Data Structures and Algorithms

Edited by:
vuquangdung

Authors:
vuquangdung
truongninhthuan
Nguyễn Việt Hà

Online version:
http://voer.edu.vn/c/8bbc580f
Outline

1. Introduction to Algorithms
2. Stack and Queue
3. Recursion
4. Binary search trees
5. Sorting
6. Graphs
7. Hashing
8. Exercises
9. Assignment problems
Contribution
Introduction to Algorithms

Problem solution

(From Wikipedia, the free encyclopedia)

Algorithms are essential to the way computers process information, because a computer program is essentially an algorithm that tells the computer what specific steps to perform (in what specific order) in order to carry out a specified task, such as calculating employees’ paychecks or printing students’ report cards. Thus, an algorithm can be considered to be any sequence of operations that can be performed by a Turing-complete system. Authors who assert this thesis include Savage (1987) and Gurevich (2000):

"...Turing's informal argument in favor of his thesis justifies a stronger thesis: every algorithm can be simulated by a Turing machine" (Gurevich 2000:1) ...according to Savage [1987], "an algorithm is a computational process defined by a Turing machine."

(Turing 2000:3)

Typically, when an algorithm is associated with processing information, data are read from an input source or device, written to an output sink or device, and/or stored for further processing. Stored data are regarded as part of the internal state of the entity performing the algorithm. In practice, the state is stored in a data structure, but an algorithm requires the internal data only for specific operation sets called abstract data types.

For any such computational process, the algorithm must be rigorously defined: specified in the way it applies in all possible circumstances that could arise. That is, any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable).

Because an algorithm is a precise list of precise steps, the order of computation will almost always be critical to the functioning of the algorithm. Instructions are usually assumed to be listed explicitly, and are described as starting 'from the top' and going 'down to the bottom', an idea that is described more formally by flow of control.

Algorithms can be expressed in many kinds of notation, including natural languages, pseudocode, flowcharts, and programming languages. Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms. Pseudocode and flowcharts are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language. Programming
languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

There is a wide variety of representations possible and one can express a given Turing machine program as a sequence of machine tables (see more at finite state machine and state transition table), as flowcharts (see more at state diagram), or as a form of rudimentary machine code or assembly code called "sets of quadruples" (see more at Turing machine).

Sometimes it is helpful in the description of an algorithm to supplement small "flow charts" (state diagrams) with natural-language and/or arithmetic expressions written inside "block diagrams" to summarize what the "flow charts" are accomplishing.

Representations of algorithms are generally classed into three accepted levels of Turing machine description (Sipser 2006:157):

- 1 High-level description:
  
  "...prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head"

- 2 Implementation description:
  
  "...prose used to define the way the Turing machine uses its head and the way that it stores data on its tape. At this level we do not give details of states or transition function"

- 3 Formal description:

  Most detailed, "lowest level", gives the Turing machine's "state table".

As it happens, it is important to know how much of a particular resource (such as time or storage) is required for a given algorithm. Methods have been developed for the analysis of algorithms to obtain such quantitative answers; for example, the algorithm above has a time requirement of O(n), using the big O notation with n as the length of the list. At all times the algorithm only needs to remember two values: the largest number found so far, and its current position in the input list. Therefore it is said to have a space requirement of O(1). (Note that the size of the inputs is not counted as space used by the algorithm.)

Different algorithms may complete the same task with a different set of instructions in less or more time, space, or effort than others. For example, given two different recipes for making potato salad, one may have peel the potato before boil the potato while the other presents the steps in the reverse order, yet they both call for these steps to be repeated for all potatoes and end when the potato salad is ready to be eaten.
The **analysis and study of algorithms** is a discipline of **computer science**, and is often practiced abstractly without the use of a specific **programming language** or implementation. In this sense, algorithm analysis resembles other mathematical disciplines in that it focuses on the underlying properties of the algorithm and not on the specifics of any particular implementation. Usually pseudocode is used for analysis as it is the simplest and most general representation.

There are various ways to classify algorithms, each with its own merits.

**Classification by implementation**

One way to classify algorithms is by implementation means.

- Recursion or iteration: A **recursive algorithm** is one that invokes (makes reference to) itself repeatedly until a certain condition matches, which is a method common to **functional programming**. **Iterative** algorithms use repetitive constructs like **loops** and sometimes additional data structures like **stacks** to solve the given problems. Some problems are naturally suited for one implementation or the other. For example, **towers of hanoi** is well understood in recursive implementation. Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.
- Logical: An algorithm may be viewed as controlled **logical deduction**. This notion may be expressed as:

  Algorithm = logic + control.

The logic component expresses the axioms that may be used in the computation and the control component determines the way in which deduction is applied to the axioms. This is the basis for the **logic programming** paradigm. In pure logic programming languages the control component is fixed and algorithms are specified by supplying only the logic component. The appeal of this approach is the elegant **semantics**: a change in the axioms has a well defined change in the algorithm.

- Serial or parallel or distributed: Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm, as opposed to **parallel algorithms** or **distributed algorithms**. Parallel algorithms take advantage of computer architectures where several processors can work on a problem at the same time, whereas distributed algorithms utilise multiple machines connected with a **network**. Parallel or distributed algorithms divide the problem into more symmetrical or asymmetrical subproblems and collect the results back together. The resource consumption in such algorithms is not only processor cycles on
each processor but also the communication overhead between the processors. Sorting algorithms can be parallelized efficiently, but their communication overhead is expensive. Iterative algorithms are generally parallelizable. Some problems have no parallel algorithms, and are called inherently serial problems.

- Deterministic or non-deterministic: Deterministic algorithms solve the problem with exact decision at every step of the algorithm whereas non-deterministic algorithm solve problems via guessing although typical guesses are made more accurate through the use of heuristics.
- Exact or approximate: While many algorithms reach an exact solution, approximation algorithms seek an approximation that is close to the true solution. Approximation may use either a deterministic or a random strategy. Such algorithms have practical value for many hard problems.

**Classification by design paradigm**

Another way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories will include many different types of algorithms. Some commonly found paradigms include:

- Divide and conquer. A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively), until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in conquer phase by merging them. A simpler variant of divide and conquer is called decrease and conquer algorithm, that solves an identical subproblem and uses the solution of this subproblem to solve the bigger problem. Divide and conquer divides the problem into multiple subproblems and so conquer stage will be more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is binary search algorithm.
- Dynamic programming. When a problem shows optimal substructure, meaning the optimal solution to a problem can be constructed from optimal solutions to subproblems, and overlapping subproblems, meaning the same subproblems are used to solve many different problem instances, a quicker approach called dynamic programming avoids recomputing solutions that have already been computed. For example, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. The main difference between dynamic programming and divide and conquer is that subproblems are more or less independent in divide and conquer, whereas subproblems overlap in dynamic programming. The difference between
dynamic programming and straightforward recursion is in caching or memoization of recursive calls. When subproblems are independent and there is no repetition, memoization does not help; hence dynamic programming is not a solution for all complex problems. By using memoization or maintaining a table of subproblems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.

- The greedy method. A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the subproblems do not have to be known at each stage; instead a "greedy" choice can be made of what looks best for the moment. The greedy method extends the solution with the best possible decision (not all feasible decisions) at an algorithmic stage based on the current local optimum and the best decision (not all possible decisions) made in previous stage. It is not exhaustive, and does not give accurate answer to many problems. But when it works, it will be the fastest method. The most popular greedy algorithm is finding the minimal spanning tree as given by Kruskal.

- Linear programming. When solving a problem using linear programming, specific inequalities involving the inputs are found and then an attempt is made to maximize (or minimize) some linear function of the inputs. Many problems (such as the maximum flow for directed graphs) can be stated in a linear programming way, and then be solved by a 'generic' algorithm such as the simplex algorithm. A more complex variant of linear programming is called integer programming, where the solution space is restricted to the integers.

- Reduction. This technique involves solving a difficult problem by transforming it into a better known problem for which we have (hopefully) asymptotically optimal algorithms. The goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithm's. For example, one selection algorithm for finding the median in an unsorted list involves first sorting the list (the expensive portion) and then pulling out the middle element in the sorted list (the cheap portion). This technique is also known as transform and conquers.

- Search and enumeration. Many problems (such as playing chess) can be modeled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for such problems. This category also includes search algorithms, branch and bound enumeration and backtracking.

- The probabilistic and heuristic paradigm. Algorithms belonging to this class fit the definition of an algorithm more loosely.
  1. Probabilistic algorithms are those that make some choices randomly (or pseudo-randomly); for some problems, it can in fact be proven that the fastest solutions must involve some randomness.
  2. Genetic algorithms attempt to find solutions to problems by mimicking biological evolutionary processes, with a cycle of random mutations yielding successive generations of "solutions". Thus, they emulate reproduction and
"survival of the fittest". In **genetic programming**, this approach is extended to algorithms, by regarding the algorithm itself as a "solution" to a problem.

3. **Heuristic** algorithms, whose general purpose is not to find an optimal solution, but an approximate solution where the time or resources are limited. They are not practical to find perfect solutions. An example of this would be **local search**, **tabu search**, or **simulated annealing** algorithms, a class of heuristic probabilistic algorithms that vary the solution of a problem by a random amount. The name "simulated annealing" alludes to the metallurgic term meaning the heating and cooling of metal to achieve freedom from defects. The purpose of the random variance is to find close to globally optimal solutions rather than simply locally optimal ones, the idea being that the random element will be decreased as the algorithm settles down to a solution.

**Classification by field of study**

Every field of science has its own problems and needs efficient algorithms. Related problems in one field are often studied together. Some example classes are **search algorithms**, **sorting algorithms**, **merge algorithms**, **numerical algorithms**, **graph algorithms**, **string algorithms**, **computational geometric algorithms**, **combinatorial algorithms**, **machine learning**, **cryptography**, **data compression** algorithms and **parsing techniques**.

Fields tend to overlap with each other, and algorithm advances in one field may improve those of other, sometimes completely unrelated, fields. For example, dynamic programming was originally invented for optimization of resource consumption in industry, but is now used in solving a broad range of problems in many fields.

**Classification by complexity**

Algorithms can be classified by the amount of time they need to complete compared to their input size. There is a wide variety: some algorithms complete in linear time relative to input size, some do so in an exponential amount of time or even worse, and some never halt. Additionally, some problems may have multiple algorithms of differing complexity, while other problems might have no algorithms or no known efficient algorithms. There are also mappings from some problems to other problems. Owing to this, it was found to be more suitable to classify the problems themselves instead of the algorithms into equivalence classes based on the complexity of the best possible algorithms for them.

**Data models**

(From Wikipedia, the free encyclopedia)
A data model is an abstract model that describes how data is represented and used.

The term data model has two generally accepted meanings:

1. A data model theory i.e. a formal description of how data may be structured and used. See also database model
2. A data model instance i.e. applying a data model theory to create a practical data model instance for some particular application. See data modeling.

Data Model Theory

A data model theory has three main components:

- The structural part: a collection of data structures which are used to create databases representing the entities or objects modeled by the database.
- The integrity part: a collection of rules governing the constraints placed on these data structures to ensure structural integrity.
- The manipulation part: a collection of operators which can be applied to the data structures, to update and query the data contained in the database.

For example, in the relational model, the structural part is based on a modified concept of the mathematical relation; the integrity part is expressed in first-order logic and the manipulation part is expressed using the relational algebra, tuple calculus and domain calculus.

Data Model Instance

Data modeling is the process of creating a data model instance by applying a data model theory. This is typically done to solve some business enterprise requirement.

Business requirements are normally captured by a semantic logical data model. This is transformed into a physical data model instance from which is generated a physical database. For more information on the tools and techniques of data modeling, see data modeling.

For example, a data modeler may use a data modeling tool to create an ERD of the Corporate data repository of some business enterprise. This model is transformed into a relational model, which in turn generates a relational database.
A data model instance may be one of three kinds (according to ANSI in 1975):

- a **conceptual schema** (data model) describes the semantics of an organization. This consists of entity classes (representing things of significance to the organization) and relationships (assertions about associations between pairs of entity classes).
- a **logical schema** (data model) describes the semantics, as represented by a particular data manipulation technology. This consists of descriptions of tables and columns, object oriented classes, and XML tags, among other things.
- a **physical schema** (data model) describes the physical means by which data are stored. This is concerned with partitions, CPUs, tablespaces, and the like.

The significance of this approach, according to ANSI, is that it allows the three perspectives to be relatively independent of each other. Storage technology can change without affecting either the logical or the conceptual model. The table/column structure can change without (necessarily) affecting the conceptual model. In each case, of course, the structures must remain consistent with the other model. The table/column structure may be different from a direct translation of the entity classes and attributes, but it must ultimately carry out the objectives of the conceptual entity class structure. Early phases of many software development projects emphasize the design of a **conceptual data model**. Such a design can be detailed into a **logical data model**. In later stages, this model may be translated into **physical data model**.

In an alternative framework, called the **Zachman Framework**, a data model instance may be one of six kinds (according to **John Zachman**, 1987):
• a **conceptual data model** (schema) consists of entity classes (representing things of significance to the organization).
• a **contextual data model** (schema) describes the semantics of an organization. This consists relationships (assertions about associations between pairs of entity classes).
• a **logical data model** (schema) describes the semantics, as represented by a particular data manipulation technology. This consists of descriptions of tables and columns, object oriented classes, and XML tags, among other things.
• a **physical data model** (schema) describes the physical means by which data are stored. This is concerned with partitions, CPUs, tablespaces, and the like.
• a **data definition** This is the actual coding of the database schema in the chosen development platform.
• a **data manipulation** describes the operations applied to the data in the schema.

The significance of this approach, according to John Zachman, is that it allows the six perspectives to be relatively independent of each other. In each case, of course, the structures must remain consistent with the other model instances although the details change. The table/column structure may be different from a direct translation of the entity classes, relationships and attributes, but it must ultimately carry out the objectives of the conceptual entity class structure and contextual relationship structure. Zachman regards each instance as a separate perspective of the database not a methodology, however development projects and software tools often proceed from conceptual data model, to contextual data model, followed by the logical data model. In later stages when the database platform is known, this model may be translated into a physical data model followed by the data definition. When the database is operational data manipulation takes place.

Different modelers may well produce different models of the same domain. This can lead to difficulty in bringing the models of different people together. Invariably, however, this difference is attributable to different levels of abstraction in the models. If the modelers agree on certain elements which are to be rendered more concretely, then the differences become less significant.

There are generic patterns that can be used to advantage for modeling business. These include the concepts PARTY (with included PERSON and ORGANIZATION), PRODUCT TYPE, PRODUCT INSTANCE, ACTIVITY TYPE, ACTIVITY INSTANCE, CONTRACT, GEOGRAPHIC AREA, and SITE. A model which explicitly includes versions of these entity classes will be both reasonably robust and reasonably easy to understand.

More abstract models are suitable for general purpose tools, and consist of variations on THING and THING TYPE, with all actual data being instances of these. Such abstract models are significantly more difficult to manage, since they are not very expressive of
real world things. More concrete and specific data models will risk having to change as the environment changes.

One approach to generic data modeling has the following characteristics:

• A generic data model shall consist of generic entity types, such as 'individual thing', 'class', 'relationship', and possibly a number of their subtypes.
• Every individual thing is an instance of a generic entity called 'individual thing' or one of its subtypes.
• Every individual thing is explicitly classified by a kind of thing ('class') using an explicit classification relationship.
• The classes used for that classification are separately defined as standard instances of the entity 'class' or one of its subtypes, such as 'class of relationship'. These standard classes are usually called 'reference data'. This means that domain specific knowledge is captured in those standard instances and not as entity types. For example, concepts such as car, wheel, building, ship, and also temperature, length, etc. are standard instances. But also standard types of relationship, such as 'is composed of' and 'is involved in' can be defined as standard instances.

This way of modeling allows the addition of standard classes and standard relation types as data (instances), which makes the data model flexible and prevents data model changes when the scope of the application changes.

A generic data model obeys the following rules:

1. Candidate attributes are treated as representing relationships to other entity types.
2. Entity types are represented, and are named after, the underlying nature of a thing, not the role it plays in a particular context. Entity types are chosen.
3. Entities have a local identifier within a database or exchange file. These should be artificial and managed to be unique. Relationships are not used as part of the local identifier.
4. Activities, relationships and event-effects are represented by entity types (not attributes).
5. Entity types are part of a sub-type/super-type hierarchy of entity types, in order to define a universal context for the model. As types of relationships are also entity types, they are also arranged in a sub-type/super-type hierarchy of types of relationship.
6. Types of relationships are defined on a high (generic) level, being the highest level where the type of relationship is still valid. For example, a composition relationship (indicated by the phrase: 'is composed of') is defined as a relationship between an 'individual thing' and another 'individual thing' (and not
just between e.g. an order and an order line). This generic level means that the type of relation may in principle be applied between any individual thing and any other individual thing. Additional constraints are defined in the 'reference data', being standard instances of relationships between kinds of things.

Examples of generic data models are ISO 10303-221, ISO 15926 and Gellish

Data organization

Another kind of data model describes how to organize data using a database management system or other data management technology. It describes, for example, relational tables and columns or object-oriented classes and attributes. Such a data model is sometimes referred to as the physical data model, but in the original ANSI three schema architecture, it is called "logical". In that architecture, the physical model describes the storage media (cylinders, tracks, and tablespaces). Ideally, this model is derived from the more conceptual data model described above. It may differ, however, to account for constraints like processing capacity and usage patterns.

While data analysis is a common term for data modeling, the activity actually has more in common with the ideas and methods of synthesis (inferring general concepts from particular instances) than it does with analysis (identifying component concepts from more general ones). {Presumably we call ourselves systems analysts because no one can say systems synthesis.} Data modeling strives to bring the data structures of interest together into a cohesive, inseparable, whole by eliminating unnecessary data redundancies and by relating data structures with relationships.

A different approach is through the use of adaptive systems such as artificial neural networks that can autonomously create implicit models of data.

Data structures

(From Wikipedia, the free encyclopedia)

In computer science, a data structure is a way of storing data in a computer so that it can be used efficiently. Often a carefully chosen data structure will allow the most efficient algorithm to be used. The choice of the data structure often begins from the choice of an abstract data structure. A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structures are implemented using the data types, references and operations on them provided by a programming language.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks. For example, B-trees are particularly well-suited
for implementation of databases, while routing tables rely on networks of machines to function.

In the design of many types of programs, the choice of data structures is a primary design consideration, as experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure. After the data structures are chosen, the algorithms to be used often become relatively obvious. Sometimes things work in the opposite direction - data structures are chosen because certain key tasks have algorithms that work best with particular data structures. In either case, the choice of appropriate data structures is crucial.

This insight has given rise to many formalized design methods and programming languages in which data structures, rather than algorithms, are the key organizing factor. Most languages feature some sort of module system, allowing data structures to be safely reused in different applications by hiding their verified implementation details behind controlled interfaces. Object-oriented programming languages such as C++ and Java in particular use classes for this purpose.

Since data structures are so crucial, many of them are included in standard libraries of modern programming languages and environments, such as C++'s Standard Template Library, the Java Collections Framework, and the Microsoft .NET Framework.

The fundamental building blocks of most data structures are arrays, records, discriminated unions, and references. For example, the nullable reference, a reference which can be null, is a combination of references and discriminated unions, and the simplest linked data structure, the linked list, is built from records and nullable references.

Data structures represent implementations or interfaces: A data structure can be viewed as an interface between two functions or as an implementation of methods to access storage that is organized according to the associated data type.

Algorithms analysis

(From Wikipedia, the free encyclopedia)

To analyze an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).
Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search of efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in asymptotic sense, i.e., to estimate the complexity function for reasonably large length of input. Big O notation, omega notation and theta notation are used to this end. For instance, binary search is said to run an amount of steps proportional to a logarithm, or in O(log(n)), colloquially "in logarithmic time". Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called hidden constant.

Exact (not asymptotic) measures of efficiency can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called model of computation. A model of computation may be defined in terms of an abstract computer, e.g., Turing machine, and/or by postulating that certain operations are executed in unit time. For example, if the sorted set to which we apply binary search has N elements, and we can guarantee that a single binary lookup can be done in unit time, then at most log2 N + 1 time units are needed to return an answer.

Exact measures of efficiency are useful to the people who actually implement and use algorithms, because they are more precise and thus enable them to know how much time they can expect to spend in execution. To some people (e.g. game programmers), a hidden constant can make all the difference between success and failure.

Time efficiency estimates depend on what we define to be a step. For the analysis to make sense, the time required to perform a step must be guaranteed to be bounded above by a constant. One must be careful here; for instance, some analyses count an addition of two numbers as a step. This assumption may not be warranted in certain contexts. For example, if the numbers involved in a computation may be arbitrarily large, addition no longer can be assumed to require constant time (compare the time you need to add two 2-digit integers and two 1000-digit integers using a pen and paper).
Stack and Queue

Stack

(From Wikipedia, the free encyclopedia)

In computer science, a stack is a temporary abstract data type and data structure based on the principle of Last In First Out (LIFO). Stacks are used extensively at every level of a modern computer system. For example, a modern PC uses stacks at the architecture level, which are used in the basic design of an operating system for interrupt handling and operating system function calls. Among other uses, stacks are used to run a Java Virtual Machine, and the Java language itself has a class called "Stack", which can be used by the programmer. The stack is ubiquitous.

A stack-based computer system is one that stores temporary information primarily in stacks, rather than hardware CPU registers (a register-based computer system).

Abstract data type

(From Wikipedia, the free encyclopedia)

As an abstract data type, the stack is a container of nodes and has two basic operations: push and pop. Push adds a given node to the top of the stack leaving previous nodes below. Pop removes and returns the current top node of the stack. A frequently used metaphor is the idea of a stack of plates in a spring loaded cafeteria stack. In such a stack, only the top plate is visible and accessible to the user, all other plates remain hidden. As new plates are added, each new plate becomes the top of the stack, hiding each plate below, pushing the stack of plates down. As the top plate is removed from the stack, they can be used, the plates pop back up, and second plate becomes the top of the stack. Two important principles are illustrated by this metaphor, the Last In First Out principle is one. The second is that the contents of the stack are hidden. Only the top plate is visible, so to see what is on the third plate, the first and second plates will have to be removed.
Operations

In modern computer languages, the stack is usually implemented with more operations than just "push" and "pop". The length of a stack can often be returned as a parameter. Another helper operation top (also known as peek and peak) can return the current top element of the stack without removing it from the stack.

This section gives pseudocode for adding or removing nodes from a stack, as well as the length and top functions. Throughout we will use null to refer to an end-of-list marker or sentinel value, which may be implemented in a number of ways using pointers.

record Node {
    data // The data being stored in the node
    next // A reference to the next node; null for last node
}

record Stack {
    Node stackPointer // points to the 'top' node; null for an empty stack
}

function push(Stack stack, Element element) { // push element onto stack
    new(newNode) // Allocate memory to hold new node
    newNode.data := element
    newNode.next := stack.stackPointer
    stack.stackPointer := newNode
}

function pop(Stack stack) { // increase the stack pointer and return 'top' node
    node := stack.stackPointer
    // You could check if stack.stackPointer is null here.
    // If so, you may wish to error, citing the stack underflow.
    return node
}
stack.stackPointer := node.next

element := node.data

return element

}

function top(Stack stack) { // return 'top' node

return stack.stackPointer.data

}

function length(Stack stack) { // return the amount of nodes in the stack

length := 0

node := stack.stackPointer

while node not null {

length := length + 1

node := node.next

}

return length

}

As you can see, these functions pass the stack and the data elements as parameters and return values, not the data nodes that, in this implementation, include pointers. A stack may also be implemented as a linear section of memory (i.e. an array), in which case the function headers would not change, just the internals of the functions.

Implementation

A typical storage requirement for a stack of n elements is O(n). The typical time requirement of O(1) operations is also easy to satisfy with a dynamic array or (singly) linked list implementation.
C++'s Standard Template Library provides a "stack" templated class which is restricted to only push/pop operations. Java's library contains a Stack class that is a specialization of Vector. This could be considered a design flaw because the inherited get() method from Vector ignores the LIFO constraint of the Stack.

Here is a simple example of a stack with the operations described above in Python. It does not have any type of error checking.

class Stack:

    def __init__(self):
        self.stack_pointer = None

    def push(self, element):
        self.stack_pointer = Node(element, self.stack_pointer)

    def pop(self):
        (e, self.stack_pointer) = (self.stack_pointer.element, self.stack_pointer.next)
        return e

    def peek(self):
        return self.stack_pointer.element

    def __len__(self):
        i = 0
        sp = self.stack_pointer
        while sp:
            i += 1
            sp = sp.next
        return i

class Node:
def __init__(self, element=None, next=None):
    self.element = element
    self.next = next

if __name__ == '__main__':
    # small use example
    s = Stack()
    [s.push(i) for i in xrange(10)]
    print [s.pop() for i in xrange(len(s))]

    The above is admittedly redundant as Python supports the 'pop' and 'append' functions to lists.

    Application

    (From Wikipedia, the free encyclopedia)

    Stacks are ubiquitous in the computing world.

    Expression evaluation and syntax parsing

    Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based machines.

    For example, The calculation: 
    
    \[(1 + 2) * 4 + 3\]
    
    can be written down like this in postfix notation with the advantage of no precedence rules and parentheses needed:

    \[1\ 2\ 4\ *\ 3\ +\]

    The expression is evaluated from the left to right using a stack:

    - push when encountering an operand and
    - pop two operands and evaluate the value when encountering an operation.
    - push the result
Like the following way (the Stack is displayed after Operation has taken place):

<table>
<thead>
<tr>
<th>Input</th>
<th>Operation</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Push operand</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Push operand</td>
<td>1, 2</td>
</tr>
<tr>
<td>+</td>
<td>Add</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Push operand</td>
<td>3, 4</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Push operand</td>
<td>12, 3</td>
</tr>
<tr>
<td>+</td>
<td>Add</td>
<td>15</td>
</tr>
</tbody>
</table>

The final result, 15, lies on the top of the stack at the end of the calculation. Example: implementation in pascal language. Using marked sequential file as data archives.

programmer : clx321

file : stack.pas

unit : Pstack.tpu

}

program TestStack;

{this program use ADT of Stack, i will assume that the unit of ADT of Stack has already existed}

uses

PStack; {ADT of STACK}

{dictionary}

const

mark = '.';

var

data : stack;
f : text;
cc : char;
ccInt, cc1, cc2 : integer;

{functions}
IsOperand (cc : char) : boolean; {JUST Prototype}
{return TRUE if cc is operand}

ChrToInt (cc : char) : integer; {JUST Prototype}
{change char to integer}

Operator (cc1, cc2 : integer) : integer; {JUST Prototype}
{operate two operands}

{algorithms}
begin
assign (f, cc);
reset (f);
read (f, cc); {first elmt}
if (cc = mark) then
begin
writeln ('empty archives !');
end
else
begin
repeat
if (IsOperand (cc)) then begin
ccInt := ChrToInt (cc);
push (ccInt, data);
end
else begin
pop (cc1, data);
pop (cc2, data);
push (data, Operator (cc2, cc1));
end;
read (f, cc); {next elmt}
until (cc = mark);
end;
close (f);
end.

Runtime memory management

A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack. For example, PostScript has a return stack and an operand stack, and also has a graphics state stack and a dictionary stack.

Forth uses two stacks, one for argument passing and one for subroutine return addresses. The use of a return stack is extremely commonplace, but the somewhat unusual use of an argument stack for a human-readable programming language is the reason Forth is referred to as a stack-based language.
Many virtual machines are also stack-oriented, including the p-code machine and the Java virtual machine.

Almost all computer runtime memory environments use a special stack (the "call stack") to hold information about procedure/function calling and nesting in order to switch to the context of the called function and restore to the caller function when the calling finishes. They follow a runtime protocol between caller and callee to save arguments and return value on the stack. Stacks are an important way of supporting nested or recursive function calls. This type of stack is used implicitly by the compiler to support CALL and RETURN statements (or their equivalents) and is not manipulated directly by the programmer.

Some programming languages use the stack to store data that is local to a procedure. Space for local data items is allocated from the stack when the procedure is entered, and is deallocated when the procedure exits. The C programming language is typically implemented in this way. Using the same stack for both data and procedure calls has important security implications (see below) of which a programmer must be aware in order to avoid introducing serious security bugs into a program.

Solving search problems

Solving a search problem, regardless of whether the approach is exhaustive or optimal, needs stack space. Examples of exhaustive search methods are brute-force and backtracking. Examples of optimal search exploring methods are branch and bound and heuristic solutions. All of these algorithms use stacks to remember the search nodes that have been noticed but not explored yet. The only alternative to using a stack is to use recursion and let the compiler do the remembering for you (but in this case the compiler is still using a stack internally). The use of stacks is prevalent in many problems, ranging from simple in-order traversals of trees or depth-first traversals of graphs to a crossword puzzle solver or computer chess game. Some of these problems can be solved by alternative data structures like a queue, when a different order of traversal is required.

Security

Some computing environments use stacks in ways that may make them vulnerable to security breaches and attacks. Programmers working in such environments must take special care to avoid the pitfalls of these implementations.

For example, some programming languages use a common stack to store both data local to a called procedure and the linking information that allows the procedure to return to its caller. This means that the program moves data into and out of the same stack that contains critical return addresses for the procedure calls. If data is moved to the wrong
location on the stack, or an oversized data item is moved to a stack location that is not large enough to contain it, return information for procedure calls may be corrupted, causing the program to fail.

Malicious parties may attempt to take advantage of this type of implementation by providing oversized data input to a program that does not check the length of input. Such a program may copy the data in its entirety to a location on the stack, and in so doing it may change the return addresses for procedures that have called it. An attacker can experiment to find a specific type of data that can be provided to such a program such that the return address of the current procedure is reset to point to an area within the stack itself (and within the data provided by the attacker), which in turn contains instructions that carry out unauthorized operations.

This type of attack is a variation on the buffer overflow attack and is an extremely frequent source of security breaches in software, mainly because some of the most popular programming languages (such as C) use a shared stack for both data and procedure calls, and do not verify the length of data items. Frequently programmers do not write code to verify the size of data items, either, and when an oversized or undersized data item is copied to the stack, a security breach may occur.

Queue

(From Wikipedia, the free encyclopedia)

A queue is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. Queues provide services in computer science, transport and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes.

The most well known operation of the queue is the First-In-First-Out (FIFO) queue process. In a FIFO queue, the first element added to in the queue will be the first one out. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element can be invoked. Unless otherwise specified, the remainder of the article will refer to FIFO queues. There are also non-FIFO queue data structures, like priority queues.

Performance
A straightforward analysis shows that for both these cases, the time needed to add or delete an item is constant and independent of the number of items in the queue. Thus we class both addition and deletion as an O(1) operation. For any given real machine+operating system+language combination, addition may take c1 seconds and deletion c2 seconds, but we aren't interested in the value of the constant, it will vary from machine to machine, language to language, etc. The key point is that the time is not dependent on n - producing O(1) algorithms.

Once we have written an O(1) method, there is generally little more that we can do from an algorithmic point of view. Occasionally, a better approach may produce a lower constant time. Often, enhancing our compiler, run-time system, machine, etc will produce some significant improvement. However O(1) methods are already very fast, and it's unlikely that effort expended in improving such a method will produce much real gain!

Basic operations

There are two basic operations associated with a queue: enqueue and dequeue. Enqueue means adding a new item to the rear of the queue while dequeue refers to removing the front item from the queue and returning it to the calling entity.

Theoretically, one characteristic of a queue is that it does not have a specific capacity. Regardless of how many elements are already contained, a new element can always be added. It can also be empty, at which point removing an element will be impossible until a new element has been added again.

A practical implementation of a queue e.g. with pointers of course does have some capacity limit, that depends on the concrete situation it is used in. For a data structure the executing computer will eventually run out of memory, thus limiting the queue size. Queue overflow results from trying to add an element onto a full queue and queue underflow happens when trying to remove an element from an empty queue.

A bounded queue is a queue limited to a fixed number of items.

FIFO queue is a queue in which the first item added is always the first one out.

LIFO queue is a queue in which the item most recently added is always the first one out.

Priority queue is a queue in which the items are sorted so that the highest priority item is always the next one to be extracted.
Priority queue

(From Wikipedia, the free encyclopedia)

A priority queue is an abstract data type in computer programming, supporting the following three operations:

- add an element to the queue with an associated priority
- remove the element from the queue that has the highest priority, and return it
- (optionally) peek at the element with highest priority without removing it

The simplest way to implement a priority queue data type is to keep an associative array mapping each priority to a list of elements with that priority. If association lists are used to implement the associative array, adding an element takes constant time but removing or peeking at the element of highest priority takes linear (O(n)) time, because we must search all keys for the largest one. If a self-balancing binary search tree is used, all three operations take O(log n) time; this is a popular solution in environments that already provide balanced trees but nothing more sophisticated. The van Emde Boas tree, another associative array data structure, can perform all three operations in O(log log n) time, but at a space cost for small queues of about O(2m/2), where m is the number of bits in the priority value, which may be prohibitive.

There are a number of specialized heap data structures that either supply additional operations or outperform the above approaches. The binary heap uses O(log n) time for both operations, but allows peeking at the element of highest priority without removing it in constant time. Binomial heaps add several more operations, but require O(log n) time for peeking. Fibonacci heaps can insert elements, peek at the maximum priority element, and increase an element's priority in amortized constant time (deletions are still O(log n)).

The Standard Template Library (STL), part of the C++ 1998 standard, specifies "priority_queue" as one of the STL container adaptor class templates. Unlike actual STL containers, it does not allow iteration of its elements (it strictly adheres to its abstract data type definition). Java's library contains a PriorityQueue class.

Applications

Bandwidth management

Priority queuing can be used to manage limited resources such as bandwidth on a transmission line from a network router. In the event of outgoing traffic queuing due to insufficient bandwidth, all other queues can be halted to send the traffic from the highest priority queue upon arrival. This ensures that the prioritized traffic (such as real-time
traffic, e.g. a RTP stream of a VoIP connection) is forwarded with the least delay and the least likelihood of being rejected due to a queue reaching its maximum capacity. All other traffic can be handled when the highest priority queue is empty. Another approach used is to send disproportionately more traffic from higher priority queues.

Usually a limitation (policer) is set to limit the bandwidth that traffic from the highest priority queue can take, in order to prevent high priority packets from choking off all other traffic. This limit is usually never reached due to high lever control instances such as the Cisco Callmanager, which can be programmed to inhibit calls which would exceed the programmed bandwidth limit.

**Discrete event simulation**

Another use of a priority queue is to manage the events in a discrete event simulation. The events are added to the queue with their simulation time used as the priority. The execution of the simulation proceeds by repeatedly pulling the top of the queue and executing the event thereon.

See also: Scheduling (computing), queueing theory

**A* search algorithm**

The A* search algorithm finds the shortest path between two vertices of a weighted graph, trying out the most promising routes first. The priority queue is used to keep track of unexplored routes; the one for which a lower bound on the total path length is smallest is given highest priority.

**Implementations**

While relying on heapsort is a common way to implement priority queues, for integer data faster implementations exist.

- When the set of keys is \( \{1, 2, \ldots, C\} \), a data structure by Emde Boas supports the minimum, maximum, insert, delete, search, extract-min, extract-max, predecessor and successor operations in \( O(\log C) \) time.

An algorithm by Fredman and Willard implements the minimum operation in \( O(1) \) time and insert and extract-min operations in \( O(\sqrt{\log n}) \) time.
Recursion

(From Wikipedia, the free encyclopedia)

Recursion in computer programming defines a function in terms of itself. One example application of recursion is in recursive descent parsers for programming languages. The great advantage of recursion is that an infinite set of possible sentences, designs, or other data can be defined, parsed, or produced by a finite computer program.

Recursive algorithms

(From Wikipedia, the free encyclopedia)

A common method of simplification is to divide a problem into sub-problems of the same type. As a computer programming technique, this is called divide and conquer, and it is key to the design of many important algorithms, as well as being a fundamental part of dynamic programming.

Virtually all programming languages in use today allow the direct specification of recursive functions and procedures. When such a function is called, the computer (for most languages on most stack-based architectures) or the language implementation keeps track of the various instances of the function (on much architecture, by using a call stack, although other methods may be used). Conversely, every recursive function can be transformed into an iterative function by using a stack.

Any function that can be evaluated by a computer can be expressed in terms of recursive functions without the use of iteration, in continuation-passing style; and conversely any recursive function can be expressed in terms of iteration.

To make a very literal example out of this: If an unknown word is seen in a book, the reader can make a note of the current page number and put the note on a stack (which is empty so far). The reader can then look the new word up and, while reading on the subject, may find yet another unknown word. The page number of this word is also written down and put on top of the stack. At some point an article is read that does not require any explanation. The reader then returns to the previous page number and continues reading from there. This is repeated, sequentially removing the topmost note from the stack. Finally, the reader returns to the original book. This is a recursive approach.

Some languages designed for logic programming and functional programming provide recursion as the only means of repetition directly available to the programmer. Such
languages generally make tail recursion as efficient as iteration, letting programmers express other repetition structures (such as Scheme's map and for) in terms of recursion.

Recursion is deeply embedded in the theory of computation, with the theoretical equivalence of mu-recursive functions and Turing machines at the foundation of ideas about the universality of the modern computer.

**Recursive programming**

(From Wikipedia, the free encyclopedia)

One basic form of recursive computer program is to define one or a few base cases, and then define rules to break down other cases into the base case. This analytic method is a common design for parsers for computer languages; see Recursive descent parser.

Another, similar form is generative, or synthetic, recursion. In this scheme, the computer uses rules to assemble cases, and starts by selecting a base case. This scheme is often used when a computer must design something automatically, such as code, a machine part or some other data.

One common example (using the Pascal programming language, in this case) is the function used to calculate the factorial of an integer:

```pascal
function Factorial(x: integer): integer;
begin
if x <= 1 then
Factorial := 1
else
Factorial := x * Factorial(x-1);
end
```

Here is the same function coded without recursion. Notice that this iterative solution requires two temporary variables; in general, recursive formulations of algorithms are often considered "cleaner" or "more elegant" than iterative formulations.

```pascal
function Factorial(x: integer): integer;
var i, temp: integer;
```
begin

temp := 1;

for i := 1 to x do
    temp := temp * i

Factorial := temp

end

Another comparison that even more clearly demonstrates the relative "elegance" of recursive functions is the Euclidean algorithm, used to compute the greatest common divisor of two integers. Below is the algorithm with recursion, coded in C:

int gcd(int x, int y)
{
    if (y == 0)
        return x;
    else
        return gcd(y, x % y);
}

Below is the same algorithm using an iterative approach:

int gcd(int x, int y)
{
    while (y != 0) {
        int r = x % y;
        x = y;
        y = r;
    }
    return x;
}
The iterative algorithm requires a temporary variable, and even given knowledge of the Euclidean algorithm it is more difficult to understand the process by simple inspection, although they are very similar in their steps.

Recursion versus iteration

In the "factorial" example the iterative implementation is likely to be slightly faster in practice than the recursive one. This is almost definite for the Euclidean Algorithm implementation. This result is typical, because iterative functions do not pay the "function-call overhead" as many times as recursive functions, and that overhead is relatively high in many languages. (Note that an even faster implementation for the factorial function on small integers is to use a lookup table.)

There are other types of problems whose solutions are inherently recursive, because they need to keep track of prior state. One example is tree traversal; others include the Ackermann function and divide-and-conquer algorithms such as Quicksort. All of these algorithms can be implemented iteratively with the help of a stack, but the need for the stack arguably nullifies the advantages of the iterative solution.

Another possible reason for choosing an iterative rather than a recursive algorithm is that in today's programming languages, the stack space available to a thread is often much less than the space available in the heap, and recursive algorithms tend to require more stack space than iterative algorithms.

**Recursive functions**

(From Wikipedia, the free encyclopedia)

Functions whose domains can be recursively defined can be given recursive definitions patterned after the recursive definition of their domain.

The canonical example of a recursively defined function is the following definition of the factorial function $f(n)$:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n-1) & \text{if } n > 0 \end{cases}$$
Given this definition, also called a recurrence relation, we work out $f(3)$ as follows:

$$f(3) = 3 \times f(3 - 1)$$

$$= 3 \times f(2)$$

$$= 3 \times 2 \times f(2 - 1)$$

$$= 3 \times 2 \times f(1)$$

$$= 3 \times 2 \times 1 \times f(1 - 1)$$

$$= 3 \times 2 \times 1 \times f(0)$$

$$= 3 \times 2 \times 1 \times 1$$

$$= 6$$

**Tail-recursive functions**

(From Wikipedia, the free encyclopedia)

Tail-recursive functions are functions ending in a recursive call. For example, the following C function to locate a value in a linked list is tail-recursive, because the last thing it does is call itself:

```c
struct node {
    int data;
    struct node *next;
};

struct node *find_value(struct node *head, int value)
{
    if (head == NULL)
        return NULL;

    if (head->data == value)
        return head;

    return find_value(head->next, value);
}
```
return head;
return find_value(head->next, value);
}

The Euclidean Algorithm function, following a similar structure, is also tail-recursive. On the other hand, the Factorial function used as an example in the previous section is not tail-recursive, because after it receives the result of the recursive call, it must multiply that result by x before returning to its caller. That kind of function is sometimes called augmenting recursive.

The Factorial function can be turned into a tail-recursive function:

function Factorial(acc: integer, x: integer): integer;
begin
if x <= 1 then
Factorial := acc
else
Factorial := Factorial(x * acc, x - 1);
end

Function should then be called by Factorial(1, x).

Notice that a single function may be both tail-recursive and augmenting recursive, such as this function to count the odd integers in a linked list:

int count_odds(struct node *head)
{
if (head == NULL)
return 0;
if (head->data % 2 == 1)
return count_odds(head->next) + 1; /* augmenting recursion */
return count_odds(head->next); /* tail recursion */

}

The significance of tail recursion is that when making a tail-recursive call, the caller's return position need not be saved on the call stack; when the recursive call returns, it will branch directly on the previously saved return position. Therefore, on compilers which support tail-recursion optimization, tail recursion saves both space and time.
Binary search trees

Introduction to binary trees

(From Wikipedia, the free encyclopedia)

A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13.

In computer science, a binary search tree (BST) is a binary treedata structure which has the following properties:

- Each node has a value.
- A total order is defined on these values.
- The left subtree of a node contains only values less than the node's value.
- The right subtree of a node contains only values greater than or equal to the node's value.

The major advantage of binary search trees is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

If a BST allows duplicate values, then it represents a multiset. This kind of tree uses non-strict inequalities. Everything in the left subtree of a node is strictly less than the value of the node, but everything in the right subtree is either greater than or equal to the value of the node.

If a BST doesn't allow duplicate values, then the tree represents a set with unique values, like the mathematical set. Trees without duplicate values use strict inequalities, meaning that the left subtree of a node only contains nodes with values that are less than the value of the node, and the right subtree only contains values that are greater.
The choice of storing equal values in the right subtree only is arbitrary; the left would work just as well. One can also permit non-strict equality in both sides. This allows a tree containing many duplicate values to be balanced better, but it makes searching more complex.

**Operations**

All operations on a binary tree make several calls to a comparator, which is a subroutine that computes the total order on any two values. In generic implementations of binary search trees, a program often provides a callback to a comparator when it creates a tree, either explicitly or, in languages that support type polymorphism, by having values be of a comparable type.

**Searching**

(From Wikipedia, the free encyclopedia)

Searching a binary tree for a specific value is a process that can be performed recursively because of the order in which values are stored. We begin by examining the root. If the value we are searching for equals the root, the value exists in the tree. If it is less than the root, then it must be in the left subtree, so we recursively search the left subtree in the same manner. Similarly, if it is greater than the root, then it must be in the right subtree, so we recursively search the right subtree. If we reach a leaf and have not found the value, then the item is not where it would be if it were present, so it does not lie in the tree at all. A comparison may be made with binary search, which operates in nearly the same way but using random access on an array instead of following links.

Here is the search algorithm in the Python programming language:

```python
def search_binary_tree(node, key):
    if node is None:
        return None # key not found
    if key < node.key:
        return search_binary_tree(node.left, key)
    elif key > node.key:
        return search_binary_tree(node.right, key)
    else: # key is equal to node key
        return node.key
```

36/174
return node.value # found key

This operation requires $O(\log n)$ time in the average case, but needs $O(n)$ time in the worst-case, when the unbalanced tree resembles a linked list.

**Insertion**

(From Wikipedia, the free encyclopedia)

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

Here's how a typical binary search tree insertion might be performed in C++:

```c++
/* Inserts the node pointed to by "newNode" into the subtree rooted at "treeNode" */
void InsertNode(struct node *&treeNode, struct node *newNode)
{
    if (treeNode == NULL)
        treeNode = newNode;
    else if (newNode->value < treeNode->value)
        InsertNode(treeNode->left, newNode);
    else
        InsertNode(treeNode->right, newNode);
}
```

The above "destructive" procedural variant modifies the tree in place. It uses only constant space, but the previous version of the tree is lost. Alternatively, as in the following Python example, we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a persistent data structure:
def binary_tree_insert(node, key, value):
    if node is None:
        return TreeNode(None, key, value, None)
    if key == node.key:
        return TreeNode(node.left, key, value, node.right)
    if key < node.key:
        return TreeNode(binary_tree_insert(node.left, key, value), node.key, node.value, node.right)
    else:
        return TreeNode(node.left, node.key, node.value, binary_tree_insert(node.right, key, value))

The part that is rebuilt uses $\Theta(\log n)$ space in the average case and $\Omega(n)$ in the worst case (see big-O notation).

In either version, this operation requires time proportional to the height of the tree in the worst case, which is $\mathcal{O}(\log n)$ time in the average case over all trees, but $\Omega(n^2)$ time in the worst case.

Another way to explain insertion is that in order to insert a new node in the tree, its value is first compared with the value of the root. If its value is less than the root's, it is then compared with the value of the root's left child. If its value is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value.

There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

**Deletion**

(From Wikipedia, the free encyclopedia)

There are several cases to be considered:

- Deleting a leaf: Deleting a node with no children is easy, as we can simply remove it from the tree.
• Deleting a node with one child: Delete it and replace it with its child.
• Deleting a node with two children: Suppose the node to be deleted is called N. We replace the value of N with either its in-order successor (the left-most child of the right subtree) or the in-order predecessor (the right-most child of the left subtree).

Once we find either the in-order successor or predecessor, swap it with N, and then delete it. Since both the successor and the predecessor must have fewer than two children, either one can be deleted using the previous two cases. In a good implementation, it is generally recommended to avoid consistently using one of these nodes, because this can unbalance the tree.

Here is C++ sample code for a destructive version of deletion. (We assume the node to be deleted has already been located using search.)

```c
void DeleteNode(struct node * & node) {
    if (node->left == NULL) {
        struct node *temp = node;
        node = node->right;
        delete temp;
    } else if (node->right == NULL) {
        struct node *temp = node;
        node = node->left;
        delete temp;
    } else if (node->right == NULL) {
        struct node *temp = node;
        node = node->left;
        delete temp;
    }
```

39/174
Although this operation does not always traverse the tree down to a leaf, this is always a possibility; thus in the worst case it requires time proportional to the height of the tree. It does not require more even when the node has two children, since it still follows a single path and does not visit any node twice.

**Traversal**

(From Wikipedia, the free encyclopedia)

Compared to linear data structures like linked lists and one dimensional arrays, which have only one logical means of traversal, tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed define the traversal type. These steps are: Performing an action on the current node (referred to as "visiting" the node); or repeating the process with the subtrees rooted at our left and right children. Thus the process is most easily described through recursion.
To traverse a non-empty binary tree in preorder, we perform the following three operations: 1. Visit the root. 2. Traverse the left subtree in preorder. 3. Traverse the right subtree in preorder.

To traverse a non-empty binary tree in inorder, perform the following operations: 1. Traverse the left subtree in inorder. 2. Visit the root. 3. Traverse the right subtree in inorder.

To traverse a non-empty binary tree in postorder, perform the following operations: 1. Traverse the left subtree in postorder. 2. Traverse the right subtree in postorder. 3. Visit the root. This is also called Depth-first traversal.

Finally, trees can also be traversed in level-order, where we visit every node on a level before going to a lower level. This is also called Breadth-first traversal.

Once the binary search tree has been created, its elements can be retrieved in order by recursively traversing the left subtree of the root node, accessing the node itself, then recursively traversing the right subtree of the node, continuing this pattern with each node in the tree as it's recursively accessed. The tree may also be traversed in pre-order or post-order traversals. The following is the implementation of these traversals:

preorder(node)
print node.value
if node.left ≠ null then preorder(node.left)
if node.right ≠ null then preorder(node.right)
inorder(node)
if node.left ≠ null then inorder(node.left)
print node.value
if node.right ≠ null then inorder(node.right)
postorder(node)
if node.left ≠ null then postorder(node.left)
if node.right ≠ null then postorder(node.right)
print node.value
All three sample implementations will require stack space proportional to the height of the tree. In a poorly balanced tree, this can be quite considerable.

**Sort**

(From Wikipedia, the free encyclopedia)

A binary search tree can be used to implement a simple but inefficient sorting algorithm. Similarly to heapsort, we insert all the values we wish to sort into a new ordered data structure — in this case a binary search tree — and then traverse it in order, building our result:

```python
def build_binary_tree(values):
    tree = None
    for v in values:
        tree = binary_tree_insert(tree, v)
    return tree

def traverse_binary_tree(treenode):
    if treenode is None: return []
    else:
        left, value, right = treenode
        return (traverse_binary_tree(left) + [value] + traverse_binary_tree(right))
```

The worst-case time of build_binary_tree is $\Theta(n^2)$ — if you feed it a sorted list of values, it chains them into a linked list with no left subtrees. For example, build_binary_tree([1, 2, 3, 4, 5]) yields the tree (None, 1, (None, 2, (None, 3, (None, 4, (None, 5, None))))).

There are several schemes for overcoming this flaw with simple binary trees; the most common is the self-balancing binary search tree. If this same procedure is done using such a tree, the overall worst-case time is $O(n \log n)$, which is asymptotically optimal for a comparison sort. In practice, the poor cache performance and added overhead in time and space for a tree-based sort (particularly for node allocation) make it inferior to other asymptotically optimal sorts such as quicksort and heapsort for static list sorting. On the other hand, it is one of the most efficient methods of incremental sorting, adding items to a list over time while keeping the list sorted at all times.
Types of binary search trees

(From Wikipedia, the free encyclopedia)

There are many types of binary search trees. AVL trees and red-black trees are both forms of self-balancing binary search trees. A splay tree is a binary search tree that automatically moves frequently accessed elements nearer to the root. In a treap ("tree heap"), each node also holds a priority and the parent node has higher priority than its children.

Performance comparisons

D. A. Heger (2004) presented a performance comparison of binary search trees. Treap was found to have the best average performance, while red-black tree was found to have the smallest amount of performance fluctuations.

Optimal binary search trees

If we don't plan on modifying a search tree, and we know exactly how often each item will be accessed, we can construct an optimal binary search tree, which is a search tree where the average cost of looking up an item (the expected search cost) is minimized.

Assume that we know the elements and that for each element, we know the proportion of future lookups which will be looking for that element. We can then use a dynamic programming solution, detailed in section 15.5 of Introduction to Algorithms by Thomas H. Cormen Sec Edition, to construct the tree with the least possible expected search cost.

Even if we only have estimates of the search costs, such a system can considerably speed up lookups on average. For example, if you have a BST of English words used in a spell checker, you might balance the tree based on word frequency in text corpuses, placing words like "the" near the root and words like "agrasia" near the leaves. Such a tree might be compared with Huffman trees, which similarly seek to place frequently-used items near the root in order to produce a dense information encoding; however, Huffman trees only store data elements in leaves and these elements need not be ordered.

If we do not know the sequence in which the elements in the tree will be accessed in advance, we can use splay trees which are asymptotically as good as any static search tree we can construct for any particular sequence of lookup operations.

Alphabetic trees are Huffman trees with the additional constraint on order, or, equivalently, search trees with the modification that all elements are stored in the leaves. Faster algorithms exist for optimal alphabetic binary trees (OABTs).
Sorting

Basic sort algorithms

(From Wikipedia, the free encyclopedia)

In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation, or reordering, of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being invented to this day (for example, library sort was first published in 2004). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide-and-conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

Classification

Sorting algorithms used in computer science are often classified by:

- Computational complexity (worst, average and best behaviour) of element comparisons in terms of the size of the list (n). For typical sorting algorithms good behavior is O(n log n) and bad behavior is Ω(n²). (See Big O notation)
- Ideal behavior for a sort is O(n). Sort algorithms which only use an abstract key comparison operation always need at least Ω(n log n) comparisons on average.
- Computational complexity of swaps (for "in place" algorithms).
- Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in place", such that only O(1) or O(log n) memory is
needed beyond the items being sorted, while others need to create auxiliary locations for data to be temporarily stored.

- Recursion. Some algorithms are either recursive or non recursive, while others may be both (e.g., merge sort).
- Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). See below for more information.
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- General method: insertion, exchange, selection, merging, etc. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.

Stability

Stable sorting algorithms maintain the relative order of records with equal keys [http://en.wikipedia.org/wiki/Strict_weak_ordering](http://en.wikipedia.org/wiki/Strict_weak_ordering) (i.e. sort key values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first coordinate:

(4, 1) (3, 7) (3, 1) (5, 6)

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

(3, 7) (3, 1) (4, 1) (5, 6) (order maintained)

(3, 1) (3, 7) (4, 1) (5, 6) (order changed)

Unstable sorting algorithms may change the relative order of records with equal keys, but stable sorting algorithms never do so. Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original data order as a tie-breaker. Remembering this order, however, often involves an additional space cost.

Sorting based on a primary, secondary, tertiary, etc. sort key can be done by any sorting method, taking all sort keys into account in comparisons (in other words, using a single composite sort key). If a sorting method is stable, it is also possible to sort multiple
times, each time with one sort key. In that case the sort keys can be applied in any order, where some key orders may lead to a smaller running time.

6.1.1. Insertion sort

(From Wikipedia, the free encyclopedia)

Insertion sort is a simple sorting algorithm, a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort, but it has various advantages:

- Simple to implement
- Efficient on (quite) small data sets
- Efficient on data sets which are already substantially sorted: it runs in O(n + d) time, where d is the number of inversions
- More efficient in practice than most other simple O(n^2) algorithms such as selection sort or bubble sort: the average time is n^2/4 and it is linear in the best case
- **Stable** (does not change the relative order of elements with equal keys)
- **In-place** (only requires a constant amount O(1) of extra memory space)
- It is an online algorithm, in that it can sort a list as it receives it.

**Algorithm**

In abstract terms, every iteration of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input. The choice of which element to remove from the input is arbitrary and can be made using almost any choice algorithm.

Sorting is typically done in-place. The resulting array after k iterations contains the first k entries of the input array and is sorted. In each step, the first remaining entry of the input is removed, inserted into the result at the right position, thus extending the result:

\[
\begin{array}{c|c|c|c}
\text{Sorted partial result} & \text{Unsorted data} \\
\hline
\leq x & \gt x & x & \ldots \\
\end{array}
\]

becomes:

\[
\begin{array}{c|c|c|c}
\text{Sorted partial result} & \text{Unsorted data} \\
\hline
\leq x & \underline{x} & \gt x & \ldots \\
\end{array}
\]
with each element > x copied to the right as it is compared against x.

The most common variant, which operates on arrays, can be described as:

1. Suppose we have a method called insert designed to insert a value into a sorted sequence at the beginning of an array. It operates by starting at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. It has the side effect of overwriting the value stored immediately after the sorted sequence in the array.
2. To perform insertion sort, start at the left end of the array and invoke insert to insert each element encountered into its correct position. The ordered sequence into which we insert it is stored at the beginning of the array in the set of indexes already examined. Each insertion overwrites a single value, but this is okay because it's the value we're inserting.

A simple pseudocode version of the complete algorithm follows, where the arrays are zero-based:

```pseudocode
insertionSort(array A)
for i <- 1 to length[A]-1 do
    value <- A[i]
    j <- i-1
    while j >= 0 and A[j] > value do
        j <- j-1
    A[j+1] <- value
```

**Good and bad input cases**

In the best case of an already sorted array, this implementation of insertion sort takes \(O(n)\) time: in each iteration, the first remaining element of the input is only compared with the last element of the sorted subsection of the array. This same case provides worst-case behavior for non-randomized and poorly implemented quicksort, which will take \(O(n^2)\) time to sort an already-sorted list. Thus, if an array is sorted or nearly sorted, insertion sort will significantly outperform quicksort.
The worst case is an array sorted in reverse order, as every execution of the inner loop will have to scan and shift the entire sorted section of the array before inserting the next element. Insertion sort takes $O(n^2)$ time in this worst case as well as in the average case, which makes it impractical for sorting large numbers of elements. However, insertion sort's inner loop is very fast, which often makes it one of the fastest algorithms for sorting small numbers of elements, typically less than 10 or so.

**Comparisons to other sorts**

Insertion sort is very similar to selection sort. Just like in selection sort, after $k$ passes through the array, the first $k$ elements are in sorted order. For selection sort, these are the $k$ smallest elements, while in insertion sort they are whatever the first $k$ elements were in the unsorted array. Insertion sort's advantage is that it only scans as many elements as it needs to in order to place the $k + 1$st element, while selection sort must scan all remaining elements to find the absolute smallest element.

Simple calculation shows that insertion sort will therefore usually perform about half as many comparisons as selection sort. Assuming the $k + 1$st element's rank is random, it will on the average require shifting half of the previous $k$ elements over, while selection sort always requires scanning all unplaced elements. If the array is not in a random order, however, insertion sort can perform just as many comparisons as selection sort (for a reverse-sorted list). It will also perform far fewer comparisons, as few as $n - 1$, if the data is pre-sorted, thus insertion sort is much more efficient if the array is already sorted or "close to sorted." It can be seen as an advantage for some real-time applications that selection sort will perform identically regardless of the order of the array, while insertion sort's running time can vary considerably.

While insertion sort typically makes fewer comparisons than selection sort, it requires more writes because the inner loop can require shifting large sections of the sorted portion of the array. In general, insertion sort will write to the array $O(n^2)$ times while selection sort will write only $O(n)$ times. For this reason, selection sort may be better in cases where writes to memory are significantly more expensive than reads, such as EEPROM or Flash memory.

Some divide-and-conquer algorithms such as quicksort and mergesort sort by recursively dividing the list into smaller sublists which are then sorted. A useful optimization in practice for these algorithms is to switch to insertion sort for "sorted enough" sublists on which insertion sort outperforms the more complex algorithms. The size of list for which insertion sort has the advantage varies by environment and implementation, but is typically around 8 to 20 elements.
Variants

D.L. Shell made substantial improvements to the algorithm, and the modified version is called Shell sort. It compares elements separated by a distance that decreases on each pass. Shell sort has distinctly improved running times in practical work, with two simple variants requiring $O(n^{3/2})$ and $O(n^{4/3})$ time.

If comparisons are very costly compared to swaps, as is the case for example with string keys stored by reference or with human interaction (such as choosing one of a pair displayed side-by-side), then using binary insertion sort can be a good strategy. Binary insertion sort employs binary search to find the right place to insert new elements, and therefore performs $\lceil \log_2(n!) \rceil$ comparisons in the worst case, which is $\Theta(n \log n)$. The algorithm as a whole still takes $\Theta(n^2)$ time on average due to the series of swaps required for each insertion, and since it always uses binary search, the best case is no longer $\Omega(n)$ but $\Omega(n \log n)$.

To avoid having to make a series of swaps for each insertion, we could instead store the input in a linked list, which allows us to insert and delete elements in constant time. Unfortunately, binary search on a linked list is impossible, so we still spend $O(n^2)$ time searching. If we instead replace it by a more sophisticated data structure such as a heap or binary tree, we can significantly decrease both search and insert time. This is the essence of heap sort and binary tree sort.

In 2004, Bender, Farach-Colton, and Mosteiro published a new variant of insertion sort called library sort or gapped insertion sort that leaves a small number of unused spaces ("gaps") spread throughout the array. The benefit is that insertions need only shift elements over until a gap is reached. Surprising in its simplicity, they show that this sorting algorithm runs with high probability in $O(n \log n)$ time.

Examples

c++ Example:

```c++
#include <iostream>

using namespace std;

bool swap(int&, int&); //Swaps Two Ints
```
void desc(int* ar, int); //Nothing Just Shows The Array Visually

int ins_sort(int*, int); //The Insertion Sort Function

int main()
{
    int array[9] = {4, 3, 5, 1, 2, 0, 7, 9, 6}; //The Original Array
    desc(array, 9);
    *array = ins_sort(array, 9);
    cout << "Array Sorted Press Enter To Continue and See the Resultant Array" << endl
    << "-------------------8<-------------------------------->8--------------";
    getchar();
    desc(array, 9);
    getchar();
    return 0;
}

int ins_sort(int* array, int len)
{
    for (int i = 0; i < len; i++)
    {
        int val = array[i];
        int key = i;
        cout << "key(Key) = " << key << "\tval(Value) = " << val << endl;
        for (; key >= 1 && array[key-1] >= val; --key)
{  
cout << "Swapping Backward\tfrom (key) " << key << " of (Value) " << array[key] << "\ttto (key) " << key-1 << " of (Value) " << array[key-1];  
cout << "\n\t" << key << "\n\t<----> " << key-1 << "\n\t( " << array[key] << "<----> " << array[key-1] << " )";  
swap(array[key], array[key-1]);  
desc(array, 9);  
}

return *array;

}

bool swap(int& pos1, int& pos2)
{
    int tmp = pos1;
    pos1 = pos2;
    pos2 = tmp;
    return true;
}

void desc(int* ar, int len)
{
    cout << endl << "Describing The Given Array" << endl;
    for (int i = 0; i < len; i++)
cout << " ________" << endl;
for (int i = 0; i < len; i++)
cout << " | " << i << " | " << "\t";  
cout << endl;
for (int i = 0; i < len; i++)
cout << " ( " << ar[i] << " ) " << "\t";  
cout << endl;
for (int i = 0; i < len; i++)
cout << "-------" << "\t";
getchar();
}

Python Example:

def insertion_sort(A):
for i in range(1, len(A)):
    key = A[i]
    j = i-1
    while(j >= 0 and A[j] > key):
        j = j-1
    A[j+1] = key
6.1.2. Selection sort

(From Wikipedia, the free encyclopedia)

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has \(\Theta(n^2)\) complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations. It works as follows:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for remainder of the list (starting at the second position)

Effectively, we divide the list into two parts: the sublist of items already sorted, which we build up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array.

Here is an example of this sort algorithm sorting five elements:

31 25 12 22 11
11 25 12 22 31
11 12 25 22 31
11 12 22 25 31

Selection sort can also be used on list structures that make add and remove efficient, such as a linked list. In this case it's more common to remove the minimum element from the remainder of the list, and then insert it at the end of the values sorted so far. For example:

31 25 12 22 11
11 31 25 12 22
11 12 31 25 22
11 12 22 31 25
11 12 22 25 31
**Implementation**

The following is a C/C++ implementation, which makes use of a `swap` function:

```c
void selectionSort(int a[], int size)
{
    int i, j, min;
    for (i = 0; i < size - 1; i++)
    {
        min = i;
        for (j = i+1; j < size; j++)
        {
            if (a[j] < a[min])
            {
                min = j;
            }
        }
        swap(a[i], a[min]);
    }
}
```

Python example:

```python
def selection_sort(A):
    for i in range(0, len(A)-1):
        min = A[i]
```
pos = i

for j in range(i+1, len(A)):
    if( A[j] < min ):
        min = A[j]
        pos = j
A[i] = min

**Analysis**

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takes n - 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n - 1 elements and so on, for \( (n - 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1) / 2 = \Theta(n^2) \) comparisons (see arithmetic progression). Each of these scans requires one swap for n - 1 elements (the final element is already in place). Thus, the comparisons dominate the running time, which is \( \Theta(n^2) \).

**Comparison to other Sorting Algorithms**

Among simple average-case \( \Theta(n^2) \) algorithms, selection sort always outperforms bubble sort and gnome sort, but is generally outperformed by insertion sort. Insertion sort is very similar in that after the kth iteration, the first k elements in the array are in sorted order. Insertion sort's advantage is that it only scans as many elements as it needs to in order to place the k + 1st element, while selection sort must scan all remaining elements to find the k + 1st element.

Simple calculation shows that insertion sort will therefore usually perform about half as many comparisons as selection sort, although it can perform just as many or far fewer depending on the order the array was in prior to sorting. It can be seen as an advantage for some real-time applications that selection sort will perform identically regardless of the order of the array, while insertion sort's running time can vary considerably. However, this is more often an advantage for insertion sort in that it runs much more efficiently if the array is already sorted or "close to sorted."
Another key difference is that selection sort always performs $\Theta(n)$ swaps, while insertion sort performs $\Theta(n^2)$ swaps in the average and worst cases. Because swaps require writing to the array, selection sort is preferable if writing to memory is significantly more expensive than reading, such as when dealing with an array stored in EEPROM or Flash.

Finally, selection sort is greatly outperformed on larger arrays by $\Theta(n \log n)$ divide-and-conquer algorithms such as quicksort and mergesort. However, insertion sort or selection sort are both typically faster for small arrays (ie less than 10-20 elements). A useful optimization in practice for the recursive algorithms is to switch to insertion sort or selection sort for "small enough" sublists.

**Variants**

Heap sort greatly improves the basic algorithm by using an implicit heap data structure to speed up finding and removing the lowest datum. If implemented correctly, the heap will allow finding the next lowest element in $\Theta(\log n)$ time instead of $\Theta(n)$ for the inner loop in normal selection sort, reducing the total running time to $\Theta(n \log n)$.

A bidirectional variant of selection sort, called cocktail sort, is an algorithm which finds both the minimum and maximum values in the list in every pass. This reduces the number of scans of the list by a factor of 2, eliminating some loop overhead but not actually decreasing the number of comparisons or swaps. Note, however, that cocktail sort more often refers to a bidirectional variant of bubble sort.

Selection sort can be implemented as a stable sort. If, rather than swapping in step 2, the minimum value is inserted into the first position (that is, all intervening items moved down), the algorithm is stable. However, this modification leads to $\Theta(n^2)$ writes, eliminating the main advantage of selection sort over insertion sort, which is always stable.

**6.1.3. Bubble sort**

(From Wikipedia, the free encyclopedia)

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (i.e. the beginning) of the list via the swaps. (Another opinion: it gets its name from the way greater elements "bubble" to the end.) Because it only uses comparisons to operate on elements, it is a comparison sort. This is the easiest comparison sort to implement.
A simple way to express bubble sort in pseudocode is as follows:

procedure bubbleSort( A : list of sortable items ) defined as:

do
swapped := false

for each i in 0 to length( A ) - 2 do:
if A[i] > A[i + 1] then
swap( A[i], A[i + 1] )
swapped := true
end if
end for
end while swapped
end procedure

The algorithm can also be expressed as:

procedure bubbleSort( A : list of sortable items ) defined as:

for each i in 1 to length(A) do:
for each j in length(A) downto i + 1 do:
swap( A[j], A[j - 1] )
end if
end for
end for
end procedure
This difference between this and the first pseudocode implementation is discussed later in the article.

**Analysis**

**Best-case performance**

Bubble sort has best-case complexity $\Omega(n)$. When a list is already sorted, bubblesort will pass through the list once, and find that it does not need to swap any elements. Thus bubble sort will make only $n$ comparisons and determine that list is completely sorted. It will also use considerably less time than $O(n^2)$ if the elements in the unsorted list are not too far from their sorted places. MKH...

**Rabbits and turtles**

The positions of the elements in bubble sort will play a large part in determining its performance. Large elements at the top of the list do not pose a problem, as they are quickly swapped downwards. Small elements at the bottom, however, as mentioned earlier, move to the top extremely slowly. This has led to these types of elements being named rabbits and turtles, respectively.

Various efforts have been made to eliminate turtles to improve upon the speed of bubble sort. Cocktail sort does pretty well, but it still retains $O(n^2)$ worst-case complexity. Comb sort compares elements large gaps apart and can move turtles extremely quickly, before proceeding to smaller and smaller gaps to smooth out the list. Its average speed is comparable to faster algorithms like Quicksort.

**Alternative implementations**

One way to optimize bubble sort is to note that, after each pass, the largest element will always move down to the bottom. During each comparison, it is clear that the largest element will move downwards. Given a list of size $n$, the $n$th element will be guaranteed to be in its proper place. Thus it suffices to sort the remaining $n - 1$ elements. Again, after this pass, the $n - 1$th element will be in its final place.

In pseudocode, this will cause the following change:

procedure bubbleSort( A : list of sortable items ) defined as:

```
n := length( A )

for i := 1 to n-1 do
```


swapped := false

n := n - 1

for each i in 0 to n do:

if A[ i ] > A[ i + 1 ] then

swap( A[ i ], A[ i + 1 ] )

swapped := true

end if

end for

while swapped

end procedure

We can then do bubbling passes over increasingly smaller parts of the list. More precisely, instead of doing $n^2$ comparisons (and swaps), we can use only $n + (n-1) + (n-2) + ... + 1$ comparisons. This sums up to $n(n + 1) / 2$, which is still $O(n^2)$, but which can be considerably faster in practice.

In practice

Although bubble sort is one of the simplest sorting algorithms to understand and implement, its $O(n^2)$ complexity means it is far too inefficient for use on lists having more than a few elements. Even among simple $O(n^2)$ sorting algorithms, algorithms like insertion sort are usually considerably more efficient.

Due to its simplicity, bubble sort is often used to introduce the concept of an algorithm, or a sorting algorithm, to introductory computer science students. However, some researchers such as Owen Astrachan have gone to great lengths to disparage bubble sort and its continued popularity in computer science education, recommending that it no longer even be taught.

The Jargon file, which famously calls bogosort "the archetypical perversely awful algorithm", also calls bubble sort "the generic bad algorithm". Donald Knuth, in his famous The Art of Computer Programming, concluded that "the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems", some of which he discusses therein.
Bubble sort is asymptotically equivalent in running time to insertion sort in the worst case, but the two algorithms differ greatly in the number of swaps necessary. Experimental results such as those of Astrachan have also shown that insertion sort performs considerably better even on random lists. For these reasons many modern algorithm textbooks avoid using the bubble sort algorithm in favor of insertion sort.

Bubble sort also interacts poorly with modern CPU hardware. It requires at least twice as many writes as insertion sort, twice as many cache misses, and asymptotically more branch mispredictions. Experiments by Astrachan sorting strings in Java show bubble sort to be roughly 5 times slower than insertion sort and 40% slower than selection sort.

Effectively sorting algorithms

Shell sort

(From Wikipedia, the free encyclopedia)

Shell sort is a sorting algorithm that is a generalization of insertion sort, with two observations:

- insertion sort is efficient if the input is "almost sorted", and
- insertion sort is typically inefficient because it moves values just one position at a time.

Implementation

The original implementation performs \( \Theta(n^2) \) comparisons and exchanges in the worst case. A minor change given in V. Pratt's book improved the bound to \( O(n \log_2 n) \). This is worse than the optimal comparison sorts, which are \( O(n \log n) \).

Shell sort improves insertion sort by comparing elements separated by a gap of several positions. This lets an element take "bigger steps" toward its expected position. Multiple passes over the data are taken with smaller and smaller gap sizes. The last step of Shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted.

Consider a small value that is initially stored in the wrong end of the array. Using an \( O(n^2) \) sort such as bubble sort or insertion sort, it will take roughly \( n \) comparisons and exchanges to move this value all the way to the other end of the array. Shell sort first moves values using giant step sizes, so a small value will move a long way towards its final position, with just a few comparisons and exchanges.
One can visualize Shellsort in the following way: arrange the list into a table and sort the columns (using an insertion sort). Repeat this process, each time with smaller number of longer columns. At the end, the table has only one column. While transforming the list into a table makes it easier to visualize, the algorithm itself does its sorting in-place (by incrementing the index by the step size, i.e. using \( i += \text{step\_size} \) instead of \( i++ \)).

For example, consider a list of numbers like \([ 13 \ 14 \ 94 \ 33 \ 82 \ 25 \ 59 \ 94 \ 65 \ 23 \ 45 \ 27 \ 73 \ 25 \ 39 \ 10 ]\). If we started with a step-size of 5, we could visualize this as breaking the list of numbers into a table with 5 columns. This would look like this:

```
13 14 94 33 82
25 59 94 65 23
45 27 73 25 39
10
```

We then sort each column, which gives us

```
10 14 73 25 23
13 27 94 33 39
25 59 94 65 82
45
```

When read back as a single list of numbers, we get \([ 10 \ 14 \ 73 \ 25 \ 23 \ 13 \ 27 \ 94 \ 33 \ 39 \ 25 \ 59 \ 94 \ 65 \ 82 \ 45 ]\). Here, the 10 which was all the way at the end, has moved all the way to the beginning. This list is then again sorted using a 3-gap sort, and then 1-gap sort (simple insertion sort).

**Gap sequence**

![The shellsort algorithm in action](image)
The gap sequence is an integral part of the shellsort algorithm. Any increment sequence will work, so long as the last element is 1. The algorithm begins by performing a gap insertion sort, with the gap being the first number in the gap sequence. It continues to perform a gap insertion sort for each number in the sequence, until it finishes with a gap of 1. When the gap is 1, the gap insertion sort is simply an ordinary insertion sort, guaranteeing that the final list is sorted.

The gap sequence that was originally suggested by Donald Shell was to begin with $N / 2$ and to halve the number until it reaches 1. While this sequence provides significant performance enhancements over the quadratic algorithms such as insertion sort, it can be changed slightly to further decrease the average and worst-case running times. Weiss' textbook[4] demonstrates that this sequence allows a worst case $O(n^2)$ sort, if the data is initially in the array as (small_1, large_1, small_2, large_2, ...), that is, the upper half of the numbers are placed, in sorted order, in the even index locations and the lower end of the numbers are placed similarly in the odd indexed locations.

Perhaps the most crucial property of Shellsort is that the elements remain k-sorted even as the gap diminishes. For instance, if a list was 5-sorted and then 3-sorted, the list is now not only 3-sorted, but both 5- and 3-sorted. If this were not true, the algorithm would undo work that it had done in previous iterations, and would not achieve such a low running time.

Depending on the choice of gap sequence, Shellsort has a proven worst-case running time of $O(n^2)$ (using Shell's increments that start with 1/2 the array size and divide by 2 each time), $O(n^3 / 2)$ (using Hibbard's increments of $2k - 1$), $O(n^4 / 3)$ (using Sedgewick's increments of $9(4i) - 9(2i) + 1$, or $4i + 1 + 3(2i) + 1$), or $O(n \log 2n)$, and possibly unproven better running times. The existence of an $O(n \log n)$ worst-case implementation of Shellsort remains an open research question.

The best known sequence is 1, 4, 10, 23, 57, 132, 301, 701. Such a Shell sort is faster than an insertion sort and a heap sort, but if it is faster than a quicksort for small arrays (less than 50 elements), it is slower for bigger arrays. Next gaps can be computed for instance with:

$$nextgap = \text{round}(gap \times 2.3)$$

**Shell sort algorithm in C/C++**

Shell sort is commonly used in programming languages; this is an implementation of the algorithm in C/C++ for sorting an array of integers. The increment sequence used in this example code gives an $O(n^2)$ worst-case running time.

```c
void shell_sort(int A[], int size)
```
{ 
    int i, j, increment, temp;
    
    increment = size / 2;
    
    while (increment > 0)
    {
        for (i=increment; i < size; i++)
        {
            j = i;
            temp = A[i];
            
            while ((j >= increment) && (A[j-increment] > temp))
            {
                j = j - increment;
            }
            A[j] = temp;
        }
        if (increment == 2)
            increment = 1;
        else
            increment = (int) (increment / 2.2);
    }
}
Shell sort algorithm in Java

The Java implementation of Shell sort is as follows:

```java
public static void shellSort(int[] a) {
    for (int increment = a.length / 2; increment > 0;
         increment = (increment == 2 ? 1 : (int) Math.round(increment / 2.2))) {
        for (int i = increment; i < a.length; i++) {
            for (int j = i; j >= increment && a[j - increment] > a[j]; j -= increment) {
                int temp = a[j];
                a[j] = a[j - increment];
                a[j - increment] = temp;
            }
        }
    }
}
```

Shell sort algorithm in Python

Here it is:

```python
def shellsort(a):
    def new_increment(a):
        i = int(len(a) / 2)
        yield i
        while i != 1:
            i = int(len(a) / 2)
            yield i
```
if i == 2:
    i = 1
else:
    i = int(numpy.round(i/2.2))
    yield i
for increment in new_increment(a):
    for i in xrange(increment, len(a)):
        for j in xrange(i, increment-1, -increment):
            if a[j - increment] < a[j]:
                break
            temp = a[j];
            a[j] = a[j - increment]
            a[j - increment] = temp
    return a

6.2.2. Heap sort

(From Wikipedia, the free encyclopedia)

Heapsort is a comparison-based sorting algorithm, and is part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of quicksort, it has the advantage of a worst-case $O(n \log n)$ runtime. Heapsort is an in-place algorithm, but is not a stable sort.

Overview

Heapsort inserts the input list elements into a heap data structure. The largest value (in a max-heap) or the smallest value (in a min-heap) are extracted until none remain, the values having been extracted in sorted order. The heap's invariant is preserved after each extraction, so the only cost is that of extraction.
During extraction, the only space required is that needed to store the heap. In order to achieve constant space overhead, the heap is stored in the part of the input array that has not yet been sorted. (The structure of this heap is described at Binary heap: Heap implementation.)

Heapsort uses two heap operations: insertion and root deletion. Each extraction places an element in the last empty location of the array. The remaining prefix of the array stores the unsorted elements.

**Variations**

- The most important variation to the simple variant is an improvement by R.W.Floyd which gives in practice about 25% speed improvement by using only one comparison in each siftdown run which then needs to be followed by a siftup for the original child; moreover it is more elegant to formulate. Heapsort's natural way of indexing works on indices from 1 up to the number of items. Therefore the start address of the data should be shifted such that this logic can be implemented avoiding unnecessary +/- 1 offsets in the coded algorithm.
- Ternary heapsort uses a ternary heap instead of a binary heap; that is, each element in the heap has three children. It is more complicated to program, but does a constant number of times fewer swap and comparison operations. This is because each step in the shift operation of a ternary heap requires three comparisons and one swap, whereas in a binary heap two comparisons and one swap are required. The ternary heap does two steps in less time than the binary heap requires for three steps, which multiplies the index by a factor of 9 instead of the factor 8 of three binary steps. Ternary heapsort is about 12% faster than the simple variant of binary heapsort.[citation needed]
- The smoothsort sorting algorithm is a variation of heapsort developed by Edsger Dijkstra in 1981. Like heapsort, smoothsort's upper bound is $O(n \log n)$. The advantage of smoothsort is that it comes closer to $O(n)$ time if the input is already sorted to some degree, whereas heapsort averages $O(n \log n)$ regardless of the initial sorted state. Due to its complexity, smoothsort is rarely used.

**Comparison with other sorts**

Heapsort primarily competes with quicksort, another very efficient general purpose nearly-in-place comparison-based sort algorithm.

Quicksort is typically somewhat faster, due to better cache behavior and other factors, but the worst-case running time for quicksort is $O(n^2)$, which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the
implementation, creating a security risk. See quicksort for a detailed discussion of this problem, and possible solutions.

Thus, because of the $O(n \log n)$ upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort.

Heapsort also competes with merge sort, which has the same time bounds, but requires $\Omega(n)$ auxiliary space, whereas heapsort requires only a constant amount. Heapsort also typically runs more quickly in practice on machines with small or slow data caches. On the other hand, merge sort has several advantages over heapsort:

- Like quicksort, merge sort on arrays has considerably better data cache performance, often outperforming heapsort on a modern desktop PC, because it accesses the elements in order.
- Merge sort is a stable sort.
- Merge sort parallelizes better; the most trivial way of parallelizing merge sort achieves close to linear speedup, while there is no obvious way to parallelize heapsort at all.
- Merge sort can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Heapsort relies strongly on random access, and its poor locality of reference makes it very slow on media with long access times.

An interesting alternative to Heapsort is Introsort which combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

**Pseudocode**

The following is the "simple" way to implement the algorithm, in pseudocode, where swap is used to swap two elements of the array. Notice that the arrays are zero based in this example.

function heapSort(a, count) is

input: an unordered array a of length count

(first place a in max-heap order)

heapify(a, count)

end := count - 1
while end > 0 do

(swap the root(maximum value) of the heap with the last element of the heap)

swap(a[end], a[0])

(decrease the size of the heap by one so that the previous max value will stay in its proper placement)

end := end - 1

(put the heap back in max-heap order)

siftDown(a, 0, end)

function heapify(a,count) is

(start is assigned the index in a of the last parent node)

start := count ÷ 2 - 1

while start ≥ 0 do

(sift down the node at index start to the proper place such that all nodes below the start index are in heap order)

siftDown(a, start, count-1)

start := start - 1

(after sifting down the root all nodes/elements are in heap order)

function siftDown(a, start, end) is

input: end represents the limit of how far down the heap to sift.

root := start

while root * 2 + 1 ≤ end do (While the root has at least one child)
child := root * 2 + 1 (root*2+1 points to the left child)

(If the child has a sibling and the child's value is less than its sibling's...)

if child < end and a[child] < a[child + 1] then

child := child + 1 (... then point to the right child instead)

if a[root] < a[child] then (out of max-heap order)

swap(a[root], a[child])

root := child (repeat to continue sifting down the child now)

else

return

The heapify function can be thought of as successively inserting into the heap and sifting up. The two versions only differ in the order of data processing. The above heapify function starts at the bottom and moves up while sifting down (bottom-up). The following heapify function starts at the top and moves down while sifting up (top-down).

function heapify(a,count) is

(end is assigned the index of the first (left) child of the root)

end := 1

while end < count

(sift up the node at index end to the proper place such that all nodes above
the end index are in heap order)

siftUp(a, 0, end)

end := end + 1

(after sifting up the last node all nodes are in heap order)

function siftUp(a, start, end) is

input: start represents the limit of how far up the heap to sift.
end is the node to sift up.

child := end

while child > start

parent := ⌊(child - 1) ÷ 2⌋

if a[parent] < a[child] then (out of max-heap order)

swap(a[parent], a[child])

child := parent (repeat to continue sifting up the parent now)

else

return

It can be shown that both variants of heapify run in O(n) time.[citation needed]

C-code

Below is an implementation of the "standard" heapsort (also called bottom-up-heapsort). It is faster on average (see Knuth. Sec. 5.2.3, Ex. 18) and even better in worst-case behavior (1.5n log n) than the simple heapsort (2n log n). The sift_in routine is first a sift_up of the free position followed by a sift_down of the new item. The needed data-comparison is only in the macro data_i_LESS_THAN_ for easy adaption.

This code is flawed - see talk page

/* Heapsort based on ideas of J.W.Williams/R.W.Floyd/S.Carlsson */

#define data_i_LESS_THAN_(other) (data[i] < other)

#define MOVE_i_TO_free { data[free]=data[i]; free=i; }

void sift_in(unsigned count, SORTTYPE *data, unsigned free_in, SORTTYPE next)
{
    unsigned i;

    unsigned free = free_in;

// sift up the free node
for (i=2*free;i<count;i+=i)
{
    if (data_i_LESS_THAN_(data[i+1])) i++;
    MOVE_i_TO_free
}

// special case in sift up if the last inner node has only 1 child
if (i==count)
    MOVE_i_TO_free

// sift down the new item next
while( ((i=free/2)>=free_in) && data_i_LESS_THAN_(next))
    MOVE_i_TO_free

    data[free] = next;

} // end heapsort

void heapsort(unsigned count, SORTTYPE *data)
{
    unsigned j;

    if (count <= 1) return;

    data-=1; // map addresses to indices 1 til count

    // build the heap structure
    for(j=count / 2; j>=1; j--)
    {
        SORTTYPE next = data[j];
        sift_in(count, data, j, next);
// search next by next remaining extremal element

for(j= count - 1; j>=1; j--) {
    SORTTYPE next = data[j + 1];
    data[j + 1] = data[1]; // extract extremal element from the heap
    sift_in(j, data, 1, next);
}

6.2.3. Quicksort

(From Wikipedia, the free encyclopedia)

Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare that, on average, makes $\Theta(n \log n)$ (big O notation) comparisons to sort n items. However, in the worst case, it makes $\Theta(n^2)$ comparisons. Typically, quicksort is significantly faster in practice than other $\Theta(n \log n)$ algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices which minimize the possibility of requiring quadratic time.

Quicksort is a comparison sort and is not a stable sort.

The algorithm

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.
The base case of the recursion are lists of size zero or one, which are always sorted. The algorithm always terminates because it puts at least one element in its final place on each iteration (the loop invariant).

In simple pseudocode, the algorithm might be expressed as:

function quicksort(array)
    var list less, pivotList, greater
    if length(array) ≤ 1
        return array
    select a pivot value pivot from array
    for each x in array
        if x < pivot then add x to less
        if x = pivot then add x to pivotList
        if x > pivot then add x to greater
    return concatenate(quicksort(less), pivotList, quicksort(greater))

Notice that we only examine elements by comparing them to other elements. This makes quicksort a comparison sort.

**Version with in-place partition**

[Diagram of quicksort partitioning process]
In-place partition in action on a small list. The boxed element is the pivot element, blue elements are less or equal, and red elements are larger.

The disadvantage of the simple version above is that it requires $\Omega(n)$ extra storage space, which is as bad as mergesort (see big-O notation for the meaning of $\Omega$). The additional memory allocations required can also drastically impact speed and cache performance in practical implementations. There is a more complicated version which uses an in-place partition algorithm and can achieve $O(\log n)$ space use on average for good pivot choices:

```
function partition(array, left, right, pivotIndex)
    pivotValue := array[pivotIndex]
    swap(array, pivotIndex, right) // Move pivot to end
    storeIndex := left - 1
    for i from left to right-1
        if array[i] <= pivotValue
            storeIndex := storeIndex + 1
            swap(array, storeIndex, i)
    swap(array, right, storeIndex+1) // Move pivot to its final place
    return storeIndex+1
```

This form of the partition algorithm is not the original form; multiple variations can be found in various textbooks, such as versions not having the storeIndex. However, this form is probably the easiest to understand.

This is the in-place partition algorithm. It partitions the portion of the array between indexes left and right, inclusively, by moving all elements less than or equal to a[pivotIndex] to the beginning of the subarray, leaving all the greater elements following them. In the process it also finds the final position for the pivot element, which it returns. It temporarily moves the pivot element to the end of the subarray, so that it doesn't get in the way. Because it only uses exchanges, the final list has the same elements as the original list. Notice that an element may be exchanged multiple times before reaching its final place.

Once we have this, writing quicksort itself is easy:
function quicksort(array, left, right)

if right > left

select a pivot index (e.g. pivotIndex := left)

pivotNewIndex := partition(array, left, right, pivotIndex)

quicksort(array, left, pivotNewIndex-1)

quicksort(array, pivotNewIndex+1, right)

**Parallelization**

Like **mergesort**, quicksort can also be easily **parallelized** due to its divide-and-conquer nature. Individual in-place partition operations are difficult to parallelize, but once divided, different sections of the list can be sorted in parallel. If we have p processors, we can divide a list of n elements into p sublists in $\Theta(n)$ average time, then sort each of these in $O\left(\frac{n}{p} \log \frac{n}{p}\right)$ average time. Ignoring the $O(n)$ preprocessing, this is **linear speedup**. Given $\Theta(n)$ processors, only $O(n)$ time is required overall.

One advantage of parallel quicksort over other parallel sort algorithms is that no synchronization is required. A new thread is started as soon as a sublist is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.

Other more sophisticated parallel sorting algorithms can achieve even better time bounds. For example, in 1991 David Powers described a parallelized quicksort that can operate in $O(\log n)$ time given enough processors by performing partitioning implicitly[1].

**Formal analysis**

From the initial description it's not obvious that quicksort takes $O(n \log n)$ time on average. It's not hard to see that the partition operation, which simply loops over the elements of the array once, uses $\Theta(n)$ time. In versions that perform concatenation, this operation is also $\Theta(n)$.

In the best case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only $(\log n)$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $O(\log n)$. But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls needs only $O(n)$ time.
all together (each call has some constant overhead, but since there are only O(n) calls at each level, this is subsumed in the O(n) factor). The result is that the algorithm uses only O(n log n) time.

An alternate approach is to set up a recurrence relation for T(n) factor), the time needed to sort a list of size n. Because a single quicksort call involves O(n) factor) work plus two recursive calls on lists of size n/2 in the best case, the relation would be:

\[ T(n) = \mathcal{O}(n) + 2T\left(\frac{n}{2}\right). \]

The master theorem tells us that \( T(n) = \Theta(n \log n) \).

In fact, it's not necessary to divide the list this precisely; even if each pivot splits the elements with 99% on one side and 1% on the other (or any other fixed fraction), the call depth is still limited to \((100 \log n)\), so the total running time is still \( O(n \log n) \).

In the worst case, however, the two sublists have size 1 and \( n - 1 \), and the call tree becomes a linear chain of \( n \) nested calls. The \( i \)th call does \( O(n-i) \) work, and\

\[ \sum_{i=0}^{n}(n - i) = \mathcal{O}(n^2). \]

\[ T(n) = \mathcal{O}(n) + T(1) + T(n-1) = \mathcal{O}(n) + T(n-1) \]

This is the same relation as for insertion sort and selection sort, and it solves to \( T(n) = \Theta(n^2) \).

**Randomized quicksort expected complexity**

Randomized quicksort has the desirable property that it requires only \( O(n \log n) \) expected time, regardless of the input. But what makes random pivots a good choice?

Suppose we sort the list and then divide it into four parts. The two parts in the middle will contain the best pivots; each of them is larger than at least 25% of the elements and smaller than at least 25% of the elements. If we could consistently choose an element from these two middle parts, we would only have to split the list at most \( 2 \log_2 n \) times before reaching lists of size 1, yielding an \( O(n \log n) \) algorithm.

Unfortunately, a random choice will only choose from these middle parts half the time. The surprising fact is that this is good enough. Imagine that you are flipping a coin over and over until you get \( k \) heads. Although this could take a long time, on average only \( 2k \) flips are required, and the chance that you won't get \( k \) heads after \( 100k \) flips is infinitesimally small. By the same argument, quicksort's recursion will terminate on average at a call depth of only \( 2 \log_2 n \). But if its average call depth is \( O(\log n) \), and each
level of the call tree processes at most $n$ elements, the total amount of work done on average is the product, $O(n \log n)$.

**Average complexity**

Even if we aren't able to choose pivots randomly, quicksort still requires only $O(n \log n)$ time over all possible permutations of its input. Because this average is simply the sum of the times over all permutations of the input divided by $n$ factorial, it's equivalent to choosing a random permutation of the input. When we do this, the pivot choices are essentially random, leading to an algorithm with the same running time as randomized quicksort.

More precisely, the average number of comparisons over all permutations of the input sequence can be estimated accurately by solving the recurrence relation:

$$C(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (C(i) + C(n - i - 1)) = 2n \ln n = 1.39n \log_2 n.$$ 

Here, $n - 1$ is the number of comparisons the partition uses. Since the pivot is equally likely to fall anywhere in the sorted list order, the sum is averaging over all possible splits.

This means that, on average, quicksort performs only about 39% worse than the ideal number of comparisons, which is its best case. In this sense it is closer to the best case than the worst case. This fast average runtime is another reason for quicksort's practical dominance over other sorting algorithms.

$$C(n) = (n-1) + C(n/2) + C(n/2)$$

$$= (n-1) + 2C(n/2)$$

$$= (n-1) + 2((n/2) - 1 + 2C(n/4))$$

$$= n + n + 4C(n/4) - 1 - 2$$

$$= n + n + n + 8C(n/8) - 1 - 2 - 4$$

$$= ...$$

$$= kn + 2^kC(n/(2^k)) - (1 + 2 + 4 + \ldots + 2^{(k-1)})$$

where $\log_2 n > k > 0$

$$= kn + 2^kC(n/(2^k)) - 2^k + 1$$
-> n\log_2 n + nC(1) - n + 1.

**Space complexity**

The space used by quicksort depends on the version used.

Quicksort has a space complexity of $O(\log n)$, even in the worst case, when it is carefully implemented such that

- in-place partitioning is used. This requires $O(1)$.
- After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most $O(\log n)$ space. Then the other partition is sorted using tail-recursion or iteration.

The version of quicksort with in-place partitioning uses only constant additional space before making any recursive call. However, if it has made $O(\log n)$ nested recursive calls, it needs to store a constant amount of information from each of them. Since the best case makes at most $O(\log n)$ nested recursive calls, it uses $O(\log n)$ space. The worst case makes $O(n)$ nested recursive calls, and so needs $O(n)$ space.

We are eliding a small detail here, however. If we consider sorting arbitrarily large lists, we have to keep in mind that our variables like left and right can no longer be considered to occupy constant space; it takes $O(\log n)$ bits to index into a list of $n$ items. Because we have variables like this in every stack frame, in reality quicksort requires $O(\log 2 n)$ bits of space in the best and average case and $O(n \log n)$ space in the worst case. This isn't too terrible, though, since if the list contains mostly distinct elements, the list itself will also occupy $O(\log n)$ bits of space.

The not-in-place version of quicksort uses $O(n)$ space before it even makes any recursive calls. In the best case its space is still limited to $O(n)$, because each level of the recursion uses half as much space as the last, and

$$
\sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.
$$

Its worst case is dismal, requiring

$$
\sum_{i=0}^{n} (n - i + 1) = \Theta(n^2)
$$

space, far more than the list itself. If the list elements are not themselves constant size, the problem grows even larger; for example, if most of the list elements are distinct,
each would require about $O(\log n)$ bits, leading to a best-case $O(n \log n)$ and worst-case $O(n^2 \log n)$ space requirement.

**Selection-based pivoting**

A selection algorithm chooses the kth smallest of a list of numbers; this is an easier problem in general than sorting. One simple but effective selection algorithm works nearly in the same manner as quicksort, except that instead of making recursive calls on both sublists, it only makes a single tail-recursive call on the sublist which contains the desired element. This small change lowers the average complexity to linear or $\Theta(n)$ time, and makes it an in-place algorithm. A variation on this algorithm brings the worst-case time down to $O(n)$ (see selection algorithm for more information).

Conversely, once we know a worst-case $O(n)$ selection algorithm is available, we can use it to find the ideal pivot (the median) at every step of quicksort, producing a variant with worst-case $O(n \log n)$ running time. In practical implementations, however, this variant is considerably slower on average.

**Competitive sorting algorithms**

Quicksort is a space-optimized version of the binary tree sort. Instead of inserting items sequentially into an explicit tree, quicksort organizes them concurrently into a tree that is implied by the recursive calls. The algorithms make exactly the same comparisons, but in a different order.

The most direct competitor of quicksort is heapsort. Heapsort is typically somewhat slower than quicksort, but the worst-case running time is always $O(n \log n)$. Quicksort is usually faster, though there remains the chance of worst case performance except in the introsort variant. If it's known in advance that heapsort is going to be necessary, using it directly will be faster than waiting for introsort to switch to it. Heapsort also has the important advantage of using only constant additional space (heapsort is in-place), whereas even the best variant of quicksort uses $\Theta(\log n)$ space. However, heapsort requires efficient random access to be practical.

Quicksort also competes with mergesort, another recursive sort algorithm but with the benefit of worst-case $O(n \log n)$ running time. Mergesort is a stable sort, unlike quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Although quicksort can be written to operate on linked lists, it will often suffer from poor pivot choices without random access. The main disadvantage of mergesort is that, when operating on arrays, it requires $\Omega(n)$ auxiliary space in the best case, whereas the variant of quicksort with in-place partitioning and tail recursion uses only $O(\log$
n) space. (Note that when operating on linked lists, mergesort only requires a small, constant amount of auxiliary storage.)

6.2.4. Merge sort

(From Wikipedia, the free encyclopedia)

In computer science, merge sort or mergesort is an \( O(n \log n) \) comparison-based sorting algorithm. It is stable, meaning that it preserves the input order of equal elements in the sorted output. It is an example of the divide and conquer algorithmic paradigm. It was invented by John von Neumann in 1945.

A merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort.

**Algorithm**

Conceptually, merge sort works as follows:

1. Divide the unsorted list into two sublists of about half the size
2. Divide each of the two sublists recursively until we have list sizes of length 1, in which case the list itself is returned
3. Merge the two sublists back into one sorted list.
Mergesort incorporates two main ideas to improve its runtime:

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted (see the `merge` function below for an example implementation).

Example: Using mergesort to sort a list of integers contained in an array:

Suppose we have an array `A` with indices ranging from `A’first` to `A’Last`. We apply mergesort to `A(A’first..A’centre)` and `A(centre+1..A’Last)` - where centre is the integer part of `(A’first + A’Last)/2`. When the two halves are returned they will have been sorted. They can now be merged together to form a sorted array.

In a simple pseudocode form, the algorithm could look something like this:

```plaintext
function mergesort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m
    else
        var middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = mergesort(left)
        right = mergesort(right)
        result = merge(left, right)
        return result
```
There are several variants for the merge() function, the simplest variant could look like this:

```plaintext
function merge(left, right)
var list result
while length(left) > 0 and length(right) > 0
  if first(left) ≤ first(right)
    append first(left) to result
    left = rest(left)
  else
    append first(right) to result
    right = rest(right)
if length(left) > 0
  append rest(left) to result
if length(right) > 0
  append rest(right) to result
return result
```

**C++ implementation**

Here is an implementation using the [STL](https://en.cppreference.com/w/cpp) algorithm `std::inplace_merge` to create an iterative bottom-up in-place merge sort:

```cpp
# include <iostream>
# include <vector>
# include <algorithm>
# include <iterator>
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
```
int main()
{
    std::vector<unsigned> data;
    for(unsigned i = 0; i < 10; i++)
        data.push_back(i);
    std::random_shuffle(data.begin(), data.end);
    std::cout << "Initial: ";
    std::copy(data.begin(), data.end(), std::ostream_iterator<unsigned>(std::cout, " "));
    std::cout << std::endl;
    for(unsigned m = 1; m <= data.size(); m *= 2)
    {
        for(unsigned i = 0; i < data.size() - m; i += m * 2)
        {
            std::inplace_merge(
                data.begin() + i,
                data.begin() + i + m,
                data.begin() + std::min<unsigned>(i + m * 2, (unsigned)data.size()));
        }
    }
    std::cout << "Sorted: ";
    std::copy(data.begin(), data.end(), std::ostream_iterator<unsigned>(std::cout, " "));
    std::cout << std::endl;
Analysis

In sorting n items, merge sort has an average and worst-case performance of \( O(n \log n) \). If the running time of merge sort for a list of length n is \( T(n) \), then the recurrence \( T(n) = 2T(n/2) + n \) follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem.

In the worst case, merge sort does approximately \( (n \lfloor \log n \rfloor - 2\lceil \log n \rceil + 1) \) comparisons, which is between \( (n \log n - n + 1) \) and \( (n \log n + n + O(\log n)) \). [2]

For large n and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches \( \alpha \cdot n \) fewer than the worst case where

\[
\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k + 1} \approx 0.2645
\]

In the worst case, merge sort does about 39% fewer comparisons than quicksort does in the average case; merge sort always makes fewer comparisons than quicksort, except in extremely rare cases, when they tie, where merge sort's worst case is found simultaneously with quicksort's best case. In terms of moves, merge sort's worst case complexity is \( O(n \log n) \)—the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.

Recursive implementations of merge sort make \( 2n - 1 \) method calls in the worst case, compared to quicksort's n, thus has roughly twice as much recursive overhead as quicksort. However, iterative, non-recursive, implementations of merge sort, avoiding method call overhead, are not difficult to code. Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in. Sorting in-place is possible but is very complicated, and will offer little performance gains in practice, even if the algorithm runs in \( O(n \log n) \) time. In these cases, algorithms like heapsort usually offer comparable speed, and are far less complex.

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort as long as the merge operation is implemented properly.
As can be seen from the procedure MergeSort, there are some complaints. One complaint we might raise is its use of $2n$ locations; the additional $n$ locations were needed because one couldn't reasonably merge two sorted sets in place. But despite the use of this space the algorithm must still work hard, copying the result placed into Result list back into m list on each call of merge. An alternative to this copying is to associate a new field of information with each key. (the elements in m are called keys). This field will be used to link the keys and any associated information together in a sorted list (keys and related informations are called records). Then the merging of the sorted lists proceeds by changing the link values and no records need to moved at all. A field which contains only a link will generally be smaller than an entire record so less space will also be used.

**Merge sorting tape drives**

Merge sort is so inherently sequential that it's practical to run it using slow tape drives as input and output devices. It requires very little memory, and the memory required does not change with the number of data elements. If you have four tape drives, it works as follows:

1. Divide the data to be sorted in half and put half on each of two tapes
2. Merge individual pairs of records from the two tapes; write two-record chunks alternately to each of the two output tapes
3. Merge the two-record chunks from the two output tapes into four-record chunks; write these alternately to the original two input tapes
4. Merge the four-record chunks into eight-record chunks; write these alternately to the original two output tapes
5. Repeat until you have one chunk containing all the data, sorted --- that is, for log $n$ passes, where $n$ is the number of records.

For the same reason it is also very useful for sorting data on disk that is too large to fit entirely into primary memory. On tape drives that can run both backwards and forwards, you can run merge passes in both directions, avoiding rewind time.

**Optimizing merge sort**

This might seem to be of historical interest only, but on modern computers, locality of reference is of paramount importance in software optimization, because multi-level memory hierarchies are used. In some sense, main RAM can be seen as a fast tape drive, level 3 cache memory as a slightly faster one, level 2 cache memory as faster still, and so on. In some circumstances, cache reloading might impose unacceptable overhead and a carefully crafted merge sort might result in a significant improvement in running time. This opportunity might change if fast memory becomes very cheap again, or if exotic architectures like the Tera MTA become commonplace.
Designing a merge sort to perform optimally often requires adjustment to available hardware, eg. number of tape drives, or size and speed of the relevant cache memory levels.

**Typical implementation bugs**

A typical mistake made in many merge sort implementations is the division of index-based lists in two sublists. Many implementations determine the middle index as outlined in the following implementation example:

```c
function merge(int left, int right)
{
    if (left < right) {
        int middle = (left + right) / 2;
    }...
```

While this algorithm appears to work very well in most scenarios, it fails for very large lists. The addition of "left" and "right" would lead to an integer overflow, resulting in a completely wrong division of the list. This problem can be solved by increasing the data type size used for the addition, or by altering the algorithm:

```c
int middle = left + (right - left) / 2;
```

Note that the following two examples do not address the issue of integer overflow but dodge it under irrelevant efficiency claims

Probably faster, and arguably as clear is:

```c
int middle = (left + right) >>> 1;
```

In C and C++ (where you don't have the >>> operator), you can do this:

```c
middle = ((unsigned) (left + right)) >>> 1;
```

See more information here: [http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html](http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html)
Comparison with other sort algorithms

Although heapsort has the same time bounds as merge sort, it requires only $\Theta(1)$ auxiliary space instead of merge sort's $\Theta(n)$, and is often faster in practical implementations. Quicksort, however, is considered by many to be the fastest general-purpose sort algorithm. On the plus side, merge sort is a stable sort, parallelizes better, and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\Theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

As of Perl 5.8, merge sort is its default sorting algorithm (it was quicksort in previous versions of Perl). In Java, the Arrays.sort() methods use mergesort or a tuned quicksort depending on the datatypes and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.

Utility in online sorting

Mergesort's merge operation is useful in online sorting, where the list to be sorted is received a piece at a time, instead of all at the beginning (see online algorithm). In this application, we sort each new piece that is received using any sorting algorithm, and then merge it into our sorted list so far using the merge operation. However, this approach can be expensive in time and space if the received pieces are small compared to the sorted list — a better approach in this case is to store the list in a self-balancing binary search tree and add elements to it as they are received.
Graphs

Graph theory

(From Wikipedia, the free encyclopedia)

In mathematics and computer science, graph theory is the study of graphs; mathematical structures used to model pairwise relations between objects from a certain collection. A "graph" in this context refers to a collection of vertices or 'nodes' and a collection of edges that connect pairs of vertices. A graph may be undirected, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be directed from one vertex to another; see graph (mathematics) for more detailed definitions and for other variations in the types of graphs that are commonly considered. The graphs studied in graph theory should not be confused with "graphs of functions" and other kinds of graphs.

History

The paper written by Leonhard Euler on the Seven Bridges of Königsberg and published in 1736 is regarded as the first paper in the history of graph theory. This paper, as well as the one written by Vandermonde on the knight problem carried on with the analysis situs initiated by Leibniz. Euler's formula relating the number of edges, vertices, and faces of a convex polyhedron was studied and generalized by Cauchy and L'Huillier, and is at the origin of topology.

More than one century after Euler's paper on the bridges of Königsberg and while Listing introduced topology, Cayley was led by the study of particular analytical forms arising from differential calculus to study a particular class of graphs, the trees. This study had many implications in theoretical chemistry. The involved techniques mainly concerned the enumeration of graphs having particular properties. Enumerative graph theory then rose from the results of Cayley and the fundamental results published by Pólya between 1935 and 1937 and the generalization of these by De Bruijn in 1959. Cayley linked his results on trees with the contemporary studies of chemical composition. The fusion of the ideas coming from mathematics with those coming from
chemistry is at the origin of a part of the standard terminology of graph theory. In particular, the term graph was introduced by Sylvester in a paper published in 1878 in Nature.

One of the most famous and productive problems of graph theory is the four color problem: "Is it true that any map drawn in the plane may have its regions colored with four colors, in such a way that any two regions having a common border have different colors?". This problem remained unsolved for more than a century and the proof given by Kenneth Appel and Wolfgang Haken in 1976 (determination of 1936 types of configurations of which study is sufficient and checking of the properties of these configurations by computer) did not convince all the community. A simpler proof considering far fewer configurations was given twenty years later by Robertson, Seymour, Sanders and Thomas.

This problem was first posed by Francis Guthrie in 1852 and the first written record of this problem is a letter of De Morgan addressed to Hamilton the same year. Many incorrect proofs have been proposed, including those by Cayley, Kempe, and others. The study and the generalization of this problem by Tait, Heawood, Ramsey and Hadwiger has in particular led to the study of the colorings of the graphs embedded on surfaces with arbitrary genus. Tait's reformulation generated a new class of problems, the factorization problems, particularly studied by Petersen and König. The works of Ramsey on colorations and more specially the results obtained by Turán in 1941 is at the origin of another branch of graph theory, the extremal graph theory.

The autonomous development of topology from 1860 and 1930 fertilized graph theory back through the works of Jordan, Kuratowski and Whitney. Another important factor of common development of graph theory and topology came from the use of the techniques of modern algebra. The first example of such a use comes from the work of the physicist Gustav Kirchhoff, who published in 1845 his Kirchhoff's circuit laws for calculating the voltage and current in electric circuits.

The introduction of probabilistic methods in graph theory, specially in the study of Erdős and Rényi of the asymptotic probability of graph connexity is at the origin of yet another branch, known as random graph theory. Research in this branch has enabled mathematicians across the globe to advance the theory of graphs significantly.

Drawing graphs

Graphs are represented graphically by drawing a dot for every vertex, and drawing an arc between two vertices if they are connected by an edge. If the graph is directed, the direction is indicated by drawing an arrow.
A graph drawing should not be confused with the graph itself (the abstract, non-graphical structure) as there are several ways to structure the graph drawing. All that matters is which vertices are connected to which others by how many edges and not the exact layout. In practice it is often difficult to decide if two drawings represent the same graph. Depending on the problem domain some layouts may be better suited and easier to understand than others.

Graph-theoretic data structures

There are different ways to store graphs in a computer system. The data structure used depends on both the graph structure and the algorithm used for manipulating the graph. Theoretically one can distinguish between list and matrix structures but in concrete applications the best structure is often a combination of both. List structures are often preferred for sparse graphs as they have smaller memory requirements. Matrix structures on the other hand provide faster access for some applications but can consume huge amounts of memory.

List structures

- Incidence list - The edges are represented by an array containing pairs (ordered if directed) of vertices (that the edge connects) and possibly weight and other data.
- Adjacency list - Much like the incidence list, each vertex has a list of which vertices it is adjacent to. This causes redundancy in an undirected graph: for example, if vertices A and B are adjacent, A's adjacency list contains B, while B's list contains A. Adjacency queries are faster, at the cost of extra storage space.

Matrix structures

- Incidence matrix - The graph is represented by a matrix of E (edges) by V (vertices), where [edge, vertex] contains the edge's data (simplest case: 1 - connected, 0 - not connected).
- Adjacency matrix - there is an N by N matrix, where N is the number of vertices in the graph. If there is an edge from some vertex x to some vertex y, then the element Mx,y is 1, otherwise it is 0. This makes it easier to find subgraphs, and to reverse graphs if needed.
- Laplacian matrix or Kirchhoff matrix or Admittance matrix - is defined as degree matrix minus adjacency matrix and thus contains adjacency information and degree information about the vertices
- Distance matrix - A symmetric N by N matrix an element Mx,y of which is the length of shortest path between x and y; if there is no such path Mx,y = infinity. It can be derived from powers of the Adjacency matrix.
Problems in graph theory

Enumeration

There is a large literature on graphical enumeration: the problem of counting graphs meeting specified conditions. Some of this work is found in Harary and Palmer (1973).

Subgraphs, induced subgraphs, and minors

A common problem, called the subgraph isomorphism problem, is finding a fixed graph as a subgraph in a given graph. One reason to be interested in such a question is that many graph properties are hereditary for subgraphs, which means that a graph has the property if and only if all subgraphs, or all induced subgraphs, have it too. Unfortunately, finding maximal subgraphs of a certain kind is often an NP-complete problem.

- Finding the largest complete graph is called the clique problem (NP-complete).

A similar problem is finding induced subgraphs in a given graph. Again, some important graph properties are hereditary with respect to induced subgraphs, which means that a graph has a property if and only if all induced subgraphs also have it. Finding maximal induced subgraphs of a certain kind is also often NP-complete. For example,

- Finding the largest edgeless induced subgraph, or independent set, called the independent set problem (NP-complete).

Still another such problem, the minor containment problem, is to find a fixed graph as a minor of a given graph. A minor or subcontraction of a graph is any graph obtained by taking a subgraph and contracting some (or no) edges. Many graph properties are hereditary for minors, which means that a graph has a property if and only if all minors have it too. A famous example:

- A graph is planar if it contains as a minor neither the complete bipartite graph $K_{3,3}$ (See the Three cottage problem) nor the complete graph $K_5$.

Another class of problems has to do with the extent to which various species and generalizations of graphs are determined by their point-deleted subgraphs, for example:

- The reconstruction conjecture

Graph coloring

Many problems have to do with various ways of coloring graphs, for example:
• The four-color theorem
• The strong perfect graph theorem
• The Erdős-Faber-Lovász conjecture (unsolved)
• The total coloring conjecture (unsolved)
• The list coloring conjecture (unsolved)

Route problems

• Hamiltonian path and cycle problems
• Minimum spanning tree
• Route inspection problem (also called the "Chinese Postman Problem")
• Seven Bridges of Königsberg
• Shortest path problem
• Steiner tree
• Three cottage problem
• Traveling salesman problem (NP-Complete)

Network flow

There are numerous problems arising especially from applications that have to do with various notions of flows in networks, for example:

Visibility graph problems

• Museum guard problem

Covering problems

Covering problems are specific instances of subgraph-finding problems, and they tend to be closely related to the clique problem or the independent set problem.

• Set cover problem
• Vertex cover problem

Applications

Applications of graph theory are primarily, but not exclusively, concerned with labeled graphs and various specializations of these.

Structures that can be represented as graphs are ubiquitous, and many problems of practical interest can be represented by graphs. The link structure of a website could be represented by a directed graph: the vertices are the web pages available at the website and a directed edge from page A to page B exists if and only if A contains a link to B. A similar approach can be taken to problems in travel, biology, computer chip design,
and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science.

A graph structure can be extended by assigning a weight to each edge of the graph. Graphs with weights, or weighted graphs, are used to represent structures in which pairwise connections have some numerical values. For example if a graph represents a road network, the weights could represent the length of each road. A digraph with weighted edges in the context of graph theory is called a network.

Networks have many uses in the practical side of graph theory, network analysis (for example, to model and analyze traffic networks). Within network analysis, the definition of the term "network" varies, and may often refer to a simple graph.

Many applications of graph theory exist in the form of network analysis. These split broadly into two categories. Firstly, analysis to determine structural properties of a network, such as the distribution of vertex degrees and the diameter of the graph. A vast number of graph measures exist, and the production of useful ones for various domains remains an active area of research. Secondly, analysis to find a measurable quantity within the network, for example, for a transportation network, the level of vehicular flow within any portion of it.

Graph theory is also used to study molecules in chemistry and physics. In condensed matter physics, the three dimensional structure of complicated simulated atomic structures can be studied quantitatively by gathering statistics on graph-theoretic properties related to the topology of the atoms. For example, Franzblau's shortest-path (SP) rings.

Graph theory is also widely used in sociology as a way, for example, to measure actors' prestige or to explore diffusion mechanisms, notably through the use of social network analysis software.

**Minimum spanning trees**

**Boruvská’s algorithms**

(From Wikipedia, the free encyclopedia)

Borůvka's algorithm is an algorithm for finding a minimum spanning tree in a graph for which all edge weights are distinct.

It was first published in 1926 by Otakar Borůvka as a method of constructing an efficient electricity network for Moravia. The algorithm was rediscovered by Choquet in 1938; again by Florek, Łukasiewicz, Perkal, Steinhaus, and Zubrzycki in 1951; and again by
Sollin some time in the early 1960s. Because Sollin was the only Western computer scientist in this list, this algorithm is frequently called Sollin's algorithm, especially in the parallel computing literature.

The algorithm begins by examining each vertex and adding the cheapest edge from that vertex to another in the graph, without regard to already added edges, and continues joining these groupings in a like manner until a tree spanning all vertices is completed. Designating each vertex or set of connected vertices a "component", pseudocode for Borůvka's algorithm is:

- Begin with a connected graph G containing edges of distinct weights, and an empty set of edges T
- While the vertices of G connected by T are disjoint:
  - Begin with an empty set of edges E
  - For each component:
    - Begin with an empty set of edges S
    - For each vertex in the component:
      - Add the cheapest edge from the vertex in the component to another vertex in a disjoint component to S
      - Add the cheapest edge in S to E
    - Add the resulting set of edges E to T.
  - The resulting set of edges T is the minimum spanning tree of G

Borůvka's algorithm can be shown to run in time O(E log V), where E is the number of edges, and V is the number of vertices in G.

Other algorithms for this problem include Prim's algorithm (actually discovered by Vojtěch Jarník) and Kruskal's algorithm. Faster algorithms can be obtained by combining Prim's algorithm with Borůvka's. A faster randomized version of Borůvka's algorithm due to Karger, Klein, and Tarjan runs in expected O(E) time. The best known (deterministic) minimum spanning tree algorithm by Bernard Chazelle is based on Borůvka's and runs in O(E \(\alpha(V)\)) time, where \(\alpha\) is the inverse of the Ackermann function.

**Kruskal’s algorithms**

(From Wikipedia, the free encyclopedia)

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.
Kruskal's algorithm

It works as follows:

• create a forest F (a set of trees), where each vertex in the graph is a separate tree
• create a set S containing all the edges in the graph
• while S is nonempty
  ◦ remove an edge with minimum weight from S
  ◦ if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
  ◦ otherwise discard that edge

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

This algorithm first appeared in Proceedings of the American Mathematical Society, pp. 48–50 in 1956, and was written by Joseph Kruskal.

Performance

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because:

• E is at most $V^2$ and $\log V^2 = 2\log V$ is $O(\log V)$.
• If we ignore isolated vertices, which will each be their own component of the minimum spanning tree anyway, $V \leq 2E$, so $\log V$ is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time; this allows the step "remove an edge with minimum weight from S" to operate in constant time. Next, we use a disjoint-set data structure to keep track of which vertices are in which components. We need to perform $O(E)$ operations, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data
structure such as disjoint-set forests with union by rank can perform $O(E)$ operations in $O(E \log V)$ time. Thus the total time is $O(E \log E) = O(E \log V)$.

Provided that the edges are either already sorted or can be sorted in linear time (for example with counting sort or radix sort), the algorithm can use more sophisticated disjoint-set data structures to run in $O(E \alpha(V))$ time, where $\alpha$ is the extremely slowly-growing inverse of the single-valued Ackermann function.

Example

This is our original graph. The numbers near the arcs indicate their weight. None of the arcs are highlighted.

AD and CE are the shortest arcs, with length 5, and AD has been arbitrarily chosen, so it is highlighted.

However, CE is now the shortest arc that does not form a loop, with length 5, so it is highlighted as the second arc.
The next arc, DF with length 6, is highlighted using much the same method.

The next-shortest arcs are AB and BE, both with length 7. AB is chosen arbitrarily, and is highlighted. The arc BD has been highlighted in red, because it would form a loop ABD if it were chosen.

The process continues to highlight the next-smallest arc, BE with length 7. Many more arcs are highlighted in red at this stage: BC because it would form the loop BCE, DE because it would form the loop DEBA, and FE because it would form FEBAD.

Finally, the process finishes with the arc EG of length 9, and the minimum spanning tree is found.

Proof of correctness

Let P be a connected, weighted graph and let Y be the subgraph of P produced by the algorithm. Y cannot have a cycle, since the last edge added to that cycle would have been within one subtree and not between two different trees. Y cannot be disconnected, since the first encountered edge that joins two components of Y would have been added by the algorithm. Thus, Y is a spanning tree of P.
It remains to show that the spanning tree $Y$ is minimal:

Let $Y_1$ be a minimum spanning tree. If $Y = Y_1$ then $Y$ is a minimum spanning tree. Otherwise, let $e$ be the first edge considered by the algorithm that is in $Y$ but not in $Y_1$. $Y_1 \cup e$ has a cycle, because you cannot add an edge to a spanning tree and still have a tree. This cycle contains another edge $f$ which at the stage of the algorithm where $e$ is added to $Y$, has not been considered. This is because otherwise $e$ would not connect different trees but two branches of the same tree. Then $Y_2 = Y_1 \cup e \setminus f$ is also a spanning tree. Its total weight is less than or equal to the total weight of $Y_1$. This is because the algorithm visits $e$ before $f$ and therefore $w(e) \leq w(f)$. If the weights are equal, we consider the next edge $e$ which is in $Y$ but not in $Y_1$. If there is no edge left, the weight of $Y$ is equal to the weight of $Y_1$ although they consist of a different edge set and $Y$ is also a minimum spanning tree. In the case where the weight of $Y_2$ is less than the weight of $Y_1$ we can conclude that $Y_1$ is not a minimum spanning tree, and the assumption that there exist edges $e, f$ with $w(e) < w(f)$ is incorrect. And therefore $Y$ is a minimum spanning tree (equal to $Y_1$ or with a different edge set, but with same weight).

Pseudocode

1 function Kruskal(G)
2 for each vertex v in G do
3 Define an elementary cluster $C(v) \leftarrow \{v\}$.
4 Initialize a priority queue $Q$ to contain all edges in $G$, using the weights as keys.
5 Define a tree $T \leftarrow \emptyset$ // $T$ will ultimately contain the edges of the MST
6 // n is total number of vertices
7 while $T$ has fewer than $n-1$ edges do
8 // edge $u,v$ is the minimum weighted route from/to v
9 ($u,v) \leftarrow Q$.removeMin()
10 // prevent cycles in $T$. add $u,v$ only if $T$ does not already contain an edge consisting of $u$ and $v$.
11 // Note that the cluster contains more than one vertex only if an edge containing a pair of
12 // the vertices has been added to the tree.
13 Let $C(v)$ be the cluster containing $v$, and let $C(u)$ be the cluster containing $u$.

14 if $C(v) \neq C(u)$ then

15 Add edge $(v,u)$ to $T$.

16 Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$.

17 return tree $T$

**Jarnik-Prim’s algorithms**

(From Wikipedia, the free encyclopedia)

Prim's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm was discovered in 1930 by mathematician Vojtěch Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Dijkstra in 1959. Therefore it is sometimes called the DJP algorithm or Jarnik algorithm.

**Description**

The algorithm continuously increases the size of a tree starting with a single vertex until it spans all the vertices.

- **Input:** A connected weighted graph $G(V,E)$
- **Initialize:** $V' = \{x\}$, where $x$ is an arbitrary node from $V$, $E'= \{\}$
- **repeat until $V'=V$:**
  - Choose edge $(u,v)$ from $E$ with minimal weight such that $u$ is in $V'$ and $v$ is not in $V'$ (if there are multiple edges with the same weight, choose arbitrarily)
  - Add $v$ to $V'$, add $(u,v)$ to $E'$
- **Output:** $G(V',E')$ is the minimal spanning tree

**Time complexity**

<table>
<thead>
<tr>
<th>Minimum edge weight data structure</th>
<th>Time complexity (total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjacency matrix, searching</td>
<td>$V^2$</td>
</tr>
<tr>
<td>binary heap (as in pseudocode below) and adjacency list</td>
<td>$O((V + E) \log(V)) = E \log(V)$</td>
</tr>
<tr>
<td>Fibonacci heap and adjacency list</td>
<td>$E + V \log(V)$</td>
</tr>
</tbody>
</table>
A simple implementation using an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge to add requires $O(V^2)$ running time. Using a simple binary heap data structure and an adjacency list representation, Prim's algorithm can be shown to run in time which is $O(E \log V)$ where $E$ is the number of edges and $V$ is the number of vertices. Using a more sophisticated Fibonacci heap, this can be brought down to $O(E + V \log V)$, which is significantly faster when the graph is dense enough that $E$ is $\Omega(V \log V)$.

Example

<table>
<thead>
<tr>
<th>Image</th>
<th>Description</th>
<th>Not seen</th>
<th>Fringe</th>
<th>Solution set</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td>This is our original weighted graph. This is not a tree because the definition of a tree requires that there are no circuits and this diagram contains circuits. A more correct name for this diagram would be a graph or a network. The numbers near the arcs indicate their weight. None of the arcs are highlighted, and vertex D has been arbitrarily chosen as a starting point.</td>
<td>C, G</td>
<td>A, B, E, F</td>
<td>D</td>
</tr>
<tr>
<td><img src="image2.png" alt="Image" /></td>
<td>The second chosen vertex is the vertex nearest to D: A is 5 away, B is 9, E is 15, and F is 6. Of these, 5 is the smallest, so we highlight the vertex A and the arc DA.</td>
<td>C, G</td>
<td>B, E, F</td>
<td>A, D</td>
</tr>
</tbody>
</table>
The next vertex chosen is the vertex nearest to either D or A. B is 9 away from D and 7 away from A, E is 15, and F is 6. 6 is the smallest, so we highlight the vertex F and the arc DF.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>9</td>
<td>G</td>
</tr>
</tbody>
</table>

The algorithm carries on as above. Vertex B, which is 7 away from A, is highlighted. Here, the arc DB is highlighted in red, because both vertex B and vertex D have been highlighted, so it cannot be used.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>9</td>
<td>G</td>
</tr>
</tbody>
</table>

In this case, we can choose between C, E, and G. C is 8 away from B, E is 7 away from B, and G is 11 away from F. E is nearest, so we highlight the vertex E and the arc EB. Two other arcs have been highlighted in red, as both their joining vertices have been used.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>9</td>
<td>G</td>
</tr>
</tbody>
</table>

Here, the only vertices available are C and G. C is 5 away from E, and G is 9 away from E. C is chosen, so it is highlighted along with the arc EC. The arc BC is also highlighted in red.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>9</td>
<td>G</td>
</tr>
</tbody>
</table>
Vertex G is the only remaining vertex. It is 11 away from F, and 9 away from E. E is nearer, so we highlight it and the arc EG. Now all the vertices have been highlighted, the minimum spanning tree is shown in green. In this case, it has weight 39.

<table>
<thead>
<tr>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>15</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Pseudo-code

**Min-heap**

**Initialization**

inputs: A graph, a function returning edge weights \( \text{weight-function} \), and an initial vertex

initial placement of all vertices in the 'not yet seen' set, set initial vertex to be added to the tree, and place all vertices in a min-heap to allow for removal of the min distance from the minimum graph.

for each vertex in graph

set \( \text{min\_distance of vertex} \) to \( \infty \)

set parent of vertex to null

set \( \text{minimum\_adjacency\_list of vertex} \) to empty list

set \( \text{is\_in\_Q of vertex} \) to true

set distance of initial vertex to zero

add to minimum-heap \( Q \) all vertices in graph.

**Algorithm**

In the algorithm description above,

nearest vertex is \( Q[0] \), now latest addition
fringe is v in Q where distance of v < ∞ after nearest vertex is removed
not seen is v in Q where distance of v = ∞ after nearest vertex is removed
The while loop will fail when remove minimum returns null. The adjacency list is set to allow a directional graph to be returned.
time complexity: V for loop, log(V) for the remove function
while latest_addition = remove minimum in Q
set is_in_Q of latest_addition to false
add latest_addition to (minimum_adjacency_list of (parent of latest_addition))
add (parent of latest_addition) to (minimum_adjacency_list of latest_addition)
time complexity: E/V, the average number of vertices
for each adjacent of latest_addition
if (is_in_Q of adjacent) and (weight-function(latest_addition, adjacent) < min_distance of adjacent)
set parent of adjacent to latest_addition
set min_distance of adjacent to weight-function(latest_addition, adjacent)
time complexity: log(V), the height of the heap
update adjacent in Q, order by min_distance
Proof of correctness
Let P be a connected, weighted graph. At every iteration of Prim's algorithm, an edge must be found that connects a vertex in a subgraph to a vertex outside the subgraph. Since P is connected, there will always be a path to every vertex. The output Y of Prim's algorithm is a tree, because the edge and vertex added to Y are connected. Let Y1 be a minimum spanning tree of P. If Y1=Y then Y is a minimum spanning tree. Otherwise, let e be the first edge added during the construction of Y that is not in Y1, and V be the set of vertices connected by the edges added before e. Then one endpoint of e is in V and the other is not. Since Y1 is a spanning tree of P, there is a path in Y1 joining the two endpoints. As one travels along the path, one must encounter an edge f joining a vertex in V to one that is not in V. Now, at the iteration when e was added to Y, f could also
have been added and it would be added instead of e if its weight was less than e. Since f was not added, we conclude that

\[ w(f) \geq w(e). \]

Let \( Y_2 \) be the graph obtained by removing f and adding e from \( Y_1 \). It is easy to show that \( Y_2 \) is connected, has the same number of edges as \( Y_1 \), and the total weights of its edges is not larger than that of \( Y_1 \), therefore it is also a minimum spanning tree of P and it contains e and all the edges added before it during the construction of V. Repeat the steps above and we will eventually obtain a minimum spanning tree of P that is identical to Y. This shows Y is a minimum spanning tree.

Shortest paths

Properties of shortest paths

(From Wikipedia, the free encyclopedia)

In graph theory, the shortest path problem is the problem of finding a path between two vertices such that the sum of the weights of its constituent edges is minimized. An example is finding the quickest way to get from one location to another on a road map; in this case, the vertices represent locations and the edges represent segments of road and are weighted by the time needed to travel that segment.

Formally, given a weighted graph (that is, a set \( V \) of vertices, a set \( E \) of edges, and a real-valued weight function \( f : E \to \mathbb{R} \)), and one element \( v \) of \( V \), find a path \( P \) from \( v \) to each \( v' \) of \( V \) so that

\[
\sum_{p \in P} f(p)
\]

is minimal among all paths connecting \( v \) to \( v' \).

Sometimes it is called the single-pair shortest path problem, to distinguish it from the following generalizations:

- The single-source shortest path problem is a more general problem, in which we have to find shortest paths from a source vertex \( v \) to all other vertices in the graph.
- The all-pairs shortest path problem is an even more general problem, in which we have to find shortest paths between every pair of vertices \( v, v' \) in the graph.

Both these generalizations have significantly more performant algorithms in practice than simply running a single-pair shortest path algorithm on all relevant pairs of vertices.
Algorithms

The most important algorithms for solving this problem are:

- Dijkstra's algorithm — solves single source problem if all edge weights are greater than or equal to zero. Without worsening the run time, this algorithm can in fact compute the shortest paths from a given start point s to all other nodes.
- Bellman-Ford algorithm — solves single source problem if edge weights may be negative.
- A* search algorithm solves for single source shortest paths using heuristics to try to speed up the search
- Floyd-Warshall algorithm — solves all pairs shortest paths.
- Johnson's algorithm — solves all pairs shortest paths, may be faster than Floyd-Warshall on sparse graphs.
- Perturbation theory; finds (at worst) the locally shortest path

Applications

Shortest path algorithms are applied in an obvious way to automatically find directions between physical locations, such as driving directions on web mapping websites like Mapquest.

If one represents a nondeterministic abstract machine as a graph where vertices describe states and edges describe possible transitions, shortest path algorithms can be used to find an optimal sequence of choices to reach a certain goal state, or to establish lower bounds on the time needed to reach a given state. For example, if vertices represents the states of a puzzle like a Rubik's Cube and each directed edge corresponds to a single move or turn, shortest path algorithms can be used to find a solution that uses the minimum possible number of moves.

In a networking or telecommunications mindset, this shortest path problem is sometimes called the min-delay path problem and usually tied with a widest path problem. e.g.: Shortest (min-delay) widest path or Widest shortest (min-delay) path.

Dijkstra’s algorithms

(From Wikipedia, the free encyclopedia)

Dijkstra's algorithm, named after its discoverer, Dutch computer scientist Edsger Dijkstra, is a greedy algorithm that solves the single-source shortest path problem for a directed graph with non negative edge weights.
For example, if the vertices (nodes) of the graph represent cities and edge weights represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between two cities.

The input of the algorithm consists of a weighted directed graph $G$ and a source vertex $s$ in $G$. We will denote $V$ the set of all vertices in the graph $G$. Each edge of the graph is an ordered pair of vertices $(u,v)$ representing a connection from vertex $u$ to vertex $v$. The set of all edges is denoted $E$. Weights of edges are given by a weight function $w: E \rightarrow [0, \infty)$; therefore $w(u,v)$ is the cost of moving directly from vertex $u$ to vertex $v$. The cost of an edge can be thought of as (a generalization of) the distance between those two vertices. The cost of a path between two vertices is the sum of costs of the edges in that path. For a given pair of vertices $s$ and $t$ in $V$, the algorithm finds the path from $s$ to $t$ with lowest cost (i.e. the shortest path). It can also be used for finding costs of shortest paths from a single vertex $s$ to all other vertices in the graph.

**Pseudo-code**

In the following algorithm, $u := \text{extract\_min}(Q)$ searches for the vertex $u$ in the vertex set $Q$ that has the least $\text{dist}[u]$ value. That vertex is removed from the set $Q$ and returned to the user. $\text{length}(u, v)$ calculates the length between the two neighbor-nodes $u$ and $v$. $\text{alt}$ on line 10 is the length of the path from the root node to the neighbor node $v$ if it were to go through $u$. If this path is shorter than the current shortest path recorded for $v$, that current path is replaced with this alt path.

1 function Dijkstra(Graph, source):
2 for each vertex $v$ in Graph: // Initializations
3 $\text{dist}[v] := \infty$ // Unknown distance function from $s$ to $v$
4 $\text{previous}[v] := \text{undefined}$
5 $\text{dist}[\text{source}] := 0$ // Distance from $s$ to $s$
6 $Q := \text{copy}(\text{Graph})$ // Set of all unvisited vertices
7 while $Q$ is not empty: // The main loop
8 $u := \text{extract\_min}(Q)$ // Remove best vertex from priority queue; returns source on first iteration
9 for each neighbor $v$ of $u$:
10 $\text{alt} = \text{dist}[u] + \text{length}(u, v)$
11 if alt < dist[v] // Relax (u,v)
12 dist[v] := alt
13 previous[v] := u

If we are only interested in a shortest path between vertices source and target, we can terminate the search at line 9 if u = target. Now we can read the shortest path from source to target by iteration:

1 S := empty sequence
2 u := target
3 while defined previous[u]
4 insert u at the beginning of S
5 u := previous[u]

Now sequence S is the list of vertices constituting one of the shortest paths from source to target, or the empty sequence if no path exists.

A more general problem would be to find all the shortest paths between source and target (there might be several different ones of the same length). Then instead of storing only a single node in each entry of previous[] we would store all nodes satisfying the relaxation condition. For example, if both r and source connect to target and both of them lie on different shortest paths through target (because the edge cost is the same in both cases), then we would add both r and source to previous[target]. When the algorithm completes, previous[] data structure will actually describe a graph that is a subset of the original graph with some edges removed. Its key property will be that if the algorithm was run with some starting node, then every path from that node to any other node in the new graph will be the shortest path between those nodes in the original graph, and all paths of that length from the original graph will be present in the new graph. Then to actually find all these short paths between two given nodes we would use path finding algorithm on the new graph, such as depth-first search.

**Running time**

The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big-O notation.
The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract-Min(Q) is simply a linear search through all vertices in Q. In this case, the running time is \( O(|V|^2 + |E|) \).

For sparse graphs, that is, graphs with many fewer than \( |V|^2 \) edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a binary heap, pairing heap, or Fibonacci heap as a priority queue to implement the Extract-Min function efficiently. With a binary heap, the algorithm requires \( O((|E| + |V|) \log |V|) \) time (which is dominated by \( O(|E| \log |V|) \) assuming every vertex is connected, i.e., \( |E| \geq |V| - 1 \)), and the Fibonacci heap improves this to \( O(|E| + |V| \log |V|) \).

**Related problems and algorithms**

The functionality of Dijkstra's original algorithm can be extended with a variety of modifications. For example, sometimes it is desirable to present solutions which are less than mathematically optimal. To obtain a ranked list of less-than-optimal solutions, the optimal solution is first calculated. A single edge appearing in the optimal solution is removed from the graph, and the optimum solution to this new graph is calculated. Each edge of the original solution is suppressed in turn and a new shortest-path calculated. The secondary solutions are then ranked and presented after the first optimal solution.

OSPF (open shortest path first) is a well known real-world implementation of Dijkstra's algorithm used in Internet routing.

Unlike Dijkstra's algorithm, the Bellman-Ford algorithm can be used on graphs with negative edge weights, as long as the graph contains no negative cycle reachable from the source vertex \( s \). (The presence of such cycles means there is no shortest path, since the total weight becomes lower each time the cycle is traversed.)

The A* algorithm is a generalization of Dijkstra's algorithm that cuts down on the size of the subgraph that must be explored, if additional information is available that provides a lower-bound on the "distance" to the target.

**Breadth-first search**

(From Wikipedia, the free encyclopedia)

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.
BFS is a uninformed search method that aims to expand and examine all nodes of a graph systematically in search of a solution. In other words, it exhaustively searches the entire graph without considering the goal until it finds it. It does not use a heuristic.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

![Pic.15 Animated example of a breadth-first search](image)

**Algorithm (informal)**

1. Put the ending node (the root node) in the queue.
2. Pull a node from the beginning of the queue and examine it.
   - If the searched element is found in this node, quit the search and return a result.
   - Otherwise push all the (so-far-unexamined) successors (the direct child nodes) of this node into the end of the queue, if there are any.
3. If the queue is empty, every node on the graph has been examined -- quit the search and return "not found".
4. Repeat from Step 2.

**C implementation**

Algorithm of Breadth-first search?

```c
void BFS(VLink G[], int v) {
  int w;
  VISIT(v); /*visit vertex v*/
  visited[v] = 1; /*mark v as visited : 1 */
```
ADDQ(Q,v);

while(!QMPQTYQ(Q)) {
    v = DELQ(Q); /*Dequeue v*/
    w = FIRSTADJ(G,v); /*Find first neighbor, return -1 if no neighbor*/
    while(w != -1) {
        if(visited[w] == 0) {
            VISIT(w); /*visit vertex v*/
            ADDQ(Q,w); /*Enqueue current visited vertex w*/
            visited[w] = 1; /*mark w as visited*/
        }
        W = NEXTADJ(G,v); /*Find next neighbor, return -1 if no neighbor*/
    }
}

Main Algorithm of apply Breadth-first search to graph G=(V,E)?

void TRAVEL_BFS(VLink G[], int visited[], int n) { 
    int i;
    for(i = 0; i < n; i ++) {
        visited[i] = 0; /* Mark initial value as 0 */
    }
    for(i = 0; i < n; i++)
        if(visited[i] == 0)
BFS(G,i);
}

**C++ implementation**

This is the implementation of the above informal algorithm, where the "so-far-unexamined" is handled by the parent array. For actual C++ applications, see the Boost Graph Library.

Suppose we have a struct:

```cpp
struct Vertex {
    ...
    std::vector<int> out;
    ...
};
```

and an array of vertices: (the algorithm will use the indexes of this array, to handle the vertices)

```cpp
std::vector<Vertex> graph(vertices);
```

the algorithm starts from start and returns true if there is a directed path from start to end:

```cpp
bool BFS(const std::vector<Vertex>& graph, int start, int end) {
    std::queue<int> next;
    std::map<int,int> parent;
    parent[start] = -1;
    next.push(start);
    while (!next.empty()) {
        int u = next.front();
        int u = next.front();
```
next.pop();

// Here is the point where you can examine the u th vertex of graph

// For example:

if (u == end) return true;

for (std::vector<int>::const_iterator j = graph[u].out.begin(); j != graph[u].out.end(); ++j) {

// Look through neighbors.

    int v = *j;

    if (parent.count(v) == 0) {

        // If v is unvisited.

        parent[v] = u;

        next.push(v);

    }

}

return false;

} 

it also stores the parents of each node, from which you can get the path.

**Features**

- Space Complexity

Since all nodes discovered so far have to be saved, the space complexity of breadth-first search is $O(|V| + |E|)$ where $|V|$ is the number of nodes and $|E|$ the number of edges in the graph. Note: another way of saying this is that it is $O(BM)$ where $B$ is the maximum branching factor and $M$ is the maximum path length of the tree. This immense demand for space is the reason why breadth-first search is impractical for larger problems.
• **Time Complexity**

Since in the worst case breadth-first search has to consider all paths to all possible nodes the time complexity of breadth-first search is \(O(|V| + |E|)\) where \(|V|\) is the number of nodes and \(|E|\) the number of edges in the graph. The best case of this search is \(o(1)\). It occurs when the node is found at first time.

• **Completeness**

Breadth-first search is complete. This means that if there is a solution breadth-first search will find it regardless of the kind of graph. However, if the graph is infinite and there is no solution breadth-first search will diverge.

• **Optimality**

For unit-step cost, breadth-first search is optimal. In general breadth-first search is not optimal since it always returns the result with the fewest edges between the start node and the goal node. If the graph is a weighted graph, and therefore has costs associated with each step, a goal next to the start does not have to be the cheapest goal available. This problem is solved by improving breadth-first search to uniform-cost search which considers the path costs. Nevertheless, if the graph is not weighted, and therefore all step costs are equal, breadth-first search will find the nearest and the best solution.

**Applications of BFS**

Breadth-first search can be used to solve many problems in graph theory, for example:

• Finding all connected components in a graph.
• Finding all nodes within one connected component
• Copying Collection, Cheney's algorithm
• Finding the shortest path between two nodes \(u\) and \(v\) (in an unweighted graph)
• Testing a graph for bipartiteness
• (Reverse) Cuthill–McKee mesh numbering

**Finding connected Components**

The set of nodes reached by a BFS are the largest connected component containing the start node.

**Testing bipartiteness**

BFS can be used to test bipartiteness, by starting the search at any vertex and giving alternating labels to the vertices visited during the search. That is, give label 0 to the
starting vertex, 1 to all its neighbours, 0 to those neighbours' neighbours, and so on. If at any step a vertex has (visited) neighbours with the same label as itself, then the graph is not bipartite. If the search ends without such a situation occurring, then the graph is bipartite.

**Usage in 2D grids for computer games**

BFS has been applied to pathfinding problems in computer games, such as Real-Time Strategy games, where the graph is represented by a tilemap, and each tile in the map represents a node. Each of that node is then connected to each of its neighbour (neighbour in north, north-east, east, south-east, south, south-west, west, and north-west).

It is worth mentioning that when BFS is used in that manner, the neighbour list should be created such that north, east, south and west get priority over north-east, south-east, south-west and north-west. The reason for this is that BFS tends to start searching in a diagonal manner rather than adjacent, and the path found will not be the correct one. BFS should first search adjacent nodes, then diagonal nodes.

**7.3.4. Bellman-Ford algorithms**

(From Wikipedia, the free encyclopedia)

The Bellman–Ford algorithm computes single-source shortest paths in a weighted digraph (where some of the edge weights may be negative). Dijkstra's algorithm accomplishes the same problem with a lower running time, but requires edge weights to be non-negative. Thus, Bellman–Ford is usually used only when there are negative edge weights.

If a graph contains a cycle of total negative weight then arbitrarily low weights are achievable and so there's no solution; Bellman-Ford detects this case.

Bellman-Ford is in its basic structure very similar to Dijkstra's algorithm, but instead of greedily selecting the minimum-weight node not yet processed to relax, it simply relaxes all the edges, and does this $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. The repetitions allow minimum distances to accurately propagate throughout the graph, since, in the absence of negative cycles, the shortest path can only visit each node at most once. Unlike the greedy approach, which depends on certain structural assumptions derived from positive weights, this straightforward approach extends to the general case.

Bellman–Ford runs in $O(V \cdot E)$ time, where V and E are the number of vertices and edges respectively.
procedure BellmanFord(list vertices, list edges, vertex source)

// This implementation takes in a graph, represented as lists of vertices
// and edges, and modifies the vertices so that their distance and
// predecessor attributes store the shortest paths.

// Step 1: Initialize graph

for each vertex v in vertices:
    if v is source then v.distance := 0
    else v.distance := infinity
    v.predecessor := null

// Step 2: relax edges repeatedly

for i from 1 to size(vertices)-1:
    for each edge uv in edges:
        u := uv.source
        v := uv.destination // uv is the edge from u to v
        if v.distance > u.distance + uv.weight:
            v.distance := u.distance + uv.weight
            v.predecessor := u

// Step 3: check for negative-weight cycles

for each edge uv in edges:
    u := uv.source
    v := uv.destination
    if v.distance > u.distance + uv.weight:
Proof of correctness

The correctness of the algorithm can be shown by induction. The precise statement shown by induction is:

Lemma. After \( i \) repetitions of for cycle:

- If \( \text{Distance}(u) \) is not infinity, it is equal to the length of some path from \( s \) to \( u \);
- If there is a path from \( s \) to \( u \) with at most \( i \) edges, then \( \text{Distance}(u) \) is at most the length of the shortest path from \( s \) to \( u \) with at most \( i \) edges.

Proof. For the base case of induction, consider \( i=0 \) and the moment before for cycle is executed for the first time. Then, for the source vertex, \( \text{source.distance} = 0 \), which is correct. For other vertices \( u \), \( u\text{-distance} = \text{infinity} \), which is also correct because there is no path from source to \( u \) with 0 edges.

For the inductive case, we first prove the first part. Consider a moment when a vertex's distance is updated by \( v\text{-distance} := u\text{-distance} + uv\text{-weight} \). By inductive assumption, \( u\text{-distance} \) is the length of some path from source to \( u \). Then \( u\text{-distance} + uv\text{-weight} \) is the length of the path from source to \( v \) that follows the path from source to \( u \) and then goes to \( v \).

For the second part, consider the shortest path from source to \( u \) with at most \( i \) edges. Let \( v \) be the last vertex before \( u \) on this path. Then, the part of the path from source to \( v \) is the shortest path from source to \( v \) with at most \( i-1 \) edges. By inductive assumption, \( v\text{-distance} \) after \( i-1 \) cycles is at most the length of this path. Therefore, \( uv\text{-weight} + v\text{-distance} \) is at most the length of the path from \( s \) to \( u \). In the \( i \)th cycle, \( u\text{-distance} \) gets compared with \( uv\text{-weight} + v\text{-distance} \), and is set equal to it if \( uv\text{-weight} + v\text{-distance} \) was smaller. Therefore, after \( i \) cycles, \( u\text{-distance} \) is at most the length of the shortest path from source to \( u \) that uses at most \( i \) edges.

When \( i \) equals the number of vertices in the graph, each path will be the shortest path overall, unless there are negative-weight cycles. If a negative-weight cycle exists and is accessible from the source, then given any walk, a shorter one exists, so there is no shortest walk. Otherwise, the shortest walk will not include any cycles (because going around a cycle would make the walk shorter), so each shortest path visits each vertex at most once, and its number of edges is less than the number of vertices in the graph.
Applications in routing

A distributed variant of Bellman–Ford algorithm is used in distance-vector routing protocols, for example the Routing Information Protocol (RIP). The algorithm is distributed because it involves a number of nodes (routers) within an Autonomous system, a collection of IP networks typically owned by an ISP. It consists of the following steps:

1. Each node calculates the distances between itself and all other nodes within the AS and stores this information as a table.
2. Each node sends its table to all neighboring nodes.
3. When a node receives distance tables from its neighbors, it calculates the shortest routes to all other nodes and updates its own table to reflect any changes.

The main disadvantages of Bellman–Ford algorithm in this setting are

- Does not scale well
- Changes in network topology are not reflected quickly since updates are spread node-by-node.
- Counting to infinity (if link or node failures render a node unreachable from some set of other nodes, those nodes may spend forever gradually increasing their estimates of the distance to it, and in the meantime there may be routing loops)

Implementation

The following program implements the Bellman–Ford algorithm in C.

```c
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

/* Let INFINITY be an integer value not likely to be confused with a real weight, even a negative one. */
#define INFINITY ((1 << 14)-1)

typedef struct {
    int source;
}
int dest;
int weight;
}

} Edge;

void BellmanFord(Edge edges[], int edgecount, int nodecount, int source)
{

int *distance = (int*) malloc(nodecount * sizeof(*distance));

int i, j;

for (i=0; i < nodecount; i++)

distance[i] = INFINITY;

/* The source node distance is set to zero. */

distance[source] = 0;

for (i=0; i < nodecount; i++) {
    for (j=0; j < edgecount; j++) {
        if (distance[edges[j].source] != INFINITY) {
            int new_distance = distance[edges[j].source] + edges[j].weight;
            if (new_distance < distance[edges[j].dest])
                distance[edges[j].dest] = new_distance;
        }
    }
}

for (i=0; i < edgecount; i++) {
    if (distance[edges[i].dest] > distance[edges[i].source] + edges[i].weight) {
puts("Negative edge weight cycles detected!");
free(distance);
return;
}
for (i=0; i < nodecount; i++) {
printf("The shortest distance between nodes %d and %d is %d\n",
source, i, distance[i]);
}
free(distance);
return;
}
int main(void)
{

/* This test case should produce the distances 2, 4, 7, -2, and 0. */

Edge edges[10] = {{0,1, 5}, {0,2, 8}, {0,3, -4}, {1,0, -2},
{2,1, -3}, {2,3, 9}, {3,1, 7}, {3,4, 2},
{4,0, 6}, {4,2, 7}};
BellmanFord(edges, 10, 5, 4);
return 0;
}
7.3.5. Johnson’s algorithms

(From Wikipedia, the free encyclopedia)

Johnson's algorithm is a way to solve the all-pairs shortest path problem in a sparse, weighted, directed graph.

First, it adds a new node with zero weight edge from it to all other nodes, and runs the Bellman-Ford algorithm to check for negative weight cycles and find h(v), the least weight of a path from the new node to node v. Next it reweights the edges using the nodes' h(v) values. Finally for each node, it runs Dijkstra's algorithm and stores the computed least weight to other nodes, reweighted using the nodes' h(v) values, as the final weight. The time complexity is O(V^2log V + VE).

Union-find problem

(From Wikipedia, the free encyclopedia)

Given a set of elements, it is often useful to break them up or partition them into a number of separate, non-overlapping sets. A disjoint-set data structure is a data structure that keeps track of such a partitioning. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

• Find: Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
• Union: Combine or merge two sets into a single set.

Because it supports these two operations, a disjoint-set data structure is sometimes called a merge-find set. The other important operation, MakeSet, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved (see the Applications section).

In order to define these operations more precisely, we need some way of representing the sets. One common approach is to select a fixed element of each set, called its representative, to represent the set as a whole. Then, Find(x) returns the representative of the set that x belongs to, and Union takes two set representatives as its arguments.

Disjoint-set linked lists

Perhaps the simplest approach to creating a disjoint-set data structure is to create a linked list for each set. We choose the element at the head of the list as the representative.
MakeSet is obvious, creating a list of one element. Union simply appends the two lists, a constant-time operation. Unfortunately, with this implementation Find requires \( \Omega(n) \) or linear time with this approach.

We can avoid this by including in each linked list node a pointer to the head of the list; then Find takes constant time. However, we've now ruined the time of Union, which has to go through the elements of the list being appended to make them point to the head of the new combined list, requiring \( \Omega(n) \) time.

We can ameliorate this by always appending the smaller list to the longer, called the weighted union heuristic. This also requires keeping track of the length of each list as we perform operations to be efficient. Using this, a sequence of \( m \) MakeSet, Union, and Find operations on \( n \) elements requires \( O(m + n \log n) \) time. To make any further progress, we need to start over with a different data structure.

**Disjoint-set forests**

We now turn to disjoint-set forests, a data structure where each set is represented by a tree data structure where each node holds a reference to its parent node. Disjoint-set forests were first described by Bernard A. Galler and Michael J. Fisher in 1964, although their precise analysis took years.

In a disjoint-set forest, the representative of each set is the root of that set's tree. Find simply follows parent nodes until it reaches the root. Union combines two trees into one by attaching the root of one to the root of the other. One way of implementing these might be:

```plaintext
function MakeSet(x)
    x.parent := x

function Find(x)
    if x.parent == x
        return x
    else
        return Find(x.parent)

function Union(x, y)
    xRoot := Find(x)
```

121/174
yRoot := Find(y)

xRoot.parent := yRoot

In this naive form, this approach is no better than the linked-list approach, because the tree it creates can be highly unbalanced, but it can be enhanced in two ways.

The first way, called union by rank, is to always attach the smaller tree to the root of the larger tree, rather than vice versa. To evaluate which tree is larger, we use a simple heuristic called rank: one-element trees have a rank of zero, and whenever two trees of the same rank are unioned together, the result has one greater rank. Just applying this technique alone yields an amortized running-time of $O(\log n)$ per MakeSet, Union, or Find operation. Here are the improved MakeSet and Union:

function MakeSet(x)
    x.parent := x
    x.rank := 0

function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    if xRoot.rank > yRoot.rank
        yRoot.parent := xRoot
    else if xRoot.rank < yRoot.rank
        xRoot.parent := yRoot
    else if xRoot != yRoot
        yRoot.parent := xRoot
        xRoot.rank := xRoot.rank + 1

The second improvement, called path compression, is a way of flattening the structure of the tree whenever we use Find on it. The idea is that each node we visit on our way to a root node may as well be attached directly to the root node; they all share the same representative. To effect this, we make one traversal up to the root node, to find out what
it is, and then make another traversal, making this root node the immediate parent of all nodes along the path. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly. Here is the improved Find:

function Find(x)

if x.parent == x

return x

else

x.parent := Find(x.parent)

return x.parent

These two techniques complement each other; applied together, the amortized time per operation is only \( O(\alpha(n)) \), where \( \alpha(n) \) is the inverse of the function \( f(n) = A(n,n) \), and \( A \) is the extremely quickly-growing Ackermann function. Since \( \alpha(n) \) is its inverse, it's less than 5 for all remotely practical values of \( n \). Thus, the amortized running time per operation is effectively a small constant.

In fact, we can't get better than this: Fredman and Saks showed in 1989 that \( \Omega(\alpha(n)) \) words must be accessed by any disjoint-set data structure per operation on average.

**Applications**

Disjoint-set data structures arise naturally in many applications, particularly where some kind of partitioning or equivalence relation is involved, and this section discusses some of them.

**Tracking the connected components of an undirected graph**

Suppose we have an undirected graph and we want to efficiently make queries regarding the connected components of that graph, such as:

- Are two vertices of the graph in the same connected component?
- List all vertices of the graph in a particular component.
- How many connected components are there?

If the graph is static (not changing), we can simply use breadth-first search to associate a component with each vertex. However, if we want to keep track of these components
while adding additional vertices and edges to the graph, a disjoint-set data structure is much more efficient.

We assume the graph is empty initially. Each time we add a vertex, we use MakeSet to make a set containing only that vertex. Each time we add an edge, we use Union to union the sets of the two vertices incident to that edge. Now, each set will contain the vertices of a single connected component, and we can use Find to determine which connected component a particular vertex is in, or whether two vertices are in the same connected component.

This technique is used by the Boost Graph Library to implement its Incremental Connected Components functionality.

Note that this scheme doesn't allow deletion of edges — even without path compression or the rank heuristic, this is not as easy, although more complex schemes have been designed that can deal with this type of incremental update.

**Computing shorelines of a terrain**

When computing the contours of a 3D surface, one of the first steps is to compute the "shorelines," which surround local minima or "lake bottoms." We imagine we are sweeping a plane, which we refer to as the "water level," from below the surface upwards. We will form a series of contour lines as we move upwards, categorized by which local minima they contain. In the end, we will have a single contour containing all local minima.

Whenever the water level rises just above a new local minimum, it creates a small "lake," a new contour line that surrounds the local minimum; this is done with the MakeSet operation.

As the water level continues to rise, it may touch a saddle point, or "pass." When we reach such a pass, we follow the steepest downhill route from it on each side until we arrive a local minimum. We use Find to determine which contours surround these two local minima, then use Union to combine them. Eventually, all contours will be combined into one, and we are done.

**Classifying a set of atoms into molecules or fragments**

In computational chemistry, collisions involving the fragmentation of large molecules can be simulated using molecular dynamics. The result is a list of atoms and their positions. In the analysis, the union-find algorithm can be used to classify these atoms into fragments. Each atom is initially considered to be part of its own fragment. The Find step usually consists of testing the distance between pairs of atoms, though other
criterion like the electronic charge between the atoms could be used. The Union merges two fragments together. In the end, the sizes and characteristics of each fragment can be analyzed.

**Connected component labeling in image analysis**

In image analysis, some of the most efficient connected component labeling algorithms make use of union-find data structure. In this type of application, the time required form union-find operations is strictly linear.

**Connectivity**

(From Wikipedia, the free encyclopedia)

In mathematics and computer science, connectivity is one of the basic concepts of graph theory. It is closely related to the theory of network flow problems. The connectivity of a graph is an important measure of its robustness as a network.

Definitions of components, cuts and connectivity

In an undirected graph G, two vertices u and v are called connected if G contains a path from u to v. Otherwise, they are called disconnected. A graph is called connected if every pair of distinct vertices in the graph is connected. A connected component is a maximal connected subgraph of G. Each vertex belongs to exactly one connected component, as does each edge.

A directed graph is called weakly connected if replacing all of its directed edges with undirected edges produces a connected (undirected) graph. It is strongly connected or strong if it contains a directed path from u to v for every pair of vertices u,v. The strong components are the maximal strongly connected subgraphs

2-connectivity is also called "biconnectivity" and 3-connectivity is also called "triconnectivity".

A cut or vertex cut of a connected graph G is a set of vertices whose removal renders G disconnected. The connectivity or vertex connectivity \( \kappa(G) \) is the size of a smallest vertex cut. A graph is called k-connected or k-vertex-connected if its vertex connectivity is k or greater. A complete graph with n vertices has no cuts at all, but by convention its connectivity is n-1. A vertex cut for two vertices u and v is a set of vertices whose removal from the graph disconnects u and v. The local connectivity \( \kappa(u,v) \) is the size of a smallest vertex cut separating u and v. Local connectivity is symmetric; that is, \( \kappa(u,v) = \kappa(v,u) \). Moreover, \( \kappa(G) \) equals the minimum of \( \kappa(u,v) \) over all pairs of vertices u,v.
Analogous concepts can be defined for edges. Thus an edge cut of $G$ is a set of edges whose removal renders the graph disconnected, the edge-connectivity $\kappa'(G)$ is the size of a smallest edge cut, and the local edge-connectivity $\kappa'(u,v)$ of two vertices $u,v$ is the size of a smallest edge cut disconnecting $u$ from $v$. Again, local edge-connectivity is symmetric. A graph is called $k$-edge-connected if its edge connectivity is $k$ or greater.

All of these definitions and notations carry over to directed graphs. Local connectivity and local edge-connectivity are not necessarily symmetric for directed graphs.

Menger's theorem

One of the most important facts about connectivity in graphs is Menger's theorem, which characterizes the connectivity and edge-connectivity of a graph in terms of the number of independent paths between vertices.

If $u$ and $v$ are vertices of a graph $G$, then a collection of paths between $u$ and $v$ is called independent if no two of them share a vertex (other than $u$ and $v$ themselves). Similarly, the collection is edge-independent if no two paths in it share an edge. The greatest number of independent paths between $u$ and $v$ is written as $\lambda(u,v)$, and the greatest number of edge-independent paths between $u$ and $v$ is written as $\lambda'(u,v)$.

Menger's theorem asserts that $\kappa(u,v) = \lambda(u,v)$ and $\kappa'(u,v) = \lambda'(u,v)$ for every pair of vertices $u$ and $v$. This fact is actually a special case of the max-flow min-cut theorem.

Computational aspects

The problem of determining whether two vertices in a graph are connected can be solved efficiently using a search algorithm, such as breadth-first search. More generally, it is easy to determine computationally whether a graph is connected (for example, by using a disjoint-set data structure), or to count the number of connected components.

By Menger's theorem, for any two vertices $u$ and $v$ in a connected graph $G$, the numbers $\kappa(u,v)$ and $\kappa'(u,v)$ can be determined efficiently using the max-flow min-cut algorithm. The connectivity and edge-connectivity of $G$ can then be computed as the minimum values of $\kappa(u,v)$ and $\kappa'(u,v)$, respectively.

In computational complexity theory, SL is the class of problems log-space reducible to the problem of determining whether two vertices in a graph are connected, which was proved to be equal to L by Omer Reingold in 2004. Hence, directed graph connectivity may be solved in $O(\log n)$ space.

Examples

- The vertex- and edge-connectivities of a disconnected graph are both 0.
• 1-connectedness is synonymous with connectedness.
• The complete graph on \( n \) vertices has edge-connectivity equal to \( n - 1 \). Every other simple graph on \( n \) vertices has strictly smaller edge-connectivity.
• In a tree, the local edge-connectivity between every pair of vertices is 1.

Properties

• Connectedness is preserved by graph homomorphisms.
• If \( G \) is connected then its line graph \( L(G) \) is also connected.
• The vertex-connectivity of a graph is less than or equal to its edge-connectivity.
  That is, \( \kappa(G) \leq \kappa'(G) \).
• If a graph \( G \) is \( k \)-connected, then for every set of vertices \( U \) of cardinality \( k \), there exists a cycle in \( G \) containing \( U \). The converse is true when \( k = 2 \).

A graph \( G \) is 2-edge-connected if and only if it has an orientation that is strongly connected.

7.5.1. Non-direction connectivity

(From Wikipedia, the free encyclopedia)

A undirected graph \( G \) is an ordered pair \( G:=(V,E) \) that is subject to the following conditions:

• \( V \) is a set, whose elements are called vertices or nodes,
• \( E \) is a set of pairs (unordered) of distinct vertices, called edges or lines

The vertices belonging to an edge are called the ends, endpoints, or end vertices of the edge.

V (and hence E) are usually taken to be finite sets, and many of the well-known results are not true (or are rather different) for infinite graphs because many of the arguments fail in the infinite case. The order of a graph is \(|V|\) (the number of vertices). A graph's size is \(|E|\), the number of edges. The degree of a vertex is the number of other vertices it is connected to by edges.
The edge set $E$ induces a symmetric binary relation $\sim$ on $V$ that is called the adjacency relation of $G$. Specifically, for each edge $\{u,v\}$ the vertices $u$ and $v$ are said to be adjacent to one another, which is denoted $u \sim v$.

For an edge $\{u, v\}$ graph theorists usually use the somewhat shorter notation $uv$.

### 7.5.2. Direction connectivity

(From Wikipedia, the free encyclopedia)

A directed graph or digraph $G$ is an ordered pair $G: = (V,A)$ with

- $V$ is a set, whose elements are called vertices or nodes,
- $A$ is a set of ordered pairs of vertices, called directed edges, arcs, or arrows.

An arc $e = (x,y)$ is considered to be directed from $x$ to $y$; $y$ is called the head and $x$ is called the tail of the arc; $y$ is said to be a direct successor of $x$, and $x$ is said to be a direct predecessor of $y$. If a path leads from $x$ to $y$, then $y$ is said to be a successor of $x$, and $x$ is said to be a predecessor of $y$. The arc $(y,x)$ is called the arc $(x,y)$ inverted.

A directed graph is called symmetric if every arc belongs to it together with the corresponding inverted arc. A symmetric loopless directed graph is equivalent to an undirected graph with the pairs of inverted arcs replaced with edges; thus the number of edges is equal to the number of arcs halved.

A variation on this definition is the oriented graph, which is a graph (or multigraph; see below) with an orientation or direction assigned to each of its edges. A distinction between a directed graph and an oriented simple graph is that if $x$ and $y$ are vertices, a directed graph allows both $(x,y)$ and $(y,x)$ as edges, while only one is permitted in an oriented graph. A more fundamental difference is that, in a directed graph (or multigraph), the directions are fixed, but in an oriented graph (or multigraph), only the underlying graph is fixed, while the orientation may vary.

A directed acyclic graph, occasionally called a dag or DAG, is a directed graph with no directed cycles.
A quiver is simply a directed graph, but the context is different. When discussing quivers emphasis is placed on representations of the graph where vector spaces are attached to the vertices and linear transformations are attached to the arcs.

**Mixed graph**

A mixed graph $G$ is a graph in which some edges may be directed and some may be undirected. It is written as an ordered triple $G := (V, E, A)$ with $V$, $E$, and $A$ defined as above. Directed and undirected graphs are special cases.

**Topological sort**

(From Wikipedia, the free encyclopedia)

An undirected graph can be viewed as a simplicial complex $C$ with a single-element set per vertex and a two-element set per edge. The geometric realization $|C|$ of the complex consists of a copy of the unit interval $[0,1]$ per edge, with the endpoints of these intervals glued together at vertices. In this view, embeddings of graphs into a surface or as subdivisions of other graphs are both instances of topological embedding, homeomorphism of graphs is just the specialization of topological homeomorphism, the notion of a connected graph coincides with topological connectedness, and a connected graph is a tree if and only if its fundamental group is trivial.

Other simplicial complexes associated with graphs include the Whitney complex or clique complex, with a set per clique of the graph, and the matching complex, with a set per matching of the graph (equivalently, the clique complex of the complement of the line graph). The matching complex of a complete bipartite graph is called a chessboard complex, as it can be also described as the complex of sets of non-attacking rooks on a chessboard.

**Examples**

The canonical application of topological sorting is in scheduling a sequence of jobs. The jobs are represented by vertices, and there is an edge from $x$ to $y$ if job $x$ must be completed before job $y$ can be done (for example, washing machine must finish before we put the clothes to dry). Then, a topological sort gives an order in which to perform the jobs. This has applications in computer science, such as in instruction scheduling, ordering of formula cell evaluation in spreadsheets, dependencies in makefiles, and symbol dependencies in linkers.
Algorithms

The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges (Θ(|V|+|E|)).

One of these algorithms works by choosing vertices in the same order as the eventual topological sort. First, find a list of "start nodes" which have no incoming edges and insert them into a queue Q (at least one such node must exist if graph is acyclic). Then,

Q ← Set of all nodes with no incoming edges

while Q is non-empty do

remove a node n from Q

output n

for each node m with an edge e from n to m do

remove edge e from the graph

if m has no other incoming edges then

insert m into Q

if graph has edges then

output error message (graph has a cycle)

If this algorithm terminates without outputing all the nodes of the graph, it means the graph has at least one cycle and therefore is not a DAG, so a topological sort is impossible. Note that, reflecting the non-uniqueness of the resulting sort, the structure Q need not be a queue; it may be a stack or simply a set.
An alternative algorithm for topological sorting is based on depth-first search. Loop through the vertices of the graph, in any order, initiating a depth first search for any vertex that has not already been visited by a previous search. The desired topological sorting is the reverse postorder of these searches. That is, we can construct the ordering as a list of vertices, by adding each vertex to the start of the list at the time when the depth first search is processing that vertex and has returned from processing all children of that vertex. Since each edge and vertex is visited once, the algorithm runs in linear time.
Hashing

Introduction to hashing algorithms

(From Wikipedia, the free encyclopedia)

Hash algorithms are designed to be fast and to yield few hash collisions in expected input domains. In hash tables and data processing, collisions inhibit the distinguishing of data, making records more costly to find.

A hash algorithm must be deterministic, i.e. if two hashes generated by some hash function are different, then the two inputs were different in some way.

Hash algorithms are usually not injective, i.e. the computed hash value may be the same for different input values. This is because it is usually a requirement that the hash value can be stored in fewer bits than the data being hashed. It is a design goal of hash functions to minimize the likelihood of such a hash collision occurring.

A desirable property of a hash function is the mixing property: a small change in the input (e.g. one bit) should cause a large change in the output (e.g. about half of the bits). This is called the avalanche effect.

Typical hash functions have an infinite domain, such as bytestrings of arbitrary length, and a finite range, such as bit sequences of some fixed length. In certain cases, hash functions can be designed with one-to-one mapping between identically sized domain and range. Hash functions that are one-to-one are also called permutations. Reversibility is achieved by using a series of reversible "mixing" operations on the function input.

Hash list

(From Wikipedia, the free encyclopedia)

In computer science, a hash list is typically a list of hashes of the data blocks in a file or set of files. Lists of hashes are used for many different purposes, such as fast table lookup (hash tables) and distributed databases (distributed hash tables). This article covers hash lists that are used to guarantee data integrity.
A hash list is an extension of the old concept of hashing an item (for instance, a file). A hash list is usually sufficient for most needs, but a more advanced form of the concept is a hash tree.

Hash lists can be used to protect any kind of data stored, handled and transferred in and between computers. Currently the main use of hash lists is to make sure that data blocks received from other peers in a peer-to-peer network are received undamaged and unaltered, and to check that the other peers do not "lie" and send fake blocks.

Usually a cryptographic hash function such as SHA-1 is used for the hashing. If the hash list only needs to protect against unintentional damage less secure checksums such as CRCs can be used.

Hash lists are better than a simple hash of the entire file since, in the case of a data block being damaged, this is noticed, and only the damaged block needs to be redownloaded. With only a hash of the file, the whole file would have to be redownloaded instead, since it would be impossible to determine which part of the file was damaged. Hash lists also protect against nodes that try to sabotage by sending fake blocks, since in such a case the damaged block can be acquired from some other source.

Often, an additional hash of the hash list itself (a top hash, also called root hash or master hash) is used. Before downloading a file on a p2p network, in most cases the top hash is acquired from a trusted source, for instance a friend or a web site that is known to have good recommendations of files to download. When the top hash is available, the hash list can be received from any non-trusted source, like any peer in the p2p network. Then the received hash list is checked against the trusted top hash, and if the hash list is damaged or fake, another hash list from another source will be tried until the program finds one that matches the top hash.

**Hash table**

(From Wikipedia, the free encyclopedia)
In computer science, a hash table, or a hash map, is a data structure that associates keys with values. The primary operation it supports efficiently is a lookup: given a key (e.g. a person's name), find the corresponding value (e.g. that person's telephone number). It works by transforming the key using a hash function into a hash, a number that is used to index into an array to locate the desired location ("bucket") where the values should be.

Hash tables support the efficient addition of new entries, and the time spent searching for the required data is independent of the number of items stored (i.e. $O(1)$.)

Choosing a good hash function

A good hash function is essential for good hash table performance. A poor choice of a hash function is likely to lead to clustering, in which probability of keys mapping to the same hash bucket (i.e. a collision) is significantly greater than would be expected from a random function. A nonzero collision probability is inevitable in any hash implementation, but usually the number of operations required to resolve a collision scales linearly with the number of keys mapping to the same bucket, so excess collisions will degrade performance significantly. In addition, some hash functions are computationally expensive, so the amount of time (and, in some cases, memory) taken to compute the hash may be burdensome.

Choosing a good hash function is tricky. Simplicity and speed are readily measured objectively (by number of lines of code and CPU benchmarks, for example), but strength is a more slippery concept. Obviously, a cryptographic hash function such as SHA-1 would satisfy the relatively lax strength requirements needed for hash tables, but their slowness and complexity makes them unappealing. However, using cryptographic hash functions can protect against collision attacks when the hash table modulus and its factors can be kept secret from the attacker or alternatively, by applying a secret salt. However, for these specialized cases, a universal hash function can be used instead of one static hash.
In the absence of a standard measure for hash function strength, the current state of the art is to employ a battery of statistical tests to measure whether the hash function can be readily distinguished from a random function. Arguably the most important test is to determine whether the hash function displays the avalanche effect, which essentially states that any single-bit change in the input key should affect on average half the bits in the output. Bret Mulvey advocates testing the strict avalanche condition in particular, which states that, for any single-bit change, each of the output bits should change with probability one-half, independent of the other bits in the key. Purely additive hash functions such as CRC fail this stronger condition miserably.

Clearly, a strong hash function should have a uniform distribution of hash values. Bret Mulvey proposes the use of a chi-squared test for uniformity, based on power of two hash table sizes ranging from 2^1 to 2^16. This test is considerably more sensitive than many others proposed for measuring hash functions, and finds problems in many popular hash functions.

Fortunately, there are good hash functions that satisfy all these criteria. The simplest class all consume one byte of the input key per iteration of the inner loop. Within this class, simplicity and speed are closely related, as fast algorithms simply don't have time to perform complex calculations.

A mathematical byte-by-byte implementation that performs particularly well is the Jenkins One-at-a-time hash, adapted here from an article by Bob Jenkins, its creator.

```c
uint32 joaat_hash(uchar *key, size_t key_len)
{
    uint32 hash = 0;
    size_t i;
    for (i = 0; i < key_len; i++) {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }
    hash += (hash << 3);
    return hash;
}
```
hash ^= (hash >> 11);
hash += (hash << 15);
return hash;
}

The avalanche behavior of this hash is shown on the right. The image was made using Bret Mulvey's AvalancheTest in his Hash.cs toolset.

Each of the 24 rows corresponds to a single bit in the 3-byte input key, and each of the 32 columns corresponds to a bit in the output hash. Colors are chosen by how well the input key bit affects the given output hash bit: a green square indicates good mixing behavior, a yellow square weak mixing behavior, and red would indicate no mixing. Only a few bits in the last byte of the output hash are weakly mixed, a performance vastly better than a number of widely used hash functions.

Many commonly used hash functions perform poorly when subjected to such rigorous avalanche testing. The widely favored FNV hash, for example, shows many bits with no mixing at all, especially for short keys. See the evaluation of FNV by Bret Mulvey for a more thorough analysis.

If speed is more important than simplicity, then the class of hash functions which consume multibyte chunks per iteration may be of interest. One of the most sophisticated is "lookup3" by Bob Jenkins, which consumes input in 12 byte (96 bit) chunks. Note, though, that any speed improvement from the use of this hash is only likely to be useful for large keys, and that the increased complexity may also have speed consequences such as preventing an optimizing compiler from inlining the hash function. Bret Mulvey analyzed an earlier version, lookup2, and found it to have excellent avalanche behavior.
One desirable property of a hash function is that conversion from the hash value (typically 32 bits) to a bucket index for a particular-size hash table can be done simply by masking, preserving only the lower k bits for a table of size 2k (an operation equivalent to computing the hash value modulo the table size). This property enables the technique of incremental doubling of the size of the hash table - each bucket in the old table maps to only two in the new table. Because of its use of XOR-folding, the FNV hash does not have this property. Some older hashes are even worse, requiring table sizes to be a prime number rather than a power of two, again computing the bucket index as the hash value modulo the table size. In general, such a requirement is a sign of a fundamentally weak function; using a prime table size is a poor substitute for using a stronger function.

Collision resolution

If two keys hash to the same index, the corresponding records cannot be stored in the same location. So, if it's already occupied, we must find another location to store the new record, and do it so that we can find it when we look it up later on.

To give an idea of the importance of a good collision resolution strategy, consider the following result, derived using the birthday paradox. Even if we assume that our hash function outputs random indices uniformly distributed over the array, and even for a hash table with 1 million indices, there is a 95% chance of at least one collision occurring before it contains 2500 records.

There are a number of collision resolution techniques, but the most popular are chaining and open addressing.

Chaining

![Hash collision resolved by chaining](image)

In the simplest chained hash table technique, each slot in the array references a linked list of inserted records that collide to the same slot. Insertion requires finding the correct
slot, and appending to either end of the list in that slot; deletion requires searching the list and removal.

Chained hash tables have advantages over open addressed hash tables in that the removal operation is simple and resizing the table can be postponed for a much longer time because performance degrades more gracefully even when every slot is used. Indeed, many chaining hash tables may not require resizing at all since performance degradation is linear as the table fills. For example, a chaining hash table containing twice its recommended capacity of data would only be about twice as slow on average as the same table at its recommended capacity.

Chained hash tables inherit the disadvantages of linked lists. When storing small records, the overhead of the linked list can be significant. An additional disadvantage is that traversing a linked list has poor cache performance.

Alternative data structures can be used for chains instead of linked lists. By using a self-balancing tree, for example, the theoretical worst-case time of a hash table can be brought down to $O(\log n)$ rather than $O(n)$. However, since each list is intended to be short, this approach is usually inefficient unless the hash table is designed to run at full capacity or there are unusually high collision rates, as might occur in input designed to cause collisions. Dynamic arrays can also be used to decrease space overhead and improve cache performance when records are small.

Some chaining implementations use an optimization where the first record of each chain is stored in the table. The purpose is to increase cache efficiency of hash table access. In order to avoid wasting large amounts of space, such hash tables would maintain a load factor of 1.0 or greater.

Open addressing

![Hash collision resolved by linear probing (interval=1)](image)
Open addressing hash tables store the records directly within the array. This approach is also called closed hashing. A hash collision is resolved by probing, or searching through alternate locations in the array (the probe sequence) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. [2] Well known probe sequences include:

linear probing

in which the interval between probes is fixed--often at 1.

quadratic probing

in which the interval between probes increases linearly (hence, the indices are described by a quadratic function).

double hashing

in which the interval between probes is fixed for each record but is computed by another hash function.

The main tradeoffs between these methods are that linear probing has the best cache performance but is most sensitive to clustering, while double hashing has poor cache performance but exhibits virtually no clustering; quadratic probing falls in-between in both areas. Double hashing can also require more computation than other forms of probing. Some open addressing methods, such as last-come-first-served hashing and cuckoo hashing move existing keys around in the array to make room for the new key. This gives better maximum search times than the methods based on probing.

A critical influence on performance of an open addressing hash table is the load factor; that is, the proportion of the slots in the array that are used. As the load factor increases towards 100%, the number of probes that may be required to find or insert a given key rises dramatically. Once the table becomes full, probing algorithms may even fail to terminate. Even with good hash functions, load factors are normally limited to 80%. A poor hash function can exhibit poor performance even at very low load factors by generating significant clustering. What causes hash functions to cluster is not well understood, and it is easy to unintentionally write a hash function which causes severe clustering.

Example pseudocode

The following pseudocode is an implementation of an open addressing hash table with linear probing and single-slot stepping, a common approach that is effective if the hash function is good. Each of the lookup, set and remove functions use a common internal function findSlot to locate the array slot that either does or should contain a given key.
record pair { key, value }

var pair array slot[0..num_slots-1]

function find_slot(key)
    i := hash(key) modulo num_slots

    // search until we either find the key, or find an empty slot.
    while ( (slot[i] is occupied) and ( slot[i].key ≠ key ) ) do
        i := (i + 1) modulo num_slots
        repeat
    return i

function lookup(key)
    i := find_slot(key)

    if slot[i] is occupied // key is in table
        return slot[i].value
    else // key is not in table
        return not found

function set(key, value)
    i := find_slot(key)

    if slot[i] is occupied
        slot[i].value := value
    else
        if the table is almost full
            rebuild the table larger (note 1)
i := find_slot(key)

slot[i].key := key

slot[i].value := value

Another example showing open addressing technique. Presented function is converting each part(4) of an internet protocol address, where NOT is bitwise NOT, XOR is bitwise XOR, OR is bitwise OR, AND is bitwise AND and << and >> are shift-left and shift-right:

// key_1,key_2,key_3,key_4 are following 3-digit numbers - parts of ip address xxx.xxx.xxx.xxx

function ip(key parts)

j := 1

do

key := (key_2 << 2)

key := (key + (key_3 << 7))

key := key + (j OR key_4 >> 2) * (key_4) * (j + key_1) XOR j

key := key AND _prime_ // _prime_ is a prime number

j := (j+1)

while collision

return key

Rebuilding the table requires allocating a larger array and recursively using the set operation to insert all the elements of the old array into the new larger array. It is common to increase the array size exponentially, for example by doubling the old array size.

function remove(key)

i := find_slot(key)

if slot[i] is unoccupied
return // key is not in the table
j := i
loop
    j := (j+1) modulo num_slots
    if slot[j] is unoccupied
        exit loop
    k := hash(slot[j].key) modulo num_slots
    if (j > i and (k <= i or k > j)) or (j < i and (k <= i and k > j)) (note 2)
        slot[i] := slot[j]
        i := j
    mark slot[i] as unoccupied

For all records in a cluster, there must be no vacant slots between their natural hash position and their current position (else lookups will terminate before finding the record). At this point in the pseudocode, i is a vacant slot that might be invalidating this property for subsequent records in the cluster. j is such a subsequent record. k is the raw hash where the record at j would naturally land in the hash table if there were no collisions. This test is asking if the record at j is invalidly positioned with respect to the required properties of a cluster now that i is vacant.

Another technique for removal is simply to mark the slot as deleted. However this eventually requires rebuilding the table simply to remove deleted records. The methods above provide O(1) updating and removal of existing records, with occasional rebuilding if the high water mark of the table size grows.

The O(1) remove method above is only possible in linearly probed hash tables with single-slot stepping. In the case where many records are to be deleted in one operation, marking the slots for deletion and later rebuilding may be more efficient.

Open addressing versus chaining

Chained hash tables have the following benefits over open addressing:
- They are simple to implement effectively and only require basic data structures.
- From the point of view of writing suitable hash functions, chained hash tables are insensitive to clustering, only requiring minimization of collisions. Open addressing depends upon better hash functions to avoid clustering. This is particularly important if novice programmers can add their own hash functions, but even experienced programmers can be caught out by unexpected clustering effects.
- They degrade in performance more gracefully. Although chains grow longer as the table fills, a chained hash table cannot "fill up" and does not exhibit the sudden increases in lookup times that occur in a near-full table with open addressing. (see right)
- If the hash table stores large records, about 5 or more words per record, chaining uses less memory than open addressing.
- If the hash table is sparse (that is, it has a big array with many free array slots), chaining uses less memory than open addressing even for small records of 2 to 4 words per record due to its external storage.

![Graph comparing average cache misses per lookup with chaining and linear probing. As the table passes the 80%-full mark, linear probing's performance drastically degrades.](image)

For small record sizes (a few words or less) the benefits of in-place open addressing compared to chaining are:

- They can be more space-efficient than chaining since they don't need to store any pointers or allocate any additional space outside the hash table. Simple linked lists require a word of overhead per element.
- Insertions avoid the time overhead of memory allocation, and can even be implemented in the absence of a memory allocator.
- Because it uses internal storage, open addressing avoids the extra indirection required for chaining's external storage. It also has better locality of reference,
particularly with linear probing. With small record sizes, these factors can yield
to better performance than chaining, particularly for lookups.
• They can be easier to serialize, because they don't use pointers.

On the other hand, normal open addressing is a poor choice for large elements, since
these elements fill entire cache lines (negating the cache advantage), and a large amount
of space is wasted on large empty table slots. If the open addressing table only stores
references to elements (external storage), it uses space comparable to chaining even for
large records but loses its speed advantage.

Generally speaking, open addressing is better used for hash tables with small records
that can be stored within the table (internal storage) and fit in a cache line. They are
particularly suitable for elements of one word or less. In cases where the tables are
expected to have high load factors, the records are large, or the data is variable-sized,
chained hash tables often perform as well or better.

Ultimately, used sensibly any kind of hash table algorithm is usually fast enough; and
the percentage of a calculation spent in hash table code is low. Memory usage is rarely
considered excessive. Therefore, in most cases the differences between these algorithms
is marginal, and other considerations typically come into play.

Coalesced hashing

A hybrid of chaining and open addressing, coalesced hashing links together chains
of nodes within the table itself. Like open addressing, it achieves space usage and
(somewhat diminished) cache advantages over chaining. Like chaining, it does not
exhibit clustering effects; in fact, the table can be efficiently filled to a high density.
Unlike chaining, it cannot have more elements than table slots.

Perfect hashing

If all of the keys that will be used are known ahead of time, and there are no more keys
than can fit the hash table, perfect hashing can be used to create a perfect hash table, in
which there will be no collisions. If minimal perfect hashing is used, every location in
the hash table can be used as well.

Perfect hashing gives a hash table where the time to make a lookup is constant in the
worst case. This is in contrast to chaining and open addressing methods, where the
time for lookup is low on average, but may be arbitrarily large. There exist methods
for maintaining a perfect hash function under insertions of keys, known as dynamic
perfect hashing. A simpler alternative, that also gives worst case constant lookup time,
is cuckoo hashing.

Probabilistic hashing
Perhaps the simplest solution to a collision is to replace the value that is already in the slot with the new value, or slightly less commonly, drop the record that is to be inserted. In later searches, this may result in a search not finding a record which has been inserted. This technique is particularly useful for implementing caching.

An even more space-efficient solution which is similar to this is use a bit array (an array of one-bit fields) for our table. Initially all bits are set to zero, and when we insert a key, we set the corresponding bit to one. False negatives cannot occur, but false positives can, since if the search finds a 1 bit, it will claim that the value was found, even if it was just another value that hashed into the same array slot by coincidence. In reality, such a hash table is merely a specific type of Bloom filter.

Robin Hood hashing

One interesting variation on double-hashing collision resolution is that of Robin Hood hashing. The idea is that a key already inserted may be displaced by a new key if its probe count is larger than the key at the current position. The net effect of this is that it reduces worst case search times in the table. This is similar to Knuth's ordered hash tables except the criteria for bumping a key does not depend on a direct relationship between the keys.

Table resizing

With a good hash function, a hash table can typically contain about 70%–80% as many elements as it does table slots and still perform well. Depending on the collision resolution mechanism, performance can begin to suffer either gradually or dramatically as more elements are added. To deal with this, when the load factor exceeds some threshold, it is necessary to allocate a new, larger table, and add all the contents of the original table to this new table. In Java's HashMap class, for example, the default load factor threshold is 0.75.

This can be a very expensive operation, and the necessity for it is one of the hash table's disadvantages. In fact, some naive methods for doing this, such as enlarging the table by one each time you add a new element, reduce performance so drastically as to make the hash table useless. However, if the table is enlarged by some fixed percent, such as 10% or 100%, it can be shown using amortized analysis that these resizings are so infrequent that the average time per lookup remains constant-time. To see why this is true, suppose a hash table using chaining begins at the minimum size of 1 and is doubled each time it fills above 100%. If in the end it contains n elements, then the total add operations performed for all the resizings is:

\[ 1 + 2 + 4 + ... + n = 2n - 1. \]
Because the costs of the resizings form a geometric series, the total cost is O(n). But it is necessary also to perform n operations to add the n elements in the first place, so the total time to add n elements with resizing is O(n), an amortized time of O(1) per element.

On the other hand, some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. One simple approach is to initially allocate the table with enough space for the expected number of elements and forbid the addition of too many elements. Another useful but more memory-intensive technique is to perform the resizing gradually:

- Allocate the new hash table, but leave the old hash table and check both tables during lookups.
- Each time an insertion is performed, add that element to the new table and also move k elements from the old table to the new table.
- When all elements are removed from the old table, deallocate it.

To ensure that the old table will be completely copied over before the new table itself needs to be enlarged, it's necessary to increase the size of the table by a factor of at least (k + 1)/k during the resizing.

Linear hashing is a hash table algorithm that permits incremental hash table expansion. It is implemented using a single hash table, but with two possible look-up functions.

Another way to decrease the cost of table resizing is to choose a hash function in such a way that the hashes of most values do not change when the table is resized. This approach, called consistent hashing, is prevalent in disk-based and distributed hashes, where resizing is prohibitively costly.

Ordered retrieval issue

Hash tables store data in pseudo-random locations, so accessing the data in a sorted manner is a very time consuming operation. Other data structures such as self-balancing binary search trees generally operate more slowly (since their lookup time is O(log n)) and are rather more complex to implement than hash tables but maintain a sorted data structure at all times. See a comparison of hash tables and self-balancing binary search trees.

Problems with hash tables

Although hash table lookups use constant time on average, the time spent can be significant. Evaluating a good hash function can be a slow operation. In particular, if simple array indexing can be used instead, this is usually faster.
Hash tables in general exhibit poor locality of reference—that is, the data to be accessed is distributed seemingly at random in memory. Because hash tables cause access patterns that jump around, this can trigger microprocessor cache misses that cause long delays. Compact data structures such as arrays, searched with linear search, may be faster if the table is relatively small and keys are cheap to compare, such as with simple integer keys. According to Moore's Law, cache sizes are growing exponentially and so what is considered "small" may be increasing. The optimal performance point varies from system to system; for example, a trial on Parrot shows that its hash tables outperform linear search in all but the most trivial cases (one to three entries).

More significantly, hash tables are more difficult and error-prone to write and use. Hash tables require the design of an effective hash function for each key type, which in some situations is more difficult and time-consuming to design and debug than the simple comparison function required for a self-balancing binary search tree. In open-addressed hash tables it's fairly easy to create a poor hash function.

Additionally, in some applications, a black hat with knowledge of the hash function may be able to supply information to a hash which creates worst-case behavior by causing excessive collisions, resulting in very poor performance (i.e., a denial of service attack). In critical applications, either universal hashing can be used or a data structure with better worst-case guarantees may be preferable.

**Hash tree**

(From Wikipedia, the free encyclopedia)

In cryptography, hash trees (also known as Merkle trees) are an extension of the simpler concept hash list, which in turn is an extension of the old concept of hashing.
Hash trees can be used to protect any kind of data stored, handled and transferred in and between computers. Currently the main use of hash trees is to make sure that data blocks received from other peers in a peer-to-peer network are received undamaged and unaltered, and even to check that the other peers do not lie and send fake blocks. Suggestions have been made to use hash trees in trusted computing systems.

How hash trees work

A hash tree is a tree of hashes in which the leaves are hashes of data blocks in, for instance, a file or set of files. Nodes further up in the tree are the hashes of their respective children. For example, in the picture hash 0 is the result of hashing hash 0-0 and then hash 0-1. That is, hash 0 = hash( hash 0-0 | hash 0-1 ).

Most hash tree implementations are binary (two child nodes under each node) but they can just as well use many more child nodes under each node.

Usually, a cryptographic hash function such as SHA-1, Whirlpool, or Tiger is used for the hashing. If the hash tree only needs to protect against unintentional damage, the much less secure checksums such as CRCs can be used.

In the top of a hash tree there is a top hash (or root hash or master hash). Before downloading a file on a p2p network, in most cases the top hash is acquired from a trusted source, for instance a friend or a web site that is known to have good recommendations of files to download. When the top hash is available, the hash tree can be received from any non-trusted source, like any peer in the p2p network. Then, the received hash tree is checked against the trusted top hash, and if the hash tree is damaged or fake, another hash tree from another source will be tried until the program finds one that matches the top hash.

The main difference from a hash list is that one branch of the hash tree can be downloaded at a time and the integrity of each branch can be checked immediately, even though the whole tree is not available yet. This can be an advantage since it is efficient to split files up in very small data blocks so that only small blocks have to be redownloaded if they get damaged. If the hashed file is very big, such a hash tree or hash list becomes fairly big. But if it is a tree, one small branch can be downloaded quickly, the integrity of the branch can be checked, and then the downloading of data blocks can start.

There are several additional tricks, benefits and details regarding hash trees. See the references and external links below for more in-depth information.

**Choosing hash functions**

(From Wikipedia, the free encyclopedia)
A hash function is a reproducible method of turning some kind of data into a (relatively) small number that may serve as a digital "fingerprint" of the data. The algorithm "chops and mixes" (i.e., substitutes or transposes) the data to create such fingerprints. The fingerprints are called hash sums, hash values, hash codes or simply hashes. (Note that hashes can also mean the hash functions.) Hash sums are commonly used as indices into hash tables or hash files. Cryptographic hash functions are used for various purposes in information security applications.

![A typical hash function at work](image)

**Properties**

Hash functions are designed to be fast and to yield few hash collisions in expected input domains. In hash tables and data processing, collisions inhibit the distinguishing of data, making records more costly to find.

A hash function must be deterministic, i.e. if two hashes generated by some hash function are different, then the two inputs were different in some way.

Hash functions are usually not injective, i.e. the computed hash value may be the same for different input values. This is because it is usually a requirement that the hash value can be stored in fewer bits than the data being hashed. It is a design goal of hash functions to minimize the likelihood of such a hash collision occurring.

A desirable property of a hash function is the mixing property: a small change in the input (e.g. one bit) should cause a large change in the output (e.g. about half of the bits). This is called the avalanche effect.

Typical hash functions have an infinite domain, such as byte strings of arbitrary length, and a finite range, such as bit sequences of some fixed length. In certain cases, hash functions can be designed with one-to-one mapping between identically sized domain and range. Hash functions that are one-to-one are also called permutations. Reversibility is achieved by using a series of reversible "mixing" operations on the function input.
Applications

Because of the variety of applications for hash functions (details below), they are often tailored to the application. For example, cryptographic hash functions assume the existence of an adversary who can deliberately try to find inputs with the same hash value. A well designed cryptographic hash function is a "one-way" operation: there is no practical way to calculate a particular data input that will result in a desired hash value, so it is also very difficult to forge. Functions intended for cryptographic hashing, such as MD5, are commonly used as stock hash functions.

Functions for error detection and correction focus on distinguishing cases in which data has been disturbed by random processes. When hash functions are used for checksums, the relatively small hash value can be used to verify that a data file of any size has not been altered.

Cryptography

A typical cryptographic one-way function is not one-to-one and makes an effective hash function; a typical cryptographic trapdoor function is one-to-one and makes an effective randomization function.

Hash tables

Hash tables, a major application for hash functions, enable fast lookup of a data record given its key. (Note: Keys are not usually secret as in cryptography, but both are used to "unlock" or access information.) For example, keys in an English dictionary would be English words, and their associated records would contain definitions. In this case, the hash function must map alphabetic strings to indexes for the hash table's internal array.

The ideal for a hash table's hash function is to map each key to a unique index, because this guarantees access to each data record in the first probe into the table. However, this is often impossible or impractical.

Hash functions that are truly random with uniform output (including most cryptographic hash functions) are good in that, on average, only one or two probes will be needed (depending on the load factor). Perhaps as important is that excessive collision rates with random hash functions are highly improbable—if not computationally infeasible for an adversary. However, a small, predictable number of collisions are virtually inevitable.

In many cases, a heuristic hash function can yield many fewer collisions than a random hash function. Heuristic functions take advantage of regularities in likely sets of keys. For example, one could design a heuristic hash function such that file names such as FILE0000.CHK, FILE0001.CHK, FILE0002.CHK, etc. map to successive indices of the table, meaning that such sequences will not collide. Beating a random hash function on
"good" sets of keys usually means performing much worse on "bad" sets of keys, which can arise naturally—not just through attacks. Bad performance of a hash table's hash function means that lookup can degrade to a costly linear search.

Aside from minimizing collisions, the hash function for a hash table should also be fast relative to the cost of retrieving a record in the table, as the goal of minimizing collisions is minimizing the time needed to retrieve a desired record. Consequently, the optimal balance of performance characteristics depends on the application.

Error correction

Using a hash function to detect errors in transmission is straightforward. The hash function is computed for the data at the sender, and the value of this hash is sent with the data. The hash function is performed again at the receiving end, and if the hash values do not match, an error has occurred at some point during the transmission. This is called a redundancy check.

For error correction, a distribution of likely perturbations is assumed at least approximately. Perturbations to a string are then classified into large (improbable) and small (probable) errors. The second criterion is then restated so that if we are given H(x) and x+s, then we can compute x efficiently if s is small. Such hash functions are known as error correction codes. Important sub-class of these correction codes are cyclic redundancy checks and Reed-Solomon codes.

Audio identification

For audio identification such as finding out whether an MP3 file matches one of a list of known items, one could use a conventional hash function such as MD5, but this would be very sensitive to highly likely perturbations such as time-shifting, CD read errors, different compression algorithms or implementations or changes in volume. Using something like MD5 is useful as a first pass to find exactly identical files, but another more advanced algorithm is required to find all items that would nonetheless be interpreted as identical to a human listener. Though they are not common, hashing algorithms do exist that are robust to these minor differences. Most of the algorithms available are not extremely robust, but some are so robust that they can identify music played on loud-speakers in a noisy room. One example of this in practical use is the service Shazam. Customers call a number and place their telephone near a speaker. The service then analyses the music, and compares it to known hash values in its database. The name of the music is sent to the user. An open source alternative to this service is MusicBrainz which creates a fingerprint for an audio file and matches it to its online community driven database.
Universal hashing

(From Wikipedia, the free encyclopedia)

Universal hashing is a randomized algorithm for selecting a hash function $F$ with the following property: for any two distinct inputs $x$ and $y$, the probability that $F(x)=F(y)$ (i.e., that there is a hash collision between $x$ and $y$) is the same as if $F$ was a random function. Thus, if $F$ has function values in a range of size $r$, the probability of any particular hash collision should be $1/r$. There are universal hashing methods that give a function $F$ that can be evaluated in a handful of computer instructions.

Introduction

Hashing has been used to associate with an input, usually a string, a small value that originally was used as an index to look up something about that input in a table. Since then hashing has found other uses. For example, two inputs might be compared by checking to see if their hash values are the same. Thus, we can see that hash functions are many-to-one mappings. The use of the word "hash" is mnemonic because the intent of a hash function is to take as many of the inputs usually encountered and assign different values to them, by scrambling them or making a hash of the inputs, using the meaning of hash from domains such as cooking. If for any given input there are too many collisions that is viewed as unfortunate.

Universal Hashing

Because a hash function is a many-to-one mapping, there must exist some set of elements that will collide under the hash function. One wants to design the hash function such that for the input sets, it is unlikely that elements collide. Proving in a mathematical sense that you are unlikely to encounter a particular set of inputs would appear to be an impossible task.

Randomized algorithms present a way of proving that you are unlikely to encounter a bad set of inputs. We can construct a Universal Class of hash functions with the property that for any given set of inputs they will scatter the inputs among the range of the function well -- essentially as well as choosing random values for those inputs. Thus, simply choosing a random function from the class allows a proof that the probabilistic expectation for any set of inputs is that they will be distributed randomly.

In fact, we are in many cases interested in only pairwise collisions. That is to say, the odds that any two inputs $x$ and $y$ collide will be approximately the same as the reciprocal of the size of the range. It might be that for any given universal class of hash functions there exist $x$, $y$ and $z$ such that if $x$ and $y$ collide then so does $z$. While some work
has been done on the set issue, universal hashing only makes statements about pairwise collisions.

**Example**

A simple universal class of hash function is all functions \( h \) of the form \( h(x) = f(g(x)) \), where \( g(x) = ax + b \pmod{p} \) with \( p \) being a prime guaranteed larger than any possible input and each combination of \( a \) and \( b \) forming a different function in the class. \( f \) then becomes a mapping function to map elements from a domain which is 0 to \( p \) to a range of say 0 to \( n-1 \). \( f \) then can simply be taking the result of \( g \pmod{n} \). There is only one \( f \) for all the functions in this class. To see why this class is universal, observe that for any two inputs and any two outputs, there are approximately \( p/n \) elements that can map to any output and for any of pair of those \( p/n \) elements you can solve the simultaneous equations in the field \( \pmod{p} \), so for any pair of inputs there is a unique pair of \( a \) and \( b \) that will take it to those elements.

Universal hashing has numerous uses in computer science, for example in cryptography and in implementations of hash tables. Since the function is randomly chosen, an adversary hoping to create many hash collisions is unlikely to succeed.

Universal hashing has been generalized in many ways, most notably to the notion of k-wise independent hash functions, where the function is required to act like a random function on any set of \( k \) inputs.

**Perfect hashing**

(From Wikipedia, the free encyclopedia)

A Perfect hash function of a set \( S \) is a hash function which maps different keys (elements) in \( S \) to different numbers. A perfect hash function with values in a range of size some constant times the number of elements in \( S \) can be used for efficient lookup operations, by placing the keys in a hash table according to the values of the perfect hash function.

A perfect hash function for a specific set \( S \) that can be evaluated in constant time, and with values in a small range, can be found by a randomized algorithm in a number of operations that is proportional to the size of \( S \). The minimal size of the description of a perfect hash function depends on the range of its function values: The smaller the range, the more space is required. Any perfect hash functions suitable for use with a hash table require at least a number of bits that is proportional to the size of \( S \). Many common implementations require a number of bits that is proportional to \( n \log(n) \), where \( n \) is the size of \( S \). This means that the space for storing the perfect hash function can be comparable to the space for storing the set.
Using a perfect hash function is best in situations where there is a large set which is not updated frequently, and many lookups into it. Efficient solutions to performing updates are known as dynamic perfect hashing, but these methods are relatively complicated to implement. A simple alternative to perfect hashing, which also allows dynamic updates, is cuckoo hashing.

A minimal perfect hash function is a perfect hash function that maps n keys to n consecutive integers -- usually [0..n-1] or [1..n]. A more formal way of expressing this is: Let j and k be elements of some set K. F is a minimal perfect hash function if F(j) = F(k) implies j = k and there exists an integer a such that the range of F is a..a+|K|-1.

A minimal perfect hash function F is order-preserving if for any keys j and k, j < k implies F(j) < F(k).
Exercises


Linked lists

Exercises 2.1

Implement a stack using a singly linked list L. The operations PUSH and POP should still take O(1) time.

Exercises 2.2

Implement a queue by a singly linked list L. The operations ENQUEUE and DEQUEUE should still take O(1) time.

Exercises 2.3

The dynamic-set operation UNION takes two disjoint sets S1 and S2 as input, and it returns a set S = S1 _ S2 consisting of all the elements of S1 and S2. The sets S1 and S2 are usually destroyed by the operation. Show how to support UNION in O(1) time using a suitable list data structure.

Exercises 2.4

Explain how to implement doubly linked lists using only one pointer value np[x] per item instead of the usual two (next and prev). Assume that all pointer values can be interpreted as k-bit integers, and define np[x] to be np[x] = next[x] XOR prev[x], the k-bit "exclusive-or" of next[x] and prev[x]. (The value NIL is represented by 0.) Be sure to describe what information is needed to access the head of the list. Show how to implement the SEARCH, INSERT, and DELETE operations on such a list. Also show how to reverse such a list in O(1) time.

Stack and Queue

Exercises 3.1
Using Figure above as a model, illustrate the result of each operation in the sequence
PUSH(S, 4), PUSH(S, 1), PUSH(S, 3), POP(S), PUSH(S, 8), and POP(S) on an initially empty stack S stored in array S[1 _ 6].

Exercises 3.2

Explain how to implement two stacks in one array A[1 _ n] in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in O(1) time.

Exercises 3.3

Using Figure above as a model, illustrate the result of each operation in the sequence
ENQUEUE(Q, 4), ENQUEUE(Q, 1), ENQUEUE(Q, 3), DEQUEUE(Q), ENQUEUE(Q, 8), and DEQUEUE(Q) on an initially empty queue Q stored in array Q[1 _ 6].

Exercises 3.4

Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

Exercises 3.5

Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a deque (double-ended queue)
allows insertion and deletion at both ends. Write four O(1)-time procedures to insert elements into and delete elements from both ends of a deque constructed from an array.

Exercises 3.6

Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

Exercises 3.7.

Show how to implement a stack using two queues. Analyze the running time of the stack operations.

**Designing algorithms**

Exercises 4.1.

Using Figure below as a model, illustrate the operation of merge sort on the array \( A = 3, 41, 52, 26, 38, 57, 9, 49 \).

Exercises 4.2.

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array \( L \) or \( R \) has had all its elements copied back to \( A \) and then copying the remainder of the other array back into \( A \).

Exercises 4.3
Insertion sort can be expressed as a recursive procedure as follows. In order to sort \(A[1\_n]\), we recursively sort \(A[1\_n-1]\) and then insert \(A[n]\) into the sorted array \(A[1\_n-1]\). Write a recurrence for the running time of this recursive version of insertion sort.

**Binary Search Trees**

Exercises 5.1

For the set of keys \(\{1, 4, 5, 10, 16, 17, 21\}\), draw binary search trees of height 2, 3, 4, 5, and 6.

Exercises 5.2

What is the difference between the binary-search-tree property and the min-heap property? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in \(O(n)\) time? Explain how or why not.

Exercises 5.3

Give a nonrecursive algorithm that performs an inorder tree walk. (Hint: There is an easy solution that uses a stack as an auxiliary data structure and a more complicated but elegant solution that uses no stack but assumes that two pointers can be tested for equality.)

Exercises 5.4

Give recursive algorithms that perform preorder and postorder tree walks in \(\Theta(n)\) time on a tree of \(n\) nodes.

Exercises 5.5

Argue that since sorting \(n\) elements takes \(\Omega(n \lg n)\) time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of \(n\) elements takes \(\Omega(n \lg n)\) time in the worst case.

Exercises 5.6

Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined?

Exercises 5.7

Write recursive versions of the TREE-MINIMUM and TREE-MAXIMUM procedures.

Exercises 5.8

Write the TREE-PREDECESSOR procedure.

Exercises 5.9

Professor Bunyan thinks he has discovered a remarkable property of binary search trees.

Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys a ∈ A, b ∈ B, and c ∈ C must satisfy a ≤ b ≤ c. Give a smallest possible counterexample to the professor's claim.

Exercises 5.10

Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

Exercises 5.11

Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

Exercises 5.12

An inorder tree walk of an n-node binary search tree can be implemented by finding the
minimum element in the tree with TREE-MINIMUM and then making n-1 calls to TREESUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

Exercises 5.13

Prove that no matter what node we start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

Exercises 5.14

Let $T$ be a binary search tree whose keys are distinct, let $x$ be a leaf node, and let $y$ be its parent. Show that key[$y$] is either the smallest key in $T$ larger than key[$x$] or the largest key in $T$ smaller than key[$x$].

Exercises 5.15

Give a recursive version of the TREE-INSERT procedure.

Exercises 5.16

Suppose that a binary search tree is constructed by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree.

Exercises 5.17

We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

Exercises 5.18

Suppose that another data structure contains a pointer to a node $y$ in a binary search tree, and suppose that $y$'s predecessor $z$ is deleted from the tree by the procedure TREE-DELETE. What problem can arise? How can TREE-DELETE be rewritten to solve this problem?

Exercises 5.19

Is the operation of deletion "commutative" in the sense that deleting $x$ and then $y$ from a
binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

Exercises 5.20

When node z in TREE-DELETE has two children, we could splice out its predecessor rather than its successor. Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be changed to implement such a fair strategy?

**Sorting**

Exercises 6.1

What are the minimum and maximum numbers of elements in a heap of height h?

Exercises 6.2

Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

Exercises 6.3

Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

Exercises 6.4

Is an array that is in sorted order a min-heap?

Exercises 6.5

Is the sequence _23, 17, 14, 6, 13, 10, 1, 5, 7, 12_ a max-heap?

Exercises 6.6
Using Figure above as a model, illustrate the operation of MAX-HEAPIFY(A, 3) on the array A = \_27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\_.

Exercises 6.7

Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MINHEAPIFY( A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

Exercises 6.8

What is the effect of calling MAX-HEAPIFY(A, i) when the element A[i] is larger than its children?

Exercises 6.9

What is the effect of calling MAX-HEAPIFY(A, i) for i > heap-size[A]/2?

Exercises 6.10

The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

Exercises 6.11

Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is \( \Omega(lg n) \).
(Hint: For a heap with \( n \) nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a path from the root down to a leaf.)

Exercises 6.12

Using Figure above as a model, illustrate the operation of HEAPSORT on the array \( A = 5, 13, 2, 25, 7, 17, 20, 8, 4 \).

Exercises 6.13

What is the running time of heapsort on an array \( A \) of length \( n \) that is already sorted in increasing order? What about decreasing order?

Exercises 6.14

Show that the worst-case running time of heapsort is \( \Omega(n \lg n) \).

Exercises 6.15

Show that when all elements are distinct, the best-case running time of heapsort is \( \Omega(n \lg n) \).

Exercises 6.16
Using Figure above as a model, illustrate the operation of PARTITION on the array A = _13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21_.

Exercises 6.17

What value of q does PARTITION return when all elements in the array A[p _ r] have the

same value? Modify PARTITION so that q = (p+r)/2 when all elements in the array A[p ? r] have the same value.

Exercises 6.18

Give a brief argument that the running time of PARTITION on a subarray of size n is Θ(n).

Exercises 6.19

How would you modify QUICKSORT to sort into nonincreasing order?

Graphs

Exercises 7.1
Attendees of a faculty party shake hands to greet each other, and each professor remembers how many times he or she shook hands. At the end of the party, the department head adds up the number of times that each professor shook hands. Show that the result is even by proving the handshaking lemma: if \( G = (V, E) \) is an undirected graph, then

**Exercises 7.2**

Show that if a directed or undirected graph contains a path between two vertices \( u \) and \( v \), then it contains a simple path between \( u \) and \( v \). Show that if a directed graph contains a cycle, then it contains a simple cycle.

**Exercises 7.3**

Show that any connected, undirected graph \( G = (V, E) \) satisfies \(|E| \geq |V| - 1\).

**Exercises 7.4**

Verify that in an undirected graph, the "is reachable from" relation is an equivalence relation on the vertices of the graph. Which of the three properties of an equivalence relation hold in general for the "is reachable from" relation on the vertices of a directed graph?

**Exercises 7.5**

Show that a hypergraph can be represented by a bipartite graph if we let incidence in the hypergraph correspond to adjacency in the bipartite graph. (Hint: Let one set of vertices in the bipartite graph correspond to vertices of the hypergraph, and let the other set of vertices of the bipartite graph correspond to hyperedges.)

**Hashing**

**Exercises 8.1**

Suppose that a dynamic set \( S \) is represented by a direct-address table \( T \) of length \( m \). Describe a procedure that finds the maximum element of \( S \). What is the worst-case performance of your procedure?

**Exercises 8.2**
A bit vector is simply an array of bits (0's and 1's). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to Represent a Dynamic Set of Distinct Elements with no Satellite Data. Dictionary Operations Should Run in O(1) Time.

Exercises 8.3

Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in O(1) time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

Exercises 8.4

We wish to implement a dictionary by using direct addressing on a huge array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use O(1) space; the operations SEARCH, INSERT, and DELETE should take O(1) time each; and the initialization of the data structure should take O(1) time. (Hint: Use an additional stack, whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

Exercises 8.5

Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of \{\{k, l\} : k \neq l \text{ and } h(k) = h(l)\}? 

Exercises 8.6

Demonstrate the insertion of the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be \(h(k) = k \mod 9\).

Exercises 8.7

Professor Marley hypothesizes that substantial performance gains can be obtained if we modify the chaining scheme so that each list is kept in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

Exercises 8.8
Suggest how storage for elements can be allocated and deallocated within the hash table itself by linking all unused slots into a free list. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

Exercises 8.9

Show that if $|U| > nm$, there is a subset of $U$ of size $n$ consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

Exercises 8.10

Suppose we wish to search a linked list of length $n$, where each element contains a key $k$ along with a hash value $h(k)$. Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?

Exercises 8.11

Suppose that a string of $r$ characters is hashed into $m$ slots by treating it as a radix-128 number and then using the division method. The number $m$ is easily represented as a 32-bit computer word, but the string of $r$ characters, treated as a radix-128 number, takes many words. How can we apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

Exercises 8.12

Consider a version of the division method in which $h(k) = k \mod m$, where $m = 2^p - 1$ and $k$ is a character string interpreted in radix $2^p$. Show that if string $x$ can be derived from string $y$ by permuting its characters, then $x$ and $y$ hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.
Assignment problems

Assignment problem 1 - Depth First Search and The N-Queens Problem

Write recursive code to implement the N-Queen problem by using depth-first search. Make your code general enough to handle any arbitrary N. For testing purposes you might want to try N = 4, which is the smallest non-trivial problem.

Now use N = 8. Print out all the possible solutions. How many are there?

Modify your code to stop as soon as one solution is found. Run with N = 9, 10, 11, etc...

Assignment problem 2 - Greedy Search and The N-Queens Problem

Write code to implement the N-Queen problem by using greedy search. Make your code general enough to handle any arbitrary N. The code should terminate as soon as one solution is found.

For testing purposes you might want to try N = 4, which is the smallest non-trivial problem. Run with ever increasing N.

Assignment problem 3 - Finding a maximum weight matching in a weighted bipartite graph

(From Wikipedia, the free encyclopedia)

There are a number of agents and a number of tasks. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task in such a way that the total cost of the assignment is minimized.

If the numbers of agents and tasks are equal and the total cost of the assignment for all tasks is equal to the sum of the costs for each agent (or the sum of the costs for each task, which is the same thing in this case), then the problem is called the Linear assignment problem. Commonly, when speaking of the Assignment problem without any additional qualification, then the Linear assignment problem is meant.

Formal mathematical definition

The formal definition of the assignment problem (or linear assignment problem) is
Given two sets, A and T, of equal size, together with a weight function \( C : A \times T \to \mathbb{R} \). Find a bijection \( f : A \to T \) such that the cost function:

\[
\sum_{a \in A} C(a, f(a))
\]

is minimized.

Usually the weight function is viewed as a square real-valued matrix \( C \), so that the cost function is written down as:

\[
\sum_{a \in A} C_{a,f(a)}
\]

The problem is "linear" because the cost function to be optimized as well as all the constraints contain only linear terms.

The problem can be expressed as a standard linear program with the objective function

\[
\sum \sum C(i, j) x_{ij}
\]

subject to the constraints

\[
\sum_{j \in A} x_{ij} = 1 \quad \text{for } i \in A ,
\]

\[
\sum_{i \in A} x_{ij} = 1 \quad \text{for } j \in A ,
\]

\[
x_{ij} \geq 0 \text{ for } i, j \in A .
\]

The variable \( x_{ij} \) represents the assignment of agent \( i \) to task \( j \), taking value 1 if the assignment is done and 0 otherwise. This formulation allows also fractional variable values, but there is always an optimal solution where the variables take integer values. This is because the constraint matrix is totally unimodular. The first constraint requires that every agent is assigned to exactly one task, and the second constraint requires that every task is assigned exactly one agent.

Suppose that a taxi firm has three taxis (the agents) available, and three customers (the tasks) wishing to be picked up as soon as possible. The firm prides itself on speedy
pickups, so for each taxi the "cost" of picking up a particular customer will depend on
the time taken for the taxi to reach the pickup point. The solution to the assignment
problem will be whichever combination of taxis and customers results in the least total
cost.

**Assignment problem 4 - Stable marriage problem**

(From Wikipedia, the free encyclopedia)

Given n men and n women, where each person has ranked all members of the opposite
sex with a unique number between 1 and n in order of preference, marry the men and
women off such that there are no two people of opposite sex who would both rather have
each other than their current partners. If there are no such people, all the marriages are
"stable".

The Gale-Shapley algorithm involves a number of "rounds" (or "iterations") where
each unengaged man "proposes" to the most-preferred woman to whom he has not
yet proposed, and she accepts or rejects him based on whether she is already engaged
to someone she prefers. If she is unengaged, or engaged to a man lower down her
preference list than her new suitor, she accepts the proposal (and in the latter case, the
other man becomes unengaged again). Note that only women can switch partners to
increase their happiness.

Algorithm

function stableMatching {

Initialize all \( m \in M \) and \( w \in W \) to free

while \( \exists \) free man \( m \) who still has a woman \( w \) to propose to {

\( w = m \)'s highest ranked such woman

if \( w \) is free

\( (m, w) \) become engaged

else some pair \( (m', w) \) already exists

if \( w \) prefers \( m \) to \( m' \)

\( (m, w) \) become engaged

\( m' \) becomes free

}
else

(m', w) remain engaged

}  
}

Using this algorithm to guarantee that:

- Everyone gets married: Once a woman becomes engaged, she is always engaged to someone. So, at the end, there cannot be a man and a woman both unengaged, as he must have proposed to her at some point (since a man will eventually propose to everyone, if necessary) and, being unengaged, she would have to have said yes.

- The marriages are stable: Let Alice be a woman and Bob be a man. They are each paired/partnered/married, but not to each other. Upon completion of the algorithm, it is not possible for both Alice and Bob to prefer each other over their current partners. If Bob prefers Alice to his current partner, he must have proposed to Alice before he proposed to his current partner. If Alice accepted his proposal, yet is not married to him at the end, she must have dumped him for someone she likes more, and therefore doesn't like Bob more than her current partner. If Alice rejected his proposal, she was already with someone she liked more than Bob.
Contribution

Collection: Data Structures and Algorithms
Edited by: vuquangdung
URL: http://voer.edu.vn/c/8bbc580f
License: http://creativecommons.org/licenses/by/3.0/

Module: Introduction to Algorithms
Authors: Nguyễn Việt Hà, truongninhthuan, vuquangdung
URL: http://www.voer.edu.vn/m/2260f3d4
License: http://creativecommons.org/licenses/by/3.0/

Module: Stack and Queue
Authors: Nguyễn Việt Hà, truongninhthuan, vuquangdung
URL: http://www.voer.edu.vn/m/f14bb7c7
License: http://creativecommons.org/licenses/by/3.0/

Module: Recursion
Authors: Nguyễn Việt Hà, truongninhthuan, vuquangdung
URL: http://www.voer.edu.vn/m/a7c2738f
License: http://creativecommons.org/licenses/by/3.0/

Module: Binary search trees
Authors: Nguyễn Việt Hà, truongninhthuan, vuquangdung
URL: http://www.voer.edu.vn/m/99ff9ed8
License: http://creativecommons.org/licenses/by/3.0/

Module: Sorting
Authors: Nguyễn Việt Hà, truongninhthuan, vuquangdung
URL: http://www.voer.edu.vn/m/80679c0f
License: http://creativecommons.org/licenses/by/3.0/

Module: Graphs
Authors: Nguyễn Việt Hà, truongninhthuan, vuquangdung
URL: http://www.voer.edu.vn/m/36385faa
License: http://creativecommons.org/licenses/by/3.0/
Module: Hashing
Authors: Nguyễn Việt Hà, truongninhthuan, vuquangdung
URL: http://www.voer.edu.vn/m/7a1a3594
License: http://creativecommons.org/licenses/by/3.0/

Module: Exercises
Authors: Nguyễn Việt Hà, truongninhthuan, vuquangdung
URL: http://www.voer.edu.vn/m/2c45fd23
License: http://creativecommons.org/licenses/by/3.0/

Module: Assignment problems
Authors: Nguyễn Việt Hà, truongninhthuan, vuquangdung
URL: http://www.voer.edu.vn/m/10fb0e72
License: http://creativecommons.org/licenses/by/3.0/
The Vietnam Open Educational Resources

Hanoi Spring is an advanced technology platform designed to allow the public to share teaching and learning materials and curriculum development. It is based on the concept of open educational resources (OER). This revolutionary concept of knowledge sharing was pioneered by MIT University and Rice University in the U.S. more than a decade ago. Since then, there has been a growing OER movement globally, and UNESCO has embraced OER as an official program in many countries around the world.

The Vietnam Foundation and its technology team in Hanoi play a leading role in the development, technology and framework for global OER. Vietnamese OER experts have been invited to Paris to advise UNESCO on its OER policy recommendations, and they have traveled to other countries to help educational institutions implement OER programs.

Under the management of the Vietnam Foundation, the Vietnam Open Educational Resources (VOER) has become a major portal for students and teachers inside and outside of Vietnam. Every day, thousands visitors access VOER (www.voer.edu.vn) to research and download materials. There are more than 21,000 knowledge modules from nearly 4,000 contributing authors; among them are well-known scientists living inside Vietnam such as Dr. Nguyen Van Hieu (Physics), Dr. Nguyen Lan Dung (Biology), and Dr. Pham Phu (Mathematics), as well as Vietnamese scholars living abroad such as Dr. Minh Do (Electrical Engineering); Dr. Thanh Truong (Chemistry) and Dr. John Vu (Information Technology).

Dr. Ray Gamble of the U.S. National Academies, and also the founding chairman of the Vietnam Foundation, observes “The participation and support of the best scholastic minds of Vietnam is a testament to the success of VOER. Now we can offer Hanoi Spring to the global OER community to further advance educational opportunity. It is a gift from the Vietnamese people to the world.”

Hanoi Spring was designed and developed over the last two years by a team of Vietnamese software experts and volunteers located in Hanoi, working with the support and direction of the Vietnam Foundation. The platform is robust; it is easy to install, manage and use. It can handle thousands of concurrent users and supports different national languages.