1. Introduction to Algorithms

1.1 Algorithm Definition

An algorithm is defined as a finite sequence of explicit instructions that, when provided with a set of input values, produces an output and then terminates. An algorithm should have the following properties:

- There must be no ambiguity in any instruction
- There should not be any uncertainty about which instruction should be executed next
- The execution of the algorithm should conclude after a finite number of steps
- Algorithms must be general enough to deal with any circumstances

The range of inputs for which an algorithm works has to be specified carefully. The same algorithm can be represented in several different ways (Using simple English sentences or pseudo code). There may exist several algorithms for solving the same problem. Different algorithms for solving the same problem may differ in their speed (Figure 1.1).

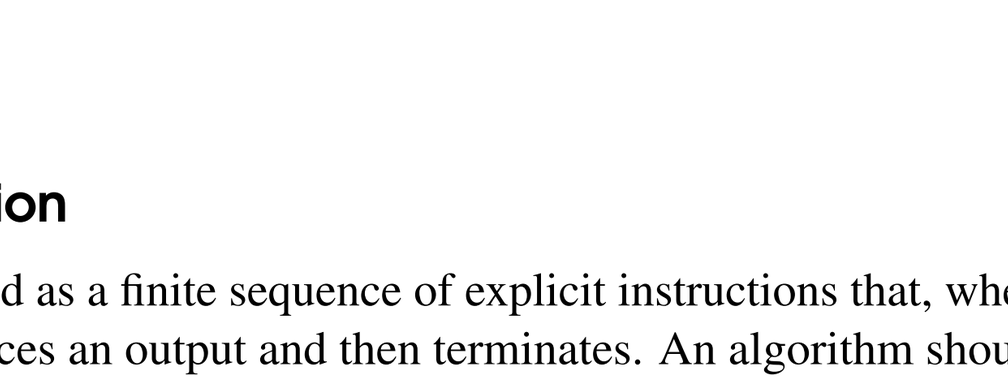


Figure 1.1: Algorithm

1.1.1 Algorithm Example - Finding GCD

The greatest common divisor of two nonnegative integers `m` and `n` (One of the numbers should be non-zero), denoted by `gcd(m,n)`, is defined as the largest integer that divides both `m` and `n`.

Euclid's Algorithm

According to Euclid's algorithm,

$$gcd(m,n) = gcd(n, m \ mod \ n)$$

$$gcd(m,0) = m$$

Complete algorithm is given in Listing 1.1.

```
while n ≠ 0 do
  r = m mod n
  m = n
  n = r
return m
```

Listing 1.1: Euclid's Algorithm

Consecutive Integer Checking Algorithm

Another method to find the gcd is given in Listing 1.2.

```
t = min(m,n)
while m mod t ≠ 0 or n mod t ≠ 0
  t = t - 1
return t
```

Listing 1.2: Consecutive integer checking algorithm

Unlike Euclid's algorithm, this algorithm does not work correctly when one of its input numbers is zero.

By finding prime factors

```
Find the prime factors of m.
Find the prime factors of n.
Identify all the common factors in the two prime expansions found in
    Step 1 and Step 2. (If p is a common factor occurring p_m and p_n
    times in m and n, respectively, it should be repeated min{p_m, p_n}
    times.)
Compute the product of all the common factors and return it as the gcd
    of m and n.
```

Listing 1.3: From prime factors

For example gcd(24,16) can be found as follows

$$\begin{aligned} 24 &= 2.2.2.3 \\ 16 &= 2.2.2.2 \\ \text{gcd}(24,16) &= 2.2.2 = 8 \end{aligned}$$

In this algorithm, the prime factorization steps are not defined unambiguously. Hence this may not be treated as a legitimate algorithm.

Consecutive primes not exceeding any given integer n > 1, can be found using an algorithm known as **sieve of Eratosthenes**.

```
for p = 2 to n do
  A[p] = p
for p = 2 to √n
  if A[p] ≠ 0
    j = p × p
    while j ≤ n do
      A[j] = 0 //mark element as eliminated
      j = j + p
```

```
//copy the remaining elements of A to array L of the primes
i = 0
for p = 2 to n do
  if A[p] ≠ 0
    L[i] = A[p]
    i = i+1
return L
```

Listing 1.4: Sieve of Eratosthenes

1.2 Fundamentals of Algorithmic Problem Solving

The sequence of steps in design and analyzing an algorithm includes the following.

1.2.1 Understanding the Problem

- Read the problem description carefully
- Ask questions for any doubts about the problem
- Do small examples by hand
- think about special cases
- Ask questions again if needed.

An input to an algorithm specifies an **instance** of the problem the algorithm solves. The set of instances the algorithm needs to handle must be specified.

1.2.2 Ascertaining the Capabilities of the Computational Device

Computers today are extremely fast. Hence, in most situations one does not have to worry about a computer being too slow for the task. For problems that are very complex, or have to process huge volumes of data, or deal with applications where the time is critical, it is better to be aware of the speed and memory available on a particular computer system.

1.2.3 Choosing between Exact and Approximate Problem Solving

Approximate algorithms may suitable for following scenarios

- Problems that simply cannot be solved exactly for most of their instances (e.g. extracting square roots)
- Available algorithms for solving a problem exactly are unacceptably slow because of the problem's intrinsic complexity
- An approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

1.2.4 Algorithm Design Techniques

An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

- Algorithm design techniques provide guidance for designing algorithms for new problems.
- Algorithm design techniques make it possible to classify algorithms according to an underlying design idea.

1.2.5 Designing an Algorithm and Data Structures

Algorithm design techniques offer helpful strategies for solving algorithmic problems, but designing an algorithm for a specific problem can still be difficult. Certain design techniques may not be suitable for the problem at hand. In some cases, it's necessary to combine multiple techniques. Choosing appropriate data structures for the operations performed by the algorithm is also very important since it may affect the speed of the algorithm.

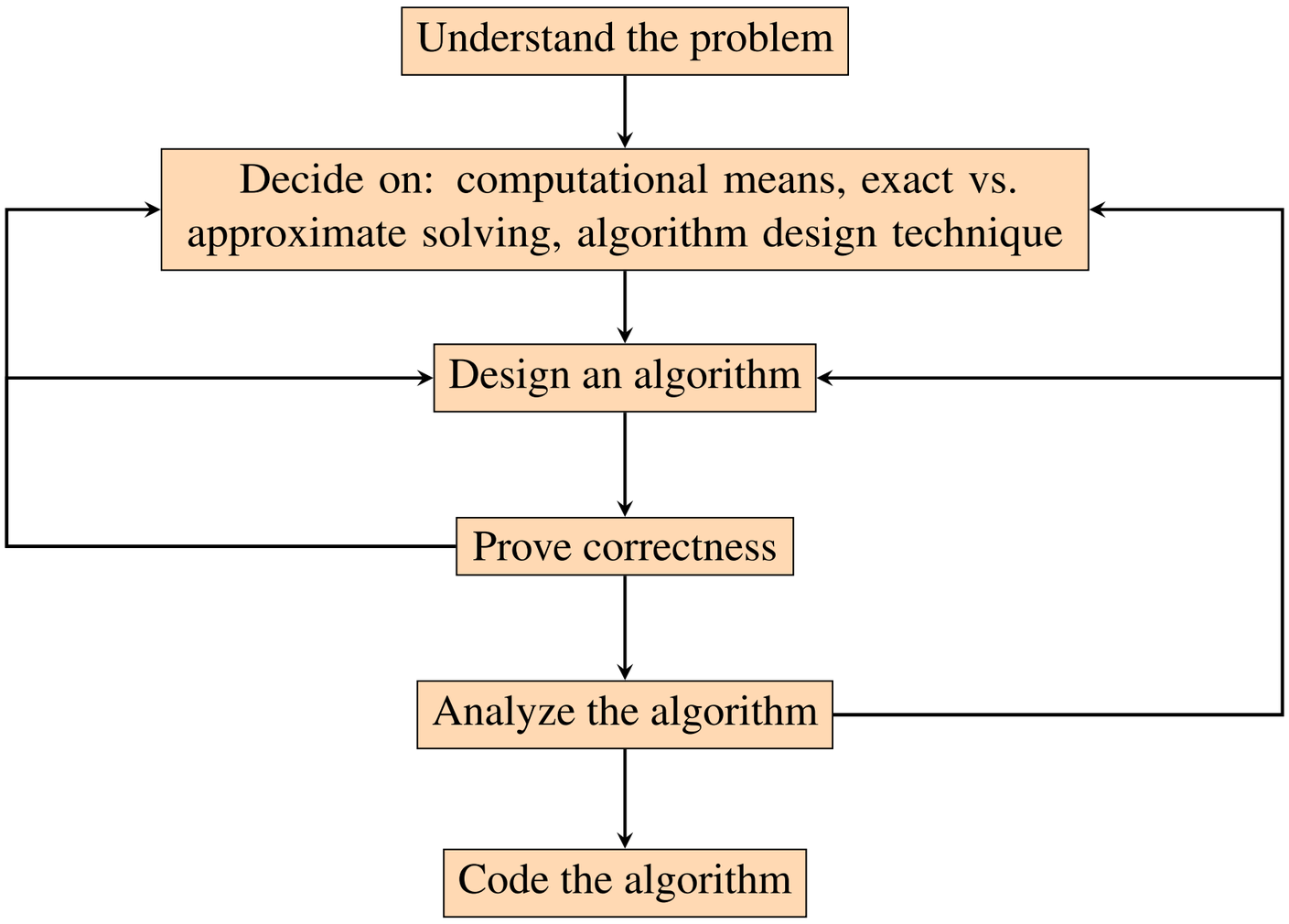


Figure 1.2: Algorithm design and analysis process

1.2.6 Methods of Specifying an Algorithm

Once an algorithm is designed it needs to be specified in some form. Two most common methods for specifying an algorithm are

- Writing sequence of steps in English
- Pseudocode, which is a mixture of a natural language and programming language-like constructs.

1.2.7 Proving an Algorithm's Correctness

Once an algorithm has been specified, it has to be ensured that the algorithm yields a required result for every legitimate input in a finite amount of time. A common technique for proving correctness is to use mathematical induction.

The concept of correctness in approximation algorithms is more complex compared to exact algorithms. In the case of an approximation algorithm, the objective is typically to demonstrate that the algorithm's error remains within a predetermined limit.

1.2.8 Analyzing an Algorithm

Efficiency

There are two kinds of algorithm efficiency:

- **time efficiency**, indicating how fast the algorithm runs
- **space efficiency**, indicating how much extra memory it uses

Simplicity

Simpler algorithms are easier to understand and easier to program. Consequently, the resulting programs usually contain fewer bugs. But simpler algorithm need not always be efficient.

Generality

To determine if two numbers are relatively prime (means they only have 1 as a common divisor), it is simpler to create an algorithm for a broader problem of finding the gcd of two numbers. The initial problem can then be solved by checking if the gcd of the two numbers is equal to 1 or not.

1.2.9 Coding an Algorithm

Having a functioning program opens up the possibility to empirically analyze the underlying algorithm. This analysis involves measuring the time it takes for the program to run with different inputs and then examining the results obtained.

1.3 Important Problem Types

1.3.1 Sorting

The sorting problem is to rearrange the items of a given list in nondecreasing order. Most common requirements are to sort lists of numbers, characters from an alphabet, character strings, records etc. In the case of records, a piece of information, known as a **key**, is to be chosen to guide sorting. Sorting makes many questions about the list easier to answer such as searching for an item in a list.

There are different types of sorting algorithms. Some are simple but slower, while others are faster but more complex. Certain algorithms perform well with randomly ordered inputs, while others excel with almost-sorted lists. Some algorithms are specifically designed for sorting small lists in fast memory, while others can be modified to sort large files stored on a disk.

A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input. Another notable feature of a sorting algorithm is the amount of extra memory the algorithm requires. An algorithm is said to be **in-place** if it does not require extra memory, except, possibly, for a few memory units.

1.3.2 Searching

The searching problem deals with finding a given value, called a **search key**, in a given set. Some searching algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays.

In certain applications, where the data changes frequently compared to the number of searches, searching needs to be considered alongside two other operations: adding an item to the data set and removing an item from it. In these situations, it is important to select data structures and algorithms that can meet the requirements of each operation in a balanced way.

1.3.3 String Processing

A string is a sequence of characters from an alphabet. Researchers have given special attention to a specific problem known as **string matching**, which involves searching for a given word in a text.

1.3.4 Graph Problems

A graph is a collection of points called vertices connected by line segments called edges. Graphs are used for modeling various applications like transportation, communication, social and economic networks, project scheduling, and games.

Basic graph algorithms include

- Graph-traversal algorithms - determine how to reach all the points in a network.
- Shortest-path algorithms- find the best route between two cities.
- Topological sorting (for graphs with directed edges) - to check if a set of courses and their prerequisites is consistent or self-contradictory.

Some graph problems are computationally challenging, such as the traveling salesman problem (TSP). TSP involves finding the shortest tour through multiple cities, visiting each city exactly once. Another difficult problem is graph coloring, which aims to assign the fewest number of colors to graph vertices so that no adjacent vertices have the same color. Graph coloring is applicable in event scheduling, where an optimal schedule can be achieved by solving the graph-coloring problem.

1.3.5 Combinatorial Problems

The traveling salesman problem and the graph-coloring problem are combinatorial problems. Combinatorial problems involve finding a combinatorial object, like a permutation, combination, or subset, that meets specific constraints. These problems may also require the combinatorial object to have additional properties, such as a maximum value or minimum cost.

Combinatorial problems are generally considered the most challenging in computing, both theoretically and practically. This difficulty arises from several factors. Firstly, the number of combinatorial objects grows rapidly as the problem size increases, even for moderate-sized instances. Secondly, there are no known algorithms that can solve most combinatorial problems within a reasonable timeframe. Many computer scientists believe that such algorithms may not even exist.

1.3.6 Geometric Problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons. Two classic problems in computational geometry are

- **The closest-pair problem** - Finding the pair of points that are closest to each other among a given set of points.
- **The convex-hull problem** - Finding the smallest convex polygon that encompasses all the points in a given set.

1.3.7 Numerical Problems

Numerical problems are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on.

Most mathematical problems in computing can only be solved approximately. This is due to the inherent difficulty in manipulating real numbers, which can only be represented approximately in a computer. Additionally, when performing numerous arithmetic operations on approximately represented numbers, round-off errors can accumulate and significantly distort the output of an algorithm that initially seemed correct.

1.4 Fundamental Data Structures

A data structure can be defined as a particular scheme of organizing related data items.

1.4.1 Linear Data Structures

The two most important elementary data structures are the array and the linked list.

Arrays

A (one-dimensional) array is a sequence of `n` items of the same data type that are stored contiguously in computer memory. Elements inside an array can be accessed using an index. Each and every element of an array can be accessed in the same constant amount of time, regardless of its location within the array. Arrays are used for implementing a variety of other data structures such as **strings**. The operations performed on strings include computing the length of a string, comparing two strings to determine their lexicographic order (alphabetical order), and concatenating two strings to form a new string.

Linked List

A linked list is a sequence of zero or more elements called **nodes**. Each node contains two kinds of information: some data and one or more links to other nodes of the linked list. In a singly linked list, each node except the last one contains a single pointer to the next element.

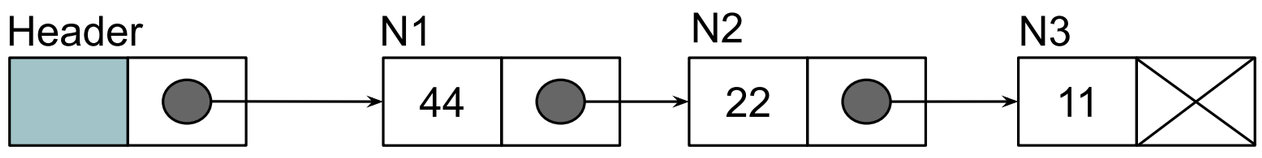


Figure 1.3: Single Linked List

To access a specific node in a linked list, the list must be traversed starting from the first node and following the pointers until the desired node is reached. Therefore, the time required to access an element in a singly linked list depends on its position within the list.

Linked lists do not require any prior reservation of computer memory, meaning that memory can be dynamically allocated as needed. This is in contrast to other data structures like arrays, which require a fixed amount of memory upfront.

Additionally, insertions and deletions can be efficiently performed in a linked list by adjusting a few appropriate pointers. This flexibility is one of the advantages of linked lists compared to other data structures.

In a doubly linked list, every node except the first and the last, contains pointers to both its successor and its predecessor.

Stack

A **stack** is a list in which insertions and deletions can only be done at the end. This end is called the **top**. The insertion operation in a stack is called **push**, and the deletion operation is called **pop**. Since the last pushed element is the first one to be popped, a stack is also known as a **LIFO** (Last In First Out) list. Stacks are used for implementing recursive algorithms,, evaluating postfix expression etc.

Queue

A **queue** is a list in which insertions takes place at one end called **rear** and deletions takes place at the other end called **front**. The insertion operation in a queue is called **enqueue**, and the deletion operation is called **dequeue**. Since the first enqueued element is the first one to be dequeued, a queue is also known as a **FIFO** (First In First Out) list. Queues are used for implementing several graph algorithms.

1.4.2 Graphs

A graph G consists of 2 sets:

1. A set V, the set of all vertices (nodes).
2. A set E, the set of all edges (arcs). E is a set of pairs of elements from V.

For example, consider the graph G shown in Figure 1.4. For this graph,

- $V = \{a, b, c, d, e\}$
- $E = \{(a,b),\ (b,c),\ (b,d),\ (c,e),\ (b,b),\ (a,d),\ (e,d)\}$

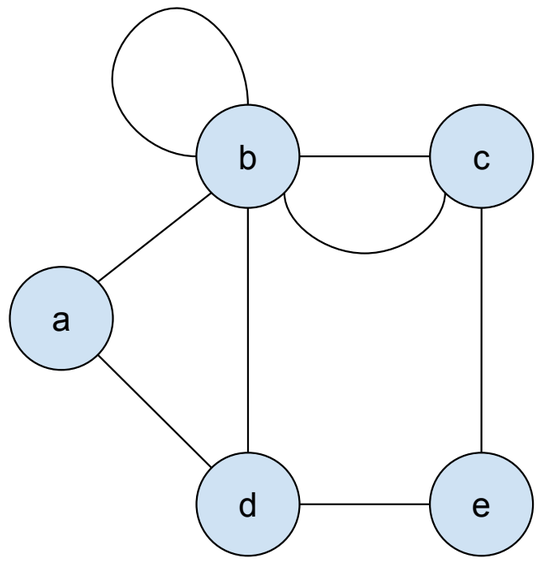


Figure 1.4: Example for graph

A **directed graph (Digraph)** is a graph G, such that $G =< V,\ E >$, where V is the set of all vertices and E is the set of ordered pairs of elements from V. If (a)

For example, consider the digraph G shown in Figure 1.5. For this graph,

- $V = \{a,b,c,d,e\}$
- $E = \{(a,b),\ (b,c),\ (b,d),\ (c,e),\ (b,b),\ (d,a),\ (d,e)\}$

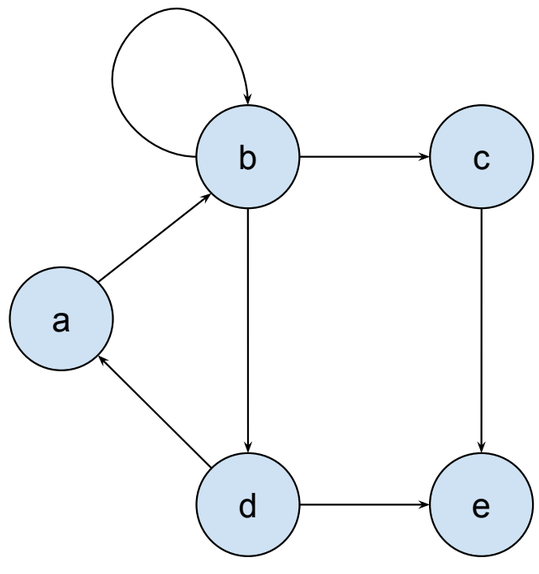


Figure 1.5: Example for digraph

A graph or digraph is called a **weighted graph** or **weighted digraph** if all the edges in it are labelled with some weights.

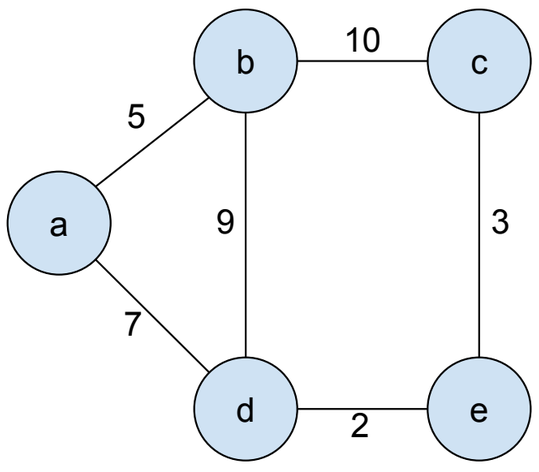


Figure 1.6: Example for weighted graph

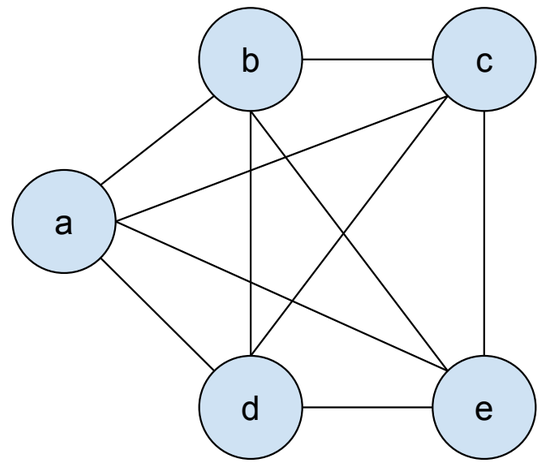


Figure 1.7: Example for complete graph

Adjacent Vertices

In a graph, a vertex a is adjacent to (or neighbour of) vertex b, if there is an edge from a to b. For example, in Figure 1.4, vertices b and d are adjacent to vertex a.

Self Loop

If there is an edge with same starting and ending vertex, then it is called a self loop or loop. For example, in Figure 1.4, edge (b,b) is a self loop.

Parallel Edges

If there is more than one edge between the same pair of vertices, then they are known as parallel edges.For example, in Figure 1.4, there are two parallel edges between vertices b and c.

Multigraph

If a graph has either self loop or parallel edges then it is called a multigraph. Hence the graph in Figure 1.4 is a multigraph.

Simple Graph

If a graph has neither a self loop nor parallel edges then it is called a simple graph. Hence the graph in Figure 1.6 is a simple graph.

Complete Graph

A graph or digraph G is called a complete graph, if each vertex v_i is adjacent to every other vertex v_j in G. Hence the graph in Figure 1.7 is a complete graph. A standard notation for the complete graph with n vertices is K_n.

Path

A path from vertex u to vertex v of a graph G can be defined as a sequence of adjacent vertices that starts with u and ends with v. If all vertices of a path are distinct, the path is said to be **simple**.

Acyclic Graph

If there is a path containing one or more edges which starts at vertex v_i and ends at v_i, then the path is known as a cycle. If a graph does not have any cycle, then the graph is called an acyclic graph.

Degree of vertex

In an undirected graph, the number of edges connected to with vertex v_i is called the degree of vertex v_i and is denoted by $degree(v_i)$. For example, in Figure 1.4, $degree(a) = 2$, $degree(d) = 3$.

For a directed graph there are two degrees,

- **Indegree** of a node is the number of edges incident on the node. For example, in Figure 1.5, $indegree(d) = 1$.

- **Outdegree** of a node is the number of edges emanating from the node. For example, in Figure 1.5, $outdegree(d) = 2$.

Dense and Sparse Graphs

A graph with relatively few possible edges missing is called dense; a graph with few edges relative to the number of its vertices is called sparse.

Connected Graph

Two vertices v_i and v_j in a graph G are said to be connected, if there is a path in G from v_i to v_j. A graph is said to be connected, if there is path between every pair of vertices in G.

A digraph is said to be **strongly connected**, if there is directed path between every pair of vertices. If a digraph is not strongly connected, but the underlying undirected graph is connected, then the graph is said to be **weekly connected**.

Representation of Graphs

- **Set Representation**
  Two sets are maintained
  1. The set of vertices V.
  2. The set of edges, E, which is the subset of $V \times V$.

  If the graph is weighted, the set E is the subset of $W \times V \times V$, where W is the set of weights.
  For example, the set representation of graph in Figure 1.4 is
  - $V = \{a,b,c,d,e\}$
  - $E = \{(a,b),\ (b,c),\ (b,d),\ (c,e),\ (b,b),\ (a,d),\ (e,d)\}$

  The set representation of the weighted graph in Figure 1.6 is
  - $V = \{a,b,c,d,e\}$
  - $E = \{(5,a,b),\ (10,b,c),\ (9,b,d),\ (3,c,e),\ (7,a,d),\ (2,e,d)\}$
- **Linked Representation**
  In the linked representation, two types of nodes are used, one for unweighted graphs and the other for weighted graphs.

(a) For weighted graph (b) For unweighted graph

Figure 1.8: Node structures for graph

For example, the linked representation of graph in Figure 1.4 is shown below.

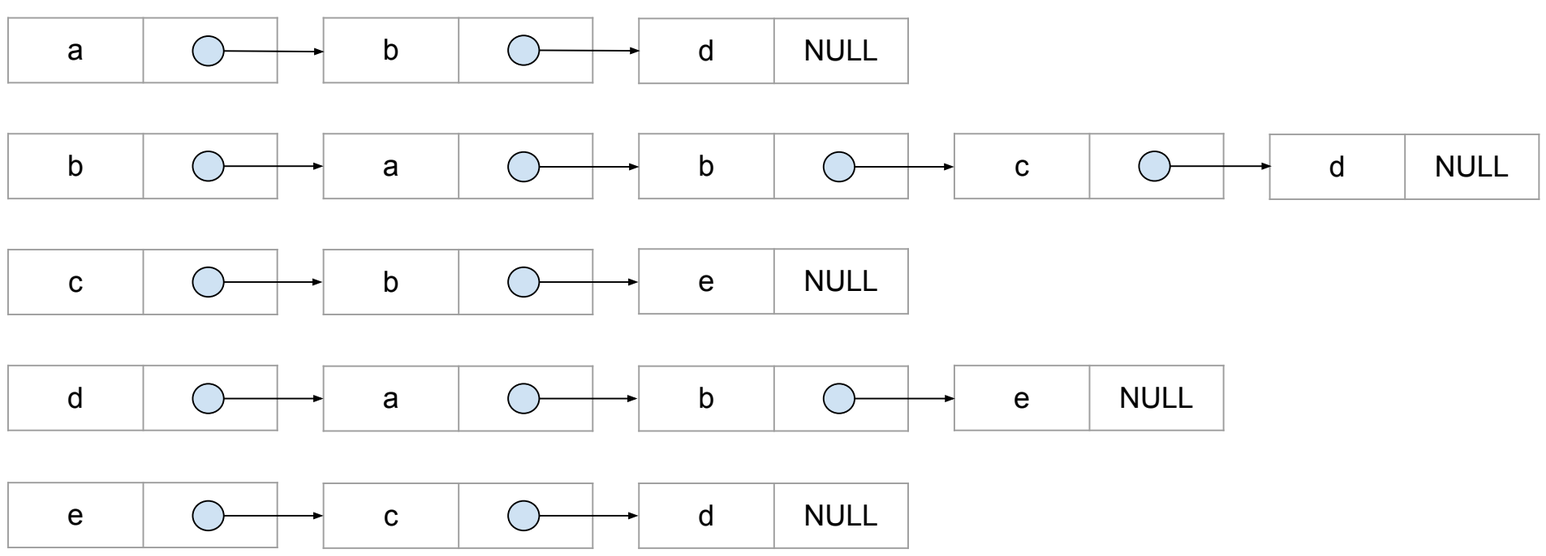


Figure 1.9: Linked representation of graph in Figure 1.4

- **Matrix Representation**
  A square matrix of order $n \times n$, where n is the number of vertices, is used to represent a graph. This matrix is known as the adjacency matrix. Entries in the adjacency matrix are defined as follows.

$$\begin{aligned} a_{ij} &= 1, \text{ If there is an edge from } v_i \text{ to } v_j \\ &= 0, \text{ otherwise} \end{aligned}$$

  In case of a multigraph, the entries in adjacency matrix will be the number of edges between the vertices, instead of 1. In case of a weighted graph, the entries in the adjacency matrix are weights of the edges between two vertices.
  The adjacency matrix representation of the graph in Figure 1.4 is shown below

$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 2 & 1 & 0 \\ 0 & 2 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

The adjacency matrix representation of the weighted graph in Figure 1.6 is shown below

$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 5 & 0 & 7 & 0 \\ 5 & 0 & 10 & 9 & 0 \\ 0 & 10 & 0 & 0 & 3 \\ 7 & 9 & 0 & 0 & 2 \\ 0 & 0 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

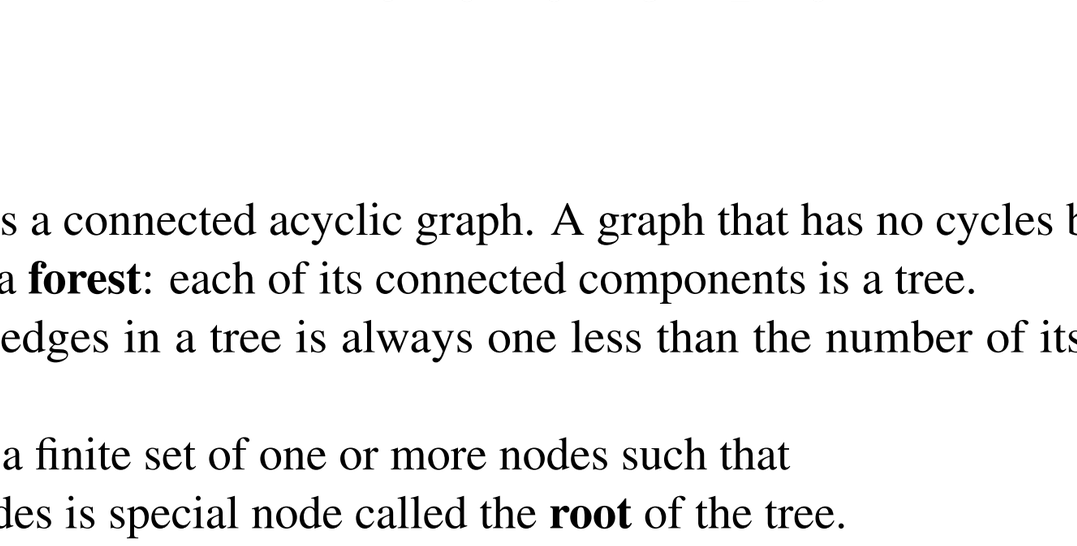
1.4.3 Trees

A tree (a free tree) is a connected acyclic graph. A graph that has no cycles but is not necessarily connected is called a **forest**: each of its connected components is a tree.

The number of edges in a tree is always one less than the number of its vertices. i.e. $|E| = |V| - 1$.

A rooted tree is a finite set of one or more nodes such that

- one of the nodes is special node called the **root** of the tree.
- the remaining nodes are partitioned into n ($n > 0$) disjoint sets $T_1, T_2, \ldots T_n$, where each T_i is a tree. $T_1, T_2, \ldots T_n$ are called subtrees of the root.



- **Node:** A tree is made up nodes. A node contains the data and links to other nodes.

- **Parent:** Parent of a node is the immediate predecessor of a node.
- **Child:** All the immediate successors of a node are called its children. The child on the left side is called the left child and the child on the right side is called the right child.
- **Root:** Root is a special node which has no parent.
- **Leaf:** The node which is at the end and does not have any child is called a leaf node.
- **Level:** Level is the rank of a node in the hierarchy. Root node is at level 0. If a node is at level l then all its children are at level $l+1$ and the parent is at level $l-1$.
- **Height:** The maximum number of nodes that is possible in a path starting from the root node to the leaf node is called the height of the tree.
- **Degree:** The maximum number of children that is possible for a node in a tree is called the degree of the node.
- **Sibling:** The nodes having the same parent are called siblings.
- **Ancestors:** For any vertex v in a tree T , all the vertices on the simple path from the root to that vertex are called ancestors of v.
- **Descendants:** All the vertices for which a vertex v is an ancestor are said to be descendants of v.

Binary Trees

A binary tree T is a finite set of nodes such that

- T is empty (called the empty binary tree) or
- T contains a special node called the root of T, and the remaining nodes of T form two disjoint binary tree T_1 and T_2 called left sub tree and right subtree respectively.

Each node in a binary tree can have at most 2 children.

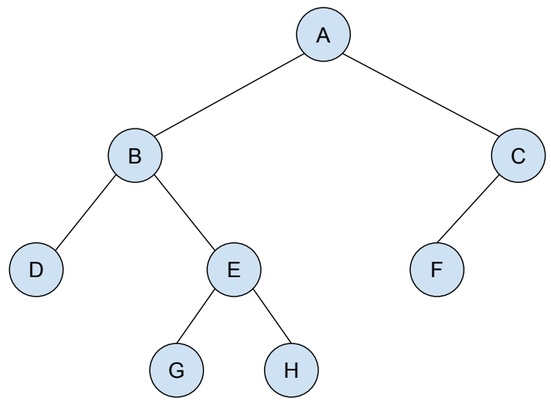


Figure 1.10: Binary Tree

Full Binary Tree

A binary tree is a full binary tree if it contains the maximum possible number of nodes at all levels.

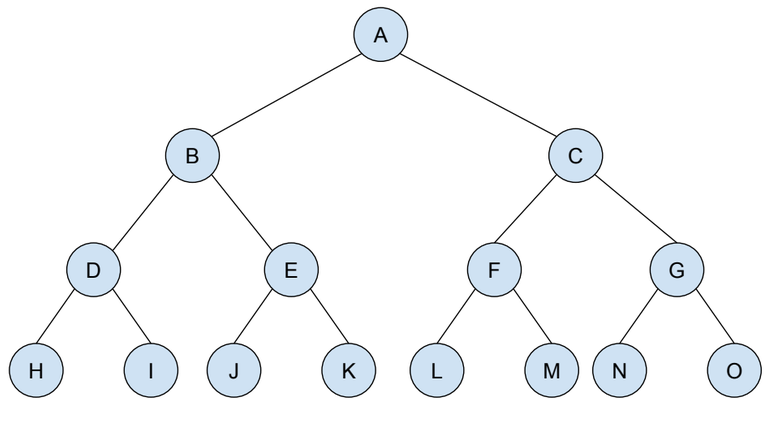


Figure 1.11: Full Binary Tree

Complete Binary Tree

A binary tree is a complete binary tree if it contains the maximum possible number of nodes at all levels except possibly the last level and all the nodes in the last level appear as far left as possible. All full binary trees are complete binary trees, bot not vice versa.

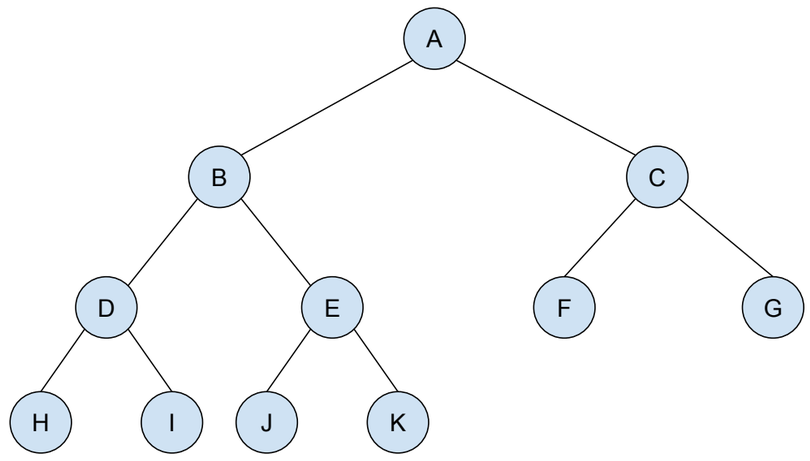


Figure 1.12: Complete Binary Tree

Binary Search Tree (BST)

A binary tree T is a binary search tree, if each node N of T satisfies the following property:

- The value of N is greater than every value in the left subtree of N and is less than every value in the right subtree of N.

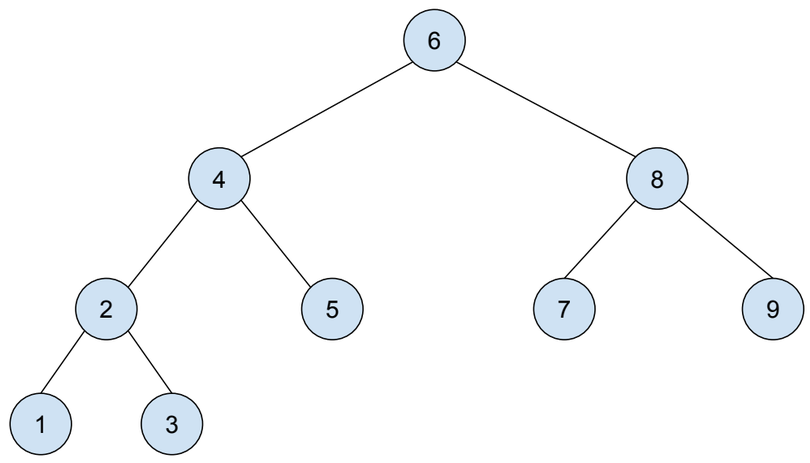


Figure 1.13: Example for Binary Search Tree

1.4.4 Sets and Dictionaries

A set is an unordered collection (possibly empty) of distinct items called elements of the set. A set can be defined in either by

- Listing of its elements (e.g., $S = \{1, 3, 5, 7, 9\}$) or by
- Specifying a property that all the set's elements and only they must satisfy
  e.g., $S = \{n: n \text{ is an odd number smaller than } 10\}$.

The most important set operations are: checking membership of a given item in a given set; finding the union of two sets, which comprises all the elements in either or both of them; and finding the intersection of two sets, which comprises all the common elements in the sets. Sets can be implemented in computer applications in two ways.

- Consider only sets that are subsets of some large set U, called the universal set. If set U has n elements, then any subset S of U can be represented by a bit string of size n, called a **bit vector**, in which the i^{th} element is 1 if and only if the i^{th} element of U is included in set S. For example, if $U = \{1,2,3,4,5,6,7,8,9,10\}$, then $S = \{1, 3, 5, 7, 9\}$ can be represented as 1010101010.
- Use the list structure to indicate the set's elements.

A **multiset**, or **bag**, an unordered collection of items that are not necessarily distinct.

In computing, the most common operations to be performed for a set or a multiset are searching for a given item, adding a new item, and deleting an item from the collection. A data structure that implements these three operations is called the **dictionary**.

1.4.5 Abstract Data Type (ADT)

An ADT is a set of abstract objects representing data items with a collection of operations that can be performed on them.

2. Fundamentals of Analysis of Algorithms

2.1 The Analysis Framework

For an algorithm, there are two kinds of efficiencies:

- **Time efficiency (Time Complexity)**: Indicates how fast the algorithm runs.
- **Space efficiency (Space Complexity)**: Indicates the amount of memory required by the algorithm in addition to the space needed for the input and output.

2.1.1 Measuring an Input's Size

Most algorithms take more time to execute with larger input sizes. Therefore, an algorithm's efficiency can be measured as a function of some parameter n, indicating the algorithm's input size.

- For searching and sorting algorithms, n could be the size of the list.
- For algorithms involving matrices, it could be the number of elements in the matrix.
- For algorithms solving problems such as checking primality of a positive integer n, it could be the number of bits b required to represent the number.

$$b = \lfloor \log_2 n \rfloor + 1$$

2.1.2 Units for Measuring Running Time

One method is to measure the time taken to run the program implementing the algorithm. But this methods has following drawbacks.

- The time taken depends on the specifications of the computer.
- Time depends on the quality of the program written and the compiler.
- It is difficult to clock the actual running time of the program.

Better method is to identify the most important operation of the algorithm, called the **basic operation**, which is the operation contributing to most of the total running time, and then compute the number of times the basic operation is executed.

The basic operation is usually the most time consuming operation in an algorithm's inner most loop. For example, in sorting algorithms, basic operation is the key comparison.

The **best-case efficiency** of an algorithm is its efficiency for the best-case input of size n, which is an input of size n for which the algorithm runs the fastest among all possible inputs of that size. The best-case analysis provides a lower bound on the running time of an algorithm.

$$C_{best}(n) = 1$$

The **average-case efficiency** yields the necessary information about an algorithm's behaviour on a "typical" or "random" input. To analyze the algorithm's average case efficiency,some assumptions are to be made about possible inputs of size n. For example, in linear search, the assumptions are

- the probability of a successful search is p $(0 \leq p \leq 1)$
- the probability of the first match occurring in the i^{th} position of the list is the same for every i, which is equal to $\frac{p}{n}$

In the case of a successful search, the probability of the first match occurring in the i^{th} position of the list is $\frac{p}{n}$ for every i, and the number of comparisons made by the algorithm in such a situation is i. In the case of an unsuccessful search, the number of comparisons will be n with the probability of such a search being $(1-p)$.

$$\begin{aligned} C_{avg}(n) &= [1.\frac{p}{n} + 2.\frac{p}{n} + 3.\frac{p}{n} + ... + i.\frac{p}{n} + ... + n.\frac{p}{n}] + n.(1-p) \\ &= \frac{p}{n}[1+2+3+...i+...+n] + n(1-p)] \\ &= \frac{p}{n}.\frac{n(n+1)}{2} + n(1-p) \\ &= \frac{p(n+1)}{2} + n(1-p) \end{aligned}$$

From this equation, if $p = 1$ (the search is successful), the average number of key comparisons made by sequential search is $\frac{n+1}{2}$, that is, the algorithm will inspect, on average, about half of the list's elements. If $p = 0$ (the search is unsuccessful), average number of key comparisons will be n, because the algorithm will inspect all n elements on all such inputs.

2.2 Asymptotic Notations and Basic Efficiency Classes

Three notations used to compare and rank order of growth of functions are

- O (big oh)
- Ω (big omega)
- Θ (big theta)

2.2.1 Informal Introduction

Informally, $O()$ is the set of all functions with a lower or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). For example,

$$\begin{aligned} n &\in O(n^2) \\ 5n+3 &\in O(n^2) \\ \frac{1}{2}n(n-1) &\in O(n^2) \end{aligned}$$

But the following function does not belong to $O(n)$

$$\begin{aligned} n3 &\notin O(n^2) \\ 0.00001n^3 &\notin O(n^2) \\ n^4+n+6 &\notin O(n^2) \end{aligned}$$

Informally, $\Omega(g(n))$ is the set of all functions with a higher or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). For example,

$$\begin{aligned} n3 &\in \Omega(n^2) \\ \frac{1}{2}n(n-1) &\in \Omega(n^2) \\ 5n+3 &\notin \Omega(n^2) \end{aligned}$$

Informally, $\Theta(g(n))$ is the set of all functions with same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). For example,

$$\begin{aligned} \frac{1}{2}n(n-1) &\in \Theta(n^2) \\ 5n^2+3 &\in \Theta(n^2) \end{aligned}$$

2.2.2 Formal Definitions

O-notation

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq c.g(n) \quad \text{for all} \quad n \geq n_0$$

For example to prove that $5n+3 \in O(n^2)$,

$$\begin{aligned} 5n+3 &\leq 5n+n \ (\text{for all} \ \ n \geq 3) = 6n \\ &\leq 6n^2 \end{aligned}$$

In this case, the value of c and n_0 are 6 and 3 respectively.

This can also be proved with another set of values for c and n_0 as shown below.

$$\begin{aligned} 5n+3 &\leq 5n+3n \ (\text{for all} \ \ n \geq 1) = 8n \\ &\leq 8n^2 \end{aligned}$$

In this case, the value of c and n_0 are 8 and 1 respectively.

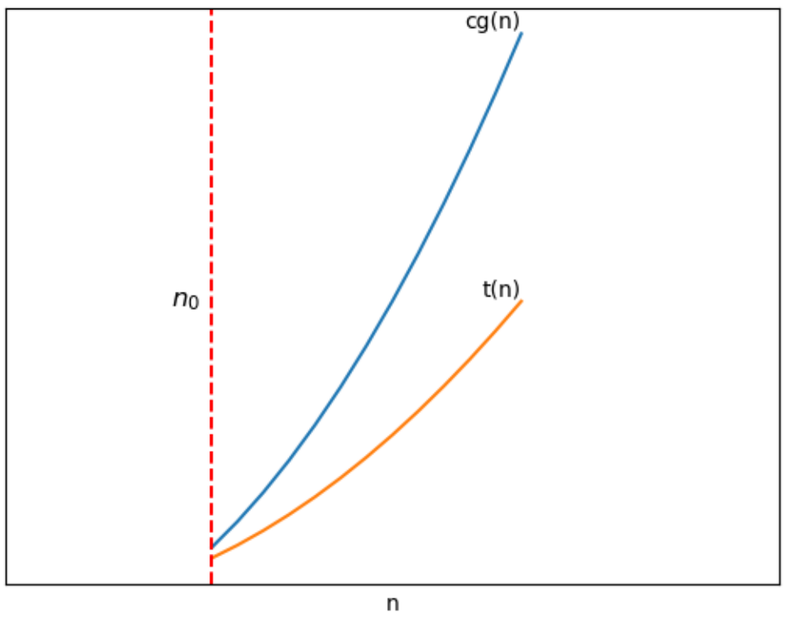


Figure 2.1: Big-oh notation: $t(n) \in O(g(n))$

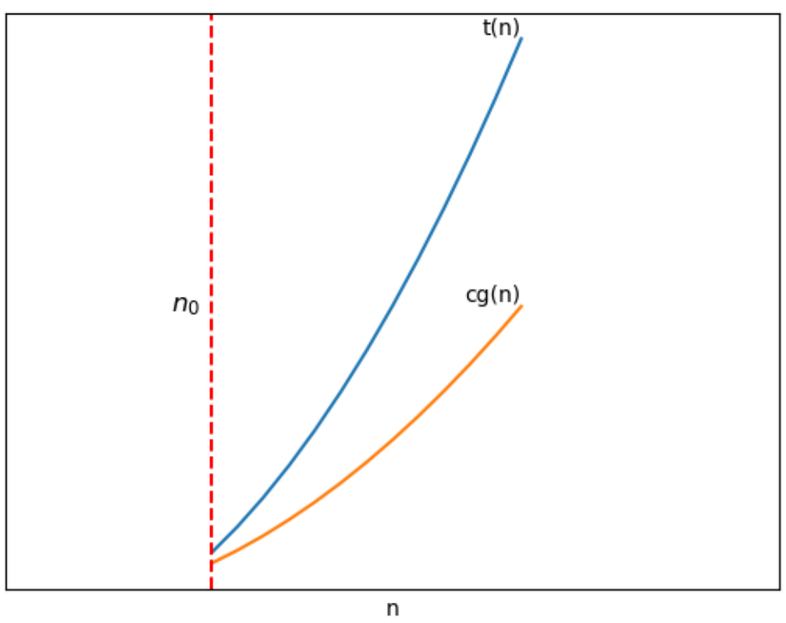


Figure 2.2: Big-omega notation: $t(n) \in \Omega(g(n))$

Ω-notation

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq c.g(n) \quad \text{for all} \quad n \geq n_0$$

For example to prove that $n^3 \in \Omega(n^2)$,

$$n^3 \geq n^2 \text{ (for all } n \geq 0)$$

In this case, the value of c and n_0 are 1 and 0 respectively.

Θ-notation

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded above and below by some constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2.g(n) \leq t(n) \leq c1.g(n) \quad \text{for all} \quad n \geq n_0$$

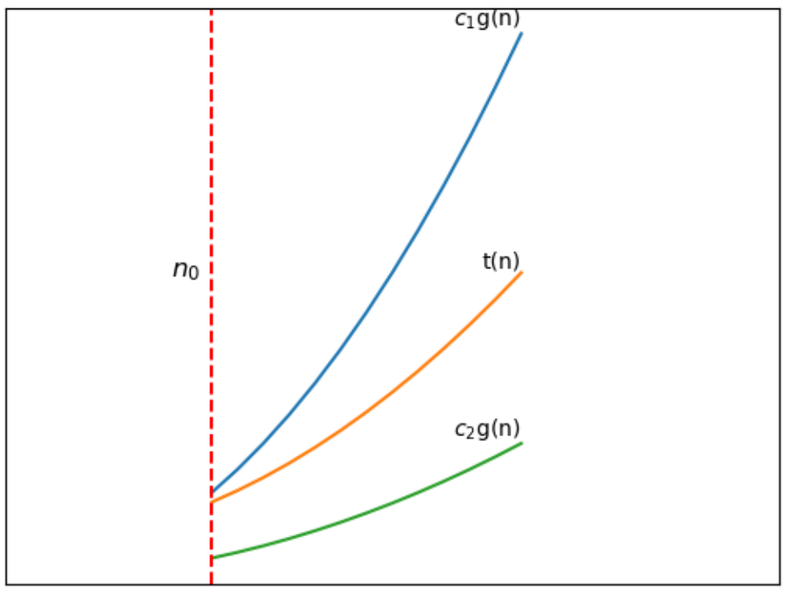


Figure 2.3: Big-theta notation: $t(n) \in \Theta(g(n))$

For example to prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$, first prove the right inequality

$$\frac{1}{2}n(n-1) \quad = \quad \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \text{ (for all } n \geq 0)$$

Then prove the left inequality

$$\frac{1}{2}n(n-1) \quad = \quad \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \geq 2) = \frac{1}{4}n^2$$

In this case, the value of $c_1 = \frac{1}{2}$, $c_2 = \frac{1}{4}$ and $n_0 = 2$.

2.2.3 Useful Property Involving the Asymptotic Notations

Theorem 2.2.1 If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(max\{g_1(n), g_2(n)\})$$

Proof

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some non-negative integer n_1 such that

$$t_1(n) \leq c_1.g_1(n) \quad \text{for all } n \geq n_1$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \leq c_2.g_2(n) \quad \text{for all } n \geq n_2$$

Let $c_3 = max\{c_1, c_2\}$ and consider $n \geq max\{n_1, n_2\}$. Adding both inequalities above will yield

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1.g_1(n) + c_2.g_2(n) \\ &\leq c_3.g_1(n) + c_3.g_2(n) = c_3(g_1(n) + g_2(n)) \\ &\leq c_3.2(max\{g_1(n), g_2(n)\}) \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(max\{g_1(n), g_2(n)\})$, with $c = 2c_3 = 2max\{c_1, c_2\}$ and $n_0 = max\{n_1, n_2\}$

This theorem can be used to check for example, whether an array has equal elements by the following two-part algorithm: first, sort the array by applying some known sorting algorithm; second, scan the sorted array to check its consecutive elements for equality. If, for example, a sorting algorithm used in the first part is in $O(n^2)$ while the second part is in $O(n)$, the efficiency of the entire algorithm will be in $O(max\{n^2, n\}) = O(n^2)$.

2.2.4 Using Limits for Comparing Orders of Growth

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n)\\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n)\\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n)\end{cases}$$

The limit-based approach is often more convenient than the one based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as L'Hopital's rule

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n!\approx\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \text{ for large values of } n$$

■ **Example 2.1** Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. ■

$$\begin{aligned}\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2} &= \frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}\\ &= \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right)\\ &= \frac{1}{2}\end{aligned}$$

Since the limit is equal to a positive constant, the functions have the same order of growth, i.e. $\frac{1}{2}n(n-1)\in\Theta(n^2)$

■ **Example 2.2** Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$ ■

$$\begin{aligned}\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}} &= \lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}\\ &= \lim_{n\to\infty}\frac{\log_2 e\frac{1}{n}}{\frac{1}{2\sqrt{n}}}\\ &= 2\log_2 e\lim_{n\to\infty}\frac{1}{\sqrt{n}}\\ &= 0\end{aligned}$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$.

■ **Example 2.3** Compare the orders of growth of $n!$ and 2^n ■

$$\begin{aligned}\lim_{n\to\infty}\frac{n!}{2^n} &= \lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}\\ &= \lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n}\\ &= \lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n\\ &= \infty\end{aligned}$$

Since the limit is equal to ∞, $n!$ has a larger order of growth than 2^n.

2.2.5 Basic Efficiency Classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops. Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops. Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Table 2.2: Basic Efficiency Classes