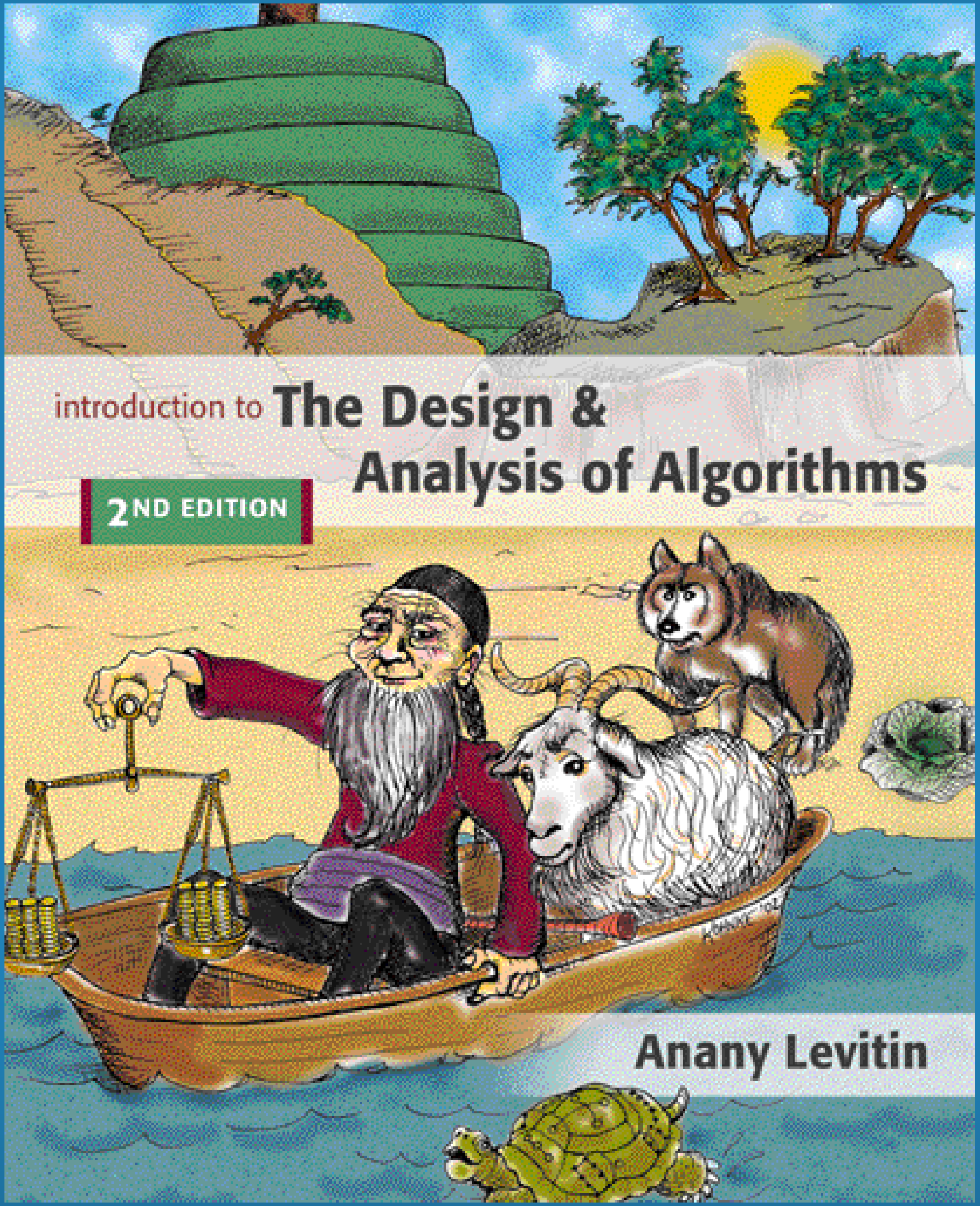
Chapter 1

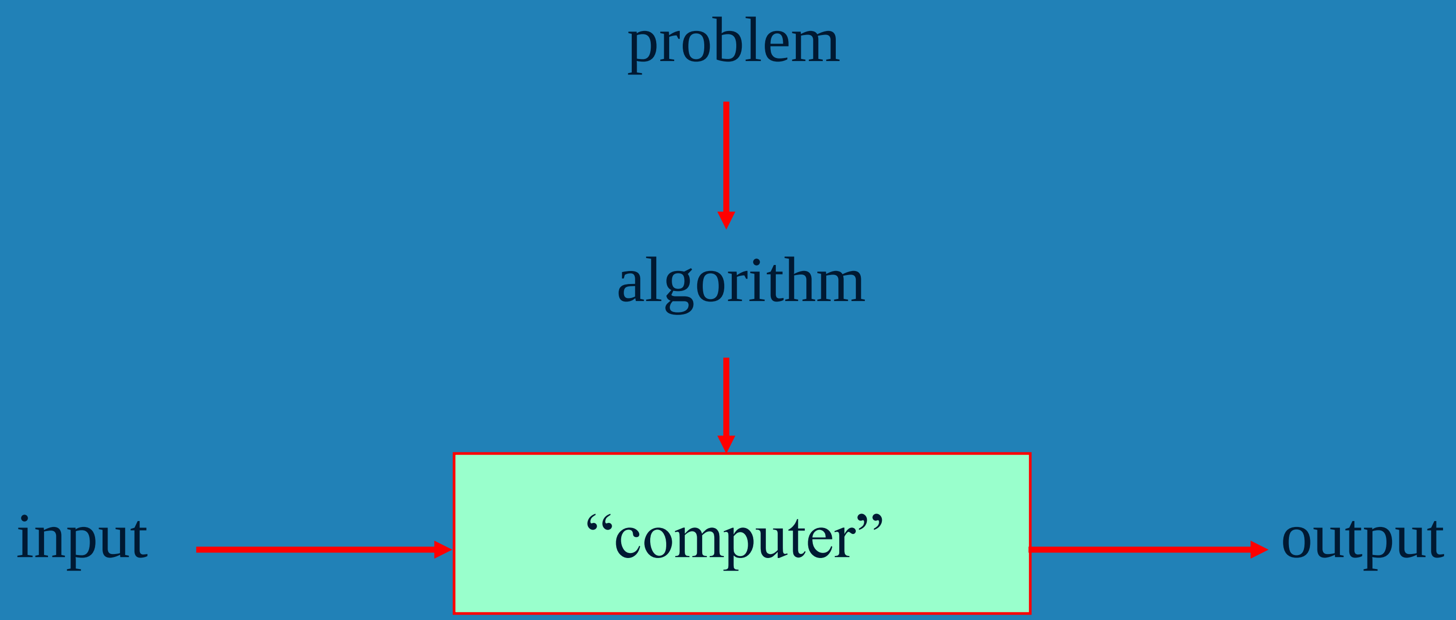
Introduction



PEARSON
Addison Wesley

What is an algorithm?

An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



Algorithm

- **An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.**

  - Can be represented various forms
  - Unambiguity/clearness
  - Effectiveness
  - Finiteness/termination
  - Correctness

Historical Perspective

- **Euclid's algorithm for finding the greatest common divisor**

- **Muhammad ibn Musa al-Khwarizmi – 9th century mathematician**
  www.lib.virginia.edu/science/parshall/khwariz.html

Example of computational problem: sorting

- **Statement of problem:**
  - ***Input:*** **A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$**
  - ***Output:*** **A reordering of the input sequence $\langle a'_1, a'_2, \ldots, a'_n \rangle$ so that $a'_i \le a'_j$ whenever $i < j$**
- **Instance: The sequence $\langle 5, 3, 2, 8, 3 \rangle$**
- **Algorithms:**
  - **Selection sort**
  - **Insertion sort**
  - **Merge sort**
  - **(many others)**

Selection Sort

- **Input: array a[1], …, a[n]**
- **Output: array a sorted in non-decreasing order**
- **Algorithm:**

for i**=1 to** n
    swap a[i] **with smallest of** a[i], …, a[n]

- Is this unambiguous? Effective?
- See also pseudocode, section 3.1

Some Well-known Computational Problems

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing
- Traveling salesman problem
- Knapsack problem
- Towers of Hanoi

Basic Issues Related to Algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness
- Efficiency (or complexity) analysis
  - Theoretical analysis
  - Empirical analysis
- Optimality

Algorithm design strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Greedy approach
- Dynamic programming
- Backtracking and branch-and-bound
- Space and time tradeoffs

Analysis of Algorithms

- How good is the algorithm?
  - Correctness
  - Time efficiency
  - Space efficiency

- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

What is an algorithm?

- Requirements:

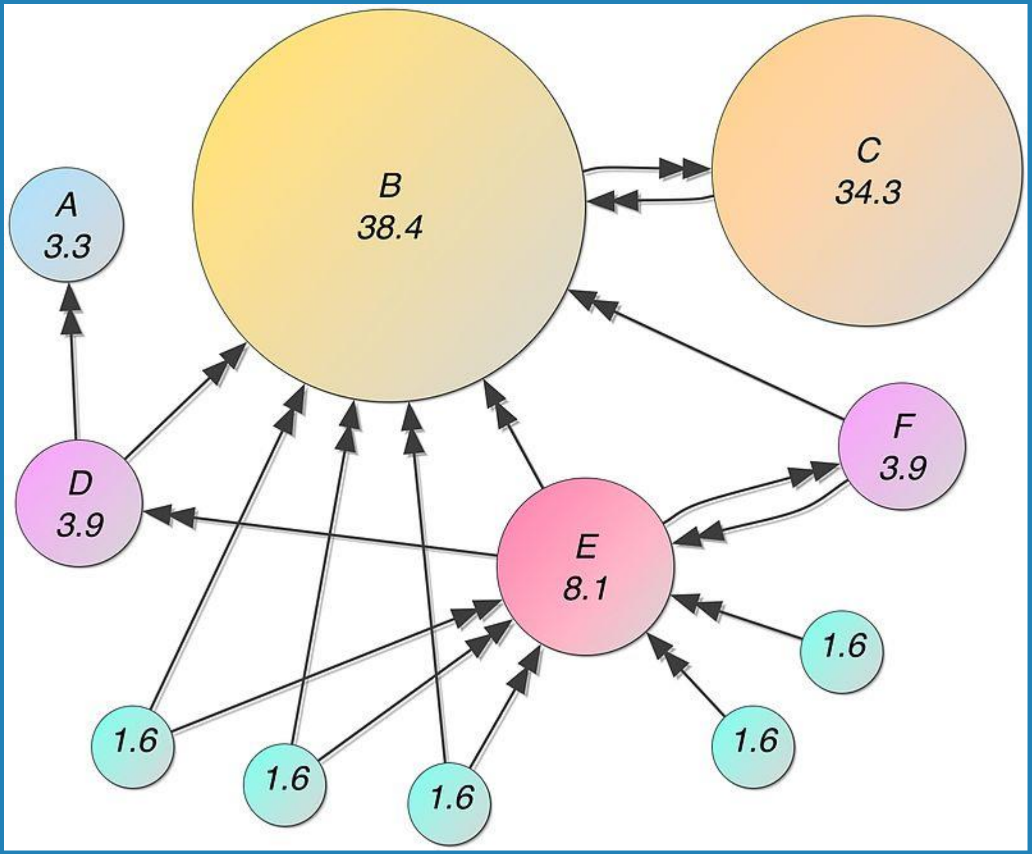
1. Finiteness
   - terminates after a finite number of steps
2. Definiteness
   - rigorously and unambiguously specified
3. Clearly specified input
   - valid inputs are clearly specified
4. Clearly specified/expected output
   - can be proved to produce the correct output given a valid input
5. Effectiveness
   - steps are sufficiently simple and basic

Why study algorithms?

- **Theoretical importance**
  - **the core of computer science**
- **Practical importance**
  - **A practitioner's toolkit of known algorithms**
  - **Framework for designing and analyzing algorithms for new problems**

Example: Google's PageRank Technology

....



A
3.3
B
38.4
C
34.3
D
3.9
E
8.1
F
3.9
1.6
1.6
1.6
1.6
1.6
1.6

Euclid's Algorithm

Problem: Find gcd(m,n), the greatest common divisor of two nonnegative, not both zero integers m and n

Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

Euclid's algorithm is based on repeated application of equality

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

until the second number becomes 0, which makes the problem trivial.

Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

Two descriptions of Euclid's algorithm

Step 1 **If *n* = 0, return *m* and stop; otherwise go to Step 2**

Step 2 **Divide *m* by *n* and assign the value of the remainder to *r***

Step 3 **Assign the value of *n* to *m* and the value of *r* to *n*. Go to Step 1.**

while $n \neq 0$ **do**

$\quad r \leftarrow m \bmod n$

$\quad m \leftarrow n$

$\quad n \leftarrow r$

return m

Other methods for computing gcd(m,n)

Consecutive integer checking algorithm

Step 1 Assign the value of min{m,n} to t

Step 2 Divide m by t. If the remainder is 0, go to Step 3; otherwise, go to Step 4

Step 3 Divide n by t. If the remainder is 0, return t and stop; otherwise, go to Step 4

Step 4 Decrease t by 1 and go to Step 2

Other methods for gcd(*m,n*) [cont.]

Middle-school procedure

Step 1 Find the prime factorization of *m*

Step 2 Find the prime factorization of *n*

Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd(*m,n*)

Sieve of Eratosthenes

Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

for $p \leftarrow 2$ **to** n **do** $A[p] \leftarrow p$

for $p \leftarrow 2$ **to** n **do**

  if $A[p] \neq 0$ //p hasn't been previously eliminated from the list

   $j \leftarrow p * p$

   while $j \leq n$ **do**

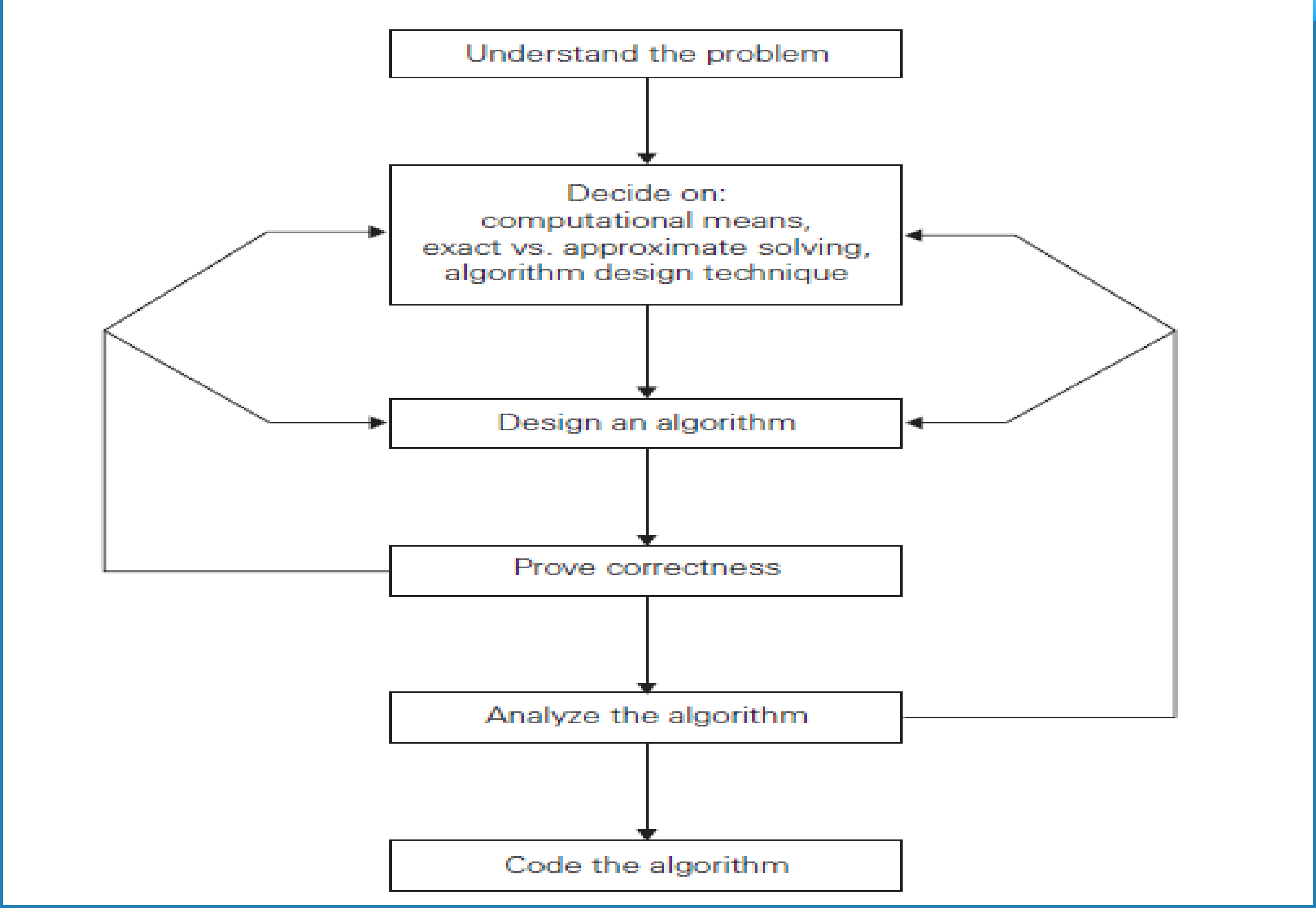
    $A[j] \leftarrow 0$ //mark element as eliminated

    $j \leftarrow j + p$

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Time complexity: O(n)

Algorithm design and analysis process



Two main issues related to algorithms

- How to design algorithms

- How to analyze algorithm efficiency

Algorithm design techniques/strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Backtracking
- Branch and bound

Analysis of algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency

- Does there exist a better algorithm?
  - optimality

Locker Door-Puzzle

Locker doors There are n lockers in a hallway, numbered sequentially from 1 to n. Initially, all the locker doors are closed. You make n passes by the lockers, each time starting with locker #1. On the ith pass, $i = 1, 2, \ldots, n$, you toggle the door of every ith locker: if the door is closed, you open it; if it is open, you close it. After the last pass, which locker doors are open and which are closed? How many of them are open?

Important problem types

- sorting
- searching
- string processing
- graph problems
- combinatorial problems
- geometric problems
- numerical problems

Sorting (I)

- Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers $<a_1, a_2, \ldots, a_n>$
  - Output: A reordering $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \le a'_2 \le \ldots \le a'_n$.
- Why sorting?
  - Help searching
  - Algorithms often use sorting as a key subroutine.
- Sorting key
  - A specially chosen piece of information used to guide sorting. E.g., sort student records by names.

Sorting (II)

- Examples of sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Heap sort …
- Evaluate sorting algorithm complexity: the number of key comparisons.
- Two properties
  - Stability: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
  - In place : A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.

Selection Sort

```
Algorithm SelectionSort(A[0..n-1])
//The algorithm sorts a given array by selection sort
//Input: An array A[0..n-1] of orderable elements
//Output: Array A[0..n-1] sorted in ascending order
for i ← 0 to n – 2 do
    min ← i
    for j ← i + 1 to n – 1 do
            if A[j] < A[min]
                        min ← j
    swap A[i] and A[min]
```

Searching

- Find a given value, called a search key, in a given set.
- Examples of searching algorithms
  - Sequential search
  - Binary search …

```
Input: sorted array a[i] < … < a[j] and key x;
m ←(i+j)/2;
while i < j and x != a[m] do
    if x < a[m] then j ← m-1
                 else  i ← m+1;
 if x = a[m] then output a[m];
```

Time: O(log n)

String Processing

- **A string is a sequence of characters from an alphabet.**
- **Text strings: letters, numbers, and special characters.**
- **String matching: searching for a given word/pattern in a text.**

Examples:

(i) searching for a word or phrase on WWW or in a Word document

(ii) searching for a short read in the reference genomic sequence

Graph Problems

- Informal definition
  - A graph is a collection of points called vertices, some of which are connected by line segments called edges.
- Modeling real-life problems
  - Modeling WWW
  - Communication networks
  - Project scheduling …
- Examples of graph algorithms
  - Graph traversal algorithms
  - Shortest-path algorithms
  - Topological sorting

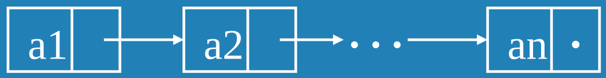
Fundamental data structures

- list
  - array
  - linked list
  - string
- stack
- queue
- priority queue/heap
- graph
- tree and binary tree
- set and dictionary

Linear Data Structures

- **Arrays**
  - **A sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's index.**
- **Linked List**
  - **A sequence of zero or more nodes each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list.**
  - **Singly linked list (next pointer)**
  - **Doubly linked list (next + previous pointers)**

- Arrays
  - fixed length (need preliminary reservation of memory)
  - contiguous memory locations
  - direct access
  - Insert/delete
- Linked Lists
  - dynamic length
  - arbitrary memory locations
  - access by following links
  - Insert/delete

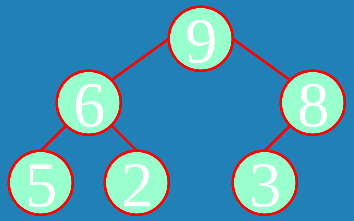


Stacks and Queues

- Stacks
  - A stack of plates
    - insertion/deletion can be done only at the top.
    - LIFO
  - Two operations (push and pop)
- Queues
  - A queue of customers waiting for services
    - Insertion/enqueue from the rear and deletion/dequeue from the front.
    - FIFO
  - Two operations (enqueue and dequeue)

Priority Queue and Heap

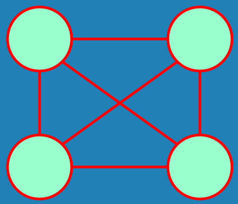
- Priority queues (implemented using heaps)
  - A data structure for maintaining a set of elements, each associated with a key/priority, with the following operations
    - Finding the element with the highest priority
    - Deleting the element with the highest priority
    - Inserting a new element
  - Scheduling jobs on a shared computer

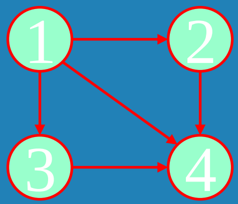


9	6	8	5	2	3

Graphs

- Formal definition
  - A graph $G = <V, E>$ is defined by a pair of two sets: a finite set V of items called **vertices** and a set E of vertex pairs called **edges**.
- **Undirected** and **directed** graphs (**digraphs**).
- What's the maximum number of edges in an undirected graph with |V| vertices?
- **Complete, dense,** and **sparse** graphs
  - A graph with every pair of its vertices connected by an edge is called complete, $K_{|V|}$

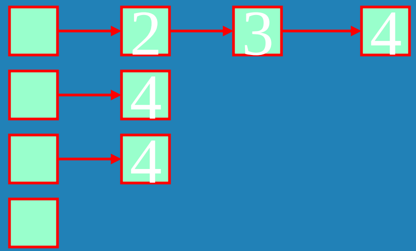




Graph Representation

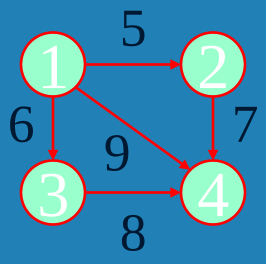
- **Adjacency matrix**
  - n x n boolean matrix if |V| is n.
  - The element on the ith row and jth column is 1 if there's an edge from ith vertex to the jth vertex; otherwise 0.
  - The adjacency matrix of an undirected graph is symmetric.
- **Adjacency linked lists**
  - A collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex.
- Which data structure would you use if the graph is a 100-node star shape?

$$\begin{matrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix}$$



Weighted Graphs

- **Weighted graphs**
  - **Graphs or digraphs with numbers assigned to the edges.**



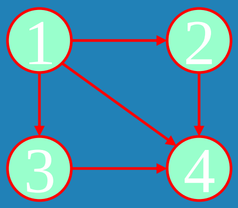
Graph Properties -- Paths and Connectivity

- **Paths**
  - **A path from vertex u to v of a graph G is defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.**
  - **Simple paths: All edges of a path are distinct.**
  - **Path lengths: the number of edges, or the number of vertices – 1.**
- **Connected graphs**

A **connected graph** is a **graph** in which it's possible to get from every vertex in the **graph** to every other vertex through a series of edges, called a path.

Graph Properties -- Acyclicity

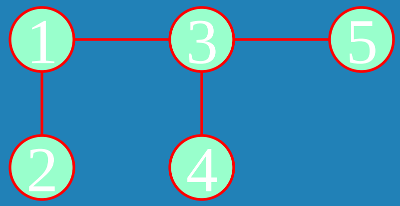
- **Cycle**
  - **A simple path of a positive length that starts and ends a the same vertex.**
- **Acyclic graph**
  - **A graph without cycles**
  - **DAG (Directed Acyclic Graph)**



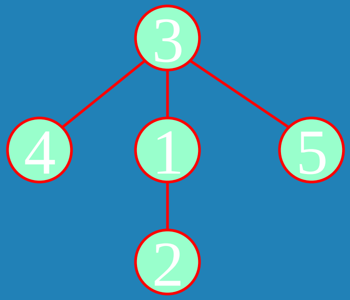
Trees

- Trees
  - A tree (or free tree) is a connected acyclic graph.
  - Forest: a graph that has no cycles but is not necessarily connected.
- Properties of trees
  - For every two vertices in a tree there always exists exactly one simple path from one of these vertices to the other. Why?
    - Rooted trees: The above property makes it possible to select an arbitrary vertex in a free tree and consider it as the root of the so called rooted tree.
    - Levels in a rooted tree.

- $|E| = |V| - 1$



rooted

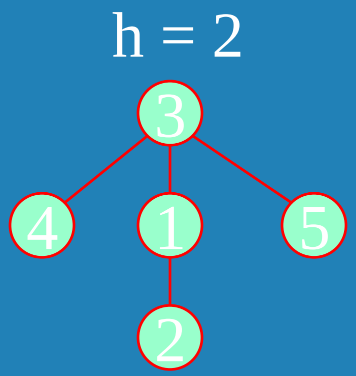


Rooted Trees (I)

- **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.
- **Descendants**
  - All the vertices for which a vertex *v* is an ancestor are said to be descendants of *v*.
- **Parent, child and siblings**
  - If *(u, v)* is the last edge of the simple path from the root to vertex *v*, *u* is said to be the parent of *v* and *v* is called a child of *u*.
  - Vertices that have the same parent are called siblings.
- **Leaves**
  - A vertex without children is called a leaf.
- **Subtree**
  - A vertex *v* with all its descendants is called the subtree of *T* rooted at *v*.

Rooted Trees (II)

- **Depth of a vertex**
  - **The length of the simple path from the root to the vertex.**
- **Height of a tree**
  - **The length of the longest simple path from the root to a leaf.**



Ordered Trees

- Ordered trees
  - An ordered tree is a rooted tree in which all the children of each vertex are ordered.
- Binary trees
  - A binary tree is an ordered tree in which every vertex has no more than two children and each children is designated s either a left child or a right child of its parent.
- Binary search trees
  - Each vertex is assigned a number.
  - A number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree.
- $\lfloor \log_2 n \rfloor \le h \le n - 1$, where h is the height of a binary tree and n the size.