Overview of Algorithms

Fall 2021

Dominic Charles Maglione

A review composed for the use of
students studying CS

Department of Computer Science
Boston University
October 26, 2021

Contents

1 Asymptotics

In mathematical analysis, **asymptotical analysis**, also known as **asymptotics**, is a method of describing limiting behavior. **Running time** is the number of computational steps an algorithm takes on an input of size n, $f(n)$ is the number of steps as a **function** of n. **Asymptotic running time** of an algorithm is a simple function that, for a sufficiently large n, is an upper/lower bound on the value of a function $f(n)$. The goal is finding the lowest/highest lower/upper bounds that hold for **all inputs** of size n.

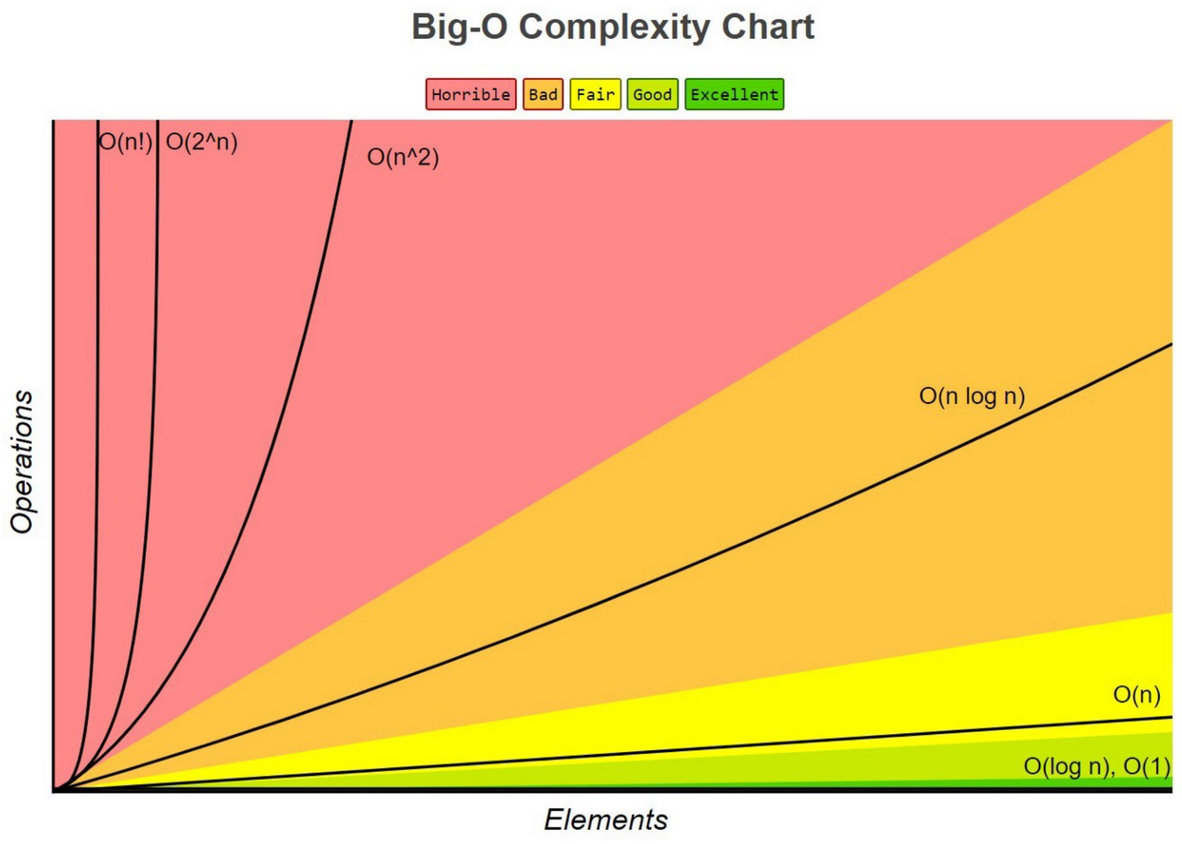


Figure 1: Big-O Complexity Chart

Analytically, let $f(n)$ be a function. Let $g(n)$ be another function $(g(n), f(n) > 0 \forall n)$. We say that $f(n)$ is $O(g(n))$ if for all sufficiently large n, the value of $f(n)$ is **bounded** by some constant $c \in \mathbb{R} \cdot g(n)$. **I.e.** $f(n) \leq c \cdot g(n)$ for any $n > n_0$.

1.1 O - Upper Bound

$f(n)$ is $O(g(n))$ if, there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Additionally, $f(n) = O(g(n))$ is about the ratio $\frac{f(n)}{g(n)}$ in the limit of large n.

Example:
$2n^2 = O(n^3) \quad (c = 1, n_0 = 2)$

Comparison:
$f(n) \leq g(n)$

Limit:
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$

Note: Big-O is not symmetric, and strictly an upper-bound notation.

1.2 Ω - Lower Bound

$f(n)$ is $\Omega g(n))$ if, there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$. Additionally, $f(n) = \Omega(g(n))$ is about the ratio $\frac{f(n)}{g(n)}$ in the limit of large n.

Example:
$\sqrt{n} = \Omega(\log n)$ or $\frac{n}{100} = \Omega(n) \quad (c = 1, n_0 = 16)$

Comparison:
$f(n) \geq g(n)$

Limit:
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > \infty \quad$ (c can be 0)

Note: Big-Ω is not symmetric, and strictly a lower-bound notation.

1.3 Θ - Tight Bound

$f(n)$ is $\Theta g(n))$ if, there exist constants $c > 0$, $n_0 > 0$ such that $O(g(n)) \cap \Theta(g(n))$ is True for all $n \geq n_0$. Additionally, $f(n) = \Theta(g(n))$ is about the ratio $\frac{f(n)}{g(n)}$ in the limit of large n.

Example:
$\frac{1}{2}n^2 - 2n = \Theta(n^2)$

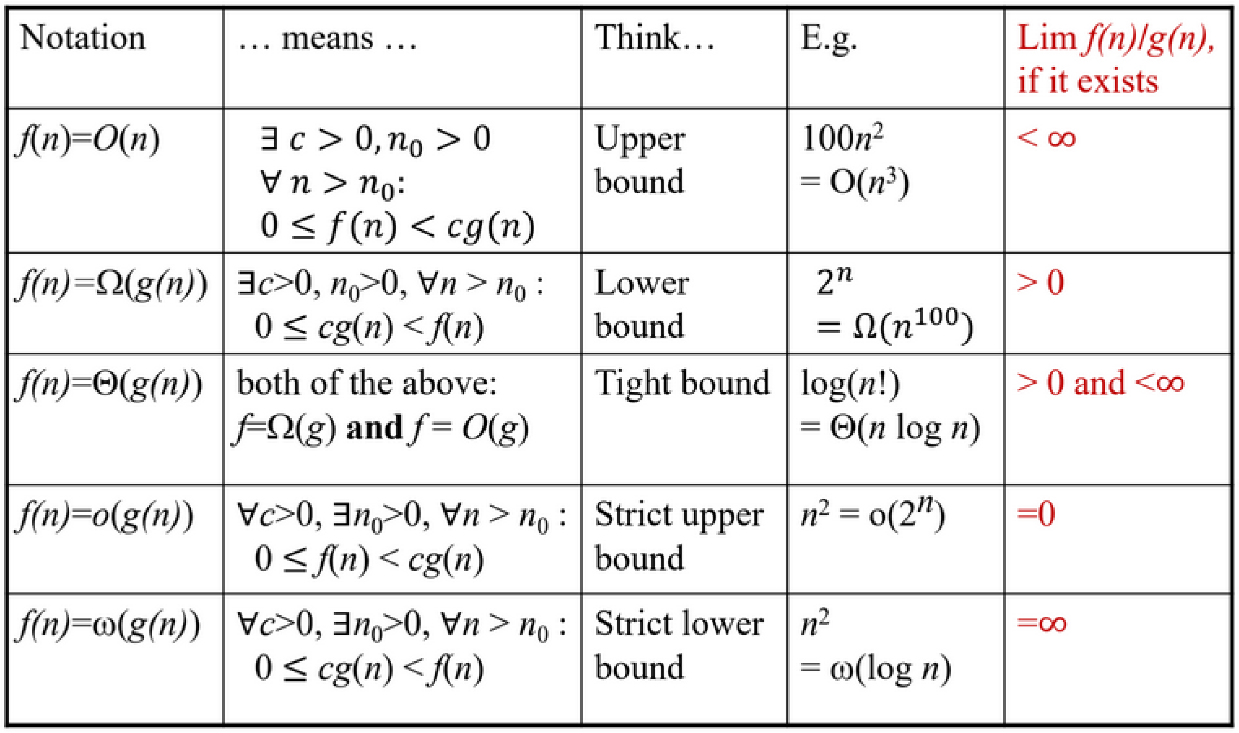
Comparison:
$f(n) = g(n)$

Limit:
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$

1.4 Common Functions

Polynomials: $a_0 + a_1 n + \cdots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.
Polynomial Time: Running time is $O(n^d)$ for some constant d independent of the input size n.
Logarithms: $\log_a n = \Theta(\log_b n)$ for all constants $a, b > 0$. For every $x > 0$, $\log n = o(n^x)$.
Exponentials: For all $r > 1$ and all $d > 0$, $n^d = o(r^n)$.
Factorial: By Sterling's Formula, $n! = (\sqrt{2\pi n})(\frac{n}{e})^n(1 + o(1)) = 2^{\Theta(n \log n)}$.

1.5 Helpful Figures



Notation	… means …	Think…	E.g.	Lim $f(n)/g(n)$, if it exists
$f(n)=O(n)$	$\exists\, c > 0, n_0 > 0$ $\forall\, n > n_0$: $0 \leq f(n) < cg(n)$	Upper bound	$100n^2 = O(n^3)$	$< \infty$
$f(n)=\Omega(g(n))$	$\exists c>0,\ n_0>0,\ \forall n > n_0$: $0 \leq cg(n) < f(n)$	Lower bound	$2^n = \Omega(n^{100})$	> 0
$f(n)=\Theta(g(n))$	both of the above: $f=\Omega(g)$ **and** $f = O(g)$	Tight bound	$\log(n!) = \Theta(n \log n)$	> 0 and $<\infty$
$f(n)=o(g(n))$	$\forall c>0,\ \exists n_0>0,\ \forall n > n_0$: $0 \leq f(n) < cg(n)$	Strict upper bound	$n^2 = o(2^n)$	$=0$
$f(n)=\omega(g(n))$	$\forall c>0,\ \exists n_0>0,\ \forall n > n_0$: $0 \leq cg(n) < f(n)$	Strict lower bound	$n^2 = \omega(\log n)$	$=\infty$

Figure 2: Overview of Asymptotic Notation

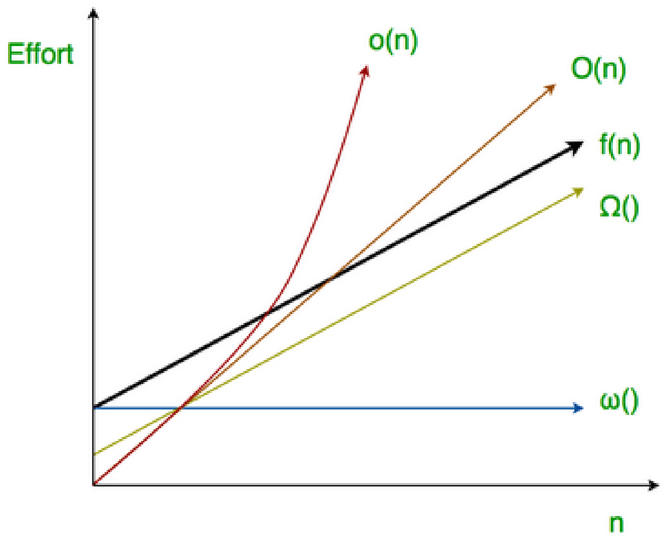


Figure 3: Graphical Representation of Asymptotic Notations

2 Data Structures

A **data structure** is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data, **i.e.**, it is an algebraic structure about data. Data structures serve as the basis for **abstract data types (ADT)**. The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

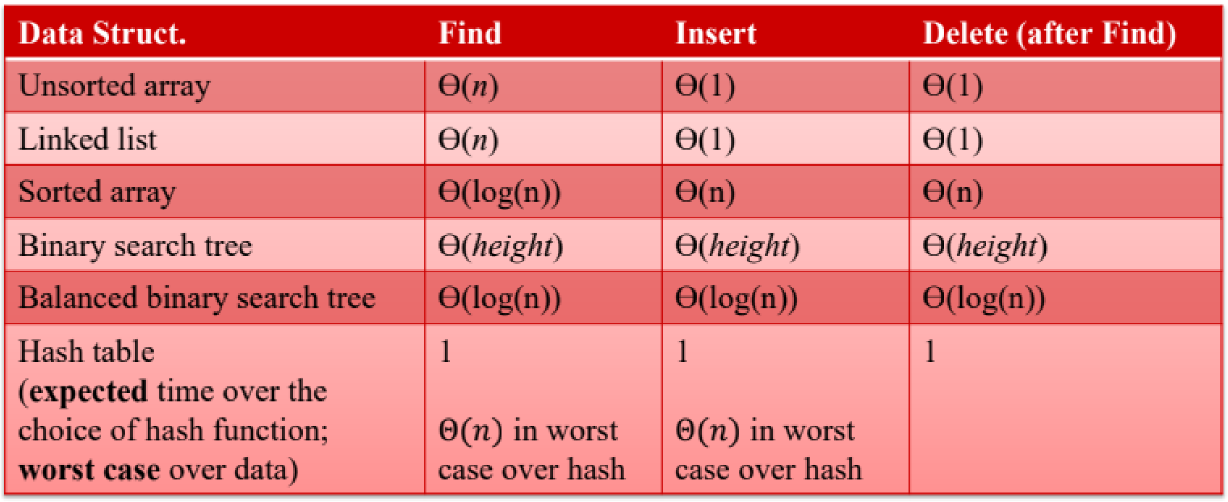
2.1 Basic Data Structures

Data Structure: Concrete Representation of Data

(i.) Array (ii.) Linked List (iii.) Binary Heap (iv.) Adjacency List Representation

Abstract Data Type: Set of Operations and Their Semantics

(i.) Priority Queue (ii.) Stack, Queue (iii.) Graph (iv.) Dictionary



Data Struct.	Find	Insert	Delete (after Find)
Unsorted array	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Sorted array	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$
Binary search tree	$\Theta(height)$	$\Theta(height)$	$\Theta(height)$
Balanced binary search tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Hash table (**expected** time over the choice of hash function; **worst case** over data)	1 $\Theta(n)$ in worst case over hash	1 $\Theta(n)$ in worst case over hash	1

Figure 4: Time Complexity's of Basic Data Structures

Lists:
$O(1)$ Time: Insert/delete anywhere there is a pointer present.

Array:
$O(1)$ Time: Append and lookup, but must pre-specify a length.

Stack ADT:
Last in, First Out (LIFO). $O(1)$ Time: Push and pop.

Queue ADT:
First in, First Out (FIFO). $O(1)$ Time: Queue and dequeue.

Dictionary:
Set of $(key, value)$ pairs.
Operations (on Dictionary S): S.insert$(key, value)$, S.Find(key), S.delete(key)

3 Bubble Sort Algorithm

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

3.1 Pseudocode

Algorithm 1: BubbleSort(A)

```
Input: $A$ is a list of sortable items
$N$ = length($A$)
for $i = 0$ to $N - 1$ do
    for $j = i$ down to $0$ do
        if $A[j] > A[j + 1]$ then
            $A[j] = A[j] + A[j+1]$
            $A[j+1] = A[j] - A[j+1]$
            $A[j] = A[j] - A[j+1]$
Output: $A$ is a list of sorted items
```

3.2 Performance

Worst-Case Performance: $O(n^2)$

Worst-Case Space Complexity: $O(n)$

3.3 Use Case

I. One of the simplest sorting algorithms to understand and implement, its $O(n^2)$ complexity means that its efficiency decreases dramatically on lists of more than a small number of elements. Even among simple $O(n^2)$ sorting algorithms, algorithms like insertion sort are usually considerably more efficient.

3.4 Appearances

Homeworks:

Homework 1: Problem 1, Part's (b) and (c), **Problem 2**

Lectures:

Lecture 1-2: Pages 22 - 24.

4 Gale-Shapley Algorithm

The **Gale–Shapley** algorithm (also known as the deferred acceptance algorithm or propose-and-reject algorithm) is an algorithm for finding a solution to the stable matching problem. It takes polynomial time, and the time is linear in the size of the input to the algorithm. It is a truthful mechanism from the point of view of the proposing participants, for whom the solution will always be optimal.

4.1 Pseudocode

Algorithm 2: StableMatching

```
Initialize all $m \in$ M and $w \in$ W to $free$
while $\exists$ $free$ man $m$ who has a woman $w$ to propose to do
    $w$ = first woman on m's list to whom m has not yet proposed
    if $\exists$ some pair (m', w) then
        if w prefers m to m' then
            m' becomes $free$
            (m, w) become engaged
    else
        (m, w) become engaged
```

4.2 Performance

Worst-Case Performance: $O(n^2)$

Worst-Case Space Complexity: $O(n^2)$

4.3 Use Case

The stable matching problem, in its most basic form, takes as input equal numbers of two types of participants (n men and n women, or n medical students and n internships, for example), and an ordering for each participant giving their preference for whom to be matched to among the participants of the other type. A stable matching always exists, and the algorithmic problem solved and proved by the Gale–Shapley algorithm is to find one.

4.4 Appearances

Homeworks:

Homework 2: Problem 1, Part's (a) and (b).

Lectures:

Lecture 1-2: Pages (68 - 78), (91 - 95).

5 Graph Theory

In mathematics, **graph theory** is the study of graphs, which are mathematical structures used to model pairwise relations between objects. A graph in this context is made up of **vertices** (also called *nodes* or *points*) which are connected by **edges** (also called *links* or *lines*). A distinction is made between undirected graphs, where edges link two vertices symmetrically, and directed graphs, where edges link two vertices asymmetrically.

5.1 Paths

A **path** in a graph is a finite or infinite sequence of edges which joins a sequence of vertices which, by most definitions, are all distinct (and since the vertices are distinct, so are the edges).

Undirected Def. A path in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_k$ with the property that each consecutive pair v_{i-1}, v_i is joined by an edge E.

Directed Def. A directed path in a directed graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_k$ with the property that each consecutive pair v_{i-1}, v_i is joined by a directed edge in E.

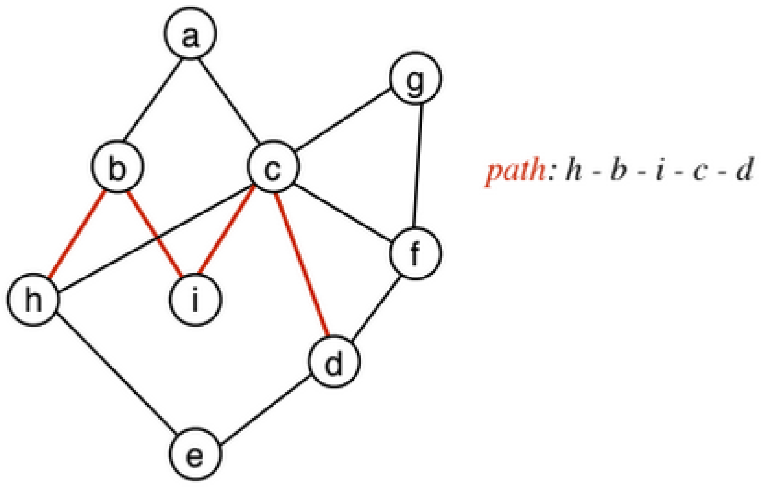


Figure 5: Depiction of a Path

5.2 Cycles

A **cycle** in a graph is a non-empty trail in which only the first and last vertices are equal. A directed cycle in a directed graph is a non-empty directed trail in which only the first and last vertices are equal. A graph without cycles is called an acyclic graph. A directed graph without directed cycles is called a directed acyclic graph. A connected graph without cycles is called a tree.

Theorem: A graph G is two colorable if and only if it has no **odd** cycle.

Bipartite Def. Every cycle C in a bipartite graph $G = (V, E)$ has an even number of nodes.

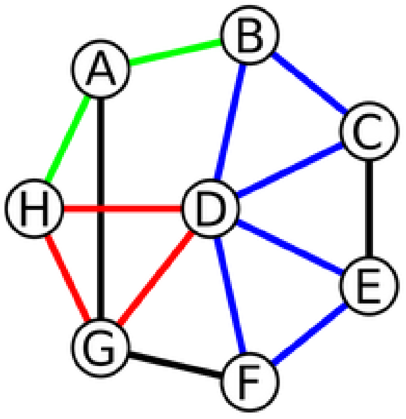


Figure 6: A Graph with Edges Colored to Illustrate Various Cycles

Cycle Detection:
The existence of a cycle in directed and undirected graphs can be determined by whether **depth-first search (DFS)** finds an edge that points to an ancestor of the current vertex (it contains a back edge). All the back edges which DFS skips over are part of cycles In an undirected graph, the edge to the parent of a node should not be counted as a back edge, but finding any other already visited vertex will indicate a back edge. In the case of undirected graphs, only $O(n)$ time is required to find a cycle in an n-vertex graph, since at most $n-1$ edges can be tree edges.

5.3 Matching

A **matching** or independent edge set in an undirected graph is a set of edges without common vertices. Finding a matching in a bipartite graph can be treated as a network flow problem. Given a matching M, an **alternating path** is a path that begins with an unmatched vertex and whose edges belong alternately to the matching and not to the matching. An augmenting path is an alternating path that starts from and ends on free (unmatched) vertices. Berge's lemma states that a matching M is maximum if and only if there is no augmenting path with respect to M.

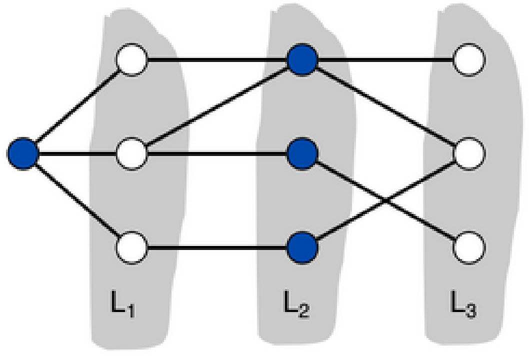


Figure 7: Depiction of a Matching in a Bipartite Graph

5.4 Reachability

Reachability refers to the ability to get from one vertex to another within a graph. A vertex s can reach a vertex t (and t is reachable from s) if there exists a sequence of adjacent vertices (i.e. a walk) which starts with s and ends with t.

In an **undirected graph**, reachability between all pairs of vertices can be determined by identifying the connected components of the graph. Any pair of vertices in such a graph can reach each other if and only if they belong to the same connected component; therefore, in such a graph, reachability is symmetric (ss reaches t iff t reaches s).

For a **directed graph** $G = (V, E)$, with vertex set V and edge set E, the reachability relation of G is the transitive closure of E, which is to say the set of all ordered pairs (s, t) of vertices in V for which there exists a sequence of vertices $v_0 = s, v_1, v_2, \ldots, v_k = t$ such that the edge (v_{i-1}, v_i) is in E for all $1 \leq i \leq k$.

5.5 Connected Components

A **component** of an undirected graph is an induced subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the rest of the graph. For example, the graph shown in the illustration has three components. A vertex with no incident edges is itself a component. A graph that is itself connected has exactly one component, consisting of the whole graph. Components are also sometimes called **connected components**.

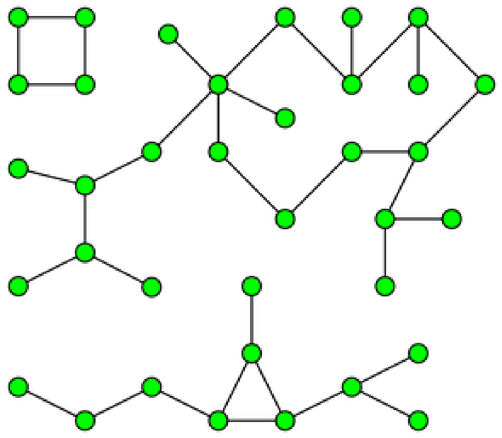


Figure 8: A Graph with Three Components

Equivalence Relations

I. **Reflexive:** There is a trivial path of length zero from any vertex to itself.

II. **Symmetric:** If there is a path from u to v, the same edges form a path from v to u.

III. **Transitive:** If there is a path from u to $v$and a path from $vtow$, the two paths may be concatenated together to form a path from u to w.

5.6 Cuts

A **cut** is a partition of the vertices of a graph into two **disjoint** subsets. Any cut determines a cut-set, the set of edges that have one endpoint in each subset of the partition. These edges are said to cross the cut. In a connected graph, each cut-set determines a unique cut, and in some cases cuts are identified with their cut-sets rather than with their vertex partitions.

Disjoint Def. Two sets are said to be disjoint sets if they have no element in common. Equivalently, two disjoint sets are sets whose intersection is the empty set.
Lecture 13:, Pages (13 - 28)

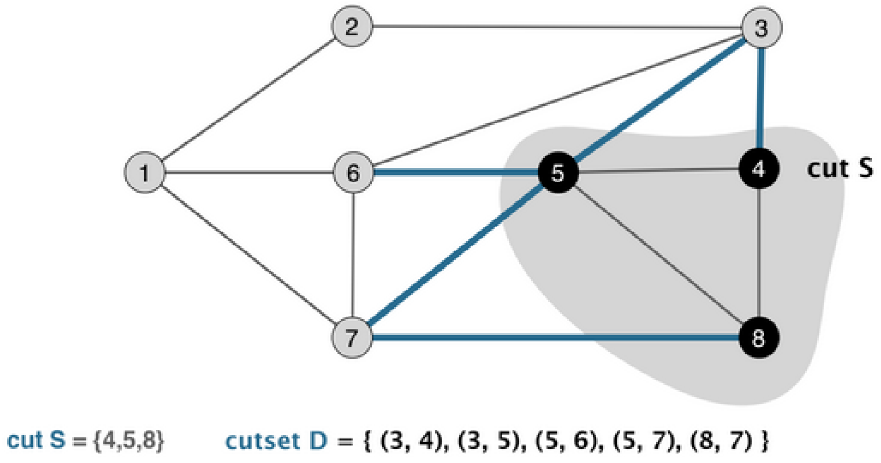


Figure 9: Depiction of a Graph and it's Cut-set

A **cut** $C = (S, T)$ is a partition of V of a graph $G = (V, E)$ into two subsets S and T. The **cut-set** of a cut $C = (S, T)$ is the set of edges that have one endpoint in S and the other endpoint in T. In other words, if s and t are specified vertices of the graph G, then an **s-t cut** is a cut in which s belongs to the set S and t belongs to the set T.

MST Cut Property. Let S be a subset of nodes. Let e be the minimum weight edge with exactly one end in S. Then e is in the **MST**.

MST Cycle Property. Let C be a cycle and let f be the maximum weight edge in C. Then f is *not* in the **MST**.

5.7 Undirected Graph

A graph (sometimes called **undirected graph** for distinguishing from a directed graph, or simple graph for distinguishing from a multigraph) is a pair $G = (V, E)$, where V is a set whose elements are called vertices (singular: vertex), and E is a set of paired vertices, whose elements are called edges (sometimes links or lines). In **Layman's Terms**,

I. An undirected graph is a set of nodes and a set of links between the nodes.

II. Each node is called a **vertex**, each link is called an **edge**, and each edge connects two vertices.

III. The order of the two connected vertices is unimportant.

IV. An undirected graph is a finite set of vertices together with a finite set of edges. Both sets might be empty, which is called the **empty graph**.

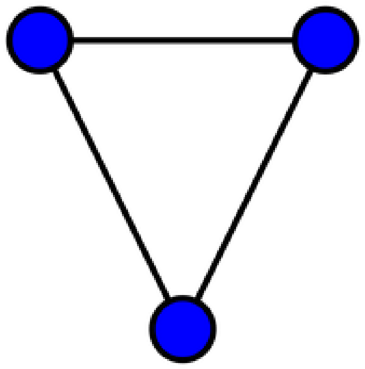


Figure 10: A Undirected Graph with 3 Vertices and 3 Edges

5.8 Directed Graph

A **directed graph** (or digraph) is a graph that is made up of a set of vertices connected by directed edges often called arcs. In formal terms, a directed graph is an ordered pair $G = (V, A)$ where,

I. V is a set whose elements are called vertices, nodes, or points.

II. A is a set of ordered pairs of vertices, called arcs, directed edges (sometimes simply edges with the corresponding set named E instead of A), arrows, or directed lines.

In **Layman's Terms**,

I. A directed graph is a finite set of vertices together with a finite set of edges. Both sets might be empty, which is called the empty graph.

II. Each edges is associated with two vertices, called its **source** and **target** vertices.

III. We say that the edge connects its source to its target.

IV. The **order** of the two connected vertices is important.

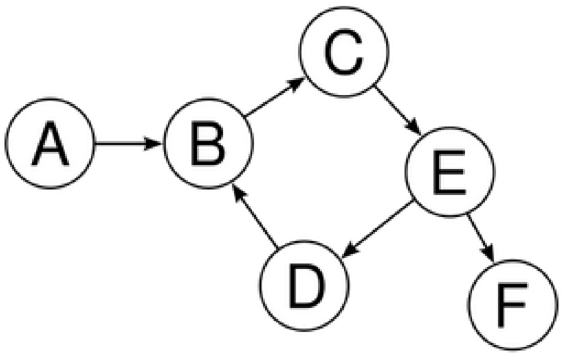


Figure 11: A Directed Graph with 6 Vertices and 6 Edges

5.9 Bipartite Graph

A **bipartite graph** (or bigraph) is a graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V. Vertex sets U and V are usually called the parts of the graph. Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles.

Undirected Def. An undirected graph $G = (V, E)$ is bipartite if the nodes can be separated into two disjoint sets U and W such that any edge in E connects a node in U to a node in W.

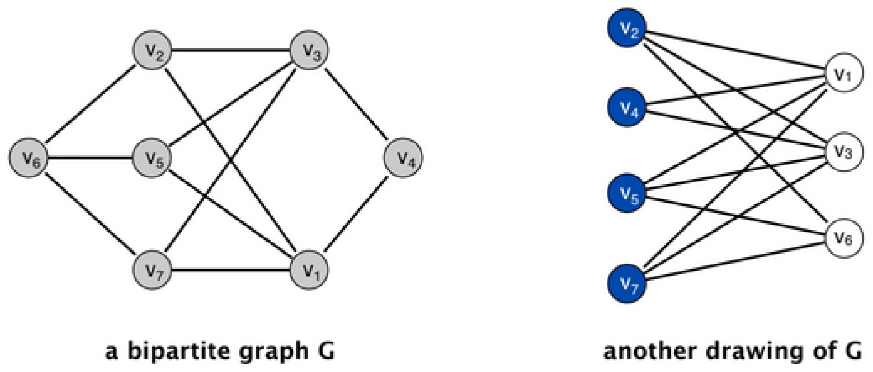


Figure 12: Depiction of a Colored Bipartite Graph

The two sets U and V may be thought of as a coloring of the graph with two colors: if one colors all nodes in U blue, and all nodes in V green, each edge has endpoints of differing colors, as is required in the graph coloring problem.

5.10 Directed Acyclic Graph

A **directed acyclic graph (DAG)** is a directed graph with **no directed cycles**. That is, it consists of vertices and edges (also called arcs), with each edge directed from one vertex to another, such that following those directions will never form a closed loop. A directed graph is a DAG if and only if it can be **topologically ordered**, by arranging the vertices as a linear ordering that is consistent with all edge directions.

Def. A DAG is a directed graph that contains no directed cycles.

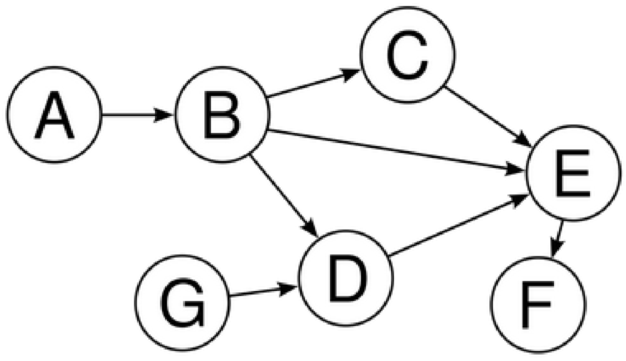


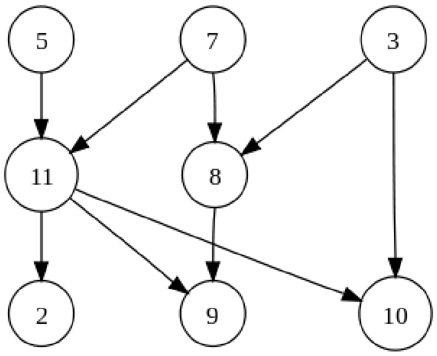
Figure 13: A Directed Acyclic Graph with 7 Vertices and 8 Edges

5.11 Topological Ordering

A topological sort or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge u, v from vertex u to vertex v, u comes before v in the ordering. A topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a **directed acyclic graph (DAG)**. Any DAG has at least one topological ordering.

Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.

Homework 6: Problem 1
Lecture 8: All Pages



The graph shown to the left has many valid topological sorts, including:

- 5, 7, 3, 11, 8, 2, 9, 10 (visual top-to-bottom, left-to-right)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

Figure 14: Depiction of a Topological Ordering with a DAG

5.12 Strongly Connected Graph

A graph is said to be **strongly connected** if every vertex is reachable from every other vertex. The strongly connected components of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. Formally, a directed graph $G = (V, E)$ such that for all pairs of vertices $u, v \in V$, there is a path from u to v and from v to u. In **Layman's Terms**,

I. A directed graph that has a path from each vertex to every other vertex.

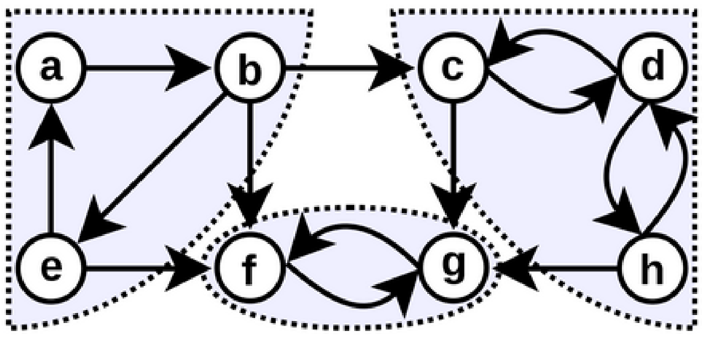


Figure 15: Depiction of a Topological Ordering with a DAG

5.13 Spanning Tree

A **spanning tree** T of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G. In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree. If all of the edges of G are also edges of a spanning tree T of G, then G is a tree and is identical to T. In **Layman's Terms**,

I. A spanning tree T of G is a **connected** tree that contains all vertices V and a subset of the edges E.

II. A graph G has multiple spanning trees, a spanning tree **always** has $n - 1$ edges.

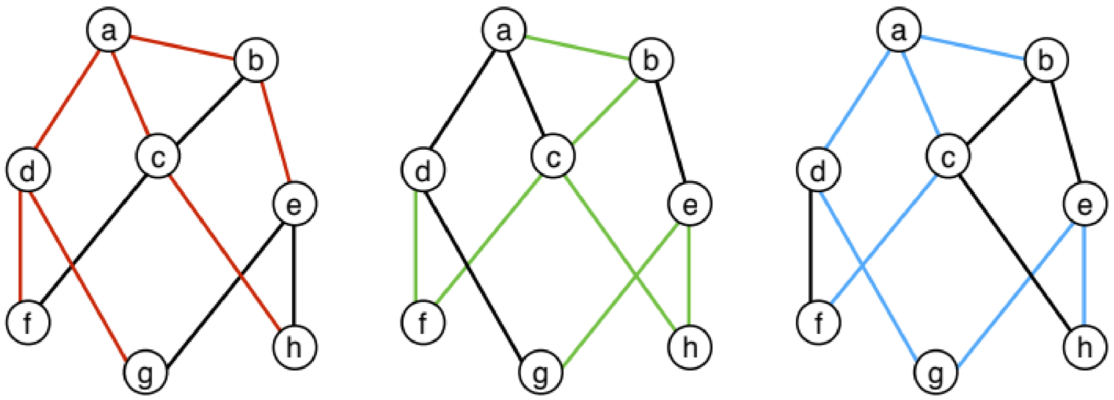
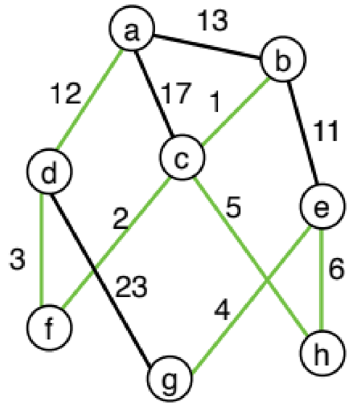


Figure 16: Depiction of Multiple Spanning Trees Derived from G

5.14 Minimum Spanning Tree

A **minimum spanning tree (MST)** or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as **small** as possible.



- there is no restriction on positive or negative edge weights.
- there is no dedicated source.
- assumption: all edge weights are distinct

Figure 17: Depiction of an MST from Figure 15

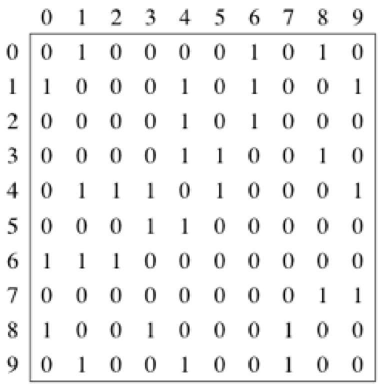
6 Graph Representation

There are several ways to represent graphs, each with its advantages and disadvantages.

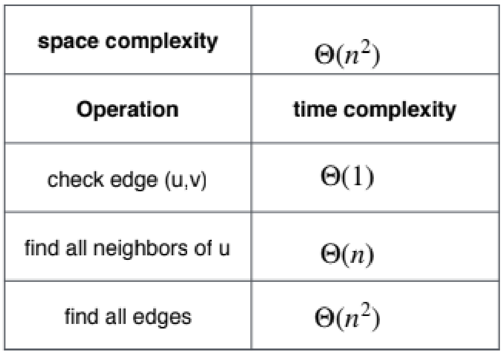
6.1 Adjacency Matrix

An **adjacency matrix** is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. For a graph with $|V|$ vertices, an adjacency matrix is a $|V| \times |V|$ matrix of 0s and 1s, where the entry in row i and column j is 1 if and only if the edge (i, j) is in the graph.

Def. A $n \times n$ binary matrix A, such that $A[i, j] = 1$ iff (i, j) is an edge.



	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0



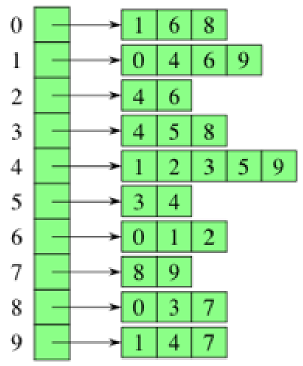
space complexity	$\Theta(n^2)$
Operation	**time complexity**
check edge (u,v)	$\Theta(1)$
find all neighbors of u	$\Theta(n)$
find all edges	$\Theta(n^2)$

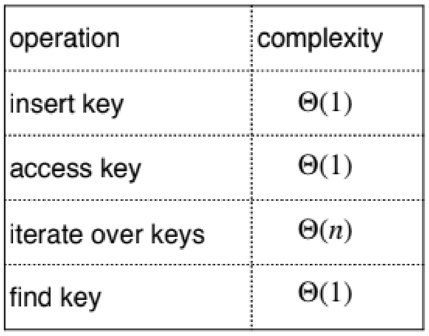
Figure 18: Depiction of an Adjacency Matrix and Complexity

6.2 Adjacency List

An **adjacency list** is a collection of unordered lists used to represent a finite graph. Each unordered list within an adjacency list describes the set of neighbors of a particular vertex in the graph. Combines **adjacency matrices** with edge lists. For each vertex i, store an array of vertices adjacent to it.

Def. For each node v there is a record containing a list of nodes to which v has edges.





operation	complexity
insert key	$\Theta(1)$
access key	$\Theta(1)$
iterate over keys	$\Theta(n)$
find key	$\Theta(1)$

Figure 19: Depiction of an Adjacency List and Complexity

7 Breadth-First Search Algorithm

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

7.1 Pseudocode

Algorithm 3: BFS(G,s)

```
Input: $G$ is a graph $G = (V, E)$, $s$ is a starting vertex of $G$
let $Q$ be a queue
label $s$ as explored
$Q$.enqueue($s$)
while $Q$ is not empty do
    $v = Q$.dequeue()
    if $v$ is the goal then
        return $v$
    forall edges from $v$ to $w$ in $G$.adjacentEdges($v$) do
        if $w$ is not labeled as explored then
            label $w$ as explored
            $Q$.enqueue($w$)
Output: Traverse all the nodes in graph $G$ (if it is connected)
```

7.2 Performance

Worst-Case Performance: $O(|V| + |E|)$

Worst-Case Space Complexity: $O(|V|)$

7.3 Use Case

BFS finds the shortest paths from a given source vertex to all other vertices, in terms of the number of edges in the paths. The algorithm is used in the famous **Dijkstra's Algorithm** for computing the shortest path. Additional uses include,

I. Best approach if the depth of the tree varies, and only part of the tree needs to be searched.

II. Finding all nodes within one connected component. **i.e.** Traversing all **reachable** nodes.

III. Testing if a given graph is **Bipartite** or not.

7.4 Appearances

Homeworks:
Homework 4: Problem 1, **Problem 2:** Part (b).

Lectures:
Lecture 5: Pages (11 - 21).
Lecture 6: Pages (14 - 26).

8 Depth-First Search Algorithm

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

8.1 Pseudocode

Algorithm 4: DFS(G,s)

Input: G is a graph $G = (V, E)$, s is a starting vertex of G

```
let $S$ be a stack
$S$.push($s$)
while $S$ is not empty do
    $v = S$.pop()
    if $v$ is not labeled as discovered then
        label $v$ as discovered
        forall edges from $v$ to $w$ in $G$.adjacentEdges($v$) do
            $S$.push($w$)
```

Output: Traverse all the nodes in graph G (if it is connected)

8.2 Performance

Worst-Case Performance: $O(|V| + |E|)$

Worst-Case Space Complexity: $O(|V|)$

8.3 Use Case

Depth-first searches are often used in simulations of games (and game-like situations in the real world). In a typical game you can choose one of several possible actions. Each choice leads to further choices, each of which leads to further choices, and so on into an ever-expanding tree-shaped graph of possibilities. Additionally, DSF is used in **topological sorting**, scheduling problems, cycle detection in graphs, and solving puzzles with only one solution, such as a maze or a sudoku puzzle.

I. Commonly used when an entire tree needs to be searched. It's easier to implement (using recursion) than BFS, and requires less space.

II. Provides a **spanning tree** when performed on an connected and undirected graph.

III. **Cycle** detection in both an undirected graph or directed graph.

IV. Used in tree-traversal algorithms, also known as tree searches.

8.4 Appearances

Homeworks:

Homework 4: Problem 1.
Homework 5: Problem 1, Part (b).

Lectures:

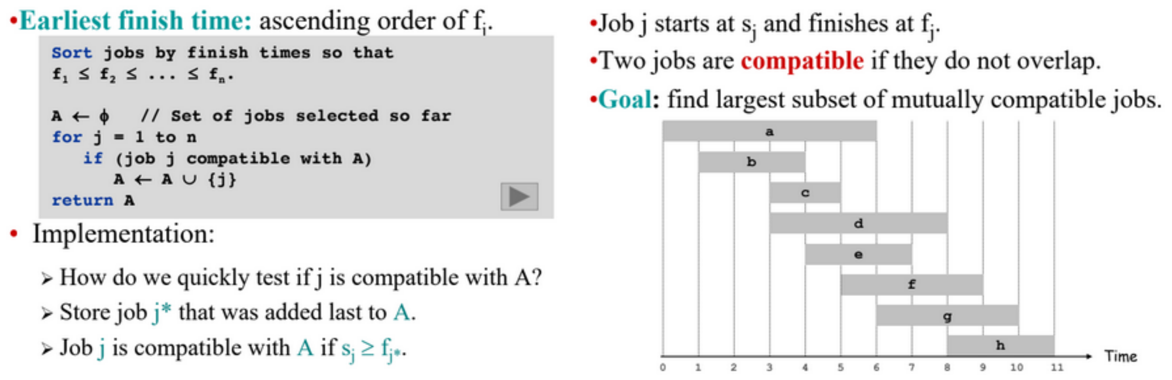
Lecture 7: Pages (26 - 55).

9 Greedy Algorithms

A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

9.1 Interval Scheduling

Interval scheduling is a class of problems in computer science, particularly in the area of algorithm design. In this case, the Greedy Stays Ahead technique is used. That is, $P(r)$: There is an optimal solution that agrees with the greedy solution in the first r jobs.



- Earliest finish time: ascending order of f_i.

```
Sort jobs by finish times so that
f₁ ≤ f₂ ≤ ... ≤ fₙ.

A ← ϕ    // Set of jobs selected so far
for j = 1 to n
   if (job j compatible with A)
      A ← A ∪ {j}
return A
```

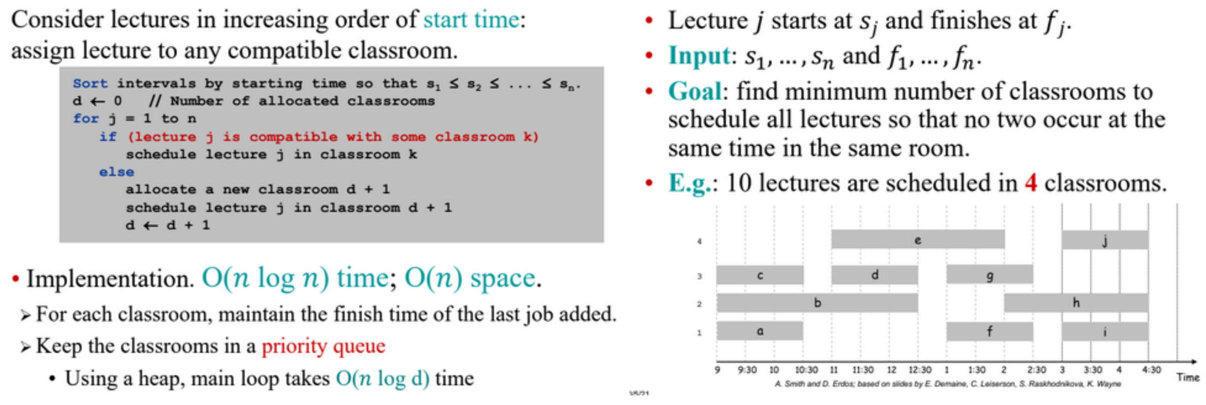
- Implementation:
  - How do we quickly test if j is compatible with A?
  - Store job j* that was added last to A.
  - Job j is compatible with A if $s_j \geq f_{j*}$.

- Job j starts at s_j and finishes at f_j.
- Two jobs are **compatible** if they do not overlap.
- **Goal:** find largest subset of mutually compatible jobs.

Figure 20: Depiction of Interval Scheduling and Implementation

9.2 Interval Partitioning

There are n lectures to be schedules and there are certain number of classrooms. Each lecture has a start time s_i and finish time f_i. The task is to schedule all lectures in **minimum** number of classes and there cannot be more than one lecture in a classroom at a given point of time.



Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s₁ ≤ s₂ ≤ ... ≤ sₙ.
d ← 0    // Number of allocated classrooms
for j = 1 to n
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
```

- Implementation. $O(n \log n)$ time; $O(n)$ space.
  - For each classroom, maintain the finish time of the last job added.
  - Keep the classrooms in a priority queue
    - Using a heap, main loop takes $O(n \log d)$ time

- Lecture j starts at s_j and finishes at f_j.
- **Input**: $s_1, \ldots, s_n$ and $f_1, \ldots, f_n$.
- **Goal**: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.
- **E.g.**: 10 lectures are scheduled in **4** classrooms.

4
3
2
1
e
j
c
d
g
b
h
a
f
i
9
9:30
10
10:30
11
11:30
12
12:30
1
1:30
2
2:30
3
3:30
4
4:30
Time
A. Smith and D. Erdos; based on slides by E. Demaine, C. Leiserson, S. Raskhodnikova, K. Wayne

Figure 21: Depiction of Interval Partitioning and Implementation

10 Advanced Data Structures

A **data structure** is a data organization, management, and storage format that enables efficient access and modification. Defined below are some advanced structures implemented in this course.

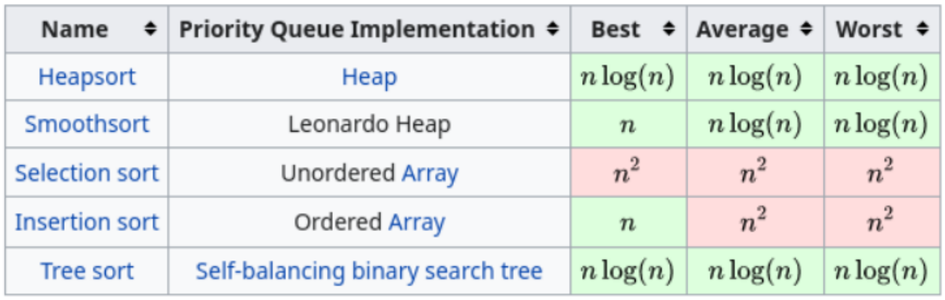
10.1 Priority Queue

A **priority queue** is an abstract data type similar to a regular queue or stack data structure in which each element additionally has a ”priority” associated with it. In a priority queue, an element with high priority is served before an element with low priority. A priority queue must at least support the following operations.

I. **is_empty:** Check whether the queue has no elements.

II. **insert_with_priority:** Add an element to the queue with an associated priority.

III. **pull_highest_priority_element:** Remove the element from the queue that has the highest priority, and return it.



Name	Priority Queue Implementation	Best	Average	Worst
Heapsort	Heap	$n\log(n)$	$n\log(n)$	$n\log(n)$
Smoothsort	Leonardo Heap	n	$n\log(n)$	$n\log(n)$
Selection sort	Unordered Array	n^2	n^2	n^2
Insertion sort	Ordered Array	n	n^2	n^2
Tree sort	Self-balancing binary search tree	$n\log(n)$	$n\log(n)$	$n\log(n)$

Figure 22: Time Complexity of Priority Queue Implementations

10.2 Binary Heap

A **binary heap** is a heap data structure that takes the form of a binary tree. Binary heaps are a common way of implementing priority queues. It has the following properties.

I. It’s a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible).

II. A Binary Heap is either **Min Heap** or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree.

This guide will focus on a **min-heap** implementation. A min-heap is a binary tree such that, the data contained in each node is less than (or equal to) the data in that node’s children, and the binary tree is complete.

Properties:

I. If node u is a parent of v, then $key(u) \leq key(v)$.

II. u may have multiple children, the heap-property doesn't imply a specific order among them.

III. It follows from the heap property that the minimum key is the root.

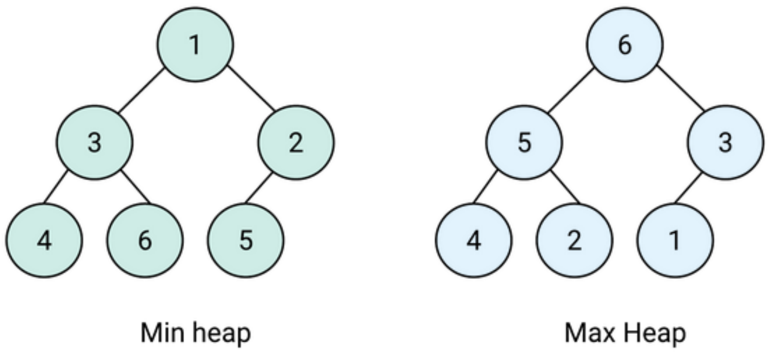
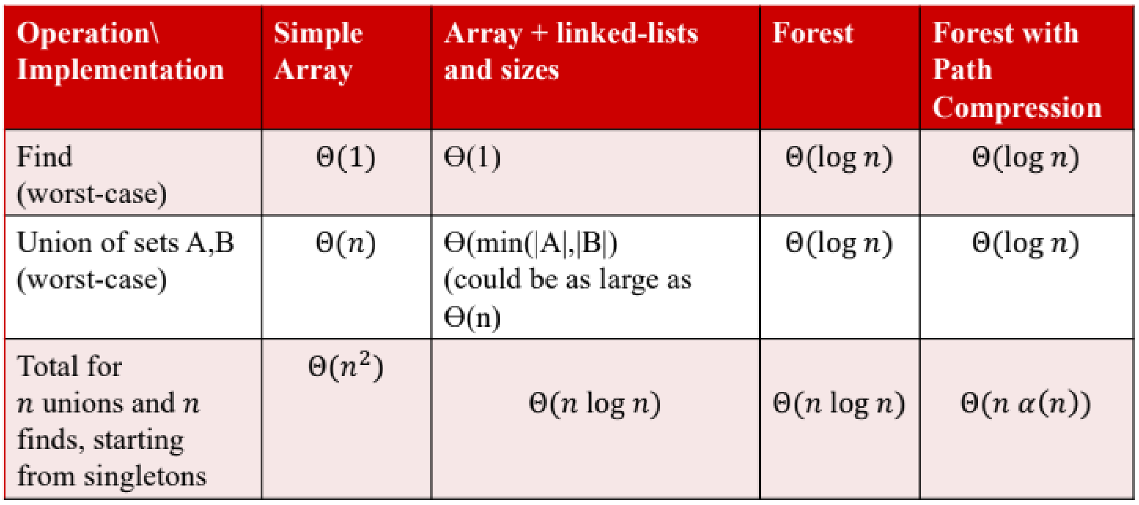


Figure 23: Depiction of a Min-Heap and Max-Heap

10.3 Union-find

A **union–find** data structure or merge–find set, is a data structure that stores a collection of disjoint (non-overlapping) sets. Equivalently, it stores a partition of a set into disjoint subsets. It provides operations for adding new sets, merging sets (replacing them by their union), and finding a representative member of a set. The last operation allows to find out efficiently if any two elements are in the same or different sets.



Operation\ Implementation	Simple Array	Array + linked-lists and sizes	Forest	Forest with Path Compression
Find (worst-case)	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
Union of sets A,B (worst-case)	$\Theta(n)$	$\Theta(\min(\lvert A\rvert,\lvert B\rvert)$ (could be as large as $\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
Total for n unions and n finds, starting from singletons	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n\ \alpha(n))$

Figure 24: Time Complexity of Union-Find Data Structure

11 Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm for finding the shortest paths between nodes in a graph. The algorithm exists in many variants. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

11.1 Pseudocode

Algorithm 5: Dijkstra(G,s)

```
Input: $G$ is a graph $G = (V, E)$, $s$ is the source vertex of $G$
create vertex set $Q$
foreach vertex $v$ in $G$ do
    dist[$v$] = INFINITY
    prev[$v$] = UNDEFINED
    add $v$ to $Q$
dist[$s$] = 0
while $Q$ is not empty do
    $u$ = vertex in $Q$ with min dist[$u$]
    remove $u$ from $Q$
    foreach neighbor $v$ of $u$ still in $Q$ do
        if dist[$v$] > dist[$u$] + length($u, v$) then
            dist[$v$] = dist[$u$] + length($u, v$)
            prev[$v$] = $u$
Output: dist[] is a list of distances from $s$ to other vertices, prev[] is a list of prior nodes
```

11.2 Performance

Worst-Case Performance: $O(|E|\log(|V|))$

Worst-Case Space Complexity: $O(|V|)$

11.3 Use Case

Dijkstra's algorithm is one of the most popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs **i.e.**, it is to find the shortest distance between two vertices on a graph. It's often used in digital mapping services, and social networking. Additionally, the algorithms efficiency can be improved if a **priority-queue** is implemented.

11.4 Appearances

Homeworks:

Homework 7: Problem 1.

Lectures:

Lecture 11: Pages (15 - 36), (41 - 46).

Lecture 12: Pages (2 - 7), (11 - 20).

12 Prim's Algorithm

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected 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 operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

12.1 Pseudocode

Algorithm 6: Prim(G, w, s)

```
Input: $G$ is a graph $G = (V, E)$, $w$ is the weights of $G$, $s$ is the source vertex of $G$
foreach $u \in G.V$ do
    $u$.key = INFINITY
    $u.\pi$ = NULL
$s$.key = 0
$Q = G.V$
while $Q$ != $\emptyset$ do
    $u$ = EXTRACT-MIN($Q$)
    foreach $v \in G.Adj[u]$ do
        if $v \in Q$ and $w(u,v) < v.key$ then
            $v.\pi = u$
            $v$.key = $w(u, v)$
Output: $T$ is a spanning tree (set of edges included in the MST), total weight $w_{min}$
```

12.2 Performance

Worst-Case Performance: $O(|E|\log(|V|))$

Worst-Case Space Complexity: $O(|V| + |E|)$

12.3 Use Case

The advantage of **Prim's algorithm** is its complexity, which is better than **Kruskal's algorithm**. Therefore, Prim's algorithm is helpful when dealing with dense graphs that have lots of edges. However, Prim's algorithm doesn't allow us much control over the chosen edges when multiple edges with the same weight occur. The reason is that only the edges discovered so far are stored inside the queue, rather than all the edges like in Kruskal's algorithm. Also, unlike Kruskal's algorithm, Prim's algorithm is a little harder to implement.

Overall: Prim's algorithm offers better time complexity.

12.4 Appearances

Lectures:

Lecture 12: Pages (27 - 30).
Lecture 13: Pages (27 - 28).

13 Kruskal's Algorithm

13.1 Pseudocode

Algorithm 7: Kruskal(G, w)

```
Input: $G$ is a graph $G = (V, E, w)$, $w$ is the weights of $G$
1 $T$ = NULL
2 $sorted$ = edges sorted by weight
3 for $u$ in $V$ do
4     make a set containing singleton $u$
5 for edge $(u,v)$ in sorted do
6     if $u$ and $v$ are in different sets then
7         $T = T + \{(u, v)\}$
8         merge the sets containing $u$ and $v$
Output: $T$ is a spanning tree (set of edges included in the MST), total weight $w_{min}$
```

13.2 Performance

Worst-Case Performance: $O(|E| \log(|V|))$

Worst-Case Space Complexity: $O(|V| + |E|)$

13.3 Use Case

Kruskal's algorithm is better used with sparse graphs, where we don't have lots of edges. However, we are examining all edges one by one sorted on ascending order based on their weight, this allows us great control over the resulting MST. Since different MSTs come from different edges with the same cost, in Kruskal's algorithm, all these edges are located one after another when sorted. Therefore, when two or more edges have the same weight, we have total freedom on how to order them. The order we use affects the resulting MST. Of course, the cost will always be the same regardless of the order of edges with the same weight. However, the edges we add to mst might be different. Another aspect to consider is that Kruskal's algorithm is **fairly easy to implement**. The only restrictions are having a good disjoint set data structure and a good sort function.

Overall: Kruskal's algorithm is better to use regarding the easier implementation and best control over the resulting **MST**.

13.4 Appearances

Lectures:

Lecture 12: Page (31).
Lecture 13: Pages (29 - 34).
Lecture 14: Pages (6 - 13).