Algorithms and Data Structures

Christian J. Rudder

October 2024

Contents

This page is left intentionally blank.

Preface

These notes are based on the lecture slides from the course:
BU CS330: Introduction to Analysis of Algorithms

Presented by:
Dora Erdos, Adam Smith
With contributions from:
S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne

Please note: These are my personal notes, and while I strive for accuracy, there may be errors. I encourage you to refer to the original slides for precise information.
Comments and suggestions for improvement are always welcome.

Prerequisites

Theorem 0.1: Common Derivatives

Power Rule: For $n \neq 0$	$\frac{d}{dx}(x^n) = n \cdot x^{n-1}$. E.g., $\frac{d}{dx}(x^2) = 2x$
Derivative of a Constant:	$\frac{d}{dx}(c) = 0$. E.g., $\frac{d}{dx}(5) = 0$
Derivative of ln:	$\frac{d}{dx}(\ln x) = \frac{1}{x}$
Derivative of $\log_a$:	$\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a}$
Derivative of $\sqrt{x}$:	$\frac{d}{dx}(\sqrt{x}) = \frac{1}{2\sqrt{x}}$
Derivative of function $f(x)$:	$\frac{d}{dx}(x) = 1$. E.g., $\frac{d}{dx}(5x) = 5$
Derivative of the Exponential Function:	$\frac{d}{dx}(e^x) = e^x$

Theorem 0.2: L'Hopital's Rule

Let $f(x)$ and $g(x)$ be two functions. If $\lim_{x\to a} f(x) = 0$ and $\lim_{x\to a} g(x) = 0$, or $\lim_{x\to a} f(x) = \pm\infty$ and $\lim_{x\to a} g(x) = \pm\infty$, then:

$$\lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$$

Where $f'(x)$ and $g'(x)$ are the derivatives of $f(x)$ and $g(x)$ respectively.

Theorem 0.3: Exponents Rules

For $a, b, x \in \mathbb{R}$, we have:

$$x^a \cdot x^b = x^{a+b} \text{ and } (x^a)^b = x^{ab}$$

$$x^a \cdot y^a = (xy)^a \text{ and } \frac{x^a}{y^a} = \left(\frac{x}{y}\right)^a$$

Note: The $:=$ symbol is short for "is defined as." For example, $x := y$ means x is defined as y.

Definition 0.1: Logarithm

Let $a, x \in \mathbb{R}$, $a > 0$, $a \neq 1$. Logarithm x base a is denoted as $\log_a(x)$, and is defined as:

$$\log_a(x) = y \iff a^y = x$$

Meaning *log* is inverse of the exponential function, i.e., $\log_a(x) := (a^y)^{-1}$.

Tip: To remember the order $log_a(x) = a^y$, think, "base a," as a is the base of our *log* and y.

Theorem 0.4: Logarithm Rules

For $a, b, x \in \mathbb{R}$, we have:

$$\log_a(x) + \log_a(y) = \log_a(xy) \text{ and } \log_a(x) - \log_a(y) = \log_a\left(\frac{x}{y}\right)$$

$$\log_a(x^b) = b \log_a(x) \text{ and } \log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

Definition 0.2: Permutations

Let $n \in \mathbb{Z}^+$. Then the number of distinct ways to arrange n objects in order is $n! := n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. When we choose r objects from n objects, it's Denoted:

$$^nP_r := \frac{n!}{(n-r)!}$$

Where $P(n, r)$ is read as "n permute r."

Definition 0.3: Combinations

Let n and k be positive integers. Where order doesn't matter, the number of distinct ways to choose k objects from n objects is it's *combination*. Denoted:

$$\binom{n}{k} := \frac{n!}{k!(n-k)!}$$

Where $\binom{n}{k}$ is read as "n choose k.", and $\binom{\cdot}{\cdot}$, the *binomial coefficient*.

Theorem 0.5: Binomial Theorem

Let a and b be real numbers, and n a non-negative integer. The binomial expansion of $(a+b)^n$ is given by:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

which expands explicitly as:

$$(a+b)^n = \binom{n}{0} a^n + \binom{n}{1} a^{n-1} b + \binom{n}{2} a^{n-2} b^2 + \cdots + \binom{n}{n-1} a b^{n-1} + \binom{n}{n} b^n$$

where $\binom{n}{k}$ represents the binomial coefficient, defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

for $0 \leq k \leq n$.

Theorem 0.6: Binomial Expansion of 2^n

For any non-negative integer n, the following identity holds:

$$2^n = \sum_{i=0}^{n} \binom{n}{i} = (1+1)^n.$$

Definition 0.4: Well-Ordering Principle

Every non-empty set of positive integers has a least element.

Definition 0.5: "Without Loss of Generality"

A phrase that indicates that the proceeding logic also applies to the other cases. i.e., For a proposition not to lose the assumption that it works other ways as well.

Theorem 0.7: Pigeon Hole Principle

Let $n, m \in \mathbb{Z}^+$ with $n < m$. Then if we distribute m pigeons into n pigeonholes, there must be at least one pigeonhole with more than one pigeon.

Theorem 0.8: Growth Rate Comparisons

Let n be a positive integer. The following inequalities show the growth rate of some common functions in increasing order:

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

These inequalities indicate that as n grows larger, each function on the right-hand side grows faster than the ones to its left.

— 1 —

Complexity Theory

1.1 Asymptotic Notation

Asymptotic analysis is a method for describing the limiting behavior of functions as inputs grow infinitely.

Definition 1.1: Asymptotic

Let $f(n)$ and $g(n)$ be two functions. As n grows, if $f(n)$ grows closer to $g(n)$ never reaching, we say that "$f(n)$ is **asymptotic** to $g(n)$."

We call the point where $f(n)$ starts behaving similarly to $g(n)$ the **threshold** n_0. After this point n_0, $f(n)$ follows the same general path as $g(n)$.

Definition 1.2: Big-O: (Upper Bound)

Let f and g be functions. $f(n)$ our function of interest, and $g(n)$ our function of comparison.

Then we say $f(n) = O(g(n))$, "$f(n)$ **is big-O of** $g(n)$," if $f(n)$ grows no faster than $g(n)$, up to a constant factor. Let n_0 be our asymptotic threshold. Then, for all $n \geq n_0$,

$$0 \leq f(n) \leq c \cdot g(n)$$

Represented as the ratio $\dfrac{f(n)}{g(n)} \leq c$ for all $n \geq n_0$. Analytically we write,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Meaning, as we chase infinity, our numerator grows slower than the denominator, bounded, never reaching infinity.

Examples:

(i.) $3n^2 + 2n + 1 = O(n^2)$

(ii.) $n^{100} = O(2^n)$

(iii.) $\log n = O(\sqrt{n})$

Proof 1.1: $\log n = O(\sqrt{n})$

We setup our ratio:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}}$$

Since $\log n$ and $\sqrt{n}$ grow infinitely without bound, they are of indeterminate form $\frac{\infty}{\infty}$. We apply L'Hopital's Rule, which states that taking derivatives of the numerator and denominator will yield an evaluateable limit:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{\frac{d}{dn}\log n}{\frac{d}{dn}\sqrt{n}}$$

Yielding derivatives, $\log n = \frac{1}{n}$ and $\sqrt{n} = \frac{1}{2\sqrt{n}}$. We substitute these back into our limit:

$$\lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = \lim_{n\to\infty} \frac{2\sqrt{n}}{n} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$$

Our limit approaches 0, as we have a constant factor in the numerator, and a growing denominator. Thus, $\log n = O(\sqrt{n})$, as $0 < \infty$. ■

Definition 1.3: Big-Ω: (Lower Bound)

The symbol Ω reads "Omega." Let f and g be functions. Then $f(n) = \Omega(g(n))$ if $f(n)$ grows no slower than $g(n)$, up to a constant factor. I.e., lower bounded by $g(n)$. Let n_0 be our asymptotic threshold. Then, for all $n \geq n_0$,

$$0 \leq c \cdot g(n) \leq f(n)$$

$$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

Meaning, as we chase infinity, our numerator grows faster than the denominator, approaching 0 asymptotically.

Examples: $n! = \Omega(2^n)$; $\frac{n}{100} = \Omega(n)$; $n^{3/2} = \Omega(\sqrt{n})$; $\sqrt{n} = \Omega(\log n)$

Definition 1.4: Big Θ: (Tight Bound)

The symbol Θ reads "Theta." Let f and g be functions. Then $f(n) = \Theta(g(n))$ if $f(n)$ grows at the same rate as $g(n)$, up to a constant factor. I.e., $f(n)$ is both upper and lower bounded by $g(n)$. Let n_0 be our asymptotic threshold, and $c_1 > 0, c_2 > 0$ be some constants. Then, for all $n \geq n_0$,

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

Meaning, as we chase infinity, our numerator grows at the same rate as the denominator.

Examples: $n^2 = \Theta(n^2)$; $2n^3 + 2n = \Theta(n^3)$; $\log n + \sqrt{n} = \Theta(\sqrt{n})$.

Definition 1.5: Little o: (Strict Upper Bound)

The symbol o reads "little-o." Let f and g be functions. Then $f(n) = o(g(n))$ if $f(n)$ grows strictly slower than $g(n)$, meaning $f(n)$ becomes insignificant compared to $g(n)$ as n grows large. Let n_0 be our asymptotic threshold. Then, for all $n \geq n_0$,

$$0 \leq f(n) < c \cdot g(n)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Meaning, as we chase infinity, the ratio of $f(n)$ to $g(n)$ shrinks to zero.

Examples: $n = o(n^2)$; $\log n = o(n)$; $n^{0.5} = o(n)$.

Definition 1.6: Little ω: (Strict Lower Bound)

The symbol ω reads "little-omega." Let f and g be functions. Then $f(n) = \omega(g(n))$ if $f(n)$ grows strictly faster than $g(n)$, meaning $g(n)$ becomes insignificant compared to $f(n)$ as n grows large. Let n_0 be our asymptotic threshold. Then, for all $n \geq n_0$,

$$0 \leq c \cdot g(n) < f(n)$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Meaning, as we chase infinity, the ratio of $g(n)$ to $f(n)$ shrinks to zero.

Examples: $n^2 = \omega(n)$; $n = \omega(\log n)$.

Definition 1.7: Asymptotic Equality ($\sim$)

The symbol $\sim$ reads "asymptotic equality." Let f and g be functions. Then $f(n) \sim g(n)$ if, as $n \to \infty$, the ratio of $f(n)$ to $g(n)$ approaches 1. I.e., the two functions grow at the same rate asymptotically. Formally,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

Meaning, as n grows large, the two functions become approximately equal.

Examples: $n + 100 \sim n$, $\log(n^2) \sim 2\log n$.

Tip: To review:

- **Big-O:** $f(n) < g(n)$ (Upper Bound); $f(n)$ grows no faster than $g(n)$.
- **Big-Ω:** $f(n) > g(n)$ (Lower Bound); $f(n)$ grows no slower than $g(n)$.
- **Big-Θ:** $f(n) = g(n)$ (Tight Bound); $f(n)$ grows at the same rate as $g(n)$.
- **Little-o:** $f(n) < g(n)$ (Strict Upper Bound); $f(n)$ grows strictly slower than $g(n)$.
- **Little-ω:** $f(n) > g(n)$ (Strict Lower Bound); $f(n)$ grows strictly faster than $g(n)$.
- **Asymptotic Equality:** $f(n) \sim g(n)$; $f(n)$ grows at the same rate as $g(n)$.

Theorem 1.1: Types of Asymptotic Behavior

The following are common relationships between different types of functions and their asymptotic growth rates:

- **Polynomials.** Let $f(n) = a_0 + a_1 n + \cdots + a_d n^d$ with $a_d > 0$. Then, $f(n)$ is $\Theta(n^d)$.
  E.e., $3n^2 + 2n + 1$ is $\Theta(n^2)$.
- **Logarithms.** $\Theta(\log_a n)$ is $\Theta(\log_b n)$ for any constants $a, b > 0$. That is, logarithmic functions in different bases have the same growth rate.
  E.g., $\log_2 n$ is $\Theta(\log_3 n)$.
- **Logarithms and Polynomials.** For every $d > 0$, $\log n$ is $O(n^d)$. This indicates that logarithms grow slower than any polynomial.
  E.g., $\log n$ is $O(n^2)$.
- **Exponentials and Polynomials.** For every $r > 1$ and every $d > 0$, n^d is $O(r^n)$. This means that exponentials grow faster than any polynomial.
  E.e., n^2 is $O(2^n)$.

1.2 Evaluating Algorithms

When analyzing algorithms, we are interested in two primary factors: time and space complexity.

Definition 2.1: Time Complexity

The **time complexity** of an algorithm is the amount of time it takes to run as a function of the input size. We use asymptotic notation to describe the time complexity of an algorithm.

Definition 2.2: Space Complexity

The **space complexity** of an algorithm is the amount of memory it uses to store inputs and subsequent variables during the algorithm's execution. We use asymptotic notation to describe the space complexity of an algorithm.

Below is an example of a function and its time and space complexity analysis.

Function 2.1: Arithmetic Series - `Fun1`(A)

Computes a result based on a length-n array of integers:

Input: A length-n array of integers.
Output: An integer p computed from the array elements.

```
1 Function Fun1(A):
2 |  p ← 0;
3 |  for i ← 1 to n − 1 do
4 |  |  for j ← i + 1 to n do
5 |  |  |  p ← p + A[i] · A[j];
6 |  |  end
7 |  end
8 return p
```

Time Complexity: For $f(n) :=$ `Fun1`(A), $f(n) = \frac{n^2}{2} = O(n^2)$. This is because the function has a nested loop structure, where the inner for-loop runs $n - i$ times, and the outer for-loop runs $n - 1$ times. Thus, the total number of iterations is $\sum_{i=1}^{n-1} n - i = \frac{n^2}{2}$.

Space Complexity: We yield $O(n)$ for storing an array of length n. The variable p is $O(1)$ (constant), as it is a single integer. Hence, $f(n) = n + 1 = O(n)$.

Additional Example: Let $f(n, m) = n^2m + m^3 + nm^3$. Then, $f(n, m) = O(n^2m + m^3)$. This is because both n and m must be accounted for. Our largest n term is n^2m, and our largest m term is m^3 both dominate the expression. Thus, $f(n, m) = O(n^2m + m^3)$.

— 2 —

Proving Algorithms

2.1 Stable Matchings

In proving the correctness of algorithms we introduce the stable matching problem. A combinatorial optimization problem that seeks to find the best possible matching between two sets of elements. When we say "best possible matching," we mean that the matching is stable, and that there is no other matching that is better.

Definition 1.1: Stable Matching

A matching is **stable** if there is no pair of elements that prefer each other over their current match.

Definition 1.2: Unstable Matching

A matching is **unstable** if there is a pair of elements that prefer each other over their current match.

I.e., in verifying a stable matching, if any one pair of elements switch partners, the matching is unstable. If no pairs swap, the matching is stable.

Scenario: *Lunch Time*

Imagine it's lunch time at elementary school, and a group of kids E ={Ena, Eda} swap lunches with A ={Ava, Adi}. They each have a list of preferences from favorite to least favorite. We visualize the following preferences:

E's Preference List

	1st	2nd
Ena	Ava	Adi
Eda	Ava	Adi

A's Preference List

	1st	2nd
Ava	Ena	Eda
Adi	Ena	Eda

Observe the following matchings:

(1.) Pairs, Ena-Ava, Eda-Adi swapped lunches.

E's Preference List

	1st	2nd
Ena	Ava	Adi
Eda	Ava	Adi

A's Preference List

	1st	2nd
Ava	Ena	Eda
Adi	Ena	Eda

This matching is **stable**. Ena and Ava prefer each other's lunches. Eda will ask Ava to trade, and Ava will refuse because she prefers Ena's lunch. Adi does the same with Ena, but they also refuse.

Tip: If it's hard to keep track who is who, here's a possible order to read in: Ena got Ava, and Ena is their 1st choice. Eda got Adi, and Eda is their 2nd choice.

Changing the preference tables,

(2.) Pairs, Ena-Adi, Eda-Ava swapped lunches.

E's Preference List

	1st	2nd
Ena	Ava	Adi
Eda	Adi	Ava

A's Preference List

	1st	2nd
Ava	Ena	Eda
Adi	Ena	Eda

This matching is **unstable**. Ena and Ava would rather eat each other's lunches.

Definition 1.3: Unique Stable Matching

A matching is **uniquely stable** if between two sets of elements, there is only one possible stable matching.

Example: If everyone uniquely prefers each other, there is only one stable matching.

(3.) Pairs, Ena-Ava, Eda-Adi swapped lunches.

E's Preference List

	1st	2nd
Ena	Ava	Adi
Eda	Adi	Ava

A's Preference List

	1st	2nd
Ava	Ena	Eda
Adi	Eda	Ena

This matching is a **unique stable matching**. If rather Ena-Adi and Eda-Ava (2nd-choice parings), then both pairs would end up swapping to their 1st-choice. **Notably**: for table sizes of $n \times n$, then $0 < n \leq 2$ **forces a unique stable matching.**

2.2 Gale-Shapley Algorithm

We will now introduce the Gale-Shapley algorithm, for which we will prove its correctness, time complexity, and space complexity.

Theorem 2.1: Gale-Shapley Algorithm

The **Gale-Shapley algorithm** is a method for finding a stable matching between two sets of elements. It is also known as the **Deferred Acceptance Algorithm**.

Algorithm: Given sets $E = e_1, \ldots, e_n$ and $A = a_1, \ldots, a_n$. Then find a stable matching:

(i.) Each $e_i \in E$ proposes to their most preferred a_j.

(ii.) For each $a_j \in A$:

(a.) If a_j is free, they accept the proposal.

(b.) If a_j is already matched, a_j either accepts or rejects. If a_j accepts, the previous match is broken.

Each e_i continues to propose to their next most preferred a_j until all e_i are matched.

Claims:

1. At least one stable matching is guaranteed.
2. Unless the table is unique, the proposing will always get their best choice unless it conflicts with another proposer.

First we will prove the correctness, then implement the algorithm and analyze its time and space complexity.

Proof 2.1: Gale-Shapley Algorithm Correctness

Claim 1: Suppose, for sake of contradiction, that some $a_j \in A$ is not matched upon termination of the algorithm. Then some $e_i \in E$ is also not matched assuming $|E| = |A|$. Then e_i must have not proposed to a_j, contradicting that e_i proposed to all elements of A. Thus, the program only terminates when all e_i are matched.

Claim 2: Suppose E proposes to A with unique first choices. Then all $a_i \in A$ must accept their first proposal. Now suppose $e_i, e_j \in E$ have a conflicting choice a_i. Then a_i gets their preference only in that case.

■

Function 2.1: Gale-Shapley Algorithm - `GS(E,A)`

Finds a stable matching between two sets of elements:

Input: Two sets, E and A, of equal size.
Output: A stable matching between E and A.

```
1  Function GS(E,A):
2      M ← ∅;
3      while there is some unmatched element in E do
4          e ← next unmatched element in E;
5          a ← next avalaible preferred choice of e;
6          if a is not yet matched then
7              match e and a;
8              add the pair (e, a) to M;
9          end
10         else
11             if a prefers e over their current match then
12                 match e and a, replacing the current match;
13                 update M accordingly;
14             end
15         end
16     end
17     return M;
18 return Matching M
```

Time Complexity: $O(n^2)$ time, where n is the number of elements in E and A. Worst-case, each element in E proposes to each element in A, i.e., $n \cdot n$ combinations to check.

Space Complexity: $O(n^2)$ space, where we store $|E| \cdot |A| = n \cdot n$ pairs.

— 3 —

Graphs and Trees

3.1 Paths and Connectivity

Graphs are similar to train networks or airline routes. They connect one location to another.

Definition 1.1: Graph

A **graph** is a collection of points, called **vertices** or **nodes**, connected by lines, called **edges**. Similarly to how a polygon has vertices connected by edges.

Definition 1.2: Undirected Graph

An **undirected graph** is a graph where the edges have no particular direction going both ways between nodes. A **degree** of a node is the number of edges connected to it.

Example: Figure (3.1) shows an undirected graph:

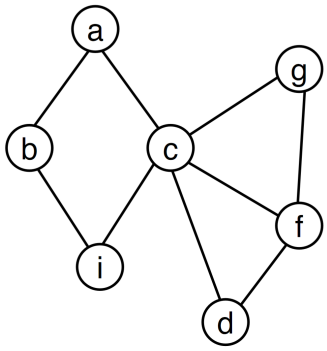


Figure 3.1: An undirected graph with 7 vertices and 9 edges.

Node a has a degree of 3, and node c has a degree of 4.

Definition 1.3: Directed Graph

A **directed graph** is where the edges have a specific direction from one node to another.

- The **indegree** of a node is the number of edges that point to it.
- The **outdegree** of a node is the number of edges that point from it.

Example: Figure (3.2) shows a directed graph:

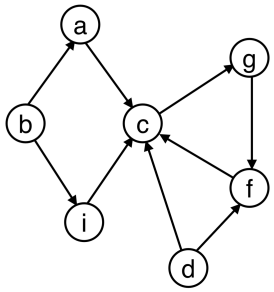


Figure 3.2: A directed graph with 7 vertices and 9 edges.

Node b has an outdegree of 2 and an indegree of 0. c has an indegree of 4 and an outdegree of 1.

Definition 1.4: Weighted Graph

A **weighted graph** is a graph where each edge has a numerical value assigned to it.

Example: Figure (3.3) shows a weighted graph:

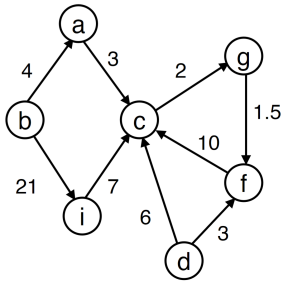


Figure 3.3: A weighted graph with 7 vertices and 9 edges.

Definition 1.5: Path

A **path** is a sequence of edges that connect a sequence of vertices. A path is **simple** if all nodes are distinct.

Example: In Figure (3.4), a simple path $h \leftrightarrow b \leftrightarrow i \leftrightarrow c \leftrightarrow d$ is shown:

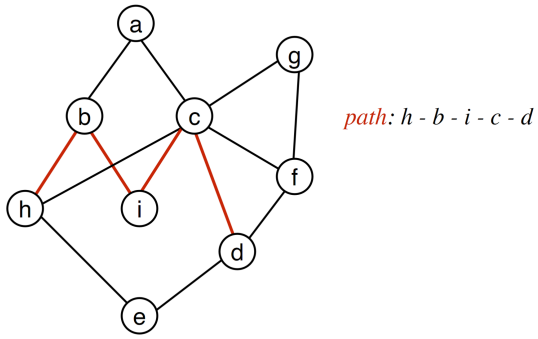


Figure 3.4: A graph with a simple path from h to d.

Definition 1.6: Connectivity

A graph is **connected** if there is a path between every pair of vertices.
A graph is **disconnected** if there are two vertices with no path between them.
Connected graphs of n nodes have at least $n - 1$ edges.

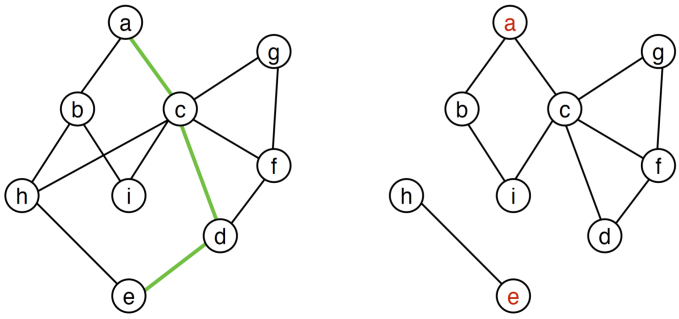


Figure 3.5: A connected graph $a \leftrightarrow c \leftrightarrow d \leftrightarrow e$ and disconnected graph.

Definition 1.7: Adjacency Matrix

An **adjacency matrix** is an $n \times n$ matrix where such that $A[i][j] = 1$ if there is an edge between nodes i and j.

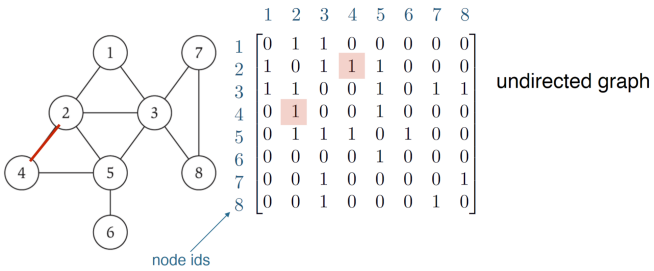


Figure 3.6: An adjacency matrix where the path $4 \leftrightarrow 2$ is highlighted ($A[2][4]$ or $A[4][2]$)

Theorem 1.1: Properties of Adjacency Matrix

The following properties hold for adjacency matrices:

- An undirected graph is symmetric about the diagonal.
- A directed graph is not symmetric about.
- A weighted graph has the weight of the edge instead of binary.

Space Complexity: $\Theta(n^2)$; **Time Complexities:**
Check edges $A[i][j]$**:** $\Theta(1)$; **List neighbors** $A[i]$: $\Theta(n)$; **List all edges:** $\Theta(n^2)$.

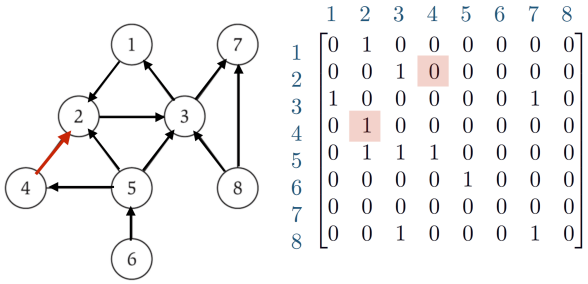


Figure 3.7: An adjacency matrix for a directed graph with path $4 \leftrightarrow 2$ highlighted ($A[2][4]$ and $A[4][2]$).

Definition 1.8: Adjacency List

An **adjacency list** is a list of keys where each key has a list of neighbors.

Often taking form as a dictionary or hash-table:
Space Complexity: $\Theta(n+m)$ for n nodes and m total edges; **Time Complexities:** **Check key:** $\Theta(1)$; **List neighbors:** $\Theta(|\text{outdegrees}|$ of key); **List all edges:** $\Theta(n+m)$; **Insert edge:** $\Theta(1)$.

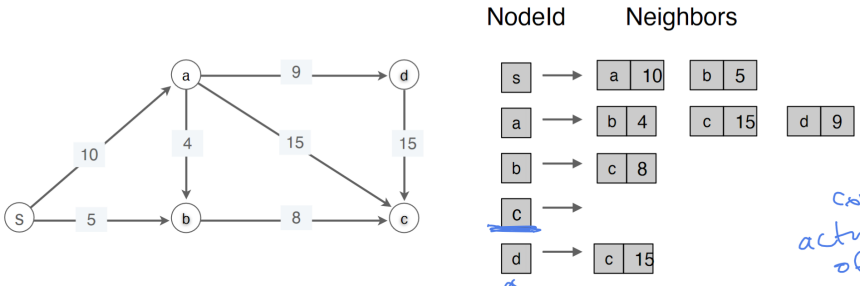


Figure 3.8: An adjacency list of a directed graph, c highlighted with no outdegrees.

3.2 Breath-First and Depth-First Search

Two general methods for traversing a graph are, **breadth-first search** and **depth-first search**.

Definition 2.1: Cycle

A **cycle** is a path that starts and ends at the same node.

Example: In the above Figure (3.7), $1 \to 2 \to 3 \to 1$ form a cycle.

Definition 2.2: Tree

A **tree** is a connected graph with no cycles. A **leaf-nodes** is the outer-most nodes of a tree. A **branch** is a path from the root to a leaf.

Theorem 2.1: Tree Identity

Let G be an undirected graph of n nodes. Then any two statements imply the third:

(i.) G is connected.

(ii.) G has $n-1$ edges.

(iii.) G has no cycles.

Definition 2.3: Rooted Trees

Let T be a tree with a designated root node r. Then each degree is considered an outdegree, called a **child**, and its indegree its **parent**.

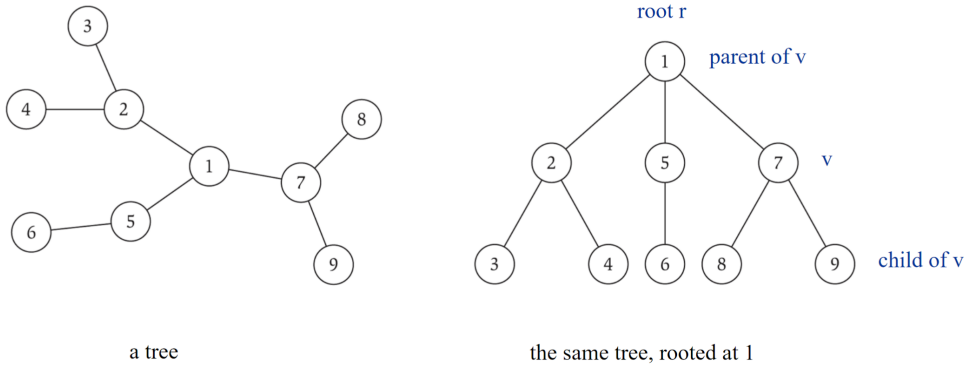


Figure 3.9: A rooted tree with root 1 and children $\{2, 5, 7\}$

Definition 2.4: Levels and Heights

The **level** of a node is the number of edges from the root. The **height** of a tree is the maximum level of any node.

Example: In Figure (3.9)'s rooted tree , node 5 has a level of 1, and the height of the tree is 2.

Definition 2.5: Breadth-First Search (BFS)

In a **breadth-first search**, we start at a node's children first before moving onto their children's children in level order.

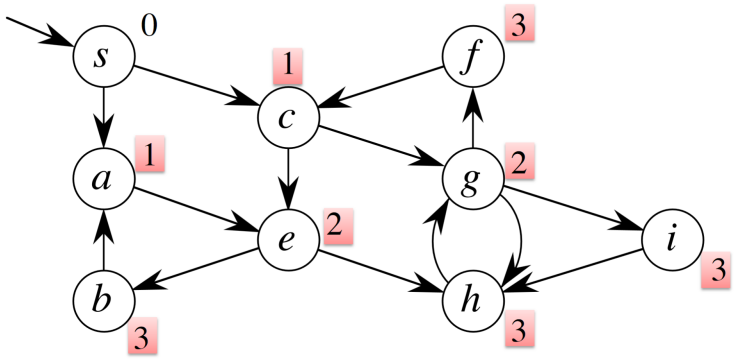


Figure 3.10: A BFS tree traversal preformed on a graph with each level enumerated

Theorem 2.2: Properties of BFS

BFS run on any graph T produces a tree T' with the following properties:

(i.) T' is a tree.

(ii.) T' is a rooted tree with the starting node as the root.

(iii.) The height of T' is the shortest path from the root to any node.

(iv.) Any sub-paths of T' are also shortest paths.

Proof 2.1: Proof of BFS

(i.) and (ii.) follow from the definition of a tree. (iii.) and (iv.) follow that since a tree contains a direct path to any given node in our parent child relationship, that path must be the shortest. ■

Tip: In a family tree, there is only one path from each ancestor to each descendant.

We create a BFS algorithm from what we know, though not the best implementation:

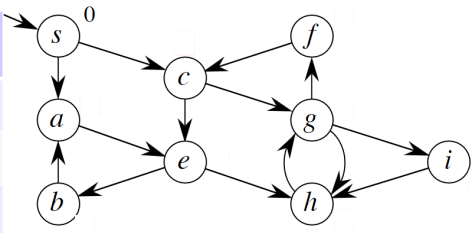
Function 2.1: BFS Algorithm - `BFS(s)`

Breadth-First Search starting from node s.

Input: Graph $G = (V, E)$ and starting node s.
Output: Levels of each vertex from s.

```
1  Function BFS(s):
2      for each v ∈ V do
3          Level[v] ← ∞;
4      end
5      Level[s] ← 0;
6      Add s to Q;
7      while Q not empty do
8          u ← Q.Dequeue();
9          for each v ∈ G[u] do
10             if Level[v] = ∞ then
11                 Add edge (u, v) to tree T (parent[v] = u);
12                 Add v to Q;
13                 Level[v] ← Level[u] + 1;
14             end
15         end
16     end
```

u	Queue contents before exploring u	… after exploring u
s	(empty)	*[a,c]*
a	*[c]*	*[c,e]*
c	*[e]*	*[e,g]*
e	*[g]*	*[g,b,h]*
g	*[b,h]*	*[b,h,f,i]*
b	*[h,f,i]*	*[h,f,i]*
h	*[f,i]*	*[f,i]*
f	*[i]*	*[i]*
i	(empty)	(empty)



Loop invariant: If the first node in the queue has level i, then the queue consists of nodes of level i possibly followed by nodes of level $i + 1$.

Consequence: Nodes are explored in increasing order of level.

Figure 3.11: A table showing the queue at each level of iteration

We analyze the time and space complexity in the below Figure (3.12):

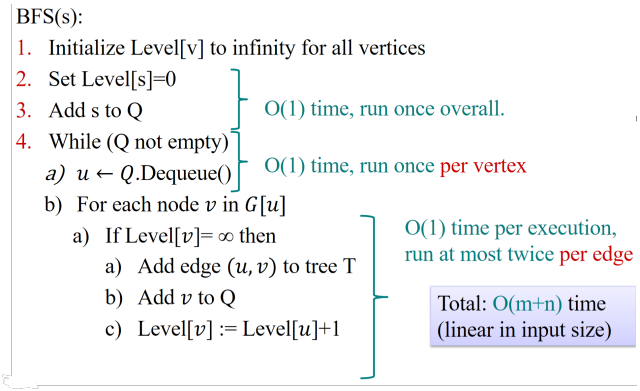


Figure 3.12: An analysis showing $O(m+n)$ for both time and space complexity

Proof 2.2: Claim 1 for BFS

Let s be the root of the BFS tree, then: **Proof:** Induction on the distance from s to u.

Base case ($u = s$): The code sets Level$[s] = 0$, and there is no path to find since the path has length 0.

Induction hypothesis: For every node u at distance $\leq i$, Claim 1 holds.

Induction step:

- Let v be a node at distance exactly $i+1$ from s. Let u be its parent in the BFS tree.
- The code sets Level$[v]$ = Level$[u]$ + 1.
- Let x be the last node before v on a shortest path from s to v. Since v is at distance $i+1$, then x must be at distance i, and so Level$[x] = i$ (by induction hypothesis).
- If $u = x$, we are done!
- If $u \neq x$, then it must be that u was explored before x, since otherwise x would be the parent of u.
- Since we explore nodes in order of level, Level$[u] \leq$ Level$[x] = i$.
- If Level$[u] = i$, then we are done.
- If Level$[u] < i$, then the path $s \sim u \to v$ has length at most i, which contradicts the assumption that the distance of v is $i+1$.

We conclude that Level$[u] = i$, Level$[v] = i+1$, and the path in the BFS tree that goes from s to u to v has length $i+1$. QED. ■

Definition 2.6: Tyes of Edges

In graphs we have three types of edges:

1. **Tree-edges:** An Edge present in a BFS tree.
2. **Forward-edges:** Edges from an ancestor to a descendant.
3. **Back-edges:** Edges from a descendant to an ancestor.
4. **Cross-edges:** Edges between which connect nodes that have no ancestor-descendant relationship.

Example: In Figure (3.10), $\{(s,c),(a,c),...\}$ are forward and tree edges, $\{(g,h)\}$ are cross edges.

The following arises from the above definitions:

Theorem 2.3: BFS does not Contain Back-edges

In a BFS tree, there are no back-edges, as they would create a cycle.

Note: In Figure (3.10), $b \to a$ is not a tree-edge, a is on level 1 and b on level 3. If this connection were to occur, it would create a cycle, breaking our tree.

Definition 2.7: Depth-First Search (DFS)

In a **depth-first search**, we recursively explore each an entire branch before moving onto the next.

Function 2.2: DFS Algorithm - `DFS(`G`)`

Depth-First Search on graph G (recursive).

Input: Graph $G = (V, E)$.
Output: Discovery and finishing times for each vertex.

```
1  Function DFS(G):
2      for each u ∈ G do
3          u.state ← unvisited;
4      end
5      time ← 0;
6      for each u ∈ G do
7          if u.state == unvisited then
8              DFS-Visit(u);
9          end
10     end
11 Function DFS-Visit(u):
12     time ← time + 1;
13     u.d ← time ;
14     // record discovery time;
15     u.state ← discovered;
16     for each v ∈ G[u] do
17         if v.state == unvisited then
18             DFS-Visit(v);
19         end
20     end
21     u.state ← finished;
22     time ← time + 1;
23     u.f ← time;
24     // record finishing time;
```

Time and Space Complexity: $O(n + m)$ where n is the number of vertices and m the number of edges.

Proof 2.3: DFS Correctness

Case 1: When s is discovered, there is a path of unvisited vertices from s to y. We use induction on the length L of the unvisited path from x to y.

- **Base case:** There is an edge (x, y), and y is unvisited.
- **Induction hypothesis:** Assume that the claim is true for all nodes reachable via k unvisited nodes.
- **Induction step:**
  - Consider u reachable via $k+1$ unvisited nodes.
  - Let z be the last node on the path before y, and z is discovered from x (by I.H.).
  - The edge (z, y) will be explored from x, ensuring that y is eventually visited.

Thus, DFS-Visit(x) explores all nodes reachable from x through a path of unvisited nodes, as required. ■

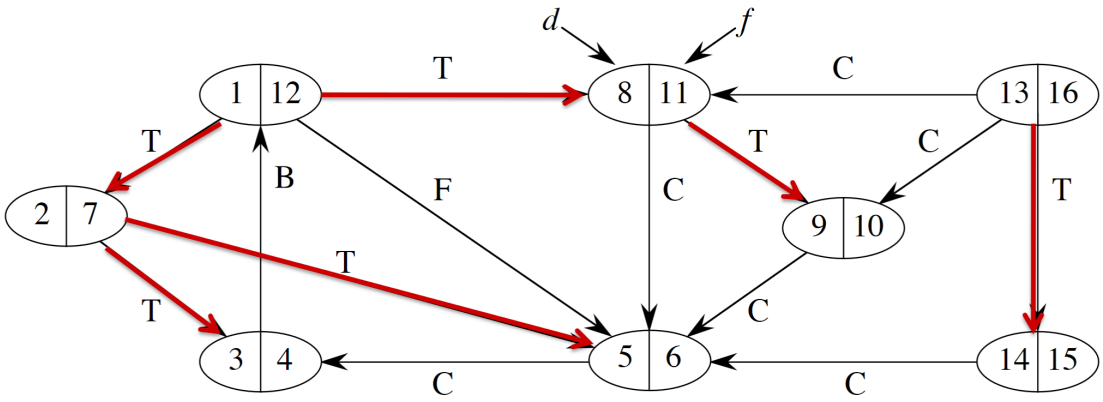


Figure 3.13: A graph showing our d discovered and f finished times, denoted in pairs (f, d).

Theorem 2.4: DFS and Cycles

Let DFS run on graph G, then:

$$(G \text{ has a cycle}) \iff (\text{DFS run reveals back edges})$$

Proof 2.4: Proof of Cycles and Back Edges

Proving (G has a cycle) $\Longleftarrow$ (DFS run reveals back edges): Every back edge creates a cycle.
Proving (G has a cycle) $\Longrightarrow$ (DFS run reveals back edges) Suppose G has a cycle:

- Let u_1 be the first discovered vertex in the cycle, and let u_k be its predecessor in the cycle.
- u_k will be discovered while exploring u_1.
- The edge (u_k, u_1) will be a back edge.

■

3.3 Directed-Acyclic Graphs & Topological Ordering

Graphs may represent a variety of relationships, such as dependencies between tasks or the flow of information.

Definition 3.1: Directed-Acyclic Graph (DAG)

A **directed-acyclic graph** is a directed graph with no cycles.

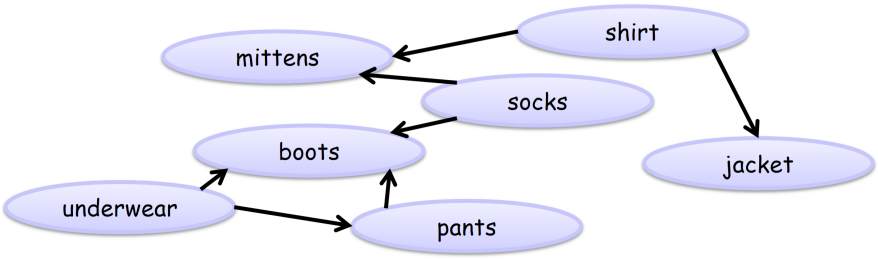


Figure 3.14: A DAG depicted by getting dressed for winter.

For each node to be processed its **dependencies** or parents must be processed first.

Definition 3.2: Topological Ordering

Given a graph, a **topological ordering** is a linear ordering of nodes such that for every edge (u, v), u comes before v.

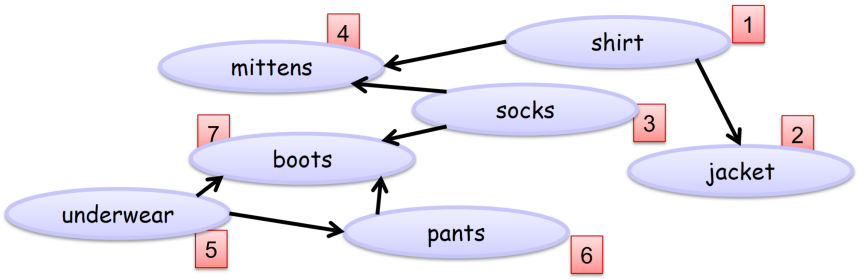


Figure 3.15: A topological ordering of the DAG in Figure (3.14) enumerated in red.

Another possible ordering of $[1, 2, 3, 4, 5, 6, 7]$ is $[5, 6, 1, 2, 3, 4, 7]$, as $5 \to 6$ is independent.

Theorem 3.1: Topological Sort

Given a DAG, a topological ordering can be found.

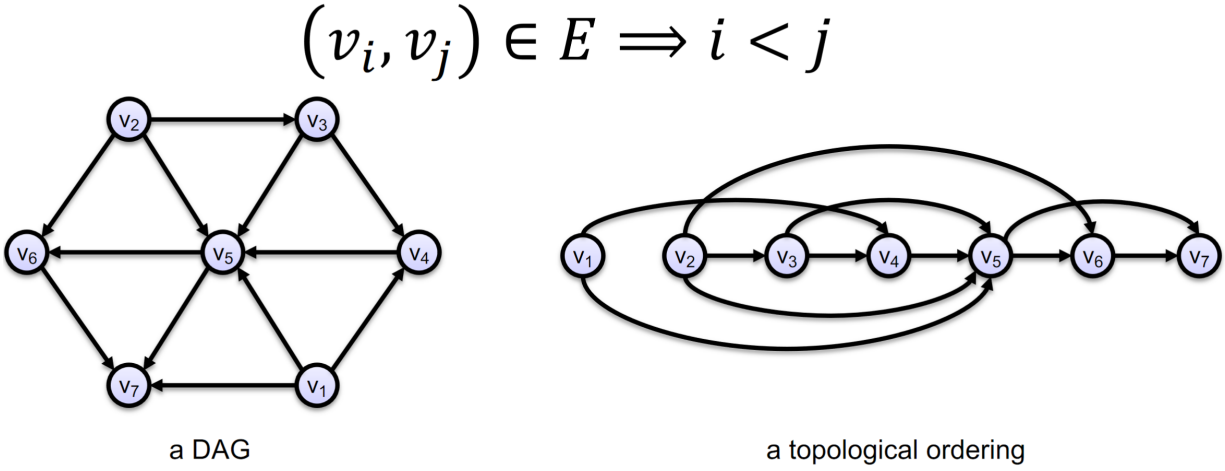


Figure 3.16: A topological sorting of a DAG E and v nodes

Proof 3.1: Topological Sort via DFS

Lemma: In a directed graph G, if (note necessarily acyclic):

- (u, v) is an edge, and
- v is not reachable from u,

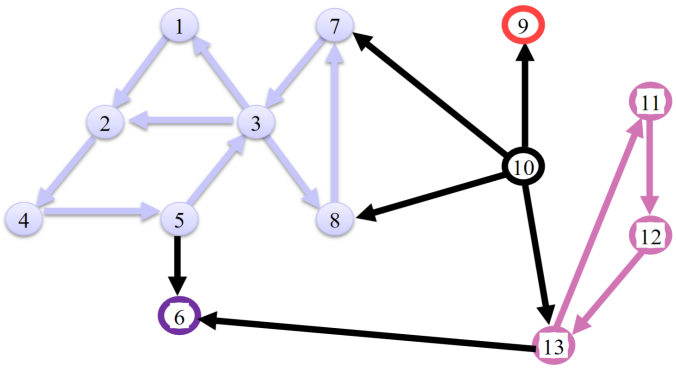
Then in every run of DFS, $u.f > v.f$.
Proof:

- If v is started before u, then the DFS-Visit(v) will terminate without reaching u (because there is no path to u).
- If u is started before v, then the edge (u, v) will be explored before u is finished.

Therefore, in all cases, $u.f > v.f$. ■

Definition 3.3: Strongly Connected Components

A **strongly connected component** is a subgraph where every node is reachable from every other node. Then we say $u \rightsquigarrow v$ and $v \rightsquigarrow u$ are **mutually reachable**.



- Observation. Two SCCs are either disjoint or equal.
- If we contract the SC components in one node we get an *acyclic* graph.

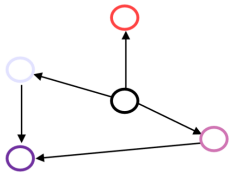


Figure 3.17: A graph with 5 strongly connected components.

— 4 —

Scheduling

4.1 Interval Scheduling

Scheduling problems arise in many areas, such as scheduling classes, tasks, or jobs. Interval scheduling is a type of scheduling problem where we want to maximize the number of tasks we can complete.

Definition 1.1: Schedule

A **schedule** is a set of tasks which we call **jobs**. Each job has a start time s_i and an end time f_i. Two jobs are **compatible** if they do not overlap.

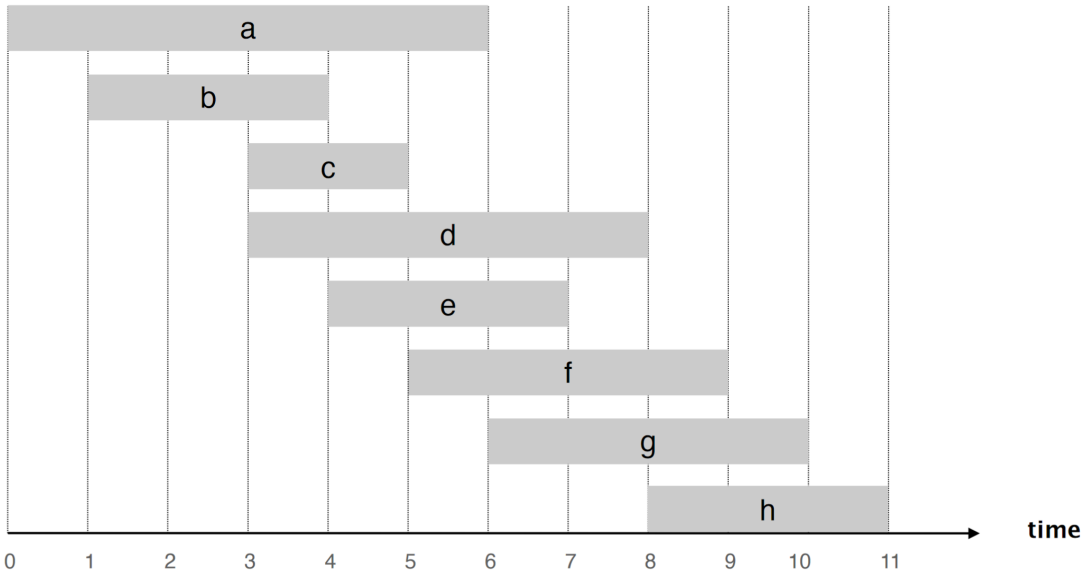


Figure 4.1: Given jobs a through h we find the largest subset of mutually compatible jobs $\{b, e, h\}$.

Definition 1.2: Greedy Algorithm

A **greedy algorithm** is an algorithm that makes the best choice at each step. I.e., it cares not about the future or big picture, only the immediate benefit, for fast computations.

Note: This definition becomes *loose*, as we encounter problems with backtracking or multiple states. As in each state it makes the best choice with the information available.

Possible Approaches: Let s_j and f_j be the start and finish times of job j.

- **[Earliest Start Time]:** Consider jobs in ascending order of s_j.
- **[Earliest Finish Time]:** Consider jobs in ascending order of f_j.
- **[Shortest Interval]:** Consider jobs in ascending order of $f_j - s_j$.
- **[Fewest Conflicts]:** For each j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

We choose the **Earliest Finish Time** approach:

Proof 1.1: Greedy Algorithm Earliest Finish Time Correctness

Let $i_1, i_2, \ldots, i_k$ denote the set of jobs selected by the greedy algorithm.
Let $j_1, j_2, \ldots, j_m$ denote the set of jobs in an optimal solution, with

$$i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r \text{ for the largest possible value of } r.$$

We can swap j_{r+1} for i_{r+1} in the optimal schedule, and it will still remain compatible. We repeat swaps until $r = k$. It's not possible that $m > k$ because j_{k+1} is compatible with i_k. ■

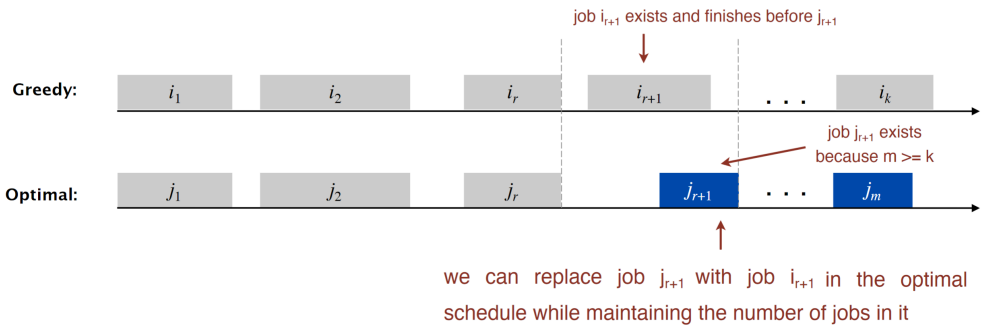


Figure 4.2: Shows that at the first divergence, i_{r+1} and .

Theorem 1.1: Interval Scheduling & Earliest Finish Time

Given a set of jobs j with start and finish times s_j and f_j, we can obtain an optimal like solution by scheduling in ascending order of f_j, choosing the next compatible job.

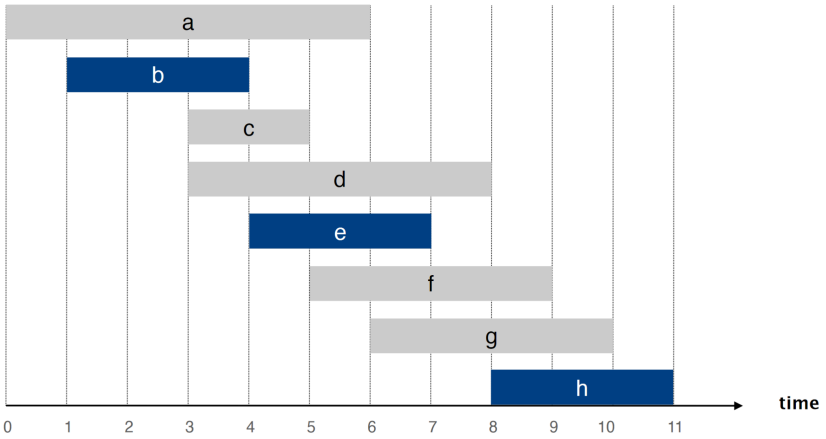


Figure 4.3: Solution to Figure (4.1) using early finish time first, yielding $\{b, e, h\}$.

Function 1.1: EarliestFinishTimeFirst Algorithm - `EFT`$(s_1, \ldots, s_n, f_1, \ldots, f_n)$

Finds the maximum set of non-overlapping jobs based on earliest finish time.

Input: A set of jobs with start times s_j and finish times f_j.
Output: The maximum set of selected jobs.

```
1 sorted_jobs ← sort(f_1, ..., f_n) // sort by finish time S ← ∅ // selected jobs
    f_last ← −∞;
2 for each j in sorted_jobs do
3 |   if f_last ≤ s_j then
4 |   |   S ← S ∪ {j};
5 |   |   f_last ← f_j;
6 |   end
7 end
8 return S
```

Time Complexity: $O(n \log n)$ assuming our sorting algorithm is $O(n \log n)$. Then we iterate through n jobs.
Space Complexity: $O(n)$ storing the input of n jobs.

4.2 Interval Partitioning

Interval partitioning generalizes our interval scheduling to multiple resources, allowing them to run in parallel.

Definition 2.1: Interval Partitioning

Given a schedule of jobs j with start and finish times s_j and f_j. We **partition** jobs into a minimal amount of k resources such that no two jobs on the same resource overlap.

Scenerio: ***Class Scheduling***
Say we have n classes and k classrooms. What are the minimum number of classrooms needed to schedule all classes?

Example: Let $n = \{a, b, c, \ldots, i\}$ be classes with start and finish times. We attempt to find the minimum number of k classrooms needed to schedule all classes.

(1.)

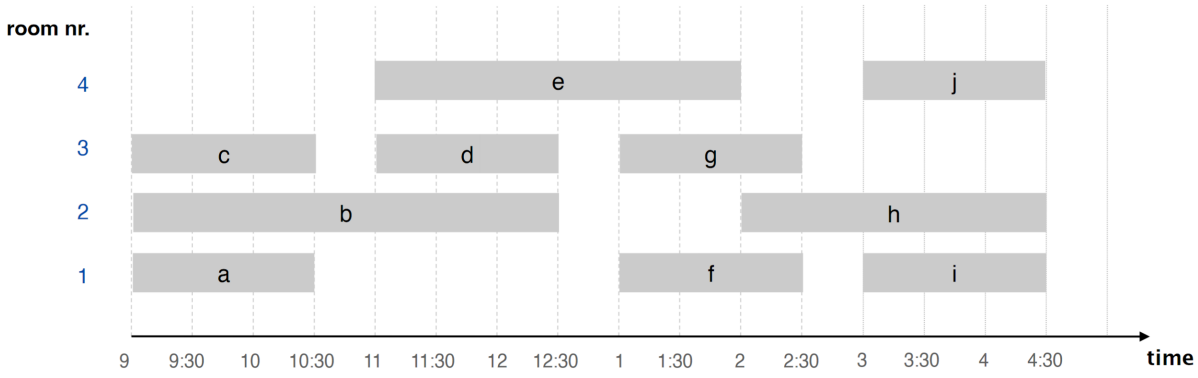


Figure 4.4: Though not optimal, here is a possible schedule where $k = 4$.

We strategies and figure out the minimum number of classrooms needed to schedule all classes in the worst-case. We observe in Example (4.2) that $\{c, b, a\}$ strictly overlap. Moreover, there are at most 3 classes overlapping at any time. Thus, we need at least 3 classrooms.

Theorem 2.1: Minimality of Interval Partitioning

Given a set of jobs j, c conflicting tasks, and k resources. We find the optimal k by $k = \max(c)$.

Possible Approaches: Let s_j and f_j be the start and finish times of job j.

- **[Earliest Start Time]:** Consider jobs in ascending order of s_j.
- **[Earliest Finish Time]:** Consider jobs in ascending order of f_j.
- **[Shortest Interval]:** Consider jobs in ascending order of $f_j - s_j$.
- **[Fewest Conflicts]:** For each j, count the number of conflicting jobs c_j.
  Schedule in ascending order of c_j.

Theorem 2.2: Interval Partitioning & Earliest Start Time

Given a set of jobs j with start and finish times s_j and f_j, we can obtain an optimal like solution by scheduling in ascending order of s_j. If two jobs overlap, we allocate a new resource.

Proof 2.1: Classroom Allocation by Early Start Time First

Let d be the number of classrooms that the algorithm allocates:

(i.) Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with all $d-1$ other classrooms.

(ii.) These d lectures each end after s_j.

(iii.) Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.

All schedules use $\geq d$ classrooms. Thus, we have d lectures overlapping at time $s_j + \epsilon$.

■

Tip: Though number of conflicts is the optimal number of classrooms, it tells us nothing about how to allocate them. We can use the **Earliest Start Time** as it identifies our next best choice and allocates conflicts as they arise.

Function 2.1: EarliestStartTimeFirst Algorithm - `EST`$(j = 1 \dots n : s_j, f_j)$

Finds an optimal schedule of lectures based on their earliest start time.

Input: A set of lectures with start times s_j and finish times f_j.
Output: Assignment of lectures to rooms.

```
1  A ← empty hash table      // A[k] contains the list of lectures assigned to
     room k sorted_class ← sort(s_1, ..., s_n)          // sort lectures by start time
2  for each c in sorted_class do
3  |  k ← find_compatible_room(c, A, Q);
4  |  if k is not None then
5  |  |  A[k].add(c);
6  |  end
7  |  else
8  |  |  d ← len(A)      // highest room id A[d + 1] ← [ ]      // open new room
   |  |  A[d + 1].add(c);
9  |  end
10 end
11 return A
```

Time Complexity: $O(n \log n)$ assuming our sorting algorithm is $O(n \log n)$. Then we iterate through n jobs.
Space Complexity: $O(n)$ storing the input of n jobs.

4.3 Priority Queues

In this section we will discuss **min-heaps** which will help us sort maintain elements in a sorted data structure.

Definition 3.1: Balanced Tree

A **balanced tree** is a tree where the height of the left and right subtrees of every node differ by at most one.

Tip: To spot this, observe each level of the tree and see whether after consecutive levels, one branch has more nodes than the other.

Below is an example of a balanced tree and an unbalanced tree:

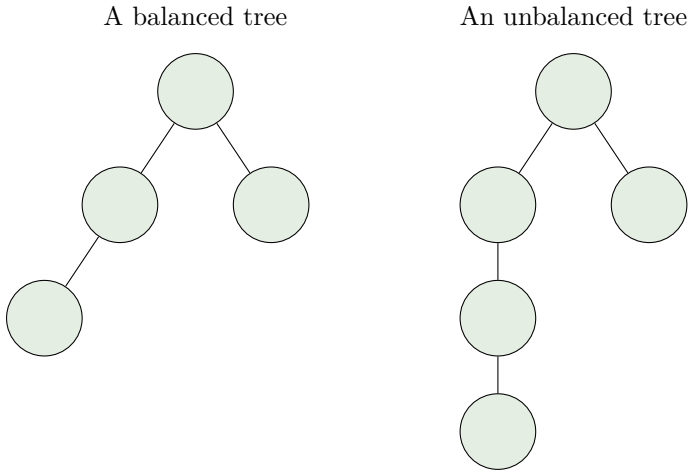


Figure 4.5: Examples of a balanced tree and an unbalanced tree

Definition 3.2: Binary Search Tree

A **binary tree** is a where each parent node p_i has at most two children c_{left} and c_{right}.
A **binary search tree** has each child ($c_{left} > \mathrm{p}_i$) and ($c_{right} < \mathrm{p}_i$).

With n nodes, there are $n - 1$ edges.

Definition 3.3: Heap

A **heap** is a binary tree with the following properties:

(i.) It is a **complete binary tree** (a binary and balanced tree).

(ii.) It is a **min-heap** if the parent node is less than its children.

(iii.) It is a **max-heap** if the parent node is greater than its children.

Operations: Suppose we have a heap of n elements:

- **PEEK:** Return the root. $O(1)$;
- **INSERT:** Add a new element. $log(n)$;
- **EXTRACT:** Remove the root. $log(n)$;
- **UPDATE:** Update an element. $log(n)$;

Tip: The difference between a min-heap and a binary search tree, is that $c_{left} \leq c_{right} \leq \mathrm{p}_i$. That is, the left and right child in a min-heap are not ordered, just less than the parent; Contrary to a binary search tree where the left child is less than the parent and the right child is greater.

We use a min-heap as a data structure to maintain a sorted list as an input.

Function 3.1: find_compatible_room - FCR(c, A, Q)

Finds a compatible room for class c based on the current room schedule.

Input: Class ID c, current schedule A, priority queue Q with room finish times.
Output: The room k compatible with class c or `None`.

```
   // c: class id, A: current schedule of room assignments
   // Q: priority queue with room finish times
1  ⟨f_k, k⟩ ← PEEK_MIN(Q)            // shows lowest ⟨key, value⟩ pair, O(1)
2  if s_c > f_k                                  // finish time in room k then
3  |  return k                                  // c is compatible with room k
4  end
5  else
6  |  return None;
7  end
```

Time Complexity: $O(n)$ as now we only need to check the minimum finish time in the priority queue.
Space Complexity: $O(n + m)$ if our min-heap is implemented as a hash table.

We can also implement a min-heap for sorting classes as well.

Function 3.2: EarliestStartTimeFirst Algorithm - `EST`($j = 1 \dots n : s_j, f_j$)

Finds an optimal schedule of lectures based on their earliest start time.

Input: Start times s_j and finish times f_j of classes.
Output: Assignment of lectures to rooms.

```
   // s_j, f_j: start and finish times of classes
 1 A ← empty hash table        // A[k] contains the list of courses assigned to
     room k Q ← empty priority queue      // contains ⟨finishTime, roomId⟩ pairs
     sorted_class ← sort(s_1, ..., s_n)                    // sort by start time
 2 for each c in sorted_class do
 3 |   k ← find_compatible_room(c, A, Q);
 4 |   if k is not None then
 5 |   |   A[k].add(c);
 6 |   |   Q.UPDATE-KEY(⟨f_k, k⟩, ⟨f_c, k⟩)        // update finish time of room k
 7 |   end
 8 |   else
 9 |   |   d ← len(A)       // highest room id A[d + 1] ← [ ]      // open new room
   |   |    A[d + 1].add(c);
10 |   |   Q.INSERT(⟨f_c, d + 1⟩);
11 |   end
12 end
13 return A
```

Time Complexity: $O(n \log n)$ as inserting into a min heap is $O(\log n)$ and reading is $O(1)$.
Space Complexity: $O(n)$ storing the input of n classes.

Theorem 3.1: Heap Array Representation

A heap H can be represented by a zero-indexed array A via:

(i.) The root is at index 0.

(ii.) The left child of node i is at index $2i + 1$.

(iii.) The right child of node i is at index $2i + 2$.

Enabling a **space complexity** of $O(n)$.

4.4 Minimizing Lateness

For situations where our tasks are forced onto a single resource, we want to minimize the lateness of tasks as much as possible.

Scenerio: *Hell in the Kitchen*
Say we have n dishes to cook for *very* important critics who place orders at t_j times. We know each dish takes p_j time to prepare. With only one kitchen, we want to minimize the lateness of each dish.

$j =$	1	2	3	4	5	6
t_j	3	2	1	4	3	2
p_j	6	8	9	9	14	15

Table 4.1: Table showing t_j and p_j start and finish times for dish j

Possible Approaches: Let s_j and f_j be the start and finish times of job j.

- **[Shortest Processing Time]:** shortest processing time t_j first.
- **[Earliest Deadline]:** earliest due time p_j first.
- **[Least Slack Time]:** least slack time $p_j - t_j$ first.

Counter Examples: Consider the following:

Shortest processing time t_j first:

	1	2
t_j	1	10
p_j	100	10

Table 4.2: Shortest processing time first

Smallest slack $p_j - t_j$ first:

	1	2
t_j	1	10
p_j	2	10

Table 4.3: Smallest slack first

Theorem 4.1: Minimizing Lateness

Given a set of jobs j with start and finish times t_j and p_j under one resource, we can obtain a like-optimal solution by scheduling in ascending order of p_j.

Tip: Rather than looking for the best solution, craft counter examples to eliminate the worst ones. The example for which you can't find a counter example, is likely the best solution.

Proof 4.1: Minizing Lateness by Earliest Deadline First

Let S^* be an optimal schedule:

- (we know S^* exists as we could exhaustively try all possible orders of jobs)

If S^* has no inversions, then $S^* = S$ by definition of the greedy schedule.
Thought experiment: let's make S^* more similar to S!

- While S^* has an inversion of consecutive jobs i, j, swap jobs i and j.

After

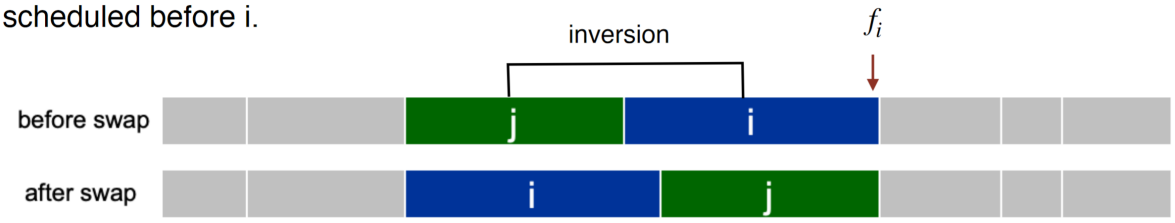
$$\leq \binom{n}{2} = O(n^2)$$

swaps, S^* has no inversions and hence is identical to S.
We know that swaps can only improve the lateness, hence we have

$$\text{Lateness}(S^*) \geq \text{Lateness}(S)$$

which means that S is optimal. ■

An inversion in a schedule S is a pair of jobs i and j such that $d_i < d_j$ but j is scheduled before i.



claim: Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Figure 4.6: Shows that at the first inversion, i and j are swapped.

Tip: Say you have to clean your room completely by ℓ time. You are confused whether to clean your bed first or your desk. The desk takes d time and the bed takes b time. Whether you choose to clean your bed or your desk first, does not make a difference, as $d+b$ will always be the same.

Function 4.1: EarliestDeadlineFirst Algorithm - `EDF`$(j = 1 \dots n : t_j, d_j)$

Schedule jobs based on their earliest deadline first.

Input: Length and deadlines of jobs.
Output: Intervals assigned to each job.

```
   // length and deadline of jobs
1  sorted ← sort(d_1, d_2, ..., d_n)   // sort by increasing deadline intervals ← empty
     list;
2  t ← 0                                                    // keep track of time
3  for each j in sorted do
       // assign job j to interval [t, t + t_j]
4      intervals.add([t, t + t_j]);
5      t ← t + t_j;
6  end
7  return intervals
```

Time Complexity: $O(n \log n)$ assuming our sorting algorithm is $O(n \log n)$. Then we iterate through n jobs.
Space Complexity: $O(n)$ storing the input of n jobs, and we maintain an array of our intervals. $n + n = 2n = O(n)$.

Applying this algorithm back to Figure (4.1), we get the following optimal schedule:

$j =$	1	2	3	4	5	6
t_j	3	2	1	4	3	2
p_j	6	8	9	9	14	15

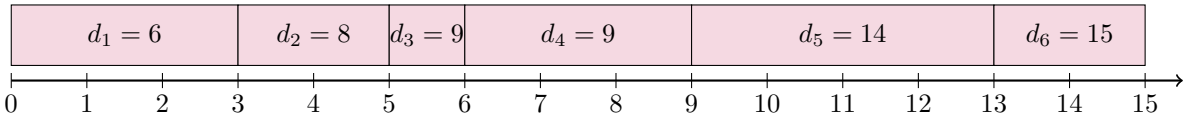


Figure 4.7: Where our number line represents time, and the rectangles the interval of each job.

Here we observe that we at most have 1 late job.

— 5 —

Greedy Algorithms

5.1 Shortest Path

Theorem 1.1: Dijkstra's Algorithm

Proposition: Suppose that there is a shortest path from nodes $u \to v$. Then any sub-path between these nodes, say $x \to y$, is also the shortest path.

Algorithm: Given a weighted graph G and a source node s,

(i.) Keep track of best distances, start at a queue with s.

(ii.) Pop off the queue, flag item as visited.

(iii.) View all children weights, update if it's the new shortest path to that node.

(iv.) Queue children in ascending order of weight (smallest→largest)

Preform steps (ii) to (iv) until the queue is empty, having visited all possible nodes.

Proof 1.1: Proof of Correctness for Dijkstra's Algorithm

Invariant: For each node $u \in S$, $d(u)$ is length of shortest path $s \rightsquigarrow u$. By induction on $|S|$:
Base case: $|S| = 1$ is true since $S = \{s\}$ and $d(s) = 0$.
Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be the next node added to S, and let (u, v) be the final edge.
- A shortest $s \rightsquigarrow u$ path plus (u, v) is an $s \rightsquigarrow v$ path of length $\pi(v)$.
- Consider any $s \rightsquigarrow v$ path P. We show that it is no shorter than $\pi(v)$.
- Let (x, y) be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it reaches y.

$$\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)$$

■

To visualize our proof consider paths the following diagram:

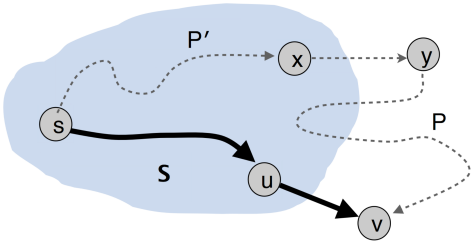


Figure 5.1: A system of subset paths (blue), and an exact point of exit $u \to v$ and $x \to y$

Since $u \to y$ and $x \to y$ are at the same exact point of exit, i.e., say $s \to v \cong s \to y$, then to go from $y \to v$ must take some additional step. Therefore, $s \to y \to v$ is longer. **E.g:** Let $s \to y := 3$ and $s \to v := 3$, and all steps take 1. Then $s \to y \to v = 4$, while $s \to v = 3$. Therefore $s \to v$ is the shortest path.

Dijkstra Example (i): To emphasize the BFS nature of Dijkstra's algorithm:

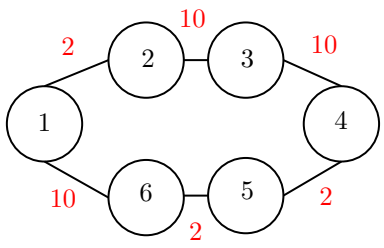


Figure 5.2: A weighted graph.

Where iterations and the queue look like:

Iteration Init	from 1	from 2	from 6	from 3	from 5
$1:0$	$1:0$	$1:0$	$1:0$	$1:0$	$1:0$
$2:\infty$	$2:2$	$2:2$	$2:2$	$2:2$	$2:2$
$3:\infty$	$3:\infty$	$3:12$	$3:12$	$3:12$	$3:12$
$4:\infty$	$4:\infty$	$4:\infty$	$4:\infty$	$4:22$	$4:14$
$5:\infty$	$5:\infty$	$5:\infty$	$5:12$	$5:12$	$5:12$
$6:\infty$	$6:10$	$6:10$	$6:10$	$6:10$	$6:10$

Queue Init	from 1	from 2	from 6	from 3	from 5
$[1]$	$[2,6]$	$[6,3]$	$[3,5]$	$[5,4]$	$[4]$

Finally visiting 4 to see 3 and 5 have already been visited, ending the algorithm as the queue's empty.

Dijkstra Example (ii):

Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

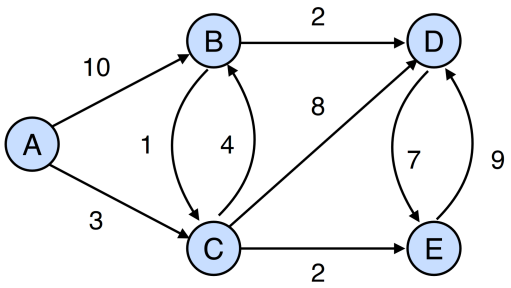


Figure 5.3: A weighted graph and its shortest paths.

In figure (5.3), the first row is 0 as $s \to s$, other nodes are assumed ∞,i.e., undefined. Each subsequent row finds the next shortest path, while updating the table about information it gathers. However we have one problem with Dijkstra's algorithm, it does not work with negative weights.

Theorem 1.2: Dijkstra's Algorithm and Negative Weights

Dijkstra's algorithm does not work with negative weights. This is because it assumes that the shortest path is the sum of the shortest paths. Therefore, if the algorithm believes it has found the shortest path, it assumes any further traversals will only increase the path length.

To illustrate the deterioration of Dijkstra's algorithm:

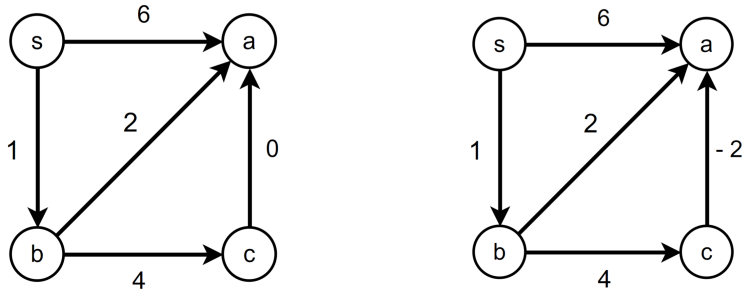


Figure 5.4: Shows two weighted graphs, one positive, and the other negative.

In Figure (5.4), our negative graph will never figure out the shortest path ($s \to b \to c \to a$=1). It will always assume ($s \to b \to a = 3$) is the shortest path. As when it looks at $c = 4$, it will think that it's impossible to yield any shorter of a path 3 as anything beyond 4 must be larger.

Function 1.1: Dijkstra Algorithm - `Dijkstra`(G, s)

Finds the shortest path in a weighted directed graph.

Input: A graph $G = (V, E)$ with adjacency list $G[u][v] = l(u, v)$ and source node s.
Output: Shortest distances $d[u]$ and parent nodes for paths.

```
1  Function Dijkstra(G, s):
2      π ← {}// hash table, current best list for v
3      d ← {}// hash table, distance of v
4      parents ← {}// parents in shortest path tree
5      Q ← PQ()// priority queue to track minimum π
6      π[s] ← 0;
7      Q.INSERT(⟨0, s⟩);
8      for v ≠ s in G do
9          π[v] ← ∞;
10         Q.INSERT(⟨π[v], v⟩);
11     end
12     while Q is not empty do
13         ⟨π[u], u⟩ ← EXTRACT-MIN(Q);
14         d[u] ← π[u];
15         for v ∈ G[u] do
16             if π[v] > d[u] + l(u, v) then
17                 DECREASE-KEY(⟨π[v], v⟩, ⟨d[u] + l(u, v), v⟩);
18                 π[v] ← d[u] + l(u, v);
19                 parents[v] ← u;
20             end
21         end
22     end
23 return d, parents
```

Time Complexity: $O(m \log n)$ where m is the number of edges and n is the number of nodes, assuming G is connected ($n - 1 \leq m$); Otherwise, $O((n + m) \log n)$.
Space Complexity: $O(n + m)$ storing the hash-table of the graph and priority queue.

5.2 Spanning trees

Definition 2.1: Spanning Tree

A **spanning tree** of a graph G is a subgraph containing edges to each $n \in G$ without cycles.

Definition 2.2: Minimum Spanning Tree (MST)

A **minimum spanning tree** of a graph G is a spanning tree with the smallest sum of edge weights.

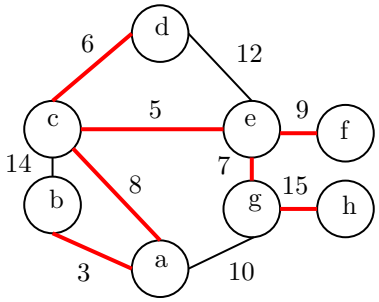


Figure 5.5: Example of an graph with MST highlighted in red.

This tree visits each node once taking the shortest path which connects all of them.
Possible Algorithms:

- **Prim's:** Start with some root node s. Grow a tree T from s outward. At each step, add to T the cheapest edge e with exactly one endpoint in T.
- **Kruskal's:** Start with $T = \emptyset$. Consider edges in ascending order of weights. Insert edge e in T unless doing so would create a cycle.
- **Reverse-Delete:** Start with $T = E$. Consider edges in descending order of weights. Delete edge e from T unless doing so would disconnect T.
- **Borůvka's:** Start with $T = \emptyset$. At each round, add the cheapest edge leaving each connected component of T. Terminates after at most $\log(n)$ rounds.

Next we revisit cycles and introduce **cuts**, which will have important implications when approaching this problem.

Definition 2.3: Endpoint

An **endpoint** is either end of an edge. So for edge $e = u \leftrightarrow v$, u and v are endpoints. If $u \rightarrow v$, then v is an endpoint of u.

Definition 2.4: Cut

Given a graph G, partitioning of the nodes into a set is called a **cut**, say G'. Nodes, with exactly one endpoint in G', are the **cut-set**.

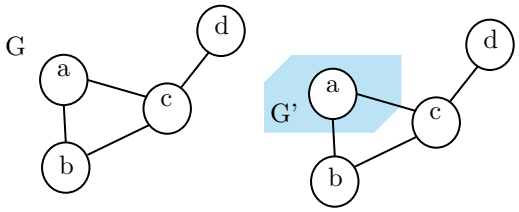


Figure 5.6: Illustration of a graph G and a cut G' of the graph.

We see in Figure (5.6) that $G' = \{a\}$ and our cut-set contains edge-pairs (a, b) and (a, c). Where the edge (d, c) is not included as the cut G' does not intersect it.

Theorem 2.1: Cycles & Cut-sets

If a cut-set crosses a cycle, then the cut-set intersects an even number of edges in the cycle. As what comes in, must come out.

Given Figure (5.6), the cut-set G' intersects the cycle (a, b, c), yielding an even cut-set.

Theorem 2.2: Cycle Property

In a graph with a cycle, the edge with the largest weight in that cycle is not in the MST. As taking an edge from a cycle does not disconnect the graph, the largest edge is not necessary.

Theorem 2.3: Cut Property

Given a graph G and a cut-set C, where e is the lightest-edge in C; e must be in the MST. As if $e \notin G$ and G is an MST, adding e creates a cycle. By the cycle property, e must replace the heaviest edge in that cycle.

Example: Given the below Figure (5.7), point e must be in the MST to connect all nodes (cut property). Say dash-edge f is the largest in its cycle, then f is not in the MST (cycle property).

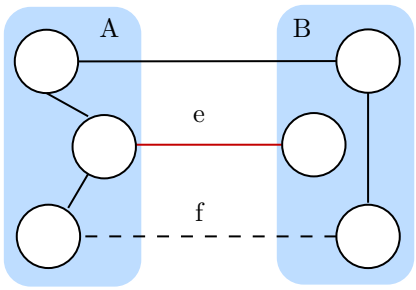


Figure 5.7: A graph cut into two disjoint sets, with a highlighted edge e and a dashed edge f.

Theorem 2.4: Prim's Algorithm

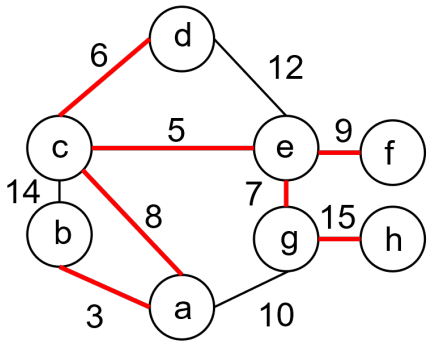
Given a connected graph G with n nodes and m edges, we produce the MST via:

(i.) Initialize an MST table T, and a priority queue Q with each n of weight ∞.

(ii.) Start a round with an arbitrary node s, evaluating children nodes v.

(iii.) Update each $T[v] = s$, if $w(s, v)$ is lighter than $T[v]$.

(iv.) End this round, take the top node in Q as the new s, repeat (ii.)-(iv.) until all $n \in T$.



NODE	PARENT
a :	NULL
b :	a
c :	a
d :	c
e :	c
g :	e
f :	e
h :	g

Figure 5.8: Prim's Alg. in the order $a \to b \to c \to e \to d \to g \to f \to h$, and a parent table.

The above example shows how our parent table will result with the following table. Next we will discuss in detail how one could approach implementation.

Tip: Live Demo of Prim's Algorithm https://www.youtube.com/watch?v=cplfcGZmX7I.

Function 2.1: Prim's Algorithm - `PALG()`

Input: A connected, undirected graph G of V nodes. With weights $w(u, v)$, and $u, v \in V$.
Output: Minimum Spanning Tree (MST) formed by edges $T < (v, \texttt{parent}[v]) >$

```
1  Q < weight, node >; // Min-heap of <key, data>
2  V.forEach((v) => {Q[v] ← ∞}); // for each v ∈ V set it's weight to ∞
3  Q[V_0] ← 0; // Picking arbitrary node V_0, pushing it to the top of Q
4  T; // Hashtable where T[u] is the parent v of u
5  while Q ≠ ∅ do
6      u ← Q.Extract();
7      foreach v ∈ G[u] do
8          if (v ∈ Q) and (w(u, v) < Q[v]) then
               // Edit the node's weight in Q then re-balance.
9              Q[v] ← w(u, v);
10             Q.DecreaseKey(v);
11             T[v] ← u;
12         end
13     end
14 end
15 return T
```

Correctness: We run a form of BFS on the graph, which touches every node. BFS creates levels each iteration, resulting in a cut-set with an end-point $G[u]$. By the cut property, any new lightest edge $w(u, v)$ is added or replaces a heavier edge in T. Thus forming an MST as all nodes are considered.
Time Complexity: $O((n+m)\log n)$. Line 7 at worse checks every adjacency, $O(n+m)$, for m edges of n nodes. Say lines 8-9 takes $O(1)$ time to find $v \in Q$ via hash-table. Line 10 takes $O(\log n)$ time to re-balance the heap. Thus, $O((n+m)\log n)$.
Space Complexity: $O(n+m)$, as we at most store the data items a hash-table representation of our graph.

Note: Lines 8 to 10 may require additional implementation. Since basic min-heaps only store weights, one might need a direct reference to each member in the heap. Say a reference hash-table R, where $R[v]$ points to v node in Q. Once we update $R[v]$, we tell the Q to sort the new v weight. We say "*DecreaseKey*" as our new weight should be lighter, bubbling up the heap.

To check if a node has been visited before, doesn't matter in our case, as we update our solution T with a better solution once it is found. We additionally discard the lightest node each round to avoid infinite loops. If one wanted to, they could use T's entries as a visited list. If entries are

undefined upon access, then they have not been visited.

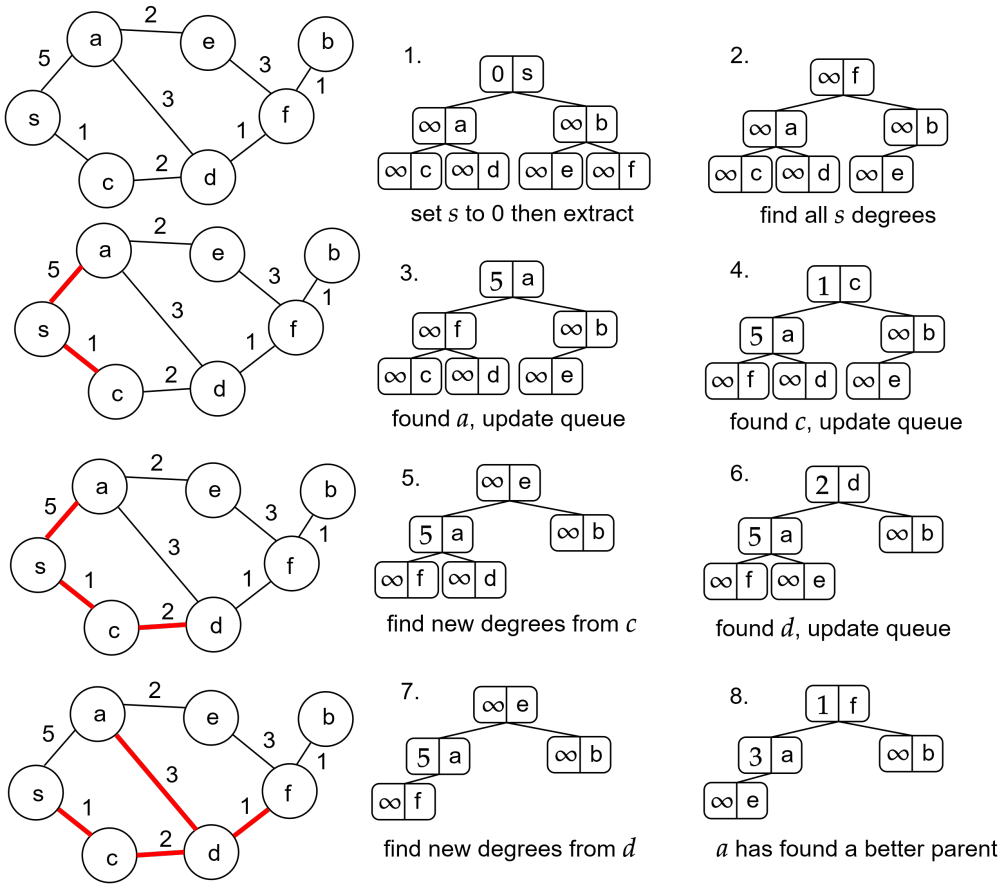


Figure 5.9: A diagram illustrating the pattern of Prim's Algorithm through each iteration.

The above diagram shows Prim's algorithm each round, pick the lightest edge at the top of the heap, check its degrees which have not been, update weights, re-balance the heap, and repeat. The algorithm will terminate when all nodes have been picked from the heap.

Union-Find Data Structures

Definition 2.5: Union-Find Data Structure

A **Union-Find** data structure is a data structure that keeps track of a set of elements partitioned into multiple disjoint subsets. It supports two useful operations: **Union:** Merge two subsets into a single subset. **Find:** Determine which subset a particular element is in.

We *could* simply use a hash-table to keep track of the parent of each node, which gives us **Find O(1); However, Union is O(n)**, as we would have to update every node to its new parent. Given a large set of n nodes, with m edges in a hash-table, to find and union all n nodes results in $\mathbf{O(n^2 + m)}$, as for every n we find, we make n updates, where our finds accumulate to the number of total edge connections.

Scenario - *Follow The Leader:* Say you have n people playing rock-paper-scissors. If n_i beats n_j, then n_j follows n_i. This creates large sets of people following a leader. When leader n_k beats n_i, n_i follows n_k with n_i's followers tagging along.

Say we are trying to figure out which component n_x is in. We ask n_x, "who is your leader," they say n_i, then n_i says n_k, and n_k replies, "I am the leader." Therefore n_x is in group n_k.

Definition 2.6: Forest

A **forest** is a collection of disjoint trees, where each tree has a **representative** r node. We may union-join trees A and B by making A be B's representative. Where $b \in B$ still point to B and B points to A.

Trees S with smaller heights should be added to bigger trees B. As height indicates the number of nodes who report to a leader. By adding the bigger tree to the smaller, we increase the time complexity of finding leaf nodes.

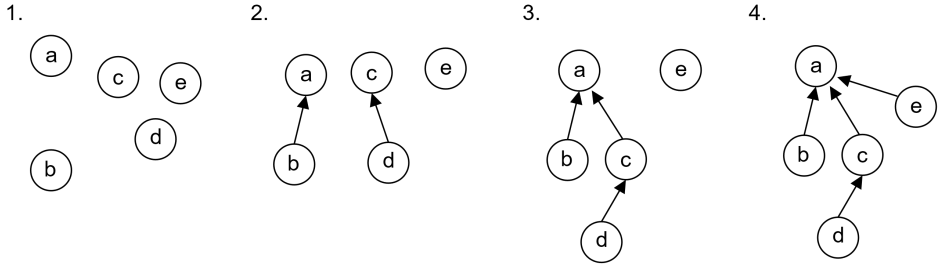


Figure 5.10: Showing disjoint nodes Union-Join with each other.

In the above figure nodes we see in step two we have two disjoint trees, with leaders a and c. In step three we join a and c by making c point to a. Finally e points to a to join the group.

We see a lot of redundancy, with nodes n_x reporting to leaders n_i, until a final leader n_k is reached. We may improve this overtime by setting n_x's leader to n_k directly.

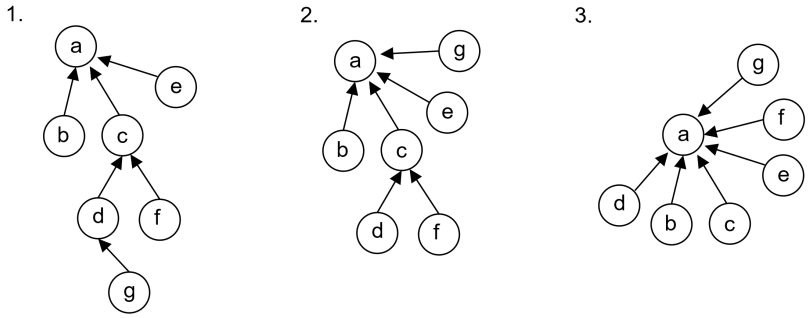


Figure 5.11: Showing a forest compress over multiple finds.

Given the figure above, we first ask g who their leader is, they report to c who reports to a. We now set g's leader to a. We do the same for f and d. Now once we ask g, f, or d who their leader is, they report to a directly without the need to traverse the tree. This is called **Path Compression**.

Definition 2.7: Path Compression

Path Compression is a technique used in Union-Find data structures to flatten the structure of the tree. When we find the leader of a node, we set the node's parent to the leader directly. This reduces the time complexity of find operations for subsequent queries.

We compare all of our techniques in the following table:

Operation\Implementation	**Simple Array**	**Forest**	**Forest with Path Compression**
Find (worst-case)	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
Union of sets A, B (worst-case)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Total for n unions and n finds, starting from singletons	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(m \cdot \alpha(m, n))$

Table 5.1: Time complexity comparison of different implementations.

Theorem 2.5: Kruskal's Algorithm

Given a connected graph G of V nodes and E edges, we produce the MST via:

(i.) Sort all $e \in E$ by weight in ascending order into an array W.

(ii.) Initialize a forest T with all V nodes as singletons.

(iii.) For each $e \in W$ Union-find its endpoints u and v.

- If u and v are in different sets, Union-join u and v.

Return the resulting forest T as the MST.

Function 2.2: Kruskal's Algorithm - `KALG()`

Input: a connected graph G of V nodes and E edges.
Output: Minimum Spanning Tree (MST) formed by forest Union-find data structure.

```
1 W[ ] ← Sort(E); // Sort all edges by weight
2 T ← new UnionFind(G); // new forest with all G's nodes as singletons
3 for i = 1 to W.size() do
4 |   (u, v) ← e; // Get the endpoints of e
5 |   if T.Find(u) ≠ T.Find(v) then
6 |   |   T.Union(u, v); // Union-join u and v
7 |   end
8 end
9 return T
```

Correctness: Sorting edges in ascending order, ensures lightest possible edge is picked first before redundancy checks with Union-find, which avoids cycles (Line 5). Any new unique edge e is added to the MST. This yields a connected graph as all edges are considered no matter their weight.
Time Complexity: $O(E \log E)$ or $O(E \log V)$. Line 1 sorts all edges, which takes $O(E \log E)$ time, where E is at most V^2 (all nodes connect to each other). Thus, $\Theta(E \log_2 V)$ is $O(E \log_2 E)$ as the exponent to reach V and E are the same. Then iterating through all E edges; actions are one-to-one for each *find-merge* operation E_i. This results in one operation per edge, rather than a compounded combination of operations, like a nested for-loops.
Space Complexity: $O(V+E)$, as we at most store the data items a hash-table representation of our graph.

Note: Lines 1 and 4, might require additional implementation such as a reference hash-table R to keep track of edges and their end-points after sorting.

— 6 —

Evaluating Recursive Algorithms