Algorithms and Data Structures

Christian J. Rudder

October 2024

Contents

Contents **1**

1 Complexity Theory **9**
1.1 Asymptotic Notation 9
1.2 Evaluating Algorithms 13

2 Proving Algorithms **14**
2.1 Stable Matchings 14
2.2 Gale-Shapley Algorithm 16

3 Graphs and Trees **18**
3.1 Paths and Connectivity 18
3.2 Breath-First and Depth-First Search 22
3.3 Directed-Acyclic Graphs & Topological Ordering 30

4 Scheduling **33**
4.1 Interval Scheduling 33
4.2 Interval Partitioning 36
4.3 Priority Queues 38
4.4 Minimizing Lateness 42

5 Greedy Algorithms **45**
5.1 Shortest Path . . . 45
5.2 Spanning trees . . . 49
Union-Find Data Structures . . . 54

6 Evaluating Recursive Algorithms **57**
6.1 Inductive Analysis . . . 57

7 Computational Algorithms **65**
7.1 Computers & Number Base Systems . . . 65
7.2 Computing Large Numbers . . . 68
7.3 Computational Efficiency . . . 75

8 Dynamic Programming **78**
8.1 Formulating Recursive Cases . . . 78
8.2 Bottom-Up Dynamic Programming . . . 81
8.3 Backtracking . . . 83
8.4 Subset Sum . . . 84
Weighted Ceiling . . . 84
Knapsack . . . 87
Unbounded Knapsack . . . 88
8.5 Shortest Paths - Bellman-Ford Algorithm . . . 92

9 Network Flow **95**
9.1 Residual Graphs . . . 95
9.2 Bipartite Matching . . . 102
9.3 Edge-disjoint Paths . . . 103
9.4 Multiple Sources & Sinks . . . 104

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];
```

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          else
10             if a prefers e over their current match then
11                 match e and a, replacing the current match;
12                 update M accordingly;
13     return 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:

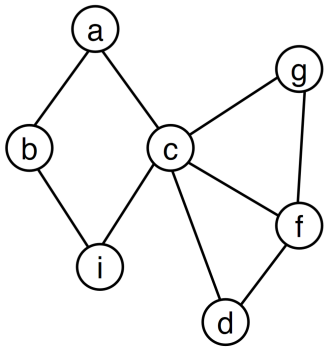


Figure 3.1: An undirected graph with 7 vertices and 9 edges.

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:

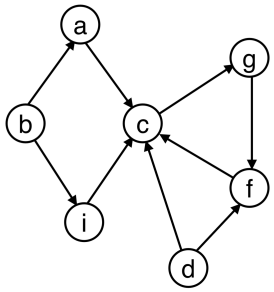


Figure 3.2: A directed graph with 7 vertices and 9 edges.

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:

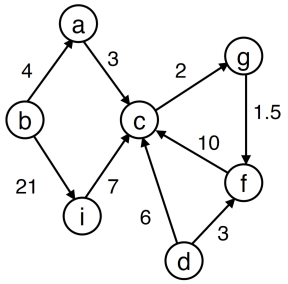


Figure 3.3: A weighted graph with 7 vertices and 9 edges.

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:

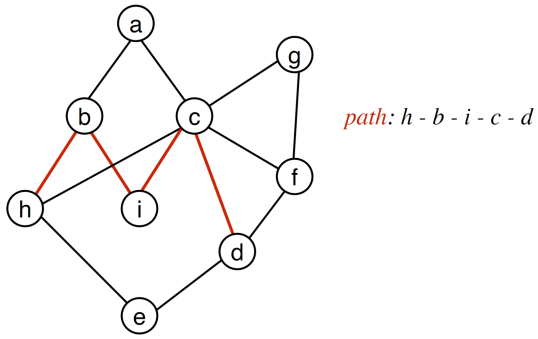


Figure 3.4: A graph with a simple path from h to d.

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.

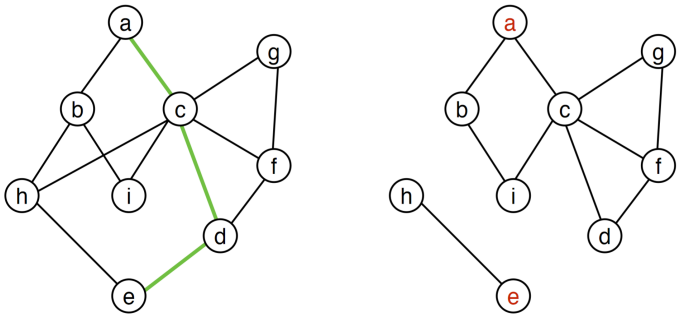


Figure 3.5: A connected graph $a \leftrightarrow c \leftrightarrow d \leftrightarrow e$ and disconnected graph.

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.

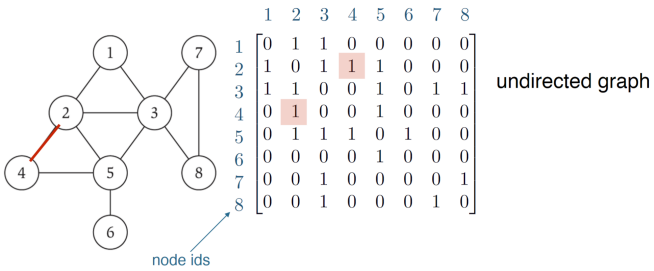


Figure 3.6: An adjacency matrix where the path $4 \leftrightarrow 2$ is highlighted ($A[2][4]$ or $A[4][2]$)

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:** $\Theta(n^2)$.

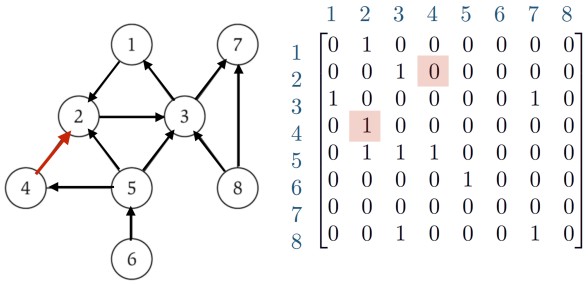


Figure 3.7: An adjacency matrix for a directed graph with path $4 \leftrightarrow 2$ highlighted ($A[2][4]$ and $A[4][2]$).

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:** $\Theta(1)$.

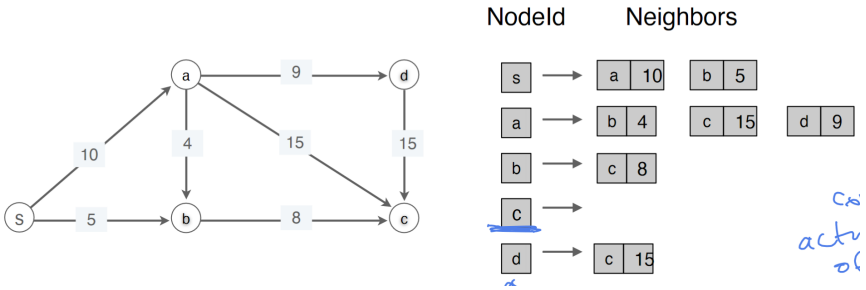


Figure 3.8: An adjacency list of a directed graph, c highlighted with no outdegrees.

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**.

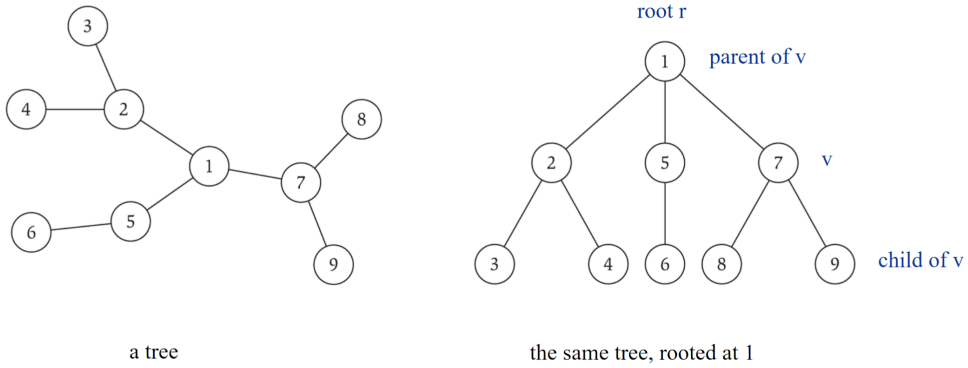


Figure 3.9: A rooted tree with root 1 and children $\{2, 5, 7\}$

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.

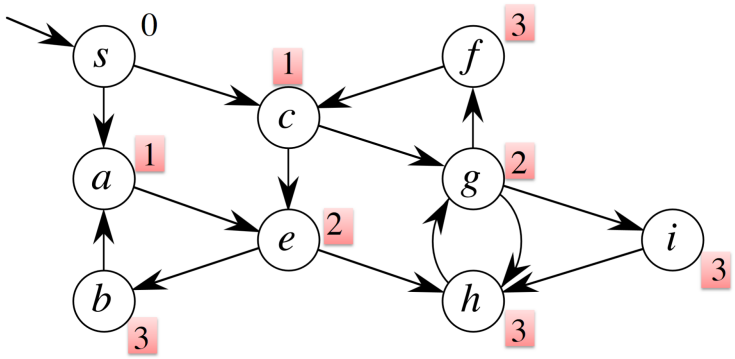


Figure 3.10: A BFS tree traversal preformed on a graph with each level enumerated

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:

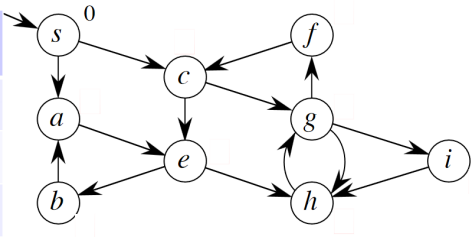
Function 2.1: BFS Algorithm - `BFS(s)`

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      Level[s] ← 0;
5      Add s to Q;
6      while Q not empty do
7          u ← Q.Dequeue();
8          for each v ∈ G[u] do
9              if Level[v] = ∞ then
10                 Add edge (u, v) to tree T (parent[v] = u);
11                 Add v to Q;
12                 Level[v] ← Level[u] + 1;
```

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):

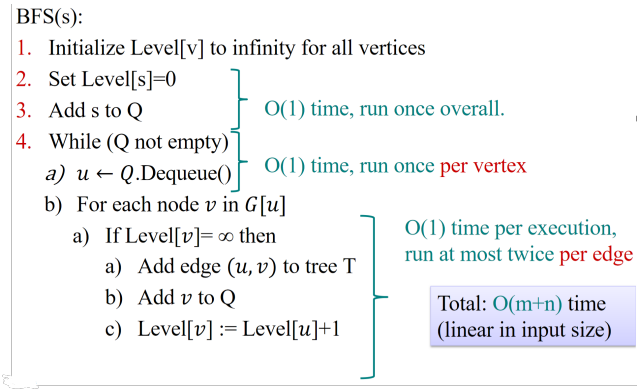


Figure 3.12: An analysis showing $O(m+n)$ for both time and space complexity

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

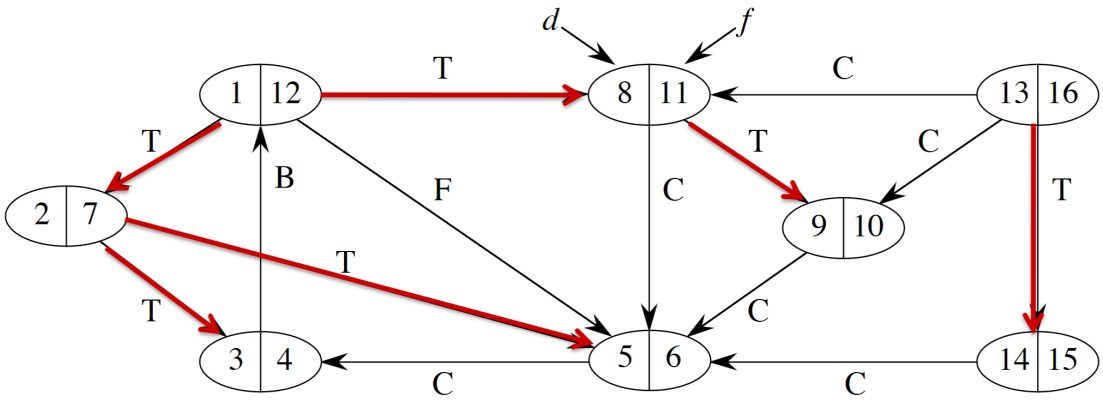


Figure 3.13: A graph showing our d discovered and f finished times, denoted in pairs (f, d).

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.

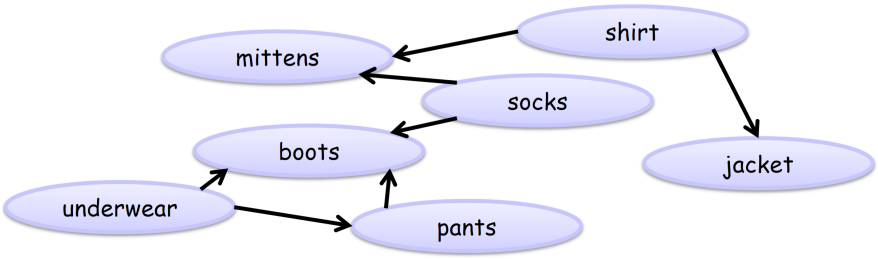


Figure 3.14: A DAG depicted by getting dressed for winter.

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.

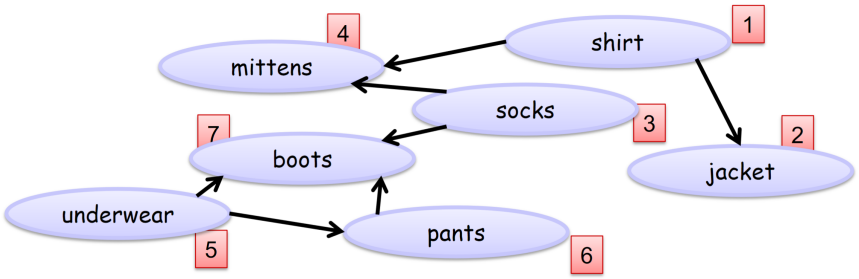


Figure 3.15: A topological ordering of the DAG in Figure (3.14) enumerated in red.

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.

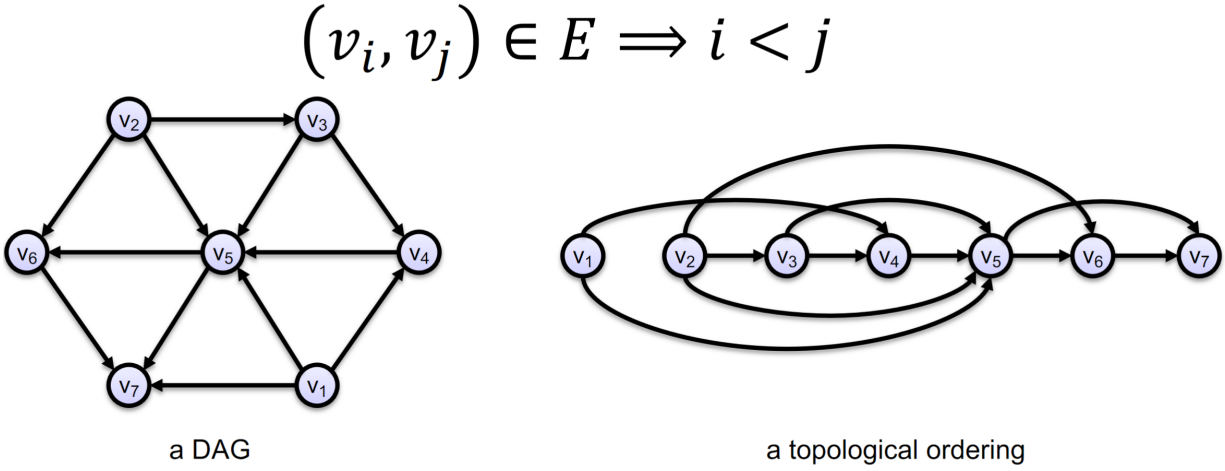


Figure 3.16: A topological sorting of a DAG E and v nodes

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,

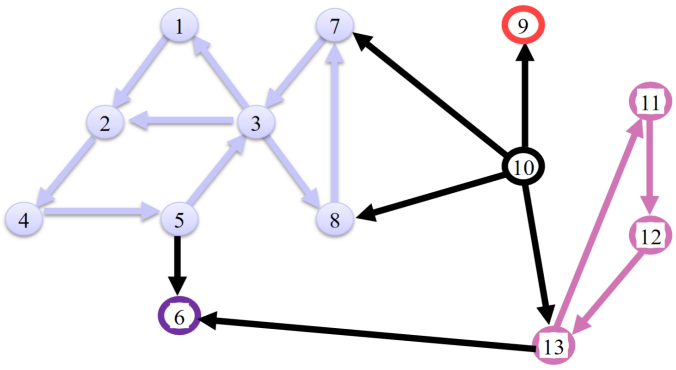
Then in every run of DFS, $u.f > v.f$.
Proof:

- 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.

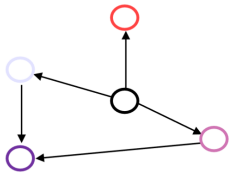


Figure 3.17: A graph with 5 strongly connected components.

— 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.

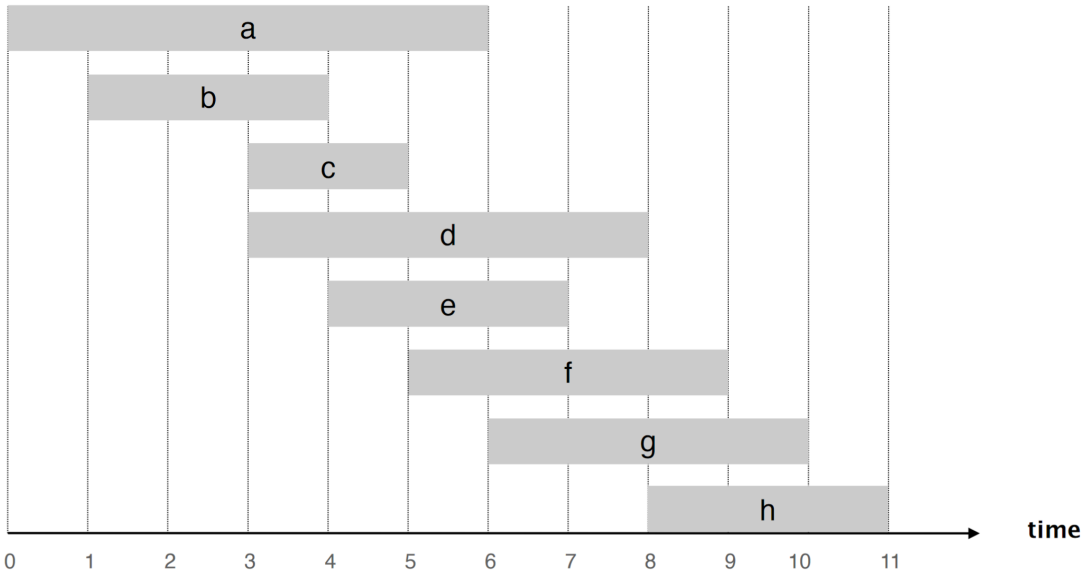


Figure 4.1: Given jobs a through h we find the largest subset of mutually compatible jobs $\{b, e, h\}$.

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. ■

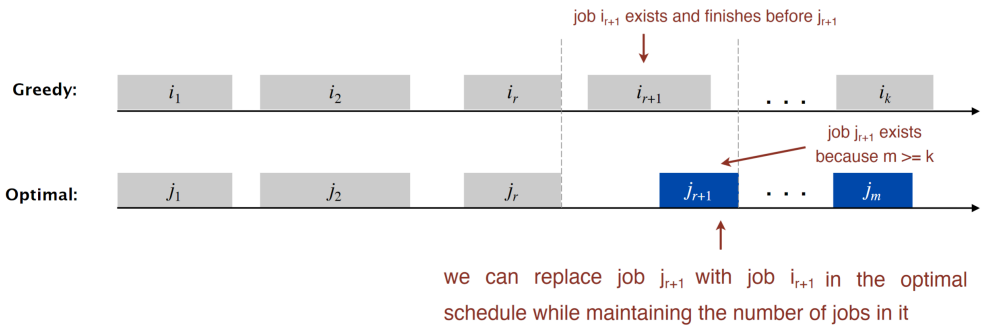


Figure 4.2: Shows that at the first divergence, i_{r+1} and .

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.

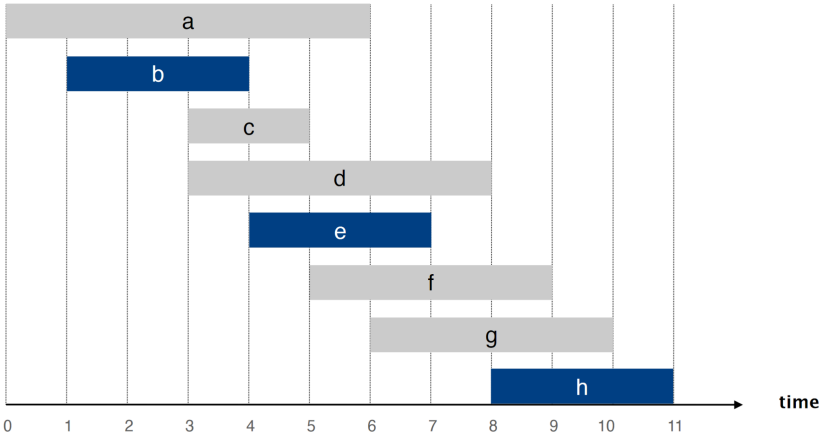


Figure 4.3: Solution to Figure (4.1) using early finish time first, yielding $\{b, e, h\}$.

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 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.

Scenario: ***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.)

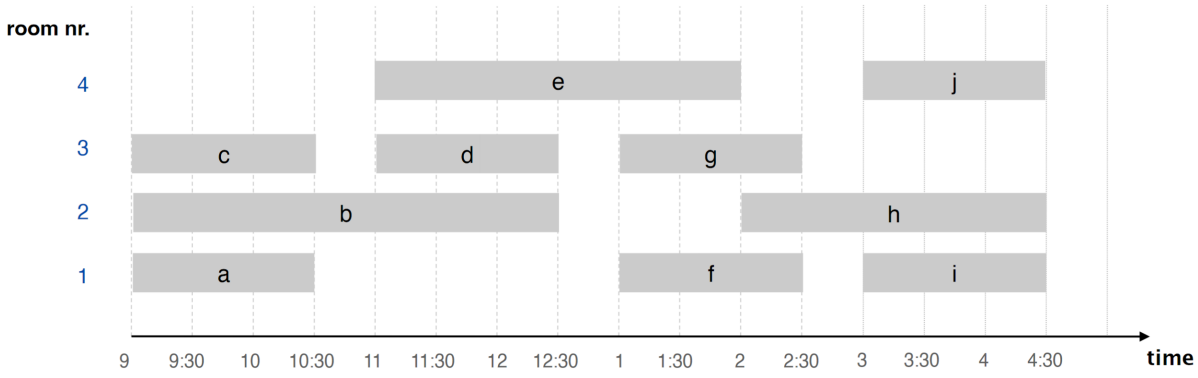


Figure 4.4: Though not optimal, here is a possible schedule where $k = 4$.

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 |   else
7 |   |   d ← len(A)      // highest room id A[d + 1] ← [ ]      // open new room
  |   |     A[d + 1].add(c);
8 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:

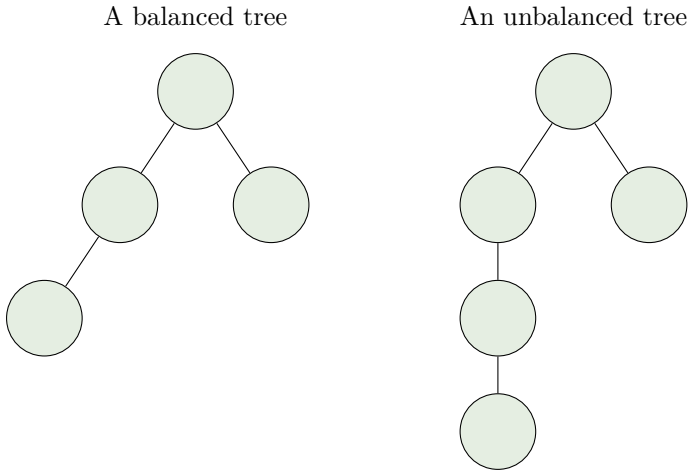


Figure 4.5: Examples of a balanced tree and an unbalanced tree

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  else
5  |  return None;
```

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 |   else
 8 |   |   d ← len(A)       // highest room id A[d + 1] ← [ ]      // open new room
   |   |    A[d + 1].add(c);
 9 |   |   Q.INSERT(⟨f_c, d + 1⟩);
10 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.

Scenario: *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

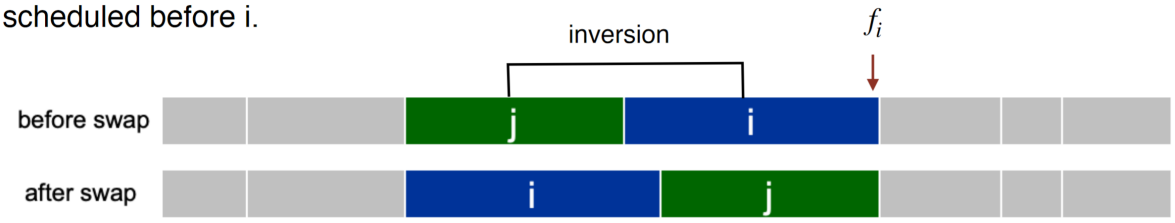
$$\leq \binom{n}{2} = O(n^2)$$

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  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

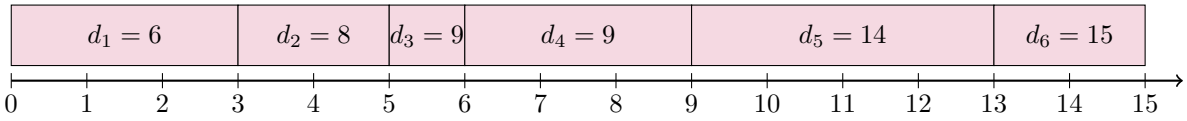


Figure 4.7: Where our number line represents time, and the rectangles the interval of each job.

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:

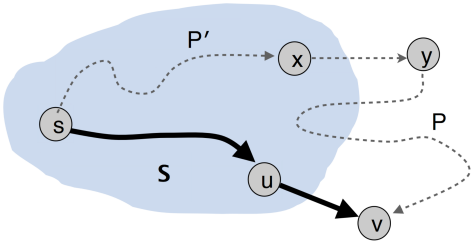


Figure 5.1: A system of subset paths (blue), and an exact point of exit $u \to v$ and $x \to y$

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:

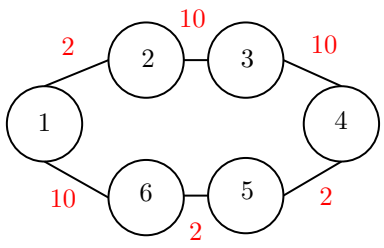


Figure 5.2: A weighted graph.

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	

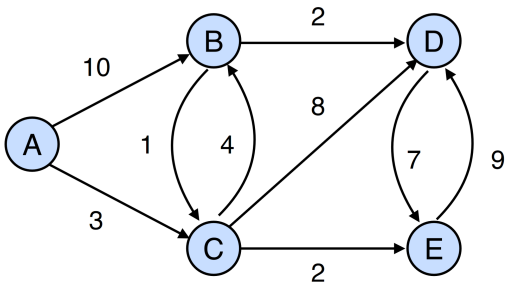


Figure 5.3: A weighted graph and its shortest paths.

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:

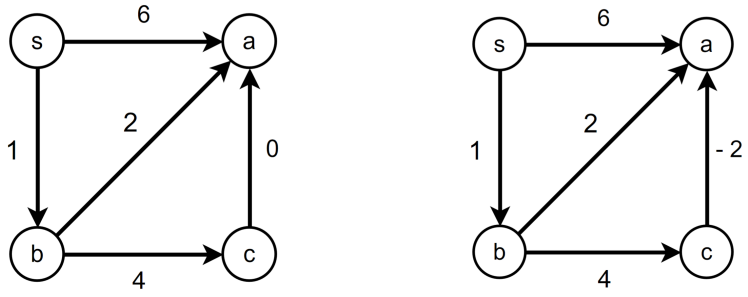


Figure 5.4: Shows two weighted graphs, one positive, and the other negative.

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     while Q is not empty do
12         ⟨π[u], u⟩ ← EXTRACT-MIN(Q);
13         d[u] ← π[u];
14         for v ∈ G[u] do
15             if π[v] > d[u] + l(u, v) then
16                 DECREASE-KEY(⟨π[v], v⟩, ⟨d[u] + l(u, v), v⟩);
17                 π[v] ← d[u] + l(u, v);
18                 parents[v] ← u;
```

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.

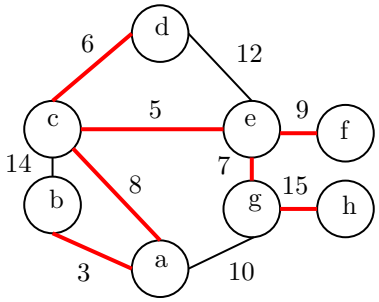


Figure 5.5: Example of an graph with MST highlighted in red.

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**.

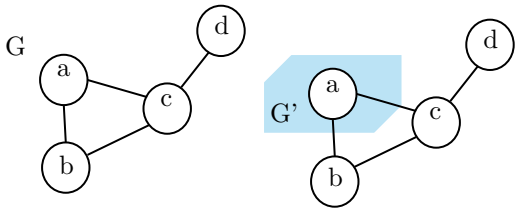


Figure 5.6: Illustration of a graph G and a cut G' of the graph.

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).

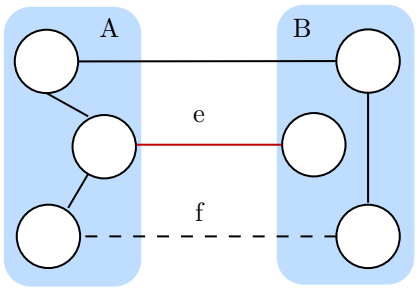


Figure 5.7: A graph cut into two disjoint sets, with a highlighted edge e and a dashed edge f.

Theorem 2.4: Prim's Algorithm

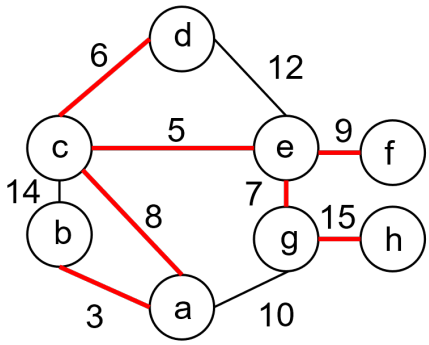
Given a connected graph G with n nodes and m edges, we produce the MST via:

(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 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.

The above diagram shows Prim's algorithm each round, pick the lightest edge at the top of the

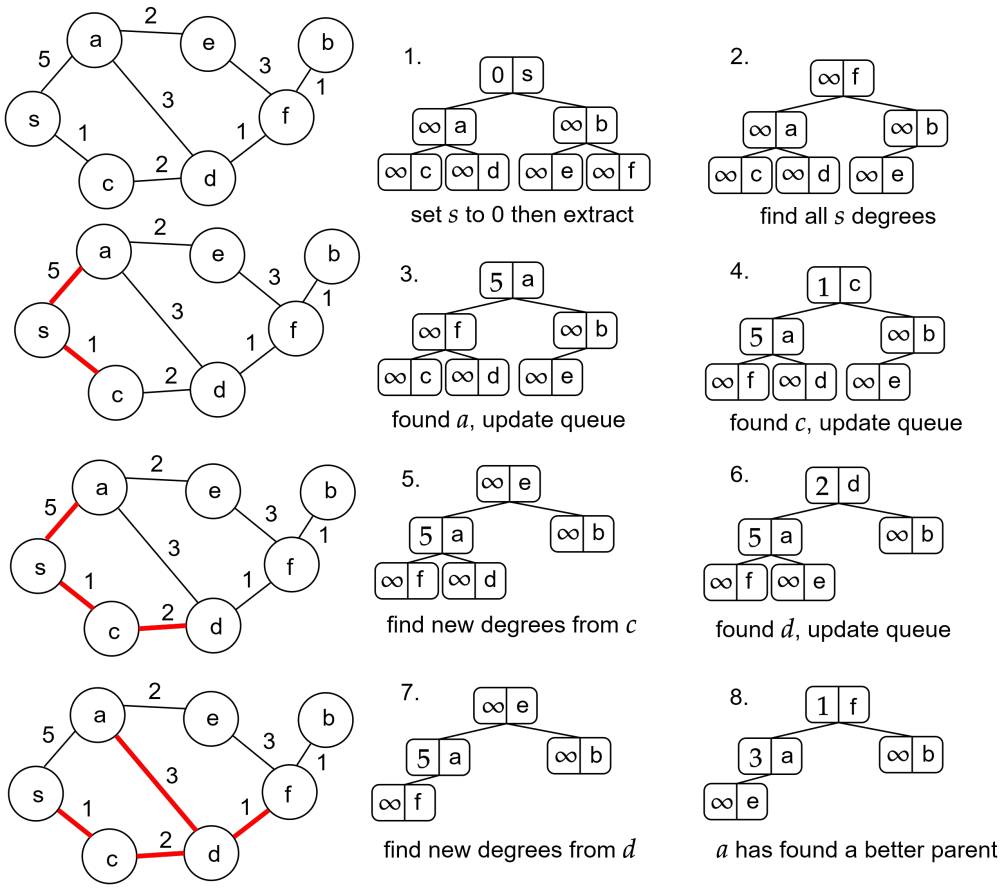


Figure 5.9: A diagram illustrating the pattern of Prim's Algorithm through each iteration.

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.

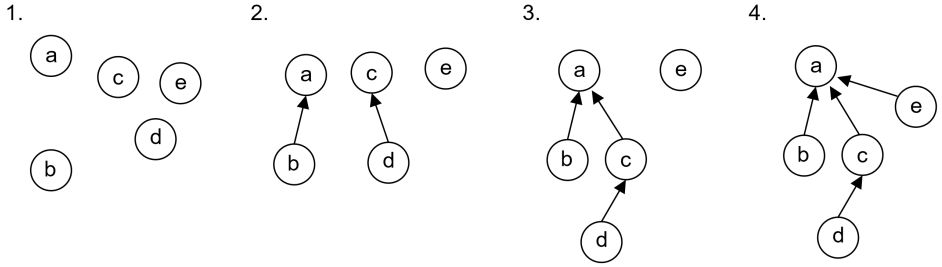


Figure 5.10: Showing disjoint nodes Union-Join with each other.

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.

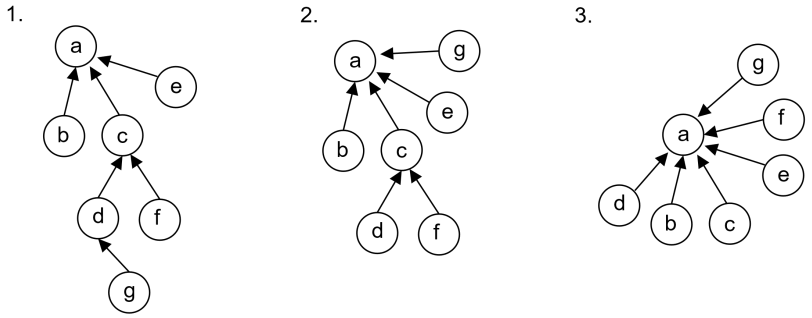


Figure 5.11: Showing a forest compress over multiple finds.

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. After finding the leader of a node in a whole component, we set the node's parent directly to that leader. This reduces the time complexity of find operations for subsequent queries.

We compare all of our techniques in the following table:

Operation\Implementation	**Simple Hash-table**	**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 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

6.1 Inductive Analysis

Recursion employs techniques of repeatedly shrinking the problem to solve its smaller halves. Often found in sorting problems, processing trees, and geometric problems.

Theorem 1.1: MergeSort

Given an array A of size n, we sort via the following:

(i.) Recursively: Split the array into two halves and wait for a return.

(ii.) Base-case: If the array has one element, return it.

(iii.) Merge: receive two halves and merge them.

The final result is a sorted array.

Theorem 1.2: QuickSort

Given an array A of size n, we sort via the following:

(i.) Choose a pivot, set i to the start and j at the end of the array.

(ii.) Increase i until $A[i] > pivot$ and decrease j until $A[j] < pivot$.

- If $A[i] < A[j]$, swap $A[i]$ and $A[j]$.
- Repeat until $i > j$, return j as the new pivot.

(iii.) The new pivot creates two sub-arrays, repeat the process on each sub-array.

The final result is A sorted in-place (on the original array without a temporary helper array).

Tip: Demos for **MergeSort**: https://www.youtube.com/watch?v=4VqmGXwpLqc and **QuickSort**: https://www.youtube.com/watch?v=Hoixgm4-P4M.

Function 1.1: Mergesort - `MSORT(A)`

Input: Array A and temporary array $temp$ of n elements.
Output: Nothing is returned, the array A is sorted by reference.

MSORT function*(A,temp,i,j)*:

```
1  if i ≥ j then
2  |  return;
3  mid ← (i + j)/2;
4  MSORT(A, temp, i, mid); // Left subarray
5  MSORT(A, temp, mid + 1, j); // Right subarray
6  merge(A, temp, i, mid, mid + 1, j); // Merge both halves
```

Merge function*(A,temp,i,leftEnd,j,rightEnd)*:

```
1  i ← lefti; // Left subarray
2  j ← righti; // Right subarray
3  k ← lefti; // Temporary array
4  while i ≤ leftEnd and j ≤ rightEnd do
5  |  if A[i] < A[j] then
6  |  |  temp[k] ← A[i]; i ← i + 1;
7  |  else
8  |  |  temp[k] ← A[j]; j ← j + 1;
9  |  k ← k + 1;
10 while i ≤ leftEnd do
11 |  temp[k] ← A[i]; i ← i + 1;
12 |  k ← k + 1;
13 while j ≤ rightEnd do
14 |  temp[k] ← A[j]; j ← j + 1;
15 |  k ← k + 1;
16 for i = lefti to rightEnd do
17 |  A[i] ← temp[i]; // Copy back sorted elements
```

Time Complexity: $O(n \log n)$ in all cases.
Space Complexity: $O(n \log n)$ for the temporary array and $O(\log n)$ for the recursion stack.

Function 1.2: Quicksort - `QSORT(A,high,low)`

Input: Array A of n elements.
Output: Sorted array A in ascending order.
Initial Call: $QSORT(A, 0, n-1)$

QSORT function*(A,low,high)*

```
1  if low < high then
2  |  pivot ← partition(A, low, high);
3  |  QSORT(A, low, pivot); // Left of the pivot
4  |  QSORT(A, pivot + 1, high); // Right of the pivot
```

Partition function*(A,low,high)*:

```
1  pivot ← A[⌊(low + high)/2⌋];
2  i ← low − 1;
3  j ← high + 1;
4  while i < j do
5  |  repeat
6  |  |  if A[i] ≥ pivot then
7  |  |  |  break;
8  |  until i ← i + 1;
9  |  repeat
10 |  |  if A[j] ≤ pivot then
11 |  |  |  break;
12 |  until j ← j − 1;
13 |  if i < j then
14 |  |  A.swap(i, j);
15 return j; // Return the index of the partition
```

Time Complexity: Average case $O(n \log n)$, Worst case $O(n^2)$.
Space Complexity: $O(n)$ for input. The sorting is done in place, and the recursion stack takes $O(\log n)$ space in the average case. Worst case space complexity is $O(n)$.

Theorem 1.3: Worst Cases - Merge and Quick Sort

- **Merge Sort:** independent of the input, always $O(n \log n)$.
- **Quick Sort:** If the array is sorted in ascending/descending order, the pivot is the smallest/largest element, respectively. This results in $O(n^2)$ time complexity, as each partition is of size $n-1$.

Let us examine merge sort at a high-level.

Function 1.3: Mergesort - `MSORT(A)`

Input: Array A and temporary array $temp$ of n elements.
Output: Nothing is returned, the array A is sorted by reference.

MSORT function:

```
1 if i ≥ j then
2 |   return;
3 mid ← (i + j)/2;
4 MSORT(A, temp, i, mid); // Left subarray
5 MSORT(A, temp, mid + 1, j); // Right subarray
6 merge(A, temp, i, mid, mid + 1, j); // Merge both halves
```

Proof 1.1: Merge Sort - Time Complexity

We make the following observations, generalizing n to each recursive call:

(i.) We make 2 recursive calls.

(ii.) Each recursive call cuts the array in half, $n/2$.

(iii.) We do $\Theta(n)$ work to merge the two halves, returning up the stack.

We define a general form for our traversals as function $T(n)$:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

number of rucursive calls — sub-divisions each frame — work each stack frame

For merge sort, we have $T(n) = 2 \cdot T(n/2) + \Theta(n)$. This means we i with input $T(n)$ and then our first recursive call is $T(n/2)$, the calls from there are $T(n/4)$, $T(n/8)$, and so on. Therefore, at any given layer k, we have 2^k calls, with each input $T(n/2^k)$. We stop when $n/2^k = 1$, which is $k = \log n$. Since $2^{\log_2 n} = n$, then $\left(\frac{n}{2^{\log_2 n}}\right) = 1$. Thus, the depth of our recursion is $\log n$, and n work is done when unraveling the stack, hence $O(n \log n)$.

■

To illustrate the above proof, we can draw a recursion tree for merge sort:

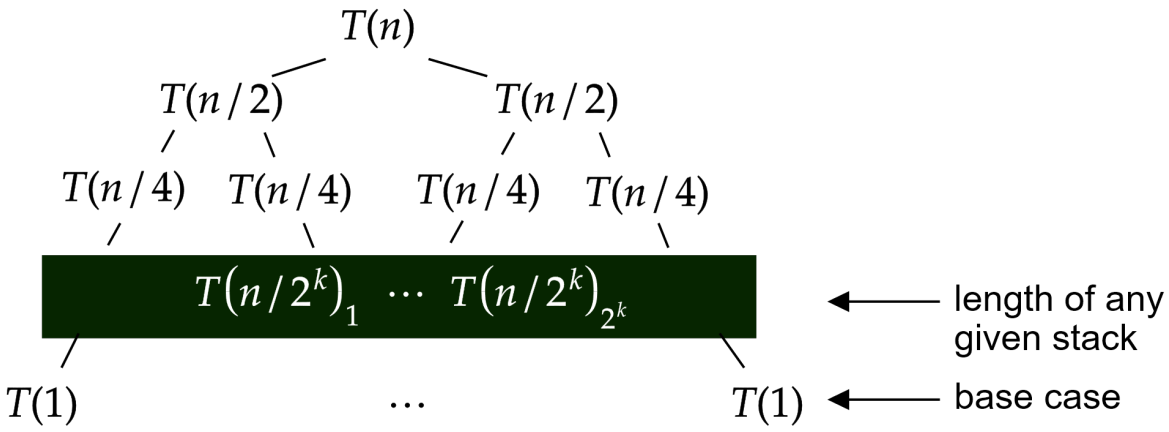


Figure 6.1: Recursion Tree for Merge Sort

Function 1.4: Quicksort - `QSORT(A)`

Input: Array A of n elements.
Output: Sorted array A in ascending order.

QSORT function:

```
1 if low < high then
2 |  pivotIndex ← partition(A, low, high);
3 |  QSORT(A, low, pivotIndex); // Left of the pivot
4 |  QSORT(A, pivotIndex + 1, high); // Right of the pivot
```

Proof 1.2: Quick Sort - Time Complexity

First does $\Theta(n)$ work to partition, then makes 2 recursive calls, and each recursive call has on average $n/2$ elements. We define a general form for our traversals as function $2T(n/2)+\Theta(n)$. Thus $\log n$ levels of recursion, each taking $\Theta(n)$ time to partition, hence $O(n \log n)$.

In worst case, $2T(n-1)+\Theta(n)$. Then we have n levels of recursion, each taking $\Theta(n)$ time to partition, hence $O(n^2)$.

■

Theorem 1.4: Proving Correctness of Recursive Functions

To prove correctness of a recursive function, we need to show:

1. **Base Case:** The base case is correct.
2. **Inductive Hypothesis:** Assume k input sizes are correct, where $k < n$, we assume the recursive calls return the correct result.
3. **Inductive Step:** Show that the problem reduces to the base case, and that intermediate steps other than the recursive calls are correct.

If all three conditions are met, then the function is correct for all n.

Theorem 1.5: Master Method for Recursive Time Complexity

The master method is a general technique for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n^d)$$

where $a \geq 1$, $b > 1$, and $f(n)$ is a given function. We consider degree d of $f(n)$:

$$\begin{aligned} d > \log_b a &\implies T(n) = \Theta(n^d) \\ d < \log_b a &\implies T(n) = \Theta(n^{\log_b a}) \\ d = \log_b a &\implies T(n) = \Theta(n^{\log_b a} \log n) \end{aligned}$$

Tip: Extended Version of the Master Method:

$$T(n) = f(n) + \sum_{i=1}^{k} a_i T(b_i n + h_i(n))$$

where $h_i(n) = O\left(\frac{n}{\log^2 n}\right)$. This is the **Akra-Bazzi Method**:
Link: https://en.wikipedia.org/wiki/Akra%E2%80%93Bazzi_method.

Examples:

- $T(n) = 2T\left(\frac{n}{2}\right) + n^3$: $(a = 2;\ b = 2;\ d = 3;)$ then $(\log_2 2 = 1 < 3)$ hence $T(n) = \Theta(n^3)$.
- $T(n) = 5T\left(\frac{n}{2}\right) + n^2$: $(a = 5;\ b = 2;\ d = 2;)$ then $(\log_2 5 \approx 2.32 > 2)$ hence $T(n) = \Theta(n^{\log_2 5})$
- $T(n) = 16T\left(\frac{n}{4}\right) + n^2$: We have $d := 2$ and $(log_4 16 = 2 = d)$ hence $T(n) = \Theta(n^{log_4 16} \log n)$

Function 1.5: Mergesort - `MSORT(A)`

Input: Array A and temporary array $temp$ of n elements.
Output: Nothing is returned, the array A is sorted by reference.

MSORT function:

```
1 if i ≥ j then
2 |   return;
3 mid ← (i + j)/2;
4 MSORT(A, temp, i, mid); // Left subarray
5 MSORT(A, temp, mid + 1, j); // Right subarray
6 merge(A, temp, i, mid, mid + 1, j); // Merge both halves
```

Proof 1.3: Correctness of Merge Sort

By strong induction on the input size n:

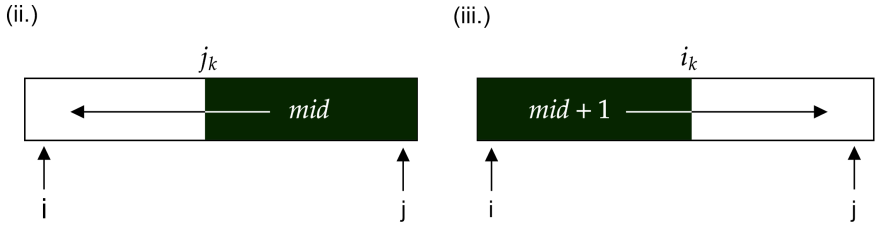
1. **Base Case:** If $i \geq j$, then the array is of size 1, and is already sorted.
2. **Inductive Hypothesis:** Assume that Merge Sort correctly sorts arrays of sizes k where $1 \leq k < n$.
3. **Inductive Step:** Our two recursive calls follow,

   (i.) $mid \leftarrow \lfloor (i + j)/2 \rfloor$

   (ii.) $MSORT(A, temp, i, mid)$

   (iii.) $MSORT(A, temp, mid + 1, j)$

Suppose (ii.) and (iii.) do not reach the base case. Then in (ii.) $mid >= j$ and in (iii.) $mid + 1 <= i$, which contradicts, as $mid := \lfloor (i + j)/2 \rfloor$, then $i \leq mid \leq j$.



Then in (ii.), the right bound mid keeps decreasing, and in (iii.), the left bound mid keeps increasing. This shirks both sub-arrays until both bounds meet. The merge function sorts after both calls, taking the next biggest in the sub-arrays, placing it in the temporary array, and copying back to the original array. Hence, by induction, the function is correct. ■

Theorem 1.6: Iterative Substitution Method (plug-and-chug)

Given a function $T(n)$ which has a recurrence relation—meaning it calls upon itself in its own definition—we may repeatedly substitute such self-references back into $T(n)$.

Given that $T(n)$ properly subdivides to a base case $T(x)$, we may derive some pattern which illustrates the state of $T(n)$ at some depth k within the recurrence. Doing so, we identify what makes our k_{th} expression hit $T(x)$.

Proof 1.4: Iterative Substitution Method (plug-and-chug) - MergeSort

Merge sort has the recurrence relation $T(n) = 2T(n/2) + \Theta(n)$, for which the base case is $T(1)$. We substitute $T(n/2)$ into $T(n)$:

$$\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n); \quad \text{and} \quad T(n/2) = 2T(n/2^2) + \Theta(n/2); \\
T(n) &= 2[\quad\quad] + \Theta(n); \text{ Prepare to substitute the recurrence } T(n/2) \\
&= 2\left[2T(n/2^2) + \Theta(n/2)\right] + \Theta(n); \text{ we evaluate } 2 \cdot \Theta(n/2) \text{ as } \Theta(2n/2) \\
&= 2^2 T(n/2^2) + \Theta(n) + \Theta(n); \text{ Simplified}
\end{aligned}$$

We won't fully simplify to observe how the recurrence builds. We continue, evaluating $T(n/2^2)$:

$$\begin{aligned}
T(n/2^2) &= 2T(n/2^3) + \Theta(n); \\
T(n) &= 2^2\left[2T(n/2^3) + \Theta(n/2^2)\right] + \Theta(n) + \Theta(n); \text{substitute again} \\
&= 2^3 T(n/2^3) + \Theta(n) + \Theta(n) + \Theta(n); \text{ Simplified}
\end{aligned}$$

We identify the pattern for the k_{th} substitution:

$$T(n) = 2^k T(n/2^k) + k \cdot \Theta(n); \text{ General form}$$

Now we identify what makes our recurrence $T(n/2^k) = T(1)$, i.e., where is $n/2^k = 1$, then $n = 2^k$, and $\log_2 n = k$. We plug this back into our general form:

$$\begin{aligned}
T(n) &= 2^{\log_2 n} T(n/2^{\log_2 n}) + \log_2 n \cdot \Theta(n); \text{ Substituting } k \\
&= n\Theta(1) + \log_2 n \cdot \Theta(n); \text{ Where } T(1) = \Theta(1) \\
&= \Theta(n) + \Theta(n)\log_2 n; \\
&= \Theta(n \log n);
\end{aligned}$$

Hence, merge sort has a time complexity of $O(n \log n)$.

■

Tip: Live Demo: https://youtu.be/Ob8SM0fz6p0?si=Z4PZQDWOXa7deEHA

— 7 —

Computational Algorithms

7.1 Computers & Number Base Systems

Definition 1.1: Turing Machine

A **Turing Machine** is a theoretical computational model used to describe the capabilities of a general-purpose **computer**. It consists of an infinite tape (memory) and a read/write head processing symbols on the tape, one at a time, according to a set of predefined rules. The machine moves left or right, reading or writing symbols, and changing states based on what it reads.

The machine **halts** once it reaches a final state or continues indefinitely. Serving as a flexible, **higher-order function** (a function which receives functions).

Definition 1.2: Von Neumann Architecture

Modern computers operate on a model known as the **Von Neumann architecture**, which consists of three primary components:

1. **Memory**: Stores data and instructions as sequences of bits.
2. **Arithmetic and Logic Unit (ALU)**: Executes operations such as addition, subtraction, multiplication, and division on numbers stored in memory.
3. **Control Unit**: Directs the execution of instructions and manages the flow of data between memory and the ALU.

Where numbers are stored in memory cells, each cell holding an integer value represented in a fixed base, typically $B = 2$, meaning **binary**. Where each digit is less than the base B. We represent integers in memory as:

$$a = \sum_{i=0}^{k-1} a_i B^i$$

where a_i represents the individual digits, and B is the base. For large integers, computations may require manipulating several memory cells to store the full number.

Example: Consider the integer $a = 13$, and let us represent it in base $B = 2$ (binary). We can express this number as a sum of powers of 2, corresponding to the binary representation of 13:

$$a = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 0 + 1 = 13$$

In binary, this is represented as the sequence of digits: $a = (1101)_2$. Here, the coefficients $a_3 = 1$, $a_2 = 1$, $a_1 = 0$, and $a_0 = 1$ correspond to the binary digits of 13.
Similarly, if we want to represent $a = 45$ in base $B = 10$ **(decimal)**, we write:

$$a = 4 \cdot 10^1 + 5 \cdot 10^0 = 40 + 5 = 45$$

In this case, the coefficients $a_1 = 4$ and $a_0 = 5$ correspond to the decimal digits of 45.

Definition 1.3: Hexadecimal

Hexadecimal base $B = 16$, using digits 0-9 and the letters A-F, where $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$, and $F = 15$. Hexadecimal is commonly used in computing due to its compact representation of binary data. For example, a **byte** (8 bits) can be represented as two hexadecimal digits, simplifying the display of binary data.

Theorem 1.1: Base $2 \leftrightarrow 16$ Conversion

Let bases $B := 2$ (binary) and $H := 16$ (hexadecimal). At a high-level:

Binary to Hexadecimal:

1. Group B digits in sets of 4, right to left. **Pad** leftmost group with 0's if necessary for a full group.
2. Compute each group, replacing the result with their H digit.
3. Finally, combine each H group.

Hexadecimal to Binary:

1. Convert each H digit into a 4 bit B group.
2. Finally, combine all B groups.

Additionally we may also trim any leading 0's.

Example:

- Binary to Hexadecimal:

$$101101111010_2 \quad \Rightarrow \quad \text{Group as ([1011] [0111] [1010])} \quad \Rightarrow \quad B7A_{16}$$

- Hexadecimal to Binary:

$$3F5_{16} \quad \Rightarrow \quad [0011]\ [1111]\ [0101]_2 \Rightarrow \quad 1111110101_2$$

The following definition is for completeness: applications of such a base are currently uncommon.

Definition 1.4: Unary

Unary, base $B = 1$. A system where each number is represented by a sequence of B symbols. Where the number n is represented by n symbols. Often used in theoretical computer science to prove the existence of computable functions.

Example: The number 5 in unary is represented by 5 symbols: $5 = \texttt{IIIII}$ or $2 = \texttt{II}$. There is no concept of 0, the absence of symbols represents 0.

Definition 1.5: Most & Least Significant Bit

In a binary number, the **most significant bit (MSB)** is the leftmost bit. The **least significant bit (LSB)** is the rightmost bit.

Example: Consider this byte (8 bits), $[1111\ 1110]_2$, the MSB = 1 and the LSB = 0.

Theorem 1.2: Adding Binary

We may use the add and carry method alike decimal addition: **Binary Addition Rules:**

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$ (add 1 to the next digit (left))

We call the last step, **carry**, as we carry our overflow to the next digit.

Example: Adding $0010\ 0011\ 0100_2$ and 0100_2:

$$\begin{array}{r} \overset{1}{} \\ 1\,1000\,0100 \\ +\,0\,0001\,0100 \\ \hline 1\,1001\,1000 \end{array}$$

Where $[1\ 1000\ 0100]_2 + [0\ 0001\ 0100]_2 = [1\ 1001\ 1000]_2$.

Theorem 1.3: Signed Binary Numbers - Two's Complement

In a **two's complement system**, an n-bit signed (positive or negative) binary number can represent values in the range $[-2^{n-1}, 2^{n-1} - 1]$. Then by most significant bit (MSB):

- If MSB is 0, the number is positive;
- If MSB is 1, the number is negative.

Conversion to Two's Complement :

1. Take an unsigned binary number and invert all bits, turning 0's to 1's and 1's to 0's.
2. Finally add 1 to the least significant bit.

Example: Converting -5 into a 4-bit two's complement:

$$\begin{aligned} 5 \to 0101 \quad & \text{(binary for 5)} \\ 1010 \quad & \text{(inverted)} \\ 1011 \quad & \text{(add 1)} \end{aligned}$$

Thus, -5 is represented as 1011 in 4-bits under two's complement.

7.2 Computing Large Numbers

In this section we discuss algorithms for computing large numbers, but first we define algorithmically, addition, subtraction, multiplication, division, and modula.

Definition 2.1: Wordsize

Our machine has a fixed **wordsize**, which is how much each memory cell can hold. Systems like 32-bit or 64-bit can hold 2^{32} (≈ 4.3 billion) or 2^{64} (≈ 18.4 quintillion) bits respectively.

We say the ALU can performs arithmetic operations at $O(1)$ time, within wordsize. Operations beyond this size we deem **large numbers**.

The game we play in the following algorithms is to compute large integers without exceeding wordsize. Moving forward, we assume our machine is a typical 64-bit system.

Function 2.1: Length of digits - $\|a\|$

We will use the notation $\|a\|$ to denote the number of digits in the integer a. For example, $\|123\| = 3$ and $\|0\| = 1$.

Definition 2.2: Computer Integer Division

Our ALU only returns the quotient after division. We denote the quotient as $\lfloor a/b \rfloor : a, b \in \mathbb{Z}$.

Our first hurdle is long division as , which will set up long addition and subtraction for success.
Scenario - Grade School Long Division: Goes as follows, take $\frac{a}{b}$. Find how times b fits into a evenly, q times. Then $a - bq$ is our remainder r.
Examples: let $a = \{12, 5, 17, 40, 89\}$, $b = \{4, 2, 3, 9, 10\}$ respectively, and base $B = 10$,

$$\begin{array}{lrlrlrlrl}
(1.) & \begin{array}{r} 3 \\ 4\overline{)12} \\ 12 \\ \hline 0 \end{array} & (2.) & \begin{array}{r} 2 \\ 2\overline{)5} \\ 4 \\ \hline 1 \end{array} & (3.) & \begin{array}{r} 5 \\ 3\overline{)17} \\ 15 \\ \hline 2 \end{array} & (4.) & \begin{array}{r} 4 \\ 9\overline{)40} \\ 36 \\ \hline 4 \end{array} & (5.) & \begin{array}{r} 8 \\ 10\overline{)89} \\ 80 \\ \hline 9 \end{array}
\end{array}$$

Take (3.), $a = 17$, $b = 3$: 3 fits into 17 five times, which is 15. 17 take away 15 is 2, our remainder. We create an algorithm to compute this process.

Key Observation: Consider the following powers of 2 of form $x = 2^n + s$, where $x, n, s \in \mathbb{Z}$:

$$\begin{aligned}
3 = 2 + 1 &= 0000\ 0011_2 \quad (1) \\
6 = 4 + 2 &= 0000\ 0110_2 \quad (2) \\
12 = 8 + 4 &= 0000\ 1100_2 \quad (3) \\
24 = 16 + 8 &= 0001\ 1000_2 \quad (4) \\
48 = 32 + 16 &= 0011\ 0000_2 \quad (5) \\
96 = 64 + 32 &= 0110\ 0000_2 \quad (6) \\
192 = 128 + 64 &= 1100\ 0000_2 \quad (7)
\end{aligned}$$

Notice that as we increase the power of 2, the number of bits shift left towards a higher-order bit. Now, instead of calculating powers of 2, we shift bits left or right, to yield instantaneous results.

Theorem 2.1: Binary Bit Shifting (Powers of 2)

Let x be a binary unsigned integer. Where "$\ll$" and "$\gg$" are left and right bit shifts:
Left Shift by k bits: $x \ll k := x \cdot 2^k$
Right Shift by k bits: $x \gg k = \lfloor x/2^k \rfloor$
Remainder: bits pushed out after right shift(s).

Example: Observe, $16 = 10000$ (4 zeros), $8 = 1000$ (3 zeros), we shift by 4 and 3 respectively:

- Instead of $3 \cdot 16$ in base 10, we can $3 \ll 4 = 48$, as $3 \cdot 2^4 = 48$.
- Conversely, Instead of $48/16$ in base 10, $48 \gg 4 = 3$, as $\lfloor 48/2^4 \rfloor = 3$.
- Catching the remainder: say we have $37/8$ base 10, then,

$$37 = 100101_2 \quad \text{and} \quad 8 = 1000_2 \text{ then } 37 \gg 3 = 4 \text{ remainder } 5,$$

as $[100101] \gg 3 = [000100]101$, where 101_2 is our remainder 5_{10}.

Function 2.2: Division with Remainder in Binary (Outline) - *QuoRem()*

For binary integers, let dividend $a = (a_{k-1} \cdots a_0)_2$ and divisor $b = (b_{\ell-1} \cdots b_0)_2$ be unsigned, with $k \geq 1$, $\ell \geq 1$, ensuring $0 \leq b \leq a$, and $b_{\ell-1} \neq 0$, ensuring $b > 0$.

We compute q and r such that, $a = bq + r$ and $0 \leq r < b$. Assume $k \geq \ell$; otherwise, $a < b$. We set $q \leftarrow 0$ and $r \leftarrow a$. Then quotient $q = (q_{m-1} \cdots q_0)_2$ where $m := k - \ell + 1$.

Input: a, b (binary integers)
Output: q, r (quotient and remainder in binary)

```
1 Function QuoRem(a, b):
2     r ← a;
3     q ← {0_{m-1} ··· 0};
4     for i ← ||a|| − ||b|| − 1 down to 0 do
5         q_i ← ⌊ r / (b ≪ i) ⌋;
6         r ← r − (q_i · (b ≪ i));
```

Time Complexity: $O(\|a\|(\|a\| - \|b\|))$. In short, line 5 we preform division on $\|a\|$ bits of decreasing size. Though **not totally necessary**, For more detail visit https://shoup.net/ntb/ntb-v2.pdf on page 60. General n cases can be found in Theorem (2.2).

Example Let $a = 47_{10} = 101111_2$ and $b = 5_{10} = 101_2$, we run *QuoRem(a, b)*. We summarize the above example as, "How many times does 101_2 fit into 101111_2?"

1. Does $5 \ll 3$ fit into 101000_2? It fits! $q = 1000_2$, $r = 101111_2 - 101000_2$.
2. Does $5 \ll 2$ fit into 0111_2? no fits! $q = 1000_2$, $r = 0111_2$.
3. Does $5 \ll 1$ fit into 0111_2? no fits! $q = 1000_2$, $r = 0111_2$.
4. Does $5 \ll 0$ fit into 0111_2? It fits! $q = 1001_2$, $r = 0111_2 - 0101$.
5. Return $q = 1001_2 = 9_{10}$, $r = 0010 = 2_{10}$

Long addition: We craft an algorithm for grade school long addition, which goes as follows:

$$\begin{array}{r} \overset{1}{2}5\,\overset{1}{3}08 \\ +\,39\,406 \\ \hline 64\,714 \end{array}$$

Where adding, $25,308 + 39,406 = 64,714$. We create an algorithm to compute this in the following function.

Here the function *QuoRem* (2.2) to return (quotient, remainder) preforms in $O(1)$ as bits are small enough for word size.

Function 2.3: Addition of Binary Integers - *Add()*

Let $a = (a_{k-1} \cdots a_0)_2$ and $b = (b_{\ell-1} \cdots b_0)_2$ be unsigned binary integers, where $k \geq \ell \geq 1$. We compute $c := a + b$ where the result $c = (c_k c_{k-1} \cdots c_0)_2$ is of length $k + 1$, assuming $k \geq \ell$. If $k < \ell$, swap a and b. This algorithm computes the binary representation of $a + b$.

Input: a, b (binary integers)
Output: $c = (c_k \cdots c_0)_2$ (sum of $a + b$)

```
1  Function Add(a, b):
2      carry ← 0;
3      for i ← 0 to ℓ − 1 do
4          tmp ← a_i + b_i + carry;
5          (carry, c_i) ← QuoRem(tmp, 2);
6      for i ← ℓ to k − 1 do
7          tmp ← a_i + carry;
8          (carry, c_i) ← QuoRem(tmp, 2);
9      c_k ← carry;
10     return c = (c_k ··· c_0)_2;
```

Note: $0 \leq carry \leq 1$ and $0 \leq tmp \leq 3$.
Time Complexity: $O(\max(\|a\|, \|b\|))$, as we iterate at most the length of the largest input.
Space Complexity: $O(\|a\| + \|b\|)$, though $c = k + 1$, constants are negligible as $k, \ell \to \infty$.

For subtracting, $5,308 - 3,406 = 1,904$, where we borrow 10 from the 5 to make 13:

$$\begin{array}{r} {\scriptstyle 4\ 10} \\ \not{5}\ 3\,08 \\ -\ 3\ 406 \\ \hline 1\ 904 \end{array}$$

Function 2.4: Subtraction of Binary Integers - *Subtract()*

Let $a = (a_{k-1} \cdots a_0)_2$ and $b = (b_{\ell-1} \cdots b_0)_2$ be unsigned binary integers, where $k \geq \ell \geq 1$ and $a \geq b$. We compute $c := a - b$ where the result $c = (c_{k-1} \cdots c_0)_2$ is of length k, assuming $a \geq b$. If $a < b$, swap a and b and set a negative flag to indicate the result is negative. This algorithm computes the binary representation of $a - b$.

Input: a, b (binary integers)
Output: $c = (c_{k-1} \cdots c_0)_2$ (difference of $a - b$)

```
1  Function Subtract(a, b):
2      borrow ← 0;
3      for i ← 0 to ℓ − 1 do
4          tmp ← a_i − b_i − borrow;
5          if tmp < 0 then
6              borrow ← 1;
7              c_i ← tmp + 2;
8          else
9              borrow ← 0;
10             c_i ← tmp;
11     for i ← ℓ to k − 1 do
12         tmp ← a_i − borrow;
13         if tmp < 0 then
14             borrow ← 1;
15             c_i ← tmp + 2;
16         else
17             borrow ← 0;
18             c_i ← tmp;
19     return c = (c_{k−1} ··· c_0)_2;
```

Note: $0 \leq borrow \leq 1$. Subtraction may produce a borrow when $a_i < b_i$.
Time Complexity: $O(\max(\|a\|, \|b\|))$, iterating at most the length of the largest input.
Space Complexity: $O(\|a\| + \|b\|)$, as the length of c is at most k, with constants negligible as $k, \ell \to \infty$.

For multiplication, $24 \cdot 16 = 384$:

$$\begin{array}{r} {\scriptstyle 2} \\ 24 \\ \times\ 16 \\ \hline 144 \\ +240 \\ \hline 384 \end{array}$$

Where $6 \cdot 4 = 24$, we write the 4 and carry the 2. Then $6 \cdot 2 = 12$ plus the carried 2 is 14. Then we multiply the next digit, 1, we add a 0 below our 144, and repeat the process. Every new 10s place we add a 0. Then we add our two products to get 384.

We create an algorithm to compute this process in the following function:

Function 2.5: Multiplication of Base-B Integers - *Mul()*

Let $a = (a_{k-1} \cdots a_0)_B$ and $b = (b_{\ell-1} \cdots b_0)_B$ be unsigned integers, where $k \geq 1$ and $\ell \geq 1$. The product $c := a \cdot b$ is of the form $(c_{k+\ell-1} \cdots c_0)_B$, and may be computed in time $O(k\ell)$ as follows:

Input: a, b (base-B integers)
Output: $c = (c_{k+\ell-1} \cdots c_0)_B$ (product of $a \cdot b$)

```
1  Function Mul(a, b):
2      for i ← 0 to k + ℓ − 1 do
3          c_i ← 0;
4      for i ← 0 to k − 1 do
5          carry ← 0;
6          for j ← 0 to ℓ − 1 do
7              tmp ← a_i · b_j + c_{i+j} + carry;
8              (carry, c_{i+j}) ← QuoRem(tmp, B);
9          c_{i+ℓ} ← carry;
10     return c = (c_{k+ℓ−1} ··· c_0)_B;
```

Note: At every step, the value of *carry* lies between 0 and $B - 1$, and the value of *tmp* lies between 0 and $B^2 - 1$.
Time Complexity: $O(\|a\| \cdot \|b\|)$, since the algorithm involves $k \cdot \ell$ multiplications.
Space Complexity: $O(\|a\| + \|b\|)$, since we store the digits of a, b, and c.

Function 2.6: Decimal to Binary Conversion - *DecToBin()*

This function converts a decimal number n into its binary equivalent by repeatedly dividing the decimal number by 2 and recording the remainders.

Input: n (a decimal number)
Output: b (binary representation of n)

```
1 Function DecToBin(n):
2     b ← empty string;
3     while n > 0 do
4         r ← n mod 2;
5         n ← ⌊n/2⌋;
6         b ← r + b;
7     return b;
```

Time Complexity: $O(\log n)$, as the number of iterations is proportional to the number of bits in n.
Space Complexity: $O(n)$, storing our input n.

Example: Converting 89 to binary given the above function:

$$\begin{aligned}
89_{10} \div 2 &= 44 \quad \text{rem } 1, \longleftarrow \textbf{ LSB} \\
44_{10} \div 2 &= 22 \quad \text{rem } 0, \\
22_{10} \div 2 &= 11 \quad \text{rem } 0, \\
11_{10} \div 2 &= 5 \quad \text{rem } 1, \\
5_{10} \div 2 &= 2 \quad \text{rem } 1, \\
2_{10} \div 2 &= 1 \quad \text{rem } 0, \\
1_{10} \div 2 &= 0 \quad \text{rem } 1. \longleftarrow \textbf{ MSB}
\end{aligned}$$

Thus, $89_{10} = 1011001_2$.

Theorem 2.2: Time Complexity of Basic Arithmetic Operations

We generalize the time complexity to a and b as n-bit integers.

(i) **Addition & Subtraction:** $a \pm b$ in time $O(n)$.

(ii) **Multiplication:** $a \cdot b$ in time $O(n^2)$.

(iii) **Quotient Remainder** quotient $q := \left\lfloor \frac{a}{b} \right\rfloor : b \neq 0, a > b$; and remainder $r := a \mod b$ has time $O(n^2)$.

7.3 Computational Efficiency

Theorem 3.1: Binary Length of a Number - $\|a\|$

The binary length of an integer a_{10} in binary representation, is given by:

$$\|a\| := \begin{cases} \lfloor \log_2 |a| \rfloor + 1 & \text{if } a \neq 0, \\ 1 & \text{if } a = 0, \end{cases}$$

as $\lfloor \log_2 |a| \rfloor + 1$ correlates to the highest power of 2 required to represent a.

Example: Think about base 10 first. Let there be a 9 digit number $d = 684,301,739$. To reach 9 digits takes 10^8; The exponent plus 1 yields $\|d\|$. Hence, $\lfloor \log_{10} d \rfloor + 1$ is $\|d\|$.

Now, let there be a 7 digit binary number $b = 1001000$, which expanded is:

$$(1 \cdot 2^6) + (0 \cdot 2^5) + (0 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (0 \cdot 2^1) + (0 \cdot 2^0) = 72,$$

Taking 6 powers of 2 to reach 72, we add 1 to get $\|b\| = 7$. Hence, $\|b\| = \lfloor \log_2 b \rfloor + 1$.
Additionally, if $a = 0_2$ then $\|a\| = 1$. as $a^0 = 1$.

Theorem 3.2: Splitting Higher and Lower Bits

Let a be a binary number with n bits. We can split a into two numbers A_1 and A_0 with $n/2$ bits each, representing the first and second halves respectively. Where:

$$A_1 := \frac{a}{2^{\lceil n/2 \rceil}} \quad \text{and} \quad A_0 := a \bmod 2^{\lceil n/2 \rceil}$$

Example: Let's start with base 10. To achieve $A_1 = 7455$ and $A_0 = 62,010$, for $a = 745,562,010$. we take the length $\|a\| := \lfloor \log_{10}(745,562,010) \rfloor + 1 = 9$, as $10^8 \leq 745,562,010 < 10^9$. Then:

$$A_1 = \frac{745,562,010}{10^{\lceil 9/2 \rceil}} = 7455, \quad \text{and} \quad A_0 = 745,562,010 \bmod 10^{\lceil 9/2 \rceil} = 62,010$$

as $10^5 \leq 62,010 < 10^6$. Likewise to finding the remainder in base 2, we can use the same bit shifting technique for base 10 (2.1). We see,

$$[745,562,010]_{10} \text{ right shift by } 5, [000,007,455]_{10} \; 62,010.$$

Hence, $62,010$ is pushed out, and our remainder. Then, we can apply the same technique to base 2. Let $a = 1111\ 1111\ 1001\ 1001_2$. We have $\|a\| := 16$, then:

$$A_1 = \frac{1111\ 1111\ 1001\ 1001_2}{2^{\lceil 16/2 \rceil}} = 1111\ 1111_2, \text{ and } A_0 = 1111\ 1111\ 1001\ 1001_2 \bmod 2^{\lceil 16/2 \rceil} = 1001\ 1001_2$$

Scenario - *Divide and Conquer Multiplication*: We are to compute,

$$A_1 2^{\lceil n/2 \rceil} + A_0 =: a \quad \times \quad b := B_1 2^{\lceil n/2 \rceil} + B_0.$$

Then we have,

$$\begin{aligned} a \cdot b &= (A_1 2^{\lceil n/2 \rceil} + A_0)(B_1 2^{\lceil n/2 \rceil} + B_0) \\ &= (A_1 2^{\lceil n/2 \rceil})(B_1 2^{\lceil n/2 \rceil}) + (A_1 2^{\lceil n/2 \rceil})B_0 + (B_1 2^{\lceil n/2 \rceil})A_0 + A_0 B_0 \\ &= (A_1 B_1)2^n + (A_1 B_0 + B_1 A_0)2^{\lceil n/2 \rceil} + A_0 B_0. \end{aligned}$$

We need to compute 4 products, (A_1B_1), $(A_1B_0), (B_1A_0)$, and (A_0B_0). We now attempt to solve them independently:

Function 3.1: Multiplication of n-bit Integers - *Multiply()*

Let a and b be n-bit integers of base 2. This algorithm recursively computes the product of a and b using a straightforward divide-and-conquer approach, without using Karatsuba's optimization.

Input: n, a, b (where a and b are n-bit integers)
Output: The product $a \times b$

```
1  Function Multiply(n, a, b):
2      if n < 2 then
3          return the result of grade-school multiplication for a × b;
4      else
5          A_1 ← a ÷ 2^{n/2};  A_0 ← a mod 2^{n/2};
6          B_1 ← b ÷ 2^{n/2};  B_0 ← b mod 2^{n/2};
7          p_1 ← Multiply(n/2, A_1, B_1);
8          p_2 ← Multiply(n/2, A_1, B_0);
9          p_3 ← Multiply(n/2, A_0, B_1);
10         p_4 ← Multiply(n/2, A_0, B_0);
11         return p_1 · 2^n + (p_2 + p_3) · 2^{n/2} + p_4;
```

Time Complexity: $O(n^2)$, as in our master method $T(n) = 4T(n/2) + O(n)$, Theorem (1.5).
Space Complexity: $O(n)$, storing $n + n$ bits for a and b, while we track $O(\log_2 n)$ depth in the recursion stack.

We appear to make no improvement, however there's a small trick to reduce the number of multiplications. We continue on the next page.

Observe our full term, $c := (A_1B_1)2^n + (A_1B_0 + B_1A_0)2^{\lceil n/2 \rceil} + A_0B_0$. Say we computed another term,

$$z := (A_1 + A_0)(B_1 + B_0) = (A_1B_1) + (A_1B_0) + (B_1A_0) + (A_0B_0).$$

Notice how z also contains (A_1B_1) and (A_0B_0), which are also in c. Say $m = (A_1B_0) + (B_1A_0)$. Let $x := (A_1B_1)$ and $y := (A_0B_0)$ then $z - x - y = m$. This reduces the number of multiplications to 3, as we only compute (A_1B_1), (A_0B_0) once, and then z.

We employ the above strategy, which is **Karatsuba's multiplication algorithm**:

Function 3.2: Karatsuba's Multiplication Algorithm - *KMul()*

Let a and b be n-bit integers of base 2. This algorithm recursively computes the product of a and b using a divide-and-conquer approach.

Input: n, a, b (where a and b are n-bit integers)
Output: The product $a \times b$

```
1  Function Multiply(n, a, b):
2      if n < 2 then
3          return the result of grade-school multiplication for a × b;
4      else
5          A_1 ← a ÷ 2^{n/2};  A_0 ← a mod 2^{n/2};
6          B_1 ← b ÷ 2^{n/2};  B_0 ← b mod 2^{n/2};
7          x ← Multiply(n/2, A_1, B_1);
8          y ← Multiply(n/2, A_0, B_0);
9          z ← Multiply(n/2, A_1 + A_0, B_1 + B_0);
10         return x · 2^n + (z − x − y) · 2^{n/2} + y;
```

Time Complexity: $O(n^{\log_2 3}) \approx O(n^{1.585})$, as in our master method $T(n) = 3T(n/2) + O(n)$, Theorem (1.5).
Space Complexity: $O(n)$.

— 8 —

Dynamic Programming

8.1 Formulating Recursive Cases

Definition 1.1: Dynamic Programming

In recursive algorithms, there are many cases where we repeat the same computations multiple times. To reduce this redundancy, we store the results of these computations in a table. This is known as **memoization**. This paradigm of programming is called **dynamic programming**.

Scenario - *Fibonacci Sequence*: The Fibonacci sequence is defined as $F_n = F_{n-1} + F_{n-2}$, with $F_0 = 0$ and $F_1 = 1$. Our first 8 terms are $\{0, 1, 1, 2, 3, 5, 8, 13\}$. To compute F_5, we do $0 + 1 = 1$, $1 + 1 = 2$, $1 + 2 = 3$, and $2 + 3 = 5$.

A recursive approach would be:

Function 1.1: Slow Fibonacci Sequence - *Fib()*

Input: n the index of the Fibonacci sequence we wish to compute.
Output: F_n the n_{th} Fibonacci number.

```
1 Function Fib(n):
2 |   if n ≤ 1 then
3 |   |   return n;
4 |   else
5 |   |   return Fib(n − 1) + Fib(n − 2);
```

Time Complexity: $O(2^n)$. Since line 6 depends on both calls, we reflect such in our recurrence relation, $T(n) = T(n-1) + T(n-2) + O(1)$, Theorem (1.5). Since we make calls of size $n-1$ and $n-2$, which are both $O(n)$, we have an exponential time complexity $O(2^n)$.

When we unravel the recursion tree, we see plenty of redundancies:

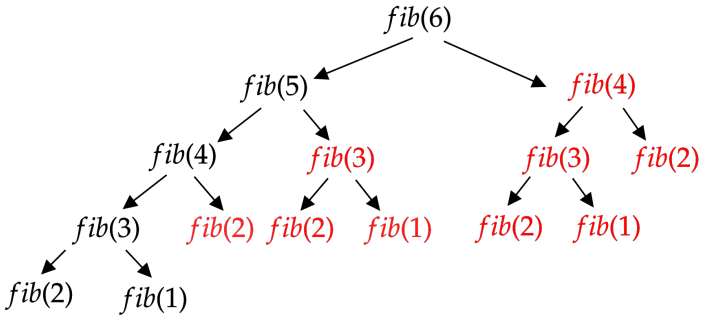


Figure 8.1: Recursion Tree for Fibonacci Sequence

We see that we've already computed F_2 and F_3 multiple times. We can store these values in a table, and use them when needed:

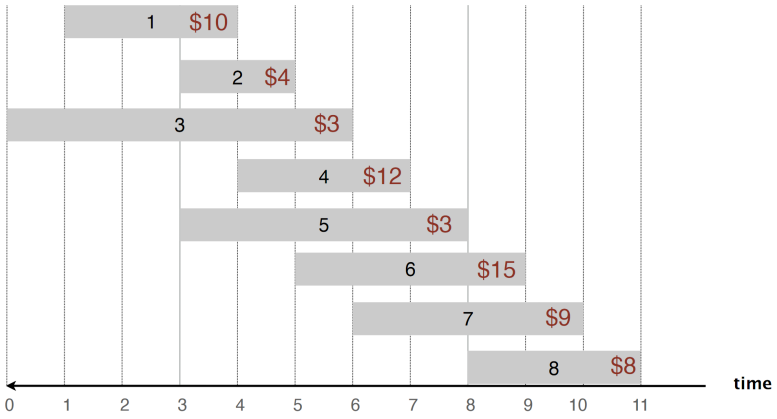
Function 1.2: Memo Fibonacci Sequence - *Fib()*

Input: n the index of the Fibonacci sequence we wish to compute.
Output: F_n the n_{th} Fibonacci number.

```
1 F[ ]; // Table to store Fibonacci numbers
2 Function Fib(n):
3 |   if n ≤ 1 then
4 |   |   return n;
5 |   else
6 |   |   if F[n] is not defined then
7 |   |   |   F[n] = Fib(n − 1) + Fib(n − 2);
8 |   |   return F[n];
```

Time Complexity: $O(n)$. Since we only need to compute F_n once, we at most recurse $n-1$ times. As we unravel we have all our necessary values stored in our table.
Space Complexity: $O(n)$. We store n and recurse at most $n-1$ times.

Scenario - *Weighted Interval Scheduling*: Say we have n paying jobs which overlap each other. We want to find the best set of jobs that allows us to maximize our profit. Recall Section (4.1)



Let us define **OPT(j)** as the maximum profit from jobs $\{1 \dots j\}$, and $\mathbf{v_j}$ as j_{th}'s value. Then $OTP(8)$, considers jobs $1 \dots 8$. Let $p(j) :=$ The largest index $i < j$, s.t., job i is compatible with j. (if none, then $p(j) = 0$). We have two cases:

$$OPT(8) = \begin{cases} OPT(7) & \text{if job 8 is not selected} \\ v_8 + OPT(p(8)) & \text{if job 8 is selected} \end{cases}$$

Then $p(8) = 5$, $p(5) = 0$, yielding \$11, which isn't the optimal solution, see job 6. If we don't choose j, then the optimal solution resides in $\{1 \dots j-1\}$. So we want to know, if our current $OPT()$ solution larger than the next solution. We derive the following cases, and algorithm:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ max\{\underbrace{v_j + OPT(p(j))}_{\text{build solution}}, \underbrace{OPT(j-1)}_{\text{next solution}}\} & \text{else} \end{cases}$$

Function 1.3: Weighted Interval Scheduling - *RecOPT()*

Compute all $OPT(j)$ recursivley, unraveling seeing which $OPT(j)$ is larger; $\mathbf{O(2^n)}$ **Time**.

```
1 Function RecOPT(j):
2     if j = 0 then
3         return 0;
4     else
5         OPT(j) ← max{v_j + RecOPT(p(j)), RecOPT(j − 1)};
6         return OPT(j);
```

Now we employ memoization to store our results in a table, and use them when needed:

Function 1.4: Memo Weighted Interval Scheduling - *OPT()*

```
1  Sort jobs by finish time; // O(n log n)
2  Compute all p(1), ..., p(n); // O(n)
3  OPT[ ]; // Table to store OPT(j)
4  Function OPT(j):
5  |  if j = 0 then
6  |  |  return 0;
7  |  else
8  |  |  if OPT[j] is not defined then
9  |  |  |  OPT[j] ← max{v_j + OPT(p(j)), OPT(j − 1)}; // O(n)
10 |  |  return OPT[j];
```

Time Complexity: $O(n \log n)$, as we are bottle-necked by our sorting algorithm. Line 10 is $O(n)$, following the same memoization pattern as the Fibonacci sequence.

8.2 Bottom-Up Dynamic Programming

In the fibonacci, sequence we don't have to compute it recursively. As shown before we can compute it linearly. Computing F_5, we do $0+1=1$, $1+1=2$, $1+2=3$, and $2+3=5$.

Function 2.1: Bottom-Up Fibonacci Sequence - *Fib()*

```
1  F[0] ← 0; F[1] ← 1; // Base cases (array of size n + 1)
2  for i ← 2 to n do
3  |  F[i] ← F[i − 1] + F[i − 2];
4  return F[n];
```

Time Complexity: $O(n)$. We compute F_n linearly, only needing to compute F_i once.

To offer intuition, recall figure (8.1), we see that that we only really take one branch of the tree. All other branches are redundant. I.e., it's almost as if we have a linear path from the root to the leaf. Hence, there's no need for recursion.

Likewise, we can compute the weighted interval scheduling problem linearly:

Function 2.2: Bottom-Up Weighted Interval Scheduling - *OPT()*

```
1 Sort jobs by finish time; // O(n log n)
2 Compute all p(1), ..., p(n); // O(n)
3 OPT[0] ← 0; // Base case (array of size n + 1)
4 for j ← 1 to n do
5 |   OPT[j] ← max{v_j + OPT[p(j)], OPT[j − 1]};
6 return OPT[n];
```

Time Complexity: $O(n \log n)$. We sort our jobs, and compute $p(1), \ldots, p(n)$ in $O(n)$ time. We then compute $OPT(j)$ linearly, only needing to compute $OPT(j)$ once.

j	0	1	2	3	4	5	6	7	8
OPT(j)	\$0	\$4	\$4	\$10	\$10	\$12	\$19	\$19	\$20

Figure 8.2: Optimal Values for Each Index j

Where,
$OTP[1] \leftarrow max\{v_1 + OPT(p(1)), OPT(0)\} = max\{4+0, 0\} = 4;$
$OPT[2] \leftarrow max\{v_2 + OPT(p(2)), OPT(1)\} = max\{4+0, 4\} = 4;$
$OPT[3] \leftarrow max\{v_3 + OPT(p(3)), OPT(2)\} = max\{10+0, 4\} = 10;$
$OPT[4] \leftarrow max\{v_4 + OPT(p(4)), OPT(3)\} = max\{3+4, 10\} = 10;$
$OPT[5] \leftarrow max\{v_5 + OPT(p(5)), OPT(4)\} = max\{12+0, 10\} = 12;$
and so on.

8.3 Backtracking

Definition 3.1: Backtracking

During recursion we may build solutions based on some number of constraints. During recursion, if we at all, hit some constraint or *dead-end*, we **backtrack** to a previous state to try another path.

Additionally, after a solution is found, we may want to trace which calls lead to our solution. This also called **backtracking**.

Given our weighted interval scheduling (WIS) problem in Figure (8.2), we want to reverse-engineer the jobs that gave us our final solution. Since we have already computed all $OPT(j)$ stored in $OPT[\,]$, and all $p(j)$, we can backtrack to find the jobs that gave us our optimal solution.

Function 3.1: Backtracking Weighted Interval Scheduling - *Backtrack()*

```
   // OPT[ ] (optimal solutions of j job) and p() (next comptible job) are
      already computed for 1,...,j
1  j ← OPT.length − 1; S ← {}; // S is our set of jobs
2  Backtrack(OPT, j);
3  Function Backtrack(OPT, j):
4  |  if v_j + OPT[p(j)] > OPT[j − 1] then
5  |  |  return S ∪ Backtrack(OPT, p(j));
6  |  else
7  |  |  return Backtrack(OTP, j − 1);
```

Correctness: In our example above, the WIS's last index was the optimal solution. However, let $8 = \$1$, then jobs $(6, 1)$ would have been the optimal solution. This leaves $OPT[8] = \$19$, rather than \$20. Line 4 finds the first occurance where we found the optimal solution. As if we first found the optimal solution at index 6, then $6, \ldots, j$ would contain $OPT(6)$. This is why we exclude the choice $(7, 3) = \$19$.

We then repeat such pattern on the next compatible job. We know the set $N := \{1 \ldots p(j)\}$ must contain an element of the optimal solution. Similar to our Dijkstra's proof (5.1), that within the optimal path, a subpath's shortest path is also optimal. We check if $p(j)$ is the first occurance of the optimal solution in N, if not we continue to backtrack.

Time Complexity: $O(n)$. At most, iterate through all n jobs, and add them to our set S.

Notice that our backtracking closely mimics our orginal recursive formula.

8.4 Subset Sum

Weighted Ceiling

Definition 4.1: Problem - Subset Sum (Weighted Ceiling)

Given a set of integers, say $S = \{3, 6, 1, 7, 2\}$, and a target sum $T = 9$, find the max subset P of S, such that $P \leq T$.

We know that when building our solution, we may pick S_i and try all other combinations with $S_{i+1}, \ldots, S_n$ where $n = |S|$. This may cause us to repeat computations as we build our solution. We now know to use dynamic programming to store our results. We start by finding subproblems:

$$S = \{3, 6, 1, 7, 2\}, T = 9; \text{ (choose 3)} \Rightarrow S' = \{6, 1, 7, 2\}, T' = 6$$

Where S' and T' are S and T after removing 3. If we kept $T = 9$, then we'd be asking, "find a max subset P of S', s.t., $P \leq 9$," which isn't our goal. If we decide $3 \in P$, then S' may also contain an optimal contribution (similar to our above Proof (3.1)). While building our solution, if $S_i > T$ we know not to consider it.

We derive the following cases, where T is a changing target sum, and S_i the value of index i (Alternitevely we could use subtraction, instead popping tail elements rather than head elements):

$$OPT(i, T) = \begin{cases} 0 & \text{if } T = 0 \\ OPT(i+1, T) & \text{if } S_i > T \\ max\{\underbrace{OPT(i+1, T)}_{\text{next solution}}, \underbrace{S_i + OPT(i+1, T - S_i)}_{\text{build solution}}\} & \text{else} \end{cases}$$

Moreover, if S_i is compatible with T, we check other solutions. In WIS (8), we only kept track of one changing variable. However, in this case, we change 2 states during each recurrence; Hence our array is two-dimensional:

	Target Sum t									
Index i	0	1	2	3	4	5	6	7	8	9
{}	0	0	0	0	0	0	0	0	0	0
4 ({2})	0	0	2	2	2	2	2	2	2	2
3 ({7,2})	0	0	2	2	2	2	2	7	7	9
2 ({1,7,2})	0	1	2	3	3	3	3	7	8	9
1 ({6,1,7,2})	0	1	2	3	3	3	6	7	8	9
0 ({3,6,1,7,2})	0	1	2	3	4	5	6	7	8	9

Table 8.1: Subset Sum Dynamic Programming Table (DP Table), where $OTP[i][t]$ is the max combination P of $S_i, \ldots, S_n$ s.t., $P \leq t$.

An additional explanation of the above table on the next page.

The above table (8.1), when we reach $i = 4$, we only have $\{2\}$ to consider. This is fine as we've already considered all 2's possible combinations. Observe a nested for-loop approach to find all pairs in $S = \{3, 6, 1, 7, 2\}$:

- Start with 3, then: $\{(3, 6), (3, 1), (3, 7), (3, 2)\}$
- Then with 6, then: $\{(6, 1), (6, 7), (6, 2)\}$
- Then with 1, then: $\{(1, 7), (1, 2)\}$
- Then with 7, then: $\{(7, 2)\}$
- Then with 2, then: $\{2\}$

Notice how we already found all 2's combinations, $(3, 2), (6, 2), (1, 2), (7, 2)$. So even though we only have 2 to consider at $i = 4$, we've already accounted for all possible combinations.

Hence the algorithm below. We change the next and build step to subtraction to allow us to mimic recursion in a bottom up approach. Here we start at the top left progressing forward, rather than the bottom right:

Function 4.1: Subset Sum - *OPT()*

```
1  S; T; OPT[ ][ ]; // Set S, Weight ceiling T, DP table OPT(i, T)
2  OPT[0][*] ← 0; // Base case (array of size 0)
3  OPT[*][0] ← 0; // Base case (array of size T = 0)
4  for i ← 1 to S.length do
5  |   for t ← 1 to T do
6  |   |   if S[i] > t then
7  |   |   |   OPT[i][t] ← OPT[i − 1][t];
8  |   |   else
9  |   |   |   OPT[i][t] ← max{OPT[i − 1][t], S[i] + OPT[i − 1][t − S[i]]};
                                  next solution   build solution
10 return OPT;
```

Time Complexity: $O(nT)$. We iterate through all n jobs, and for each job, we iterate through all T target sums.

The above table (8.1) shows our best possible combination at $OPT[0][9]$; However, we don't know which elements contributed to our solution. We backtrack to find such elements on the next page.

In our table below, ignore the fact that the numbers come out nicely. Where $OPT[0][9] = 9$, could have been $OPT[0][9] = 8$, if we excluded 2 and 3 from our orginal set.

We want to know at each stage, what S_i we picked to obtain T. Just like, in our WIS problem (3.1), we want to know the first occurance of the optimal solution existing. I.e., which S_i was first to contribute to our solution.

Below we give the algorithm to compute such, and an explanation below the table:

Function 4.2: Backtracking Subset Sum - *Backtrack()*

```
1 i ← 0; t ← T; S ← {}; // S is our set of jobs
2 Backtrack(OPT, i, t);
3 Function Backtrack(OPT, i, t):
      // Where OTP.length is the number of rows
4     while i < OPT.length do
5         if OPT[i][t] > OPT[i + 1][t] then
6             S ← S ∪ {S[i]};
7             t ← t − S[i];
8         i ← i + 1;
9     return S;
```

Time Complexity: $O(n)$. At most, iterate through all n jobs, and add them to our set S.

	Target Sum t									
Index i	0	1	2	3	4	5	6	7	8	9
{}	0	0	0	0	0	0	0	0	0	0
4 ({2})	0	0	2	2	2	2	2	2	2	2
3 ({7,2})	0	0	2	2	2	2	2	7	7	9
2 ({1,7,2})	0	1	2	3	3	3	3	7	8	9
1 ({6,1,7,2})	0	1	2	3	3	3	6	7	8	9
0 ({3,6,1,7,2})	0	1	2	3	4	5	6	7	8	9

Here we know 3 combinations did not start our solution, neither did 6 or 1. However, 7 combinations did. We know that $7 \in P$. We reduce T to 2, and our first element to contribute to the optimal solution was 2. We reduce again hitting 0, hence, $P = \{7, 2\}$.

Knapsack

Definition 4.2: Problem - Subset Sum (Knapsack)

Given a set S of n items, each with a weight w_i and value v_i, and a knapsack of capacity W, find a subset P of S, s.t., $\sum_{i \in P} w_i \leq W$ and $\sum_{i \in P} v_i$ is the highest value achievable.

To the right is an example of a knapsack instance. Say we want to solve this by repeatedly taking some combination of weights and their totals. Each time we take an item, we can no longer choose it, and our knapsack weight and value increase. Our base case must be when we run out of items. We also know to step back when we exceed our weight limit.

We want our build step to track the value in some way, decrement our choices, and decrease our weight limit. Then if we don't pick the item, we move to the next item with no change in weight or value. This becomes strikingly similar to the previous subset sum problem with minor adjustments:

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit $W = 11$)

$$OPT(i,w) = \begin{cases} 0 & \text{if } i == 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ max\{\underbrace{OPT(i-1,w)}_{\text{next solution}}, \underbrace{v_i + OPT(i-1, w-w_i)}_{\text{build solution}}\} & \text{else} \end{cases}$$

weight limit w

subset of items 1, ..., i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = max-profit subset of items 1, ..., i with weight limit w.

We achieve the a similar table as we saw before in the subset sum problem.

We skip the implementation as the function is the same as Function (4.1), with the only difference being that we add v_i instead of S_i at build step. Again the runtime is the dimension of our DP table, $O(nW)$.

Function 4.3: Backtracking Knapsack- *Backtrack()*

```
1 i ← OPT.length; w ← W; S ← {}; // S is our set of jobs
2 while i > 0 AND w > 0 do
3 |   if OPT[i][w] > OPT[i − 1][w] then
4 |   |   S ← S ∪ {i};
5 |   |   w ← w − w_i;
6 |   i ← i − 1; // In both cases we i − 1
7 return S;
```

Time Complexity: $O(n)$.

Unbounded Knapsack

Definition 4.3: Problem - Unbounded Knapsack

Given a set S of n items, each with a weight w_i and value v_i, and a knapsack of capacity W, find a subset P of S, s.t., $\sum_{i \in P} w_i \leq W$ and $\sum_{i \in P} v_i$ is the highest value achievable. There are infinite copies of each item.

The same logic applies from the above knapsack problem. However, we can now take an item multiple times, meaning we may have to iterate over all possible choices between each item. We proceed as follows:

$$OPT(i,w) = \begin{cases} 0 & \text{if } i == 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\limits_{j=0,\ldots,\infty} \{\underbrace{OPT(i-1,w)}_{\text{next solution}}, \underbrace{j \cdot v_i + OPT(i-1, w - j \cdot w_i)}_{\text{build solution}}\} & \text{else} \end{cases}$$

At each build step we iterate $j \to \infty$, until $w_i \cdot j > w$. This third step may seem 3-dimensional, as in our $n \times W$ table, each entry has another W entries. However, there's no need to store all W entries, as we only need to store j.

Moreover, we don't want to directly store j in our table OPT as it would overwrite the v_i value. Instead, we keep two tables both $n \times W$, so that we can read both values v_i and its j.

Let us call our choices j table C and our value table M. We illustrate this relationship by imagining the two tables sandwiched on top of each other. Then for every i and w we can reference both values at the same time:

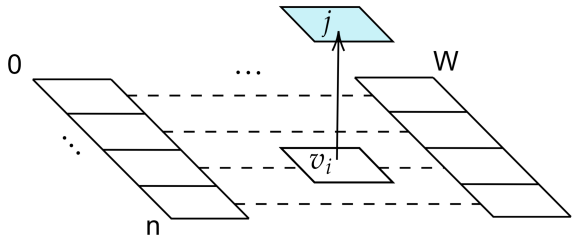


Figure 8.3: Illustrating the relationship between C and M, where v_i relates to its respective j

We implement the following algorithm:

Function 4.4: Unbounded Knapsack - *UnboundedKnapsack()*

```
1  M ← (n + 1) × (W + 1) table // DP table
2  C ← (n + 1) × (W + 1) table // Number of copies
3  M[0][∗] ← 0 and M[∗][0] ← 0;
4  for i ← 1 to n do
5  |  for w ← 1 to W do
6  |  |  M[i][w] ← M[i − 1][w]; // Start with previous solution
7  |  |  m ← 1;
8  |  |  while m · w_i ≤ w do
9  |  |  |  val ← m · v_i + M[i − 1][w − m · w_i];
10 |  |  |  if val > M[i][w] then
11 |  |  |  |  M[i][w] ← val;
12 |  |  |  |  C[i][w] ← m;
13 |  |  |  m ← m + 1;
14 return M, C;
```

Time Complexity: $O(n \times W^2)$ Though we said our tables won't need to be W deep, we still iterate at most W times for each w.
Space Complexity: $O(n \times W)$, as M and C are both $n \times W$, hence $nW + nW = O(nW)$.

Our backtracking algorithm is almost identical to the previous knapsack problem (4.3), however, we now have to consider the number of copies $C[i][w]$.

Function 4.5: Backtracking Unbounded Knapsack - *BKBacktrack()*

```
1 sol ← ∅;
2 i ← n and w ← W;
3 while i > 0 and w > 0 do
4 |   sol ← sol ∪ {i · C[i][w]}; // Add item i with its count C[i][w]
5 |   w ← w − C[i][w] · w_i;
6 |   i ← i − 1;
7 return sol;
```

Time Complexity: $O(n)$, just like before, as we traverse vertically up our $n \times W$ table, finding at what point $M[i][w]$ change. This indicates i was used in the solution.

Unbounded knapsack (Bottom-Up Approach)

Alternatively we can use a bottom-up approach with a slight modification to our recursive formula. Instead of iterating over j, we iterate over w, continuously taking v_j until we exceed w or move onto the next item. This is shown by:

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & \text{if } i = 0, \\ \mathrm{OPT}(i-1, w) & \text{if } w_i > w, \\ \max\left(\mathrm{OPT}(i-1, w), v_i + \mathrm{OPT}(i, w - w_i)\right) & \text{otherwise.} \end{cases}$$

Same scenario, though now we build our solution $n \times W$ starting from the top left, rather than the bottom right. Meaning we start with one item, then two, then three, and so on.

Each iteration we find the largest weight we can take as we grow our constraint w. While growing w if the solution at at iteration i is greater than the last, we take the item. This will consider all possible combinations.

For an exercise, consider the previous table (9), and follow the table left to right, top to bottom.

Function 4.6: Unbounded Knapsack - *UnboundedKnapsack2()*

```
1  M ← (n + 1) × (W + 1) table* // DP table
2  M[0][∗] ← 0 and M[∗][0] ← 0 // Set first row and column to 0
3  for i ← 1 to n do
4  |  for w ← 1 to W do
5  |  |  if w_i > w then
6  |  |  |  M[i][w] ← M[i − 1][w];
7  |  |  else
8  |  |  |  val ← v_i + M[i][w − w_i] // Take one more copy of i
9  |  |  |  if val > M[i − 1][w] then
10 |  |  |  |  M[i][w] ← val;
11 |  |  |  else
12 |  |  |  |  M[i][w] ← M[i − 1][w];
13 return M;
```

Time Complexity: $O(n \times W)$, as we iterate over each item and weight once.

Function 4.7: Traceback Solution for Unbounded Knapsack - *Traceback()*

```
1  i ← n // Start from the last item
2  w ← W // Start from the maximum weight
3  sol ← ∅ // Initialize the solution set
4  while i > 0 and w > 0 do
5  |  if w ≥ w_i and (v_i + M[i][w − w_i] > M[i − 1][w]) then
6  |  |  sol.add(i) // Add item i to the solution
7  |  |  w ← w − w_i // Reduce the weight
8  |  else
9  |  |  i ← i − 1 // Move to the previous item
10 return sol;
```

Time Complexity: $O(n + W)$, as we traverse the DP table in reverse order.

8.5 Shortest Paths - Bellman-Ford Algorithm

Revisiting the shortest path problem (5.1), Dijkstra's failed to account for negative edge weights. This was a result of fixing nodes too early in the algorithm, as it assumes paths can only get larger beyond each point. We look to correct this by considering all possible paths from the source node.

Theorem 5.1: Bellman-Ford Algorithm

Given a connected graph G with n nodes and a source node s, begin with setting every node's distance as ∞ and s as 0. Keep a parent-child list to build our solution. Let $d(v) :=$ "current distance of $s \to v$," and $w(v,u) :=$ "edge-weight $v \to u$." Then, for $n-1$ iterations:

(i.) Iterate over all nodes $v \in G$ starting with s.

(ii.) For each v, iterate out-degrees u, and evaluate $w(v,u)$.

(iii.) If $d(v) + w(v,u) < d(u)$, update $d(u) = d(v) + w(v,u)$.

(iv.) Update parent-child list with u as child of v.

Say we start with s, and examine all out-degrees u. We update their distances $d(s) + w(s,u)$. Now all u nodes have a distance to contribute to their out-degrees. As hash-tables are unordered it is possible we reach a node $d(v) = \infty$, before an in-degree of v updates it. If such happens we skip the node, as it is not reachable from s.

Thus using the idea that the sub-paths of the shortest path are also shortest paths, we gather that each iteration a new shortest path is found. This allows us to find newer shortest paths.

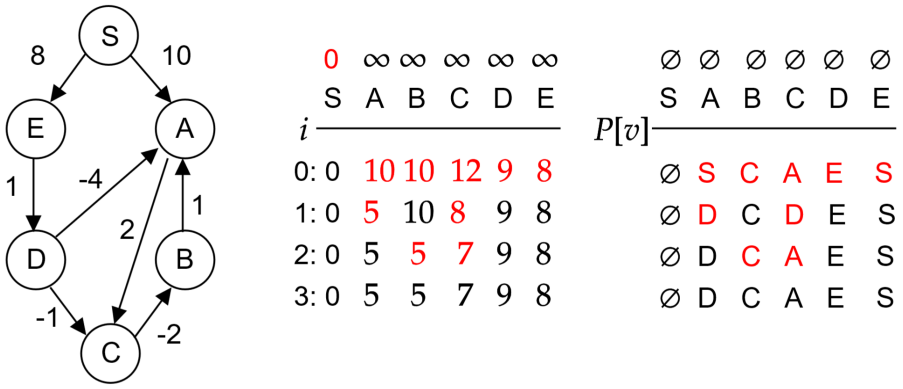


Figure 8.4: Bellman-Ford Algorithm, where i depicts iterations and $P[v]$ parents of v.

Tip: Bellman-Ford Alg. Live Demo: https://www.youtube.com/watch?v=obWXjtg0L64.

Our first iteration goes as follows, $d(S) = 0$. We check $min\{\text{evaluation weight}, \text{current weight}\}$.

- **S:** $d(S) + w(S, A) = 0 + 10; d(A) \leftarrow min\{10, \infty\}$
  $d(S) + w(S, E) = 0 + 8; d(E) \leftarrow min\{8, \infty\}$
- **A:** $d(A) + w(A, C) = 10 + 2; d(C) \leftarrow min\{12, \infty\}$
- **B:** $d(B) = \infty$, currently unreachable from S; skip.
- **C:** $d(C) + w(C, B) = 12 + (-2); d(B) \leftarrow min\{10, \infty\}$
- **D:** $d(D) = \infty$, currently unreachable from S; skip.
- **E:** $d(E) + w(E, D) = 8 + 1; d(D) \leftarrow min\{9, \infty\}$.

Notice how in the second iteration in Figure (8.4), A and C are updated as a direct consequence of us now being able to evaluate paths leaving D. This insight gives us the following theorem:

Theorem 5.2: Bellman-Ford Alg. - Early Termination

We may end the algorithm early if no updates are made in an iteration, as there are no new shortest paths to evaluate.

Though just like Dijkstra's algorithm, Bellman-Ford also has an Achilles' heel. If a negative cycle exists, the algorithm will loop indefinitely, as there will always be a new shortest path. Moreover on the next page.

Definition 5.1: Negative Cycle

A negative cycle is a cycle whose total weight is negative.

Below find the shortest path from $a \rightarrow c$:

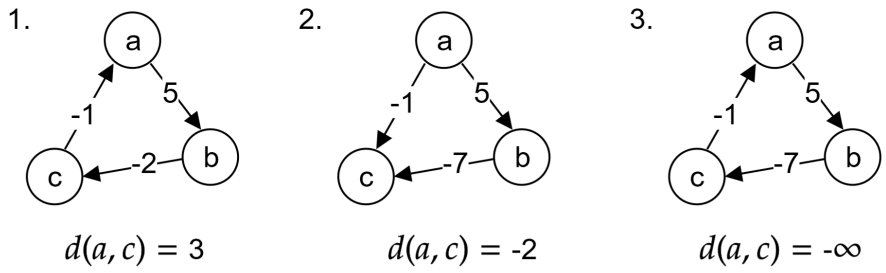


Figure 8.5: Three graphs: 1. A positive cycle, 2. A DAG, 3. A negative cycle.

Examining 3. in Figure (8.5), if we run Bellman-Ford on this graph, we will continuously find a new shortest path to c. We then update the shortest path for $a \rightarrow c$, subsequently updating a new shortest path $a \rightarrow b$, and so on.

Theorem 5.3: Bellman-Ford Alg. - Negative Cycle Detection

To detect a negative cycle, run Bellman-Ford for $n-1$ iterations. If a new shortest path is found in the last iteration, a negative cycle exists. As by then, we should have solidified a solution.

We identify the choices we make in Figure (8.4)'s routine into a recursive formula:

$$OPT(i,v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s, \\ +\infty & \text{if } i = 0 \text{ and } v \neq s, \\ \min \left\{ \begin{array}{l} OPT(i-1,v), \\ \min\limits_{u \in G(v)} (OPT(i-1,u) + w(u,v)) \end{array} \right\} & \text{if } i > 0. \end{cases}$$

Say we are evaluating node v. We first take the last iteration's shortest path $d(v)$ and compare it to possible paths d'. To find such d' we iterate over all in-degrees $u \to v$ and see if $d(u)+w(u,v) < d(v)$. If so, we update $d(v) = d(u) + w(u,v)$.

In figure (8.4) we see on our second iteration, A and C are updated upon evaluating their new in-degree D. Below is pseudo-code for the Bellman-Ford algorithm, returning a DP table M and a parent list *parents*.

Function 5.1: Bellman-Ford - *BellmanFord()*

```
1  G ← Graph // Graph G with n nodes
2  parents ← length-n table // Parents list for shortest paths tree
3  M ← n × n table // DP table M[i][v] = OPT(i, v)
4  M[0][∗] ← ∞ // Set all base values
5  M[0][s] ← 0 // Base case for source
6  for i ← 1 to n − 1 do
7  |  for v ∈ G do
8  |  |  M[i][v] ← M[i − 1][v]; // Copy previous iteration
9  |  |  for u ∈ G[v] do
10 |  |  |  if M[i − 1][u] > M[i][v] + G[v][u] then
11 |  |  |  |  M[i][u] ← M[i][v] + G[v][u];
12 |  |  |  |  parents[u] ← v;
13 return M, parents;
```

Time Complexity: $O(nm)$. We iterate $n-1$ times for $n+m$ edges. Then $O(n(n+m)) = (n^2+nm)$; however, even in tree structures $m = n-1$. So here m could be much larger, s.t., $m \geq n$. Therefore, nm dominates n^2 in the expression. Hence, $O(nm)$.

Let's again examine the above Figure at $i = 1$. Say we reach line 7, with $v = D$. We then copy it's previous iteration, 9, and iterate over it's out-degrees u. We reach $u = A$, where $M[i-1][A] = 10$ and $M[i][D] = 9$. We see $10 > 9 + (-4)$, and update $M[i][A] = 5$ and update $parents[A] = D$.

— 9 —

Network Flow

Network flow techniques are versatile tools with applications spanning various fields. They solve problems like optimizing airline schedules, matching tasks in distributed systems, and detecting network intrusions. From image segmentation to project selection, these methods reduce complex systems into manageable frameworks, showcasing their broad real-world impact.

9.1 Residual Graphs

Definition 1.1: Flow Network

A graph $G = (V, E)$ of V vertices and E edges, carrying a flow of data such that:

1. There is a source node s where data enters the stream.
2. There is a sink node t where data exits the stream.
3. Each edge (u, v) has capacity $c(u, v)$, the maximum flow that can pass through it.

Definition 1.2: Source-Sink Cut (s-t cut)

A partition of a flow graph into two sets A and B such that $s \in A$ and $t \in B$. Namely, $B := \overline{A}$.

Definition 1.3: Capacity

The capacity of a cut $C(A, B)$ is the sum of the edge capacities leaving A to B, i.e.,

$$C(A, B) := \sum_{u \in A, v \in B} c(u, v) \tag{9.1}$$

Below shows a flow network with source s and sink t nodes, with a capacity cut drawn around nodes s and b.

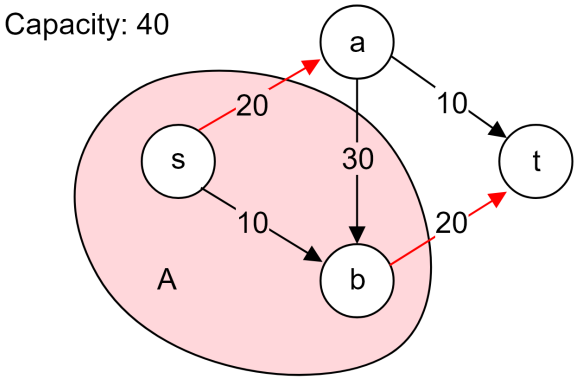


Figure 9.1: Flow Network with Capacity Cut A

Here the sum of the edge weights leaving A is $c(s,a) + c(b,t) = 20 + 20 = 40$.

Definition 1.4: Minimum Cut (Min Cut)

A cut $C(A,B)$ such that the capacity obtained is the minimum possible value for all cuts in the graph.

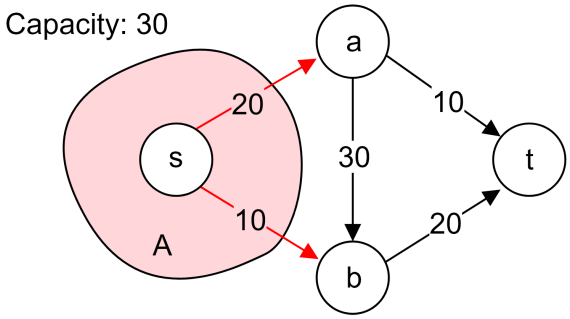


Figure 9.2: Flow Network with Minimum Cut

Where the minimum cut is $C(A,B) = 20$, as $c(s,a) + c(s,b) = 10 + 20 = 30$. Note, despite this cut being around the source node, not all minimum cuts result in this manner.

Definition 1.5: Graph Flow

For edges (u, v) in a graph $G = (V, E)$, the flow $f(u, v)$ denotes the amount of data flowing from u to v. For a valid **s-t flow**, f in G follows the following constraints:

(i) Capacity constraint: for all $(u, v) \in G$, $0 \leq f(u, v) \leq c(u, v)$

(ii) Flow conservation: for all $v \in V \setminus \{s, t\}$, $\sum_{u \to v \in V} f(u, v) = \sum_{v \to w \in V} f(v, w)$

I.e., (i) data flows within capacity limits, and (ii) data isn't created or destroyed in the network.

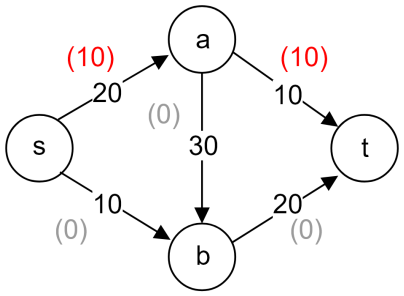


Figure 9.3: Flow Network with Flow a valid flow f, as what comes in, is what comes out.

Definition 1.6: Maximum Flow (Max Flow)

The maximum flow in a network is the maximum amount of data that can be sent from the source to the sink. **Integral Flow,** is a flow where all edge flows are whole integers.

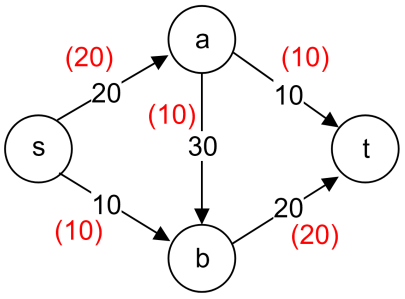


Figure 9.4: Flow Network with Maximum Flow

Definition 1.7: Flow Value Lemma

For any flow f the value $V(f)$ and any s-t cut $C(A, B)$, the net-flow sent across the cut is equal to the flow leaving s, i.e., for $u \in A$ and $v \in B$:

$$V(f) = \sum_{\{u \to v\}} f(u, v) - \sum_{\{u \leftarrow v\}} f(v, u) \tag{9.2}$$

Subtracting flows entering preserves conservation, as what comes in, is sent out.

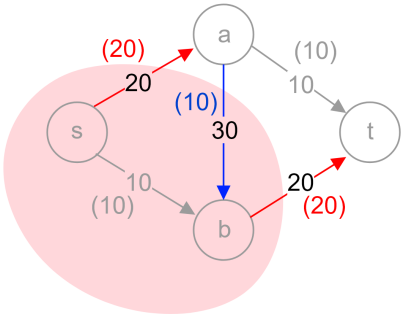


Figure 9.5: An s-t cut with net-flow equaling $V(f)$: $(20 + 20) - (10) = 30 = V(f)$.

Definition 1.8: Weak Duality

For any flow f and any s-t cut $C(A, B)$, the value of the flow is at most the capacity of the cut, i.e.,

$$V(f) \leq C(A, B) \tag{9.3}$$

Definition 1.9: Max Flow-Min Cut Theorem

By weak duality, if $V(f) = C(A, B)$ then f is the max-flow and $C(A, B)$ the min-cut. As by conservation, $V(f)$ cannot get any larger. If $C(A, B)$ weren't the min cut, $V(f) < C(A, B)$ bottle-necked by some other smaller capacity.

One could approach this problem iteratively by **augmenting** the graph by picking the largest path and bottle-necking the flow via the lowest capacity edge.

Though the biggest problem with this approach is that, how do we consider all possible paths? We could use recursion finding. However, this could lead to an inefficiencies or even infinite loops.

When filling out our graph, we may want to keep track of what choices we made, and how much flow we have left to give. For this we introduce the following data structure:

Definition 1.10: Residual Graph

Given a graph $G = (V, E)$ with flow f, we denote its residual graph $G_f = (V, E_f)$, where E_f are a new set of edges generated from augmentations. Between two nodes u and v in G_f, we have the following edges:

- Flow spent from u to v, $f(u, v)$, is shown as a **backward edge** c_b of that capacity,

$$c_b(u, v) := f(u, v)$$

- Flow left to spend is shown as a **forward edge** c_f with capacity of what's left,

$$c_f(u, v) := c(u, v) - f(u, v)$$

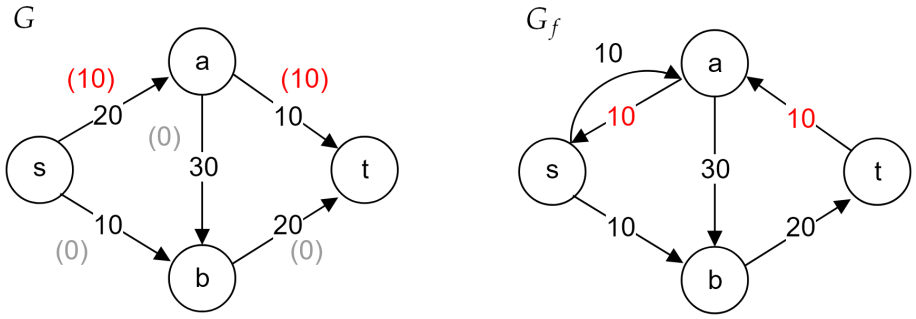


Figure 9.6: A graph G and its residual graph G_f.

Our residual graph tells us, going from s to t, what paths we can take, and how much flow we have left to spend. We harness this into the following algorithm:

Theorem 1.1: Ford-Fulkerson Algorithm (FFA)

Given a flow network $G = (V, E)$, with source s and sink t, the Ford-Fulkerson algorithm finds the max flow by iteratively augmenting the graph.

1. Initialize flow $f(u, v) = 0$ for all $(u, v) \in E$.
2. While there exists a path p from s to t in G_f, augment the flow along p.
3. Return the flow f.

Notably, if a backward edge $c_b(u, v)$ is taken from $s \to t$, then $f(u, v) \in G$ is decreased by $c_b(u, v)$. Additionally, each output is an integral flow.

In essence, each round we find a path from $s \to t$ in G_f, take that path and update both tables G and G_f with the new flow. Do so until we no longer can reach t from s in G_f.

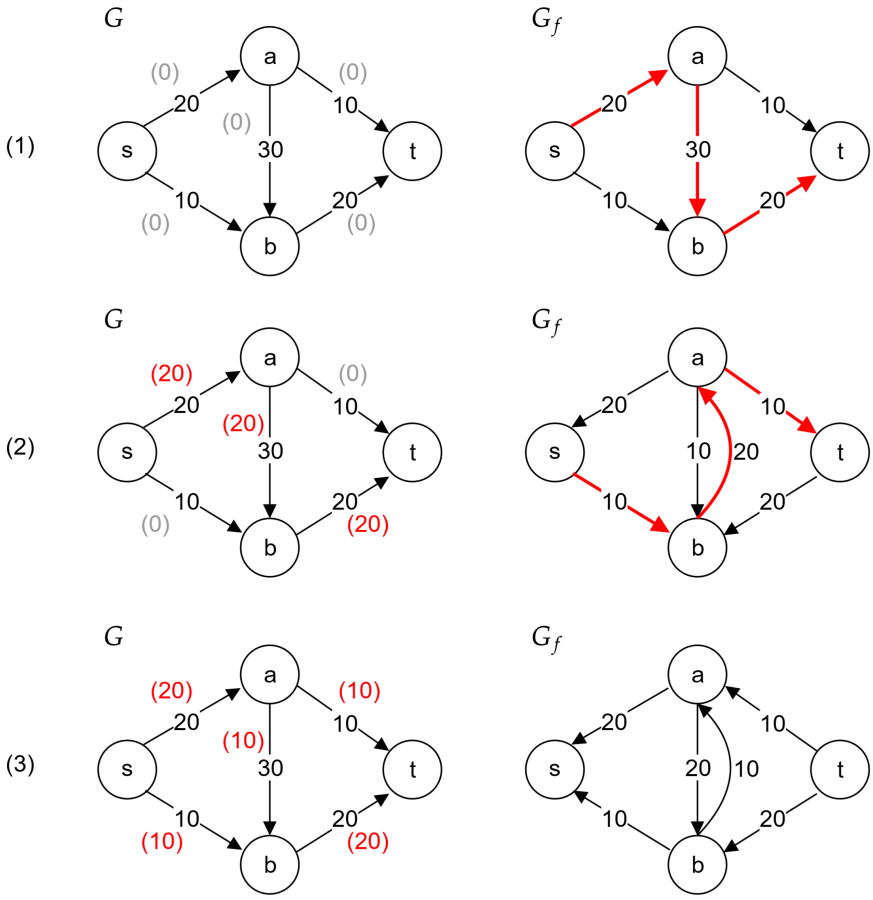


Figure 9.7: Ford-Fulkerson Algorithm in action, terminating at 3 where $s \to t$ is blocked.

Theorem 1.2: Ford-Fulkerson Max flow-Min Cut Consequence

At termination, max flow is obtained. Moreover, nodes from s to blocked nodes b form a cut $C(A, B)$, which is the min cut.

Algorithm on next page.

Function 1.1: Ford-Fulkerson - *Ford-Fulkerson*(G, s, t, c)

Input: G: graph, s: source, t: sink, c: capacities
Output: f: maximum flow

```
1 Function Ford-Fulkerson(G, s, t, c):
2     foreach e ∈ E do
3         f(e) ← 0 // Initialize flow on each edge to zero
4     G_f ← residual(G, c, f) // Construct the residual graph
5     while there exists path P := s → t in G_f do
6         f ← Augment(f, c, P) // Augment the flow along path P
7         Update G_f along P;
8     return f;
```

Time Complexity: $O(mnC)$, where m is the number of edges, n is the number of nodes, and C is the value of the maximum flow. Line 5 takes $O(m)$ time to find an augmenting path in the residual graph. Lines 6-7 take $O(n)$ time to update the flow along the path. Since the flow increases by at least 1 unit in each iteration, and each iteration takes $O(m + n) = O(m)$ time, the while loop runs at most C times. Therefore, the total time complexity is $O(mnC)$.

Function 1.2: Augmentation for Ford-Fulkerson - *Augment*(f, c, P)

Input: f: current flow, c: capacities, P: augmenting path
Output: f: updated flow

```
1 Function Augment(f, c, P):
2     b ← bottleneck(P) // Minimum residual capacity of an edge in P
3     foreach e ∈ P do
4         if e ∈ E then
5             f(e) ← f(e) + b // Update flow on forward edge
6         else
7             f(e) ← f(e) − b // Update flow on reverse edge
8     return f;
```

Time Complexity: $O(n)$, where n is the number of nodes in G. At worst our path P contains all nodes.

9.2 Bipartite Matching

In this section we revisit matching problems alike what we saw with Gale-Shapely, Section (2.1). Here we find the maximum number of matches between two sets of elements, L and R, where each element from both sets may partially match some set of elements from the other.

Definition 2.1: Bipartite Graph

Given a graph $G = (V, E)$, such that $V := L \cup R$, where E connects nodes L to R, a cut which separates V into the two sets L and R is called a **Bipartition**.

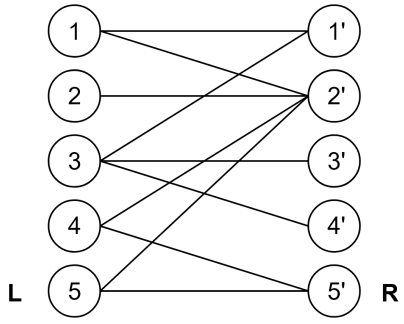


Figure 9.8: A Bipartite Graph $G(V, E)$ cut into two sets $L := \{1, \ldots, 5\}$ and $R := \{1', \ldots, 5'\}$.

Think, if we had water flowing from L to R, what maximum-flow would arise, how many networks could we utilize?

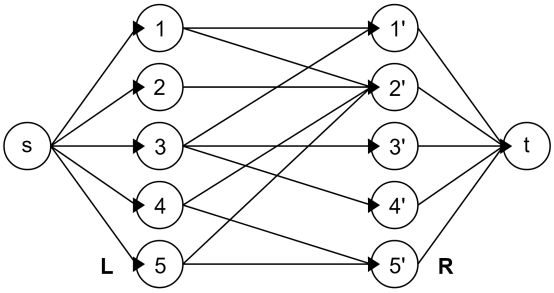


Figure 9.9: Figure (9.8) with a source node s added to L and sink t added to R.

Theorem 2.1: Max-Matching & Max-Flow

Given a bipartite graph $G = (L \cup R, E)$ with a source and sink from $L \to R$, the flow which runs through nodes $L \to R$ is the maximum matching, each edge a capacity of 1.

The below diagram demonstrates a maximum matching achieved through a max-flow algorithm.

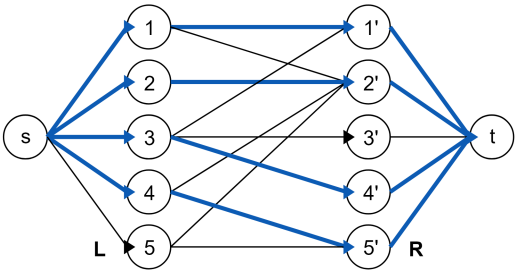


Figure 9.10: A maximum matching found through max-flow.

9.3 Edge-disjoint Paths

Definition 3.1: Edge-disjoint Paths

Given a graph $G = (V, E)$, two paths p_1 and p_2 are edge-disjoint if they share no edges in common. Though, they may share nodes.

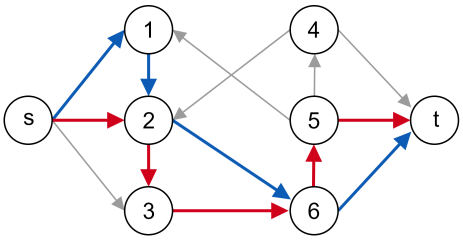


Figure 9.11: Two edge-disjoint paths p_1(red) and p_2(blue) from s to t.

Theorem 3.1: Max-Edge Disjoint Paths

The max-number of edge-disjoint paths from sink to source is the max-flow.

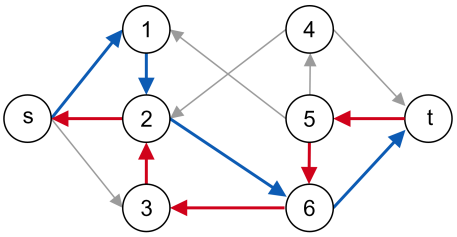


Figure 9.12: A **Residual Graph** of Figure (9.11) after one round of FFA, where p_1(red) is taken.

By the Ford-Fulkerson algorithm, the residual graph blocks the same path from being taken again.

9.4 Multiple Sources & Sinks

Scenario - *Farmer's Dilemma*: There are multiple c communities in the area which require x number of resources, and f farmers. The farmers need know with the routes they have, are they meeting community needs?

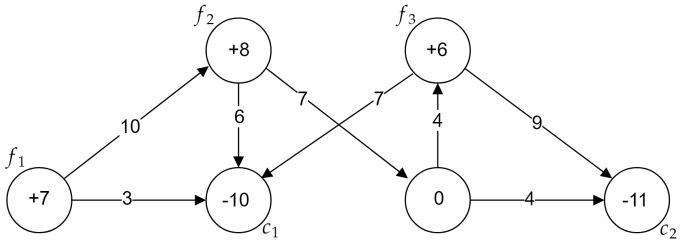


Figure 9.13: A graph showing f_j farms' variation connections to c_k communities. (+) denoting the amount of resources added and (-) the amount of resources taken.

Definition 4.1: Super Source & Sink

Given a graph $G = (V, E)$, with multiple sources $s_1, \ldots, s_f$ and sinks $t_1, \ldots, t_c$. A node which acts as a source for all sources is called a **Super Source**, and a node which acts as a sink for all sinks, a **Super Sink**.

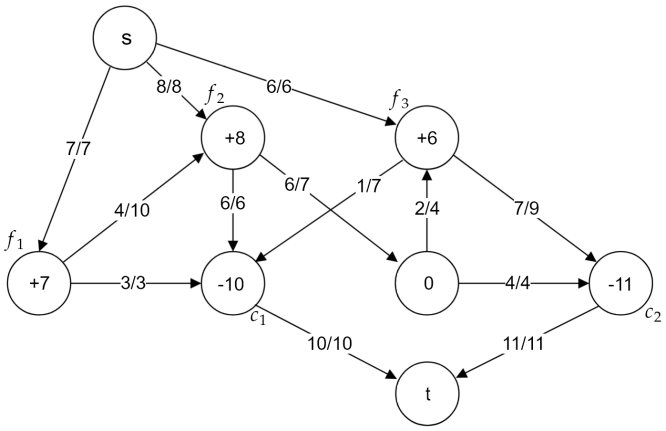


Figure 9.14: A graph with a super source s and super sink t on Figure (9.13).

Converting the graph into a flow network allows us to find the max-flow, via algorithms like Ford-Fulkerson. To verify if demands are met, we check that the edges entering the super sink t are fully saturated.

The strategy is to identify a **Supply** and **Demand** relationship. If this relationship can be established, a bipartite graph can be formed, and a source and sink added to find the max-flow.

Scenario - *Eldritch Curse*: A wizardry school requires j adjunct students on k days to help keep an Eldritch Curse sealed beneath the school. Students W_z, $z := \{1, \ldots, n\}$, give a list A of availability on D_y, $y := \{1, \ldots, m\}$ days. The school needs to know if they have enough students to keep the curse sealed.

We identify: **Suppliers** students W_z, and **demands** as days D_y. We now form a bipartite graph.

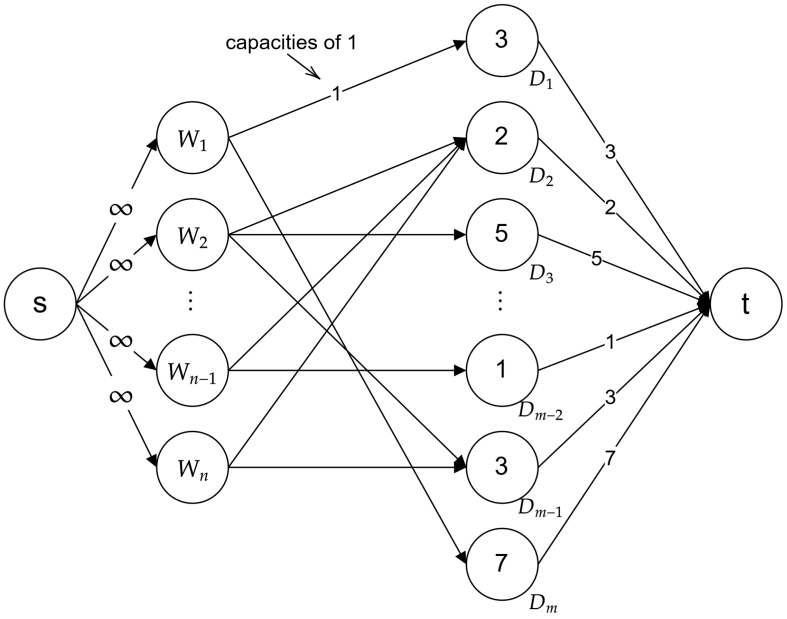


Figure 9.15: A network flow of W_z's availability on D_y nodes showing the students needed.

Here the capacities leaving source node s are infinite, as they will be bottleneck by the students' availability. We match each student to their days all with a capacity of 1. We take all capacities C and flows F entering the sink node t and verify if $\sum_{c \in C} c = \sum_{f \in F} f$, or $\sum_{c \in C} c \leq \sum_{f \in F} f$ to verify that we at least have enough students to keep the curse sealed (allowing flow to be more than the capacity).

Theorem 4.1: Supply & Demand Verification

Given a flow network $G = (V, E)$ with a super source and sinks s and t. If all flows F and capacities C entering t satisfy, $\sum_{c \in C} c = \sum_{f \in F} f$, demands are met. Then **minimum** demands are met if we allow flows to exceed capacities entering t. Then we verify, $\sum_{c \in C} c \leq \sum_{f \in F} f$.

Scenario - *Image Segmentation*: Given an image $I(V, E)$, V pixels, and E neighboring pixels, we want to separate the foreground from the background. We ask a visual specialist to give us three variables:

- $a_i \geq 0$: the likelihood of pixel i being in the foreground.
- $b_i \geq 0$: the likelihood of pixel i being in the background.
- $p_{ij} \geq 0$: the penalty for disconnecting pixels i and j (higher similarities, higher penalties).

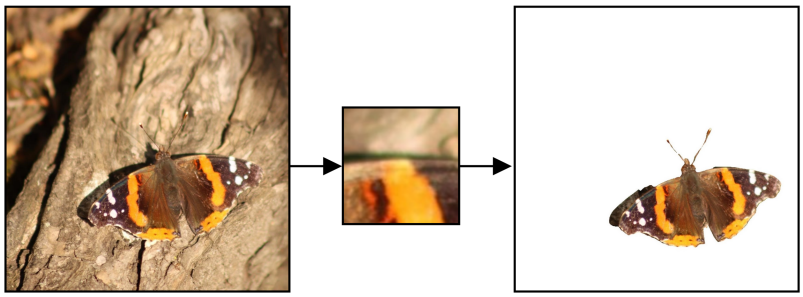


Figure 9.16: A high-level diagram of image segmentation.

We partition the image into two sets, via cut $C(A, B)$, A the foreground and B the background. We the find a partition maximizing the likelihood of pixels in the foreground and background, while minimizing the penalties. Hence,

$$\max\left\{\sum_{i\in A} a_i + \sum_{j\in B} b_j - \sum_{(i,j)\in E} p_{ij}\right\}.$$

However, we don't on hand have an algorithm to solve the maximum cut problem. We can however invert the max-cut into a min-cut problem.

Theorem 4.2: Min-Max Inverse Function

Given a min function $f(x)$, the max function $-f(x)$ is equivalent to the min function $f(-x)$.

We convert the previous function into the following min-cut problem: $\min\left\{-\sum a_i - \sum b_j + \sum p_{ij}\right\}$.
Our Trick: If we add a constant C to our deciding function, the result is still the same, just shifted, as all terms are still the same relative to each other. E.g., $\max\{x\} = \max\{x-C\} = \min\{-x+C\} = \min\{-x\}$, within a constant C. We add back $\sum a_i + \sum b_j$ from the general set V and group terms yielding:

$$\min\left\{\left(\sum_{j\in V} b_j - \sum_{i\in A} a_i\right) + \left(\sum_{i\in V} a_i - \sum_{j\in B} b_j\right) + \sum_{(i,j)\in E} p_{ij}\right\} = \min\left\{\sum_{j\in B} a_j + \sum_{i\in A} b_i + \sum_{(i,j)\in E} p_{ij}\right\},$$

Giving us mis-labeling of a and b in V. Despite the inversion, it retains the same selection.

Network Flow Setup

We take $I(V, E)$ and convert it into a graph $G = (V, E)$, where each neighboring pixel i and j are connected by an edge (i, j), with p_{ij} capacity. In the residual graph G', we add two edges to represent flow from either $i \to j$ and $j \to i$.

Definition 4.2: Anti-parallel Edges

iven an edge e and endpoints i and j, the anti-parallel edges are $i \to j$ and $j \to i$, both independent of each other.

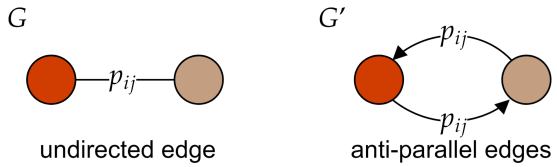


Figure 9.17: Showing an edge in G vs. its anti-parallel edges in G'.

Definition 4.3: Net-flow of Anti-parallel Edges

The net flow of the anti-parallel edges from a residual graph on endpoints i and j is given by,

$$f(i, j) = f(i, j') - f(j, i').$$

This means the flow through endpoints i and j could be negative, zero, or positive. If negative, we have more flow from $j \to i$ than $i \to j$. We now apply this to the Ford-Fulkerson algorithm.

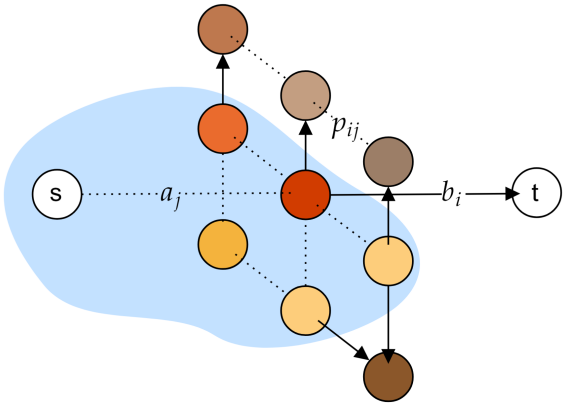


Figure 9.18: A flow network G where the min-cut is highlighted in blue. Solid lines represent edges leaving the min-cut (not all connections from s and t to and from the image are shown).

The derived min-cut yields the optimal partitioning of the image, separating foreground from background.