Computer Science Fundamentals:
Intro to Algorithms, Systems, & Data Structures

Christian J. Rudder

October 2024

Contents

This page is left intentionally blank.

Preface

Big thanks to **Christine Papadakis-Kanaris**
for teaching Intro. to Computer Science II,
Dora Erdos and **Adam Smith**
for teaching BU CS330: Introduction to Analysis of Algorithms
With contributions from:
S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne,
at Boston University

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 $0 \le k \le n$.

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. **Note:** $1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$.

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

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

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, $\underline{f(n) \text{ is } \Theta(n^d).}$
  E.e., $3n^2 + 2n + 1$ is $\Theta(n^2)$.
- **Logarithms.** $\underline{\Theta(\log_a n) \text{ is } \Theta(\log_b n)}$ 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 $d > 0$, $\underline{\log n \text{ is } O(n^d).}$ 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 $d > 0$, $\underline{n^d \text{ is } O(r^n).}$ This means that exponentials grow faster than any polynomial.
  E.e., n^2 is $O(2^n)$.

Proof 1.2: $O(logn) = O(n)$

We prove that $\log_2 n$ is $O(n)$ by induction. Claim: for all $n \geq 1$, $\log_2 n \leq n$.
Inductive step: Assume $n = k$ for some $k \geq 1$. We show that $n = k + 1$ holds.

$$
\begin{aligned}
\log_2(k+1) &\leq \log_2(2k) && \text{(Choosing } 2k \text{ as a convenient upper bound.)} \\
\log_2(2k) &= \log_2 k + \log_2 2 && \text{(Product Rule)} \\
\log_2(2k) &= \log_2 k + 1 && \text{(Simplifying.)} \\
\log_2(k+1) &\leq \log_2 k + 1 && \text{(Substituting.)}
\end{aligned}
$$

Hence, $\log_2(k+1) \leq k + 1$. Thus, by induction, $\log_2 n$ is $O(n)$. ■

Definition 1.8: Time Complexity & Space Complexity

In terms of input size, The **time complexity** measures the run-time of an algorithm. **Space complexity** measures the memory usage of an algorithm. Both are expressed in asymptotic notation.

Function 1.1: 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.

```
Function Fun1(A):
    p ← 0;
    for i ← 1 to n − 1 do
        for j ← i + 1 to n do
            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, $f(n, m) = O(n^2m + m^3)$.

1.2 Simple Sorting Algorithms

This section covers simple sorting algorithms to get a sense of asymptotic analysis. Here we account all structures into our space complexities. If we only accounted for additional space, all the following algorithms would have $O(1)$ space complexity.

Function 2.1: Bubble Sort – `bubble_sort(A)`

Given an array A, of $\{0, \ldots, n\}$ elements. Iterate through the entire list, swapping elements such that if $A[i] > A[i+1]$, swap (Sorting in ascending order). The sort is done when there's nothing left to swap.

Input: An array A of n elements.
Output: The array A sorted in non-decreasing order.

```
n ← |A|;
swapped ← True;
while swapped do
    swapped ← False;
    for i ← 0 to n − 2 do
        if A[i] > A[i + 1] then
            swap A[i] and A[i + 1];
            swapped ← True;
```

Time Complexity: Best: $O(n)$ (already sorted, one pass detects no swaps); Average: $O(n^2)$; Worst: $O(n^2)$ (reverse-sorted). **Space Complexity:** $O(n)$ the array itself.

Function 2.2: Insertion Sort – `insertion_sort(A)`

Given an array A, of $\{0, 1, \ldots, n\}$ elements. Have $A[0]$ be the sorted portion, and $A[1, \ldots, n]$ be the unsorted portion. Take an element from the unsorted, and slide it down the sorted portion till it finds its place.

Input: An array A of n elements.
Output: The array A sorted in non-decreasing order.

```
n ← |A|;
for i ← 1 to n − 1 do
    j ← i;
    while j ≥ 1 and A[j] < A[j − 1] do
        swap A[j] and A[j − 1];
        j ← j − 1;
```

Time Complexity: Best: $O(n)$ (already sorted, no shifts); Average: $O(n^2)$; Worst: $O(n^2)$ (reverse-sorted). **Space Complexity:** $O(n)$ the array itself.

Function 2.3: Selection Sort – `selection_sort(A)`

Given an array A, of $\{0, \ldots, n\}$ elements, take $A[0]$ and iterate to $A[n]$, while doing so maintain the largest element found, and swap with $A[n]$. Now $A[n]$ is sorted, take $A[0]$ and iterate to $A[n-1]$.

Input: An array A of n elements.
Output: The array A sorted in non-decreasing order.

```
n ← |A|;
for i ← 0 to n − 2 do
    minIdx ← i;
    for j ← i + 1 to n − 1 do
        if A[j] < A[minIdx] then
            minIdx ← j;
    swap A[i] and A[minIdx];
```

Time Complexity: Best: $O(n^2)$; Average: $O(n^2)$; Worst: $O(n^2)$.
Space Complexity: $O(n)$ the array itself.

To summarize,

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Table 1.1: Time Complexities of Simple Sorting Algorithms

Tip: Consider the following animations:

- **Bubble Sort:** https://www.youtube.com/watch?v=hahrx5WUeNI
- **Insertion Sort:** https://youtu.be/Q1JdRUh1_98?si=j34DBzZAQkGdMkxU
- **Selection Sort:** https://www.youtube.com/watch?v=Iccmrk2ZWoc

Code examples can be found here: https://github.com/Concise-Works/Algorithms/blob/main/Implementations/alg.py

— 2 —

Memory Management

2.1 CPU Arichitecture

This section provides a high-level overview of the CPU to provide context/motivation for the following algorithms and data structures.

Definition 1.1: Central Processing Unit (CPU)

The **CPU (Central Processing Unit)**, is a hardware component that *computes* instructions within a computer. Abstract models that define interfaces between hardware and software for a CPU are called **instruction set architectures (ISA)**.

Possible operations are detailed as **opcodes** (operation codes), which are numeric identifiers for each instruction. Moreover, the ISA defines supported data types, **registers (temporary storage locations)**, and addressing modes (ways to access memory).

ISA's are defines the instruction set, which allows for flexibility in hardware performance needs. This various categories:

- **CISC (Complex Instruction Set Computing):** Large number of complex instructions (multiple operations per instruction).
- **RISC (Reduced Instruction Set Computing):** Small set of simple/efficient instructions.
- **VLIW (Very Long Instruction Word):** Enables instruction parallelism (simultaneous execution).
- **EPIC (Explicitly Parallel Instruction Computing):** More explicit control over parallel execution.

Smaller more theoretical architectures exists such as **MISC (Minimal Instruction Set Computing)** and **OISC (One Instruction Set Computing)**, which are not used in practice. Popular CPU architectures include x86_64, and ARM64 (64-bit), originating from x86 and ARM (32-bit).

The implementation of a CPU on a circuit board is called a **microprocessor**. Multiple CPUs on a single circuit board are **multi-core processors**, where each *core* is a fully functional CPU.

Definition 1.2: CPU Anatomy: Von Neumann Architecture

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

- **ALU (Arithmetic Logic Unit):** Performs arithmetic and logical operations (e.g., addition, subtraction, AND, OR).
- **Control Unit (CU):** Directs the operation of the CPU, fetching and decoding instructions, and controlling the flow of data.
- **Memory Unit (MU):** Manages data storage and retrieval, including registers and cache memory.

All these components have volatile memory, lost when the computer is turned off.

Definition 1.3: CPU Execution Flow

The **CPU execution flow** is the sequence of operations that the CPU performs to execute a program. It typically follows these steps:

1. **Fetch:** Fetches the next instruction from memory.
2. **Decode:** Decodes the fetched instruction to associated opcode and operands.
3. **Execute:** Perform decoded operation using the ALU or other components.
4. **Store:** Save results of the operation back into memory or registers.

This cycle is repeated until the program completes or an interrupt occurs.

Definition 1.4: Registers

Registers are small, high-speed storage locations within the CPU that hold data temporarily during execution. Common types of registers include:

- **General-Purpose Registers (GPR):** Hold general data storage and manipulation.
- **Special-Purpose Registers:** For specific functions, such as a reference to the current line of code.
- **Floating-Point Registers:** Floating-point arithmetic (e.g., decimal numbers).

Registers are faster than main memory (RAM) and are used to store frequently accessed data during program execution.

The following is an example of the primary registers in the x86-32 (IA-32) architecture, which is a CISC architecture.

Register	**Size**	**Purpose**
EAX	32-bit	Accumulator (arithmetic / return value)
EBX	32-bit	Base register (data pointer)
ECX	32-bit	Counter (loops, shifts)
EDX	32-bit	Data register (I/O, multiply/divide)
ESI	32-bit	Source index (string / memory ops)
EDI	32-bit	Destination index (string / memory ops)
EBP	32-bit	Base/frame pointer (stack-frame anchor)
ESP	32-bit	Stack pointer
EIP	32-bit	Instruction pointer (program counter)
EFLAGS	32-bit	Flags / status register (ZF, CF, OF...)

Table 2.1: Primary registers of the x86-32 (IA-32) architecture. **Note:** Registers are prefixed with 'E' for 32-bit, 'R' for 64-bit in x86-64.

Definition 1.5: Machine Code & Compilation

Code is separated into two main areas of memory management, the program itself, and the data in transit during execution. The program itself is broken up such as follows:

- **Text Segment:** The part of the program which contains the executable code.
- **Data Segment:** The part of the program which contains global and static variables.
- **Machine Code:** The compiled code of the program, which is executed by the CPU.

Once the code compiles, our data segment is further divided into two parts in memory:

- **Initialized Data:** Data given a value before the program starts (global variables).
- **Uninitialized Data:** Data yet to be assigned (local variables), which are zeroed at program start.

By memory we mean the **RAM (Random Access Memory)** hardware component, which stores temporary data, constantly communicating with the CPU or external storage (e.g., hard drive, SSD). Each memory cell is IDed by a unique monotonic **address**, often in hexadecimal format(e.g., 0xF00, 0xF01, etc).

Definition 1.6: Operating System (OS)

Implemented ISAs only provide an interface to the CPU; Programmers must design how their systems utilize the CPU (e.g., file and memory management), such software is called an **operating system (OS)**.

Tip: In an analogous sense, say we have a train riding service. The ISA would be the specifications of the trains, rails, routes, and stations needed. The physical implementation of trains, rails, and stations would be the CPU. The OS would be the train schedule system, managing external factors such as workers and other tasks effecting the train service.

Definition 1.7: The Kernel

The **kernel** is a **process** (a program) vital for OS operation, always running with the highest priority. It is the only program that can directly interact with the CPU and various hardware components.

Other processes running on the system are called **user processes**. This is where applications and other user-level programs run. If a user wishes to perform a task that requires hardware access (e.g., writing/reading files), they must request the kernel called a **system call (syscall)**. System calls provide an **Application Programming Interface (API)** for user processes to interact with the kernel.

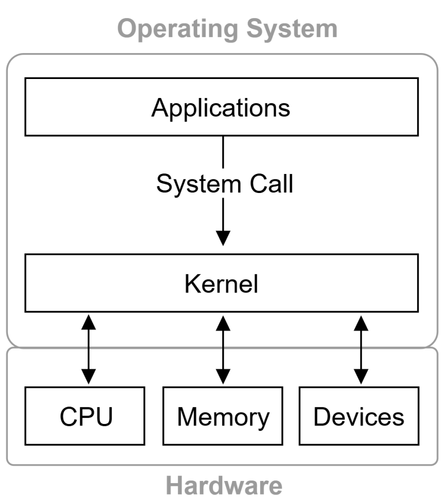


Figure 2.1: User-level applications make syscalls to the kernel to access hardware resources.

Definition 1.8: Bus

A **bus** is a collection of physical signal lines (wires or pins) and protocols that carry data, addresses, and control signals between components inside a computer (e.g. CPU, memory, I/O devices) or between multiple boards and peripherals. There are two main types of buses:

- **Serial Bus:** Transfers data one bit at a time over a single channel (e.g., USB).
- **Parallel Bus:** Transfers multiple bits simultaneously over multiple channels (e.g., PCI).

Definition 1.9: Device Drivers

The kernel exposes generic interfaces to various sub-systems (e.g., file system) that user processes can use to perform tasks; **Device drivers** implement such interfaces, translating generic system calls into hardware-specific operations for specific devices (e.g., disk drives, network cards, etc.). Drivers must be loaded into kernel space.

This text does not concern assembly code, so **do not** get caught in the specifics of this Example:

Example 1.1: Assembly Code

An assembly example demonstrating initialized (.data) and uninitialized (.bss) data sections:

```
section .data                   ; Initialized data section
    num1    dd  7               ; num1 is initialized to 7
    num2    dd  3               ; num2 is initialized to 3

section .bss                    ; Uninitialized data section
    temp    resd 1              ; temp is reserved (uninitialized)
    result  resd 1              ; result is reserved (uninitialized)

section .text                   ; Code section
    global _start

_start:
    mov eax, [num1]             ; Load num1 into eax
    mov [temp], eax             ; Store num1 in temp
    mov ebx, [num2]             ; Load num2 into ebx
    add eax, ebx                ; Add num2 to eax (eax = num1 + num2)
    mov [result], eax           ; Store the sum in result
; Exit syscall removed for simplicity
```

In this example, ‘num1’ and ‘num2’ are initialized before execution, while ‘temp’ and ‘result’ are uninitialized and only receive values during program execution. ■

2.2 Code Security

At a *very* high-level, vulnerabilities exploited by hackers stem from flaws that the programmer forgot to consider (i.e., bugs). To learn more on cybersecurity, consider our other text here.

Definition 2.1: Proper Encapsulation

Proper encapsulation is the practice of hiding implementation details and exposing only necessary interfaces to prevent unauthorized access or modification.

Example 2.1: Student Class

Consider a simple 'Student' class in an object-oriented programming language:

```
public class Student {
    private String name; // Private field, not accessible outside
        the class
    private int age;     // Private field, not accessible outside
        the class

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { // Public method to access name
        return name;
    }
    // Other methods...
}
```

Upon creating a new student instance `new Student("Alice", 20)`, the name and age are private, preventing direct access via **dot notation** (e.g., `student.name`). The only way to access the name is through the public method `getName()`. Here we do not have a method for accessing age. ■

Definition 2.2: Risks of Accessing Main Memory

Programs accesses main memory (RAM) to read and write data; **It's critical** that such references to RAM are abstracted to avoid malicious or accidental access of data.

For example, in Java when users print objects, instead of printing the object's memory address, it prints the `toString()` method, which **by default** prints the class name and hash code of the object.

In conclusion, there are significant risks when dealing with memory management.

2.3 Stack Data Structures

Let's talk about our first data structure, the stack:

Definition 3.1: Stack

A **stack data structure** is a collection of elements that follows a **Last In, First Out** (LIFO) principle. I.e., in a stack of plates, the last added plate is the first one to be removed, not the middle or bottom/first plate. Each *plate* in the stack is called a **stack frame**.

A **call stack** is a stack which keeps track of function calls in a program as well as any local variables within such functions.

This is why we say a variable is in **scope**, as when a function is taken off the stack, or a new stack frame is placed on top, the variables in the previous or discarded stack frame are **no longer accessible**.

My_Program

```
var birth_time
var name

void Main()
        birth_time = time.Now()
        var age = Calculate_Age()
        print(age)

int Calculate_Age()
        var today = time.Now()
        var age = today - birth_time
        return age
```

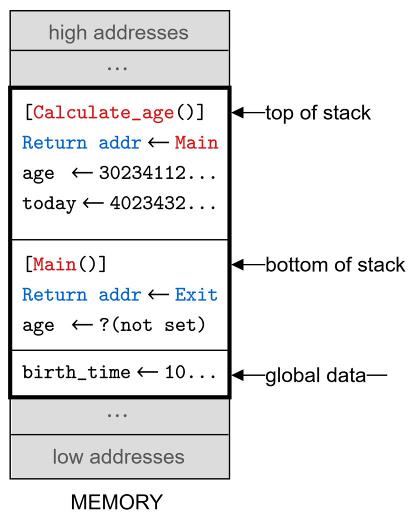


Figure 2.2: Here is a simplified look at how memory manages the stack. On the left is our program written in some abstract language, and on the right is the call stack in memory (simplified). The program has a global 'birth_time' variable, which is initialized in the `Main` function. The `Main` function then calls the `Calculate_Age` function which uses the 'birth_time' variable to calculate the 'age' via the difference of the current time and the 'birth_time.' Looking at the memory, we see at the bottom of our memory contains global variables accessible to any frame. Next, is the bottom of the stack, containing a return address to exit the program, while awaiting the result of the function call for 'age'. The top of our stack contains another frame that we will return the value a new 'age' (not the same as the one before) not accessible from the main function. This new frame also contains a new local variable 'today.' Once this function returns, `Main` will have the result of its local variable 'age.'. Concretely, the 'age' variable in both the `Main` and `Calculate_Age` function are completely separate despite sharing the same *name*.

Please Note: The above figure is a simplified version; This presentation derivatives from what actually happens for teaching sake. In the following pages we define the stack frame in more detail.

Tip: A lot of demonstrations (including this text) will show the stack growing **upwards**; This is strictly because it's easier to visualize and does not accurately portray what a stack really does or looks like. In the following pages we will clear this up, and show how the stack actually grows from top-to-bottom. Of course, there is always room for deviation if a developer wishes to implement a stack in some other arbitrary way. Nonetheless, the following is what one might typically expect in a stack implementation.

Definition 3.2: Stack Frame Anatomy

Under the x86-32 calling convention Two registers keep track our place in the stack:

- **Base Pointer (BP/EBP):** Points to the base (i.e. "bottom") of the current function's stack frame.
- **Stack Pointer (SP/ESP):** Points to the "top" of the current function's stack frame, i.e., the next free byte where a push would land.

When the program starts, the operating system *reserves* a contiguous region of memory for the stack. By convention, the *bottom* of that region lies at a higher address, and the stack "grows downward" toward lower addresses as data is pushed. If the stack pointer ever moves past the reserved limit—a **stack overflow** occurs.

A single **stack frame** itself is a contiguous block of memory in which the function stores:

- **Parameters:** The arguments passed in by the caller,
- **Return Address:** The address of the next instruction to execute after the function returns,
- **Old Base Pointer:** The caller's 'EBP', saved so that on return we can restore the previous frame,
- **Local Variables:** Space for any locals or temporaries that the function needs.

This is why variables in previous or new functions calls become "**out of scope**" (no longer accessible), as they belong to some other stack frame; When it comes to **Global Variables**, they live in a separate region of memory, defined by the **data segment** (1.5).

Moreover, a call to a new function invokes the **call** instruction, this automatically pushes the return address to the current frame onto the stack. Additionally, the CPU reserves the **EAX** register for the return value (number or address) of a function. When the function returns, it can place its result in 'EAX', and the caller can retrieve it from there. During constant use the 'EAX' register may contain **garbage** data from previous use, unless explicitly set to zero or some other value.

High Addresses		
Contents	**Offset**	**Notes**
(Parameters 3, 4, ...)	$EBP + 16, +20, \ldots$	Third-and-onward arguments, if any.
Parameter 2	$EBP + 12$	Second argument passed on stack.
Parameter 1	$EBP + 8$	First argument passed on stack.
Return Address	$EBP + 4$	Auto-pushed by the `call` instruction.
Old EBP (Saved BP)	$EBP + 0$	The caller's base pointer
Current Frame (locals/temporaries)		
Local Variable 1	$EBP - 4$	First 4-byte local (or smallest slot).
Local Variable 2	$EBP - 8$	Next 4-byte local or part of a larger object.
...	$\vdots$	(additional locals at $EBP - 12,\ -16,\ \ldots$)
Low Addresses		

Table 2.2: Typical x86-32 Stack-Frame Layout, where offsets are typically a multiple of 4 bytes.

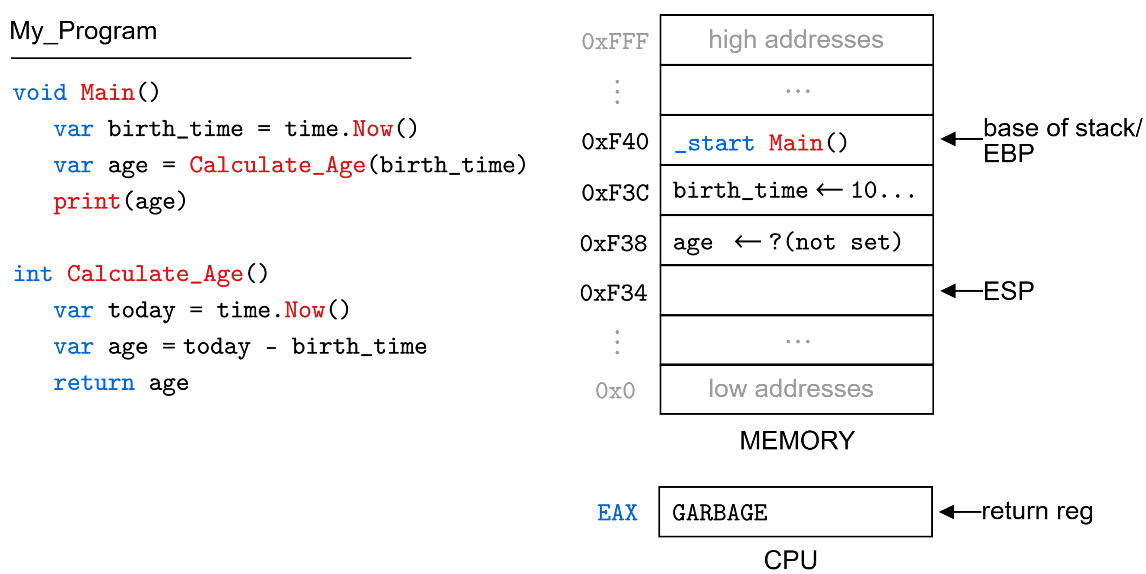


Figure 2.3: Revisiting Figure (2.2) with slight alterations to the code: This is a snapshot of the code executing right before `CalculateAge(birth_time)` is called. For simplicity sake, let's say the stack begins at address `0xF40` (Hexadecimal), growing downwards. Here the base of the stack and the EBP are one and the same. We include the CPU's EAX (return register), which contains garbage. Address `0xF38` is currently just reserved space for 'age'.

My_Program

```
void Main()
   var birth_time = time.Now()
   var age = Calculate_Age(birth_time)
   print(age)

int Calculate_Age()
   var today = time.Now()
   var age = today - birth_time
   return age
```

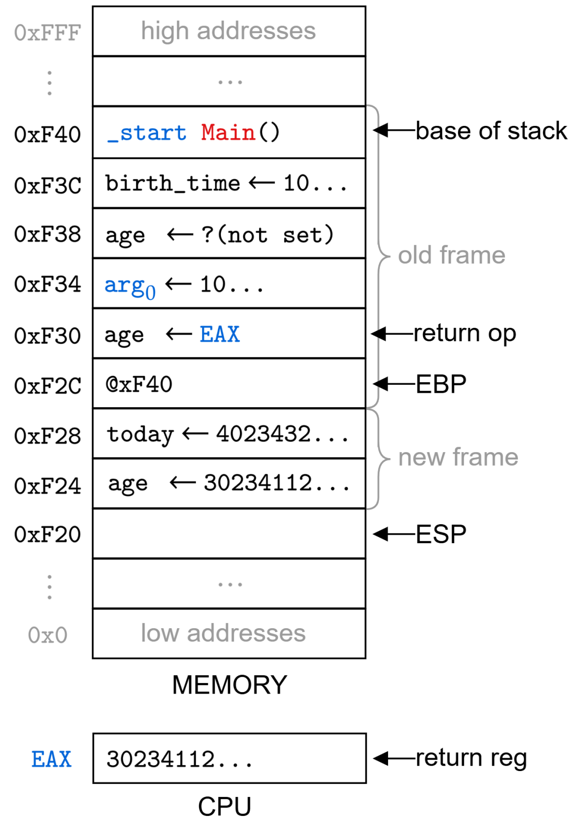


Figure 2.4: Revisiting Figure (2.3) at the moment the function `CalculateAge(birth_time)` has supplied its return value to the EAX register, and is about to return. We see that before calling `CalculateAge(birth_time)`: The old frame pushed it's arguments (`birth_time`) onto the stack, then the return address (IP/Next Instruction) onto the stack, and finally the old EBP (Base Pointer) onto the stack. The 'new frame' then sets the saved EBP address to the current EBP, concluding the old frame into the 'new frame'. Moreover, since the offset looks for local variables below `0xF40`, the above 'birth_time' and 'age' are **out of scope** for the 'new frame', vice-versa. **Note:** This is still a high-level abstraction of what actually happens sequentially with opcodes; Nonetheless, this is the fundamental idea of how a stack works.

This concludes our discussion on stack structures; We continue with the heap structure next.

2.4 Heap Data Structures

So far we have simply said global data is declared in the **data segment** of memory. There is a second segment of memory that builds ontop of this called the **heap**:

Definition 4.1: Heap – Dynamic vs. Static Memory

When a program runs there is a **static** (fixed) region reserved for the program's data segment (local/global variables). During execution, more objects may be created, needing additional memory; A new region of memory is reserved **dynamically**, building upwards from the top of the data segment, called the **heap**.

Language protocols either **manually** (e.g., Assembly, C) or **automatically** (e.g., Python, Java) manage this memory:

- **Manual Memory Management:** The programmer must explicitly allocate and deallocate memory using functions like 'malloc' and 'free' in C.
- **Automatic Memory Management:** The language runtime automatically allocates and deallocates memory, often using a **garbage collector** to reclaim unused memory (no variables pointing to it).

Unlike the stack, this allows values to be accessed from anywhere in the program, regardless of the function call or scope.

We discuss hash tables more in-depth in the following section, for now we provide a high-level idea:

Definition 4.2: Hash Table

A **hash table** uses a **hash function**, taking a **key** (e.g., number or string) and producing a fixed-sized **hash** value (index), creating a table of mappings (key→hash). At such indices lies data associated with the key, enabling fast data retrievals.

A **universal hash function** is a hash function that uniformly distributes hashes across the hash table.

Note: The input in many context (typically cryptographic), may be called 'data' or 'message'; The output: hash, checksum, fingerprint, or digest.

Definition 4.3: Arrays in Memory

Arrays list elements sequentially in memory. A reference to an array is a pointer to the first element. To terminate reading an array, we must either know the size of said array or have some **sentinel value** (e.g., 'null' or '0') to indicate the end of the array.

Consider the following examples:

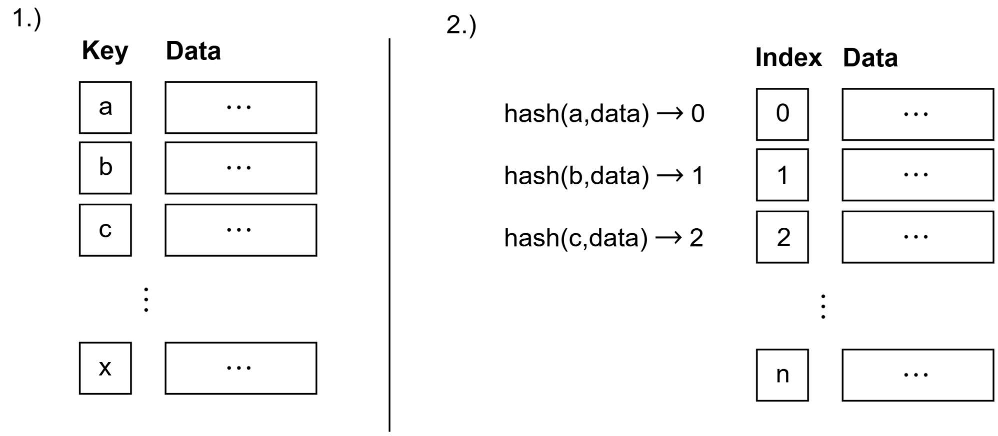


Figure 2.5: On the left (1) demonstrates a typical diagram one might find when learning about hash tables. Here a through x are the keys, which house some type of data. On the right (2) shows a slightly more detailed version, which emphasizes that keys a–x are hashed to indices 0–n in the hash table. The data could be any other value (e.g., number, string, or object). Moreover, hash tables under the hood are arrays, with each index pointing to whatever data is associated with the key.

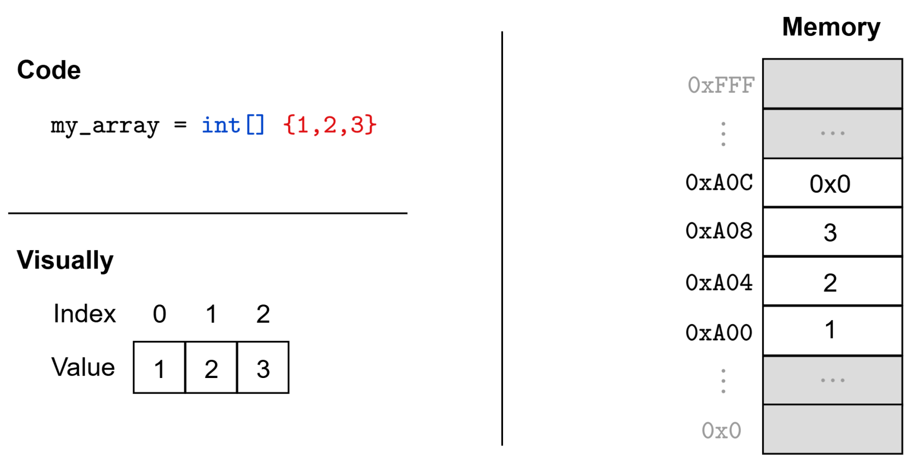


Figure 2.6: Here there are three sections breaking down how arrays look: In code, typical diagram depictions (Visually), and in memory. The code depiction illustrates a toy language creating an array of numbers ($[1, 2, 3]$). The visual depiction shows how indices relate to values. The memory depiction shows how the array is laid out in contiguous memory locations, where `0x0` is the sentinel value (a bit-pattern of all zeros). In code, `my_array` holds the address `0xA00`.

Objects behave very similarly to arrays, but with a few key differences:

Definition 4.4: Objects in Memory

An **Object** (or **struct**), is a collection of key-value pairs, where each key is called a **field** or **attribute** and each value can be any data type (e.g., number, string, or address).

Attributes are stored in array like fashion, where each element is a fixed-offset from the head (start) of the object. The object itself is a pointer to the first element. Accessing attributes works differently in compiled (e.g., C) vs. interpreted (e.g., Python) languages:

- **Compiled Languages:** There is no lookup, as the compiler has *hardcoded* the offsets of each attribute interaction (e.g., 'object.attribute' is translated to a direct memory access).
- **Interpreted Languages:** The interpreter looks up a hash table lookup for the attribute name.

Depending on the use case, objects may be stored in the heap or stack:

- **Static Objects:** An objects whose size is known at compile time can be allocated on the stack. I.e., no changes to the object are made after creation (e.g., Math and Time objects, which purely exist to compute).
- **Dynamic Objects:** Often just called **objects**, are allocated on the heap, allowing for dynamic resizing and modification (e.g., a student object with attributes like 'name', 'age', and 'grades' that can change over time).

Languages like Java push this even further by allowing both static (shared) and dynamic (personal) fields within a class.

Definition 4.5: Object-oriented – Classes, Interfaces, & Polymorphism

Object-oriented programming is a paradigm where objects are the main building blocks of the program. A **class** is a blueprint for defining how an object will behave once **instantiated** (created). In this paradigm, functions are called **methods**, as they are defined and used within the class (i.e., globally does not exist in independence).

Some languages (e.g., Java, C++) support **interfaces** (or protocols), which specify a set of methods that implementing classes must provide. Although the terminology varies (abstract classes, traits, protocols, etc.), they all ultimately describe capabilities an object must fulfill.

Inheritance is the main motivation behind classes and interfaces, enabling a **child** (sub) class to reuse or extend the functionality of a **parent** (super) class. To further reduce redundancy, **Polymorphism** allows objects to **override** (redefine) methods of the same signature (variable name) to accept different types of data or behave differently based on their context.

Example 4.1: Java – Classes, Interfaces, Abstracts, Inheritance, & Polymorphism

Consider the following Java code:

```java
public interface Animal { // Rough blueprint
    void eat(); void sleep(); void sound();
}

public abstract class Cat implements Animal { // Partial blueprint
    @Override
    public void eat() {
        System.out.println("Cat eats fish");
    }

    @Override
    public void sound() {
        System.out.println("Meow");
    }

    // Abstract method to demonstrate subclass-specific behavior
    public abstract void run();
}

public class Cheetah extends Cat { // Inheritance from parent Cat class
    @Override
    public void run() {
        System.out.println("Cheetah runs at 120 km/h");
    }

    @Override
    public void sound() {
        System.out.println("Chirp");
    }
}

public class Main {
    public static void main(String[] args) {
        // Polymorphism: reference is Animal, instance is Cheetah
        Animal anim = new Cheetah();
        anim.eat();        // calls Cat.eat()
        anim.sound();      // calls Cheetah.sound()
    }
}
```

Java polymorphism allows parent types to host children instances as seen with `Animal anim = new Cheetah();`, but `Cheetah chet = new Cheetah();` also works. This allows us to create arrays of different animal types:
E.g., `Animal[] zoo = {new Cheetah(), new Lion(), new Elephant()};`, assuming they all implement the `Animal` interface. ■

Strings are not what they seem:

Definition 4.6: Strings & Characters in Memory

A **character** is represented by a numeric code unit:

- In C, a single `char` (1 byte) typically holds an ASCII code (0–127). Characters beyond U+FFFF use two `char` values, a **surrogate pair**.
- In Java, `char` is a 16-bit UTF-16 code unit (U+0000..U+FFFF). ASCII values (0–127) map directly to the same Unicode code points. We can take advantage of the encoding:

```
char c = 'A';
System.out.println((int)c);  // prints 65, since 'A' is U+0041
```

This allows us to do things like checking for valid characters:

```
if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
    // c is in 'a'..'z' or 'A'..'Z'
}
```

We can also perform arithmetic on `char`:

```
char c = 'A';                // U+0041 (65)
char next = (char)(c + 1);   // 'B' (66)
```

Typically, a **string** is stored as a contiguous array of **characters**. In low-level languages (e.g. C), that array ends with a null terminator (`\0`) and literal strings reside in the data segment. In higher-level languages (e.g. Java, Python), strings are full objects with methods. For e.g.,

C:

- String literals (e.g. `"Hello"`) are placed in the (often read-only) data segment.
- Runtime-constructed strings (via `malloc`, `strcpy`, etc.) live on the heap.

Java:

- Compile-time literals are **interned** (stored as a single shared copy) into the **String Constant Pool** section (specially reserved on the heap).
- Any other `String` (e.g. via `new String(...)`, concatenation, or user input) also resides on the heap but outside the pool.
- Because Java strings are immutable, interning lets multiple references share the same character data.

With slight alterations to our code in Figure (2.5), we illustrate heaps and arrays:

My_Program

```
var birth_time
var name = "Ren"

void Main()
   birth_time = Date.Now()
   var age = Calculate_Age()
   print(age)

int Calculate_Age()
   var today = Date.Now()
   var age = Date.Diff(today,birth_time)
   return age
```

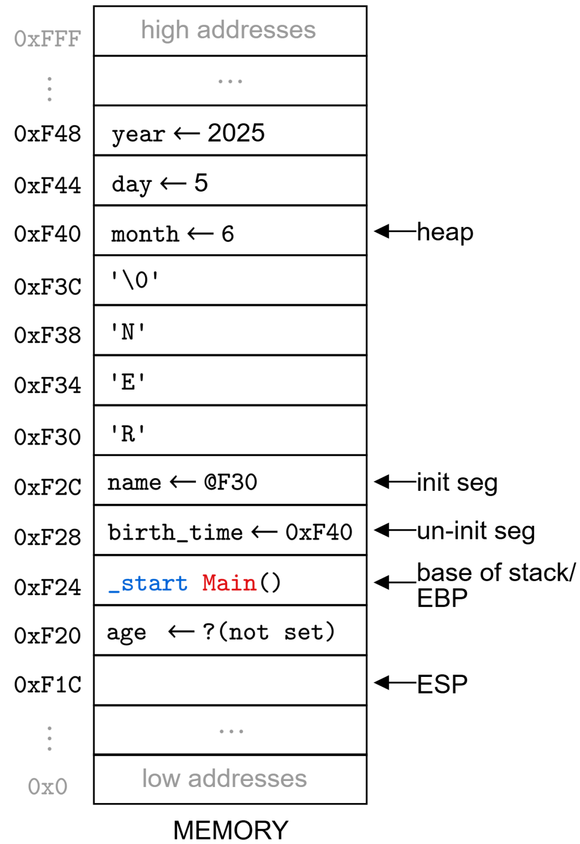


Figure 2.7: Here `birth_time` and `name` are global variables. Following the C convention, `birth_time` is placed in the uninitialized data segment, while `name` is placed in the initialized data segment. Since `name` is a string, it holds a reference to the first character in the string, which is stored contiguously in the initialized data segment ending with a null terminator ('\0'). During execution, `Date.Now()` a method call from a `Date` object is called; This method returns a new object, which is placed on the heap with its attributes (`month`, `day`, `year`) stored contiguously in memory. **Note:** Methods such as `Date.Diff()` are code (not data), which do not live in the heap or stack.

Definition 4.7: Factory Method

A **factory method** is a function that creates and returns an object, often initializing it with default values or parameters. In Figure (2.7), the method `Date.Now()` is a factory method that creates a new `Date` object with the current date and time.

The below illustration summarizes the heap and stack in memory:

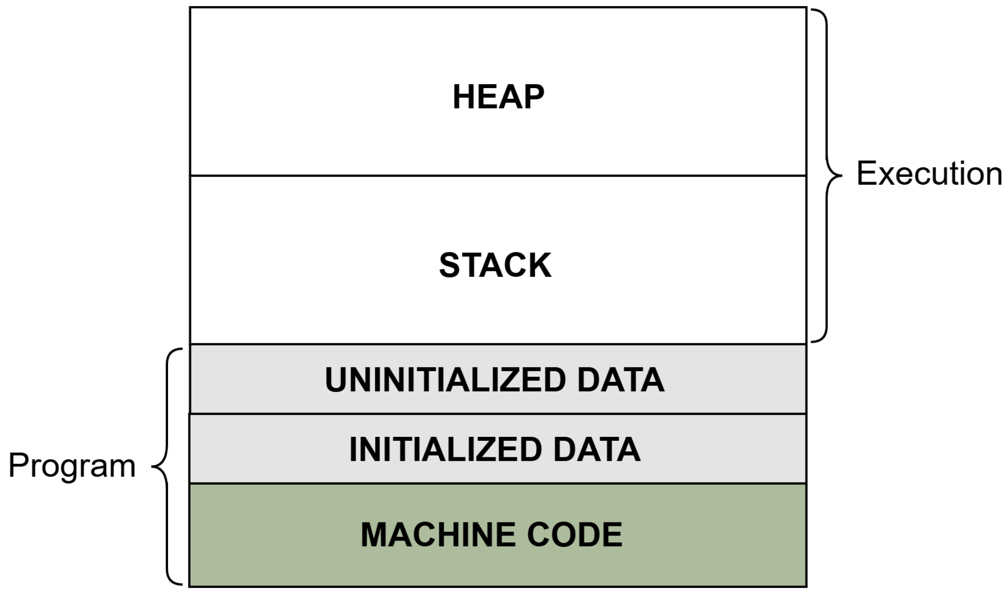


Figure 2.8: The above figure demonstrates the relationship from bottom-to-top the order at which data is loaded into memory. First the program compiles to machine code and loaded by the OS into memory. From there, provisions to the data segment (static memory: uninitialized and initialized) are made. Depending on the OS, some objects may have already been loaded into the heap, which are referenced by initialized data segment variables. Then as functions are called, the stack grows downwards within its allotted memory space. During execution of each stack frame, new objects may be placed on the heap, referenced by variables in the stack or data segment. Then depending on the language, a garbage collector periodically checks for objects with no references (i.e., no variables pointing to them) and deallocates them; Alternatively, the program explicitly deallocates memory using functions like 'free' in C.

2.5 Hashing & Collisions

In the previous section we lightly touched on the topic of hashing in Definition (4.2). This section will dive into more detail and difficulties collisions in hashing.

Definition 5.1: Collisions

A **collision** occurs when two different keys hash to the same index in a hash table. This is an unavoidable issue in hashing when keys begin to exceed the available indices.

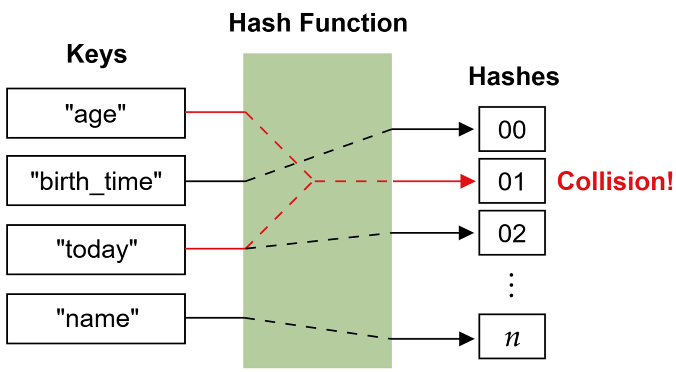


Figure 2.9: Four keys (‘age’, ‘birth_time’, ‘today’, ‘name’) go through a hash function to n possible indices. Keys, ‘birth_time’ and ‘name’, find a unique one-to-one mapping; However, ‘age’ and ‘today’ both hash to the same index, causing a collision.

Example 5.1: Simple Hashing Algorithm

Consider the hashing algorithm H, it takes the first ASCII value modulo the size of the table. Concretely, $H(k) := \text{ASCII}(k[0]) \ \% \ n$, where n is the size of the table.

Given the function H, we consider the following keys under a hash table of size 10:

- **Key:** ‘apple’ $\rightarrow$ ASCII value $= 97 \rightarrow H(\text{apple}) = 97 \ \% \ 10 = 7$
- **Key:** ‘banana’ $\rightarrow$ ASCII value $= 98 \rightarrow H(\text{banana}) = 98 \ \% \ 10 = 8$
- **Key:** ‘bread’ $\rightarrow$ ASCII value $= 98 \rightarrow H(\text{bread}) = 98 \ \% \ 10 = 8$

Here, we see that ‘banana’ and ‘bread’ both hash to index 8, causing a collision. ■

One could have a superb hashing algorithm, but when space is tight, collisions are inevitable. We’ll look at two particular methods for dealing with this issue.

2.5.1 Open Addressing

Our first method:

Definition 5.2: Open Addressing

Open addressing is a collision resolution method where, upon a collision, the algorithm searches for the next available slot via a probing sequence.

Wrap Around: the algorithm uses a modulo operation (e.g., Given a table size of 10 and request for index 12, the algorithm would use 12 % 10 = 2).

Time Complexity: $O(n)$, where n is the number of elements in the hash table. For example, say the only free index is at 0 with all other indices occupied. If we hash to index 1, the algorithm will have to walk all n indices to find the free index at 0. Changing the probe method only switches order of indices checked, not the worst case.
Space Complexity: $O(n)$, where n is the size of the hash table (no additional space).

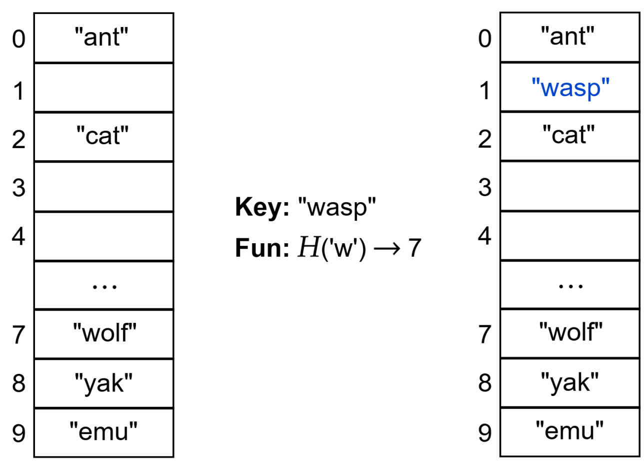


Figure 2.10: On the left is an existing hash table of 10 elements filled with various keys. The middle shows the insertion of a new key, 'wasp', which the function H hashes to index 7; However, index 7 already occupied. The algorithm walks through the table, wrapping around to the beginning, finding a free index at 1. The right shows the final state of the hash table with 'wasp' inserted at index 1.

Definition 5.3: Linear Probing

Linear Probing in open addressing refers to sequentially checking each index for an available slot (e.g., Figure 2.10).

Definition 5.4: Quadratic Probing

Given a universal hashing function $H(x)$, a **quadratic probing** resolves collisions by defining,

$$h(x,k) := (H(x) + k^2)\ \%\ n$$

Where k defines the number of collisions, and n hash table size. The algorithm may **never discover** particular cells due to its even probing style. We **terminate execution** once n indices have been checked, avoiding an infinite loop.

So why even use quadratic probing?

Definition 5.5: Clustering

Clustering is a phenomenon in open addressing where multiple keys form contiguous *runs* of occupied indices. This degrades linear probing performance; Such is the main motivation behind quadratic probing.

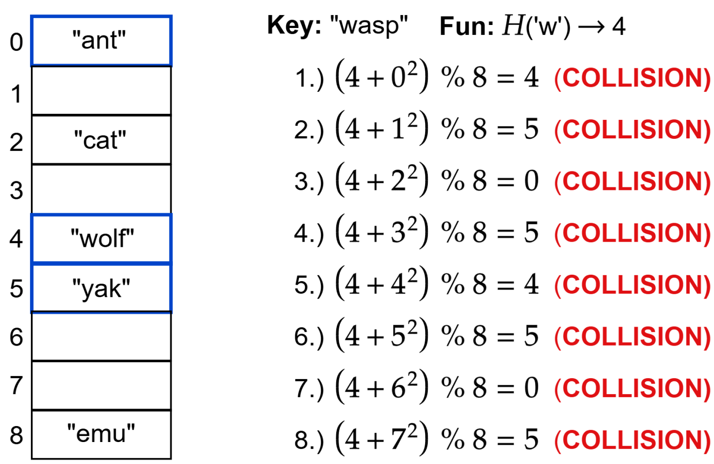


Figure 2.11: On the left is an existing hash table of 8 elements. We attempt to insert a new key, 'wasp', which hashes to index 4; Though, 4 in occupied. The algorithm continues with $(4+1^2)\ \%\ 8 = 5$, which is also occupied. After some probing, it appears only indices $4, 5, 0$ are appearing, from which are all occupied. The algorithm terminates at its n-th attempt with $(4+7^2)\ \%\ 8 = 5$. No spaces were found. One could imagine that if the table were larger or $0, 4, 5$ were free, the algorithm would have had better success.

Definition 5.6: Double Hashing

Double hashing resolves collisions by using two hash functions; Given, $H_1(x)$ and $H_2(x)$ uniform hashing functions, we define the probe sequence $h(x, k)$ as:

$$h(x, k) := \big(H_1(x) + k \cdot H_2(x)\big) \ \% \ n,$$

Where k is the collision count, and n is the hash table size. $H_2(x)$ is chosen such that:

- The hash satisfies $0 < H_2(x) < n$ (i.e., The result is likely taken by modulo n).
- It is pair-wise independent from $H_1(x)$ (i.e., not a transformation of/related to $H_1(x)$).
- Computationally inexpensive to evaluate.
- All outputs of $H_2(x)$ are relatively prime to n (does not share any common factors other than 1), ensuring all entries are probed.

We elaborate on the need for relatively prime numbers:

Theorem 5.1: Probing Period

A **period** defines the number of unique elements before the sequence begins to repeat. This cycle length is defined as the ratio:

$$\frac{n}{\gcd(n, H_2(x))}$$

Where n is the hash table size, and $H_2(x)$ is each hash output on an arbitrary x input. We ideally want $\gcd(n, H_2(x)) = 1$ for each x to achieve a full period of n. Hence, if n is a power of 2, $H_2(x)$ may uniformly provide odd numbers; Otherwise, for that particular key, it will only partially probe the table.

Without too much number theory, we attempt to intuitively understand the theorem:

Proof 5.1: Length of Probing Period

In terms of modulo, n defines a cycle of n elements, $0, 1, \ldots, n-1$; Each element is called a **residue class** (i.e., all possible remainders). E.g., 8 has residue classes 0–7 . Given a finite set of integers $\mathcal{H}$ (i.e., our hash function), all $\mathcal{H}_i$ need be co-prime to n to exhaust all residue classes. We exclude all $\mathcal{H}_i \geq n$, as n's cycle is definitively over (also by Definition 5.6). E.g., $1 \ \% \ 8 = 1$, $9 \ \% \ 8 = 1$. We pick a fixed-hash $h := \mathcal{H}_i$ such that $\gcd(n, h) = d > 1$. Recall that we calculate $(k \cdot h) \ \% \ n$ for each k collision. Hence if h and n factors intersect at d, then the $k = n/d_{th}$ collision will produce a multiple of n, terminating prematurely at 0.

$\blacksquare$

Let's try an example to see how this works:

Example 5.2: Double Hashing without co-primes

Consider a $H_1(x)$ function which always causes collisions, forcing the use the $H_2(x)$ function:

$$H_1(x) := x^0 \quad H_2(x) := x^2 \ \% \ (n-1)$$

Giving us the full function:

$$h(x,k) := \left(x^0 + k \cdot \left(x^2 \ \% \ (n-1)\right)\right) \ \% \ n$$

Observe when $n = 12$, with key $x = 3$, while increasing k collisions:

$k \cdot (3^2 \ \% \ 11) \ \% \ 12$	$h(x,k)$
$0 \cdot 9 \ \% \ 12$	0
$1 \cdot 9 \ \% \ 12$	9
$2 \cdot 9 \ \% \ 12$	6
$3 \cdot 9 \ \% \ 12$	3
$4 \cdot 9 \ \% \ 12$	0
$5 \cdot 9 \ \% \ 12$	9
$6 \cdot 9 \ \% \ 12$	6
$7 \cdot 9 \ \% \ 12$	3
$8 \cdot 9 \ \% \ 12$	0
$9 \cdot 9 \ \% \ 12$	9
$10 \cdot 9 \ \% \ 12$	6

Since $\gcd(12, 9) = 3$, the probing period is $12/3 = 4$, only touching indices $\{0, 3, 9, 6\}$. ■

2.5.2 Searching: Insertion & Deletion

Things become trickier with a populated hash table with previous deletions:

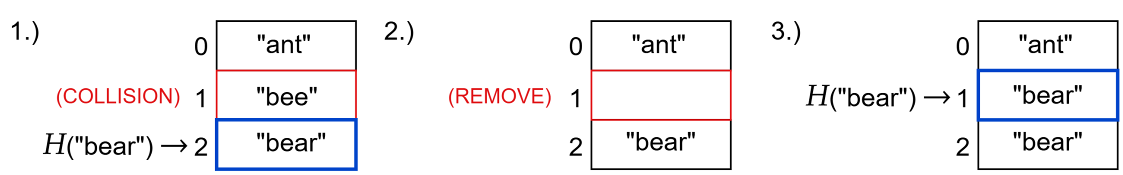


Figure 2.12: Demonstrates complications when inserting into a table blindly. 3.) niaively inserts "bear" at index 1 despite it already existing in the table. This is caused by a previous collision (1) and removal of the collider (2).

Theorem 5.2: Safe Insertion

To safely insert a key into a hash table, we create three distinctions for each cell:

- **Empty:** The index is empty, and a key can be inserted.
- **Occupied:** The index is occupied (blocked) by another key.
- **Deleted:** This index was previously occupied but free.

We may proceed as normal for events Empty and Occupied, but Deleted requires special handling; First, take note of the first **deleted index** then probe the table:

- **Empty:** If an empty index is found, insert at the first deleted index.
- **Occupied/Deleted:** Continue probing.
- **Duplicate** If the same key is found, insert any data, otherwise terminate.

Insertion at the first deleted index is safe, as if the key already existed in the table, it would have been inserted at any found empty index.

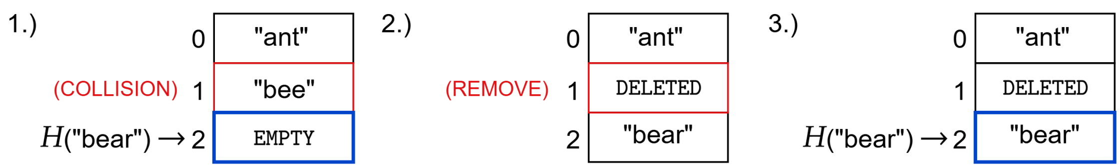


Figure 2.13: Revisiting Figure 2.12 at (3) we continue to probe after finding a deleted index. This leads us to find the already inserted key, “bear”, at index 2.

2.5.3 Separate Chaining & Linked Lists

A problem we have with open addressing is that it requires a lot of array space; If we run out of space, we have to resize the table, **which is costly**:

Theorem 5.3: Resizing a Hash Table

Resizing a hash table requires rehashing all keys, which is $O(n)$, where n is the number of elements in the table, excluding the computation cost of hashing each key again.

This brings us to the motivation for our second method of collision resolution:

Definition 5.7: Separate Chaining

Separate chaining is a collision resolution method where each index contains a **bucket**, which contains all keys that hash to such index. This saves contiguous memory space, as the array can stay a fixed size while holding object references living in the heap.

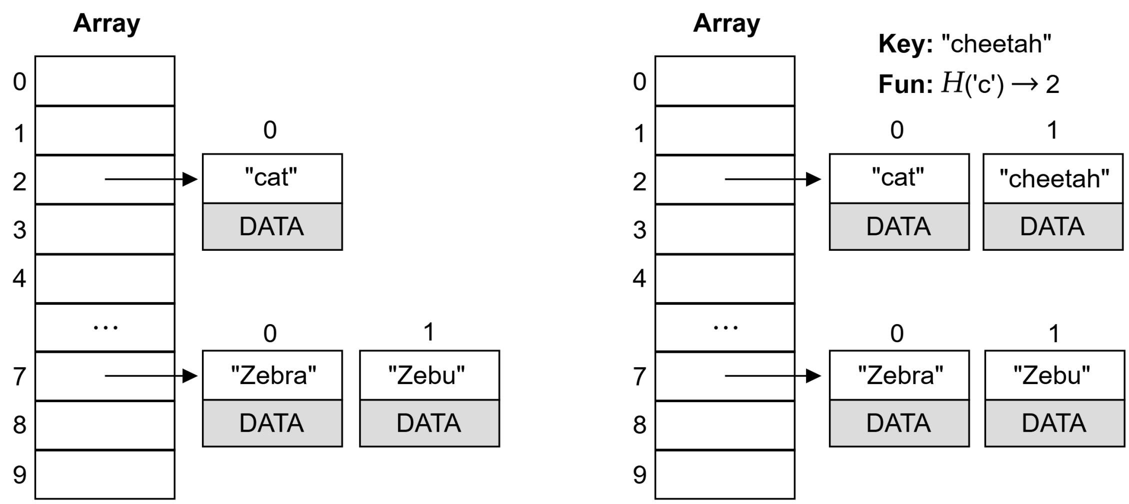


Figure 2.14: The left an array which points to another array (the bucket). The right shows the insertion of "cheetah" into the table, indexing at 2, adding to the end of the bucket. Here it's unclear how the data for each element is stored (perhaps a nested array). Additionally, the **same problem** of resizing occurs, as if a bucket is another array, we still have to resize it (there is no need to rehash the keys in buckets).

Definition 5.8: Resizing an Array

Many languages provide a method for resizing an array; However, this is costly as perhaps the next contiguous memory cell for the array is not available in memory. Hence, a new memory region is found and all elements are copied to a new array of larger size, typically $O(n)$, where n is the number of elements in the array.

Tip: In languages like Java or GO, resizing an array (ArrayList or Slice) is done by creating a new array of double the size and copying all elements over.

We introduce a new data structure to solve this problem:

Definition 5.9: Linked List

A **linked list** is a data structure consisting of single objects called **nodes**. Nodes are *linked* together via a reference to the next node in the list. This means there are no indices, and the next node can live anywhere in memory. The first element is called the **head**, and the last element is called the **tail**.

Each node at the very least contains a **value** and a **next** pointer to the next node in the list. Since a node is an object, an indefinite number of fields can be added to its class definition.

Definition 5.10: Common Types of Linked Lists

- **Singly Linked List:** Each node contains a reference to the next node.
- **Doubly Linked List:** Each node contains a reference to both the next and previous nodes.
- **Circular Linked List:** The last node points back to the first node, creating a cycle.

Time Complexity: Search is $O(n)$, as we may need to traverse the entire list.

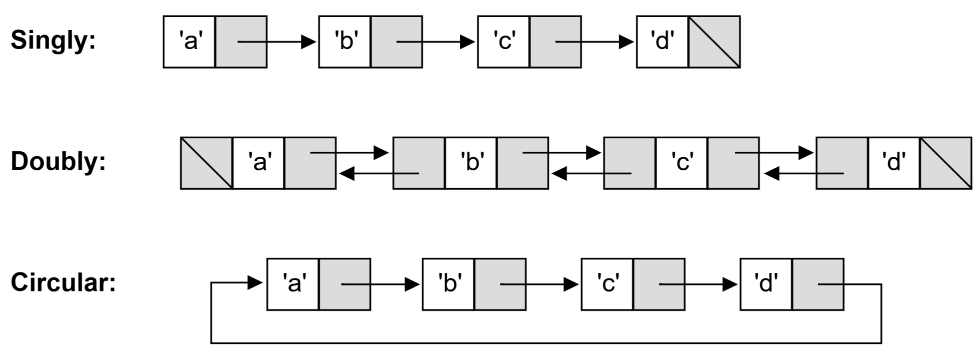


Figure 2.15: A visual representation of a singly, doubly, and circular linked lists.

Theorem 5.4: Open Addressing vs. Separate Chaining

If updates to the table are **rare** (rehashing on resize), choose open addressing. If updates are **frequent**, consider separate chaining.

It's worth noting that, separate chaining with linked lists requires **more memory** overhead for each node object.

We revisit Figure 2.14 with a linked list implementation:

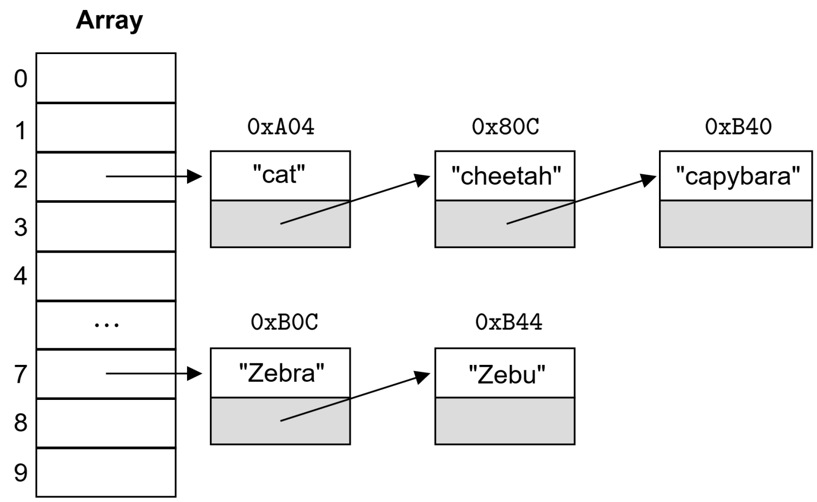


Figure 2.16: Here we see an array with two buckets, each bucket is a linked list. Each node points to the next node in the list. In particular, each node lives in at an ambiguous memory location.

Definition 5.11: Insertion & Deletion in Linked Lists

Inserting or deleting relies on shifting pointers around. To make sure references aren't lost to the linked list, we always point to a **dummy head** node, which holds no data and always points to the first actual node.

Say we have two nodes, A and B sequentially in a singly linked list referenced by `my_llist` (points to dummy head). it has one dot operator, `my_llist.next` (the next node in the list).

- **Inserting:** We attempt to add a new node C,
  - **At the head:** Set `my_llist.next = C` and `C.next = A`, $O(1)$.
  - **At the tail:** Traverse the list until via `my_llist.next` until a null `.next` is found, set the previous node's `.next` to C and `C.next = null`, $O(n)$.
  - **In the middle:** Traverse the list until node A is found, set `C.next = A.next`, then `A.next = C.next`, $O(n)$.
- **Deleting:** Now we have three sequential nodes, A, B, and C.
  - **The head:** Set `my_llist.next = A.next`, discards references to A, $O(1)$.
  - **The tail:** Traverse and set `B.next = null`, $O(n)$ (discards C).
  - **In the middle:** Traverse and set `A.next = B.next`, $O(n)$ (discards B).

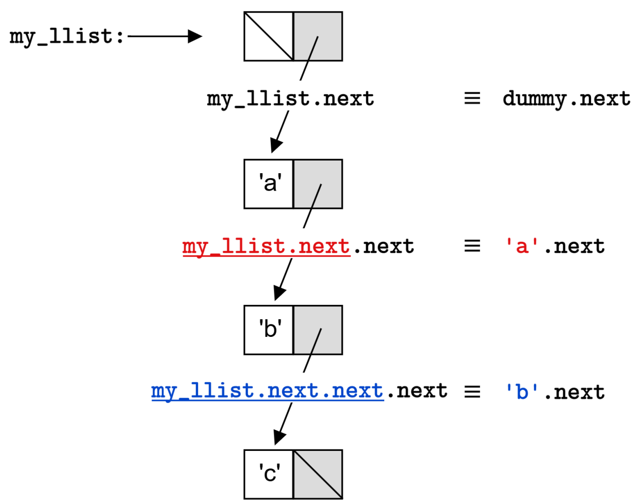
Visualizations on the next page.

Example 5.3: Insertion & Deletion in Linked Lists – Corollary

If we know the position of the node to be inserted or deleted, we can directly access it via a pointer. Revisiting Definition 5.11, we can insert or delete in constant time $O(1)$, instead of traversing the list. To demonstrate deletion, we have three sequential nodes, A, B, and C:

- **The head:** Set `my_llist.next = my_llist.next.next`, discards references to A.
- **The tail:** Set `my_llist.next.next.next = null`, discards C.
- **In the middle:** Set `my_llist.next.next = my_llist.next.next.next`, discards B.

This is a bit hard to read, we visualize it as follows:



In other words, `my_llist` is the dummy head node, `my_llist.next` returns the dummy node's `next` pointer (an address). If we *act* on such address, we access the address's fields. Hence, `my_llist.next.next` $\equiv$ `A.next`. Again, `A.next` $\equiv$ `B` (the address). So,

$$
\begin{aligned}
\texttt{my\_llist.next.next.next} &\equiv \texttt{A.next.next} \\
&\equiv \texttt{B.next} \\
&\equiv \texttt{C} \quad \text{(the address)}
\end{aligned}
$$

Still, it's not advisable to code like this, primarily because of readability. ■

2.5.4 Load Factor & Performance Metrics

We want to be pre-emptive at avoiding collisions and resizing the table. The following measurement tracks this:

Definition 5.12: Load Factor

The **load factor** α of a hash table is defined as the ratio of the number of elements n to the size of the table m:

$$\alpha := \frac{n}{m}$$

- **Open Addressing:** $\alpha < 0.5$.
- **Separate Chaining:** $\alpha < 1$.

Once α exceeds the optimal threshold, we should **consider resizing** the table. Upon good load conditions, we generally expect $\Theta(1)$ and $\Theta(1 + \alpha)$ time complexity for insertion and deletion, respectively with open addressing and separate chaining.

Method	Insertion	Deletion
Open Addressing	$\Theta(1)$ average, $O(n)$ worst	$\Theta(1)$ average, $O(n)$ worst
Separate Chaining	$\Theta(1 + \alpha)$ average, $O(n)$ worst	$\Theta(1 + \alpha)$ average, $O(n)$ worst

Table 2.3: Time-complexity comparison of insertion and deletion in open addressing vs. separate chaining (where α is the load factor).

Operation	**Dynamic Array**	**Singly Linked List**
Insert at head	$O(n)$ (shift all elements)	$O(1)$
Insert at tail	$O(1)$	$O(n)$
Insert in middle	$O(n)$	$O(n)$
Delete at head	$O(n)$	$O(1)$
Delete at tail	$O(1)$	$O(n)$
Delete in middle	$O(n)$	$O(n)$
Search for element	$O(n)$	$O(n)$
Random access	$O(1)$	$O(n)$ (traversal)

Table 2.4: Inserting or deleting elements in dynamic arrays (growing) versus singly linked lists. In particular, maintaining order within a dynamic array forces shifting of elements upon insertion or deletion.

2.6 Virtual Memory

2.6.1 Problem Space of Virtual Memory

Virtual memory solves three problems [4]:

- **Not enough memory, Memory fragmentation, and Security**

Definition 6.1: Not Enough Memory

Back then, computer memory was expensive, and many computers had very little memory (e.g., 4–1 GiB or even less). CPUs could only support up to 4 GiB of memory, as CPUs were 32-bit (2^{32} addresses = $2^{32} bytes = 4GiB$). On the other hand, 64-bit CPUs can support up to 2^{64} addresses = 16 million TB of memory.

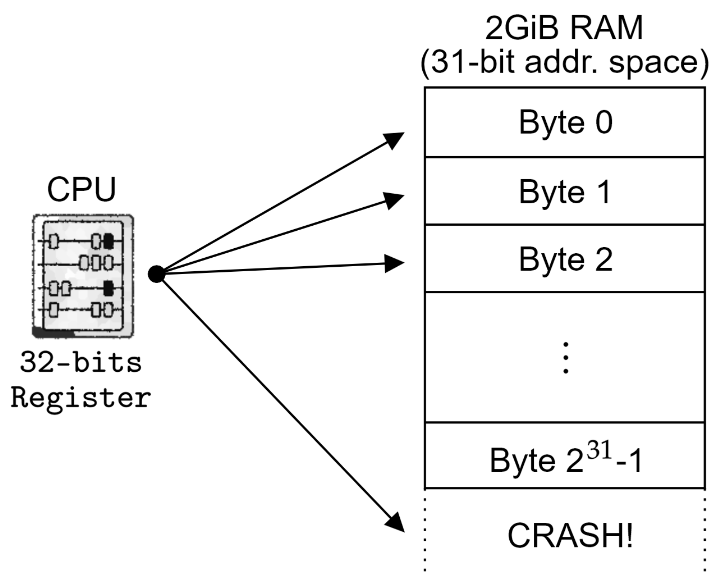


Figure 2.17: A 32-bit CPU accessing 2 GiBs of RAM, where a crash happens when trying to access beyond the 31-bit address space.

The next problem deals with multiple processes allocating and deallocating memory:

Definition 6.2: Memory Fragmentation

Memory can be thought of as a big array, where each cell is a resource a program can use. We want memory usage to be contiguous (i.e., no gaps or holes). So say we have an array

$$[O, O, O, O]$$

Where O represents free space in our array, each cell 1 GiB of space. If we have processes A and B take 1 and 2 GiBs respectively, we might have a memory layout of:

$$[A, B, B, O]$$

If we then free process A, we might have a memory layout of:

$$[O, B, B, O]$$

Now, if another process C needs 2 GiBs of memory, it will not be able to find a contiguous space of 2 GiBs, even though we have 2 GiBs of free space. This is called **memory fragmentation**.

Now finally we have the problem of protecting memory from other processes:

Definition 6.3: Memory Security

In a multi-process system, processes may have collisions when trying to access the same memory space. For example, if process A is a weather service and process B is some finance service, we don't want the weather service to overwrite the same memory space where the finance service is storing critical data. This is called **memory security**.

So in theory we want to give each process its own portion of memory, to solve overlapping access:

Definition 6.4: Virtual Memory

To solve process memory collisions, we give each process its own fictional view of memory, called **virtual memory**. Though for this to work, each virtual view is mapped to an actual place in the original memory we call **physical memory**.

Virtual and physical addresses are the cells spaces themselves.

Consider the figure below showing how virtual memory works in theory:

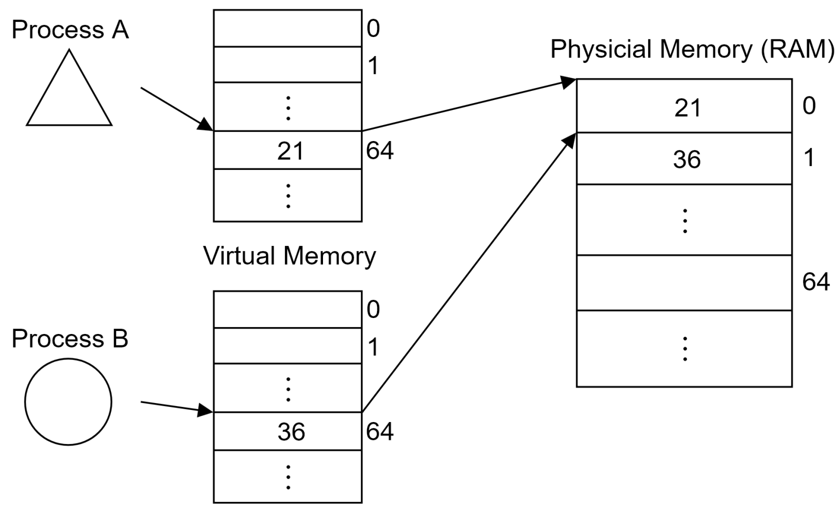


Figure 2.18: Processes A and B write to memory cell 64 in their view of memory, but in reality they map to different physical memory cells (0 and 1 respectively).

A quick aside:

Theorem 6.1: Physical Memory the CPU Accesses

In reality the CPU can access the physical memory of many other devices (e.g., hard drives, SSDs, etc.). In addition, the OS takes up some of the physical memory as well. The rest is left to programs to use. The program allocated space is the memory we refer to going forward as the **physical memory**.

Virtual memory solves the three problems we mentioned before:

Definition 6.5: Virtual Memory & Not Enough Memory (Swapping)

The physical memory can be much smaller than what a program thinks it has in virtual memory. When a program tries to access memory it does not have, the OS will **swap** physical memory to external storage to free up space. This means, while a program is not using a portion of memory at a given time, the OS can swap it in and out depending on system needs.

Memory that is swapped out is called **swap-memory**. Every time a we try to access such absent memory in our mappings, it's called a **page fault** (more on this later).

Definition 6.6: Virtual Memory & Fragmentation

Virtual memory allows programs to think they have contiguous memory. This simplifies their memory management, while in reality the OS manages the split memory mappings.

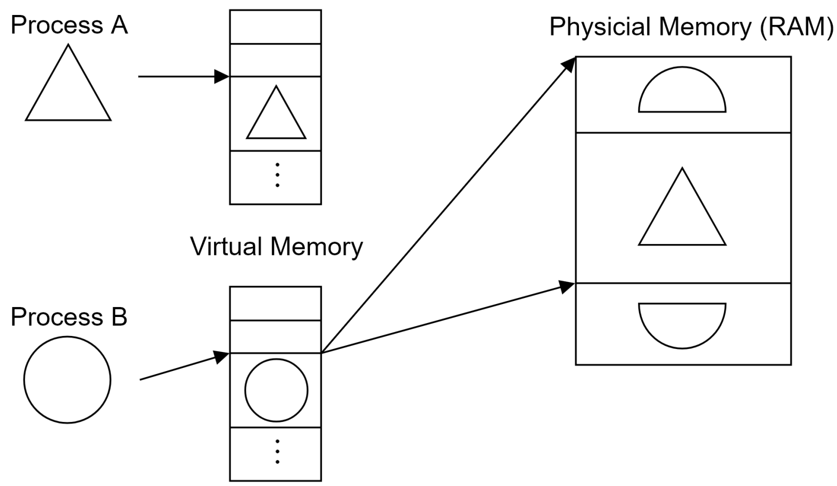


Figure 2.19: Here two processes A and B are using the same physical memory space. Both think they have contiguous memory, but in reality, B is split into two parts in the physical memory.

Definition 6.7: Virtual Memory & Security

Memory cells are collision safe as memory mappings are not shared in physical memory; However, this would be inefficient if say A and B depend on a secondary process C. In this case, A and B may share the same mapping to C's memory.

2.6.2 Virtual Memory Implementation (Page Tables)

We discuss the mapping mechanism further in linking virtual to physical memory.

Definition 6.8: Page Tables

The OS keeps a **page table**, where each entry is a mapping of a virtual memory cell to a physical one, called a **Page Table Entry (PTE)**. CPUs work with words (32 bits = 4 bytes), so the page table has one entry for every word (address) in the virtual memory space.

We make the following observation:

Theorem 6.2: Theoretical Page Table Space Complexity

If our CPU 32-bits, then there are 2^{32} addresses. CPUs speak in words, hence we'd need 2^{30} words, and thus $\approx$ 1 billion PTEs (4 GiB per table). This is too much memory to practically implement for each process.

We solve the above problem, via the following strategy:

Definition 6.9: Memory Chunking (Pages)

Instead of mapping each address to a PTE—which would be too large. We chunk the memory into **pages** reducing the number of PTEs needed. Typically, page sizes are some power of 2, most commonly 4 KiB (4096 bytes), meaning 1024 words per page. This reduces our page table requirement from 4 GiB to 4 MiB (1 Million PTEs). Despite moving 4 KiBs of data at a time, this works well in practice, as adjacent memory is often accessed together.

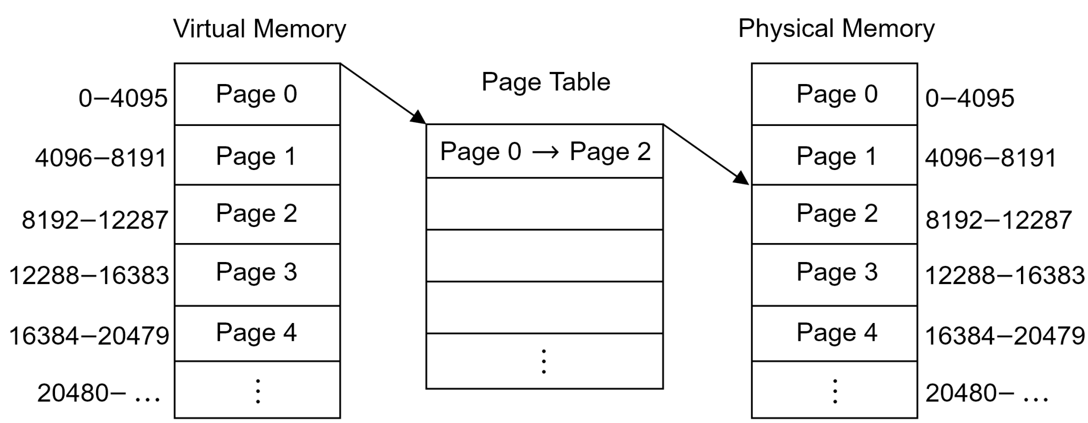


Figure 2.20: A 4 MiB page table, each PTE 4 KiB (4096 bytes, 1024 words), showing mappings.

One must ask themselves:

- What is the relationship of how the Page Numbers are separated.
- Can we determine the page number given a specific address?
- How might we convert a virtual to a physical address given some address?

We discuss the following on the next page.

Example 6.1: Converting Virtual to Physical Memory (intuition)

We may find conversions from virtual to physical memory via the following formula:

$$\text{PA} = \underbrace{(\text{VA } \% \text{ Page Size})}_{\text{offset}} + \underbrace{(\text{Page Size} \cdot \text{Physical Page Number})}_{\text{Physical Page starting index}}$$

Where % is modulo and Page Number $= \left\lfloor \frac{\text{Addr.}}{\text{Page Size}} \right\rfloor$. **However,** this is not the most efficient way, and is strictly for educational/intuition building purposes.

Given the Figure (2.20), $104 \to 8296$ from $104 + 4096 * 2 = 8296$. ■

Theorem 6.3: Converting Virtual to Physical Memory

Addresses are binary numbers, typically represents as hexadecimal (base 16) numbers. The first 12 bits of an address is the **offset**. The remaining higher-order bits yields the **page number**. To find the physical address, take the last 12 bits of the address, index the higher-order bits into the page table, and append the offset to the physical page number.

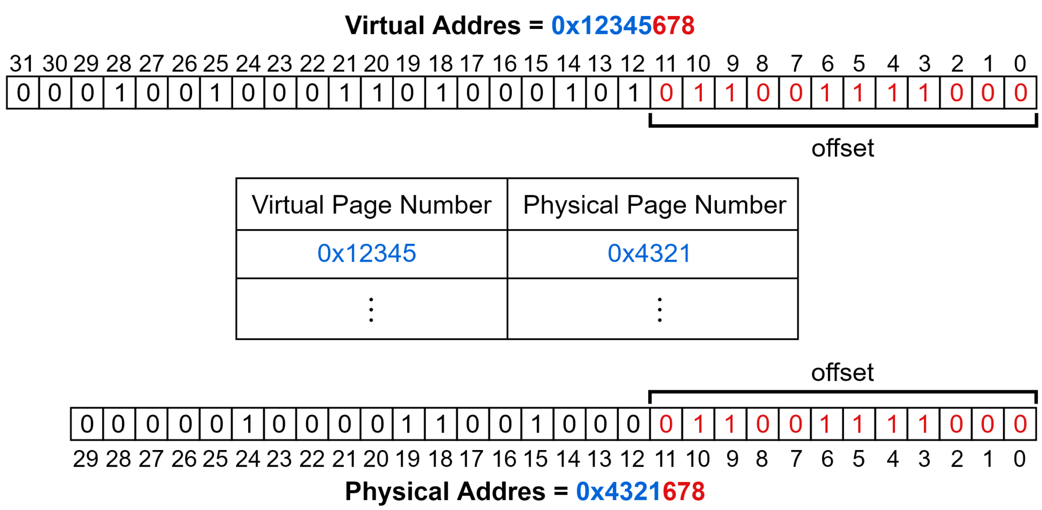


Figure 2.21: The conversion of 0x123456 (32-bit, 4 GiB) → 0x4321678 (30-bit, 1 GiB). Recall that if a accessing a page that is not in memory, causes a page fault, from which the OS with begin to swap memory with external storage devices.

2.6.3 Page Faults & Translation Lookaside Buffer (TLB)

So far we have discussed how the mapping system of virtual memory works, but now we pivot to how the OS handles swapping data in and out of memory on page faults.

Definition 6.10: Swapping on Page Faults

A **page fault** occurs when a program tries to access a page that does not have a mapping in the page table. The OS then uses a replacement strategy to evict a page in the page table, swapping it out to external storage. The most common strategy is to evict the **least recently used (LRU)** page, though other strategies such as **first in first out (FIFO)** and **least frequently used (LFU)** are used in some applications.

A page is **dirty** if it has been written to. In such case, we write back its contents to external storage before swapping. If the page is not dirty, we may discard it on swap.

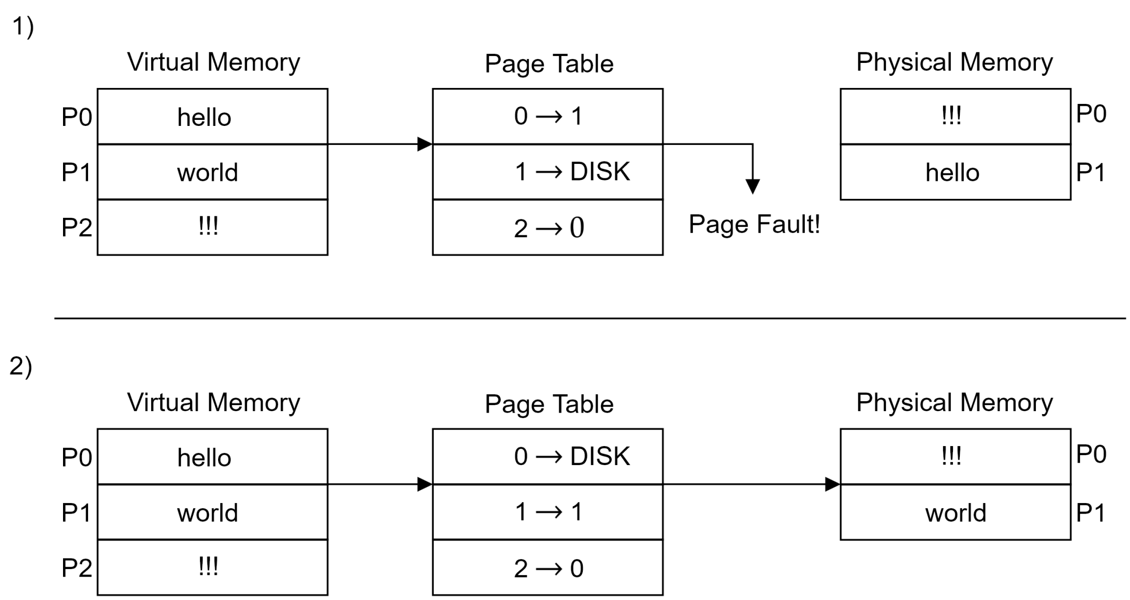


Figure 2.22: 1) A page fault occurs when trying to access page 1. 2) The OS has swapped out page 0 to external storage (DISK), and now page 1 can be accessed. Note: This diagram is for illustrative purposes, virtual memory does not contain data, only addresses binding to physical memory.

Definition 6.11: Direct Memory Access (DMA)

Page faults are expensive, as swapping data with I/O devices may take some time. Modern CPUs have a **Direct Memory Access (DMA)** module, which allows I/O devices to access memory directly, while the CPU completes other tasks.

Still our routine of mapping is expensive in it of itself.

Definition 6.12: Translation Lookaside Buffer (TLB)

The mapping process includes three steps:

1. Index the page table: Access Memory (RAM).
2. Convert Addresses: Computation.
3. Interact: Access Memory (RAM).

To speed this up, we create a small cache of the most recent Page Table Lookups, called the **Translation Lookaside Buffer (TLB)**. If a page in not in the TLB, we must preform the translation then load it into the TLB for future use.

Every successful TLB lookup is called a **hit**, while a failed lookup is called a **miss**. The **hit time** and **miss time** are the time it takes to access the TLB and page table respectively (measured in CPU clock cycles). The **hit rate** is the ratio of hits to the total number of lookups.

Modern CPU architectures usually have two TLBs, one for instructions (**ITLB**) and one for data (**DTLB**).

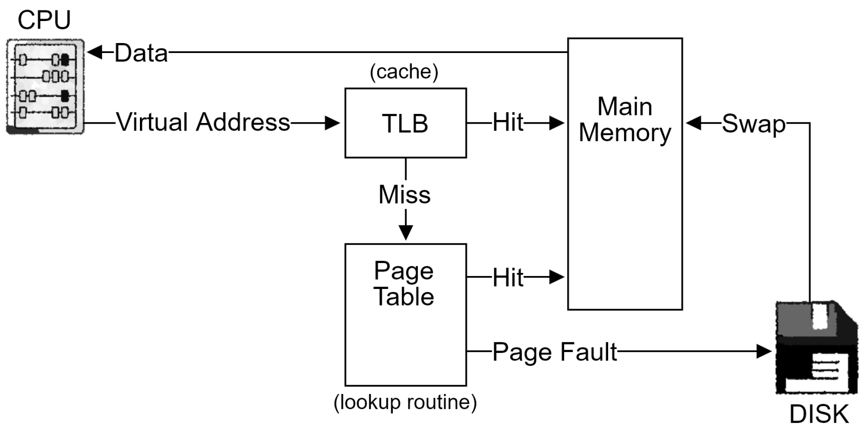


Figure 2.23: Demonstrating how a TLB lookup effects the memory access flow. CPU attempts to access memory providing the virtual address to the TLB. (Happy path) A hit occurs, memory is accessed and passed to the CPU. (Ok path) A TLB miss occurs, causing a page table lookup. A page table hit occurs, the mapping is cached and the memory is accessed. (Sad path) both a TLB and page table miss occurs, causing a page fault. The OS must decide which page to swap out. After swapping, the addressed is cached and the memory is accessed.

So far what we've been covering actually resides within the CPU architecture:

Definition 6.13: Memory Management Unit (MMU)

The **Memory Management Unit (MMU)** is a hardware component that manages the mapping of virtual to physical memory. It is responsible for translating virtual addresses to physical addresses, and it also handles page faults and TLB lookups.

2.6.4 Multi-level Page Tables

Let's define the problem space of multi-level page tables [**?**]:

Theorem 6.4: Single Level Page Table Cost

In a 32-bit system, each page table requires 4 MiB of memory. If we had 100 programs running, that's easily 400 MiB of memory.

Additionally, if we move page tables to disk (external storage), we lose any reference we had to them in an attempt to free up memory. So we need to keep some reference oracle in memory.

This highlights the need for a more efficient way

Definition 6.14: Multi-level Page Tables

Recall that in a 32-bit system, we have 4 GiB of memory. In single level page tables, we chunk our addresses into 4 KiBs (4096 bytes), and load all of them at once. That's

$$4 \text{ KiB} \cdot 1024 \text{ PTEs } = 4096 \text{ KiB} = 4 \text{ MiB}$$

We wish to offload data without losing references. To do this we allocate a single chunk of memory (4 KiB) to serve as an reference oracle, called the **page directory** or **1st level page table**. Each entry in the page directory points to other chunks from which we are safe to load and unload out of memory. These referenced chunks are called the **2nd level page tables**. Each entry in the 2^{nd} level page table points to a physical page.

Definition 6.15: Converting Virtual to Physical Memory (Multi-level)

In multi-level page tables, the first 12 bits are the **offset**, the next 10 bits are the index into the **second-level page table**, and the last 10 bits are the index into the **first-level page directory**. These tables are treated as arrays: indexing the first table yields the address of the second table, and indexing the second table yields the physical page number.

Recall, in single-level page tables the higher 20 bits are the page number and the index, which yields the physical page number. Multi-level breaks away from this, and instead as illustrated below:

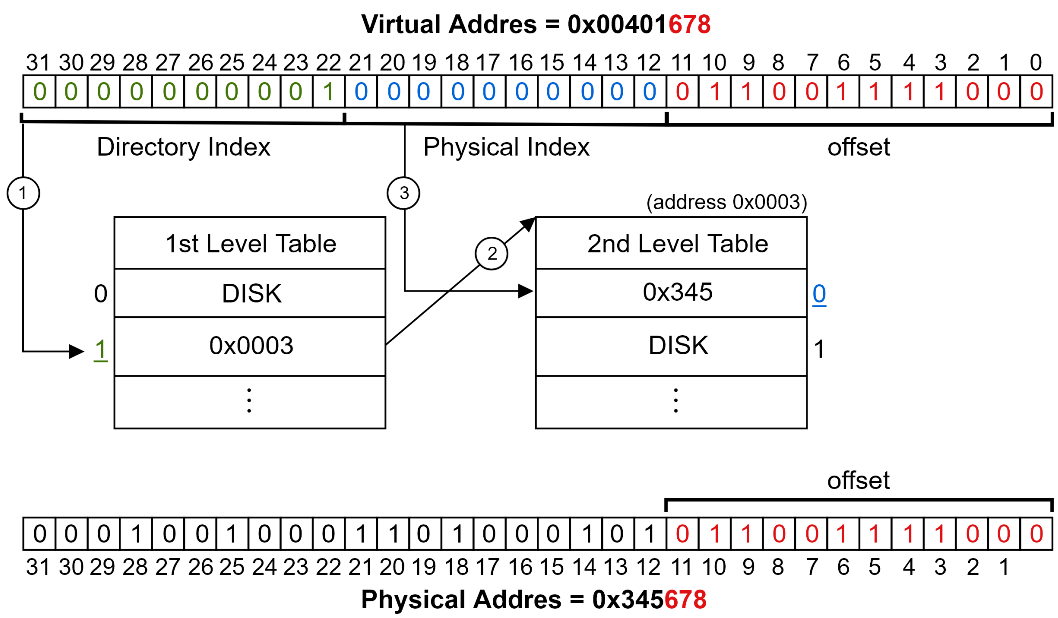


Figure 2.24: A multi-level page table, where the first level is a page directory, and the second level is a page table which points to physical pages.

Definition 6.16: Multi-level Page Scaling (64-bit)

Increasing the number of bits allows us to scale the number of levels in our page table. In 64-bit architectures using 4-level paging (e.g., x86-64), only the lower 48 bits are used. The virtual address is divided into five parts:

- 12 bits for the page offset
- 9 bits for the 4$^{\text{th}}$-level page table (PT)
- 9 bits for the 3$^{\text{rd}}$-level page directory (PD)
- 9 bits for the 2$^{\text{nd}}$-level page directory pointer table (PDPT)
- 9 bits for the 1$^{\text{st}}$-level page map level 4 (PML4)

Modern versions of Windows and Linux support 5 levels, which allows for a maximum of 129 PiB, while 4 gives us 256 TiB.

Finally we talk about the performance of our lookups:

Definition 6.17: Effective Memory Access Time (EMAT)

Effective Memory Access Time (EMAT) accounts for the time it takes to access memory in the presence of a Translation Lookaside Buffer (TLB), multi-level paging, and potential page faults. Given:

- t = TLB access time; m = Memory access time
- S = Page fault service time; n = Number of page levels
- h = TLB hit rate; p = Page hit rate (1 - page fault rate)

The EMAT is computed as:

$$\text{EMAT} = h(t+m) + (1-h)\,(t + p(n \cdot m) + (1-p)S)$$

Essentially [?],

```
EMAT=
TLB hit*(TLB access time + Memory access time)
+ TLB Miss*(TLB access time + PageHit*[n * memory access time]
+ PageMiss*PageFaultServiceTime)
```

Example 6.2: Three-Level Paging System

Suppose the system has:

- $n = 3$ page levels
- $t = 5$ ns (TLB lookup)
- $m = 100$ ns (memory access time)
- $\alpha = 0.80$ (TLB hit rate)

Then the EMAT is:

$$\begin{aligned}\text{EMAT} &= (5+100)\cdot 0.80 + (5 + 3\cdot 100 + 100)\cdot(1-0.80)\\ &= 105 \cdot 0.80 + 405 \cdot 0.20 = 84 + 81 = \boxed{165\text{ ns}}\end{aligned}$$

For a better TLB hit rate $\alpha = 0.98$:

$$\text{EMAT} = 105 \cdot 0.98 + 405 \cdot 0.02 = 102.9 + 8.1 = \boxed{111\text{ ns}} \qquad [?]$$

■

— 3 —

Computational Algorithms

3.1 Information Theory

3.1.1 Defining Information

The following sections gets us thinking about how we might represent and encode information in a computer system. This slightly deals with probability, but we will not cover it in depth. [13].

Definition 1.1: Information

Information measures the amount of uncertainty about a given fact provided some data.

Example 1.1: Playing Deck of Cards

Given a 52-card deck, a card is drawn at random. One of the following data points is revealed:

a) The card is a heart (13 possibilities).

b) The card is not the Ace of Spades (51 possibilities).

c) The card is the "Suicide King," i.e., King of Hearts (1 possibility). ■

Definition 1.2: Quantifying Information

Given a discrete (finite) random variable X with n possible outcomes $(x_1, x_2, \ldots, x_n)$ and a probability $P(X) = p_i$ for each outcome x_i, the **information content** of X is defined as:

$$I(X_i) := \log_2 \left(\frac{1}{p_i} \right)$$

Where $1/p_i$ is the probability of x_i, while Log base 2 measures how many bits (0 or 1) are needed to represent the outcome.

Example 1.2: Generalizing Information Content

A heart drawn from a 52-card deck may be represented as follows:

$$I(\text{heart}) = \log_2\left(\frac{1}{13/52}\right) \approx 2 \text{ bits}$$

More generally, we may redefine the information content as follows:

$$I(\text{data}) = \log_2\left(\frac{1}{M \cdot (1/N)}\right) = \log_2\left(\frac{N}{M}\right)$$

Where N is the total number of possible outcomes (e.g., 52 cards in a deck), and M is the number of outcomes that match the data (e.g., 13 hearts in a deck). Hence, $M \cdot (1/N)$ is the amount of information received from the data. Consider two more examples:

- **Information in one coin flip:** $\log_2(2/1) = 1$ bit ($N := 2, M := 1$).
- **Rolling 2 dice:** $\log_2(36/1) \approx 5.17$ or 6 bits ($N := 36, M := 1$).

■

Definition 1.3: Entropy

The **entropy** of a discrete random variable X is the average amount of information contained in all possible outcomes of X. It is defined as:

$$H(X) := E(I(X)) = \sum_{i=1}^{N} p_i \cdot \log_2\left(\frac{1}{p_i}\right)$$

Where function E is the expected value (i.e., average) of the information content $I(X)$ across all outcomes of X. This conveys how many bits b are needed to represent the outcomes of X:

- $b < H(X)$: Information is lost (i.e., not all outcomes can be represented).
- $b = H(X)$: An optimal representation.
- $b > H(X)$: Redundancy (i.e., not an efficient use of resources.).

Tip: For refreshers on $\sum$ consider our other text: Concise Works: Discrete Math.

Example 1.3: The Entropy of Four Choices

Consider a discrete random variable and it's possible outcomes $X := \{A, B, C, D\}$:

choice_i	p_i	$\log_2(1/p_i)$
A	1/3	1.58 bits
B	1/2	1 bit
C	1/12	3.58 bits
D	1/12	3.58 bits

Hence, the entropy of X is:

$$\begin{aligned} H(X) := \sum_{i=1}^{4} p_i \cdot \log_2\left(\frac{1}{p_i}\right) &= \left(\frac{1}{3} \cdot 1.58\right) + \\ &\left(\frac{1}{2} \cdot 1\right) + \\ &\left(\frac{1}{12} \cdot 3.58\right) + \\ &\left(\frac{1}{12} \cdot 3.58\right) + \\ &\approx 1.626 \,\text{bits} \end{aligned}$$

The entropy of X is approximately 1.626 bits, meaning that on average, we should be able to represent the outcomes of X using less than 2 bits per outcome.

■

Let's discuss how we might go about representing our outcomes:

Definition 1.4: Encoding

An **encoding** is an unambiguous mapping from a set of symbols to a set of bit strings:

- **Fixed-length encoding:** Uses a fixed number of bits to represent each symbol.
- **Variable-length encoding:** Uses a different number of bits for each symbol.

Example 1.4: Encoding Four Symbols

Consider the four symbols A, B, C, D and each possible encoding for them:

	Encoding for each symbol				Encoding for,
	A	B	C	D	"ABBA"
1.)	`00`	`01`	`10`	`11`	`00 01 01 00`
2.)	`01`	`1`	`000`	`001`	`01 1 1 01`
3.)	`0`	`1`	`10`	`11`	`0 1 1 0`

(1) Is a fixed-length encoding, (2) is a variable-length encoding, and (3) is also a variable-length encoding and uses fewer bits; **However**, it is ambiguous. Depending on how our program reads the string, it may group and misinterpret the bits.

E.g., (3) could be, "0 11 0" (A D A) or " 0 1 10" (A B C). Hence, an invalid encoding. ■

Theorem 1.1: Binary Tree Encoding

Binary trees may represent unambiguous encodings, where each symbol is a leaf node, and each edge represents the next bit. Since each path is unique, the encoding is unambiguous.

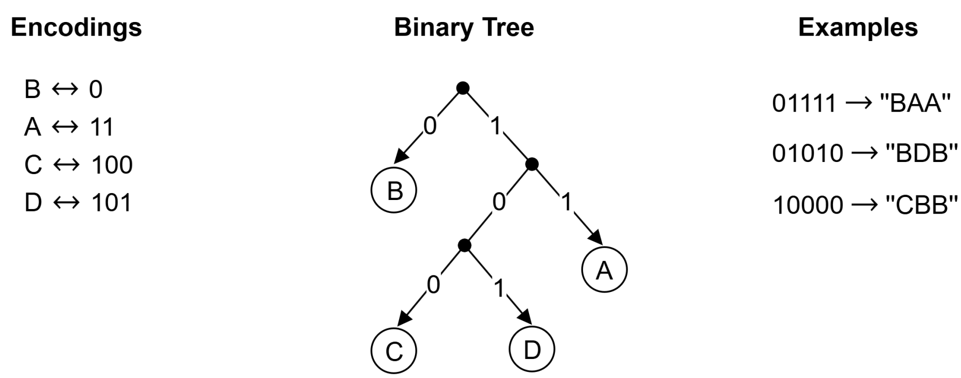


Figure 3.1: Encodings start at the root, each edge taken writes the next bit.

Theorem 1.2: Binary Tree Encoding – Fixed-Length

A fixed-length encoding is optimal when the number of symbols n bear an equal probability of occurrence. In a binary tree encoding, all leaves have the same depth.

I.e., for n symbols, each have a probability of $1/n$, hence an entropy of:

$$H(X) = \sum_{i=1}^{n} \left(\frac{1}{n} \cdot \log_2 \left(\frac{1}{1/n} \right) \right) = \log_2(n)$$

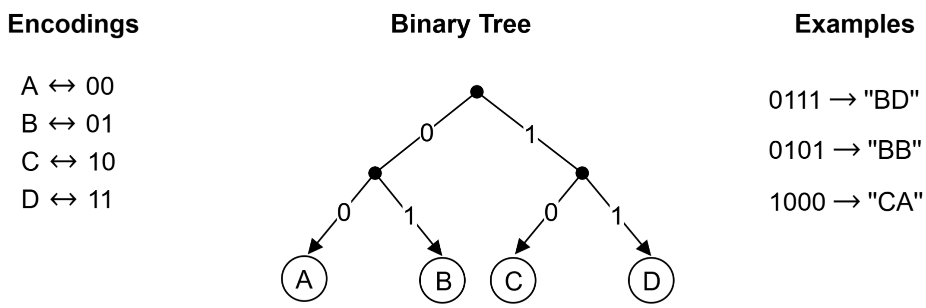


Figure 3.2: A fixed-length encoding for four symbols, represented as a binary tree.

Theorem 1.3: Choosing Variable-Length Encoding

If a symbol A has the high probability of occurrence, it should be represented with the shortest possible bit string. Vice versa, if symbol B has the low probability, then it should receive the longest bit string.

E.g., in Figure (3.1), the symbol B may be assumed to have the highest probability of occurrence, with C and D having the lowest.

Theorem 1.4: Variable vs. Fixed-Length Encoding

Though we would like to use a variable-length encoding in theory, a fixed-length encoding for complex data structures is provides simplicity and scalability.

Moving forward we focus on such fixed-length encodings.

3.1.2 Efficient Encodings

Now that we are familiar with Binary Tree Encodings (Theorem 1.1), optimal encodings:

Definition 1.5: Huffman's Algorithm

Build a subtree with the two least probable symbols. Such subtree is now considered a single symbol with a probability equal to the sum of the previous two. Do so until all symbols are combined into a single tree, called a **Huffman tree**, an optimal variable-length encoding.

Time Complexity: $O(n \log n)$, where n is the number of symbols. The actual encoding is done in $O(n)$ time, the $\log n$ factor comes from the need to sort the symbols by probability.

Symbol	Probability
A	1/3
B	1/2
C	1/12
D	1/12

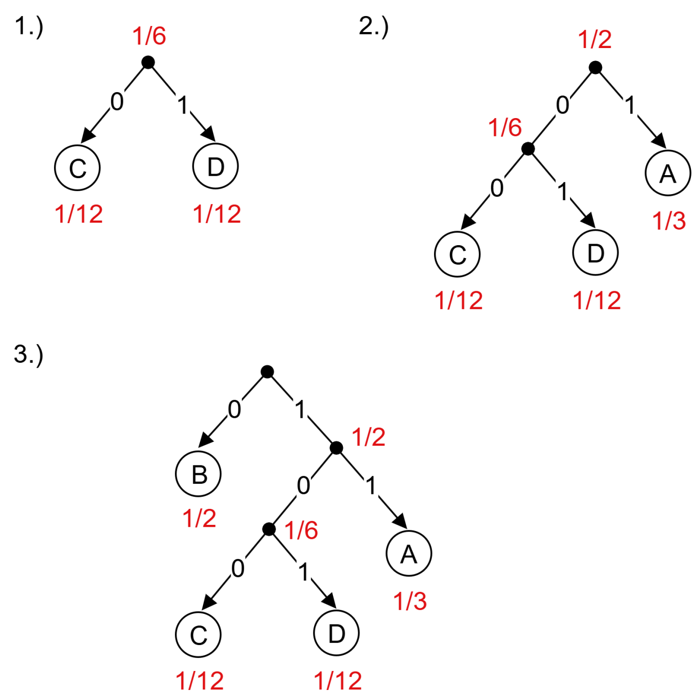


Figure 3.3: A Huffman tree for the symbols A, B, C, and D with their respective probabilities. (1) C and D are combined first having the highest probabilities, $1/12$, which combine to a subtree with a probability of $1/6$. (2) A is added, $1/3$ combining with the subtree $C + D$ to form a new subtree with a probability of $1/2$. (3) Finally, B is added, which has the highest probability of $1/2$, resulting in the final Huffman tree.

This brings us to the field of compression:

Definition 1.6: Lossless Data Compression

Lossless data compression is a technique that reduces the size of a file without losing any information. This is often achieved by encoding chunks of redundancy into a single symbol.

3.1.3 Error Detection & Correction

Two people Alice and Bob were trying to communicate a game of heads or tails over a noisy channel:

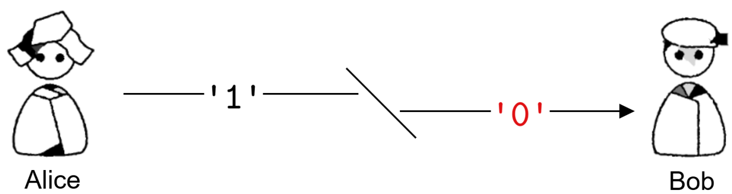


Figure 3.4: Alice and Bob trying to communicate '1' for heads, but the message is corrupted, and Bob receives a '0' instead.

As we see above, such a single bit error can be catastrophic. To mitigate this, our encodings must be more robust.

Definition 1.7: Hamming Distance

A **Hamming distance** is the number of bit positions in which two same length encodings differ. If there is a single bit error, the Hamming distance is 1. If there is none, then the two encodings are identical.

Example 1.5: Hamming Distance

The Hamming distance between the encodings,

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

is 2, as there are two bit positions that differ (marked in red). ■

Definition 1.8: Single-bit Error Detection

If two valid distinct encodings have a Hamming distance of 1, then a single-bit error may occur; Therefore if we increase the minimum Hamming distance to 2, we can detect single-bit errors. This may be done by adding a parity bit to the end of the encoding:

- **Even Parity:** Once added, the total number of 1's in the encoding is even.
- **Odd Parity:** Once added, the total number of 1's in the encoding is odd.

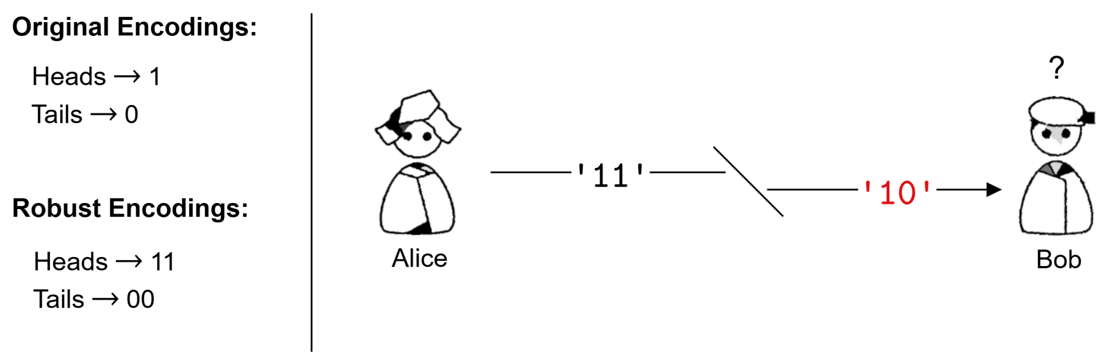


Figure 3.5: Alice and Bob using even parity to detect single-bit errors. Bob receives a single bit error and immediately detects it by checking the parity. **Note:** This does not help us if more than one bit is flipped.

Theorem 1.5: Detecting Multi-bit Errors

To detect E errors within an encoding, a minimum Hamming distance of $E + 1$ is required between each code word.

Theorem 1.6: Error Correction

Let there be code A and B, if the Hamming distance between them is $2E + 1$, then we can correct E errors. This is because the set of possible errors A_e and B_e will not overlap, allowing us to deduce the original encoding.

3.2 Number Base System Encodings

We now explore how to represent our base two or any other base number in a fixed-length encoding.

Definition 2.1: Number Base Fixed-length Encoding

Each memory cell in the computers stack is 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 2.1: Binary & Decimal Representations

Consider the integer, $a = 13$, in base, $B := 2$ **(binary)**, using Definition 2.1:

$$a = \underbrace{(1 \cdot 2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0)}_{\text{binary: } 1101} = 8 + 4 + 0 + 1 = 13$$

Here, the coefficients $a_3 = 1$, $a_2 = 1$, $a_1 = 0$, and $a_0 = 1$ correspond to the binary digits of 13, where each power of 2 represents the binary place value. Similarly, if we want to represent, $a = 45$, in base, $B := 10$ **(decimal)**:

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

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

We continue to common bases that one may encounter in computing and mathematics:

Definition 2.2: Hexadecimal

Hexadecimal base $B = 16$, using digits 0-9 and the letters A-F, where:

$$A = 10,\ B = 11,\ C = 12,\ D = 13,\ E = 14, \text{ and } F = 15.$$

Hexadecimal is commonly used in computing due to its compact representation of binary data. E.g., A **byte** (8 bits) can be represented as two hexadecimal digits, simplifying the display of binary data.

Theorem 2.1: 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 2.2: Base $2 \leftrightarrow 16$ Conversion

- Binary to Hexadecimal:

$$101101111010_2 \quad \Rightarrow \quad \text{Group as } \underbrace{[1011]}_{11}\ \underbrace{[0111]}_{7}\ \underbrace{[1010]}_{10} \quad \Rightarrow \quad B7A_{16}$$

- 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 seldom.

Definition 2.3: Unary

Unary, base $B = 1$. A system where each number is represented by a sequence of the same symbol. This system is more theoretical than practical given today's systems.

Given a toy unary system, we may represent numbers with a single symbol "I": $5 =$ `IIIII` or $2 =$ `II`. The absence of symbols may represent 0.

Definition 2.4: 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.

E.g., In the byte (8 bits), $[1111\ 1110]_2$, the MSB $= 1$ and the LSB $= 0$.

Theorem 2.2: Adding Binary

We may use the add and carry method alike decimal addition:

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

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

Example 2.3: Binary Addition

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

$$\begin{array}{r} \overset{}{1\,1000\,0}\overset{1}{1}00 \\ +\,0\,0001\,0100 \\ \hline 1\,1001\,1000 \end{array}$$

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

Definition 2.5: 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 2.4: Two's Complement Conversion

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

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

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

Definition 2.6: Big & Little Endian

Big Endian and **Little Endian** are two ways of storing multi-byte data in memory:

- **Big Endian:** The most significant byte (MSB) is stored at the lowest memory address (How English reads and writes).
- **Little Endian:** The least significant byte (LSB) is stored at the lowest memory address.

Subsequent bytes are stored in increasing order. **Note:** A single byte has no endianness.

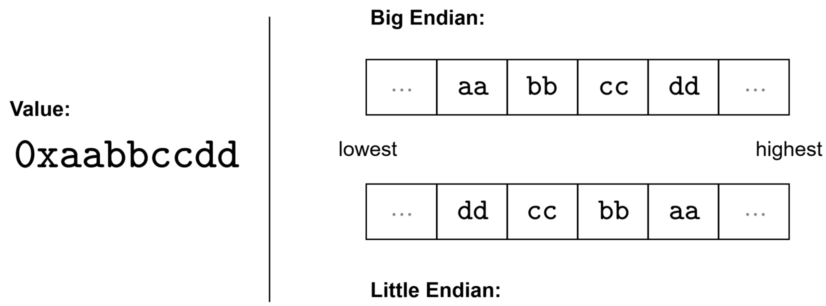


Figure 3.6: Big Endian vs Little Endian representation of $0xaabbccdd$ from lowest to highest memory addresses. Recall that $0x$ denotes a hexadecimal number, and aa is a whole byte (8 bits).

Tip: The name "endian" comes from *Gulliver's Travels* by Jonathan Swift, where Gulliver encounters two factions of people while traveling to the island of Lilliput. One faction, the **Big-Endian**, believes that eggs should be cracked at the big end, while the other faction, the **Little-Endian**, believes that eggs should be cracked at the little end.

Theorem 2.3: Commincation & Endianess

When communicating between systems, it is crucial to agree on the endianness of the data being sent. If one system uses big-endian and the other uses little-endian, the data will be misinterpreted. E.g., According to RFC 793, all the fields in network headers must be big-endian.

Note: When debugging one might encounter an issue with endianness. Suppose we read the little-endian value $0xaabbccdd$ two bytes at a time. This gives us $A_1 = 0xccdd$ and $A_2 = 0xaabb$. The computer read both values $B_1 = ddcc$ and $B_2 = bbaa$, translating it before sending it back to the debugger. This can cause issues as this is far beyond the original value.

3.3 Computing Large Numbers

Before moving forward, we must understand on paper how computation works at a mathematical level. Then we will re-visit encodings at the bit & byte level of abstraction [12].

Definition 3.1: 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} ($\approx$ 4.3 billion) or 2^{64} ($\approx$ 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 3.1: 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 $\|0\| = 1$.

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.

Definition 3.2: Computer Integer Division

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

Example 3.1: Long Division

let $a = \{12, 5, 17, 40, 89\}$, $b = \{4, 2, 3, 9, 10\}$ respectively, and base $B = 10$,

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

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 $x, n, s \in \mathbb{Z}$:

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

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 3.1: 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 3.2: Bit Shifting in Base 2

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 3.2: 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 $\quad r \leftarrow a$;
3 $\quad q \leftarrow \{0_{m-1} \cdots 0\}$;
4 $\quad$ **for** $i \leftarrow \|a\| - \|b\| - 1$ ***down to*** 0 **do**
5 $\quad\quad q_i \leftarrow \left\lfloor \dfrac{r}{b \ll i} \right\rfloor$;
6 $\quad\quad r \leftarrow r - (q_i \cdot (b \ll i))$;

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 (3.2).

Note: The above function is laid out in a more general form in its original text. But the above suffices for our purposes. **Remember:** The ALU can handle basic arithmetic operations in $O(1)$ time, as long as the numbers are small enough to fit in a single word.

Example 3.3: QuoRem - Quotient and Remainder

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?"

For q (quotient) and r (remainder):

1. Does $5 \ll 3$ fit into 101111_2?
   - **It fits!** As $[0000\ 0101]_2 \ll 3 = [0010\ 1000]_2$, is less than or equal to 101111_2.
   - **So:** $q_1 = 1000_2$ (3 zero shifts), $r_1 = (101111_2 - 101000_2) = 0111_2$.
2. Does $5 \ll 2$ fit into 0111_2?
   - **No fits!** As $[0000\ 0101]_2 \ll 2 = [0001\ 0100]_2$, is greater than 0111_2.
   - **So:** nothing is changed, $q_2 = 1000_2$, $r_2 = 0111_2$.
3. Does $5 \ll 1$ fit into 0111_2?
   - **No fits!** As $[0000\ 0101]_2 \ll 1 = [0000\ 1010]_2$, is greater than 0111_2,
   - **So:** nothing is changed, $q_2 = 1000_2$, $r_2 = 0111_2$.
4. Does $5 \ll 0$ fit into 0111_2?
   - **It fits!** As $[0000\ 0101]_2 \ll 0 = [0000\ 0101]_2$, is less than or equal to 0111_2.
   - **So:** $q_2 = 0001_2$ (0 zero shifts), then $q_1 + q_2 = 1001_2$, $r_2 = 0111_2 - 0101 = 0010_2$.
5. **Return:** $q = 1001_2 = 9_{10}$, $r = 0010 = 2_{10}$

We may verify this in decimal, $47 = 5 \cdot 9 + 2$ (dividend = divisor · quotient + remainder). ■

Note: The above example touches on the "Division Algorithm," which is not technically an algorithm, but rather a theorem. To learn more, consider: Concise Works: Number Theory

Scenario - Grade School 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.

Function 3.3: 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 $a + b$)

```
Function Add(a, b):
    carry ← 0;
    for i ← 0 to ℓ − 1 do
        tmp ← a_i + b_i + carry;
        (carry, c_i) ← QuoRem(tmp, 2);
    for i ← ℓ to k − 1 do
        tmp ← a_i + carry;
        (carry, c_i) ← QuoRem(tmp, 2);
    c_k ← carry;
    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:

$$\begin{array}{r} \overset{4}{\cancel{5}}\,\overset{10}{3}\,08 \\ -\,3\;406 \\ \hline 1\;904 \end{array}$$

Function 3.4: 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 $a - b$)

```
Function Subtract(a, b):
    borrow ← 0;
    for i ← 0 to ℓ − 1 do
        tmp ← a_i − b_i − borrow;
        if tmp < 0 then
            borrow ← 1;
            c_i ← tmp + 2;
        else
            borrow ← 0;
            c_i ← tmp;
    for i ← ℓ to k − 1 do
        tmp ← a_i − borrow;
        if tmp < 0 then
            borrow ← 1;
            c_i ← tmp + 2;
        else
            borrow ← 0;
            c_i ← tmp;
    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, $24 \cdot 16 = 384$:

$$
\begin{array}{r}
\overset{2}{2}4 \\
\times\ \ 16 \\
\hline
144 \\
+\,240 \\
\hline
384
\end{array}
$$

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 3.5: 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 $a \cdot b$)

```
Function Mul(a, b):
    for i ← 0 to k + ℓ − 1 do
        c_i ← 0;
    for i ← 0 to k − 1 do
        carry ← 0;
        for j ← 0 to ℓ − 1 do
            tmp ← a_i · b_j + c_{i+j} + carry;
            (carry, c_{i+j}) ← QuoRem(tmp, B);
        c_{i+ℓ} ← carry;
    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 3.6: 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)

```
Function DecToBin(n):
    b ← empty string;
    while n > 0 do
        r ← n mod 2;
        n ← ⌊n/2⌋;
        b ← r + b;
    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 3.4: Decimal to Binary Conversion

Let $n = 89_{10}$. We apply the function *DecToBin(n)* to show $89_{10} = 1011001_2$:

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

■

Theorem 3.2: 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 $O(n^2)$.

3.4 Manipulating Numbers in Base 2

Theorem 4.1: 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 4.1: Base 10 and Base 2 Lengths

Think of base 10 and 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 4.2: 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 4.2: Splitting & Shifting Base 10 and 2

Given $a = 745,562,010_{10}$, we split it in half as follows:

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

We can use the same bit shifting technique from Theorem (3.1) for base 10:

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

Given $[1111\ 1111\ 1001\ 1001_2]$ (32665_{10}), which is 16 bits long, we can split it in half, as follows:

$$A_1 = \frac{1111\ 1111\ 1001\ 1001_2}{2^{\lceil 16/2 \rceil}} = 1111\ 1111_2$$

$$A_0 = 1111\ 1111\ 1001\ 1001_2 \bmod 2^{\lceil 16/2 \rceil} = 1001\ 1001_2$$ ■

— 4 —

Circuits and Logic

4.1 Representing Information

4.1.1 Electricity & Information: Volts, Amps, & Watts

Figuring out how to represent information is tricky: Nature encodes information in DNA, though it may be hard to store because of decay (This is an active area of research). Punching holes in cards was a common method of storing information, but it's difficult to manipulate [13]. Ideally:

- **Inexpensive:** We want to reproduce at scale with low costs.
- **Stable:** Reliably store information for long periods.
- **Mutable:** The ability to manipulate information easily.

Definition 1.1: Electricity & Information

Electricity is a flow of electrons, which can be used to represent information. We can use the presence or absence of an electric current to represent binary values:

- **1** for presence of current;
- **0** for absence of current.

This is the basis of digital electronics and computing.

This is great for our applications, as electricity is relatively inexpensive given the scale of production.

Theorem 1.1: Noise & Error Accumulation

We ought to keep in mind that electricity is not perfect. Though we design systems to measure information, slight inaccuracies or environmental factors may introduce noise, which over time corrupts information.

It's important that we understand the difference between analog and digital systems:

Definition 1.2: Analog vs. Digital Circuits

A **circuit** is a closed path through which electricity flows, allowing us to manipulate and measure electrical signals.

An **analog** system is one that uses continuous signals to represent information, while a **digital** system uses discrete values (e.g., binary) to represent information.

Example 1.1: Real World Analog vs. Digital

Vinyl records are analog, as the grooves on the record represent sound waves continuously. In contrast, a digital system would be a CD or MP3 file, where sound is represented as discrete samples of the original sound wave. ■

Our main focus will be on digital systems, representing the strength of electricity as binary values. First we will briefly understand the terminology used in electrical systems:

Definition 1.3: Voltage, Amps, & Watts

Definition wise we have the following terms in electrical systems:

- **Voltage (Volts):** The potential difference between two points in an electrical circuit, measured in volts (V).
- **Amperage (Amps):** The flow of electric **current**, measured in amperes (A/I).
- **Resistance (Ohms):** The opposition to the flow of electric current, are ohms (Ω/R).
- **Power (Watts):** The rate at which electrical energy is transferred, are watts (W).

We calculate all such as follows:

- **Voltage:** $V = I \cdot R$ (Voltage = Current × Resistance).
- **Current:** $I = P/V$ (Current = Power / Voltage).
- **Resistance:** $R = V/I$ (Resistance = Voltage / Current).
- **Power:** $P = V \cdot I$ (Power = Voltage × Current).

These ratios between Voltage, Current, and Resistance are part of **Ohm's Law**.

Let's understand this with a common analogy to water flow:

Example 1.2: Water & Electric Flow Analogy

Imagine a water pipe system:

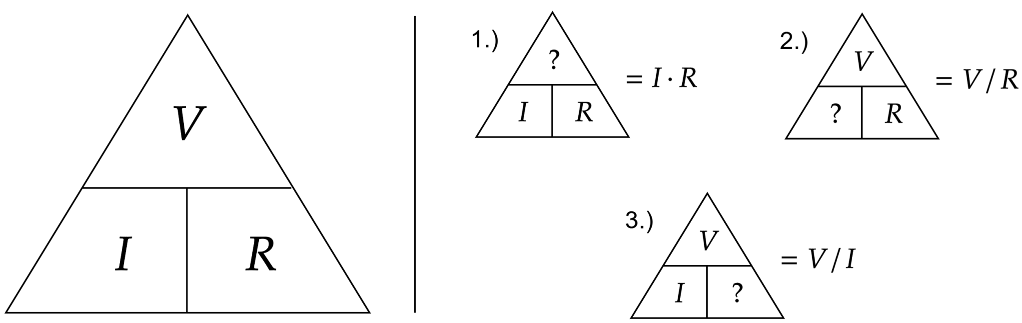
- **Voltage** is the water pressure in the pipes, the force pushing water through the system.
- **Current** is the amount of water flowing through the pipes at any given time.
- **Resistance** is the size of the pipes, which affects how easily water can flow.
- **Power** is the total amount of water that flowed through the system over time.

■

The relationship between Voltage, Current, and Resistance has a handy visualization:

Definition 1.4: Ohm's Triangle

Ohm's Triangle is a visual representation of the relationship between Voltage, Current, and Resistance. If any two values are known, the third can be calculated using the triangle:



Here, Voltage (V) is at the top, with Current (I) and Resistance (R) at the bottom corners:

1. **Voltage** is unknown: $V = I \cdot R$.
2. **Current** is unknown: $I = V/R$.
3. **Resistance** is unknown: $R = V/I$

A common mnemonic to remember is "Viral" for VIR (Voltage, Current, Resistance).

Now for completeness sake, we distinguish the following:

Definition 1.5: Energy vs. Power

Energy is the capacity to do work, measured in joules (J). **Power** is the rate at which work/energy is done or used, measured in watts (W). This is given by the formulation:

$$P = E/t$$

where P is power, E is energy, and t is time.

Example 1.3: Energy-Power Water Analogy

Continuing with the water analogy:

- **Energy** is the total amount of water stored in a tank.
- **Power** is how fast water flows out of the tank per second.

If we have a large tank (more energy), and water flows out slowly, we have high energy but low power. Conversely, if we open the tap wide (high power), we use up the water quickly. ■

We will wrap up such with a final analogy that uses numbers:

Example 1.4: Mathematical Water Analogy

- **Water Gun:** Imagine a water gun with very high pressure granted by the resistance of its small nozzle, so only a little water comes out.
  - Pressure (Voltage) = 10 V
  - Water Flow (Current) = 1 A
  - Power = 10 V × 1 A = 10 W
- **Large Hose:** Now, consider a large fire hose with lower pressure but a much wider opening with less resistance, allowing a lot of water to flow.
  - Pressure (Voltage) = 2 V
  - Water Flow (Current) = 5 A
  - Power = 2 V × 5 A = 10 W

Both systems consumed the same amount of power (10 W), despite supporting different voltages, currents, and possibly energy supplies. **Question:** What is the resistance of each system? ■

4.1.2 Combinational Devices

We now focus on the conduits of representing information digitally:

Definition 1.6: Digital Current Encoding Threshold

Given a line of voltage V, which we measure, V_{TH} serves as a threshold:

$$\text{0-bit} < V_{TH} < \text{1-bit}$$

In practice, we have noise ϵ in our measurements, making it hard to discern $V_{TH} + \epsilon$ from $V_{TH} - \epsilon$. To mitigate this, we pad the threshold from both sides called the **forbidden zone**:

$$\text{0-bit} \leq V_L < \text{"Forbidden Zone"} < V_H \leq \text{1-bit}$$

Where V_L (low-level) and V_H (high-level) are the region markers for valid voltage distinction.

Definition 1.7: Combinational Device

A **combinational device** is follows four specifications (spec.) called the, **static discipline**:

- **Input:** A set of input signals (i.e., measuring voltage levels).
- **Output:** A set of output signals (i.e., outputting voltage levels).
- **Functional Spec:** A mapping of all possible input combinations to an output value.
- **Timing Spec:** Detailing an upper bound t_{PD} **(Propagation Delay)**, which is the minimum amount of time needed for the output to stabilize on a new value after an input change.

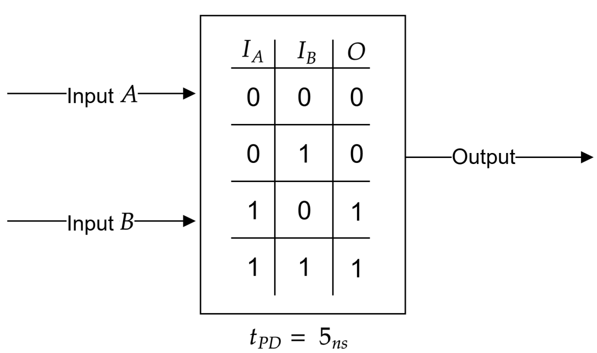


Figure 4.1: A combinational device with inputs A and B, and a truth table detailing mappings towards the output. The $t_{PD} = 5_{ns}$ (nanoseconds).

Definition 1.8: Combinational Digital Systems

A combinational device may also be made up of multiple other combinational devices. It must follow that:

- Each device is indeed a combinational device.
- Every input is connected to a single output.
- Each parent input will at most visit the same child input once (i.e., no cycles).

The t_{PD} of the system is the sum of sub-devices t_{PD}'s along a path such that it is the maximum such t_{PD} path in the system.

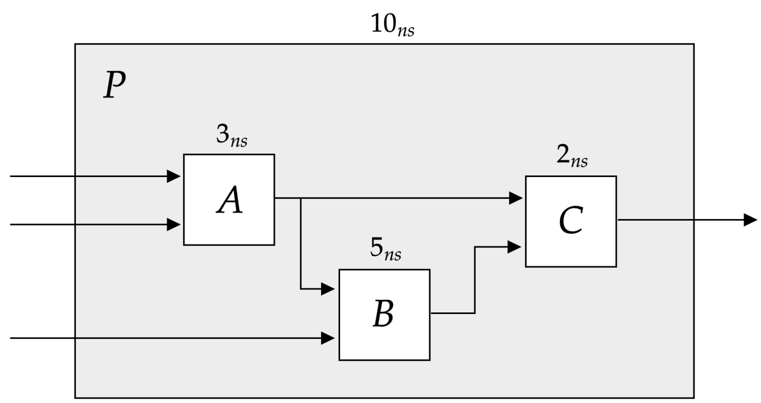


Figure 4.2: A combinational digital system, with a parent device P and children devices A, B and C. We abstract away the mappings focusing on the components and their connections. We see that there are no cycles and all sub components are also combinational devices; Hence, the parent system is a combinational device. The t_{PD} of the system is 10_{ns}, as the longest path takes $A \to B \to C = 3_{ns} + 5_{ns} + 2_{ns} = 10_{ns}$, the effective bottleneck of the system.

Though this introduces a new problem:

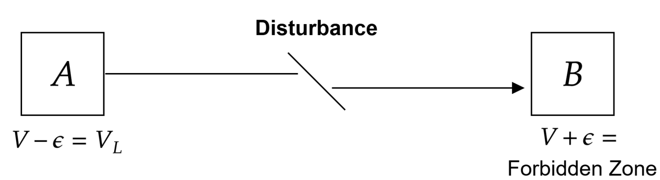
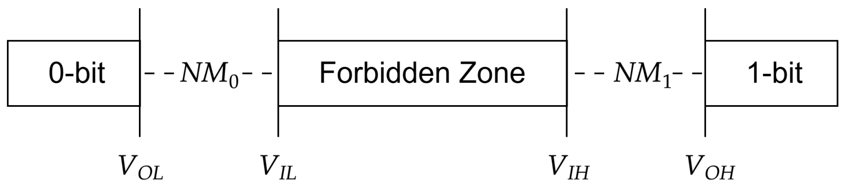


Figure 4.3: Combinational devices A and B communicate; However, A's output (V) is dangerously close to V_L, over the wire there is a disturbance, causing the input of B to enter the forbidden zone.

We offer a simple fix to this problem, by loosening up the thresholds during certain phases:

Definition 1.9: Noise Margins

To mitigate noise from outputs of a combinational device, we decrease the *forbidden zone* (FZ) for the receiving device. The overlap between the output's FZ and the input's FZ is called the **noise margin**. Concretely, we define the following:



Where, V_{OL} and V_{OH} are the output bounds, while V_{IL} and V_{IH} are the new input bounds. Then NM_0 is the noise margin for the 0-bit, and NM_1 is the noise margin for the 1-bit. The smallest of the two is called the **noise immunity** of the device (i.e., the worst case that must be supported).

Now when building our systems or combinational devices we must standardize how a particular device behaves on inputs and outputs to account for the worst case noise.

Definition 1.10: Voltage Transfer Characteristics (VTC)

The **Voltage Transfer Characteristics** (VTC) is a graphical representation which shows how a device's inputs affect its outputs after stabilization. The horizontal axis measures the input voltage, while the vertical axis measures the output voltage.

- **Horizontal Axis (V_{in}):** Contains V_{IL} and V_{IH}:

$$V_{in} \leq V_{IL} \text{ (0-bit)} \quad \text{and} \quad V_{in} \geq V_{IH} \text{ (1-bit)}$$

  Otherwise, the input is in the forbidden zone.

- **Vertical Axis (V_{out}):** Contains V_{OL} and V_{OH}:

$$V_{OL} < \text{Invalid Outputs} < V_{OH} \quad \text{such that} \quad V_{in} < V_{OL}, V_{in} > V_{OH}$$

  I.e., if the input is already in the forbidden zone, the output is irrelevant.

It's given that the device must perform properly such that a $V_{in} > V_{IH}$ will always yield a $V_{out} > V_{OH}$, and a $V_{in} < V_{IL}$ will always yield a $V_{out} < V_{OL}$. Each device has its own VTC, plotting the input-output relationship. The resulting curve is the **VTC** of the device.

Diagram next page.

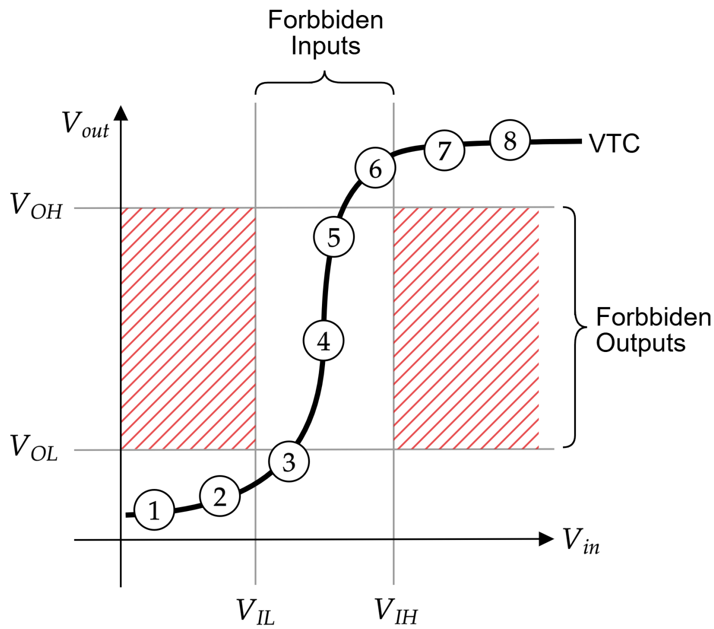


Figure 4.4: A Voltage Transfer Characteristics (VTC) diagram, showing the input-output relationship of a device. The horizontal axis represents the input voltage, while the vertical axis represents the output voltage. The invalid output regions are shaded in red. The VTC is the bold line that crosses each point. Possible points: (1-2) received a low input and output reading, (3-6) undefined, and (7-8) high input and output reading. E.g., an inverter device (inverts logic) would be a vertical flip of the above VTC curve.

Notice how in Figure (4.4) the center white region is taller than it is wide:

Theorem 1.2: Properties of VTC – Gain & Nonlinearity

Since more leeway is allowed for input voltages, the following suffices $V_{OH} - V_{OL} > V_{IH} - V_{IL}$. We can compactly write this as:

- **Width of the transition (x-axis):** $\Delta V_{in} = V_{IH} - V_{IL}$.
- **Height of the swing (y-axis):** $\Delta V_{out} = V_{OH} - V_{OL}$.

Since $\Delta V_{out} > \Delta V_{in}$, the **gain** (average slope) satisfies: (avg.) gain $= \dfrac{\Delta V_{out}}{\Delta V_{in}} > 1$.

Because of this ratio (gain > 1) small deviations (wiggles) in the input are amplified (exaggerated) in the output, which **regenerates** the signal (i.e., the output is a reinforced version of the input). The slope of the VTC must be **nonlinear** to ensure flat stable regions around 0 and 1 bits, and steep transitions between the forbidden zones (as seen in Figure 4.4).

4.1.3 Building Transistors: The Chemistry of Silicon

To even begin to manage currents and voltages, we will need a way to control the flow of electricity:

Definition 1.11: Transistor

A **transistor** is a small electronic semiconductor device. A **semiconductor** (e.g., silicon) is a material with electrical conductivity between that of a **conductor** (great electricity conductor) and an **insulator** (inhibits electric flow). Transistors fall into two broad families:

- **Bipolar Junction Transistor (BJT):** a current-controlled device with three terminals (pins),
  - **Emitter (E):** current flows *out*.
  - **Base (B):** controls operation.
  - **Collector (C):** current flows *in*.
- **Field-Effect Transistor (FET):** a voltage-controlled device with three terminals,
  - **Source (S):** current flows *in*.
  - **Gate (G):** controls operation.
  - **Drain (D):** current flows *out*.

Low-power transistors are molded in an epoxy (resin) package. Higher-power transistors often use a metal tab or "can" that you bolt to a **heat sink** (a metal object that dissipates heat).

Pin order and package style vary by model; check the **part number** and manufacturer's **datasheet** for exact details [2, 10].

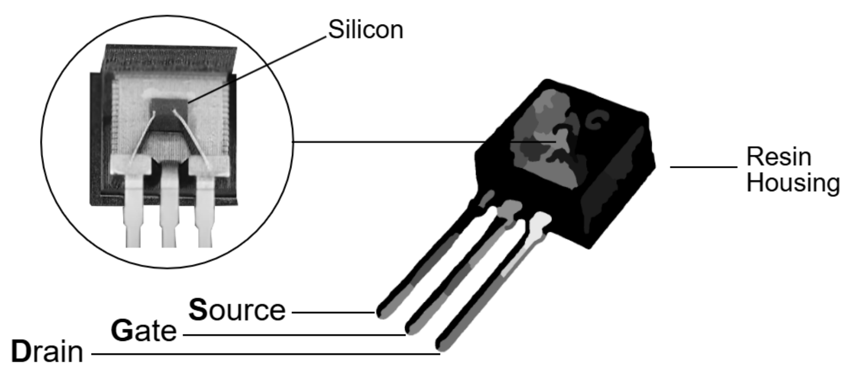


Figure 4.5: Cross-section of a discrete transistor: a silicon die (center) is bonded to three metal leads, all encased in an epoxy package. A metal tab (not shown) may be added for heatsinking.

Theorem 1.3: FETs over BJTs

A BJT needs continuous base current, which wastes energy. A MOSFET only requires its gate to be charged or discharged (i.e., voltage applied or removed), which is more efficient.

Now we briefly step into chemistry for completeness sake to understand differing silicon charges:

Definition 1.12: Anotomy of an Atom

An **atom** is the smallest unit of matter that retains the properties of an element. It consists of three main subatomic particles:

- **Protons:** Positively charged particles found in the nucleus.
- **Neutrons:** Neutral particles also found in the nucleus (same size as protons).
- **Electrons:** About the same charge as proton, but negative, and about 1800x smaller and lighter than a proton.

Protons and neutrons are tightly packed together in a space called the **nucleus**, gaining the name **nucleons**; Electrons orbit the nucleus at discrete distances called **shells** or **energy levels**. The number of protons in the nucleus defines the element (i.e., specifications). E.g., 79 protons will always be gold.

Opposite charges attract, causing an **orbital space**, in which subatomic particles never collide (i.e., alike orbiting planets). Neutrons act as a buffer between protons (e.g., Silver is stable with 60 or 62 neutrons, but unstable with 61). Atoms with different number of neutrons are called **isotopes**, latin for "same place". Electrons may jump between shells and atoms. If there is a greater number of electrons to protons, the atom is **negatively charged** (anions), otherwise it is **positively charged** (cations) [5].

Definition 1.13: Periodic Table

The **Periodic Table of Elements** organizes all known elements by the number of protons in their nuclei. This is called an **atomic number** (e.g., gold's atomic number is 79). Elements are abbreviated from their latin translations (e.g., gold is **aurum**, AU, which means "shining dawn"). There are 118 elements, with 80 being stable and the rest being unstable isotopes. Anything past 82 protons (lead) is unstable, undergoing radioactive decay.

Tip: The periodic table is complete, hence movies that claim "we discovered a new element!" truly deserve science-fiction as their defining genre.

We'll stop with the chemistry dive after these next two critical definitions

Definition 1.14: Shell Capacities & Valence Electrons

The first shell of any atom can hold up to 2 electrons, and the second 8. From 1-20 periodic elements, the third and fourth shells can hold 8 and 2 respectively. A *full* shell is considered **stable**, otherwise it is **unstable**. This arrangement of electrons within the shells is called the **electron configuration** (EC) of the atom. An EC is written as a n-tuple, starting with the inner-most shell (e.g., 2, 8, 8, 2 for calcium).

The outer most shell is called the **valence shell**. An atom's **valency** (the number of electrons in the valence shell) determines whether a chemical reaction will occur. If an atom is stable (i.e., full valence shell), it will not react with other atoms. Unstable atoms *strive* to become stable by either gaining, losing, or sharing electrons with other atoms [9].

Definition 1.15: Chemical Bonds – Molecules & Compounds

The act of atoms joining together (e.g., sharing electrons, which is called a **covalent bond**), forms a **molecule**. Concretely, a molecule is a merger of two or more elements. We use subscripts to denote the number of atoms in a molecule (e.g., H_2O is water, with two hydrogen atoms and one oxygen atom). **Compounds** are a subset class of molecules that consists only of two more more **different** elements (e.g., H_2O is a compound, but O_2 is not, as it only has one element, oxygen) [14].

Now to what we've been waiting for:

Definition 1.16: Doping – N-type & P-type Silicon

Silicon has 14 atoms, with an EC of (2, 8, 4); Hence silicon is unstable. If we view silicon (Si) as a 3D lattice (a string of Si atoms in 3D grid), each Si atom will share its four valence electrons with it neighbors to become stable (covalent bonding). This creates a **silicon crystal**.

Adding another element to the silicon lattice is called **doping**. We are interested in two types of doping [10]:

- **N-type:** When adding an element like phosphorus (P), EC of (2, 8, 5), is added to the silicon lattice, one electron goes unused after the covalent bonding. This free electron creates a **negative charge carrier** (hence N-type).
- **P-type:** Conversely, adding boron (B), EC of (2, 8, 3), creates a **positive charge carrier** (hence P-type). This is because boron won't have enough to share with its neighbors, causing **holes** (absence of electrons), overall lowering the density of electrons.

Let's visualize what we've learned so far:

Figure 4.6: An image of a silicon crystal [8].

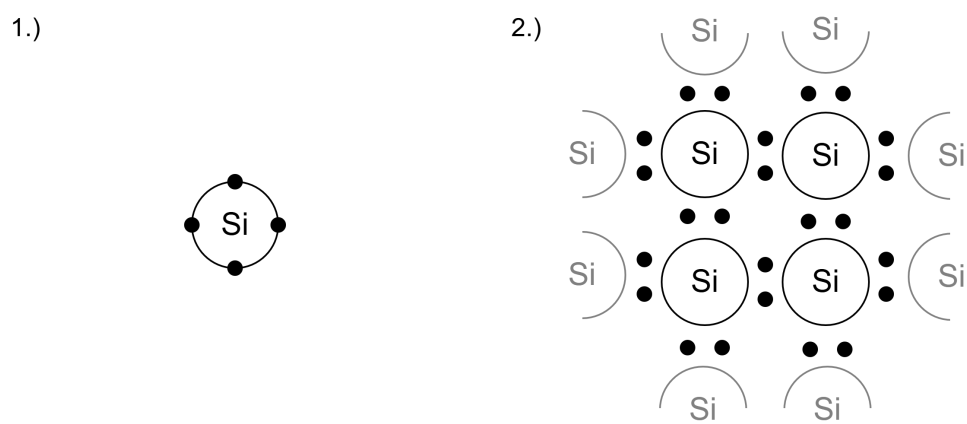


Figure 4.7: (1) Shows a single silicon atom (Si) and its valence electrons (4 black dots). (2) Shows a flattened silicon lattice where neighboring Si atoms share their electrons to become stable. This creates an electron configuration of (2, 8, 8) for surrounded Si atoms.

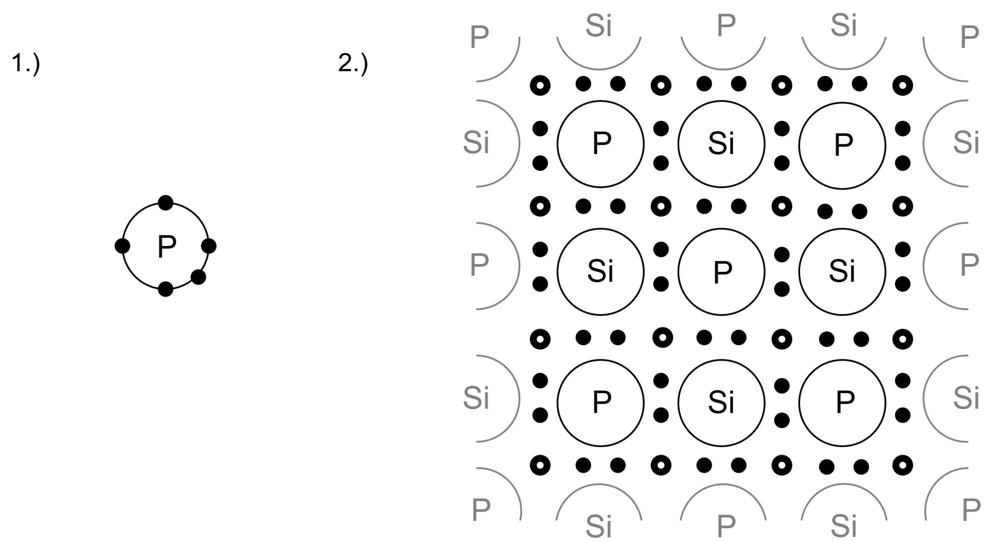


Figure 4.8: (1) Shows a single phosphorus atom (P) and its valence electrons (5 black dots). (2) A silicon lattice where phosphorus is doped, creating free electrons (thick dots with holes).

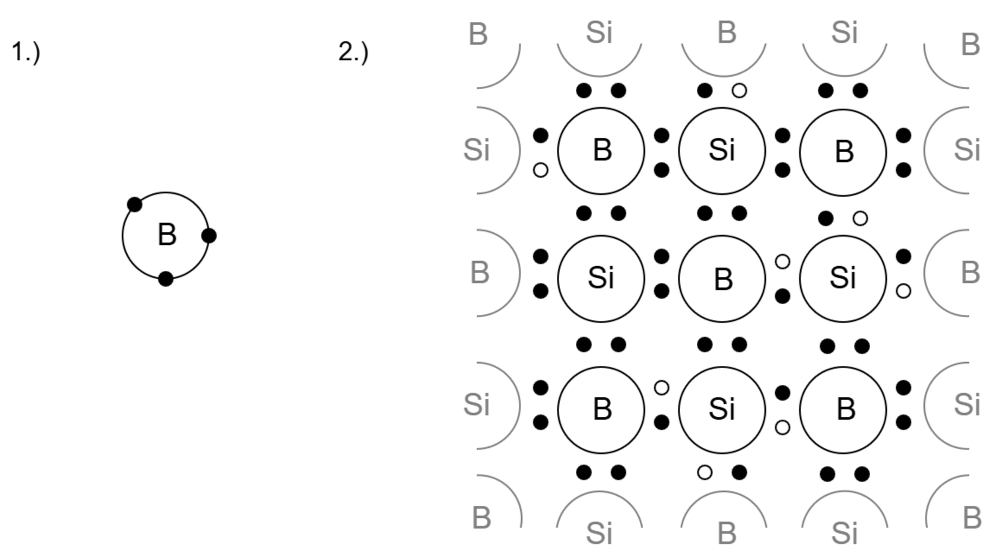


Figure 4.9: (1) Shows a single boron atom (B) and its valence electrons (3 black dots). (2) Shows a flattened silicon lattice where boron is doped, creating holes (halo dots).

Definition 1.17: N-type & P-type Junctions

When a N-type and P-type material are placed next to each other creates a **PN-junction**. At this junction, we get a **depletion region**, where N-type electrons and P-type holes cross; This leaves a slightly positive region on the N-type side and a slightly negative region on the P-type side. This manifests an electric field, creating a barrier, preventing further flow across the junction [10].

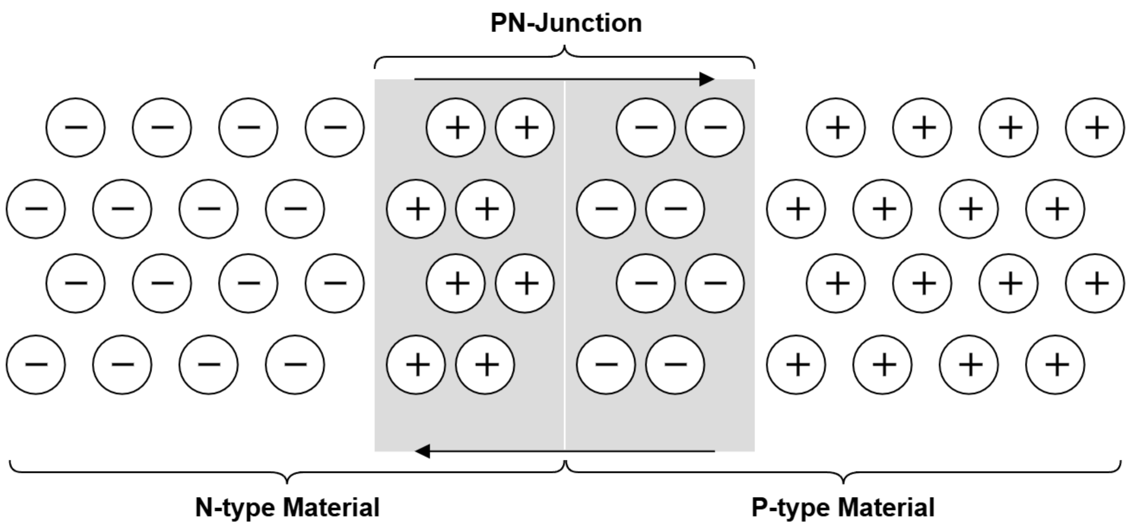


Figure 4.10: A PN-junction, where the depletion region is shown in gray.

Definition 1.18: MOSFET – High-Level Overview

A **MOSFET** (Metal-Oxide-Semiconductor Field-Effect Transistor) is a type of FET that uses its gate to control the flow of current. It consists of two default starting states:

- **Enhancement:** Normally **off** (i.e., no current flows), until such voltage is applied:
  - **N-Channel Enhancement:** A positive voltage.
  - **P-Channel Enhancement:** A negative voltage.
- **Depletion:** Normally **on** (i.e., current flows), until such voltage is applied:
  - **N-Channel Depletion:** A negative voltage.
  - **P-Channel Depletion:** A positive voltage.

Definition 1.19: MOSFET – Anatomy of N-Channel

A MOSFET **N-Channel Enhancement** is constructed as follows:

- **Substrate:** A base-layer of P-type material from which all parts will build upon.
- **Source & Drain:** Two motes are dug at either ends of the substrate and filled with N-type material; One for our **source** and the other for our **drain**. Two metal contacts are placed on these motes (our terminals); A body of metal is connected to the bottom of the substrate (**base/body terminal**), which connects to the source terminal.
- **Gate:** A metal contact pad is placed between the motes on top of the SiO_2 layer, forming the **gate terminal**. A layer of silicon dioxide (SiO_2) is sandwiched between the gate and the substrate. Since SiO_2 is a superb insulator, it prevents the gate terminal from touching/interacting with the substrate.
- **Channeling:** SiO_2 is a **dielectric** material, meaning that when a charge is applied to one side, the opposite charge builds on the other side, creating an electric field. When a positive charge is applied, it attracts negative electrons from the other side, creating a **channel** (i.e., bridge) between the two motes (source to drain), allowing current.

The **M**etal from gates, the **O**xide from the SiO_2 layer, the **S**emiconductor from the substrate and motes, and the **FET** from the field-effect, gives us the name **MOSFET**.

For **N-Channel Depletion**, A *thin* N-type substrate-channel is already present, bridging the two motes (source and drain) together. Once a negative charge is applied to the gate, positive holes are attracted, weakening the channel; This effectively stops the flow of current.

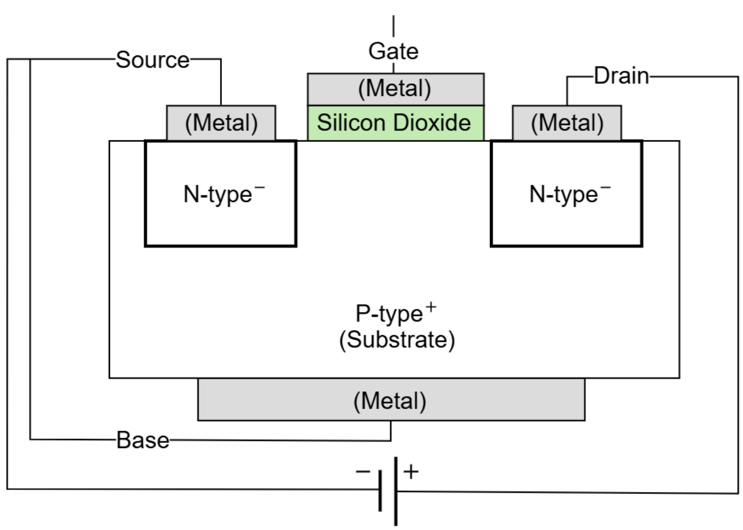


Figure 4.11: N-Channel Enhancement (off). Negative battery side to source, positive to drain.

Theorem 1.4: Flow of Electrons

Recall that a body of electrons that are negatively charged (low potential), have a surplus of electrons. A positive charge (high potential) reflects a deficit of electrons. Therefore, when given the chance (alike water), electrons will flow to fill the void.

Tip: An empty stomach has a high potential for food, while a full stomach has a low potential, as there's not much more room left to stuff food into.

Definition 1.20: MOSFET – N-Channel Battery Configuration

One battery is used to power the MOSFET, and another to control the gate. The source connects to the negative, and the drain to the positive side of the battery.

The gate is connected to a second battery, which can be either positive or negative, depending on the MOSFET type. The **other** end of this battery is connected to the source terminal.

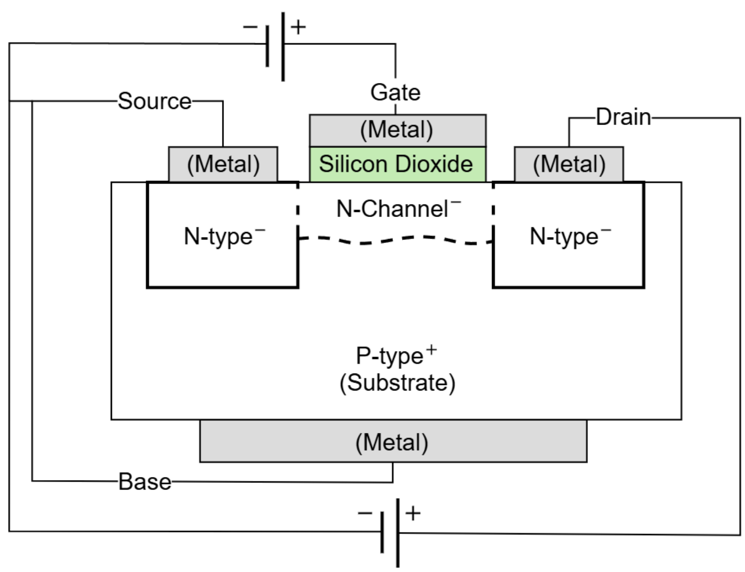


Figure 4.12: N-Channel Enhancement (on). Negative battery side to gate, positive to source. Positive charge given to the gate attracts negative electrons on the other side of the SiO_2 layer, creating a channel between the source and drain.

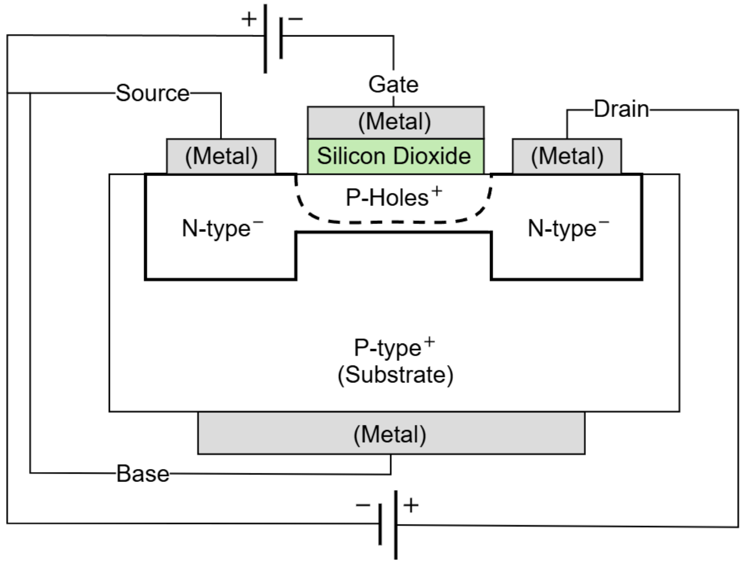


Figure 4.13: N-Channel Depletion (on). Negative charge to gate, creates holes into the channel.

Definition 1.21: MOSFET – Anatomy of P-Channel

The P-Channel variation follows the same logic as the N-Channel Definition (1.19); Instead, we swap N-type for P-type materials, and vice versa. Then apply negative for Enhancement and positive for Depletion on the gate to open or close the channel respectively (1.18). **In particular**, source now connects to a high potential and drain to a low potential.

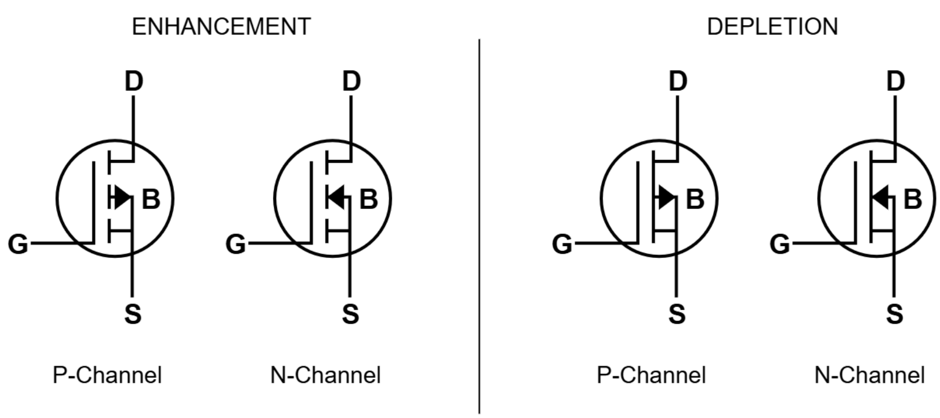


Figure 4.14: MOSFET symbols, **G** (gate), **S** (source), **D** (drain), and **B** (body) terminals.

4.1.4 Logic Gates & Functional Completeness

This section will cover how we take MOSFETs and use them to build logic gates.

Definition 1.22: Gate-Source Voltage V_{GS}

Recall Definition (1.20), the gate battery's opposite end is connected to the source terminal. This serves as a zero-volt reference for the gate terminal. The difference in potential between the gate and source terminals is called the **gate-source voltage** (V_{GS}):

$$V_{GS} = V_G - V_S,$$

Once V_{GS} exceeds the threshold voltage V_{TN} (for N-channel) or is below the threshold voltage V_{TP} (for P-channel), the MOSFET turns on, allowing current to flow from source to drain.

Tip: Notice that source takes from gate, i.e., for N-channel, the gate must overcome the source's negative charge. Hence we must exceed a threshold voltage to turn on the MOSFET. For P-channel, the same logic applies, but in reverse, as the components are implemented in a complementary manner.

Definition 1.23: Pull-Up & Pull-Down Switches

Let V_{DD} be the positive supply ("logical 1") and ground (0 V) be "logical 0." A CMOS logic gate uses:

- **Pull-Down Switch** (Off: 0, On: 1): N-channel enhancement. If $V_{GS} > V_{TN}$, source connects to drain, producing a logical 1.
- **Pull-Up Switch** (Off: 1, On: 0): P-channel enhancement. If $V_{GS} < V_{TP}$, source connects to V_{DD}, producing a logical 1.

Tip: Think of pull-down as "pulling down to the ground" to allow electrons to escape.

Additionally, the "DD" in V_{DD} does not stand for anything; it was made not to be confused with V_D, the voltage at the drain terminal. Though, unimaginative, it is simply convention.

Definition 1.24: CMOS Logic Gate

A **CMOS logic gate** is a circuit that uses **C**omplementary **MOS**FETs to perform logical operations. It consists of:

- **Pull-Down Network (PDN):** N-channel MOSFETs (NFET) connected to ground.
- **Pull-Up Network (PUN):** P-channel MOSFETs (PFET) connected to V_{DD}.

The output is high when the PUN is active and the PDN is inactive, and vice versa.

Pull-up	Pull-down	Digital
on	off	"1"
off	on	"0"
on	on	undefined
off	off	disconn.

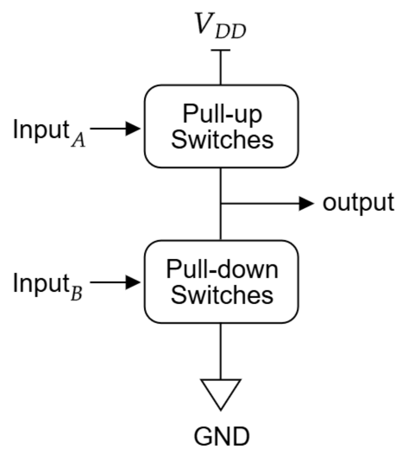
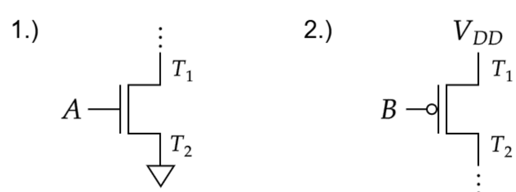


Figure 4.15: Simple CMOS logic gate, where **GND** stands for ground, V_{DD} for positive supply. **Note:** That even if the circuit is disconnected, the output may still *"remember"* its last state for some time until the charge dissipates.

Definition 1.25: Simplified NFET & PFET Symbols

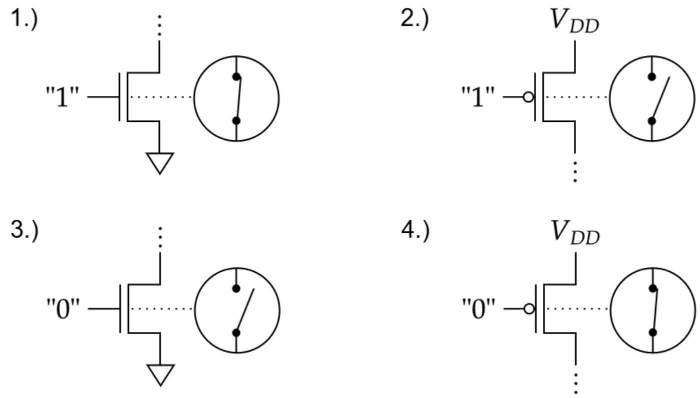
Below are input gates A and B, with two other terminals T_1 and T_2, simplifying Figure(4.15):



(1) NFET, T_1 output, and T_2 ground; (2) PFET, T_1 is V_{DD} (typically 1V), and T_2 output [6].

Definition 1.26: Open & Closed Circuits – NFET & PFET Logic

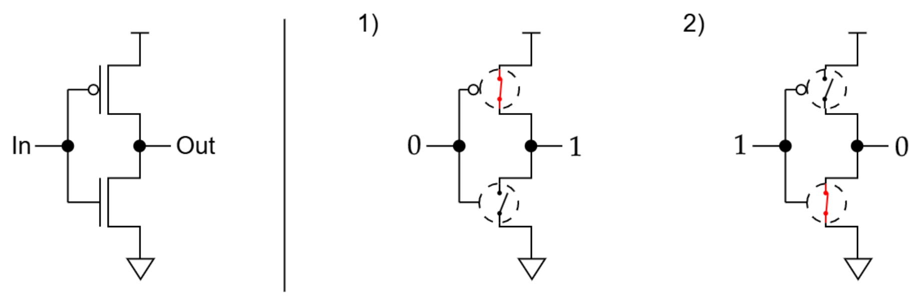
A **closed circuit** is a complete path for current to flow, while an **open circuit** is an incomplete path:



(1) An NFET is closed when given a digital 1 (high voltage), while (2) a PFET, is open; (3) NFET, open when given 0 (low voltage), while (4) a PFET is closed.

Definition 1.27: MOSFET Logic Gate – Not

Below shows a logic gate, NOT, using MOSFETs (NFET and PFET):



Both the NFET and PFET share the same input. The top line represents V_{DD} (positive supply), while the bottom line is ground (triangle). (1) input low, NFET is open, PFET is closed, output is high from V_{DD}; (2) input high, NFET is closed, PFET is open, output is low from ground. **Important:** A black dot connecting two or more lines represents a connection.

Definition 1.28: Functional Completeness

it can be used to express all possible Boolean functions. For example, the set of operators {AND, OR, NOT} is functionally complete. However, we can do better. The NAND gate (and NOR gate) by itself is functionally complete.

Example 1.5: Functionally Complete Sets of Operators

1. {NAND} alone

$$\begin{aligned} \neg A &= A\,\mathrm{NAND}\,A, \\ A \wedge B &= \neg\big(A\,\mathrm{NAND}\,B\big) = \big(A\,\mathrm{NAND}\,B\big)\;\mathrm{NAND}\;\big(A\,\mathrm{NAND}\,B\big), \\ A \vee B &= (A\,\mathrm{NAND}\,A)\;\mathrm{NAND}\;(B\,\mathrm{NAND}\,B). \end{aligned}$$

2. {NOR} alone

$$\begin{aligned} \neg A &= A\,\mathrm{NOR}\,A, \\ A \vee B &= \neg\big(A\,\mathrm{NOR}\,B\big) = \big(A\,\mathrm{NOR}\,B\big)\;\mathrm{NOR}\;\big(A\,\mathrm{NOR}\,B\big), \\ A \wedge B &= (A\,\mathrm{NOR}\,A)\;\mathrm{NOR}\;(B\,\mathrm{NOR}\,B). \end{aligned}$$

3. $\{\wedge, \neg\}$ alone

$$\begin{aligned} \neg A &= \neg A, \\ A \wedge B &= A \wedge B, \\ A \vee B &= \neg\big(\neg A \wedge \neg B\big). \end{aligned}$$

4. $\{\vee, \neg\}$ alone

$$\begin{aligned} \neg A &= \neg A, \\ A \vee B &= A \vee B, \\ A \wedge B &= \neg\big(\neg A \vee \neg B\big). \end{aligned}$$

5. $\{\rightarrow, \neg\}$ alone

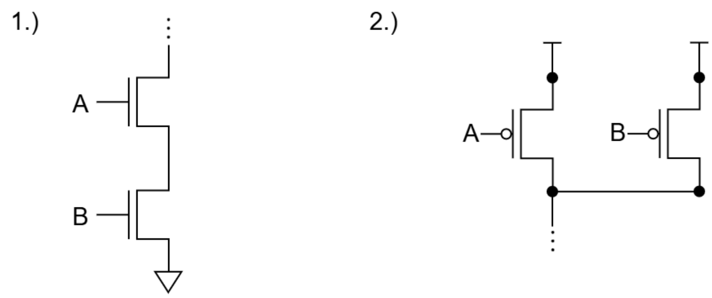
$$\begin{aligned} \neg A &= \neg A, \\ A \vee B &= \neg A \rightarrow B, \\ A \wedge B &= \neg\big(A \rightarrow \neg B\big). \end{aligned}$$

Recall, $A \rightarrow B$ is logically equivalent to $\neg A \vee B$. ■

Let's brainstorm possibilities for an AND and NAND, and elaborate on Defintition (1.24):

Theorem 1.5: Balancing Series & Parallel Connections

It's important in a CMOS circuit that the PUN and PDN are in fact complements of each other. Below illustrates two types of connections:



(1) Shows a NFET **series** connection (i.e., sequentially connected). Theoretically in isolation, this represents an AND gate ($A \cdot B$), meaning both A and B must be high to close the circuit.

(2) Shows a PFET in **parallel** connection (i.e., side-by-side connected). As per complementarity, this represents the NAND gate ($\overline{A} + \overline{B} = \overline{A \cdot B}$), by De Morgan's Law; Either A or B must be low to close the circuit.

Hence to **balance**, between the PUN and PDN, we must ensure that:

- each NFET series requires a PFET parallel.
- each PFET series requires a NFET parallel.

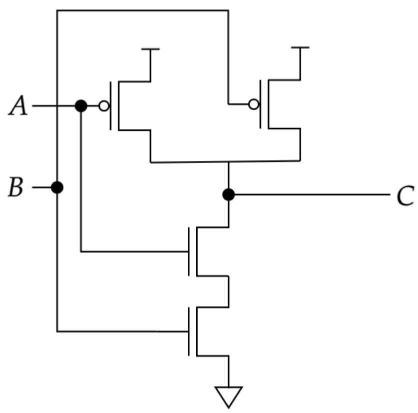
This keeps our networks complementary, ensuring that <u>when one is closed, the other is open</u>.

Tip: Think about how to create this before moving to the next page. understand that the placement of the output determines the logic of the circuit. Since PUNs default to high, think about how that might affect the output.

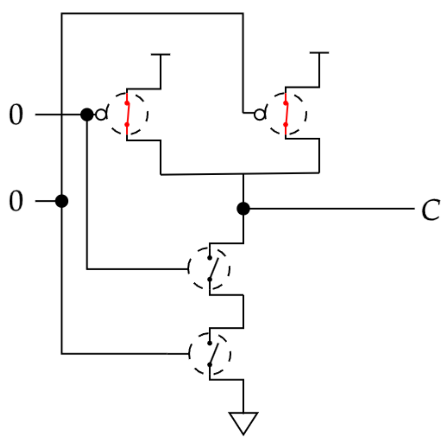
In CMOS, there must be both a PUN and PDN, think about how to balance the (2), from the above Theorem (1.5), to create a NAND gate.

Theorem 1.6: CMOS Logic Gate – NAND

Combining both networks in Theorem (1.5) yields a NAND gate in the following configuration:

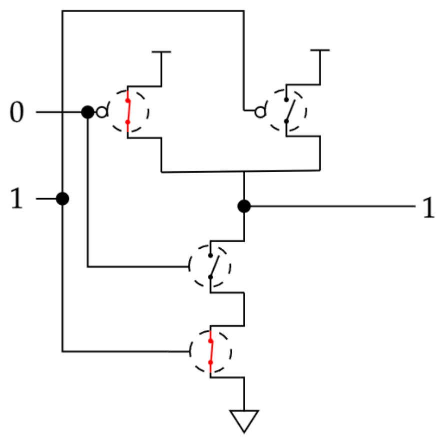


A	B	C
0	0	1
0	1	1
1	0	1
1	1	0



A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Figure 4.16: Both inputs are (0,0), PUN closed, PDN open, hence the output is high (1).



A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Figure 4.17: Both inputs are (0,1), PUN half-closed reaching the output, PDN half-closed, but can't reach the output; Hence the output is high (1).

(Taking a break from writing. I intended to write all the way to show how with the PUN and PDN configurations achieve binary addition, subtraction, multiplication, and division. But I think this is enough for now. I will come back to it later.)

— 5 —

Proving Algorithms

5.1 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.**

5.2 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.

```
Function GS(E,A):
    M ← ∅;
    while there is some unmatched element in E do
        e ← next unmatched element in E;
        a ← next avalaible preferred choice of e;
        if a is not yet matched then
            match e and a;
            add the pair (e, a) to M;
        else
            if a prefers e over their current match then
                match e and a, replacing the current match;
                update M accordingly;
    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.

Exercise 2.1: Given sets $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_n\}$ (True/False):

1. If a_i and b_j rank each other first, then in any stable matching, a_i and b_j must be matched.
2. If a_i and b_j rank each other last, then in any stable matching, a_i and b_j are never matched.
3. If A's are proposing to B's, with a_i and a_j both preferring b_k, but a_i ranked b_k higher, then a_i will be matched with b_k.

Exercise 2.2: Given the following preference lists, in any stable matching:

1. What is the highest ranked match Alice can receive from her list?
2. What is the lowest ranked match Alice can receive from her list?
3. Is there a unique stable matching?
4. Dolly put Dakar last on their list. Therefore, Dolly won't be matched with Dakar (True/False).

Hospital	**1st**	**2nd**	**3rd**	**4th**	**5th**
Acton	Bart	Edgar	Alice	Dolly	Carol
Boston	Alice	Carol	Edgar	Bart	Dolly
Chicago	Carol	Alice	Dolly	Edgar	Bart
Dakar	Dolly	Bart	Carol	Alice	Edgar
Erie	Edgar	Dolly	Bart	Carol	Alice

Resident	**1st**	**2nd**	**3rd**	**4th**	**5th**
Alice	Erie	Chicago	Boston	Dakar	Acton
Bart	Erie	Acton	Dakar	Boston	Chicago
Carol	Dakar	Boston	Chicago	Acton	Erie
Dolly	Acton	Erie	Chicago	Boston	Dakar
Edgar	Boston	Acton	Dakar	Erie	Chicago

Answers on the next page.

Answer 2.1:

1. **True.** The proposing side gets their best pick. If there are collisions, then the other side gets their best pick.
2. **False.** If all other matches prefer each other over a_i and b_j, then a_i and b_j will be matched.
3. **False.** b_k will choose their best pick, either a_i or a_j.

Answer 2.2:

1. **Chicago.** If Residents propose first, Alice and Bart collide in preferring Erie. Though Bart ranks higher on Erie's list. Alice then prefers Chicago, for which no one else prefers.
2. **Boston.** If Hospitals propose first, Boston gets their 1st choice, Alice.
3. **No.** Matchings change depending on who proposes first.
4. **No.** If Hospitals propose first, Dakar matches with Dolly, as no other Hospitals prefer Dolly.

Tip: The key is to look for collisions in preferences. If there are no collisions, the proposing side gets their best pick. If there are collisions, the other side gets their best pick. After each collision is resolved look again for new collisions.

In the above example, if Residents propose first, after sorting out Alice and Bart's collision, everyone else got their first pick, leaving Chicago to Alice. If Hospitals propose first, they all get their first pick having no collisions.

— 6 —

Graphs and Trees

6.1 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 (6.1) shows an undirected graph:

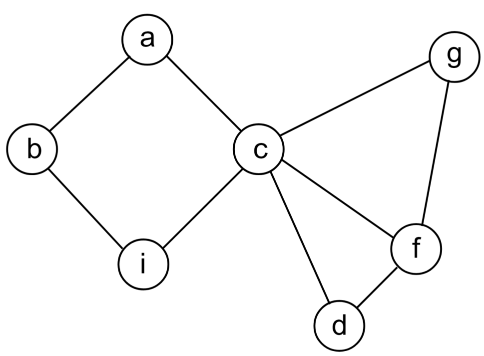


Figure 6.1: An undirected graph with 7 vertices and 9 edges.

Node a has a degree of 2, and node c has a degree of 5.

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 (6.2) shows a directed graph:

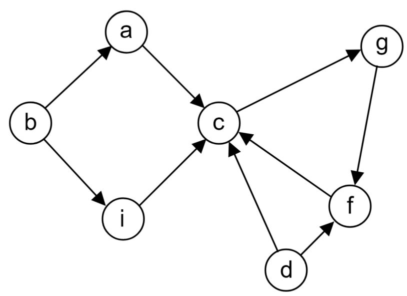


Figure 6.2: A directed graph with 7 vertices and 9 edges.

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 (6.3) shows a weighted graph:

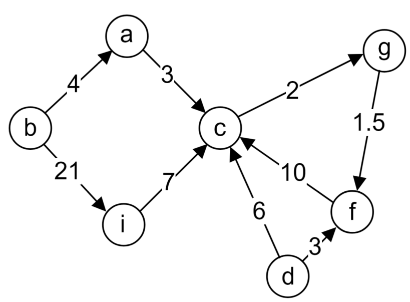


Figure 6.3: A weighted graph with 7 vertices and 9 edges.

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 (6.4), a simple path $h \leftrightarrow b \leftrightarrow i \leftrightarrow c \leftrightarrow d$ is shown:

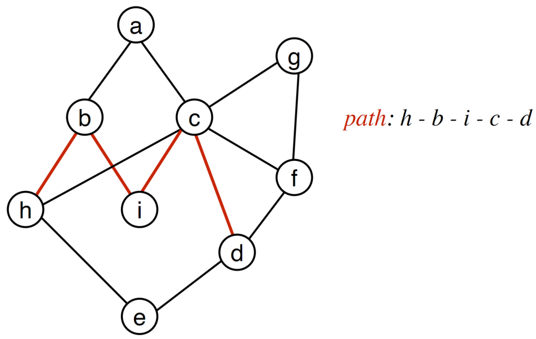


Figure 6.4: A graph with a simple path from h to d.

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.

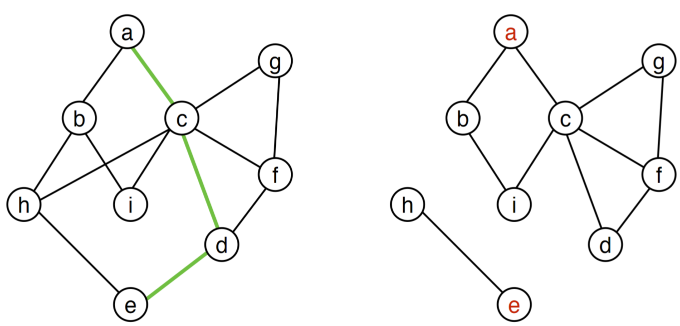
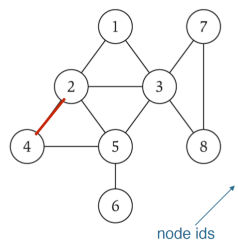


Figure 6.5: A connected graph $a \leftrightarrow c \leftrightarrow d \leftrightarrow e$ and disconnected graph.

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.



$$
\begin{array}{c c}
 & \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} &
\begin{bmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$

undirected graph

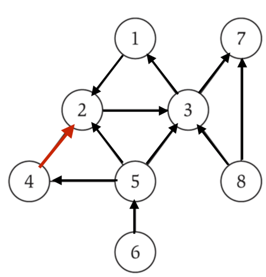
Figure 6.6: An adjacency matrix where the path $4 \leftrightarrow 2$ is highlighted ($A[2][4]$ or $A[4][2]$)

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:**
Index edge $A[i][j]$**:** $\Theta(1)$; **List neighbors** $A[i]$: $\Theta(n)$; **List all edges:** $\Theta(n^2)$.



$$
\begin{array}{c c}
 & \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} &
\begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$

Figure 6.7: An adjacency matrix of a directed graph with path $4 \to 2$ highlighted.

Definition 1.8: Adjacency List

An **adjacency list** is a list of keys where each key has a list of neighbors; This often takes the form of a hash-table: **Space Complexity:** $\Theta(n+m)$ for n nodes and m total edges; **Time Complexities: Index key:** $\Theta(1)$; **List key neighbors:** $\Theta(\#\text{ of outdegrees})$; **List all edges:** $\Theta(n+m)$; **Insert edge:** $\Theta(1)$. **Note:** $m \leq n^2$ (all nodes connected to all nodes), though typically in practice, $m < n$.

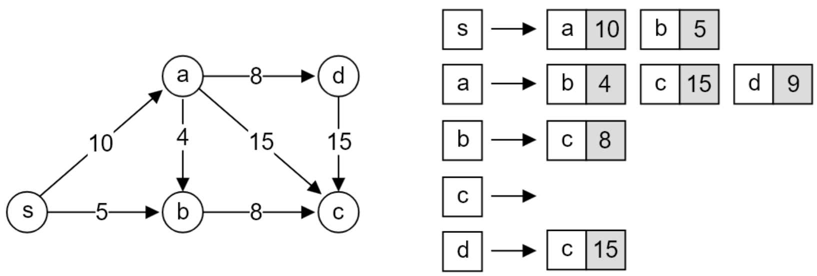


Figure 6.8: An adjacency list of a directed graph, notably c has no outdegrees.

6.2 Breath-First and Depth-First Search

6.2.1 High-Level Overview

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 (6.7), $1 \to 2 \to 3 \to 1$ form a cycle.

Definition 2.2: Trees

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. **Interior nodes** have at least one child.

Tip: Watch this entire section: BFS & DFS (Edge Types, Traversal Orders, & Code).

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

A tree **root** is a designated starting node, which all other nodes follow. The root has no parent/ancestors (proceeding node), and all other nodes are descendants (children).

A **subtree** is a tree formed by picking another node as the root, representing a subset of the original tree.

Definition 2.4: Levels and Heights

The **level** (or **depth**) of a node v in a tree is the number of edges on the path from the root to v, so level(root) $= 0$. The **height** of a node v is the number of edges on the longest downward path from v to any leaf. The **height** of a tree is defined as the height of its root node.

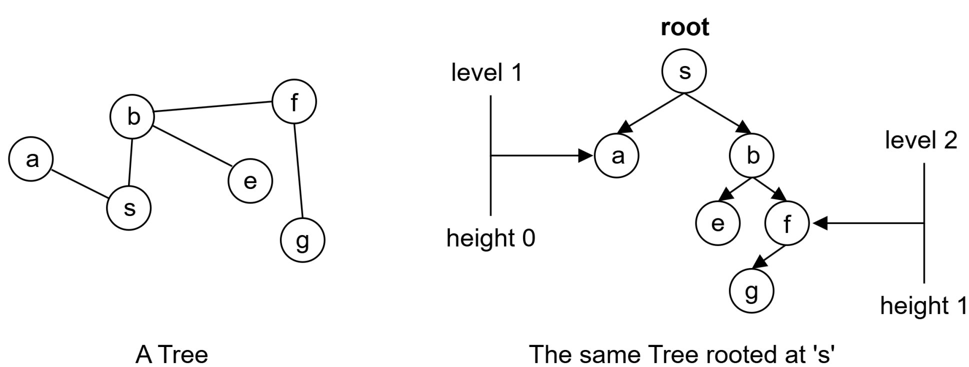


Figure 6.9: The right shows a rooted tree with root s, with a and b as direct children. A possible subtree starts with node b, only consisting of $\{b, e, f, g\}$. **Note:** A tree is not necessarily directed, while a rooted tree is (floods outwards from the root)

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.

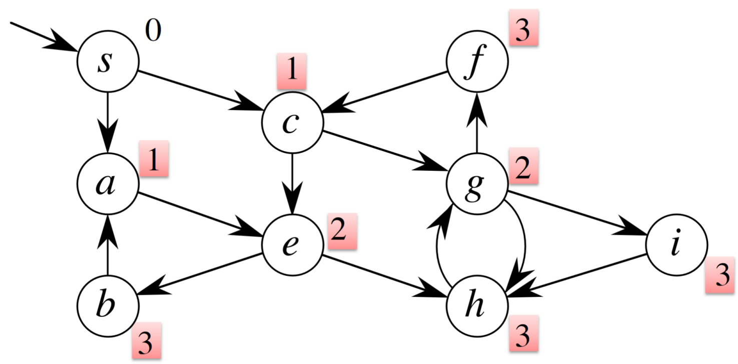


Figure 6.10: A BFS tree traversal preformed on a graph with each level enumerated

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:

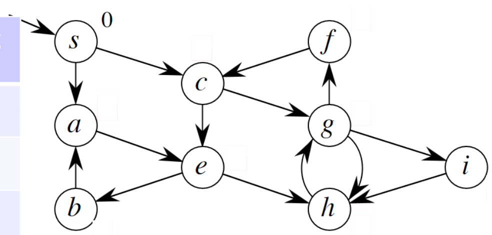
Function 2.1: BFS Algorithm - BFS(s)

Breadth-First Search starting from node s.

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

```
Function BFS(s):
    for each v ∈ V do
        Level[v] ← ∞;
    Level[s] ← 0;
    Add s to Q;
    while Q not empty do
        u ← Q.Dequeue();
        for each v ∈ G[u] do
            if Level[v] = ∞ then
                Add edge (u, v) to tree T (parent[v] = u);
                Add v to Q;
                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 6.11: A table showing the queue at each level of iteration

We analyze the time and space complexity in the below Figure (6.12):

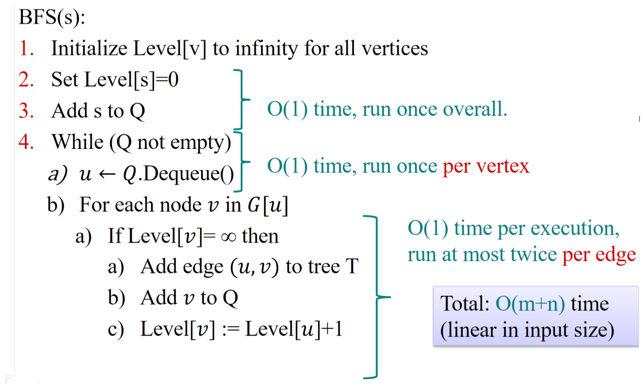


Figure 6.12: An analysis showing $O(m+n)$ for both time and space complexity

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 $i + 1$. ■

Definition 2.6: 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.

```
Function DFS(G):
    for each u ∈ G do
        u.state ← unvisited;
    time ← 0;
    for each u ∈ G do
        if u.state == unvisited then
            DFS-Visit(u);
Function DFS-Visit(u):
    time ← time + 1;
    u.d ← time ;
    // record discovery time;
    u.state ← discovered;
    for each v ∈ G[u] do
        if v.state == unvisited then
            DFS-Visit(v);
    u.state ← finished;
    time ← time + 1;
    u.f ← time;
    // 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. ∎

6.2.2 Edge Classifications – Directed Graphs

To build our intuition around edge classifications, we'll follow a walkthrough of DFS & BFS on directed graphs to build our intuition. We'll follow the below format:

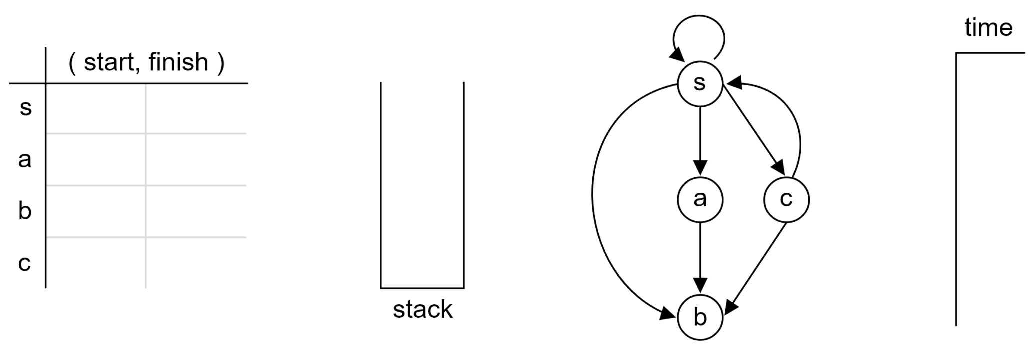


Figure 6.13: A table for start and finish times of each visited nodes, our stack to keep track of traversals, the graph that shall be traversed, and a timer for DFS (BFS will use a queue).

Let's start by filling out a partial DFS traversal:

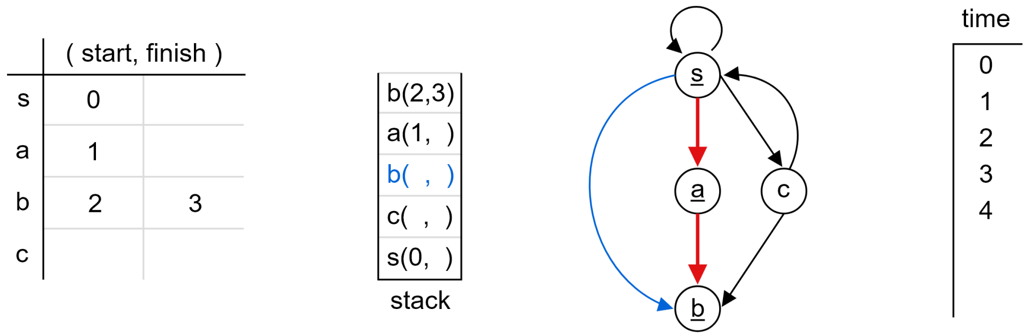


Figure 6.14: So far we have asked s to put all their children on the stack at timestamp 0. We first investigate the first child a's branch at time 1. It gives us b, for which we time stamp (2,3) for start and finish. We highlight the first the alternative b branch in blue for distinction. The edges that are in our (start, finish) table are **Tree Edges**.

Definition 2.7: Forward Edge

A **non-tree edge** that connects an already discovered node to a descendant in the DFS tree is called a **forward edge**. Concretely, (u, v) is a forward edge if given $u(s_1, f_1)$ and $v(s_2, f_2)$ of node id and a start finish time tuple, we have: $s_1 < s_2 < f_2 < f_1$.
Where the **start time** of u is less than v, and the **finish time** of u is greater than v.

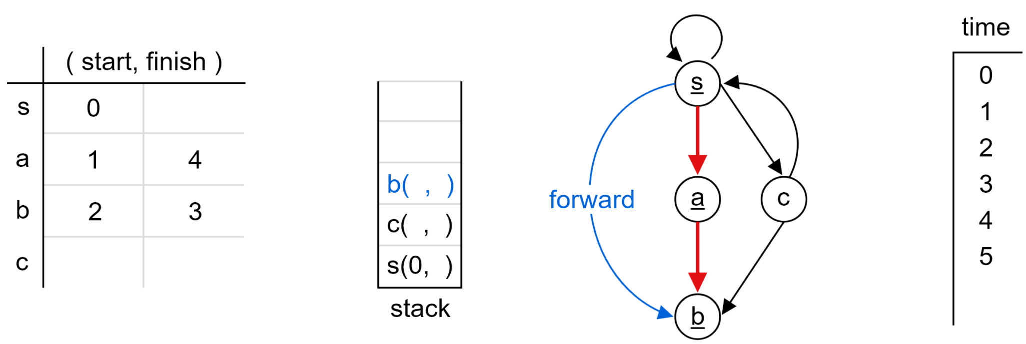


Figure 6.15: Continuing from Figure (6.14), b and a are taken off the stack. The next b on the stack is a **forward edge**, as s has a smaller start time, and a foreseeable finish time than the table's b.

The next two are **cross and back edges**:

Definition 2.8: Cross Edge

A **non-tree edge** that connects branches of a tree is called a **cross edge**. In particular, (u, v) is a cross edge if v is not a descendant or ancestor of u. Concretely, given $u(s_1, f_1)$ and $v(s_2, f_2)$ of node id and a start finish time tuple, we have: $s_2 < s_1 < f_2 < f_1$.
Where the **start and finish time** of u is greater than v.

Definition 2.9: Back Edge

A **non-tree edge** that connects a node to an ancestor in the tree is called a **back edge**. Concretely, given $u(s_1, f_1)$ and $v(s_2, f_2)$ of node id and a start finish time tuple, we have: $s_2 < s_1 < f_1 < f_2$.
Where the **start time** of u is greater than v, but the **finish time** of is less than v.

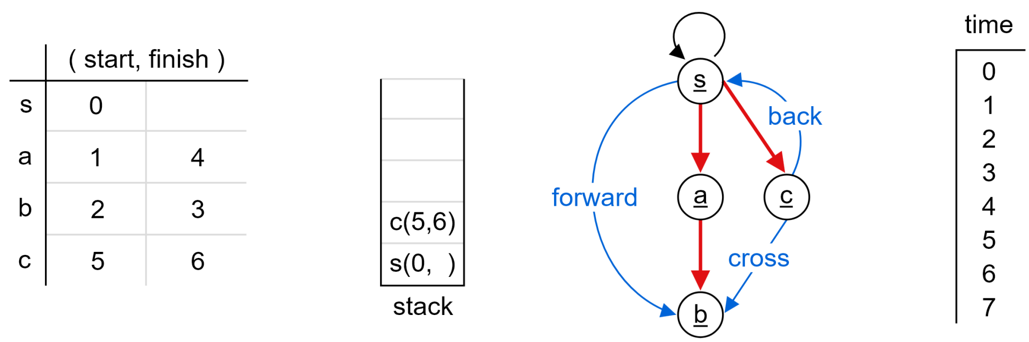


Figure 6.16: Continuing from Figure (6.15), b and a are taken off the stack. Next on the stack is c, which reveals children b and s; Here, b has already been processed (cross edge), and s discovered, but yet to be finished (back edge). There is nothing else to explore, so we finish c. **Note:** There is no classifications for self referential edges.

Next we'll see how our classifications work on BFS.

Starting with the same graph as before we instead run BFS:

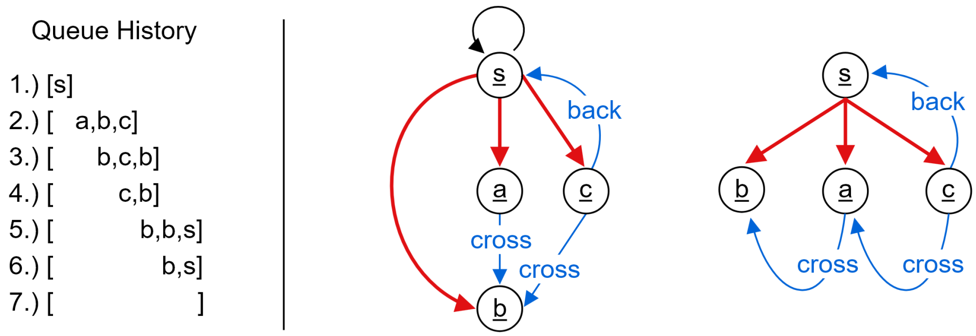


Figure 6.17: We start with s modeling the queue history on the left. The first graph is our original graph finished, and on the right it is redrawn for clarity.

Theorem 2.3: DFS & BFS Directed Graphs

Between DFS and BFS, the following holds:

- **DFS**: Tree, back, forward, and cross edges (all edges).
- **BFS**: Tree, cross, and back edges (no forward edges).

Additionally, one may find the following theorem helpful:

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. ■

Next we deal with undirected graphs briefly and give a summary:

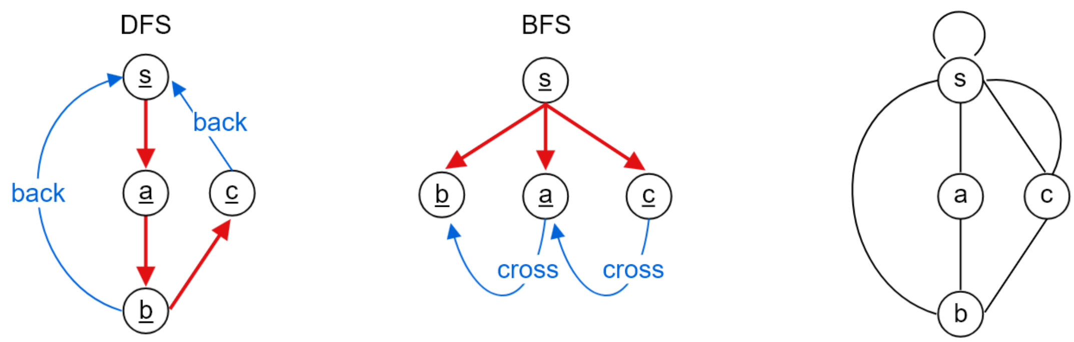


Figure 6.18: Here DFS and BFS are shown, with the original graph on the far right. Here it's quite peculiar to have a double-edge in an undirected graph. In our case it's redundant and may be counted as a single edge.

Summary:
This allows us to summarize the edge classifications for both directed and undirected graphs in the below table:

Graph Type	**DFS Edge Types**	**BFS Edge Types**
Directed Graphs	Tree, Back, Forward, Cross	Tree, Back, Cross
Undirected Graphs	Tree, Back	Tree, Cross

Table 6.1: Edge classifications for DFS and BFS in directed and undirected graphs.

Edge Type	**Start Condition**	**Finish Condition**
Forward Edge	less	greater
Cross Edge	greater	greater
Back Edge	greater	less

Table 6.2: Summary of start and finish conditions for edge types by comparing the starting node with its endpoint. E.g., in a forward edge (u, v), the start time of u is less than v, but the finish time is greater.

6.3 Tree Types & Traversals

6.3.1 Binary Trees: 2 Children

First we'll define a basic binary tree and then all its extensions.

Definition 3.1: Binary Tree

A **binary tree** is a tree where each parent node has at most two children.

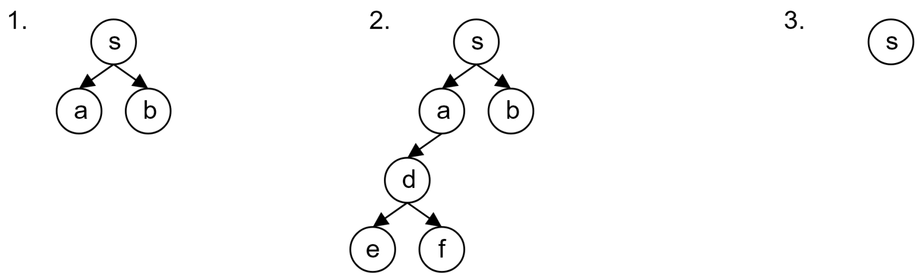


Figure 6.19: These are three examples of binary trees, including a single node (3).

To simply do a binary tree traversal, we may employ the following methods:

Function 3.1: DFS Binary Tree Traversal - `DFS`(T)

Depth-First Search on binary tree T (recursive).

Input: Binary tree T.
Output: Nodes in pre-order, in-order, and post-order.

```
Function DFS(T):
    if T is not empty then
        Visit node T;
        DFS(T.left);
        DFS(T.right);
```

Time Complexity: Given n nodes, and a force of a tree format, we have $n-1$ edges. Therefore with m edges, $n+m = n+(n-1)$, hence $O(n)$ time.
Space Complexity: again, we have $O(n)$ space for the stack.

→ **BFS has the same complexities as per the same reasoning as the above function.**

The order in the above function is called **pre-order traversal**;

Definition 3.2: Order Traversals

The order of traversal refers to the order in which nodes are visited/processed:

- **Pre-order:** Visit the node, then left, then right.
- **In-order:** Visit left, then the node, then right.
- **Post-order:** Visit left, then right, then the node.

In particular, **in-order traversal** is simply a run of **BFS** as it processes the tree level by level starting from the root.

Below is an example of all three:

Example 3.1: Binary Tree Traversals (Part 1)

```
# Pre-order Traversal
def PreOrder(T):
    if T is not None:
        visit(T)
        PreOrder(T.left)
        PreOrder(T.right)

# In-order Traversal
def InOrder(T):
    if T is not None:
        InOrder(T.left)
        visit(T)
        InOrder(T.right)

# Post-order Traversal
def PostOrder(T):
    if T is not None:
        PostOrder(T.left)
        PostOrder(T.right)
        visit(T)

# Level-order Traversal
def LevelOrder(T):
    BFS(T)
```

■

Now to show how this actually looks:

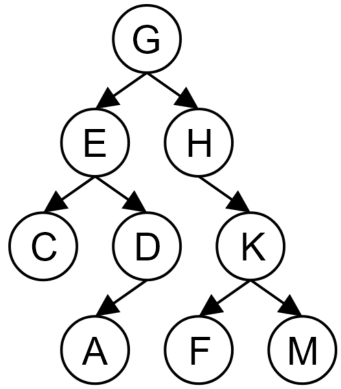


Figure 6.20: The above binary tree has the following traversals:

- **Pre-order:** G, E, C, D, A, H, K, F, M
- **In-order:** C, E, A, D, G, H, F, K, M
- **Post-order:** C, A, D, E, F, M, K, H, G
- **Level-order:** G, E, H, C, D, K, A, F, M

Notably, if there is no left or right child to explore for that particular traversal order, that node then and there is processed. For example, in in-order traversal, since C has no left child, it is processed first. The same goes for Post-order, where D has no right child, so it is processed.

Definition 3.3: Types of Binary Trees

There are several types of binary trees, each with its own properties:

- **Full Binary Tree:** Every node has either 0 or 2 children.
- **Complete Binary Tree:** All levels are fully filled except possibly the last level, which is filled from left to right.
- **Perfect Binary Tree:** All interior nodes have two children and all leaves are at the same level.
- **Balanced Binary Tree:** The height of the left and right subtrees of every node differ by at most one.

Note: A full binary tree is not necessarily complete, and a complete binary tree is not necessarily full.

Observe the following examples of binary trees:

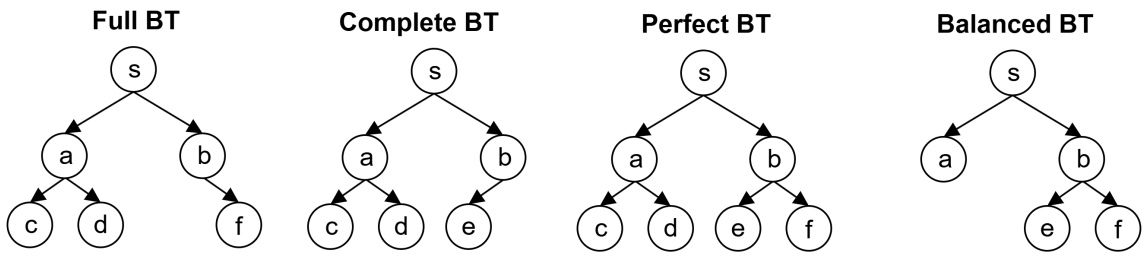


Figure 6.21: These are examples of different types of binary trees (BT).

6.3.2 Binary Search Tree

A binary search tree is a stricter form of a binary tree, which gives us an efficient search algorithm.

Definition 3.4: Binary Search Tree (BST) – Overview

A **binary search tree** is a binary tree where:

$$\text{left child} < \text{parent node} \leq \text{right child}$$

Note: A Binary Tree **is not** a Binary Search Tree, unless it satisfies the above conditions.

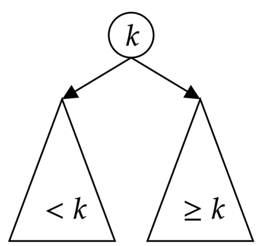


Figure 6.22: Visual Representation of Definition (3.4), where k is the parent node with its respective left (less than) and right (greater than or equal to) nodes.

The next few pages we discuss **Searching, Insertion, and Deletion** in a binary search tree.

Function 3.2: Binary Search Tree (BST) – Searching

Given a BST root node T and a desired value v, we can search for v in T as follows:

Input: BST root node T, value v.
Output: Node containing v or null if not found.

```
Function Search(T, v):
    while T is not null do
        if T.value = v then
            return T;
        if v < T.value then
            T ← T.left;
        else
            T ← T.right;
    return null;
```

Time Complexity: $O(h)$ where h is the height of the tree. In a balanced BST, this is $O(\log n)$, but in the worst case (unbalanced), it can be $O(n)$.

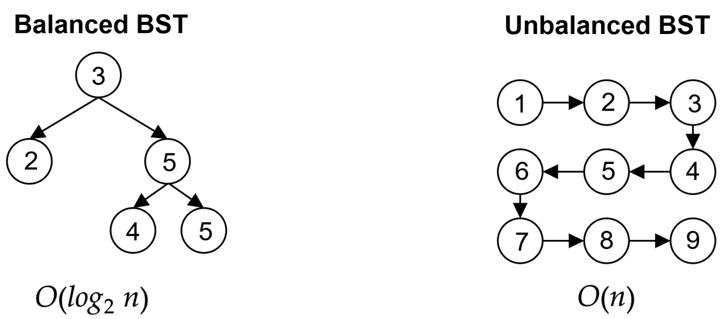


Figure 6.23: Searching for a value in a binary search tree. The worst case is a noodle (all right nodes), forcing n traversals, i.e., the height (within a constant). One could imagine these nodes as complex objects, where the value v is a key, and the node itself contains more data.

Theorem 3.1: Property of Binary Search Trees

In a binary search tree, for any node N:

- All values in the left subtree of N are less than the value of N.
- All values in the right subtree of N are greater than or equal to the value of N.

Insertion is the same an searching, except we insert upon reaching a null node.

Function 3.3: Binary Search Tree (BST) – Insertion

Given a BST root node T and a value v, we can insert v into T as follows:

Input: BST root node T, value v.
Output: Updated binary search tree with v inserted.

```
Function Insert(T, v):
    parent ← null; current ← T;
    while current is not null do
        parent ← current;
        if v < current.value then
            current ← current.left;
        else
            current ← current.right;

    // After the while loop terminates
    if parent is null then
        return v ; // Tree was empty, v becomes root
    if v < parent.value then
        parent.left ← v;
    else
        parent.right ← v;
    return T;
```

Time Complexity: $O(h)$ where h is the height of the tree. In a balanced BST, this is $O(\log n)$, but in the worst case (unbalanced), it can be $O(n)$.

Deletion is a bit more complex, as we have to consider removing intermediate nodes:

Theorem 3.2: Binary Search Tree (BST) – Deletion

When deleting a node from a binary search tree, we must consider three cases:

- v **has no children:** Remove the parent's reference to v.
- v **has one child:** Replace the parent's reference to v's child.
- v **has two children:** Replace the parent's reference to v with either:
  - The **L**argest value in v's **L**eft subtree (in-order predecessor).
  - The **R**unt (smallest) value in v's **R**ight subtree (in-order successor).

This ensures that the binary search tree properties are maintained after deletion.

Function 3.4: Binary Search Tree (BST) – Deletion

Given a BST root node T and a value v, we can delete v from T as follows:

Input: BST root node T, value v.
Output: Updated binary search tree with v deleted.

```
Function Delete(T, v):
    parent ← null; current ← T;
    while current is not null and current.value ≠ v do
        parent ← current;
        if v < current.value then
            current ← current.left;
        else
            current ← current.right;
    if current is null then
        return T ; // Value not found
    // Case 1: Node has at most one child
    if current.left is null or current.right is null then
        if current.left is not null then
            child ← current.left;
        else
            child ← current.right;
        if parent is null then
            return child ; // Deleted root
        if parent.left = current then
            parent.left ← child;
        else
            parent.right ← child;
        return T;
    // Case 2: Node has two children
    succParent ← current;
    succ ← current.right ; // Runt of Right subtree
    while succ.left is not null do
        succParent ← succ;
        succ ← succ.left;
    current.value ← succ.value;
    if succParent.left = succ then
        succParent.left ← succ.right;
    else
        succParent.right ← succ.right;
    return T;
```

Time Complexity: $O(h)$ where h is the height of the tree. In a balanced BST, this is $O(\log n)$, but in the worst case (unbalanced), it can be $O(n)$.

Consider an example as we build a binary search tree and performing the above operations:

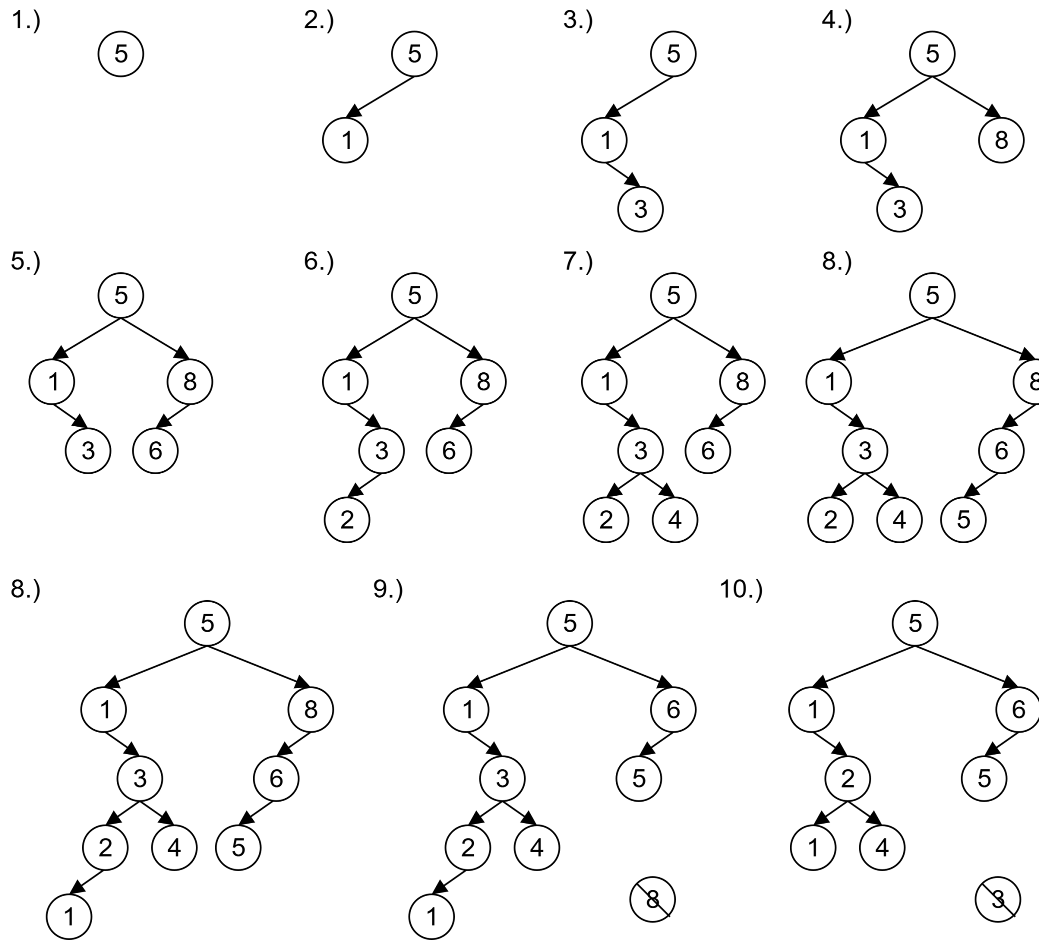


Figure 6.24: **Note:** We use the Largest Left child (in-order predecessor) to replace the 2-children case. From 1–9 we insert values. From 10–11 we delete. In (10) we delete 8 (1-child case), we replace its parent's reference to 8 with its only child 6. In (11) we delete 3 (2-children case), we replace its parent's reference to 3 with the largest left child, which is 2. One could imagine the in-order successor (smallest right child) would have 4 instead of 2, with 2 and 1 on its left subtree.

6.3.3 2-3 Trees: 3 Children Search

We can keep extending the binary tree to allow for more children, which allows us to search faster [11].

Definition 3.5: Ternary & N-ary Trees

A **ternary tree** is a tree where each parent node has at most three children. An **n-ary tree** is a tree where each parent node can have up to n children.

Definition 3.6: 2-3 Tree – Overview

A **2-3 tree** is a **complete** search tree where a parent may contain 1 or 2 keys:

- **1 Key (1K):** This parent has 0 or 2 children.
- **2 Keys (2K):** This parent has 0 or 3 children.

The 1K node abides by the same rules as a BST:

$$\text{left child} < \text{parent node} \leq \text{right child}$$

For 2K let us define the left key as k_1 and the right key as k_2, then the rules are :

$$\text{left child} < k_1 \leq \text{middle child} < k_2 \leq \text{right child}$$

Since it is a complete tree, height of n nodes is $O(\log n)$.

Time Complexity: Searching, inserting, and deleting in a 2-3 tree is $O(\log n)$, where n is the number of keys in the tree. The performance gain comes from 3 children possibility: if the tree has only 3 children, it would be $O(\log_3 n)$, which is faster than $O(\log_2 n)$; However, the tree has have a mix of 1K and 2K nodes. So that's why in the worst case, we have $O(\log_2 n)$. I.e., $\lfloor \log_3 n \rfloor \leq h \leq \lfloor \log_2 n \rfloor$, where h is the height of the tree.

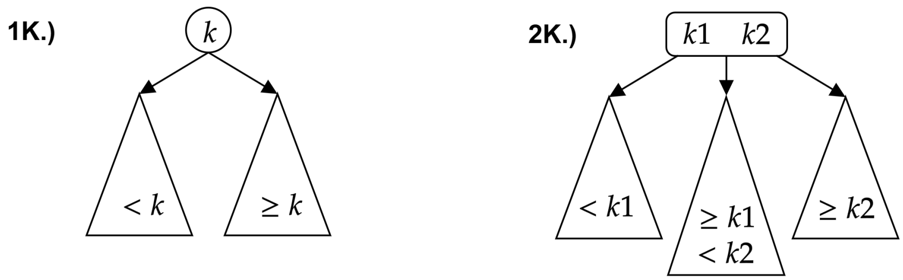


Figure 6.25: Definition (3.4), k_1 and k_2 are parent keys with its left, middle, and right nodes.

Definition 3.7: 2-3 Tree – Search & Insert

Searching is similar to a BST (3.2), we compare the desired value v to the keys (If $v < k_1$, we search in the left child. If $k_1 \leq v < k_2$, we search in the middle child. If $v \geq k_2$, we search in the right child. If the child is null, it does not exist.)

Inserting a value v into a 2-3 tree involves the following steps after a search:

- **Null:** If no nodes exists, create a new $1k$ node of v.
- **1K Node:** Insert v in sorted order, becoming a 2K node (no children).
- **2K Node:** Insert v in sorted order (3 keys), then split:
  - **No Parent:** Push the middle key up taking the remaining left (LK) and right (RK) keys and create two new children satisfying 1K node rules (3.6).
  - **Parent is 1K:** The parent becomes a 2K node. The remaining LK and RK keys are again placed respectively satisfying the 2K node rules (3.6).
  - **Parent is 2K:** The parent momentarily becomes a 3K node with 4 children. These children are split are split between k_1 and k_2 respectively after the middle key is pushed up.

The next page provides a more complex example demonstrating the "Parent is 2K" case.

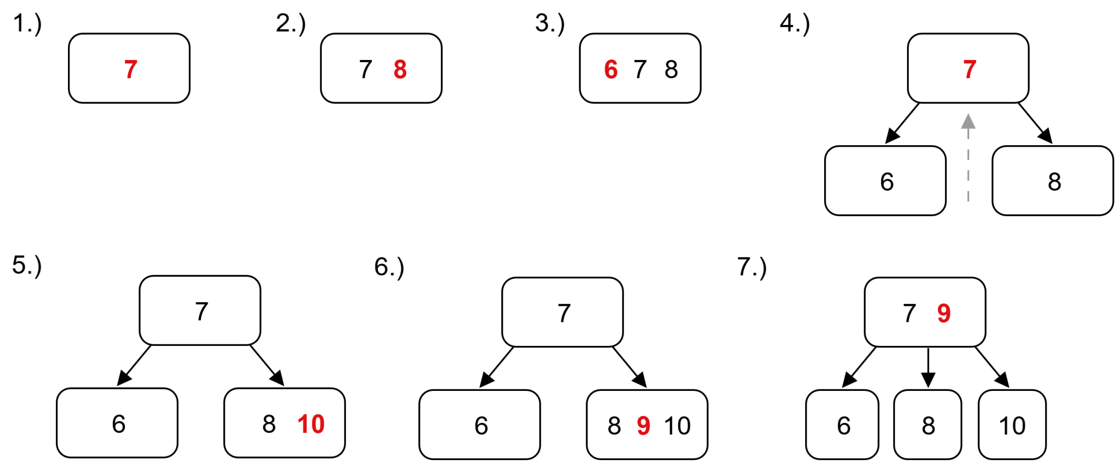


Figure 6.26: Visual Representation of Definition (3.7). In (3), there are three keys, initiating a split, pushing up 7. In particular, 1 & 4 are 1K nodes, and 2 & 7 are 2k nodes.

Definition 3.8: 2-3 Tree – Deletion

To delete a value v after search in a 2-3 tree, we follow these steps:

1. **Leaf Node:** If v is in a leaf node, remove it as follows:
   - **2K Parent:**
     a) **2K Child-Node:** If v sits in a child 2K **leaf**-node, remove v.
     b) **1K Child:** Say v is removed from a 1K child. Then one parent key will merge with another child into a valid 1K configuration.
   - **1K Parent:**
     a) **2K Child-Node:** If v sits in a child 2K **leaf**-node, remove v.
     b) **2K Sibling:** If v is removed from a 1K child and its sibling is a 2K node, rotate the parent key and closest-order sibling key to balance the tree back into a stable 1K configuration.
     c) **1K Children:** If both children are 1K nodes, v is removed. The parent key is copy/merged with the remaining child (2K w/ no children), and the original parent (OP) is emptied. We now need to maintain the complete tree property
        - **Parent Cases:** Balancing the tree requires taking from a parent key. If OP's parent is the root, we promote OP as the new root, deleting the old.
        i. **Adgecent 2K Node:** If the OP has a 2K direct sibling, adopt OP's parent key, then rotate the closest-order key from the sibling to the parent. OP

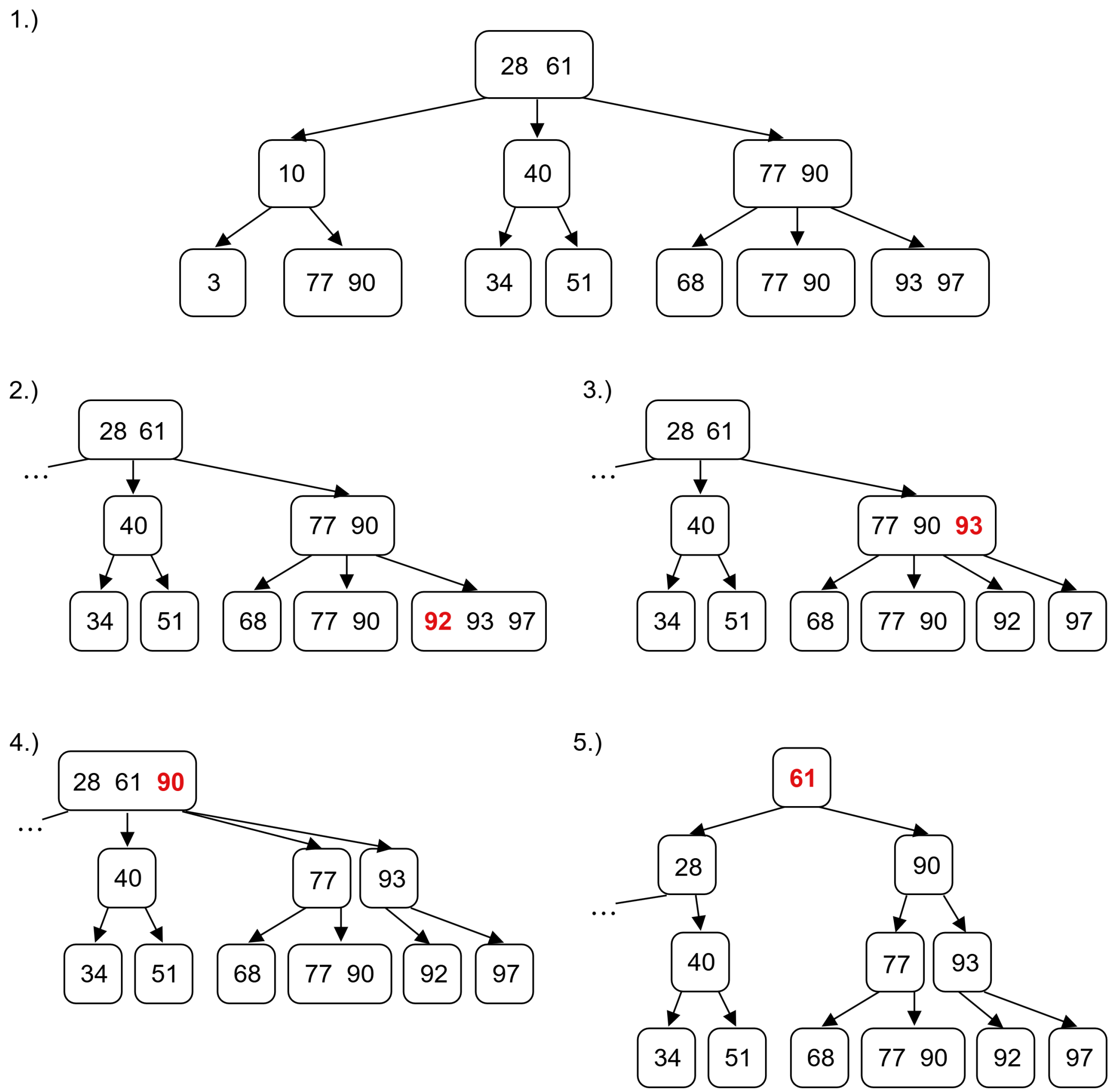


Figure 6.27: Visual Representation of Definition (3.7) of the "Parent is 2K" case: In (2) we start out inserting 92, overflowing the current node, (3) 93 is pushed up, splitting the previous node $\{92, 97\}$ into total 4 children, (4) 90 is pushed up, splitting node $\{77, 93\}$, each taking their respective two children from the previous children-overflow, (5) and finally, 61 is pushed up in the same manner.

now adopts the closest-order child from the sibling, balancing the tree from (1C:3C)→(2C:2C) #Children.

ii. **No 2K Sibling:** If OP has no 2K sibling, adopt its closest-order parent key, then it merge with a sibling 1K node (becomes a 2K node). This should balance the tree from (1C:2C)→(3C) #Children. If the parent becomes illegal, repeat the above steps.

2. **Intermediate/Root Node:** If v is not a leaf node, swap v with either its in-order predecessor (largest left child) or successor (smallest right child) repeatedly until it's a leaf node (If $k2$, take largest middle child). Then follow the above steps to delete v.

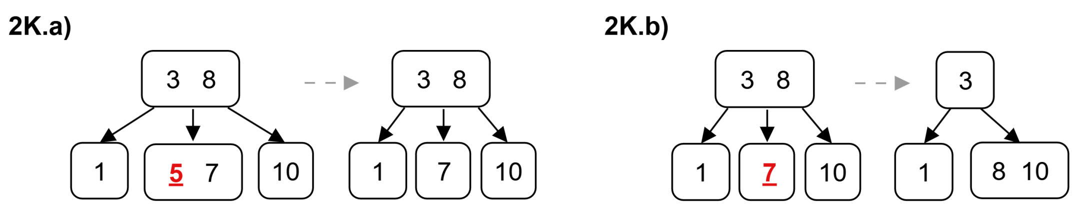


Figure 6.28: Without Loss of Generality, Definition (3.8)'s bullets – 2K Parent (a & b).

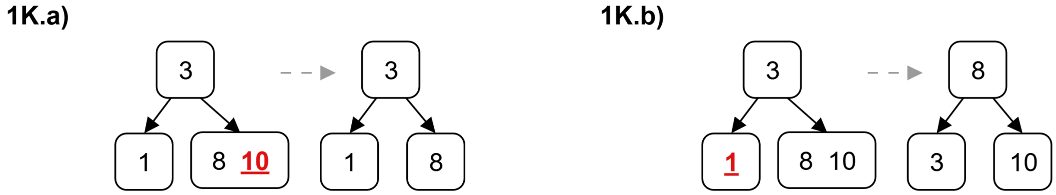


Figure 6.29: Definition (3.8)'s bullets – 1K Parent (a & b).

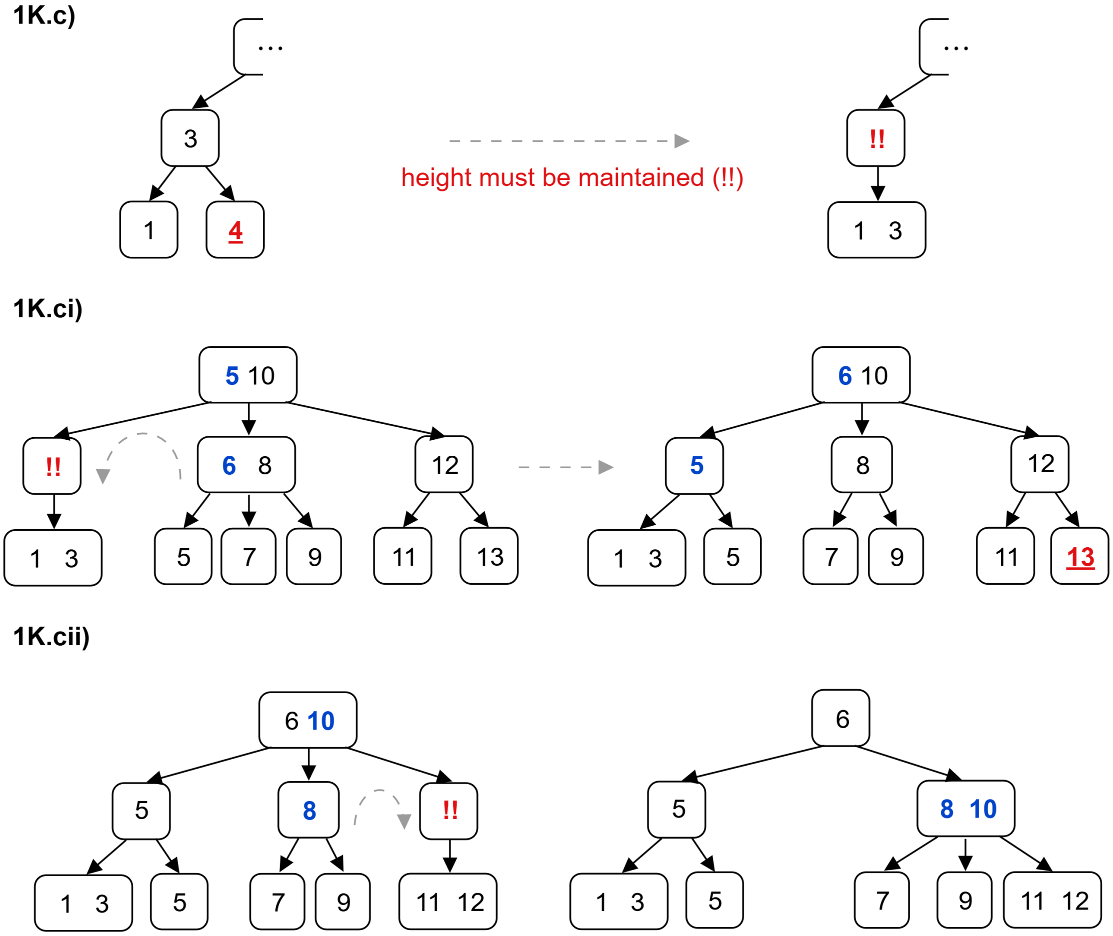


Figure 6.30: Definition (3.8)'s bullets – 1K Parent (c). (1K.c) 4 is deleted, causing 1 and 3 to merge; However the height must be maintained for a complete tree. (1K.ci) parent sibling keys rotate, beginning a deletion on 13. (1k.cii) parent key and sibling merge.

Lastly, the intermediate/root deletion:

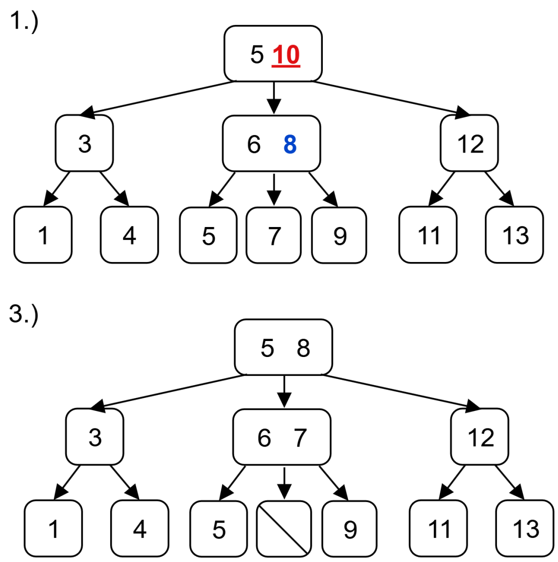


Figure 6.31: Definition (3.8)'s bullets – Intermediate/Root Node deletion. (1) We desire to delete 10, but it's not a leaf node, hence we swap with its in-order predecessor, 8. (2) 10 Still isn't a leaf node, we swap with 7. (3) 10 is finally a leaf node, we delete it. (4) The tree is rebalanced.

Other cases (without loss of generality) and implementation is left as an exercise to the reader.

6.4 Directed-Acyclic Graphs & Topological Ordering

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

Definition 4.1: Directed-Acyclic Graph (DAG)

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

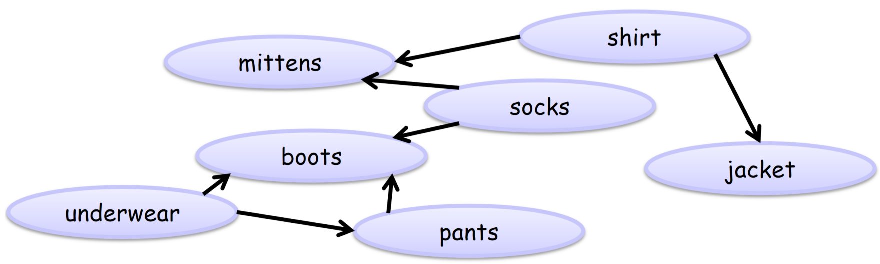


Figure 6.32: A DAG depicted by getting dressed for winter. For each node to be processed its **dependencies** or parents must be processed first.

Definition 4.2: 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.

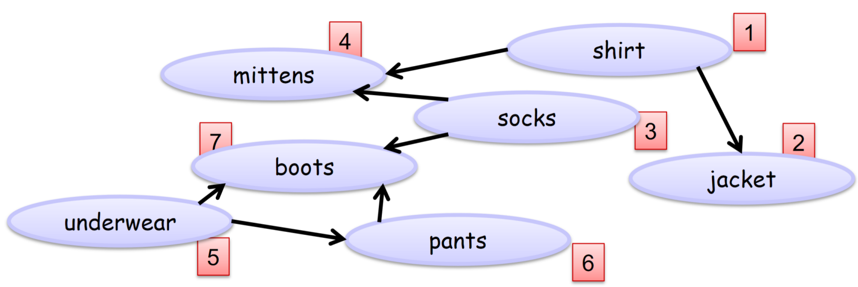


Figure 6.33: A topological ordering of the DAG in Figure (6.32) enumerated in red. Another possible ordering of $[1, 2, 3, 4, 5, 6, 7]$ is $[5, 6, 1, 2, 3, 4, 7]$, as $5 \to 6$ is independent.

Theorem 4.1: Topological Sort

Given a DAG, a topological ordering can be found.

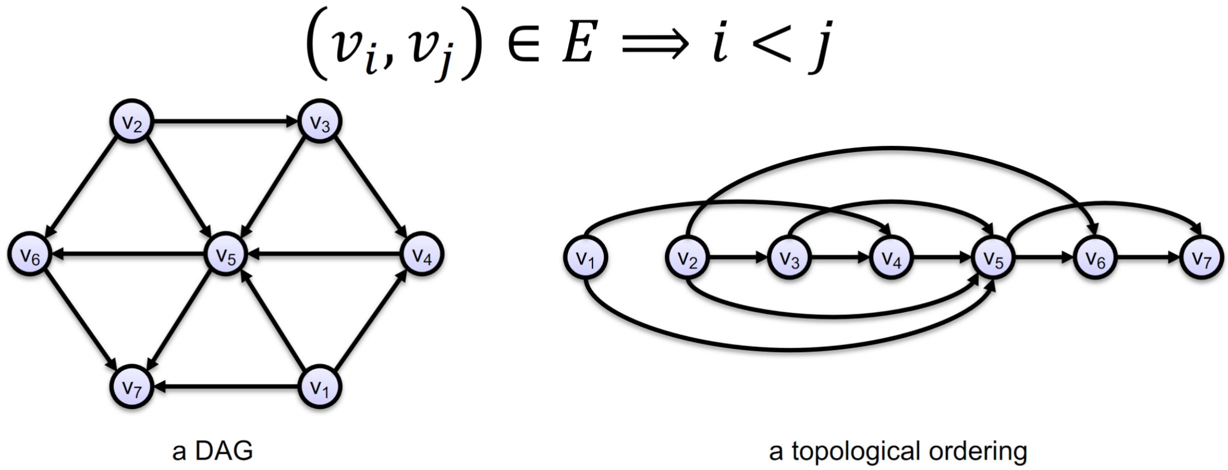


Figure 6.34: A topological sorting of a DAG E and v nodes

Proof 4.1: 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 4.3: 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**.

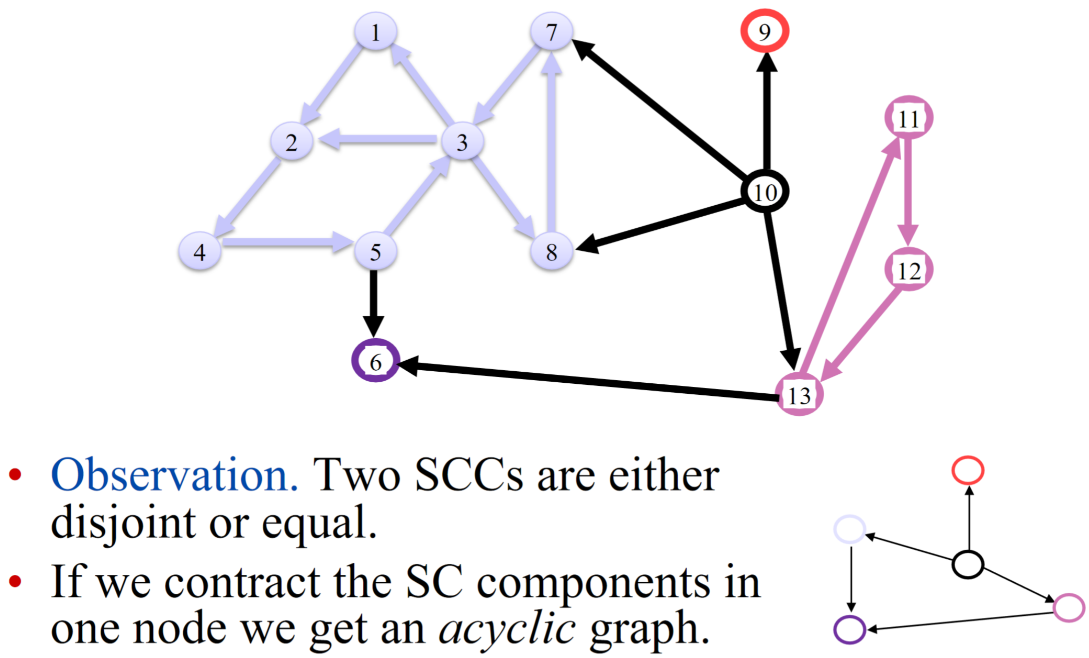
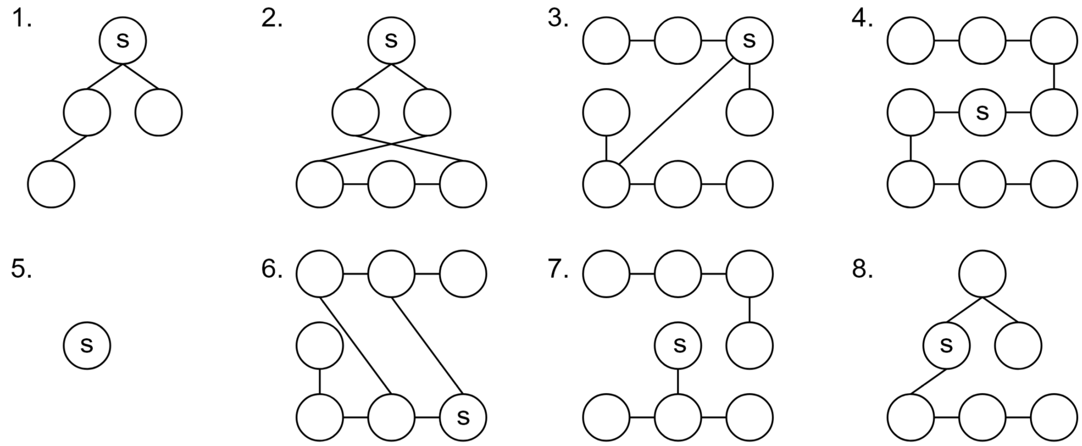


Figure 6.35: A graph with 5 strongly connected components.

Exercise 4.1: Based on the images which are trees, and if not, why?



Exercise 4.2: Based on Exercise (4.1) above, starting with the s nodes, what are possible results of BFS and DFS?

Exercise 4.3: What type of edges do BFS and DFS traversals produce on directed and undirected graphs?

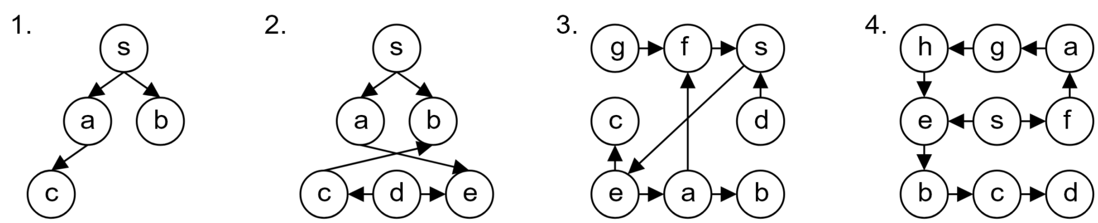
Exercise 4.4: BFS can find shortest paths, does DFS find longest paths (True/False)?

Exercise 4.5: To find a path and return such path from node s to node t in a graph, should we use BFS or DFS?

Exercise 4.6: To detect cycles in a graph, should we use BFS or DFS, or both?

Exercise 4.7: Say we wanted to find the shortest round-trip path, that is, the minimal path starting from origin node s visiting a set of nodes exactly once returning to s. How should we use BFS or DFS?

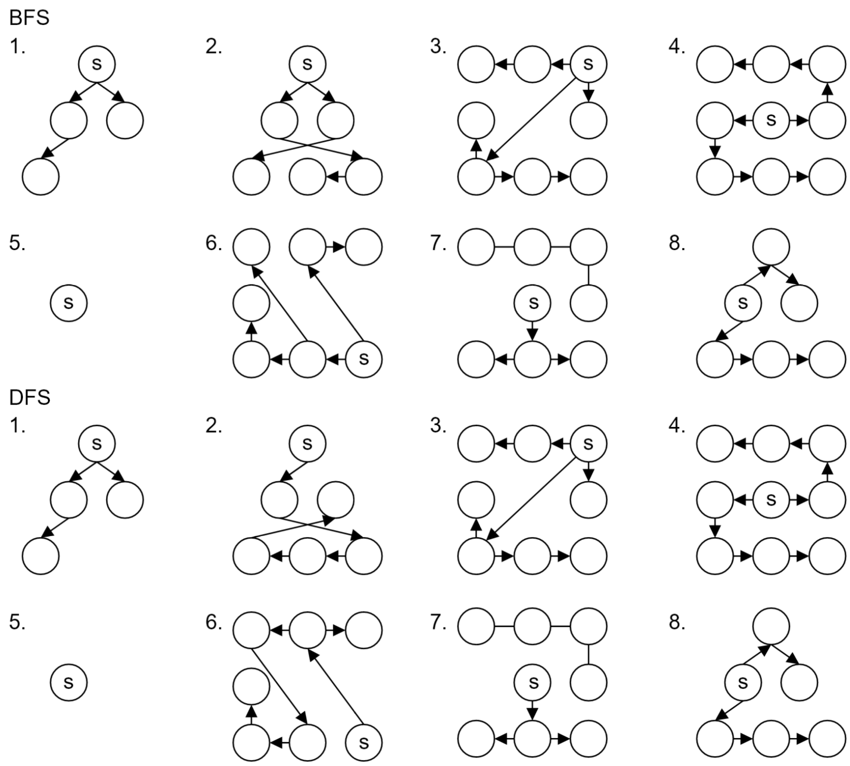
Exercise 4.8: Which of the below images are DAGs.



Exercise 4.9: Based on Exercise (4.8) above, what are possible topological orderings?

Answer 4.1: 1, 3, 4, 5, 8 are trees. 2 has a cycle, 6 has a cycle, 7 is disconnected.

Answer 4.2: There are other possible solutions depending on order of traversal. For 7, BFS and DFS don't alone handle disconnected graphs. Though modifications could be made to handle such.



Answer 4.3: Undirected - BFS: Tree edges, DFS: Tree edges, Back edges. Directed - BFS: Tree edges, Back edges, Cross edges, DFS: Tree edges, Back edges, Forward edges, Cross edges [1, 7, 3]

Answer 4.4: False, as of today, there is no known algorithm to find the longest path in a graph.

Answer 4.5: BFS, it finds shortest child-parent connections, giving us a direct path from s to t.

Answer 4.6: Both, as DFS when it encounters a node that has already been visited, it has found a cycle. BFS when two branches meet, a cycle is found.

Answer 4.7: BFS, one implementation is to run BFS from s to all nodes, the first cycle found is the shortest round-trip path.

Answer 4.8: 1, 2, and 4 are DAGs. 3 has a cycle (f,s,e,a).

Answer 4.9:

1. $[s, a, b, c]$, $[s, b, a, c]$, $[s, a, c, b]$.
2. $[s, a, d, c, b, e]$, $[d, c, s, a, e, b]$ or $[d, c, s, a, b, e]$
4. $[s, f, a, g, h, e, b, c, d]$

— 7 —

Scheduling

7.1 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.

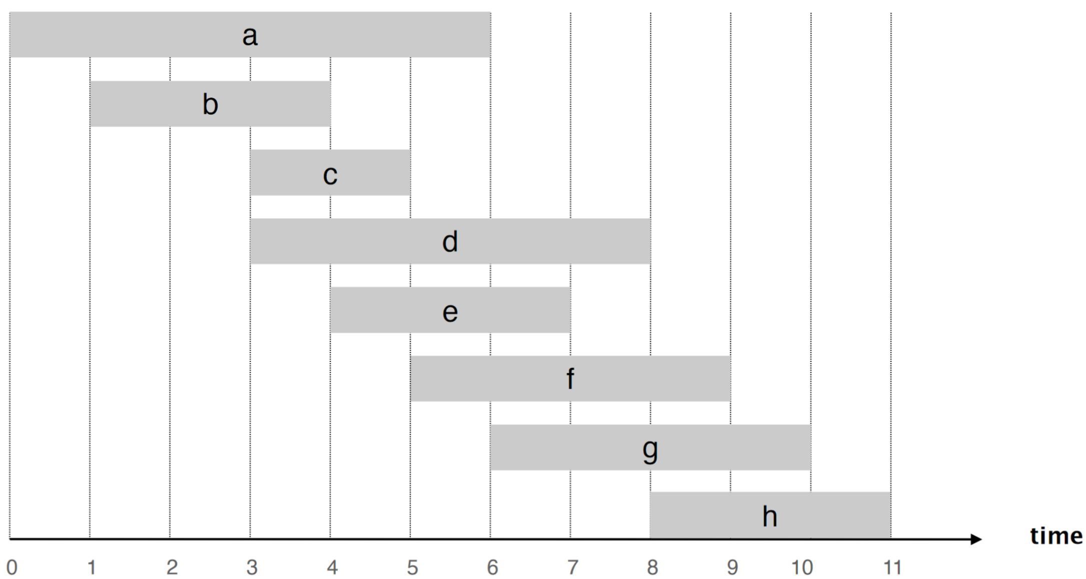


Figure 7.1: Given jobs a through h we find the largest subset of mutually compatible jobs $\{b, e, h\}$.

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. ■

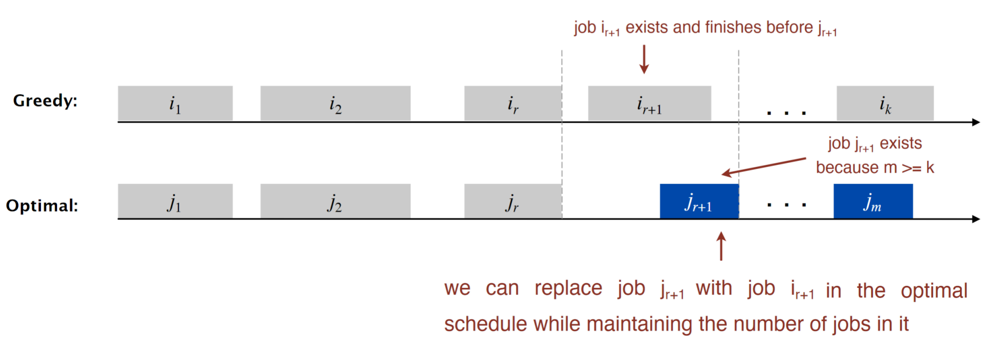


Figure 7.2: Shows that at the first divergence, i_{r+1} and .

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.

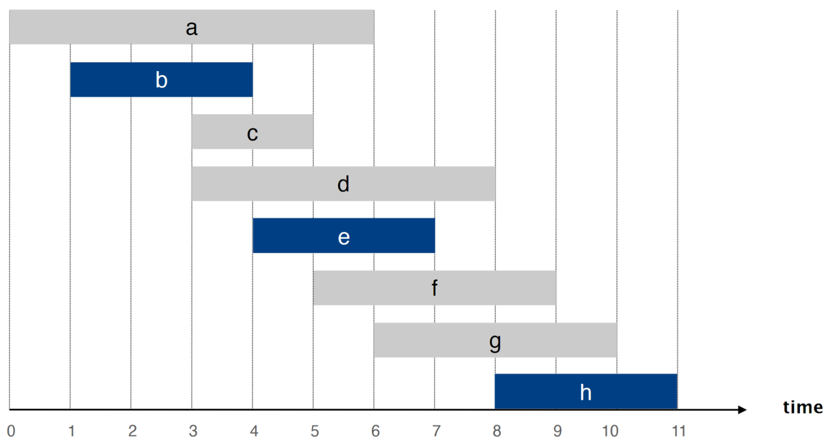


Figure 7.3: Solution to Figure (7.1) using early finish time first, yielding $\{b, e, h\}$.

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.

sorted_jobs $\leftarrow$ sort$(f_1, \ldots, f_n)$ `// sort by finish time` $S \leftarrow \emptyset$ `// selected jobs`
  $f_{\text{last}} \leftarrow -\infty$;
for *each j in sorted_jobs* **do**
  if $f_{last} \leq s_j$ **then**
    $S \leftarrow S \cup \{j\}$;
    $f_{\text{last}} \leftarrow f_j$;
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.

7.2 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.)

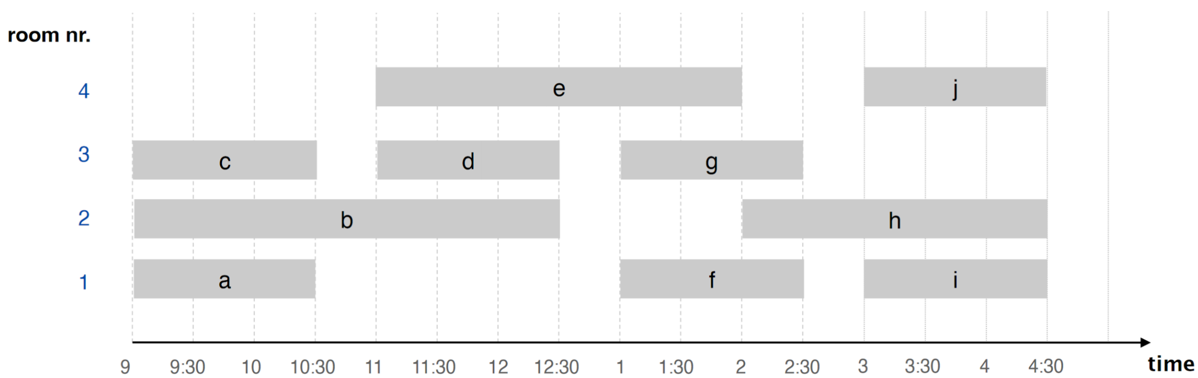


Figure 7.4: Though not optimal, here is a possible schedule where $k = 4$.

We strategies and figure out the minimum number of classrooms needed to schedule all classes in the worst-case. We observe in Example (7.2) 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.

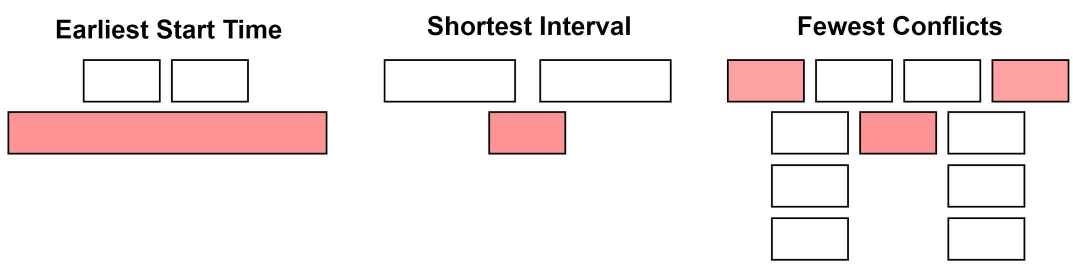


Figure 7.5: Counter examples of Earliest Start Time, Shortest Interval, and Fewest Conflicts. Earliest time fails to see the later two smaller jobs, shortest interval fails to see two longer jobs, fewest conflicts fails to see the sequential solution of 4 jobs because it has too many conflicts.

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 $s_j + \epsilon$.

■

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

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.

$\mathcal{A} \leftarrow$ empty hash table `// ` $\mathcal{A}[k]$ `contains the list of lectures assigned to room` k sorted_class $\leftarrow$ sort$(s_1, \ldots, s_n)$ `// sort lectures by start time`
for *each c in sorted_class* **do**
  $k \leftarrow$ find_compatible_room(c, $\mathcal{A}$, Q);
  if k *is not None* **then**
    $\mathcal{A}[k]$.add(c);
  else
    $d \leftarrow$ len($\mathcal{A}$) `// highest room id` $\mathcal{A}[d+1] \leftarrow [\,]$ `// open new room`
    $\mathcal{A}[d+1]$.add(c);
return $\mathcal{A}$

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.

7.3 Priority Queues: Min & Max Heaps

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:

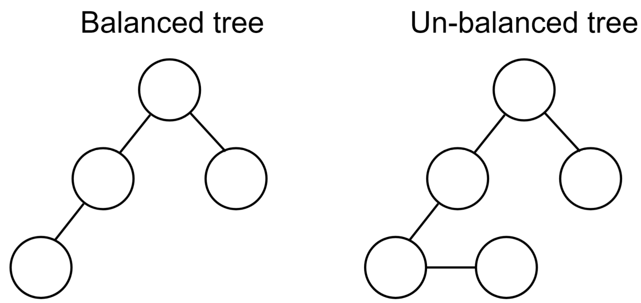


Figure 7.6: Examples of a balanced tree and an unbalanced tree

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 Tree

A **heap** (not to be confused with a memory 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. $log(n)$;

We may also represent a heap as an array:

Definition 3.4: Heap as Array

heap can be represented as an array A where:

- The root is at index 0.
- The left child of node i is at index $2i + 1$.
- The right child of node i is at index $2i + 2$.

This representation allows us to efficiently access parent and child nodes, instead of using whole node objects.

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
⟨f_k, k⟩ ← PEEK_MIN(Q)          // shows lowest ⟨key, value⟩ pair, O(1)
if s_c > f_k                     // finish time in room k then
    return k                     // c is compatible with room k
else
    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
$\mathcal{A} \leftarrow$ empty hash table // $\mathcal{A}[k]$ contains the list of courses assigned to room k
$Q \leftarrow$ empty priority queue // contains $\langle$finishTime, roomId$\rangle$ pairs
sorted_class $\leftarrow$ sort($s_1, \dots, s_n$) // sort by start time
for *each c in sorted_class* **do**
  $k \leftarrow$ find_compatible_room(c, $\mathcal{A}$, Q);
  if *k is not None* **then**
    $\mathcal{A}[k]$.add(c);
    Q.UPDATE-KEY($\langle f_k, k\rangle, \langle f_c, k\rangle$) // update finish time of room k
  else
    $d \leftarrow$ len($\mathcal{A}$) // highest room id
    $\mathcal{A}[d+1] \leftarrow [\,]$ // open new room
    $\mathcal{A}[d+1]$.add(c);
    Q.INSERT($\langle f_c, d+1\rangle$);
return $\mathcal{A}$

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 $O(n)$.

7.4 Minimizing Lateness

Scenario: ***Swamped in Assignments***
Say we have j assignments, each j assignment takes p_j time to prepare, and a d_j deadline. We want to finish all assignments, but minimize the overall lateness.

$j =$	1	2	3	4	5
p_j	3	2	1	4	2
d_j	4	7	8	8	10

Table 7.1: Table showing d_j and p_j deadlines and prepare times for j assignments.

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

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

Counter Examples: Consider the following:

Shortest processing time p_j first:

	p_j	d_j
1	1	100
2	10	10

Table 7.2: If shortest p is picked, 1 will be placed first, despite its deadline being 100. Then 2 will be placed, which will be late by 1, since $p_1 + p_2 = 11$, missing its 10 deadline.

Smallest slack $d_j - p_j$ first:

	p_j	d_j
1	1	2
2	4	4

Table 7.3: 4−4 would have the smallest slack, so it's placed first, but this means 1 will be late by 3, since $4 + 1 = 5$ and $d_1 = 2$. Vice-versa, if 1 is first, then 2 is only late by 1.

Theorem 4.1: Minimizing Lateness

Given a set of jobs j with prepare and deadline times p_j and d_j under one resource, a like-optimal solution is obtained by scheduling deadlines in ascending order.

Tip: Video version of this section: https://youtu.be/wdqkJ7VICkc?t=973

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. ■

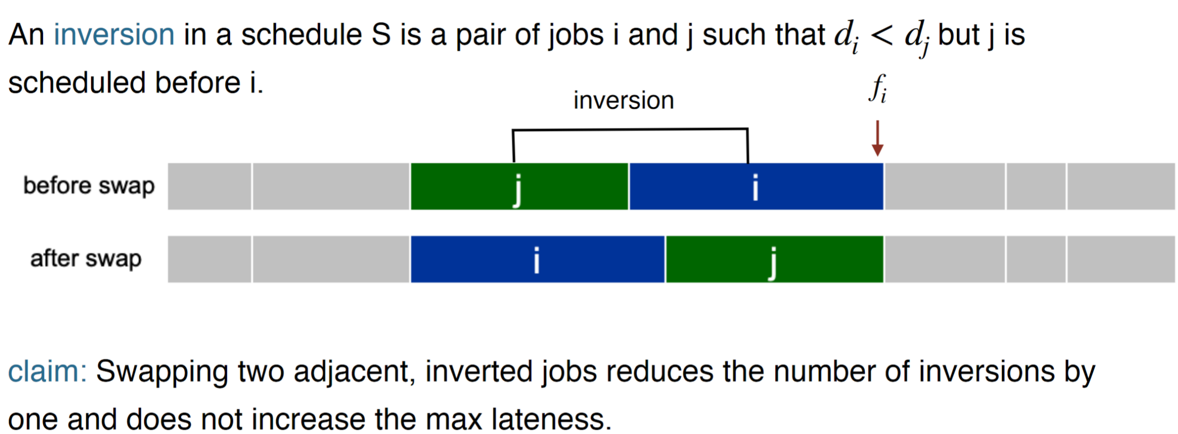


Figure 7.7: Shows that at the first inversion, i and j are swapped.

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
sorted ← sort(d_1, d_2, ..., d_n)   // sort by increasing deadline intervals ← empty list;
t ← 0                                                  // keep track of time
for each j in sorted do
    // assign job j to interval [t, t + t_j]
    intervals.add([t, t + t_j]);
    t ← t + t_j;
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 (7.1), we get the following optimal schedule:

$j =$	1	2	3	4	5
p_j	3	2	1	4	2
d_j	4	7	8	8	10

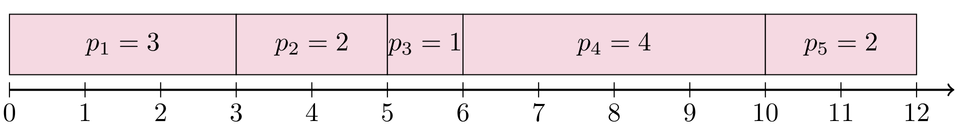


Figure 7.8: Sorting by earliest deadline first, we get the optimal schedule. Here, p_4 and p_5 only have a 2, totalling to 4 lateness.

— 8 —

Greedy Algorithms

8.1 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:

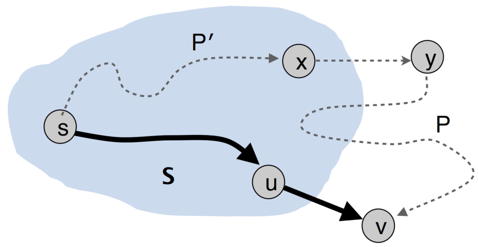


Figure 8.1: A system of subset paths (blue), and an exact point of exit $u \to v$ and $x \to y$

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:

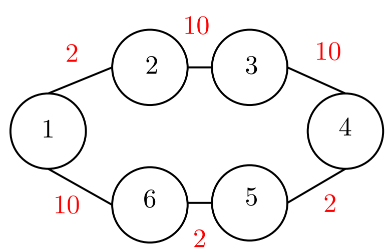


Figure 8.2: A weighted graph.

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	

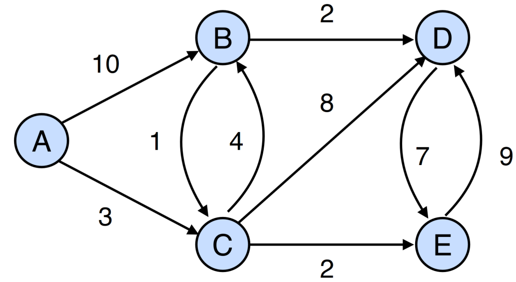


Figure 8.3: A weighted graph and its shortest paths.

In figure (8.3), 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:

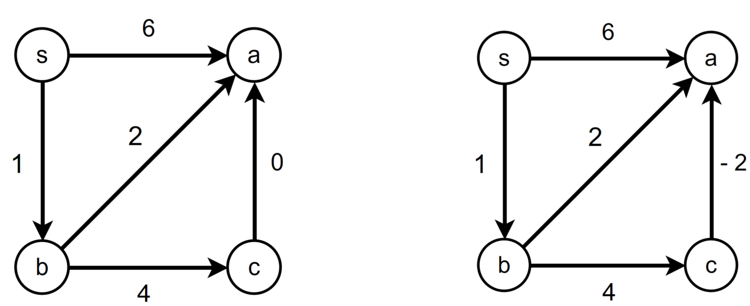


Figure 8.4: Shows two weighted graphs, one positive, and the other negative.

In Figure (8.4), 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.

```
Function Dijkstra(G, s):
    π ← {}// hash table, current best list for v
    d ← {}// hash table, distance of v
    parents ← {}// parents in shortest path tree
    Q ← PQ()// priority queue to track minimum π
    π[s] ← 0;
    Q.INSERT(⟨0, s⟩);
    for v ≠ s in G do
        π[v] ← ∞;
        Q.INSERT(⟨π[v], v⟩);
    while Q is not empty do
        ⟨π[u], u⟩ ← EXTRACT-MIN(Q);
        d[u] ← π[u];
        for v ∈ G[u] do
            if π[v] > d[u] + l(u, v) then
                DECREASE-KEY(⟨π[v], v⟩, ⟨d[u] + l(u, v), v⟩);
                π[v] ← d[u] + l(u, v);
                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.

Tip: Video Demo of Dijkstra's Algorithm:
https://youtu.be/gaXTSp2BBdY?si=Egc7bAv4SRPgtffa

8.2 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.

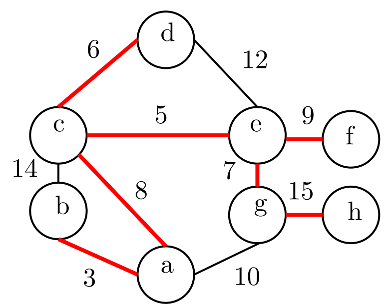


Figure 8.5: Example of an graph with MST highlighted in red.

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 $u \to v$, 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**.

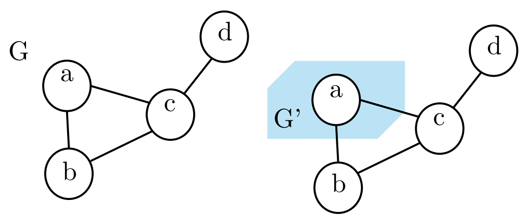


Figure 8.6: Illustration of a graph G and a cut G' of the graph.

We see in Figure (8.6) 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 (8.6), 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 (8.7), 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).

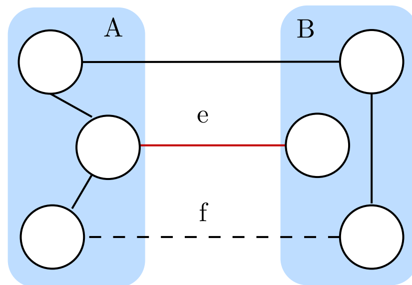


Figure 8.7: A graph cut into two disjoint sets, with a highlighted edge e and a dashed edge f.

Theorem 2.4: Prim's Algorithm

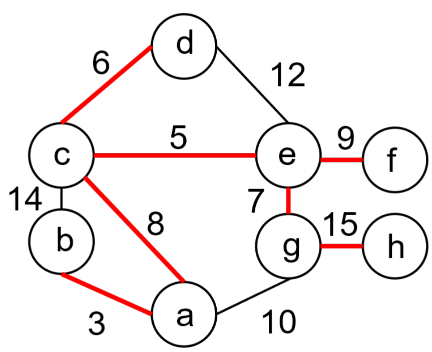
Given a connected graph G with n nodes and m edges, we produce the MST via:

(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 8.8: 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 $V.forEach((v) =$¿ $\{Q[v] \leftarrow \infty\})$; `// for each` $v \in V$ `set it's weight to` ∞
3 $Q[V_0] \leftarrow 0$; `// 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** $Q \neq \emptyset$ **do**
6     $u \leftarrow Q.Extract()$;
7   **foreach** $v \in G[u]$ **do**
8   **if** $(v \in Q)$ *and* $(w(u, v) < Q[v])$ **then**
            `// Edit the node's weight in` Q `then re-balance.`
9             $Q[v] \leftarrow w(u, v)$;
10             $Q.DecreaseKey(v)$;
11             $T[v] \leftarrow u$;
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, $O((n+m)\log n)$ or $O(m \log n)$.
Space Complexity: $O(n+m)$, as we at most store the data items a hash-table representation of our graph.

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

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

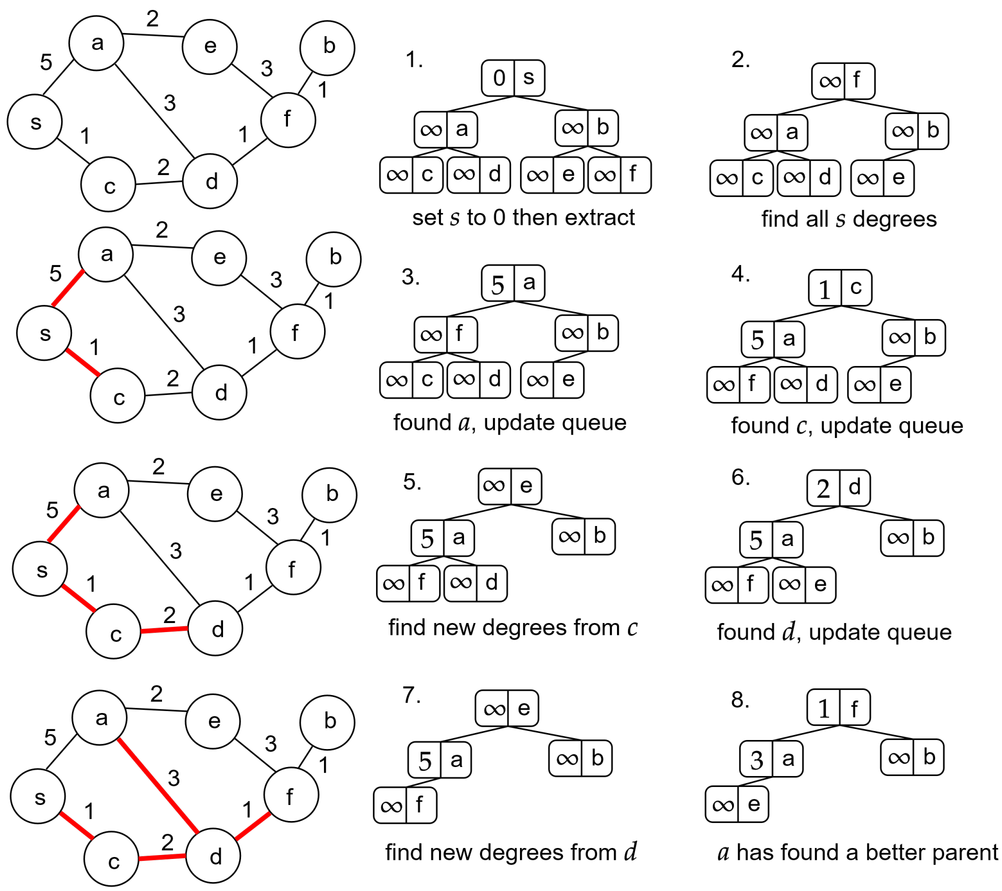


Figure 8.9: A diagram illustrating the pattern of Prim's Algorithm through each iteration.

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

Proof 2.1: Prim's Hash-tables vs. Min-heaps Implementation

Say we had a hash-table of priorities where $R[v]$ returns weight $w(u, v)$:

- **Hash-table - Extract:** $O(n)$, search all indices, **Decrease-Key:** $O(1)$, update $R[v]$.
- **Min-heap - Extract:** $O(1)$, top element, **Decrease-Key:** $O(\log n)$, re-balance heap.

```
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();                         O(n + m)
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
```

line 5 stores all n nodes to be visited $O(n)$, line 7 iterates all neighbors for each n. Let all edges be m, then $\sum_{v \in n} \text{degree}(v) = m$. Since line 6 is $O(1)$, we have $O(n+m)$ for iterating all edges. Now for every edge, we might evaluate Decrease-Key, which is $O(\log n)$. This gives us $O((n+m)\log n)$, or $O(m \log n)$.

When using a hash-table, line 6 becomes $O(n)$, and Decrease-Key is $O(1)$. Thus, $O((n(n)+m)\cdot 1) = O(n^2+m)$ or $O(n^2)$. Hence,

Hash-table: $O(n^2)$. **Min-heap:** $O(m \log n)$.

For any connected graph $n - 1 \leq m \leq \frac{n(n-1)}{2}$, where m is the number of edges. Most often m is much less than n, making the min-heap the better choice. E.g., say $m = \frac{n^2}{2}$, then $O(n^2)$ vs. $O(n^2 \log n)$, hash-table wins. However, say $m = \frac{n}{2}$, then $O(n^2)$ vs. $O(n \log n)$, min-heap wins. ■

Theorem 2.5: Prim's Hash-tables vs. Min-heaps

For Prim's algorithm,
when $m \leq n$, min-heap is better ($O(n \log n) < O(n^2)$). When m is significantly larger than n, hash-tables are better ($O(n^2) < O(n^2 \log n)$).

8.2.1 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.

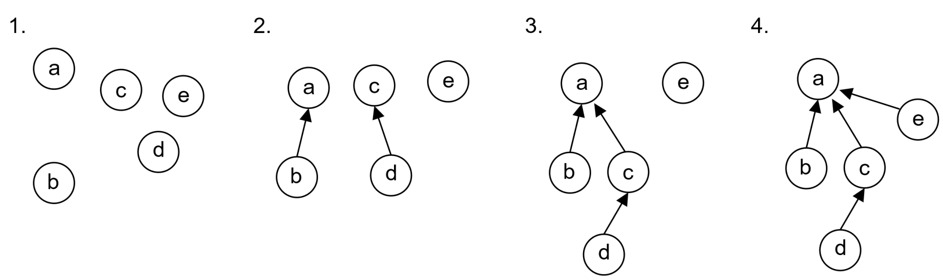


Figure 8.10: Showing disjoint nodes Union-Join with each other.

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.

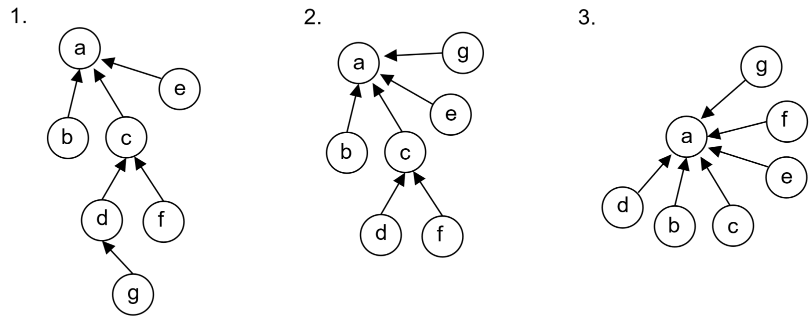


Figure 8.11: Showing a forest compress over multiple finds.

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	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta\big(m \cdot \alpha(m,n)\big)$

Table 8.1: Time-complexity comparison of different implementations. Where $\alpha(m,n)$ is the inverse Ackermann function, which grows very slowly.

Theorem 2.6: 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.

Exercise 2.1: Let there be a graph $G(V, E)$ with an MST T and a shortest paths tree S. Now suppose a weight of 1 is added to each edge in G.

1. Will the MST T change?
2. Will the shortest paths tree S change?

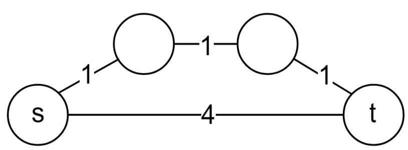
Exercise 2.2: Let there be a graph $G(V, E)$ with an MST T and a shortest paths tree S. Suppose a new edge e is added to G.

1. The weight of e is the heaviest in G.
   a) Will the MST T change?
   b) Will the shortest paths tree S change?
2. The weight of e is the lightest in G.
   a) Will the MST T change?
   b) Will the shortest paths tree S change?

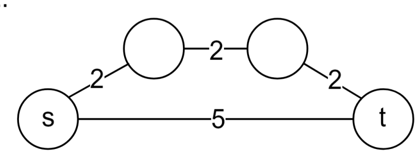
Answer 2.1:

1. **No.** All edges in T are still the lightest edges in G.
2. **Yes.** Shortest paths is not MST. Here is a counter example:

1.



2.

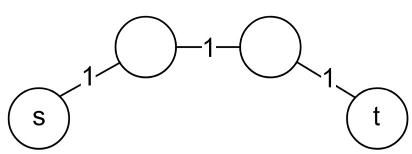


The shortest path from s to t changed when all edges were increased by 1.

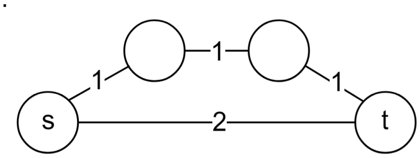
Answer 2.2:

1. a) **No.** By the cycle property, the heaviest edge is not necessary in the MST.
   b) **Yes.** The new edge may present a shorter path. Here is a counter example:

1.



2.



2. a) **Yes.** By the cycle property, an existing edge will be replaced by a lighter edge.
   b) **Yes.** Consider a long expensive path, a new edge may present a shorter path.

— 9 —

Evaluating Recursive Algorithms

9.1 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://youtu.be/ZLBmc0qf174?si=2b0Kxy9xxPHos_6A and **QuickSort**: https://youtu.be/FWWqLJIUFZM?si=ZPhuvIduWAGXH_mv.

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)*:

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

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

```
i ← lefti; // Left subarray
j ← righti; // Right subarray
k ← lefti; // Temporary array
while i ≤ leftEnd and j ≤ rightEnd do
    if A[i] < A[j] then
        temp[k] ← A[i]; i ← i + 1;
    else
        temp[k] ← A[j]; j ← j + 1;
    k ← k + 1;
while i ≤ leftEnd do
    temp[k] ← A[i]; i ← i + 1;
    k ← k + 1;
while j ≤ rightEnd do
    temp[k] ← A[j]; j ← j + 1;
    k ← k + 1;
for i = lefti to rightEnd do
    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.

Below is a graphical representation of merge sort:

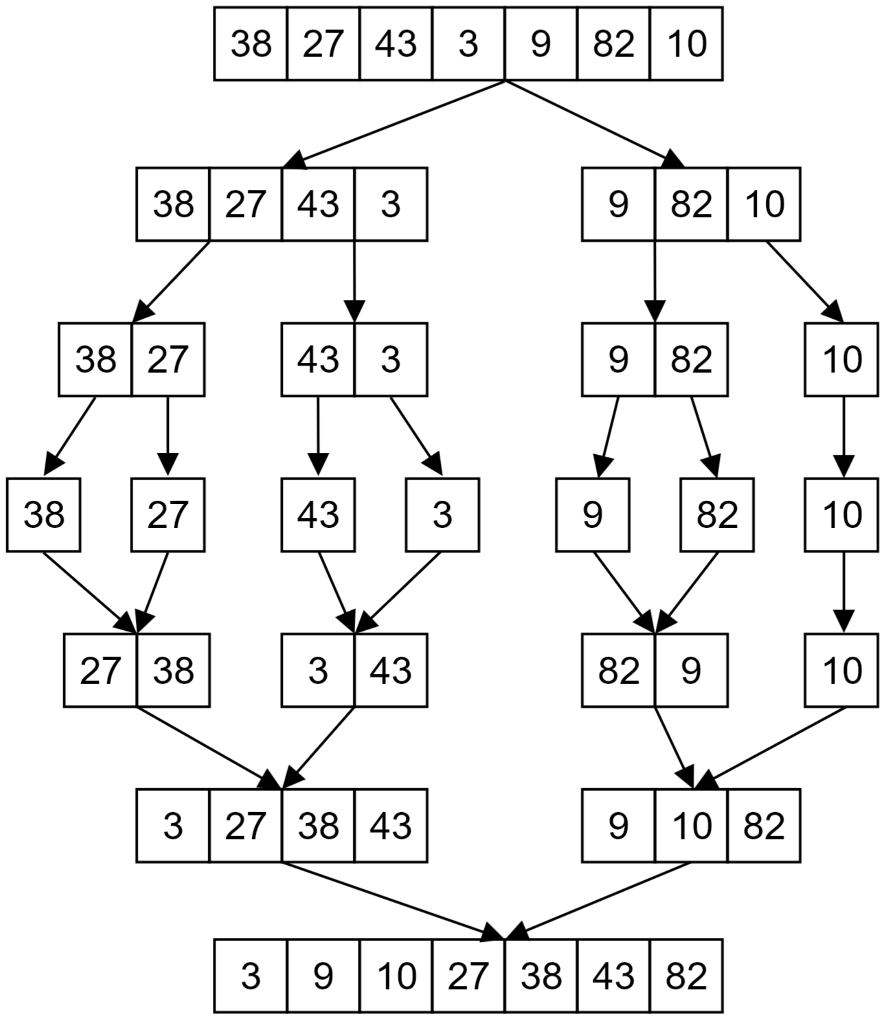


Figure 9.1: Here is an array of 7 elements. The first partition happens on index 3 as $\lfloor (0+6)/2 \rfloor = 3$. This continues all the way down until we hit our base case; During the merge step, we take a temporary array and fill in the missing elements from the two joining halves. Here the next smallest element is picked from each half, if one finishes first, we copy the rest of the other half into the temporary array. Finally, we copy back the temporary array to the original array.

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)*

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

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

```
pivot ← A[⌊(low + high)/2⌋];
i ← low − 1;
j ← high + 1;
while i < j do
    repeat
        if A[i] ≥ pivot then
            break;
    until i ← i + 1;
    repeat
        if A[j] ≤ pivot then
            break;
    until j ← j − 1;
    if i < j then
        A.swap(i, j);
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:

```
if i ≥ j then
    return;
mid ← (i + j)/2;
MSORT(A, temp, i, mid); // Left subarray
MSORT(A, temp, mid + 1, j); // Right subarray
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 $T(n)$:

$$T(n) = \underset{\substack{\text{number of} \\ \text{rucursive calls}}}{a} \cdot \underbrace{T\left(\frac{n}{b}\right)}_{\substack{\text{sub-divisions} \\ \text{each frame}}} + \underbrace{f(n)}_{\substack{\text{work each} \\ \text{stack frame}}}$$

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:

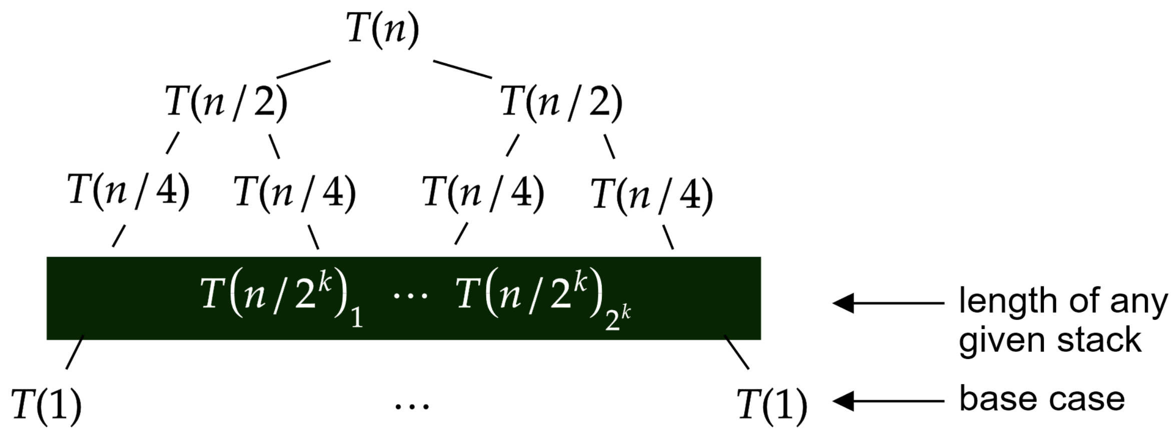


Figure 9.2: Recursion Tree for Merge Sort

Function 1.4: Quicksort - `QSORT(A)`

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

QSORT function:

if $low < high$ **then**
  $pivotIndex \leftarrow$ `partition`$(A, low, high)$;
  $QSORT(A, low, pivotIndex)$; `// Left of the pivot`
  $QSORT(A, pivotIndex + 1, high)$; `// 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:

if $i \geq j$ **then**
  return;
$mid \leftarrow (i+j)/2$;
$MSORT(A, temp, i, mid)$; `// Left subarray`
$MSORT(A, temp, mid+1, j)$; `// Right subarray`
$merge(A, temp, i, mid, mid+1, j)$; `// Merge both halves`

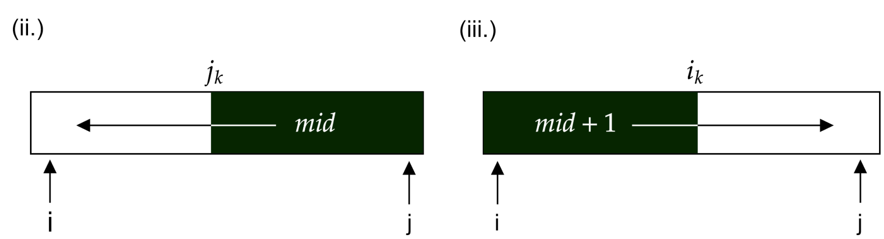
Proof 1.3: Correctness of Merge Sort

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 $T(n)$:

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

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

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

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

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

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: Evaluating Recursion – Master Method

9.1.1 Efficiency of Multiplication: Karatsuba's Algorithm

Recall Section (3.4), which discusses manipulating base 2. There is a problem when trying to multiply two n-bit integers, as we have to compute n^2 products, which is inefficient. We can attempt to reduce the number of products by using a divide-and-conquer approach.

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

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, $(A_1 B_1)$, $(A_1 B_0), (B_1 A_0)$, and $(A_0 B_0)$. We now attempt to solve them independently:

Function 1.6: 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.

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

Function *Multiply*(n, a, b)**:**
  if $n < 2$ **then**
    return the result of grade-school multiplication for $a \times b$;
  else
    $A_1 \leftarrow a \div 2^{n/2}$; $A_0 \leftarrow a \bmod 2^{n/2}$;
    $B_1 \leftarrow b \div 2^{n/2}$; $B_0 \leftarrow b \bmod 2^{n/2}$;
    $p_1 \leftarrow Multiply(n/2, A_1, B_1)$;
    $p_2 \leftarrow Multiply(n/2, A_1, B_0)$;
    $p_3 \leftarrow Multiply(n/2, A_0, B_1)$;
    $p_4 \leftarrow Multiply(n/2, A_0, B_0)$;
    return $p_1 \cdot 2^n + (p_2 + p_3) \cdot 2^{n/2} + p_4$;

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.

As one might of noticed, we didn't improve the time complexity whatsoever. This is where we employ a clever mathematical trick.

Observe the 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 some 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 1.7: 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 $a \times b$

```
Function Multiply(n, a, b):
    if n < 2 then
        return the result of grade-school multiplication for a × b;
    else
        A_1 ← a ÷ 2^{n/2};  A_0 ← a mod 2^{n/2};
        B_1 ← b ÷ 2^{n/2};  B_0 ← b mod 2^{n/2};

        x ← Multiply(n/2, A_1, B_1);
        y ← Multiply(n/2, A_0, B_0);
        z ← Multiply(n/2, A_1 + A_0, B_1 + B_0);
        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: $O(n)$.

Exercise 1.1: Evaluate the following functions with the Master Method and Substitution:

1. makes 2 recursive calls, recursive calls of length $n/3$, and does $\Theta(n)$ work.
2. makes 4 recursive calls, recursive calls of length $n/3$, and does $\Theta(n)$ work.
3. makes 9 recursive calls, recursive calls of length $n/3$, and does $\Theta(n^2)$ work.

Note: Helpful log identities: $\log_a 2b = 2\log_a b$; $a^{\log_a b} = b$; $a^{\log_b c} = c^{\log_b a}$.

Exercise 1.2: Find the recurrence relation for the below function then solve it:

Function 1.8: Mystery Function - MYST(n)

```
if n ≤ 1 then
    return 1
else if x > 6 then
    return MYST(n/2) + O(n)
else if x > 3 then
    return MYST(n/3) + O(n)
else
    return MYST(n − 1) + O(n)
```

Exercise 1.3: Find the recurrence relation for the below function then solve it:

Function 1.9: Mystery Function - MYST2(n)

```
if n ≤ 1 then
    return 1
else
    return MYST2(n − 1) + MYST2(n − 1)
```

Exercise 1.4: Find the recurrence relation for the below function then solve it:

Function 1.10: Mystery Function - MYST3(A,n)

Takes an array A, and an integer n. The min, max functions returns the smallest and largest numbers in A respectively.

```
if A.length ≤ 1 then
    return 1
else
    MYST3(A[0,...,n/2]), min(A);
    MYST3(A[0,...,n/2]), max(A);
    MYST3(A[0,...,n/2]), 0;
```

Exercise 1.5: Are the following statements "Always True," "Sometimes True," or "Never True."

1. If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.
2. If $f(n) = O(g(n))$, then $g(n) - f(n) = \Omega(g(n))$.
3. If $f(n) = O(g(n))$, then $f(n) = \Omega(g(n))$.
4. If $f(n) = O(g(n))$, then $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

Answer 1.1:

1. $2^k\ T\left(\frac{n}{3^k}\right) + k\Theta(n)$: Start by rewriting as $2^{\log_3 n}\ T\left(\frac{n}{3^{\log_3 n}}\right) + \log_3 n\Theta(n)$. Simplifying this gives $2^{\log_3 n} + \Theta(1) + n^{\log_3 n}$. Using the identity $a^{\log_b c} = c^{\log_b a}$, we find that the expression becomes $n^{\log_3 2} + n\log_3 n$. Since $nlog_3(n) = log_3(n^2) = O(n)$, the result simplifies further to $n^{\log_3 2} + O(n)$. Noting that $n^{\log_3 2} < O(n)$ (because $\log_3 2$ is a fraction), the final result is $O(n)$.

2. $4^k\ T\left(\frac{n}{3^k}\right) + k\Theta(n)$: Rewriting gives $4^{\log_3 n} + n^{\log_3 n} = n^{\log_3 4} + O(n)$. Since $n^{\log_3 4} > n$, the final result is $O(n^{\log_3 4})$.

3. $9^k\ T\left(\frac{n}{3^k}\right) + k\Theta(n^2)$: Start by observing as $9^{\log_3 n} = 3^{2\cdot\log_3 n} = 3^{\log_3 n^2} = n^2$. Then $n^2 \cdot \Theta(1) + (\log_3 n)\Theta(n^2) = n^2 + n^2\log_3 n$, hence $O(n^2\log_3 n)$.

Note: Log magnitudes: $\log_2(8) = 3$; $\log_8(2) = \frac{1}{3}$; $\log_2(\frac{1}{8}) = -3$; $\log_8(\frac{1}{2}) = -\frac{1}{3}$; $\log_2(\frac{1}{8}) = -3$.

Answer 1.2: Equation: $T(n-2) + \Theta(n)$, Time Complexity: $O(n^2)$.

Answer 1.3: Equation: $2T(n-1) + \Theta(1)$. General Case: $2^kT(n-k) + k\Theta(1)$, solve $n - k = 0$. Base Case $(n = k)$: $2^nT(0) + n\Theta(1) = 2^n\Theta(1) + n\Theta(1)$. Hence $O(2^n)$.

Answer 1.4: Equation: $T(n/2) + \Theta(n)$, Recurrence: $O(nlog_2)$.

Answer 1.5:

1. Sometimes True: $f(n) = g(n) = n$, then $2^{f(n)} = O(2^{g(n)})$. $f(n) = 2n, g(n) = n$, then $2^{f(2n)} \neq O(2^{g(n)})$ as $4^n > 2^n$.

2. Sometimes True: $f(n) = g(n) = n$, then $g(n) - f(n) = 0 \neq \Omega(g(n))$. $f(n) = n, g(n) = 2n$, then $n2 - n = n = O(g(n))$.

3. Sometimes True: $f(n) = \Theta(g(n)) = O(g(n))$. $f(n) = 1, g(n) = n$, then $1 \neq \Omega(n)$.

4. Never True: $g(n)$ should dominate the expression, hence $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$.

— 10 —

Dynamic Programming

10.1 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:

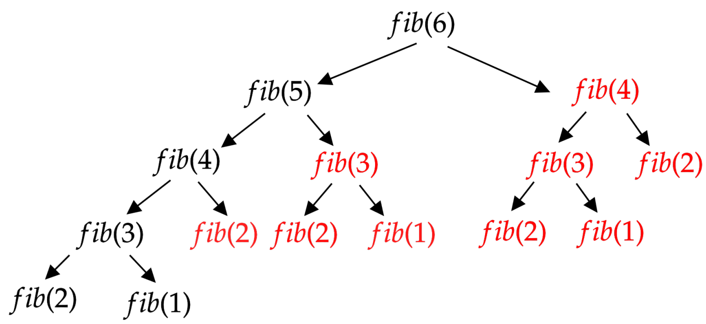


Figure 10.1: Recursion Tree for Fibonacci Sequence

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.

$F[\,]$; `// Table to store Fibonacci numbers`
Function *Fib(n)***:**
    if $n \leq 1$ **then**
        return n;
    else
        if $F[n]$ *is not defined* **then**
            $F[n] = Fib(n-1) + Fib(n-2)$;
        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 (7.1)

Let us define $\mathbf{OPT(j)}$ 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**.

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

10.2 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()*

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

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

To offer intuition, recall figure (10.1), 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()*

```
Sort jobs by finish time; // O(n log n)
Compute all p(1), ..., p(n); // O(n)
OPT[0] ← 0; // Base case (array of size n + 1)
for j ← 1 to n do
    OPT[j] ← max{v_j + OPT[p(j)], OPT[j − 1]};
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 10.2: 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.

10.3 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 (10.2), 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` $1, \ldots, j$

1 $j \leftarrow OPT.length - 1$; $S \leftarrow \{\}$; `//` 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** $S \cup Backtrack(OPT, p(j))$;

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 (8.1), 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.

10.4 Subset Sum

10.4.1 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 10.1: 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 (10.1), 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()*

S; T; $OPT[\,][\,]$; `// Set` S`, Weight ceiling` T`, DP table` $OPT(i, T)$
$OPT[0][*] \leftarrow 0$; `// Base case (array of size 0)`
$OPT[*][0] \leftarrow 0$; `// Base case (array of size` $T = 0$`)`
for $i \leftarrow 1$ **to** $S.length$ **do**
    for $t \leftarrow 1$ **to** T **do**
        if $S[i] > t$ **then**
            $OPT[i][t] \leftarrow OPT[i-1][t]$;
        else
            $OPT[i][t] \leftarrow max\{\underbrace{OPT[i-1][t]}_{\text{next solution}}, \underbrace{S[i] + OPT[i-1][t - S[i]]}_{\text{build solution}}\}$;
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 (10.1) 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()*

$i \leftarrow 0$; $t \leftarrow T$; $S \leftarrow \{\}$; `//` S `is our set of jobs`
$Backtrack(OPT, i, t)$;
Function *Backtrack(OPT, i, t)*:
  `//` `Where` $OTP.length$ `is the number of rows`
  while $i < OPT.length$ **do**
    if $OPT[i][t] > OPT[i+1][t]$ **then**
      $S \leftarrow S \cup \{S[i]\}$;
      $t \leftarrow t - S[i]$;
    $i \leftarrow i + 1$;
  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, $P = \{7, 2\}$.

10.4.2 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()*

$i \leftarrow OPT.length$; $w \leftarrow W$; $S \leftarrow \{\}$; // S is our set of jobs
while $i > 0$ *AND* $w > 0$ **do**
  if $OPT[i][w] > OPT[i-1][w]$ **then**
    $S \leftarrow S \cup \{i\}$;
    $w \leftarrow w - w_i$;
  $i \leftarrow i - 1$; // In both cases we $i - 1$
return S;

Time Complexity: $O(n)$.

10.4.3 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:

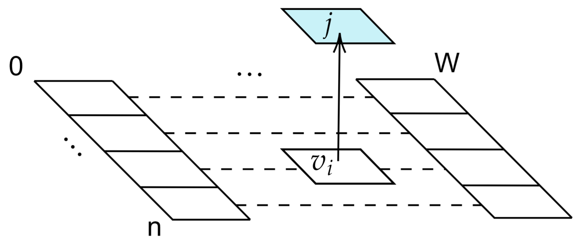


Figure 10.3: Illustrating the relationship between C and M, where v_i relates to its respective j

We implement the following algorithm:

Function 4.4: Unbounded Knapsack - *UnboundedKnapsack()*

$M \leftarrow (n+1) \times (W+1)$ table `// DP table`
$C \leftarrow (n+1) \times (W+1)$ table `// Number of copies`
$M[0][*] \leftarrow 0$ and $M[*][0] \leftarrow 0$;
for $i \leftarrow 1$ **to** n **do**
    for $w \leftarrow 1$ **to** W **do**
        $M[i][w] \leftarrow M[i-1][w]$; `// Start with previous solution`
        $m \leftarrow 1$;
        while $m \cdot w_i \leq w$ **do**
            $val \leftarrow m \cdot v_i + M[i-1][w - m \cdot w_i]$;
            if $val > M[i][w]$ **then**
                $M[i][w] \leftarrow val$;
                $C[i][w] \leftarrow m$;
            $m \leftarrow m + 1$;
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()*

$sol \leftarrow \emptyset$;
$i \leftarrow n$ and $w \leftarrow W$;
while $i > 0$ ***and*** $w > 0$ **do**
    $sol \leftarrow sol \cup \{i \cdot C[i][w]\}$; `// Add item` i `with its count` $C[i][w]$
    $w \leftarrow w - C[i][w] \cdot w_i$;
    $i \leftarrow i - 1$;
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:

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

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 (10.4.2), and follow the table left to right, top to bottom.

Function 4.6: Unbounded Knapsack - *UnboundedKnapsack2()*

```
M ← (n + 1) × (W + 1) table* // DP table
M[0][*] ← 0 and M[*][0] ← 0 // Set first row and column to 0
for i ← 1 to n do
    for w ← 1 to W do
        if w_i > w then
            M[i][w] ← M[i − 1][w];
        else
            val ← v_i + M[i][w − w_i] // Take one more copy of i
            if val > M[i − 1][w] then
                M[i][w] ← val;
            else
                M[i][w] ← M[i − 1][w];
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()*

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

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

10.5 Shortest Paths - Bellman-Ford Algorithm

Revisiting the shortest path problem (8.1), 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.

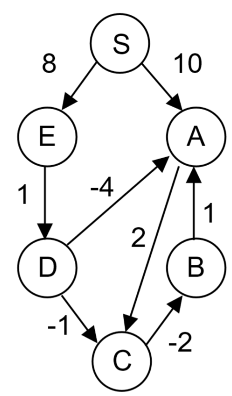
(ii.) For each v, iterate out-degrees u, and evaluate $w(v, u)$.

(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.



	0	∞	∞	∞	∞	∞
i	S	A	B	C	D	E
0:	0	10	10	12	9	8
1:	0	5	10	8	9	8
2:	0	5	5	7	9	8
3:	0	5	5	7	9	8

	∅	∅	∅	∅	∅	∅
$P[v]$	S	A	B	C	D	E
	∅	S	C	A	E	S
	∅	D	C	D	E	S
	∅	D	C	A	E	S
	∅	D	C	A	E	S

Figure 10.4: 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 $\underline{min\{\text{evaluation weight}, \text{current weight}\}}$.

- **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 (10.4), 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 $a \to c$:

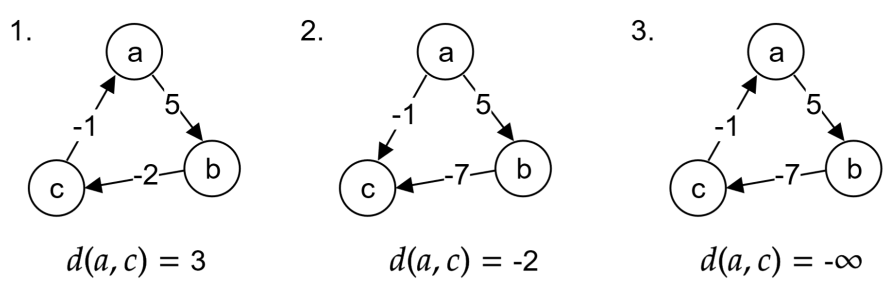


Figure 10.5: Three graphs: 1. A positive cycle, 2. A DAG, 3. A negative cycle.

Examining 3. in Figure (10.5), 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 $a \to c$, subsequently updating a new shortest path $a \to b$, 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 (10.4)'s routine into a recursive formula:

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

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 (10.4) 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 $G \leftarrow$ Graph `// Graph` G `with` n `nodes`
2 *parents* $\leftarrow$ length-n table `// Parents list for shortest paths tree`
3 $M \leftarrow n \times n$ table `// DP table` $M[i][v] = OPT(i,v)$
4 $M[0][*] \leftarrow \infty$ `// Set all base values`
5 $M[0][s] \leftarrow 0$ `// Base case for source`
6 **for** $i \leftarrow 1$ **to** $n-1$ **do**
7   **for** $v \in G$ **do**
8         $M[i][v] \leftarrow M[i-1][v]$; `// Copy previous iteration`
9   **for** $u \in G[v]$ **do**
10   **if** $M[i-1][u] > M[i][v] + G[v][u]$ **then**
11                 $M[i][u] \leftarrow M[i][v] + G[v][u]$;
12   *parents*$[u] \leftarrow v$;
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 $parents[A] = D$.

— 11 —

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.

11.1 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.,

$$C(A, B) := \sum_{u \in A, v \in B} c(u, v) \tag{11.1}$$

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

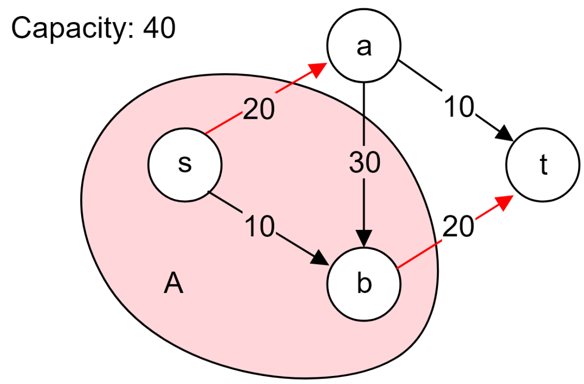


Figure 11.1: Flow Network with Capacity Cut A

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.

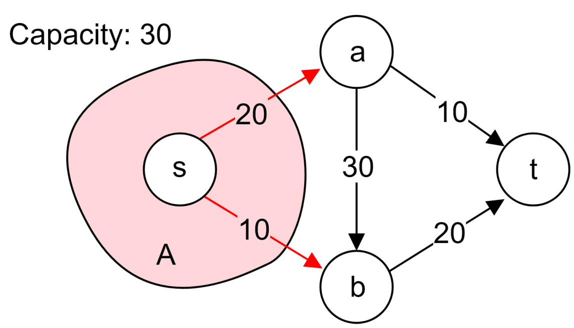


Figure 11.2: Flow Network with Minimum Cut

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.

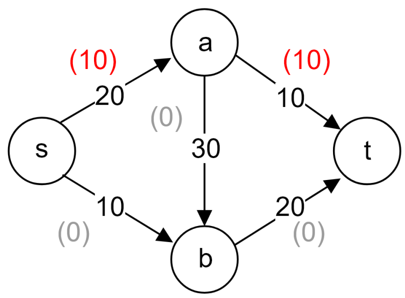


Figure 11.3: Flow Network with Flow a valid flow f, as what comes in, is what comes out.

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.

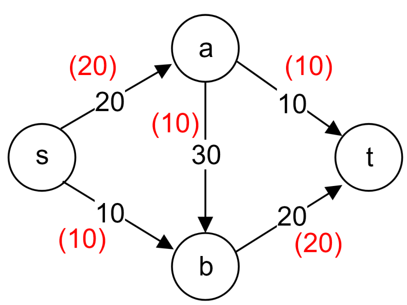


Figure 11.4: Flow Network with Maximum Flow

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 $v \in B$:

$$V(f) = \sum_{\{u \to v\}} f(u,v) - \sum_{\{u \leftarrow v\}} f(v,u) \tag{11.2}$$

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

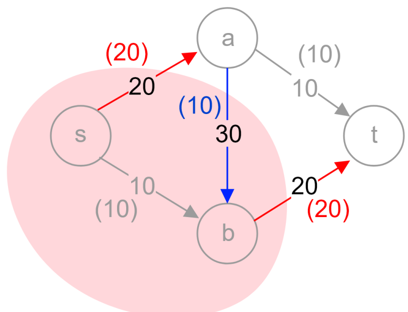


Figure 11.5: An s-t cut with net-flow equaling $V(f)$: $(20+20)-(10)=30=V(f)$.

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.,

$$V(f) \leq C(A,B) \tag{11.3}$$

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,

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

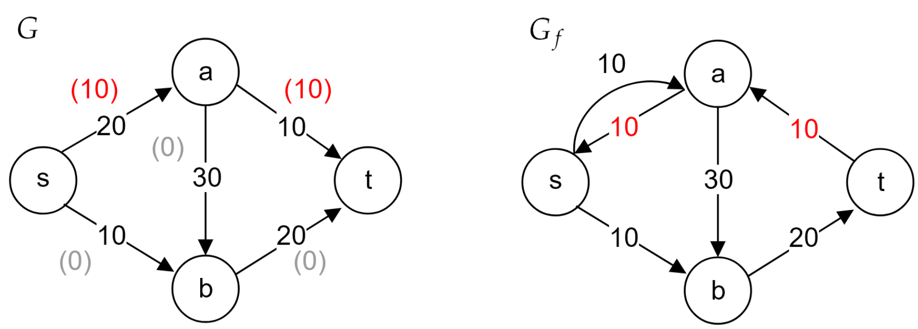


Figure 11.6: A graph G and its residual graph G_f.

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.

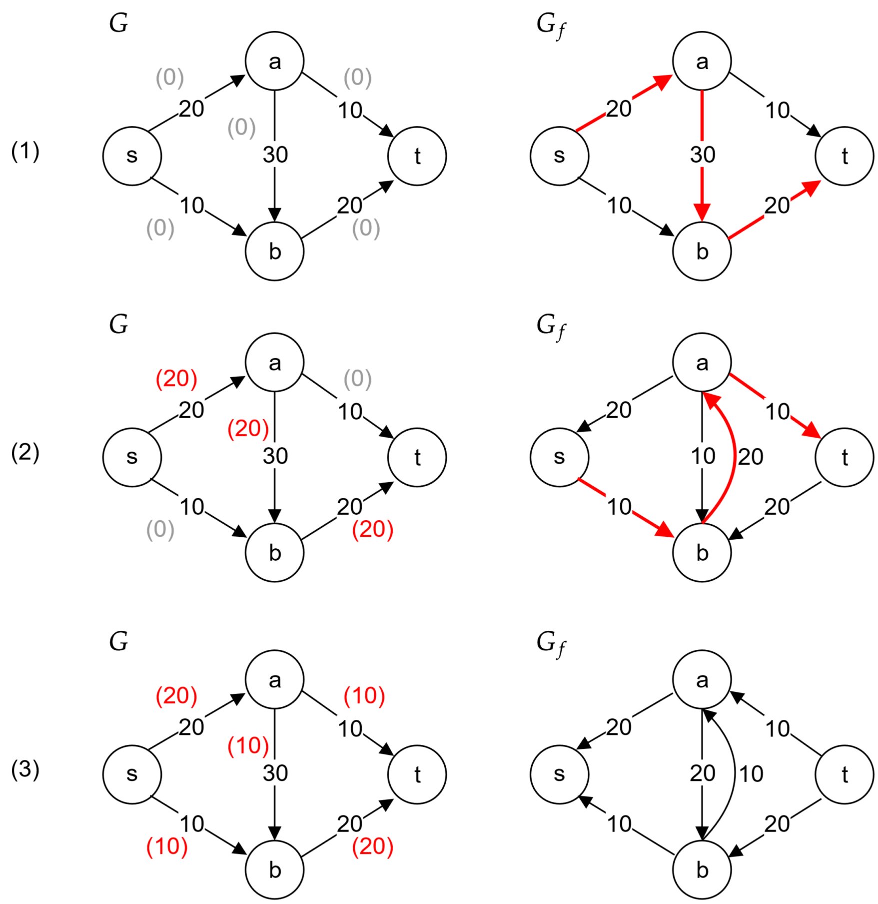


Figure 11.7: Ford-Fulkerson Algorithm in action, terminating at 3 where $s \to t$ is blocked.

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):
2     foreach e ∈ E do
3         f(e) ← 0 // Initialize flow on each edge to zero
4     G_f ← residual(G, f) // Construct the residual graph
5     while there exists path P := s → t in G_f do
6         f ← Augment(f, 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

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

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

Theorem 1.3: Residual Graph Min-cut Corollary

Given a network-flow that has reached its max-flow, a min cut can be found in one of two ways using an s-t cut $C(A, B)$:

- **From s**: Let A be the partition of nodes reachable from s in the residual graph. Then the cut formed by a is a min cut.
- **To t**: Let B be the partition of nodes that reach t in the residual graph. Then the cut formed by B is a min cut.

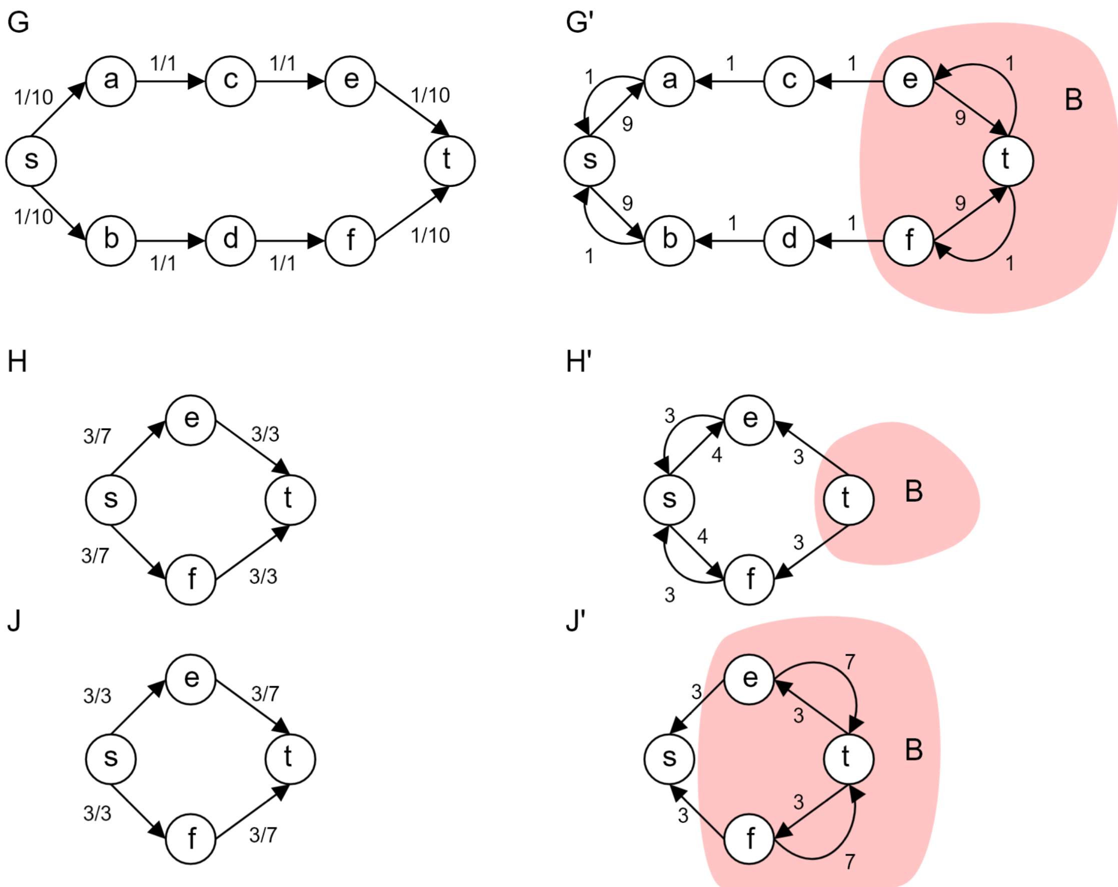


Figure 11.8: Network flows graphs adjacent to their residual graphs showing min-cut partitions.

The above graph shows examples of $C(A, B)$ cuts forming min-cuts by nodes in B reaching the sink t in the residual graph.

11.2 Bipartite Matching

In this section we revisit matching problems alike what we saw with Gale-Shapely, Section (5.1). 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**.

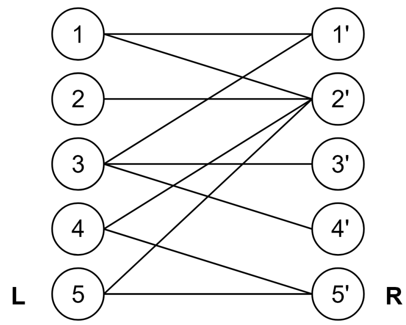


Figure 11.9: A Bipartite Graph $G(V, E)$ cut into two sets $L := \{1, \ldots, 5\}$ and $R := \{1', \ldots, 5'\}$.

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

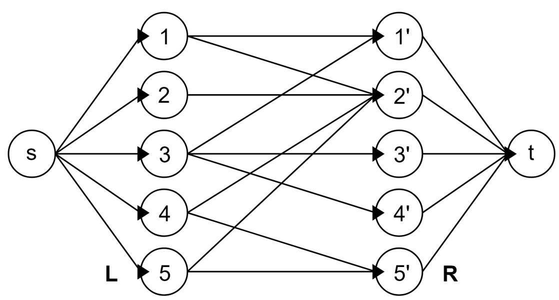


Figure 11.10: Figure (11.9) with a source node s added to L and sink t added to R.

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.

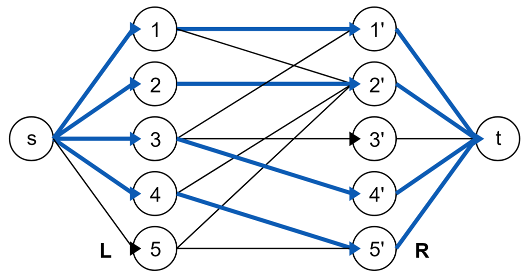


Figure 11.11: A maximum matching found through max-flow.

11.3 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.

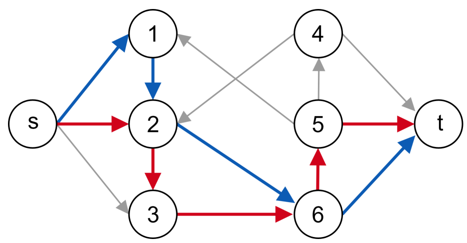


Figure 11.12: Two edge-disjoint paths p_1(red) and p_2(blue) from s to t.

Theorem 3.1: Max-Edge Disjoint Paths

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

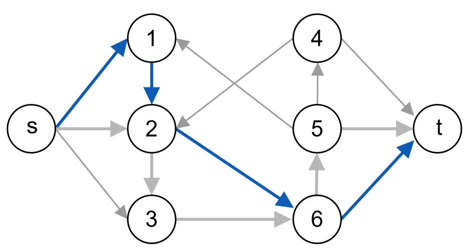


Figure 11.13: Figure (11.12) of flow 2, now of flow 1 after deleting p_1(*red*).

Deleting a path subtracts 1 from the flow. Hence, the number of unique paths equates to max-flow.

11.4 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?

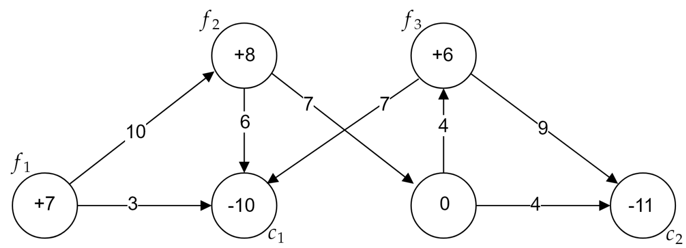


Figure 11.14: A graph showing f_j farms' variation connections to c_k communities. (+) denoting the amount of resources added and (-) the amount of resources taken.

Definition 4.1: Super Source & Sink

Given a graph $G = (V, E)$, with multiple sources $s_1, \dots, s_f$ and sinks $t_1, \dots, t_c$. 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**.

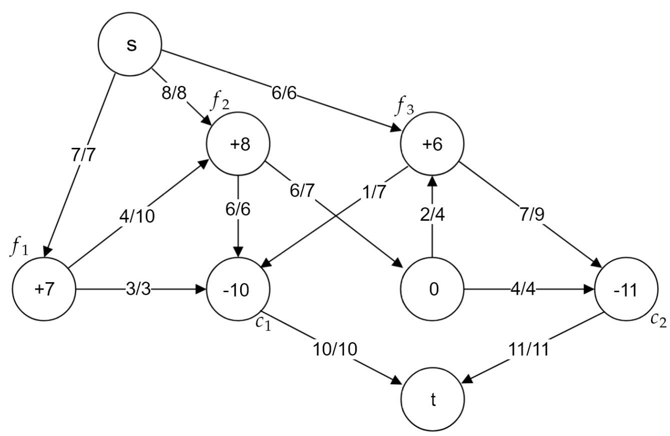


Figure 11.15: A graph with a super source s and super sink t on Figure (11.14).

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 needs 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 for each day.

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

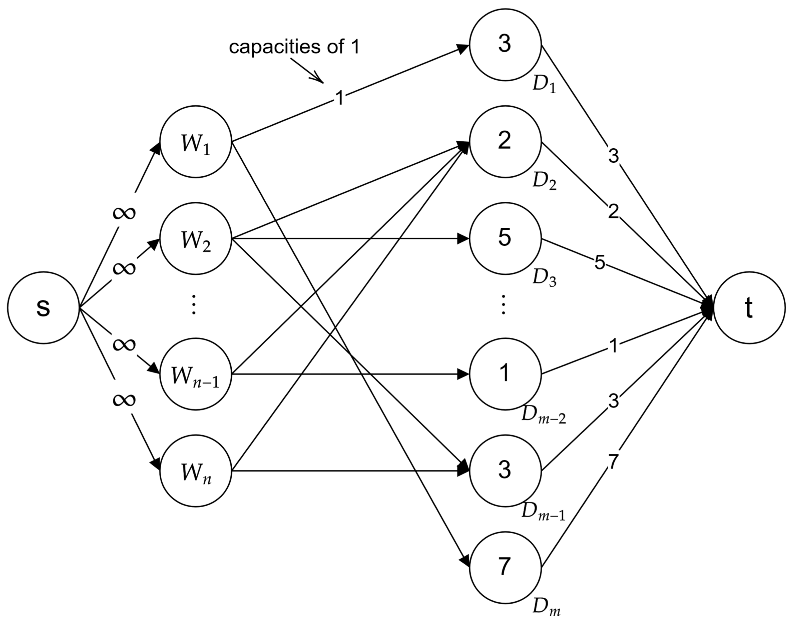


Figure 11.16: A network flow of W_z's availability on D_y nodes showing the students needed.

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 $\sum\limits_{c \in C} c = \sum\limits_{f \in F} f$. This also ensures that we at least meet the minimum demands.

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, $\sum\limits_{c \in C} c = \sum\limits_{f \in F} f$, demands are met. This also ensures that we at least meet the **minimum demands**.

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).

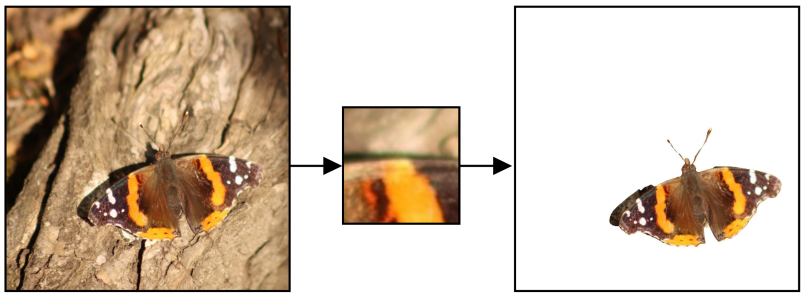


Figure 11.17: A high-level diagram of image segmentation.

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-labelings 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

Given 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.

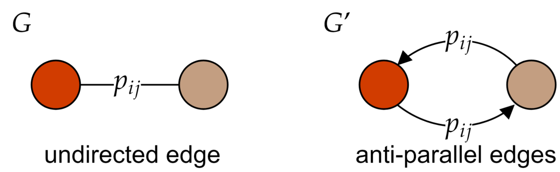


Figure 11.18: Showing an edge in G vs. its anti-parallel edges in G'.

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.

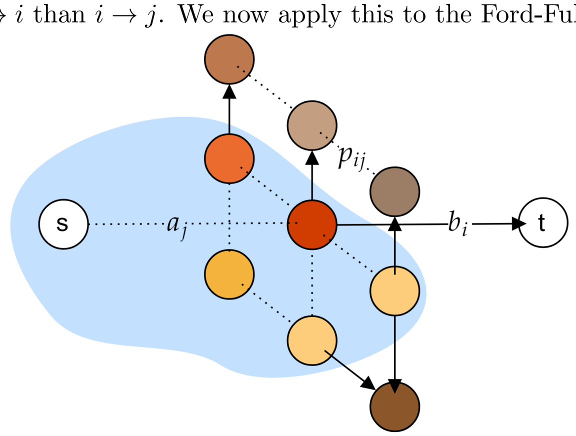


Figure 11.19: A flow network G where the min-cut is highlighted in blue. Solid lines represent edges leaving the min-cut (not all connections from s and t to and from the image are shown).

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

— 12 —

Computational Hardness

12.1 P, NP, NP-Complete, EXP

Definition 1.1: Decision Problem

A **decision problem** is a problem that can be answered with a simple "yes" or "no" answer.

Definition 1.2: Complexity Classes

A **complexity class** is a set of problems of related complexity, solvable under the same system constraints. A problem x is in complexity class X if they share similar characteristics in complexity. A problem is **solvable** if it runs in polynomial time. Notable classes:

- **P** - Solvable in polynomial time
- **NP** - Verifiable ("yes") in Polynomial time (nondeterministic polynomial time).
- **NP-Hard** - Not necessarily NP, a decision problem, nor has a solution. If a problem H is NP-Hard, all problems L in NP can be reduced to H. I.e., H can run as a sub-routine solving all NP. It is said that NP-Hard problems are at least as hard as NP problems.
- **NP-Complete** - NP and NP-Hard.
- **EXP** - Solvable in exponential time.

Tip: Alan Turing (1912-1954) was a British mathematician and logician. In 1936, he introduced the concept of the Turing machine, laying the groundwork for theory of computation. During World War II Turing broke the German Enigma encrypted code shifting the tides of the war. He after worked on early computers and artificial intelligence, proposing the "Turing Test", defining machine intelligence. Tragically, Turing's life was cut short by personal and professional persecution against his homosexuality, dying at 41.

Example 1.1: P, NP, NP-Hard, NP-Complete, EXP

- **P** - Sorting algorithms.
- **NP** - Travelling Salesman Problem (1.2).
- **NP-Hard** - Halting Problem (1.1).
- **NP-Complete** - Boolean Satisfiability Problem (1.2).
- **EXP** - Towers of Hanoi (1.3), or unoptimized Fibonacci.

■

Function 1.1: Halting Problem - *HaltingProblem()*

```
1 Function Halts(program, input):
2 |  if program(input) doesn't halts then
3 |  |  return "This program does not halt."
4 |  else
5 |  |  while true do
  |  |  |  // Loop forever
6 return Halts(Halts, Halts) // Run Halts on Halts with an input of itself
```

Explanation: The Halting Problem is a decision problem that determines whether a given program will halt (terminate) or run forever on a given input. This problem is undecidable, meaning there is no general algorithm that can solve the Halting Problem for all possible program-input pairs.

The above is an example of a paradoxical function. Line 2 checks if the program does not halt, if so, it returns that the program does not halt. Otherwise, it loops forever. Though if line 2 is true, that means the inner program looped forever—a contradiction—if the inner program never halted, the outer function could have never evaluated it. If it did halt, the outer function would have ran forever. A similar paradox, "(1) The following statement is true. (2) The previous statement is false."

Despite the contrived example, the root is that given on any input it is impossible to automatically detect all cases. IDEs (Integrated Development Environments) today can detect some cases, but not all. Another example in the same class would be program which identifies **Dead-code** (code that won't run). These problems are NP-complete, as if dead-code is solved, then the Halting Problem could be reduced to it.

Tip: John von Neumann (1903-1957) was a Hungarian-American mathematician and polymath who made fundamental contributing to, mathematics, physics, economics, and computer science. He is best known for developing the "*Von Neumann Architecture.*"

Function 1.2: 3-SAT

```
Function 3-SAT(formula):
    for each clause in formula do
        if clause is not satisfied then
            return false
    return true
```

Explanation: 3-SAT is a Boolean Satisfiability Problem (SAT) where each clause has exactly 3 literals. E.g.,

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge \cdots \wedge (\neg x_1 \vee \neg x_2 \vee x_3).$$

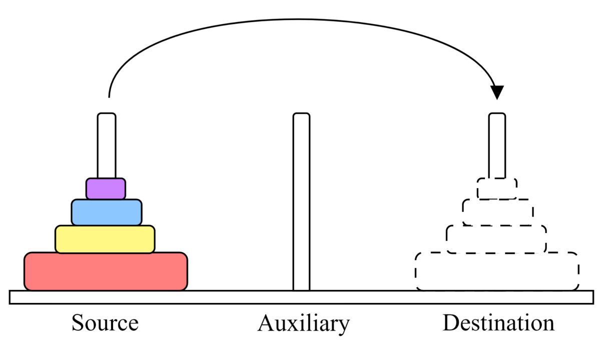
$\wedge :=$ AND, $\vee :=$ OR, $\neg :=$ NOT. Given an input formula, the function determines if there's a possible true/false assignment to evaluate the formula to true. The best approach today is by brute force, trying all possible assignments.

This problem is NP-Complete, as any function could theoretically be reduced to a true-false table for which an SAT function could be solved.

Function 1.3: Towers of Hanoi

```
Function TowersOfHanoi(n, source, target, auxiliary):
    if n = 1 then
        // Move disk 1 from source to target
    else
        TowersOfHanoi(n-1, source, auxiliary, target);
        // Move disk n from source to target
        TowersOfHanoi(n-1, auxiliary, target, source);
```

Explanation: There are n different sized disks stacked in decreasing order of size and three rods. Rules: **(1).** One disk is moved at a time. **(2).** Each turn, a top disk moves from one stack to another. **(3).** Bigger over smaller disks are prohibited. Given the above, $O(2^n)$ (1.5).



For emphasis and clarity:

Theorem 1.1: Complexity Classes Corollary

- **P=NP, NP≠P -** all P are NP as they are verifiable in polynomial time. Though as of today, no proof exists that all verifiable problems are solvable in polynomial time.
- **X reducing to Y -** If X reduces to Y and Y is solvable in $O(n^2)$, then X is also solvable in polynomial time. However, this does not mean that Y's solution alone determines the overall complexity of solving X. The reduction itself may introduce additional overhead, or Y could be invoked multiple times during the solution of X.
- **Verification -** To verify an example is given and a criteria checked. E.g., is this path at least 10 units long? However to verify "No" requires the program to provide proof that the solution is not possible. I.e., to verify a "No" answer, we must check all possible solutions for which we don't even know how to compute.

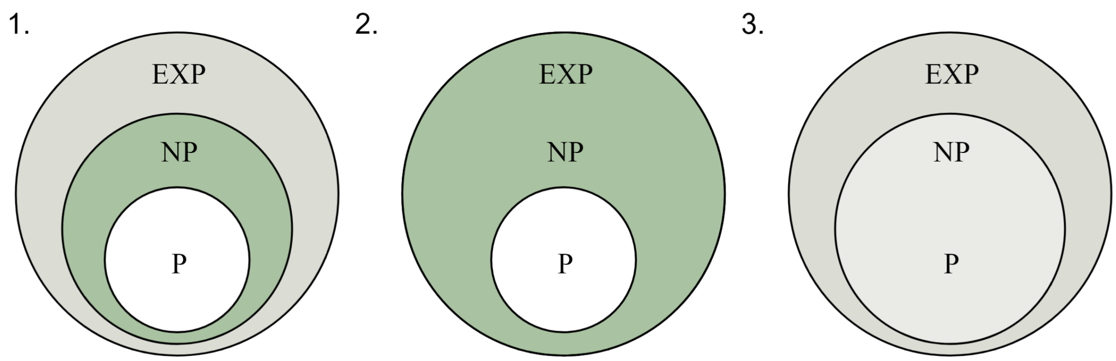
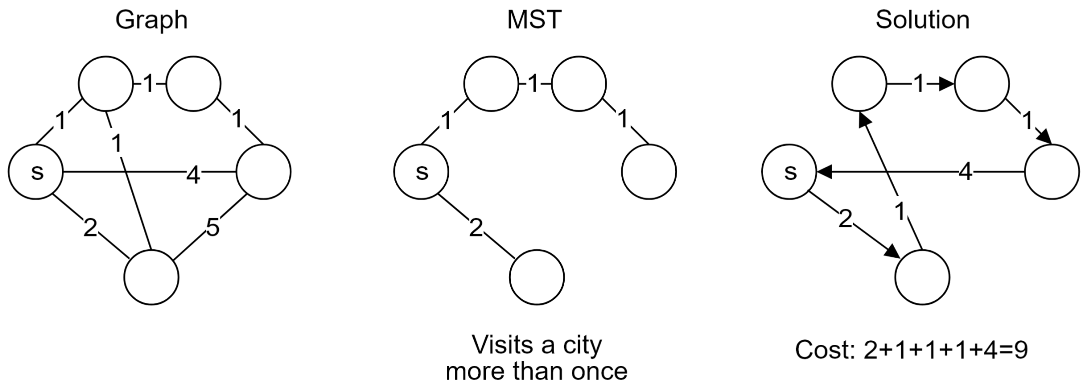


Figure 12.1: Complexity Classes Ven Diagram of P, NP, EXP

In Figure (12.1), **(1)** is the general consensus on the state of complexity classes. We don't know whether all NP are P, or whether all EXP are NP. **(2)** shows that it could be that all EXP are NP, but we don't know. Likewise, **(3)** shows that it could be that all NP are P, but we don't know.

Theorem 1.2: The Traveling Salesman Problem (TSP)

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? This problem is NP-Complete.



Note: that this does not mean MST, as MST finds the minimal cost to connect all nodes. It isn't strictly shortest paths from the origin to all nodes and back, as we'd double-count nodes.

NP Verification:
"Does this path visit all nodes once returning to the origin in less than k units?"

— 13 —

Summary

This page is left intentionally blank.

Growth Rates: $1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n! < n^n$ (0.8) — **(htbl)**:= hash-table
O**(Upper Bound)**: $f(n) < g(n) \implies \lim_{n\to\infty}\frac{f(n)}{g(n)} = 0$ (1.2) — **(conn.)**:= connected
Ω**(Lower Bound)**: $f(n) > g(n) \implies \lim_{n\to\infty} 0 < \frac{f(n)}{g(n)}$ (1.3)
Θ**(Upper & Lower Bound)**: $f(n) = g(n) \implies 0 < \frac{f(n)}{g(n)} < \infty$ (1.4)

Interval Scheduling: sort by finish time ascending order, choose next compatible job (1.1)
Interval Partition: sort by start time, find next available resource (2.2)
Min Lateness: sort by finish time, take next available job (4.1)

Master Method:

$T(n) = aT\left(\frac{n}{b}\right) + f(n^d)$: (1.5)
$d > \log_b a \implies T(n) = \Theta(n^d)$
$d < \log_b a \implies T(n) = \Theta(n^{\log_b a})$
$d = \log_b a \implies T(n) = \Theta(n^{\log_b a}\log n)$

Big-Os:
logs: $\log n = O(\sqrt{n}) = O(n)$(1.2)(1.2)
edges: $m \leq O(n^2)$ typically $m < n$ (1.8)
long $(\times, \div)$ **:** $O(n^2)$ (3.2)
Karatsuba: $O(n^{\log_2 3}) \approx O(n^{1.59})$ (11)
sorting: $O(n\log n)$ (1.1)

BFS: $O(n+m)$ (6.2.1). Siblings before children, shortest paths tree.
DFS: $O(n+m)$ (6.2.1). Children before siblings, not longest paths.
Heap: sorting $O(n\log n)$, operations $O(\log n)$ except for extract-min $O(1)$ (7.3).
Dijkstra:$O((n+m)\log n)$;$O(m\log n)$(conn.);$O(n^2)$(htbl). Min paths from s (no neg.paths) (8.1).
Prim's: $O(m\log n)$. $O(n^2)$(htbl). MST, dijkstra like routine, (2.4).
Kruskal's: $O(m\log n)$ or $O(m\log m)$. MST, sort & pick edges ascending, union-find (2.6).
Bellman-Ford: $O(nm)$. Shortest paths from s via relax rounds on n (no negatives cycles) (5.1).
Ford-Fulkerson: $O(mnC)$, $C :=$ largest capacity, finds max flow (5.1).

P: decision problems solvable in polynomial time (12.1).
NP: decision problems verifiable in polynomial time. E.g., "is this path $\geq k$."
NP-Complete: a problem X that if solved in polynomial time, all NP problems may use it as a subroutine to be solved in polynomial time.
NP-Hard: an NP-Complete problem which may not be in NP or have a solution.
EXP: decision problems solvable in exponential time.

Substitution Examples: (1.6)

(a) $2^k\, T\left(\frac{n}{3^k}\right) + k\Theta(n)$: $2^{log_3 n}\, T\left(\frac{n}{3^{log_3 n}}\right) + log_3 n\Theta(n)$. Simplifying: $2^{log_3 n} + \Theta(1) + nlog_3 n$. Then apply the identity $a^{log_b c} = c^{log_b a}$. Then $n^{log_3 2} + log_3(n)n$. Then,$log_3(n)n = log_3(n^2) = O(n)$. Then $n^{log_3 2} < O(n)$, as $log_3 2$ must be some fraction rooting n. hence $O(n)$

(b) $4^k\, T\left(\frac{n}{3^k}\right) + k\Theta(n)$: $4^{log_3 n} + n^{log_3 n} = n^{log_3 4} + O(n)$. Then $O(n^{log_3 4})$ as $log_3 4 > 1$

(c) $9^k\, T\left(\frac{n}{3^k}\right) + k\Theta(n^2)$: $9^{log_3 n} = 3^{2\cdot\log_3 n} = 3^{\log_3 n^2} = n^2$. So $n^2 + n^2 log_3 n = O(n^2 log_3 n)$

Log Rules: $\log_2(8) = 3, \log_8(2) = \frac{1}{3}, \log_2(\frac{1}{8}) = -3, \log_8(\frac{1}{2}) = -\frac{1}{3}, n^{\log_b a} = a^{\log_b n}, n^{\log_n a} = a$
$\log(ba) = \log b + \log a, \log(\frac{b}{a}) = \log b - \log a, \log_b a = \frac{\log a}{\log b}, 2\log a = \log a^2$ (0.4).

Dynamic Programming:

Weighted Int. Sched. ($O(n \log n)$): sort by finish time, find next compatible job (8)

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

Subset Sum($O(nT)$)(10.4.1):

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

Knapsack($O(nW)$)(10.4.2):

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

Knapsack Unbounded (unoptimal)($O(nW^2)$)(10.4.3):

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

Knapsack Unbounded (optimal)($O(nW)$)(7):

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

(10.1)

Function 0.1: Slow Fibonacci Sequence - *Fib()*

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

Function *Fib(n)*:
  if $n \leq 1$ **then**
    return n;
  else
    return $Fib(n-1) + Fib(n-2)$;

Time Complexity: $O(2^n)$. $T(n) = T(n-1) + T(n-2) + O(1)$, Theorem (1.5).

Function 0.2: Memo Fibonacci Sequence - *Fib()*

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

$F[\,]$; `// Table to store Fibonacci numbers`
Function *Fib(n)*:
  if $n \leq 1$ **then**
    return n;
  else
    if $F[n]$ *is not defined* **then**
      $F[n] = Fib(n-1) + Fib(n-2)$;
    return $F[n]$;

Time Complexity: $O(n)$. **Space Complexity:** $O(n)$.

Function 0.3: Bottom-Up Fibonacci Sequence - *Fib()*

$F[0] \leftarrow 0$; $F[1] \leftarrow 1$; `// Base cases (array of size` $n+1$`)`
for $i \leftarrow 2$ **to** n **do**
  $F[i] \leftarrow F[i-1] + F[i-2]$;
return $F[n]$;

Time Complexity: $O(n)$.

(2.2)

Function 0.4: Bottom-Up Weighted Interval Scheduling - *OPT()*

Sort jobs by finish time; `//` $O(n \log n)$
Compute all $p(1), \ldots, p(n)$; `//` $O(n)$
$OPT[0] \leftarrow 0$; `// Base case (array of size` $n+1$`)`
for $j \leftarrow 1$ **to** n **do**
  $OPT[j] \leftarrow max\{v_j + OPT[p(j)], OPT[j-1]\}$;
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.

Function 0.5: Backtracking Weighted Interval Scheduling - *Backtrack()*

`//` $OPT[\ ]$ `(optimal solutions of` j `job) and` $p()$ `(next comptible job) are already computed for` $1, \ldots, j$
1 $j \leftarrow OPT.length - 1$; $S \leftarrow \{\}$; `//` 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** $S \cup Backtrack(OPT, p(j))$;
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 (8.1), 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.

(4.1)

Function 0.6: Subset Sum - *OPT()*

S; T; $OPT[\,][\,]$; `// Set` S`, Weight ceiling` T`, DP table` $OPT(i, T)$
$OPT[0][*] \leftarrow 0$; `// Base case (array of size 0)`
$OPT[*][0] \leftarrow 0$; `// Base case (array of size` $T = 0$`)`
for $i \leftarrow 1$ **to** $S.length$ **do**
    for $t \leftarrow 1$ **to** T **do**
        if $S[i] > t$ **then**
            $OPT[i][t] \leftarrow OPT[i-1][t]$;
        else
            $OPT[i][t] \leftarrow max\{\underbrace{OPT[i-1][t]}_{\text{next solution}}, \underbrace{S[i] + OPT[i-1][t-S[i]]}_{\text{build solution}}\}$;
return OPT;

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

Function 0.7: Backtracking Subset Sum - *Backtrack()*

$i \leftarrow 0$; $t \leftarrow T$; $S \leftarrow \{\}$; `//` S `is our set of jobs`
$Backtrack(OPT, i, t)$;
Function *Backtrack(OPT, i, t)***:**
    `// Where` $OTP.length$ `is the number of rows`
    while $i < OPT.length$ **do**
        if $OPT[i][t] > OPT[i+1][t]$ **then**
            $S \leftarrow S \cup \{S[i]\}$;
            $t \leftarrow t - S[i]$;
        $i \leftarrow i + 1$;
    return S;

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

Knapsack is skipped as it is similar to Subset Sum and Unbounded Knapsack (4.3). Below is the unoptimal version of Unbounded Knapsack (4.4).

Function 0.8: Unbounded Knapsack - *UnboundedKnapsack()*

$M \leftarrow (n+1) \times (W+1)$ table `// DP table`
$C \leftarrow (n+1) \times (W+1)$ table `// Number of copies`
$M[0][*] \leftarrow 0$ and $M[*][0] \leftarrow 0$;
for $i \leftarrow 1$ **to** n **do**
    for $w \leftarrow 1$ **to** W **do**
        $M[i][w] \leftarrow M[i-1][w]$; `// Start with previous solution`
        $m \leftarrow 1$;
        while $m \cdot w_i \leq w$ **do**
            $val \leftarrow m \cdot v_i + M[i-1][w - m \cdot w_i]$;
            if $val > M[i][w]$ **then**
                $M[i][w] \leftarrow val$;
                $C[i][w] \leftarrow m$;
            $m \leftarrow m + 1$;
return M, C;

Time Complexity: $O(n \times W^2)$ **Space Complexity:** $O(n \times W)$

Function 0.9: Backtracking Unbounded Knapsack - *BKBacktrack()*

$sol \leftarrow \emptyset$;
$i \leftarrow n$ and $w \leftarrow W$;
while $i > 0$ ***and*** $w > 0$ **do**
    $sol \leftarrow sol \cup \{i \cdot C[i][w]\}$; `// Add item` i `with its count` $C[i][w]$
    $w \leftarrow w - C[i][w] \cdot w_i$;
    $i \leftarrow i - 1$;
return sol;

Time Complexity: $O(n)$.

(4.6)

Function 0.10: Unbounded Knapsack - *UnboundedKnapsack2()*

```
M ← (n + 1) × (W + 1) table* // DP table
M[0][*] ← 0 and M[*][0] ← 0 // Set first row and column to 0
for i ← 1 to n do
    for w ← 1 to W do
        if w_i > w then
            M[i][w] ← M[i − 1][w];
        else
            val ← v_i + M[i][w − w_i] // Take one more copy of i
            if val > M[i − 1][w] then
                M[i][w] ← val;
            else
                M[i][w] ← M[i − 1][w];
return M;
```

Time Complexity: $O(n \times W)$.

Function 0.11: Traceback Solution for Unbounded Knapsack - *Traceback()*

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

Time Complexity: $O(n + W)$.

Bibliography

[1] Dfs, graph modeling. https://courses.cs.washington.edu/courses/cse417/21wi/lecture/06-DFS-model.pdf, 2021. Lecture 6.

[2] What is a transistor? YouTube video, https://youtu.be/AwXp6jVaTV4?si=s4-UwlgglmiCBPso, 2022.

[3] P.-Y. Chen. Chapter 22.1: Breadth-first search – clrs solutions. https://walkccc.me/CLRS/Chap22/Problems/22-1/, 2017–2024. Built by P.-Y. Chen, made with Material for MkDocs. Accessed: 2024-06-15.

[4] Computerphile. But, what is virtual memory?, 2023. Accessed: 2025-04-18.

[5] CrashCourse. The nucleus: Crash course chemistry #1. YouTube video, https://youtu.be/FSyAehMdpyI?si=h5ngW3IvcCTiOoL_, 2013. Published February 12, 2013.

[6] EngMicroLectures. Building logic gates from MOSFET transistors. https://youtu.be/1rZyGL1K5QI?si=ODgxP84kuHqVq6Bi, ?? 2013. [Online; accessed 2025-07-13].

[7] Rong Ge and Will Long (Scribe). Graph algorithms: Types of edges. https://courses.cs.duke.edu/fall17/compsci330/lecture12note.pdf, 2017. Lecture 12.

[8] Getty Images. Getty Images Stock Photo 700832601. https://www.thoughtco.com/thmb/TTC7l9oab0l_A2_xRPryeoHSvXc=/2092x1433/filters:no_upscale():max_bytes(150000):strip_icc()/GettyImages-700832601-5bb602c0c9e77c002609fe08.jpg. Thumbnail image hosted on ThoughtCo via Getty Images. Accessed: 2025-07-07.

[9] Infinity Learn. Concept of valency - introduction — atoms and molecules. YouTube video, August 2018. Accessed: 6 July 2025.

[10] The Engineering Mindset. Mosfet explained - how mosfet works. YouTube video, https://youtu.be/AwRJsze_9m4?si=whrgrmMmzrychH2c, 2024.

[11] David M. Mount. Cmsc 420: Lecture 6 – 2-3 trees. Lecture notes Lecture 6, University of Maryland Department of Computer Science, 2020. MIT OpenCourseWare.

[12] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, version 2 edition, 2008. Electronic version distributed under Creative Commons Attribution-NonCommercial-NoDerivs 3.0.

[13] Chris Terman. 6.004 computation structures, 2017. Undergraduate course, Spring 2017.

[14] Wayne Breslyn (Dr. B.). Molecule vs compound: Examples and practice. YouTube video, 2013. Accessed: 6 July 2025.