Design and Analysis of Algorithms (DAA)

Rupak R. Gupta (RRG)
Aaditya Joil (Gojo)

2024

Contents

Chapter 1

Introduction to Analysis of Algorithms

1.1 Algorithms

Informally, an **algorithm** is any well-defined computational procedure that solves a problem.

Every algorithm must satisfy the following properties:

Definiteness Every step in an algorithm must be clear and unambiguous.

Finiteness Every algorithm must produce a result within a finite number of steps.

Effectiveness Every instruction must be executed in a finite amount of time.

Input and Output Every algorithm must take zero or more number of inputs and must produce at least one output as result.

Correctness The proposed algorithm should produce correct and unambiguous results.

1.2 Performance of an Algorithm

The performance of an algorithm can be measured by the metrics of time and space complexities.

1.2.1 Time Complexity

The **time complexity** is the computational complexity that describes the amount of computer time it takes to run an algorithm. It is calculated by assuming that each elementary operation takes a constant amount of time and finding out the total number of elementary operations. It is usually stated as a function of the size (n) of the input.

1.2.2 Space Complexity

The **space complexity** of an algorithm or a data structure is the amount of computer memory required to solve an instance of the computational problem. It is the memory required by an algorithm until it executes completely and contains the input space as well as the auxiliary space. Like *time complexity* it also is represented as a function of the size (n) of the input.

1.3 Asymptotic Notations

The time and space complexity varies drastically from one algorithm to another, even varying for the same algorithm with different inputs. So we represent the functions in terms of the rate at which they grow. This representation is known as *asymptotic notation*[1].

1.3.1 $\mathcal{O}$-Notation

$$\mathcal{O}(g(n)) \coloneqq \{f(n) \,|\, \exists\, c > 0\ \exists\, n_0 > 0\ \forall\, n \geq n_0 \,:\, 0 \leq f(n) \leq cg(n)\} \tag{1.1}$$

It provides an asymptotic upper bound on the rate of growth of a function. If a function $f(n)$ is $\mathcal{O}(g(n))$ it means that $f(n)$ grows at a rate that is at most as fast as $g(n)$ as $n \to \infty$.

Q.1. Find the $\mathcal{O}$-notation for $f(n) \coloneqq 6n^3 + n^2 + 3n + 3$.

$f(n) \coloneqq 6n^3 + n^2 + 3n + 3$

$$\begin{aligned}
6n^3 + n^2 + 3n + 3 &\leq 6n^3 + n^2 + 3n + n && \forall\, n \geq 3 \\
&\leq 6n^3 + n^2 + 4n \\
6n^3 + n^2 + 4n &\leq 6n^3 + n^2 + n^2 && \forall\, n \geq 4 \\
&\leq 6n^3 + 2n^2 \\
6n^3 + 2n^2 &\leq 6n^3 + n^3 && \forall\, n\,(n^3 \geq 2n^2) \text{ or } \forall\, n \geq 2 \\
&\leq 7n^3 \\
\therefore f(n) &\leq 7n^3 && (\because (a \leq b) \wedge (b \leq c) \implies (a \leq c))
\end{aligned}$$

Thus, for all $n \geq 2$, $f(n) \leq cn^3$, with $c = 7$. Thus, $\boxed{f(n) \in \mathcal{O}(n^3)}$. **(Ans.)**

o-Notation

$$o(g(n)) \coloneqq \{f(n) \,|\, \forall\, c > 0\ \exists\, n_0 > 0\ \forall\, n \geq n_0 \,:\, 0 \leq f(n) < cg(n)\} \tag{1.2}$$

o-notation is defined in a similar way to $\mathcal{O}$-notation, but it provides a stricter upper bound for the asymptotic growth of a function. If a function $f(n)$ is $o(g(n))$ it means that $f(n)$ is guaranteed to grow at a lesser rate as compared to $g(n)$ as $n \to \infty$. The following equation holds true if $f(n) \in o(g(n))$:

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

1.3.2 Ω-Notation

$$\Omega(g(n)) \coloneqq \{f(n) \,|\, \exists\, c > 0\ \exists\, n_0 > 0\ \forall\, n \geq n_0 \,:\, 0 \leq cg(n) \leq f(n)\} \tag{1.3}$$

It provides an asymptotic lower bound for the rate of growth of a function. If a function $f(n)$ is defined $\Omega(g(n))$ it means that $f(n)$ grows at a rate that is at least as slow as $g(n)$ as $n \to \infty$.

Q.2. Find out Ω-notation for the function $4n^3 + 2n + 8$.

If given $f(n)$ is $\Omega(g(n))$ then: $c \cdot g(n) \leq f(n)$.

[1] We use this term because we evaluate the rate of growth as $n \to \infty$.

Let $c = 4$ and $g(n) = n^3$

$$\begin{aligned}
&\therefore c \cdot g(n) \leq f(n) \\
&\therefore 4n^3 \leq 4n^3 + 2n + 8 \\
&\therefore 0 \leq 2n + 8
\end{aligned}$$

which is true $\forall n (n \geq 0)$. Thus, $\boxed{f(n) \in \Omega(n^3)}$. **(Ans.)**

Q.3. Find the Ω-notation for $f(n) \coloneqq 4 \cdot 2^n + 3n$.

If given $f(n)$ is $\Omega(g(n))$ then: $c \cdot g(n) \leq f(n)$.

Let $c \in (0, 4]$ and $g(n) = 2^n$.

$$\begin{aligned}
&\therefore c \cdot g(n) \leq f(n) \\
&\therefore c \cdot 2^n \leq 4 \cdot 2^n + 3n \\
&\therefore (c - 4)2^n \leq 3n
\end{aligned}$$

which is true $\forall n (n \geq 0)$. Thus, $\boxed{f(n) \in \Omega(2^n)}$. **(Ans.)**

ω-Notation

$$\omega(g(n)) \coloneqq \{ f(n) \,|\, \forall\ c > 0\ \exists\ n_0 > 0\ \forall\ n \geq n_0\ :\ 0 \leq cg(n) < f(n) \} \tag{1.4}$$

ω-notation is defined in a similar way to Ω-notation, but it provides a stricter lower bound for the asymptotic growth of a function. If a function $f(n)$ is $\omega(g(n))$ it means that $f(n)$ is guaranteed to grow at a greater rate as compared to $g(n)$ as $n \to \infty$. The following equation holds true if $f(n) \in \omega(g(n))$:

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

It is important to note that o and ω are complementary in nature:

$$\phi(n) \in o(\psi(n)) \iff \psi(n) \in \omega(\phi(n)) \tag{1.5}$$

1.3.3 Θ-Notation

$$\Theta(g(n)) \coloneqq \{ f(n) \,|\, \exists\ c_1, c_2 > 0\ \exists\ n_0 > 0\ \forall\ n \geq n_0\ :\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \} \tag{1.6}$$

It provides asymptotic upper and lower bounds for the rate for growth of a function. If a function $f(n)$ is defined $\Theta(g(n))$ it means that $f(n)$ grows at a rate that is at most as fast as AND at least as slow as $g(n)$ as $n \to \infty$.

Another definition for Θ-notation is as follows:

$$\Theta(g(n)) \coloneqq \{ f(n) \mid f(n) \in \mathcal{O}(g(n)) \land f(n) \in \Omega(g(n)) \} = \mathcal{O}(g(n)) \cap \Omega(g(n)) \tag{1.7}$$

Q.4. Find Θ-notation for $f(n) \coloneqq 27n^2 + 16n$.

If a function $f(n)$ is defined as $\Theta(g(n))$ then: $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Let $c_1 \in (0, 27]$, $c_2 > 27$ and $g(n) = n^2$

$$\begin{aligned}
&\therefore c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \\
&\therefore c_1 \cdot n^2 \leq 27n^2 + 16n \leq c_2 \cdot n^2 \\
&\therefore (c_1 - 27)n^2 \leq 16n \leq (c_2 - 27)n^2
\end{aligned}$$

The first inequality $(c_1 - 27)n^2 \leq 16n$ holds true $\forall\ n \geq 0$.
The second inequality $16n \leq (c_2 - 27)n^2$ can be further evaluated as follows:

$$\begin{aligned}
&\therefore 16n \leq (c_2 - 27)n^2 \\
&\therefore n((c_2 - 27)n - 16) \geq 0 \\
&\therefore n \in \left[\frac{16}{c_2 - 27}, \infty\right)
\end{aligned}$$

Thus, for $c_1 \in (0, 27]\,,\ c_2 > 27,\ n_0 > \dfrac{16}{c_2 - 27}$ and $g(n) = n^2$, $\boxed{f(n) \in \Theta\big(n^2\big)}$. **(Ans.)**

Q.5. Find Θ-notation for $f(n) := 5n^3 + n^2 + 3n + 2$.

If a function $f(n)$ is defined as $\Theta(g(n))$ then: $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Let us find the value of $g(n)$ for the first inequality $c_1 \cdot g(n) \leq f(n)$. Let $c_1 \in (0, 5]$ and $g(n) = n^3$.

$$\begin{aligned}
&\therefore c_1 \cdot g(n) \leq f(n) \\
&\therefore c_1 \cdot n^3 \leq 5n^3 + n^2 + 3n + 2 \\
&\therefore (c_1 - 5)n^3 \leq n^2 + 3n + 2
\end{aligned}$$

which is true $\forall\ n \geq 0$. Thus, $f(n) \in \Omega\big(n^3\big)$.

Now let us find the value for $g(n)$ for the second inequality $f(n) \leq c_2 \cdot g(n)$.

$$\begin{aligned}
5n^3 + n^2 + 3n + 2 &\leq 5n^3 + n^2 + 3n + n && \forall n (n \geq 2) \\
&\leq 5n^3 + n^2 + 4n \\
5n^3 + n^2 + 4n &\leq 5n^3 + n^2 + n^2 && \forall n (n \geq 4) \\
&\leq 5n^3 + 2n^2 \\
5n^3 + 2n^2 &\leq 5n^3 + n^3 && \forall n (n^3 \geq 2n^2) \implies \forall n (n \geq 2) \\
&\leq 6n^3 \\
\therefore f(n) &\leq 6n^3 && (\because (a < b) \land (b < c) \implies (a < c))
\end{aligned}$$

Thus, for all $n > 2,\ f(n) < cn^3$, with $c = 6$. Thus, $f(n) \in \mathcal{O}\big(n^3\big)$.

Since $f(n) \in \Omega\big(n^3\big)$ and $f(n) \in \mathcal{O}\big(n^3\big)$, $\boxed{f(n) \in \Theta\big(n^3\big)}$. **(Ans.)**

1.4 Solving Recurrences

Recurrence relations are relations of an indexed variable which involve the presence of the same variable but with a lesser index. The elements of such a relation form sequences, conversely a recurrence relation may also be used to denote a sequence.

A recurrence relation of an indexed variable a_k is defined as follows: $a_k := \begin{cases} f(a_i) & k > i > k_0 \\ c_0 & k = k_0 \end{cases}$

There are various different methods to solving a recurrence relation, ket us have a look at some of them.

1.4.1 Substitution Method

The first method of solving a recurrence relation is to constantly substitute the “lesser” version of the variable recursively until we reach the base case.

e.g. Suppose we have been given a recurrence relation as follows:

$$a_k = \begin{cases} 3a_{k-1} + 1 & \forall k > 1 \\ 2 & k = 1 \end{cases}$$

To solve this recurrence relation, we first write the relation: $a_k = 3a_{k-1} + 1$.
Now upon constant backtracking of the a_{k-1} term:

$$\begin{aligned} a_k &= 3a_{k-1} + 1 \\ &= 3(3a_{k-2} + 1) + 1 = 9a_{k-2} + 1 + 3 \\ &= 9(3a_{k-3} + 1) + 1 + 3 = 27a_{k-3} + 1 + 3 + 9 \\ a_k &= 3^i a_{k-i} + \textstyle\sum_{r=1}^{i} 3^{r-1} \end{aligned} \quad \textit{Noticing a pattern}$$

We need to perform this operation until we reach the base case, i.e., the term a_1 appears in the above equation, this occurs when the value of $k - i = 1 \implies i = k - 1$.

$$\begin{aligned} a_k &= 3^{k-1} a_1 + \textstyle\sum_{r=1}^{k-1} 3^{r-1} \\ &= 3^{k-1}2 + (1 + 3 + 3^2 + \cdots + 3^{k-2}) \\ &= 3^{k-1}2 + 1 \cdot \frac{3^{k-1} - 1}{3 - 1} \\ a_k &= 3^{k-1}2 + \frac{3^{k-1} - 1}{2} \end{aligned} \quad \textit{k-1 terms in the sum}$$

The recurrence relation equates out to $\boxed{a_k = 3^{k-1}2 + \frac{3^{k-1} - 1}{2}}$. **(Ans.)**

e.g. Suppose we have been given a recurrence relation as follows:

$$a_k = \begin{cases} a_{k/2}/2 & k = 2^n > 1 \\ 1 & k = 1 \end{cases}$$

To solve this recurrence relation, we first write the relation: $a_k = a_{k/2}/2$.

Now upon constant backtracking of the $a_{k/2}$ term:

$$\begin{aligned} a_k &= a_{k/2}/2 \\ &= a_{k/4}/4 \\ &= a_{k/8}/8 \\ a_k &= a_{k/2^i}/2^i \end{aligned} \quad \textit{Noticing a pattern}$$

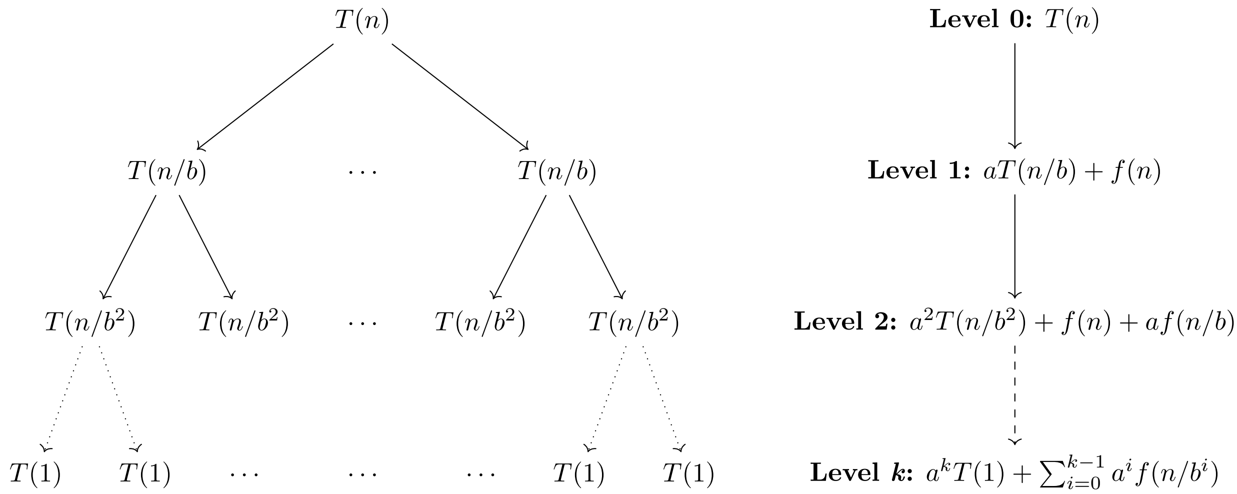
We need to perform this operation until we reach the base case, i.e., the term a_1 appears in the above equation, this occurs when the value of $k/2^i = 1 \implies i = \lg k$.

$$\begin{aligned} a_k &= a_1/2^{\lg k} \\ &= a_1/k \\ &= 1/k \end{aligned}$$

The recurrence relation equates out to $\boxed{a_k = 1/k}$. **(Ans.)**

1.4.2 Recursion Tree

Consider the following recurrence relation; $T(n) = \begin{cases} aT(n/b) + f(n) & n > 1 \\ 1 & n = 1 \end{cases}$



where $k = \log_b n$, this is derived from the fact that at the k^{th} level, the value of n/b^k becomes 1.

1.4.3 Master's Theorem

From the above recursion tree, it follows that:

$$T(n) = n^{\log_b a} \cdot T(1) + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \tag{1.8}$$

Depending on the value of $a,\ b$ and $f(n)$, a few results have been already derived. They are as follows:

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & f(n) \in \mathcal{O}(n^{\log_b(a)-\varepsilon}) \\ \Theta(n^{\log_b a}\log_b^{k+1} n) & f(n) \in \Theta(n^{log_b(a)}\log_b^k n) \\ \Theta(f(n)) & f(n) \in \Omega(n^{\log_b(a)+\varepsilon}) \wedge c_{reg} \end{cases}$$

where c_{reg} is the regularity condition given as $af(n/b) \leq cf(n)$; where $c < 1$

Derivation of Master's Theorem: Case 1

From eq. (1.8), $T(n) = n^{\log_b a} + \sum_{i=0}^{\log_b n-1} a^i f(n/b^i)$

$$\because f(n) \in \mathcal{O}(n^{\log_b(a)-\varepsilon})$$

$$\begin{aligned}
\therefore \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) &\leq c_1 \sum_{i=0}^{\log_b n-1} a^i (n/b^i)^{\log_b a-\varepsilon} \\
&\leq c_1 n^{\log_b(a)-\varepsilon} \sum_{i=0}^{\log_b n-1} \frac{a^i}{b^{i(\log_b(a)-\varepsilon)}} \\
&\leq c_1 n^{\log_b(a)-\varepsilon} \sum_{i=0}^{\log_b n-1} \frac{a^i b^{i\varepsilon}}{(b^{\log_b(a)})^i} \\
&\leq c_1 n^{\log_b(a)-\varepsilon} \sum_{i=0}^{\log_b n-1} \frac{a^i b^{i\varepsilon}}{a^i} \\
&\leq c_1 n^{\log_b(a)-\varepsilon} \sum_{i=0}^{\log_b n-1} b^{i\varepsilon} \\
&\leq c_1 n^{\log_b(a)-\varepsilon} \cdot \frac{b^{\varepsilon(\log_b n)} - 1}{b^\varepsilon - 1} \\
\therefore \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) &\leq c_1 n^{\log_b(a)-\varepsilon} \frac{n^\varepsilon - 1}{b^\varepsilon - 1}
\end{aligned}$$

$$\because c_1 n^{\log_b(a)-\varepsilon} \frac{n^\varepsilon - 1}{b^\varepsilon - 1} \leq c_1 n^{\log_b(a)-\varepsilon} \frac{n^\varepsilon}{b^\varepsilon - 1} = \frac{c_1}{b^\varepsilon - 1} \cdot n^{\log_b(a)}$$

$$\therefore T(n) \leq \frac{c_1}{b^\varepsilon - 1} \cdot n^{\log_b(a)} + n^{\log_b a}$$

$$\therefore T(n) \in \mathcal{O}(n^{\log_b a})$$

Derivation of Master's Theorem: Case 2

From eq. (1.8), $T(n) = n^{\log_b^a} + \sum_{i=0}^{\log_b n-1} a^i f(n/b^i)$

$\because f(n) \in \Theta(n^{\log_b a} \log_b^k n)$

$\therefore c_1 \sum_{i=0}^{\log_b n-1} a^i (n/b^i)^{\log_b a} \log_b^k (n/b) \leq \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \leq c_2 \sum_{i=0}^{\log_b n-1} a^i (n/b^i)^{\log_b a} \log_b^k (n/b)$

$\therefore c_1 n^{\log_b a} \log_b^k (n/b) \sum_{i=0}^{\log_b n-1} \frac{a^i}{b^{i \log_b a}} \leq \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \leq c_2 n^{\log_b a} \log_b^k (n/b) \sum_{i=0}^{\log_b n-1} \frac{a^i}{b^{i \log_b a}}$

$\therefore c_1 n^{\log_b a} \left(\log_b^k n - \log_b^k b\right) \sum_{i=0}^{\log_b n-1} \frac{a^i}{(b^{\log_b a})^i} \leq \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \leq c_2 n^{\log_b a} \left(\log_b^k n - \log_b^k b\right) \sum_{i=0}^{\log_b n-1} \frac{a^i}{(b^{\log_b a})^i}$

$\therefore c_1 n^{\log_b a} \left(\log_b^k n - 1\right) \sum_{i=0}^{\log_b n-1} (a^i/a^i) \leq \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \leq c_2 n^{\log_b a} \left(\log_b^k n - 1\right) \sum_{i=0}^{\log_b n-1} (a^i/a^i)$

$\therefore c_1 n^{\log_b a} \left(\log_b^k n - 1\right) \sum_{i=0}^{\log_b n-1} 1 \leq \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \leq c_2 n^{\log_b a} \left(\log_b^k n - 1\right) \sum_{i=0}^{\log_b n-1} 1$

$\therefore c_1 n^{\log_b a} \left(\log_b^{k+1} n - \log_b n\right) \leq \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \leq c_2 n^{\log_b a} \left(\log_b^{k+1} n - \log_b n\right)$

$\therefore n^{\log_b a} + c_1 n^{\log_b a} \left(\log_b^{k+1} n - \log_b n\right) \leq n^{\log_b a} + \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \leq n^{\log_b a} + c_2 n^{\log_b a} \left(\log_b^{k+1} n - \log_b n\right)$

$\therefore n^{\log_b a} + c_1 n^{\log_b a} \left(\log_b^{k+1} n - \log_b n\right) \leq T(n) \leq n^{\log_b a} + c_2 n^{\log_b a} \left(\log_b^{k+1} n - \log_b n\right)$

$\therefore T(n) \in \Theta\left(n^{\log_b a} \log_b^{k+1} n\right)$

In most cases, the value of k is 0. Thus, $T(n) \in \Theta\left(n^{\log_b a} \log_b n\right)$

Derivation of Master's Theorem: Case 3

From eq. (1.8), $T(n) = n^{\log_b a} + \sum_{i=0}^{\log_b n-1} a^i f(n/b^i)$
And thus we prove, $T(n) \in \Omega\left(f(n)\right)$.

1.5 Algorithm Design Techniques

1.5.1 Implementation

- Recursion or Iteration
- Procedural or Declarative
- Serial or Parallel
- Deterministic or Non-Deterministic
- Exact (optimal) or Approximate ($\mathcal{NP}$-Hard)

1.5.2 Design

- Greedy
- Divide and Conquer

- Dynamic Programming (DP)
- Linear Programming
- Reduction

1.5.3 Other Classifications

- By Research Area (Search, Sort, String, Graph)
- By Complexity
- Randomized Algorithm
- Branch and Bound and Backtracking

1.6 Sorting Algorithms

1.6.1 Selection Sort

Selection sort is a sorting algorithm that sorts an array by repeatedly swapping the minimum element in the unsorted part with the first unsorted element.

For example, consider sorting the following array in ascending order:

$$[11, \quad 9, \quad 5, \quad 7, \quad 4]$$

The sorted array must look like

$$[4, \quad 5, \quad 7, \quad 9, \quad 11]$$

Here is how the selection sort algorithm works for sorting an array in ascending order:

Pass 1 We claim that the minimum element is 11. We will search for any element less than 11 in the rest of the array, and if it exists, swap it and 11.

$$\underbrace{11 \quad 9}_{\times} \quad 5 \quad 7 \quad 4 \qquad \text{Minimum: } 9$$

$$11 \quad \underbrace{9 \quad 5}_{\times} \quad 7 \quad 4 \qquad \text{Minimum: } 5$$

$$11 \quad 9 \quad \underbrace{5 \quad 7}_{\checkmark} \quad 4 \qquad \text{Minimum: } 5$$

$$11 \quad 9 \quad \underbrace{5 \quad 7 \quad 4}_{\times} \qquad \text{Minimum: } 4$$

Thus, the array obtained after the first pass is $[4, 9, 5, 7, 11]$.

Pass 2 We start from the first element of the unsorted array, *i.e.* 9 and claim it as the minimum. We shall then search for any element in the unsorted that may be lesser than 9, and swap it with 9.

$$\mathbf{4} \quad \underbrace{9 \quad 5}_{\times} \quad 7 \quad 11 \qquad\qquad \text{Minimum: } 5$$

$$\mathbf{4} \quad 9 \quad \underbrace{5 \quad 7}_{\checkmark} \quad 11 \qquad\qquad \text{Minimum: } 5$$

$$\mathbf{4} \quad 9 \quad \underbrace{5 \quad 7 \quad 11}_{\checkmark} \qquad\qquad \text{Minimum: } 5$$

Thus, the array obtained after the second pass is $[4, 5, 9, 7, 11]$.

Pass 3

$$\mathbf{4} \quad \mathbf{5} \quad \underbrace{9 \quad 7}_{\times} \quad 11 \qquad\qquad \text{Minimum: } 7$$

$$\mathbf{4} \quad \mathbf{5} \quad 9 \quad \underbrace{7 \quad 11}_{\checkmark} \qquad\qquad \text{Minimum: } 7$$

Thus, the array obtained after the third pass is $[4, 5, 7, 9, 11]$[2].

Pass 4

$$\mathbf{4} \quad \mathbf{5} \quad \mathbf{7} \quad \underbrace{9 \quad 11}_{\checkmark} \qquad\qquad \text{Minimum: } 9$$

Thus, the array obtained after the fourth pass is $[4, 5, 7, 9, 11]$.

Therefore, the final array obtained after selection sort is $\boxed{[4, 5, 7, 9, 11]}$.

Algorithm 1: Selection Sort Algorithm

Input: Array A of n elements

Output: Array A sorted in ascending order

```
for i = 1 to n − 1 do
    // Assume the ith element is the minimum
    min_index ← i;
    for j = i + 1 to n do
        if A[j] < A[min_index] then
            min_index ← j;
        end
    end
    // Swap the found minimum element with the ith element
    if min_index ≠ i then
        temp ← A[i];
        A[i] ← A[min_index];
        A[min_index] ←temp;
    end
end
```

The time complexity of selection sort is $\mathcal{O}(n^2)$. The space complexity of this algorithm is $\mathcal{O}(1)$.

[2] Although the array appears to be sorted, the algorithm still needs to complete all iterations.

1.6.2 Insertion Sort

Insertion sort is a sorting algorithm that sorts an array by trying to "insert" a certain element in its correct place by comparing it with the all elements that are before(or after) it.

For example, consider sorting the following array in ascending order:

$$[11, \qquad 9, \qquad 5, \qquad 7, \qquad 4]$$

The sorted array must look like

$$[4, \qquad 5, \qquad 7, \qquad 9, \qquad 11]$$

Here is how the insertion sort algorithm works for sorting an array in ascending order:

Pass 1

$$\underbrace{\mathbf{11} \quad 9}_{\times} \quad 5 \quad 7 \quad 4 \qquad \text{Shift}$$

Thus, the array obtained after the first pass is $[9, 11, 5, 7, 4]$.

Pass 2

$$\mathbf{9} \quad \underbrace{\mathbf{11} \quad 5}_{\times} \quad 7 \quad 4 \qquad \text{Shift}$$

$$\underbrace{\mathbf{9} \quad 5}_{\times} \quad \mathbf{11} \quad 7 \quad 4 \qquad \text{Shift}$$

Thus, the array obtained after the second pass is $[5, 9, 11, 7, 4]$.

Pass 3

$$\mathbf{5} \quad \mathbf{9} \quad \underbrace{\mathbf{11} \quad 7}_{\times} \quad 4 \qquad \text{Shift}$$

$$\mathbf{5} \quad \underbrace{\mathbf{9} \quad 7}_{\times} \quad \mathbf{11} \quad 4 \qquad \text{Shift}$$

$$\underbrace{\mathbf{5} \quad 7}_{\checkmark} \quad \mathbf{9} \quad \mathbf{11} \quad 4 \qquad \text{Do not shift}$$

Thus, the array obtained after the third pass is $[5, 7, 9, 11, 4]$.

Pass 4

$$\mathbf{5} \quad \mathbf{7} \quad \mathbf{9} \quad \underbrace{\mathbf{11} \quad 4}_{\times} \qquad \text{Shift}$$

$$\mathbf{5} \quad \mathbf{7} \quad \underbrace{\mathbf{9} \quad 4}_{\times} \quad \mathbf{11} \qquad \text{Shift}$$

$$\mathbf{5} \quad \underbrace{\mathbf{7} \quad 4}_{\times} \quad \mathbf{9} \quad \mathbf{11} \qquad \text{Shift}$$

$$\underbrace{\mathbf{5} \quad 4}_{\times} \quad \mathbf{7} \quad \mathbf{9} \quad \mathbf{11} \qquad \text{Shift}$$

Thus, the array obtained after the fourth pass is $[4, 5, 7, 9, 11]$.

Therefore, the final array obtained after insertion sort is $\boxed{[4, 5, 7, 9, 11]}$.

The best-case time complexity of insertion sort is $\mathcal{O}(n)$ and worst-case time complexity is $\mathcal{O}(n^2)$. The space complexity of this algorithm is $\mathcal{O}(1)$.

Algorithm 2: Insertion Sort Algorithm

Input: Array A of n elements

Output: Array A sorted in ascending order

```
for i = 2 to n do
    key ← A[i];
    j ← i − 1;
    // Shift elements of A[1...i − 1] that are greater than key to one
       position ahead
    while j > 0 and A[j] > key do
        A[j + 1] ← A[j];
        j ← j − 1;
    end
    // Insert the key at the correct position
    A[j + 1] ← key;
end
```

1.6.3 Heap Sort

Heap sort is a sorting algorithm that sorts an array by making use of the characteristic property of the heap data structure. We repeatedly find the maximum element of an array, place it at the end and consider the final elements to be sorted.

Heaps

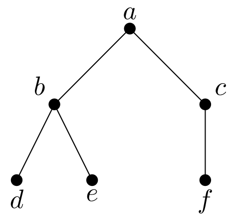
A heap is a special type of binary tree which has the characteristic property that the children of a particular node have values which are *lesser* than the value of their parent. Such a heap is called a *max-heap* since the maximum element is at the root.

A *min-heap* is a heap where the children have values which are greater than the value of the parent.

We will almost talk exclusively about *max-heaps* in this section until specified otherwise.

Relation between Arrays and Trees

A tree can be represented using a single array, this is done by making clever use of indices of the array. The root of the tree is the first element in the array, all the subsequent children of a node at index i are located at the $(2i)^{\text{th}}$ and $(2i+1)^{\text{th}}$ positions in the array.



$[a$	b	c	d	e	$f]$
1	2	3	4	5	6

Algorithm 3: Build Max-Heap

Input: Array A of length n

Output: A max-heap represented in the array A

$n \leftarrow$ length(A);
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
  Max-Heapify(A, i);
end

Algorithm 4: Max-Heapify

Input: Array A, index i

Output: The subtree rooted at index i is a max-heap

$l \leftarrow 2i$ `// left child`
$r \leftarrow 2i + 1$ `// right child`
if $l \leq$ *heap-size*(A) ***and*** $A[l] > A[i]$ **then**
  $largest \leftarrow l$;
end
else
  $largest \leftarrow i$;
end
if $r \leq$ *heap-size*(A) ***and*** $A[r] > A[largest]$ **then**
  $largest \leftarrow r$;
end
if $largest \neq i$ **then**
  Swap $A[i]$ and $A[largest]$;
  Max-Heapify($A, largest$);
end

Algorithm 5: Heap Sort

Input: Array A of length n

Output: Sorted array A

Build Max-Heap(A);
$n \leftarrow$ length(A);
for $i \leftarrow n$ **downto** 2 **do**
  Swap $A[1]$ with $A[i]$;
  heap-size(A) $\leftarrow$ heap-size(A) $- 1$;
  Max-Heapify($A, 1$);
end

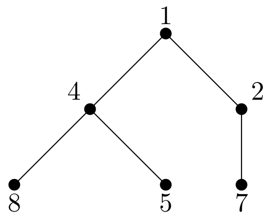
For example, consider sorting the following array in ascending order:

$$[1, \quad 4, \quad 2, \quad 8, \quad 5, \quad 7]$$

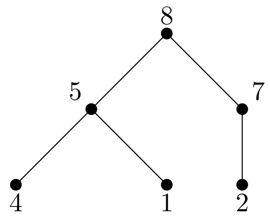
The sorted array must look like

$$[1, \quad 2, \quad 4, \quad 5, \quad 7, \quad 8]$$

Here is how the heap sort algorithm works for sorting an array in ascending order:
Build the Heap

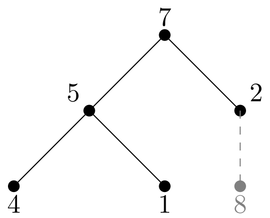


[1, 4, 2, 8, 5, 7]



[8, 5, 7, 4, 1, 2]

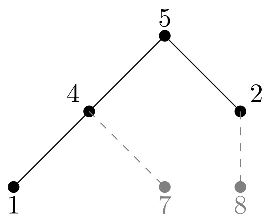
Pass 1



[7, 5, 2, 4, 1, **8**]

Thus, the array obtained after the first pass is $[7, 5, 2, 4, 1, 8]$.

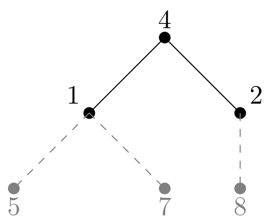
Pass 2



[5, 4, 2, 1, **7**, **8**]

Thus, the array obtained after the second pass is $[5, 4, 2, 1, 7, 8]$.

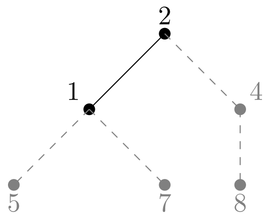
Pass 3



[4, 1, 2, **5**, **7**, **8**]

Thus, the array obtained after the third pass is $[4, 1, 2, 5, 7, 8]$.

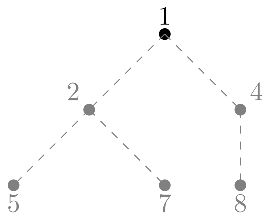
Pass 4



[2, 1, **4**, **5**, **7**, **8**]

Thus, the array obtained after the fourth pass is $[2, 1, 4, 5, 7, 8]$.

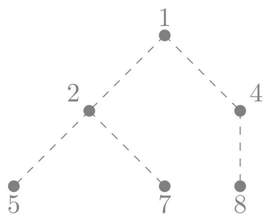
Pass 5



[1, **2**, **4**, **5**, **7**, **8**]

Thus, the array obtained after the fifth pass is $[1, 2, 4, 5, 7, 8]$.

Pass 6



[**1**, **2**, **4**, **5**, **7**, **8**]

Thus, the array obtained after the sixth pass is $[1, 2, 4, 5, 7, 8]$.

Therefore, the final array obtained after heap sort is $\boxed{[1, 2, 4, 5, 7, 8]}$.

The time complexity of heap sort is $\mathcal{O}(n \lg n)$. The space complexity of this algorithm is $\mathcal{O}(1)$.

Chapter 2

Graph Algorithms

2.1 Graphs

A graph $G = (V, E)$ is defined as a set of nodes or vertices V connected to each other by edges E where $E \subseteq V \times V$.

A graph can be:

- Directed or Undirected
- Cyclic or Acyclic
- Weighted or Unweighted
- Connected or Disconnected
- Bipartite
- Multigraphs
- Hierarchical (Trees)
- Unit-degree (Linked Lists)

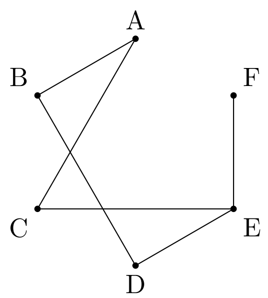


Figure 2.1: An unweighted undirected acyclic and connected graph

Graphs can be used in modelling real world networks such as electrical circuitry, motor roadways, computer networks, etc.

2.2 Elementary Graph Algorithms

Notation

An edge from vertex u to vertex v is denoted as the pair $(u, v) \in E$. For an undirected graph, $(u, v) \in E \iff (v, u) \in E$ (Holy crap symmetric relation).

The set of all vertices connected from vertex u is given by $Adj[u]$.

$$Adj[u] \coloneqq \{v \in V \mid (u, v) \in E\}$$

A vertex $v \in V$ might have certain *attributes*, which are denoted using the dot (.) notation.

- The current **parent** (previous vertex) of v is denoted as $v.\pi$.
- The current **shortest distance** of v from some source vertex is denoted as $v.d$.
- The current **color** (WHITE, GRAY or BLACK) of v relevant to a graph traversal is denoted as $v.color$.
- The **finishing time** of v in a graph traversal is denoted as $v.f$.
- The current **minimum weight** of any edge connecting v to any vertex in the graph is denoted as $v.key$.

For directed graphs, a weight function $w : E \to \mathbb{R}$ is used to map each edge to a weight.

If an edge does not exist, we can store a NIL value as its corresponding entry, though for many problems it is convenient to use a value such as 0 or ∞.

$$w(u, v) = \text{NIL} \iff (u, v) \notin E$$

2.2.1 Representation of Graphs

Consider the following graph:

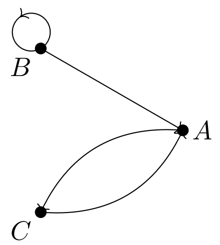


Figure 2.2: An unweighted directed cyclic and connected graph

Adjacency Matrices

An adjacency matrix $[a_{ij}]$ is a boolean matrix of order $|V| \times |V|$ defined as follows:

$$[a_{ij}] \coloneqq \begin{cases} 1, & \text{if } (i, j) \in E \\ 0, & \text{otherwise} \end{cases}$$

For example, the adjacency matrix of the graph in fig. 2.2 would be:

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

or

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

The space requirements of an adjacency matrix representation would vary as $\mathcal{O}\left(|V|^2\right)$.

Adjacency Lists

For an adjacency list representation, we maintain a linked list of vertices for each vertex that stores the vertices that are connected from that vertex. When a new edge is added to the graph, the destination vertex must be appended to the list corresponding to the source vertex.

For example, the adjacency list of the graph in fig. 2.2 would be:

$$\begin{aligned} \text{A} &: \boxed{\text{C}} \to \varnothing \\ \text{B} &: \boxed{\text{A}} \to \boxed{\text{B}} \to \varnothing \\ \text{C} &: \boxed{\text{A}} \to \varnothing \end{aligned}$$

The space requirements of an adjacency list representation would vary as $\mathcal{O}(|E| + |V|)$.

2.2.2 Depth-First Search

Depth-First Search (DFS)

Algorithm 6: Depth-First Search (DFS)

```
Input: Graph G = (V, E)
Output: Discovery and finishing times of all vertices
foreach vertex u ∈ V do
    u.color ← WHITE;
    u.π ← NIL;
time ← 0;
foreach vertex u ∈ V do
    if u.color = WHITE then
        DFS-Visit(u);
```

Algorithm 7: DFS-Visit(u)

```
u.color ← GRAY;
time ← time + 1;
u.d ← time;
foreach vertex v ∈ Adj[u] do
    if v.color = WHITE then
        v.π ← u;
        DFS-Visit(v);
u.color ← BLACK;
time ← time + 1;
u.f ← time;
```

2.2.3 Breadth-First Search

Breadth-First Search (BFS)

Algorithm 8: Breadth-First Search (BFS)

Input: Graph $G = (V, E)$, source vertex s

Output: Shortest path from s to all other vertices

```
foreach vertex u ∈ V \ {s} do
    u.color ← WHITE;
    u.d ← ∞;
    u.π ← NIL;
s.color ← GRAY;
s.d ← 0;
s.π ← NIL;
Q ← ∅;
Enqueue(Q, s);
while Q ≠ ∅ do
    u ← Dequeue(Q);
    foreach vertex v ∈ Adj[u] do
        if v.color = WHITE then
            v.color ← GRAY;
            v.d ← u.d + 1;
            v.π ← u;
            Enqueue(Q, v);
    u.color ← BLACK;
```

2.2.4 Topological Sorting

Algorithm 9: Topological Sort

Input: Directed Acyclic Graph (DAG) $G = (V, E)$

Output: A topological ordering of vertices

```
Perform DFS on G to compute finishing times u.f for each vertex u;
As each vertex is finished, insert it onto the front of a linked list;
return the linked list of vertices;
```

The time complexity of topological sorting is $\mathcal{O}(|V| + |E|)$. The space complexity of this algorithm is $\mathcal{O}(|V|)$.

2.2.5 Strongly Connected Components

A subgraph H of a graph G is said to be a Strongly Connected Component (SCC) iff for every pair of vertices u and v in $H.V$ v is *reachable* from u and u is *reachable* from v, *i.e.*,

$$H \text{ is an SCC of } G \iff H \subseteq G \land \forall u \in H.V \ \forall v \in H.V (u \rightsquigarrow v \land v \rightsquigarrow u).$$

We can determine the SCCs of a graph using **Kosaraju's algorithm**.

Algorithm 10: Kosaraju's Algorithm

Input: Graph $G = (V, E)$

Output: Strongly connected components of G

Call DFS on G to compute finishing times $u.f$ for each vertex u;
Compute $G^\top$ (the transpose of G);
Call DFS on $G^\top$, but in the main loop of DFS, consider vertices in order of decreasing $u.f$ (from the original DFS);
Each tree in the depth-first forest of $G^\top$ is a strongly connected component;

The time complexity of Kosaraju's algorithm is $\mathcal{O}(|V| + |E|)$. The space complexity of this algorithm is $\mathcal{O}(|V|)$.

2.2.6 Trees

An acyclic and connected graph is known as a tree. If a tree is directed, then the graph is known as a DAG.

Trees are used to represent hierarchical relationships, indicating that the parents "come first" before the children.

2.3 Spanning trees

Spanning trees of a graph are special trees which contain every node in the graph without forming any cycles.

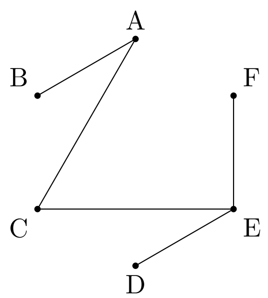


Figure 2.3: A spanning tree for the graph in fig. 2.1

A spanning tree will always have exactly $|V| - 1$ edges.

A **complete graph** will have $|V|^{|V|-2}$ spanning trees (Cayley's formula).

2.3.1 Minimum spanning trees

A Minimum Spanning Tree (MST) is a special spanning tree where the sum of the all the edges which have been included is the minimum possible for that particular graph.

For an unweighted, graph *any* spanning tree is a valid minimum spanning tree as well, as the total weight of the tree will always be $|V| - 1$ no matter what.

2.3.2 Generic algorithm

The generic algorithm to find the MST is as follows:

Algorithm 11: Generic MST

$T \leftarrow \varnothing$;
while T *does not form a spanning tree* **do**
    Find an edge (u, v) which is safe for T;
    $T \leftarrow T \cup \{(u, v)\}$;
return T;

A safe edge is an edge which maintains the loop invariant before and during the execution of the loop.

2.3.3 Kruskal's algorithm

This algorithm is an example of a greedy[1] algorithm as we include the edges the least possible weight without forming a cycle.

Algorithm 12: Kruskal's Algorithm

Input: Graph $G = (V, E)$, edge weights $w(e)$ for all $e \in E$
Output: A minimum spanning tree T
$T \leftarrow \varnothing$;
Sort the edges of E into non-decreasing order by weight w;
foreach *edge* (u, v) *in the sorted edge list* **do**
    if u *and* v *are in different components* **then**
        Add edge (u, v) to T;
        Union the sets containing u and v;
return T;

We make use of the *disjoint set* data structure to make sure that no cycles are formed at the inclusion of a new edge into the tree.

The best-case time complexity of Kruskal's algorithm is $\mathcal{O}(|E| \lg |E|)$ and the worst-case time-complexity is $\mathcal{O}(|E| \lg |V|)$. The space complexity of this algorithm is $\mathcal{O}(|E| + |V|)$.

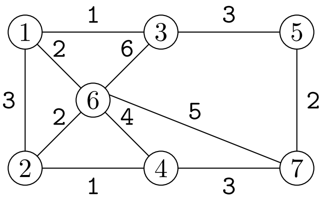
For example, consider finding the MST of the following weighted graph:

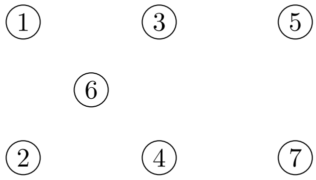
Initialise the tree

$$T = \{\}$$

Sort the edges

[1] Refer the chapter regarding greedy algorithms.



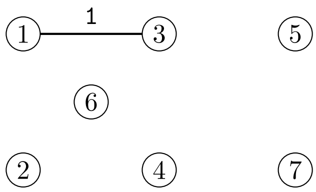


$$\underset{\text{weight}}{E} = \left\{\underset{1}{(1,3)}, \underset{1}{(2,4)}, \underset{2}{(1,6)}, \underset{2}{(2,6)}, \underset{2}{(5,7)}, \underset{3}{(1,2)}, \underset{3}{(3,5)}, \underset{3}{(4,7)}, \underset{4}{(4,6)}, \underset{5}{(6,7)}, \underset{6}{(3,6)}\right\}$$

Iterating

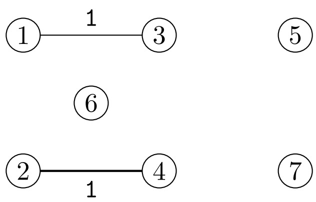
Pass 1

$$T = \{(1,3)\}$$



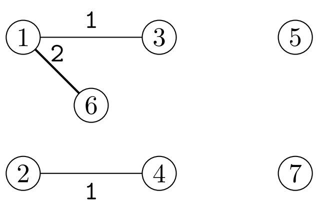
Pass 2

$$T = \{(1,3), (2,4)\}$$



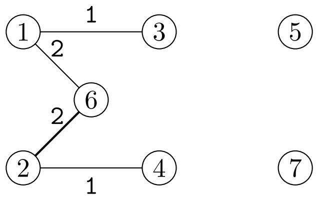
Pass 3

$$T = \{(1,3), (2,4), (1,6)\}$$



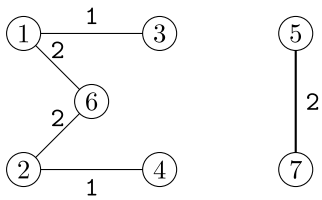
Pass 4

$$T = \{(1,3), (2,4), (1,6), (2,6)\}$$



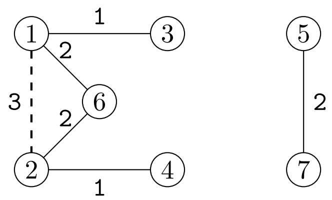
Pass 5

$$T = \{(1,3),(2,4),(1,6),(2,6),(5,7)\}$$



Pass 6

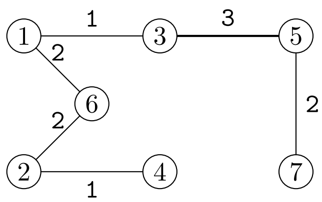
$$T = \{(1,3),(2,4),(1,6),(2,6),(5,7)\}$$



The edge $(1,2)$ forms a cycle so it is not included.

Pass 7

$$T = \{(1,3),(2,4),(1,6),(2,6),(5,7),(3,5)\}$$



$|T| = 6 = |V| - 1$. Therefore, the algorithm can stop now. Alternatively, we can say that all subsequent passes (**Passes 8, 9, 10, 11, 12**) lead to an edge which causes the formation of a cycle. Hence, we can safely say that $\boxed{\text{a spanning tree has been formed}}$.

The cost of the above MST is $1+2+2+1+3+2 = 11$. **(Ans.)**

2.3.4 Prim's Algorithm

Algorithm 13: Prim's Algorithm

Input: Graph $G = (V, E)$, edge weights $w(e)$ for all $e \in E$, starting vertex r

Output: A minimum spanning tree T

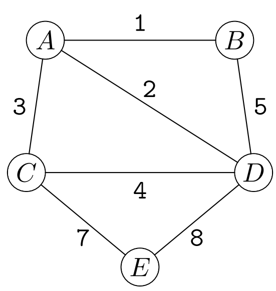
foreach *vertex* $u \in V$ **do**
    $u.key \leftarrow \infty$;
    $u.\pi \leftarrow$ NIL;
$r.key \leftarrow 0$;
$Q \leftarrow \varnothing$ `// Initialise the Priority Queue`
foreach *vertex* $u \in V$ **do**
    $Q \cdot \text{insert}(u)$;
while $Q \neq \varnothing$ **do**
    $u \leftarrow \text{Extract-Min}(Q)$;
    foreach *vertex* $v \in Adj[u]$ **do**
        if $v \in Q$ *and* $w(u, v) < v.key$ **then**
            $v.\pi \leftarrow u$;
            $v.key \leftarrow w(u, v)$;
return $\{(v, v.\pi) : v \in V, v.\pi \neq \text{NIL}\}$;

The following three-pointed loop invariant is followed for every iteration:

1. $A = \{(v, v.\pi) \mid v \in V \setminus (\{r\} \cup Q)\}$.
2. The vertices already placed into the minimum spanning tree are those in V/Q.
3. For all vertices $v \in Q$, $v \neq \text{NIL} \implies v.key < \infty \,\land\, v.key = w(e_0)$ where $e_0 = (v, v.\pi) \,|e_0 =$ light edge[2] connecting v to some vertex already placed into the minimum spanning tree.

The best-case as well as worst-case time complexity of Prim's algorithm is $\mathcal{O}(|E| \lg |V|)$. The space complexity of this algorithm is $\mathcal{O}(|E| + |V|)$.

For example, consider finding the MST of the following weighted graph:

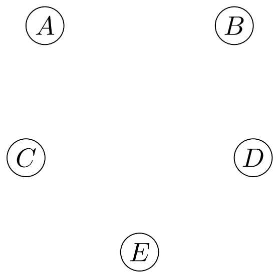


Assuming $r = A$

After inserting all elements in the priority queue Q

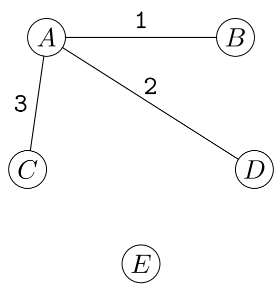
[2]Kabhi padhai bhi kar lo.

$$Q = \left[\underset{0}{A}, \quad \underset{\infty}{B}, \quad \underset{\infty}{C}, \quad \underset{\infty}{D}, \quad \underset{\infty}{E}\right]$$



After pass 1

$$Q = \left[\underset{1}{B}, \quad \underset{2}{D}, \quad \underset{3}{C}, \quad \underset{\infty}{E}\right]$$



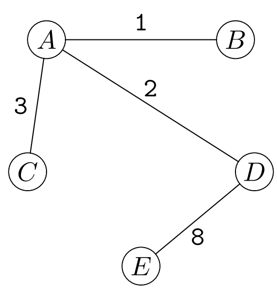
After pass 2

$$Q = \left[\underset{2}{D}, \quad \underset{3}{C}, \quad \underset{\infty}{E}\right]$$

No new additions to the tree as an edge of greater weight is being introduced.

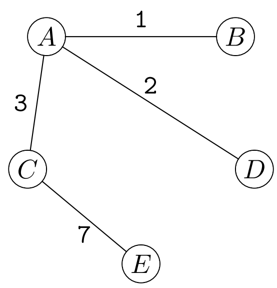
After pass 3

$$Q = \left[\underset{3}{C}, \quad \underset{8}{E}\right]$$



After pass 4

$$Q = \begin{bmatrix} E \\ {\scriptstyle 8} \end{bmatrix}$$



After pass 5

$$Q = \varnothing$$

No new additions to the tree as all elements in adjacency list of E are out of Q.

Since, the priority queue is empty, we can stop the algorithm.
$\boxed{\text{The MST has been formed successfully}}$. The cost of the above MST is $1 + 2 + 3 + 7 = 10$. **(Ans.)**

2.4 Shortest Path Algorithms

2.4.1 Single-Source Shortest Path Algorithm

Dijkstra's Algorithm

Algorithm 14: Dijkstra's Algorithm

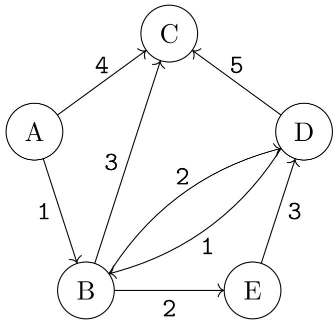
Input: Graph $G = (V, E)$, edge weights $w(e) \geq 0$, starting vertex s

Output: Shortest path distances from s to all vertices

foreach *vertex* $u \in V$ **do**
    $u.d \leftarrow \infty$;
    $u.\pi \leftarrow$ NIL;
$s.d \leftarrow 0$;
$Q \leftarrow \varnothing$ `// Initialise the Priority Queue`
foreach *vertex* $u \in V$ **do**
    Q·insert(u);
while $Q \neq \varnothing$ **do**
    $u \leftarrow$ Extract-Min(Q);
    foreach *vertex* $v \in Adj[u]$ **do**
        if $v.d > u.d + w(u, v)$ **then**
            $v.d \leftarrow u.d + w(u, v)$;
            $v.\pi \leftarrow u$;
            Q·Reduce-Key($v, v.d$);

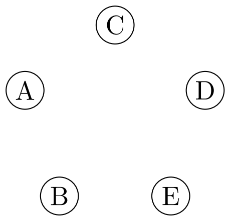
The best-case time complexity of Dijkstra's algorithm is $\mathcal{O}(|V| \lg |V| + |E| \lg |V|)$ and the worst-case time-complexity is $\mathcal{O}\left(|V|^2\right)$. The space complexity of this algorithm is $\mathcal{O}\left(|V^2|\right)$ or $\mathcal{O}(|E| + |V|)$ depending on the data structure used.

For example, consider applying Dijkstra's Algorithm on the following graph with the starting vertex $s = A$:



After inserting all elements in the priority queue Q

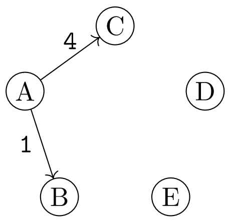
$$Q = \left[\underset{0}{A}, \quad \underset{\infty}{B}, \quad \underset{\infty}{C}, \quad \underset{\infty}{D}, \quad \underset{\infty}{E}\right]$$



Element	Distance	Parent
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

After pass 1

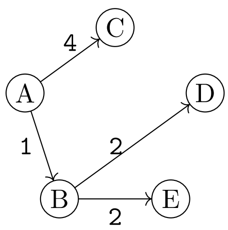
$$Q = \left[\underset{1}{B}, \quad \underset{4}{C}, \quad \underset{\infty}{D}, \quad \underset{\infty}{E}\right]$$



Element	Distance	Parent
A	0	NIL
B	1	A
C	4	A
D	∞	NIL
E	∞	NIL

After pass 2

$$Q = \left[\underset{3}{D}, \quad \underset{3}{E}, \quad \underset{4}{C}\right]$$



Element	Distance	Parent
A	0	NIL
B	1	A
C	4	A
D	3	B
E	3	B

After pass 3

$$Q = \left[\underset{3}{E}, \quad \underset{4}{C}\right]$$

No changes as the condition to perform relaxation does not occur.

After pass 4

$$Q = \begin{bmatrix} \underset{4}{C} \end{bmatrix}$$

No changes as the condition to perform relaxation does not occur.

After pass 5

$$Q = \varnothing$$

No changes as the adjacency list of C is empty.

Since, the priority queue is empty, we can stop the algorithm. The following table states the shortest distances from A to every other node.

Element	Distance	Parent
A	0	NIL
B	1	A
C	4	A
D	3	B
E	3	B

(Ans.)

Note

When there exist negative weight cycles, one can just travel along that cycle and get an even lesser distance. This can go on forever. Hence, ANY shortest distance algorithm is incorrect in such cases.

Bellman-Ford Algorithm

Algorithm 15: Bellman-Ford Algorithm

Input: Graph $G = (V, E)$, edge weights $w(e)$, starting vertex s

Output: Shortest path distances from s to all vertices, or report if a negative-weight cycle exists

foreach *vertex* $u \in V$ **do**
    $u.d \leftarrow \infty$;
    $u.\pi \leftarrow$ NIL;
$s.d \leftarrow 0$;
for $i = 1$ **to** $|V| - 1$ **do**
    foreach *edge* $(u, v) \in E$ **do**
        if $v.d > u.d + w(u, v)$ **then**
            $v.d \leftarrow u.d + w(u, v)$;
            $v.\pi \leftarrow u$;
foreach *edge* $(u, v) \in E$ **do**
    if $v.d > u.d + w(u, v)$ **then**
        return "Negative-weight cycle exists";

The best-case as well as worst-case time complexity of the Bellman-Ford algorithm is $\mathcal{O}(|E||V|)$. The space complexity of this algorithm is $\mathcal{O}(|V|)$.

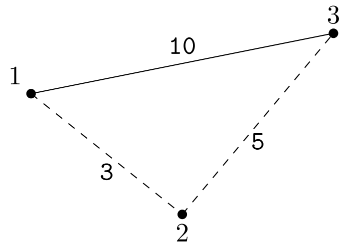
figure

2.4.2 All-Pairs Shortest Paths Algorithms

All-Pairs Shortest Paths Algorithm

All-Pairs Shortest Paths (APSP) algorithm.

Consider the ways to get to a certain node from a certain predefined node in a given graph.



In the above graph, the current path from 1 to 3 has a weight of **10**, with one edge being traversed. If we somehow increase the number of edges to 2, then we can go from 1 to 2 and then 2 to 3 and have a reduced path of **8** with 2 edges being traversed.

The above procedure is known as the "Extend-Shortest-Path" Procedure. The algorithm for it is as follows:

Algorithm 16: Extend-Shortest-Path

Input: $L^{(r)}$=Matrix of shortest paths with r edges, W=Weight matrix containing the weight of all the edges

Output: $L^{(r+1)}$=Matrix of shortest paths with r edges

if $l_{ij}^{(r)} < l_{ik}^{(r)} + w_{kj}$ **then**

  $l_{ij}^{(r+1)} \leftarrow l_{ik}^{(r)} + w_{kj}$;

  `// Extend the shortest path`

else

  $l_{ij}^{(r+1)} \leftarrow l_{ij}^{(r)}$;

  `// Keep the same path`

We repeatedly extend the shortest path for $|V|$ times. As the maximum number of edges we need to cross to get to a particular node is $|V|-1$. The last extension is performed to make sure that we have covered self paths as well.

Algorithm 17: APSP

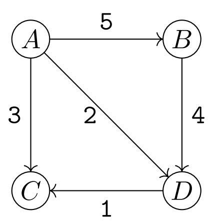
Input: Weight Matrix W

Output: M = The shortest path matrix containing where m_{ij} is the weight of the shortest path from i to j

$L \leftarrow \varnothing, M \leftarrow \varnothing$;
for $i = 1$ **to** $|V|$ **do**
    for $j = 1$ **to** $|V|$ **do**
        $l_{ij} \leftarrow 0$ **if** $(i = j)$ **else** ∞;
        `// Initialise` $L^{(0)}$
for $r = 1$ **to** $|V|$ **do**
    for $i = 1$ **to** $|V|$ **do**
        for $j = 1$ **to** $|V|$ **do**
            for $k = 1$ **to** $|V|$ **do**
                $M \leftarrow$Extend-Shortest-path(L, W);
                `// Alternatively` $m_{ij} \leftarrow \min(l_{ij}, l_{ik} + w_{kj})$
    $L \leftarrow M$;
return M;

The time complexity of the APSP algorithm is $\mathcal{O}\left(|V|^4\right)$. The space complexity of this algorithm is $\mathcal{O}\left(|V|^2\right)$.

Consider we have to apply the APSP algorithm on the following graph:



$$W = \begin{pmatrix} \infty & 5 & 3 & 2 \\ \infty & \infty & \infty & 4 \\ \infty & \infty & \infty & \infty \\ \infty & \infty & 1 & \infty \end{pmatrix}$$

Before Iterating

$$L^{(0)} = \begin{bmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

After pass 1

$$L^{(1)} = \begin{bmatrix} 0 & 5 & 3 & 2 \\ \infty & 0 & \infty & 4 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

After pass 2

$$L^{(2)} = \begin{bmatrix} 0 & 5 & 3 & 2 \\ \infty & 0 & 5 & 4 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

After pass 3

$$\because L^{(3)} = L^{(2)} = \begin{bmatrix} 0 & 5 & 3 & 2 \\ \infty & 0 & 5 & 4 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

After this, no extensions lead to a shorter path for any of the pairs, hence the algorithm is complete and we can stop. $L^{(3)}$ contains the required single source shortest path lists for every starting node. **(Ans.)**

Floyd-Warshall algorithm

Although, we perform a similar computation here as in the generic APSP algorithm, the intuition behind this algorithm is a bit different.

In this algorithm, the values $d_{ij}^{(k)}$ represent the distance to be travelled from node i to node j by somehow travelling to node k first and then to j from k. This covers all the different possibilities to get from i to j, including the direct path as well. We then $\vee$ these possibilities to get the final answer. (Remember Discrete Structures).

But the above algorithm only helps us to determining the existence of a path. Instead we use the min operation to find the minimum path instead of using the $\vee$ operator defined for booleans.

Algorithm 18: Floyd-Warshall Algorithm

Input: Graph $G = (V, E)$, weight matrix $W = (w_{ij})$ for all i, j

Output: Shortest path distances between every pair of vertices

$n \leftarrow |V|$;
for $k = 1$ **to** n **do**
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      $d[i][j] \leftarrow \min(d[i][j], d[i][k] + d[k][j])$;
return d;

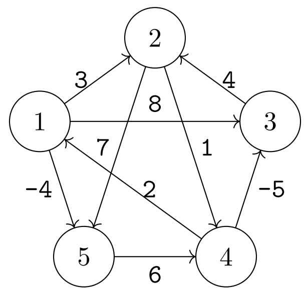
The time complexity of the Floyd-Warshall algorithm is $\mathcal{O}\left(|V|^3\right)$. The space complexity of this algorithm is $\mathcal{O}\left(|V|^2\right)$.

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right), & k \geq 1 \end{cases}$$

$$\Pi_{ij}^{(0)} = \begin{cases} \text{NIL}, & i = j \vee w_{ij} = \infty, \\ i, & i \neq j \wedge w_{ij} < \infty \end{cases}$$

$$\Pi_{ij}^{(k)} = \begin{cases} \Pi_{ij}^{(k-1)}, & d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \Pi_{kj}^{(k-1)}, & d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

For example, consider the following graph:



$$D^0 = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \qquad \Pi^0 = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^1 = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \qquad \Pi^1 = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^2 = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \qquad \Pi^2 = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^3 = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \qquad \Pi^3 = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^4 = \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & 1 & -5 & 0 & -2 \\ 8 & 7 & 1 & 6 & 0 \end{bmatrix} \qquad \Pi^4 = \begin{bmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

$$D^5 = \begin{bmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 4 & 4 & 0 & 5 & 3 \\ 2 & 1 & -5 & 0 & -2 \\ 8 & 7 & 1 & 6 & 0 \end{bmatrix} \qquad \Pi^5 = \begin{bmatrix} \text{NIL} & 1 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

We have completed all the 5 iterations, the matrix D^5 contains the single source shortest path lists from a node i in it's i^{th} row. Similarly, the corresponding row of Π^5 contains the required parent lists for the single source shortest paths. **(Ans.)**

Chapter 3

Divide and Conquer

The divide and conquer approach is a powerful strategy for designing asymptotically efficient algorithms. A famous example of the divide and conquer algorithm is *merge sort*, an asymptotically efficient sorting algorithm.

In the divide and conquer method, we are given an instance of a problem and we solve it recursively until it becomes small enough (base case) that we know the answer directly. In the recursive cases, we solve the problem as follows:

1. **Divide** the problem into one or more sub-problems that are smaller instances of the same problem.
2. **Conquer** the sub-problems by recursively solving them.
3. **Combine** the sub-problems to form a solution to the original sub-problem.

We have seen how to solve the recurrences using various different methods, we will use those methods to analyse the time and space complexities of the algorithms discussed in the first chapter.

3.1 Binary Search

The purpose of a searching algorithm is to find a specific element present in a data structure.[1]

The naive linear search algorithm used in appendix A.2 has asymptotic time complexity of $\Theta(n)$ in the average and worst case scenario. There exists a more efficient algorithm for searching based on the divide and conquer technique called **binary search**.

[1] Read more about the searching problem in appendix A

Algorithm 19: Binary Search - Recursive (BS-Recursive)

Input: Sorted array A of n elements, Element b to be searched, *low* and *high*- the index of smallest and largest element of subarray.

Output: Index of b in A

```
1 if low > high then
2     return −1;
3 mid ← (low + high)/2;
4 if A[mid] = b then
5     return mid;
6 else if A[mid] < b then
7     return BS-Recursive(A, b, mid + 1, high);
8 else
9     return BS-Recursive(A, b, low, mid − 1);
```

The initial call to BS-Recursive$(A, b, 1, n)$ provides us with the index of b.

3.1.1 Why this specific algorithm

The above algorithm seems pretty arbitrary in terms of the method used and it is not visible at first glance how it relates to the divide and conquer method.

Consider line 4 of the above binary search algorithm, we compute the index of the middlemost element of the array and compare it with the element to be found. If the middlemost element of the array is the required element b, then we directly return *mid*.

If this does not occur then we compare the middlemost element and b depending on their difference, we **narrow down the position of b to the left or right subarray of A**. We are allowed to narrow the position down because we know that A is actually a sorted array. Therefore, if b is larger than the middlemost element then we can say that it is in the right subarray of A. Otherwise it is in the left subarray of A.

3.1.2 Divide and Conquer

1. **Divide** the array into half
2. **Conquer** the subarray by finding the position of the element in this subarray.

There is no combine step as we only have one recursive call to a smaller sub-problem.

Another popular variation of the binary search algorithm is as follows where we use tail recursion elimination to make use of an iterative statement:

Algorithm 20: Binary Search

Input: Sorted array A of n elements, Element b to be searched

Output: Index of b in A

$high \leftarrow n$, $low \leftarrow 1$;
while $high \geq low$ **do**
    $mid \leftarrow \left\lfloor \dfrac{low + high}{2} \right\rfloor$;
    if $A\left[mid\right] = b$ **then**
        return mid;
    else if $A\left[mid\right] < b$ **then**
        $high \leftarrow mid - 1$;
    else
        $low \leftarrow mid + 1$;
return -1;

The binary search algorithm has time complexity $O(\lg n)$ and the space complexity is $O(\lg n)$ (for the recursive algorithm) and $O(1)$ (for the iterative algorithm).

For example, consider searching for the element 5 the following array:

$$[1, \quad 2, \quad 4, \quad 5, \quad 7, \quad 8]$$

Pass 1

$$[\underset{low}{1}, \quad 2, \quad \underset{mid}{\mathbf{4}}, \quad 5, \quad 7, \quad \underset{high}{8}]$$

The value of *mid* element is less than the value to be searched (5), so search the right sub-array.

Pass 2

$$[1, \quad 2, \quad 4, \quad \underset{low}{5}, \quad \underset{mid}{\mathbf{7}}, \quad \underset{high}{8}]$$

The value of *mid* element is more than the value to be searched (5), so search the left sub-array.

Pass 3

$$[1, \quad 2, \quad 4, \quad \underset{low,\ mid,\ high}{\mathbf{5}}, \quad 7, \quad 8]$$

The value of *mid* element is equal to the value to be searched (5), so return the current value of $mid(4)$.

3.2 Quick Sort

3.3 Merge Sort

Appendix A

Important Algorithms

A.1 Searching

The purpose of a searching algorithm is to find a specific element present in a data structure. Usually, the problem is stated as follows:

> Given a sequence of elements, $\langle a_1, a_2, a_3, \cdots, a_n \rangle$, find the position of an element b in the sequence.

A.2 Linear Search

To solve this problem, the naive approach would be to search all elements of the array (note that we will be using fixed length arrays to represent sequences), stopping when we find the required element.

Algorithm 21: Linear Search (Naive approach)

Input: Array A of n elements, Element b to be searched
Output: Index of b in A

for $i \leftarrow 1$ **to** n **do**
  if $A[i] = b$ **then**
    return i;
  end
end
return -1;

We can easily analyse the average-case (required element is somewhere close to the middle of the array) time complexity of the above algorithm as follows.

$$
\begin{aligned}
T(n) &= \underbrace{c_0 + c_0 + \cdots + c_0}_{n/2 \text{ times}} \\
&= \frac{c_0}{2} \cdot n \\
T(n) &\in \Theta(n)
\end{aligned}
$$

We read about a better sorting algorithm in chapter 3.

A.3 Sorting

The purpose of a sorting algorithm is to sort a given set of records based on the value of a particular field present in the record. The values upon which the records are sorted are called the *keys* of the record. The remaining fields of the record are known as *satellite data*.

Usually, the problem is stated as follows:

> Given a sequence of elements, $\langle a_1, a_2, a_3, \cdots, a_n \rangle$, return it's permutation $\langle a'_1, a'_2, a'_3, \cdots, a'_n \rangle$, where $a'_1 \leq a'_2 \leq a'_3 \leq \cdots \leq a'_n$.

A.3.1 Why sorting?

Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms. There are several reasons:

- An application needs to inherently sort information and prepare results. Example: Banks need to sort cheques by cheque number.
- Algorithms often use sorting as a key subroutine.
- Sorting is a problem of historical interest leading to development of various different forms of algorithm solving techniques.
- A non-trivial lower bound for sorting can be proven for sorting. Since, the upper bound of many sorting algorithms match lower bound asymptotically, we can say that certain sorting algorithms are optimal.

Acronyms

APSP All-Pairs Shortest Paths 31–33

BFS Breadth-First Search 21

DAA Design and Analysis of Algorithms 1

DAG Directed Acyclic Graph 21, 22

DFS Depth-First Search 20–22

DP Dynamic Programming 11

Gojo Aaditya Joil 1

MST Minimum Spanning Tree 23, 25, 26, 28

RRG Rupak R. Gupta 1

SCC Strongly Connected Component 22

$$\boxed{\begin{bmatrix} \mathbb{R} & \mathfrak{R} \\ \mathcal{g} & \infty \end{bmatrix} \binom{0}{\mathbb{J}}^{2}}$$

The End.