CHAPTER 4

ALGORITHMS

Syllabus: Algorithms: Analysis, Asymptotic notation, Notions of space and time complexity, Worst- and average-case analysis; Design: Greedy approach, Dynamic programming, Divide-and-conquer; Tree and graph traversals, Connected components, Spanning trees, Shortest paths; Hashing, Sorting, Searching. Asymptotic analysis (best, worst, average cases) of time and space, Upper and lower bounds, Basic concepts of complexity classes – P, NP, NP-hard, NP-complete.

4.1 INTRODUCTION

In computer science, an algorithm provides a set of step-by-step instructions for solving various problems. The set of instructions should be unambiguous, finite and provide a clear termination point. An algorithm accepts some input values and produces outputs. The algorithm can be built in simple English language or in programming language. This subject helps in solving and analysing complex problems through logical thinking.

4.2 ALGORITHM

An algorithm is a well-defined procedure which takes a set of values as an input and produces a set of values as the output.

4.2.1 Properties of Algorithm

There are certain properties algorithms have:

1. An algorithm has zero or more inputs.
2. It should produce at least one output.
3. It should terminate within the finite time.
4. It should be deterministic (ambiguous).
5. It is generic to all programming languages.

4.2.2 Steps to Solve a Problem

1. Define the problem, which means what is input and what should be the output.
2. Identify the constraints which are to be satisfied.
3. Design the algorithm for a given problem.
4. Draw the flowchart.
5. Verify the process.
6. Implement the coding.
7. Perform time and space analysis.

4.2.3 Algorithm Analysis

If a problem has more than one solution, then the technique used to decide which solution is best is known as algorithm analysis.

4.2.3.1 A Posteriori and A Priori Analysis

Analysis is done on two parameters: time complexity and space complexity. Time complexity means time taken by an algorithm to solve a problem. Space complexity means memory space required by an algorithm to solve a problem. There are two types of analysis:

1. ***A Posteriori* Analysis:** It is a software- and hardware-dependent analysis. It keeps on changing from one system to other system. This type of analysis tells the exact time and space complexity, meaning how much an algorithm takes time and space for solving a problem on the given system.
2. ***A Priori* Analysis:** It is a software- and hardware-independent analysis. It is constant for all the systems. This type of analysis tells the approximate time and space complexity. It is the determination of the order of magnitude of a statement, meaning how many times a statement is executing.

4.2.3.2 Asymptotic Analysis

Suppose there are given two algorithms for a task, how do we find out which one is better?

This can be done by actually implementing both the algorithms and running them on the same computer with different input values. Then the comparison is made in terms of time. Algorithm that takes less time is considered better. But there are so many problems with this approach.

1. It might be possible that for certain inputs first algorithms performs better than second and for other set of inputs second algorithm performs better. So,the decision of choosing best among them becomes difficult.
2. With the change in operating machine, performance of algorithm varies. On one machine first algorithm can work better and on another machine, the second works better. This is another problem associated with this approach.

Above issues can be handled using asymptotic analysis techniques for analysing algorithms. In asymptotic analysis, analysis of algorithm in done in terms of input size. Actual running time is not computed, variation of time with input size is calculated.

Now, understand that how the issues with above technique are resolved by asymptotic analysis technique using the below example:

Example 4.1

Let us consider an example of a search problem (searching a given item) in a sorted array. One way to search is linear search (the order of growth is linear) and the other way is binary search. Suppose we have two machines with different processing speeds. We run linear search on the fast machine and binary search on slow machine. For small input size, the linear search will perform better. But as we increase the size of input, binary search will perform better. The reason behind this is,the order of growth of binary search is logarithmic and that of linear search is linear. So, by using this technique the issues of machine dependency are resolved.

4.2.4 Asymptotic Notation

Asymptotic notation is based on two assumptions, which hold in most of the cases and have their importance. It is very much important to understand the assumptions and limitations of asymptotic notations:

1. **Input size:** It is interesting to know how the running time of an algorithm grows for a large input size n.
2. **Constant:** The running time of an algorithm also depends on various constants. But for a large input n, the constant factors would be ignored.

For representing best-case, average-case, and worst-case complexity, there are three different notations as given in the following sub-sections.

4.2.4.1 Theta Notation (Θ)

Theta notation shows the tightest upper bound and the tightest lower bound to the given function (Fig. 4.1). It is used to define the average-case running time. For a given function $f(n)$, we denote $\Theta(g(n))$ as $\Theta(g(n)) = \{f(n)$: there exists positive constants $c_1 > 0$, $c_2 > 0$ and $n_0 > 0$ such that $0 < c_1(g(n)) \leq f(n) \leq c_2(g(n)) \ \forall \ n \geq n_0\}$

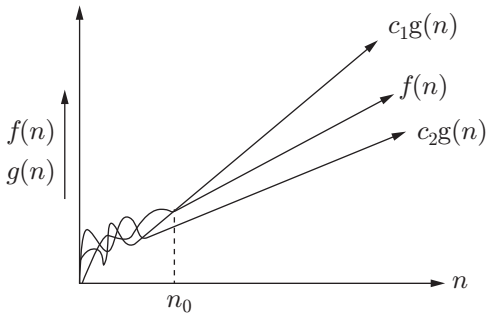


Figure 4.1 Theta notation (Θ)

Problem 4.1: Consider the following functions:

$$f(n) = n$$

$$g(n) = \frac{n}{8}$$

For $f(n) = \Theta(g(n))$, find the values of c_1, c_2 and n_0.

Solution: From the definition of theta (Θ) notation, we have

$$0 < c_1(g(n)) \leq f(n) \leq c_2(g(n)) \quad \forall n \geq n_0$$

By putting $c_1 = 1, 2, 3$ to satisfy the above condition, we get

$$c_1(g(n)) \leq f(n)$$

$$1\left(\frac{n}{8}\right) \leq n$$

$$c_1 = 1$$

By putting $c_2 = 1, 2, 3$ to satisfy the above condition, we get

$$f(n) \leq c_2(g(n))$$

$$8\left(\frac{n}{8}\right) \leq n$$

$$c_2 = 8$$

By putting $n_0 = 1, 2, 3, \ldots$, to satisfy the above condition, we get $n_0 = 1$.

4.2.4.2 Big-O Notation (O)

Big-O notation shows the upper bound to the given function (Fig. 4.2). It is used to define the worst-case running time. For a given function $f(n)$, we denote $O(g(n))$ as $O(g(n)) = \{f(n)$: there exist positive constants $c > 0$ and $n_0 > 0$ such that $0 < f(n) \leq c(g(n)) \forall n \geq n_0\}$

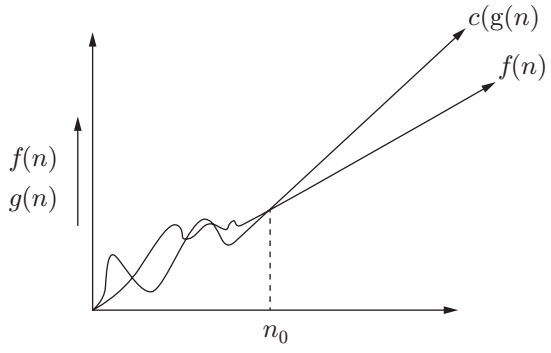


Figure 4.2 | Big-O notation (O).

Problem 4.2: Consider the following functions:

$$f(n) = n$$

$$g(n) = n$$

For $f(n) = O(g(n))$, find the values of c and n_0.

Solution: From the definition of Big-O notation, we have

$$f(n) \leq c(g(n)) \quad \forall n \geq n_0$$

By putting $c = 1$ and $n = 1$, we get

$$n \leq c \cdot n \ \forall \ n \geq n_0$$

The above condition is satisfied. So, $c = 1$ and $n_0 = 1$.

4.2.4.3 Omega Notation (Ω)

Omega notation shows the lower bound to the given function (Fig. 4.3). It is used to define the best-case running time. For a given function $f(n)$, we denote $\Omega(g(n))$ as $\Omega(g(n)) = \{f(n)$: there exist positive constants $c > 0$ and $n_0 > 0$ such that $0 < c(g(n)) \leq f(n) \ \forall \ n \geq n_0\}$.

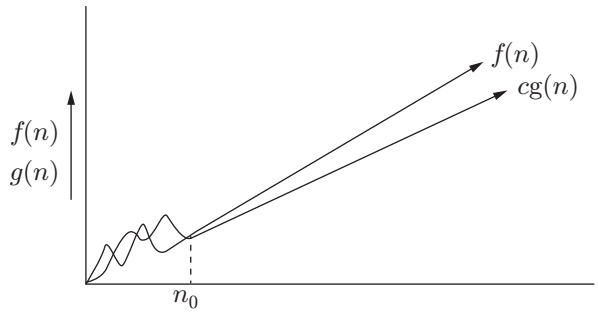


Figure 4.3 | Omega notation (Ω).

Problem 4.3: Consider the following functions:

$$f(n) = n^2$$

$$g(n) = n$$

For $f(n) = \Omega(g(n))$, find values of c and n_0.

Solution: From the definition of Omega notation, we have

$$c(g(n)) \leq f(n) \ \forall \ n \geq n_0$$

By putting $c = 1$ and $n = 1$, we get

$$c \cdot n \leq n^2 \ \forall \ n \geq n_0$$

The above condition is satisfied. So, $c = 1$ and $n_0 = 1$.

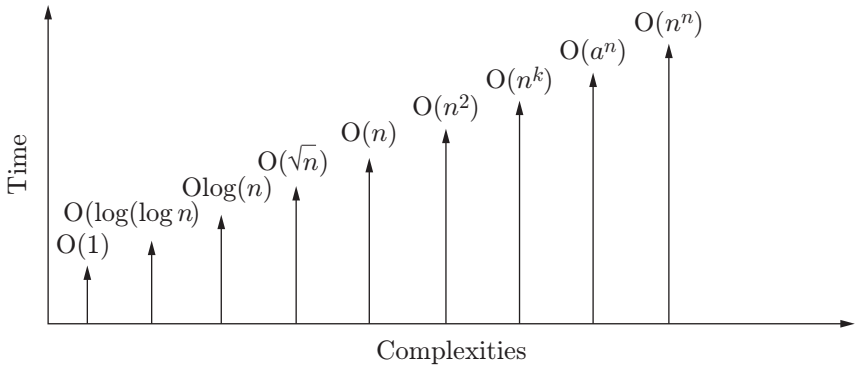


Figure 4.4 Time complexities in increasing order.

4.2.4.4 Small-o Notation (o)

Small-o represents an upper bound which is not a tight upper bound. It is also used to define the worst-case running time. For a given function $f(n)$, we denote $o(g(n))$ as $o(g(n)) = \{f(n)$: there exist positive constants $c > 0$ and $n_0 > 0$ such that $0 < f(n) < c(g(n)) \ \forall \ n \geq n_0\}$.

4.2.4.5 Small Omega Notation (ω)

Small omega (ω) represents a lower bound which is not a tight lower bound. It is used to define the best-case running time. For a given function $f(n)$, we denote $\omega(g(n))$ as $\omega(g(n)) = \{f(n)$: there exist positive constants $c > 0$ and $n_0 > 0$ such that $0 < c(g(n)) < f(n) \ \forall \ n \geq n_0\}$.

4.2.4.6 Properties of Asymptotic Notations

1. **Transitivity:** All asymptotic notations show transitivity.
   If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.
   If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
   If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$.
   If $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then $f(n) = o(h(n))$.
   If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then $f(n) = \omega(h(n))$.
2. **Reflexivity:** Only Theta, Big-O, and Omega notations show reflexivity. Small-o and small-omega do not show reflexive property.
   $f(n) = \Theta(f(n)) \quad f(n) = O(f(n)) \quad f(n) = \Omega(f(n))$
3. **Symmetric:** Only Theta notation is symmetric.
   $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
4. **Transpose symmetry:**
   $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
   $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

4.2.4.7 Types of Complexities

1. Constant time complexity: $O(1)$
2. Logarithmic time complexity: $O(\log(\log n))$, $O(\sqrt{\log n})$ and $O(\log n)$
3. Linear time complexity: $O(\sqrt{n})$ and $O(n)$
4. Polynomial time complexity: $O(n^k)$, where k is a constant and is >1
5. Exponential time complexity: $O(a^n)$, where $a > 1$

The complexities in increasing order are represented in Fig. 4.4.

4.2.5 Recurrence Relations

Recurrence relations are recursive definitions of mathematical functions or sequences. For example, the recurrence relation defines the famous Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, …:

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = 1$$
$$f(0) = 1$$

Given a function defined by a recurrence relation, we want to find a 'closed form' of the function. In other words, we would like to eliminate recursion from the function definition. There are several techniques for solving recurrence relations, and these are discussed in the following sub-sections.

4.2.5.1 Iteration Method

This method is accurate but involves a lot of algebraic expressions which are difficult to keep track of. It can get very challenging for further complicated recurrence relations.

Problem 4.4: We have $T(n) = 1 \quad$ if $n = 1$

$$T(n) = 4T\left(\frac{n}{2}\right) + n \quad \text{if } n > 1$$

$$T(n) = 4T\left(\frac{n}{2}\right) + n = 4\left(4T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$$

$$= 4\left(4\left(4T\left(\frac{n}{8}\right)+\frac{n}{4}\right)+\frac{n}{2}\right)+n$$

$$= 64T\left(\frac{n}{8}\right)+4n+2n+n$$

$$= n+2n+4n+64T\left(\frac{n}{8}\right)$$

$$= n+2n+4n+\cdots+2^j n+\cdots+4^i T\left(\frac{n}{2^i}\right)$$

Find the time complexity when $(n/2^i) = 1$.

Solution:

We have $i = \log n$. So, the time complexity of n can be represented as

$$T(n) = n+2n+4n+8n+2^i n+4^{\log n}\mathrm{O}(1)$$

We get $$n\left(\sum_{i=0}^{\log(n-1)} 2^i\right)+4^{\log n}\mathrm{O}(1)$$

We Know that $$\sum_{k=0}^{m} x^k = \frac{x^{m+1}-1}{x-1}$$

So, $$n\left(\frac{2^{\log(n-1)+1}-1}{2-1}\right)+4^{\log n}\mathrm{O}(1)$$

Solving, we get

$$n2^{\log n} - n + 4^{\log n}\mathrm{O}(1) \Rightarrow n^2 - n + n^{\log 4}\mathrm{O}(1)$$

$$\Rightarrow 2n^2\mathrm{O}(1) - n \Rightarrow \mathrm{O}(n^2)$$

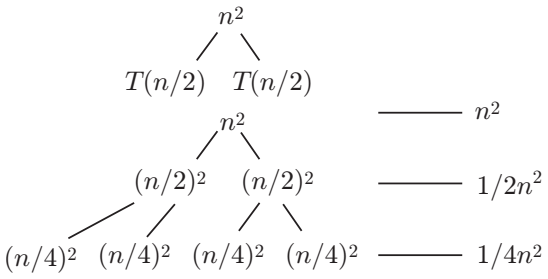
So, the running time complexity is given as: $T(n) = \mathrm{O}(n^2)$.

4.2.5.2 Recursion Tree

A recursion tree is useful for visualizing what happens when a recurrence is iterated. It represents a tree of recursive calls and the amount of work done at each call is shown with the help of a diagram.

Example 4.2

Consider the recurrence $T(n) = 2T(n/2) + n^2$.



How deep does the tree go?

We stop at the leaf, and we know we are at a leaf when we have a problem of size 1:

$$1 = \left(\frac{n}{2^i}\right)^2 \Rightarrow n^2 = 2^{2i} \Rightarrow n = 2^i \Rightarrow i = \log n$$

The amount of work done is given as

$$\sum_{i=0}^{\log n} \frac{n^i}{2} = \Theta n^2$$

This is geometrically decreasing in size, so it would not get any bigger than n^2.

4.2.5.3 Master Theorem

Master theorem provides a method to solve various recurrence relations. Although all the relations can't be solved using this method, but it is one of the useful method of solving complex relations of the form:

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

where a and b are constants. *f is a function of n. this recursive algorithm breaks up the input into sub-problems of size n/b each.*

Three cases of master theorem are as follows:

1. **Case 1:** If $f(n)$ is $\mathrm{O}(n^{\log_b a-\varepsilon})$ then $T(n)$ is $\Theta(n^{\log_b a})$. The rate of growth at leaves is faster than f, more work is done on leaves.
2. **Case 2:** If $f(n)$ is $\Theta(n^{\log_b a})$ then $T(n)$ as $\Theta(n^{\log_b a}\log n)$. The rate of growth of leaves is same as f. So, the same amount of work is done at every level of tree. The leaves grow at the same rate as f, so the same order of work is done at every level of the tree. The tree has $\mathrm{O}(\log\ n)$ times the work done on one level, yielding $T(n)$ as $\Theta(n^{\log_b a}\log n)$.
3. **Case 3:** $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$. Here f grows faster than the number of leaves, which means that asymptotically the total amount of work is dominated by the work done at the root node. For the upper bound, we also need an extra smoothness condition on f in this case, namely, that $af(n/b) \le cf(n)$ for some constant $c < 1$ and large n. In this case, $T(n)$ is $\Theta(f(n))$.

Example 4.3

Consider the following recurrence:

$$T(n) = 4T\left(\frac{n}{2}\right)+n$$

In the given relation, $a = 4$ and $b = 2$.

For this recurrence, there are $a = 4$ sub-problems, each dividing the input by $b = 2$, and the work done on each call is $f(n) = n$. Thus, $n^{\log_b a}$ is n^2 and $f(n)$ is $\mathrm{O}(n^{2-\varepsilon})$ for $\varepsilon = 1$, and Case 1 applies. Thus, $T(n)$ is $\Theta(n^2)$.

Example 4.4

Consider the following recurrence:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

For this recurrence, there are again $a = 4$ sub-problems, each dividing the input by $b = 2$; but now the work done on each call is $f(n) = n^2$. Again, $n^{\log_b a}$ is n^2 and $f(n)$ is thus $\Theta(n^2)$, so Case 2 applies. Thus, $T(n)$ is $\Theta(n^2 \log n)$. Note that increasing the work on each recursive call from linear to quadratic has increased the overall asymptotic running time by only a logarithmic factor.

Example 4.5

Consider the following recurrence:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

For this recurrence, there are again $a = 4$ sub-problems, each dividing the input by $b = 2$, but now the work done on each call is $f(n) = n^3$. Again, $n^{\log_b a}$ is n^2 and $f(n)$ is thus $\Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$. Moreover, $4(n/2)^3 \leq kn^3$ for $k = 1/2$, so Case 3 applies. Thus, $T(n)$ is $\Theta(n^3)$.

4.3 HASHING

Hashing is technique that transforms a variable length input to a fixed length output. This technique is used for indexing, which helps in constant time search for insertion, deletion and information retrieval.

4.3.1 Hash Table

A hash table is a data structure used for hashing. It is an associative array that contains keys with values. In an array, indices are required to be integer, but a hash table allows floating point numbers, strings, array or structures as keys. Hash tables provide efficient lookup operation, that is, whenever a key is given, hash table provides associated value in almost constant time complexity. This transformation is done using hash function.

4.3.2 Hashing Functions

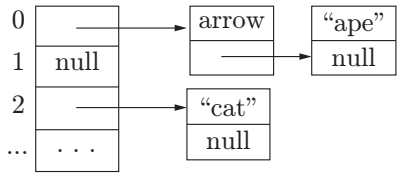
A hashing function is a transformation function, coded to return a value based on key. If h is a hash function that takes key k as parameter, will compute the index at which value is placed. If h is a hash function that takes key as a parameter, it will compute index where value is placed. It is important to note that hash function returns the same index value every time for same key.

4.3.3 Collisions

Collision occurs when one key value stores more than one value. Hash function producesthe same position for two values. Hash table cannot ensure to have single value corresponding to a key. This is because a hash table has fixed number of keys. Any new entry in hash table, after all the slots are filled, causes collision. However, good hash tables use some mechanism to avoid the situations of collisions or to handle them efficiently.

4.3.3.1 Separate Chaining

1. If more than one item is assigned the same hash index, "chain" them together.
2. In hash table, each position act as bucket that can store multiple data items.
3. There are the following two implementations:
   - Each bucket is itself an array: The disadvantages of bucket as array are
     (a) Memory wastage
     (b) Problem of overflow occurs when we try to add new item to a full bucket.
   - Each bucket is a linked list: The disadvantage of bucket as linked list is
     Pointers require more memory.



4.3.3.2 Open Addressing

1. Collision occurs if the hash function produces the same position occupied by another item. In this case, another open position is searched.

   Example: Hash code produced for 'White' is 22. But position 22 is already occupied, so it is placed at position 23.

2. There are three ways of finding an open position. This process is known as **probing**. To search any item in hash table, probing may be required.

   Example: When item "white" is searched, first we look for it at position 22. If it is not found there, we move to next location. The search will be stopped only when an empty position is encountered.

0	Arrow
1	
2	Cat
3	
4	Eye
5	
...	... ...
22	When
23	White
24	Yellow
25	Zero

4.3.3.3 Linear Probing

In case of linear probing the probe sequence is h(key), h(key) + 1, h(key) + 2, …, uptill required.

Examples: The item 'Ape' ($h = 0$) will be placed at position 1, because position 0 is already occupied. then 'Ball' ($h = 1$): try to place it at position 1, 1+1, 1+2. Position 3 is open so it will be placed here.

Advantage: If open position is available, linear probing will definitely find it.

Disadvantage: If there are clusters of filled positions, then length of subsequent probes increases. Probe length is the number of positions considered during a probe.

0	Arrow
1	Ape
2	Cat
3	Ball
4	Eye
5	
...	... ...
22	When
23	White
24	Yellow
25	Zero

4.3.3.4 Quadratic Probing

The probe sequence is

h(key), h(key) + 1, h(key) + 4, h(key) + 9, …,
wrapping around as necessary

The offsets are perfect squares: $h + 12$, $h + 22$, $h + 32$, ….

Examples:

a. 'Ape' ($h = 0$): try 0, 0 + 1 − open!
b. 'Ball' ($h = 1$): try 1, 1 + 1, 1 + 4, 1 + 9 − open!

Advantage: It reduces clustering.

0	Arrow
1	Ape
2	Cat
3	
4	Eye
5	Ball
...	... ...
22	When
23	White
24	Yellow
25	Zero

Disadvantage: It may fail to find an existing open position. For example, let table size = 10, and x denote the occupied blocks. Trying to insert a key with h(key) = 0 offsets the probe sequence shown in italic.

0	x		5	x	25
1	x	1 81	6	x	16 36
2			7		
3			8		
4	x	4 64	9	x	9 49

4.3.3.5 Double Hashing

It uses the following two hash functions:

1. h_1 computes the hash code.
2. h_2 computes the increment for probing.

The probe sequence is: h_1, $h_1 + h_2$, $h_1 + 2 \times h_2$, ….

Examples:

1. h_1 = our previous h
2. h_2 = number of characters in the string
3. 'Ape' ($h_1 = 0$, $h_2 = 3$): try 0, 0 + 3 − open!
4. 'Ball' ($h_1 = 1$, $h_2 = 4$): try 1 − open!

Advantages:

1. It combines the good features of linear and quadratic probing:
2. It reduces clustering
3. It will find an open position if there is one, provided the table size is a prime number

0	Arrow
1	Ball
2	Cat
3	Ape
4	Eye
5	
...	... ...
22	When
23	White
24	Yellow
25	Zero

4.3.3.6 Removing Items Under Open Addressing

Consider the following scenario:

Using linear probing:

(a) Insert 'Ape' ($h = 0$): Try 0, 0 + 1 − open!
(b) Insert 'Ball' ($h = 1$): Try 1, 1 + 1, 1 + 2 − open!
(c) Remove 'Ape'.
(d) Search for 'Ape': Try 0, 0 + 1 − no item.
(e) Search for 'Ball': Try 1 − no item, but 'Ball' is further down in the table.

1. On removing an item from a position, leave a special value in that position to indicate that an item was removed.
2. There are three types of positions: occupied, empty and removed.
3. Stop probing on encountering an empty position, but not when encountering a removed position.
4. Insert items in either empty or removed positions.

0	Arrow
1	1
2	Cat
3	Ball
4	Eye
5	
...	... ...
22	When
23	White
24	Yellow
25	Zero

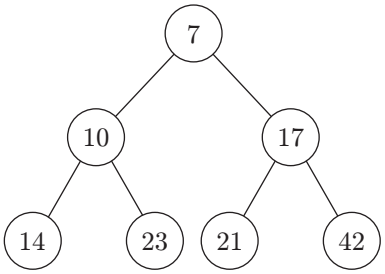
4.4 BINARY HEAP

A binary heap is a binary tree with two additional constraints:

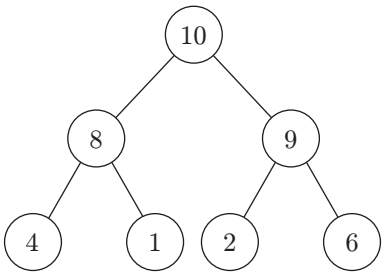
1. Binary heap should satisfy the property of complete binary tree.
2. It should satisfy the heap ordering property, that is, all the nodes should be either greater than or equal to or less than or equal to its children.

According to the heap ordering property, binary heaps are of two types:

1. **Min-Heap Tree:** In Min heap ordering, the value of each node is less or equal to its child node. Minimum element of the tree will be the root element. The largest element exists in the last level. The tree should be an almost complete binary tree.



2. **Max-Heap Tree:** In max heap, each node in the tree will be greater or equal to its child node. The maximum element of tree will be the root element. The smallest element exists in the last level. The tree should be an almost complete binary tree.



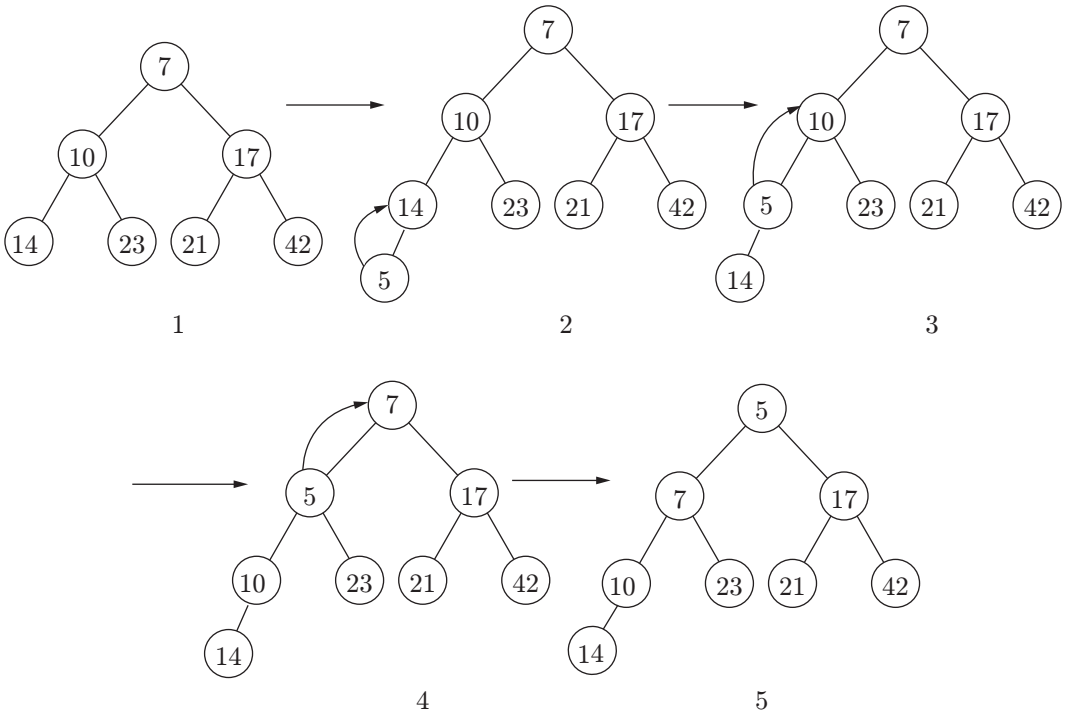
4.4.1 Insertion in Min-Heap Tree

The following are the steps to follow for insertion in the min-heap tree:

1. Add the new element to the last level of heap.
2. Compare the new element with its parent node.
3. If the order is correct then stop, otherwise swap the element with its parent.
4. Perform steps 2 and 3 until a new element is at position.

Example 4.6

1. Insert the given value (5) in the given min-heap tree.
2. Element 5 is inserted at position of left child of node 14. To satisfy the property of min-heap, swap (5, 14).
3. To satisfy the property of min-heap, swap (5, 10).
4. To satisfy the property of min-heap, swap (5, 7).
5. Now, element 5 is in its right position. This is the final tree.



4.4.2 Deletion in Min-Heap Tree

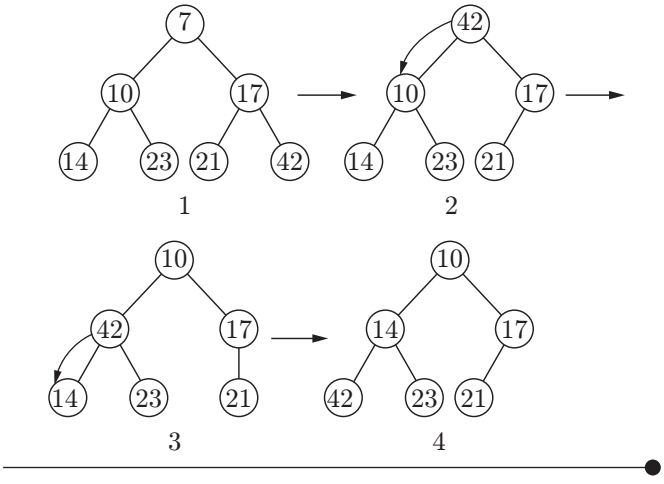
The following are the steps to follow for deletion in the min-heap tree:

1. Root of the heap is replaced with the element at the last level.
2. New root element is compared with its children; if it maintains ordering then stop.
3. Otherwise, swap the root with one of its children and return to previous step. (Smaller element is swapped for min-heap and larger child for max-heap)

Example 4.7

1. Deletion always occurs at the root level, so element 7 will be deleted. Before deletion of element 7 at the root, the element (42) is swapped to the root node from the extreme right-hand side of the last level and then 7 is deleted.
2. To maintain the min-heap property of the tree, swap 42 and min(left-child, right-child), which means swap 42 with 10.
3. To maintain the min-heap property of the tree, swap 42 and min(left-child, right-child), which means swap 42 with 14.

4. This represents the final tree, which is in the min-heap property.



4.4.3 Time Complexity

1. The worst-case running time for insertion is O(log n) because at most (log n) swaps are required to stable the min-heap property of the tree.
2. The worst-case running time for deletion is O(log n) because at most (log n) swaps are required to stable the min-heap property of the tree.

Some important points of binary heap are as follows:

1. Because of its structure, a heap with height k will have between 2^k and $2^{k+1} - 1$ elements. Therefore, a heap with n elements will have height $= \log_2 n$.
2. In heap data structure, element with highest or lowest priority is stored as root element. Heap is not considered as sorted structure, rather it is partially ordered. There is no relation among nodes at a given level.

   Heap is very useful data structure for implementing priority queues. Element with highest priority can be easily removed.
3. Since a binary heap is a complete binary tree, it can be easily represented as an array, and an array-based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by $2 \times I + 1$ and right child by $2 \times I + 2$.

4.5 SEARCHING AND SORTING

Searching and sorting are the most common activities performed by the computer. These operations are the most important part for managing the data.

1. **Searching** is the algorithmic process of finding an item in a collection of items. The search result is either true or false depending on whether or not the item is present or not in the list.
2. **Sorting** is ordering a list of objects according to a given condition or property. For example, ordering a set of numbers in an ascending or descending order.
3. In **internal sorting**, the data which is to be sorted fits into the main memory. Internal sorting has an advantage of the random access nature of the main memory.
4. In **external sorting**, large amount of data is sorted. This large amount of data cannot be fit in main memory, so the data is placed in auxiliary memory. When sorting has to perform, small chunks of data are brought into main memory, sorted and placed back to a temporary file. At last all the sub-files are merged in one large file.
5. In **stable sorting**, the order of equal elements is preserved in a sorted list. Some of the sorting algorithms that show characteristics of stable sort are insertion sort, bubble sort, merge sort, etc.; whereas heap sort, quick sort, etc., are not stable sorts.
6. **In-place sorting:** In this, the input is transformed using a data structure with a small, constant amount of extra storage space. The input is usually overwritten by the output as the algorithm executes. Quicksort is commonly described as an in-place algorithm.
7. **Measurement of time complexity:** The time complexity of a sorting algorithm is measured by the number of critical operations performed. Examples of critical operations are as follows:
   - Key comparisons
   - Movements of records
   - Interchanges of two records
8. We use asymptotic analysis to denote the time efficiency of an algorithm. Efficiency of an algorithm also depends on the type of data for input, so we define best-, worst- and average-case efficiencies.

4.5.1 Linear Search

Linear search is a sequential way of finding an element in a given list. Linear search is a special case of brute force search. Its worst case is proportional to the number of elements in list.

4.5.1.1 Pseudocode for Linear Search

```
LINEAR_SEARCH (A[ ], x)
1. i = 1
2. while (A[i] = = x and i < n)
3. i = i + 1
4. if (A[i] = = x)
5. return i
6. else return 0
```

4.5.1.2 Time Complexity

From the analysis of above pseudocode, it can be observed that with the growth of input list, average and worst case complexities also grow linearly. In general, if there are n elements in the list, the worst-case requires n

comparisons. So, the complexity of linear search can be expressed in terms of some linear function.

So, running time $T(n) = O(n)$.

4.5.2 Binary Search

Binary search relies on a divide-and-conquer strategy to find an element within an already sorted list. It compares the element to be searched with the middle element of array. If the element is equal to the middle element, the search stops here and the position of element is returned. If it is less or greater than the middle element, then the values of sub-array are adjusted accordingly. The algorithm returns a unique value, that is, position of the element in the list, if the element is present.

4.5.2.1 Pseudocode for Binary Search

```
BINARY_SEARCH (A, value, left, right)
1. if right < left
2. return 'not found'
3. mid = floor((right-left)/2)+left
4. if A[mid] = value
5. return mid
6. if value < A[mid]
7. return BINARY_SEARCH(A, value, left, mid-1)
8. else
9. return BINARY_SEARCH(A, value, mid+1, right)
```

4.5.2.2 Time Complexity

The running time of the binary search is $\Theta(\log n)$.

4.5.3 Bubble Sort

Bubble sort algorithm is a comparison-based algorithm. It compares each element in the list with the next element. Elements are swapped if required. In every pass, one element comes to its position. This process repeats until a pass is made without disturbing any of the elements from their position.

4.5.3.1 Pseudocode for Bubble sort

```
BUBBLE_SORT(A)
1. for i ← 1 to n-1
2. for j ← 1 to n-1
3. If(A(j) > A(J+1))
4. Temp = A(j)
5. A(j) = A(j+1)
6. A(j+1) = Temp
```

Example 4.8

Sort the given elements (7, 5, 2, 4, 3 and 9) by using bubble sort.

After the first pass, we get

7, **5**, 2, 4, 3, 9

5, **7**, **2**, 4, 3, 9

5, 2, **7**, **4**, 3, 9

5, 2, 4, **7**, **3**, 9

5, 2, 4, 3, **7**, **9**

After the first pass, the first largest element (9) is on the top of array. So we have

5, 2, 4, 3, 7, 9

After the second pass, we get

2, 4, 3, 5, 7, 9

After the third pass, we get

2, 3, 4, 5, 7, 9

After the fourth pass, we get

2, 3, 4, 5, 7, 9

After the fifth pass, we get

2, 3, 4, 5, 7, 9

The sorted list is 2, 3, 4, 5, 7, 9

4.5.3.2 Time Complexity

It can be seen from the above example that in the bubble sort, each element passes one at a time and is placed in its correct position.

So, the number of passes = (number of elements − 1) = $(n - 1)$.

Cost per pass = $O(n)$.

Total cost = $O(n)(n - 1)$.

So, the worst-case runtime complexity is $O(n^2)$.

The average-case and best-case runtime complexity is $O(n^2)$.

4.5.4 Selection Sort

The selection sort first selects the smallest element from the unsorted list. Then that element is swapped with the first element in the list. In the next step, the size of the list is reduced by one because one element is present at its position. Next, the smallest element is swapped with the second element in list, and so on.

4.5.4.1 Pseudocode for Selection Sort

```
SELECTION_SORT(A)
1. for j ← 1 to n-1
2. smallest ← j
3. for i ← j + 1 to n
4. if A[ i ] < A[ smallest]
5. smallest ← i
6. Exchange A[ j ] ↔ A[ smallest]
```

Example 4.9

Sort the given elements (29, 64, 73, 34 and 20) by using selection sort.

We have

29, 64, 73, 34, 20

After the first pass, we get

20, 64, 73, 34, 29

After the second pass, we get

20, 29, 73, 34, 64

After the third pass, we get

20, 29, 34, 73, 64

After the fourth pass, we get

20, 29, 34, 64, 73

After the fifth pass, we get

20, 29, 34, 64, 73

The sorted list is 20, 29, 34, 64, 73.

4.5.4.2 Time Complexity

Number of passes = (Number of elements − 1)

Cost per pass = Swap + Comparisons = 1 + O(n)

Total cost = (n − 1)[1 + O(n)]

So, the worst-case runtime complexity is O(n^2).

The average-case and best-case runtime complexity is O(n^2).

4.5.5 Insertion Sort

The insertion sort inserts each element into its proper position. It chooses one element, inserts it to its position and shifts the rest of the list by one location. This process is repeated until no input element remains. Step-by-step procedure is given as follows.

1. Take an element from unsorted list.
2. Compare that element with sorted list from right to left.
3. Shift the list.

Here, it is assumed that first element is already sorted.

4.5.5.1 Pseudocode for Insertion Sort

```
INSERTION_SORT(A)
1. for j = 2 to n
2. key  ← A [j]
3. // Insert A[j] into the sorted sequence A[1..j-1]
4. j  ← i - 1
5. while i > 0 and A[i] > key
6. A[i+1]   ← A[i]
7. i  ← i - 1
8. A[j+1]   ← key
```

Example 4.10

Sorted part is bold in text and unsorted part is non-bold.

Suppose **30**, 10, 75, 33, 65 needs to be sorted using insertion sort.

After the first pass, we get

10, 30, 75, 33, 65

After the second pass, we get

10,30,75, 33, 65

After the third pass, we get

10, 30, 35, 75, 65

After the fourth pass, we get

10, 30, 35, 65, 75

The sorted list is 10, 30, 35, 65, 75.

4.5.5.2 Recurrence Relation of Insertion Sort

The recurrence relation for insertion sort is

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ T(n-1) + \Theta(n), & n > 1 \end{cases}$$

4.5.5.3 Time Complexity

In sorting, the most expensive part is the comparison of two elements. Zero comparisons are required to insert first element, one for second element, two comparisons for third element, and so on. Atmost $N - 1$ comparisons are required for the last element in the list.

Complexity is: $1 + 2 + 3 + 4 \cdots + (N - 1) = \mathrm{O}(n^2)$

The worst-case running time is O(n^2)

The best-case running time is O(n)

4.5.6 Heap Sort

Heap sort is one of the comparison based sort that uses heap data structure. It is a special case of selection sort. Its typical implementation is not stable, but can be made stable. Heap can be built by the following two types:

1. **Min-Heap:** The parent node will be less than or equal to its children nodes. So, the minimum element will be at root.
2. **Max-Heap:** The parent node will be greater than or equal to its children nodes. The maximum element will be at the root.

4.5.6.1 Heap Sort Algorithm for Sorting in Increasing Order

1. First build a max-heap from the input List.
2. In max heap, the largest element is at the root. So, swap the root with last element. Re-heapify the

tree. Repeat the above steps until the size of the heap is greater than 1.

- **Heapify function:** Heapify is an important subroutine for manipulating max-heaps. Its inputs are an array A and an index i into the array. When Heapify is called, it is assumed that the binary tree rooted at left(i) and right(i) is max-heap, but that $A[i]$ may be smaller than its children, thus violating the max-heap property.
- **Build Heap function:** Heapify procedure is used in a bottom-up to convert an array A into a max-heap. The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \cdots n]$ are all leaves of the tree, and so each is a one-element heap to begin with. The procedure BuildHeap goes through the remaining nodes of the tree and runs Heapify on each node.

4.5.6.2 Pseudocode for Heapify Function

```
HEAPIFY(A, i)
1. le <- left(i)
2. ri <- right(i)
3. if (le<=heapsize) and (A[le]>A[i])
4. largest <- le
5. else
6. largest <- i
7. if (ri<=heapsize) and (A[ri]>A[largest])
8. largest <- ri
9. if (largest != i)
10. exchange A[i] <-> A[largest]
11. HEAPIFY(A, largest)
```

4.5.6.3 Pseudocode for BuildHeap Function

```
BUILD_HEAP(A)
1. heapsize <- length(A)
2. for i <- floor( length/2 ) down to 1
3. Heapify(A, i)
```

4.5.6.4 Pseudocode for Heap Sort

```
HEAP_SORT(A)
1.   BuildHeap(A)
2.   for i <- length(A) down to 2
3.      exchange A[1] <-> A[i]
4.      heapsize <- heapsize -1
5.      Heapify(A, 1)
```

4.5.6.5 Recurrence Relation of Heapify Function

The recurrence relation of Heapify() is

$$T(n) \leq \left\{ T\left(\frac{2n}{3}\right) + \Theta(1) \right\}$$

4.5.6.6 Time Complexity

The time complexity of Heapify() is O(log n). The time complexity of BuildHeap() is O(n log n), and the overall time complexity of the heap sort is O(n log n).

4.5.6.7 Applications of Heap Sort

1. It sorts a nearly sorted (or k-sorted) array.
2. It sorts k largest (or smallest) elements in an array.

Example 4.11

Let {6, 5, 3, 1, 8, 7, 2, 4} be the list that we want to sort from the smallest to the largest. (**Note**: For 'building the heap' step, larger nodes do not stay below smaller node parents. They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap binary tree.)

1. Build the heap:

Heap	Newly Added Element	Swap Elements
Nil	16	
16	20	
20 16		16 20
20 16 11	11	
20 16 11 8	8	
20 16 11 8	18	
20 16 11 8 18		16 18
20 18 11 8 16	14	
20 18 11 8 16 14		11 14
20 18 14 8 16 11		

2. Sort the list:

Heap	Swap Elements	Delete Element	Sorted Array	Details
20 18 14 8 16 11	20, 11			Swap 20 and 11 in order to delete 20 from the heap.
11 18 14 8 16 **20**		20		Delete 20 from the heap and add to the sorted array.
11, 18, 14, 8, 16, **20**	11, 18		20	Swap 11 and 18 as they are not in order in the heap.
18 11 14 8 16 **20**	11 16		20	Swap 11 and 16 as they are not in order in the heap.
17, 15, 16, 7, 10, **19** 18 16 14 8 11 **20**	18 11		20	Swap 18 and 11 in order to delete 17 from the heap.
11 16 14 8 **18 20**		18	20 18	Delete 18 from the heap and add to the sorted array.
11 16 14 8 **18 20**	11 14		20 18	Swap 11 and 14 as they are not in order in the heap.
11 14 16 8 **18 20**	14 16		20 18	Swap 14 and 16 in order to delete 14 from the heap.
16 14 11 8 **18 20**			20 18 16	Delete 16 from the heap and add to the sorted array.
8 14 11 **16 18 20**	8 14		20 18 16	Swap 14 and 8 as they are not in order in the heap.
14 8 11 **16 18 20**	14 11		20 18 16	Swap 14 and 11 in order to delete 14 from the heap.
10, 7, **15**, **16**, **17**, **19** 11 8 **14 16 18 20**		14	20 18 16 14	Delete 14 from the heap and add to the sorted array.
11 8 **14 16 18 20**	11 8		20 18 16 14	Swap 11 and 8 in order to delete 10 from the heap.
8, **11**, **15**, **16**, **17**, **19**		11	20 18 16 14 11	Delete 11 from the heap and add to the sorted array.
7, **10**, **15**, **16**, **17**, **19** **8 11 14 16 18 20**		8	20 18 16 14 11 8	Delete 8 from the heap and add to the sorted array.
			20 18 16 14 11	Completed

4.5.7 Merge Sort

Merge sort is a divide-and-conquer algorithm. It divides the input array in two subarrays, calls itself for the two sub-arrays and then merges the two sorted arrays. The MERGE(A, p, q, r) function is used for merging the two arrays. The MERGE(A, p, q, r) is a key process that assumes that array[l$\cdots m$] and array[$m + 1 \cdots r$] are sorted and merges the two sorted sub-arrays into one. Merge sort is a stable sort. It is not an in-place sorting technique.

4.5.7.1 Pseudocode for Merge Sort

```
MERGE_SORT (A, p, r)
1. if p < r
2. Then q = floor[(p + r)/2]
3. MERGE (A, p, q)
4. MERGE (A, q + 1, r)
5. MERGE (A, p, q, r)
```

Pseudocode for MERGE function is given as follows:

```
MERGE (A, p, q, r)
1. n1 ← q − p + 1
2. n2 ← r − q
3. create arrays L[1...n1 + 1] and R[1...n2 + 1]
4. For i ← 1 to n1
5. doL[i] ← A[p + i − 1]
6. For j ← 1 to n2
7. Do R[j] ← A[q + j]
```

```
8. L[n1 + 1]  ← ∞
9. R[n2 + 1]  ← ∞
10. i ← 1
11. j ← 1
12. For k  ← p to r
13. Do if L[i]  ≤ R[j]
14. then A[k]  ← L[i]
15. i ← i + 1
16. Else A[k]  ← R[j]
17. j ← j + 1
```

4.5.7.2 Recurrence Relation of Merge Sort

The recurrence relation for merge sort is

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & n > 1 \end{cases}$$

4.5.7.3 Time Complexity

The above recurrence can be solved using either recurrence tree method or master theorem. It falls under case 2 of the master theorem, and the solution of the recurrence is O(n log n). The time complexity of merge sort in all three cases (worst, average and best) is the same, as merge sort always divides the array in two sub-arrays and takes linear time to merge the two arrays.

4.5.7.4 Applications of Merge Sort

1. Merge sort is useful for sorting linked lists in O(n log n) time. Other n log n algorithms such as heap sort and quick sort (average case n log n) cannot be applied to linked lists.
2. It is used in inversion count problem.
3. It is used in external sorting.

Note: Merge sort is not preferable for smaller-size element array.

Example 4.12

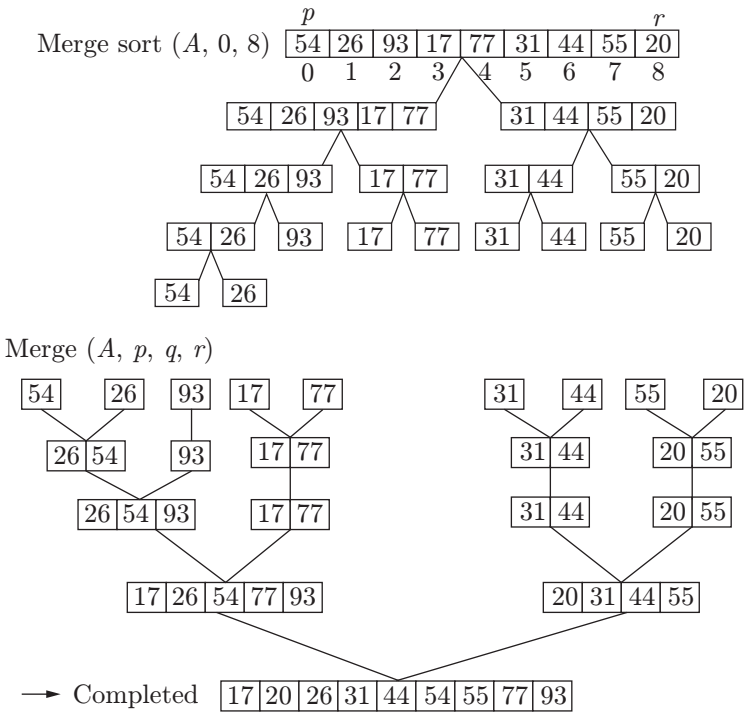
Sort the given list of elements: (54, 26, 93, 17, 77, 31, 44, 55 and 20).

Step 1: A recursive merge sort is called on the array of elements. Merge sort is a recursive function which divides an array into two parts; and calls itself recursively to divide the array into sub-parts until each part is having one element.

Step 2: After dividing the array, merge sort calls the MERGE function to put the subarray into a sorted order.

Array A:

54	26	93	17	77	31	44	55	20



4.5.8 Quick Sort

Quicksort is also divide-and-conquer strategy of sorting a list of elements like merge sort. This divides array of elements into sub-array $S[p...r]$. The concept is described as follows:

1. **Divide:** Partition $S[p...r]$ into two sub-arrays $S[p...q-1]$ and $S[q+1...r]$ such that each element of $S[p...q-1]$ is less than or equal to $S[q]$, which is, in turn, less than or equal to each element of $S[q+1...r]$. Compute the index q as part of this partitioning procedure.
2. **Conquer:** Sort the two sub-arrays $S[p...q-1]$ and $S[q+1...r]$ by recursive calls to quicksort.
3. **Combine:** Since the sub-arrays are sorted in place, no work is needed to combine them; the entire array S is now sorted.

4.5.8.1 Pseudocode for Quick Sort

```
QUICKSORT(S, P, r)
1. If p < r
2. then q <- PARTITION(S, p, r)
3. QUICKSORT(S, p, q-1)
4. QUICKSORT(S, q+1, r)
```

4.5.8.2 Pseudocode for Partition

```
PARTITION(S, p, r)
1. x <- S[r]
2. i <- p-1
3. for j <- p to r-1
4. do if S[j] <= x
5. then i <- i+1
6. swap S[i] <-> S[j]
7. swap S[i+1] <-> S[r]
8. return i+1
```

4.5.8.3 Recurrence Relation of Quick Sort

1. **For worst case:** The worst case of quick sort occurs when the pivot we picked turns out to be the least element of the array to be sorted, in every step (i.e. in every recursive call). A similar situation will also occur if the pivot happens to be the largest element of the array to be sorted. Then recurrence relation of quick sort is
$$T(n) = T(1) + T(n-1) + \Theta(n)$$
2. **For best case:** The best case of quicksort occurs when the pivot we picked happens to divide the array into two almost equal parts, in every step. Thus, we have $k = n/2$ and $n - k = n/2$ for the original array of size n. Then the recurrence relation of quick sort is given as
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

4.5.8.4 Time Complexity

The best-case and average-case running time of quick sort = $O(n \log n)$.

The worst-case running time of quick sort = $O(n^2)$.

Note: The worst case of quick sort occurs when the given elements are already sorted or almost sorted.

Example 4.13

Sort the given list of elements: (38, 81, 22, 48, 13, 69, 93, 14, 45, 58, 79, 72)

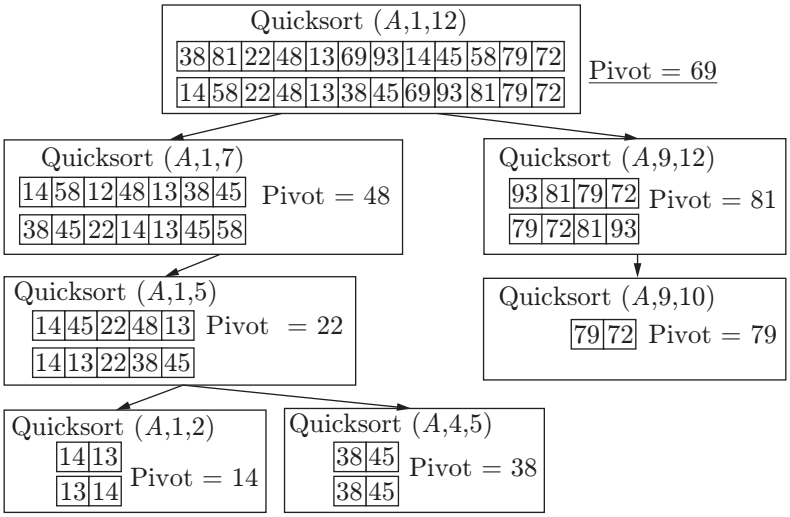
Step 1: Choose the pivot element which is at position [(left + right)/2].

Step 2: Now in partitioning phase: During the partitioning process:

(a) Elements at the left of the pivot are less than or equal to that of pivot.

(b) Elements at the right of pivot are greater than that of pivot.

The tree of recursive call is given as follows:



4.5.9 Randomized Quick Sort

In the randomized version of quick sort, we impose a distribution on the input. This does not improve the worst-case running time, but it improves the probability of getting the average-case running time.

In this version, we choose a random key for the pivot. Assume that the procedure Random(a, b) returns a random integer in the range [a, b); there are $b - a + 1$ integers in the range, and the procedure is equally likely to return one of them. The new partition procedure simply implements the swap before actually partitioning.

4.5.9.1 Pseudocode for Randomized Quick Sort

```
RANDOMIZED_PARTITION (A, p, r)

1. i ← RANDOM (p, r)
2. Exchange A[p] ← A[i]
3. return PARTITION (A, p, r)
```

The pseudocode for randomized quick sort has the same structure as quick sort, except that it calls the randomized version of the partition procedure.

```
RANDOMIZED_QUICKSORT (A, p, r)

1. If p<r then
2. Q ← RANDOMIZED_PARTITION (A, p, r)
3. RANDOMIZED_QUICKSORT (A, p, q)
4. RANDOMIZED_QUICKSORT (A, q+1, r)
```

4.5.9.2 Time Complexity

The best- and average-case running time of randomized quick sort = O(n log n).

The worst-case running time of randomized quick sort = O(n^2).

Note: The worst case of randomized quick sort occurs when all the given elements are equal.

4.5.10 Counting Sort

Counting sort is a sorting technique based on keys within a specific range. It works by counting the number of objects having distinct key values (kind of hashing), and then doing some arithmetic to calculate the position of each object in the output sequence.

4.5.10.1 Pseudocode for Counting Sort

```
COUNTING_SORT(A[], B[], k)
1. for i = 1 to k do
2. C[i] = 0
3. for j = 1 to length(A) do
4. C[A[j]] = C[A[j]] + 1
5. for 2 = 1 to k do
6. C[i] = C[i] + C[i-1]
7. for j = 1 to length(A) do
       B[C[A[j]]] = A[j]
       C[A[j]] = C[A[j]] - 1
```

Although this may look complicated, it is actually a very simple and clever algorithm.

1. An array A[ ] stores the initial data to be sorted.
2. An array C[ ] is used to count the occurrences of the data values.
3. An array B[ ] is used to store the final, sorted, list.
4. The first `for` loop initializes C[ ] to zero.
5. The second `for` loop increments the values in C[ ], according to their frequencies in the data.
6. The third `for` loop adds all the previous values, making C[ ] contain a cumulative total.
7. The fourth `for` loop writes out the sorted data into the array B[ ].

4.5.10.2 Time Complexity

O($n + k$), where n is the number of elements in the input array and k is the range of input.

Note:

1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted.
2. It is not a comparison-based sorting. Its running time complexity is O(n) with space proportional to the range of data.
3. This can be used as subroutine to some other algorithm. It is often used as a subroutine to another sorting algorithm, such as radix sort.
4. It uses a partial hashing to count the occurrence of the data object in O(1).
5. It can be extended to work for negative inputs also.

Example 4.14

Consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

The steps to sort elements through counting sort are as follows:

1. Take a count array to store the count of each unique object:
   Index: 0 1 2 3 4 5 6 7 8 9
   Count: 0 2 2 0 1 1 0 1 0 0
2. Modify the count array such that each element at each index stores the sum of the previous counts:
   Index: 0 1 2 3 4 5 6 7 8 9
   Count: 0 2 4 4 5 6 6 7 7 7

The modified count array indicates the position of each object in the output sequence.

Table 4.1 | Sorting techniques

Sorting Algorithm	Average Time	Best Time	Worst Time	Space	Stability	In-Place Sorting
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant	Stable	Yes
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant	Stable	Yes
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$	Constant	Stable	Yes
Heap sort	$O(n \times \log(n))$	$O(n \times \log(n))$	$O(n \times \log(n))$	Constant	Instable	Yes
Merge sort	$O(n \times \log(n))$	$O(n \times \log(n))$	$O(n \times \log(n))$	Depends	Stable	No
Quick sort	$O(n \times \log(n))$	$O(n \times \log(n))$	$O(n^2)$	Constant	Stable	Yes
Count sort	$O(n)$	$O(n)$	$O(n)$	Constant	Stable	No

3. Output each object from the input sequence followed by decreasing its count by 1.
   (a) Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.
   (b) Put data 1 at index 2 in the output. Decrease count by 1 to place the next data 1 at an index 1 smaller than this index.
   (c) Put data 2 at index 4 in the output. Decrease count by 1 to place the next data 2 at an index 3 smaller than this index.
   (d) Now, move to the next element 3 which has count 4, and index 4 is already filled by 2 which means element 3 does not exist in the given input list.
   (e) Follow this process for all the elements.

4.5.11 Comparison of Sorting Techniques

The comparison of the various sorting techniques is given in Table 4.1.

4.6 GRAPH

A graph $G = (V, E)$ is a set of vertices and edges where each edge consist of pair of vertices.

1. Finite set of vertices are known as nodes of graph.
2. (u, v) is a pair of vertices called edge.
3. For directed graph (u, v) is an ordered pair that represents an edge from node u to node v.

4.6.1 Types of Graph

There are two types of graphs:

1. **Simple graph:** A graph is called a simple graph (Fig. 4.5) if
   - It contains no self-loop.
   - It does not have any parallel edge.

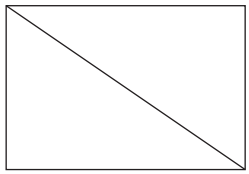


Figure 4.5 | Simple graph.

2. **Multigraph:** A graph with a self-loop and parallel edges is called a multigraph (Fig. 4.6).

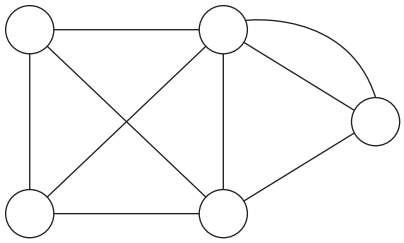


Figure 4.6 | Multigraph.

4.6.2 Types of Simple Graph

4.6.2.1 Null Graph

If in the given graph there is no edge, then it is called a null graph. In other words, a null graph does not have any edge between any two vertices (Fig. 4.7).

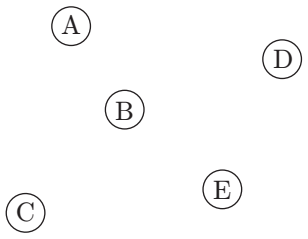


Figure 4.7 | Null graph.

4.6.2.2 Regular Graph

A graph is called a regular graph if all of its vertices have the same degree (Fig. 4.8).

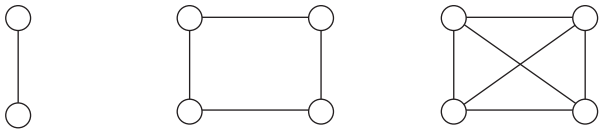


Figure 4.8 | Regular graph.

If a graph has n number of vertices and it is k-regular, where k is a positive constant, then the number of edges $= (n \times k)/2$.

4.6.2.3 Complete Graph

In the given graph, if every vertex is adjacent to all other remaining vertices, then that graph is called a complete graph (Fig. 4.9). If a graph is a complete graph and is having n vertices, then it is called a K_n graph.

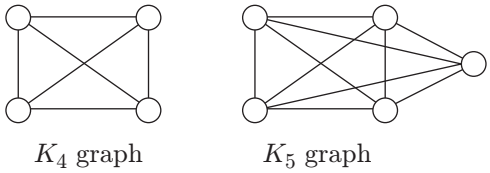


Figure 4.9 | Complete graphs.

A complete graph having n vertices has $[n \times (n-1)]/2$ edges.

Note: The number of simple graphs possible with n vertices $= 2^{[n(n-1)]/2}$.

4.6.3 Graph Representation

Graph data structure is useful in many real-time applications like they are used to represent complex networks. The network represented by graph can be of city defining paths, of telephone or circuit network.

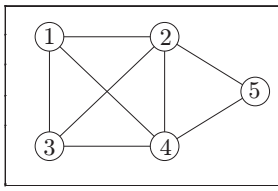
The following two are the most commonly used representations of a graph:

1. Adjacency matrix
2. Adjacency list

Incidence matrix and incidence list are two other representations for graph. Any representation can be chosen depending on the situation.

4.6.3.1 Adjacency Matrix

An adjacency matrix is a two-dimensional array of size $V \times V$, where V represents the number of vertices in a graph. If adj$[i][j] = 1$, this indicates there exists an edge between vertices i and j; otherwise, the value will be 0. The adjacency matrix for undirected graph is symmetric. Weights can also be represented by this matrix. The value adj$[i][j] = w$ means weight of edge from i to j is w.



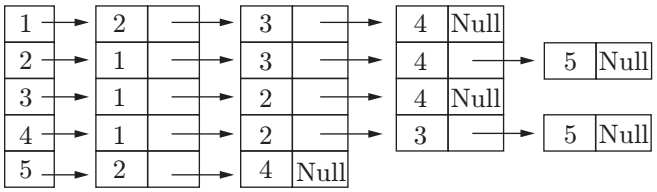
Vertex	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	1	1
3	1	1	0	1	0
4	1	1	1	0	1
5	0	1	0	1	0

This type of representation is very easy for implementation purpose. O(1) time is required to remove an edge. Testing that an edge exists between two nodes or not can be computed in O(1) time complexity.

Space requirement is more for both dense and sparse matrices is O(V^2). Addition of a vertex takes O(V^2) time.

4.6.3.2 Adjacency List

To implement adjacency list, an array of linked list is used. Array size is kept equal to the total number of vertices. For each vertex i, linked list of its adjacent nodes is stored in array$[i]$. Weights of edges can also be stored in linked list nodes. The representation of adjacency list is as follows:



The space required for adjacency list is: O(V+E), which is less than that of adjacency matrix. In worst-case, space consumed is O(V^2). Addition of vertex is easy, but finding an edge between vertices, say u and v, is not efficient. It requires O(V) complexity.

4.7 GREEDY APPROACH

A greedy algorithm is an algorithm for solving problems in an optimised way. In this approach,the algorithms execute repeatedly to maximize the outcome, considering the local conditions for some global problem. For some problems, it guarantees the optimal solution while for some it does not. The main strategy is to implement the problem so that it requires minimal resources.

The important terms are given as follows:

1. **Solution space:** The set of all possible solutions for given n inputs is called solution space.
2. **Feasible solution:** The feasible solution is one of the solutions from the solution space which satisfies the given condition.
3. **Optimal solution:** The optimal solution is one of the feasible solutions which optimizes our goal.

4.7.1 Applications of Greedy Approach

The following are the applications of the greedy approach:

1. Knapsack problem (fractional)
2. Job sequence with deadline
3. Huffman coding
4. Minimum cost spanning tree:
   a. Kruskal's algorithm
   b. Prim's algorithm
5. Singlesource shortest path:
   a. Dijkstra's algorithm
   b. Bellman–Ford algorithm

4.7.2 Fractional Knapsack

In this problem, items are given along with its weight and profit. The target is to maximize the profit considering the weight constraint. Unlike 0-1 knapsack problem, fraction of item can be considered. Fractional knapsack problem can be solved by Greedy approach.

Greedy Algorithm: The solution can be obtained by making different choices. At each stage, the best possible choice is made. In fractional knapsack, first of all profit value/weight ratios are calculated and then sorted in descending order. Item with highest value/weight ratio is chosen and added in the collection. The maximum weight is checked after adding each item. If the entire item cannot be included, then fraction of it is added to the collection.

Example 4.15

There are n items in a store. For $i = 1, 2, \ldots, n$, item i has weight $w_i > 0$ and worth $v_i > 0$. A thief can carry a maximum weight of W pounds in a knapsack. In this version of a problem, the items can be broken into smaller pieces, so the thief may decide to carry only a fraction x_i of object i, where $0 \leq x_i \leq 1$. Item i contributes $x_i w_i$ to the total weight in the knapsack, and $x_i v_i$ to the value of the load.

In symbol, the fraction knapsack problem can be stated as follows.

Maximize $nS_i = 1x_i v_i$ subject to constraint $nS_i = 1x_i w_i \leq W$. It is clear that an optimal solution must fill the knapsack exactly; otherwise, we could add a fraction of one of the remaining objects and increase the value of the load. Thus, in an optimal solution, $nS_i = 1x_i w_i = W$.

4.7.2.1 Pseudocode for Fractional Knapsack

```
Greedy-fractional-knapsack (w, v, W)

1. for i =1 to n
2. Do x[i] =0
3. weight = 0
4. while weight < W
5. Do i = best remaining item
6. if ( weight + w[i] ≤ W)
7. Then x[i] = 1
8. weight = weight + w[i]
9. else
10. x[i] = (w - weight) / w[i]
11. weight = W
12. Return x
```

4.7.2.2 Time Complexity

If the ratio of v_i/w_i is already sorted in decreasing order, then the time taken by the while loop will be O(n). So, the total time required will be O(n log n). If heap data structure is used, which keeps highest value/weight ratio at root. The construction of heap will take O(n) time and while loop will take O(log n) time (heap property is restored for each root removal). This does not improve the worst case, but it is faster if small number of items are to be filled in knapsack.

Example 4.16

Consider the following details:

1. Knapsack size = 20
2. Number of objects = 3

Object	Obj1	Obj2	Obj3
Profit	25	24	15
Weight	18	15	10

Solution:

Step 1: Calculate ratio (profit/weight):

Object	Obj1	Obj2	Obj3
Profit	25	24	15
Weight	18	15	10
Ratio	1.38	1.6	1.5

Step 2: Put according to decreasing ratio (profit/weight):

Object	Obj2	Obj3	Obj1
Profit	24	15	25
Weight	15	10	18
Ratio	1.6	1.5	1.38

Step 3: Put items into the knapsack till the knapsack is full:

(a) Put object 2 into the knapsack; it will add (value = 24 and weight = 15) to the knapsack. Knapsack has 5 units capacity remaining.
(b) Now object 3 cannot be put completely into the knapsack, so put a fraction of it. Add (value = 7.5 and weight = 5) to the knapsack. Knapsack is completely full, which means no other object can be put into it.

Step 4: Total knapsack value = 31.5 (which is optimum).

4.7.3 Huffman Coding

Huffman coding is a common technique of encoding, includinglossless data compression. The algorithm's output can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). Huffman coding is based on the frequency of occurrence of a data item (pixel in images). The main aim of Huffman coding is to encode the data that occurs frequently using less number of bits. Code books store codes that are constructed for images. Both the code book and encoded data are transmitted to decode the information.

Huffman codes can be constructed using following two ideas:

1. For an optimum code, symbols that have high probability should be assigned shorter code words.
2. To make optimum prefix code, two symbols that occur least frequently should be assigned same length.

4.7.3.1 Principle of Huffman Codes

1. Starting with the two least probable symbols, γ and δ, of an alphabet A, if the code word for γ is $[m]0$, the code word for δ would be $[m]1$, where $[m]$ is a string of 1s and 0s.
2. Now, the two symbols can be combined into a group, which represents a new symbol ψ in the alphabet set.
3. The symbol ψ has the probability $P(\gamma) + P(\delta)$. Recursively, determine the bit pattern $[m]$ using the new alphabet set.

4.7.3.2 Pseudocode for Huffman Coding

```
HUFFMAN(f[1...n])
1. T = empty binary tree
2. Q = priority queue of pairs (i, f[i]),
   i = 1...n, with f as comparison key
3. for each k = 1...n   – 1
4. i = extractMin(Q)
5. j = extractMin(Q)
6. f[n + k] = f[i] + f[j]
7. insertNode(T, n + k) with children i, j
8. insertRear(Q,(n + k, f[n + k]))
9. return T
```

4.7.3.3 Time Complexity

From the above pseudocode, we can see that with each iteration,the problem size is reduced by 1. Hence, there are exactly n iterations. The ith iteration consists of locating the two minimum values in a list of length $n - i + 1$. This is a linear operation, and so Huffman's algorithm clearly has a time complexityof $O(n^2)$.

However, it would be faster to sort the weights initially, and then maintain two lists. The first list consists of weights that have not been combined yet, and the second list consists of trees that have been formed by combining weights. This initial ordering is obtained at the cost of $O(n \log n)$. Obtaining the minimum two trees at each step then consists of two comparisons (comparing the heads of the two lists, and then comparing the larger item with the item after the smaller). The ordering of the second list can be maintained cheaply by using a binary search to insert new elements. Since at step i there are $i - 1$ elements in the second list, $O(\log i)$ comparisons are needed for insertion. Over the entire duration of the algorithm, the cost of keeping this list sorted is $O(n \log n)$. Therefore, the overall time complexity of Huffman's algorithm is $O(n \log n)$.

Example 4.17

In the following example five different symbols are given along with their frequency.

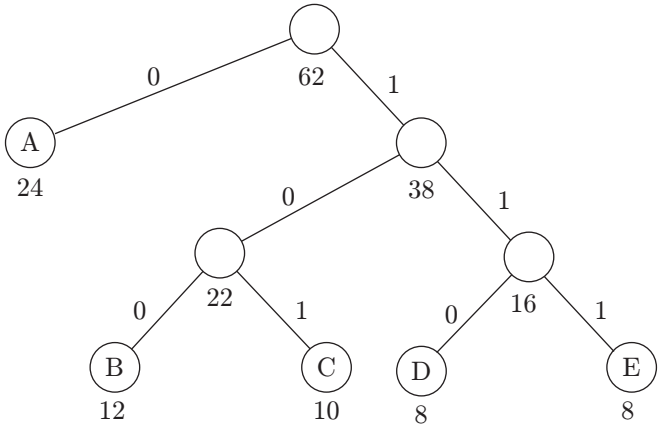
Symbol	Frequency
A	24
B	12
C	10
D	8
E	8

There are a total of 186 bits (with 3 bits per code word).

Here, the least frequency symbols are E and D. First of all, we will connect them and frequency of node will become 16. Then B and C are combined to make the

frequency 22. These will be two new nodes. In the next step, these two nodes are brought together and act as subordinate to the root node. A also acts as subordinate to root node.

4.7.3.4 Code Tree According to Huffman



Symbol	Frequency	Code	Code Length	Total Length
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24

The total length is 138 bits.

4.7.4 Minimum Spanning Tree

Spanning tree is a sub-graph that connects all the vertices together. A single graph can have multiple spanning trees. Minimum spanning tree for a weighted undirected graph is the spanning tree with minimum total weights. The weight of spanning tree is addition of weights of all the edges in the spanning tree. An MST with N vertices, has $N - 1$ edges for a given graph.

4.7.4.1 Prim's Algorithm

Prim's algorithm is a method to find minimum spanning tree. In this algorithm minimum spanning tree formed should always be connected. Prim's algorithm is an example of Greedy approach.

Properties of Prim's Algorithm

Prim's algorithm has the following properties:

1. Prime's algorithm always results into a single tree.
2. Tree grows until it covers all the vertices of the graph.
3. At each step, an edge is added to the graph that has minimum weight and is the neighbour to the previous vertices.

Steps of Prim's Algorithm

1. Find the minimum weight edge from graph and mark it.
2. Consider the neighboring edges of the selected edges and mark the minimum weight edge among those edges.
3. Continue this until we cover n-1 edges.
4. The resultant tree is minimum spanning tree and it should always be connected.

Input: Undirected connected weighted graph $G = (V, E)$ in adjacency list representation, source vertex s in V and w is an array representing weights of edges.

Output: $p[1\cdots|V|]$, representing the set of edges composing an MST of G.

Note: The field $\pi(v)$ names the parent of v in the tree.

Key(v) is the minimum weight of any edge connecting v to a vertex in tree. Key(u)←∞ means there is no such edge.

```
PRIM (G, w, s)
1. for each u in V(G)
2. do key(u) ← ∞
3. π(v) ← NIL
4. key [r] ← 0
5. Q ← V [G]
6. while Q != empty
7. do u ← Extract-Min(Q)
8. for each v in Adjacent[u]
9. do if v ∈ Q and w(u, v) < key [u]
10. then π[v] ← u
11. key [v] ← w(u, v)
```

Time Complexity

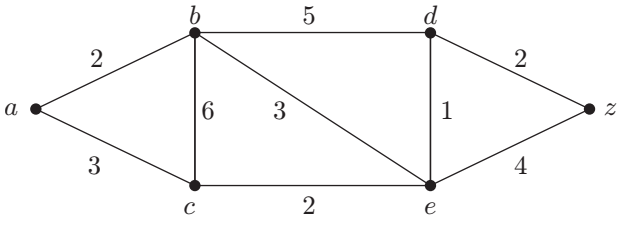
Prims algorithm use min heap data structure and its time complexity is O($V\log V+E\log V$).

Maximum edges in tree, $E = V - 1$

Therefore, O($V\log V + (V - 1)\log V$) = O($2V\log V - \log V$)

Thus, complexity = O($V\log V$)

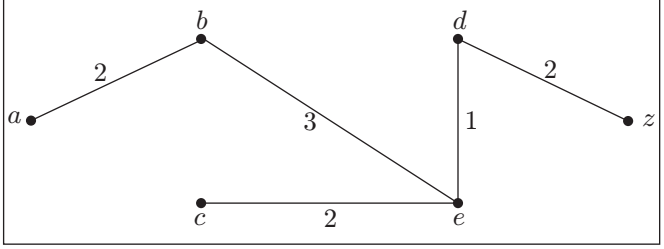
Problem 4.5: Use Prim's algorithm to find an MST in the following weighted graph. Use alphabetical order to break ties.



Solution: Prim's algorithm will proceed as follows.

1. First we add edge $\{d, e\}$ of weight 1.
2. Next, we add edge $\{c, e\}$ of weight 2.
3. Next, we add edge $\{d, z\}$ of weight 2.
4. Next, we add edge $\{b, e\}$ of weight 3.
5. And finally, we add edge $\{a, b\}$ of weight 2.

This produces a MST of weight 10.



4.7.4.2 Kruskal's Algorithm

One another algorithm to find the minimum spanning tree is Kruskal's. This method considers all minimum edges of graph one by one in increasing order and gives a resultant minimum spanning tree. In Kruskal's algorithm, it is not necessary for a tree to be connected.

Implementation steps:

1. Start from the source node.
2. Scan the weights and include the minimum weights to the tree in increasing order.
3. Stop if $N - 1$ edges are included.

Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Data Structure

Before formalizing the above idea, let us quickly understand the disjoint-set data structure:

1. **Make-SET(v):** It creates a new set whose only member is pointed to by v. Note that for this operation, v must already be in a set.
2. **FIND-SET(v):** It returns a pointer to the set containing v.
3. **UNION(u, v):** It unites the dynamic sets that contain u and v into a new set that is a union of these two sets.

Algorithm

Starting with an empty set A, select the shortest edge that has not been chosen or rejected at every step, regardless of the position of this edge in the graph. The pseudocode for Kruskal algorithm is given:

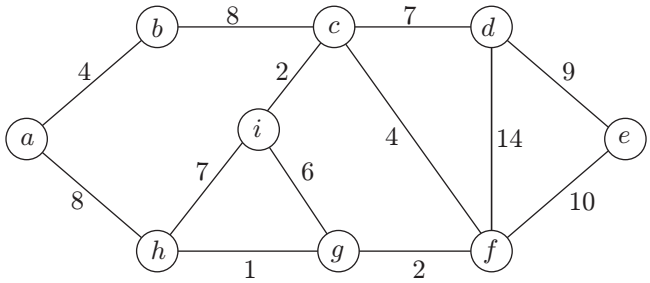
```
KRUSKAL(V, E, w)
1. A ← {φ} ▷ Set A will ultimately contain the
   edges of the MST
2. For each vertex v in V[G]
3. Do MAKE-SET(v)
4. Sort E into non-decreasing order by weight w
5. For each edge (u, v)∈ E, taken in
   non-decreasing order by weight w
6. do if FIND-SET(u)  ≠ FIND-SET(v)
7. Then A ← A ∪ {(u, v)}
8. UNION(u, v)
9. Return A
```

Time Complexity

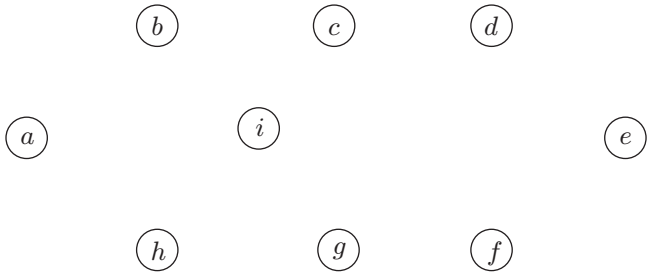
The edge weight can be compared in constant time. Initialization of priority queue takes $O(E \log E)$ time by repeated insertion. At each iteration of `while` loop, minimum edge can be removed in $O(\log E)$ time, which is $O(\log V)$, since the graph is simple. The total running time is $O((V + E) \log V)$, which is $O(E \log V)$ since the graph is simple and connected.

Example 4.18

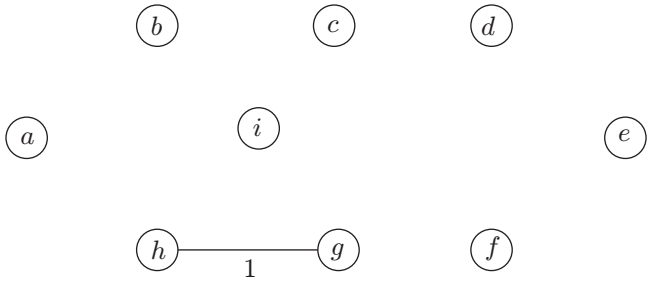
Let us use the Kruskal's algorithm to solve the following example.



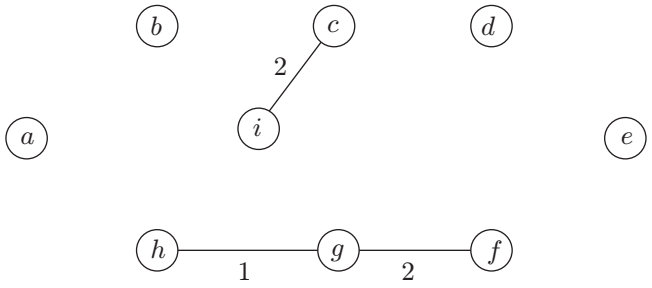
Step 1: Take only the vertices of graph.



Step 2: Choose the minimum weight edge from the original graph and draw that edge in disconnected graph containing n connected components as shown below:



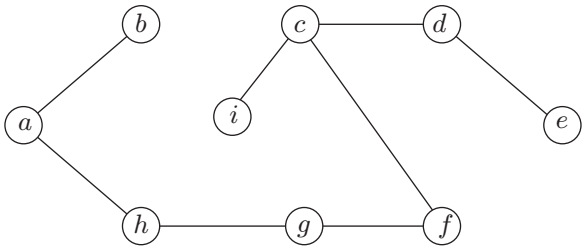
Step 3: Now select the next minimum weight edge and draw the corresponding edge in disconnected graph.



Here, there are two edges containing weight 2, add all possible edges of same weight until they form loop in the tree (since tree doesnot contain any loop but a graph may contain). That is why edge(g,f) and edge(i,c) both are added.

Step 4: Continue this process and check for the next smallest edge and add it into tree. Do this till we get $(n-1)$ edges in the tree.

The final minimum spanning tree would be:



4.7.5 Single-Source Shortest Path

Given a weighted graph G, find the shortest path from a given vertex to every other vertex in G. Note that we can solve this problem quite easily with BFS traversal in the special case when all weights are equal. The Greedy approach to this problem is repeatedly selecting the best choice from those available at that time.

4.7.5.1 Dijkstra's Algorithm

Dijkstra's algorithm is a single source shortest path algorithm. It calculates the shortest distance starting from the start node till another node in the graph. While calculating the shortest distance, it excludes the longer distance paths. Dijkstra's algorithm does not operate on negative weights. For calculating shortest distance with negative edges, Bellman-Ford algorithm is used. Steps to perform Dijkstra's algorithm are as follows:

1. Assign all the nodes with distance infinity and 0 to initial node.Distance of start node s permanent and of all the other nodes is temporary.Set start node as active node.
2. Calculate the temporary distance of neighbours of active nodes. Distance is calculated by adding up the weights associated with those edges.
3. Update the distance, and set the current node an antecessor if such a calculated distance of a node is smaller as the current one. This step is also called update and is Dijkstra's central idea.
4. Set the node as active node which has minimal temporary distance. Mark the distance as permanent.
5. Repeat steps 2 to 4 until there are no nodes left with a permanent distance, whose neighbours still have temporary distances.

Dijkstra's Pseudocode

```
DIJKSTRA (G , S):
1.  for each vertex v in Graph:
2.  distance[v] := infinity
3.  previous[v] := undefined
4.  distance[ S ] := 0
5.  Q := the set of all nodes in Graph
6.  While Q is not empty:
7.  u := node in Q with smallest distance[ ]
8.  remove u from Q
9.  for each neighbour v of u:
10. alt := distance[u] + distance_between(u, v)
11. If alt < distance[v]
12. distance[v] := alt
13. previous[v] := u
14. Return previous[ ]
```

Time Complexity

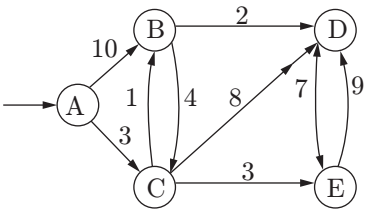
Like Prim's algorithm, Dijkstra's algorithm runs in $O(E \log V)$ time.

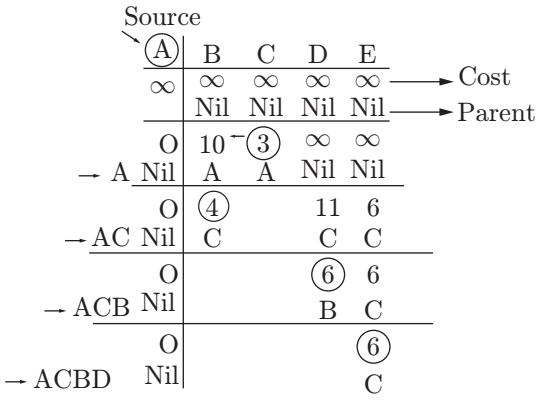
1. Dijkstra's algorithm with list: $O(V^2)$
2. Dijkstra's algorithm with modified binary heap: $O((E+V)\log V)$
3. Dijkstra's algorithm with Fibonacci heap: $O(E + V \log V)$

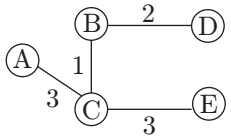
Example

The following are the steps:

1. Start from the source node; find the distance of all nodes reachable from the source.
2. Then pick the node which has the minimum distance, from the pool of nodes.
3. Calculate again the distance from the selected node, and follow the above two steps till the entire nodes are selected.







4.7.5.2 Bellman–Ford Algorithm

The Bellman–Ford algorithm is used to compute single source shortest path to all other vertices of weighted directed graph. This algorithm also considers negative weights of graph edges. This algorithm is slower but is capable of even finding a negative weight cycle in graph.

Bellman-Ford Pseudocode

```
1. d[s] ← 0
2. for each v ∈ V - {s}
3. do d[v] ← ∞
4. for i ← 1 to |V| - 1 do
5. for each edge (u, v) ∈ E do
6. if d[v] > d[u] + w(u, v) then
7. d[v] ← d[u] + w(u, v)
8. π[v] ← u
9. for each edge (u, v) ∈ E
10. do if d[v] > d[u] + w(u, v)
11. then report that a negative-weight
    cycle exists
```

Time Complexity

The time complexity of Bellman-Ford algorithm is $O(VE)$, where $E = V^2$, so the worst-case running time of Bellman–Ford algorithm is $O(V^3)$.

4.8 GRAPH TRAVERSAL

Graph traversal means visiting all the nodes in a graph in some order. In graph traversal, some nodes may be visited more than once. This is because it is not necessary to know that the node has been explored before or not. The re-traversal becomes more for dense graph, which increase computation time. Tree traversal is a special case of graph traversal. The following are two graph traversal methods:

1. Breadth-first traversal (BFT)
2. Depth-first traversal (DFT)

4.8.1 Breadth-First Traversal

The breath-first traversal starts from the root node and then traverses all its neighbours. Then for each neighbour, its corresponding unvisited neighbour is processed. The pseudocode for BFT is given as follows:

```
BST (V)
1. visited(V) =1
2. add(V, Q)
3. while(Q is not empty)
4. u = delete(Q)
5. for all x adjacent to u
6. if( x is not visited)
7. visited(x) = 1
8. add(Q, x)
```

4.8.1.1 Time Complexity

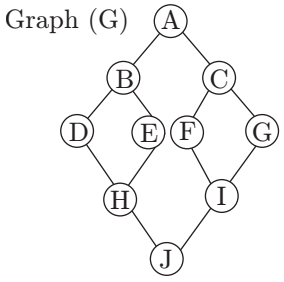
The time complexity of BFT is $O(V + E)$, where V is the number of vertices in the graph and E is the number of edges in the graph.

4.8.1.2 Application of BFT

The following are the applications of BFT. It can be used to find out

1. whether the given graph is connected.
2. the number of connected components in a graph.
3. whether the given graph contains a cycle.
4. the shortest path if all the edges have the same weight.
5. whether the given graph is a bipartite graph.

Problem 4.6: Consider the graph $G = (V, E)$ and find the BFT starting node A.



Solution: A is the starting node and Q is a queue data structure to store the values.

BFT (A):

1. Visited (A) = 1
Add (Q, A)

A				

2. u = delete (Q) → A
For all adjacent nodes to A which are not already added in the queue, Add (Q, X), where X = (B, C) is the adjacent node to A.

B	C			

3. u = delete (Q) → B
For all adjacent nodes to B which are not already added in the queue, Add (Q, X), where X = (D, E) is the adjacent node to B.

C	D	E		

4. u = delete (Q) → C
For all adjacent nodes to C which are not already added in the queue, Add (Q, X), where X = (F, G) is the adjacent node to C.

D	E	F	G	

5. u = delete (Q) → D
For all adjacent nodes to D which are not already added in the queue, Add (Q, X), where X = (H) is the adjacent node to D.

E	F	G	H	

6. u = delete (Q) → E
For all adjacent nodes to E which are not already added in the queue, Add (Q, X), where X = (Φ) is the adjacent node to E.

F	G	H		

7. u = delete (Q) → F
For all adjacent nodes to F which are not already added in the queue, Add (Q, X), where X = (I) is the adjacent node to F.

G	H	I		

8. u = delete (Q) → G
For all adjacent nodes to G which are not already added in the queue, Add (Q, X), where X = (I) is the adjacent node to G.

H	I			

9. u = delete (Q) → H
For all adjacent nodes to H which are not already added in the queue, Add (Q, X), where X = (J) is the adjacent node to H.

I	J			

10. u = delete (Q) → I
For all adjacent nodes to I which are not already added in the queue, Add (Q, X), where X is the adjacent node to I.

J				

11. u = delete (Q) → J
For all adjacent nodes to J which are not already added in the queue, Add (Q, X), where X is the adjacent node to J.

12. The queue is empty, which means graph is traversed completely.

4.8.2 Depth-First Traversal

As already discussed, there are many ways of tree traversal. Consider the tree given in Fig. 4.10:

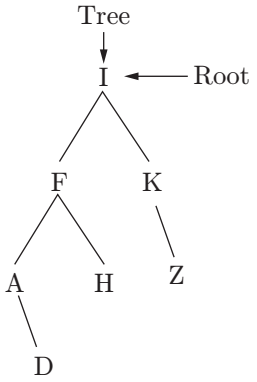


Figure 4.10 | DFT of a tree.

Preorder traversal for given tree: I F A D H K Z

Preorder traversal also results in depth first traversal. This is because the traversal goes deeper before traversing its siblings. In this tree, the descendants of F, that is, A, H, D are traversed first and after that sibling of F, that is, K are traversed.

4.8.2.1 Pseudocode for DST

```
DFT (V)
1. visited (V) =1
2. for all x adjacent to V
3. if (x is not visited)
4. visited (x) =1
5. DFT (x)
```

The above is the recursive code for DFT.

4.8.2.2 Time Complexity

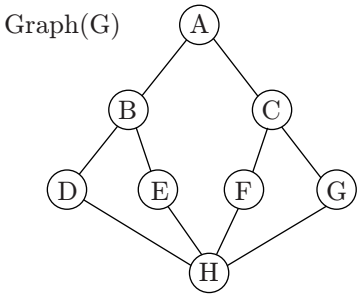
The time complexity of DFT is O($V + E$), where V is the number of vertices in the graph and E is the number of edges in the graph.

4.8.2.3 Application of DFT

The following are the applications of BFT. It can be used to find out

1. whether the given graph is connected.
2. the number of connected components in a graph.
3. whether the given graph contains a cycle.
4. whether the given directed graph is strongly connected.

Problem 4.7: Consider the graph $G = (V, E)$ and find the DFT starting node A.



Solution: A is the starting node and S is a stack data structure to store the values. Stack is internally used by the recursive call.

```
DFT (A)
1. Visited (A) =1
PUSH(S, A)
```

A				

2. POP(S) → A
For all adjacent nodes to A which are not already added in the stack, `PUSH (S, X)`. Every neighbour element will be stored in the stack from right to left. Here, X = (B, C) is the adjacent node to A.

B	**C**			

3. POP(S) → B
For all adjacent nodes to B which are not already added in the stack, `PUSH (S, X)`. Every neighbour element will be stored in the stack from right to left. Here, X = (D, E) is the adjacent node to B.

D	**E**	**C**		

4. POP (S) → D
For all adjacent nodes to D which are not already added in the stack, `PUSH (S, X)`. Every neighbour element will be stored in the stack from right to left. Here, X = (H) is the adjacent node to D.

H	**E**	**C**		

5. POP (S) → H
For all adjacent nodes to H which are not already added in the stack, `PUSH (S, X)`. Every neighbour element will be stored in the stack from right to left. Here, X = (F, G) is the adjacent node to H.

F	**G**	**E**	**C**	

6. POP (S) → F
For all adjacent nodes to F which are not already added in the stack, `PUSH (S, X)`. Every neighbour element will be stored in the stack from right to left. Here, X = (Φ) is the adjacent node to F.

G	**E**	**C**		

7. POP (S) → G
For all adjacent nodes to G which are not already added in the stack, `PUSH (S, X)`. Every neighbour element will be stored in the stack from right to left. Here, X = (Φ) is the adjacent node to G.

E	**C**			

8. POP (S) → E
For all adjacent nodes to E which are not already added in the stack, `PUSH (S, X)`. Every neighbour element will be stored in the stack from right to left. Here, X = (Φ) is the adjacent node to E.

C				

9. POP (S) C
For all adjacent nodes to C which are not already added in the stack, `PUSH (S, X)`. Every neighbour element will be stored in the stack from right to left. Here, X = (Φ) is the adjacent node to C.

10. The stack is empty, which means the graph is traversed completely.

4.9 DYNAMIC PROGRAMMING

Dynamic programming is a method for solving complex problems by breaking them down into simpler sub-problems. It is applicable to problems exhibiting the properties of overlapping sub-problemsand optimal substructure.

Dynamic programming algorithms are used for optimization (e.g. finding the shortest path between two points, or the fastest way to multiply many matrices). A dynamic programming algorithm will examine all possible ways to solve the problem and will pick the best solution.

4.9.1 Applications of Dynamic Programming

The following are the applications of dynamic programming:

1. Fibonacci series
2. 0/1 knapsack
3. Largest common subsequence
4. All-pair shortest path

4.9.2 Fibonacci Series

Fibonacci series is the sequence produced by adding the previous two numbers in the sequence. First two numbers in sequences are 0 1 or 1 1. The recurrence relation to find the next term in the sequence is given below:

$$F_n = F_{n-1} + F_{n-2}$$

with initial values $F_1 = 0$ and $F_2 = 1$ or $F_1 = 1$ and $F_2 = 1$.

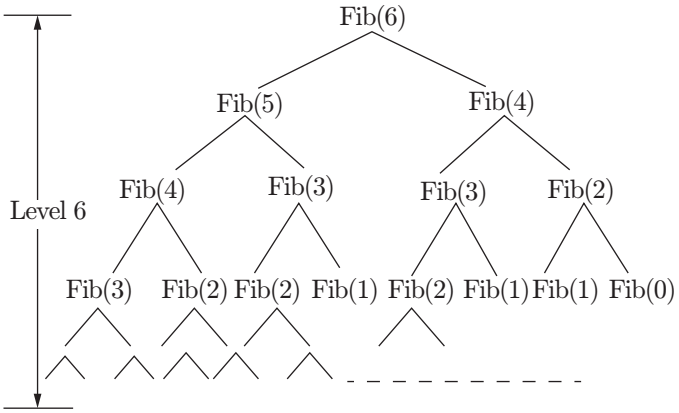
Example: Fibonacci series 0, 1, 1, 2, 3, 5, 8, 13, 21, ….

4.9.2.1 Recurrence Relation of Fibonacci

$$\text{fib}(n) = \begin{cases} 0, & \text{if } n < 0 \\ 1, & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2), & \text{if } n > 1 \end{cases}$$

Example 4.19

The solution of Fibonacci through recursion tree is shown as follows:



Total number of nodes = $(2^n - 1)$.

So, the total number of function calls = 2^n, and one function call takes O(1) time.

The total running time = $O(2^n)$.

Note: Most of the recursive programs contain very less number of distinct calls and more number of repeated sub-problems or function calls. These problems are known as overlapping sub-problems.

4.9.2.2 Dynamic Solution of Fibonacci

In dynamic programing technique, we solve every problem exactly once and store the results in memory. That result is used later to get an optimum output (Fig. 4.11).

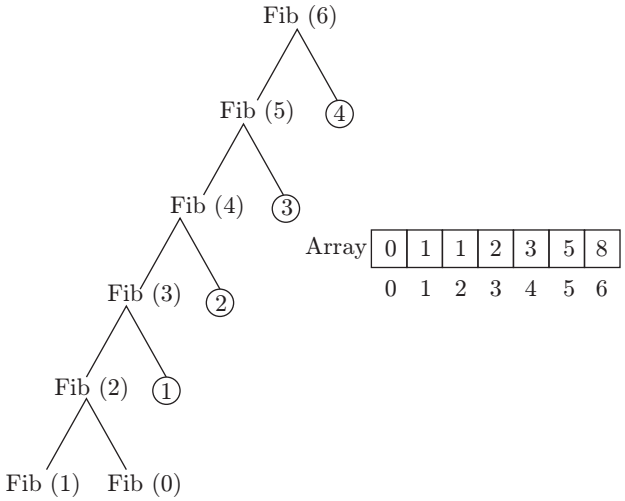


Figure 4.11 Dynamic programming approach for Fibonacci.

Pseudocode for Dynamic Solution of Fibonacci Series

```
1. Algorithm Fast-Fibonacci(n)
2. Let fib[0] and fib[1] be 1
3. for each i from 2 to n, do
4. Let fib[i] = fib[i - 2] + fib[i - 1]
5. end of loop
6. return fib[n]
```

Time Complexity

1. Fibonacci series without dynamic programing [space = $O(n)$ and time = $O(2^n)$]
2. Fibonacci series with dynamic programing [space = $O(n)$ and time = $O(n)$]

4.9.3 0/1 Knapsack

One of the problems in combinatorial optimization is knapsack problem. In this problem, the target is to maximize the profit by choosing items. Each item is associated with a weight and profit. The collection of items is formed such that the chosen total weight should be less than or equal to the given limit. The profit can be as large as possible.

4.9.3.1 Recurrence Relation 0/1 Knapsack

The recurrence relation of 0/1 Knapsack problem is

$$\mathrm{KS}(n,m)$$

$$= \begin{cases} 0, & \text{if } m = 0 \text{ or } n = 0 \\ \mathrm{KS}(n-1,m) & \text{if } m < w[n] \\ \mathrm{Max}[\{\mathrm{KS}(n-1,m-w[n]+p[n])\} & \text{otherwise} \\ \quad \{\mathrm{KS}(n-1,m)+0\}], & \end{cases}$$

4.9.3.2 Time Complexity

The time complexity of the knapsack problem is as follows:

1. Without dynamic programing = $\mathrm{O}(2^n)$
2. With dynamic programing = $\mathrm{O}(m^n)$

It is one of the NPC problems.

4.9.4 Longest Common Subsequence

A subsequence of a given sequence is just the given subsequence in which zero or more symbols are left out.

Let sequence $(S) = \{A, B, B, A, B, B\}$, subsequence $(S_1) = \{A, A\}$, subsequence $(S_2) = \{A, B, A, B\}$ and subsequence $(S_3) = \{B, A, B, A\}$.

4.9.4.1 Common Subsequence

In the given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y, if Z is a subsequence of both X and Y.

Example 4.20

$X = \{A, B, B, A, B, B\}$

$Y = \{B, A, A, B, A, A\}$

$Z = \{B, B, A\}$ is a common subsequence. But $Z = \{A, B, A, B\}$ is not a common subsequence.

4.9.4.2 Recurrence Relation of LCS

The recurrence relation of longest common subsequence is

$$\mathrm{LCS}(i,j)$$

$$= \begin{cases} 0, & \text{if } i=0 \text{ or } j=0 \\ 1+\mathrm{LCS}(i-1,j-1), & \text{if } X[i]=Y[j] \\ \mathrm{Max}[\{\mathrm{LCS}(i-1,j)\}\{\mathrm{LCS}(i,j-1)\}], & \text{if } X[i]\neq Y[j] \end{cases}$$

4.9.4.3 Time Complexity

Complexity	LCS with Recursive Solution	LCS with Dynamic Programing
Time	If $(m = n)$ LCS $(m, n) = \mathrm{O}(2^n)$ Else LCS$(m, n) = \mathrm{O}(2^{m+n})$	$\mathrm{T}(n) = \mathrm{O}(m^n)$
Space	$\mathrm{O}(m + n)$	$\mathrm{O}(mn)$

4.10 ALL-PAIR SHORTEST PATH

Given a directed graph $G = (V, E)$, where each edge (v, w) has a non-negative cost $C[v, w]$, for all pairs of vertices (v, w), find the cost of the lowest-cost path from v to w. It is a generalization of the single-sourceshortest-path problem.

4.10.1 Floyd's Algorithm

Let $A^k(i, j)$ be the minimum cost required to go from vertex i to vertex j for which all intermediate vertices are in the set $(0, 1, 2, 3, \ldots, k)$ (Fig. 4.12).

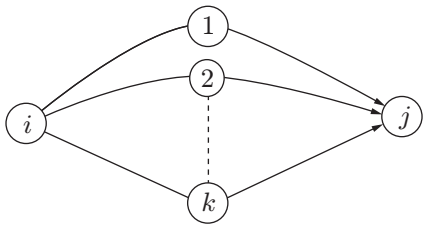


Figure 4.12 Different paths from i to j.

4.10.1.1 Recurrence Relation of Floyd's Algorithm

The recurrence relation of Floyd's algorithm is

$$A^k(i,j) = \mathrm{Min}\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}$$

4.10.1.2 Time Complexity

The running time complexity: $T(n) = \mathrm{O}(V^3)$.

4.11 CONCEPTS OF COMPLEXITY CLASSES

4.11.1 P-Complexity Class

P in computational complexity theory is known as polynomial time or deterministic time. This is the most basic complexity class. This consists of all the problems that

are solvable in polynomial time. These problems can be solved in $O(n^k)$ time, where k is some constant. In class P, problems are efficiently solvable or traceable.

A language L is in class P iff there exists some Turing machine M such that:

1. Turing machine runs in polynomial time for all the inputs.
2. For all y in L, M must return 1, otherwise 0.

P includes many natural problems like linear programming, finding maximum matching and calculating the greatest common divisor. Prime number is also a part of class P now.

4.11.2 NP Complexity Class

NP class consists of problems that are verifiable in polynomial time. This class contains those languages that can be defined by second order logic.

Note: The most important open question in complexity theory is the P = NP problem (Fig. 4.13).

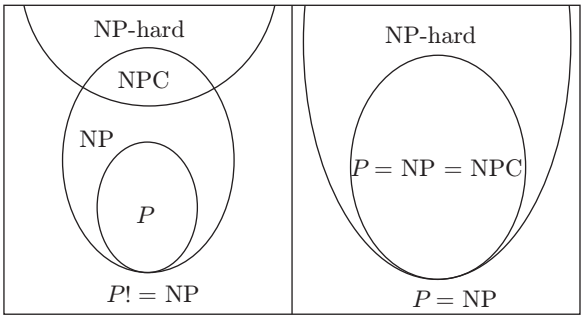


Figure 4.13 Relation between P, NP, NP-complete and NP-hard problems.

4.11.3 NP-Complete

These are the hardest problems in NP. In the computational complexity theory, the complexity class NP-complete (abbreviated NP-C or NPC) is a class of decision problems. A decision problem L is NP-complete if it is in the set of NP problems and also in the set of NP-hard problems.

NP-complete is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time; NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a non-deterministic TM. A problem p in NP is also NP-complete if every other problem in NP can be transformed into p in polynomial time.

The following list contains some well-known problems that are NP-complete when expressed as decision problems:

1. Boolean satisfiability problem (SAT)
2. Knapsack problem
3. Hamiltonian path problem
4. Travelling salesman problem
5. Subgraph isomorphism problem
6. Subset sum problem
7. Clique problem
8. Vertex cover problem
9. Independent set problem
10. Dominating set problem
11. Graph colouring problem

4.11.4 NP-Hard

NP-hard is a class that includes problems which are "at least as hard as hardest problem in NP". A problem is said to be NP hard if

1. It is in NP
2. It is NP-complete

This means a problem H is NP-hard if any problem L has polynomial time reduction from L $\rightarrow$ H. The problem can be a decision problem, optimization problem or a search problem.

Examples of NP-hard problem is the traveling salesman problem. It is an optimization problem of finding the least-cost cyclic route through all nodes of a weighted graph. There are decision problems that are NP-hard but not NP-complete, for example, the halting problem. This is the problem which asks, 'given a program and its input, will it run forever?' That is a yes/no question, so this is a decision problem.

Note:

1. Let 'A' be an NP-complete problem and 'B' be an unknown problem; if 'A' is a polynomial that reduces to 'B' ($A \leq_p B$), then 'B' is NP-hard.
2. Let 'A' be an NP-hard problem and 'B' be an unknown problem; if 'A' is a polynomial that reduces to 'B' ($A \leq_p B$), then 'B' is NP-hard.
3. Let 'A' be a P-class problem and 'B' be an unknown problem; if ($B \leq_p A$), then 'B' is a P-class problem.
4. Let 'A' be a P-class problem and 'B' be an unknown problem; if ($B \leq_p A$), then 'B' is an NP-class problem.

Problem 4.8: Both P and NP are closed under the operation of:

(A) Union (B) Intersection
(C) Kleene's closure (D) None of these

Solution: (D) Both P and NP are closed under concatenation and Kleene's closure. The Kleene closure of a language L, denoted by L^* is defined as

$$\{w_1 w_2 \ldots w_k - k \in N,\ w_i \in L \forall i = 1, 2, \ldots, k\}.$$

P is closed under Kleene's closure as for all $L \in P$, $L^* \in P$.

NP is closed under Kleene's closure as for all $L \in$ NP, $L^* \in$ NP.

IMPORTANT FORMULAS

1. Theta notation (Θ): $0 < c_1(g(n)) \le f(n) \le c_2(g(n))$ $\forall n \ge n_0$
2. Big-O notation (O): $0 < f(n) \le c(g(n))$ $\forall n \ge n_0$.
3. Omega notation (Ω): $0 < c(g(n)) \le f(n)$ $\forall n \ge n_0$.
4. Small-o notation (o): $0 < f(n) < c(g(n))$ $\forall n \ge n_0$.
5. Small omega notation (ω): $0 < c(g(n)) < f(n)$ $\forall n \ge n_0$.
6. Master theorem: $T(n) = aT(n/b) + f(n)$.

   Case 1: $f(n) = O(n^{\log_b a - \varepsilon})$, then $T(n) = \Theta(n^{\log_b a})$.

   Case 2: $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

   Case 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$.

   $af(n/b) \le cf(n)$ for some constant $c < 1$ and large n, then $T(n) = \Theta(f(n))$.
7. Linear search: $O(n)$.
8. Binary search: $\Theta(\log n)$.
9. Bubble sort: $O(n^2)$.
10. Selection sort: $O(n^2)$.
11. Insertion sort: $O(n^2)$.
12. Heap sort: $O(n \log n)$.
13. Merge sort: $O(n \log n)$.
14. Quick sort: $O(n^2)$ (worst case).
15. Quick sort: $O(n \log n)$ (average case).
16. Counting sort: $O(n + k)$.
17. If a graph has n number of vertices and it is k-regular, where k is a positive constant, then the number of edges $= (n \times k)/2$.
18. A complete graph having n vertices has $[n\ (n-1)]/2$ edges.
19. Number of simple graphs possible with n vertices $= 2^{[n(n-1)]/2}$.
20. Prim's algorithm: $O(E \log V)$.
21. Kruskal's algorithm: $O((V + E) \log V)$.
22. Dijkstra's algorithm: $O(E \log V)$.
23. Bellman–Ford algorithm: $O(V^3)$.
24. BFT and DFT: $O(V + E)$.

SOLVED EXAMPLES

1. Consider $T(n) = 6T(n/2) + n^3$; then $T(n)$ is equal to

(a) $(n/2)$ (b) $O(n^3 \log n)$
(c) $(n^2 \log n)$ (d) $O(n^3)$

Solution:
$T(n) = 6\,T(n/2) + n^3$; By using Master Theorem.
$a = 6,\ b = 2,\ f(n) = n^3$
$g(n) = (n^{\log_b a}) = n^{\log_2 6} = \Theta(n^{2.58})$
$f(n) = \Omega(n^{\log_2 6 + \varepsilon})$, $f(n)$ is larger than $g(n)$, but is not a larger polynomial.
By case 3, $T(n) = \Theta(f(n)) = n^3$

Ans. (d)

2. Solve the following recurrence relation: $T(n) = 4T(n/2) + n$.

(a) $O(n^2 \log n)$ (b) $O(n^2)$
(c) $O(n/2)$ (d) $O(\log n)$

Solution:
$T(n) = 4\,T(n/2) + n$; By using Master Theorem.
$a = 4,\ b = 2,\ f(n) = n$
$g(n) = (n^{\log_b a}) = n^{\log_2 4} = \Theta(n^2)$
$f(n) = O(n^{\log_2 4 - \varepsilon})$, $f(n)$ is smaller than $g(n)$, but is not a smaller polynomial.
By case 1, $T(n) = \Theta(n^2)$

Ans. (b)

3. Consider the following statements:

1. An algorithm is the number of steps to be performed to solve a problem.
2. An algorithm is the number of steps and their implementation in any language to a given problem to solve a problem.
3. To solve a given problem there may be more than one algorithm.

Which of the following is true?

(a) 1 is correct. (b) 2 is correct.
(c) 1 and 3 are correct. (d) 2 and 3 are correct.

Solution: Both statements 1 and 3 are correct.

Ans. (c)

4. Compilers require efficient searching strategies for which they rely on

(a) Hash tables (b) String matching
(c) Binary search tables (d) Binary search trees

Solution: Hash tables have less complexity, so they are used by the compiler for efficient searching.

Ans. (a)

5. Consider a binary search tree having n elements. The time required to search a given element will be:

(a) $O(n)$ (b) $O(\log n)$
(c) $O(n^2)$ (d) $O(n \log n)$

Solution: The time required for search operation in a binary search depends on the height of the tree. In a complete binary tree with n nodes, search operation takes $\Theta(\log_2 n)$ in worst case scenario. But if all the nodes of tree are in linear chain, then the same operations will take $O(n)$ in worst-case scenario. So, option (a) is correct.

Ans. (a)

6. The average time required to perform a successful sequential search for an element in an array $A(1: n)$ is given by

(a) $\log n$ (b) n^2
(c) $(n + 1)/2$ (d) $(n \times n - 1/2)$

Solution: The average time required for sequential search or linear search in this case is $(n + 1)/2$.

Ans. (c)

7. Which sort will operate in quadratic time relative to the number of elements in the array (on the average)?

(a) Heap sort (b) Bubble sort
(c) Radix sort (d) Merge sort

Solution: The average time for bubble sort is $O(n^2)$, so it will operate in quadratic time.

Ans. (b)

8. Binary expression tree is traversed in _____ traversal.

(a) Postorder (b) Preorder
(c) Both A and B (d) Reverse polish order

Solution: Binary tree is traversed in both preorder as well as postorder traversal.

Ans. (c)

9. The time complexity of searching an element in a linked list of length n will be

(a) $O(n)$ (b) $O(n^2 - 1)$
(c) $O(\log n)$ (d) $O(\log n^2)$

Solution: The worst-case time complexity for searching an element in a linked list is $O(n)$, so option (a) is correct. If in the question it is not mentioned about (best or worst or average) time complexity then we always calculate as worst time complexity.

Ans. (a)

10. A hashing algorithm uses linear probing; the average search time will be less if the load factor

(a) is less than one (b) equals one
(c) is greater than one (d) equals two

Solution: As load time is the ratio of number of records present to the total number of records, so is load time is less than 1, the probability of collisions decreases and hence, search time reduces.

Ans. (a)

11. If a node has K children in B tree, then the node contains exactly _____ keys.

(a) K^2 (b) $K - 1$ (c) $K + 1$ (d) $\sqrt{K}$

Solution: The number of keys $= K - 1$.

Ans. (b)

12. Big-O estimate for the factorial function, that is, $n!$, is given by

(a) $O(n!)$ (b) $O(n^n)$
(c) $O(n \times n!)$ (d) $O(\sqrt{n})$

Solution: Factorial function $f(n) = n! = n(n - 1) \times (n - 2) \ldots 1$
$n! = n\,(n - 1)(n - 2) \ldots 1 \leq n \times n \ldots \times n = (n^n)$
So, Big O Estimation of $f(n!) = O(n^n)$

Ans. (b)

13. Let A and B be two $n \times n$ matrices. The efficient algorithm to multiply the two matrices has the time complexity $O(n^x)$, where x is _____.

Solution: Strassen's surprising algorithm can multiply two $n \times n$ matrices in $\Theta(n^{\log 7})$ time, which is equal to $O(n^{2.81})$ time.

Ans. (2.81)

14. The searching technique in which less number of comparisons are done is

(a) Binary searching
(b) Linear search searching
(c) Hashing
(d) Woline's search

Solution:

Search Complexity	Average Case	Worst Case
Binary	O(log n)	O(n)
Hash table	O(1)	O(n)
Linear search	O(n)	O(n)

Thus, hash table takes less search time.

Ans. (c)

15. Which of the following statements is false about Prim's algorithm?

(a) The time complexity is O(E log V) using a binary heap.
(b) It may use binomial max-heap to represent the priority queue.
(c) The time complexity is O($E + V$ log V) using a Fibonacci heap.
(d) Initially the roots key and nodes are set to zero.

Solution:
(a) True, The time complexity is O(E log V) using a binary heap.
(c) True, Prim's algorithm's time complexity is O($E + V$ log V) using a Fibonacci heap.
(d) True, Initial distance to the roots key and nodes is kept Zero.

Only option (b) is false.

Ans. (b)

16. Express the following recurrence relation in asymptotic notations:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

(a) $\Theta(n \log n)$ (b) $\Theta(n \log^2 n)$
(c) $\Theta(n^2)$ (d) $O(n^3)$

Solution: By using Master Theorem,
$T(n) = 2T(n/2) + n$

$a = 2,\ b = 2,\ f(n) = n$

$g(n) = (n^{\log_b a}) = n^{\log_2 2} = \Theta(n)$

Order of $g(n)$ = order of $f(n)$, so by case 2, $T(n) = \Theta(f(n) \cdot \log n) = \Theta(n \log n)$

Ans. (a)

17. What is the height of n-node, k-ary heap?

(a) $\Theta(\log_k n)$ (b) $\Theta(nk)$
(c) $\Theta(k \log_2 n)$ (d) $\Theta(n/k)$

Solution: If h is the height of k-ary heap ($h \geq 2$) H, then the number of nodes 'n' in H such that

$$\frac{k(h)-1}{k-1} < n \leq \frac{k(h+1)-1}{k-1}$$

So $h = \Theta(\log_k n)$.

Ans. (a)

18. Solve the following given recurrence relation: $T(n) = 4T(n/2) + n^2$.

(a) $\Theta(n^2)$ (b) $\Theta(n^2 \log_2 n)$
(c) $\Theta(n \log_2 n)$ (d) None of the above

Solution: Using the master theorem, we have $a = 4,\ b = 2$ and $f(n) = n^2$. So

$$g(n) = n^{\log b \cdot a} = n^{\log 2 \cdot 4} = n^2$$

Here, order of $f(n)$ = order of $g(n)$, so we get

$$T(n) = O(n^{\log b \cdot a} \log_2 n) = O(n^2 \log_2 n)$$

Ans. (b)

19. Solve the following given recurrence relation: $T(n) = 16T(n/4) + n^3$.

(a) $\Theta(n \log_2 n)$ (b) $\Theta(n^2 \log_2 n)$
(c) $\Theta(n^3)$ (d) None of the above

Solution: From the master theorem, we have $a = 16,\ b = 4$ and $f(n) = n^2$. So

$$g(n) = n^{\log b \cdot a} = n^{\log 4 \cdot 16} = n^2$$

Order of $f(n)$ > order of $g(n)$, so we get

$$T(n) = O(f(n))$$

Ans. (c)

20. Consider the following two functions:

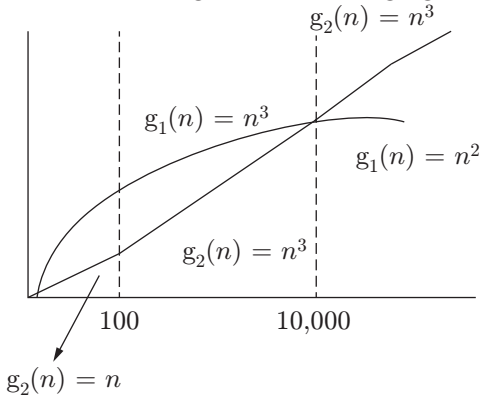
$$g_1(n) = \begin{cases} n^3, & \text{for } 0 \leq n \leq 10,000 \\ n^3, & \text{for } 0 \leq n \geq 10,000 \end{cases}$$

$$g_2(n) = \begin{cases} n, & \text{for } 0 \leq n \leq 10,000 \\ n^3, & \text{for } n > 10,000 \end{cases}$$

Which of the following is true?

(a) $g_1(n)$ is $O(g_2(n))$. (b) $g_1(n)$ is $O(n^3)$.
(c) $g_2(n)$ is $O(g_1(n))$. (d) $g_2(n)$ is $O(n)$.

Solution: Looking at the following figure:



Therefore, we get

$$n^2 \leq n^3 \quad \text{for } N \geq 10,000$$

$$g_1(n) = O(g_2(n))$$

Ans. (a)