

Tribhuvan University
Institute of Science and Technology
2079

Bachelor Level / Fifth Semester / Science
Computer Science and Information Technology (CSC314)
Design and Analysis of Algorithms

Full Marks: 60 + 20 + 20 Pass Marks: 24 + 8 + 8 Time: 3 Hours

Candidates are required to give their answers in their own words as far as practicable.

The figures in the margin indicate full marks.

Group A
Answer any TWO

1. Explain the divide-and-conquer strategy for problem-solving. Describe the worst-case linear time selection algorithm and analyze its complexity. (10)

Divide-and-Conquer Strategy for Problem-Solving

Divide-and-Conquer is a fundamental algorithmic technique where a problem is solved by breaking it down into smaller sub-problems that are similar to the original problem. The strategy involves three main steps:

1. **Divide:** Break the problem into smaller sub-problems.
2. **Conquer:** Solve each sub-problem recursively. If the sub-problems are small enough, solve them directly.
3. **Combine:** Merge the solutions of the sub-problems to form the solution to the original problem.

This approach is effective because it simplifies complex problems, allowing solutions to be built from the solutions of smaller problems. Common examples include merge sort, quicksort, and binary search.

Worst-Case Linear Time Selection Algorithm

The worst-case linear time selection algorithm, also known as the **Median of Medians algorithm** or **Blum-Floyd-Pratt-Rivest-Tarjan (BFPRT) algorithm**, is used to find the k-th smallest element in an unsorted array with a guaranteed linear time complexity. The steps are as follows:

1. **Divide the Array:** Split the array into groups of five elements each.
2. **Sort and Find Medians:** Sort each group and find the median of each group.

3. **Median of Medians:** Use the Median of Medians as the pivot to partition the array.
4. **Partition the Array:** Partition the array around the pivot.
5. **Recur:** Depending on the position of the pivot, recursively find the k-th smallest element in the relevant partition.

Analysis of Complexity

1. **Divide:** Dividing the array into groups of five takes $O(n)$ time.
2. **Sort and Find Medians:** Sorting each group of five elements and finding medians also takes $O(n)$ time because there are $O(n/5)$ groups, and sorting each group takes constant time.
3. **Median of Medians:** Finding the median of the medians involves a recursive call on a smaller array of size $n/5$, which contributes $O(n)$ time.
4. **Partition:** Partitioning the array around the pivot takes $O(n)$ time.

Thus, the recurrence relation for the worst-case time complexity is:

$$T(n) \leq T(5n) + T(107n) + O(n)$$

This recurrence solves to $O(n)$, indicating that the Median of Medians algorithm runs in linear time.

Conclusion

The divide-and-conquer strategy simplifies complex problems by dividing them into smaller, manageable sub-problems, solving them recursively, and combining their solutions. The worst-case linear time selection algorithm, through clever partitioning and selection of pivots, ensures a linear time complexity for finding the k-th smallest element, making it highly efficient for large datasets.

Divide and Conquer Strategy

Divide and Conquer is a problem-solving technique that involves breaking down a complex problem into smaller, more manageable subproblems. These subproblems are then solved independently, and their solutions are combined to solve the original problem.

[1. Divide-and-conquer algorithm - Wikipedia](#)



en.wikipedia.org

[2. brainly.in](https://brainly.in)

brainly.in

[3. Divide and Conquer Algorithm - Working, Advantages & Disadvantages - ScholarHat](#)



www.scholarhat.com

[4. www.numerade.com](https://www.numerade.com)

www.numerade.com

The general approach consists of three steps:

1. **Divide:** Break the problem into smaller subproblems of the same type.
2. **Conquer:** Solve the subproblems recursively. If the subproblem is small enough, solve it directly.
3. **Combine:** Combine the solutions of the subproblems to obtain the solution to the original problem.

Example: Merge Sort is a classic example of the divide and conquer approach. It divides an unsorted array into two halves, recursively sorts the halves, and then merges the sorted halves to produce the final sorted array.

[1. Divide and conquer algorithms \(article\) - Khan Academy](#)



www.khanacademy.org

[2. Merge sort algorithm overview \(article\) - Khan Academy](#)



www.khanacademy.org

Worst-Case Linear Time Selection Algorithm

The problem of finding the k-th smallest element in an unsorted array can be solved efficiently using a divide and conquer approach called the **median-of-medians algorithm**. While its average-case time complexity is linear, it guarantees a worst-case linear time complexity.

[1. Median of medians - Wikipedia](#)



en.wikipedia.org

Algorithm:

1. **Divide the array into groups of 5 elements each.**
2. **Find the median of each group.**
3. **Recursively find the median of the medians.** This becomes the pivot.
4. **Partition the array around the pivot.**
5. **Recursively apply the algorithm to the appropriate subarray based on the position of k relative to the pivot.**

Complexity Analysis:

- **Time complexity:**
 - Dividing the array into groups of 5 takes $O(n)$ time.
 - Finding the median of each group takes constant time, so finding all medians takes $O(n)$ time.
 - Recursively finding the median of medians takes $T(n/5)$ time.
 - Partitioning the array around the pivot takes $O(n)$ time.
 - Recursively solving the subproblem takes $T(n/c)$ time, where c is a constant greater than 1.

The recurrence relation for the time complexity is:

$$T(n) = T(n/5) + T(7n/10) + O(n)$$

- Using techniques like the Master Theorem, it can be shown that $T(n)$ is $O(n)$, ensuring a worst-case linear time complexity.
- **Space complexity:** $O(1)$ extra space is used, making it an in-place algorithm.

Key points:

- The median-of-medians technique is crucial for guaranteeing the worst-case linear time complexity.
- The algorithm effectively reduces the problem size in each recursive call, leading to the linear time-bound.

By carefully analyzing the algorithm and its recurrence relation, we can establish that the worst-case time complexity for finding the k-th smallest element using the median-of-medians algorithm is linear.

Worst-Case Linear Time Selection Algorithm

The **Median of Medians** algorithm is an example of a worst-case linear time selection algorithm. It is used to find the k-th smallest element in an unsorted array with a guaranteed $O(n)$ time complexity.

Steps of the Algorithm:

1. **Divide the Array:**
 - Split the array into groups of at most 5 elements.
2. **Find the Median of Each Group:**
 - Sort each group and determine the median of each group.
 - This can be done in constant time since the groups are of fixed size (at most 5).
3. **Median of Medians:**
 - Collect all the medians and recursively find the median of these medians. This median will be the pivot.
4. **Partition the Array:**
 - Use the median of medians as a pivot to partition the array into two sub-arrays:
 - Elements less than or equal to the pivot.
 - Elements greater than the pivot.
5. **Recur:**
 - Determine the position of the pivot.
 - If the position is the k-th smallest, return the pivot.
 - Otherwise, recursively apply the selection algorithm to the appropriate sub-array.

Example:

Let's illustrate with an example where we find the 5th smallest element in the array:

Example:

Let's illustrate with an example where we find the 5th smallest element in the array:

$\{9, 3, 2, 7, 6, 5, 4, 8, 1, 0\}$

1. **Divide into groups of 5:**

$\{9, 3, 2, 7, 6\}, \{5, 4, 8, 1, 0\}$

2. **Find medians of each group:**

Medians: $\{6, 4\}$

3. **Median of medians:** (Finding median of $\{6, 4\}$)

- Since there are only two medians, 6 is chosen as the median of medians.

4. **Partition around 6:**

$\{3, 2, 5, 4, 1, 0\}$ (left), $\{7, 9, 8\}$ (right)

5. **Recur:**

- Since the 5th smallest element is in the left partition, recursively apply the algorithm to $\{3, 2, 5, 4, 1, 0\}$ to find the 5th smallest element.

Complexity Analysis:

1. **Divide the Array:**

- Dividing into groups of 5: $O(n)$

2. **Find Medians of Each Group:**

- Sorting each group of 5: $O(n)$
- Finding medians: $O(n/5)$

- Finding medians: $O(n/5)$

3. Median of Medians:

- Recursively finding the median of the medians: $T(n/5)$

4. Partition the Array:

- Partitioning around the pivot: $O(n)$

The recurrence relation for the time complexity is:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

The solution to this recurrence is $T(n) = O(n)$, ensuring linear time complexity.

Alternative Example: Quickselect Algorithm

Quickselect is another selection algorithm that is similar to quicksort but optimized for selecting the k -th smallest element. While its average time complexity is $O(n)$, its worst-case time complexity can be $O(n^2)$. However, with good pivot choices (like using Median of Medians), Quickselect can achieve linear time in practice.

Steps of Quickselect:

1. Choose a Pivot:

- Randomly or using Median of Medians.

2. Partition the Array:

- Partition the array into elements less than the pivot and elements greater than the pivot.

3. Recur:

- If the pivot is the k -th smallest element, return it.
- Otherwise, recursively apply the algorithm to the appropriate partition.

Summary

The Median of Medians algorithm guarantees worst-case linear time complexity by carefully choosing the pivot to balance the partitions. Quickselect, on the other hand, has an average-case linear time complexity but can be enhanced to achieve better performance using the Median of Medians strategy. Both algorithms exemplify efficient methods for selection in unsorted arrays.

2. Write the dynamic programming algorithm for matrix chain multiplication. Find the optimal parenthesization for the matrix chain product ABCD with the size of each is given as $A_{5 \times 10}$, $B_{10 \times 15}$, $C_{15 \times 20}$, $D_{20 \times 30}$. (10)

Matrix chain multiplication is a classic optimization problem where you need to find the most efficient way to multiply a chain of matrices. The goal is to determine the optimal order of multiplication that minimizes the total number of scalar multiplications required.

Problem Definition

Given a sequence of matrices A_1, A_2, \dots, A_n , where matrix A_i has dimensions $p_{i-1} \times p_i$, find the most efficient way to multiply these matrices. The task is to determine the order of matrix multiplication that minimizes the total number of scalar multiplications.

Dynamic Programming Approach

1. Define the Problem with States:

- Let $m[i, j]$ represent the minimum number of scalar multiplications needed to compute the product $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$.
- Let $s[i, j]$ be the index at which the optimal split occurs for multiplying $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$.

2. Initialization:

- For single matrices, $m[i, i] = 0$ because no multiplication is needed.

3. Recursive Relation:

- For $i < j$, the cost of multiplying $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ is:

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1} \cdot p_k \cdot p_j)$$

- Here, k is the index at which the matrix chain is split.

4. Algorithm Steps:

Explanation:

- Initialization:** The matrix `m` is initialized to store the minimum number of scalar multiplications, and `s` is initialized to store the split points.
- Filling the Tables:** The nested loops iterate over all possible subproblems, computing the minimum multiplication cost for each subchain of matrices.
- Optimal Parenthesization:** The function `print_optimal_parens` recursively prints the optimal parenthesization of the matrix chain product.

This algorithm runs in $O(n^3)$ time and uses $O(n^2)$ space, making it efficient for solving the matrix chain multiplication problem.

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. We are given a sequence (chain) $\langle A_1; A_2; \dots; A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 \dots A_n : (15.5)$$

We can evaluate the expression (15.5) using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of matrices is **fully parenthesized** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1; A_2; A_3; A_4 \rangle$, then we can fully parenthesize the product $A_1 A_2 A_3 A_4$ in five distinct ways:

$$\langle A_1 . A_2 . A_3 A_4 \rangle ; \langle A_1 . . A_2 A_3 / A_4 \rangle ; \langle . . A_1 A_2 / . A_3 A_4 \rangle ; \langle . . A_1 . A_2 A_3 // A_4 \rangle ; \dots \langle A_1 A_2 / A_3 / A_4 \rangle :$$

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. The standard algorithm is given by the following pseudocode, which generalizes the SQUARE-MATRIX-MULTIPLY procedure from Section 4.2. The attributes *rows* and *columns* are the numbers of rows and columns in a matrix.

MATRIX-MULTIPLY(A;B/

1 2 3 4 5 6 7 8 9

if $A:\text{columns} \neq B:\text{rows}$
error "incompatible dimensions"

else let C be a new $A:\text{rows}$ $B:\text{columns}$ matrix **for** $i \in 1$ **to** $A:\text{rows}$

for $j \in 1$ **to** $B:\text{columns}$ $c_{ij} \leftarrow 0$

for $k \in 1$ **to** $A:\text{columns}$ $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$

return C

We can multiply two matrices A and B only if they are **compatible**: the number of columns of A must equal the number of rows of B . If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr . In what follows, we shall express costs in terms of the number of scalar multiplications.

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1; A_2; A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100 , 100×5 , and 5×50 , respectively. If we multiply according to the parenthesization $((A_1 A_2) A_3)$, we perform $10 \times 100 \times 5 = 5000$ scalar multiplications to compute the 10×5 matrix product $A_1 A_2$, plus another $10 \times 5 \times 50 = 2500$ scalar multiplications to multiply this matrix by A_3 , for a total of 7500 scalar multiplications. If instead we multiply according to the parenthesization $(A_1 (A_2 A_3))$, we perform $100 \times 5 \times 50 = 25,000$ scalar multiplications to compute the 100×50 matrix product $A_2 A_3$, plus another $10 \times 100 \times 50 = 50,000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

We state the **matrix-chain multiplication problem** as follows: given a chain $\langle A_1; A_2; \dots; A_n \rangle$ of n matrices, where for $i = 1, 2, \dots, n$, matrix A_i has dimension

$p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \dots A_n$ in a way that minimizes the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75,000).

▶ Matrix Chain Multiplication - Dynamic Programming

3. What do you mean by Backtracking? Explain the backtracking algorithm for solving 0-1 knapsack problem and find the solution for the problem given below:
(10)

Backtracking algorithms are often used to solve constraint satisfaction problems. The 0—1 Knapsack Problem is an example problem that can be solved by backtracking. (There are other approaches (greedy, dynamic programming) for this problem.)

Backtracking is a general algorithmic technique for solving problems incrementally, one piece at a time, and removing those solutions that fail to satisfy the constraints of the problem at any point of time (often referred to as "pruning"). It involves exploring all possible options to solve a problem and undoing ("backtracking") those decisions that lead to a dead end or invalid solution, thus systematically searching for a solution.

▶ Knapsack problem using Backtracking

Group B

Answer any EIGHT

1. Explain the iterative algorithm used to find the GCD of two given numbers and analyze their complexity. (5)

```
def gcd(a, b):  
  
    while b != 0:  
  
        a, b = b, a % b  
  
    return a
```

Iterative Algorithm to Find the GCD

The **Greatest Common Divisor (GCD)** of two integers a and b is the largest positive integer that divides both a and b without leaving a remainder. One efficient way to compute the GCD is using the **Euclidean Algorithm**, which can be implemented iteratively.

Steps of the Iterative Euclidean Algorithm:

1. **Initialize Variables:**

- Let a and b be the two integers, with $a \geq b$.

2. **Iterative Process:**


- While $b \neq 0$:
 - Compute the remainder r of a divided by b , i.e., $r = a \% b$.
 - Update a to b and b to r .

3. **Termination:**

- When b becomes 0, a contains the GCD of the original numbers.

Pseudocode

python

 Copy code

```
def gcd_iterative(a, b):  
    while b != 0:  
        a, b = b, a % b  
    return a
```

example

Find the GCD of 56 and 98:

1. $a = 98, b = 56$
 - Compute $98 \% 56 = 42$
 - Update $a = 56, b = 42$
2. $a = 56, b = 42$
 - Compute $56 \% 42 = 14$
 - Update $a = 42, b = 14$
3. $a = 42, b = 14$
 - Compute $42 \% 14 = 0$
 - Update $a = 14, b = 0$

When $b = 0$, $a = 14$, so the GCD of 56 and 98 is 14.

Complexity Analysis


The Euclidean algorithm has a time complexity of $O(\log(\min(a, b)))$. This can be analyzed by observing that the size of the numbers reduces approximately by half with each iteration.

1. Number of Steps:

- Each step involves a division operation and the size of the number b (the smaller of the two numbers) decreases significantly.
- Specifically, if the numbers are large, the number of iterations required to reduce b to zero is proportional to the number of digits in b , which is logarithmic in its value.

2. Mathematical Insight:

- The algorithm's efficiency is based on the fact that the remainder operation ($a \% b$) significantly reduces the size of the numbers involved.
- The number of iterations can be shown to be $O(\log(\min(a, b)))$ through the properties of the Fibonacci sequence, where the Euclidean algorithm has its worst-case scenario.

In summary, the iterative Euclidean algorithm  a highly efficient method to compute the GCD of two numbers with a time complexity of $O(\log(\min(a, b)))$.

2. Generate the prefix code for the string "CYBER CRIME" using the Huffman algorithm and find the total number of bits required. (5)

Huffman Coding

Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.

Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

How Huffman Coding works?

Suppose the string below is to be sent over a network.

B	C	A	A	D	D	D	C	C	A	C	A	C	A	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Initial string

Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of $8 * 15 = 120$ bits are required to send this string.

Using the Huffman Coding technique, we can compress the string to a smaller size.

Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.

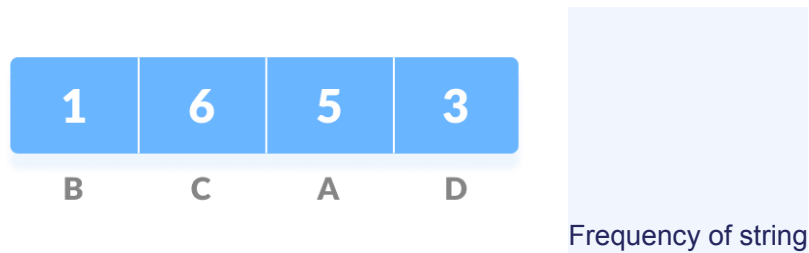
Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix code ie. a code associated with a character should not be

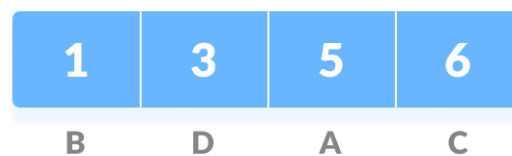
present in the prefix of any other code. The tree created above helps in maintaining the property.

Huffman coding is done with the help of the following steps.

1. Calculate the frequency of each character in the string.



2. Sort the characters in increasing order of the frequency. These are



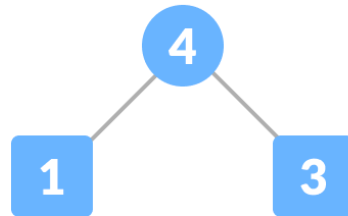
stored in a priority queue Q .

Characters sorted according to the frequency

3. Make each unique character as a leaf node.
4. Create an empty node z . Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z . Set the value of the z as the sum of the above two minimum



* A C



B D

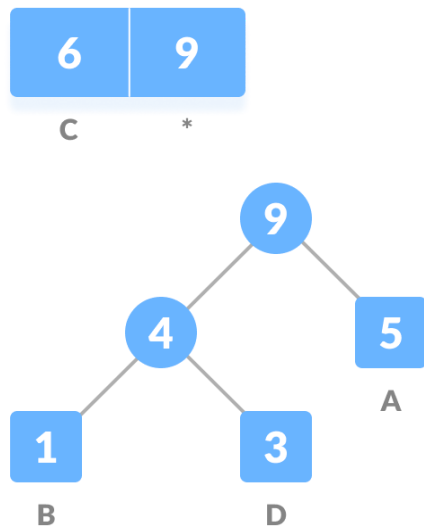
frequencies.

numbers

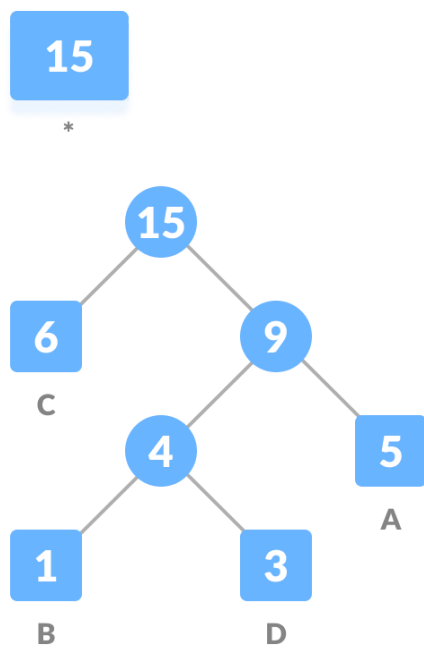
Getting the sum of the least

5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure above).
6. Insert node z into the tree.

7. Repeat steps 3 to 5 for all the characters.

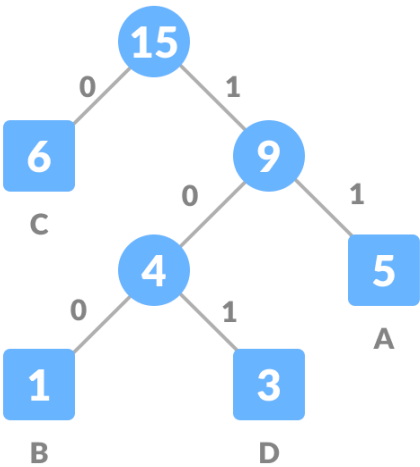


Repeat steps 3 to 5 for all the characters.



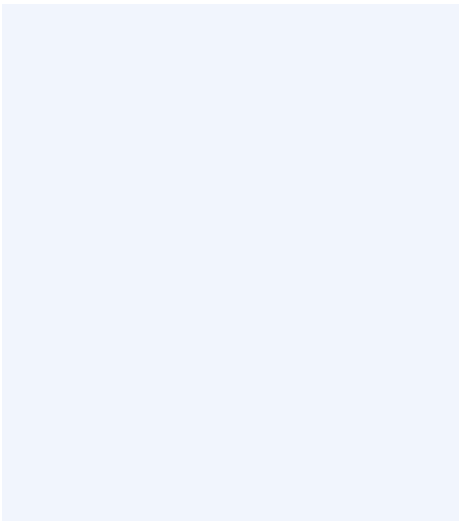
Repeat steps 3 to 5 for all the characters.

8. For each non-leaf node, assign 0 to the left edge and 1 to the right



edge.

right edge



Assign 0 to the left edge and 1 to the

For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

Character	Frequency	Code	Size
A	5	11	$5 \times 2 = 10$
B	1	100	$1 \times 3 = 3$
C	6	0	$6 \times 1 = 6$
D	3	101	$3 \times 3 = 9$

4 * 8 = 32 bits

15 bits

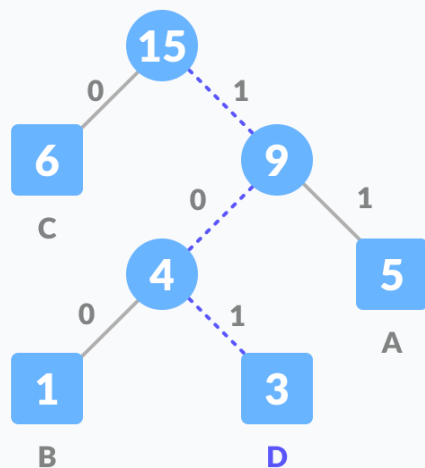
28 bits

Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to $32 + 15 + 28 = 75$.

Decoding the code

For decoding the code, we can take the code and traverse through the tree to find the character.

Let 101 is to be decoded, we can traverse from the root as in the figure below.



Decoding

Huffman Coding Algorithm

create a priority queue Q consisting of each unique character.

sort then in ascending order of their frequencies.

for all the unique characters:

```
create a newNode

extract minimum value from Q and assign it to leftChild of newNode

extract minimum value from Q and assign it to rightChild of newNode

calculate the sum of these two minimum values and assign it to the value of
newNode

insert this newNode into the tree

return rootNode
```

3. Define tractable and intractable problems. Illustrate the vertex cover problem with an example. (5)

Tractable and Intractable Problems

Tractable Problems

A problem is considered **tractable** if it can be solved by an algorithm whose running time is bounded by a polynomial function of the input size. In simpler terms, the time it takes to solve the problem grows reasonably slowly as the problem size increases. Examples of tractable problems include sorting, searching, and finding the shortest path between two points in a graph.


Intractable Problems

An **intractable** problem is one for which no known polynomial-time algorithm exists. These problems typically require exponential time to solve, making them impractical for large input sizes. Many optimization and decision problems fall into this category.

Vertex Cover Problem

The **vertex cover problem** is a classic example of an intractable problem. Given an undirected graph, a vertex cover is a subset of vertices such that

every edge of the graph has at least one endpoint in the subset. The goal is to find the minimum-sized vertex cover.

 [Vertex Cover Problem- Approximation Algorithm || Solved with Example ...](#)

Tractable and Intractable Problems

Tractable Problems:

- Tractable problems are those for which there exists an algorithm that can solve the problem in polynomial time (i.e., the time complexity of the algorithm is $O(n^k)$ for some constant k).
- These problems are considered efficiently solvable and feasible for large inputs.
- Examples include sorting algorithms like QuickSort, MergeSort, and searching algorithms like Binary Search.

Intractable Problems:

- Intractable problems are those for which no known polynomial-time algorithm exists. Solving these problems requires super-polynomial time (e.g., exponential time, factorial time).
- These problems are considered not efficiently solvable, especially as the input size grows.
- Examples include many NP-complete problems, such as the Traveling Salesman Problem (TSP), the Knapsack Problem, and the Vertex Cover Problem.

Vertex Cover Problem

The Vertex Cover problem is a classic example of an NP-complete problem, which means it is intractable and no polynomial-time algorithm is known to solve it for all cases.

Definition:

Given an undirected graph $G=(V,E)$, a vertex cover is a subset $C \subseteq V$ such that every edge in E is incident to at least one vertex in C . The goal is to find the smallest possible vertex cover for G .

4. Find the edit distance between the string "ARTIFICIAL" and "NATURAL" using dynamic programming. (5)

5. Write short notes on:
- a) Best, Worst, and average case complexity
 - b) Greedy Strategy (5)

. In particular, problems to which greedy algorithms apply have optimal substructure. One major difference between greedy algorithms and dynamic programming is that instead of first finding optimal solutions to sub- problems and then making an informed choice, greedy algorithms first make a “greedy” choice—the choice that looks best at the time—and then solve a resulting subproblem, without bothering to solve all possible related smaller subproblems. Surprisingly, in some cases this strategy works!

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. This chapter explores optimization problems for which greedy algorithms provide optimal solutions. Before reading this chapter, you should read about dynamic programming in Chapter 15, particularly Section 15.3.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We shall first examine, in Section 16.1, a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution. We shall arrive at the greedy algorithm by first considering a dynamic-programming approach and then showing that we can always make greedy choices to arrive at an optimal solution. Section 16.2 reviews the basic elements of the greedy approach, giving a direct approach for proving greedy algorithms correct. Section 16.3 presents an important application of greedy techniques: designing data-compression (Huffman) codes. In Section 16.4, we investigate some of the theory underlying combinatorial structures called “matroids,” for which a greedy algorithm always produces an optimal solution. Finally, Section 16.5 applies matroids to solve a problem of scheduling unit-time tasks with deadlines and penalties.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that we can view as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 23), Dijkstra's algorithm for shortest paths from a single source (Chapter 24), and Chvátal's greedy set-covering heuristic (Chapter 35). Minimum-spanning-tree algorithms furnish a classic example of the greedy method. Although you can read

Overview of Greedy Algorithms

Greedy algorithms are a powerful technique in algorithm design, used to solve optimization problems. They work by making the most advantageous choice at each step with the hope that these local choices will lead to a globally optimal solution. Here's a brief overview based on your description:

Key Concepts

1. Local Optimal Choice:

- Greedy algorithms make decisions based on the immediate benefit, hoping these choices will lead to the best overall solution.
- This approach simplifies the problem-solving process but doesn't always guarantee an optimal solution for every problem.

2. Comparison with Dynamic Programming:

- **Dynamic Programming (DP):** Solves problems by breaking them down into simpler subproblems and solving each subproblem only once, storing the results to avoid redundant computations. It's often used for problems where choices are dependent on previous decisions.
- **Greedy Algorithms:** Focus on making the best local choice at each step without considering the entire problem space. It's generally simpler and more efficient but may not always provide an optimal solution.

Greedy Algorithm Applications

1. Activity-Selection Problem:

- **Problem:** Select the maximum number of non-overlapping activities from a given list.
- **Greedy Approach:** Always select the next activity that finishes the earliest. This approach works optimally for this problem, and the algorithm can be implemented efficiently.

2. Huffman Coding:

- **Problem:** Compress data by assigning variable-length codes to characters based on their frequencies.
- **Greedy Approach:** Build a Huffman Tree by repeatedly merging the two least frequent nodes. This method ensures that characters with higher frequencies have shorter codes, leading to an optimal encoding.

3. Minimum Spanning Tree (MST):

- **Algorithm:** Prim's and Kruskal's algorithms.
- **Greedy Approach:**

- **Kruskal's Algorithm:** Add edges in increasing order of weight, ensuring no cycles are formed.
 - **Prim's Algorithm:** Grow the MST by adding the smallest edge that connects a new vertex to the growing MST.
4. **Shortest Path (Dijkstra's Algorithm):**
- **Problem:** Find the shortest path from a source to all other nodes in a graph with non-negative edge weights.
 - **Greedy Approach:** Continuously expand the shortest known path from the source node to other nodes.
5. **Set-Covering Problem:**
- **Chvátal's Greedy Heuristic:** Iteratively select the set that covers the largest number of uncovered elements.

Application to Combinatorial Structures

Matroids:

- **Definition:** A combinatorial structure that generalizes the notion of linear independence in vector spaces. Greedy algorithms can be applied to matroid problems to find optimal solutions.
- **Example:** Scheduling unit-time tasks with deadlines and penalties can be efficiently solved using matroid properties.

Summary

- **Greedy Algorithms:** Make the best local choice at each step with the goal of finding a globally optimal solution.
- **Dynamic Programming vs. Greedy:** DP is used for problems with overlapping subproblems and optimal substructure, while Greedy algorithms are suitable for problems where local decisions lead to global optimality.
- **Applications:** Include activity selection, Huffman coding, MST algorithms, shortest path algorithms, and set-covering problems.
- **Matroids:** Provide a theoretical foundation for applying greedy algorithms to more complex problems.

These concepts form a crucial part of algorithm design and optimization, with greedy methods offering efficient solutions for many practical problems.

6. Solve the following recurrence relations using the master's method:
 - a. $T(n) = 2T(n/4) + kn^2$, $n > 1$
 - b. $T(n) = 5T(n/4) + kn$, $n > 1$
7. Solve the following linear congruences using the Chinese Remainder Theorem:

$$X \equiv 1 \pmod{2}$$

$$X \equiv 3 \pmod{5}$$

$$X \equiv 6 \pmod{7}$$

Solving Linear Congruences Using Chinese Remainder Theorem

Understanding the Problem

We have three linear congruences:

- $X \equiv 1 \pmod{2}$
- $X \equiv 3 \pmod{5}$
- $X \equiv 6 \pmod{7}$

We need to find a value of X that satisfies all three congruences simultaneously using the Chinese Remainder Theorem (CRT).

Applying the Chinese Remainder Theorem

Step 1: Calculate M

- $M = m_1 * m_2 * m_3 = 2 * 5 * 7 = 70$

Step 2: Calculate M_i and y_i

- $M_1 = M / m_1 = 70 / 2 = 35$
- $M_2 = M / m_2 = 70 / 5 = 14$
- $M_3 = M / m_3 = 70 / 7 = 10$

Find y_i such that $M_i * y_i \equiv 1 \pmod{m_i}$

- $35 * y_1 \equiv 1 \pmod{2} \Rightarrow y_1 = 1$
- $14 * y_2 \equiv 1 \pmod{5} \Rightarrow y_2 = 2$
- $10 * y_3 \equiv 1 \pmod{7} \Rightarrow y_3 = 3$

Step 3: Calculate X

- $X = (a_1 * M_1 * y_1 + a_2 * M_2 * y_2 + a_3 * M_3 * y_3) \pmod{M}$
- $X = (1 * 35 * 1 + 3 * 14 * 2 + 6 * 10 * 3) \pmod{70}$
- $X = (35 + 84 + 180) \pmod{70}$
- $X = 299 \pmod{70}$
- $X = 59$

Solution

Therefore, the solution to the system of linear congruences is $X \equiv 59 \pmod{70}$. This means any integer of the form $59 + 70k$, where k is an integer, will satisfy the given congruences.

The smallest positive solution is 59.

8. Find the MST from the following graph using Kruskal's algorithm. (5)

▶ 3.5 Prims and Kruskals Algorithms - Greedy Method

▶ 6.6 Kruskals Algorithm for Minimum Spanning Tree- Greedy method | D...

▶ Kruskal's Algorithm

▶ Prim's Algorithm

9. Trace the quick sort algorithm for sorting the array $A[] = \{15, 7, 6, 23, 18, 34, 25\}$ and write its best and worst complexity. (5)

Quicksort Algorithm

Quicksort is a sorting algorithm based on the divide and conquer approach where

1. An array is divided into subarrays by selecting a pivot element (element selected from the array).

While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quicksort Algorithm

1. Select the Pivot Element

There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.



Select a pivot

element

2. Rearrange the Array

Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

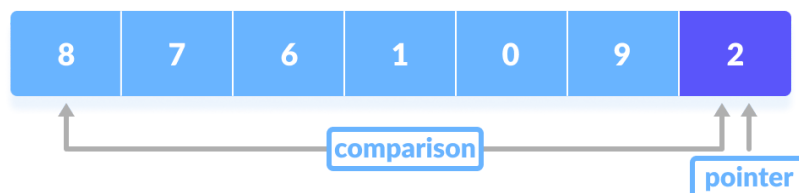


Put all the smaller

elements on the left and greater on the right of pivot element

Here's how we rearrange the array:

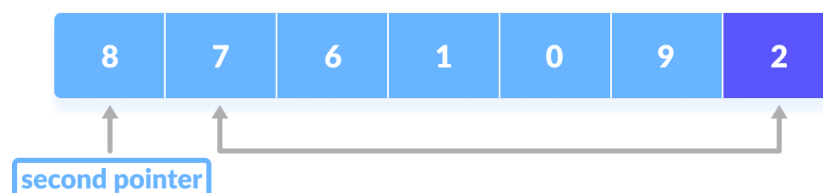
1. A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.



Comparison

of pivot element with element beginning from the first index

2. If the element is greater than the pivot element, a second pointer is set for that

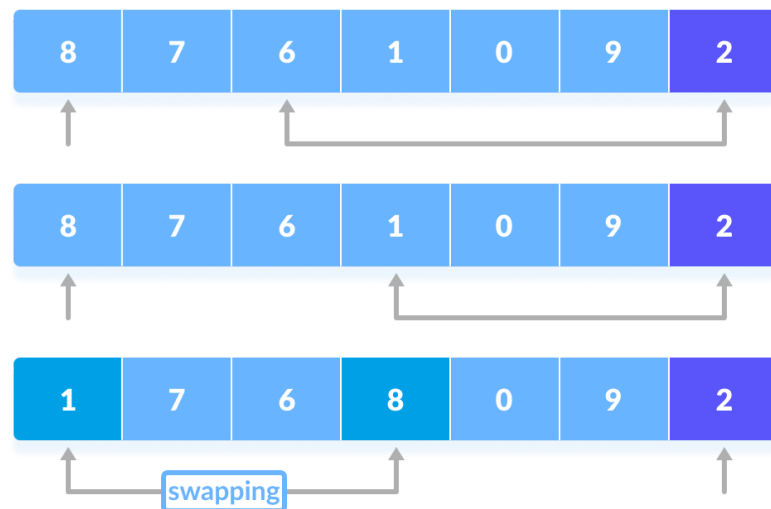


element.

If the

element is greater than the pivot element, a second pointer is set for that element.

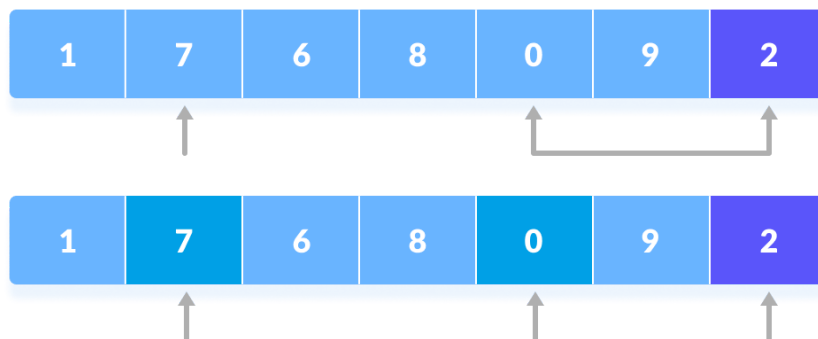
3. Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found



earlier.

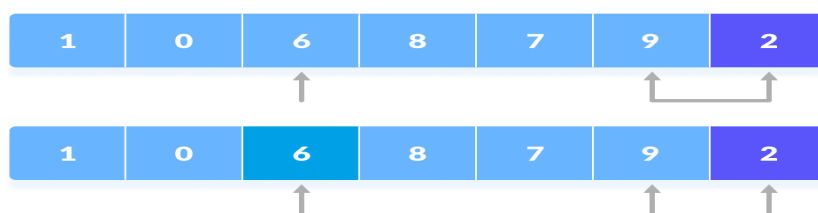
compared with other elements.

4. Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.



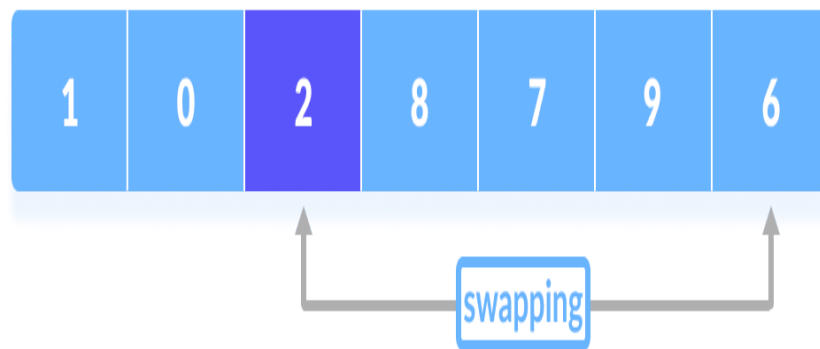
repeated to set the next greater element as the second pointer.

5. The process goes on until the second last element is reached.



goes on until the second last element is reached.

6. Finally, the pivot element is swapped with the second pointer.



Finally, the

pivot element is swapped with the second pointer.

3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately. And, step 2 is repeated.

```
quicksort(arr, pi, high)
```



Select pivot element

of in each half and put at correct place using recursion

The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

Quick Sort Algorithm

Visual Illustration of Quicksort Algorithm

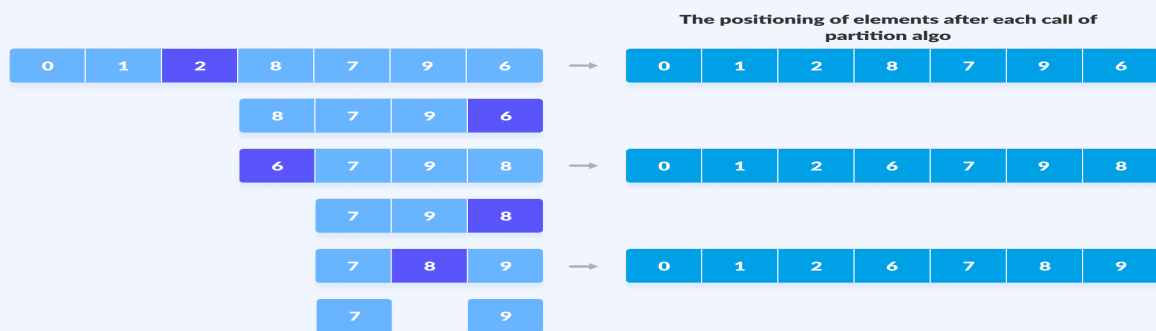
You can understand the working of quicksort algorithm with the help of the illustrations below.

`quicksort(arr, low, pi-1)`



Sorting the elements on the left of pivot using recursion

`quicksort(arr, pi+1, high)`



Sorting the elements on the right of pivot using recursion

The following procedure implements **quicksort**:

QUICKSORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      QUICKSORT( $A, q + 1, r$ )
```

To **sort** an entire array A , the initial call is **QUICKSORT**($A, 1, A.length$).

Partitioning the array

The key to the algorithm is the **PARTITION** procedure, which rearranges the subarray $A[p \dots r]$ in place.

PARTITION(A, p, r)

```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```