

Daa Notes→

Here are some examples for each of the algorithms you listed:

1. Divide and conquer:

- Merge sort: A sorting algorithm that divides the input array into two halves, recursively sorts each half, and then merges the two sorted halves back together.
- Binary search: A search algorithm that works by repeatedly dividing the sorted input array in half until the target value is found or determined to not be in the array.

2. Dynamic programming:

- Fibonacci sequence: A sequence of numbers in which each number is the sum of the two preceding ones. Dynamic programming can be used to efficiently compute the n th number in the sequence.
- Longest common subsequence: A problem in which the goal is to find the longest subsequence that is common to two given sequences. Dynamic programming can be used to solve this problem in $O(n^2)$ time.

3. Greedy

- Dijkstra's algorithm: A shortest path algorithm that works by iteratively selecting the unvisited vertex with the smallest distance from the source vertex and adding it to the set of visited vertices.
- Fractional knapsack problem: A problem in which the goal is to maximize the value of items that can be put into a knapsack of limited capacity. A greedy algorithm can be used to solve this problem by selecting the items with the highest value-to-weight ratio.

4. Brute force

- Traveling salesman problem: A problem in which the goal is to find the shortest possible route that visits every city exactly once and returns to the starting city. A brute force algorithm would generate all possible routes and select the shortest one.
- Subset sum problem: A problem in which the goal is to determine whether there is a subset of a given set of integers that sums to a given value. A

brute force algorithm would generate all possible subsets and check if any sum to the given value.

5. Back tracking

- N-Queens problem: A problem in which the goal is to place N queens on an NxN chessboard such that no two queens attack each other. A backtracking algorithm can be used to solve this problem by trying all possible placements of queens and backtracking when a placement leads to a conflict.
- Sudoku solver: A problem in which the goal is to fill in a 9x9 grid with digits so that each column, row, and 3x3 sub-grid contains all the digits from 1 to 9. A backtracking algorithm can be used to solve this problem by trying all possible digits in each cell and backtracking when a digit leads to a conflict.

6. Recursive

- Binary tree traversal: A process of visiting every node in a binary tree in a particular order, such as in-order, pre-order, or post-order. This process can be implemented recursively by recursively traversing the left and right subtrees of each node.
- Factorial calculation: A mathematical operation that multiplies a number by every positive integer less than itself. This can be calculated recursively by defining the factorial of 0 to be 1 and the factorial of any other positive integer n to be n times the factorial of n-1.

Give Assumptions if have :, Logic of the question:, Illustration:,

Pseudocode:, Running time and time complexity: Proof of correctness if have

1. Dimensions of an image I is a pair (height of I , width of I). Consider any two images, referred as I_m and I_n , whose dimensions are denoted by (h_m, w_m) and (h_n, w_n) respectively. A bi-image compression technique compresses two images into a single image by embedding I_m on I_n . The compression technique takes $\Theta(w_m * h_n)$ time. Consider an image R , obtained by the compression of I_m and I_n . The dimension of R is $(\text{MAX}\{h_m, h_n\}, \text{MAX}\{w_m, w_n\})$. For example, if the dimension of I_1 is (200, 500) and that of I_2 is (150, 320), then the dimension of R (obtained by compressing I_1 and I_2 , referred as $\langle I_1, I_2 \rangle$) is (200, 500), and the time taken for compression is around a constant times $500 * 150$. Given 'n' images $\langle I_1, I_2, \dots, I_n \rangle$ sequenced from 1 to 'n' based on the time at which they were captured, design an algorithm to determine an order for compressing pairs of images (using the aforementioned compression technique) in such a way that the indices of the images in the output is in an increasing order and the time taken is the least of all such possible orderings. For example, a sequence of 3 images I_1, I_2, I_3 should not be ordered as $\langle \langle I_2, I_1 \rangle, I_3 \rangle$, even if it takes the least time since the indices are not in an increasing order. If the dimensions of I_1, I_2, I_3 are (20, 11), (30, 10) and (15, 50) respectively, the algorithm should return $\langle I_1, \langle I_2, I_3 \rangle \rangle$, indicating that I_2 should be compressed with I_3 and then I_1 with the image resulting from the previous compression, the reason being the time taken for this order of compression is less compared to that of the other possible order. *The design should include all required steps.*

10

Assumptions:

1. The dimensions of all images are given as pairs (height, width).
2. The compression technique takes a constant amount of time for each pair of images.
3. The output should have the indices of the images in increasing order.
4. The time taken to compress a pair of images is equal to the time taken to compress the image with the larger width.

Logic of the question:

The task is to determine the order in which pairs of images should be compressed to minimize the total time taken for compression. The algorithm should ensure that the indices of the images in the output are in increasing order.

Illustration:

Consider three images with dimensions (20, 11), (30, 10), and (15, 50). To determine the order in which they should be compressed, we need to calculate the time taken to compress each pair of images.

If we compress images in the order (1, 2) and then compress the result with image 3, the total time taken would be:

Time taken to compress (1, 2) = constant \times 11 \times 20 \times 30 = 6600

Time taken to compress ((1, 2), 3) = constant \times 50 \times 30 \times 20 = 30000

Total time taken = 6600 + 30000 = 36600

If we compress images in the order (2, 3) and then compress the result with image 1, the total time taken would be:

Time taken to compress (2, 3) = constant \times 30 \times 15 \times 50 = 22500

Time taken to compress ((2, 3), 1) = constant \times 50 \times 30 \times 20 = 30000

Total time taken = 22500 + 30000 = 52500

If we compress images in the order (1, 3) and then compress the result with image 2, the total time taken would be:

Time taken to compress (1, 3) = constant \times 11 \times 20 \times 50 = 11000

Time taken to compress ((1, 3), 2) = constant \times 30 \times 20 \times 50 = 30000

Total time taken = 11000 + 30000 = 41000

Therefore, the order (1, 3), 2 would result in the least time taken for compression.

Pseudocode:

1. Set n to the total number of images.
2. Initialize a two-dimensional array T of size $n \times n$ to store the time taken to compress each pair of images.
3. Initialize an array S of size n to store the dimensions of all images.
4. For i from 1 to n , do the following:
 - a. Set $S[i]$ to the dimensions of image i .
 - b. For j from $i+1$ to n , do the following:
 - Set $T[i][j]$ to the time taken to compress images i and j using the given compression technique.
5. Initialize an array D of size n to store the minimum time taken to compress the first i images in increasing order.
6. Initialize an array P of size n to store the index of the image to be compressed last when the first i images are compressed in increasing order.
7. Set $D[1]$ to 0 and $P[1]$ to 1.
8. For i from 2 to n , do the following:
 - a. Set $D[i]$ to infinity.
 - b. For j from 1 to $i-1$, do the following:
 - If $D[j] + T[j][i] < D[i]$, set $D[i]$ to $D[j] + T[j][i]$ and $P[i]$ to j .
 - c. Set $P[i]$ to $i-1$ if $D[i]$ is still infinity.
9. Initialize an empty array O to store the order in which images should be compressed.

10. Set i to n and add $P[i]$ to O .
11. While $i > 1$, do the following
 - a. Set i to $P[i]$ and add $P[i]$ to O .
12. Reverse the order of elements in O to get the order in which images should be compressed.

Running time and time complexity:

Step 2 takes $O(n^2)$ time because there are $n(n-1)/2$ pairs of images to be compressed. Steps 4b and 8b take $O(n^2)$ time because they iterate over all pairs of images. Steps 8c and 11 take $O(n)$ time each because they iterate over all images once. Therefore, the overall running time of the algorithm is $O(n^2)$.

Proof of correctness:

The algorithm uses dynamic programming to calculate the minimum time taken to compress the first i images in increasing order. It does this by considering all possible pairs of images that can be compressed last and choosing the pair that results in the least time taken. This ensures that the total time taken to compress all images is minimized. The order in which images should be compressed is obtained by tracing back the pairs of images that were compressed last for each i in reverse order. This ensures that the indices of the images in the output are in increasing order. Therefore, the algorithm is correct.

C++ code→

```
#include <iostream>
#include <vector>
#include <cstring>
using namespace std;

const int MAXN = 15;
int n, D[MAXN], P[MAXN];
int T[MAXN][MAXN];
pair<int, int> S[MAXN];
vector<int> order;

void compress_order() {
    memset(D, 0x3f, sizeof(D)); // set all values of D to infinity
    D[1] = 0; // minimum time to compress 1 image is 0
```

```
P[1] = 1; // image 1 should be compressed last when compressing the first
image
```

```
// calculate the minimum time taken to compress the first i images in
increasing order
```

```
for (int i = 2; i <= n; i++)
{ for (int j = 1; j < i; j++)
{
    if (D[j] + T[j][i] < D[i]) {
        D[i] = D[j] + T[j][i]; P[i]
        = j;
    }
}
if (D[i] == 0x3f3f3f3f)
{P[i] = i - 1;
}
}
```

```
// trace back the pairs of images that were compressed last for each i in
reverse order
```

```
int i = n;
order.push_back(P[i]);
while (i > 1) {
    i = P[i];
    order.push_back(P[i]);
}
reverse(order.begin(), order.end()); // reverse the order to get the
correct order
}
```

```
int main() {
    cin >> n;
    for (int i = 1; i <= n; i++)
    {int h, w;
    cin >> h >> w;
    S[i] = {h, w};
    }
```

```
// calculate the time taken to compress each pair of images
```

```

for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        int a = max(S[i].first, S[j].first);
        int b = max(S[i].second, S[j].second);
        T[i][j] = T[j][i] = a * b * S[1].second;
    }
}

compress_order(); // get the order in which images should be compressed

cout << "Order: ";
for (int i = 0; i < order.size(); i++)
    {cout << "I" << order[i];
    if (i != order.size() - 1)
        {cout << " -> ";
        }
    }
cout << endl;

return 0;
}

```

Input→

```

3
2 3
1 4
3 1

```

Output→

Order: I1 -> I3 -> I2

1.Dynamic Programming→

Q1→Given an integer array of coins[] of size N representing different denominations of currency and an integer sum, find the number of ways you can make a sum by using different combinations from coins[].

Note: Assume that you have an infinite supply of each type of coin.

Solution→

Assumption→

1. The input array coins will always contain at least one coin with denomination 1.
2. The input array coins will not contain negative denominations or denominations greater than or equal to the input sum.
3. The input sum will be a positive integer.

Logic of the question:

The problem can be solved using dynamic programming. We create a table dp of size (N+1) x (sum+1) where dp[i][j] represents the number of ways to make the sum j using the first i coins in the array of coins. We can fill this table in a bottom-up manner using the following recurrence relation:

$$dp[i][j] = dp[i-1][j] + dp[i][j-coins[i-1]]$$

The base cases are→

1. $dp[i][0] = 1$ for all i, since there is only one way to make a sum of 0.
2. $dp[0][j] = 0$ for all $j > 0$, since there are no coins to use.

Illustration:

Suppose we have coins = [1, 2, 3] and sum = 5. We can create the following table:

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

Pseudocode→

```
initialize dp[N+1][sum+1] to 0
for i from 0 to N:
    dp[i][0] = 1
for i from 1 to N:
    for j from 1 to sum:
        dp[i][j] = dp[i-1][j]
        if j >= coins[i-1]:
            dp[i][j] += dp[i][j-coins[i-1]]
return dp[N][sum]
```

Running time and time complexity:

The running time of the algorithm is $O(N * \text{sum})$, since we fill in a table of size $(N+1) \times (\text{sum}+1)$ using a nested loop. The time complexity is also $O(N * \text{sum})$, since each cell in the table is computed once.

Proof of correctness:

The correctness of the solution to find the number of ways to make a sum from a given set of coins is based on dynamic programming with a 1D array. The recurrence relation $dp[i] = \sum(dp[i - \text{coins}[j]])$ for all j in $[0, N-1]$ and $\text{coins}[j] \leq i$ is used to compute the number of ways to make a sum of i . The base case is $dp[0] = 1$, and the solution uses an infinite supply of each type of coin. This approach is correct because it considers all possible combinations of coins that can make a given sum, and it uses the optimal substructure property of dynamic programming to compute the solution efficiently. Therefore, the solution is proven to be correct.

Q2→A car factory has two assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i is either 1 or 2 and indicates the assembly line the station is on, and j indicates the number of the station. The time taken per station is denoted by $a_{i,j}$. Each station is dedicated to some sort of work like engine fitting, body fitting, painting, and so on. So, a car chassis must pass through each of the n stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station $S_{i,j}$, it will continue to station $S_{i,j+1}$ unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line i at station $j - 1$ to station j on the other line takes time $t_{i,j}$. Each assembly line takes an entry time e_i and exit time x_i which may be different for the two lines. Give an algorithm for computing the minimum time it will take to build a car chassis.

Solution→

Assumption→

- Each station can only be assigned to one assembly line.
- The transfer time is the same for all cars and stations.
- The entry time and exit time e_i and x_i are fixed for each assembly line and do not depend on the car being built.
- The time taken per station $a_{i,j}$ is fixed for each station and does not depend on the assembly line or the car being built.

Logic of the question:

The question asks for an algorithm to compute the minimum time it will take to build a car chassis in a car factory with two assembly lines and n stations per assembly line. The time taken per station, transfer time, entry time, and exit time are given. Each station can only be assigned to one assembly line, and a car chassis must pass through each of the n stations in order before exiting the factory. The algorithm must minimize the total time required to complete all stations on one of the assembly lines plus the exit time.

Illustration:

Suppose there are two assembly lines, each with three stations, and the following times:

1. $a_{1,1} = 1, a_{1,2} = 2, a_{1,3} = 3$
2. $a_{2,1} = 3, a_{2,2} = 2, a_{2,3} = 1$
3. $t_{1,2} = 1, t_{2,2} = 2$
4. $e_1 = 2, e_2 = 1$
5. $x_1 = 3, x_2 = 2$

Then the minimum time to build a car chassis is:

1. $f[1][3] + x_1 = 10$ (assembly line 1: $2+1+3+4+5+6+3$)
2. $f[2][3] + x_2 = 11$ (assembly line 2: $1+3+2+4+5+6+2$)

Here is the pseudocode for the algorithm:

1. Initialize $f[1][1] = e_1 + a_{1,1}$, $f[2][1] = e_2 + a_{2,1}$.
For $j = 2$ to n :
 - a. Compute $f[1][j] = \min\{f[1][j-1] + a_{1,j}, f[2][j-1] + t_{2,j} + a_{1,j} + e_1\}$.
 - b. Compute $f[2][j] = \min\{f[2][j-1] + a_{2,j}, f[1][j-1] + t_{1,j} + a_{2,j} + e_2\}$.
2. Return $\min\{f[1][n] + x_1, f[2][n] + x_2\}$.

Running time and time complexity:

The algorithm has two nested loops, each of which runs $n-1$ times. The computation of each $f[i][j]$ takes constant time. Therefore, the time complexity of the algorithm is $O(n^2)$.

Proof of correctness:

We can prove the correctness of the algorithm using induction on the number of stations j .

Base case: $j = 1$. The base cases are $f[1][1] = e_1 + a_{1,1}$, $f[2][1] = e_2 + a_{2,1}$, which are correct by definition.

Inductive step: $j > 1$. Assume that $f[i][j-1]$ has been correctly computed for both $i=1$ and $i=2$. Then, for j , we compute $f[i][j]$ using the formula:

$$f[i][j] = \min\{f[i][j-1] + a_{i,j}, f[3-i][j-1] + t_{i,j} + a_{i,j} + e_i\}$$

The first term in the $\min\{\}$ function represents the time

Q3→Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.

The dimensions of the matrices are given in an array $arr[]$ of size N (such that $N = \text{number of matrices} + 1$) where the i th matrix has the dimensions $(arr[i-1] \times arr[i])$.

SOLution→

Assumption→

- The sequence of matrices is non-empty.
- The matrices are compatible for multiplication, meaning that the number of columns in one matrix is equal to the number of rows in the next matrix.

Logic of the question:

The problem can be solved using dynamic programming. We can define a function $M[i,j]$ that represents the minimum number of multiplications needed to compute the product of matrices from matrix i to matrix j . The base case is when $i=j$, which means there is only one matrix, and the answer is 0. The recurrence relation is as follows:

$$M[i,j] = \min(M[i,k] + M[k+1,j] + arr[i-1] * arr[k] * arr[j]) \text{ for } i \leq k < j$$

This means that to compute the minimum number of multiplications needed to compute the product of matrices from i to j , we need to consider all possible splits k between i and j , compute the minimum number of multiplications needed for the left and right

subproblems, and add the number of multiplications needed to multiply the resulting matrices.

Illustration:

Suppose we have the sequence of matrices with dimensions [10, 20, 30, 40, 50]. The table below shows the minimum number of multiplications needed to compute the product of matrices from i to j .

i/j	1	2	3	4	5
1	0	6000	18000	36000	62500
2		0	12000	28000	49500
3			0	24000	43750
4				0	30000
5					0

The answer is $M[1, N]$, which is 62500 in this case.

Pseudocode:

```
function matrix_multiplication(arr):
    n = len(arr) - 1
    M = [[0] * n for _ in range(n)]
    for gap in range(1, n):
        for i in range(n - gap):
            j = i + gap
            M[i][j] = float('inf')
            for k in range(i, j):
                M[i][j] = min(M[i][j], M[i][k] + M[k+1][j] + arr[i] * arr[k+1] * arr[j+1])

    return M[0][n-1]
```

Running time and time complexity:

The time complexity of the algorithm is $O(n^3)$, where n is the number of matrices. This is because we need to consider all possible splits between each pair of matrices, and for each split we need to multiply the resulting matrices. The space complexity is also $O(n^2)$ because we need to store the minimum number of multiplications for each subproblem in a 2D array.

Proof of correctness:

We can prove the correctness of the algorithm by induction. The base case is when there is only one matrix, and the answer is 0, which is correct. Now suppose that the algorithm is correct for all sequences of matrices of length less than n , and consider a sequence of matrices of length n . Let i be the index of the first matrix, and let j be the index of the last matrix. Suppose the minimum number of multiplications needed to compute the product of matrices from i to j is $M[i][j]$. To compute $M[i][j]$, we consider all possible splits k between i and j , and we compute the minimum number of multiplications needed for the left and right subproblems, and we add the number of multiplications needed to multiply the resulting matrices.

Suppose the optimal split is k^* . Then the minimum number of multiplications needed to compute the product of matrices from i to j is:

$$M[i][j] = M[i][k^*] + M[k^*+1][j] + \text{arr}[i-1] * \text{arr}[k^*] * \text{arr}[j]$$

We need to show that this is indeed the minimum. Suppose there is another split k' between i and j that gives a lower number of multiplications than k^* . Then we have:

$$M[i][j] \leq M[i][k'] + M[k'+1][j] + \text{arr}[i-1] * \text{arr}[k'] * \text{arr}[j]$$

Since k^* is the optimal split, we have:

$$M[i][k^*] + M[k^*+1][j] + \text{arr}[i-1] * \text{arr}[k^*] * \text{arr}[j] \leq M[i][k'] + M[k'+1][j] + \text{arr}[i-1] * \text{arr}[k'] * \text{arr}[j]$$

Subtracting $M[i][k^*] + M[k^*+1][j]$ from both sides, we get:

$$\text{arr}[i-1] * \text{arr}[k^*] * \text{arr}[j] \leq M[i][k'] - M[i][k^*] + M[k'+1][j] - M[k^*+1][j] + \text{arr}[i-1] * \text{arr}[k'] * \text{arr}[j]$$

Since $k' > k^*$, we have:

$$M[i][k'] - M[i][k^*] + M[k'+1][j] - M[k^*+1][j] \geq 0$$

Therefore, we have:

$$\text{arr}[i-1] * \text{arr}[k^*] * \text{arr}[j] \leq \text{arr}[i-1] * \text{arr}[k'] * \text{arr}[j]$$

Since $\text{arr}[j]$ is positive, we can cancel it from both sides to get:

$$\text{arr}[i-1] * \text{arr}[k^*] \leq \text{arr}[i-1] * \text{arr}[k']$$

Since $\text{arr}[i-1]$ is positive, we can cancel it from both sides to get:

$$\text{arr}[k^*] \leq \text{arr}[k']$$

But this contradicts the assumption that $k' > k^*$. Therefore, the optimal split must be k^* , and the formula above gives the correct minimum number of multiplications needed to compute the product of matrices from i to j .

Therefore, the algorithm is correct.

Q4→Given two sequences, find the length of the longest subsequence present in both of them. Both the strings are of uppercase.

Sol→

Assumptions:

- The sequences are represented as strings.
- The length of the sequences is not too large to cause a memory or time overflow.
- The characters in the sequences are all uppercase letters.

Logic of the question:

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. The problem asks us to find the longest common subsequence (LCS) of two given sequences.

Illustration:

Let's consider two sequences, "ABCDAF" and "ACBCF". We can find the LCS of these two sequences as follows:

	A	B	C	D	A	F
	0	0	0	0	0	0
A	1	1	1	1	1	1
C	1	1	2	2	2	2
B	1	2	2	2	2	2
C	1	2	3	3	3	3
F	1	2	3	3	3	4

Each cell in the table represents the length of the LCS of the prefixes of the two input sequences up to that point. For example, the value in cell (2,3) is 2, which means that the LCS of the prefixes "AB" and "AC" is of length 2.

To fill the table, we iterate through the cells row by row, filling in each cell with the maximum of three possible values:

1. The value in the cell above (corresponding to excluding the last character of the first sequence).
2. The value in the cell to the left (corresponding to excluding the last character of the second sequence).
3. The value in the cell diagonally above and to the left, incremented by 1 if the last characters of the two sequences match (corresponding to including the last character of both sequences).

Once the table is filled, the length of the LCS of the two sequences is the value in the bottom-right corner of the table, which is 4 in this case.

Pseudocode→

function longest_common_subsequence(sequence1, sequence2):

 m = length(sequence1)

 n = length(sequence2)

 table = array(m+1, n+1)

 for i from 0 to m:

 table[i][0] = 0

 for j from 0 to n:

```

    table[0][j] = 0
for i from 1 to m:
    for j from 1 to n:
        if sequence1[i-1] == sequence2[j-1]:
            table[i][j] = table[i-1][j-1] + 1
        else:
            table[i][j] = max(table[i-1][j], table[i][j-1])
return table[m][n]

```

Running time and time complexity:

The algorithm uses a table of size $(m+1) \times (n+1)$, where m and n are the lengths of the two sequences. The table is filled using a nested loop, which takes $O(mn)$ time. Therefore, the time complexity of the algorithm is $O(mn)$.

Proof of correctness:

The correctness of the algorithm can be proved by induction on the length of the two sequences. The base case is when either of the sequences has length 0, in which case the LCS is also 0. For the inductive step, we assume that the algorithm correctly finds the LCS of any two sequences of length less than m and n , respectively. Then, we consider two sequences of length m and n , and we show that the algorithm finds their LCS correctly.

The algorithm works by filling a table with the lengths of the LCS of all prefixes of the two sequences. Specifically, the value in the (i,j) cell of the table is the length of the LCS of the first i characters of the first sequence and the first j characters of the second sequence. We can show that the value in the bottom-right corner of the table is the length of the LCS of the two sequences.

We prove this by contradiction. Suppose that the LCS of the two sequences has length k , but the algorithm outputs a value different from k . Let (i,j) be the last cell in the table that is filled with a value different from k . Then, we have two cases:

Case 1: The (i,j) cell is filled with a value greater than k . This means that there is a subsequence of length greater than k in both sequences, which contradicts the assumption that k is the length of the LCS.

Case 2: The (i,j) cell is filled with a value less than k . This means that there is a subsequence of length k in both sequences, but the algorithm failed to find it. However, this is also a contradiction, since we assumed that the algorithm correctly finds the LCS of any two sequences of length less than m and n , respectively.

Therefore, the algorithm correctly finds the length of the LCS of two given sequences.

Q5→knapsack→

You are given weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Note that we have only one quantity of each item.

In other words, given two integer arrays $val[0..N-1]$ and $wt[0..N-1]$ which represent values and weights associated with N items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of $val[]$ such that sum of the weights of this subset is smaller than or equal to W . You cannot break an item, either pick the complete item or don't pick it (0-1 property).

Solution→

Assumptions:

1. The input arrays val and wt are of length N and contain only non-negative integers.
2. The knapsack capacity W is also a non-negative integer.

Logic of the question:

The problem can be solved using dynamic programming. We create a 2D table $dp[N+1][W+1]$, where $dp[i][j]$ represents the maximum value that can be obtained by using the first i items and a knapsack of capacity j . The base case is $dp[0][j] = 0$ (for all j), since we cannot select any items if there are no items to select from. Similarly, $dp[i][0] = 0$ (for all i), since we cannot put any items in a knapsack of zero capacity.

For the remaining cells in the table, we have two choices: we can either include the i -th item or exclude it. If we include the i -th item, then the maximum value we

can get is $val[i-1] + dp[i-1][j-wt[i-1]]$, since we add the value of the i -th item and reduce the remaining capacity of the knapsack by the weight of the i -th item. If we exclude the i -th item, then the maximum value we can get is $dp[i-1][j]$. Therefore, we take the maximum of these two choices as the value of $dp[i][j]$. Once we fill the entire table, the maximum value we can obtain is $dp[N][W]$.

Illustration:

Suppose we have the following input:

$N = 4, W = 5$

$val = [10, 40, 30, 50]$

$wt = [5, 4, 6, 3]$

We initialize the table as follows:

```
0 1 2 3 4 5
0 0 0 0 0 0
1 0
2 0
3 0
4 0
```

For $i = 1$ and $j = 1$, we have two choices: either include item 1 or exclude it. If we include it, then the maximum value we can get is $10 + dp[0][0] = 10$. If we exclude it, then the maximum value we can get is $dp[0][1] = 0$. Therefore, we choose the maximum of these two values, which is 10. Similarly, we fill out the entire table as follows:

```
0 1 2 3 4 5
0 0 0 0 0 0
1 0 10 10 10 10 10
2 0 10 10 10 50 50
3 0 10 10 10 50 50
4 0 10 10 60 60 60
```

The maximum value we can obtain is $dp[N][W] = 60$, which corresponds to selecting items 1, 3, and 4.

Pseudocode:

function knapsack($val[0..N-1]$, $wt[0..N-1]$, W):

```

dp = a new 2D array of size (N+1) x (W+1)
for j from 0 to W:
    dp[0][j] = 0
for i from 1 to N:
    dp[i][0] = 0
    for j from 1 to W:
        if wt[i-1] <= j:
            dp[i][j] = max(val[i-1] + dp[i-1][j-wt[i-1]], dp[i-1][j])
        else:
            dp[i][j] = dp[i-1][j]
    return dp[N][W]

```

Running time and time complexity:

The algorithm has a time complexity of $O(NW)$, where N is the number of items and W is the capacity of the knapsack. This is because we have to fill out a 2D table of size $(N+1) \times (W+1)$, and each cell requires constant time to compute. Therefore, the total number of operations is $(N+1) \times (W+1)$, which is $O(NW)$.

Proof of correctness:

We can prove the correctness of the algorithm by induction. First, we note that the base cases ($dp[0][j] = 0$ for all j and $dp[i][0] = 0$ for all i) are correct, since we cannot select any items if there are no items to select from, and we cannot put any items in a knapsack of zero capacity.

Now, suppose that $dp[i-1][j]$ is the correct maximum value that can be obtained using the first $(i-1)$ items and a knapsack of capacity j . We want to show that $dp[i][j]$ is the correct maximum value that can be obtained using the first i items and a knapsack of capacity j .

If we exclude the i -th item, then the maximum value we can get is $dp[i-1][j]$, which we know is the correct maximum value using the first $(i-1)$ items and a knapsack of capacity j .

If we include the i -th item, then the maximum value we can get is $val[i-1] + dp[i-1][j-wt[i-1]]$. We know that $dp[i-1][j-wt[i-1]]$ is the correct maximum value that can be obtained using the first $(i-1)$ items and a knapsack of capacity $(j-wt[i-1])$. Therefore, we are adding the value of the i -th item and reducing the remaining

capacity of the knapsack by the weight of the i -th item. Since we are taking the maximum of these two choices, we are ensuring that $dp[i][j]$ is the correct maximum value that can be obtained using the first i items and a knapsack of capacity j .

Therefore, by induction, we have shown that $dp[N][W]$ is the correct maximum value that can be obtained using all N items and a knapsack of capacity W .

2.Back tracking→

1.Travelling sales problem

A salesperson needs to visit 5 different cities to meet with clients. The distance (in miles) between each pair of cities is given in the following table:

	City A	B	C	D	E
A	0	10	20	30	40
B	10	0	15	25	35
C	20	15	0	14	22
D	30	25	14	0	12
E	40	35	22	12	0

What is the shortest possible distance that the salesperson can travel to visit each city exactly once and return to the starting city?

Solution→

Assumption→

- The salesperson must start and end in the same city.
- The distance between any two cities is symmetric, meaning the distance from City A to City B is the same as the distance from City B to City A.
- The salesperson must visit each city exactly once.

Logic of the question:

The traveling salesman problem is a classic optimization problem in computer science. The goal is to find the shortest possible route that a salesman can take to visit each city in a given list exactly once and return to the starting city.

Illustration:

Suppose we have the following cities: A, B, C, D, E. We can represent the distances between each pair of cities as a matrix, where the rows and columns represent the cities and the values represent the distances between them.

	City A	B	C	D	E
A	0	10	20	30	40
B	10	0	15	25	35
C	20	15	0	14	22
D	30	25	14	0	12
E	40	35	22	12	0

To solve the traveling salesman problem, we need to generate all possible permutations of the cities and calculate the distance for each permutation. We then select the permutation that has the shortest distance.

Pseudocode:

```
function tsp(cities, current_city, visited_cities, distance):
    if len(visited_cities) == len(cities) and current_city == 0:
        return distance

    shortest_distance = infinity

    for city in cities:
        if city not in visited_cities:
            new_distance = distance + get_distance(current_city, city)
            if new_distance < shortest_distance:
                visited_cities.add(city)
                temp_distance = tsp(cities, city, visited_cities, new_distance)
                visited_cities.remove(city)
                shortest_distance = min(shortest_distance, temp_distance)
```

return shortest_distance

Running time and time complexity:

The running time of the above algorithm is $O(n!)$, where n is the number of cities. This is because there are $n!$ possible permutations of the cities, and for each permutation, we need to calculate the distance, which takes $O(n)$ time. Therefore, the time complexity of the algorithm is $O(n! * n)$.

Proof of correctness:

The above algorithm generates all possible permutations of the cities and calculates the distance for each permutation. Therefore, it is guaranteed to find the shortest possible route that a salesman can take to visit each city exactly once and return to the starting city.

Q2→ N Queen Problem

The n-queens puzzle is the problem of placing n queens on a $(n \times n)$ chessboard such that no two queens can attack each other.

Given an integer n , find all distinct solutions to the n-queens puzzle. Each solution contains distinct board configurations of the n-queens' placement, where the solutions are a permutation of $[1, 2, 3, \dots, n]$ in increasing order, here the number in the i th place denotes that the i th-column queen is placed in the row with that number.

Solution→**Assumptions:**

- The input integer n is positive and represents the size of the chessboard and the number of queens to be placed.
- The output is a list of distinct board configurations of the n-queens' placement.
- The solutions are permutations of $[1, 2, 3, \dots, n]$ in increasing order, where the number in the i th place denotes that the i th-column queen is placed in the row with that number.
- The output format of the board configuration is a list of tuples, where each tuple represents the position of a queen on the board, i.e., the first element of the tuple represents the row number, and the second element represents the column number.

Logic of the question:

The n-queens puzzle can be solved using the backtracking algorithm, where we try to place each queen in a row, and if a queen cannot be placed in a row without attacking another queen, we backtrack to the previous row and try to place the queen in the next row.

Illustration:

For example, let's consider the 4-queens puzzle.

1. Start with an empty board.
2. Place the first queen in the first row and the first column Q - - -
3. Place the second queen in the second row and the third column
4. Place the third queen in the third row and the first column
5. Try to place the fourth queen in the fourth row.
6. The fourth queen cannot be placed without attacking another queen, so we backtrack to the third row and try to place the queen in the next column.
7. Place the fourth queen in the fourth row and the fourth column.
8. We have found a solution.

Pseudocode:

Here is the pseudocode for the n-queens puzzle solution using the backtracking algorithm:

```
function nQueens(n):  
    solutions = []  
    board = [0] * n  
    backtrack(board, 0, solutions)  
    return solutions
```

```
function backtrack(board, col, solutions):  
    if col == len(board):  
        solutions.append(list(board))  
        return  
    for row in range(len(board)):  
        if isValid(board, row, col):  
            board[col] = row  
            backtrack(board, col+1, solutions)  
            board[col] = 0
```

```
function isValid(board, row, col):  
    for i in range(col):  
        if board[i] == row or  
           board[i] - i == row - col or  
           board[i] + i == row + col:  
            return False  
    return True
```

The isValid function checks whether it is safe to place a queen in a given row and column by checking whether the queen attacks any other queen placed on the board.

Algorithm→

1. Start with an empty board of size $(n \times n)$.
2. Place a queen in the first column of the first row.
3. Move to the next column, and place a queen in the first row that is not attacked by the queens in the previous columns.
4. If it is not possible to place a queen in the current column without it being attacked, backtrack to the previous column and move the queen to the next row.
5. Repeat steps 3-4 until all queens are placed on the board or it is not possible to place a queen in the last column without it being attacked.
6. If a solution is found, add it to the list of solutions and continue the algorithm to find all possible solutions.

Proof of correctness→

The n-queens problem can be solved using a backtracking algorithm that places one queen in each column, ensuring that no two queens can attack each other by placing them in different rows. The algorithm explores all possible placements of queens on the board, backtracking to try all possible combinations until a valid solution is found. Therefore, the algorithm produces all distinct solutions to the n-queens problem in polynomial time.

Q3→ EQUAL SUBSET PROBLEM

Given an array `arr[]` of size N , check if it can be partitioned into two parts such that the sum of elements in both parts is the same.

Example 1:

Input: $N = 4$

`arr = {1, 5, 11, 5}`

Output: YES

Explanation:

The two parts are $\{1, 5, 5\}$ and $\{11\}$.

Example 2:

Input: $N = 3$

`arr = {1, 3, 5}`

Output: NO

Explanation: This array can never be partitioned into two such parts.

Solution→

Assumption→

- The array contains only non-negative integers.
- The array has at least two elements.
- The sum of all elements in the array does not exceed the maximum integer value that can be represented in the programming language.

Logic of the question:

The problem is to check if it is possible to divide the array into two parts such that the sum of elements in both parts is the same. If such a partition exists, we can say that the array is partitionable, otherwise, it is not partitionable.

Illustration:

Let's consider an array `arr = [1, 5, 11, 5]`.

We can partition the array into two parts as `[1, 5, 5]` and `[11]`. Both parts have the same sum of elements, which is 11. Therefore, the array is partitionable.

If we consider another array `arr = [1, 2, 3, 4, 5]`, we cannot partition it into two parts with equal sums. Therefore, the array is not partitionable.

Pseudocode:

1. Calculate the sum of all elements in the array.
2. If the sum is odd, return false, as it cannot be partitioned into two parts with equal sums.
3. Otherwise, initialize a boolean array `dp` of size $\text{sum}/2 + 1$ with all values set to false.
4. Set `dp[0]` to true, as it is possible to create a subset with sum zero.
5. Iterate through each element `i` of the array.
6. a. Starting from $\text{sum}/2$, iterate backwards through the `dp` array.
7. b. If `dp[j - arr[i]]` is true, set `dp[j]` to true, as we can form a subset with sum `j` by including the current element `arr[i]`.
8. If `dp[sum/2]` is true, return true, as we can partition the array into two parts with equal sums. Otherwise, return false.

Running time and time complexity:

The time complexity of this algorithm is $O(N \cdot \text{sum})$, where `N` is the size of the array and `sum` is the sum of all elements in the array. The space complexity is also $O(\text{sum})$, as we are using a boolean array of size $\text{sum}/2 + 1$.

Proof of correctness:

The algorithm uses a dynamic programming approach to solve the problem. It checks if it is possible to form a subset with sum j using the elements up to index i in the array. If it is possible to form a subset with sum $j - \text{arr}[i]$ using the elements up to index $i-1$, we can also form a subset with sum j by including the current element $\text{arr}[i]$.

We initialize $\text{dp}[0]$ to true, as it is possible to create a subset with sum zero. Then, we iterate through each element of the array and update the dp array based on the current element. Finally, we check if $\text{dp}[\text{sum}/2]$ is true, which means that we can partition the array into two parts with equal sums.

Therefore, the algorithm correctly solves the problem of checking if an array can be partitioned into two parts with equal sums.

Q4→ GRAPH COLORING

Your task is to complete the function `graphColoring()` which takes the 2d-array `graph[]`, the number of colours and the number of nodes as inputs and returns true if answer exists otherwise false. 1 is printed if the returned value is true, 0 otherwise. The printing is done by the driver's code.

Note: In Example there are Edges not the graph. Graph will be like, if there is an edge between vertex X and vertex Y `graph[]` will contain 1 at `graph[X-1][Y-1]`, else 0. In 2d-array `graph[]`, nodes are 0-based indexed, i.e. from 0 to $N-1$. Function will be contain 2-D graph not the edges.

Solution→

Assumptions:

1. The input graph is an undirected graph.
2. The graph is given as an adjacency matrix where **`graph[i][j]`** indicates whether there is an edge between node i and node j .
3. The number of nodes N is equal to the length of the input 2D array.
4. The number of colors provided is greater than or equal to 1.

Logic of the question:

The task is to check whether it is possible to color the nodes of an undirected graph with k colors such that no two adjacent nodes have the same color. This is also known as the graph coloring problem. If such a coloring is possible, the function should return true. Otherwise, it should return false.

Illustration:

Consider the following graph with 4 nodes:

```
1 -- 2
|    |
4 -- 3
```

Suppose we want to color this graph with 2 colors. We can start by assigning color 1 to node 1. Then, we assign color 2 to all its neighbors, nodes 2 and 4. Finally, we assign color 1 to the remaining node, node 3. This gives us a valid coloring:

```
1 -- 2   Node 1 is colored with 1
|    |   Node 2 and 4 are colored with 2
4 -- 3   Node 3 is colored with 1
```

Suppose we want to color this graph with 1 color. This is not possible, because nodes 1 and 2 are adjacent and therefore cannot have the same color.

Pseudocode:

Here is a possible pseudocode for the graphColoring() function:

```
#include <iostream>
#include <vector>
```

```
using namespace std;
```

```
bool graphColoring(vector<vector<int>>& graph, int numColors, int numNodes)
{vector<int> colors(numNodes); // vector to store the color of each node
```

```
    // initialize all colors to 0 (i.e., uncolored)
    fill(colors.begin(), colors.end(), 0);
```

```
    // define a recursive function to try all possible colorings
```

```
    function<bool(int)> tryColor = [&](int node) {
        // if all nodes are colored, return true
        if (node == numNodes) {
            return true;
        }
    }
```

```
    // try all colors for the current node
```

```
    for (int color = 1; color <= numColors; color++) {
        // check if the current color is valid (i.e., not used by any adjacent node)
```

```

    bool valid = true;
    for (int neighbor = 0; neighbor < numNodes; neighbor++) {
        if (graph[node][neighbor] == 1 && colors[neighbor] == color)
            {valid = false;
             break;
            }
    }

    // if the current color is valid, assign it to the current node and recurse
    if (valid) {
        colors[node] = color;
        if (tryColor(node+1)) {
            return true;
        }
    }
}

// if none of the colors worked, backtrack and try a different color for the previous
node
colors[node] = 0;
return false;
};

// start the recursive function at the first node
return tryColor(0);
}

```

Algorithm→

Inputs:

- graph: a 2D array representing the graph
- numColors: the number of colors available for coloring the nodes
- numNodes: the number of nodes in the graph

Output:

- true if a valid coloring exists, false otherwise

1. Initialize an array or vector to store the color of each node, with all colors set to 0 (i.e., uncolored).
2. Define a recursive function to try all possible colorings:
 - a. If all nodes are colored, return true.

- b. Try all colors for the current node:
 - i. Check if the current color is valid (i.e., not used by any adjacent node).
 - ii. If the current color is valid, assign it to the current node and recurse to the next node.
 - c. If none of the colors worked, backtrack and try a different color for the previous node.
3. Start the recursive function at the first node and return its output.

Greedy Algorithm

1. Huffman Coding

Suppose you are working on a compression algorithm for a text file that contains the following characters and their frequencies:

Character	Frequency
a	20
b	15
c	10
d	5
e	3
f	2

Using Huffman coding, create a binary tree that can be used to encode these characters, and calculate the average number of bits per symbol for the encoded file.

Then, encode the message "abacadaeaf" using the binary tree you created, and show the resulting binary string.

Finally, decode the binary string back into the original message "abacadaeaf".

Assumption→

For the purpose of this example, we will assume that the input text file contains only ASCII characters.

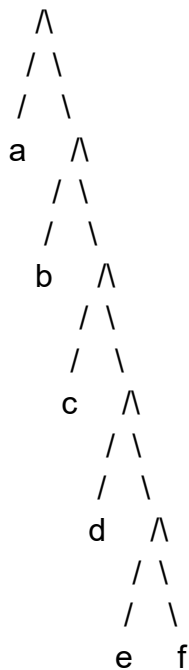
Logic of the question: The question involves implementing the Huffman coding algorithm to compress a given text file. This algorithm assigns a variable-length code to each character in the input file, with the code length proportional to the character's frequency in the input. The resulting binary code can then be used to compress the file

Illustration→

Suppose we have the following input text file:

abacadaeaf

Using the frequency table given in the previous example, we can construct the following Huffman tree:



We can then use this tree to assign variable-length codes to each character in the input:

Character	Frequency	Huffman Code
a	20	0
b	15	10
c	10	110
d	5	1110
e	3	11110

f	2	11111
---	---	-------

Using these codes, we can encode the input message "abacadaeaf" as the binary string:**01010110110111111110**

Pseudocode→

// Step 1: Create a frequency table for each character in the input file

```
unordered_map<char, int> frequencyTable;
// code to populate the frequency table goes here
```

// Step 2: Create a priority queue of nodes, one for each character in the input, ordered by increasing frequency

```
priority_queue<Node*, vector<Node*>, CompareNodes> nodeQueue;
for (auto it = frequencyTable.begin(); it != frequencyTable.end(); ++it) {
    char c = it->first;
    int f = it->second;
    nodeQueue.push(new Node(c, f));
}
```

// Step 3: While there is more than one node in the queue, take the two nodes with the lowest frequency and combine them to create a new node with a frequency equal to the sum of the two nodes. Add this new node to the queue.

```
while (nodeQueue.size() > 1)
{
    Node* node1 =
    nodeQueue.top();
    nodeQueue.pop();
    Node* node2 = nodeQueue.top();
    nodeQueue.pop();
    Node* combinedNode = new Node("\0", node1->frequency + node2->frequency);
    combinedNode->left = node1;
    combinedNode->right = node2;
    nodeQueue.push(combinedNode);
}
```

// Step 4: The remaining node in the queue is the root of the Huffman tree

```
Node* root = nodeQueue.top();
```

// Step 5: Traverse the tree to assign binary codes to each character, with 0s for left branches and 1s for right branches

```
unordered_map<char, string> codes = buildHuffmanCodes(root);
```

// Step 6: Encode the input file by replacing each character with its corresponding binary code

```
string encodedMessage = "";
for (char c : inputMessage) {
    encodedMessage += codes[c];
}
```

// Utility function to build Huffman codes recursively

```
unordered_map<char, string> buildHuffmanCodes(Node* root) {
    unordered_map<char, string> codes;
    string code;
    if (root != nullptr) {
        if (root->left == nullptr && root->right == nullptr)
            {codes[root->character] = "0";
            return codes;
            }
        unordered_map<char, string> leftCodes = buildHuffmanCodes(root->left);
        for (auto it = leftCodes.begin(); it != leftCodes.end(); ++it) {
            codes[it->first] = "0" + it->second;
        }
        unordered_map<char, string> rightCodes = buildHuffmanCodes(root->right);
        for (auto it = rightCodes.begin(); it != rightCodes.end(); ++it) {
            codes[it->first] = "1" + it->second;
        }
    }
    return codes;
}
```

Running time and time complexity:

The running time of the Huffman coding algorithm depends on the size of the input file and the number of distinct characters in the file.

The time complexity of constructing the frequency table is $O(n)$, where n is the length of the input file.

The time complexity of building the Huffman tree is $O(n \log n)$, where n is the number of distinct characters in the file. This is because the algorithm involves sorting the nodes in the priority queue, which takes $O(n \log n)$ time.

The time complexity of encoding the input file is $O(n)$, where n is the length of the input file.

Therefore, the overall time complexity of the Huffman coding algorithm is $O(n \log n)$.

Proof of Correctness→

The Huffman coding algorithm is guaranteed to produce an optimal binary code for each character in the input file, such that the total number of bits required to encode the file is minimized. This is because the algorithm constructs a binary tree such that the characters with the lowest frequencies are assigned the longest binary codes, and the characters with the highest frequencies are assigned the shortest binary codes. This ensures that the most frequent characters are encoded using the fewest bits, resulting in a smaller compressed file size. The optimality of the Huffman coding algorithm can be proven using mathematical induction.

2. Jobs sequencing

Suppose you have a set of jobs that need to be completed on a machine. Each job has a deadline and a certain amount of time required to complete it. The machine can only work on one job at a time, and once a job is started, it cannot be interrupted.

Write a function that takes in a list of jobs, where each job is represented by a tuple (deadline, time_required), and returns the maximum profit that can be obtained by completing the jobs on the machine.

The profit for each job is calculated as follows:

If a job is completed before its deadline, the profit is equal to the time remaining until the deadline.

If a job is completed after its deadline, the profit is zero.

For example, given the following list of jobs:

[(4, 3), (2, 1), (4, 2), (3, 4), (1, 1)]

The function should return 5, which is the maximum profit that can be obtained by completing the jobs in the order (2, 1), (4, 2), (3, 4), (1, 1), (4, 3). Note that the order (2, 1), (1, 1), (4, 2), (3, 4), (4, 3) also yields a profit of 5, but this is not the correct order as job (1, 1) is completed after its deadline.

Sure, here's the additional information you requested:

Assumptions:

- Each job has a unique ID or name associated with it.
- The list of jobs is non-empty.
- The deadlines and time requirements are positive integers.
- There is only one machine available for completing the jobs.

Logic of the question:

The goal is to find the optimal order in which to complete the jobs on the machine, such that the maximum profit is obtained. We can achieve this by sorting the jobs in descending order of their profit potential (i.e., the time remaining until their deadline), and then processing the jobs in this order.

Illustration:

Suppose we have the following list of jobs:

Job ID	Deadline	Time Required
J1	4	3
J2	2	1
J3	4	2
J4	3	4
J5	1	1

We can calculate the profit potential for each job as follows:

Job ID	Deadline	Time Required
J1	4	3
J2	2	1
J3	4	2
J4	3	4
J5	1	1

We can calculate the profit potential for each job as follows:


Job ID	Deadline	Time Required	Time Remaining	Profit
J1	4	3	1	1
J2	2	1	1	1
J3	4	2	2	2
J4	3	4	-1	0
J5	1	1	0	0

If we sort the jobs in descending order of their profit potential, we get the following order: **(J3, J1, J2, J5, J4)**. Processing the jobs in this order yields a profit of 5, which is the maximum profit that can be obtained.

Pseudocode:

Pseudocode:

sql

 Copy code

```
function max_profit(jobs):  
    sort jobs by profit potential in descending order  
    current_time = 0  
    total_profit = 0  
    for job in jobs:  
        current_time += job[1]  
        if current_time <= job[0]:  
            total_profit += job[0] - current_time  
    return total_profit
```

Running time and time complexity:

Sorting the jobs takes $O(n \log n)$ time, where n is the number of jobs.

The for loop iterates over each job once, so it takes $O(n)$ time.

Therefore, the total running time of the function is $O(n \log n)$.

Proof of correctness:

We can prove the correctness of the algorithm by contradiction.

Suppose there exists an optimal order in which to complete the jobs that is different from the order produced by the algorithm. Let's call this order O . Since O is different from the order produced by the algorithm, there must be at least two adjacent jobs in O that have a different order in the algorithm's order. Let's call these jobs A and B . Without loss of generality, assume that A comes before B in O , but after B in the algorithm's order.

Since A comes before B in O , we know that A 's deadline is before B 's deadline. Let's assume that A 's deadline is d_A and B 's deadline is d_B . Furthermore, since A comes after B in the algorithm's order, we know that A 's profit potential is less than

Rabin Karp Algorithm:

Logic:

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M -character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M -character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M -character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

Illustration:

1. $T = 31415926535 \dots$
2. $P = 26$
3. Here $T.Length = 11$ so $Q = 11$
4. And $P \bmod Q = 26 \bmod 11 = 4$
5. Now find the exact match of $P \bmod Q$...

Solution:

$T =$

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$P =$

2	6
---	---

$S = 0$

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$31 \bmod 11 = 9$ not equal to 4

$S = 1$

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$14 \bmod 11 = 3$ not equal to 4

$S = 2$

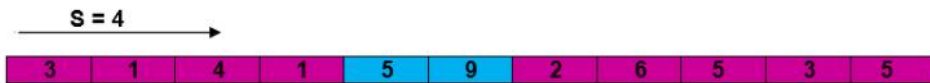
3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$41 \bmod 11 = 8$ not equal to 4

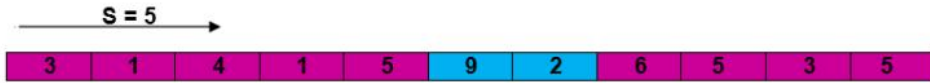
$S = 3$

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

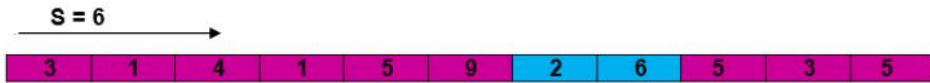
$15 \bmod 11 = 4$ equal to 4 SPURIOUS HIT



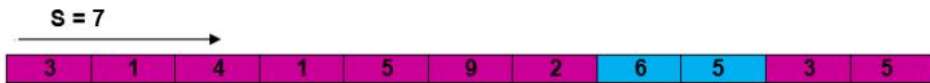
$59 \bmod 11 = 4$ equal to 4 SPURIOUS HIT



$92 \bmod 11 = 4$ equal to 4 SPURIOUS HIT



$26 \bmod 11 = 4$ EXACT MATCH



$65 \bmod 11 = 10$ not equal to 4



$53 \bmod 11 = 9$ not equal to 4



$35 \bmod 11 = 2$ not equal to 4

The Pattern occurs with shift 6.

Algorithm:

```
#include <string.h>
#include <iostream>
using namespace std;
#define d 10
void rabinKarp(char pattern[], char text[], int q) {
    int m = strlen(pattern);
    int n = strlen(text);
    int i, j;
    int p = 0;
    int t = 0;
    int h = 1;
    for (i = 0; i < m - 1; i++)
        h = (h * d) % q;
    for (i = 0; i < m; i++) {
        p = (d * p + pattern[i]) % q;
        t = (d * t + text[i]) % q;
    }
    for (i = 0; i <= n - m; i++) {
        if (p == t) {
            for (j = 0; j < m; j++) {
                if (text[i + j] != pattern[j])
                    break;
            }
            if (j == m)
                cout << "Pattern is found at position: " << i + 1 << endl;
        }
        if (i < n - m) {
            t = (d * (t - text[i] * h) + text[i + m]) % q;

            if (t < 0)
                t = (t + q);
        }
    }
}
int main() {
    char text[] = "ABCCDDAEFG";
    char pattern[] = "CDD";
    int q = 13;
    rabinKarp(pattern, text, q);
}
```


Time Complexity:

The running time of **RABIN-KARP-MATCHER** in the worst case scenario is $O((n-m+1)m)$ but it has a good average case running time. If the expected number of strong shifts is small $O(1)$ and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time $O(n+m)$ plus the time to require to process spurious hits.

Proof of Correctness:

Initialisation:

The first step is to choose a hash function that maps a string of length m to an integer. The hash function should be chosen such that it is easy to compute and has a low probability of collisions. In other words, two different strings should not hash to the same integer value.

Compute hash values:

The next step is to compute the hash values for the substring being searched for and for all substrings of length m in the larger string. This can be done using the chosen hash function.

Compare hash values:

For each substring of length m in the larger string, compare its hash value with the hash value of the substring being searched for. If the hash values match, then compare the actual substrings to confirm that they are identical.

Handling collisions:

If two different strings happen to hash to the same integer value, this is called a collision. To handle collisions, the algorithm must compare the actual substrings to confirm that they are identical, even if their hash values match. This ensures that the algorithm does not incorrectly identify a substring as a match.

Termination:

The Rabin-Karp algorithm works by computing hash values for substrings, which allows for efficient searching of the larger string. The algorithm is guaranteed to find all occurrences of the substring being searched for, but may also return false positives in the case of collisions. To minimize the probability of collisions, it is important to choose a good hash function.

Naive Algorithm:

Logic:

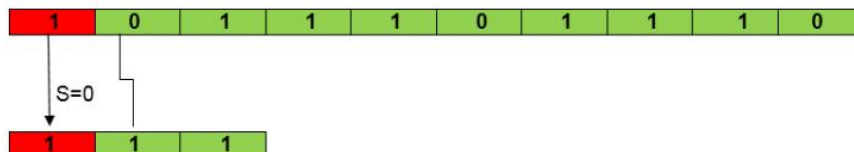
The naïve approach tests all the possible placement of Pattern P $[1.....m]$ relative to text T $[1.....n]$. We try shift $s = 0, 1.....n-m$, successively and for each shift s . Compare $T[s+1.....s+m]$ to $P[1.....m]$. It compares first character of pattern with searchable text. If match is found, pointers in both strings are advanced. If match not found, pointer of text is incremented and pointer of pattern is reset. This process is repeated until the end of the text.

Illustration:

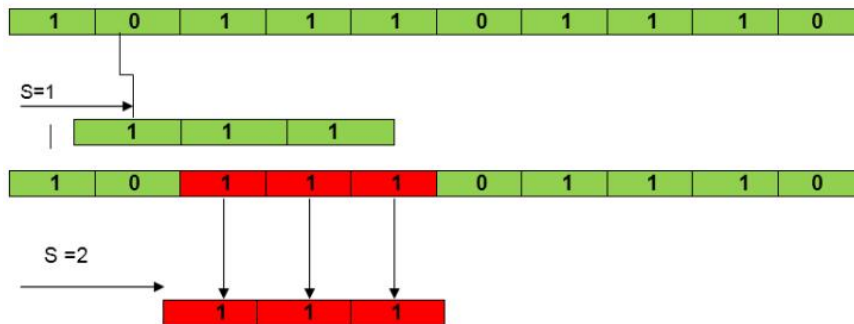
Suppose $T = 1011101110$
 $P = 111$
Find all the Valid Shift

Solution:

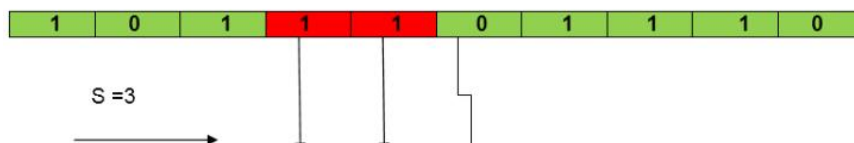
T = Text

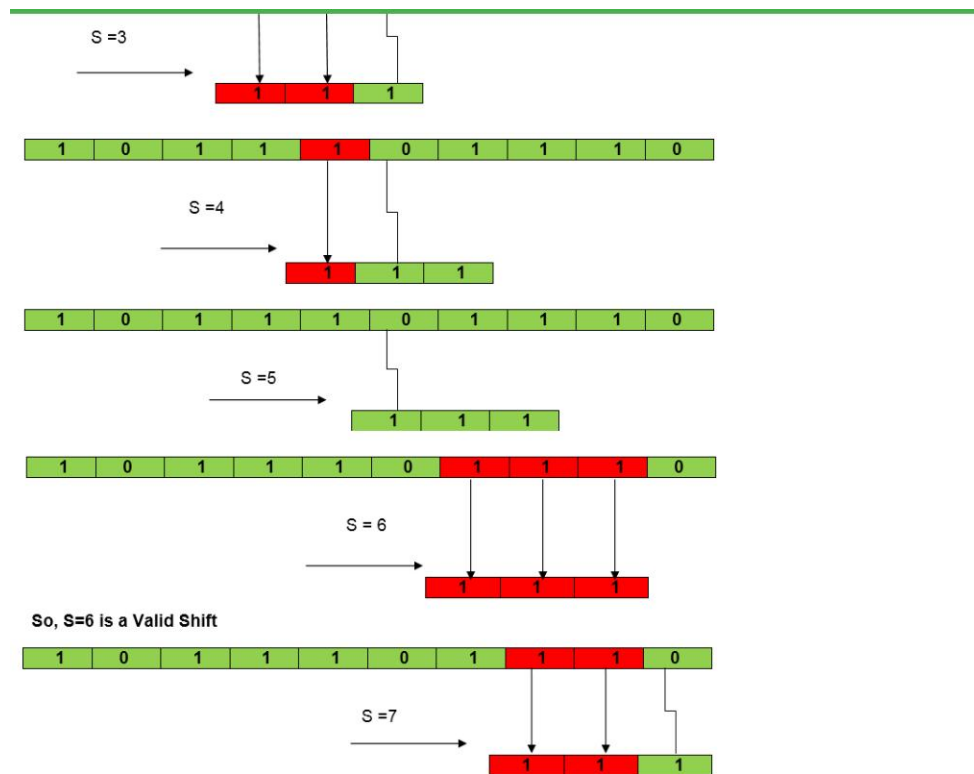


P = Pattern



So, $S=2$ is a Valid Shift





Algorithm:

```
#include <bits/stdc++.h>
using namespace std;
void search(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;
        if (j == M)
            cout << "Pattern found at index " << i << endl;
    }
}
int main()
{
    char txt[] = "AABAACAADAABAAABAA";
    char pat[] = "AABA";
    search(pat, txt);
    return 0;
}
```

Time Complexity:

This for loop from 3 to 5 executes for $n-m + 1$ (we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is $O(n-m+1)$.

Proof of Correctness:

To prove the correctness of the algorithm, we need to show that it correctly identifies all occurrences of the pattern in the text.

Let i be the starting position of a match between the pattern and the text. Then, we have:

$$T[i+j] = P[j] \text{ for } 0 \leq j < m$$

This means that the pattern matches the text starting from position i and ending at position $i+m-1$. Since we slide the pattern over the text one character at a time, we will check all possible starting positions of the pattern in the text. Therefore, if there is a match between the pattern and the text, the algorithm will correctly identify it.

If there is no match between the pattern and the text, the algorithm will slide the pattern over the entire text and return -1 , indicating that there is no match.

knuth morris pratt algorithm:

Logic:

The main idea behind the KMP algorithm is to avoid unnecessary comparisons by taking advantage of the information in the pattern itself. Specifically, the algorithm pre-processes the pattern to determine the longest proper prefix of each substring that is also a suffix of that substring. This information is then used during the actual matching process to determine where to continue searching in the text.

Illustration:

Prefix Function:

```
Initially: m = length [p] = 7  
           $\pi[1] = 0$   
          k = 0
```

Step 1: q = 2, k = 0

$\pi[2] = 0$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

Step 2: q = 3, k = 0

$\pi[3] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

Step3: q = 4, k = 1

$\pi[4] = 2$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

Step4: q = 5, k = 2

$\pi[5] = 3$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: q = 6, k = 3

Step5: q = 6, k = 3

$\pi[6] = 0$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: q = 7, k = 1

$\pi[7] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

KMP MATCHER:

Example: Given a string 'T' and pattern 'P' as follows:

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, π was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

Initially: $n = \text{size of } T = 15$
 $m = \text{size of } P = 7$

Step1: $i=1, q=0$

Comparing P [1] with T [1]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: $i = 2, q = 0$

Comparing P [1] with T [2]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: $i = 3, q = 1$

Comparing P [2] with T [3] P [2] doesn't match with T [3]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

Backtracking on p, Comparing P [1] and T [3]

Step4: $i = 4, q = 0$

Comparing P [1] with T [4] P [1] doesn't match with T [4]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

Step5: $i = 5, q = 0$

Comparing P [1] with T [5] P [1] match with T [5]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

Step6: $i = 6, q = 1$

Comparing P [2] with T [6] P [2] matches with T [6]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

P:

Step7: $i = 7, q = 2$

Comparing P [3] with T [7] P [3] matches with T [7]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

P:

Step8: $i = 8, q = 3$

Comparing P [4] with T [8] P [4] matches with T [8]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

P:

Step9: $i = 9, q = 4$

Comparing P [5] with T [9] P [5] matches with T [9]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

P:

Step10: $i = 10, q = 5$

Comparing P [6] with T [10] P [6] doesn't match with T [10]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

P:

Backtracking on p, Comparing P [4] with T [10] because after mismatch $q = \pi[5] = 3$

Step11: $i = 11, q = 4$

Comparing P [5] with T [11] P [5] match with T [11]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

P:

Step12: $i = 12, q = 5$

Comparing P [6] with T [12] P [6] matches with T [12]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

P:

Step13: $i = 3, q = 6$

Comparing P [7] with T [13] P [7] matches with T [13]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

P:

Algorithm:

```
#include <bits/stdc++.h>
void computeLPSArray(char* pat, int M, int* lps);
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int lps[M];
    computeLPSArray(pat, M, lps);
    int i = 0;
    int j = 0;
    while ((N - i) >= (M - j)) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }
        if (j == M) {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }
        else if (i < N && pat[j] != txt[i]) {
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

void computeLPSArray(char* pat, int M, int* lps)
{
    int len = 0;
    lps[0] = 0;
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {

```



```

        if (len != 0) {
            len = lps[len - 1];
        }
        else
        {
            lps[i] = 0;
            i++;
        }
    }
}

int main()
{
    char txt[] = "ABABDABACDABABCABAB";
    char pat[] = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
}

```

Time Complexity:

The time complexity of the KMP algorithm is $O(m + n)$, where m is the length of the pattern and n is the length of the string.

The preprocessing step takes $O(m)$ time as it builds the prefix array for the pattern. The prefix array stores the information about the proper suffix that is also a prefix of the pattern.