KLE TECH.

KLE Technological University

Creating Value
Leveraging Knowledge

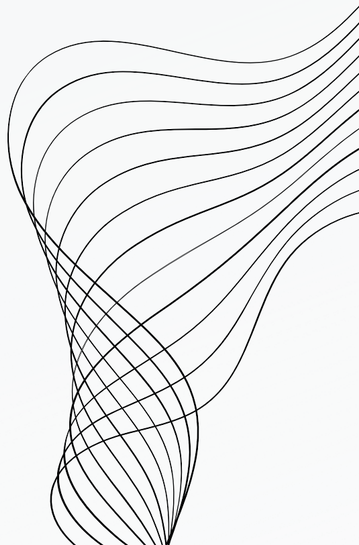
DEPARTMENT OF ELECTRONICS
AND
COMMUNICATION ENGINEERING

DR. M. S. SHESHGIRI COLLEGE OF ENGINEERING AND TECHNOLOGY, BELAGAVI CAMPUS

ASSESSMENT-1

DATA STRUCTURES AND APPLICATIONS

BY PRAVEEN MAGADUM



ALGORITHMS

01 ITERATIVE ALGORITHM

02 RECURSIVE ALGORITHM

03 BACKTRACKING ALGORITHM

04 DEVIDE AND CONQUER ALGORITHM

05 DYNAMIC PROGRAMMING

06 GREEDY ALGORITHM

07 BRANCH AND BOUND

08 BRUTE FORCE ALGORITHM

09 RANDOMIZED APPROACH

ALGORITHMS

An algorithm is a procedure used for solving a problem or performing a computation. Algorithms act as an exact list of instructions that conduct specified actions step by step in either hardware- or software-based routines.

Algorithms are widely used throughout all areas of IT. In mathematics and computer science, an algorithm usually refers to a small procedure that solves a recurrent problem. Algorithms are also used as specifications for performing data processing and play a major role in automated systems.

Types of algorithms

Search engine algorithm

Encryption algorithm

Greedy algorithm

Recursive algorithm

Backtracking algorithm

Divide-and-conquer algorithm

Dynamic programming algorithm

Brute-force algorithm

Sorting algorithm

Hashing algorithm

Randomized algorithm

TYPES OF ALGORITHMS

1) Iterative Algorithm

- An iterative algorithm executes steps in iterations. It aims to find successive approximation in sequence to reach a solution

- Iteration is when the same procedure is repeated multiple times. Some examples were long division, the Fibonacci numbers, prime numbers, and the calculator game. Some of these used recursion as well, but not all of them. bunch of successive integers, or repeat a procedure a given number of times.

Example 1

```
// File processing line by line

#include<stdio.h>
int main() {
FILE *file = fopen("log.txt", "r");
if (file == NULL) {
printf("Error opening file.\n");
return 1;}
char line[256];
while (fgets(line, sizeof(line), file) != NULL)
{
// Process each line of the file
printf("Line: %s", line);
// Perform analysis or filtering
// Uncomment the line below to add a delay between processing each line
// usleep(500000); // Pause for 500 milliseconds (0.5 seconds)
}
fclose(file);
return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 1 :

File processing line by line
Opening a file-O(1) doesnot depend on the size of the file
Reading line by line and processing it –O(n)
Therefore ,
T(n) =O(n)

EXAMPLE 2:

```
// Timer Program to count the time in Reverse way

#include <stdio.h>
#include <unistd.h>

int main() {
    int timer = 10; // Start from 10 seconds

    while (timer >= 0)
 {
        printf("Timer: %d\n", timer);
        sleep(1); // Pause for 1 second
        timer--;
    }

    printf("Time's up!\n");

    return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 2 :
Timer

- Initialization – O(1) because it involves a simple assignment operation
- While Loop – O(timer) The while loop iterates until the "timer" variable becomes less than 0. The number of iterations in the while loop is equal to the initial value of "timer" plus one, as the loop will execute even when "timer" is 0 before the decrement operation. Therefore, the time complexity of the while loop is O(timer).
- Print Statement – O(1) it involves the printing the value of the timer which is constant
- Sleep function will not have any time complexcity
- Therefore the total time complexcity will be O(n).

TYPES OF ALGORITHMS

2) Recursive Algorithm

- This algorithm calls itself repeatedly until it solves a problem. Recursive algorithms call themselves with a smaller value every time a recursive function is invoked.
- The classic example of recursive programming involves computing factorials. The factorial of a number is computed as that number times all of the numbers below it up to and including 1. For example, factorial(5) is the same as 5*4*3*2*1 , and factorial(3) is 3*2*1 .

EXAMPLE 1 :

```
//Exponential Calculation

#include <stdio.h>
int power(int base, int exponent)
{
if (exponent == 0) return 1;
else
return base * power(base, exponent - 1);
}
int main()
{
int base, exponent;
printf("Enter the base number: "); scanf("%d", &base);
printf("Enter the exponent: "); scanf("%d", &exponent);
int result = power(base, exponent);
printf("%d raised to the power of %d is %d\n", base, exponent, result);

return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 1:
Exponential calculation

- The time complexity of this program can be analyzed by looking at the recursive function power.
- The power function calculates the result of raising base to the power of exponent. It uses recursion to repeatedly multiply base by itself for exponent number of times.
- In the worst case scenario, where the exponent is a positive integer, the power function will be called recursively exponent number of times. This results in a linear relationship between the input size (exponent) and the number of function calls.
- Therefore, the time complexity of the power function is O(exponent).
- T(n) = O(exponent)

Example 2 :

```
// Program to find the factorial of a number 5

#include <stdio.h>
int f(int n)
{
 // Stop condition
 if (n == 0 || n == 1)
  return 1;
 // Recursive condition
 else
  return n * f(n - 1);
}

// Driver code
int main()
{
 int n = 5;
 printf("factorial of %d is: %d", n, f(n));
 return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 2 :
Program to find the factorial of number 5

T(n) =1 for n=0
T(n) = 1+T(n-1) for n>0
So,
Time complexity = 0(n)

TYPES OF ALGORITHMS

3) Backtracking Algorithm

- This algorithm finds a solution to a given problem in incremental approaches and solves it one piece at a time
- Backtracking is a general algorithmic technique used for finding solutions to problems by incrementally building candidates and backtracking when they are determined to be invalid or unsuitable. It involves exploring all possible paths or choices and undoing certain choices when they are found to be incorrect. Backtracking is often used in combination with recursive algorithms.

EXAMPLE 1 :

```
// C program to print all permutations with duplicates

#include <stdio.h>
#include <string.h>
void swap(char* x, char* y)
{
 char temp;
 temp = *x;
 *x = *y;
 *y = temp;
}

void permute(char* a, int l, int r)
{
 int i;
 if (l == r)
  printf("%s\n", a);
 else {
  for (i = l; i <= r; i++) {
   swap((a + l), (a + i));
 permute(a, l + 1, r);
 swap((a + l), (a + i)); // backtrack
 }
 }
}
```

```
/* Driver code */
int main()
{
 char str[] = "ABC";
 int n = strlen(str);
 permute(str, 0, n - 1);
 return 0;
}
```

EXAMPLE 2 :
//N queens Algorithm

```
#include <stdio.h>
#define N 8
int isSafe(int board[N][N], int row, int col) {
    int i, j;
    // Check the left side of the current row
    for (i = 0; i < col; i++)
        if (board[row][i])
            return 0;
```

```
// Check the upper diagonal on the left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return 0;
    // Check the lower diagonal on the left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return 0;
    return 1;
}
int solveNQueens(int board[N][N], int col) {
 // Base case: If all queens are placed, return true
 if (col >= N)
 return 1;
 // Consider each row in the current column
 for (int i = 0; i < N; i++) {
 // Check if the queen can be placed in the current position
 if (isSafe(board, i, col)) {
 // Place the queen
 board[i][col] = 1;
 // Recursively solve the problem for the next column
 if (solveNQueens(board, col + 1))
```

```
            return 1;
        // If placing the queen in the current position doesn't lead to a solution,
        // backtrack by removing the queen from the current position
        board[i][col] = 0;
      }
    }
    // If no queen can be placed in the current column, return false
    return 0;
}

int main() {
    int board[N][N] = {0};

    if (solveNQueens(board, 0))
        printBoard(board);
    else
        printf("No solution found.");

    return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 1 :
C program to print all permutations with duplicates

- T(n)=O(N * N!)
- i.e. there are N! permutations and it requires O(N) time to print a permutation.

Example 2 :
N queens Problem

- Let N be the number of variables (queens in this case) and M be the number of possible values (positions on the chessboard).
- In the N-Queens problem, N is equal to M. The worst-case time complexity of backtracking is generally exponential
-
- T(n)= O(M^N).
- In the N-Queens problem,
- T(n)=O(N^N) ; since N = M.

TYPES OF ALGORITHMS

4) Devide and Conquer

- This common algorithm is divided into two parts. One part divides a problem into smaller subproblems. The second part solves these problems and then combines them together to produce a solution.
- Divide and conquer is a problem-solving technique that involves breaking down a complex problem into smaller, more manageable subproblems. The main idea is to divide the problem into smaller parts, solve each part independently, and then combine the solutions to solve the original problem.

EXAMPLE 1 :

```
// C code to implement quicksort

#include <stdio.h>
// Function to swap two elements
void swap(int* a, int* b)
{
 int t = *a;
 *a = *b;
 *b = t;
}
// Partition the array using the last element as the pivot
int partition(int arr[], int low, int high)
{
 // Choosing the pivot
 int pivot = arr[high];
 // Index of smaller element and indicates
 // the right position of pivot found so far
```

```
int i = (low - 1);
 for (int j = low; j <= high - 1; j++) {
  // If current element is smaller than the pivot
  if (arr[j] < pivot) {
   // Increment index of smaller element
   i++;
   swap(&arr[i], &arr[j]);
  }
 }
 swap(&arr[i + 1], &arr[high]);
 return (i + 1);
}
// The main function that implements QuickSort
// arr[] --> Array to be sorted,
// low --> Starting index,
// high --> Ending index
void quickSort(int arr[], int low, int high)
{
        if (low < high) {
```

```
  // pi is partitioning index, arr[p]
  // is now at right place
  int pi = partition(arr, low, high);

  // Separately sort elements before
  // partition and after partition
  quickSort(arr, low, pi - 1);
  quickSort(arr, pi + 1, high);
 }
}
// Driver code
int main()
{
int arr[] = { 10, 7, 8, 9, 1, 5 };
        int N = sizeof(arr) / sizeof(arr[0]);
        // Function call
        quickSort(arr, 0, N - 1);
        printf("Sorted array: \n");
        for (int i = 0; i < N; i++)
                printf("%d ", arr[i]);
        return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 1 :
Implementing the quick sort

- Best Case: omega (N * log N)
- Average Case: Theta (N * logN)
- Worst Case: O(N^2)

EXAMPLE 2 :

```
// C program for Merge Sort

#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
 int i, j, k;
 int n1 = m - l + 1;
 int n2 = r - m;
// Create temp arrays
        int L[n1], R[n2];

        // Copy data to temp arrays L[] and R[]
        for (i = 0; i < n1; i++)
                L[i] = arr[l + i];
        for (j = 0; j < n2; j++)
                R[j] = arr[m + 1 + j];
```

```
// Merge the temp arrays back into arr[l..r
 i = 0;
 j = 0;
 k = l;
 while (i < n1 && j < n2) {
  if (L[i] <= R[j]) {
   arr[k] = L[i];
   i++;
  }
  else {
   arr[k] = R[j];
   j++;
  }
  k++;
 }
// Copy the remaining elements of L[],
          // if there are any
          while (i < n1) {
                    arr[k] = L[i];
                    i++;
                    k++;
          }
```

```
// Copy the remaining elements of R[],
 // if there are any
 while (j < n2)
                {
  arr[k] = R[j];
  j++;
  k++;
 }
}

// l is for left index and r is right index of the
// sub-array of arr to be sorted
void mergeSort(int arr[], int l, int r)
{
                    if (l < r)
              {
  int m = l + (r - l) / 2;
// Sort first and second halves
                        mergeSort(arr, l, m);
                        mergeSort(arr, m + 1, r);
```

```
 merge(arr, l, m, r);
 }
}

// Function to print an array
void printArray(int A[], int size)
{
 int i;
 for (i = 0; i < size; i++)
  printf("%d ", A[i]);
 printf("\n");
}
// Driver code
int main()
{
 int arr[] = { 12, 11, 13, 5, 6, 7 };
 int arr_size = sizeof(arr) / sizeof(arr[0]);
```

```
printf("Given array is \n");
 printArray(arr, arr_size);
 mergeSort(arr, 0, arr_size - 1);
 printf("\nSorted array is \n");
 printArray(arr, arr_size);
 return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Implementing the merge sort

- Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
- $T(n) = 2T(n/2) + \theta(n)$
- $T(n) = O(N \log(N))$

TYPES OF ALGORITHMS

5)Dynamic Algorithm

- This algorithm solves problems by dividing them into subproblems. The results are then stored to be applied for future corresponding problems.
- It typically involves solving the subproblems in a bottom-up manner and storing their solutions in a table or array to avoid redundant computations. Dynamic programming is particularly useful when the problem has overlapping subproblems and exhibits optimal substructure.

EXAMPLE 1 :

```
// Finding the Fibonacci number in a sequence
#include <stdio.h>
int fibonacci(int n) {
    int fib[n + 1];
    int i;
    fib[0] = 0; // Base case: F(0) = 0
    fib[1] = 1; // Base case: F(1) = 1

    for (i = 2; i <= n; i++) {
        // Calculate the current Fibonacci number by adding the previous two numbers
        fib[i] = fib[i - 1] + fib[i - 2];
    }
    return fib[n];
}
int main() {
    int n;
    printf("Enter the value of n: ");
    scanf("%d", &n);
    printf("The %dth Fibonacci number is: %d\n", n, fibonacci(n));
    return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 1 :

Finding the Fibonacci number in a sequence
T(n)= O(n) ; because it iterates from 2 to n, performing a constant-time operation for each iteration.

EXAMPLE 2:

```
// solving the problem of finding the minimum number of coins needed to make change for a given amount

#include <stdio.h>
#include <limits.h>
int minCoins(int coins[], int n, int amount) {
    int dp[amount + 1]; // Table to store the minimum number of coins for each amount
    int i, j;

    // Initialize the table with infinity
    for (i = 1; i <= amount; i++) {
        dp[i] = INT_MAX;
    }

    // Base case: The minimum number of coins needed to make change for 0 is 0
    dp[0] = 0;

    // Compute the minimum number of coins for each amount
    for (i = 1; i <= amount; i++) {
        // Try each coin denomination
        for (j = 0; j < n; j++) {
            if (coins[j] <= i && dp[i - coins[j]] != INT_MAX) {
                // If using the current coin leads to a smaller number of coins needed, update the table
```

```
                dp[i] = dp[i] < 1 + dp[i - coins[j]] ? dp[i] : 1 + dp[i - coins[j]];
            }
        }
    }

    return dp[amount];
}

int main() {
    int coins[] = {1, 5, 10, 25};
    int n = sizeof(coins) / sizeof(coins[0]);
    int amount;

    printf("Enter the amount: ");
    scanf("%d", &amount);

    int result = minCoins(coins, n, amount);

    if (result != INT_MAX) {
        printf("Minimum number of coins needed: %d\n", result);
    } else {
        printf("No solution found.\n");
    }

    return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 2 :

- Solving the problem of finding the minimum number of coins needed to make change for a given amount
- -Solving the problem of finding the minimum number of coins needed to make change for a given amount
- The time complexity of the dynamic programming approach to solve the minimum coin change problem is O(n * amount), where n is the number of coin denominations and amount is the given amount.
- The algorithm iterates over each amount from 1 to the given amount and tries each coin denomination, resulting in an overall time complexity of O(n * amount).
- T(n)=O(n*amount)

TYPES OF ALGORITHMS

6) Greedy Algorithm

- This algorithm solves optimization problems by finding the locally optimal solution, hoping it is the optimal solution at the global level. However, it does not guarantee the most optimal solution.

- it makes the best choice at the current moment without considering the overall future consequences. Greedy algorithms often provide efficient solutions but may not always guarantee the best or globally optimal solution..

EXAMPLE 1 :

```
// Function to find the minimum number of coins needed to make change

#include <stdio.h>
// Function to find the minimum number of coins needed to make change
int coinChangeGreedy(int coins[], int numCoins, int targetAmount)
 {
    // Sort the coins in descending order
    for (int i = 0; i < numCoins - 1; i++) {
        for (int j = 0; j < numCoins - i - 1; j++) {
            if (coins[j] < coins[j + 1]) {
                int temp = coins[j];
                coins[j] = coins[j + 1];
                coins[j + 1] = temp;
            }
        }
    }

    int numCoinsUsed = 0;

    // Iterate through the coins and select the largest denomination that is less than or equal to the remaining amount
    for (int i = 0; i < numCoins; i++) {
        while (targetAmount >= coins[i]) {
            targetAmount -= coins[i];
            numCoinsUsed++;
```

```
 }
    }

    return numCoinsUsed;
}

int main() {
    int coins[] = {25, 10, 5, 1};  // Available coin denominations
    int numCoins = sizeof(coins) / sizeof(coins[0]);

    int targetAmount = 63;  // Target amount for which change is to be made

    int minCoins = coinChangeGreedy(coins, numCoins, targetAmount);

    printf("Minimum number of coins needed: %d\n", minCoins);

    return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 1 :
Function to find the minimum number of coins needed to make change

- The time complexity of the Coin Change Problem using a greedy approach in C depends on the number of coins (numCoins) and the target amount (targetAmount).
- Overall, the time complexity of the greedy approach for the Coin Change Problem is O(numCoins^2) or O(numCoins * log(numCoins)) depending on the sorting algorithm used.

EXAMPLE 2 :

```
// Greedy algorithm to solve the Activity Selection Problem

#include <stdio.h>
// Structure to represent an activity
typedef struct {
    int start;
    int finish;
} Activity;
int compareActivities(const void* a, const void* b) {
    Activity* activityA = (Activity*)a;
    Activity* activityB = (Activity*)b;

    return activityA->finish - activityB->finish;
}
// Greedy algorithm to solve the Activity Selection Problem
void activitySelectionGreedy(Activity activities[], int numActivities) {
   qsort(activities, numActivities, sizeof(Activity), compareActivities); // Sort activities based on their finish time

   printf("Selected Activities:\n");

   int previousFinishTime = -1;
```

```
    for (int i = 0; i < numActivities; i++) {
        if (activities[i].start >= previousFinishTime) {
            printf("Activity %d: Start Time = %d, Finish Time = %d\n", i+1, activities[i].start, activities[i].finish);
            previousFinishTime = activities[i].finish;
        }
    }
}

int main() {
    Activity activities[] = {
        {1, 4},
        {3, 5},
        {0, 6},
        {5, 7},
        {3, 9},
        {5, 9},
        {6, 10},
        {8, 11},
      {8, 12},
      {2, 14},
      {12, 16}
   };
```

```
    int numActivities = sizeof(activities) / sizeof(activities[0]);

    activitySelectionGreedy(activities, numActivities);

    return 0;
}
```

TIME COMPLEXITY CALCULATIONS

- Greedy algorithm to solve the activity selection problem
- The time complexity of this greedy algorithm is dominated by the sorting operation, which has a time complexity of O(numActivities * log(numActivities)).
- using an efficient sorting algorithm like quicksort or mergesort. The subsequent iteration over the sorted list has a linear time complexity of O(numActivities).
- T(n) = O(numactivities)

TYPES OF ALGORITHMS

7) Brute force Algorithm

- This algorithm iterates all possible solutions to a problem blindly, searching for one or more solutions to a function.
- It involves systematically checking all possible options or combinations without applying any optimization or heuristics. While brute force guarantees finding the optimal solution, it can be computationally expensive and inefficient for large problem spaces.

EXAMPLE 1 :

```
// Brute force algorithm for finding pattern index

#include <stdio.h>
#include <string.h>
int bruteForceStringMatch(const char* text, const char* pattern)
{
    int n = strlen(text);
    int m = strlen(pattern);
    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (text[i + j] != pattern[j])
                break;
        }
        if (j == m)
            return i;  // Pattern found at index i
    }
    return -1;  // Pattern not found
}
```

```
int main() {
    const char* text = "Hello, World!";
    const char* pattern = "World";
    int index = bruteForceStringMatch(text, pattern);
    if (index != -1)
        printf("Pattern found at index %d\n", index);
    else
        printf("Pattern not found\n");
    return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 1 :

- Brute force algorithm for finding pattern index
- In a basic brute-force approach, the algorithm iterates through the text and checks for matches with the pattern at each position. The worst-case scenario occurs when the pattern matches at the last position of the text or does not match at all. In this case, the algorithm needs to compare each character of the pattern with the corresponding character in the text for every possible starting position.
- Therefore, the time complexity of a brute-force pattern matching algorithm is approximately O((n - m + 1) * m), where n is the length of the text and m is the length of the pattern.
- The term (n - m + 1) represents the number of possible starting positions in the text where the pattern could match, and m represents the number of character comparisons needed for each starting position.
- T(n)= O((n - m + 1))

EXAMPLE 2 :

// PROGRAM TO CHECK THE GIVEN NUMBER IS PRIME NUMBER OR NOT A PRIME NUMBER

```
#include <stdio.h>
int isPrime(int num)
 {
    if (num <= 1)
        return 0;  // Not a prime
    for (int i = 2; i <= num / 2; i++) {
        if (num % i == 0)
            return 0;  // Not a prime
    }
    return 1;  // Prime number
}
int main()
 {
   int num = 17;
   if (isPrime(num))
       printf("%d is a prime number\n", num);
   else
       printf("%d is not a prime number\n", num);
   return 0;
}
```

TIME COMPLEXITY CALCULATIONS

- Program to check the given number is a prime number or not a prime number
- if a given number n is prime. In a brute-force approach, we iterate from 2 to n and check if n is divisible by any number in that range. If we find a divisor, then n is not prime. If we don't find any divisors, then n is prime.
- In this approach, we iterate n times, checking each number in the range. Therefore, the time complexity of a brute-force prime number check algorithm is approximately O(n).
- T(n) =O(n)

TYPES OF ALGORITHMS

8) Branch and Bound

- Branch and bound is a general algorithmic technique used for solving optimization problems, especially those with a large search space.
- It involves exploring the search space by systematically dividing it into smaller subproblems or branches and bounding the search based on certain criteria or constraints. By pruning unpromising branches, it can significantly reduce the number of possibilities to consider, improving efficiency.

EXAMPLE 1 : Knapsack Problem

```
#include <stdio.h>
#include <stdbool.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}
int knapsack(int values[], int weights[], int capacity, int n) {
    if (n == 0 || capacity == 0)
        return 0;

    if (weights[n - 1] > capacity)
        return knapsack(values, weights, capacity, n - 1);

    int included = values[n - 1] + knapsack(values, weights, capacity - weights[n - 1], n - 1);
    int excluded = knapsack(values, weights, capacity, n - 1);

    return max(included, excluded);
}
int main() {
    int values[] = {60, 100, 120};
    int weights[] = {10, 20, 30};
    int capacity = 50;
    int n = sizeof(values) / sizeof(values[0]);
    int maxProfit = knapsack(values, weights, capacity, n);
    printf("Maximum profit: %d\n", maxProfit);

    return 0;
}
```

EXAMPLE 2: SALESMAN PROBLEM

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

#define N 4

int graph[N][N] = {
    {0, 10, 15, 20},
    {10, 0, 35, 25},
    {15, 35, 0, 30},
    {20, 25, 30, 0}
};

int minCost = INT_MAX;
bool visited[N] = {false};

void tsp(int current, int cost, int count) {
    if (count == N && graph[current][0] > 0) {
        minCost = (cost + graph[current][0] < minCost) ? cost + graph[current][0] : minCost;
        return;
    }
```

```
    for (int i = 0; i < N; i++) {
        if (!visited[i] && graph[current][i] > 0) {
            visited[i] = true;
            tsp(i, cost + graph[current][i], count + 1);
            visited[i] = false;
        }
    }
}

int main() {
    visited[0] = true;
    tsp(0, 0, 1);

    printf("Minimum cost of TSP: %d\n", minCost);

    return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 1 :

- Knapsack Problem
- Time Complexity = Exponential, O(2^n)

Example 2 :

- Salesman problem
- Time Complexity= Exponential, O(N!)

TYPES OF ALGORITHMS

9) Randomized Approach

- This algorithm reduces running times and time-based complexities. It uses random elements as part of its logic.
- They use random numbers or randomization techniques to solve problems or improve the efficiency of algorithms. Randomized algorithms often provide approximate solutions or outputs that have a high probability of being correct, but they may not always be deterministic or guarantee the optimal solution.

EXAMPLE 1 : Rolling a dice to get random numbers

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int rollDice() {
 return rand() % 6 + 1;
}

int main() {
 srand(time(NULL));

    printf("Rolling the dice...\n");
    int diceResult = rollDice();
    printf("The dice rolled: %d\n", diceResult);

    return 0;
}
```

TIME COMPLEXITY CALCULATIONS

Example 1 :

- Rolling a dice to get random numbers

- Overall, the time complexity of simulating rolling a dice using randomization in C can be considered to be constant, assuming that the random number generation functions (rand() and srand()) have constant time complexities in the specific environment in which the code is executed.
- T(n)=O(1)

EXAMPLE 2 :

// Generating a random numbers with upper and lower limit using rand() function

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{
    int lower, upper, count,i,num;
    srand(time(0));
    printf("enter the upper limit");
    scanf("%d",&upper);
     printf("enter the lower limit");
    scanf("%d",&lower);
     printf("enter the count limit");
    scanf("%d",&count);
   FILE *fptr;
fptr=fopen("bhuvan.txt","w");
if(fptr==NULL)
{
printf("error");
   exit(1);
}
```

```
else
    {
        fprintf(fptr,"%d\n",count);
       for (i = 0; i < count; i++)
       {
           int num = (rand() %
        (upper - lower + 1)) + lower;
        printf("%d ", num);

        fprintf(fptr,"%d\n",num);
        }}
        fclose(fptr);
        return 0;
        }
```

TIME COMPLEXITY CALCULATIONS

- Generating a random numbers with upper and lower limits using rand() function

- The time complexity of generating n random numbers would be O(n). This is because you would need to call the random number generation function n times to generate all the numbers.
- T(n)=O(n)

Advantages of Algorithms:

- It is a step-wise representation of a solution to a given problem, which makes it easy to understand
- An algorithm uses a definite procedure.
- It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
- Every step in an algorithm has its own logical sequence so it is easy to debug.
- By using algorithm, the problem is broken down into smaller pieces or steps hence, it is easier for programmer to convert it into an actual program.

- **<u>Disdvantages of Algorithms:</u>**

- Alogorithms is Time consuming.
- Difficult to show Branching and Looping in Algorithms.
- Big tasks are difficult to put in Algorithms.

THANK YOU SO MUCH!