Lab Report

Course Code : CSE 232

Course Title : Algorithm Lab

Submitted To

Ms. Fatema Tuj Johora

Department of CSE,

Daffodil International University

Submitted By

Rakibul Hasan Akash

ID: 221-15-5688

Sec : 61_V

Department of CSE

Daffodil International University.

Date of submission : 23-11-2023

Contents

1. Linear Search:

Introduction:

Linear search, also known as sequential search, is a straightforward method for finding a target element within a list. It involves checking each element in the list sequentially until the desired element is found or the end of the list is reached.

Application:

Linear search is commonly used in scenarios where the data is unsorted, or the dataset is relatively small. It's suitable for simple applications where efficiency is not a critical concern.

Complexity:

- **Time Complexity:** O(n) - Linear search has a time complexity of O(n), where 'n' is the number of elements in the list. In the worst case, it may need to traverse the entire list.
- **Space Complexity:** O (1) - Linear search has a constant space complexity as it uses only a few variables, regardless of the input size.

Advantage:

- **Simplicity:** Linear search is easy to understand and implement.
- **Applicability:** It can be applied to both sorted and unsorted lists.
- **No Pre-sorting:** Unlike some other searching algorithms, linear search does not require the data to be sorted.

Disadvantage:

- **Inefficiency:** Linear search becomes inefficient for large datasets, as it checks each element one by one.
- **Limited Efficiency:** It is not suitable for scenarios where frequent or fast searches are required, especially in large datasets.

This C code provides a basic implementation of linear search with user input for testing. Users can enter the target element, and the program will output whether the element was found in the array and its index if applicable.

```
#include <stdio.h>

int linear_search(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;  // Return the index if the target is found
        }
```

```
    }
    return -1;  // Return -1 if the target is not found}
int main () {
    int myArray[] = {1, 4, 2, 7, 5, 9};
    int arraySize = sizeof(myArray) / sizeof(myArray[0]);

    int targetElement;
    printf("Enter the target element to search: ");
    scanf("%d", &targetElement);

    int result = linear_search(myArray, arraySize, targetElement);

    if (result != -1) {
        printf("Target %d found at index %d.\n", targetElement, result);
    } else {
        printf("Target %d not found in the list.\n", targetElement);
    }

    return 0;
}
```

Output:

Enter the target element to search: 5.

Target 5 found at index 4.

Enter the target element to search: 3.

Target 3 not found in the list.

Enter the target element to search: 9.

Target 9 found at index 5.

2. Binary Search:

Introduction:

Binary search is an efficient algorithm for finding a target element in a sorted array. It works by repeatedly dividing the search range in half, comparing the target value to the middle element, and narrowing down the search until the target is found or the search range becomes empty.

Application:

Binary search is particularly useful when working with large, sorted datasets, such as databases or arrays. It is more efficient than linear search for large datasets because it eliminates half of the remaining elements with each comparison.

Complexity:

- **Time Complexity:** O (log n) - Binary search has a logarithmic time complexity, where 'n' is the number of elements in the array. It efficiently reduces the search space with each comparison.
- **Space Complexity:** O (1) - Binary search has a constant space complexity as it uses only a few variables.

Advantage:

- **Efficiency:** Binary search is significantly faster than linear search, especially for large datasets.
- **Reduced Comparisons:** It reduces the number of comparisons needed to find the target element compared to linear search.

Disadvantage:

- **Sorted Data Requirement:** Binary search requires the data to be sorted. If the data is unsorted, a preprocessing step is needed.
- **Limited Applicability:** It may not be suitable for certain types of data structures, like linked lists, where random access is not efficient.

Implementation code in:

```
#include <stdio.h>

// Function to binary search

int binary_search(int arr[], int low, int high, int target) {

   while (low <= high) {
```

```
        int mid = low + (high - low) / 2;

        if (arr[mid] == target)
            return mid; // Target found
        if (arr[mid] < target)
            low = mid + 1; // the left half
        else
            high = mid - 1; //  the right half
    }
    return -1; // Target not found.
}
int main () {
    int myArray[] = {1, 2, 4, 5, 7, 9};
    int arraySize = sizeof(myArray) / sizeof(myArray[0]);
    int targetElement;
    printf("Enter the target element to search: ");
    scanf("%d", &targetElement);
    int result = binary_search(myArray, 0, arraySize - 1, targetElement);
    if (result != -1) {
        printf("Target %d found at index %d.\n", targetElement, result);
    } else {
        printf("Target %d not found in the list.\n", targetElement);
    }
    return 0.
}
```

Output:

Enter the target element to search: 5.

Target 5 found at index 3.

Enter the target element to search: 8.

Target 8 not found in the list.

3. Bubble Sort:

Introduction:

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Application:

Bubble sort is not the most efficient sorting algorithm, and it is typically used for educational purposes or when simplicity is more important than performance. It is rarely used in practical applications for large datasets due to its quadratic time complexity.

Complexity:

- Time Complexity: O(n^2) - Bubble sort has a quadratic time complexity, making it inefficient for large datasets.
- Space Complexity: O(1) - Bubble sort has a constant space complexity as it only uses a few variables.

Advantages:

- Simplicity: Bubble sort is easy to understand and implement.
- No Additional Memory: It doesn't require additional memory since it operates directly on the input array.

Disadvantages:

- Inefficiency: Bubble sort is not efficient for large datasets, as its time complexity is quadratic.
- Lack of Adaptability: It does not adapt well to partially sorted arrays; it performs the same number of swaps regardless of initial order.

Implementation:

```
#include <stdio.h>

void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int flag = 0;
        for (int j = 0; j < n - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
```

```
            // Swap elements
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
            flag = 1;
        }
    }
    if (flag == 0) {
        // If no swapping occurred in the inner loop, the array is already sorted
        break;
    }
  }

  // Print the sorted array directly in the sorting function
  printf("\sorted array is: \n");
  for (int i = 0; i < n; i++) {
    printf("%d\t", arr[i]);
  }
}

int main() {
  int size;
  printf("\nEnter the size of the array:");
  scanf("%d", &size);

  int arr[size];
  printf("\nEnter the elements of the array:");
```

```
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
    // Call bubble_sort function from main
    bubble_sort(arr, size);
    return 0;
}
```

Output:

Enter the size of the array: 5

Enter the elements of the array: 4 2 7 1 5

Sorted array is:

1 2 4 5 7

4. Insertion Sort:

Introduction:

Insertion sort is a simple sorting algorithm that builds the final sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, it performs well for small datasets or mostly sorted datasets.

Application:

Insertion sort is often used when the number of elements is small, or the dataset is partially sorted. It is considered efficient for small datasets due to its simplicity.

Complexity:

- Time Complexity: O(n^2) - In the worst case, when the input array is in reverse order. However, it can perform better than other quadratic algorithms, such as bubble sort or selection sort, in practice.
- Space Complexity: O(1) - Insertion sort has a constant space complexity as it only uses a few variables.

Advantages:

- Simple Implementation: Insertion sort is easy to understand and implement.
- Efficiency for Small Datasets: It can be more efficient than other quadratic sorting algorithms for small datasets.

Disadvantages:

- Inefficiency for Large Datasets: Insertion sort becomes inefficient for large datasets due to its quadratic time complexity.
- Not Suitable for Random Access Memory: It may not be suitable for linked lists or situations where random access is costly.

Implementation:

```
#include <stdio.h>
void insertion_sort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        // Move elements of arr[0..i-1] that are greater than key to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
void print_array(int arr[], int n) {
    printf("\nSorted array is: \n");
    for (int i = 0; i < n; i++) {
        printf("%d\t", arr[i]);
    }
}
```

```
int main() {
    int size;
    printf("\nEnter the size of the array:");
    scanf("%d", &size);
    int arr[size];
    printf("\nEnter the elements of the array:");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
    insertion_sort(arr, size);
    print_array(arr, size);
    return 0;
}
```

Output:

Enter the size of the array: 8

Enter the elements of the array: 9 3 6 1 8 2 7 5

Sorted array is:

1 2 3 5 6 7 8 9

5. Selection Sort:

Introduction:

Selection sort is a simple sorting algorithm that divides the input list into a sorted and an unsorted region. The algorithm repeatedly selects the smallest (or largest) element from the unsorted region and swaps it with the first element in the unsorted region.

Application:

Selection sort is often used for small datasets or in situations where the cost of swapping elements is less than the cost of comparisons. However, its quadratic time complexity makes it less efficient than more advanced algorithms for large datasets.

Complexity:

- Time Complexity: O(n^2) - Selection sort has a quadratic time complexity, making it inefficient for large datasets.
- Space Complexity: O(1) - Selection sort has a constant space complexity as it uses only a few variables.

Advantages:

- Simplicity: Selection sort is straightforward to understand and implement.
- In-Place Sorting: It performs sorting in-place, meaning it doesn't require additional memory.

Disadvantages:

- Inefficiency for Large Datasets: Selection sort becomes inefficient for large datasets due to its quadratic time complexity.
- Not Stable: Selection sort is not a stable sorting algorithm, meaning it may change the relative order of equal elements.

Code Example (Selection Sort in C):

```
#include<stdio.h>

int main(){


  int i, j, count, temp, number[25];


  printf("How many numbers u are going to enter?: ");

  scanf("%d",&count);

  printf("Enter %d elements: ", count);

  // Loop to get the elements stored in array

  for(i=0;i<count;i++)

    scanf("%d",&number[i]);

  // Logic of selection sort algorithm

  for(i=0;i<count;i++){

    for(j=i+1;j<count;j++){

      if(number[i]>number[j]){

        temp=number[i];
```

```
            number[i]=number[j];
            number[j]=temp;
        }
      }
    }

  printf("Sorted elements: ");
  for(i=0;i<count;i++)
     printf(" %d",number[i]);

  return 0;
}
```

Output:

How many numbers u are going to enter? 6

Enter 6 elements: 12 4 8 1 9 6

Sorted elements: 1 4 6 8 9 12

6. Quicksort:

Introduction:

Quicksort is a divide-and-conquer sorting algorithm that works by partitioning an array into two subarrays, then sorting the subarrays independently. The key step is the partitioning process, where an element is chosen as the "pivot," and the array is rearranged such that elements smaller than the pivot are on its left, and elements larger than the pivot are on its right. The process is then applied recursively to the subarrays.

Application:

Quicksort is widely used due to its efficiency for average and typical cases. It is often used in practice for large datasets and is the algorithm of choice for many programming languages' standard libraries for sorting.

Complexity:

- Time Complexity: O(n log n) - On average and best-case scenarios. However, in the worst-case scenario (if the pivot is always the smallest or largest element), it can degrade to O(n^2).
- Space Complexity: O(log n) - Quicksort typically has a logarithmic space complexity due to its recursive nature.

Advantages:

- Efficiency: Quicksort is very efficient on average and typical cases.
- In-Place Sorting: It sorts the array in-place, requiring only a small constant amount of additional memory.

Disadvantages:

- Worst-Case Scenario: The worst-case time complexity is O(n^2), although this can be mitigated by using randomized or carefully chosen pivots.
- Not Stable: Quicksort is not a stable sorting algorithm, meaning it may change the relative order of equal elements.

Implementation:

```
#include<stdio.h>

void quicksort(int number[25],int first,int last){

  int i, j, pivot, temp;


  if(first<last){

    pivot=first;

    i=first;

    j=last;


    while(i<j){

      while(number[i]<=number[pivot]&&i<last)

        i++;

      while(number[j]>number[pivot])

        j--;

      if(i<j){
```

```
            temp=number[i];
            number[i]=number[j];
            number[j]=temp;
          }
        }

        temp=number[pivot];
        number[pivot]=number[j];
        number[j]=temp;
        quicksort(number,first,j-1);
        quicksort(number,j+1,last);

    }
}

int main(){
    int i, count, number[25];

    printf("How many elements are u going to enter?: ");
    scanf("%d",&count);

    printf("Enter %d elements: ", count);
    for(i=0;i<count;i++)
        scanf("%d",&number[i]);

    quicksort(number,0,count-1);
    printf("Order of Sorted elements: ");
    for(i=0;i<count;i++)
```

```
    printf(" %d",number[i]);


  return 0;

}
```

Output:

How many elements are u going to enter?: 8

Enter 8 elements: 9 3 6 1 8 2 7 5

Order of Sorted elements: 1 2 3 5 6 7 8 9

7. Merge Sort :

Introduction:

Merge sort is a divide-and-conquer algorithm that divides an array into two halves, recursively sorts each half, and then merges the sorted halves. The merging step involves comparing elements from the two halves and combining them in sorted order. Merge sort is known for its stability and guarantees O(n log n) time complexity.

Application:

Merge sort is widely used for sorting large datasets and is often the algorithm of choice for external sorting where data is too large to fit into memory.

Complexity:

- **Time Complexity:** O(n log n) - Merge sort has a guaranteed time complexity of O(n log n) in all cases.
- **Space Complexity:** O(n) - Merge sort typically requires additional space for temporary storage during the merging process.

Advantages:

- **Stability:** Merge sort is a stable sorting algorithm, preserving the relative order of equal elements.
- **Predictable Performance:** It guarantees a consistent O(n log n) time complexity.

Disadvantages:

- **Space Complexity:** Merge sort requires additional space for merging, making it less memory-efficient in comparison to in-place sorting algorithms.

- **Not In-Place:** Merge sort is not an in-place sorting algorithm, which means it needs additional memory for the merging process.

Code Example (Merge Sort in C):

```
#include <stdio.h>

#define SIZE 100

int array1[SIZE];

int array2[SIZE];


void merge(int low, int mid, int high) {
   int i, j, k;


   for(i=low, j=mid+1, k=low; i<=mid && j<=high; k++) {
      if(array1[i]<array1[j]) {
         array2[k]=array1[i++];
      } else {
         array2[k] = array1[j++];
      }
   }
   while(i<=mid) {
      array2[k++]=array1[i++];
   }
   while(j<=high) {
      array2[k++]=array1[j++];
   }


   for (int l = 0; l <high+1 ; ++l) {
      array1[l]=array2[l];
   }
}
```

```
void sort(int low, int high) {
   if(low<high) {
      int mid= (low + high)/2;
      sort(low,mid);
      sort(mid+1, high);
      merge(low, mid, high);
   }
   else
      return;
}
int main(void) {
   int n;
   printf("\nEnter number of elements: ");
   scanf("%d", &n);
   printf("\nEnter %d elements: ", n);
   for(int i=0; i<n; i++) {
      scanf("%d", &array1[i]);
   }
   printf("\nArray after sorting is : ");

   sort(0, n-1);

   for(int i=0; i<n; i++) {
      printf("%d ", array1[i]);
   }
   return 0;
}
```

Output:

Enter number of elements: 8

Enter 8 elements: 9 3 6 1 8 2 7 5

Array after sorting is : 1 2 3 5 6 7 8 9

8. Greedy Algorithms:

Introduction:

An algorithm is designed to achieve optimum solution for a given problem. In the Greedy Algorithm approach, decisions are made from the given solution domain. Being greedy, the closest solution that seems to provide an optimum solution is chosen. Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Advantages of Greedy Algorithms:

1. **Simplicity:** Easy to understand and implement.
2. **Efficiency:** Often linear or close to linear time complexity.
3. **Local Optimum:** Suitable for problems where locally optimal choices lead to satisfactory overall solutions.
4. **Space Efficiency:** Low space complexity.

Disadvantages of Greedy Algorithms:

1. **No Backtracking:** Irreversible decisions, no undoing choices.
2. **No Guarantee of Optimality:** Doesn't always find the best solution.
3. **Dependency on Problem Structure:** Success depends on problem characteristics.
4. **Not Suitable for All Problems:** Limited applicability to complex or constraint-heavy problems.
5. **Limited Scope:** Effective for optimization, less so for intricate interactions.

9. Coin Change:

Introduction:

The Coin Change problem is a classic algorithmic problem that involves determining the number of ways to make change for a given amount of money using a given set of coin denominations. The goal is to find all possible combinations of coins that sum up to the given amount.

Application:

The Coin Change problem has various real-world applications, including:

1. Currency Systems**:** Designing efficient currency systems where the goal is to minimize the number of coins needed for any amount.
2. Vending Machines: Programming vending machines to give change in the optimal way.
3. Dynamic Programming: The Coin Change problem is often used as an introductory example in dynamic programming.

Complexity:

- Time Complexity**:** The time complexity depends on the approach used. The basic recursive approach has exponential time complexity, but dynamic programming solutions can achieve O(coins * amount) using memorization or tabulation.
- Space Complexity: The space complexity also depends on the approach. Recursive solutions may require a large number of function calls, resulting in a large call stack. Dynamic programming solutions can optimize space complexity using memorization or tabulation.

Advantage:

1. Versatility: The Coin Change problem serves as a fundamental problem in algorithmic thinking and dynamic programming, helping learners grasp key concepts.
2. Algorithmic Insight: Solving the Coin Change problem provides insight into the design of algorithms that involve optimal substructure and overlapping subproblems.

Disadvantage:

1. Exponential Time Complexity: The naive recursive approach has exponential time complexity, making it impractical for large inputs.
2. Optimization Challenges: Designing an optimal solution that minimizes both time and space complexity can be challenging, especially for beginners.
3.

Implement In C:

```
#include <stdio.h>
void calculateChange(int denominations[], int numDenominations, int remainingAmount) {
    int coinCount = 0;
    for (int i = 0; i < numDenominations; i++) {
        while (remainingAmount >= denominations[i]) {
            printf("%d ", denominations[i]);
            remainingAmount -= denominations[i];
            coinCount++;
        }
    }
    printf("\nMinimum coins needed: %d\n", coinCount);
}

int main() {
    int numDenominations;
    printf("Enter how many denominations: ");
    scanf("%d", &numDenominations);

    int denominations[numDenominations];
    printf("Enter the denominations in descending order: ");
    for (int i = 0; i < numDenominations; i++)
        scanf("%d", &denominations[i]);

    int targetAmount;
    printf("Enter target Amount: ");
    scanf("%d", &targetAmount);
```

```
    printf("Coins used for change: ");
    calculateChange(denominations, numDenominations, targetAmount);

    return 0;
}
```

Output:

Enter how many denominations: 4

Enter the denominations in descending order: 25 10 5 1

Enter target Amount: 63

Coins used for change: 25 25 10 1 1 1

Minimum coins needed: 6

10. Fractional Knapsack:

Introduction:

The Fractional Knapsack problem involves selecting items with specific weights and values to maximize the total value in a knapsack while respecting the knapsack's weight capacity. Unlike 0/1 Knapsack, here, fractions of items can be taken.

Application:

- Used in resource allocation problems where items can be divided into fractions.
- Can be applied to maximize profit in various scenarios, such as stock investments.

Complexity:

The time complexity of Fractional Knapsack is usually O(n log n), where "n" is the number of items. This complexity arises from sorting the items based on their value-to-weight ratios.

Advantage:

- Provides an optimal solution for maximizing value within the given weight capacity.
- A greedy approach allows for relatively simple and efficient implementations.

Disadvantage:

- Greedy algorithms might not always yield the globally optimal solution in certain scenarios.
- Fractional solutions may not be suitable for problems where items cannot be divided.

Implement with C :

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

struct Item {
    int profit, weight;
};

static int compare(struct Item a, struct Item b) {
    double r1 = (double)a.profit / (double)a.weight;
    double r2 = (double)b.profit / (double)b.weight;
    return r1 > r2;
}

double fractionalKnapsack(int W, struct Item arr[], int N) {
    qsort(arr, N, sizeof(struct Item), compare);

    double finalValue = 0.0;

    for (int i = 0; i < N; i++) {
        if (arr[i].weight <= W) {
            W -= arr[i].weight;
            finalValue += arr[i].profit;
        } else {
            finalValue += arr[i].profit * ((double)W / (double)arr[i].weight);
            break;
        }
    }
```

```
    return finalValue;
}

int main() {
    int knapsackSize, numItems;

    printf("\nEnter the size of the knapsack: ");
    scanf("%d", &numItems);

    struct Item itemList[numItems];

    printf("\nEnter the weight: ");
    for (int i = 0; i < numItems; i++) {
        scanf("%d", &itemList[i].weight);
    }

    printf("\nEnter the profit: ");
    for (int i = 0; i < numItems; i++) {
        scanf("%d", &itemList[i].profit);
    }

    printf("\nEnter the knapsack capacity: ");
    scanf("%d", &knapsackSize);

    printf("Maximum value in Knapsack = %lf\n", fractionalKnapsack(knapsackSize, itemList,
numItems));

    return 0;
```

}

11. 0-1 knapsack:

Introduction:

Fractional Knapsack is a classical optimization problem in which the goal is to select a fraction of items, each with a specified weight and value, to maximize the total value while respecting a weight constraint (knapsack capacity). Unlike 0/1 Knapsack, in Fractional Knapsack, items can be divided to maximize the overall value.

Application:

Fractional Knapsack has practical applications in various domains, such as:

- Resource Allocation: Selecting projects based on their return on investment.
- Inventory Management: Deciding the quantity of items to stock based on their profitability and weight.
- Finance: Portfolio optimization, where each item represents a financial asset with a return and risk.

Complexity:

The time complexity of the Fractional Knapsack algorithm is generally O(n log n), where 'n' is the number of items. This is often due to sorting the items based on their value-to-weight ratios.

Advantages:

- Flexibility: Fractional Knapsack allows for fractional parts of items to be included, providing more flexibility in choosing items.
- Greedy Approach: The fractional knapsack problem can be efficiently solved using a greedy algorithm, making it suitable for real-world applications.

Disadvantages:

- Greedy Heuristic: While the greedy approach works well for Fractional Knapsack, it might not guarantee an optimal solution for all types of knapsack problems.
- Limited to Continuous Items: Fractional Knapsack assumes that items can be divided, which may not be realistic for certain types of items.

Here's a simple example of solving Fractional Knapsack using a greedy algorithm in C:

```
#include <stdio.h>

#include <stdlib.h>
```

```
typedef struct {
    int weight;
    int value;
    double ratio; // value-to-weight ratio
} Item;

int compare(const void* a, const void* b) {
    return ((Item*)b)->ratio - ((Item*)a)->ratio;
}

double fractionalKnapsack(Item items[], int n, int capacity) {
    qsort(items, n, sizeof(Item), compare);

    double totalValue = 0.0;
    int currentWeight = 0;

    for (int i = 0; i < n; i++) {
        if (currentWeight + items[i].weight <= capacity) {
            currentWeight += items[i].weight;
            totalValue += items[i].value;
        } else {
            double remainingCapacity = capacity - currentWeight;
            totalValue += (remainingCapacity / items[i].weight) * items[i].value;
            break;
        }
    }

    return totalValue;
```

```
}

int main() {
    int n, capacity;

    printf("Enter the number of items: ");
    scanf("%d", &n);

    Item items[n];

    printf("Enter the weight and value of each item:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &items[i].weight, &items[i].value);
        items[i].ratio = (double)items[i].value / items[i].weight;
    }

    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);

    double maxValue = fractionalKnapsack(items, n, capacity);

    printf("Maximum value that can be obtained = %.2lf\n", maxValue);

    return 0;
}
```

Output:

Enter the number of items: 4

Enter the weight and value of each item:

2 10

5 25

1 7

3 15

Enter the capacity of the knapsack: 8

Maximum value that can be obtained = 38.50

12. Depth First Search:

Introduction:

Breadth First Search or BFS is a vertex-based algorithm used for finding the shortest path in a graph. The BFS technique traverses any graph in the breadth-ward motion. It then uses a queue for remembering that the next vertex must begin the search after reaching a dead end in any iteration. It basically selects one vertex at a time (by visiting and marking it). Then it visits the adjacent one to store it in the queue. Thus, BFS uses a Queue Data Structure that works on FIFO (First In, First Out).

Applications of Breadth First Search:

1. Breadth First Search (BFS) can find the shortest path and minimum spanning tree for unweighted graphs. In an unweighted graph, the shortest path has the least number of edges, and BFS always reaches a vertex from a source using the minimum number of edges.
2. GPS Navigation systems use BFS to find all neighboring locations.
3. Broadcasting in networks uses BFS to reach all nodes.

Time Complexity:

The time complexity of BFS is also O(V + E), where V is the number of nodes and E is the number of edges in the graph. This is because BFS visits each vertex once and examines all its edges, so the total number of operations is also proportional to the number of vertices plus the number of edges.

Outcome in BFS:

Breadth-First Search (BFS) is an algorithm for traversing or searching a graph in a level-wise manner, starting from a source vertex. The outcome of a BFS traversal is a list of visited vertices, explored in the order they were encountered. It is particularly useful for finding the shortest path in unweighted graphs and for exploring all vertices in a connected component. It ensures that vertices are visited in a breadth-first order, and it can help in tasks such as determining connectivity and distances in a graph.

Advantages of Breadth First Search (BFS):

- BFS is guaranteed to find the shortest path between the source and any other vertex in an unweighted graph. This makes it a useful algorithm in finding the shortest distance between two points, such as in navigation or routing problems.
- BFS is a complete algorithm, which means it will always find a solution if one exists.
- BFS is used in various applications such as social network analysis, web crawling, and image processing.
- BFS can handle disconnected graphs by exploring each connected component one by one.
- BFS can be easily parallelized, which means that it can take advantage of multiple processors to speed up the search.

Disadvantages of Breadth First Search (BFS):

- BFS requires more memory than DFS, as it needs to store all the nodes at each level.
- Although BFS has a time complexity of O(V+E), it can be slower than DFS in practice, especially when the graph is dense.
- BFS may not be suitable for large graphs as it can become slow and memory-intensive.
- BFS may not be the best algorithm for path finding in weighted graphs, as it may not always find the shortest path.

Breadth First Search Solved Examples

1.When is breadth first search optimal Solution?

Breadth First Search (BFS) is optimal when we want to find the shortest path between two nodes in an unweighted graph or a graph with the same weight on all edges.

BFS explores all the neighbors of a node before moving on to its neighbors' neighbors, and so on. When BFS finds the goal node, it has necessarily explored all the nodes that are at the same

or shorter distance from the starting node. This guarantees that the path found by BFS is always the shortest path in terms of the number of edges between the start and goal nodes.

Implementation Of BFS traversal.

To implement DFS traversal, you need to take the following stages.

Step 1: Create a stack with the total number of vertices in the graph as the size.

Step 2: Choose any vertex as the traversal's beginning point. Push a visit to that vertex and add it to the stack.

Step 3 - Push any non-visited adjacent vertices of a vertex at the top of the stack to the top of the stack.

Step 4 - Repeat steps 3 and 4 until there are no more vertices to visit from the vertex at the top of the stack.

Step 5 - If there are no new vertices to visit, go back and pop one from the stack using backtracking.

Step 6 - Continue using steps 3, 4, and 5 until the stack is empty.

Step 7 - When the stack is entirely unoccupied, create the final spanning tree by deleting the graph's unused edges.

BFS code:

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

int a[20][20], q[20] = {0},
     visited[20] = {0}, n, i, j, f = 0, r = -1;

void bfs(int v) {
   q[++r] = v;
   visited[v] = 1;
  while (f <= r) {
       v = q[f];
       printf("%d ", v);
       for (i = 0; i < n; i++) {
          if (a[v][i] && visited[i] == 0) {
             q[++r] = i;
             visited[i] = 1;
          }
       }
       f++;
   }
}
```

```
int main() {
   int v;
   printf("Enter the number of vertices: ");
   scanf("%d", &n);
   printf("Enter the graph data from the
            matrix:\n");

for (i = 0; i < n; i++) {
       for (j = 0; j < n; j++) {
          scanf("%d", &a[i][j]
       }
   }
   printf("Enter the Starting vertex: ");
   scanf("%d", &v);

   printf("The nodes which are reachable are: ");
   bfs(v);

   return 0;
}
```

Output:

Enter the number of vertices: 4

Enter the graph data from the matrix:

0 1 0 0

1 0 0 0

0 0 0 1

0 0 1 0

Enter the Starting vertex: 2

The nodes which are reachable are: 2 3

13. Swapping with a third variable:

```
#include <stdio.h>
void swap(int a, int b) {
   int temp;
   temp = a;
   a = b;
   b = temp;
   printf("\nAfter swapping: a = %d b = %d", a, b);
}

int main() {
   int a, b;
   printf("Enter the two numbers: ");
   scanf("%d %d", &a, &b);
   printf("\nBefore swapping: a = %d b = %d", a, b);
   swap(a, b);
   return 0;
}
```

14. Swapping without a third variable:

```
#include <stdio.h>
int main() {
   int a, b;
   printf("\nEnter two numbers: ");
   scanf("%d %d", &a, &b);
   printf("\nBefore swapping: a = %d b = %d", a, b);
   a = a + b;
   b = a - b;
```

```
    a = a - b;
    printf("\nAfter swapping: a = %d b = %d", a, b);
    return 0;
}
```

Swapping using pointers:

```
#include <stdio.h>
void swap(int *a, int *b) {
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int a, b;
    printf("Enter the two numbers: ");
    scanf("%d %d", &a, &b);
    printf("\nBefore swapping: a = %d b = %d", a, b);
    swap(&a, &b);
    printf("\nAfter swapping: a = %d b = %d", a, b);
    return 0;
}
```

15. Find the factorial of a number using recursion:

```
#include <stdio.h>
int factorial(int n) {
   if (n == 1) {
      return 1;
   } else {
      return n * factorial(n - 1);
   }
}
int main() {
   int n;
   printf("\nEnter the number: ");
   scanf("%d", &n);
   int result = factorial(n);
   printf("Factorial of %d is: %d", n, result);
   return 0;
}
```

16. Fibonacci series using recursion:

```
#include <stdio.h>
int fibonacci(int i) {
   if (i == 0) {
      return 0;
   } else if (i == 1) {
      return 1;
   } else {
      return fibonacci(i - 1) + fibonacci(i - 2);
   }
}
```

```
int main() {
    int n;
    printf("\nEnter the number: ");
    scanf("%d", &n);
    printf("\nSeries is: ");
    for (int i = 0; i < n; i++) {
        printf("%d\t", fibonacci(i));
    }
    return 0;
}
```