DESIGN AND ANALYSIS OF ALGORITHMS

PRACTICAL FILE

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF

BACHELOR OF TECHNOLOGY

(Information Technology**)**

Submitted by:

Name: Shivay Bhandari

Branch: (D3 IT-B2)

CRN: 1921142

URN: 1905398

Submitted to:

Dr. Manjot Kaur

Department of Information Technology,

Guru Nanak Dev Engineering College,

Ludhiana-141001

INDEX

EXPERIMENT NO. 1

1. Implement binary search algorithm and compute its time complexity.

1. Binary search is a searching algorithm which uses the Divide and Conquer technique to perform search on a sorted data. In each iteration or in each recursive call, the search gets reduced to half of the array. So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

C++ program to implement Binary Search :

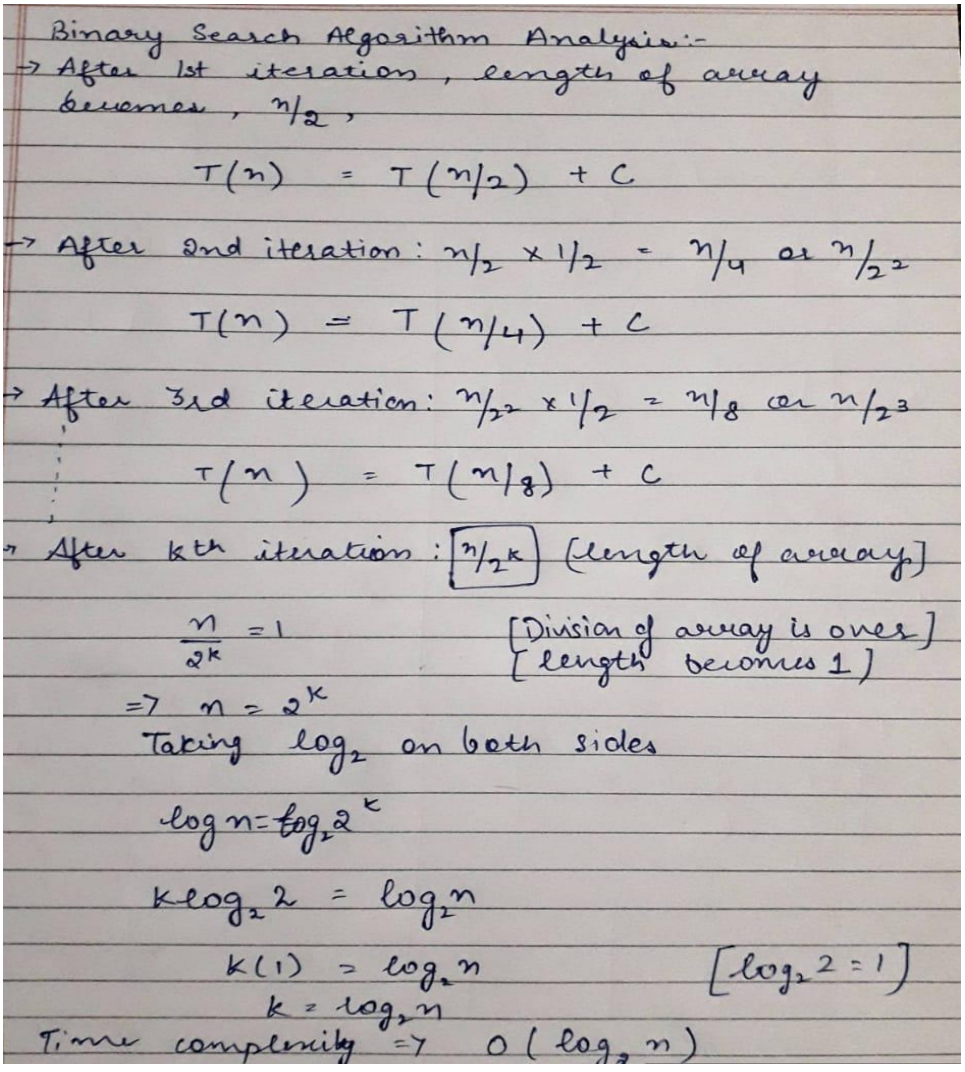
```
#include <bits/stdc++.h>

using namespace std;

int binarySearch(int arr[], int l, int r, int x)

{

        if (r >= l) {

                int mid = l + (r - l) / 2;

                if (arr[mid] == x)

                        return mid;

                if (arr[mid] > x)

                        return binarySearch(arr, l, mid - 1, x);

                return binarySearch(arr, mid + 1, r, x);

        }

        return -1;

}

int main(void)

{

        int arr[] = { 2, 3, 4, 10, 40 };

        int x = 10;

        int n = sizeof(arr) / sizeof(arr[0]);

        int result = binarySearch(arr, 0, n - 1, x);
```

```
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Output:

```
/tmp/1cOlAsKkGr.o/1921036
Element is present at index 3
```

Analysis:



EXPERIMENT 02

2. Implement merge sort algorithm and demonstrate divide and conquer technique.

2. Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

C++ program for Merge Sort:

```
#include <iostream>

using namespace std;

void merge(int array[], int const left, int const mid, int const right)

{

        auto const subArrayOne = mid - left + 1;

        auto const subArrayTwo = right - mid;

        auto *leftArray = new int[subArrayOne],

                *rightArray = new int[subArrayTwo];

        for (auto i = 0; i < subArrayOne; i++)

                leftArray[i] = array[left + i];

        for (auto j = 0; j < subArrayTwo; j++)

                rightArray[j] = array[mid + 1 + j];


        auto indexOfSubArrayOne = 0,

                indexOfSubArrayTwo = 0;

        int indexOfMergedArray = left;

        while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo <
subArrayTwo) {

                if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {
```

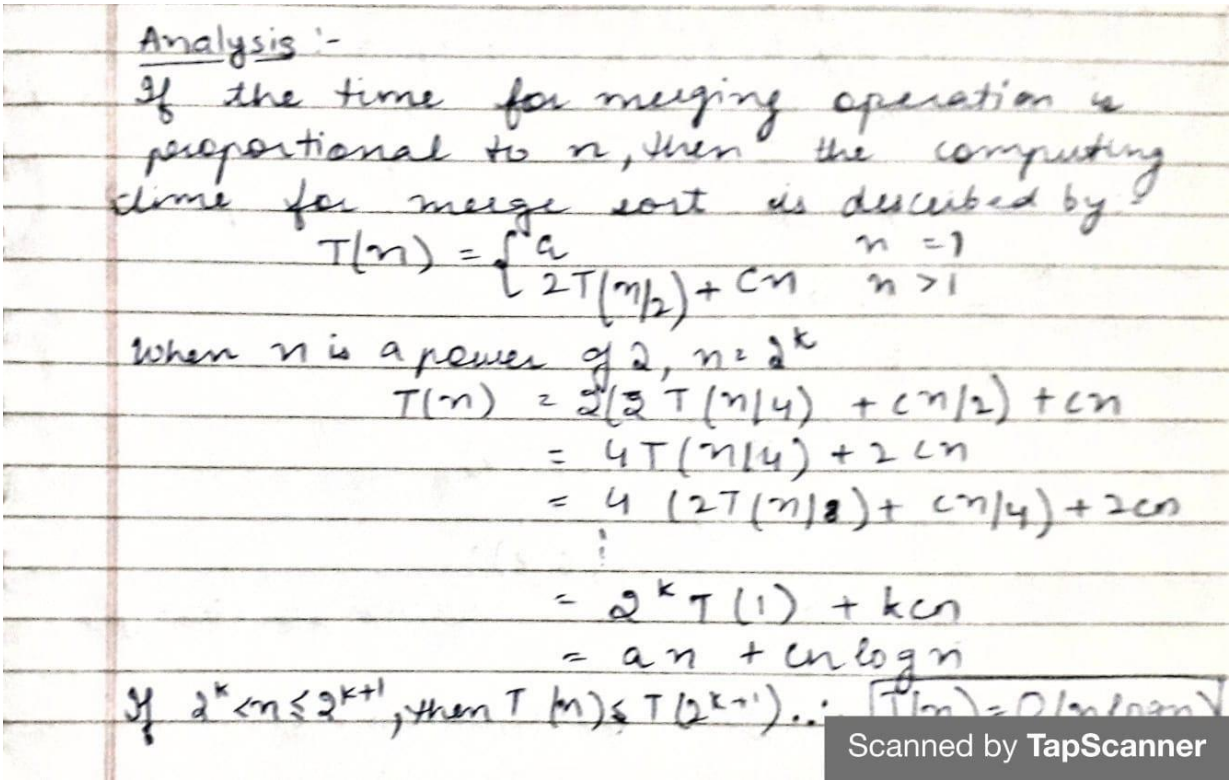
```
                array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
                indexOfSubArrayOne++;
            }
            else {
                array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
                indexOfSubArrayTwo++;
            }

            indexOfMergedArray++;
        }
        while (indexOfSubArrayOne < subArrayOne) {
            array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
            indexOfMergedArray++;
        }
        while (indexOfSubArrayTwo < subArrayTwo) {
            array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
            indexOfMergedArray++;
        }
}
void mergeSort(int array[], int const begin, int const end)
{
        if (begin >= end)
            return; // Returns recursively
```

```
        auto mid = begin + (end - begin) / 2;
        mergeSort(array, begin, mid);
        mergeSort(array, mid + 1, end);
        merge(array, begin, mid, end);
}
void printArray(int A[], int size)
{
        for (auto i = 0; i < size; i++)
                cout << A[i] << " ";
}
int main()
{
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        auto arr_size = sizeof(arr) / sizeof(arr[0]);

        cout << "Given array is \n";
        printArray(arr, arr_size);

        mergeSort(arr, 0, arr_size - 1);

        cout << "\nSorted array is \n";
        printArray(arr, arr_size);
        return 0;
}
```

OUTPUT:

```
/tmp/bRjjUqweut.o/1921036
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
```

Analysis:



Analysis :-

If the time for merging operation is proportional to n, then the computing time for merge sort is described by

$$T(n) = \begin{cases} a & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

When n is a power of 2, $n = 2^k$

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn\log n \end{aligned}$$

If $2^k < n \le 2^{k+1}$, then $T(n) \le T(2^{k+1})$ ∴ $T(n) = O(n \log n)$

EXPERIMENT 03

3. Analyze the time complexity of Quick-sort algorithm.

3. QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

C++ implementation of QuickSort :

```
#include <bits/stdc++.h>

using namespace std;

void swap(int* a, int* b)

{

	int t = *a;

	*a = *b;

	*b = t;

}

int partition (int arr[], int low, int high)

{

	int pivot = arr[high];

	int i = (low - 1);

	for (int j = low; j <= high - 1; j++)

	{

		if (arr[j] < pivot)

		{

			i++;

			swap(&arr[i], &arr[j]);
```

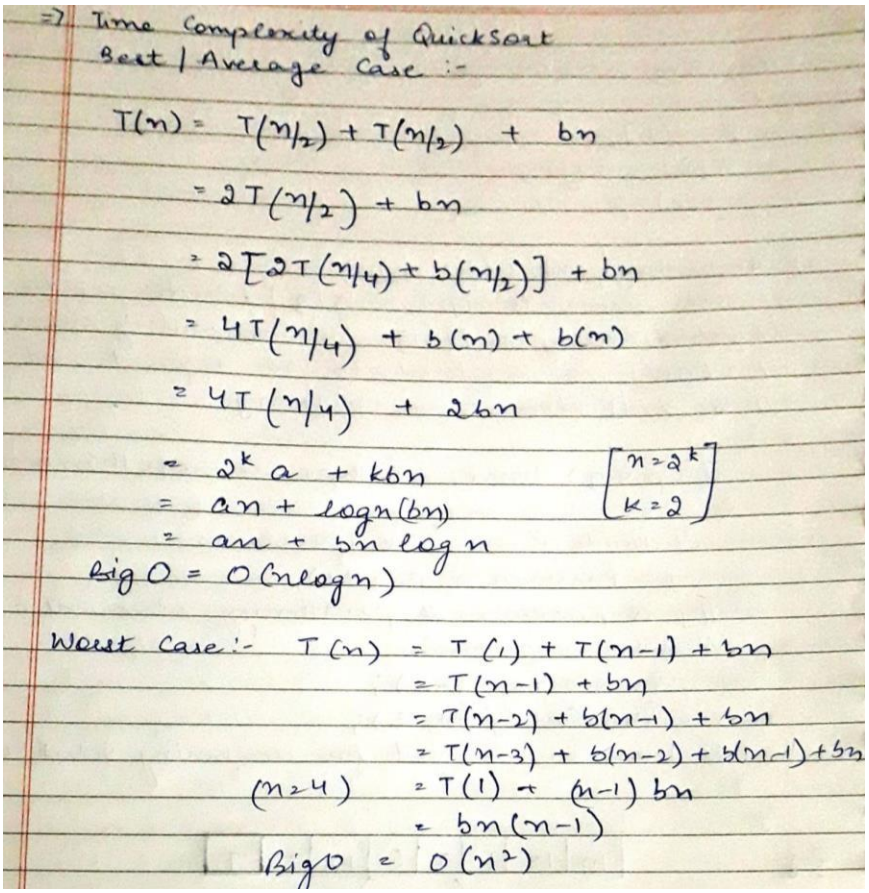
```
            }
        }
        swap(&arr[i + 1], &arr[high]);
        return (i + 1);
}
void quickSort(int arr[], int low, int high)
{
        if (low < high)
        {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
}
void printArray(int arr[], int size)
{
        int i;
        for (i = 0; i < size; i++)
            cout << arr[i] << " ";
        cout << endl;
}
int main()
{
```

```
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

OUTPUT:

```
/tmp/bRjjUqweut.o/1921036
Sorted array:
1 5 7 8 9 10
```

ANALYSIS:



EXPERIMENT 04

4. Solve minimum cost spanning tree problem using greedy method.

4. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph. There are two algorithms: Prim's and Kruskal's.

C++ Program to solve MST problem using Prim's algorithm:

```
#include <bits/stdc++.h>
using namespace std;
int minKey(int key[], bool mstSet[])
{
	// Initialize min value
	int min = INT_MAX, min_index;

	for (int v = 0; v < V; v++)
		if (mstSet[v] == false && key[v] < min)
			min = key[v], min_index = v;

	return min_index;
}
void printMST(int parent[], int graph[V][V])
{
	cout<<"Edge \tWeight\n";
	for (int i = 1; i < V; i++)
		cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
}
void primMST(int graph[V][V])
{
	int parent[V];

	int  key[V];

	bool mstSet[V];
	for (int i = 0; i < V; i++)
		key[i] = INT_MAX, mstSet[i] = false;
```

```
        key[0] = 0;
        parent[0] = -1;
        for (int count = 0; count < V - 1; count++)
        {
                int u = minKey(key, mstSet);
                mstSet[u] = true;
                for (int v = 0; v < V; v++)
                if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
        }
        printMST(parent, graph);
}
int main()
{
        int graph[V][V] = { { 0, 2, 0, 6, 0 },
                            { 2, 0, 3, 8, 5 },
                            { 0, 3, 0, 0, 7 },
                            { 6, 8, 0, 0, 9 },
                            { 0, 5, 7, 9, 0 } };
        primMST(graph);

        return 0;
}
```

Output:

```
/tmp/EfejomHCcP.o/1921036
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

Analysis:

- If adjacency list is used to represent the graph, then using breadth first search, all the vertices can be traversed in O(V + E) time.
- We traverse all the vertices of graph using breadth first search and use a min heap for storing the vertices not yet included in the MST.
- To get the minimum weight edge, we use min heap as a priority queue.
- Min heap operations like extracting minimum element and decreasing key value takes O(logV) time.

So, overall time complexity

= O(E + V) x O(logV)

= O((E + V)logV)

= O(ElogV)

This time complexity can be improved and reduced to O(E + VlogV) using Fibonacci heap.

C++ program for Kruskal's algorithm:

```
#include <bits/stdc++.h>
using namespace std;
class DSU {
	int* parent;
	int* rank;
public:
	DSU(int n)
	{
		parent = new int[n];
		rank = new int[n];
		for (int i = 0; i < n; i++) {
			parent[i] = -1;
			rank[i] = 1;
		}
	}
	int find(int i)
	{
		if (parent[i] == -1)
```

```
                return i;
        return parent[i] = find(parent[i]);
    }
    void unite(int x, int y)
    {
        int s1 = find(x);
        int s2 = find(y);
        if (s1 != s2) {
            if (rank[s1] < rank[s2]) {
                parent[s1] = s2;
                rank[s2] += rank[s1];
            }
            else {
                parent[s2] = s1;
                rank[s1] += rank[s2];
            }
        }
    }
};

class Graph {
    vector<vector<int> > edgelist;
    int V;
public:
    Graph(int V) { this->V = V; }
    void addEdge(int x, int y, int w)
```

```
    {
        edgelist.push_back({ w, x, y });
    }
    void kruskals_mst()
    {
        sort(edgelist.begin(), edgelist.end());
        DSU s(V);
        int ans = 0;
        cout << "Following are the edges in the "
                "constructed MST"
             << endl;
        for (auto edge : edgelist) {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];
            if (s.find(x) != s.find(y)) {
                s.unite(x, y);
                ans += w;
                cout << x << " -- " << y << " == " << w
                     << endl;
            }
        }
        cout << "Minimum Cost Spanning Tree: " << ans;
    }
};
```

```
int main()
{
        Graph g(4);
        g.addEdge(0, 1, 10);
        g.addEdge(1, 3, 15);
        g.addEdge(2, 3, 4);
        g.addEdge(2, 0, 6);
        g.addEdge(0, 3, 5);
        g.kruskals_mst();
        return 0;
}
```

Output:

```
/tmp/EfejomHCcP.o/1921036
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

Analysis:

Worst case time complexity of Kruskal's Algorithm= O(ElogV) or O(ElogE)

- The edges are maintained as min heap.
- The next edge can be obtained in O(logE) time if graph has E edges.
- Reconstruction of heap takes O(E) time.
- So, Kruskal's Algorithm takes O(ElogE) time.
- The value of E can be at most $O(V^2)$.
- So, O(logV) and O(logE) are same.

EXPERIMENT 05

5. Implement greedy algorithm to solve single-source shortest path problem.

5. **Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm)** is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. The algorithm creates a tree of shortest paths from the starting vertex, the source, to all other points in the graph.

A C++ program for Dijkstra's single source shortest path algorithm:

```
#include <limits.h>

#include <stdio.h>

#define V 9

int minDistance(int dist[], bool sptSet[])

{

        int min = INT_MAX, min_index;

        for (int v = 0; v < V; v++)

                if (sptSet[v] == false && dist[v] <= min)

                        min = dist[v], min_index = v;

        return min_index;

}

int printSolution(int dist[], int n)

{

        printf("Vertex Distance from Source\n");

        for (int i = 0; i < V; i++)

                printf("%d \t\t %d\n", i, dist[i]);

}

void dijkstra(int graph[V][V], int src)

{
```

```
        int dist[V];
        bool sptSet[V];
        for (int i = 0; i < V; i++)
                dist[i] = INT_MAX, sptSet[i] = false;
        dist[src] = 0;
        for (int count = 0; count < V - 1; count++) {
                int u = minDistance(dist, sptSet);
                sptSet[u] = true;
                for (int v = 0; v < V; v++)
                        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                                && dist[u] + graph[u][v] < dist[v])
                                dist[v] = dist[u] + graph[u][v];
        }
        printSolution(dist, V);
}
int main()
{
        int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                                { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                                {0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                                { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                                { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                                { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                                { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
```

```
                { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;

}
```

Output:

```
/tmp/EUvjAu7LBL.o/1921036
Vertex Distance from Source
0       0
1       4
2       12
3       19
4       21
5       11
6       9
7       8
8       14
```

Analysis:

Case-01:

This case is valid when-

- The given graph G is represented as an adjacency matrix.
- Priority queue Q is represented as an unordered list.

Here,

- A[i,j] stores the information about edge (i,j).
- Time taken for selecting i with the smallest dist is O(V).
- For each neighbor of i, time taken for updating dist[j] is O(1) and there will be maximum V neighbors.
- Time taken for each iteration of the loop is O(V) and one vertex is deleted from Q.
- Thus, total time complexity becomes $O(V^2)$.

Case-02:

This case is valid when-

- The given graph G is represented as an adjacency list.
- Priority queue Q is represented as a binary heap.

Here,

- With adjacency list representation, all vertices of the graph can be traversed using BFS in O(V+E) time.
- In min heap, operations like extract-min and decrease-key value takes O(logV) time.
- So, overall time complexity becomes O(E+V) x O(logV) which is O((E + V) x logV) = O(ElogV)
- This time complexity can be reduced to O(E+VlogV) using Fibonacci heap.

EXPERIMENT 06

6. Use dynamic programming to solve Knapsack problem.

6. In this Knapsack algorithm type, each package can be taken or not taken. Besides, the thief cannot take a fractional amount of a taken package or take a package more than once. This type can be solved by Dynamic Programming Approach.

C++ program to solve knapsack problem using dynamic programming:

```
#include  <iostream>

using namespace std;

int max(int x, int y) {

  return (x > y) ? x : y;

}

int knapSack(int W, int w[], int v[], int n) {

  int i, wt;

  int K[n + 1][W + 1];

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

    for (wt = 0; wt <= W; wt++) {

      if (i == 0 || wt == 0)

      K[i][wt] = 0;

      else if (w[i - 1] <= wt)

        K[i][wt] = max(v[i - 1] + K[i - 1][wt - w[i - 1]], K[i - 1][wt]);

      else

     K[i][wt] = K[i - 1][wt];

    }

  }

  return K[n][W];

}
```

```
int main() {
   cout << "Enter the number of items in a Knapsack:";
   int n, W;
   cin >> n;
   int v[n], w[n];
   for (int i = 0; i < n; i++) {
      cout << "Enter value and weight for item " << i << ":";
      cin >> v[i];
      cin >> w[i];
   }
   cout << "Enter the capacity of knapsack";
   cin >> W;
   cout << knapSack(W, w, v, n);
   return 0;
}
```

Output:

```
/tmp/ytNpjdXh9M.o/1921036
Enter the number of items in a Knapsack:4
Enter value and weight for item 0:10
5
Enter value and weight for item 1:20
60
Enter value and weight for item 2:30
70
Enter value and weight for item 3:40
80
Enter the capacity of knapsack100
50
```

Analysis:

- Each entry of the table requires constant time O(1) for its computation.
- It takes O(nw) time to fill (n+1)(w+1) table entries.

- It takes O(n) time for tracing the solution since tracing process traces the n rows.
- Thus, overall O(nw) time is taken to solve 0/1 knapsack problem using dynamic programming.

EXPERIMENT 07

7. Solve all pairs shortest paths problem using dynamic programming. (Floyd-Warshall Algorithm)

7. The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

C++ Program for Floyd Warshall Algorithm:

```
#include <bits/stdc++.h>

using namespace std;

#define V 4

void floydWarshall(int graph[][V])

{

	int dist[V][V], i, j, k;

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

		for (j = 0; j < V; j++)

			dist[i][j] = graph[i][j];

	for (k = 0; k < V; k++) {

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

			for (j = 0; j < V; j++) {

				if (dist[i][j] > (dist[i][k] + dist[k][j])

					&& (dist[k][j] != INF

						&& dist[i][k] != INF))

					dist[i][j] = dist[i][k] + dist[k][j];

			}

		}

	}
```

```
        printSolution(dist);
}
void printSolution(int dist[][V])
{
        cout << "The following matrix shows the shortest "
                        "distances"
                        " between every pair of vertices \n";
        for (int i = 0; i < V; i++) {
                for (int j = 0; j < V; j++) {
                        if (dist[i][j] == INF)
                                cout << "INF"
                                        << "      ";
                        else
                                cout << dist[i][j] << " ";
                }

                cout << endl;
        }
}
int main()
{
        int graph[V][V] = { { 0, 5, INF, 10 },
                                                { INF, 0, 3, INF },
                                                { INF, INF, 0, 1 },
                                                { INF, INF, INF, 0 } };
        floydWarshall(graph);
```

```
    return 0;
}
```

Output:

```
/tmp/ytNpjdXh9M.o/1921036
The following matrix shows the shortest distances between every pair of
    vertices
0    5   8   9
INF  0   3   4
INF  INF     0   1
INF  INF     INF     0
```

Analysis:

- Floyd Warshall Algorithm consists of three loops over all the nodes.
- The inner most loop consists of only constant complexity operations.
- Hence, the asymptotic complexity of Floyd Warshall algorithm is $O(n^3)$.
- Here, n is the number of nodes in the given graph.

EXPERIMENT 08

8. Use backtracking to solve 8-queens' problem.

8. The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due toclashes then we backtrack and return false.

C++ program to solve 8 Queen Problem using backtracking:

```
#include<iostream>
using namespace std;
int grid[10][10];
void print(int n) {
for (int i = 0;i <= n-1; i++) {
for (int j = 0;j <= n-1; j++) {
cout <<grid[i][j]<< " ";
}
cout<<endl;
}
cout<<endl;
cout<<endl;
}
bool isSafe(int col, int row, int n) {
for (int i = 0; i < row; i++) {
if (grid[i][col]) {
return false;
}
}
for (int i = row,j = col;i >= 0 && j >= 0; i--,j--) {
if (grid[i][j]) {
return false;
```

```
}
}
for (int i = row, j = col; i >= 0 && j < n; j++, i--) {
if (grid[i][j]) {
return false;
}
}
return true;
}
bool solve (int n, int row) {
if (n == row) {
print(n);
return true;
}
bool res = false;
for (int i = 0;i <=n-1;i++) {
if (isSafe(i, row, n)) {
grid[row][i] = 1;
res = solve(n, row+1) || res;//if res ==false then backtracking will occur
//by assigning the grid[row][i] = 0
grid[row][i] = 0;
}
}
return res;
}
int main()
{
ios_base::sync_with_stdio(false);
cin.tie(NULL);
int n;
cout<<"Enter the number of queen"<<endl;
```

```
cin >> n;
for (int i = 0;i < n;i++) {
for (int j = 0;j < n;j++) {
grid[i][j] = 0;
}
}
bool res = solve(n, 0);
if(res == false) {
cout << -1 << endl;
} else {
cout << endl;
}
return 0;
}
```

OUTPUT:

Analysis:

- The analysis of the code is a little bit tricky. The for loop in the N-QUEEN function is running from 1 to N (N, not n. N is fixed and n is the size of the problem i.e., the number of queens left) but the recursive call of N-QUEEN(row+1, n-1, N, board) (T(n−1)T(n−1)) is not going to run N times because it will run only for the safe cells.
- Since we have started by filling up the rows, so there won't be more than n (number of queens left) safe cells in the row in any case.
- So, this part is going to take n∗T(n−1)n∗T(n−1) time.
- One can think that the term O(N2)(O((n−2)!))O(N2)(O((n−2)!)) will dominate if N is large enough but this is not going to happen.
- Think about placing 1 queen on a 4x4 chessboard. Even if the size of the board (N) is quite greater than the number of queen (n), the algorithm will just find a place for the queen and then terminate (if n==0 → return TRUE).
- So it is not going to depend on N and thus, the running time will be O(n!)O(n!).

EXPERIMENT 09

9. Solve sum of subsets problem using backtracking.

9. Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

C++ Program:

```
#include <bits/stdc++.h>

using namespace std;

 #define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))

static int total_nodes;

 // prints subset found

void printSubset(int A[], int size)

{

    for(int i = 0; i < size; i++)

    {

        cout<<" "<< A[i];

    }

    cout<<"\n";

}

 // qsort compare function

int comparator(const void *pLhs, const void *pRhs)

{

    int *lhs = (int *)pLhs;

    int *rhs = (int *)pRhs;

    return *lhs > *rhs;

}
```

```
void subset_sum(int s[], int t[],
                int s_size, int t_size,
                int sum, int ite,
                int const target_sum)
{
   total_nodes++;
    if( target_sum == sum )
   {
      // We found sum
      printSubset(t, t_size);
       // constraint check
      if( ite + 1 < s_size && sum - s[ite] + s[ite + 1] <= target_sum )
      {

         // Exclude previous added item and consider next candidate
         subset_sum(s, t, s_size, t_size - 1, sum - s[ite], ite + 1, target_sum);
      }
      return;
   }
   else
   {
      // constraint check
      if( ite < s_size && sum + s[ite] <= target_sum )
      {
         // generate nodes along the breadth
```

```
        for( int i = ite; i < s_size; i++ )
        {
            t[t_size] = s[i];
            if( sum + s[i] <= target_sum )
            {
                // consider next level node (along depth)
                subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
            }
        }
    }
}
// Wrapper that prints subsets that sum to target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));
    int total = 0;
    // sort the set
    qsort(s, size, sizeof(int), &comparator);
    for( int i = 0; i < size; i++ )
    {
        total += s[i];
    }
    if( s[0] <= target_sum && total >= target_sum )
    {
```

```
        subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);
    }
    free(tuplet_vector);
}
int main()
{
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
    int target = 53;
    int size = ARRAYSIZE(weights);
    generateSubsets(weights, size, target);
    cout << "Nodes generated " << total_nodes;
    return 0;
}
```

Output:

```
/tmp/yLTPrhAKWC.o/1905334
8 9 14 22
 8 14 15 16
 15 16 22
Nodes generated 68
```

Analysis:

- It is intuitive to derive the complexity of sum of the subset problem. In the state-space tree, at level i, the tree has 2i nodes. So, given n items, the total number of nodes in the tree would be $1 + 2 + 2^2 + 2^3 + .. 2^n$.

- $T(n) = 1 + 2 + 2^2 + 2^3 + .. 2^n = 2^{n+1} – 1 = O(2^n)$

Thus, sum of sub set problem runs in exponential order.

- Worst case time complexity: $O(2^n)$
- Best case time complexity: O(1)

EXPERIMENT 10

10. Implement Boyer-Moore Algorithm.

10. The Boyer Moore algorithm is **a searching algorithm in which a string of length n and a pattern of length m is searched**. It prints all the occurrences of the pattern in the Text. Like the other string matching algorithms, this algorithm also preprocesses the pattern.

C++ program for Boyer Moore Algorithm:

```
# include <string.h>
# include <stdio.h>
 # define NO_OF_CHARS 256
 // A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b) ? a : b;
}
 // The preprocessing function for Boyer Moore's bad character heuristic
void badCharHeuristic(char *str, int size, int badchar[NO_OF_CHARS])
{
    int i;
     // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;
     // Fill the actual value of last occurrence of a character
    for (i = 0; i < size; i++)
        badchar[(int) str[i]] = i;
}
 void search(char *txt, char *pat)
{
    int m = strlen(pat);
```

```
    int n = strlen(txt);
     int badchar[NO_OF_CHARS];
     badCharHeuristic(pat, m, badchar);
     int s = 0; // s is shift of the pattern with respect to text
    while (s <= (n - m))
    {
        int j = m - 1;
         while (j >= 0 && pat[j] == txt[s + j])
            j--;
         if (j < 0)
        {
            printf("\n pattern occurs at shift = %d", s);
             s += (s + m < n) ? m - badchar[txt[s + m]] : 1;
         }
         else
            s += max(1, j - badchar[txt[s + j]]);
    }
}
 /* Driver program to test above funtion */
int main()
{
    char txt[] = "ABAAABCD";
    char pat[] = "ABC";
    search(txt, pat);
    return 0;
}
```

Output:

```
/tmp/HaTHZn4Z9L.o/1905334
pattern occurs at shift = 4
```

Analysis:

- Pre-processing phase in O (m + σ) time and space complexity
- Searching phase in O (mn) time complexity
- 3n text character comparisons in the worst case

O(n / m) best performance