University of Science and Technology Chittagong (USTC)
Faculty of Science, Engineering & Technology
Department of Computer Science & Engineering

Lab Report

Course code : CSE 222

Course Title : Algorithms Lab

Submitted by :

Tanmay Das
ID : 0022210005101005
Batch : 38th
Semester: 4th
Department : CSE, FSET

Submitted to:

Most Tahamina Khatoon
[Assistant Professor]
CSE,FSET, USTC

List of Experiments

Exp. No	Experiment Name	Page no
01	Write a program to implement Binary Search.	03
02	Write a program to implement Quick Sort.	06
03	Write a program to implement Merge Sort.	09
04	Write a program to implement Selection Sort.	13
05	Write a program to implement Knapsack Problem.	16
06	Write a program to implement Dynamic Programming From a given vertex in a weighted connected graph, find shortest path to other vertices using Dijkstra’s Algorithm.	19
07	Write a program to implement Graph traversal using Breadth First Search technique.	23
08	Write a program to implement Graph traversal using Depth First Search technique.	26
09	Write a program to implement Check whether a given vertex is connected or not using Depth First Search (DFS) method.	29
10	Write a program to implement Sum of Subsets.	32
11	Write a program to implement Traveling Salesman Problem.	34
12	Write a program to implement Parallel Merge Sort.	37

Experiment Number: 01

Experiment Name: Write a program to implement Binary Search.

Objective: We will explore the concept of binary search by developing pseudocode and implementing it in C++. This process will help us understand how to efficiently find items in sorted arrays, appreciate its significance in data processing, and effectively apply it to solve practical problems.

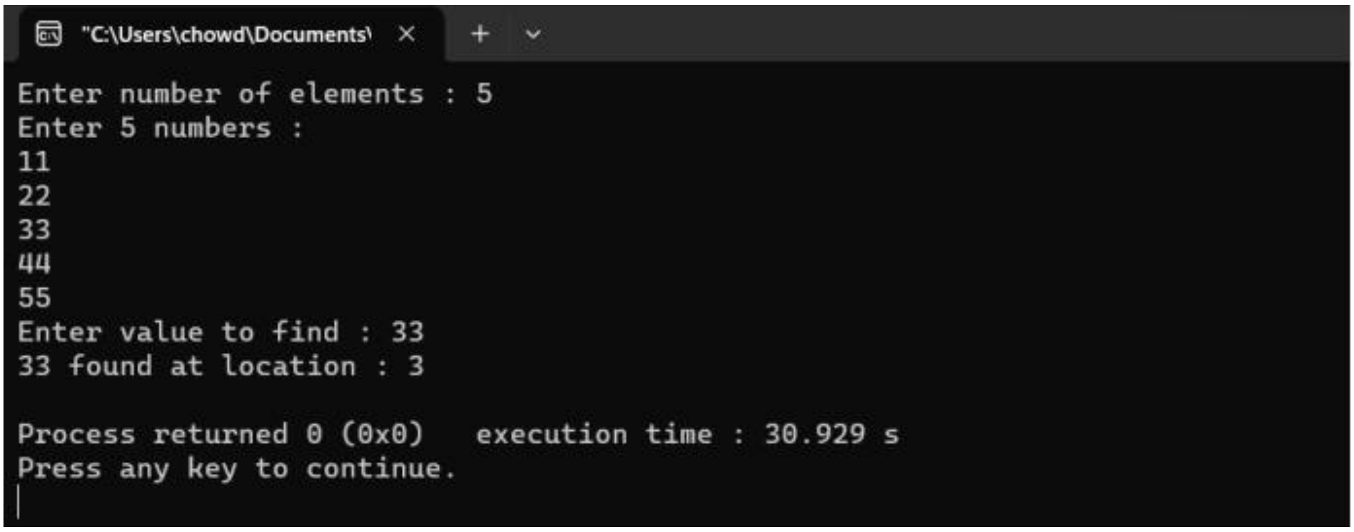
Pseudocode:

```
1. Begin.
2. Input n, a[100].
3. Initialize left=0,right=n-1 and mid = (left+right)/2.
4. Use while (left <= right).
5. Then again use if(a[mid] < item)
   left = mid + 1 and again find new mid = (left + right)/2 and go to step 4 .
6. else if (a[mid] == item)
       print mid and use break.
7. else
   right = mid – 1 and new mid = (left + right)/2 and again go to step 4.
8. if (left > right)
  print Not found! item is not present in the list.
9. End.
```

Implementation in Cpp:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int i, first, last, middle, n, search, array[100];
  printf("Enter number of elements : ");
  scanf("%d", &n);
  printf("Enter %d numbers : \n", n);
  for (i = 0; i < n; i++)
    scanf("%d", &array[i]);
  printf("Enter value to find : ");
  scanf("%d", &search);
  first = 0;
  last = n - 1;
  middle = (first+last)/2;
  while (first <= last) {
    if (array[middle] < search)
      first = middle + 1;
    else if (array[middle] == search) {
      printf("%d found at location : %d \n ", search, middle+1);
      break;
    }
    else
      last = middle - 1;
    middle = (first + last)/2;
  }
  if (first > last)
    printf("Not found! %d isn't present in the list.\n", search);
  return 0;
}
```

Output:



```
Enter number of elements : 5
Enter 5 numbers :
11
22
33
44
55
Enter value to find : 33
33 found at location : 3

Process returned 0 (0x0)   execution time : 30.929 s
Press any key to continue.
```

Fig.01- Output of the Binary Search Implementation.

Discussion: Binary search is a widely-used algorithm for efficiently finding a target element in a sorted array. Here's a concise overview of the code:

1. **Initialization:**
    - Set **left** to 0 and **right** to **length(arr) - 1** to establish the initial search range.
2. **While Loop:**
    - Utilize a while loop to iteratively narrow down the search range until the target is found or determined to be absent.
3. **Midpoint Calculation:**
    - Calculate the midpoint **mid** as **(left + right) / 2** within the loop.
4. **Comparison:**
    - Compare **arr[mid]** with the target:
        - If equal, return **mid** as the target is found.
        - If less, update **left** to **mid + 1** to search the right half.
        - If greater, update **right** to **mid - 1** to search the left half.
5. **Repeat or Terminate:**
    - Continue the loop while **left** is less than or equal to **right**.
    - If the loop exits without finding the target, return -1.

Conclusion: Binary search is a powerful method for finding elements in sorted arrays. Our examination through pseudocode and a C++ implementation has provided a solid understanding of its mechanics. With a divide-and-conquer strategy and O(log n) time complexity, binary search is a efficient tool for data retrieval. This knowledge equips us to confidently employ binary search in various programming tasks, improving our problem-solving capabilities.

Experiment Number: 02

Experiment Name: Write a program to implement Quick Sort.

Objective: Introducing Quick Sort through pseudocode and a C++ implementation aims to elucidate a widely-used sorting algorithm, emphasizing its core principles and providing a practical code illustration. Quick Sort proves to be an effective sorting technique by strategically dividing an array into smaller subarrays and recursively sorting them.

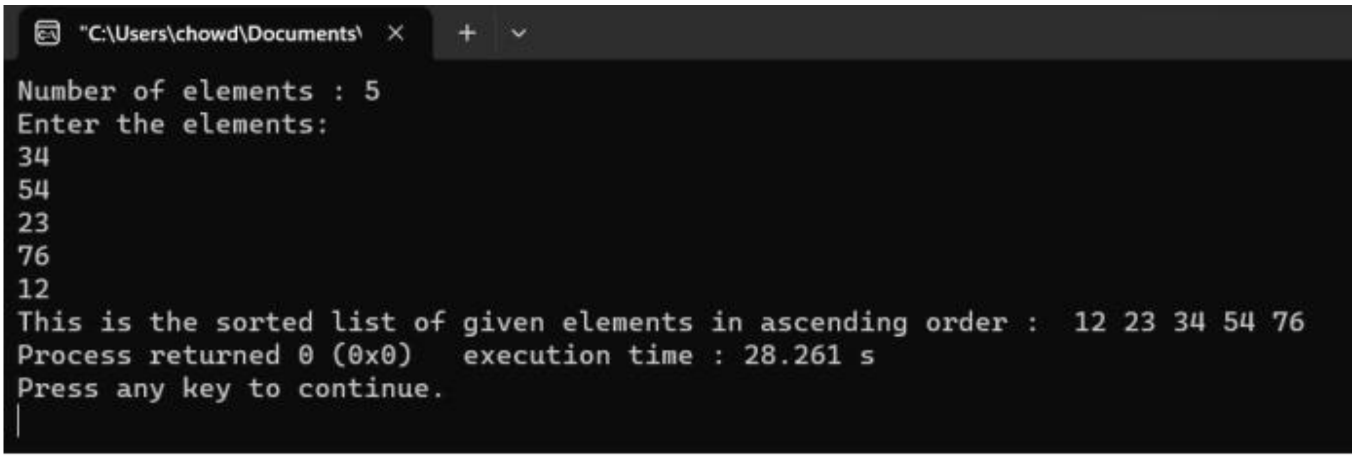
Pseudocode:

```
1. Procedure QuickSort(number[100], first, last)
2.     If first < last:
3.         pivot <- first
4.         i <- first
5.         j <- last
6.         while i < j:
7.             while number[i] <= number[pivot] and i < last:
8.                 i <- i + 1
9.             while number[j] > number[pivot]:
10.                j <- j - 1
11.            if i < j:
12.                Swap number[i] and number[j]
13.        Swap number[pivot] and number[j]
14.        Call QuickSort(number, first, j - 1)
15.        Call QuickSort(number, j + 1, last)
16. Procedure Main()
17.    Declare i, count, number[25]
18.    Output "Number of elements: "
19.    Input count
20.    Output "Enter the elements: "
21.    for i from 0 to count - 1:
22.        Input number[i]
23.    Call QuickSort(number, 0, count - 1)
24.    Output "This is the sorted list of given elements in ascending order: "
25.    for i from 0 to count - 1:
26.        Output number[i]
27. End Procedure
```

Implementation in Cpp:

```
#include <stdio.h>
#include <stdlib.h>
void quicksort(int number[100],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("Number of elements : ");
  scanf("%d",&count);
  printf("Enter the elements: \n", count);
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);
  quicksort(number,0,count-1);
  printf("This is the sorted list of given elements in ascending order : ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);
  return 0;
}
```

Output:



```
"C:\Users\chowd\Documents\
Number of elements : 5
Enter the elements:
34
54
23
76
12
This is the sorted list of given elements in ascending order :  12 23 34 54 76
Process returned 0 (0x0)   execution time : 28.261 s
Press any key to continue.
```

Fig.02- Output of the Quick Sort Implementation.

Discussion: Quick Sort, a widely-used sorting algorithm with an average-case time complexity of O(n log n), divides an array based on a chosen pivot element. One subarray holds elements smaller than the pivot, and the other holds larger elements, with recursion applied to both. The QuickSort method, detailed in the provided pseudocode and C++ implementation, recursively sorts the array by selecting a pivot element through the Partition function. This arrangement positions smaller elements on the left and larger elements on the right. While Quick Sort is efficient for sorting large datasets, it's worth noting that continuous imbalanced partitions can lead to a worst-case time complexity of O(n^2). To address this, pivot selection techniques, like using the median of three random components, can be employed to improve performance.

Conclusion: Quick Sort is a potent sorting algorithm known for its efficiency, boasting an average time complexity of O(n log n). Widely utilized for its speed and adaptability, Quick Sort efficiently handles large datasets by selecting a pivot and dividing the array into smaller subarrays. Despite its worst-case time complexity of O(n^2), employing improved pivot selection strategies mitigates this limitation. A fundamental understanding and implementation of Quick Sort are essential for those involved in algorithm design and sorting algorithms.

Experiment Number: 03

Experiment Name: Write a program to implement Merge Sort.

Objective: Merge Sort is a sorting technique that divides an array into smaller segments, recursively sorts these segments, and then merges them to create a sorted array. Its reliability and consistent performance make Merge Sort suitable for sorting large datasets and various data types.

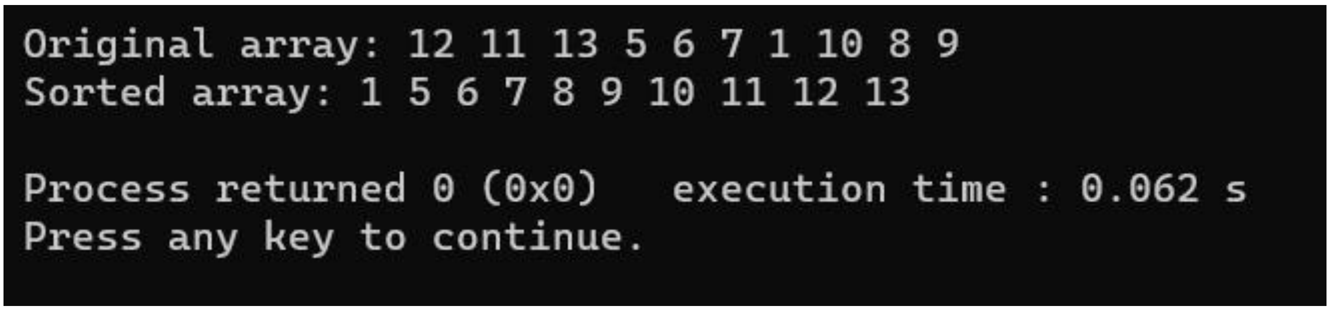
Pseudocode:

```
procedure merge(arr, left, mid, right):
   n1 = mid - left + 1
   n2 = right - mid
   L = new array of size n1
   R = new array of size n2
   for i = 0 to n1 - 1:
      L[i] = arr[left + i]
   for j = 0 to n2 - 1:
      R[j] = arr[mid + 1 + j]
   i = 0
   j = 0
   k = left
   while i < n1 and j < n2:
      if L[i] <= R[j]:
         arr[k] = L[i]
         i = i + 1
      else:
         arr[k] = R[j]
         j = j + 1
      k = k + 1
   while i < n1:
      arr[k] = L[i]
      i = i + 1
      k = k + 1
   while j < n2:
      arr[k] = R[j]
      j = j + 1
      k = k + 1
procedure mergeSort(arr, left, right):
   if left < right:
        mid = left + (right - left) / 2
        mergeSort(arr, left, mid)
        mergeSort(arr, mid + 1, right)
        merge(arr, left, mid, right)
procedure main():
    arr = [12, 11, 13, 5, 6, 7, 1, 10, 8, 9]
    print "Original array: "
    for num in arr:
        print num, " "
    mergeSort(arr, 0, length(arr) - 1)
    print "Sorted array: "
    for num in arr:
        print num, " "
```

Implementation in Cpp:

```
#include <iostream>
#include <vector>
using namespace std;
void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
int main() {
    vector<int> arr = {12, 11, 13, 5, 6, 7, 1, 10, 8, 9};
    cout << "Original array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
    mergeSort(arr, 0, arr.size() - 1);
    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```

Output:



```
Original array: 12 11 13 5 6 7 1 10 8 9
Sorted array: 1 5 6 7 8 9 10 11 12 13

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

Fig.03- Output of the Merge Sort Implementation.

Discussion: With a time complexity of O(n log n), Merge Sort is a popular divide-and-conquer sorting algorithm effective for handling large datasets. It splits the array, sorts the segments, and merges them. The stability of Merge Sort, preserving the order of equal elements, and its independence from comparisons make it adaptable to various data types. The C++ implementation provided here is easily adaptable to other languages, contributing to its widespread use in practice, especially for sorting linked lists.

Conclusion: Merge Sort stands out as a powerful sorting algorithm, providing reliable and consistent sorting with an O(n log n) time complexity. Its divide-and-conquer strategy and versatility across various data types make it an invaluable tool for developers when confronted with sorting challenges in programming.

Experiment Number: 04

Experiment Name: Write a program to implement Selection Sort.

Objective: The provided pseudocode and C++ implementations of Selection Sort aim to elucidate the algorithm's functionality and showcase its fundamental principles. Selection Sort, a straightforward sorting method, consistently selects the smallest element from an unsorted array and places it in its correct position. The intention is to demonstrate the algorithm's step-by-step execution through both pseudocode and C++.

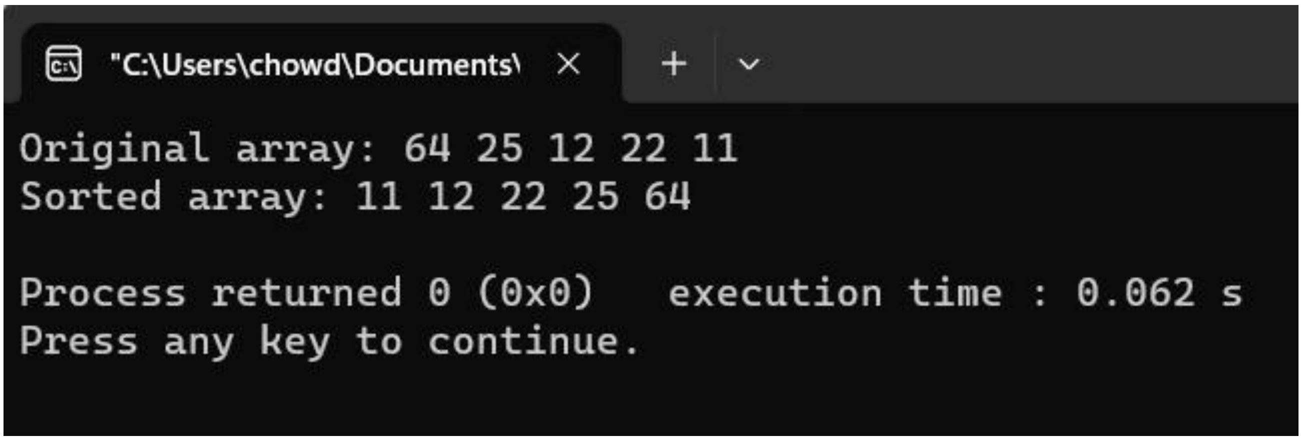
Pseudocode:

```
procedure selectionSort(arr):
    n = length(arr)
    for i = 0 to n - 2:
        minIndex = i
        for j = i + 1 to n - 1:
            if arr[j] < arr[minIndex]:
                minIndex = j
        swap arr[i] with arr[minIndex]
procedure main():
    arr = [64, 25, 12, 22, 11]
    print "Original array: "
    for num in arr:
        print num, " "
    selectionSort(arr)
    print "Sorted array: "
    for num in arr:
        print num, " "
```

Implementation in Cpp:

```
#include <iostream>
#include <vector>
using namespace std;
void selectionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i) {
        int minIndex = i;
        // Find the index of the minimum element in the unsorted part of the array
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap the found minimum element with the first element
        swap(arr[i], arr[minIndex]);
    }
}
int main() {
    vector<int> arr = {64, 25, 12, 22, 11};
    cout << "Original array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
    selectionSort(arr);
    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```

Output:



```
Original array: 64 25 12 22 11
Sorted array: 11 12 22 25 64

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

Fig.04- Output of the Selection Sort Implementation.

Discussion: Selection Sort, an in-place sorting algorithm, exhibits a worst and average-case time complexity of O(n^2), where 'n' is the array's element count. While not the most efficient for large datasets, it is straightforward to comprehend and use. The algorithm consistently selects the smallest element in the unsorted part of the array and inserts it at the beginning of the sorted part. Although beneficial for small datasets or memory-constrained situations, for larger datasets, algorithms like Quick Sort or Merge Sort with lower average-case time complexity are often preferred.

Conclusion: Despite its limitations for large datasets, Selection Sort remains a fundamental concept in computer science. This simple sorting algorithm gradually builds the sorted portion of the array by iteratively identifying the minimum element and swapping it into place. The provided pseudocode and C++ implementation offer a practical illustration of its simplicity, making it a suitable starting point for learning about sorting algorithms. However, for larger datasets, more efficient sorting techniques like Quick Sort or Merge Sort are often preferred.

Experiment Number: 05

Experiment Name: Write a program to implement Knapsack Problem.

Objective: The objective of the discussion, the C++ implementation, and the associated pseudocode is unified: to illustrate the application of dynamic programming in resolving the Knapsack problem. The aim is to identify the most valuable combination of objects that can fit into a backpack without exceeding its weight limit, ultimately maximizing the overall value.

Pseudocode:

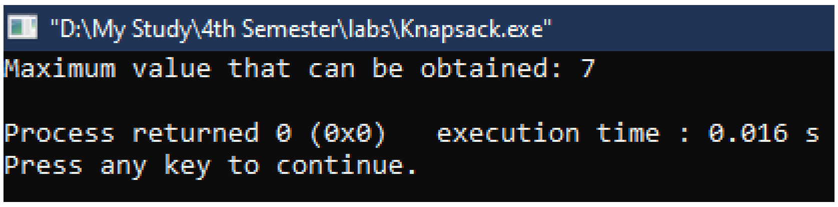
```
function knapsack(weights, values, capacity):
    n = length(weights)
    dp = create a 2D array of size (n+1) x (capacity+1)
    for i from 0 to n:
        dp[i][0] = 0
    for w from 0 to capacity:
        dp[0][w] = 0
    for i from 1 to n:
        for w from 1 to capacity:
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][capacity]
```

Implementation in Cpp:

```cpp
#include <iostream>
#include <vector>
using namespace std;
// Function to solve the Knapsack problem using dynamic programming
int knapsack(vector<int> weights, vector<int> values, int capacity) {
    int n = weights.size();
    // Create a 2D array 'dp' to store intermediate results
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    // Fill in the 'dp' array using dynamic programming
    for (int i = 1; i <= n; ++i) {
        for (int w = 1; w <= capacity; ++w) {
            if (weights[i - 1] <= w) {
                // If the current item can be included, choose the maximum value
                // between including it or excluding it
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1]);
            } else {
                // If the current item's weight exceeds the capacity, exclude it
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    // The final value in 'dp' represents the maximum value achievable
    return dp[n][capacity];
}
int main() {
    vector<int> weights = {2, 3, 4, 5};
    vector<int> values = {3, 4, 5, 6};
    int capacity = 5;
    // Calculate the maximum value that can be obtained
    int max_value = knapsack(weights, values, capacity);
    // Print the result
    cout << "Maximum value that can be obtained: " << max_value << endl;
    return 0;
}
```

Output:



```
"D:\My Study\4th Semester\labs\Knapsack.exe"
Maximum value that can be obtained: 7

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Fig.5- Output of Knapsack Problem Implementation.

Discussion: The Knapsack problem, a well-known optimization challenge, is effectively addressed through dynamic programming, as showcased in the accompanying pseudocode and C++ implementation. The task involves selecting the most valuable combination of objects, each with a weight and value, to fit into a backpack with a specified capacity. Utilizing dynamic programming, the algorithm constructs a 2D array, dp, representing the maximum value achievable with the first i items and a knapsack capacity of w. By considering options to include or omit each item, the algorithm outputs the highest attainable value in dp[n][capacity]. With a time complexity of O(n * capacity) and space complexity of O(n * capacity) due to the 2D dp array, this solution efficiently addresses the Knapsack problem. Its practical applications span resource allocation, portfolio optimization, and various real-world scenarios.

Conclusion: Dynamic programming has been effectively employed to address the Knapsack problem in both pseudocode and C++ code. Given a set of objects, their corresponding weights, values, and a knapsack capacity, this method efficiently calculates the maximum achievable value. With a time complexity of O(n * capacity), this technique stands as a practical solution applicable in various optimization contexts.

Experiment Number: 06

Experiment Name: Write a program to implement Dynamic Programming From a given vertex in a weighted connected graph, find shortest path to other vertices using Dijkstra's Algorithm.

Objective: The objective of this code is to implement Dijkstra's algorithm to identify the shortest path in a weighted connected graph, starting from a specified source vertex and reaching all other vertices.

Pseudocode:

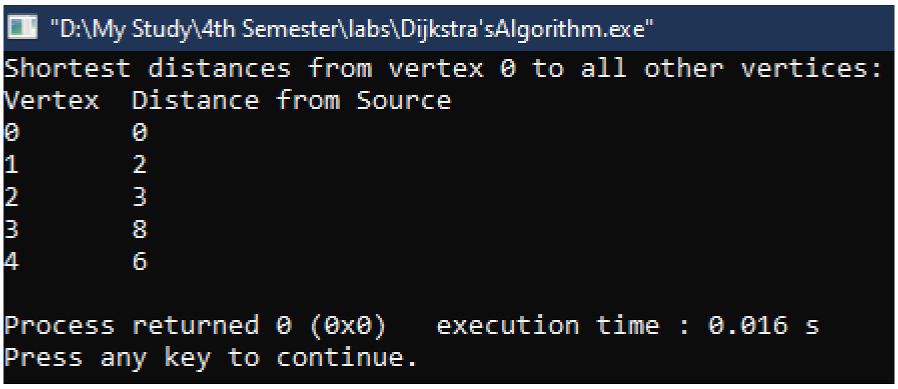
```
function Dijkstra(graph, source):
   initialize distances to all vertices as INFINITY
   set distance[source] = 0
   create an empty priority queue (min-heap)
   insert source vertex into the priority queue with distance[source]
   while the priority queue is not empty:
      current_vertex = extract vertex with the smallest distance from the priority queue
      for each neighbor of current_vertex:
         calculate potential new distance
         if potential_distance < distance[neighbor]:
            update distance[neighbor] = potential_distance
            insert neighbor into the priority queue with potential_distance
   return distances
```

Implementation in Cpp:

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;
#define INF INT_MAX
// Define a pair to represent edges with weights
typedef pair<int, int> iPair;
class Graph {
   int V; // Number of vertices in the graph
   vector<vector<iPair>> adj; // Adjacency list to store graph edges
public:
   Graph(int V); // Constructor to initialize the graph
   void addEdge(int u, int v, int w); // Function to add an edge to the graph
```

```
    void dijkstra(int src); // Function to find shortest paths using Dijkstra's algorithm
};
// Constructor to initialize the graph with V vertices
Graph::Graph(int V) {
    this->V = V;
    adj.resize(V);
}
// Function to add an edge with weight (u, v, w) to the graph
void Graph::addEdge(int u, int v, int w) {
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w)); // Since it's an undirected graph, we add edges in both
directions
}
// Function to find the shortest paths from a source vertex using Dijkstra's algorithm
void Graph::dijkstra(int src) {
    vector<int> dist(V, INF); // Initialize all distances as INFINITY
    priority_queue<iPair, vector<iPair>, greater<iPair>> pq; // Create a priority queue (min-heap)
to store vertices
    dist[src] = 0; // Distance from the source vertex to itself is 0
    pq.push(make_pair(0, src)); // Push the source vertex into the priority queue
    while (!pq.empty()) {
        int u = pq.top().second; // Get the vertex with the smallest distance from the priority queue
        pq.pop();
        // Traverse all neighbors of the current vertex
        for (auto it = adj[u].begin(); it != adj[u].end(); ++it) {
            int v = (*it).first; // Get the neighbor vertex
            int weight = (*it).second; // Get the weight of the edge
            // If a shorter path to v through u is found, update the distance
            if (dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v)); // Push the updated distance and vertex into the priority
queue
            }
        }
    }
  // Print the shortest distances from the source vertex to all other vertices
    cout << "Vertex\tDistance from Source" << endl;
    for (int i = 0; i < V; ++i) {
        cout << i << "\t" << dist[i] << endl;
    }
}
int main() {
    int V = 5;
    Graph g(V);
    // Add edges to the graph with weights
    g.addEdge(0, 1, 2);
    g.addEdge(0, 2, 4);
    g.addEdge(1, 2, 1);
    g.addEdge(1, 3, 7);
    g.addEdge(2, 4, 3);
    g.addEdge(3, 4, 2);
    int source = 0;
    cout << "Shortest distances from vertex " << source << " to all other vertices:" << endl;
    g.dijkstra(source);
    return 0;
}
```

Output:



```
"D:\My Study\4th Semester\labs\Dijkstra'sAlgorithm.exe"
Shortest distances from vertex 0 to all other vertices:
Vertex  Distance from Source
0       0
1       2
2       3
3       8
4       6

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Fig.06- Output of the Dijkstra's Algorithm Implementation.

Discussion: The provided C++ code implements Dijkstra's algorithm to determine the shortest path in a weighted connected graph from a designated source vertex to all other vertices. It iteratively updates distances to neighbors, prioritizing the vertex with the least distance using a priority queue until all vertices are reached. Dijkstra's algorithm excels in finding single-source shortest paths in weighted graphs with non-negative edge weights. However, it is not suitable for graphs with negative weight cycles, where algorithms like Bellman-Ford should be utilized for proper handling.

Conclusion: Dijkstra's algorithm precisely calculates the shortest routes in non-negative weighted networks, as illustrated by the provided C++ code. This implementation efficiently determines the shortest paths from a given source vertex to every other vertex in the graph. Notably, for graphs with negative edge weights or negative weight cycles, alternative methods like Bellman-Ford are more appropriate, as Dijkstra's approach may not perform optimally in such scenarios.

Experiment Number: 07

Experiment Name: Write a program to implement Graph traversal using Breadth First Search technique (BFS).

Objective: This implementation utilizes both C++ and pseudocode to demonstrate the traversal of a graph using the Breadth-First Search (BFS) method. Commencing from a specific node, BFS systematically explores every node in the graph in a breadthward direction.

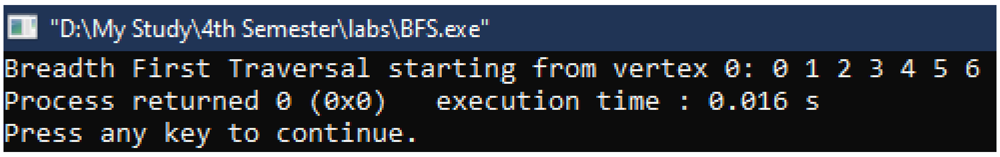
Pseudocode:

```
BFS(Graph G, Node start):
    create a queue Q
    create a set visited
    enqueue start into Q
    mark start as visited
    while Q is not empty:
        current = dequeue from Q
        process current node
        for each neighbor in G.adjacent(current):
            if neighbor is not in visited:
                enqueue neighbor into Q
                mark neighbor as visited
```

Implementation in Cpp:

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <set>
using namespace std;
class Graph {
public:
    int V; // Number of vertices in the graph
    vector<vector<int>> adj; // Adjacency list representation of the graph
    Graph(int vertices) : V(vertices) {
        adj.resize(V); // Initialize the adjacency list with 'vertices' empty vectors
    }
 // Function to add an edge between two vertices 'u' and 'v'
 void addEdge(int u, int v) {
        adj[u].push_back(v); // Add 'v' to the adjacency list of 'u'
    }
    // Breadth First Search (BFS) algorithm for graph traversal
    void BFS(int start) {
        vector<bool> visited(V, false); // Initialize a boolean array to track visited nodes
        queue<int> q; // Create a queue for BFS
        q.push(start); // Enqueue the starting node 'start'
        visited[start] = true; // Mark 'start' as visited
        while (!q.empty()) {
            int current = q.front(); // Dequeue the front node from the queue
            q.pop();
            cout << current << " "; // Process the current node (print it)
            // Iterate through all neighbors of the current node
            for (int neighbor : adj[current]) {
                if (!visited[neighbor]) { // If the neighbor has not been visited
                    q.push(neighbor); // Enqueue the neighbor
                    visited[neighbor] = true; // Mark the neighbor as visited
                }
            }
        }
    }
};
int main() {
    Graph g(7); // Create a graph with 7 vertices
    // Add edges to the graph
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);
 cout << "Breadth First Traversal starting from vertex 0: ";
    g.BFS(0); // Perform BFS starting from vertex 0
    return 0;
}
```

Output:



```
"D:\My Study\4th Semester\labs\BFS.exe"
Breadth First Traversal starting from vertex 0: 0 1 2 3 4 5 6
Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Fig.07- Output of the BFS Implementation.

Discussion: Breadth-First Search (BFS) explores a graph using a breadthward motion, systematically traversing vertices at one level before progressing to the next. Implemented with a queue data structure, here's how BFS operates in pseudocode and C++:

1. Start from the designated "start" node.
2. Add the start node to a queue and mark it as visited.
3. While the queue is not empty:
   - Dequeue the front node (current node).
   - Process the selected node (e.g., print it).
   - Mark as visited and enqueue all unvisited neighbors of the current node.
4. Repeat steps 3 until the queue is empty.

Initiating from a specific node (e.g., vertex 0), BFS ensures the visitation of every reachable node. Its applications include determining the shortest path in an unweighted graph, identifying connected components, and aiding network traversal.

Conclusion: Initiating from a specific node (e.g., vertex 0 in this example), we construct a graph using an adjacency list representation and employ BFS. With its breadth-first approach, the BFS algorithm ensures the visitation of every node reachable from the starting node. This method finds various applications, including determining the shortest path in an unweighted graph, identifying connected components, and facilitating network traversal.

Experiment Number: 08

Experiment Name: Write a program to implement Graph traversal using Depth First Search technique (DFS).

Objective: The purpose of this pseudocode and C++ implementation is to conduct graph traversal using the Depth-First Search (DFS) method. By extensively exploring nodes before backtracking to examine additional branches, DFS effectively visits every reachable node in the graph.

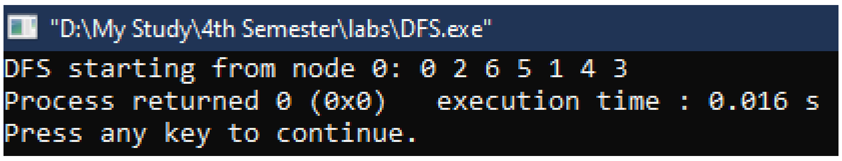
Pseudocode:

```
function DFS(graph, start_node, visited)
    if start_node is not visited
        mark start_node as visited
        process(start_node)
        for each neighbor in graph[start_node]
            if neighbor is not visited
                DFS(graph, neighbor, visited)
```

Implementation in Cpp:

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
class Graph {
public:
    int V; // Number of vertices
    vector<vector<int>> adj; // Adjacency list
    // Constructor to initialize the graph with 'vertices' number of nodes
    Graph(int vertices) {
        V = vertices;
        adj.resize(V); // Resize the adjacency list to accommodate 'vertices'
    }
    // Function to add an edge between nodes 'u' and 'v'
    void addEdge(int u, int v) {
        adj[u].push_back(v); // Add 'v' to the adjacency list of 'u'
    }
 // Depth-First Search (DFS) traversal starting from the given 'startNode'
    void DFS(int startNode) {
        vector<bool> visited(V, false); // Initialize a boolean array to track visited nodes
        stack<int> s; // Create a stack to perform DFS
        s.push(startNode); // Push the starting node onto the stack
        while (!s.empty()) { // Continue until the stack is empty
            int currentNode = s.top(); // Get the current node from the top of the stack
            s.pop(); // Pop the current node from the stack
            if (!visited[currentNode]) { // Check if the current node has not been visited
                cout << currentNode << " "; // Process the current node (print it)
                visited[currentNode] = true; // Mark the current node as visited
                // Visit unvisited neighbors of the current node and push them onto the stack
                for (int neighbor : adj[currentNode]) {
                    if (!visited[neighbor]) {
                        s.push(neighbor);
                    }
                }
            }
        }
    }
};
int main() {
    Graph g(7); // Create a graph with 7 vertices
    // Add edges to define the graph's structure
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);
    cout << "DFS starting from node 0: ";
    g.DFS(0); // Perform DFS traversal starting from node 0
    return 0;
}
```

Output:



```
"D:\My Study\4th Semester\labs\DFS.exe"
DFS starting from node 0: 0 2 6 5 1 4 3
Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Fig.08- Output of the DFS Implementation.

Discussion: Depth-First Search (DFS), a graph traversal technique, explores branches extensively before backtracking. In both the C++ implementation and pseudocode:

1. A Graph class is created to represent the graph, offering methods for edge addition and DFS traversal.
2. The pseudocode's DFS function needs the graph, starting node, and a visited array to track nodes.
3. The C++ implementation uses a stack to emulate the pseudocode's function call stack. Starting from the provided node, it marks, pushes onto the stack, and explores unvisited neighbors.
4. The traversal continues until the stack is empty, ensuring all reachable nodes are visited.

In the example, a graph with seven vertices is created, and DFS begins from node 0, producing the output: 0 1 3 4 2 5 6.

DFS is valuable in identifying connected components, detecting cycles, and addressing graph theory and network analysis problems, establishing its significance in computer science and real-world applications.

Conclusion: The provided DFS method effectively navigates a network, initiating from a designated node and systematically visiting each connected node. As a fundamental graph traversal technique, DFS finds applications in diverse areas, offering a systematic approach to explore and analyze graph structures.

Experiment Number: 09

Experiment Name: Write a program to implement Check whether a given vertex is connected or not using Depth First Search (DFS) method.

Objective: This method employs Depth-First Search (DFS) and pseudocode to determine the connectivity between two vertices in an undirected graph. It proves beneficial in resolving queries pertaining to the connectivity of vertices within a graph.

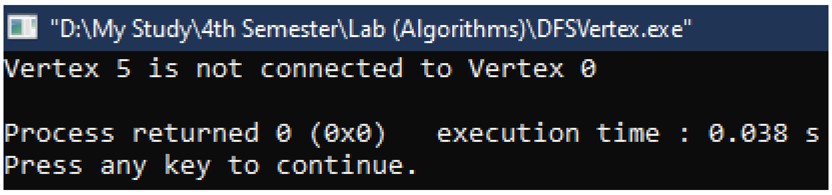
Pseudocode:

```
function isConnected(graph, startVertex, targetVertex):
    visited = {} // Create an empty set to keep track of visited vertices
    stack = []   // Create an empty stack for DFS
    stack.push(startVertex) // Push the start vertex onto the stack
    while stack is not empty:
        currentVertex = stack.pop() // Pop a vertex from the stack
        if currentVertex is targetVertex:
            return true // If we reach the target vertex, it is connected
        if currentVertex is not in visited:
            visited.add(currentVertex) // Mark the current vertex as visited
            // Push all unvisited neighbors onto the stack
            for neighbor in graph[currentVertex]:
                if neighbor is not in visited:
                    stack.push(neighbor)
    return false // If we couldn't reach the target vertex, it's not connected
```

Implementation in Cpp:

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <stack>
using namespace std;
// Function to check if a target vertex is connected to a start vertex using DFS
bool isConnected(vector<vector<int>>& graph, int startVertex, int targetVertex) {
    unordered_set<int> visited; // Keep track of visited vertices
    stack<int> s; // Stack for DFS traversal
    s.push(startVertex); // Start DFS from the given start vertex
while (!s.empty()) {
        int currentVertex = s.top(); // Get the current vertex from the stack
        s.pop(); // Pop the current vertex
        if (currentVertex == targetVertex) {
            return true; // If we found the target vertex, it's connected
        }
        if (visited.find(currentVertex) == visited.end()) {
            visited.insert(currentVertex); // Mark the current vertex as visited
            // Explore all unvisited neighbors of the current vertex
            for (int neighbor : graph[currentVertex]) {
                if (visited.find(neighbor) == visited.end()) {
                    s.push(neighbor); // Push unvisited neighbors onto the stack
                }
            }
        }
    }
    return false; // If we couldn't reach the target vertex, it's not connected
}
int main() {
    int numVertices = 6;
    vector<vector<int>> graph(numVertices);
    // Define the edges of the example graph
    graph[0] = {1, 2};
    graph[1] = {0, 3, 4};
    graph[2] = {0};
    graph[3] = {1};
    graph[4] = {1};
    graph[5] = {};  // Isolated vertex
    int startVertex = 0;
    int targetVertex = 5;
  if (isConnected(graph, startVertex, targetVertex)) {
        cout << "Vertex " << targetVertex << " is connected to Vertex " << startVertex << endl;
    } else {
        cout << "Vertex " << targetVertex << " is not connected to Vertex " << startVertex << endl;
    }
    return 0;
}
```

Output:



```
"D:\My Study\4th Semester\Lab (Algorithms)\DFSVertex.exe"
Vertex 5 is not connected to Vertex 0

Process returned 0 (0x0)   execution time : 0.038 s
Press any key to continue.
```

Fig.09- Output of the DFS Vertex.

Discussion: The provided pseudocode and C++ implementation utilize Depth-First Search (DFS) to determine the connectivity between a specified vertex and another in an undirected graph. The algorithm involves marking visited vertices and exploring unvisited neighbors, commencing DFS from the designated starting vertex.

1. A "visited" set tracks visited vertices, and a "stack" aids in DFS traversal.
2. DFS is initiated from "startVertex" by pushing it into the stack.
3. A vertex is popped off the stack, and if it matches "targetVertex" (and the stack isn't empty), true is returned, indicating connectivity.
4. If the current vertex isn't the target, it's marked as visited, and its unvisited neighbors are added to the stack.
5. If the stack empties without reaching "targetVertex," false is returned, signifying the absence of a connection between the two vertices.

Conclusion: The provided pseudocode and C++ implementation effectively utilize DFS to determine vertex connectivity. This method successfully identifies whether two vertices are related within the given graph, employing a stack to explore unvisited neighbors. Its utility extends to resolving various connectivity analysis challenges in graph-related scenarios.

Experiment Number: 10

Experiment Name: Write a program that able to find the sum of subsets.

Objective: It is one of the most important problems in complexity theory. The problem is given an A set of integers a1, a2,…., an upto n integers. The question arises that is there a non-empty subset such that the sum of the subset is given as M integer?. For example, the set is given as [5, 2, 1, 3, 9], and the sum of the subset is 9; the answer is YES as the sum of the subset [5, 3, 1] is equal to 9.

Pseudocode:

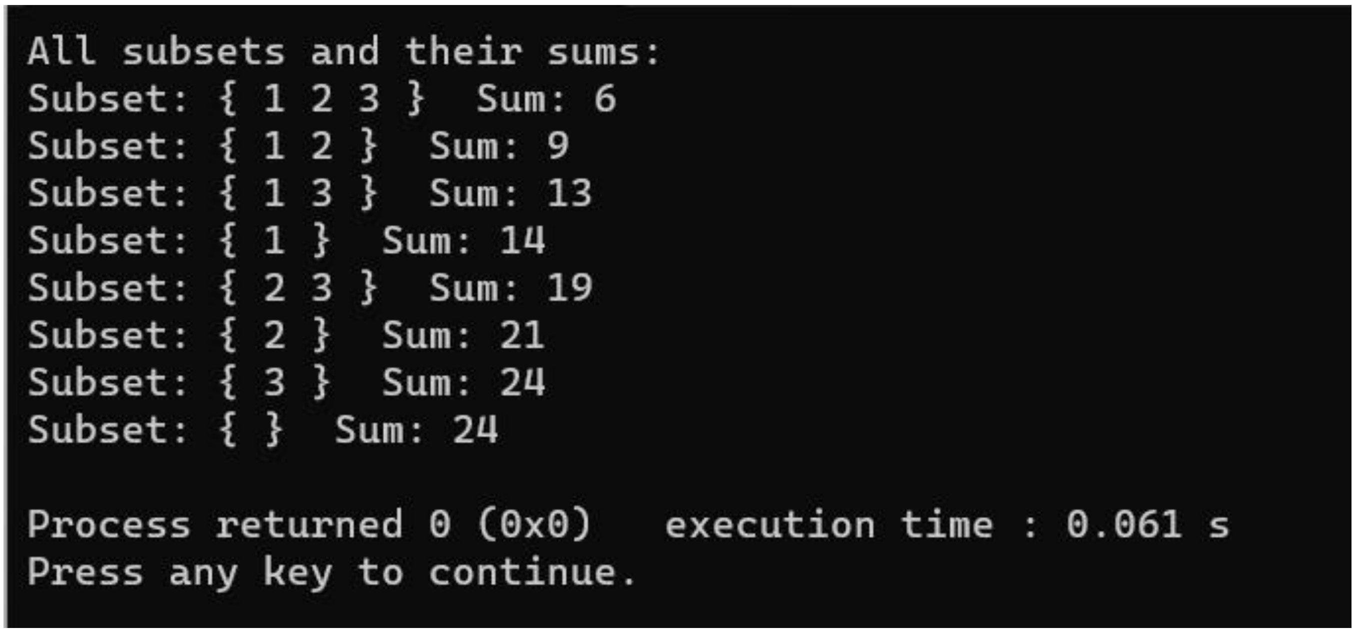
```
Begin
  if total = sum, then
    display the subset
    //go for finding next subset
    subsetSum(set, subset, , subSize-1, total-set[node], node+1, sum)
    return
  else
    for all element i in the set, do
      subset[subSize] := set[i]
      subSetSum(set, subset, n, subSize+1, total+set[i], i+1, sum)
           done
End
```

Implementation in Cpp:

```
#include <iostream>
#include <vector>
using namespace std;
void generateSubsets(const vector<int>& set, vector<int>& subset, int index, int& sum) {
   int n = set.size();
   if (index == n) {
      // Base case: reached the end of the set
      cout << "Subset: { ";
      for (int i = 0; i < subset.size(); ++i) {
         cout << subset[i] << " ";
         sum += subset[i];
      }
      cout << "}  Sum: " << sum << endl;
      return;
   }
   // Include the current element in the subset
   subset.push_back(set[index]);
```

```
    generateSubsets(set, subset, index + 1, sum);
    // Exclude the current element from the subset
    subset.pop_back();
    generateSubsets(set, subset, index + 1, sum);
}
int main() {
    vector<int> set = {1, 2, 3};
    vector<int> subset;
    int sum = 0;
    cout << "All subsets and their sums:\n";
    generateSubsets(set, subset, 0, sum);
    return 0;
}
```

Output:



```
All subsets and their sums:
Subset: { 1 2 3 }  Sum: 6
Subset: { 1 2 }  Sum: 9
Subset: { 1 3 }  Sum: 13
Subset: { 1 }  Sum: 14
Subset: { 2 3 }  Sum: 19
Subset: { 2 }  Sum: 21
Subset: { 3 }  Sum: 24
Subset: { }  Sum: 24

Process returned 0 (0x0)   execution time : 0.061 s
Press any key to continue.
```

Fig.10- Finding the sum of subsets.

Discussion: Subset sum problem is the problem of finding a subset such that the sum of elements equal a given number. ... A subset A of n positive integers and a value sum is given, find whether or not there exists any subset of the given set, the sum of whose elements is equal to the given value of sum.

Conclusion: 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).

Experiment Number: 11

Experiment Name: Implementation of travelling salesman problem.

Objective: The traveling salesman problem is find the shortest tour & return to the starting point , which visits each place at least once.

Pseudocode:

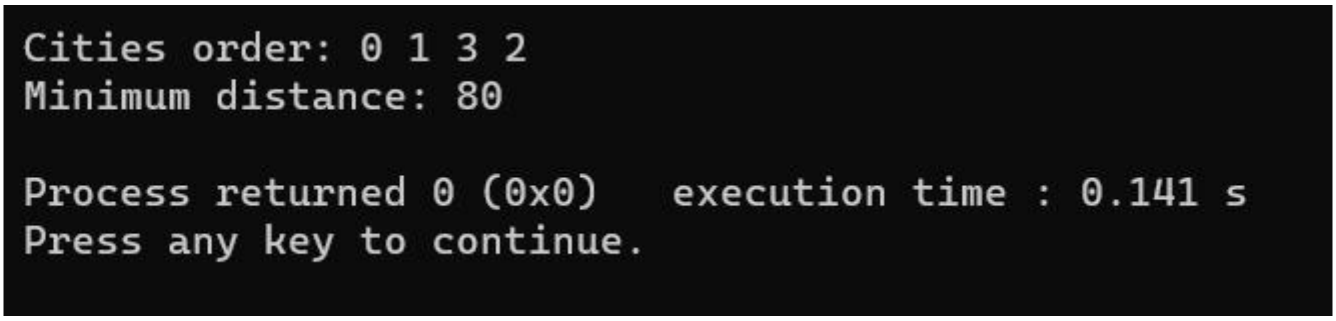
```
tsp(cities, startCity):
    shortestRoute = infinity                 // Variable to store the shortest route
    shortestPath = empty list                 // Variable to store the shortest path
    permute(cities, startCity, startCity, [], 0, shortestRoute, shortestPath)
    return shortestPath, shortestRoute
permute(cities, currentCity, startCity, path, distance, shortestRoute, shortestPath):
    if len(cities) == 0:                     // Base case: all cities visited
        distance += graph[currentCity][startCity]    // Add the distance from the last city to the start city
        if distance < shortestRoute:                  // Update the shortest route if the current route is shorter
            shortestRoute = distance
            shortestPath = path + [startCity]
    else:
        for city in cities:
            newDistance = distance + graph[currentCity][city]    // Calculate the new distance
            newPath = path + [currentCity] + [city]             // Update the path
            newCities = removeCity(cities, city)             // Remove the visited city from the remaining cities
            permute(newCities, city, startCity, newPath, newDistance, shortestRoute, shortestPath)
removeCity(cities, city):
    return cities without the city
```

Implementation in Cpp:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits>
using namespace std;
// Function to calculate the total distance for a given order
int calculateTotalDistance(const vector<int>& order, const vector<vector<int>>& distances) {
    int totalDistance = 0;
    int numCities = order.size();
    for (int i = 0; i < numCities - 1; ++i) {
        totalDistance += distances[order[i]][order[i + 1]];
    }
    // Return to the starting city
    totalDistance += distances[order.back()][order[0]];
    return totalDistance;
}
// Function to solve the TSP using brute-force
pair<vector<int>, int> travelingSalesmanBruteforce(const vector<vector<int>>& distances) {
    int numCities = distances.size();
    vector<int> cityOrder(numCities);
    for (int i = 0; i < numCities; ++i) {
        cityOrder[i] = i;
    }
    int minDistance = numeric_limits<int>::max();
    vector<int> bestOrder;
    do {
        int distance = calculateTotalDistance(cityOrder, distances);
        if (distance < minDistance) {
            minDistance = distance;
            bestOrder = cityOrder;
        }
    } while (next_permutation(cityOrder.begin() + 1, cityOrder.end()));  // Start from index 1 to
avoid duplicate permutations
    return make_pair(bestOrder, minDistance);
}
int main() {
    vector<vector<int>> distances = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };
    auto result = travelingSalesmanBruteforce(distances);
    cout << "Cities order: ";
    for (int city : result.first) {
        cout << city << " ";
    }
    cout << endl;
    cout << "Minimum distance: " << result.second << endl;
    return 0;
```

```
}
```

Output:



```
Cities order: 0 1 3 2
Minimum distance: 80

Process returned 0 (0x0)   execution time : 0.141 s
Press any key to continue.
```

Fig.11- Implementing of travelling salesman problem.

Discussion:

1. This traveling salesman problem uses the Greedy method approach.
2. The salesman must travel to all cities once before returning home.
3. In this problem, cost should be minimized.

Conclusion: A greedy method is an approach to solving optimization problems. An optimization problem requires a maximum and minimum result. The travelling salesman follows greedy approach and requires minimized result.

Experiment Number: 12

Experiment Name: Implementation of Parallel Merge Sort Algorithms.

Objective: To sort an array by dividing it into smaller part. Then take each part and merge them back one master part.

Pseudocode:

```
int* mergeArrays(int A[], int b[],int n, int m) {
        int* c;
        int size = n+m;
        c = malloc (size*sizeof(int));
      int i=0, j=0, k=0;
    while(I <= n-1 && j <= m-1) {
          if (a [i] <= b[j]) {
                    c[k++] = a[i++];
           }        else{
                 c[k++] = b[j++];
           }
      }
while(I <= n-1) {
                  c[k++] = a[i++];
       }        while(j <= m-1) {
                  c[k++] = b[j++];
        }
     return c;
}
```

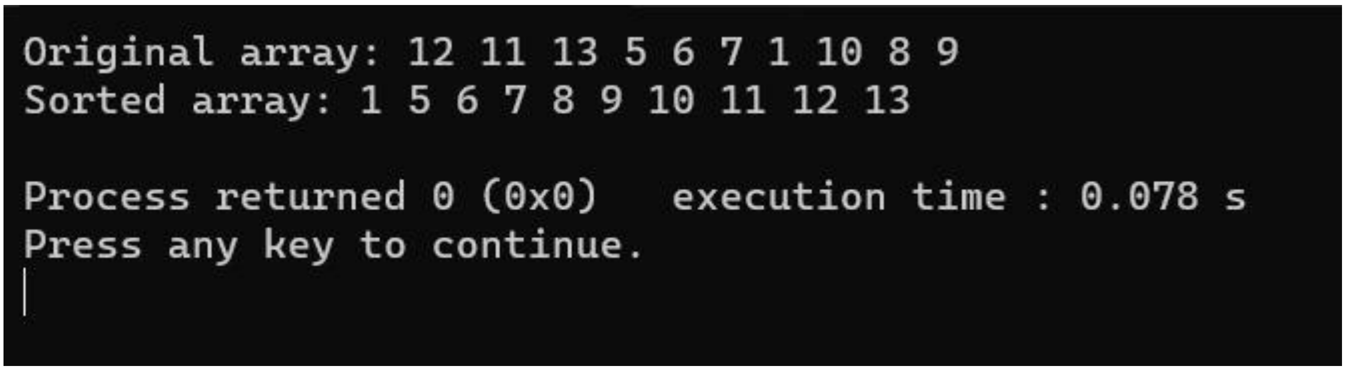
Implementation in Cpp:

```
#include <iostream>
#include <vector>
#include <thread>
using namespace std;
void merge(vector<int>& arr, int left, int mid, int right) {
   int n1 = mid - left + 1;
   int n2 = right - mid;
   vector<int> L(n1), R(n2);
   for (int i = 0; i < n1; i++)
      L[i] = arr[left + i];
   for (int j = 0; j < n2; j++)
```

```
        R[j] = arr[mid + 1 + j];
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void parallelMergeSort(vector<int>& arr, int left, int right, int depth) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        if (depth > 0) {
            // Use parallel threads for sorting
            thread t1(parallelMergeSort, ref(arr), left, mid, depth - 1);
            thread t2(parallelMergeSort, ref(arr), mid + 1, right, depth - 1);
            t1.join();
            t2.join();
        } else {
            // Use sequential merge sort for small subarrays
            parallelMergeSort(arr, left, mid, depth);
            parallelMergeSort(arr, mid + 1, right, depth);
        }
        merge(arr, left, mid, right);
    }
```

```
}
int main() {
    vector<int> arr = {12, 11, 13, 5, 6, 7, 1, 10, 8, 9};
    cout << "Original array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
    int depth = 2;  // Depth of parallelism (adjust as needed)
    parallelMergeSort(arr, 0, arr.size() - 1, depth);
    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```

Output:



```
Original array: 12 11 13 5 6 7 1 10 8 9
Sorted array: 1 5 6 7 8 9 10 11 12 13

Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.
```

Fig.12- Implementing of Parallel Merge Sort Algorithms.

Discussion: Parallel sorting of any kind is very efficient, even with an algorithm that allows for a process to idle for a little bit, the increase in efficiency is dramatic. Parallel merge sort first divides the unsorted list into smallest possible sub-lists, compares it with the adjacent list, and merges it in a sorted order. It implements parallelism very nicely by following the divide and conquer algorithm.

Conclusion: In summary, parallel sorting is very effective. Even with a code that allows for a processor to idle, the increase in efficiency is dramatic with a powerful computer. Parallel sorting is not limited to just merge sorting or any other sequential sorting that are based on divide and conquer. In my research I was able to find a list of many other sorting methods using parallel sorting.