1. Program Definition:
Sort a given set of elements using Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

**Aim:**

The aim of this program is to sort ‘n’ randomly generated elements using Quick sort and plot the graph of the time taken to sort n elements versus n.

**Implementation:**

- Call a function to generate list of random numbers (integers)
- Record clock time
- Call a Quick sort function to sort n randomly generated elements.
- Record clock time.
- Measure difference in clock time to get elapse time to sort n elements using Quick sort.
- Print the Sorted ‘n’ elements and time taken to sort.
- Repeat the above steps for different n values.

**Pseudo code:**

1. Declare time variables
   
   struct timeval tv;
   
   double start,end;

2. call clock( ) function to record the start time in terms of seconds before sorting
   
   gettimeofday(&tv,NULL);
   
   start = tv.tv_sec+(tv.tv_usec/1000000.0);

3. Generate ‘n’ elements randomly using rand () function
   
   int a[100],n;
   
   Print “enter the total number to be generated”
   
   Read n
   
   For i= 0 to n-1
   
   a[i]=rand( )
   
4. Call Quick sort function to sort n elements

ALGORITHM Quick sort (A[1….r])
ADA LAB

// Sorts a sub array by quick sort
// Input: A sub array A[l..r] of A[0..n-1], defined by its left and right indices l and r
// Output: The sub array A[l..r] sorted in non decreasing order

if 1 < r
    s = Partition(A[l..r]) // s is a split position
    Quick sort(A[l...s-1])
    Quick sort(A[s+1...r])

ALGORITHM Partition(A[l...r])

// Partition a sub array by using its first element as a pivot
// Input: A sub array A[l...r] of A[0...n-1] defined by its left and right indices l and r (l < r)
// Output: A partition of A[l...r], with the split position returned as this function’s value

p=A[l]
i=l;
j=r+1;
repeat
    repeat i= i+1 until A[i] >= p
    repeat j=j-1 until A[j] <= p
    Swap(A[i],A[j])
until i >= j
Swap(A[i],A[j]) // Undo last Swap when i >= j
Swap(A[l],A[j])
Return j

5. Call clock() function to record the end time in terms of seconds after sorting
   gettimeofday(&tv,NULL);
   end = tv.tv_sec+(tv.tv_usec/1000000.0);

6. Calculate the time in terms of seconds required to sort n elements using Quick sort i.e elapse time
   elapse_time = (end-start);
7. Repeat the above steps for different ‘n’ values i.e 10, 15, 20, 25, 30

**Result:**

- Randomly generated ‘n’ numbers are sorted using Quick sort.
- Calculate the elapse time required for different ‘n’ elements

<table>
<thead>
<tr>
<th>N (no of elements)</th>
<th>Elapse time (Total time required in terms of seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

- Plot a graph of N versus Elapse time
2. **Program Definition:**

Using OpenMP, implement a parallelized Merge Sort algorithm to sort given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of \( n \), the number of elements in the list to be sorted and plot a graph of the time taken versus \( n \). The elements can be read from a file or can be generated using the random number generator.

**Aim:**

The aim of this program is to sort ‘\( n \)’ randomly generated elements using parallelized Merge sort algorithm and Plotting the graph of the time taken to sort \( n \) elements versus \( n \).

**Implementation:**

- Call a function to generate list of random numbers (integers)
- Record clock time
- Call a Merge sort function to sort \( n \) randomly generated elements.
- Merge sort function calls itself recursively for the two half portions
- Create two pragma sections two parallelize the call
- Record clock time.
- Measure difference in clock time to get elapse time to sort \( n \) elements using Merge sort.
- Print the Sorted ‘\( n \)’ elements and time taken to sort.
- Repeat the above steps for different \( n \) values.

**Pseudo code:**

1. Declare time variables
   ```
   struct timeval tv;
   double start, end;
   ```
   call the function `omp_set_num_threads(2)` which is in the headerfile `omp.h` to create two threads
2. Generate ‘\( n \)’ elements randomly using `rand()` function
   ```
   int a[100].n;
   ```
   Print “enter the total number to be generated”
   Read \( n \)
   for i= 0 to n-1
3. call clock( ) function to record the start time in terms of seconds before sorting
   gettimeofday(&tv,NULL);
   start=tv.tv_sec+(tv.tv_usec/1000000.0);

4. Call Parallelized Merge sort function to sort n elements

ALGORITHM Merge sort (A[0…n-1])
   // Sorts array A[0..n-1] by Recursive merge sort
   // Input : An array A[0..n-1] elements
   // Output : Array A[0..n-1] sorted in non decreasing order
   If n > 1
      Create two omp parallel sections
      In first omp section
      Copy A[0…(n/2)-1] to B[0…(n/2)-1]
      Mergesort (B[0…(n/2)-1])
      In second omp section
      Copy A[0…(n/2)-1] to C[0…(n/2)-1]
      Mergesort (C[0…(n/2)-1])
      Merge(B,C,A)

ALGORITHM Merge (B[0…p-1], C[0…q-1],A[0…p+q-1])
   // merges two sorted arrays into one sorted array
   // Input : Arrays B[0..p-1] and C[0…q-1] both sorted
   // Output : Sorted array A[0…p+q-1] of the elements of B and C
   I = 0;
   J = 0;
   K= 0;
   While I < p and j < q do
      If B[i] <= C[j]
         A[k]= B[I];      I= I+1;
      Else
5. call clock( ) function to record the end time in terms of seconds after sorting 
   gettimeofday(&tv,NULL);
   end=tv.tv_sec+(tv.tv_usec/1000000.0);

6. Calculate the time in terms of seconds required to sort n elements using Merge sort i.e elapse time
   
elapse_time = (end-start);
   Print "elapse_time".

7. Repeat the above steps for different ‘n’ values i.e 10, 15, 20, 25, 30

**Result:**
- Randomly generated ‘n’ numbers are sorted using Merge sort.
- Calculate the elapse time required for different ‘n’ elements

<table>
<thead>
<tr>
<th>N (no of elements)</th>
<th>Elapse time (Total time required in terms of seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
3. Program Definition:

   a. Obtain the topological ordering of vertices in a given digraph.
   b. Compute the transitive closure of a given directed graph using Warshall's algorithm.

Aim:

The aim of this program is to obtain topological ordering of vertices of a given digraph.

Implementation:

1. Repeatedly, identify in a remaining digraph a source, which is a vertex with no incoming edges.
2. Delete it along with all the edges outgoing from it.
3. The order in which the vertices are deleted yields a solution to the topological sorting problem.

Pseudo code:

output ← Empty list that will contain the sorted nodes
stack ← Set of all nodes with no outgoing edges(sources)
indegree←List having indegree of each vertex

1. Read the adjacency matrix
2. Find the indegree of each vertex
3. Push in to the stack all the vertices with indegree zero.
Repeat the steps 4-7 till stack is empty
4. Delete the vertex at the top of the stack that is current source insert that vertex in to output array
5. Delete all the vertices on outgoing edges from the current source(decrement indegree by 1)
6. If indegree of the vertex becomes zero push the vertex in to the stack if it not present already
7. Print all the vertices of the output array.
B. Warshall’s Algorithm

AIM: To find Transitive closure

**Algorithm**

//Input: Adjacency matrix of digraph
//Output: R, transitive closure of digraph

- Accept no. of vertices
- Call graph function to read directed graph
- Set R[ ] <- digraph matrix // get R {r(i,j)} for k=0
- Print digraph
- Repeat for k = 1 to n
  - Repeat for i = 1 to n
    - Repeat for j = 1 to n
      - \( R(i,j) = 1 \) if
        \[ \{ r_{ij}^{(k-1)} = 1 \ \text{OR} \ \ r_{ik}^{(k-1)} = 1 \ \text{OR} \ \ r_{kj}^{(k-1)} = 1 \} \]
  - Print R
4. Program Definition:

Implement 0/1 Knapsack problem using dynamic programming.

AIM:

We are given a set of $n$ items from which we are to select some number of items to be carried in a knapsack (BAG). Each item has both a weight and a profit. The objective is to choose the set of items that fits in the knapsack and maximizes the profit.

Given a knapsack with maximum capacity $W$, and a set $S$ consisting of $n$ items, Each item $i$ has some weight $w_i$ and benefit value $b_i$ (all $w_i$, $b_i$ and $W$ are integer values).

Problem: How to pack the knapsack to achieve maximum total value of packed items?

USING: Dynamic programming

It gives us a way to design custom algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency).

ALGORITHM

Algorithm:

$\text{//(n items, W weight of sack) Input: n, w_i, v_i and W – all integers}$

$\text{//Output: V(n,W)}$

$\text{// Initialization of first column and first row elements}$

- Repeat for $i = 0$ to $n$
  - set $V(i,0) = 0$
- Repeat for $j = 0$ to $W$
  - Set $V(0,j) = 0$

$\text{//complete remaining entries row by row}$

- Repeat for $i = 1$ to $n$
  - repeat for $j = 1$ to $W$
    - if ($w_i <= j$) $V(i,j) = \max\{ V(i-1,j), V(i-1,j-w_i) + v_i \}$
    - if ($w_i > j$) $V(i,j) = V(i-1,j)$
  - Print $V(n,W)$
5. Program Definition:
From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

AIM:
Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

Algorithm:
// Input: A weighted connected graph G ={V,E}, source s
// Output dv: the distance-vertex matrix
• Read number of vertices of graph G
• Read weighted graph G
• Print weighted graph
• Initialize distance from source for all vertices as weight between source node and other vertices, i, and none in tree

// initial condition
• For all other vertices,
  • dv[i] = wt_graph[s,i], TV[i]=0, prev[i]=0
  • dv[s] = 0, prev[s] = s // source vertex
• Repeat for y = 1 to n
  v = next vertex with minimum dv value, by calling FindNextNear()
  Add v to tree.
  For all the adjacent u of v and u is not added to the tree,
    if dv[u] > dv[v] + wt_graph[v,u]
      then dv[u]= dv[v] + wt_graph[v,u] and prev[u]=v.
findNextNear

//Input: graph, dv matrix
//Output: j the next nearest vertex
  o  minm = 9999
  o  For k =1 to n
      if k vertex is not selected in tree and
      if dv[k] < minm
        {  minm = dv [ k]
          j=k
        }
  o  return j

Analysis

Like Prim's algorithm, Dijkstra's algorithm runs in $O(|E|\log|V|)$ time.
6. Program Definition:

Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

Objective: Kruskal's Algorithm for computing the minimum spanning tree is directly based on the generic MST algorithm. It builds the MST in forest.

Kruskal's Algorithm

Start with an empty set A, and select at every stage the shortest edge that has not been chosen or rejected, regardless of where this edge is situated in graph.

- Initially, each vertex is in its own tree in forest.
- Then, algorithm consider each edge in turn, order by increasing weight.
- If an edge \((u, v)\) connects two different trees, then \((u, v)\) is added to the set of edges of the MST, and two trees connected by an edge \((u, v)\) are merged into a single tree.
- On the other hand, if an edge \((u, v)\) connects two vertices in the same tree, then edge \((u, v)\) is discarded.

Kruskals algorithm can be implemented using disjoint set data structure or priority queue data structure.

I. Kruskal's algorithm implemented with disjoint-sets data structure.

Disjoint-Sets Data Structure

- Make_Set \((v)\)
  Create a new set whose only member is pointed to by \(v\). Note that for this operation \(v\) must already be in a set.
- FIND_Set
  Returns a pointer to the set containing \(v\).
- UNION \((u, v)\)
  Unites the dynamic sets that contain \(u\) and \(v\) into a new set that is union of these two sets.
ADA LAB
MST_KRUSKAL (G, w)

1. A ← {} // A will ultimately contains the edges of the MST
2. for each vertex v in V[G]
   do Make_Set (v)
3. Sort edge of E by nondecreasing weights w
4. for each edge (u, v) in E
   do if FIND_SET (u) ≠ FIND_SET (v)
   then A = AU{(u, v)}
   UNION (u, v)
5. Return A

Analysis

The for-loop in lines 5-8 per forms 'Union' and 'find' operations in O(|E|) time. When the disjoint-set forest implementation with the weighted union and path compression heuristic is used, the O(|E|) 'union' and 'find' operators involved in the for-loop in lines 5-8 will have worst-case time O(|E| lg * |E|). Kruskal's algorithm requires sorting the edge of E in line 4.

We know that the fastest comparison-based sorting algorithm will need $\Theta(|E| \lg |E|)$ time to sort all the edges. The time for sorting all the edges of G in line 4 will dominates the time for the 'union' and 'find' operations in the for-loop in lines 5-8. When comparison-based sorting algorithm is used to perform sorting in line 4. In this case, the running time of Kruskal's algorithm will be O(|E| lg |E|).

- O(V) time to initialize.
- O(E lg E) time to sort edges by weight
- O(E α (E, V)) = O(E lg E) time to process edges.

If all edge weights are integers ranging from 1 to |V|, we can use COUNTING_SORT (instead of a more generally applicable sorting algorithm) to sort the edges in O(V + E) = O(E) time. Note that V = O(E) for connected graph. This speeds up the whole algorithm to take only O(E α (E, V)) time; the time to process the edges, not the time to sort them, is now the dominant term.
Knowledge about the weights won't help speed up any other part of the sort in line 4 uses the weight values.

If the edges weights are integers ranging from 1 to constant W, we can again use COUNTING_SORT, which again runs in \(O(W + E) = O(E)\) time. Note that \(O(W + E) = O(E)\) because \(W\) is a constant. Therefore, the asymptotic bound of the Kruskal's algorithm is also \(O(E \alpha (E, V))\).

II. Kruskal's algorithm implemented with priority queue data structure.

**MST_KRUSKAL (G)**

1. for each vertex \(v\) in \(V[G]\) do
2. define set \(S(v) \leftarrow \{v\}\)
3. Initialize priority queue \(Q\) that contains all edges of \(G\), using the weights as keys.
4. \(A \leftarrow \{\} \) // \(A\) will ultimately contain the edges of the MST
5. While \(A\) has less than \(n-1\) edges do
6. Let set \(S(v)\) contains \(v\) and \(S(u)\) contain \(u\)
7. IF \(S(v) \neq S(u)\) then
   - Add edge \((u, v)\) to \(A\)
   - Merge \(S(v)\) and \(S(u)\) into one set i.e., union
8. Return \(A\)

**Analysis**

- The edge weight can be compared in constant time.
- Initialization of priority queue takes \(O(E \log E)\) time by repeated insertion.
- At each iteration of while-loop, minimum edge can be removed in \(O(\log E)\) time, which is \(O(\log V)\), since graph is simple.
- The total running time is \(O((V + E) \log V)\), which is \(O(E \log V)\) since graph is simple and connected.
Example Step by Step operation of Kurskal's algorithm.

Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be representative. Let's choose vertex g arbitrarily.

Step 2. The edge (c, i) creates the second tree. Choose vertex c as representative for second tree.

Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as representative.

Step 4. Edge (a, b) creates a third tree.
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.

Step 6. Edge (g, i) is the next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.

Step 7. Instead, add edge (c, d).

Step 8. If we add edge (h, i), edge(h, i) would make a cycle.

Step 9. Instead of adding edge (h, i) add edge (a, h).
Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.

II Kruskal’s algorithm implemented with priority queue data structure.

MST_KRUSKAL (G)

1. for each vertex v in V[G] do
2.   define set S(v) ← {v}
3. Initialize priority queue Q that contains all edges of G, using the weights as keys.
4. A ← { } // A will ultimately contains the edges of the MST
5. While A has less than n-1 edges do
6.   Let set S(v) contains v and S(u) contain u
7.   IF S(v) ! S(u) then
      Add edge (u, v) to A
      Merge S(v) and S(u) into one set i.e., union
8. Return A
Analysis

- The edge weight can be compared in constant time.
- Initialization of priority queue takes $O(E \log E)$ time by repeated insertion.
- At each iteration of while-loop, minimum edge can be removed in $O(\log E)$ time, which is $O(\log V)$, since graph is simple.
- The total running time is $O((V + E) \log V)$, which is $O(E \log V)$ since graph is simple and connected.
7. Program Definition:

a. Print all the nodes reachable from a given starting node in a digraph using BFS method.

b. Check whether a given graph is connected or not using DFS method.

Breadth First Search algorithm used in

Objective: Like depth first search, BFS traverse a connected component of a given graph and defines a spanning tree.

Algorithm Breadth First Search

- BFS starts at a given vertex, which is at level 0.
- In the first stage, visit all vertices at level 1.
- In the second stage, we visit all vertices at second level.
- These new vertices, which are adjacent to level 1 vertices, and so on.
- The BFS traversal terminates when every vertex has been visited.

A queue data structure is used to keep track of vertex in each level.

Pseudo code:

**BREADTH FIRST SEARCH (G, S)**

**Input:** A graph G and a vertex.

**Output:** Edges labeled as discovery and cross edges in the connected component.

Create a Queue Q.
ENQUEUE (Q, S) // Insert S into Q.
While Q is not empty do
   for each vertex v in Q do
      for all edges e incident on v do
         if edge e is unexplored then
            let w be the other endpoint of e.
            if vertex w is unexpected then
ADA LAB
- mark e as a discovery edge
- insert w into Q
else
mark e as a cross edge

BFS label each vertex by the length of a shortest path (in terms of number of edges) from the start vertex.

Example

Step 1

Step 2

Step 3

Step 4
Step 5

Step 6

Step 7

Step 8

Step 9
Starting vertex (node) is S
Solid edge = discovery edge.
Dashed edge = error edge (since none of them connects a vertex to one of its ancestors).

As with the depth first search (DFS), the discovery edges form a spanning tree, which in this case its called the BSF-tree.

Applications of BFS

BSF is used to solve following problem

i) Testing whether graph is connected.

ii) Computing a spanning forest of graph.

iii) Computing, for every vertex in graph, a path with the minimum number of edges between start vertex and current vertex or reporting that no such path exists.

iv) Computing a cycle in graph or reporting that no such cycle exists.

Analysis

The lines added to BFS algorithm take constant time to execute and so the running time is the same as that of BFS which is O(V + E).

Breadth First Search algorithm used in
ADA LAB
  • Prim’s MST algorithm.
  • Dijkstra’s single source shortest path algorithm.

Like depth first search, BFS traverse a connected component of a given graph and defines a spanning tree.

**Depth first search (DFS)**

**Objective** : Depth first search (DFS) is useful for

- Find a path from one vertex to another
- Whether graph is connected or not
- Computing a spanning tree of a connected graph.

**Algorithm Depth First Search**

**Input** : Adjacency matrix G.

**Output** : spanning tree.

- Algorithm starts at a specific vertex S in G, which becomes current vertex.
- Algorithm traverse graph by any edge \((u, v)\) incident to the current vertex \(u\).
- If the edge \((u, v)\) leads to an already visited vertex \(v\), then backtrack to current vertex \(u\).
- If, on other hand, edge \((u, v)\) leads to an unvisited vertex \(v\), then go to \(v\) and \(v\) becomes current vertex.
- We proceed in this manner until we reach to "deadend".
- At this point start back tracking.
- The process terminates when backtracking leads back to the start vertex.
- In order to keep track of parent vertices Stack is used in DFS as soon as a vertex is visited it is pushed to stack.
- And while back tracking if there are no unvisited vertices connected to it the vertex is popped out.
Edger in DFS Tree/Graph

- Edges leads to new vertex are called discovery or tree edges
- Edges lead to already visited are called back edges.

Pseudo code

**DEPTH FIRST SEARCH (G, v)**

**Input:** A graph G and a vertex v.

**Output:** Edges labeled as discovery and back edges in the connected component.

For all edges e incident on v do

If edge e is unexplored then

\[ w \leftarrow \text{opposite}(v, e) \] // return the end point of e distant to v

If vertex w is unexplained then

- mark e as a discovery edge
- Recursively call DSF(G, w)

else

- mark e as a back edge

Example:

The following is an example where graph G has 6 vertices and 8 edges.

The numerator in each node implies to push value and denominator determines popped value.
Solid Edge = discovery or tree edge
Dashed Edge = back edge.

Each vertex has two time stamps:

- The **first** time stamp records when vertex is first discovered (push in to stack).
- The **second** time stamp records when the search finishes examining adjacency list of vertex (popped out of stack).
Applications of DFS algorithm

DFS can be used for the following...

Testing whether graph is connected.

Computing a spanning forest of graph.

Computing a path between two vertices of graph or equivalently reporting that no such path exists.

Computing a cycle in graph or equivalently reporting that no such cycle exists.

Analysis

The running time of DSF is $O(V + E)$.

Consider vertex $u$ and vertex $v$ in $V[G]$ after a DFS. Suppose vertex $v$ in some DFS-tree. Then we have $d[u] < d[v] < f[v] < f[u]$ because of the following reasons

1. Vertex $u$ was discovered before vertex $v$; and
2. Vertex $v$ was fully explored before vertex $u$ was fully explored.

Note that converse also holds: if $d[u] < d[v] < f[v] < f[u]$ then vertex $v$ is in the same DFS-tree and a vertex $v$ is a descendent of vertex $u$.

Suppose vertex $u$ and vertex $v$ are in different DFS-trees or suppose vertex $u$ and vertex $v$ are in the same DFS-tree but neither vertex is the descendent of the other. Then one vertex was discovered and fully explored before the other was discovered i.e., $f[u] < d[v]$ or $f[v] < d[u]$.

Consider a directed graph $G = (V, E)$. After a DFS of graph $G$ we can put each edge into one of four classes:

- A tree edge is an edge in a DFS-tree.
- A back edge connects a vertex to an ancestor in a DFS-tree. Note that a self-loop is a back edge.
- A forward edge is a nontree edge that connects a vertex to a descendent in a DFS-tree.
ADA LAB

- A cross edge is any other edge in graph G. It connects vertices in two different DFS-tree or two vertices in the same DFS-tree neither of which is the ancestor of the other.

Time Complexity

The maximum number of possible edges in the graph G if it does not have cycle is |V| - 1. If G has a cycles, then the number of edges exceeds this number. Hence, the algorithm will detects a cycle at the most at the Vth edge if not before it. Therefore, the algorithm will run in O(V) time.
8. Program Definition:

Find a subset of a given set \( S = \{s_1, s_2, \ldots, s_n\} \) of \( n \) positive integers whose sum is equal to a given positive integer \( d \). For example, if \( S = \{1, 2, 5, 6, 8\} \) and \( d = 9 \) there are two solutions\{1,2,6\} and \{1,8\}. A suitable message is to be displayed if the given problem instance doesn't have a solution.

AIM: An instance of the Subset Sum problem is a pair \((S, t)\), where \( S = \{x_1, x_2, \ldots, x_n\} \) is a set of positive integers and \( t \) (the target) is a positive integer. The decision problem asks for a subset of \( S \) whose sum is as large as possible, but not larger than \( t \).

This problem is NP-complete.

This problem arises in practical applications. Similar to the knapsack problem we may have a truck that can carry at most \( t \) pounds and we have \( n \) different boxes to ship and the \( i \)th box weighs \( x_i \) pounds. The naive approach of computing the sum of the elements of every subset of \( S \) and then selecting the best requires exponential time. Below we present an exponential time exact algorithm.

- Algorithm:
  - accept \( n \): no of items in set
  - accept their values, \( s_k \) in increasing order
  - accept \( d \): sum of subset desired
  - initialise \( x[i] = -1 \) for all \( i \)
  - check if solution possible or not
  - if possible then call \( \text{SumOfSub}(0,1,\text{sum of all elements}) \)
• **SumOfSub** (s, k, r)

//Values of x[j], 1 <= j < k, have been determined
//Node creation at level k taking place: also call for creation at level K+1 if possible
// s = sum of 1 to k-1 elements and r is sum of k to n elements
//generating left child that means including k in solution

• Set x[k] = 1

• If (s + s[k] = d) then subset found, print solution

• If (s + s[k] + s[k+1] <= d)

  then **SumOfSum** (s + s[k], k+1, r – s[k])

//Generate right child i.e. element k absent

• If (s + r - s[k] >= d) AND (s + s[k+1] )<=d

  THEN { x[k]=0;
  **SumOfSub** (s, k+1, r – s[k])
9. **Program Definition:**

Implement any scheme to find the optimal solution for the Travelling Salesperson problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation.

**TSP Approximation Algorithm (Known as Christofides Heuristics)**

**Algorithm:**

1. Compute MST graph T.
2. Compute a minimum-weighted matching graph M.
3. Combine T and M as edge set and Compute an Euler Cycle.
4. Traverse each vertex taking shortcuts to avoid visited nodes.

- **What is a Minimum-weighted Matching?**
  - It creates a MWM on a set of the nodes having an odd degree.
- **Why odd degree?**
  - Property of Euler Cycle
- **Why 1.5 TSP?**
  - MST < Euler Cycle = MWM+MST <= 1.5 TSP
  - (MWM = ½ MST)
10. **Program Definition:**

10. Find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm.

**Objective:** Like Kruskal's algorithm, Prim's algorithm is based on a generic MST algorithm.

**Algorithm**

Choose a node and build a tree from there selecting at every stage the shortest available edge that can extend the tree to an additional node.

- Prim's algorithm has the property that the edges in the set A always form a single tree.
- We begin with some vertex \( v \) in a given graph \( G = (V, E) \), defining the initial set of vertices A.
- In each iteration, we choose a minimum-weight edge \((u, v)\), connecting a vertex \( v \) in the set A to the vertex \( u \) outside of set A.
- The vertex \( u \) is brought in to A. This process is repeated until a spanning tree is formed.
- Like Kruskal's algorithm, here too, the important fact about MSTs is we always choose the smallest-weight edge joining a vertex inside set A to the one outside the set A.
- The implication of this fact is that it adds only edges that are safe for A; therefore when the algorithm terminates, the edges in set A form a MST

**Pseudo code**

MST_PRIM (G, \( w \), \( v \))

1. \( Q \leftarrow V[G] \)
2. for each \( u \) in Q do
3. \( \text{key} [u] \leftarrow \infty \)
4. \( \text{key} [r] \leftarrow 0 \)
5. \( \pi[r] \leftarrow \text{Nil} \)
6. while queue is not empty do
7. \( u \leftarrow \text{EXTRACT_MIN} (Q) \)
8. for each \( v \) in Adj\([u]\) do
9. if \( v \) is in Q and \( w(u, v) < \text{key} [v] \)
Analysis

The performance of Prim’s algorithm depends on how we choose to implement the priority queue $Q$.

```latex
10. \text{then } \pi[v] \leftarrow w(u, v)
11. \text{key } [v] \leftarrow w(u, v)
```
11. Program Definition:

Implement All-Pairs Shortest Paths Problem using Floyd’s algorithm. Parallelize this algorithm, implement it using OpenMP and determine the speed-up achieved.

AIM:

The Floyd–Warshall algorithm (sometimes known as the WFI Algorithm or Roy–Floyd algorithm) is a graph analysis algorithm for finding shortest paths in a weighted graph (with positive or negative edge weights). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices though it does not return details of the paths themselves. The algorithm is an example of dynamic programming.

Floyd’s Algorithm

- Accept no. of vertices
- Call graph function to read weighted graph // w(i,j)
- Set D[ ] <- weighted graph matrix // get D {d(i,j)} for k=0
- // If there is a cycle in graph, abort. How to find?
- Repeat for k = 1 to n
  - Repeat for i = 1 to n
    - Repeat for j = 1 to n
      - D[i,j] = min {D[i,j], D[i,k] + D[k,j]}
  - Print D
12. **Program Definition:**

Implement N Queen's problem using Back Tracking.

**AIM:**

The object is to place queens on a chess board in such a way as no queen can capture another one in a single move

- Recall that a queen can move horz, vert, or diagonally an infinite distance
  
  - This implies that no two queens can be on the same row, col, or diagonal
- We usually want to know how many different placements there are

- Using Backtracking Techniques

**Algorithm:**

/* outputs all possible acceptable positions of n queens on n x n chessboard */

// Initialize x [ ] to zero

// Set k = 1 start with first queen

Repeat for i = 1 to n // try all columns one by one for k

  - if Place (k, i) true then

    { x(k) = i // place k\textsuperscript{th} queen in column i
      
      if (k=n) **all queens placed and hence print output** (x[ ])
      
      else NQueens(K+1,n) //try for next queen

    }

**4-Queens**

![Diagram of 4-Queens placement](image-url)
• Place (k,i)
/* finds if k\textsuperscript{th} queen in k\textsuperscript{th} row can be placed in column i or not; returns true if queen can be placed */
// x[1,2, . . . k-1] have been defined
//queens at (p, q) & (r, s) attack if |p-r| = |q-s|
• Repeat for j = 1 to (k-1)
    if any earlier j\textsuperscript{th} queen is in i\textsuperscript{th} column ( x[j]= i)
or in same diagonal ( abs(x[ j] - i) = abs( j - k ) )
    then k\textsuperscript{th} queen cannot be placed (return false)
• return true (as all positions checked and no objection)