GH Raisoni College of Engineering and Management
Wagholi, Pune

Department of Computer Engineering

Third Year Bachelor of Technology

DESIGN AND ANALYSIS OF ALGORITHMS (BCOP1931)

Lab Manual

Prepared By
Faculty Name: Mrs. Mansi Bhonsle

G. H. Raisoni College of Engineering and Management, Wagholi, Pune 412207

Department: Computer Engineering

Course Details

<u>Subject: Design & Analysis of Algorithm Lab (BCOP19312)</u>

CLASS: TY BTECH **DIVISION: A& B**

Internal Marks: 25 **External marks: Nil**

COURSE OUTCOME	
CO1	To develop problem-solving abilities using mathematical theories.
CO2	To analyze the performance of algorithms with time complexity.
CO3	To apply algorithmic design strategies with optimization problems.
CO4	To solve real-life problems with various designs of algorithms.

List of Experiments

Sr. No.	List of Laboratory Assignments	CO Mapping	Software Required
1	Write a C++ program to find factorial of a given number using (i)Recursion (ii) Iteration Compare the time and space complexity of both the designs.	CO1, CO2	CodeBlock
2	Implement a Binary search program with a Divide and Conquer design strategy for n numbers using C++. Discuss Best, Average, and Worst time complexity.	CO1, CO3	CodeBlock
3	Sort a given set of n integer elements using the QuickSort method and compute its time complexity. Run the program for varied values of n and record the time taken to sort. The elements can be read by a user or can be generated using the random number generator. Demonstrate using C++ how the divide and conquer method works along with its time complexity analysis: worst case, average case, and best case.	CO2, CO3	CodeBlock
4	A business house has several offices in different countries; they want to lease phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. The business house wants to connect all its offices with a minimum total cost. Solve the problem by suggesting appropriate data structures in C++.	CO2, CO3	CodeBlock
5	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. Write the program in C++.	CO2, CO3	CodeBlock
6	Implement a program in C++ for the 0/1 Knapsack problem using the Dynamic Programming method.	CO3	CodeBlock
7	Write C++ program to implement Travelling Sales Person problem using Dynamic programming.	CO3, CO4	CodeBlock
8	Implement a C++ program for solving N-Queen's problem using Backtracking. (Assume N=4)	CO3, CO4	CodeBlock

9	Implement Travelling salesman problem using branch and bound approach using C++.	CO4	CodeBlock
10	Write C++ Program to demonstrate the implementation of the Rabin-Karp Algorithm with a discussion of time complexity.	CO4	CodeBlock
	Content beyond syllabus		
11	Matrix Chain Multiplication using Dynamic Programming	CO4	CodeBlock
12	Optimization Algorithms for complexity problems	CO3	CodeBlock
	Conduction of Practical :		
	1. Every student has to execute practicals with proper inputs and outputs. 2. Every student has to write roll no and name on the output file. 3. Report must show on the same day of execution of programs. 4. Marks distribution: Program + Conduction + Viva+ I/O = 10, Tools=10, File=5		

Experiment No. 1

Aim: Write a C++ program to find the factorial of a given number using (i) Recursion and (ii) Iteration. Compare the time and space complexity of both the designs.

Theory: Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function. a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function; otherwise, it will go into an infinite loop. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

A function that calls itself is known as a recursive function and this technique is known as recursion. A recursion instruction continues until another instruction prevents it. Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function α either calls itself directly or calls a function β that in turn calls the original function α. The function α is called the recursive function.

Example − a function calling itself.

```
int function(int value) {
  if(value < 1)
    return;
  function(value - 1);
  printf("%d ",value);
}
```

Properties

A recursive function can go infinite like a loop. To avoid infinite running of a recursive function, there are two properties that a recursive function must have −

Base criteria − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.

Progressive approach − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Implementation

Many programming languages implement recursion by means of stacks. Generally, whenever a function (caller) calls another function (callee) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies that the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.

Activation Records

This activation record keeps the information about local variables, formal parameters, return address, and all information passed to the caller function.

Analysis of Recursion

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of the latest enhanced CPU systems, recursion is more efficient than iterations.

Time Complexity

In the case of iterations, we take a number of iterations to count the time complexity. Likewise, assuming everything is constant, we try to figure out the number of times a recursive call is being made, in case of recursion. A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

Space Complexity

Space complexity is counted as what amount of extra space is required for a module to execute. In the case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in the case of recursion, the system needs to store an activation record each time a recursive call is made. Hence, it is considered that the space complexity of a recursive function may go higher than that of a function with iteration.

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code. We can write such codes iteratively with the help of stack data structure.

Note that both recursive and iterative programs have problem-solving, i.e., every recursive program can be written iteratively and vice versa is also true. A recursive program has greater space requirements than an iterative program as all functions will remain in the stack until the base case is reached. Also, it has greater time requirements because of function calls and returns overhead.

Pseudocode for Iteration method:

```
function binarySearch(a, value, left, right)
    while left ≤ right
        mid := floor((right-left)/2)+left
        if a[mid] = value
            return mid
        if value < a[mid]
            right := mid-1
        else
            left  := mid+1
    return not found
```

Pseudocode for Recursive method:

```
function binarySearch(a, value, left, right)
    if right < left
        return not found
    mid := floor((right-left)/2)+left
    if a[mid] = value
        return mid
    if value < a[mid]
        return binarySearch(a, value, left, mid-1)
    else
        return binarySearch(a, value, mid+1, right)
}
```

Program:Factorial using recursive function

```
#include<iostream>
using namespace std;

int factorial(int n);

int main()
{
    int n;

    cout << "Enter a positive integer: ";
    cin >> n;

    cout << "Factorial of " << n << " = " << factorial(n);

    return 0;
}

int factorial(int n)
{
   if(n > 1)
      return n * factorial(n - 1);
   else
      return 1;
}
```

Output:

```
Enter a positive integer: 5
Factorial of 5 = 120
```

Program: factorial using iterative function

```
#include <iostream>
using namespace std;
int fact_iter(int n)
{
   int result = 1;
   for (int i = 1; i <= n; i++)
   {
      result *= i;
   }
   return result;
}
int main()
{
   int n;
   while (1)
   {
        cout<<"Enter interger (0 to exit): ";
        cin>>n;
        if (n == 0)
            break;
        cout<<fact_iter(n)<<endl;
    }
    return 0;
}
```

Output:

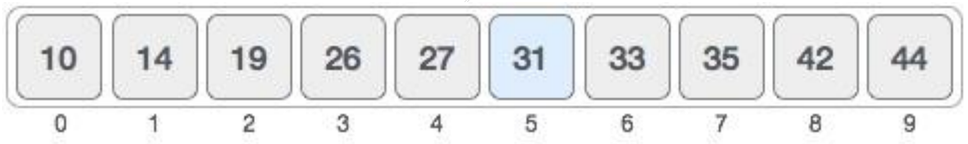
```
Enter integer (0 to exit): 5
120
Enter integer (0 to exit): 4
24
Enter integer (0 to exit): 3
6
```

Conclusion: Thus, we have studied factorial programs with recursive and iterative design strategies. The space complexity is slightly increased in recursive functions with the inclusion of an internal stack in execution.

Experiment No. 2

Aim: Implement a Binary Search program with a Divide and Conquer design strategy for n numbers using C++.

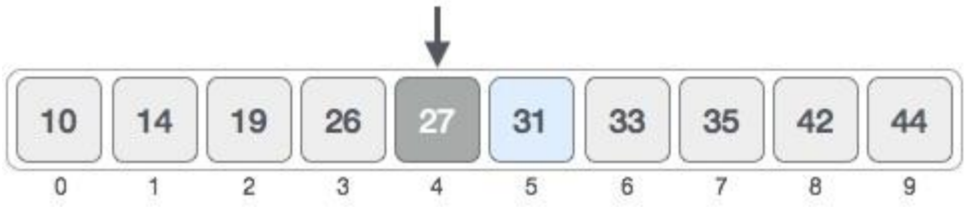
Theory: A **binary search algorithm** is a technique for locating a particular value in a sorted list. The method makes progressively better guesses, and closes in on the location of the sought value by selecting the middle element in the span (which, because the list is in sorted order, is the median value), comparing its value to the target value, and determining if it is greater than, less than, or equal to the target value. A guessed index whose value turns out to be too high becomes the new upper bound of the span, and if its value is too low that index becomes the new lower bound. Only the sign of the difference is inspected: there is no attempt at an interpolation search based on the size of the difference. Pursuing this strategy iteratively, the method reduces the search span by a factor of two each time, and soon finds the target value or else determines that it is not in the list at all. A binary search is an example of a dichotomic divide and conquer search algorithm. For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



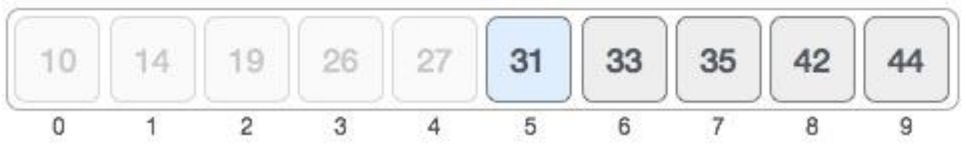
First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

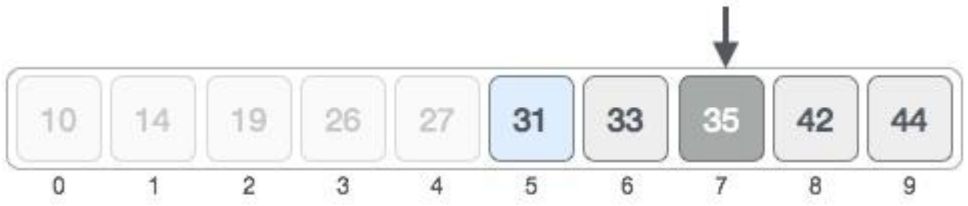


We change our low to mid + 1 and find the new mid value again.

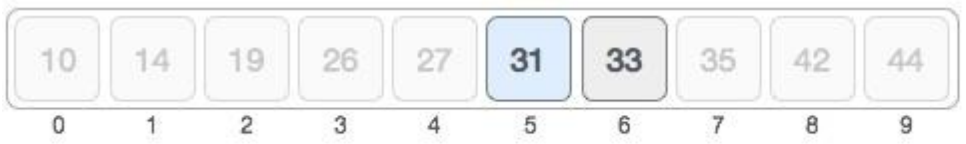
low = mid + 1

mid = low + (high - low) / 2

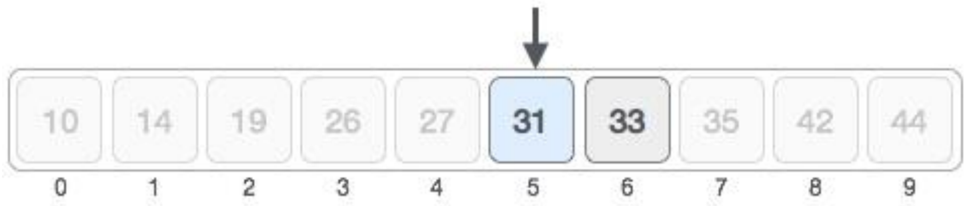
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



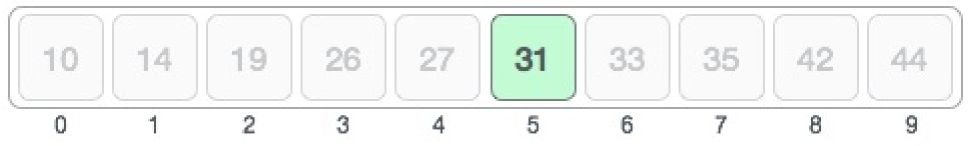
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very fewer numbers.

Algorithm: Iterative

```
BS(A,item)
{
low=0;
high=n-1;
while(low<=high)
{
mid=(low+high)/2
if(A[mid]>value)
high=mid-1
else if(A[mid]<value)
low=mid+1
else
return mid;
}
}
```

Algorithm: Recursive

```
BS(A,item,low,high)
{
If(high<low)
Return -1
Mid=(low+high)/2
if(A[mid]>item)
return BS(A,item,low,mid-1)
else if(A[mid]<item)
return BS(A,item,mid+1,high)
else
return mid
}
```

Time Complexity T(n)= O(log n)

Space Complexity S(n)= O(n)

Program: Iterative

```
/* C++ Program - Binary Search */
#include<iostream>
using namespace std;
int main()
{
        int n, i, arr[50], search, first, last, middle;
        cout<<"Enter total number of elements :";
        cin>>n;
        cout<<"Enter "<<n<<" number :";
        for (i=0; i<n; i++)
        {
                cin>>arr[i];
        }
        cout<<"Enter a number to find :";
cin>>search;
first = 0;
last = n-1;
middle = (first+last)/2;
while (first <= last)
{
        if(arr[middle] < search)
        {
                first = middle + 1;

        }
        else if(arr[middle] == search)
        {
                cout<<search<<" found at location "<<middle+1<<"\n";
                break;
        }
        else
        {
                 last = middle - 1;
        }
        middle = (first + last)/2;
}
if(first > last)
{
        cout<<"Not found! "<<search<<" is not present in the list.";
}
return 0;
}
```

Output:

```
Enter the total number of elements: 5
Enter 5 numbers: 2
4
6
8
9
Enter a number to find: 8
8 found at location 4
```

Program: Recursive

```
#include <iostream>
using namespace std;
int bs(int a[],int l,int r,int x)
{
 if(r>=l)
 {
  int m=l+(r-l)/2;
  if(a[m]==x)
  return m;
  else if(a[m]>x)
  return bs(a,l,m-1,x);
  else
  return bs(a,m+1,r,x);
 }
 return -1;
}
int main()
{
   int a[10],n,i,key,res;
```

```
    cout << "\nEnter size of array" << endl;
    cin>>n;
    cout<<"\nEnter the Array in Ascending order::\t";
    for(i=0;i<n;i++)
    {
     cin>>a[i];
    }
    cout<<"\nEntered Array Is ::\t";
    for(i=0;i<n;i++)
    {
      cout<<"\n";
      cout<<a[i];
    }
    cout<<"\nEnter the element to be searched::\t";
    cin>>key;
    res=bs(a,0,n-1,key);
    if(res==-1)
    cout<<"\nElement not found  ";
    else
    cout<<"\nElement found at position "<<res;
    return 0;
}
```

Output:

```
Enter size of array
5
Enter the Array in Ascending order:    2
4
6
7
```

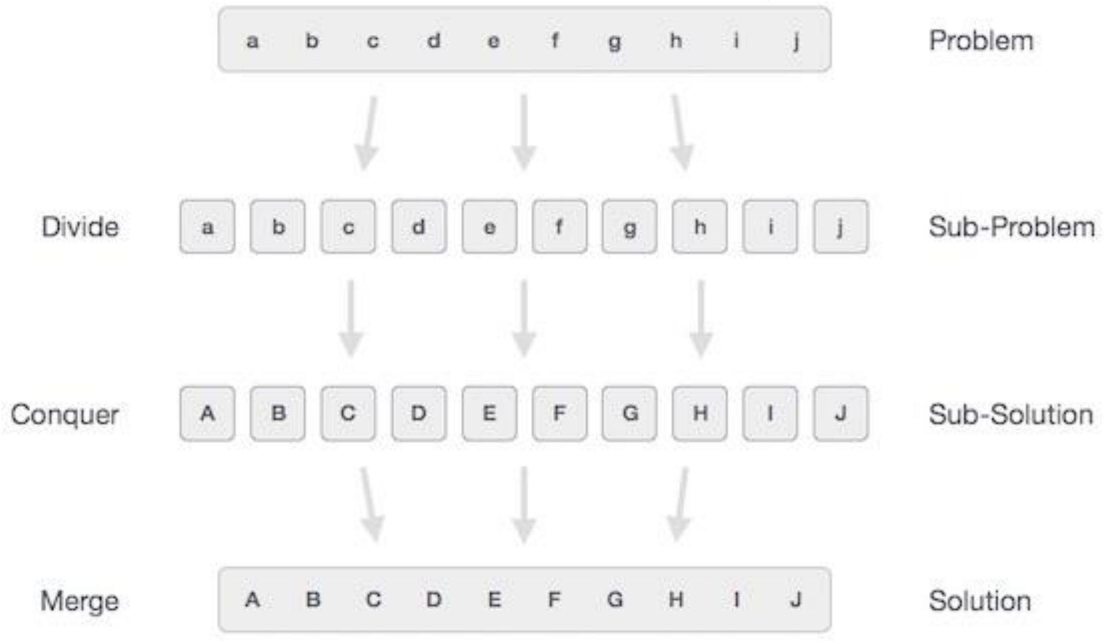
```
9
Entered Array Is:
2
4
6
7
9
Enter the element to be searched:      5
Element not found
```

Conclusion: Thus, we have studied and implemented a binary search program using the divide and conquer strategy. The best time complexity is made when the middle elements as search elements. The time complexity of the best case is O(1).

Experiment No. 3

Aim: Sort a given set of n integer elements using the Quicksort method and compute its time complexity. Run the program for varied values of n and record the time taken to sort. The elements can be read by a user or can be generated using the random number generator. Demonstrate using C++. How the divide and conquer method works along with its time complexity analysis: worst case, average case, and best case.

Theory: In the divide and conquer approach, the problem at hand, is divided into smaller sub-problems, and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



Broadly, we can understand the divide-and-conquer approach in a three-step process.

- **Divide/Break**

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

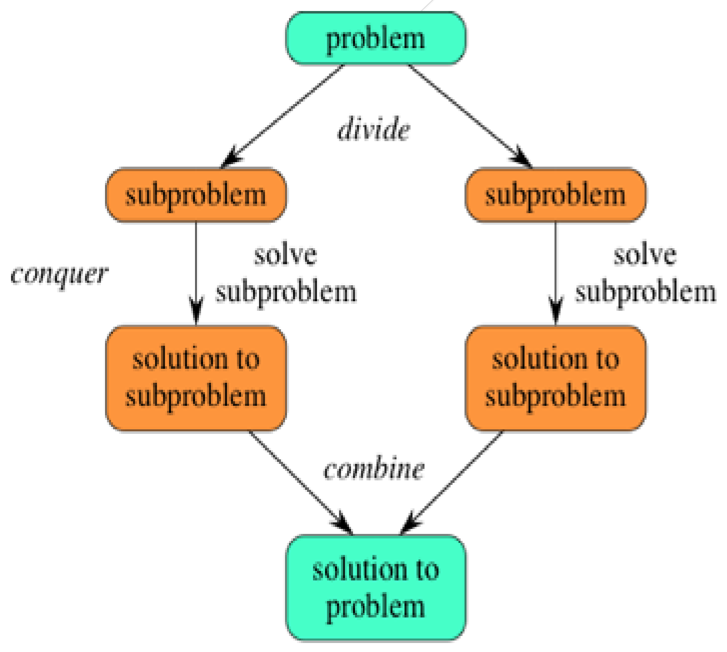
- **Conquer/Solve**

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

- **Merge/Combine**

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution to the original problem. These algorithmic approaches work recursively and conquer & merge steps work so close that they appear as one.

You can easily remember the steps of a divide-and-conquer algorithm as *divide, conquer, and combine*. Here's how to view one step, assuming that each divide step creates two subproblems (though some divide-and-conquer algorithms create more than two):



Advantages:

- Solving difficult problems

- Algorithm efficiency
- Parallelism
- Memory access
- Roundoff control

Program: Quicksort is a highly efficient sorting algorithm and is based on the partitioning of an array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), where n is the number of items.

Analysis of Quicksort:

Time taken by Quicksort, in general, can be written as follows.

T(n) = T(k) + T(n-k-1) + \theta(n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the pivot.

The time taken by Quicksort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks the greatest or smallest element as a pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for the worst case. T(n) = T(0) + T(n-1) + \theta(n)

Best Case: The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.

T(n) = 2T(n/2) + \theta(n)

The solution for the above recurrence is \theta (nLogn). It can be solved using case 2 of the Master Theorem.

Average Case:

To do an average case analysis, we need to consider all possible permutations of the array and calculate the time taken by every permutation which doesn't look easy.

Algorithm:

```
Quicksort(A,P,r)
1.if P<r
2.then q<-partition(A,P,r)
3. Quicksort(A,P,q)
4. Quicksort(A,q+1,r)
Algorithm: partition(A,P,r)
{
X<-A[i]
i<-P-1
j<-r+1
while(true)
repeat j=j-1 until A[j]<=x
repeat i-i+1 until A[j]>=x
if (i<j)
then exchange A[i]<->A[j]
else
return j
```

Program:

```
 #include <iostream>
 using namespace std;
 void quick_sort(int[],int,int);
int partition(int[],int,int);
 int main()
{
    int a[50],n,i;
    cout<<"How many elements?";
    cin>>n;
    cout<<"\nEnter array elements:";
     for(i=0;i<n;i++)
        cin>>a[i];

    quick_sort(a,0,n-1);
    cout<<"\nArray after sorting:";
    for(i=0;i<n;i++)
        cout<<a[i]<<" ";
     return 0;
}
 void quick_sort(int a[],int l,int u)
{
    int j;
    if(l<u)
    {
        j=partition(a,l,u);
        quick_sort(a,l,j-1);
        quick_sort(a,j+1,u);
    }
}
int partition(int a[],int l,int u)
```

```
{
    int v,i,j,temp;
    v=a[l];
    i=l;
    j=u+1;
    do
    {
        do
            i++;
        while(a[i]<v&&i<=u);
        do
            j--;
        while(v<a[j]);
        if(i<j)
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }while(i<j);
    a[l]=a[j];
    a[j]=v;
    return(j);
}
```

Output:

```
How many elements?5
Enter array elements:9
```

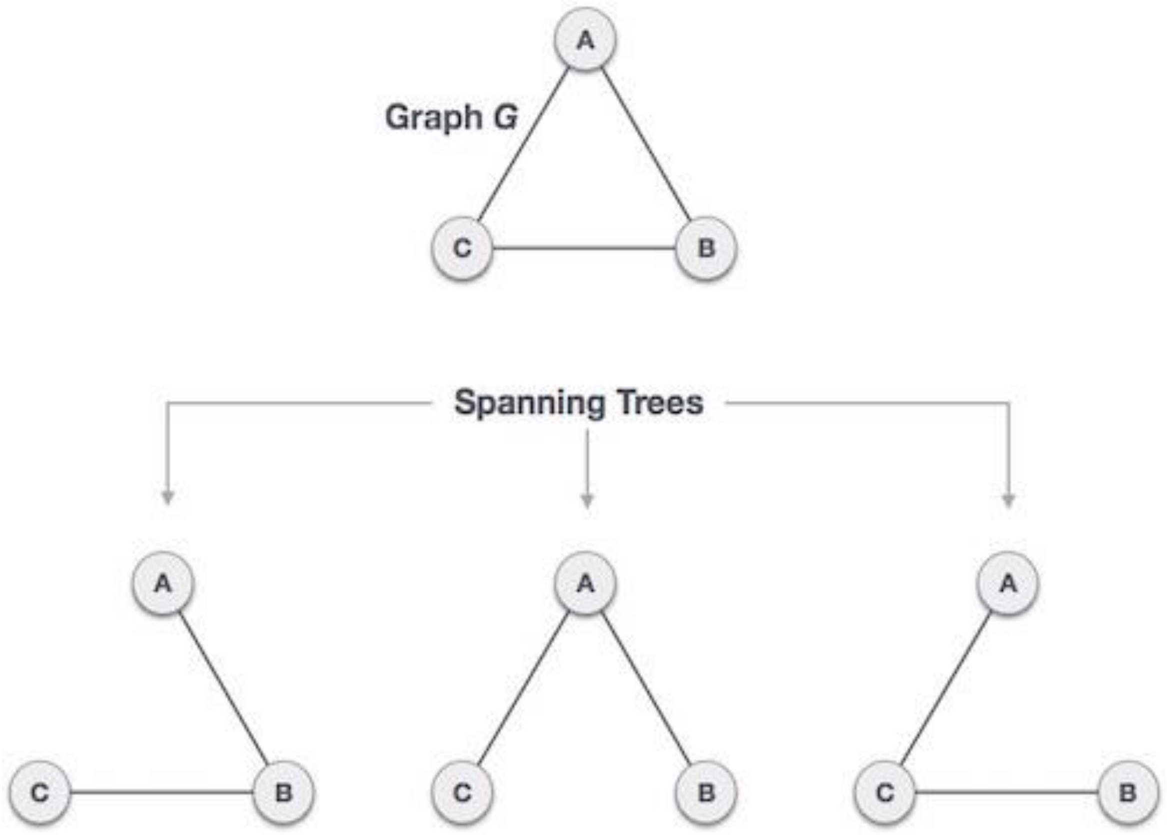
```
3
8
4
6
Array after sorting: 3 4 6 8 9
```

Conclusion: Thus, we have studied and implemented quick sorting using the divide and conquer strategy. The pivot element is responsible for partition types of quicksort. The best and average time complexity is O(nlogn). For an unbalanced partition, the worst-case time complexity is O (n^2).

Experiment No. 4

Aim: A business house has several offices in different countries; they want to lease phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. The business house wants to connect all its offices with a minimum total cost. Solve the problem by suggesting appropriate data structures in C++.

Theory: A spanning tree is a subset of Graph G, which has all the vertices covered with the minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected. By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have a maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above-addressed example, n is 3, hence $3^{3-2} = 3$spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.
-

Mathematical Properties of Spanning Tree:

- Spanning tree has n-1 edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have a maximum n^{n-2} number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G, and disconnected graphs do not have spanning trees.

Application of Spanning Tree:

A spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common applications of spanning trees are −

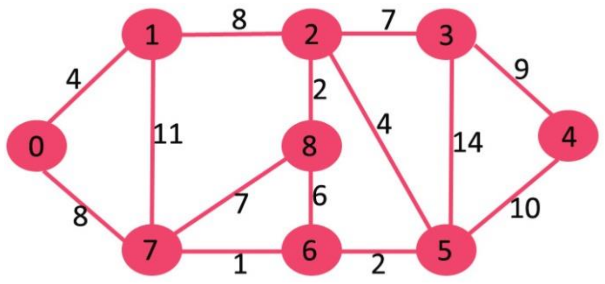
- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

Prim's Minimum Spanning Tree (MST)

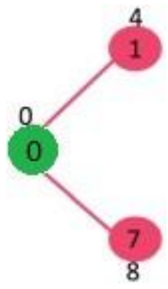
Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.
A group of edges that connects two sets of vertices in a graph is called cut in graph theory. *So, at* every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and the other contains the rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim's Algorithm Work? The idea behind Prim's algorithm is simple; a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

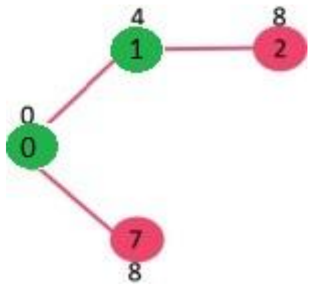
Let us understand with the following example:



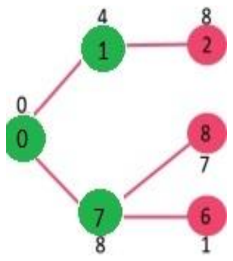
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with the minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



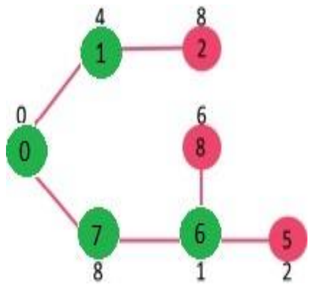
Pick the vertex with minimum key value and not already included in MST (not in mstSet). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



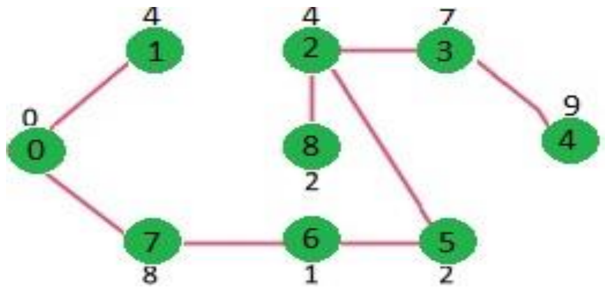
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until *mstSet* includes all vertices of a given graph. Finally, we get the following graph.



Algorithm:

1) Create a set *mstSet* that keeps track of vertices already included in MST.

2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.

3) While mstset doesn't include all vertices

  a) Pick a vertex *u* which is not there in *mstSet* and has a minimum key value.

  b) Include *u* to mstset

  c) Update the key value of all adjacent vertices of *u*.

To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if the weight of edge *u*-*v* is less than the previous key value of *v*, update the key value as the weight of *u*-*v*.

Program:

```
#include <iostream>
using namespace std;
int i,j,k,a,b,v,u,n,ne=1;
int low,mincost=0,cost[10][10];
int visited[10]={0};
int main()
{
    cout<<"Prims algorithm\n";
    cout<<"\nEnter number of vertices:\t";
    cin>>n;
    cout<<"\nEnter the adjacency matrix:\n";
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            cin>>cost[i][j];
            if(cost[i][j]==0)
                cost[i][j]=999;
        }
    }
    visited[1]=1;
   // printf("\nThe edges of Minimum Cost spanning tree are:\t");
    while(ne<n)
    {
        for(i=1,low=999;i<=n;i++)
        {
            for(j=1;j<=n;j++)
            {
                if(cost[i][j]<low)
                {
                    if(visited[i]!=0)
                    {
                    low=cost[i][j];
                    a=u=i;
                    b=v=j;
                    }
                }
            }
        }
        if(visited[u]==0||visited[v]==0)
        {
            cout<<"\n edge cost= "<<low;
            mincost+=low;
            visited[b]=1;
        }
        cost[a][b]=cost[b][a]=999;
    }
    cout<<"\nMinimum Cost= "<<mincost;
    return 0;
}
```

Output:

```
Enter the number of vertices in the graph: 4
Enter the number of edges in the graph: 5
Enter 2 vertices and weight of edge:
First vertex: 0
Second vertex: 1
Weight: 10
Enter 2 vertices and weight of edge:
First vertex: 0
Second vertex: 2
```

```
Weight: 6
Enter 2 vertices and weight of edge:
First vertex: 0
Second vertex: 3
Weight: 5
Enter 2 vertices and weight of edge:
First vertex: 1
Second vertex: 3
Weight: 15
Enter 2 vertices and weight of edge:
First vertex: 2
Second vertex: 3
Weight: 4

Edge (2, 3) with weight 4 is selected
Edge (0, 3) with weight 5 is selected
Edge (0, 2) with weight 6 is discarded
Edge (0, 1) with weight 10 is selected
```

Conclusion: Thus, we have studied and implemented a minimum spanning tree using the Greedy strategy. Kruskal’s Algorithm gives a unique minimum spanning tree for a given graph.

Experiment No. 5

Aim: From a given vertex in a weighted connected graph, find the shortest paths to other vertices using Dijkstra's algorithm. Write the program in C++.

Theory: Among all the algorithmic approaches, the simplest and most straightforward approach is the Greedy method. In this approach, the decision is taken on the basis of currently available information without worrying about the effect of the current decision in the future.
Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. The greedy method is easy to implement and quite efficient in most cases. Hence, we can say that the greedy algorithm is an algorithmic paradigm based on a heuristic that follows local optimal choice at each step with the hope of finding the global optimal solution.
In many problems, it does not produce an optimal solution though it gives an approximate (near-optimal) solution in a reasonable time.
Components of Greedy Algorithm
Greedy algorithms have the following five components −

- A candidate set − A solution is created from this set.
- A selection function − Used to choose the best candidate to be added to the solution.
- A feasibility function − Used to determine whether a candidate can be used to contribute to the solution.
- An objective function − Used to assign a value to a solution or a partial solution.
- A solution function − Used to indicate whether a complete solution has been reached.
-

Areas of Application

Greedy approach is used to solve many problems, such as

- Finding the shortest path between two vertices using Dijkstra's algorithm.
- Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.

Dijkstra’s Algorithm:

Dijkstra’s algorithm solves the single-source shortest-paths problem on a directed weighted graph G = (V, E), where all the edges are non-negative (i.e., $w(u, v) \geq 0$ for each edge $(u, v) \in E$).

In the following algorithm, we will use one function Extract-Min(), which extracts the node with the smallest key.

Analysis

The complexity of this algorithm is fully dependent on the implementation of the Extract-Min function. If the extract min function is implemented using linear search, the complexity of this algorithm is $O(V^2 + E)$. In this algorithm, if we use the min-heap on which the Extract-Min() function works to return the node from Q with the smallest key, the complexity of this algorithm can be reduced further.

Dijkstra’s algorithm is very similar to trees. Like Prim’s MST, we generate a SPT (shortest-path tree) with a given source as the root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex which is in the other set (set not yet included) and has a minimum distance from the source. Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices

Algorithm:

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign the distance value as 0 for the source vertex so that it is picked first.

3) While sptSet doesn’t include all vertices

a) Pick a vertex u which is not there in sptSet and has a minimum distance value.

b) Include u to sptSet.

c) Update the distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of the distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Program:

```
#include<iostream>

#include<conio.h>

#include<stdio.h>

using namespace std;

int shortest(int ,int);

int cost[10][10],dist[20],i,j,n,k,m,S[20],v,totcost,path[20],p;

main()

{

int c;

cout <<"enter no of vertices";

cin >> n;

cout <<"enter no of edges";

cin >>m;

cout <<"\nenter\nEDGE Cost\n";

for(k=1;k<=m;k++)
```

```
{
cin >> i >> j >>c;
cost[i][j]=c;
}
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
if(cost[i][j]==0)
cost[i][j]=31999;
cout <<"enter initial vertex";
cin >>v;
cout << v<<"\n";
shortest(v,n);
 }
 int shortest(int v,int n)
{
int min;
for(i=1;i<=n;i++)
{
S[i]=0;
dist[i]=cost[v][i];
}
path[++p]=v;
```

```
S[v]=1;

dist[v]=0;

for(i=2;i<=n-1;i++)

{

k=-1;

min=31999;

for(j=1;j<=n;j++)

{

if(dist[j]<min && S[j]!=1)

{

min=dist[j];

k=j;

}

}

if(cost[v][k]<=dist[k])

p=1;

path[++p]=k;

for(j=1;j<=p;j++)

cout<<path[j];

cout <<"\n";

//cout <<k;

S[k]=1;
for(j=1;j<=n;j++)
if(cost[k][j]!=31999 && dist[j]>=dist[k]+cost[k][j] && S[j]!=1)
 dist[j]=dist[k]+cost[k][j];
}
}
```

OUTPUT

```
enter no of vertices6
enter no of edges11
enter
EDGE Cost
1 2 50
1 3 45
1 4 10
2 3 10
2 4 15
3 5 30
4 1 10
4 5 15
5 2 20
5 3 35
6 5 3
enter initial vertex1
```

```
1

1 4

1 4 5

1 4 5 2

1 4 5 2 3
```

Conclusion: Thus, we have studied and implemented Dijkstra’s Algorithm using the Greedy strategy. The algorithm only works on positive numbers only. Also, if the node is not connected then it will not reachable by any means of traversing.

Experiment No. 6

Aim: Implement in C++, the 0/1 Knapsack problem using the Dynamic Programming method.

Theory: Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike divide and conquer, these sub-problems are not solved independently. Rather, the results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, the dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions to sub-problems are combined in order to achieve the best solution.

So we can say that −

- The problem should be able to be divided into smaller overlapping sub-problem.
- An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Dynamic algorithms use memorization.

Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated by the overall optimization of the problem.

In contrast, to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use memorization to remember the output of already solved sub-problems. Dynamic Programming is also used in optimization problems. Like the divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, the Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

Overlapping Sub-Problems

Similar to the Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exist.

For example, Binary Search does not have an overlapping sub-problem. Whereas the recursive program of Fibonacci numbers has many overlapping sub-problems.

Optimal Sub-Structure

A given problem has Optimal Substructure Property if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

For example, the Shortest Path problem has the following optimal substructure property −

If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

Steps of Dynamic Programming Approach

The dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

A thief is robbing a store and can carry a maxᵢmal weight of ***W*** into his knapsack. There are ***n*** items and weight of **ith** item is w_i and the profit of selecting this item is p_i. What items should the thief take?

Dynamic-Programming Approach

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution ***S*** is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maxᵢmum weight **w**.

The algorithm takes the following inputs

- The maxᵢmum weight **W**
- The number of items **n**
- The two sequences $v = <v_1, v_2, \ldots, v_n>$ and $w = <w_1, w_2, \ldots, w_n>$

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

Program: Knapsack problem

```
#include <iostream>

using namespace std;

int knaps(int n,int m,int w[],int p[])

{

    int i,j;

    int knapsack[n+1][m+1];

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

        knapsack[0][j]=0;

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

        knapsack[i][0]=0;

    for(i=1;i<=n;i++)

    {

        for(j=1;j<=m;j++)

        {

            if(w[i-1]<=j)

                knapsack[i][j]=max(knapsack[i-1][j],p[i-1]+knapsack[i-1][j-w[i-1]]);

            else

                knapsack[i][j]=knapsack[i-1][j];

        }

    }
    return knapsack[n][m];
}
int main()
{
   int i,j,n,m;
   cout<<"\nEnter number of item:\t";
   cin>>n;
   int w[n];
   int p[n];
   cout<<"\nEnetr weight & price of items:\t";
   for(i=0;i<n;i++)
   {
       cin>>w[i]>>p[i];
   }
   cout<<"\nEnter capacity of knapsack:\t";
   cin>>m;
   int result=knaps(n,m,w,p);
   cout<<"\nMaximum value that can be stored is:\t"<<result;
    return 0;
}
```

Output:

```
Enter number of item:   3

Enter weight & price of items:

2 1

3 2

4 5

Enter capacity of knapsack:     6

The maximum value that can be stored is:    6
```

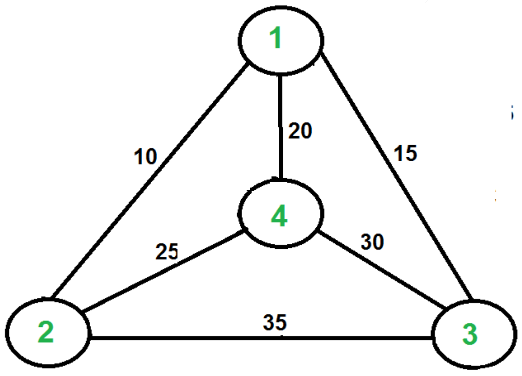
Conclusion: Thus, we have studied and implemented the 0-1 knapsack problem using Dynamic Programming. The algorithm may be solved by the table format method also but take extra memory.

Experiment No. 7

Aim: Write a C++ program to implement the Travelling Sales Person problem using Dynamic programming.

Theory: Travelling Salesman Problem (TSP): Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP-hard problem. There is no polynomial-time know solution for this problem.

DynamicProgramming:

Let the given set of vertices be {1, 2, 3, 4,…..,n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, I as the ending point, and all vertices appearing exactly once. Let the cost of this path have cost (i), and the cost of the corresponding Cycle would cost (i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks

simple so far. Now the question is how to get cost(i)?
To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i.*
We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

If the size of S is 2, then S must be {1, i},

C(S, i) = dist(1, i)

Else if size of S is greater than 2.

C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don't have nth in them.

Using the above recurrence relation, we can write a dynamic programming-based solution. There are at most $O(n*2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2*2^n)$. The time complexity is much less than O(n!) but still exponential. The space required is also exponential. So, this approach is also infeasible even for a slightly higher number of vertices.

Algorithm:

1) Let 1 be the starting and ending point for the salesman.
2) Construct MST with 1 as root using Prim's Algorithm.
3) List vertices visited in the preorder walk of the constructed MST and add 1 at the end.

Program:

```
#include<iostream>
```

```
using namespace std;

#define INT_MAX 999999

int n=4;
int dist[10][10] = {
        {0,20,42,25},
        {20,0,30,34},
        {42,30,0,10},
        {25,34,10,0}
};
int VISITED_ALL = (1<<n) -1;

int dp[16][4];



int  tsp(int mask,int pos){

            if(mask==VISITED_ALL){
                        return dist[pos][0];
            }
            if(dp[mask][pos]!=-1){
               return dp[mask][pos];
            }

            //Now from current node, we will try to go to every other node and take the min ans
            int ans = INT_MAX;

            //Visit all the unvisited cities and take the best route
            for(int city=0;city<n;city++){
                if((mask&(1<<city))==0){

                        int newAns = dist[pos][city] + tsp( mask|(1<<city), city);
                        ans = min(ans, newAns);
                }

        }

        return dp[mask][pos] = ans;
}

int main(){
   /* init the dp array */
   for(int i=0;i<(1<<n);i++){
       for(int j=0;j<n;j++){
           dp[i][j] = -1;
       }
   }
        cout<<"Travelling Salesman Distance is "<<tsp(1,0);
return 0;
}
```

Output:

```
Travelling Salesman Distance is : 85
```

Conclusion: Thus, we have studied and implemented the traveling salesperson problem using Dynamic Programming) provided a set of cities and the distance between every pair of cities

Experiment No. 8

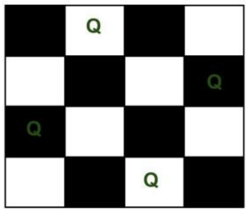
Title: Implement a Java/C++ program for solving N-Queen's problem using Backtracking. (Assume N=4)

Theory:

Backtracking: Backtracking is a very general technique that can be used to solve a wide variety of problems in a combinatorial enumeration. Many of the algorithms to be found in succeeding chapters are backtracking in various guises. It also forms the basis for other useful techniques such as branch-and-bound and alpha-beta pruning, which find wide application in operations research (and, more generally, discrete optimization) and artificial intelligence, respectively. As an introductory problem, consider the puzzle of trying to determine all ways to place n non-taking queens on an n by n chessboard. Recall that, in the game of chess, the queen attacks any piece that is in the same row, column, or diagonal. To gain insight, we first consider the specific value of n = 4. We start with an empty chessboard and try to build up our solution column by column, starting from the left. We can keep track of the approaches that we explored so far by maintaining a backtracking tree whose root is the empty board and where each level of the tree corresponds to a column on the board; i.e., to the number of queens that have been placed so far. Figure 3.1 shows the backtracking tree. This tree is constructed as follows: Starting from an empty board, try to place a queen in column one. There are four positions for the queen, which correspond to the four children of the root. In column one, we first try the queen in row 1, then row 2, etc., from left to right in the tree. After successfully placing a queen in some column we then proceed to the next column and recursively try again. If we get stuck at some column k, then we backtrack to the previous column k − 1, where we again try to advance the queen in column k − 1 from its current position. Observe that the tree is constructed in preorder. All nodes at level n represent solutions; for n = 4 there are two solutions.

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 to clashes then we backtrack and return false.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, the following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

```
{ 0,  1,  0,  0}

{ 0,  0,  0,  1}

{ 1,  0,  0,  0}

{ 0,  0,  1,  0}
```

Time Complexity for this algorithm is **exponential** because of recursive calls and backtracking algorithm.
Space Complexity is **O(n)** because in the worst case, our recursion will be N level deep for an NxN board.

Algorithm:

```
1) Start in the leftmost column

2) If all queens are placed

    return true
```

3) Try all rows in the current column. Do the following for every tried row.

a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if the placing queen here leads to a solution.

b) If placing the queen in [row, column] leads to a solution then return true.

c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.

3) If all rows have been tried and nothing worked, return false to trigger backtracking.

* C++ Program to Solve N-Queen Problem

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#define N 8
using namespace std;

/* print solution */
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            cout<<board[i][j]<<"  ";
        cout<<endl;
    }
}

/* check if a queen can be placed on board[row][col]*/
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;
    for (i = 0; i < col; i++)
    {
        if (board[row][i])
            return false;
    }
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
    {
        if (board[i][j])
            return false;
    }

    for (i = row, j = col; j >= 0 && i < N; i++, j--)
    {
        if (board[i][j])
            return false;
    }

    return true;
}

/*solve N Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++)
    {
        if ( isSafe(board, i, col) )
        {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1) == true)
                return true;
            board[i][col] = 0;
        }
    }
    return false;
}

/* solves the N Queen problem using Backtracking.*/
bool solveNQ()
{
    int board[N][N] = {0};
    if (solveNQUtil(board, 0) == false)
    {
        cout<<"Solution does not exist"<<endl;
        return false;
    }
    printSolution(board);
    return true;
}

// Main
int main()
```

```
{
    solveNQ();
    return 0;
}
```

Output:

```
1  0  0  0  0  0  0  0
0  0  0  0  0  0  1  0
0  0  0  0  1  0  0  0
0  0  0  0  0  0  0  1
0  1  0  0  0  0  0  0
0  0  0  1  0  0  0  0
0  0  0  0  0  1  0  0
0  0  1  0  0  0  0  0
```

Conclusion: Thus, we studied and implemented a C++ program for solving N-Queen's problem using Backtracking by placing N chess queens on an N×N chessboard so that no two queens attack each other.

Experiment No. 9

Aim: Implement traveling salesman problem using branch and bound approach using C++.

Theory: TSP PROBLEM- As researchers in the area of algorithm design know, the Traveling Salesperson Problem (TSP) is one of many combinatorial optimization problems in the set NP-complete. The only known solutions are of exponential complexity and are therefore intractable.

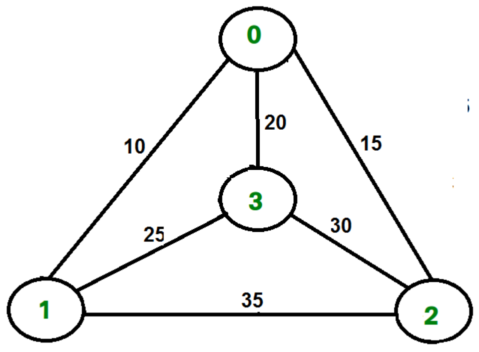
Recall that the TSP assumes that the distances between n cities are specified in a cost matrix that specifies the non-negative cost between any pair of cities. A salesperson is given the task of starting at city 1, traveling to each of the other cities, and then returning to city 1. Each city must be visited exactly once and no cities can be skipped. The goal is to find the sequence of cities that starts and ends with city 1 such that the overall cost of the tour is minimized.

When one first encounters this classic problem, it seems relatively easy to solve. Perhaps that is because the problem itself is easy to understand. But consider the fact that there are (n – 1)! tours in an n-city problem since each tour must start and end with city 1. So only for toy-size problems, say n <= 14, would a brute force evaluation of all possible tours and subsequent computation of the minimum tour be feasible. For a problem with 4 cities, the 3! = 6 possible tours are:

1, 2, 3, 4, 1
1, 2, 4, 3, 1
1, 3, 2, 4, 1
1, 3, 4, 2, 1
1, 4, 2, 3, 1
1, 4, 3, 2, 1

Two different fairly well-known branch and bound approaches to solving the TSP will be explored and implemented in Java. Several moderate-sized problems of sizes 20, 24, and 33 will be used to test the implementations.

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.



For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 0-1-3-2-0. The cost of the tour is 10+25+30+15 which is 80.

As seen in the previous articles, in the Branch and Bound method, for the current node in the tree, we compute a bound on the best possible solution that we can get if we down this node. If the bound on the best possible solution itself is worse than the current best (best computed so far), then we ignore the subtree rooted with the node. Note that the cost through a node includes two costs. 1) Cost of reaching the node from the root (When we reach a node, we have this cost computed) 2) Cost of reaching an answer from the current node to a leaf (We compute a bound on this cost to decide whether to ignore subtree with this node or not).

- In cases of a **maximization problem**, an upper bound tells us the maximum possible solution if we follow the given node. For example, in 0/1 knapsack we used the Greedy approach to find an upper bound.

- In cases of a **minimization problem**, a lower bound tells us the minimum possible solution if we follow the given node. For example, in the Job Assignment Problem, we get a lower bound by assigning the least cost job to a worker.

In branch and bound, the challenging part is figuring out a way to compute a bound on the best possible solution. Below is an idea used to compute bounds for the Traveling salesman problem.

Time Complexity: The worst-case complexity of Branch and Bound remains the same as that of the Brute Force clearly because in the worst case, we may never get a chance to prune a node. Whereas, in practice, it performs very well depending on the different instances of the TSP. The complexity also depends on the choice of the bounding function as they are the ones deciding how many nodes to be pruned.

Program:

```
#include<iostream>
using namespace std;
int main()
{
int i,j,k,n,min,g[20][20],c[20][20],s,s1[20][1],s2,lb;
cout << ("\n TRAVELLING SALESMAN PROBLEM");
cout << ("\n Input number of cities:");
cin >> n;
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
c[i][j]=0;
}}
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if(i==j)
continue;
else{
```

```
cout<<"input"<<i<<"to"<<j<<"cost:";
cin>>c[i][j];
}
}
}
for(i=2;i<=n;i++)
{
g[i][0]=c[i][1];
}
for(i=2;i<=n;i++)
{
for(j=2;j<=n;j++)
{
if(i!=j)
g[i][j]=c[i][j]+g[j][0];
}
}
for(i=2;i<=n;i++)
{
for(j=2;j<=n;j++)
{
if(i!=j)
break;
}
}
for(k=2;k<=n;k++){

if(i!=k && j!=k){
if((c[i][j]+g[i][k])<(c[i][k]+g[k][j]))
{
```

```
g[i][j]=c[i][j]+g[j][k];
s1[i][j]=j;
}
else
{
g[i][1]=c[i][k]+g[k][j];
s1[i][1]=k;
}
}
}
min=c[1][2]+g[2][1];
s=2;
for(i=3;i<n;i++)
{
if((c[i][i]+g[i][i])<min)
{
min=c[1][i]+g[i][1];
s=i;
}
}
int y=g[i][1]+g[i][j]+g[i][i];
lb=(y/2);
cout<<"Edge Matrix";
for(i=1;i<=n;i++)
{
cout<<"\n";
for(j=1;j<=n;j++)
{
cout<<"\t"<<c[i][j];
}
```

```
}
cout<<"\n min"<<min;
cout<<"\n\b"<<lb;
for(i=2;i<=n;i++)
{
if(s!=i && s1[s][1]!=i)
{
s2=i;
}
}
cout<<"\n"<<1<<"-->"<<s<<"-->"<<s1[s][1]<<"-->"<<s2<<"-->"<<1<<"\n";

return (0);
}
```

Output:

```
TRAVELING SALESMAN PROBLEM
Input number of cities: 3
input1to2cost:20
input1to3cost:12
input2to1cost:33
input2to3cost:23
input3to1cost:34
input3to2cost:12
Edge Matrix
      0      20     12
      33      0     23
      34     12      0

 min21
3915 with the path  1-->2-->3-->1
```

Conclusion: Thus, we have studied and implemented the Travelling salesman problem using the branch and bound approach using C++.

Experiment No. 10

Aim: Design and implement the presence of the Hamiltonian Cycle in an undirected Graph G of n vertices using C++.

Theory: Hamiltonian Path is a path in a directed or undirected graph that visits each vertex exactly once. The problem to check whether a graph (directed or undirected) contains a Hamiltonian Path is NP-complete, and so is the problem of finding all the Hamiltonian Paths in a graph.

A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains the Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

Input:

A 2D array graph[V][V] where V is the number of vertices in the graph and graph[V][V] is an adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

Output:

An array path[V] should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}. There are more Hamiltonian Cycles in the graph like {0, 3, 4, 2, 1, 0}

```
(0)--(1)--(2)

 |   / \   |

 |  /   \  |

 | /     \ |

(3)-------(4)
```

And the following graph doesn't contain any Hamiltonian Cycle.

```
(0)--(1)--(2)
 |   / \   |
 |  /   \  |
 | /     \ |
(3)       (4)
```

There is a simple relation between the problems of finding a Hamiltonian path and a Hamiltonian cycle. In one direction, the Hamiltonian path problem for graph G is equivalent to the Hamiltonian cycle problem in a graph H obtained from G by adding a new vertex and connecting it to all vertices of G. Thus, finding a Hamiltonian path cannot be significantly slower (in the worst case, as a function of the number of vertices) than finding a Hamiltonian cycle. In the other direction, the Hamiltonian cycle problem for a graph G is equivalent to the Hamiltonian path problem in the graph H obtained by copying one vertex v of G, v', that is, letting v' have the same neighborhood as v, and by adding two dummy vertices of degree one, and connecting them with v and v', respectively.[2] The Hamiltonian cycle problem is also a special case of the traveling salesman problem, obtained by setting the distance between two cities to one if they are adjacent and two otherwise, and verifying that the total distance traveled is equal to n.

Program:

```
/*
 * C++ Program to Find Hamiltonian Cycle
 */
#include <iostream>
#include <cstdio>
#include <cstdlib>
#define V 5
using namespace std;
void printSolution(int path[]);
/*  check if the vertex v can be added at index 'pos' in the Hamiltonian Cycle */
```

```
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
    if (graph [path[pos-1]][v] == 0)
        return false;
   for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;
    return true;
}

/* solve hamiltonian cycle problem */
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    if (pos == V)
    {
        if (graph[ path[pos-1] ][ path[0] ] == 1)
            return true;
        else
            return false;
    }

    for (int v = 1; v < V; v++)
    {
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;
            if (hamCycleUtil (graph, path, pos+1) == true)
                return true;
            path[pos] = -1;
        }
    }
    return false;
}

/* solves the Hamiltonian Cycle problem using Backtracking.*/
bool hamCycle(bool graph[V][V])
{
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false)
    {
        cout<<"\nSolution does not exist"<<endl;
        return false;
    }
    printSolution(path);
    return true;
}
/* Main */
void printSolution(int path[])
{
    cout<<"Solution Exists:";
    cout<<" Following is one Hamiltonian Cycle \n"<<endl;
    for (int i = 0; i < V; i++)
        cout<<path[i]<<"  ";
    cout<< path[0]<<endl;
}

int main()
```

```
{
  bool graph1[V][V] = {{0, 1, 0, 1, 0},
                     {1, 0, 1, 1, 1},
                     {0, 1, 0, 0, 1},
                     {1, 1, 0, 0, 1},
                     {0, 1, 1, 1, 0},
                    };
  hamCycle(graph1);
   bool graph2[V][V] = {{0, 1, 0, 1, 0},
                     {1, 0, 1, 1, 1},
                     {0, 1, 0, 0, 1},
                     {1, 1, 0, 0, 0},
                     {0, 1, 1, 0, 0},
                    };
   hamCycle(graph2);
   return 0;
}
```

```
Solution Exists: Following is one Hamiltonian Cycle
0  1  2  4  3  0
```

Conclusion: Thus, we have implemented the presence of the Hamiltonian Cycle in an undirected Graph G of n vertices using C++. The program is used a decision-based problem-solving strategy.

Content Beyond Syllabus

Experiment No.11

Aim: Given a series of n arrays (of appropriate sizes) to multiply: A1×A2×⋯×An Determine where to place parentheses to minimize the number of multiplications. Multiplying an i×j array with a j×k array takes i×j×k array. Use Matrix chain multiplication.

Theory: Problem: Given a sequence of matrices A1,A2,…, An insert parentheses so that the product of the matrices, in order, is unambiguous and needs the minimal number of multiplication. Assume that the matrix dimensions allow multiplication, in order for Matrix multiplication is associative: A1(A2A3)=(A1A2)A3
A product is unambiguous if no factor is multiplied on both the left and the right and all factors are either a single matrix or an unambiguous product (in parentheses)
Number of Multiplications: Multiplying an i×j and a j×k matrix requires ijk multiplications. Each element of the product requires j multiplications, and there are ik elements
Number of Parenthesizations
Example:
Given the matrices A1, A2, A3, A4
Assume the dimensions of A1=d0×d1, etc Below are the five possible parenthesizations of these arrays, along with the number of multiplications:
(A1A2)(A3A4):d0d1d2+d2d3d4+d0d2d4
((A1A2)A3)A4:d0d1d2+d0d2d3+d0d3d4
(A1(A2A3))A4:d1d2d3+d0d1d3+d0d3d4
A1((A2A3)A4):d1d2d3+d1d3d4+d0d1d4
A1(A2(A3A4)):d2d3d4+d1d2d4+d0d1d4

The number of parenthesizations is at least T(n)≥T(n−1)+T(n−1), Since the number with the first element removed is T(n−1) , which is also the number with the last removed. Thus the number of parenthesizations is Ω(2n)

The number is actually T(n)=∑k=1n−1T(k)T(n−k) which is related to the Catalan numbers.
This is because the original product can be split into 2 subproducts in k , places. Each split is to be parenthesized optimally. This recurrence is related to the Catalan numbers.
Characterizing the Optimal Parenthesization : An optimal parenthesization of A1…An

must break the product into two expressions, each of which is parenthesized or is a single array. Assume the break occurs at position k. In the optimal solution, the solution to the product A1…Ak must be optimal Otherwise, we could improve A1…An, by improving A1…Ak But the solution to A1…An is known to be optimal This is a contradiction Thus the solution to A1…An is known to be optimal.

Principle of Optimality: This problem exhibits the Principle of Optimality: The optimal solution to product A1…An contains the optimal solution to two subproducts.
Thus, we can use Dynamic Programming: Consider a recursive solution. Then improve its performance with memorization or by rewriting bottom-up Matrix Dimensions: Consider matrix product A1×⋯×An

Let the dimensions of the matrix Ai
be di−1×di

Thus the dimensions of matrix product Ai×⋯×Aj
is di−1×dj

Recursive Solution

Let M[i,j]

represent the number of multiplications required for matrix product Ai×⋯×Aj

For 1≤i≤j<n

M[i,i]=0
since no product is required

The optimal solution of Ai×Aj
must break at some point, k, with i≤k<j

Thus, M[i,j]=M[i,k]+M[k+1,j]+di−1dkdj

Thus, M[i,j]={0mini≤k<j{M[i,k]+M[k+1,j]+di−1dkdj}if i=jif i<j

This is easily expressed as a recursive function (with exponential complexity)
Complexity: P(1) = 1, P(n) = Sum (k=1, n-1) P(k)P(n-k), P(n) = Ω(fn/n32)

Program:

```
#include <iostream>
#include <climits>
using namespace std;

// Function to find the most efficient way to multiply
// a given sequence of matrices
int MatrixChainMultiplication(int dims[], int i, int j)
{
    // base case: one matrix
    if (j <= i + 1) {
        return 0;
    }

    // stores the minimum number of scalar multiplications (i.e., cost)
    // needed to compute matrix `M[i+1] … M[j] = M[i…j]`
    int min = INT_MAX;

    // take the minimum over each possible position at which the
    // sequence of matrices can be split

    /*
        (M[i+1]) × (M[i+2]………………M[j])
        (M[i+1]M[i+2]) × (M[i+3…………M[j])
        …
        …
        (M[i+1]M[i+2]…………M[j-1]) × (M[j])
    */

    for (int k = i + 1; k <= j - 1; k++)
    {
        // recur for `M[i+1]…M[k]` to get an `i × k` matrix
        int cost = MatrixChainMultiplication(dims, i, k);

        // recur for `M[k+1]…M[j]` to get an `k × j` matrix
        cost += MatrixChainMultiplication(dims, k, j);

        // cost to multiply two `i × k` and `k × j` matrix
        cost +=    dims[i] * dims[k] * dims[j];

        if (cost < min) {
            min = cost;
        }
    }

    // return the minimum cost to multiply `M[j+1]…M[j]`
    return min;
}

// Matrix Chain Multiplication Problem
int main()
{
    // Matrix `M[i]` has dimension `dims[i-1] × dims[i]` for `i = 1…n`
    // input is `10 × 30` matrix, `30 × 5` matrix, `5 × 60` matrix
    int dims[] = { 10, 30, 5, 60 };
    int n = sizeof(dims) / sizeof(dims[0]);

    cout << "The minimum cost is " << Matrix Chain Multiplication (dims, 0, n - 1);

    return 0;
}
```

Output: The minimum cost is 4500

Conclusion: Thus, we have studied and implemented Matrix chain multiplication using dynamic programming provided a series of n arrays (of appropriate sizes) to multiply: A1×A2×⋯×An.

Experiment No. 12

Aim: To study optimization Algorithms (case study)

Theory: Optimization Algorithms -Optimization refers to a procedure for finding the input parameters or arguments to a function that result in the minimum or maximum output of the function.

The most common type of optimization problem encountered in machine learning is **continuous function optimization**, where the input arguments to the function are real-valued numeric values, e.g., floating-point values. The output from the function is also a real-valued evaluation of the input values.

We might refer to problems of this type as continuous function optimization, to distinguish from functions that take discrete variables and are referred to as combinatorial optimization problems.

There are many different types of optimization algorithms that can be used for continuous function optimization problems, and perhaps just as many ways to group and summarize them.

One approach to grouping optimization algorithms is based on the amount of information available about the target function that is being optimized that, in turn, can be used and harnessed by the optimization algorithm.

Generally, the more information that is available about the target function, the easier the function is to optimize if the information can effectively be used in the search.

Perhaps the major division in optimization algorithms is whether the objective function can be differentiated at a point or not. That is, whether the first derivative (gradient or slope) of the function can be calculated for a given candidate solution or not. This partitions algorithm into those that can make use of the calculated gradient information and those that do not.

- Differentiable Target Function?
    - Algorithms that use derivative information.
    - Algorithms that do not use derivative information.

We will use this as the major division for grouping optimization algorithms in this tutorial and look at algorithms for differentiable and non-differentiable objective functions.

Note: this is not an exhaustive coverage of algorithms for continuous function optimization, although it does cover the major methods that you are likely to encounter as a regular practitioner.

Differentiable Objective Function

A differentiable function is a function where the derivative can be calculated for any given point in the input space.

The derivative of a function for a value is the rate or amount of change in the function at that point. It is often called the slope.

- **First-Order Derivative**: Slope or rate of change of an objective function at a given point.

The derivative of the function with more than one input variable (e.g. multivariate inputs) is commonly referred to as the gradient.

- **Gradient**: Derivative of a multivariate continuous objective function.

A derivative for a multivariate objective function is a vector, and each element in the vector is called a partial derivative, or the rate of change for a given variable at the point assuming all other variables are held constant.

- **Partial Derivative**: Element of a derivative of a multivariate objective function.

We can calculate the derivative of the objective function, which is the rate of change of the rate of change in the objective function. This is called the second derivative.

- **Second-Order Derivative**: Rate at which the derivative of the objective function changes.

For a function that takes multiple input variables, this is a matrix and is referred to as the Hessian matrix.

- **Hessian matrix**: Second derivative of a function with two or more input variables.

Simple differentiable functions can be optimized analytically using calculus. Typically, the objective functions that we are interested in cannot be solved analytically.

Optimization is significantly easier if the gradient of the objective function can be calculated, and as such, there has been a lot more research into optimization algorithms that use the derivative than those that do not.

Some groups of algorithms that use gradient information include:

- Bracketing Algorithms
- Local Descent Algorithms
- First-Order Algorithms
- Second-Order Algorithms

Bracketing Algorithms

Bracketing optimization algorithms are intended for optimization problems with one input variable where the optima are known to exist within a specific range.

Bracketing algorithms are able to efficiently navigate the known range and locate the optima, although they assume only a single optimum is present (referred to as unimodal objective functions).

Some bracketing algorithms may be able to be used without derivative information if it is not available.

Examples of bracketing algorithms include:

- Fibonacci Search
- Golden Section Search
- Bisection Method

Local Descent Algorithms

Local descent optimization algorithms are intended for optimization problems with more than one input variable and a single global optimum (e.g., unimodal objective function).

Perhaps the most common example of a local descent algorithm is the line search algorithm.

- Line Search

There are many variations of the line search (e.g. the Brent-Dekker algorithm), but the procedure generally involves choosing a direction to move in the search space, then performing a bracketing type search in a line or hyperplane in the chosen direction.

This process is repeated until no further improvements can be made.

The limitation is that it is computationally expensive to optimize each directional move in the search space.

First-Order Algorithms

First-order optimization algorithms explicitly involve using the first derivative (gradient) to choose the direction to move in the search space.

The procedures involve first calculating the gradient of the function, then following the gradient in the opposite direction (e.g. downhill to the minimum for minimization problems) using a step size (also called the learning rate).

The step size is a hyperparameter that controls how far to move in the search space, unlike “local descent algorithms” that perform a full line search for each directional move.

A step size that is too small results in a search that takes a long time and can get stuck, whereas a step size that is too large will result in zig-zagging or bouncing around the search space, missing the optima completely.

First-order algorithms are generally referred to as gradient descent, with more specific names referring to minor extensions to the procedure, e.g.:

- Gradient Descent
- Momentum
- Adagrad
- RMSProp
- Adam

The gradient descent algorithm also provides the template for the popular stochastic version of the algorithm, named Stochastic Gradient Descent (SGD) which is used to train artificial neural networks (deep learning) models.

The important difference is that the gradient is appropriated rather than calculated directly, using prediction error on training data, such as one sample (stochastic), all examples (batch), or a small subset of training data (mini-batch).

The extensions designed to accelerate the gradient descent algorithm (momentum, etc.) can be and are commonly used with SGD.

- Stochastic Gradient Descent
- Batch Gradient Descent
- Mini-Batch Gradient Descent

Second-Order Algorithms

Second-order optimization algorithms explicitly involve using the second derivative (Hessian) to choose the direction to move in the search space.

These algorithms are only appropriate for those objective functions where the Hessian matrix can be calculated or approximated.

Examples of second-order optimization algorithms for univariate objective functions include:

- Newton’s Method
- Secant Method

Second-order methods for multivariate objective functions are referred to as Quasi-Newton Methods.

- Quasi-Newton Method

There are many Quasi-Newton Methods, and they are typically named for the developers of the algorithm, such as:

- Davidson-Fletcher-Powell
- Broyden-Fletcher-Goldfarb-Shanno (BFGS)
- Limited-memory BFGS (L-BFGS)

Now that we are familiar with the so-called classical optimization algorithms, let’s look at algorithms used when the objective function is not differentiable.

Non-Differential Objective Function

Optimization algorithms that make use of the derivative of the objective function are fast and efficient.

Nevertheless, there are objective functions where the derivative cannot be calculated, typically because the function is complex for a variety of real-world reasons. Or the derivative can be calculated in some regions of the domain, but not all, or is not a good guide.

Some difficulties on objective functions for the classical algorithms described in the previous section include:

- No analytical description of the function (e.g. simulation).
- Multiple global optima (e.g. multimodal).
- Stochastic function evaluation (e.g. noisy).
- Discontinuous objective function (e.g. regions with invalid solutions).

As such, there are optimization algorithms that do not expect first- or second-order derivatives to be available.

These algorithms are sometimes referred to as black-box optimization algorithms as they assume little or nothing (relative to the classical methods) about the objective function.

A grouping of these algorithms includes:

- Direct Algorithms
- Stochastic Algorithms
- Population Algorithms

Let’s take a closer look at each in turn.

Direct Algorithms

Direct optimization algorithms are for objective functions for which derivatives cannot be calculated.

The algorithms are deterministic procedures and often assume the objective function has a single global optimum, e.g., unimodal.

Direct search methods are also typically referred to as a “*pattern search*” as they may navigate the search space using geometric shapes or decisions, e.g. patterns.

Gradient information is approximated directly (hence the name) from the result of the objective function comparing the relative difference between scores for points in the search space. These direct estimates are then used to choose a direction to move in the search space and triangulate the region of the optima.

Examples of direct search algorithms include:

- Cyclic Coordinate Search
- Powell’s Method
- Hooke-Jeeves Method
- Nelder-Mead Simplex Search

Stochastic Algorithms

Stochastic optimization algorithms are algorithms that make use of randomness in the search procedure for objective functions for which derivatives cannot be calculated.

Unlike the deterministic direct search methods, stochastic algorithms typically involve a lot more sampling of the objective function but are able to handle problems with deceptive local optima.

Stochastic optimization algorithms include:

- Simulated Annealing
- Evolution Strategy
- Cross-Entropy Method

Population Algorithms

Population optimization algorithms are stochastic optimization algorithms that maintain a pool (a population) of candidate solutions that together are used to sample, explore, and hone in on an optimum.

Algorithms of this type are intended for more challenging objective problems that may have noisy function evaluations and many global optima (multimodal), and finding a good or good enough solution is challenging or infeasible using other methods.

The pool of candidate solutions adds robustness to the search, increasing the likelihood of overcoming local optima.

Examples of population optimization algorithms include:

- Genetic Algorithm
- Differential Evolution
- Particle Swarm Optimization

Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Books

- Algorithms for Optimization, 2019.
- Essentials of Metaheuristics, 2011.
- Computational Intelligence: An Introduction, 2007.
- Introduction to Stochastic Search and Optimization, 2003.

Conclusion: Thus, we studied optimization algorithms in detail.