DESIGN AND ANALYSIS OF ALGORITHMS

SUBMITTED BY: SHAGUN
181210049
CSE II year(G-2)

TEAM PARTNER: YUVRAJ SINGH CHAMPAWAT
181210064
CSE II year(G-2)

SUBMITTED TO: Dr. CHANDRA MAURYA

Table of Content:

Divide and Conquer

Finding Peak Element of a given Array

Divide and Conquer Algorithm

Divide and Conquer is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Finding Peak Element of a given Array

A Peak Element is an element that is greater than its neighbours

An element A[i] of an Array (A) is a Peak element if :-

1. A[i-1] <= A[i] >= A[i+1] for 0 <= i <= N-1
2. A[i-1] <= A[i] if i= N-1
3. A[i] >= A[i+1] if i = 0

An Array can have multiple peak elements we have to find any of them.

Example :-

Input : [8, 9 ,10, 2, 5, 6]

Output : The peak element is : 10

Input : [8, 9, 10, 12, 15]

Output : The peak element is : 15

Input : [10, 8, 6, 5, 2]

Output : The peak element is : 10

Why Divide and Conquer Algorithm?

We can solve this problem by testing all the elements for peak element by running linear search on the array and then return the element which is greater than its neighbours. We need to handle two special cases for this. First, when Array is in ascending order last element is the peak element and Second, when Array is in descending order then first element is the peak element. The problem with this approach is that its worst case time complexity is O(n).

We can solve this problem easily in O(log(n)) time by using an idea similar to that of Binary Search, that is by using Divide and Conquer Technique.

Divide and Conquer Algorithm approach

We calculate the mid index and if the mid element is greater than both of its neighbours then return that element as Peak element.

If the right neighbor of mid is greater than the mid element, then we find the peak element recursively in the right side of the Array.

If the left neighbor of mid is greater than the mid element, then we find the peak element recursively in the left side of the Array.

Algorithm of problem :

```
findPeakElement( A, low, high, n)
        mid=(low+high)/2
        if ((mid = 0  or  A[mid - 1] <= A[mid])  and  (mid = A.length - 1  or  A[mid + 1] <= A[mid]))
                Then return (mid)
        if (mid - 1 >= 0   and   A[mid - 1] > A[mid])
                Then return  findPeakElement(A, left, mid-1)
        Else     return  findPeakElement(A, mid+1, right)
```

BACKTRACKING

Find all possible permutations of a String

Backtracking Algorithm

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons each partial candidate c (“backtracks”) as soon as it determines that ‘ c ‘ cannot possibly be completed to a valid solution.

Backtracking can be applied only for problems which admit the concept of a “partial candidate solution” and a relatively quick test of whether it can possibly be completed to a valid solution.

Find all possible permutations of a String

Permutations of a String means all possible combinations of characters of that String.

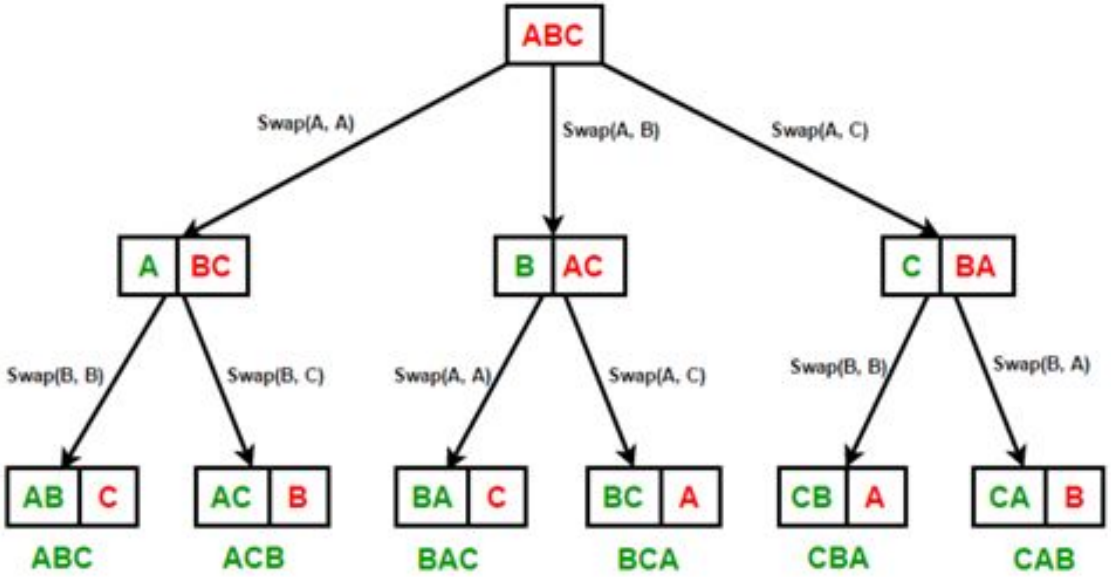
Here we will find all possible combinations of characters, that is, all possible permutations of a String having distinct or similar characters.

Example :-

Let String be : ABC

It has 6 possible permutations, those are

ABC, ACB, BAC, BCA, CAB, CBA



Recursion Tree for string "ABC"

Why Backtracking Algorithm?

We solved this problem using Backtracking Algorithm because it is the most efficient way to solve the computational problems related to finding possible combinations or paths. As, in this Algorithm the solutions are determined incrementally and it discards each partial candidates as soon as it determines that that particular can not be the part of solution. Thus, we left with few possibilities and we have to now find solutions in them only.

The time complexity of this solution is O(n.n!) as there are n! Possible combinations of a string having n distinct characters and time taken by each combination is O(n).

Backtracking Algorithm approach

The idea behind finding all possible permutations of a String using Backtracking Algorithm is to swap each of the remaining characters in the string with its first character and then find all the permutations of the remaining characters using a recursive call. The base case of the recursion is that when the string is left with only one unprocessed element.

Note - This approach can be used for the Strings either having all distinct elements or may have some similar elements.

Algorithm of problem :

```
Permutations ( string, index, n )

    If  ( index = n - 1 )

        Then    print(string)

    For ( i = index, i < n, i++ )

        Boolean    check = shouldSwap( string, index , i )
```

If (check)

Then Swap (String[i], String[j])

Permutations (String, i + 1, n)

Swap (String[i], String[j])

```
shouldSwap ( string,  start,  curr )

        for (int i = start; i < curr; i++)

                if (str[i] == str[curr])

                        Then    return false;

                Else    return true;
```

GREEDY METHOD

Activity Selection Problem

Greedy Method Algorithm

A Greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

In mathematical optimization, greedy algorithms optimally solve combinatorial problems having the properties of matroids, and give constant-factor approximations to optimization problems with submodular structure.

Activity Selection Problem

In Activity Selection problem, we are provided with a set of activities and the start and end time of each activity, we need to find the maximum number of activities that a single person can perform in given time assuming that a person can perform only a single activity at a time.

Example, of such type of problem is scheduling a room or chamber for different events each having its own time requirements.

Example :-

Input :

$(1, 4), (3, 5), (0, 6), (5, 7), (3, 8), (5, 9), (6, 10), (8, 11), (8, 12), (2, 13), (12, 14)$

Output:

$(1, 4), (5, 7), (8, 11), (12, 14)$

Why Greedy Method Algorithm?

The activity selection problem is a problem concerning the selection of non-conflicting activities to perform within a given time frame, given a set of activities each marked by a start and finish time. Let's assume there exists n activities with each of them being represented by a start and finish time as si and fi respectively. Two activities i and j are said to be non-conflicting if they have either si = fj or fi = sj.

Time complexity of solving this problem with Greedy Algorithm is O(nlog(n)).

Greedy Method Algorithm approach

We can solve this problem of activity selection by Greedy Method Algorithm by initially sorting all the activities in increasing order of their finish times and create a set S to store the selected activities. We initialize the set with the first activity and then from the second activity onwards, we include the activity in activities list if start time of the activity is greater or equal to the finish time of the last selected activity. Then we will repeat this procedure for each activity involved.

Algorithm of problem:

```
selectActivity ( sorted_activities )
        Int   k = 0
        Set <int>  out
        out.insert(0)
        For  ( i = 1, i < sorted_activities.size(), i++ )
                If  ( sorted_activities[i].start >= sorted_activities[k].finish )
                        out.insert(i)
```

```
            k = i

For ( int i : out )

        Print ( “{“ + sorted_activities[i].start + “, ” + sorted_activities[i].finish + “}” )
```

DYNAMIC PROGRAMMING

The Levenshtein distance (Edit distance) problem

Dynamic Programming Algorithm

Dynamic Programming(DP) is a very powerful technique to solve particular class of problems. It demands very elegant formulation of the approach and simple thinking.

If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again. If the given problem can be broken up into smaller subproblems and these smaller subproblems are in turn divided into still-smaller ones, and in this process, if you observe some overlapping subproblems, then it’s a big hint for DP. Also, the optimal solutions to the subproblems contribute to the optimal solution of the given problem.

The Levenshtein (Edit distance) problem

Edit Distance is a way of quantifying for dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into other. The Levenshtein distance between two words is the minimum number of single character edits(i.e. Insertions, deletions and substitutions) require to change one word into the other.Each of these operations has unit cost.

The Edit distance problem has an optimal substructure. This means the problem can be broken down into smaller, simpler “subproblems”, which can be broken down into yet simpler subproblems, and so on, until finally, the solution becomes trivial.

- **Problem: C**onvert string X[1..m] to Y[1..n] by performing edit operations on string X.

- **Sub-problem:** Convert substring X[1..i] to Y[1..j] by performing edit operations on single substring X.

Example

The Levenshtein distance between `kitten` and `sitting` is 3. A minimal edit script that transforms the former into the latter is:

```
kitten -> sitten (substitution of s for k)

sitten -> sittin (substitution of i for e)

sittin -> sitting (addition of g at the end)
```

Dynamic Programming Approach

1. A matrix is initialized measuring in the (m, n) cell the Levenshtein distance between the m-character prefix of one with the n-prefix of the other word.
2. The matrix can be filled from the upper left to the lower right corner.
3. Each jump horizontally or vertically corresponds to an insert or a delete, respectively.
4. The cost is normally set to 1 for each of the operations.
5. The diagonal jump can cost either one, if the two characters in the row and column do not match else 0, if they match. Each cell always minimizes the cost locally.
6. This way the number in the lower right corner is the Levenshtein distance between both words.

Why Dynamic Programming Algorithm?

Dynamic programming helps you to write the optimized code.The problem that you solved in exponential time complexity, you can solve it in O(n^2)time complexity.

As it is a recursive programming technique, it reduces the line code. One of the major advantages of using dynamic programming is it speeds up the processing as we use previously calculated references.

In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.

Algorithm of Problem:

```
findEditDistance(X,n,Y,m)

        if(m==0)      return n

        if (n==0)     return m

        cost=(X.charAt(m - 1) == Y.charAt(n-1)) ?  0 : 1

       return  minimum(findEditDistance(X, m - 1,  Y, n) +1,

                findEditDistance(X, m, Y, n-1) + 1,

                findEditDistance((X, m-1, Y, n-1 ) + cost)
```

BRANCH AND BOUND

Travelling Salesman Problem

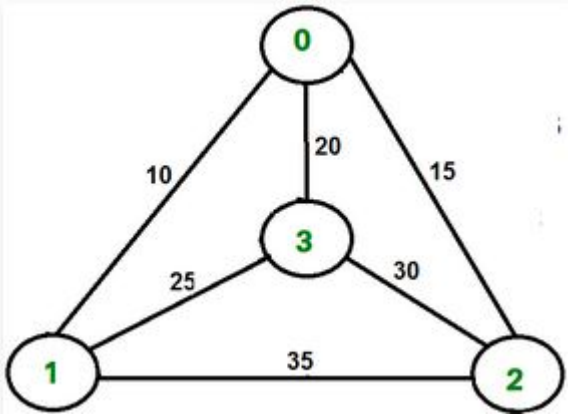
Branch and Bound

Branch and Bound (BnB) is a general programming paradigm used, for example, in operations research to solve hard combinatorial optimization problems. Branching is the process of spawning subproblems, and bounding refers to ignoring partial solutions that cannot be better than the current best solution. To this end, lower and upper bounds *L* and *U* are maintained. Since global control values on the solution quality improve over time, branch-and-bound is effective in solving optimization problems, in which a cost-optimal assignment to the problem variables has to be found.

The Travelling Salesman Problem

Given a set of cities and 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 figure on 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.



Why Branch and Bound Algorithm?

In solving the traveling salesman problem using branch and bound approach, while generating the node for every node, we find out the cost and we follow the path which is taking us to the minimized or maximized, i.e. optimized solution node.

So, the branch and bound approach will inspect less subproblems and thus saves computation time and saves storage space.

Branch and Bound Programming Approach

In Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we go down this node. If the bound on best possible solution itself is worse than 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 current node to a leaf (We compute a bound on this cost to decide whether to ignore subtree with this node or not).

Algorithm of Problem:

```
static void TSP(int adj[][])

        curr_path[] = new int[N + 1]   and    curr_bound = 0

        Arrays.fill(curr_path, -1)

         Arrays.fill(visited, false)

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

                curr_bound += (firstMin(adj, i) +  secondMin(adj, i))

                 curr_bound = (curr_bound==1)? curr_bound/2 + 1 :  curr_bound/2

                  visited[0] = true;

              curr_path[0] = 0;      TSPRec(adj, curr_bound, 0, 1, curr_path)
```