DESIGN AND ANALYSIS OF ALGORITHMS

MADE BY-ANAND PANDEY & RAGHAV SHUKLA

CSE II YEAR 181210009/181210038

1. DYNAMIC PROGRAMMING

- DYNAMIC PROGRAMMING (USUALLY REFERRED TO AS **DP**) IS A VERY POWERFUL TECHNIQUE TO SOLVE A PARTICULAR CLASS OF PROBLEMS. IT DEMANDS VERY ELEGANT FORMULATION OF THE APPROACH AND SIMPLE THINKING AND THE CODING PART IS VERY EASY. THE IDEA IS VERY SIMPLE, 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.. SHORTLY *'REMEMBER YOUR PAST'* :) . IF THE GIVEN PROBLEM CAN BE BROKEN UP IN TO SMALLER SUB-PROBLEMS AND THESE SMALLER SUBPROBLEMS ARE IN TURN DIVIDED IN TO STILL-SMALLER ONES, AND IN THIS PROCESS, IF YOU OBSERVE SOME OVER-LAPPING SUBPROBLEMS, THEN ITS A BIG HINT FOR DP. ALSO, THE OPTIMAL SOLUTIONS TO THE SUBPROBLEMS CONTRIBUTE TO THE OPTIMAL SOLUTION OF THE GIVEN PROBLEM.

TWO WAYS OF SOLVING DP PROBLEMS

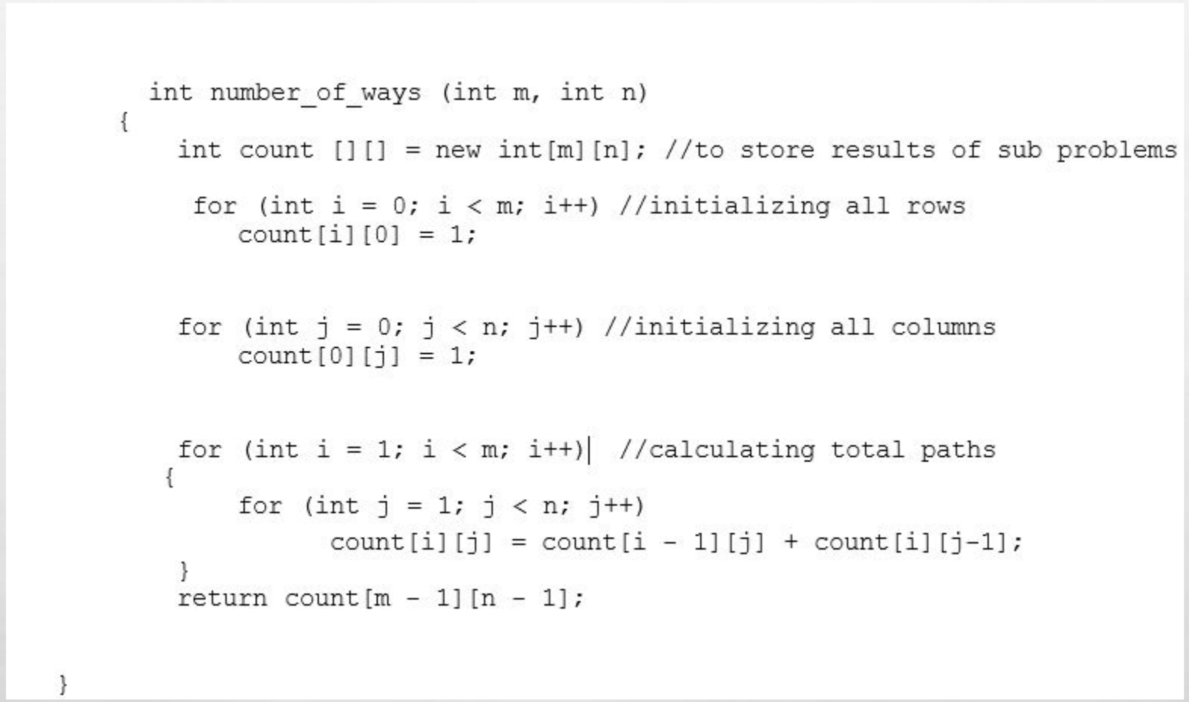
- **1.) TOP-DOWN :** START SOLVING THE GIVEN PROBLEM BY BREAKING IT DOWN. IF YOU SEE THAT THE PROBLEM HAS BEEN SOLVED ALREADY, THEN JUST RETURN THE SAVED ANSWER. IF IT HAS NOT BEEN SOLVED, SOLVE IT AND SAVE THE ANSWER. THIS IS USUALLY EASY TO THINK OF AND VERY INTUITIVE. THIS IS REFERRED TO AS ***MEMOIZATION***.
- **2.) BOTTOM-UP :** ANALYZE THE PROBLEM AND SEE THE ORDER IN WHICH THE SUB-PROBLEMS ARE SOLVED AND START SOLVING FROM THE TRIVIAL SUBPROBLEM, UP TOWARDS THE GIVEN PROBLEM. IN THIS PROCESS, IT IS GUARANTEED THAT THE SUBPROBLEMS ARE SOLVED BEFORE SOLVING THE PROBLEM. THIS IS REFERRED TO AS ***DYNAMIC PROGRAMMING***.

EXAMPLE OF DYNAMIC PROGRAMMING

COUNT ALL POSSIBLE PATHS FROM TOP LEFT TO BOTTOM RIGHT OF A M X N MATRIX

THE PROBLEM IS TO COUNT ALL THE POSSIBLE PATHS FROM TOP LEFT TO BOTTOM RIGHT OF A M X N MATRIX WITH THE CONSTRAINTS THAT ***FROM EACH CELL YOU CAN EITHER MOVE ONLY TO RIGHT OR DOWN***

SOLUTION ALGORITHM



```
     int number_of_ways (int m, int n)
   {
       int count [][] = new int[m][n]; //to store results of sub problems

        for (int i = 0; i < m; i++) //initializing all rows
           count[i][0] = 1;


       for (int j = 0; j < n; j++) //initializing all columns
           count[0][j] = 1;


       for (int i = 1; i < m; i++)  //calculating total paths
      {
           for (int j = 1; j < n; j++)
                 count[i][j] = count[i - 1][j] + count[i][j-1];
       }
       return count[m - 1][n - 1];


}
```

FOR THE EXPLANATION PART , I AM ATTACHING MY SELF MADE VIDEO

LEETCODE

CODING INTERVIEW PROBLEMS

2. GREEDY ALGORITHM

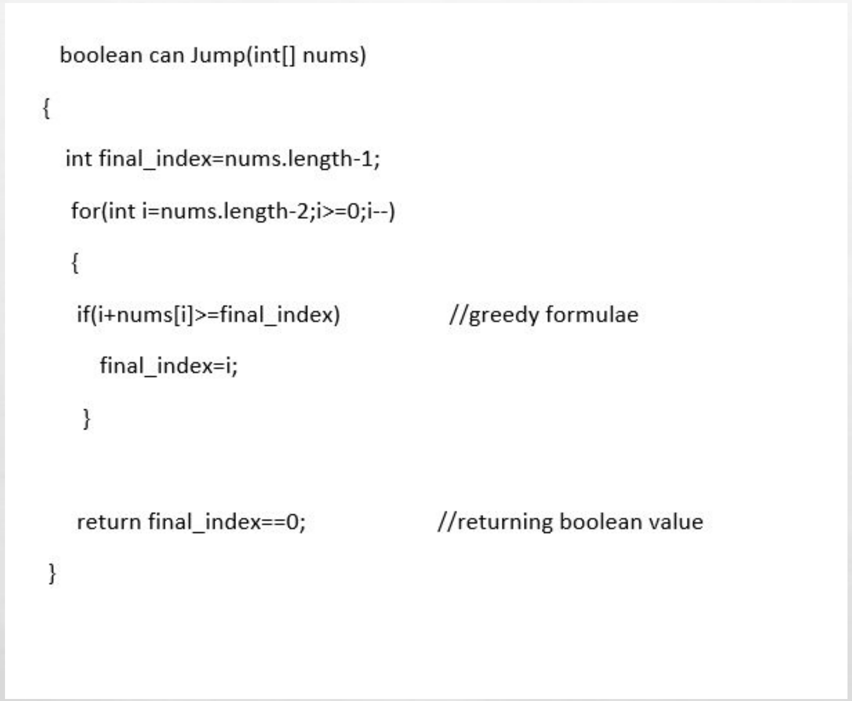
A **GREEDY ALGORITHM IS** A SIMPLE, INTUITIVE **ALGORITHM** THAT **IS** USED IN OPTIMIZATION PROBLEMS. THE **ALGORITHM** MAKES THE OPTIMAL CHOICE AT EACH STEP AS IT ATTEMPTS TO FIND THE OVERALL OPTIMAL WAY TO SOLVE THE ENTIRE PROBLEM

EXAMPLE OF GREEDY METHOD

THE JUMP GAME

- GIVEN AN ARRAY OF NON-NEGATIVE INTEGERS, YOU ARE INITIALLY POSITIONED AT THE FIRST INDEX OF THE ARRAY.
- EACH ELEMENT IN THE ARRAY REPRESENTS YOUR MAXIMUM JUMP LENGTH AT THAT POSITION.
- DETERMINE IF YOU ARE ABLE TO REACH THE LAST INDEX.

SOLUTION ALGORITHM



```
  boolean can Jump(int[] nums)
{
   int final_index=nums.length-1;
    for(int i=nums.length-2;i>=0;i--)
    {
     if(i+nums[i]>=final_index)              //greedy formulae
        final_index=i;
      }

     return final_index==0;                  //returning boolean value
 }
```

FOR THE EXPLANATION PART ,
I AM ATTACHING MY SELF MADE VIDEO

LEETCODE

CODING INTERVIEW PROBLEMS

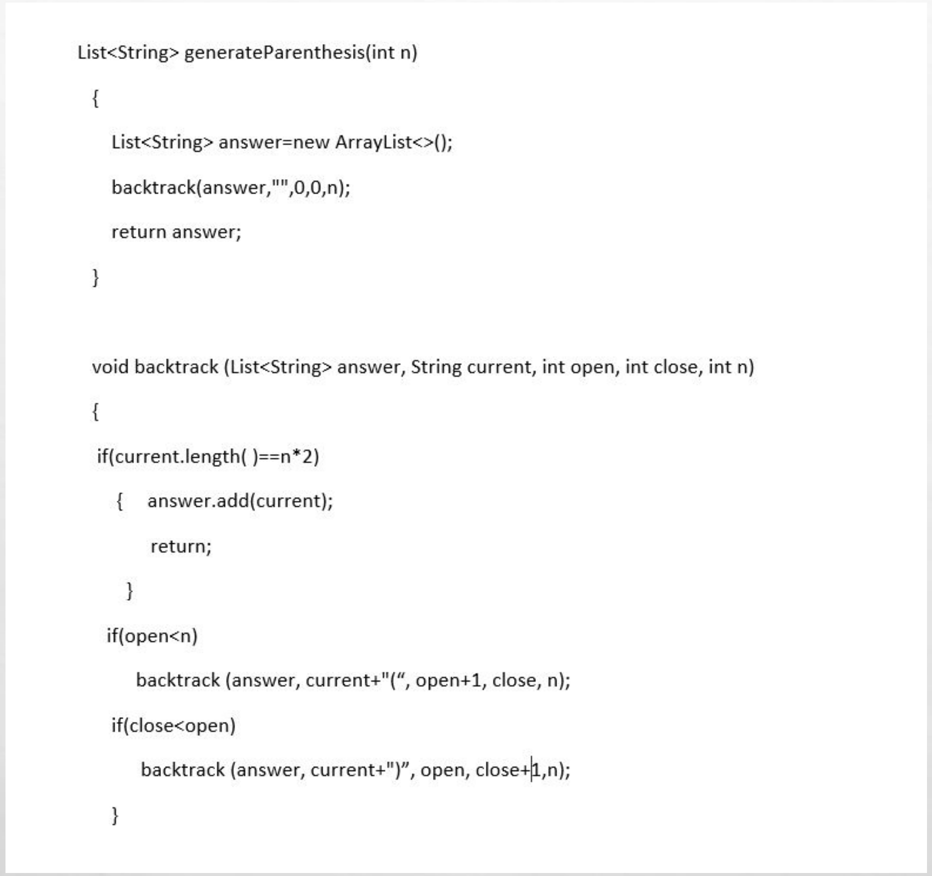
3. BACKTRACKING

- **BACKTRACKING** IS A TECHNIQUE BASED ON ALGORITHM TO SOLVE PROBLEM. IT USES RECURSIVE CALLING TO FIND THE SOLUTION BY BUILDING A SOLUTION STEP BY STEP INCREASING VALUES WITH TIME. IT REMOVES THE SOLUTIONS THAT DOESN'T GIVE RISE TO THE SOLUTION OF THE PROBLEM BASED ON THE CONSTRAINTS GIVEN TO SOLVE THE PROBLEM.
- BACKTRACKING ALGORITHM IS APPLIED TO SOME SPECIFIC TYPES OF PROBLEMS,
- DECISION PROBLEM USED TO FIND A FEASIBLE SOLUTION OF THE PROBLEM.
- OPTIMIZATION PROBLEM USED TO FIND THE BEST SOLUTION THAT CAN BE APPLIED.
- ENUMERATION PROBLEM USED TO FIND THE SET OF ALL FEASIBLE SOLUTIONS OF THE PROBLEM.

EXAMPLE OF BACKTRACKING

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.
For example, given *n* = 3, a solution set is:
["((()))", "(()())", "(())()", "()(())", "()()()"]

SOLUTION ALGORITHM



```
List<String> generateParenthesis(int n)
  {
     List<String> answer=new ArrayList<>();
     backtrack(answer,"",0,0,n);
     return answer;
  }

  void backtrack (List<String> answer, String current, int open, int close, int n)
  {
   if(current.length( )==n*2)
      {    answer.add(current);
           return;
        }
    if(open<n)
        backtrack (answer, current+"(“, open+1, close, n);
     if(close<open)
         backtrack (answer, current+")”, open, close+1,n);
     }
```

FOR THE EXPLANATION PART ,
I AM ATTACHING MY SELF MADE VIDEO

4. DIVIDE AND CONQUER

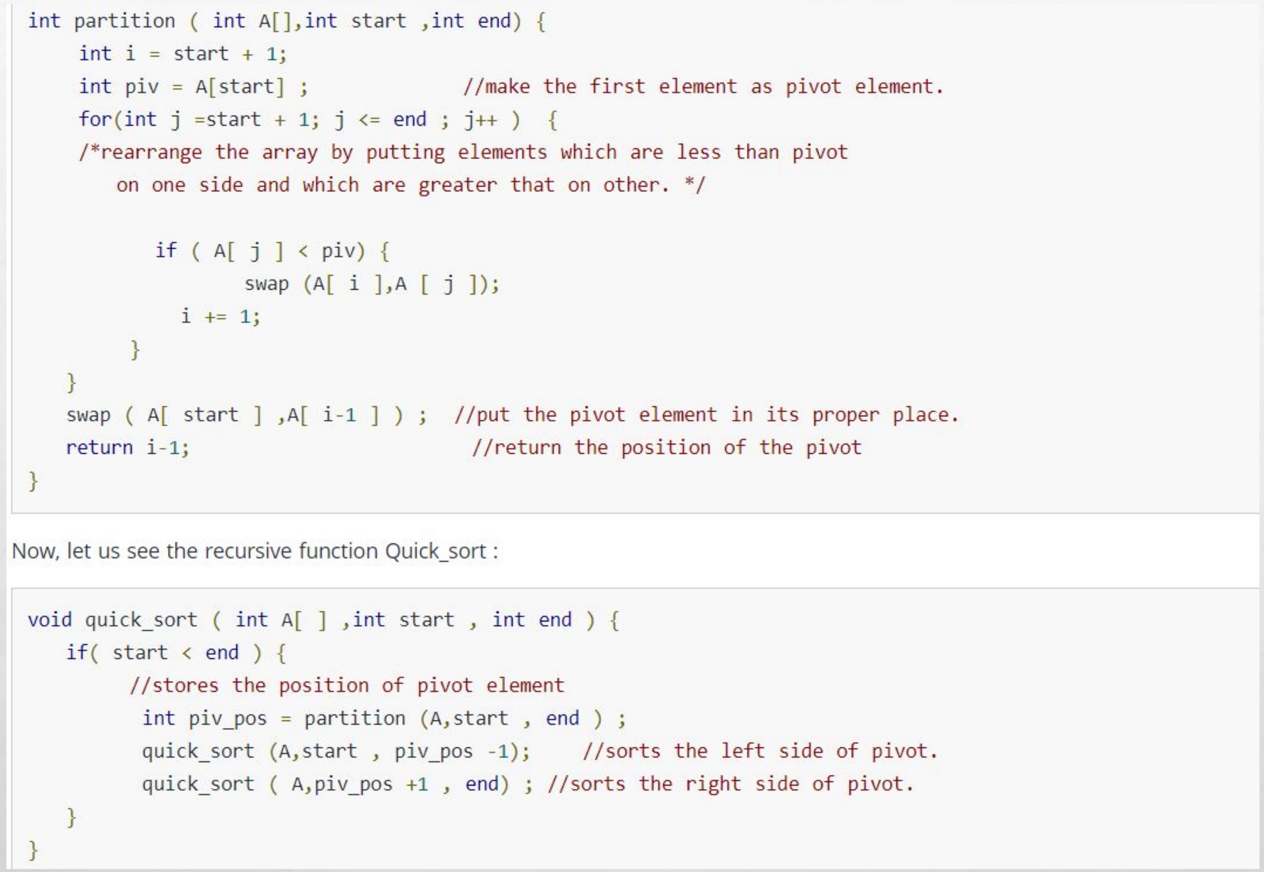
IN **DIVIDE AND CONQUER APPROACH**, A PROBLEM IS DIVIDED INTO SMALLER PROBLEMS, THEN THE SMALLER PROBLEMS ARE SOLVED INDEPENDENTLY, AND FINALLY THE SOLUTIONS OF SMALLER PROBLEMS ARE COMBINED INTO A SOLUTION FOR THE LARGE PROBLEM.

- GENERALLY, DIVIDE-AND-CONQUER ALGORITHMS HAVE THREE PARTS −
- **DIVIDE THE PROBLEM** INTO A NUMBER OF SUB-PROBLEMS THAT ARE SMALLER INSTANCES OF THE SAME PROBLEM.
- **CONQUER THE SUB-PROBLEMS** BY SOLVING THEM RECURSIVELY. IF THEY ARE SMALL ENOUGH, SOLVE THE SUB-PROBLEMS AS BASE CASES.
- **COMBINE THE SOLUTIONS** TO THE SUB-PROBLEMS INTO THE SOLUTION FOR THE ORIGINAL PROBLEM.

EXAMPLE OF DIVIDE AND CONQUER

- SORT THE GIVEN ARRAY USING QUICK SORT(DIVIDE AND CONQUER)

SOLUTION ALGORITHM



```
int partition ( int A[],int start ,int end) {
    int i = start + 1;
    int piv = A[start] ;            //make the first element as pivot element.
    for(int j =start + 1; j <= end ; j++ )  {
    /*rearrange the array by putting elements which are less than pivot
       on one side and which are greater that on other. */

          if ( A[ j ] < piv) {
                 swap (A[ i ],A [ j ]);
            i += 1;
        }
   }
   swap ( A[ start ] ,A[ i-1 ] ) ;  //put the pivot element in its proper place.
   return i-1;                      //return the position of the pivot
}
```

Now, let us see the recursive function Quick_sort :

```
void quick_sort ( int A[ ] ,int start , int end ) {
   if( start < end ) {
        //stores the position of pivot element
         int piv_pos = partition (A,start , end ) ;
         quick_sort (A,start , piv_pos -1);    //sorts the left side of pivot.
         quick_sort ( A,piv_pos +1 , end) ; //sorts the right side of pivot.
   }
}
```

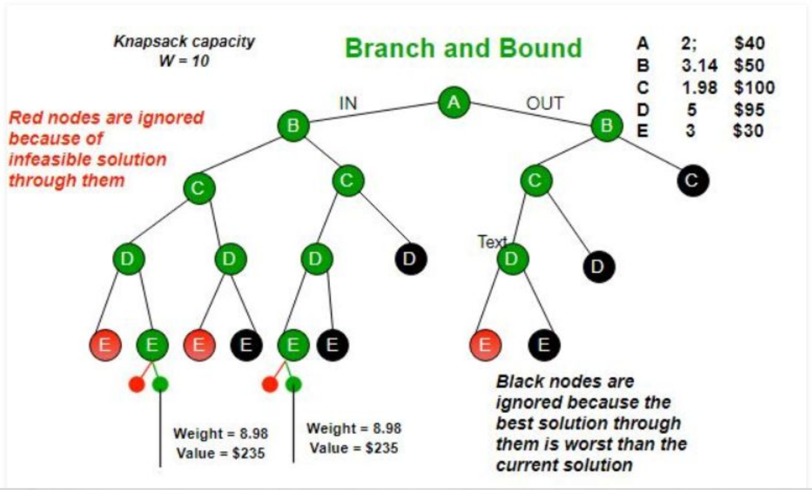
5. BOUND AND BRANCH

- **BRANCH AND BOUND** (**BB**, **B&B**, OR **BNB**) IS AN ALGORITHM DESIGN PARADIGM FOR DISCRETE AND COMBINATORIAL OPTIMIZATION PROBLEMS, AS WELL AS MATHEMATICAL OPTIMIZATION. A BRANCH-AND-BOUND ALGORITHM CONSISTS OF A SYSTEMATIC ENUMERATION OF CANDIDATE SOLUTIONS BY MEANS OF STATE SPACE SEARCH: THE SET OF CANDIDATE SOLUTIONS IS THOUGHT OF AS FORMING A ROOTED TREE WITH THE FULL SET AT THE ROOT.

EXAMPLE OF BRANCH AND BOUND

Let's see the Branch and Bound Approach to solve the **0/1 Knapsack problem**: The Backtracking Solution can be optimized if we know a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, we can simply ignore this node and its subtrees. So we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.

Example bounds used in below diagram are, A down can give $315, B down can $275, C down can $225, D down can $125 and E down can $30.


Knapsack capacity W = 10
Branch and Bound
A 2; $40
B 3.14 $50
C 1.98 $100
D 5 $95
E 3 $30
IN
OUT
Red nodes are ignored because of infeasible solution through them
Text
Weight = 8.98 Value = $235
Weight = 8.98 Value = $235
Black nodes are ignored because the best solution through them is worst than the current solution

THANK YOU

I HAVE SOLVED MORE PROBLEMS ON THE ABOVE ALGORITHMS ON MY TUBE CHANNEL “INTERVIEW AALA”