Table of Contents

1. GREEDY ALGORITHM

Greedy algorithms are simple and intuitive which is used for optimization problems. It selects the optimal choice in every step to find the optimal path to solve the problem. In some problems greedy algorithms are useful such as Huffman encoding. On the other hand, greedy algorithm does not find any optimal solutions for many problems such as find the largest sum path in a tree. The algorithm select the largest available number at each step so it does not find the largest sum because it focuses the one step to make a decision based on the current step information without consider the overall problem.

A greedy algorithm takes all the information for the particular problem and it sets a rule for elements to add the solution at every step. The greedy algorithm can be selected to solve problems if the problems contain two approaches:

Greedy Choice Feature – Optimal solution depends on the optimal choices on each step.

Optimal Structure – Entire problem's solution depends on the optimal solution on sub-problems.

To use the greedy algorithm in the problem, the first step is to identify an optimal structure or optimal subproblems in the problem. Then what the solution will contain is determined. The next step is to create an iterative path to solve all of the subproblems and build a solution. In some cases, greedy algorithms fail to find the optimal solution that is because they don't consider all of the data and the entire problem. The solution selection depends on the choices it has made so far. However, it is not aware of future choices it could make.

To sum up, a greedy algorithm always makes the choice that seems to be the best at that moment which means that it makes a locally optimal choice in the hope that this choice will lead to a globally-optimal solution.

Greedy algorithm has some pros and cons:

a. It is easy to find if a problem solve with a greedy algorithm or not.
b. Finding a run time for greedy algorithm is easier than the other algorithms.
c. It is difficult to prove that a greedy algorithm is correct. It requires more compiles with different data.

2. RECURSIVE ALGORITHM

The recursion is a process that function calls itself directly or indirectly and the corresponding function is named as a recursive function. To solve a problem with recursive:

1. Solve a sub-problem which is a smaller instance of the same problem

2. Use the solution of the first step to solve the original problem.

A recursive algorithm calls itself with smaller input values. It obtains the solution for the current input by applying simple operations to the returned value for the smaller input. A problem can be solved with a recursive algorithm when a problem can be separated into smaller problems and this smaller problems reduce to easily solvable problems. Using recursive algorithm, some problems can be solve easily such ass DFS of Graph, Towers of Hanoi.

Recursive algorithms require more memory and computation compared with the iterative algorithm. However, they are simpler. Recursive algorithm separates problem into smaller parts to solve problem by applying the same algorithm to each part and then combine the results. A recursive definition is defined in terms of itself.

3. DYNAMIC PROGRAMMING ALGORITHM

Dynamic programming is a very powerful method to solve a particular class of problems. The idea of the algorithm is that if you have a problem with input and then save the result for future reference to avoid solving the same problem again. It refers to 'remember the past'. Dynamic programming is a method for solving complex problems. The concept is to remember and make use of solution to smaller sub-problems that are solved. It makes it possible to solve problems more efficiently than recursive algorithms. It trades space for time.

Instead of solving all the solution's different states (long time and no space), it takes up space for storing solutions of all the sub-problems to save time for later. These steps are called "Memoization".

Sub-problems are parts of the original problem. Sub-problems look like a reworded version of the entire problem. Sub-problems build on each other to obtain the solution to the original problem. The process steps of dynamic programming are:

1. Identify the sub-problems in words
2. Write the sub-problem as a recurring mathematical decision.
3. Solve the original problem using Step 1 and Step 2.
4. Determine the dimensions of the memorization array and the direction in which it should be filled.

Dynamic Programming offers two different methods to solve the problem.

1. **Top-down with Memoization**
   In this method, the problem is solved by recursively finding the result to smaller sub-problems. When solving a sub-problem, the result is cached so that it ends up solving it repeatedly if it's called multiple times. The saved result can be returned. This method of storing the results of an already solved problem is called Memoization.

2. Bottom-up with Tabulation

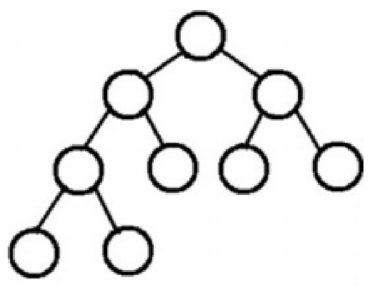
   Tapulation is the opposite of Memoization approach and it avoids recursion. The problem is solved by bottom-up which means solving all the related sub-problems first. This is typically done by filling an n-dimensional table. Based on the results that are inside of the table, the solution to the entire problem is computed.

4. QUESTION

We are given a complete binary tree where nodes and edges have positive weights. Node weights are stored in a 1-dimensional array WN. Edge weights are stored in a 2-dimensional array WE where 0 denotes no edge. Node indices are given form top-to-bottom and left-to-right (e.g. root node index is 1, its left child 2 and so on...).

Starting at the root of the tree and moving to either one of the children from the current node, the goal is to find the minimum total weight (i.e. sum of node and edge weights) path from the root to any one of the leaves.

Example:



Node weights array: ***WN*** = [3 4 2 6 1 9 8 8 5]

Edge weights array: ***WE*** =

$$\left[\begin{array}{ccccccccc}
0 & 1 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 6 & 2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 9 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]$$

Output: Min total weight path includes nodes 1-2-5 with total weight 11.

5. SOLUTIONS

1. Implement the greedy algorithm (i.e., write a function) of choosing the child with smallest sum of edge and node weights each time.

```
/** GREEDY FUNCTION **/
public static void GreedyFunc() {
   Arrays.sort(WN);

   int col;
```

```
  int nodePath;
  int parent=0;
  int weightSum=0;
  int child1=1;
  int child2=0;
  // Weight of node calculation
  while(parent<WN.length) {
    for(col = 0;col<WE[0].length;col++) {
      if(WE[parent][col]!=0) break;
    }
    if(col==9) break;
    nodePath = WE[parent][col] + WN[child1];
    if(nodePath<WE[parent][col+1] + WN[child1+1]) {
      //Left node
      weightSum+=nodePath;
      parent=child1;
    }
    else {
      //Right node
      weightSum+=WE[parent][col+1] + WN[child1+1];
      parent=child1 + 1;
    }
    child2=parent;
    child1=(parent*2)+1;
    if (child1>WN.length) {
      break;
    }
  }
  System.out.println("Total Weight is: " + (weightSum += WN[0]) + " Child Index is: " + child2 + "
Child is: " + WN[child2]);
}
```

2. Implement a recursive algorithm (i.e., write a function) to find the minimum total weight. You must determine the input parameters. Also, give the time complexity of a recursive algorithm that implements this formulation? Show your work.

```
/** RECURSIVE FUNCTION **/
static int x=0;
static int leftSum;
static int compress;
static int left=0;
public static int RecursiveFunc(int parent) {
  int[] weightSum= {0,0};
  int check=0;
  if(x==0) {
    Arrays.sort(WN);
    x++;
    check++;
  }
  int weight=0;
  int col=0;
  int child=0;
  int rightWeight=0;
  int leftWeight=0;
  for(col = 0;col<WE[0].length;col++) {
    if(WE[parent][col]!=0) break;
  }

  // Right Leaf Recursive
  if(parent*2+2<WN.length){
    child=(parent*2)+2;
    rightWeight=WE[parent][col+1]+WN[child];
    if(child*2+2<WN.length)
      if(leftSum==0)
        weightSum[1]+=RecursiveFunc(child);
      else
        weightSum[0]+=RecursiveFunc(child);

  }

  if(((parent*2+1)*2)+1<WN.length  &&  child*2+2>WN.length)
    compress=1;

  // Left Leaf Recursive
  if(parent*2+1<WN.length){
    if(check==1) {
      weightSum[1]+=rightWeight;
      leftSum=1;
    }
    child=(parent*2)+1;
    leftWeight=WE[parent][col]+WN[child];
    if(child*2+1<WN.length)
      if(leftSum==1)
        weightSum[0]+=RecursiveFunc(child);
      else
        weightSum[1]+=RecursiveFunc(child);
  }

  if(child*2+1>=WN.length &&  child*2+2>=WN.length) {
    if(leftWeight<rightWeight)
      return leftWeight;
    else
      return rightWeight;

  }

  if(compress==1) {
        if(leftWeight+weightSum[0]<rightWeight) {
            compress=0;
            return leftWeight+weightSum[0];
        }
        else {
            compress=0;
            weightSum[0]=0;
            return rightWeight;
        }
    }

    if(weightSum[0]+leftWeight<weightSum[1]) {
        weightSum[0]+=leftWeight+WN[0];
        System.out.println("Total Weight is: " + (weightSum[0]) );
        return weightSum[0];
    }
    else {
        weightSum[1]+=WN[0];
        System.out.println("Total Weight is: " + weightSum[1]);
        return weightSum[1];
    }

}
```

The time complexity is the number of operations that an algorithm performs to complete any task. The algorithm that performs the process in the smallest number of operations is considered the fastest one in terms of time complexity. These recursive algorithms divide nodes into left and right to continue to the process of the recursive algorithm so that the time complexity of it O(logn). It separates the problem into sub-problems and one by one solving them and then, it finds the correct solution using those sub-problems' solutions.

3. **Implement a dynamic programming algorithm to solve the problem. You must determine the input parameters. Also, give the time complexity of your dynamic programming solution? Show your work.**

```
/** DYNAMIC PROGRAMMING FUNCTION **/
public static void DynamicFunc() {
    Arrays.sort(WN);
    int parent=0;
    int child=1;
    int col;
    int weightSum=0;
    int rightWeight=0;
    int leftWeight=0;
```

```
  int rightMinWeight=0;
  int leftMinWeight=0;

  // Left Node
  while(child<WN.length) {
    for(col = 0;col<WE[0].length;col++) {
      if(WE[parent][col]!=0) break;
    }
    if(child==1)
      weightSum=WE[parent][col]+WN[child];
    else {
      //Same Depth
      if(((parent*2+2)*2)+2>=WN.length &&  child*2+1>=WN.length) {

leftWeight+=Math.min(WE[parent][col]+WN[child],WE[parent][col+1]+WN[parent*2+2]);
      }
      //Right Node Final Depth
      if((child*2+1<WN.length &&  ((parent*2+2)*2)+2>=WN.length)) {
        rightWeight+=WE[parent][col+1]+WN[child+1];
        leftWeight+=WE[parent][col]+WN[parent*2+1];
      }
      //After last element left tree find min element
      if(child*2+1>WN.length) {
        leftMinWeight=Math.min(leftWeight, rightWeight)+weightSum+WN[0];
      }
    }
    parent=child;
    child=child*2+1;
  }

  // Right Node
  parent=0;
  child=2;
  weightSum=0;
  rightWeight=0;
  leftWeight=0;

  while(child<WN.length) {
    for(col = 0;col<WE[0].length;col++) {
      if(WE[parent][col]!=0) break;
    }
    if(child==2)
      weightSum=WE[parent][col+1]+WN[child];
    else {
      //Same Depth
      if(((parent*2+1)*2)+1>=WN.length &&  child*2+2>=WN.length) {
        if(leftWeight==0)

rightWeight+=Math.min(WE[parent][col+1]+WN[child],WE[parent][col]+WN[parent*2+1]);
      }
```

```
            //Right Node Final Depth
            if(((parent*2+1)*2+1<WN.length &&  child*2+2>=WN.length)) {
              rightWeight+=WE[parent][col+1]+WN[child];
              leftWeight+=WE[parent][col]+WN[parent*2+1];
            }
            //After last element right tree find min element
            if(child*2+1>WN.length) {
              if(leftWeight!=0)
                rightMinWeight=Math.min(leftWeight, rightWeight);
              else
                rightMinWeight=rightWeight;
            }
            //After last element left tree find min element
            if(child*2+2>WN.length) {
              if(leftWeight!=0)
                rightMinWeight=Math.min(leftWeight, rightWeight)+weightSum+WN[0];
              else if(rightWeight!=0)
                rightMinWeight=weightSum+WN[0]+rightWeight;
            }
          }
          parent=child;
          child=child*2+2;
        }
        weightSum=Math.min(rightMinWeight, leftMinWeight);
        System.out.println("Total Weight is: " + weightSum);
}
```

In dynamic programming, the time complexity is the number of unique states or sub-problems * time taken per sub-problem. For instance, let's say a problem contains n unique sub-problems and each state is solved in a constant time so the time complexity is O(n*1) which is linear. This is the power of dynamic programming. It allows for solving complex problems efficiently. In this dynamic programming implementation, the time complexity is O(n). It is a linear function. In the algorithm, the result is stored in a memory so the same sub-problems do not require solving again and again.

4. In your main function:

```
public static void main(String[] args) {

    /** Call Greedy Algorithm */
    System.out.println("------------------------------");
    System.out.println("Greedy algorithm is running");
    long startGreedy = System.nanoTime();
    GreedyFunc();
    long endGreedy = System.nanoTime();
    long elapsedTimeforGreedy = endGreedy - startGreedy;
    System.out.println("Greedy Algorithm - Elapsed time is " + elapsedTimeforGreedy + " ns");
```

```
  System.out.println("------------------------------");

  /** Call Recursive Algorithm */
  System.out.println("Recursive algorithm is running");
  long startRecursive = System.nanoTime();
  RecursiveFunc(0);
  long endRecursive = System.nanoTime();
  long elapsedTimeforRecursive = endRecursive - startRecursive;
  System.out.println("Recursive Algorithm - Elapsed time is " + elapsedTimeforRecursive + " ns");
  System.out.println("------------------------------");

  /** Call Dynamic Programming Algorithm */
  System.out.println("Dynamic programming is running");
  //int parentindex=0,childindex=2,clmn;
  long startDynamic = System.nanoTime();
  DynamicFunc();
  long endDynamic = System.nanoTime();
  long elapsedTimeforDynamic = endDynamic - startDynamic;
  System.out.println("Dynamic Programming Algorithm - Elapsed time is " +
elapsedTimeforDynamic + " ns");
  System.out.println("------------------------------");
}
```

a. **Show that the greedy algorithm does not solve this problem optimally.**

A greedy algorithm takes all the information for the particular problem and it sets a rule for elements to add the solution at every step. The greedy algorithm can be selected to solve problems if the problems contain two approaches:

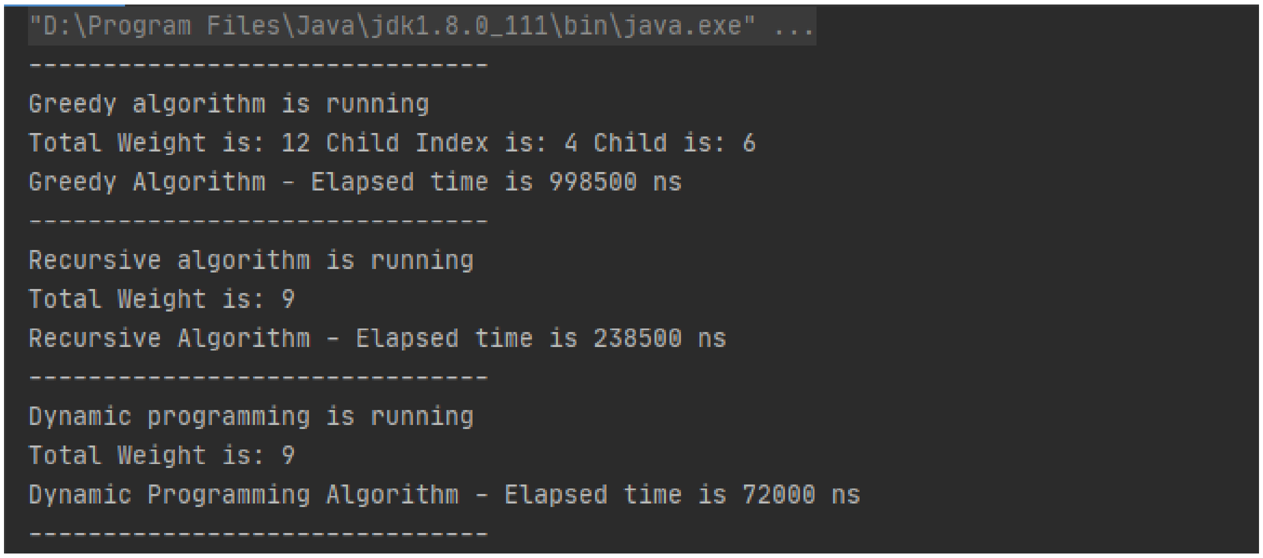
1. An optimal solution depends on the optimal choices on each step.
2. The entire problem's solution depends on the optimal solution to sub-problems.

In this question, the entire problem's optimal solution does not depend on the sub-problems. Optimal choices in each step does not gives the correct answer for every input arrays.

For example, arrays of the question are given below:

```
static int[] WN = {3,4,2,6,1,9};
static int[][] WE= {{0,1,5,0,0,0},{0,0,0,6,2,0},{0,0,0,0,0,9},{0,0,0,0,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0}};
```

In this example, the result is 9. The shortest path contains nodes 1-3 and the total weight is 9. However, using the Greedy Algorithm which is implemented above shows the solution below. The result weight is 12 and the nodes were selected as 1-2-6. In that case, greedy algorithm gives the wrong answer. It fails. That is why greedy algorithm does not solve this problem optimally.



```
"D:\Program Files\Java\jdk1.8.0_111\bin\java.exe" ...
-------------------------------
Greedy algorithm is running
Total Weight is: 12 Child Index is: 4 Child is: 6
Greedy Algorithm - Elapsed time is 998500 ns
-------------------------------
Recursive algorithm is running
Total Weight is: 9
Recursive Algorithm - Elapsed time is 238500 ns
-------------------------------
Dynamic programming is running
Total Weight is: 9
Dynamic Programming Algorithm - Elapsed time is 72000 ns
-------------------------------
```

b. **Run each of the recursive and dynamic functions with three different input sizes an compute the actual running times (in milliseconds or seconds) of these three algorithms. You will need to calculate the time passed before and after making a call to each function. Provide a 2x3 table involving the actual running times.**

In different input sizes every algorithms run in different times. The output of recursive function and dynamic programming is the same all the time because those algorithms control each nodes and paths in the tree. On the other hand, greedy algorithm sometimes gives the correct solution but sometimes it fails to find a optimal solution. The greedy algorithm focus on the process step another saying current step when running. So that, greedy selects the optimal solution for sub-problem but it does not always give the correct result.

The actual running times of those algorithms change depend on input sizes. Every time the dynamic programming algorithm is run faster than the other two algorithms. The recursive algorithm is much faster than the greedy algorithm. The slowest algorithm is a greedy algorithm. The actual running times were find as nanoseconds because the millisecond function of java gives 0 or 1 at any time. To compare differences of functions are seen more clearly with nanosecond. 10^6 ns gives 1 ms.

Comparing three algorithm running times depend on input is shown table below.

	INPUT 1	**INPUT 2**	**INPUT 3**
	static int[] *WN* = {3,4,2,6,1,9}; static int[][] *WE*= {{0,1,5,0,0,0}, {0,0,0,6,2,0}, {0,0,0,0,0,9},	static int[] *WN* = {3,4,2,6,1,9,8,8,5}; static int[][] *WE*= {{0,1,5,0,0,0,0,0,0}, {0,0,0,6,2,0,0,0,0},	static int[] *WN* = {5,7,9,11,3,4,6,9,10,1,2,7}; static int[][] *WE*= {{0,2,3,0,0,0,0,0,0,0,0,0}, {0,0,0,1,5,0,0,0,0,0,0,0},

	{0,0,0,0,0,0}, {0,0,0,0,0,0}, {0,0,0,0,0,0}};	{0,0,0,0,0,9,3,0,0}, {0,0,0,0,0,0,0,6,4}, {0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0}};	{0,0,0,0,0,9,1,0,0,0,0,0}, {0,0,0,0,0,0,0,3,2,0,0,0}, {0,0,0,0,0,0,0,0,0,1,7,0}, {0,0,0,0,0,0,0,0,0,0,0,3}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, };
Greedy Algorithm	1202800 ns	734000 ns	5069600 ns
Recursive Algorithm	245800 ns	276100 ns	644200 ns
Dynamic Programming	102300 ns	94500 ns	312600 ns

- Greedy Algorithm

Input Parameters	Actual Running Times (nanosecond)	Output
static int[] *WN* = {3,4,2,6,1,9}; static int[][] *WE*= {{0,1,5,0,0,0}, {0,0,0,6,2,0}, {0,0,0,0,0,9}, {0,0,0,0,0,0}, {0,0,0,0,0,0}, {0,0,0,0,0,0}};	1202800 nanosecond	Total weight is 12
static int[] *WN* = {3,4,2,6,1,9,8,8,5}; static int[][] *WE*= {{0,1,5,0,0,0,0,0,0}, {0,0,0,6,2,0,0,0,0}, {0,0,0,0,0,9,3,0,0}, {0,0,0,0,0,0,0,6,4}, {0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0}};	734000 nanosecond	Total weight is 11
static int[] *WN* = {5,7,9,11,3,4,6,9,10,1,2,7}; static int[][] *WE*=		

{{0,2,3,0,0,0,0,0,0,0,0,0}, {0,0,0,1,5,0,0,0,0,0,0,0}, {0,0,0,0,0,9,1,0,0,0,0,0}, {0,0,0,0,0,0,0,3,2,0,0,0}, {0,0,0,0,0,0,0,0,0,1,7,0}, {0,0,0,0,0,0,0,0,0,0,0,3}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, };	5069600 nanosecond	Total weight is 20

- Recursive Algorithm

Input Parameters	Actual Running Times (nanosecond)	Output
static int[] *WN* = {3,4,2,6,1,9}; static int[][] *WE*= {{0,1,5,0,0,0}, {0,0,0,6,2,0}, {0,0,0,0,0,9}, {0,0,0,0,0,0}, {0,0,0,0,0,0}, {0,0,0,0,0,0}};	490800 nanosecond	Total weight is 9
static int[] *WN* = {3,4,2,6,1,9,8,8,5}; static int[][] *WE*= {{0,1,5,0,0,0,0,0,0}, {0,0,0,6,2,0,0,0,0}, {0,0,0,0,0,9,3,0,0}, {0,0,0,0,0,0,0,6,4}, {0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0}};	276100 nanosecond	Total weight is 11
static int[] *WN* = {5,7,9,11,3,4,6,9,10,1,2,7}; static int[][] *WE*= {{0,2,3,0,0,0,0,0,0,0,0,0}, {0,0,0,1,5,0,0,0,0,0,0,0}, {0,0,0,0,0,9,1,0,0,0,0,0}, {0,0,0,0,0,0,0,3,2,0,0,0}, {0,0,0,0,0,0,0,0,0,1,7,0}, {0,0,0,0,0,0,0,0,0,0,0,3}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0},	644200 nanosecond	Total weight is 15

{0,0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0,0}, };		

- Dynamic Programming

Input Parameters	Actual Running Times (nanosecond)	Output
static int[] *WN* = {3,4,2,6,1,9}; static int[][] *WE*= {{0,1,5,0,0,0}, {0,0,0,6,2,0}, {0,0,0,0,0,9}, {0,0,0,0,0,0}, {0,0,0,0,0,0}, {0,0,0,0,0,0}};	102300 nanosecond	Total weight is 9
static int[] *WN* = {3,4,2,6,1,9,8,8,5}; static int[][] *WE*= {{0,1,5,0,0,0,0,0,0}, {0,0,0,6,2,0,0,0,0}, {0,0,0,0,0,9,3,0,0}, {0,0,0,0,0,0,0,6,4}, {0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0}};	94500 nanosecond	Total weight is 11
static int[] *WN* = {5,7,9,11,3,4,6,9,10,1,2,7}; static int[][] *WE*= {{0,2,3,0,0,0,0,0,0,0,0,0}, {0,0,0,1,5,0,0,0,0,0,0,0}, {0,0,0,0,0,9,1,0,0,0,0,0}, {0,0,0,0,0,0,0,3,2,0,0,0}, {0,0,0,0,0,0,0,0,0,1,7,0}, {0,0,0,0,0,0,0,0,0,0,0,3}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0,0,0}, };	312600 nanosecond	Total weight is 15

REFERENCES

1. https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/
2. https://en.wikipedia.org/wiki/Greedy_algorithm
3. https://brilliant.org/wiki/greedy-algorithm/#:~:text=A%20greedy%20algorithm%20is%20a,to%20solve%20the%20entire%20problem.
4. https://www.geeksforgeeks.org/greedy-algorithms/
5. https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/properties-of-recursive-algorithms#:~:text=Here%20is%20the%20basic%20idea,to%20solve%20the%20original%20problem.
6. https://www.geeksforgeeks.org/recursion/
7. https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/recursion
8. https://www.cs.odu.edu/~toida/nerzic/content/recursive_alg/rec_alg.html
9. https://en.wikipedia.org/wiki/Recursion_(computer_science)
10. https://developer.ibm.com/technologies/linux/articles/l-recurs/
11. https://www.sparknotes.com/cs/recursion/whatisrecursion/section1/
12. https://computersciencewiki.org/index.php/Recursion
13. https://www.codechef.com/wiki/tutorial-dynamic-programming
14. https://www.educative.io/edpresso/what-is-dynamic-programming?aid=5082902844932096&utm_source=google&utm_medium=cpc&utm_campaign=edpresso-dynamic&gclid=CjwKCAiAuoqABhAsEiwAdSkVVAlzarWmCgd9VDq7QlUyHa27JaCqACgsslq7xSDKX4YljQt21kHHxhoCp1kQAvD_BwE
15. https://en.wikipedia.org/wiki/Dynamic_programming
16. https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/
17. https://www.freecodecamp.org/news/demystifying-dynamic-programming-3efafb8d4296/
18. https://www.freecodecamp.org/news/an-intro-to-algorithms-dynamic-programming-dd00873362bb/
19. https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2G1pAq0OO0