Optimal Path Dynamic Programing

Submitted by:

Ahmed Bahaa Ibrahim 16P6057
Amr Mohamed El-Sayed Badawi 16P3034
Youssef Ossama Eid 16P8178
Basma Magdy hassanien 16P8187
Menna Allah alaa Mohamed 16P6017
Rana Fathy Mohamed 16P6070

Submitted to:

DR Gamal Abdel Shaf

Table of Contents

Introduction:

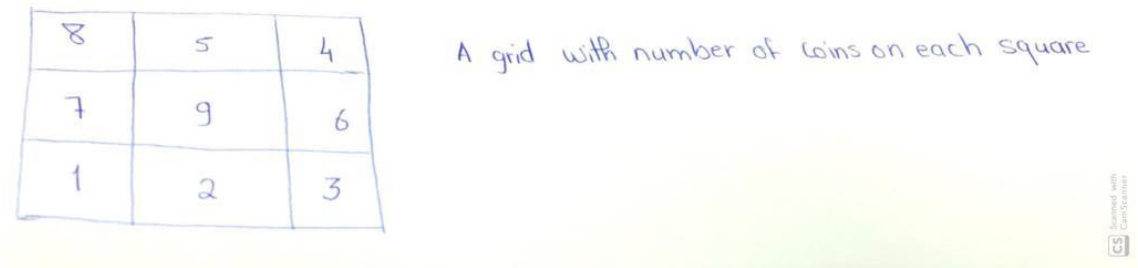
What Is Dynamic Programming?

In Brief, Dynamic Programming is a general, powerful algorithm design technique (for things like shortest path problems). You can also think of dynamic programming as a kind of exhaustive search. Ordinarily, this is a **terrible** idea, as an exhaustive search (usually) produces exponential time complexity. However, If you "brute force" carefully- you can get your algorithm down from exponential time to polynomial time- just like our greedy algorithm from earlier.

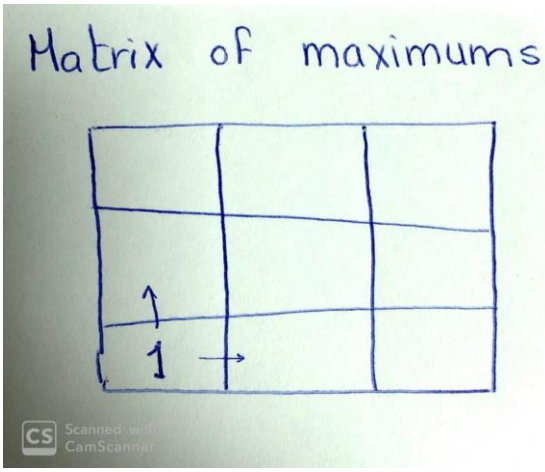
This technique involves "**memorization**' (keeping a record/memo of relevant computations). The idea here is a bit like having a scratch pad. You write down and reuse the solutions you have already computed. The solutions on your scratch pad are not problems you ultimately care about for their own sake. Rather, They are the sub-problems you would previously have solved through recursion.

Steps of the Algorithm :

Finding Optimal Paths:

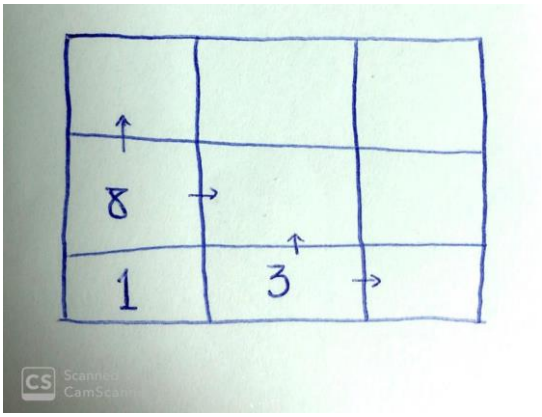


The bottom left square is (0,0) and the top left is (2,2)

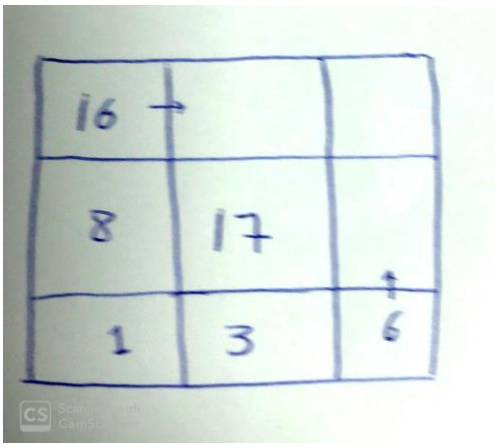


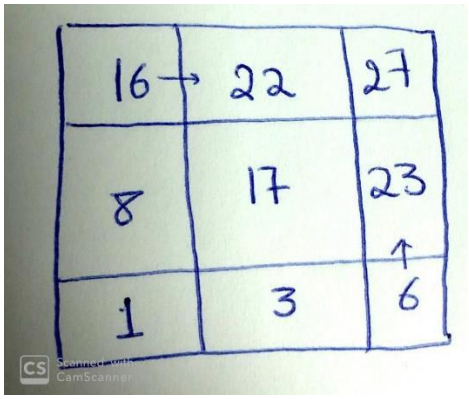
If we start at the bottom left corner and want to travel to the top right in the way prescribed, it's apparent on inspection that before our first move (while we still stand at 0,0) We would have 1 coin.

Let's record that on a memo, here that is going to take the form of another matrix which records the maximum possible number of coins for each square.



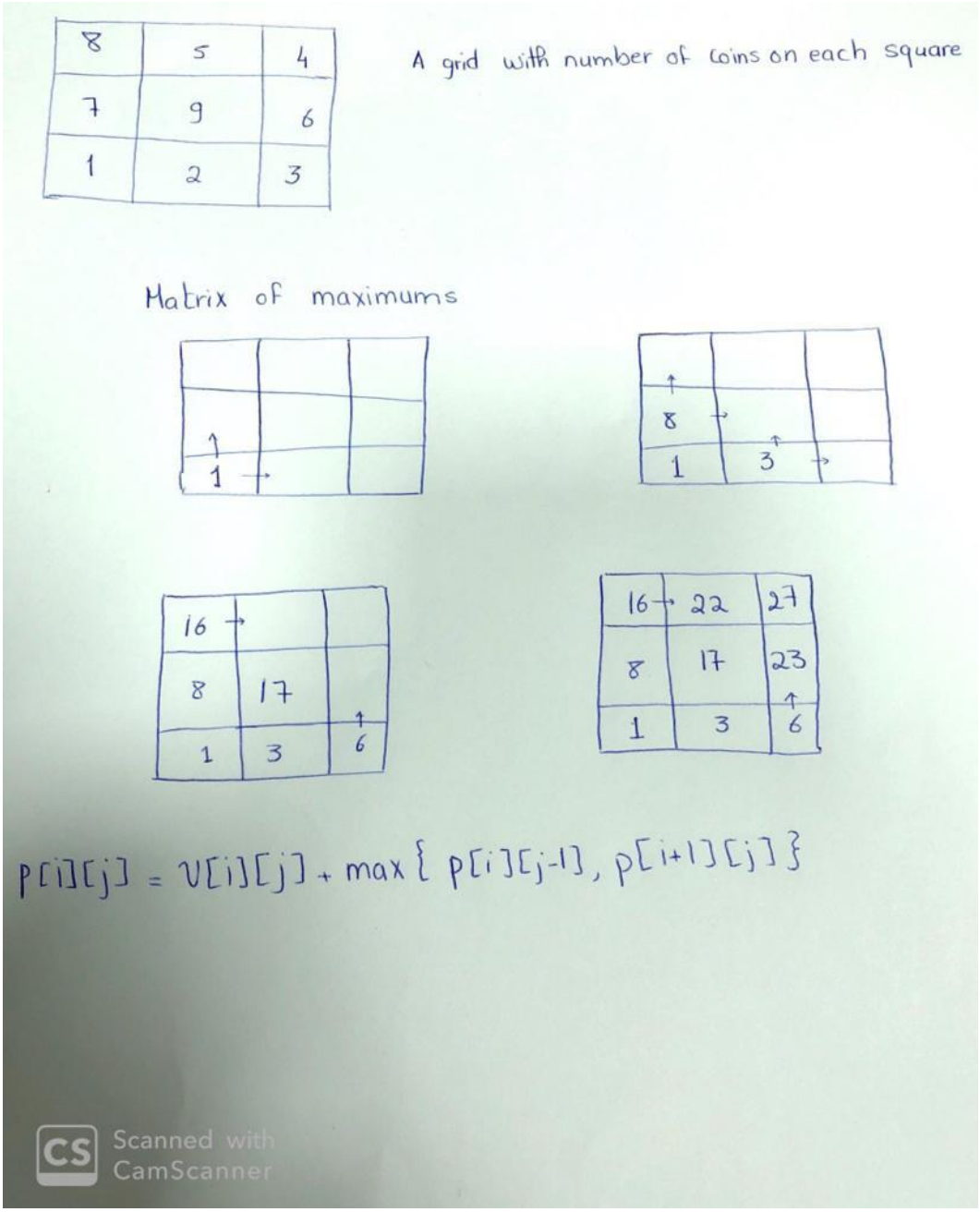
Now we have our base case, and we know the number of coins on each square- we can proceed by working out the maximums for the bottom row and column (moving in the required way, there is only one path to get to these squares)





We repeat this process with the first column then we fill in the middle to get the following complete matrix

We can then see that the maximum number of coins is 27. We did this without recursion because we never recomputed our old results.



Pseudo code:

```
function MaxCoins(A){
  //First finding the values of P and Q
  const x_axis = A.length
  const y_axis= A[0].length
  //setting up the "matrix of maximums"

  let memo=[]
  for (let i=0; i<x_axis ; i++){
     memo[i] = []
   }

  //Base Case
  memo[0][0] = A[0][0]
  //Bottom Row
  for (let i=1 ; i<x_axis; i++){
    memo[i][0] = memo[i-1][0]+A[i][0]
  }
  //Bottom Column
  for (let i=1 ; i<y_axis; i++){
    memo[0][i] = memo[0][i-1]+A[0][i]
  }
//Everywhere else
  for (let i=1; i<x_axis; i++){
    for (let k=1; k<y_axis; k++){
       if (memo[i-1][k]>memo[i][k-1]){
          memo[i][k] = memo[i-1][k]+A[i][k]
       }else {
          memo[i][k] = memo[i][k-1]+A[i][k]
       }
    }
  }
  return memo[x_axis - 1][y_axis - 1 ]
 }
```

Java code:

```
package optimalpath;

public class OptimalPath
{
   public static void main(String[] args) {
     int mat[][] = { { 8, 5, 4 },
                     {7, 9, 6 },
                     { 1, 2 , 3 } };
     // print2D(mat);
     int coins=maxCoins(mat);
      System.out.println("Max Coins : "+coins);
   }
   public static int maxCoins(int[][] A)
   {
   int row=A[0].length;
   int column=A.length;
   int[][] memo=new int[row][column];
   int start=A[row-1][0];
   int end=A[0][column-1];
   //base case
   memo[row-1][0] =start;
   //bottom row
   for(int i=1;i<column;i++)
   {
   memo[row-1][i]=memo[row-1][i-1]+A[row-1][i];
   }
   //Left Column
   for(int i=row-2;i>=0;i--)
   {
   memo[i][0]=memo[i+1][0]+A[i][0];
   }
   //Rest
   for(int i=1;i<=column-1;i++)
   {
   for(int j=row-2;j>=0;j--)
   {
```

```
    memo[j][i]= A[j][i]+((memo[j+1][i]>=memo[j][i-1])? memo[j+1][i]:memo[j][i-
1]);
    }
    }

      return memo[0][column-1];
    }
    public static void print2D(int mat[][])
    {
        // Loop through all rows
        for (int i = 0; i < mat.length; i++)
        {
            // Loop through all elements of current row
            for (int j = 0; j < mat[i].length; j++)
                System.out.print(mat[i][j] + " ");
        System.out.println("");
        }
    }
}
```

Complexity:

Tracing pseudo code to get T(n) :

```
function MaxCoins(A){
 //First finding the values of P and Q
 const x_axis = A.length.  ————————————————->1
 const y_axis= A[0].length————————————————->1
 //setting up the "matrix of maximums"

 let memo=[]
 for (let i=0; i<x_axis ; i++){
    memo[i] = [] ————————————> n-1
  }

 //Base Case
 memo[0][0] = A[0][0]
 //Bottom Row
 for (let i=1 ; i<x_axis; i++){
   memo[i][0] = memo[i-1][0]+A[i][0]————————————>n-1
 }
 //Bottom Column
 for (let i=1 ; i<y_axis; i++){
   memo[0][i] = memo[0][i-1]+A[0][i]————————->n-1
```

```
 }
//Everywhere else
 for (let i=1; i<x_axis; i++){ ——————————>n-1
   for (let k=1; k<y_axis; k++){——————————>n-I-1
      if (memo[i-1][k]>memo[i][k-1]){
         memo[i][k] = memo[i-1][k]+A[i][k]
      }else {
         memo[i][k] = memo[i][k-1]+A[i][k]
      }
   }  ———->T(n) =(n—1+1)(n-1)/2
 }
 return memo[x_axis - 1][y_axis - 1 ]————> 1
}
```

Finally,

$$T(n) = o(n^2)$$

As it’s the biggest complexity we get

When tracing pseudo code:

Worst, Best and Average Cases are all the same as We must go through all the matrix till the end point so the complexity will remain equal at any condition , which is o(n^2).

Note:

It’s not the only algorithm that finds the optimal path as Bellman Ford's algorithm is used to find the shortest paths.

Bellmanford

Code:

```
function BellmanFord(list vertices, list edges, vertex source)
   ::distance[],predecessor[]

   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) about the shortest path
   // from the source to each vertex

   // Step 1: initialize graph
   for each vertex v in vertices:
       distance[v] := inf             // Initialize the distance to all
vertices to infinity
       predecessor[v] := null         // And having a null predecessor

   distance[source] := 0              // The distance from the source to
itself is, of course, zero

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges:
       if distance[u] + w < distance[v]:
           error "Graph contains a negative-weight cycle"

   return distance[], predecessor[]
```

Complexity

$$T(n) = o(n^2)$$