Lab Week 6: Dynamic Programming

1. **Coin row revisited**: Design a dynamic programming algorithm for the "coin row problem" but change the constraint to be: You may not pick up a new coin until you have passed up at least two coins rather than one coin as in the problem discussed in the screencast. Thus, you goal is pick up the maximum amount of money picking up coins **leaving at least two coins between each picked up coin**.

   Example:

1	2	3	4	5	6	7	8	9	10	11	12
1	2	10	5	2	3	11	1	1	1	1	13

   -- the answer would be to pick up coins 3(10), 7(11), 12(13) for a total of 34.

   A. Specify the function that represents the quantity to be optimized.

   In this case let maxVal (n) represent the maximum value of a feasible solution of picking up the coins in the coin row.

   B. Give the **recurrence relation** that describes the optimal substructure of the problem using maxVal (n). Don't forget to specify the base case(s).

   $maxVal(n) = max\{maxVal(n - 3) + c_n, maxVal(n - 1), maxVal(n - 2)\}$
   base cases $maxVal(0) = 0$, $maxVal(1) = c_1$, $maxVal(2) = c_2$

   C. Give the **specification of the table** that you would use in a bottom up programmatic solution. Specify the dimensions of the table and what each entry in the table represents.

   The table will be of n + 1 length because we are adding a 0 entry. Each entry will the optimal value of picking from the first coin to the i^{th} coin, which is the table entry number.

   D. Write the **pseudo code of the algorithm** for filling in the table that you would use in a bottom up programmatic solution. That is convert the recurrence relation (part **B.**) to an **iterative** algorithm.

```
maxVal(0) = 0;
maxVal(1) = c1;
maxVal(2) = c2;
for i = 3 to n
        maxVal(i) = max{maxVal(i – 3) + ci, maxVal(i – 2), maxVal(i - 1)};
```

   E. Write the **pseudo code of the algorithm** for tracing back through the table to find the set of compatible tasks that gives the maximum total value.

```
Trace[];
i = coinset.length;
while (i > 0);
        if (maxVal(i) == maxVal(i - 1)) then
                trace.append(i - 1);
                i = i – 1;
        else if (maxVal(i) == maxVal(i - 2)) then
                trace.append(i - 2);
                i = i – 2;
        else
```

```
                trace.append(i - 3);
                i = i – 3;
    return trace reversed;
```

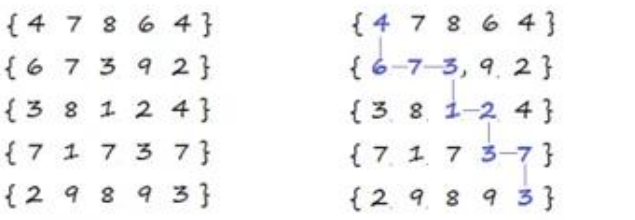
F. Write the asymptotic complexity of filling in the table.

Filling in the table takes O(n) time since we only have to go through the given list of n coins once.

2. Minimum cost corner to corner

Given an mxn matrix where each cell has a cost associated with it, find the minimum cost to reach the last cell C[m-1,n-1] of the matrix starting from the cell C[0,0]. You can only move one cell to the right or one cell down from any cell. That is for cell C[i,j] you can only move to C[i+1,j] or C[i,j+1].

For example,



The highlighted path shows the minimum cost path having cost of 36.

A. Specify the function that represents the quantity to be optimized.

In this case let minCost (i, j) represent the minimum value of a feasible solution of moving from top left to bottom right.

B. Give the **recurrence relation** that describes the optimal substructure of the problem using minCost (i, j). Don't forget to specify the base case(s).

minCost(i, j) = min{minCost(i – 1, j), minCost(i, j - 1)} + $c_{i,j}$
base cases: minCost(0, 0) = 0

C. Give the **specification of the table** that you would use in a bottom up programmatic solution. Specify the dimensions of the table and what each entry in the table represents.

The table dimensions would be m+1 x n+1 to add an additional 0 row and column. Initialize minCost(i, 0) = large number and minCost(0, j) = large number. Each entry will represent the minimum cost it took to get to that position in the matrix.

D. Write the **pseudo code of the algorithm** for filling in the table that you would use in a bottom up programmatic solution. That is convert the recurrence relation (part **B.**) to an **iterative** algorithm.

```
for i = 0 to m
        minCost(i, 0) = large number
for j = 0 to n
        minCost(0, j) = large number
```

```
for i = 1 to m //start at [1,1]
        for j = 1 to n
                minCost(i, j) = min{minCost(i – 1, j), minCost(i, j - 1)} + c_{i,j}
return minCost(m, n);
```

E. Write the **pseudo code of the algorithm** for tracing back through the table to find the set of compatible tasks that gives the maximum total value.

```
Trace[];
i = m;
j = n;
Trace.append([i, j]);
While (i > 1 and j > 1) do
        if minCost(i, j) == minCost(i, j - 1) + c_{i,j} then
                j = j - 1;
                trace.append([i, j]);
        else
                i = i - 1;
                trace.append([i, j]);
while (i != 1) do
        i = i – 1;
        trace.append([i, j]);
while (j != 1) do
        j = j -1;
        trace.append([i, j]);
return trace reversed;
```

F. Write the asymptotic complexity of filling in the table.

Filling in the table takes O(m * n) since we are going through the 2d list and filling out each value once.