# Homework 3 - training problems

Pratyay Dutta (Codeforces ID : pdutt005)

February 8, 2024

## Contents

# 1 Ski - Submission ID : 245443837

**Explanation**: To find the longest path possible in a grid of locations where the path exists only if the destination height is lesser than the source height, is basically equivalent to finding the longest path in a non weighted(cost of traveling from one place to other is constant) directed acyclic graph.
I found similarity with the SSSP problem talked about in class. Instead of finding the shortest path from a single source, we have to find the longest path from all possible sources.
The brute force method is to find the longest possible paths from each location and get the maximum of them.
In my dp implementation :
**State**: Our dp array is a 2D matrix where $dp_(ij)$ = longest possible path from location i,j. The dp matrix is initialised by putting all the values in the matrix to -1.
**Subproblem**: There can be two cases:

- If there is no adjacent location whose height is less than the current location, then the value $dp_(ij)$ remains 1.
- If there exists any such adjacent location, the longest path from the current location $dp_(ij)$ = the maximum of the longest paths from the adjacent nodes +1

**Base case**: The base case is always 1. This implies that if there is no place to go from a particular location, the maximum path length from that node is 1.
**Answer** : Maximum value in the dp matrix after construction.
In my implementation, I create a function $compute(i, j)$ which:

- If $dp_(ij)$ has a value stored, returns that value.
- If not, then, for every adjacent location which has a height lower than the current location, it computes the longest path from each adjacent node $compute(n, m)$ where $n, m$ is a subset of $adjacent(i, j)$. It stores the maximum of (the longest path from current location(if there is no other adjacent site available), $1 + compute(n, m)$ and returns that value.
- If there is no path to go from a certain node, it puts that value to 1(path length from that location is 1)

We create a *longestpath* variable and initialise it with 0. We loop over every location starting from 0,0 to r,c and update the *longestpath* variable as we compute the longest paths of each location. After the loop ends, the value of the variable *longestpath* is the longest possible path in the the given grid of locations.
**Time complexity**: We have to fill our dp matrix which has dimensions $rxc$ where r is the number of rows and c is the number of columns. To fill up the matrix, for each location, we have to make 4 computations so the number of operations is 4*r*c.
Time complexity to find the maximum value of the dp matrix = O(rc). Therefore the complexity of our algorithm is $O(4rc + rc) = O(5rc) = O(rc)$

## 2 Gridiron Gauntlet - Submission ID : 245417551

**Explanation**: We have to find the maximum time till when Wario can stay alive given the attacks and timings of each attack. This problem is like finding the length of a longest continuous path in a grid with obstacles. In this problem, there is an extra constraint which is time. This can be understood as an additional dimension and the problem can be treated as finding the longest continuous path in a 3D grid with obstacles where we can go back and forth in space but we can go only front in the time dimension. The obstacles are the rows and columns at each time stamp through which the Chucks would go through. A brute force approach will be to find the longest path possible from every point in the 3D matrix. We need to memoize this brute force method. In my dp implementation:
**State** : $s_(t, i, j)$ is the maximum time that Wario can survive till time t at a given location i,j.
**Subproblems** : We check if it is possible for Wario to stay at the current location in the current time. There are two cases:

- It is possible: We then check for the previous time stamp in all possible locations from where it could have reached the location including the case where Wario does not move. $s_(t, i, j)$ = maximum $(1 + s_(t-1, n, m)$ (where location n,m is adjacent or equal to location i,j), $s_(t-1, i, j)$ (if it not possible to stay in any adjacent or same cell, then longest time Wario can surivive till current time in the current location is the longest time he could survive till the last time stamp at the same location))
- It is not possible : $s_(t, i, j)$ value remains unchanged.

**Base case**:

1. If at time t, there is attack on row i or column j, $s_(t, i, j) = 0$ (Wario cannot survive here from the start)
2. For all other cells in the 3D matrix, it is possible to be alive for at least 1 round. Therefore, if there is no attack $s_(t, i, j) = 1$

**Answer** : Maximum value in the 3D dp matrix after construction.
In my implementation, I initialise the dp matrix with all cell values=1 (Base case 2).I first make the rows and columns of the grid. If a track number is 5 or less, then its a row else it is a column. We loop over all the attacks and set all the cells where there is attack to 0(Base case 1).
We loop through our matrix from $s_(1, 0, 0)$ to $s_(T, 5, 5)$ updating the value of each cell. According to our formulation, if $s_(t, i, j)! = 0$ (possible to survive at current location at current time) then we check for all adjacent locations at previous time stamp. I used $safe$ variable to keep track of the value in the cell of the same location at previous time stamp. If any cell is not blocked by attack then we update the value of the current cell to (value in possible adjacent cell+1). If all cells including the same one is blocked in the previous time stamp then the current cell value remains the same as the $safe$.
After, we have constructed the dp matrix, our answer is the maximum value in the matrix. Therefore we flatten the 3D matrix and find the maximum value in it. That is our answer.
**Time complexity** : The dimensions of our dp matrix is Tx5x5. To fill each cell, we need 5 computations. Therefore, time complexity to fill up dp matrix = $O(25T) = O(T)$ where T is the maximum time till chucks are released.
Time complexity to find maximum value = checking each cell in the matrix = $O(25T) = O(T)$
Therefore, total time complexity of our algorithm = O(T).

# 3 Symmetry makes perfect - Submission ID : 245459436

**Explanation** : This is essentially a problem to find the minimum edit distance to a palindrome with different costs for each operation. The logic to make a palindrome using editing is to check for each interval of each interval length in the string. If the first and last elements are equal then it has a possibility to be a palindrome and we can move to the following substring. If they are not equal then we can either add the last element before the first or the first element after the last one. In our dp implementation:
**State** : $s_(i, j)$ = number of changes to make X[i:j] a palindrome.
**Subproblems** : There are two cases:

- If X[i]==X[j], then it can be a palindrome and we move on to the next substring without changing the value because no editing is required here i.e $s_(i, j) = s_(i + 1, j - 1)$
- If X[i]!=X[j], then editing is required. The possible edits are(it is given that we can only add candies and not replace them):
  1. Add item X[i] to the end of substring X[i:j]. $s_(i, j) = s_(i + 1, j) + cost(X[i])$
  2. Add item X[j] to the beginning of the substring X[i:j]. $s_(i, j) = s_(i, j - 1) + cost(X[j])$

  The minimum of the possible edits is updated i.e $s_(i, j) = \min(s_(i + 1, j) + cost(X[i]), s_(i, j - 1) + cost(X[j]))$

**Base Case** : There can be no substring o negative or 0 length. Therefore $s_(i, j) = 0$ for i¿=j.
**Answer** : Number of changes to make entire X a palindrome i.e i=0 and j=n-1 where n is the length of the string. Therefore answer is $s_(0, n - 1)$
I did a recursive implementation of the abovementioned recursion. I made a dictionary *cost* which contains the prices of each candy. The function $symmetry_r(i, j)$ fills up the i,jth cell of the dp matrix. We initialise our dp matrix to -1. If the value of dp[i][j] is not -1, that means it has been calculated and recorded before, that value is returned. If i¿=j, value 0 is returned (base case). According to our formulation, if the candies at the extremities of the sub string are equal, then no edit is required and we return the edit distance for the substring $s_(i + 1, j - 1)$ after saving the value. If the candies at extremities are unequal then edit is required and we store and return the minimum edit distance for :

- Placing the first candy at the end of the substring and returning the cost for editing substring $s_(i + 1, j)$+cost(first candy).
- Placing the last candy at the beginning of the substring) and returning the cost for editing substring $s_(i, j - 1)$+cost(last candy)

Our answer is the element $s_(0, n - 1)$. Therefore to get our answer, we find $symmetry_r(0, n - 1)$
**Time complexity** : We have to fill up the dp matrix which has dimensions nxn where n is the length of the string. Time complexity to fill up dp matrix = $O(n^2)$. Time complexity to find the value of a known cell in the matrix after construction = O(1). therefore, total time complexity = $O(n^2)$.