# Homework 7

● Graded

**Student**

Abhishek Soundalgekar

**Total Points**

**40 / 40 pts**

**Question 1**

**P1** **10** / 10 pts

✔ **– 0 pts** Correct

**– 2 pts** Incorrect Subproblem definition

**– 3 pts** Incorrect Recurrence relation

**– 2 pts** Incorrect Pseudocode

**– 1 pt** Incorrect Base case

**– 1 pt** Incorrect Final answer location

**– 1 pt** Incorrect Time complexity

**Question 2**

**P2** **10** / 10 pts

✔ **– 0 pts** Correct

**– 2 pts** Incorrect subproblem definition

**– 3 pts** Incorrect recursive relation

**– 2 pts** Incorrect/Missing pseudocode

**– 1 pt** Incorrect/Missing Base case

**– 1 pt** Incorrect/Missing Final answer

**– 1 pt** Incorrect/Missing Time Complexity

**Question 3**

**P3** **10** / 10 pts

✔ **– 0 pts** Correct

**– 3 pts** Wrong sub-problem

**– 3 pts** Wrong recurrence

**– 2 pts** Wrong base cases

**– 2 pts** Wrong time complexity

**– 10 pts** Wrong

**Question 4**

**P4** **10** / 10 pts

**– 2 pts** Part 1 incorrect

**– 3 pts** Part 2 incorrect

**– 2 pts** Part 3 incorrect

**– 1 pt** Part 4 incorrect

✔ **– 0 pts** Correct

Question assigned to the following page: 1

CSCI 570 Spring 2025

Homework 7

Name: Abhishek Soundalgekar

USC ID: 2089011000

Question 1

Answer 1

1.1 Subproblems to be solved:

We need to calculate the number of valid paths to each tile $(j, i)$ in the grid, where $j \in \{1, 2, 3\}$ denotes the row and $i \in \{1, 2 \dots n\}$ denotes the column.

Let OPT[j][i] represent the number of ways to reach tile $(j, i)$ without stepping on weak tiles. Given by the badTile[j][i] value (0 for strong tile, 1 for weak tile) if the movement is permitted.

The base case initialize paths from the starting column and subsequent columns build on these values.

Question assigned to the following page: 1

1.2 Recurrence Relation for the subproblems :

Every column where $i \geq 2$ and row $j$ :
if Bad Tile $[j][i] = 1$, then $OPT[j][i] = 0$ (tile is weak)

else :
For $j=1$ ; $OPT[1][i] = OPT[1][i-1] + OPT[2][i-1]$

For $j=2$ ; $OPT[2][i] = OPT[1][i-1] + OPT[2][i-1] + OPT[3][i-1]$

for $j=3$ ; $OPT[3][i] = OPT[2][i-1] + OPT[3][i-1]$

1.3 Pseudocode :

Initialize $OPT[3][n]$ as a $3 \times n$ array filled with 0

```
for j from 1 to 3:                  } // Base Case
    if Bad Tile [j][1] == 0;        } // first column
        OPT[j][1] = 1               } // i=1
```

Base cases :
For the first column ($i=1$) :
$OPT[j][1] = 1$ if Bad Tile $[j][1] = 0$ (tile is strong)
else 0.

Question assigned to the following page: 1

```
// Fill the OPT table for column 2 to n

for i from 2 to n:
    for j from 1 to 3:
        if BadTile[j][i] == 1:
            OPT[j][i] = 0

        else:
            if j == 1:
                OPT[j][i] = OPT[1][i-1] + OPT[2][i-1]
            elif j == 2:
                OPT[j][i] = OPT[1][i-1] + OPT[2][i-1]
                          + OPT[3][i-1]
            else: // j == 3
                OPT[j][i] = OPT[2][i-1] + OPT[3][i-1]
        end for
end for

total = OPT[1][n] + OPT[2][n] + OPT[3][n]
return total
```

The total number of valid paths is the sum of OPT[1][n], OPT[2][n] and OPT[3][n] (the last column of the OPT table). This totals for all paths ending at strong tiles in the final column.

Question assigned to the following page: 2

Question 2
Answer 2

2.1 Subproblems to be solved

We need to determine the maximum coins obtainable from bursting all balloons within a given interval. Let OPT[i][j] represent the maximum coins achievable by bursting all balloons between indices i and j (exclusive) in a modified array that includes virtual boundary balloons with values 1 at both ends.

$$newnum = [1 + nums[0], nums[1], \ldots nums[n-1], 1]$$

2.2 Recurrence Relation

Any interval $(i, j)$ with $j \geq i+2$ (atleast 1 balloons in between) the last balloon to burst in this interval is balloon $k$ $(i < k < j)$. When balloon $k$ is the last to burst, the coins collected from that action are:

$$newnum[i] \times newnum[k] \times newnum[j]$$

Here, the balloons at $i$ & $j$ are still intact. In addition, we would have already optimally burst the balloons in the interval $(i, k)$ & $(k, j)$.

$$OPT(i,j) = \max_{i<k<j} \{OPT(i,k) + OPT(k,j) + [newnum[i] \times newnum[k] \times newnum[j]]\}$$

Question assigned to the following page: 2

2-3 Pseudocode :

```
Initialize n = length(nums)

newnum[0] = 1          } // new array of size n+2
newnum[n+1] = 1        }

for i from 1 to n:
    newnum[i] = nums[i-1]

// initialize OPT table of (n+2) x (n+2)
// OPT[i][j] represents max coins for bursting ballons(i, j)

create 2D array OPT[0 ... n+1][0 ... n+1] and set
all entries to 0
// m is length of interval (j-i)
for m from 2 to n+1:
    for i from 0 to n+1-m:
        j = i + m
        // k is last balloon to burst between i and j
        for k from i+1 to j-1:
            OPT[i][j] = max(OPT[i][j], OPT[i][k] +
            OPT[k][j] + newnum[i] * newnum[k] * newnum[j])
        end for
    end for
end for
return OPT[0][n+1]  // Final answer at OPT[0][n+1]
```

Question assigned to the following page: 2

2.4 a) Base case:

For every $i$, set:

$OPT(i, i+1) = 0$ (since no balloon between $i$ & $i+1$)

b) Final Answer at $DP(0, n+1)$

$n \rightarrow$ no. of original balloons.

2.5 Complexity Analysis

Time complexity:

3 nested for loops:

total time complexity is $O(n^3)$

Space complexity:

A 2D OPT table of size $(n+2) \times (n+2)$ is used, so the space complexity is $O(n^2)$

Question assigned to the following page: 3

Question 3
Answer 3

3.1 Sub Problems to be solved :

We will compute the number of ways to form the first i elements of array B using the first j elements of array A.

Let OPT [i][j] represent this count. Here, i ranges from 0 to m (length of B), and j ranges from 0 to n (length of A). This captures all possible partial matches between B and subsequences of A.

3.2 Recurrence Relation & Base case :

if A[j-1] == B[i-1] then:
    OPT [i][j] = OPT [i-1][j-1] + OPT [i][j-1]
(Either include A[j-1] to match B[i-1] or exclude it).
else :
    OPT [i][j] = OPT [i][j-1]  (Exclude A[j-1]).

Question assigned to the following page: 3

Base cases:

OPT [0][j] = 1 for all j (one way to form an empty B by deleting all elements of A).

OPT [i][0] = 0 for all i > 0 (no way to form a non-empty B from an empty A).

3.3 Runtime complexity:

The above algorithm runs in O(mn) time. The OPT table has dimensions (m+1) × (n+1), and each entry OPT [i][j] is computed in constant time using the recurrence. Thus, the total operations are proportional to m * n ensuring an optimal solution.

Space complexity:

OPT [n][m] table is a 2 dimensional array of size n×m giving us the overall space complexity of O(nm)

Question assigned to the following page: 4

Question 4
Answer 4

→ An integer array A of size N, where each element in the array satisfies $1 \leq A[i] \leq N$

→ Determine the longest consecutive subsequence that can be formed from A

→ Design an efficient Dynamic Programming algorithm to compute the maximum length of any consecutive subsequence in A.

4.1] Sub problems to be solved :

We need to find the length of the longest consecutive subsequence in the given array A.

Therefore, for each element 'i', we define the subproblem as follows :

OPT [i] = the length of the longest consecutive subsequence ending at index 'i'.

We check for previous indices $j < i$ where $A[j] = A[i] - 1$

Question assigned to the following page: 4

4.2 Recurrence Relation:

if A[i]-1 already exists in the OPT table
then OPT [A[i]] = OPT [A[i]-1] + 1
else
OPT [A[i]] = 1

Use hash map here, it allows us to efficiently store and retrieve the values.

4.3 Iterative Pseudocode:

Base Case
If N=0 ie; the array is empty we return 0
If N=1 then return 1 (length of the longest subsequence)

Pseudocode

Initialize OPT table as an empty hash table and longest_sequence = 0

Question assigned to the following page: 4

```
for i=0 to N-1:
    if A[i]-1 exist in OPT
        then OPT[A[i]] = OPT[A[i]-1]+1
    else
        OPT[A[i]] = 1
    longest_sequence = max(longest_sequence, OPT(A[i]))
return longest_sequence
```

longest_sequence will return the maximum length of any consecutive subsequence in A.

4.4] Complexity Analysis

Iterating through array A only once, and for each element, we check if A[i]-1 exist in the OPT table & update the OPT (hash map) value for A[i] → this takes O(N) time
The total time complexity is O(N)

Space Complexity

We create an OPT hash map that stores N entries
Space complexity : O(N)