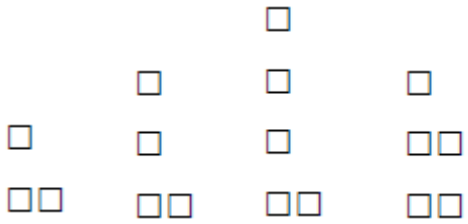
DESIGN AND ANALYSIS OF ALGORITHMS
(CS-2009)
SEMESTER PROJECT SPRING 2024

Part A:- Hassan Imran (22i-0813)
Part B:- Hamza Ahmad (22i-1003)

PROBLEM #1 (Structure arrangements with N Blocks)



The solution utilizes dynamic programming to efficiently count the number of valid structures given the number of blocks *n*. The algorithm defines subproblems based on the maximum height of the tower, allowing for the calculation of valid structures for varying heights.

A bottom-up approach has been employed, iterating through all possible heights of the maximum tower and updating the number of valid structures for each height. The algorithm utilizes a dynamic programming table *dp* of size *n + 1* to store the number of valid structures for each possible number of blocks, optimizing space usage

Within the nested loop, the algorithm updates a running total of possible structures while ensuring that the current maximum tower's height is not exceeded.

The DP table is filled backward to ensure that subproblems are not updated for the current maximum height until they have been utilized, allowing for accurate counts of valid structures.

The inner loop only updates from blocks down to maxTower because it is not possible to make a tower of height greater than the available blocks.

The function returns the total count of valid structures minus 1 to exclude the empty structure, as it is assumed to be a valid structure initially.

Pseudo Code

```
countStructures ( n )
    dp = array of size (n + 1) // initialized with 0
    dp[0] = 1  //assume 0 tower valid as subproblem

    // Iterate through all possible heights of the max tower
    for maxTower = 1 to n:
        blocks = n to maxTower:
            dp[blocks] += dp[blocks - maxTower]

    return dp[n] - 1 // exclude the empty structure
```

- We iterate all possible heights for the first tower i.e. 1 to *n.*
- For a certain height of max tower, let's say 5, (considering case of 10 blocks).
    - We now have 10 - 5 = 5 remaining blocks.
    - Height of the next tower can be 4 at maximum.
- Hence, we check out list to see how many combinations can be made for 5 blocks where maximum tower does not exceed 4, and add this number to our running count for 10 blocks so far.
- The formula was used:
    - ***dp[blocks] += dp[blocks - maxTower]***
- At this point so far we have counted the case (4,3,2,1) so the value of *dp[10]* is 1.
- *dp[5]* returns 2 since (4,1) & (3,2).
- So, the subproblems are used such that the max tower (5) is combined with these solved subproblems to make (5,4,1) & (5,3,2).
- Iterations go on similarly calculating for all heights.

Example with n =10

Update log of the list

	init	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
[0]	1	1	1	1	1	1	1	1	1	1	1
[1]	0	1	1	1	1	1	1	1	1	1	1
[2]	0	0	1	2	1	1	1	1	1	1	1
[3]	0	0	1	2	2	2	2	2	2	2	2
[4]	0	0	0	1	2	2	2	2	2	2	2
[5]	0	0	0	1	2	3	3	3	3	3	3
[6]	0	0	0	1	2	3	4	4	4	4	4
[7[	0	0	0	0	2	3	4	5	5	5	5
[8]	0	0	0	0	1	3	4	5	6	6	6
[9]	0	0	0	0	1	3	5	6	7	8	8
[10]	0	0	0	0	1	3	5	6	8	9	**10**

The final result is returned by **(dp[n] - 1)** because of our initial assumption

Complexity Analysis

- The outer loop iterates over all possible heights of the maximum tower, ranging from 1 to n, hence having complexity *n*.
- The inner loop iterates from *n* down to the current *maxTower*, updating the dynamic programming table, and therefore, also having time complexity *n*.
- $\therefore n(n) \Rightarrow \mathbf{O(n^2)}$

PROBLEM #2 (Rail track Bombing)

The best way to explain this solution is by illustrating an example.

Considering this track with 2 Bombs available.
We consider attacking one by one on all positions of the current track using the for loop:

(4) | **(5)** **(1)** **(2)**

Similarly left to right we test placing second bomb on every index

(5) | **(1)** **(2)** and **(5)** **(1)** | **(2)**

The next call is made, reducing number of bombs

(1) **(2)**

/// since there is no bomb left we have reached the base case and we return the corresponding strategic value from out table i.e. *i*2*

(2) // in this case we are on the last depot and start = end so we return the value.

- Left subproblems are directly retrieved from the table
- Similarly, all the calls are made and the iteration with bomb that yields minimum value i.e. with the most optimal attack returns
- Successive returns with the optimal attacks result in the maximum decrease in strategic value of the railroad

Complete dry run:

4 | 5 1 2

5 | 1 2 5 1 | 2

0 **2** 5 0

4 5 | 1 2

20 1 | 2

0 0

4 5 1 | 2

29 | 0

Successive minimums of sub problems add up to give the minimum strategic value for the bigger attack.

As can be traced in the example, the remaining MSV after 2 optimal attacks is 2.

We can trace it back and tell that this occurs when we place bombs the following way:

(4)= | =(5)= | =(1)==(2)

So only depots (1)==(2) remain .

Description of Sub-problems and Optimality Substructure:

Parameterization of Sub-problems: The sub-problems are parameterized by three variables: bombs, start, and end, representing the number of remaining bombs, the starting index of the current sub-problem, and the ending index of the current sub-problem, respectively.

Overall Goal Reflection:

The overall goal is reflected in these sub-problems by recursively solving smaller

sub-problems (with fewer bombs and smaller segments of the railroad) and combining their solutions to find the optimal strategy for the entire railroad segment.

Justification of Optimality Substructure: The optimality substructure is present because the optimal solution to the larger problem can be constructed from the optimal solutions of its sub-problems. This is evident as the minimum strategic value for a given segment can be obtained by considering all possible attacks and choosing the one that yields the minimum value, which is recursively determined by optimal solutions to smaller sub-problems.

Recursive Formula:

```
for (int i = start; i < end; ++i) {
   int strategicValue = value[start][i] + minStrategicValue (bombs - 1, i + 1, end);
   minimumValue = min(minimumValue, strategicValue);
}
return minimumValue;
```

This formula calculates the strategic value for each possible attack position by recursively combining the strategic values of left and right sub-problems, considering the placement of bombs at different positions along the railroad track.

Base Case :

```
if (bombs == 0 || start == end)
    return value[start][end];
```

//Either the bombs finish or the end of the railroad has been reached

Order of Solving Sub-problems:

The pre-processing step follows a bottom-up approach to populate the value table efficiently; the actual solution of the problem using the MinStrategicValue function employs a top-down recursive strategy.

Pseudo Code

```
int n, m;
Depots[n];
Value[n][n]; // dp table for subproblems

// Bottom-up DP to populate the value matrix
// Calculate all possible subproblems
// pre-processing

// Fill diagonal with zeros
for i from 0 to n - 1:
   value[i][i] = 0;

// Fill the rest using the formula
for l from 1 to n - 1:
   for i from 0 to n - l - 1:
      j = i + l;
      value[i][j] = depots[i] * depots[j] + value[i + 1][j] + value[i][j - 1] - value[i + 1][j - 1];

// Call Function after processing
minStrategicValue = minStrategicValue(m, 0, n - 1);

minStrategicValue(bombs, start, end):
   // Base case: bombs finished or last depot reached
   if bombs == 0 OR start == end:
      return value[start][end];

   minimumValue = INFINITY;
```

```
// Iterate over all possible attack's positions for current problem
for i from start to end - 1:
   strategicValue = value[start][i] + minStrategicValue (bombs - 1, i + 1, end);
   // left subproblem retrieved from table
   // right subproblem returns
   minimumValue = min (minimumValue, strategicValue);
   // combined with right subproblem we get the minimum considering our first attack was at i
   // recursively we repeat returning minimum strategic values
   // hence we end up with the most optimal attack

return minimumValue;
```

Complexity Analysis:

- **$O(n^2 \times m)$**
- The space complexity is primarily determined by the space required to store the value matrix, resulting in **$O(n^2)$** space usage.

PROBLEM #3 (Minimum Not Presented Number in sub-arrays)

{ 1 2 1 2 1 2 0 1 2 1 2 3 3 2 1 0 }

For this problem, a bottom-up approach has been adopted. The algorithm iterates through the array twice; the first time from left to right and the second time from right to left. The purpose is to identify the mark at which the cut is to be made such that the MNPN in the two resulting segments is the same (only two segments are being made).
3 additional arrays are required:

1) *m1* (of size *n+1*. This will store the MNPNs as the loop iterates from left to right)
2) *m2* (of size *n+1*. This will store the MNPNs as the loop iterates from right to left)
3) *pn* (Present Numbers. The size is *n+2*, initialized with 0s. It is marked 1 for each index that is discovered in the input array. This array will act as our dynamic programming array as it will ensure that present numbers are not marked as MNPNs.)

Thus, making the total space complexity of the algorithm *4(n+1)*.
segment1 and *segment2* can be considered structure objects, made to store the indices of the segments.

(*Note: the names of the arrays and variables may be different in the source code. The above notations are for the pseudocode*)

Pseudo Code

```
segmentsForMNPN ( array, n )
START
    mnpn ← 0
    pn ← {0}

    for i ← 0 to n -1
        pn[array[i]] ← 1
        WHILE mnpn is in pn
            INCREMENT mnpn
        m1[i] ← mnpn


    mnpn ← 0
    pn ← {0}
    for i ← n -1 to 0
        pn[array[i]] ← 1
        WHILE mnpn is in pn
            INCREMENT mnpn
        m2[i] ← mnpn

    reverseArray ( m2, n+1 )

    for i ← 0 to n
        IF m1[i] = m2[i+1]
            segment1 ← (0, i)
            segment2 ← (i+1, n)
            PRINT 2, segment1, segment2
            END
    PRINT -1
END
```

Complexity Analysis

- Both the forward and backward loops run *n* times.
- In the worst-case event of an array like [*8, 7, 6, 5, 4, 3, 2, 1, 0*], the MNPN will be incremented *n+1* times only once per loop.
- This, as a result, makes the worst-case complexity of each loop *n + (n + 1)*, and therefore *2n + 1*.
- Consequently, the added worst-case complexity of these two loops becomes *2(2n+1)*, and therefore *4n + 2*.
- The MNPN array obtained from iterating through the array backwards i.e. *m2*, is reversed in order to facilitate comparison. The complexity of array reversal is *n/2*.
- Moving on to the final loop which identifies the cutoff point of the two segments, in the worst-case i.e. if the MNPN does not match for the entirety of the two arrays, the complexity becomes *n*.
- $\therefore (4n + 2) + (n/2) + (n) \Rightarrow$ **O(n)**

PROBLEM #4 (Maximum Power Level across 2-D Matrix)

11	5	**7**	9	**5**
7	**10**	3	**6**	7

The algorithm makes use of a 1-D array *u* of size *r+1*, where *r* is the number of rows in the matrix. This serves as our dynamic programming array, as the solutions are kept track of as the route is formed

By iterating column-wise throughout the matrix and adding the values of each row to *u*, a new matrix *Q* is generated in each iteration.

At the $(c\text{-}1)^{th}$, we can extract the maximum value from *u* to obtain our optimal solution.

Pseudo Code

```
getMax (matrix, rows, cols )
    START
        IF rows ==1
            PRINT max (matrix[0])
        END

    APPEND row of 0s to matrix
        u ← first column of matrix
        for c ← (1 to cols - 1):
            Q ← empty matrix
            for r ← 0 to rows
                Q[r][c] = u[r] + matrix [r][c+1]
            Diagonal values of matrix ← -1
            for i ← 0 to rows+1
            u[i] = max(matrix[i])
        PRINT max ( u )
        END
```

Complexity Analysis

- Let number of rows = *M*, number of columns = *N*
- An additional row of zeroes of size *N* is appended to the matrix to facilitate row skipping. The time complexity of this operation is *O(1)*.
- *u* is initialized with the value of the first column of the input matrix. As *M* values are copied, the loop runs *M* times.
- The outer loop starts from the second column (the first column has already been covered in *u*) and runs until the second last column has been traversed, thereby having complexity *N – 2*.
- The first inner loop runs *M+1* times, accounting for the additional row of zeroes appended to the matrix. This loop adds the values of each row to *u*.
- The second inner loop stores the maximum value of each row in *u*. This is done in (*N* x *M*) steps.
- After the outer loop has run its course, another loop of size *M* is needed to obtain the maximum value from *u*.
- $\therefore 1 + M + (N - 2)((M+1) + (M \times N)) + M \Rightarrow \mathbf{O(M \times N^2)}$