Theory Assignment-1: ADA Winter-2023

Anay Chauhan (2021013) Pranjal Bharti (2021080)

---

1. **Question 1** Consider the problem of putting L-shaped tiles (L-shaped consisting of three squares) in an n × n square-board. You can assume that n is a power of 2. Suppose that one square of this board is defective and tiles cannot be put in that square. Also, two L-shaped tiles cannot intersect each other. Describe an algorithm that computes a proper tiling of the board. Justify the running time of your algorithm.

Ans: This problem can be solved by using a divide and conquer algorithm. Before constructing the algorithm however, we will first list the given details, constraints and variables.

Given:-
The dimensions of the board is n X n squares where n is a power of 2.
The size of the tile L is 3 squares.
Minimum size of the board is 2 X 2.
Assume that the input is of the following form:-
1.n(Size of the side of the board)
2.i j(Consider that the board is a 2d-matrix A with all squares with an initial value of 0 except A[i][j] which is initialized to -1 representing the missing square on the board)

Algorithm:-
Since we are constructing a divide and conquer algorithm, we will have to divide the problem into smaller subproblems and also have to identify a base case for the recursion.

Preprocessing:- Before this however, we will have to find out the position of the 1 missing tile. Since we have assumed that we are given the coordinates of the missing square as an input, we don't have to actually find the position of the hole. Therefore, we will simply initialize the value of A[i][j] to -1. (Non-zero value means either the square is missing or has already been covered)

Base case:- The base case will be the one where the size of the board is 2 X 2. In this case, we can simply place a single L-shaped tile on the board depending on the position of the missing tile.
Pre-division step:-Now we will place the L-shaped tile at the centre of the board depending on the position of the empty/missing square. For eg.- if the missing square is in the first quadrant, place the tile in such a way that the tile will cover one square each in the 2nd, 3rd and 4th quadrants respectively. Once we have placed this tile,

we can update the value of the squares which have been covered by this tile to 1.(In the recursive step, when another tile is placed over 3 other squares, the value of these squares will be updated to the number of the respective tile. For eg.- if squares A[1][1], A[1][2] and A[2][2] are covered by a L shaped tile on the second call of this function, these 3 squares will be updated to 2)
Divide step:-Now we have a board with 1 square in each quadrant with a single missing or already covered tile. Therefore, we will divide the board into 4 smaller boards(subproblems) of dimension - (n/2) X (n/2)
Conquer step:-Now, we will recursively solve these subproblems by recursively calling the function. For each quadrant, we will have to first find the location of the covered/missing square and place a tile accordingly, update the values of these newly covered squares and then once again divide the board into 4 smaller boards.

Running time of algorithm:-
1.The base case itself will take O(1) time since we are placing a single tile in a single possible manner.
2.The division into the 4 subproblems/boards also takes O(1) time.
3.Recursively solving these 4 subproblems will be T(n/2) each, considering the total running time of the algorithm on the initial board is T(n).

From the above statements, we get the following recurrence relation:-
T(n) = 4T(n/2) + C

Applying Master's Theorem on this relation, we obtain the running time of the algorithm as $O(n^2)$.

2. **Question 2** Suppose we are given a set L of n line segments in 2D plane. Each line segment has one endpoint on the line y = 0, one endpoint on the line y = 1 and all the 2n points are distinct. Give an algorithm that uses dynamic programming and computes a largest subset of L of which every pair of segments intersects each other. You must also give a justification why your algorithm works correctly.

Ans: Since we have to solve this question using dynamic programming, we will first have to construct a Recurrence relation and then try to use memoization in order to optimise the time complexity and make our algorithm faster. Before we construct the algorithm, we will first list the given details and inputs.

Given:-
Assume that the input is of the following form:-
1.n(number of line segments)
2.n pairs of the form 'i j'(where i is the x coordinate of the line segment lying on y = 0 and j is the x coordinate of the line segment lying on y = 1.

Algorithm:-
The main algorithm which we will use is the algorithm for computing the Longest Increasing Subsequence. However, before executing this step, we will first have to go

through a preprocessing step.

Preprocessing:-

1. First we will have to create two arrays y0 and y1, each of length n, inside which we will store the x coordinates lying on the lines y=0 and y=1 respectively.
2. Now we will choose one of these 2 arrays for sorting. Let's choose y1 in this case.
3. Now we will sort this array(y1) in descending order and after this, we will have to shuffle the other array(y0) in such a way that the corresponding x coordinates of the line segment lying on y=1 and y=0 are at the same indices in both the arrays.
4. For example - if inputs are as follows:
   2 4
   3 3
   4 5
   We will have two arrays y0 = [2,3,4] and y1 = [4,3,5]. On sorting y1 in descending ordering we get y1 = [5,4,3]. Finally, we will shuffle y0 accordingly to obtain y0 = [4,2,3].
5. Now, we simply have to apply LIS algorithm on the array y0.

The Longest Increasing Subsequence algorithm is as follows:

Define problem:-For any $1 \leq m \leq n$, define LIS(m) to be the length of the longest increasing subsequence in A[1 : m] which ends with A[m]. We are interested in $max_{1 \leq m \leq n} LIS(m)$.

Base case:-LIS(0) = 0.

Recursive formula:-For all $m \geq 1$:
LIS(m) = $1 + max_{1 \leq j < m A[j] \leq A[m]} LIS(j)$

Subproblem:-The subproblems in this algorithm will be as follows:
Given A[1 : m] suppose we ask not for an LIS in A[1 : m], but for the longest increasing subsequence that ends with A[m].(where m is less than n)

Pseudocode:-

Now we will construct a dynamic programming algorithm to solve the above Recursive formula efficiently.

This can be done as follows:

procedure LIS(A[1 : n]):
2: //Returns the Longest Increasing Subsequence

3: Allocate space L[0 : n] //L[m] will contain LIS(m).
4: //Maintain parent[1 : n] which is useful for recovery. Initialized to 0
5: L[0] = 0. // Base Case.
6: for $1 \leq m \leq n$ do:
7: L[m] = 1 + $max_{1j<mA[j]\leq A[m]}$ L[j] //Naively, takes O(n) time.
8: Set parent[m] to be the j which maximizes. If no such j, parent remains 0.
9: m = arg maxL[1 : n] // L[m] contains the length of the optimal LIS
10: Define a to be the reverse of (A[m], A[parent[m]], A[parent[parent[m]]], . . . ,) till parent becomes 0.

Applying this Longest Increasing Subsequence algorithm on the array y0 will essentially give us the number of line segements l from the given line segments such that each one of them intersects the others.

Running time of algorithm:- Without using Dynamic programming, if we sought out to find the number of Longest Increasing Subsequences in the given array, it would have taken $O(2^n)$ time. However, on using the DP algorithm as mentioned above(similar to the one taught in class), we can obtain the result of just the LIS algorithm in $O(n^2)$ time.
Apart from this, we will also require O(nlogn) time to sort the array y1 using Mergesort Algorithm.

This gives us the resultant running time as $O(n^2)$.

Justification:-
We can use induction to prove the correctness of our algorithm. Let's take the base where just a single line segment is present. In this case, when we sort the array y1 in descending order and then calculate the LIS of the corresponding array y0, we will get the answer as 0 which is correct since the single line segment is not intersecting any other line segment.
Let us assume that this algorithm holds true for a set of n line segments. We now have to prove it's correctness for n+1 line segments. This means that another line segment is added to our pre-existing set of n line segments.
To prove this, let us assume that the new line segment added is such that it doesn't intersect any other line segment(either both the x coordinates lying on y=0 and y=1 are the least or the most amongst as compared to both the line segments). Let's say that they are the most. On descending sort of y1, the line segment's x coordinates will become the first element of both y1 and y0. As a result, the corresponding x coordinate lying on y=0(in the array y0) will not be considered in any Longest Increasing Subarray(since it is the highest element and is at the 0th index itself). We can prove the same for the case when the x coordinates of the newly added line segment are the least.
Also, we can use a similar approach to prove the same for the (n+1)th line segment if it intersects a few line segments. Since our algorithm is true for k=1,n and n+1, we can safely say that this algorithm holds true for every set of n line segments.

3. **Question 3** Suppose that an equipment manufacturing company manufactures si units in

the i-th week. Each week's production has to be shipped by the end of that week. Every week, one of the three shipping agents A, B and C are involved in shipping that week's production and they charge in the following:

- Company A charges a rupees per unit.
- Company B charges b rupees per week (irrespective of the number of units), but will only ship for a block of 3 consecutive weeks.
- Company C charges c rupees per unit but returns a reward of d rupees per week, but will not ship for a block of more than 2 consecutive weeks. It means that if si unit is shipped in the i-th week through company C, then the cost for i-th week will be csi-d.

The total cost of the schedule is the total cost to be paid to the agents. If si unit is produced in the i-th week, then si unit has to be shipped in the i-th week. Then, give an efficient algorithm that computes a schedule of minimum cost. (Hint: use dynamic programming)

Ans: This question is sort of an optimization problem wherein we have to minimize the total cost of shipping across a span of multiple weeks using a combination of shipping from three companies - A, B and C. First we will see what is already known to us and the input format.

Given:-
1. The number of weeks
2. An array S containing the number of units to be produced each week.
3. It is known that company A charges a rupees/unit.
4. It is known that company B charges b rupees/week irrespective of units to be shipped.
5. It is known tat company C charges c rupees/week with a discount of d rupees/week.

Algorithm:-
To solve this, we can either use the naive approach which will calculate every possible way of payment for shipping or we could use a Dynamic Programming approach which will be much faster.
Therefore, we will go with the DP approach for this problem.

Preprocessing:-For the preprocessing part of this algorithm, we will simply initialize a function cost(i,j) which takes in two parameter: i which is the number of the week and j which is a counter for counting the number of consecutive weeks shipping company C was used(which can not be more than 2).

Subproblems and Recurrence Relation:- This problem will have three types of subproblems (1 each for every company)

- Company A => a*s[i] + cost(i-1,0)
- Company B => 3*b + cost(i-3,0) [this will only hold when i-3 $\geq$ 0]
- Company C => c*s[i] - d + cost(i-1,j+1)

Base case:-The base case will be when the value of i becomes negative, upon which the function will return 0 and ultimately stop.

Initial Subproblem:-
Since we are taking a bottom up approach, the initial subproblem will be cost(n,0).

Pseuodocode/Explanation of DP Algorithm:-
In order to solve the above recurrence relation in an efficient manner, we will use a dynamic programmimg algorithm. We will utilize a 2D-matrix for memoization to reduce the running time our algorithm.
Let the 2D-matrix be called dp with it's dimensions being dp[n][3] (to store weekly costs for n weeks for each of the 3 companies)

The pseudocode is as follows:

cost(i,j):

$i \leftarrow n$
$j \leftarrow 0$
if $i < 0$ **then**
end if
$costA \leftarrow a * s[i] + cost(i-1, 0)$
$costB \leftarrow infinity$
if $i - 3 > 0$ **then**
    $costB \leftarrow 3 * b + cost(i-3, 0)$
end if
$costC \leftarrow infinity$
if $j < 2$ **then**
    $costC \leftarrow c * s[i] + d + cost(i-1, j+1)$
end if

Running time of algorithm:-
The running time of the algorithm is O(3*n) which is essentially O(n) itself when we are using a dynamic programming approach. If we had gone with a simple recursive approach, we would have ended up with an algorithm with a running time of $O(3^n)$.

Acknowledgement:

1. https://www.comp.nus.edu.sg/ sanjay/cs3230/dandc.pdf
2. https://www.geeksforgeeks.org/tiling-problem-using-divide-and-conquer-algorithm/
3. Lecture 7 LIS notes pdf (uploaded on classroom by the professor)