Analysis and Design of Algorithms

Theory Assignment 1

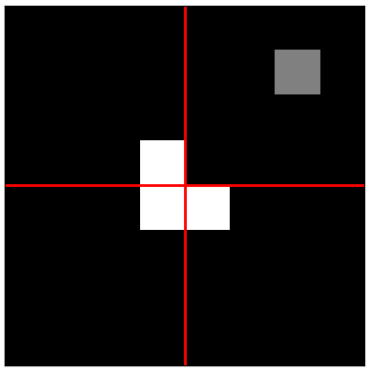
Mehar Khurana (2021541) — Nalish Jain (2021543)

February 24, 2023

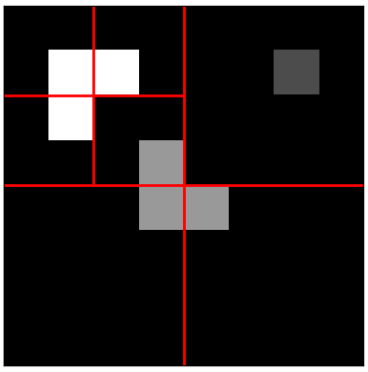
Question 1

1.1 Description

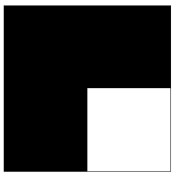
Consider the first step where we split the $n \times n$ board (where $n = 2^p$) into four squares (as shown in Figure 1(a)). We would have a defective square in one of the four sections (top left section in this case). We take each of the 'center' squares (shown in white) of the other sections out of consideration for the moment, since we can fill these with an L-shaped tile at a later stage.



(a) Dividing the $n \times n$ board into four sections



(b) Further iterations



(c) Base case

Figure 1

Now in each of the sections, we have a subproblem of the same kind as we had in the $n \times n$ board given to us, i.e., we need to fill these smaller boards with L-tiles, with one square empty in each of them. At each iteration, we divide a square of side length m into four squares of side length $m/2$ each. $m/2$ is a whole number since $m = 2^k$. As we keep dividing the sections into smaller and smaller squares, we will eventually have just 2×2 boards left, which will be the base case.

We can now start filling in all the 2×2 boards, each of which would have one empty square. As we fill these up, we find spaces for more L-tiles around the corners of these squares, spaces that we had previously left out. As we make our way back up the subproblems, we finally reach the stage in Figure 1(a)[1] again, where we fill in the last L-tile.

1.2 Pseudocode

Pseudocode pushed to next page due to less space.

Input Format: We are given an $n \times n$ array of 0's with the defective square marked as -1.

Explanation: As we go deeper, leaving an L-tile in each iteration, we denote 'leaving' (or marking the L-tile as "defective") by assigning -2 to that index. Then we apply the split function to each of the 4 smaller boards obtained on splitting. Whenever we mark an L-tile, we assign a counter value to each of the 3 squares in that L-tile, with each L-tile in the $n \times n$ board getting a different count value. After we have stepped into and returned from the split functions in the 4 subproblems, we finally fill in the L-tile we "left" (or marked as "defective") initially and mark that with the counter value.

The base case is that of 2×2 boards ($n = 2$) as explained in the problem description (1.1) and Figure 1(c) above.

[1] All figures built in jupyter notebooks using **matplotlib**.

Algorithm 1

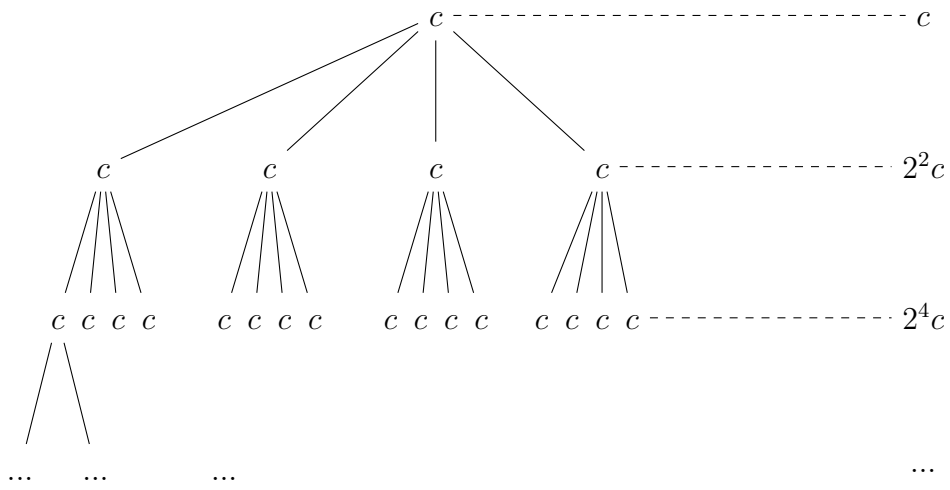
function SPLIT($arr, n, startX, startY, defX, defY$)
  global counter
  if $n == 2$ **then**
    for i **in** $0, 1$ **do**
      for j **in** $0, 1$ **do**
        if $arr[startX + i][startY + j] == 0$ **then**
          $arr[startX + i][startY + j] \leftarrow counter$
    $counter \leftarrow counter + 1$; **return**
  if $defX > startX + n/2 - 1$ **then**
    if $defY > startY + n/2 - 1$ **then**
      $arr[startX + n/2 - 1][startY + n/2 - 1] \leftarrow -2$
      $arr[startX + n/2 - 1][startY + n/2] \leftarrow -2$
      $arr[startX + n/2][startY + n/2 - 1] \leftarrow -2$
      SPLIT$(arr, n/2, startX + n/2, startY + n/2, defX, defY)$
      SPLIT$(arr, n/2, startX, startY, startX + n/2 - 1, startY + n/2 - 1)$
      SPLIT$(arr, n/2, startX + n/2, startY, startX + n/2, startY + n/2 - 1)$
      SPLIT$(arr, n/2, startX, startY + n/2, startX + n/2 - 1, startY + n/2)$
      $arr[startX + n/2 - 1][startY + n/2 - 1] \leftarrow counter$
      $arr[startX + n/2 - 1][startY + n/2] \leftarrow counter$
      $arr[startX + n/2][startY + n/2 - 1] \leftarrow counter$
    else
      $arr[startX + n/2 - 1][startY + n/2 - 1] \leftarrow -2$
      $arr[startX + n/2 - 1][startY + n/2] \leftarrow -2$
      $arr[startX + n/2][startY + n/2] \leftarrow -2$
      SPLIT$(arr, n/2, startX + n/2, startY, defX, defY)$
      SPLIT$(arr, n/2, startX, startY, startX + n/2 - 1, startY + n/2 - 1)$
      SPLIT$(arr, n/2, startX + n/2, startY + n/2, startX + n/2, startY + n/2)$
      SPLIT$(arr, n/2, startX, startY + n/2, startX + n/2 - 1, startY + n/2)$
      $arr[startX + n/2 - 1][startY + n/2 - 1] = counter$
      $arr[startX + n/2 - 1][startY + n/2] = counter$
      $arr[startX + n/2][startY + n/2] = counter$
  else
    if $defY > startY + n/2 - 1$ **then**
      $arr[startX + n/2 - 1][startY + n/2 - 1] = -2$
      $arr[startX + n/2][startY + n/2 - 1] = -2$
      $arr[startX + n/2][startY + n/2] = -2$
      SPLIT$((arr, n/2, startX, startY + n/2, defX, defY))$
      SPLIT$((arr, n/2, startX + n/2, startY, startX + n/2, startY + n/2 - 1))$
      SPLIT$((arr, n/2, startX + n/2, startY + n/2, startX + n/2, startY + n/2))$
      SPLIT$(arr, n/2, startX, startY, startX + n/2 - 1, startY + n/2 - 1))$
      $arr[startX + n/2 - 1][startY + n/2 - 1] \leftarrow counter$
      $arr[startX + n/2][startY + n/2 - 1] \leftarrow counter$
      $arr[startX + n/2][startY + n/2] \leftarrow counter$
    else
      $arr[startX + n/2][startY + n/2 - 1] \leftarrow -2$
      $arr[startX + n/2 - 1][startY + n/2] \leftarrow -2$
      $arr[startX + n/2][startY + n/2] \leftarrow -2$
      SPLIT$((arr, n/2, startX, startY, defX, defY))$
      SPLIT$((arr, n/2, startX + n/2, startY, startX + n/2, startY + n/2 - 1))$
      SPLIT$((arr, n/2, startX + n/2, startY + n/2, startX + n/2, startY + n/2))$
      SPLIT$((arr, n/2, startX, startY + n/2, startX + n/2 - 1, startY + n/2))$
      $arr[startX + n/2][startY + n/2 - 1] \leftarrow counter$
      $arr[startX + n/2 - 1][startY + n/2] \leftarrow counter$
      $arr[startX + n/2][startY + n/2] \leftarrow counter$
  $counter \leftarrow counter + 1$

1.3 Complexity

Recurrence relation:

$$T(n) = 4T(n/2) + c,$$

where c is the constant time to fill in an L-tile.



At any level k, $s = 2^{k-1}c$. Hence, $T(n) = c + 2^2c + 2^4c + ... + 2^{2(h-1)}c = \frac{c(4^k-1)}{4-1}$, where h is the height of the tree. It can be observed that $h = \log_2 n$.

Hence, $T(n) = \frac{(4^{log_2 n}-1)c}{3} = \frac{n^2-1}{3}c = O(n^2)$

1.4 Proof (using the Principle of Mathematical Induction)

Base Case: $n = 2$
We have a 2×2 board, say A. If any one square of A is defective, we can trivially fill in the L-tile (which is formed by the other 3 squares in the 2×2 board)

Inductive Hypothesis:
Any $2^p \times 2^p$ board can be filled in with L-tiles if any one square in the board is defective.

Inductive Step:
To Prove: Given that a $2^p \times 2^p$ board that has a defective square can be filled in completely with L-tiles, we can (completely) fill in L-tiles on a $2^{p+1} \times 2^{p+1}$ board too.
Since

$$2^{p+1} \times 2^{p+1} = 4 \cdot (2^p \times 2^p),$$

A $2^{p+1} \times 2^{p+1}$ board can be split into 4 $(2^p \times 2^p)$ boards, say B_1, B_2, B_3 and B_4. Assume that board B_4 has the defective square, then (without loss of generality) we can mark 3 squares (shown in white in Figure 1(a)) as "defective". Now we know that any $2^p \times 2^p$

board can be filled in with L-tiles if any one square in the board is defective. So, we can fill in each of B_1, B_2, and B_3 using the Inductive Hypothesis, and later fill in the "defective" pieces with an L-tile.

Q.E.D., by the Principle of Mathematical Induction

Question 2

2.1 Pre-processing

Input format: Lines are taken in the form of pairs (x_1, x_2) where x_1 is the x-coordinate of line on $y = 0$ and x_2 is the x-coordinate of line $y = 1$.
Sorting: First, we sort the line according to the x_1 coordinates using MergeSort (algorithm taught in the DSA course) **in descending order**.

2.2 Subproblem Description

In any set of lines that intersect each other (mutually), the x_1 coordinates of the lines must be in increasing order, and the corresponding x_2 coordinates must be in decreasing order, as shown in the diagram below.

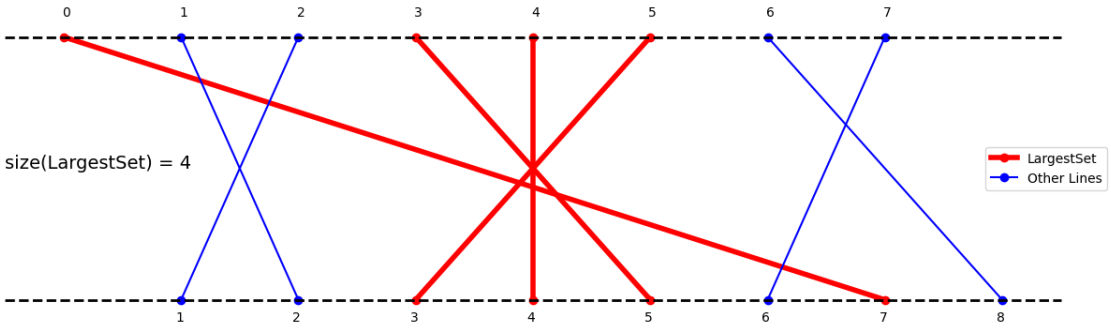


Figure 2: An example that shows the use of Longest Increasing Subsequence

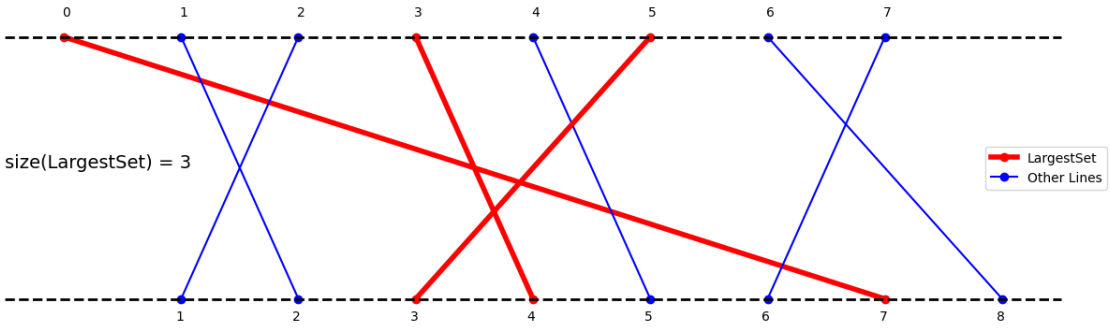


Figure 3: If we flip the x_2 coordinates of any two lines in the LargestSet, we lose one line from the largest set, meaning that the x_2 subsequence must be increasing if x_2s are sorted in descending order.

It can also be seen that the idea stated above holds true for the converse statement, i.e., in any set of lines that intersect each other (mutually), if the x_1 coordinates of the

lines are taken in decreasing order, the corresponding x_2 coordinates must be in ascending order. So, if we sort the x_1 coordinates in descending order, we just need to find the Longest Increasing Subsequence in the array of the corresponding x_2 coordinates.

After our preprocessing stage (Sorting), we have an array formed by corresponding x_2 coordinates. Any two lines i, j will intersect only if x_2 coordinate of line $i < x_2$ coordinate of line j and x_1 coordinate of line $i > x1$ coordinate of line j. Since x_1s are already arranged in descending order we now need to find the longest increasing subsequence (i.e., x_2 coordinates of the largest set of lines that intersect).

Using a bottom-up approach: As explained above, subproblems in our algorithm will be similar to the subproblems in the longest increasing subsequence, that is if LS is the longest subsequence starting from index i then it can be broken into $1 + LS$ starting from index j if $a[i] < a[j]$.

We have used a tabulation approach to solve the problem. The recurrence relation is

$$MaxSet(i) = max(MaxSet(j)) + 1 \; \forall \; j < i \text{ and } a[j] < a[i]$$

and our optimal solution will be

$$\max_{i \in N} MaxSet(i)$$

2.3 Pseudocode

Pseudocode pushed to next page due to less space.

Explanation:
We first iterate through the input set of lines and for each line i, looking at all previous lines $prev$ and check if $prev$ intersects with i and if the length of the largest set of intersecting lines ending in $prev + 1$ is greater than the length of the largest set of intersecting lines ending in i. If so, we update $dp1[i]$ and $hash[i]$.

After the iteration, we output the maximum value in the $dp1$ vector, which denotes the length of the largest set of intersecting lines.

We then create the largest set of intersecting lines by backtracking through the $hash$ vector, starting from the last line with the maximum value in the $dp1$ vector. We add each line to a set $LargestSet$ and continue until we reach the first line in the largest set.

Finally, we return the length of $LargestSet$ and the set itself.

2.4 Complexity

Since we have used two for loops that loop over $i \in n$ and $j \in n$, the time complexity of our algorithm is $O(n^2)$.

Algorithm 2 code to output the largest set of lines that mutually intersect

```
Sort(setOfLines)  //According to the x_1 coordinate of lines in descending order
procedure LargestSet(setOfLines)
    n ← size(setOfLines)
    dp1 ← a vector of integers of size n, initialized with ones
    hash ← a vector of integers of size n
    maxi ← 1
    last ← 0
    for i ← 0 to n − 1 do
        hash[i] ← i
        for prev ← 0 to i − 1 do
            if setOfLines[prev].x_2 < setOfLines[i].x_2 and 1 + dp1[prev] > dp1[i] then
                dp1[i] ← 1 + dp1[prev]
                hash[i] ← prev
        if dp1[i] > maxi then
            maxi ← dp1[i]
            last ← i
    output maxi
    LargestSet ← {}
    LargestSet.Insert(setOfLines[last])
    while hash[last] ≠ last do
        last ← hash[last]
        LargestSet.Insert(setOfLines[last])
    return size(LargestSet), LargestSet
```

Question 3

3.1 Preprocessing

No preprocessing stage is required in our algorithm.
Input Format: The function takes input an array containing the number of units produced in a week. For the first function call the secondLastAgent and LastAgent are set to 'D'.

3.2 Subproblem Description

Using a top-down approach: The subproblem that the algorithm solves is to find the minimum cost of shipping equipment produced by the manufacturing company in a given week, given the last two agents used, the number of units to be shipped in that week, the current week number, and the previously computed solutions for subproblems.

The subproblem can be defined recursively as follows: Let

$$minimumShipmentCost(agent1, agent2, s, w, dp, i)$$

be the minimum cost of shipping s units produced by the manufacturing company in week w, given that the second last agent used was $agent1$ and the last agent used was $agent2$, and the previously computed solutions for subproblems are stored in the dp array with state i.

3.3 Complexity

Recurrence relation:

$$dp[i][0] = min(a \cdot si + dp[i+1][0],\ 3b + dp[i+3][0],\ c \cdot si - d + dp[i+1][1])$$

$$dp[i][1] = min(a \cdot si + dp[i+1][0],\ 3b + dp[i+3][0],\ c \cdot si - d + dp[i+1][2])$$

$$dp[i][2] = min(a \cdot si + dp[i+1][0],\ 3b + dp[i+3][0])$$

$$\textbf{Optimal Solution} = min(dp[i][0],\ dp[i][1],\ dp[i][2])$$

Time Complexity: $O(3n) = O(n)$, since $3n$ is the size of the dp Array.

3.4 Pseudocode

Explanation: The function *minShipmentCost* takes in the second last and last agents used, the number of units to be shipped on a day array, the current week number, a 2D

Algorithm 3

procedure MINSHIPMENTCOST($secondLastAgent, lastAgent, units[], week, dp[], dpIndex$)
  if $week > size(units)$ **then return** 0
  if $dp[week, dpIndex] \neq -1$ **then return** $dp[week, dpIndex]$
  $chosenA,\ chosenB,\ chosenC \leftarrow \infty$
  $chosenA \leftarrow units[week] + a \cdot minShipmentCost(lastAgent, A, units, week + 3, dp, 0)$
  if $week + 3 \leq size(units)$ **then**
    $chosenB \leftarrow 3b + minShipmentCost(B, B, units, week + 1, dp, 0)$
  if $secondLastAgent = C$ and $lastAgent = C$ **then**
    pass
  else
    $chosenC \leftarrow c \cdot units[week] - d + minShipmentCost(lastAgent, C, units, week + 1, dp, dpIndex + 1)$
  $dp[week, dpIndex] \leftarrow min(chosenA, chosenB, chosenC)$
return $dp[week, dpIndex]$

array to store the computed results (dp), and an index ($dpIndex$) for the current state of dp which actually represents the number of days C has worked consecutively.

The function first checks if the current week number exceeds the total number of weeks (size of $units$), in which case it returns 0. If the result for the current state is already computed and stored in the dp array, it is returned.

We then proceed to compute the minimum cost of shipping for the current week by considering all three possible agents.

For agent A, the cost is computed as the product of the number of units produced in the current week and the cost per unit charged by agent A, plus the minimum cost of shipping for the next week, and the $dpIndex$ is set to 0 since C did not work.

For agent B, the cost is computed as a fixed amount charged by agent B for three consecutive weeks, plus the minimum cost of shipping after the next three weeks, and the $dpIndex$ is set to 0 since C did not work.

For agent C, the cost is computed as the product of the number of units produced in the current week and the cost per unit charged by agent C, minus the reward per week offered by agent C, plus the minimum cost of shipping for the next week and the $dpIndex$ is set to $dpIndex + 1$ since C worked for that day.

The last and second last agents for $week\ i$ are set to the agent who worked on $week\ i - 1$ and $week\ i - 2$.

The function then returns the minimum of the costs computed for all three agents. The result is also stored in the dp array for future reference.