Theory Assignment-1: ADA Winter-2023

Shivoy Arora (2021420) Chehak Malhotra (2021141)

1. Consider the problem of putting L-shaped tiles (L-shaped consisting of three squares) in an n $\times$ 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.

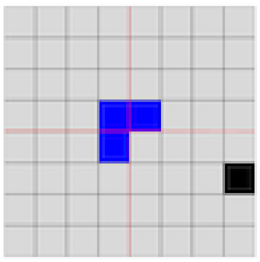
Ans:

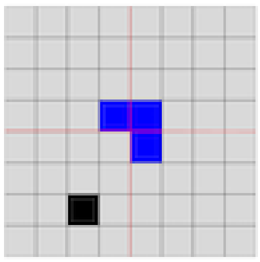
a) In the **preprocessing step**, we will find the coordinates of the defective tile in the given board. We can do this by using a simple nested for loop for each row and column of the board, until we find the defective tile. we will store the coordinated as follows:

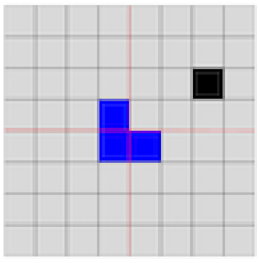
- $xBroken \leftarrow$ coordinate of the defective tile
- $yBroken \leftarrow$ coordinate of the defective tile

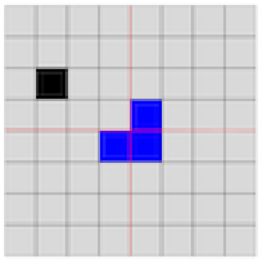
The time complexity of this step is $O(n^2)$.

b) **Subproblems** can be defined as breaking the board into 4 equal parts. We will add one L-shaped tile at the center of the breaking so that one unit part of the L-shaped tile is in three of the sub-boards. We will see the orientation of the L-shaped tile by the following way according to the defective tile:


Orientation 1


Orientation 2


Orientation 3


Orientation 4

Then when the sub-problem is called then either the coordinates of the defective tile is given or the coordinates of the unit of tile of the L-shaped tile that is in that coordinate is given for the coordinates of the defective tile.

c) **Combining** the algorithm is a tail recursion so no extra work is required.

e) **Recurrence relation:** $\mathrm{T}(n) = 4\mathrm{T}(\frac{n}{2}) + C$

Solution: Time Complexity $= O(n^2)$, using Master's Theorem.

Analysis: Since in this algorithm we are visiting each tile once for the preprocessing and second time while filling. And, since the width and height of the board is n $\times$ n and the number of tiles is n^2, the time complexity is $O(n^2)$.

d)

Algorithm 1 Function for filling the L-shaped tile.

```
Require: $n$ is the number of tiles
Ensure: $xCoord, yCoord, xBroken, yBroken \leq n$
  function fillTile($xCoord, yCoord, xBroken, yBroken, n$) then
    if $xBroken >= (xCoord + n)/2$ && $yBroken >= (yCoord + n)/2$ then
      Fill tile according to the orientation 1
    else if $xBroken < (xCoord + n)/2$ && $yBroken >= (yCoord + n)/2$ then
      Fill tile according to the orientation 2
    else if $xBroken >= (xCoord + n)/2$ && $yBroken < (yCoord + n)/2$ then
      Fill tile according to the orientation 3
    else if $xBroken < (xCoord + n)/2$ && $yBroken < (yCoord + n)/2$ then
      Fill tile according to the orientation 4
    end if
  end function
```

Algorithm 2 Putting the L-shaped tiles.

```
Require: $n$ is the number of tiles
Ensure: All the tiles are covered by L-shaped tiles without overlapping
  function putTiles($xCoord, yCoord, xBroken, yBroken, n$) then
    fillTile($xCoord, yCoord, xBroken, yBroken, n$)
    if $n = 2$ then
      return
    end if

    if $xBroken >= (xCoord + n)/2$ && $yBroken >= (yCoord + n)/2$ then
      putTiles($xCoord + n/2, yCoord + n/2, xBroken, yBroken, n/2$)
    else
      putTiles($xCoord + n/2, yCoord + n/2, xCoord + n/2, yCoord + n/2, n/2$)
    end if

    if $xBroken < (xCoord + n)/2$ && $yBroken >= (yCoord + n)/2$ then
      putTiles($xCoord, yCoord + n/2, xBroken, yBroken, n/2$)
    else
      putTiles($xCoord, yCoord + n/2, xCoord, yCoord + n/2, n/2$)
    end if

    if $xBroken >= (xCoord + n)/2$ && $yBroken < (yCoord + n)/2$ then
      putTiles($xCoord + n/2, yCoord, xBroken, yBroken, n/2$)
    else
      putTiles($xCoord + n/2, yCoord, xCoord + n/2, yCoord, n/2$)
    end if

    if $xBroken < (xCoord + n)/2$ && $yBroken < (yCoord + n)/2$ then
      putTiles($xCoord, yCoord, xBroken, yBroken, n/2$)
    else
      putTiles($xCoord, yCoord, xCoord, yCoord, n/2$)
    end if
  end function
```

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:

a) In the **preprocessing step**, we store the lines in a array(L) of 2-tuples where each 2-tuple represent a line in the 2D plane. And in the 2-tuples, the element at the 1^{st} index represents the x-coordinate of the line segment at y=1 line, and the 2^{nd} index represents the x-coordinate of the line segment at y=0 line.

Then after the data is represented in the way told before we sort(using any sorting algorithm like mergesort, quicksort which were taught in the DSA course) the array according to the 1^{st} coordinate of the 2-tuple.

b) The **subproblem** is the largest subset of i segments where each pair of lines intersect each other, i.e., the the dynamic programming array $dp[i]$ represents the largest subset of i to n segments where each pair of lines intersect each other.

c)

$$dp[k] = \max_{i \leftarrow k+1 \ to \ n-1} (dp[i]) + 1, where\ k \in [0, n)$$

d) The largest number in the array dp is the **solution** to the problem.

f) **Runtime**

Time Complexity: $O(n^2)$, since during the iterative step we are using a nested loop to compare all the elements after the i^{th} value, $\forall i \in [0, n)$

Space Complexity: $O(n)$, since only a new array dp with n elements is created e)

Algorithm 3 Largest subset of line segments that intersect each other.

```
Require: $L$ is the set of line segments
Ensure: Each line segment in $L$ is following the format given above
  $dp[n-1] = 1$
  for $i \leftarrow n-2$ to 0 do
    for $j \leftarrow i+1$ to $n-1$ do
      maxSet $\leftarrow 1$
      if $L[j][1] < L[i][1]$ then
        maxSet $\leftarrow$ max(maxSet, $dp[j] + 1$)
      end if
    end for
    $dp[i] \leftarrow$ maxSet
  end for

  # Finding the largest subset from the $dp$ array maxSet $\leftarrow 1$
  for $i \leftarrow 0$ to $n-1$ do
    maxSet $\leftarrow$ max(maxSet, $dp[i]$)
  end for
  return maxSet
```

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:

a) In the **preprocessing step**, we create a 2D matrix dp of size $n \times 3$.

And the weekly production of the company till the n^{th} is stored in an array s of size n.

b) The **subproblem** is the minimum shipping cost till the i^{th} week with the best of a, b and c chosen in the i^{th} week, i.e., the dynamic programming array $dp[i][a]$, $dp[i][b]$ and $dp[i][c]$ represents the minimum shipping cost till the i^{th} week when a, b and c are chosen in the i^{th} week respectively.

c)

$$dp[i][a] = min \begin{cases} dp[i-1][a] \\ dp[i-1][b] \\ dp[i-1][c] \end{cases} + a * s[i]$$

$$dp[i][b] = min \begin{cases} dp[i-3][a] \\ dp[i-3][b] \\ dp[i-3][c] \end{cases} + 3b$$

$$dp[i][c] = min \begin{cases} (c * s[i-1]) - d + min \begin{cases} dp[i-2][a] \\ dp[i-2][b] \end{cases} \\ min \begin{cases} dp[i-1][a] \\ dp[i-1][b] \end{cases} \end{cases} + c * s[i] - d$$

d)

$$min \begin{cases} dp[n-1][a] \\ dp[n-1][b] \\ dp[n-1][c] \end{cases} \quad \textit{is the solution to the problem.}$$

f) **Runtime**

Time Complexity: $O(n)$, since we are using a single loop to iterate through the array of size n, and filling a matrix of size $n \times 3$.

Space Complexity: $O(n)$, since only a new array dp with $n \times 3$ elements is created

e)

Algorithm 4

```
Require: $s_i$ is the number of units in the $i^{th}$ week, a is the charge per unit by Company A,
  b is the charge per week by Company B, c is the charge per unit by Company c, d is the
  reward per week given by Company C
Ensure: Company B will only ship for a block of 3 consecutive weeks and Company C will
  not ship for more than 2 consecutive weeks. The units must be shipped in the week in which
  they have been produced.
  for i in range 0 to n do
    if i==0 then
      $dp[i][a] = as_i$
      $dp[i][b] = \infty$
      $dp[i][c] = cs_i - d$
    else if i==1 then
      $dp[i][a] = min(dp[i-1][a], dp[i-1][b], dp[i-1][c]) + as_i$
      $dp[i][b] = \infty$
      $dp[i][c] = min(dp[i-1][a], dp[i-1][b]) + cs_i - d$
    else if i==2 then
      $dp[i][a] = min(dp[i-1][a], dp[i-1][b], dp[i-1][c]) + as_i$
      $dp[i][b] = 3 * b$
      $dp[i][b] = min((c-s_i) - d + min(dp[i-2][a], dp[i-2][b]), min(dp[i-1][a], dp[i-1][b])) +$
      $cs_i - d$
    else
      $dp[i][a] = min(dp[i-1][a], dp[i-1][b], dp[i-1][c]) + as_i$
      $dp[i][b] = min(dp[i-3][a], dp[i-3][b], dp[i-3][c]) + 3 * b$
      $dp[i][b] = min((c-s_i) - d + min(dp[i-2][a], dp[i-2][b]), min(dp[i-1][a], dp[i-1][b])) +$
      $cs_i - d$
    end if
  end for
  return $min(dp[n-1][a], dp[n-1][b], dp[n-1][c])$
```

Acknowledgement:

- Aditi Saxena (2021371)
- Akanksha Singal (2021008)