Theory Assignment-1: ADA Winter-2023

Ashutosh Gera (2021026) Naman Aggarwal (2021070)

23rd February, 2023

---

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:

PreProcessing:

As a Pre-information/result, we are using the fact/theorem we proved in our *Discrete Mathematics* course which stated "It is **always** possible to use L-shaped tiles to tile a board of size $n \times n$ (where n is a power of 2) with any one tile removed". Based on this result, we can say that our algorithm will always work as long as n is a power of 2.

Basic Idea/ Intuition:

We have a $n \times n$ square board and a given tile T, and we have to tile the complete board except for the tile T using L-shaped tiles; it is given that n is a power of 2.

We think an approach based on the **Divide and Conquer** paradigm in which we will divide our original board ($n \times n$) into **4 boards** (each of size $n/2$). Let's name these sub-boards as b_1, b_2, b_3 and b_4. One of these sub-boards(say, b_1) would be having the Tile T. Thus, we would be able to **recursively** tile the board b_1. But, the other 3 boards need to be completely tiled, which the recursion won't be able to handle directly. To solve that, we'll *ask* recursion to tile the other three boards (b_2, b_3, b_4) considering the tiles $T2, T3, T4$ respectively as the *defective* tiles for each of them (for visual reference, refer here). In that way, while *combining* the *sub-problems*, we will be able to tile the combination of the tiles $T2, T3, T4$ using an L-shaped tile, thus, eventually tiling the complete board except for the tile T and our problem is solved :);
For the **base** case, we will have $n = 2$ in which we can trivially tile the board.

Precise description of the sub-problems:

We have a $n \times n$ board, where n is a power of 2. We will be dividing this board into 4 boards, each of size $n/2 \times n/2$. Hence, our problem is divided into **four sub-problems**, each sub-problem is a **board of size** $(n/2 \times n/2)$, which needs to be completely tiled except one tile(which would be among $T, T2, T3$ and $T4$).

Combining the sub-problems:

As a result of our sub-problems, we will have boards b_1, b_2, b_3, b_4, all completely tiled except the tiles $T, T2, T3, T4$ respectively (notation same as used under *Intuition*). We will **tile the combination of** $T2, T3, T4$ **using an L-shaped tile** and combine the boards b_1, b_2, b_3, b_4, thus, our complete board would be tiled except the tile T.

PseudoCode of the algorithm:

Input: We are given n, the size of the board, which is a power of 2 and tile T which is the defected Tile
Output: *void*, our function(below) would be tiling the complete board except the tile T using L-shaped tiles.

Algorithm 1 Tiling a complete $n \times n$ board except one tile T using L-shaped tiles

```
tiling(n, T):
//base case
if n == 2 then
    tile the board with an L-shaped tile except for the tile T
    return
end if
Divide the board into 4 n/2 × n/2 sub-boards named b1, b2, b3, b4
Get the locations/coordinates of the tiles T2, T3, T4
//Induction Hypothesis:
tiling(n/2, T)      // for b1
tiling(n/2, T2)     // for b2
tiling(n/2, T3)     // for b3
tiling(n/2, T4)     // for b4
// Inductive step (small calculation)
Put an L-shaped tile in the combination of tiles T2, T3, T4
return
//Board has been tiled
```

Ans: (contd.)

Running time analysis of our algorithm:

In our algorithm, "tiling with T" can be interpreted as *printing the location of* T *etc* and thus would take $O(1)$ amount of time. Furthermore, dividing the board into 4 sub-boards also takes $O(1)$ time. Getting the location of tiles also takes $O(1)$ time. Let the sum of all this constant time be denoted by a constant c. Then we have further 4 recursive calls, each of size *half* of the initial problem. Hence, our final **recursive** relation comes out to be:

$$T(n) = 4 \times T(n/2) + c; \qquad T(1) = 0$$

(Since n is always a power of 2, thus $n = 1$ (2^0) would be the base case)

Using **Master's Theorem** on this recurrence relation, we can conclude that the run-time of our algorithm is $\theta(n^2)$.

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:

PreProcessing:

1) There will be two arrays (y0,y1) storing x coordinates on line y=0 and y=1.
2) Now do merge sort on array y0 ensuring that index wise mapping of x coordinates in y0 and y1 remains same.
For ex:
Before Sorting : y0 = [1,5,2,6] ; y1 = [5,7,9,2]
After Sorting : y0 = [1,2,5,6] ; y1 = [5,9,7,2]

Description of the SubProblem:

If i_{th} line segment is chosen and the previous COS is given then what is the maximum number of line segments from ith to nth lines that could be added to COS.
COS $\rightarrow$ Current Optimal Set.

Recurrence relating the subProblem: Our recurrence relation is:

$T(n) = 2 \times T(n-1) + c;$
$T(0) = 0, T(1) = 1$

$2 \times T(n-1) \rightarrow$ at every step we are looking if current line segment can be a part of COS and if it can be then we make a choice to either choose it or not, in both the cases problem set reduces by one as we move onto next line segment there by two recursive calls are made each with T(n-1). Hence, our problem is divided into 2 subproblems, each of size one less than the problem and then those are related by doing some constant time taking operations.

Identifying Subproblem which solves the original problem:

In the *2D array, DP* we will have to traverse for all the values in DP[0] to find the maximum number of intersections, here rows indicate the ith chosen line segment and jth column indicate the minimum X-coordinate value of COS on line y=1. Thereby choosing the 0th line segment will tell us the maximum number of intersections in set from 0 to n line segments, but there will be many sets so we will have to iterate through those sets to get the max number of intersections.

PseudoCode of our algorithm:

Notations:
$y_0 \rightarrow$ array of x coordinates of line segments on y=0.
$y_1 \rightarrow$ array of x coordinates of line segments on y=1.
$DP[numOflineSegments][max(y_1 + 1)] \rightarrow$ is a dp array to store computations where rows indicate the ith choosen line segment and jth column indicate the minimum xcoordinate value of set on line y=1.
minCOS $\rightarrow$ this is minimum value of x coordinate on y=1 for the current choosen optimal set.
currentSegment $\rightarrow$ index of current choosen segment.
$l \rightarrow$ max number of line segments.

Algorithm:

- The approach is inspired from the solution of the classic longest-increasing subsequence DP problem.
- we will choose i_{th} line segment and check if that line segment can be a part of the Current Optimal Set, COS (a set chosen out of 'i' line segments where j line segments intersect with each other, $j \leq i$).
- If i_{th} line segment can't be a part of COS then we will skip this line segment and check for others.
- If i_{th} line segment can be a part of COS, then we will have to make choice to include i_{th} line segment in set or not as it could be the case a particular line segment could be a part of COS, but due to this line segment others can't be a part of COS as those lines don't intersect with current chosen line segment thereby it is important to consider both possibilities of selecting (or not) the current line segment in COS even if it can be a part of COS.
- In order to check if the current chosen line segment i intersects all the other line segments in COS, we know that x-coordinate of all line segments in COS on y=0 will be less than that of the current as we had sorted them earlier so now if x-coordinate of the current chosen line segment on line y=1 is less than the x-coordinate of all line segments in COS on line y=1 then we can say that the current chosen line segment will intersect all of the line segments on COS.
- Instead of traversing the x-coordinate on y=1 for COS we can maintain the minimum value of x-coordinate on y=1 in COS and update it whenever there is a new line segment added to COS. Thereby we will only be dependent on the minimum value of x-coordinate on y=1 in COS to check if the current chosen line segment can be a part of COS.
- **Memoization** when choosing ith line segment we find the max possible number of intersections from i to n line segments provided we have been a COS this computed value will be stored in DP array.

Algorithm 2 Computing the largest subset L in which every pair of segments intersects each other

```
opt(minCOS, currentSegment, l, DP):
if currentSegment ≥ l then
  return 0
end if
if DP[currentSegment][minCOS] ≠ NULL then
  return DP[currentSegment][minCOS]
end if
if y1[currentSegment] < minCOS then
  numOfIntersectionsOnSelecting ← opt(y1[currentSegment], currentSegment+1, l, DP) +
  1
  numOfIntersectionsOnNotSelecting ← opt(minCOS, currentSegment+1, l, DP)
  DP[currentSegment][minCOS] ← max(numOfIntersectionsOnNotSelecting, numOfInter-
  sectionsOnSelecting)
else
  numOfIntersectionsOnNotSelecting ← opt(minCOS, currentSegment+1, l, DP)
  DP[currentSegment][minCOS] ← numOfIntersectionsOnNotSelecting
end if
return DP[currentSegment][minCOS]
```

Ans: (contd.)

Running time analysis of our algorithm:

Our algorithm has a recursive structure, with each recursive call reducing the input size by 1. The recursion ends when the currentSegment value is greater than or equal to l.

- Comparison operations (e.g., currentSegment ≥ l) take constant time, O(1).
- Accessing an element in the DP array using indices takes constant time, O(1).
- The max() function takes constant time, O(1).
- The opt() function is called recursively at most n times.

As each recursive call can lead to two or more recursive calls, resulting in a binary tree with n levels. Hence, the worst case time complexity of our algorithm is O(n^2). We have used **memoization** to avoid redundant computations but the worst case complexity will still remain the same.
Hence, our algorithm is **O(n^2)**.

Justification of correctness of our algorithm:

We can prove the correctness of our algorithm using PMI.

For the base case, n = 0, it returns 0 which is trivially true.
For the Inductive Hypothesis we assume it is true for a set of k line-segments.and output is opt(minCOS, 0, numOfLineSegments, DP).

Inductive step In our algorithm, we are considering both the cases If i_{th} line segment can be a part of COS, then we will have to make choice to include i_{th} line segment in set or not as it could be the case a particular line segment could be a part of COS, but due to this line segment others can't be a part of COS. And we are returning max(1+opt(minCOS, 0, numOfLineSegments, DP), opt(minCOS, 0, numOfLineSegments + 1, DP). Thus, it is also returning the number of elements in the largest subset L which satisfies our required condition. Thus, output for (k+1) is also true.

Thus, By principle of mathematical Induction, Our algorithm is **correct**.

3. Suppose that an equipment manufacturing company manufactures s_i 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 s_i unit is shipped in the i-th week through company C, then the cost for i-th week will be $c \times s_i$ - d.

The total cost of the schedule is the total cost to be paid to the agents. If s_i unit is produced in the i-th week, then s_i 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:

Description of the subproblem:

From the i_{th} week, what is the optimal minimum value for transportation until the last (n_{th}) week.
$L(i, n) \leftarrow$ the minimum value of transportation from i_{th} week to n_{th} week for a particular company.

Recurrence relating the sub-problems:

Broadly, the **recurrence relation** would be like:
$T(n) = 2 \times T(n-1) + T(n-3) + c; \qquad T(0) = 0$

Explaination: $T(n-1) \rightarrow$ if we choose company A for the current week, then we can choose ANY company for the next week, reducing the problem size by 1.
$T(n-1) \rightarrow$ Choosing company C will also reduce the problem size by 1, but we cannot use C for more than 2 consecutive weeks. That condition would be dealt with using constant operations.
$T(n-3) \rightarrow$ IF we choose company B, then for 3 weeks we **must** use it, therefore,

reducing the problem size by 3.
$c \rightarrow$ Constant time operations which would include addition, comparison, if-else conditions.

$L(i,n) = \min[a \times s_i + L(i+1,n), b + L(i+3,n), c \times s_i + L(i+2,n), c \times s_i + L(i+1,n)]$ (given that c has not been used for 2 consecutive weeks, this will be better illustrated in our pseudocode)

Clearly, the sizes of the sub-problems are decreasing. The **Topological order** of our DP algorithm is ***increasing i from*** $0....n$

Identifying Subproblem which solves the original problem:

$L(0,n) \rightarrow$ Our original problem

Explaination: Starting recursive calls from 0_{th} week, we will calculate the minimum cost of transportation from i_{th} week to last (n_{th}) week and which transporter is used for that week. The cost will be returned to $(i-1)_{th}$ week which will add this cost and current week's cost to get the required minimum cost of transportation from i_{th} week to last (n_{th}) week.

Final Answer $\rightarrow$ will be returned by the main function call and in the DP array (Our *memo*), it could be found using $\min(DP[0][0], DP[1][0], DP[2][0])$.

PseudoCode of our algorithm:

Notations:
$w_i \rightarrow$ i-th week number
$c_a \rightarrow$ Cost per unit of using company A for that week
$c_b \rightarrow$ cost for using company B for that week
$c_c \rightarrow$ cost for using compaany C per unit for that week
$d \rightarrow$ discount for that week if using company C.
$n \rightarrow$ total number of weeks.
$DP \rightarrow$ a 2D array of size $3 \times n$ where rows $0, 1, 2$ correspond to companies A, B, C respectively and column number denotes the week number, i.e. $DP[i][j] \rightarrow$ min cost of using i_{th} company in j_{th} week.

Algorithm:

- For each w_i, the user has 3 options to choose vendors for transportation to supply s_i supplies that week:
  - If the user chooses A, he will pay $s_i \times c_a$ + cost(week(i+1))
  - If the user chooses B then he will pay $c_{bi} + c_{b(i+1)} + c_{b(i+2)}$ + cost(week(i+3)). As for minm 3 weeks he is bound to use B.
  - If the user chooses C then he will pay $c_c \times s_i$ + cost(week(i+1)) - d. Here, we will maintain a **flag**
- In every case we check if flag C has value ≥ 2 that means vendor C has been used for 2 consecutive weeks so we can choose only from A or B and **RESET** flag.

- Before finding cost of vendor j for i_{th} week and minimum cost for subsequent weeks we also check in DP array, if in i_{th} week vendor j is used then what is the cost of transportation from i to n_{th} week. If we find it in the array, we use the value; otherwise, we make a recursive call to find and store the value. [**Memoization**]
- We will then check the minimum value the user has to spend upon choosing any of the available vendors. This value is then returned!

Algorithm 3 Computing the schedule with minimum cost

OPT(w_i, c)
if $w_i \geq n$ or $n = 0$ **then**
  return 0
end if
if $n < 3$ **then**
  if DP[0][w_i] $\neq$ NULL **then**
    costA = DP[0][w_i]
  else
    costA = OPT($w_{(i+1)}$, 0) + $s_i \times c_a$
  end if
  if DP[2][w_i] $\neq$ NULL **then**
    costC = DP[2][w_i]
  else
    costC = OPT($w_{(i+1)}$, $c+1$) + $s_i \times c_c - d$
  end if
  DP[0][w_i] = costA
  DP[2][w_i] = costC
  return min($costA, costC$)
end if
if $c < 2$ **then**
  if DP[0][w_i] $\neq$ NULL **then**
    costA = DP[0][w_i]
  else
    costA = OPT($w_{(i+1)}$, 0) + $s_i \times c_a$
  end if
  if DP[1][w_i] $\neq$ NULL **then**
    costB = DP[1][w_i]
  else
    costB = OPT($w_{(i+3)}$, 0) + $c_{bi} + c_{b(i+1)} + c_{b(i+2)}$
  end if
  if DP[2][w_i] $\neq$ NULL **then**
    costC = DP[2][w_i]
  else
    costC = OPT($w_{(i+1)}$, $c+1$) + $s_i \times c_c - d$
  end if
  DP[0][w_i] = costA
  DP[1][w_i] = costB
  DP[2][w_i] = costC
  return min($costA, costB, costC$)
end if

Algorithm 4 contd.

```
if c ≥ 2 then
  if DP[0][w_i] ≠ NULL then
    costA = DP[0][w_i]
  else
    costA = OPT(w_(i+1), 0) + s_i × c_a
  end if
  if DP[1][w_i] ≠ NULL then
    costB = DP[1][w_i]
  else
    costB = OPT(w_(i+3), 0) + c_bi + c_b(i+1) + c_b(i+2)
  end if
  DP[0][w_i] = costA
  DP[1][w_i] = costB
  return min(costA, costB)
end if
```

Ans: (contd.)

Running time analysis of our algorithm:

Our algorithm finds the optimal solution by finding cost for every week i from 1 to n. In i_{th} week we check cost for usage of A,B,C companies. Since we are also using DP and **memoization** to save computation we at max will be solving 3*n subproblems, each subproblem would be related in O(1) time, thereby time complexity comes out to be O(1) × O(n) = **O(n)**.

Acknowledgement: Lecture slides.