ADA Assignment 1

Anirudh S. Kumar
Roll Number - 2021517
IIIT - Delhi
anirudh21517@iiitd.ac.in

Vartika
Roll Number - 2021571
IIIT - Delhi
vartika21571@iiitd.ac.in

February 24, 2023

1 L Shaped Tiles

1.1 Problem

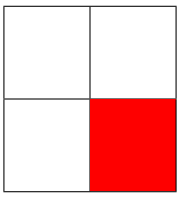
Consider the problem of putting L-shaped tiles (L-shaped consisting of three squares) in an `n` × `n` square-board. You can assume that `n` is a power of 2. Suppose that one square of this board is defective and tiles cannot be put in that square. Also, two L-shaped tiles cannot intersect each other.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

1.2 Proof

1 Pre Processing

Iterate through the board and find which cell is the defective cell. Time complexity for this process is $\mathcal{O}(n^2)$ where $n \times n$ is the size of the board

2 Base Case: `n = 2` i.e. it's a 2×2 board

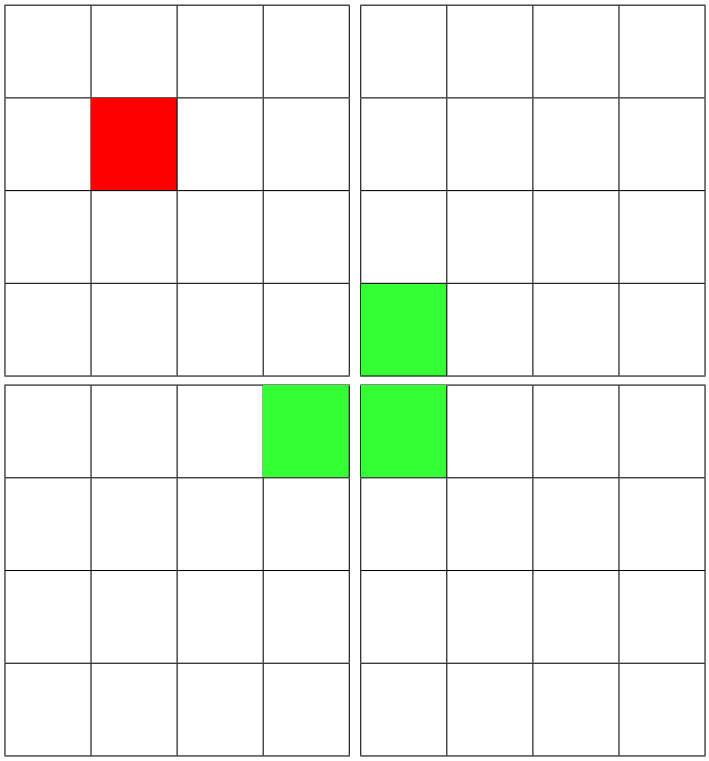


One square is defective. We can place one L shaped tile in the remaining space.

3 Sub Problem

(1) Divide the board into 4 boards of size $n/2 \times n/2$ each
(2) Check which board out of these 4 has the defective tile

(3) Of the 3 sub boards which don't have a defective tile, assume there is an L shaped tile between them like shown in the figure.
(4) Recursively apply the algorithm for all the boards now. We can assume that the positions which already have a piece of tile as defective.



4 How to combine solution

Place an L shaped tile in the location we assumed in (3) in Sub Problem.

5 Recursive Algorithm

Ensure: $board \leftarrow int[n][n]$
Require: defective $\leftarrow$ the defective cell
1: **if** $n = 2$ **then**
2: Place a L shaped tile on the board
3: **else**
4: Find which quadrant the defective cell belongs to
5: Place an L shaped tile at the common corner of the other 3 quadrants
6: Mark these cells as defective
7: Run the algorithm for all the quadrants
8: **end if**

6 Time complexity

We need to find the defective tile for every board of size `n x n`, which will take n^2 comparisons.

In the Divide Step, we break each problem into 4 problems of size $n^2/4$ each. Therefore, we will get the following recurrence relation for $T(n^2)$

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

Let $n^2 = k$, where k is the number of tiles on the board, then our recurrence relation in $T(k)$ now becomes

$$T(k) = 4T(k/4) + c$$

Looking at this equation, we can see that it can easily be solved using Master's Theorem for $a = 4$, $b = 4$ and $d = 0$ Since $d < \log_b(a)$, we will get time tome complexity as :-

$$\mathcal{O}(k^{\log_b a}) = \mathcal{O}(k) = \mathcal{O}(n^2)$$

Therefore, final time complexity of the algorithm comes out to be

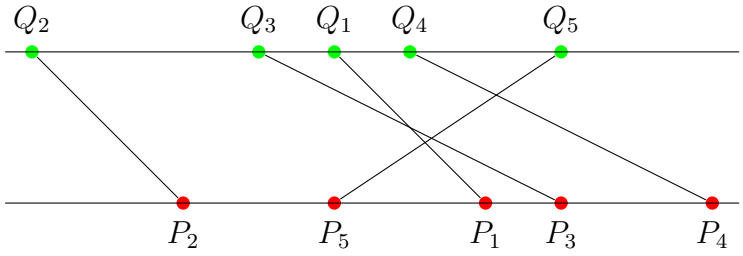
$$\mathcal{O}(n^2 + \mathcal{O}(n^2)) = \mathcal{O}(n^2)$$

□

2 Largest Subset of L

2.1 Problem

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.



2.2 Proof

1 Pre Processing

Let $L = [\{x_1, x_2\}]_n$ be the list of line segments where x_1 is the coordinate of the point at $y = 0$, and x_2 is the coordinate of the point at $y = 1$.

Sort the given list of line segments w.r.t x_1 in decreasing order, and construct a list $a[i] = [x_2 \text{ for } \{x_1, x_2\} \text{ in } L]$

2 Sub Problem

$d[i]$ stores the length of the largest increasing subsequence upto and including $a[i]$.

Initialize $d[i] = 1 \ \forall i \in \{0, 1, 2, \ldots n\}$. Our task is now to find $d[i]$ given we know all $d[j], j < i$

3 Recurrence Relation

If $d[i] = 1$, then it's the only element in the subsequence, but if $d[i] > 1$, then $\exists j < i$, a$[j]$ < a$[i]$ such that $d[j] = d[i] - 1$ i.e. There must be some subsequence of length $d[i] - 1$ which ends at a$[j]$

Converting the above statement into a recurrence relation, we get the following recurrence relation of $d[i]$

$$d[i] = \max\left(1, \max_{\substack{j<i, \\ \text{a}[j] < \text{a}[i]}} (d[j] + 1)\right)$$

4 Sub Problem that solves the original problem

The solution to the problem will be $\max(d[i])$, and then find the actual subsequence corresponding to the index with maximum $d[i]$ value.

To achieve this, we will use an auxilary array p, which will store the j value we have discussed above, i.e., the j for which $d[i]$ is maximum.

Once we have reached the end of the algorithm and found i for which $d[i]$ is maximum, we will backtrack using this auxilary array to find the subsequence, and hence our subset.

5 Algorithm

```
Require: a ← int[n]
Require: L ← int[n][2]                          ▷ Array of line segments in sorted order
d ← int[n]
d[i] = 1 ∀i
p ← int[n]
p[i] = −1 ∀i
for i in range(0, n) do                          ▷ Calculating d and p
    for j in range (0, i) do
        if a[j] < a[i] then
            d[i] = d[j] + 1
            p[i] = j
        end if
    end for
end for
idx ← 0
val ← d[0]
for i in range(0, n) do                          ▷ Finding the maximum value of i
    if d[i] > val then
        val ← d[i]
        idx ← i
    end if
end for
set ← {}                                         ▷ making a set to add all line segments into
while idx ≠ −1 do
    set.add(L[idx])
    idx ← p[idx]
end while
    return set
```

6 Proof of Correctness

For any pair of lines $P_1(x_1, x_2), Q_1(x_3, x_4)$, where x_1, x_3 are the x-coordinates on $y = 0$ and x_2, x_4 are the x-coordinates on $y = 1$, they will only intersect if $(x_1 - x_3)\dot{(}x_2 - x_4) < 0$. In our case, we ensure that $x_1 - x_3 > 0$ by sorting the list of line segments in decreasing order.

Therefore, we only need to check for all pairs (x_2, x_4) such that $x_2 < x_4$. In any increasing sequence $x_1, x_2, \ldots x_k$, all the pair will mutually intersect.

This reduces our problem to finding the Longest Increasing Subsequence in an array where the line segmemnts are sorted in decreasing order by their point on $y = 0$

7 Time Complexity

We can observe that for every i, we must check all $j < i$ to find the maximum $d[j]$. This will take $\mathcal{O}(n)$ number of operations for every i

Since we are performing this opertation for every i, therefore our final time complexiy comes out to be $\mathcal{O}(n^2)$ □

2.3 $\mathcal{O}(n \log(n))$ approach

1 Pre Processing

Let $L = [\{x_1, x_2\}]_n$ be the list of line segments where x_1 is the coordinate of the point at $y = 0$, and x_2 is the coordinate of the point at $y = 1$.

Sort the given list of line segments w.r.t x_1 in decreasing order, and construct a list $a[i] = [x_2 \text{ for } \{x_1, x_2\} \text{ in } L]$

2 Sub-problem

$dp[i]$ stores smallest element for which an increasing subsequence of length i exists. For example, for the array $[6, 4, 5, 7, 1]$, $dp[i] = [-\infty, 1, 5, 7, \infty, \dots]$

Here, ∞ signifies that there isn't a increasing subsequence of length i

3 Base Case

$dp[i] = \infty \ \forall i \in \{1, 2, 3, \dots n\}$ where n is total number of segments, and $dp[0] = -\infty$

4 Sub-problem which solves the original problem

The value of i for which $dp[i]$ is the largest finite number in the array, will give us the length of the longest increasing subsequence. Remember that i gives us the length of the LIS and $dp[i]$ is the value at which the LIS of length i ends.

5 Algorithm

Require: L ← int[n][2] ▷ Array of line segments in sorted order
Require: a ← int[n] ▷ The same a mentioned in pre processing
1: d ← int[n+1]
2: $d[0] = -\infty$
3: $d[i] = \infty \ \forall i > 0$
4: **for** i in range(0, n) **do**
5: l ← smallest value of l such that $d[l] > a[i]$
6: **if** $d[l-1] < a[i]$ $a[i] < d[l]$ **then**
7: $d[l] \leftarrow a[i]$
8: **end if**
9: **end for**
10: ans ← 0
11: **for** i in range(0, n+1) **do**
12: **if** $d[i] = \infty$ **then**
13: **break**
14: **end if**
15: ans ← $d[i]$
16: **end for**

6 Proof of Correctness

For any pair of lines $P_1(x_1, x_2), Q_1(x_3, x_4)$, where x_1, x_3 are the x-coordinates on $y = 0$ and x_2, x_4 are the x-coordinates on $y = 1$, they will only intersect if $(x_1 - x_3)\dot{(}x_2 - x_4) < 0$. In our case, we ensure that $x_1 - x_3 > 0$ by sorting the list of line segments in decreasing order.

Therefore, we only need to check for all pairs (x_2, x_4) such that $x_2 < x_4$. In any increasing sequence $x_1, x_2, \ldots x_k$, all the pair will mutually intersect.

This reduces our problem to finding the Longest Increasing Subsequence in an array where the line segmemnts are sorted in decreasing order by their point on $y = 0$

7 Time Complexity

Time complexity of sorting the array during preprocessing is $\mathcal{O}(n \log(n))$

For finding the value of l as shows in (5) of the algorithm, we notice that d is always a non decreasing subarray. Therefore to find l such that $d[l] > a[i]$, we can use binary search. Time complexity of the binary search comes out to be $\mathcal{O}(\log(n))$

We iterating over the length of a and performing binary search $\forall i$. Therefore, final complexity of the algorithm comes out to be $\mathcal{O}(n \log(n))$ □

3 Shipments

3.1 Problem

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 $cs_i - d$.

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.

3.2 Proof

1 Sub Problem

$dp[i]$ is the optimal cost till ith week without considering company C , and $dp'[i]$ is the optimal cost till the i^{th} week considering C

2 Recurrence Relation

For the 0^{th} week, no products have been shipped, therefore $dp[0] = dp'[0] = 0$.

We can't use B for shipping until we know number of weeks > 2. Therefore, For the first week, cost of shipping by A is

$$dp[1] = as_1$$

Minimum cost of shipping using either A or C is minimum of shipping cost by A and shipping cost by C

$$dp'[1] = \min\begin{pmatrix} dp[1], \\ cs_1 - d \end{pmatrix}$$

For the second week, Minimum cost of shipping by A is minimum cost till last week + cost shipping by A in second week is

$$dp[2] = dp'[1] + as_2$$

Minimum of cost of shipping by either A or C is **min**(cost of shipping by A in week 2, cost of shipping till week 1 + cost of shipping by C in week 2, cost of shipping by C in two consecutive weeks *i.e.* shipping by C in present week and in last week)

$$dp'[2] = \min\begin{pmatrix} dp[2], \\ dp[1] + cs_2 - d, \\ dp[0] + c(s_2 + s_1) - 2d \end{pmatrix}$$

From third week onwards, Cost of shipping by A will change slightly as now we can ship using B also. Therefore, while calculating A we will also take care of cost of shipping by B in 3 consecutive weeks.

$$dp[i] = \min\begin{pmatrix} dp'[i-1] + as_i, \\ dp'[i-3] + 3b \end{pmatrix}$$

Calculation of dp' will remain same

$$dp'[i] = \min\begin{pmatrix} dp[i], \\ dp[i-1] + cs_i - d, \\ dp[i-2] + c(s_i + s_{i-1}) - 2d \end{pmatrix}$$

3 Sub Problem that will solve the original problem

$dp'[n]$ will give us the minimum cost of shipping over n weeks

4 Algorithm

Require: a ← int ▷ Cost of shipping of company A per unit
Require: b ← int ▷ Cost of shipping of company B per week
Require: c ← int ▷ Cost of shipping of company C per unit
Require: d ← int ▷ Reward per week on shipping by company C
Require: s ← int[n] ▷ number of items to be shipped in i^{th} week

```
dp ← int[n]
dp' ← int[n]
dp[0] = dp'[0] = 0.
dp[1] = as_1
dp'[1] = min(cs_1 − d, dp[1])
dp[2] = dp'[1] + as_2
dp'[2] = min(dp[2], dp[1] + cs_2 − d, dp[0] + c(s_2 + s_1) − 2d)
for i in range(3, n) do
    dp[i] = min(dp'[i − 1] + as_i, dp'[i − 3] + 3b)
    dp'[i] = min(dp[i], dp[i − 1] + cs_i − d, dp[i − 2] + c(s_i + s_{i−1}) − 2d)
end for
    return dp'[n]
```

5 Time Complexity

As we are iterating through the array only once, this means that the time complexity of our algorithm is $\mathcal{O}(n)$ where n is the number of weeks

4 Acknowledgements

- Dev Mittal - 2021038 for Q3
- CP-Algorithms for Question 2 $\mathcal{O}(n \log(n))$ solution