Theory Assignment 1

CSE222: *Algorithm Design & Analysis*

Divyajeet Singh (2021529)

February 25, 2023

Problem-1

Consider the problem of putting L-shaped tiles (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.

Input

A square board S of side N ($N = 2^k;\ k \geq 1$).
A defective square $P = (x, y)$ on the board ($1 \leq x, y \leq N$, measured from top-left given in the format of (row, column)).

Output

A proper tiling of S, wherein each tile may be represented using a unique integer.

Solution

The given problem is a naturally recursive problem. To solve the problem, we use the Divide & Conquer paradigm. Using this technique, we divide the board into four equal sub-boards and recursively tile each board until the entire board is tiled.

Like any typical Divide & Conquer algorithm, we can break down the solution into three key steps, namely, **Divide**, **Conquer**, and **Combine**.

The Divide Step

The board is recursively divided into four square sub-boards of side length $\frac{N}{2}$, with a base case of $N = 2$. Figure (1) shows a rough diagram depicting this step.

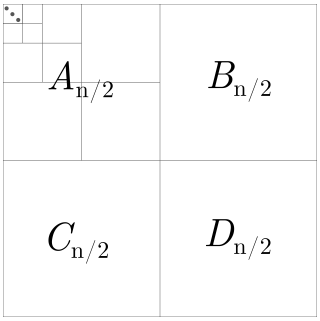

Figure 1: Recursive Division of S_n, a board of side n

As shown above, for any $N = n$, we label the four $2^{n-1} \times 2^{n-1}$ sub-boards as:

1. $A_{n/2}$: The top-left sub-board.
2. $B_{n/2}$: The top-right sub-board.
3. $C_{n/2}$: The bottom-left sub-board.
4. $D_{n/2}$: The bottom-right sub-board.

If $T(S_n, P)$ denotes the problem of tiling S_n of size $n \times n$ with defective square $P = (x, y)$, then, for each of these sub-problems, we decide the defective squares and sub-problems as follows:

1. If P lies in $A_{n/2}$:
   (a) $T\left(A_{n/2}, (x, y)\right)$ is the problem of tiling $A_{n/2}$.
   (b) $T\left(B_{n/2}, \left(\frac{n}{2}, 1\right)\right)$ is the problem of tiling $B_{n/2}$.
   (c) $T\left(C_{n/2}, \left(1, \frac{n}{2}\right)\right)$ is the problem of tiling $C_{n/2}$.
   (d) $T\left(D_{n/2}, (1, 1)\right)$ is the problem of tiling $D_{n/2}$.
2. If P lies in $B_{n/2}$:
   (a) $T\left(A_{n/2}, \left(\frac{n}{2}, \frac{n}{2}\right)\right)$ is the problem of tiling $A_{n/2}$.
   (b) $T\left(B_{n/2}, \left(x, y - \frac{n}{2}\right)\right)$ is the problem of tiling $B_{n/2}$.
   (c) $T\left(C_{n/2}, \left(1, \frac{n}{2}\right)\right)$ is the problem of tiling $C_{n/2}$.
   (d) $T\left(D_{n/2}, (1, 1)\right)$ is the problem of tiling $D_{n/2}$.
3. If P lies in $C_{n/2}$:
   (a) $T\left(A_{n/2}, \left(\frac{n}{2}, \frac{n}{2}\right)\right)$ is the problem of tiling $A_{n/2}$.
   (b) $T\left(B_{n/2}, \left(\frac{n}{2}, 1\right)\right)$ is the problem of tiling $B_{n/2}$.
   (c) $T\left(C_{n/2}, \left(x - \frac{n}{2}, y\right)\right)$ is the problem of tiling $C_{n/2}$.
   (d) $T\left(D_{n/2}, (1, 1)\right)$ is the problem of tiling $D_{n/2}$.
4. If P lies in $D_{n/2}$:
   (a) $T\left(A_{n/2}, \left(\frac{n}{2}, \frac{n}{2}\right)\right)$ is the problem of tiling $A_{n/2}$.
   (b) $T\left(B_{n/2}, \left(\frac{n}{2}, 1\right)\right)$ is the problem of tiling $B_{n/2}$.
   (c) $T\left(C_{n/2}, \left(1, \frac{n}{2}\right)\right)$ is the problem of tiling $C_{n/2}$.
   (d) $T\left(D_{n/2}, \left(x - \frac{n}{2}, y - \frac{n}{2}\right)\right)$ is the problem of tiling $D_{n/2}$.

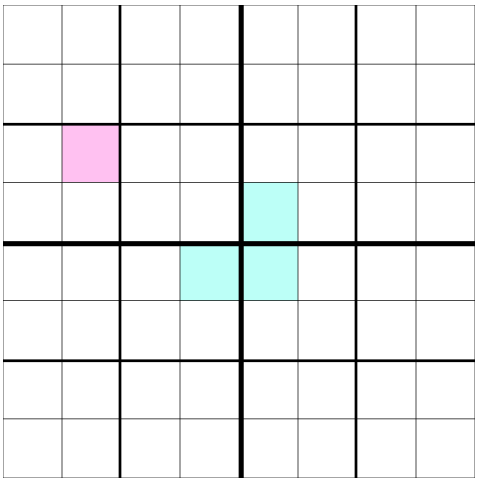

Figure 2: Division into sub-problems when $P = (3, 2)$ lies in A_4 of S_8

Note: In the case of sub-problems except when P lies in $A_{n/2}$, the coordinates of the defective square are modified such that it lies in that particular sub-board.

Figure (2) shows the division of an example board S_8 into sub-problems when the defective tile is represented by $P = (3, 2)$, lies in A_4. The pink tile is *defective*, and the blue tiles are *defect*s introduced to solve the sub-problems. The other cases follow by rotational symmetry.

Note: The above procedure is applied on all four sub-problems, namely $A_{n/2}$, $B_{n/2}$, $C_{n/2}$, and $D_{n/2}$, recursively until the base case is reached.

The Conquer Step

The conquer step is the most important step in the Divide & Conquer paradigm. In this step, the base case of the problem, i.e., $N = 2$ is solved. In the base case $T(S_2, (x_0, y_0))$ where $P_0 = (x_0, y_0)$ is defective, only one of four possibilities can occur:

1. $P_0 = A_1$, i.e. the top-left square is defective.
2. $P_0 = B_1$, i.e. the top-right square is defective.
3. $P_0 = C_1$, i.e. the bottom-left square is defective.
4. $P_0 = D_1$, i.e. the bottom-right square is defective.

In each of these cases, we can tile the board by placing an L-shaped tile such that it does not touch the defective square. This is a constant time operation, since it only requires us to place a tile in a (given) particular position.

The Combine Step

For the final step, the solutions of the sub-problems are combined to obtain the solution of the original problem. For any $N = n$, when all four of its sub-problems are solved, the only remaining *defective* (untiled) squares are three of the four center-most squares of the board. The L-shaped gap so formed can be filled by placing an L-shaped tile. The combine step is also done within constant time, since it only requires us to place a tile in a (given) particular position.

Figure (3) shows a rough illustration of this step.

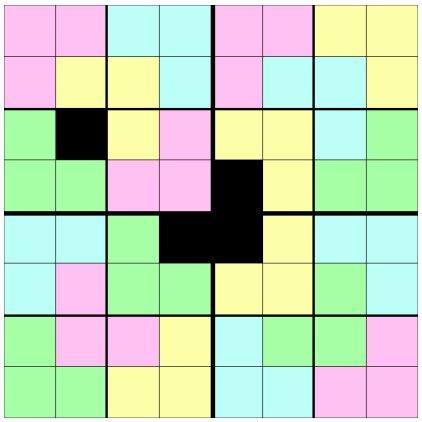

Figure 3: Filling the L-shaped gap formed in the center by placing a tile

The complete Recursive Algorithm

The complete recursive algorithm consists of the following steps:

1. Recursively divide the board of size $N = n$ into four sub-boards until the base case of $N = 2$ is reached.

2. Solve the sub-problems recursively as described above, by placing tiles without touching any defective square.

3. Combine the solutions of the sub-problems by placing an L-shaped tile in the gap formed by the three defective squares in the middle.

Pseudocode for the Place-tile subroutine

A subroutine, PLACE-TILE(S, X), is used to place an L-shaped tile (numbered X) in the given board S_2 without touching any defective square.

Figure (4) shows a diagram of how a tile is placed. Other base case solutions follow by rotational symmetry. Note that in any case, however, the tile is never placed on the defective square. A tile is *placed* by assigning a unique number to the squares it covers.

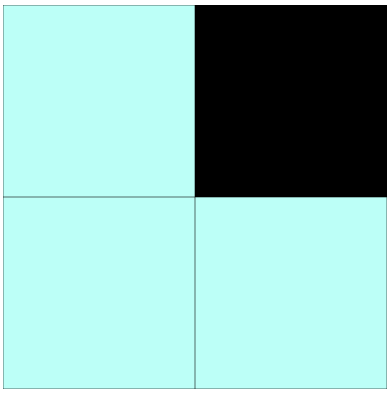

Figure 4: Tiling a 2×2 board without touching any defective square (base case S_2)

The subroutine also updates the unique label X which it has already used to place a tile. Algorithm (1) shows a pseudocode for the subroutine PLACE-TILE(S, X).

Algorithm 1 An algorithm to place one L-shaped tile in given 2×2 board without touching any defective square

```
1:  procedure PLACE-TILE(S[1 : 2][1 : 2], X):
2:      if S[1, 1] is defective then
3:          S[1, 2] = S[2, 1] = S[2, 2] ← X
4:      else if S[1, 2] is defective then
5:          S[1, 1] = S[2, 1] = S[2, 2] ← X
6:      else if S[2, 1] is defective then
7:          S[1, 1] = S[1, 2] = S[2, 2] ← X
8:      else
9:          S[1, 1] = S[1, 2] = S[2, 1] ← X
10:     end if
11:     X ← X + 1
12:     return S
13: end procedure
```

Pseudocode for the complete recursive algorithm

The pseudocode for the complete recursive algorithm is given in Algorithm (2).

Algorithm 2 An algorithm to tile a board of size $N = 2^k$ with one defective square with L-shaped tiles

```
1: procedure TILE(S[1 : N][1 : N], P(x, y)):
2:     if N = 2 then
3:         return PLACE-TILE(S, X)
4:     else
5:         n_0 ← N/2
6:         if x ≤ n_0 and y ≤ n_0 then
7:             P_1 ← (x, y)
8:             P_2 ← (n_0, 1)
9:             P_3 ← (1, n_0)
10:            P_4 ← (1, 1)
11:        else if x ≤ n_0 and y > n_0 then
12:            P_1 ← (n_0, n_0)
13:            P_2 ← (x, y − n_0)
14:            P_3 ← (1, n_0)
15:            P_4 ← (1, 1)
16:        else if x > n_0 and y ≤ n_0 then
17:            P_1 ← (n_0, n_0)
18:            P_2 ← (n_0, 1)
19:            P_3 ← (x − n_0, y)
20:            P_4 ← (1, 1)
21:        else
22:            P_1 ← (n_0, n_0)
23:            P_2 ← (n_0, 1)
24:            P_3 ← (1, n_0)
25:            P_4 ← (x − n_0, y − n_0)
26:        end if
27:        TILE(S[1 : n_0][1 : n_0], P_1)
28:        TILE(S[1 : n_0][n_0 + 1 : N], P_2)
29:        TILE(S[n_0 + 1 : N][1 : n_0], P_3)
30:        TILE(S[n_0 + 1 : N][n_0 + 1 : N], P_4)
31:        PLACE-TILE(S[n_0 : n_0 + 1][n_0 : n_0 + 1], X)
32:        return S
33:    end if
34: end procedure
```

Runtime Analysis of the Algorithm

It is easy to see that the recurrence relation for the above algorithm is given by:

$$T(N) = 4T\left(\frac{N}{2}\right) + C \tag{1}$$

with the base case of $T(2) = 1$.
We simplify the recurrence relation and solve it mathematically as follows:

$$\begin{aligned}
T(N) &= 4T\left(\frac{N}{2}\right) + C \\
&= 4\left(4T\left(\frac{N}{4}\right) + C\right) + C \\
&= 4\left(4\left(4T\left(\frac{N}{8}\right) + C\right) + C\right) + C \\
&= \cdots \\
&= 4^i\ T\left(\frac{N}{2^i}\right) + C\sum_{j=0}^{i-1} 4^j \qquad (2) \\
&= 4^i\ T\left(\frac{N}{2^i}\right) + C\left(\frac{4^i - 1}{3}\right) \\
&= 4^i\ T\left(\frac{N}{2^i}\right) + C_0(4^i - 1) \qquad \left(\text{assuming } C_0 = \frac{C}{3}\right) \qquad (3)
\end{aligned}$$

where i is the number of steps needed to reach the base case. The base case is reached when:

$$\begin{aligned}
\frac{N}{2^i} = 2 &\implies N = 2^{i+1} \\
&\implies N \geq 2^i \implies i \leq \log_2 N \qquad (4)
\end{aligned}$$

Therefore, the solution to the recurrence relation becomes (at $i = \log_2 N$):

$$\begin{aligned}
T(N) &\leq 4^i\ T\left(\frac{N}{2^i}\right) + C_0\ 4^i \\
&= 4^{\log_2 N}\ T\left(2\right) + C_0\ 4^{\log_2 N} \\
&= 2^{2\log_2 N}(1) + C_0\ 2^{2\log_2 N} \\
&= 2^{\log_2 N^2} + C_0\ 2^{\log_2 N^2} \\
&= C_1 N^2 \qquad (\text{assuming } C_1 = 1 + C_0) \qquad (5)
\end{aligned}$$

Since $\exists\ c \in \mathbb{R}$ such that $T(N) \leq cN^2\ \ \forall N \geq 2$, we conclude that the algorithm runs in $O(N^2)$ time, where $N = 2^k$ is the side of the input board.

Problem-2

Suppose we are given a set L of n line segments in 2D plane. Each line segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$. 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.

Input

A list L of n line segments.
Each line segment is of the form (x_0, x_1), representing its x-coordinates on the lines $y = 0$ and $y = 1$ respectively.

Output

The length of a largest subset of L of which every pair of segments intersects each other.

Solution

The given problem can be solved with the Longest Decreasing Subsequence algorithm which is implemented using Dynamic Programming paradigm. The solution makes use of an optimized version of the LDS algorithm which uses binary search and runs in $O(n \log n)$ time for a list of n numbers.

Terminologies and Definitions

Definition 1 (Inversion) *Let $P[1 : p]$ be a list of p numbers. An inversion of P is a pair of indices (i, j) such that $i < j$ and $P[i] > P[j]$.*

Definition 2 (Longest Decreasing Subsequence (LDS)) *Let $P[1 : p]$ be a list of p numbers. The LDS of P is the longest subset $Q[1 : q]$ of P such that the Q is a monotonically non-increasing array, i.e. $Q[i] \geq Q[i + 1]$ $\forall$ $1 \leq i \leq q$.*

Definition 3 (Largest Intersecting Set) *Let $L[1 : n]$ be a list of n line segments. The largest intersecting set of L is the largest subset $Q[1 : q]$ of L such that $Q[i]$ intersects $Q[j]$ $\forall$ $1 \leq i, j \leq q$.*

Preprocessing

We first sort the input list L by the x_0-coordinate of each line segment. This can be done using any common sorting algorithm, such as MERGE-SORT($L[1 : n]$) routine covered in class.
The sorting step can be achieved in $\Theta(n \log n)$ time.

Reason for Sorting:
When L is sorted in the aforementioned fashion, then two line segments intersect each other if and only if their x_1-coordinates form an inversion. This means that more than two line segments mutually intersect each other if and only if their x_1-coordinates are in monotonically non-increasing order. Hence, the longest decreasing subsequence of the x_1-coordinates of sorted L forms the *largest intersecting set* of L.

Figure 5 illustrates the said idea.

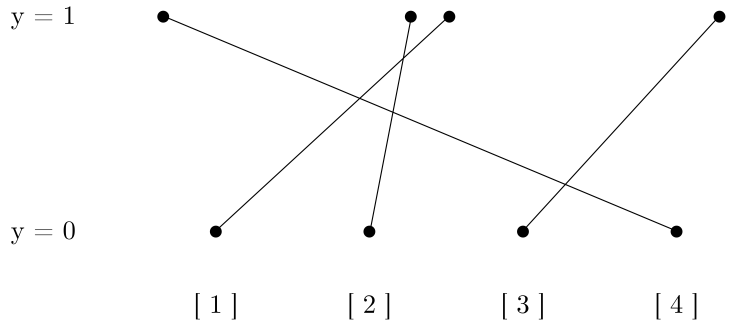

Figure 5: An example of $L[1:4]$ sorted by x_0-coordinates

Note: No tie-breaker is needed for sorting since all the points are unique.

Decomposition into Sub-Problems

The given problem of $L[1:n]$ is broken down into overlapping sub-problems of the form $L[1:i]\ \forall\ 1 \leq i \leq n$. For each problem, say $S[i]$, we intend to find the length of a *largest intersecting set* of $L[1:i]$.
Such a subset can be found by finding the largest subset of $L[1:i-1]$ that intersects $L[i]$. Hence, this decomposition of the given problem into smaller sub-problems guarantees substructure optimality. This is because at any step, we can find the optimal solution using the optimal solutions obtained in the previous steps.

Recurrence Relation

Let $S[i]$ be the length of a *largest intersecting set* of $L[1:i]$. Then, the following recurrence relation can be defined:

$$S[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 + \max_{1 \leq j < i}(S[j]) & \text{if } \exists\ 1 \leq j < i \text{ such that } L[j] \text{ intersects } L[i] \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

Using the above recurrence relation, we can find the length of a *largest intersecting subset* of $L[1:i]\ \forall\ 1 \leq i \leq n$ and maintain it in an $n \times 1$ dynamic programming table.

Base Case and Solvable Sub-Problems

The base case of the above recurrence relation is $S[1] = 1$. This is because the length of the *largest intersecting subset* of $L[1:1]$ is 1, as $L[1]$ must intersect itself.

The sub-problem that solves the original problem is $S[n]$, the solution to $L[1:n]$.

The complete Algorithm

The complete algorithm consists of the following steps:

1. Sort the input list L by the x_0-coordinate of each line segment.
2. In the sorted array, find the LDS of the x_1-coordinates of the line segments.
3. The length of the LDS is the length of a *largest intersecting subset* of L.

Pseudocode for the Lower-bound subroutine

To complete the LDS algorithm, we use a subroutine LOWER-BOUND$(A[1:n], x)$, a binary-search like algorithm that returns the index of the first element in the range $[1, n)$ that is not less than x.

The pseudocode for the subroutine is given in Algorithm (3).

Algorithm 3 An algorithm to find the lower-bound of a value in a sorted array

```
1:  procedure LOWER-BOUND(A[1 : n], x):
2:      low ← 1
3:      high ← n
4:      while low < high do
5:          mid ← ⌊(low+high)/2⌋
6:          if A[mid] < x then
7:              low ← mid + 1
8:          else
9:              high ← mid
10:         end if
11:     end while
12:     return low
13: end procedure
```

Pseudocode for the comlete algorithm

The pseudocode for the complete algorithm is given in Algorithm (4).

Algorithm 4 An algorithm to find the length of a *largest intersecting subset* of L

```
1: procedure LARGEST-INTERSECTING-SUBSET(L[1 : n]):
2:     MERGE-SORT(L[1 : n])
3:     X[1 : n] ← 0
4:     for i ← 1 to n do
5:         X[i] ← L[i].x_1
6:     end for
7:     len ← 1
8:     dp[1 : n] ← 0
9:     dp[1] ← X[1]
10:    for i ← 2 to n do
11:        dp[i] ← 1
12:        if X[i] < dp[len] then
13:            len ← len + 1
14:            dp[len] ← X[i]
15:        else
16:            j ← LOWER-BOUND(dp[len : 1], X[i])
17:            dp[j] ← X[i]
18:        end if
19:    end for
20:    return len
21: end procedure
```

Runtime Analysis of the Algorithm

The following operations are performed in the algorithm:

1. MERGE-SORT($L[1:n]$) This step takes $\Theta(n \log n)$ time, where n is the number of line segments in L.

2. Copy the x_1-coordinates of the line segments into a new array $X[1:n]$. This step takes $\Theta(n)$ time, as we linearly loop over L once.

3. Calculation of LDS of $X[1:n]$, which takes $\Theta(n \log n)$ time using by binary-search inside a linear loop.

Dominated by the sorting step and the LDS step, the time complexity of the algorithm is $\Theta(n \log n)$.

Proof of Correctness

The correctness of the algorithm can be proved by the principle of mathematical induction.

Claim 1 *The algorithm is correct and gives the optimal solution. It gives the length of the largest (possible) intersecting subset of* $L[1:n]$.

Proof.

Base Case

The base case of the induction is $n = 1$, where we consider the set $L[1:1]$ consisting of 1 line segment. The length of the *largest intersecting subset* of $L[1:1]$ is 1. This is because $L[1]$ must intersect itself.

Induction Hypothesis

Assume that the claim holds true $\forall j \leq k$ for some $k \leq n$, i.e. the algorithm produces the correct result for every subset $L[1:j]$ $(j \leq k)$. The claim that the algorithm works correctly for $L[1:k+1]$ must be proved.

Inductive Step

Let $L[k+1]$ be the current line segment in consideration, that could be added to $L[1:k]$. Then ony three (mutually exclusive) cases are possible:

1. **Case 1** *$L[k+1]$ does not intersect any line segment in $L[1:k]$.*
   The length of the *largest intersecting subset* of $L[1:k+1]$ is 1. This is because $L[k+1]$ must intersect (at least) itself.
2. **Case 2** *$L[k+1]$ intersects some (not all) line segment(s) in $L[1:k]$.*

Let $m = \max_{1 \leq j < k}(S[j])$. Then, m is the maximum number of mutually intersecting line segments that also intersect with $L[k+1]$. The length of the *largest intersecting subset* of $L[1 : k+1]$ is $1 + m$. [1]

3. **Case 3** *$L[k+1]$ intersects all the line segments in $L[1 : k]$.*
   The length of the *largest intersecting subset* of $L[1 : k+1]$ is $k+1$. This is because $L[k+1]$ intersects all $k+1$ line segments in $L[1 : k+1]$ including itself.

According to the assumption, the size of the *largest intersecting subset*s of $L[1 : j]$ are given by $S[j]$ $\forall j \leq k$. The size of the *largest intersecting subset* of $L[1 : k+1]$ can be found using the above. In each case, the subset so produced is indeed the largest, as it is the only subset that satisfies the constraints.

Hence, the claim is proved. □

[1] The algorithm finds m using the Longest Decreasing Subsequence algorithm. Refer to *Reason for Sorting* under **Preprocessing**.

Problem-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 three shipping agents A, B, and C are involved in shipping the week's production. Their charges are given as follows:

- 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, and 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 units are produced in the i^{th} week, then they must be shipped in the i^{th} week. Give an efficient algorithm that computes a schedule of minimum cost.

Input

A list s of the number of units manufactured in n weeks.
a, the fare charged by company A per unit.
b, the fare charged by company B per week.
c, the fare charged by company C per unit.
d, the total discount given by company C per week.

Output

The minimum cost that can be obtained using an optimal schedule.

Solution

The given problem can be efficiently solved using Dynamic Programming paradigm. Dynamic programming allows for the calculation of the cost of an optimal schedule that ends at the i^{th} week ($\forall\ 1 \leq i \leq n$).

Notations Used

Notation 1 (A_i) *Let $A_i = as_i$ denote the cost of shipping the production of the i^{th} week using company A.*

Notation 2 (B_i) *Let $B_i = 3b$ denote the cost of shipping the production of weeks $i-2$ to i using company B. Note that B_i is a constant (independent of the current week, i).*

Notation 3 (C_i) *Let $C_i = cs_i - d$ denote the cost of shipping the production of the i^{th} week using company C.*

Notation 4 ($M[i, X]$) *Let $M[i, X]$ denote the minimum cost of shipping till the end of the i^{th} week using company X for the i^{th} week*

Notation 5 ($M[i]$) *Let $M[i] = [M[i, A], M[i, B], M[i, C]]$ represent the i^{th} row of the dynamic programming table, M.*

Notation 6 ($Paths[i]$) *Let $Paths[i]$ denote the set of schedules that use C for the i^{th} week. It only considers the last two weeks.*

$$Paths[i] = (M[i-1, A], M[i-1, B], M[i-2, A] + C_{i-1}, M[i-2, B] + C_{i-1})$$

Decomposition into Sub-Problems

The given problem of $s[1:n]$ is broken down into overlapping sub-problems of the form $s[1:i]\ \forall\ 1 \leq i \leq n$. For each problem, say $M[i]$, we intend to find the minimum cost of shipping till the end of the i^{th} week. Such optimal cost can be found by considering the optimal cost of shipping till the end of the $(i-1)^{th}$ week.
This decomposition of the given problem into smaller sub-problems guarantees sub-structure optimality. This is because at any step, we can find the optimal solution using the optimal solutions obtained in the previous steps.

Recurrence Relation

To solve this problem, we maintain an $n \times 3$ ($n \geq 3$) dynamic programming table, M, for tabulation. M contains an extra row of zeros at the beginning.
Then, the following recurrence relations can be defined:

$$M[i, A] = \begin{cases} 0 & \text{if } i = 0 \\ A_1 & \text{if } i = 1 \\ \min M[i-1] + A_i & \text{otherwise} \end{cases} \tag{7}$$

$$M[i, B] = \begin{cases} 0 & \text{if } i = 0 \\ \infty & \text{if } 0 < i < 3 \\ \min M[i-3] + B_i & \text{otherwise} \end{cases} \tag{8}$$

$$M[i, C] = \begin{cases} 0 & \text{if } i = 0 \\ C_1 & \text{if } i = 1 \\ \min Paths[i] + C_i & \text{otherwise} \end{cases} \tag{9}$$

The recurrence relation for $M[i, B]$ ensures that company B is selected in blocks of 3 consecutive weeks only.
The recurrence relation for $M[i, C]$ ensures that company C can be chosen for no more than 2 consecutive weeks.

Base Case and Solvable Sub-Problems

The base case of the above given recurrence relations for the three companies are:

$$M[i, A] = \begin{cases} 0 & \text{if } i = 0 \\ A_1 & \text{if } i = 1 \end{cases} \tag{10}$$

$$M[i, B] = \begin{cases} 0 & \text{if } i = 0 \\ \infty & \text{if } 0 < i < 3 \end{cases} \tag{11}$$

$$M[i, C] = \begin{cases} 0 & \text{if } i = 0 \\ C_1 & \text{if } i = 1 \end{cases} \tag{12}$$

The following table represents the base case for the dynamic programming array, M:

Weeks	Company A	Company B	Company C
0	0	0	0
1	A_1	∞	C_1
2	$\min M[1] + A_2$	∞	$\min M[1] + C_2$
3	$\min M[2] + A_3$	$\min M[0] + B_3$	$\min Paths[3] + C_3$

Even if $n < 3$, we still create $M[1:4][1:3]$ and fill it with the base case values.
The sub-problem that solves the original problem is given by $\min M[n]$, the minimum cost of shipping till the end of the n^{th} week.

Note: Due to the given constraints, B cannot be selected in weeks 1 and 2 when $n < 3$. Hence, the base case for $M[i, B] = \infty$ for $1 \leq i < 3$.

The complete Algorithm and Pseudocode

The complete algorithm consists of the following steps:

1. Initialize the dynmaic programming table following the base case.
2. If $n > 3$, iterate over i from 4 to n and tabulate M using the above given recurrence relations.
3. Return $\min M[n]$. This is the minimum cost of shipping till the end of the n^{th} week.

The pseudocode for the complete algorithm is given in Algorithm (5). [2]

Algorithm 5 An algorithm to find the minimum cost of shipping till the end of the n^{th} week.

procedure FIND-MIN-COST($s[1:n]$, a, b, c, d):
    $N \leftarrow \max(4, n+1)$
    $M[1:N][1:3] \leftarrow 0$
    $M[1] \leftarrow [A_1, \infty, C_1]$
    $M[2] \leftarrow [\min M[1] + A_2, \infty, \min M[1] + C_2]$
    $M[3] \leftarrow [\min M[2] + A_3, \min M[0] + B_3, \min Paths[3] + C_3]$
    if $n > 3$ **then**
        for $i \leftarrow 4$ to n **do**
            $M[i, A] \leftarrow \min M[i-1] + A_i$
            $M[i, B] \leftarrow \min M[i-3] + B_i$
            $M[i, C] \leftarrow \min Paths[i] + C_i$
        end for
    end if
    return $\min M[n]$
end procedure

Runtime Analysis of the Algorithm

The algorithm initializes the dynamic programming table in constant time. The most computationally heavy part of the algorithm is the linear loop applied on i from 4 to n. This is a linear time operation. The three atomic operations used inside the loop simply update the dynamic programming table, and hence take constant time.

Hence, the time complexity of the algorithm is $\Theta(n)$, where n is the number of weeks.

[2]The pseudocode of the algorithm makes use of notations introduced in the section **Notations Used** to make it concise. While implementing the algorithm, notations like A_i or $Paths[i]$ can be coded as macros or smaller $\Theta(1)$ subroutines.

Bibliography

1. **GeeksforGeeks:** Tiling Problem using Divide & Conquer algorithm
2. **GeeksforGeeks:** Longest Decreasing Subsequence