Theory Assignment-1: ADA Winter-2023

Deepanshu Dabas (2021249)   Ankush Gupta (2021232)

---

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: We will use divide and conquer paradigm.

Subproblems

The subproblem for the tiling problem involves filling a square board of size $2^k X 2^k$ (where 2^k = n) with L shaped tiles where one cell is missing. We divide the board into four quadrants of size $2^{k-1} X 2^{k-1}$ and recursively fill each quadrant until we reach a board of size $2X2$ as, in this case, we need to place a single L shaped tile to cover the missing cell.

Combine

To combine the subproblem solutions, we place four L-shaped tiles in each quadrant except the one with the missing cell. We use the missing cell's location to determine which quadrant it is in, and then we fill the quadrant containing the missing cell recursively.

Pseudo-code

```
Function fill_board(board, start_x, start_y, size, missing_x, missing_y):
    if size = 2 then
        for i ← start_x to start_x + size − 1 do
            for j ← start_y to start_y + size − 1 do
                if (i, j) ≠ (missing_x, missing_y) then
                    board[i][j] ← 1;
                end
            end
        end
        return
    end
    tile_number ← 0;
    center ← size / 2;
    missing_quadrant ← 0;
    if missing_x ≥ start_x + center then
        missing_quadrant += 2;
        start_x += center;
    end
    if missing_y ≥ start_y + center then
        missing_quadrant += 1;
        start_y += center;
    end
    for i from 0 to 3 do
        if i ≠ missing_quadrant then
            tile_x ← start_x + (i % 2) * center;
            tile_y ← start_y + (i / 2) * center;
            board[tile_x][tile_y] ← tile_number;
            board[tile_x + center][tile_y + center] ← tile_number;
            board[tile_x + (i + 1) % 2 * center][tile_y + i / 2 * center] ←
              tile_number;
            tile_number += 1;
        end
    end
    fill_board(board, start_x, start_y, center, start_x + center - 1, start_y +
      center - 1);
    fill_board(board, start_x, start_y + center, center, start_x + center - 1,
      start_y + center);
    fill_board(board, start_x + center, start_y, center, start_x + center,
      start_y + center - 1);
    fill_board(board, start_x + center, start_y + center, center, missing_x,
      missing_y);
```

Call the function with the board of size $2^k \times 2^k$ and the coordinates of the missing cell (where $2^k = n$):

```
fill_board(board, 0, 0, k, missing_x, missing_y)
```

Runtime Analysis
Recursively problem is divided into 4 subproblems of size n/2.
Let c be constant time taken by algorithm at each level of recursion. Therefore ,by above recurrence relations is:- T(n)=4T($\frac{n}{2}$) + c
Using master theorem and From above recurrence relationship, we know a=4. b=2 and *f(n)=c* $\log_b a = \log_2 4 = 2$
By master theorem,
Case 1: If $f(n) = O(n^c)$ where $c < \log_b a = 2$, then $T(n) = O(n^{\log_b a}) = O(n^2)$.

Case 2: If $f(n) = O(n^c)$ where $c = \log_b a = 2$, then $T(n) = O(n^{\log_b a} \log n) = O(n^2 \log n)$.

Case 3: If $f(n) = O(n^c)$ where $c > \log_b a = 2$, then $T(n) = O(f(n))$.

Since $f(n) = c = O(1)$, which is less than $\log_b a = 2$, we are in Case 1. Therefore, the solution to the recurrence relation is $T(n) = O(n^2)$. Solution of recurrence relation =time complexity for above algorithm i.e. $O(n^2)$

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: Here is the bottom up DP algorithm for above question.

Precomputation
The input set of line segments is sorted based on their x-coordinates.

Subproblems
We need to find the largest subset of segments in a given list that intersects each other.
Let DP(i) be the size of the largest intersecting subset of segments that includes the ith segment.
The base case is DP(0) = 0, which means the size of the largest intersecting subset of segments that includes the empty set is 0.
Then, for each i, DP(i) can be defined as follows: DP(i) = max DP(j) + 1 for all j ¡ i, where j is the largest index such that segment j intersects segment i, or j = 0 if no such segment exists.
We use bottom up approach to build upto larger subproblems(i=n) starting from smaller subproblems(j=0).

The subproblem that solves the original problem is dp[n], i.e., the largest subset of line segments that intersect with each other up to index n.

Psuedocode

```
Function LargestIntersectingSubset(L):
    Sort(L) ;          // Sort L in ascending order of start point of segments
    Initialize dp as an array of size |L| + 1, where dp[i] is a pair of (int, array of Segments)
    Set dp[0] = (0, ∅);
    for i from 1 to |L| do
        Initialize IncludeSize as 0 and IncludeSubset as an empty array of Segments
        for j from i − 1 down to 0 do
            if j = 0 or Intersects(L[i − 1], L[j − 1]) then
                if dp[j].first > IncludeSize then
                    Set IncludeSize to dp[j].first and IncludeSubset to dp[j].second ;
                end
                Increment IncludeSize by 1 and add L[i − 1] to IncludeSubset ;
            end
        end
        if include_size > dp[i − 1].first then
            Set dp[i] to (include_size, include_subset) ;
        end
        else
            Set dp[i] to dp[i − 1] ;
        end
    end
    return dp[|L|] ;
```

Proof Of Correctness

Proof. We prove the correctness of the algorithm by induction on the size of the input list.

Base case: For the input list of size 0, the algorithm correctly returns the empty set as the largest subset of segments that intersect each other.

Inductive hypothesis: Suppose that the algorithm correctly computes the largest subset of segments that intersects each other for all input lists of size $n-1$ or smaller.

Inductive step: Let S be an input list of size n, and let T be the largest subset of segments in S that intersects each other. We will show that the algorithm correctly computes T.

Suppose that the i-th segment of S is not in T. Then T is also a subset of the first $i-1$ segments of S, and the algorithm correctly computes the largest subset of segments that intersects each other for this subset by the induction hypothesis. Therefore, the i-th segment cannot be added to T without violating the intersection property.

Suppose that the i-th segment of S is in T. Let T' be the largest subset of segments in $S[1:i-1]$ that intersects each other and includes the i-th segment. Then T' is a candidate for T.

Let T'' be the largest subset of segments in $S[1:i-1]$ that intersects each other and does not include the i-th segment. By the induction hypothesis, the algorithm correctly computes T''.

If T'' has the same size as T', then both subsets are equally good solutions for $S[1:i]$, and the algorithm chooses T' because it includes the i-th segment.

If T'' is larger than T', then T is a subset of T''. Therefore, T' cannot be the largest subset of segments in $S[1:i]$ that intersects each other and includes the i-th segment.

Therefore, the algorithm correctly computes the largest subset of segments in S that intersects each other.

□

Runtime Analysis
1) Sorting the list of segments L takes $O(n \log n)$ time, where n is the length of L.
2) Initializing the array dp takes $O(n)$ time.
3) For each segment in L, the algorithm checks all previous segments in reverse order. Since there are n segments and for each segment, the algorithm checks at most $n-1$ previous segments, this step takes $O(n^2)$ time.
4) Other Steps involved takes constant time. Therefore ,from above the time complexity is $O(n^2)$ as $O(n^2) > O(n \log n) > O(n) > constant$

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 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. (Hint: use dynamic programming).

Ans: We will use bottom up dynamic programming approach to solve given question
Subproblems
We can involve any of the three agents for shipping in a week, given we can fulfill individual requirements of shipping agents.
If we involve company C, we can't ship with it for more than 2 consecutive weeks.
If we involve company B, we need to ship with for 3 consecutive weeks.

Let $dp[i][j]$ denote the minimum cost of scheduling shipments starting from week i, where j represents the number of consecutive weeks Company C has been used for shipping.

Recurrence relation:-

The base case is $dp(n+1,0) = dp(n+1,1) = dp(n+1,2) = 0$.

$dp(i,0) = min(dp(i+1,2)+a*s[i], dp(i+3,2)+b*s[i]+b*s[i+1]+b*s[i+2]) if n-i+1 >= 3$

$dp(i,0) = dp(i+1,2) + a*s[i] if n-i+1 < 3$

$dp(i,j) = min(dp(i+1,j-1)+c*s[i]-d, dp(i+1,2)+a*s[i]) if n-i+1 < 3 and j > 0$

$dp(i,j) = min(dp(i+1,2)+a*s[i], min(dp(i+3,2)+b*s[i]+b*s[i+1]+b*s[i+2], dp(i+1,j-1)+c*s[i]-d)) if n-i+1 >= 3 and j > 0$

where,

$s[i]$: Number of units manufactured in week i.

a, b, c, d: Cost and reward parameters for shipping agents A, B, and C.

n: Total number of weeks to consider.

The final solution will be provided by $dp[1][2]$ by using bottom up approach dynamic programming

---

```
Function minimumCost(s, a, b, c, d, n):
    dp ← 2D array of size (n + 1) × 3, initialized to ∞
    dp[n + 1][0] ← dp[n + 1][1] ← dp[n + 1][2] ← 0
    for i ← n down to 1 do
        for j ← 0 to 2 do
            if j = 0 then
                if n − i + 1 < 3 then
                    dp[i][j] ← dp[i + 1][2] + a · s[i];
                else
                    dp[i][j] ←
                      min(dp[i+1][2]+a·s[i], dp[i+3][2]+b·s[i]+b·s[i+1]+b·s[i+2]);
            else
                if n − i + 1 < 3 then
                    dp[i][j] ← min(dp[i+1][j−1] + c · s[i] − d, dp[i+1][2] + a · s[i]);
                else
                    dp[i][j] ← min(dp[i + 1][2] + a · s[i], min(dp[i + 3][2] + b · s[i] +
                      b · s[i + 1] + b · s[i + 2], dp[i + 1][j − 1] + c · s[i] − d));
    return dp[1][2];
```

Runtime Analysis

For each i, the algorithm only needs to compute the values of $dp[i][j]$ For $j = 0, 1, 2$, which requires constant time. The outer loop iterates over n values of i, so the total time complexity is proportional to n.

Also, the constant factors in the time complexity, such as the cost of arithmetic operations and memory access, are also constant and do not depend on n.

Therefore, the time complexity of the algorithm is $O(\text{n})$.