CSCI 570 – ANALYSIS OF ALGORITHMS HOMEWORK – 3

1. (**Greedy**) Given n rods of lengths $L_1, L_2, \ldots, L_n$, respectively, the goal is to connect all the rods to form a single rod. The length and the cost of connecting two rods are equal to the sum of their lengths. Devise a greedy algorithm to minimize the cost of forming a single rod.

ANS:

ALGORITHM TO FORM A SINGLE ROD MINIMIZING THE COST USING GREEDY APPROACH:

To minimize the cost of connecting all the rods into a single rod, we can use a greedy algorithm based on the following approach:

STEP 1: Create a min-heap using all the lengths of the rods L_1, L_2,.....L_n. Initialize a cost variable **total_cost** to keep a track of the total cost for merging the rods.

STEP 2: At each step, identify and remove two rods with the smallest lengths (first_min & second_min using 2 pop operations from min-heap) from the min-heap and merge them into a single rod. The cost of merging these two rods is equal to the sum of their lengths (first_min + second_min). Hence, increase our cost variable by sum (first_min + second_min) i.e., **total_cost** += first_min + second_min.

STEP 3: Insert the merged rod length i.e., first_min + second_min into the min-heap.

STEP 4: Repeat steps 2 and 3 until there is only one rod is left, which will be the final rod.

STEP 5: Cost variable (total_cost) will give us the total cost of forming a single rod.

2. (**Greedy**) During a summer music festival that spans m days, a music organizer wants to allocate time slots in a concert venue to various artists. However, each time slot can accommodate only one performance, and each performance lasts for one day.

   There are N artists interested in performing at the festival. Each artist has a specific deadline, D_i, indicating the last day on which they can perform, and an expected audience turnout, A_i, denoting the number of attendees they expect to draw if they perform on or before their deadline. It is not possible to schedule an artist's performance after their deadline, meaning an artist can only be scheduled on days 1 through D_i.

   The goal is to create a performance schedule that maximizes the overall audience turnout. The schedule can assign performances for n artists over the course of m days.

   Note: The number of performances (n) is not greater than the total number of artists (N) and the available days (m), i.e., $n \leq N$ and $n \leq m$. It may not be feasible to schedule all artists before their deadlines, so some performances may need to be skipped.

   (a) Let's explore a situation where a greedy algorithm is used to allocate n performance to m days consecutively, based on their increasing deadlines D_i. If, due to this approach, a task ends up being scheduled after its specified deadline D_i, it is excluded (not scheduled). Provide an counterexample to demonstrate that this algorithm does not consistently result in the best possible solution.

   (b) Let's examine a situation where a greedy algorithm is employed to distribute n performance across m days without any gaps, prioritizing performances based on their expected turnouts A_i in decreasing order.

   If, as a result of this approach, a performance ends up being scheduled after its specified deadline D_i, it is omitted from the schedule (not scheduled). Provide a counterexample to illustrate that this algorithm does not consistently produce the most advantageous solution.

   (c) Provide an efficient greedy algorithm that guarantees an optimal solution to this problem without requiring formal proof of its correctness.

ANS:

a) Let us consider a situation where a greedy algorithm is used to allocate n performance to m days consecutively, based on their increasing Deadlines D_i.

Given: If the performance is scheduled after Deadline D_i by following this algorithm, it should be excluded (not scheduled).

COUNTER EXAMPLE:

Let us consider there are 4 Artists with D_1, D_2, D_3 and D_4 as their respective deadlines and the audience turnout for each of their performances be A_1, A_2, A_3 and A_4 respectively. Let the summer music festival is organized on 3 days i.e., m = 3.

Artist 1 → $D_1 = 1$, $A_1 = 10$

Artist 2 → $D_2 = 2$, $A_2 = 20$

Artist 3 → $D_3 = 3$, $A_3 = 30$

Artist 4 → $D_4 = 3$, $A_4 = 40$

Maximum number of performances that can be allocated is n i.e., 3 in the best-case greedy approach.

GREEDY APPROACH BASED ON INCREASING DEADLINES OF ARTISTS:

1. If we sort artists based on their increasing deadlines we get [D_1, D_2, D_3, D_4 or D_1, D_2, D_4, D_3].
2. As the festival is only for 3 days, we can accommodate maximum of 3 performances (one on each day).

   In case 1, Greedy Algorithm will yield:
   - Audience Turnout = $A_1 + A_2 + A_3 = 10 + 20 + 30 = 60$

   In case 2, Greedy Algorithm will yield:
   - Audience Turnout = $A_1 + A_2 + A_4 = 10 + 20 + 40 = 70$

Therefore, the maximum audience turnout that can be obtained by using the greedy algorithm based on the increasing deadlines of artists is 70. Whereas the optimal solution will yield audience turnout of 90.

OPTIMAL SOLUTION:

Skip the Artist 1 and schedule Artist 2, Artist 3 and Artist 4 performances on Day 1, Day 2 and Day 3 respectively i.e., the total audience turnout will be

- Audience Turnout = $A_2 + A_3 + A_4 = 20 + 30 + 40 = 90$ [> 70 produced by Greedy Approach]

Hence, we can say that performances scheduled using greedy algorithm based on increasing deadlines D_i will not produce the optimal outcome in terms of audience turnout.

b) Let us consider a situation where a greedy algorithm is used to allocate n performance to m days consecutively, based on their decreasing order of audience turnouts A_i.

Given: If the performance is scheduled after Deadline D_i by following this algorithm, it should be excluded (not scheduled).

COUNTER EXAMPLE:

Let us consider there are 4 Artists with D_1, D_2, D_3 and D_4 as their respective deadlines and the audience turnout for each of their performances be A_1, A_2, A_3 and A_4 respectively. Let the summer music festival is organized on 3 days i.e., m = 3.

Artist 1 → $D_1 = 1$, $A_1 = 10$

Artist 2 → $D_2 = 2$, $A_2 = 20$

Artist 3 → $D_3 = 3$, $A_3 = 30$

Artist 4 → $D_4 = 3$, $A_4 = 40$

Maximum number of performances that can be allocated is n i.e., 3 in the best-case greedy approach.

GREEDY APPROACH BASED ON AUDIENCE TURNOUTS OF PERFORMANCES:

1. If we sort artists based on their decreasing order of audience turnouts (greater audience turnout should be prioritized first], we get the scheduling order [A_4, A_3, A_2, A_1].
2. As the festival is only for 3 days, we can accommodate maximum of 3 performances (one on each day).

Day 1 → Artist 4 will be allocated for performing on day as A_4 is our first priority (based on max audience turnout) $D_4 = 3 > 1$(Day allocated) → Deadline is on or before Day 3 but given allocation on Day 1. Hence, this holds true and the audience turnout for Day 1 is A_4 i.e., 40.

Day 2 → Artist 3 will be allocated for performing on day as A_3 is our next priority (based on max audience turnout) $D_3 = 3 > 1$(Day allocated) → Deadline is on or before Day 3 but given allocation on Day 2. Hence, this holds true and the audience turnout for Day 2 is A_3 i.e., 30.

Day 3 → At this point we are left with 2 artists Artist 1 and Artist 2 but we cannot schedule any performance or allocate any of these artists because Day 3 > $D_1 = 1$ & $D_2 = 2$. Hence, we cannot schedule any performance on Day 3 and the audience turnout for Day 3 is 0.

- Total Audience Turnout for the Festival = Sum (Audience Turnout on each day)

=> **Total Audience Turnout = 40 + 30 + 0 = 70**

OPTIMAL SOLUTION:

Skip the Artist 1 and schedule Artist 2, Artist 3 and Artist 4 performances on Day 1, Day 2 and Day 3 respectively i.e., the total audience turnout will be

- Audience Turnout = $A_2 + A_3 + A_4 = 20 + 30 + 40 = 90$ [> 70 produced by Greedy Approach]

Hence, we can say that performances scheduled using greedy algorithm based on decreasing order of audience turnouts A_i will not produce the optimal outcome in audience turnout.

c) Efficient Greedy Algorithm that guarantees an Optimal Solution.

GREEDY APPROACH FOR OPTIMAL SOLUTION:

Festival is held on m days. Given that number of artists (N) is ≥ Number of performances (n) and Number of days festival commences is (m) ≥ Number of performances (n).

STEP 1: Initialize a variable called Total_Audience_Turnout = 0 to keep a track of total audience turnout, Performance_Schedule = [ ] to keep track of allocation on each day and Current_Deadline = m, Current_max to traverse.

STEP 2: Sort all the Artists in Descending order of their Deadlines (also indicating the Artist number).

STEP 3: Now initialize a max-heap and insert all the Audience Turnout values of Artists whose Deadline = Current_Deadline.

STEP 4: Pop the max Audience Turnout value from the max heap (root) and update Current_max to the value popped from the max-heap

- Increase out Total_Audience_Turnout += Current_max
- Append (Artist i, Current_max, Current_Deadline) to Performance Schedule
- If Current_Deadline > 1 then Current_Deadline -= 1 and Repeat Steps 3 and 4, Else Stop

Repeat this process until Current_Deadline is 1 and the iteration stops.

STEP 5: The Total_Audience_Turnout will give us the value of Maximum Audience Turnout possible and the Performance Schedule gives out the values of which artist (Artist i) is performing on what day and what is the maximum audience turnout for each day.

3. (**Master Theorem**) The recurrence $T(n) = 7T(n/2) + n^2$ describes the running time of an algorithm ALG. A competing algorithm ALG' has a running time of $T'(n) = aT'(n/4) + n^2 \log n$ What is the largest value of a such that ALG' is asymptotically faster than ALG?

ANS:

The Master Theorem

$T(n) = a \cdot T(n/b) + f(n)$, $a \geq 1$ and $b > 1$

Let $c = \log_b a$.

Case 1: (only leaves)
if $f(n) = O(n^{c-\varepsilon})$, then $T(n) = \Theta(n^c)$ for some $\varepsilon > 0$.

Case 2: (all nodes)
if $f(n) = \Theta(n^c \log^k n)$, $k \geq 0$, then $T(n) = \Theta(n^c \log^{k+1} n)$

Case 3: (only internal nodes)
if $f(n) = \Omega(n^{c+\varepsilon})$, then $T(n) = \Theta(f(n))$ for some $\varepsilon > 0$.

To determine the largest value of "a" such that ALG' is asymptotically faster than ALG, we can use the Master Theorem. The Master Theorem provides a framework for solving recurrence relations of the form:

$$T(n) = aT(\tfrac{n}{b}) + f(n)$$

In our case, for ALG, $a = 7, b = 2$, and $f(n) = n^2$.

$=> c = log_b^a$ => $c = log_2^7 = 2.8073549221$

Hence, it falls under Case 1 of Master's Theorem as $n^{log_2^7} > n^2$. This states that the run-time complexity of the ALG will be $O(n^{log_2^7})$

For ALG', $a = a, b = 4$, and $f(n) = n^2 \log n$.

$=> c = log_b^a$
=> $c = log_4^a < log_2^7$ (2.8073549221)

Solving the inequation, we get

=> $1 \leq a < 49$ (in the world of computer science → $\boldsymbol{a = 48}$ maximum possible value)

When $log_4^a \leq 2$, by case 2 and case 3 of the Master Theorem, we won't get a faster algorithm than ALG. As in the question it requires to find a maximum possible a value where ALG' is faster than ALG, we can take $\boldsymbol{a = 48}$ as maximum value possible so that (ALG' is faster than ALG).

4. (**Master Theorem**) Consider the following algorithm `StrangeSort` which sorts n distinct items in a list A.

   (a) If $n \leq 1$, return A unchanged.

   (b) For each item $x \in A$, scan A and count how many other items in A are less than x.

   (c) Put the items with count less than $n/2$ in a list B.

   (d) Put the other items in a list C.

   (e) Recursively sort lists B and C using `StrangeSort`.

   (f) Append the sorted list C to the sorted list B and return the result.

   Formulate a recurrence relation for the running time $T(n)$ of `StrangeSort` on an input list of size n. Solve this recurrence to get the best possible $O(\cdot)$ bound on $T(n)$.

ANS:

To formulate a recurrence relation for the running time $T(n)$ of StrangeSort, let's analyze each step of the algorithm:

(a) If $n \leq 1$, return A unchanged.

In this case, the algorithm does nothing, so it takes constant time: $T(1) = O(1)$.

(b) For each item $x \in A$, scan A and count how many other items in A are less than x.

This step involves scanning the entire list A for each element x, resulting in a time complexity of $O(n^2)$ for this step.

(c) Put the items with count less than $n/2$ in a list B.

This step involves selecting items based on a count criterion, which takes linear time: $O(n)$.

(d) Put the other items in a list C.

This step involves selecting items not included in list B, which also takes linear time: $O(n)$.

(e) Recursively sort lists B and C using StrangeSort. Let $T\_B(n/2)$ be the time it takes to sort list B of size $n/2$, and $T\_C(n/2)$ be the time it takes to sort list C of size $n/2$. We will use these recursive calls in our recurrence relation.

(f) Append the sorted list C to the sorted list B and return the result.

This step involves combining the sorted lists B and C, which takes linear time: $O(n)$.

Now, let's formulate the recurrence relation for $T(n)$:

- $T(n) = O(1)$ if n ≤ 1 (Base case)
- $T(n) = O(n^2) + T\_B(n/2) + T\_C(n/2) + O(n)$ if $n > 1$

Now, we can solve this recurrence relation using the Master Theorem.

$T(n) = O(n^2) + 2T\left(\frac{n}{2}\right) + O(n)$

Comparing this to the Master Theorem's generic form:

$T(n) = aT(\frac{n}{b}) + f(n)$

We have:

$a = 2$ (Two recursive calls: one for B and one for C)

$b = 2$ (The input size is divided in half)

$f(n) = O(n^2) + O(n)$ (non-recursive work)

$=> f(n) = O(n^2) + O(n) \approx O(n^2)$ [Dominant Term]

Since $f(n) = O(n^2)$ is polynomially larger than $n^{log\,\frac{a}{b}}$, we are in case 3 of the Master Theorem.

Case 3 of the Master Theorem states that if $f(n)$ is polynomially larger than $n^{log\,\frac{a}{b}}$, then the recurrence has a solution of the form:

$T(n) = \theta(f(n))$

In our case, $\boldsymbol{T(n) = \theta\,(n^2)}$.

So, the best possible bound on the running time of StrangeSort is $\boldsymbol{O(n^2)}$.

5. (**Master Theorem**) For the given recurrence equations, solve for $T(n)$ if it can be found using the Master Method. Else, indicate that the Master Method does not apply.

(a) $T(n) = T(n/2) + 2^n$

(b) $T(n) = 5T(n/5) + n \log n – 1000n$

(c) $T(n) = 2T(n/2) + \log^2 n$

(d) $T(n) = 49T(n/7) - n^2 \log n^2$

(e) $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

ANS:

The Master Theorem

$T(n) = a \cdot T(n/b) + f(n)$, $a \geq 1$ and $b > 1$

Let $c = \log_b a$.

Case 1: (only leaves)
if $f(n) = O(n^{c-\varepsilon})$, then $T(n) = \Theta(n^c)$ for some $\varepsilon > 0$.

Case 2: (all nodes)
if $f(n) = \Theta(n^c \log^k n)$, $k \geq 0$, then $T(n) = \Theta(n^c \log^{k+1} n)$

Case 3: (only internal nodes)
if $f(n) = \Omega(n^{c+\varepsilon})$, then $T(n) = \Theta(f(n))$ for some $\varepsilon > 0$.

(a) $T(n) = T(n/2) + 2^n$

Here, $a = 1, b = 2$, and $f(n) = 2^n$.

$=> c = log_b^a$
$=> c = log_2^1 \approx 0$

i.e., **Case 3** where $f(n)$ dominates the leaves and hence, the $\boldsymbol{T(n) = \theta(2^n)}$ for the given recurrence equation (a).

(b) $T(n) = 5T\left(\frac{n}{5}\right) + n \log n - 1000n$

Here, $a = 5$, $b = 5$, and $f(n) = n \log n - 1000n$

$=> c = log_b^a$
$=> c = log_5^5 = 1$

=> Comparing n *and* $n \log n$ (as $n \log n$ is dominant term for huge values of n)

i.e., **Case 2** where the complexities of the leaves and $f(n)$ are asymptotically equal and hence, the $\boldsymbol{T(n) = \theta(n \log^2 n)}$ for the given recurrence equation (b).

c) $T(n) = 2T(n/2) + \log^2 n$

Here, $a = 2$, $b = 2$, and $f(n) = \log^2 n$.

$=> c = log_b^a$
$=> c = log_2^2 = 1$

$=> (n > \log^2 n)$

i.e., **Case 1** where leaves dominates the $f(n)$ and hence, the $\boldsymbol{T(n) = \theta(n)}$ for the given recurrence equation (c).

d) $T(n) = 49T(n/7) - n\log^2 n$

Here, $a = 49$, $b = 7$, and $f(n) = -n\log^2 n$

Master's Theorem doesn't apply as $f(n)$ is a negative function.

(e) $T(n) = 3T\left(\frac{n}{4}\right) + n\ log\ n$

Here, $a = 3$, $b = 4$, and $f(n) = n\ log\ n$.

$=> c = log_b^a$
$=> c = log_4^3$

$=> As\ log_4^3 < 1$

Regularity Check: Find value of c < 1 such that $af(\frac{n}{b}) \leq cf(n)$ + we can write the above relation as $\frac{3n}{4}\log\frac{n}{4} \leq c\ n\ log\ n$ and $c = \frac{3}{4}$ will satisfy the equation.

i.e., **Case 3** where $f(n)$ dominates the $leaves$ and hence, the $\boldsymbol{T(n) = \theta(n\ log\ n)}$ for the given recurrence equation (e).

6. (**Divide-and-Conquer**) We know that binary search on a sorted array of size n takes $\Theta(\log n)$ time. Design a similar divide-and-conquer algorithm for searching in a sorted singly linked list of size n. Discuss its worst-case runtime complexity.

ANS:

To design a divide-and-conquer algorithm for searching in a sorted singly linked list of size n, we can adapt the binary search concept. Binary search on arrays works efficiently because we can directly access elements in constant time by their indices. In a singly linked list, direct indexing is not possible, so we will use a divide-and-conquer approach that involves splitting the list into two halves recursively.

DIVIDE AND CONQUER ALGORITHM:

We have Global_Start = Head of our sorted Singly Linked List. Initialize two pointers Start and End, initially pointing to the head of the linked list.

STEP 1: If the address value at Start pointer is pointing to NULL, STOP.

End pointer (2i) moves 2 places for each move of Start Pointer (i). When the End pointer points to null, Start pointer points to the mid element of our sorted singly linked list.

STEP 2: Compare the search query (element to be searched) with the value at the mid node (the one start pointer is pointing).

STEP 3:

- If they are equal, return value at mid, indicating that the element is found and STOP.
- If the query value is less than the value at mid, update the address value at mid to NULL, Move the Start pointer and End pointer to Global Start and again perform Steps 1 to 3.
- If the query value is more than the value at mid, Point the Start pointer and End pointer to mid+1 and again perform Steps 1 to 3.

STEP 4: Repeat the process recursively on the selected half until the element is found or Terminate

The algorithm terminates when the element is found, or the search range is empty. Now, let's analyze the worst-case runtime complexity:

In each step, we reduce the size of the search space by half, which is similar to binary search on arrays. Therefore, at each recursive step, we are left with approximately half the remaining elements. The worst-case time complexity is logarithmic in nature, just like binary search on an array but traversing takes n:

$$T(n) = T(n/2) + O(n)$$

Here, $T(n)$ represents the time complexity for a list of size n. Using the Master Theorem, this recurrence relation falls into case 3:

$T(n) = T(n/2) + O(n)$

$a = 1, b = 2, and\ f(n) = O(n).$

$=> c = log_b^a => c = log_2^1$

$=> As\ log_2^1 < 1$

Therefore, the time complexity of the divide-and-conquer algorithm for searching in a sorted singly linked list of size n is: **Case 3** where $f(n)$ dominates the $leaves$ and hence, the $\boldsymbol{T(n) = \theta(n)}$ for the given recurrence equation (e). So, the worst-case runtime complexity is $\boldsymbol{O(n)}$.

7. (**Divide-and-Conquer**) We know that mergesort takes $\Theta(n \log n)$ time to sort an array of size n. Design a divide-and-conquer mergesort algorithm for sorting a singly linked list. Discuss its worst-case runtime complexity.

ANS:

Designing a divide-and-conquer merge sort algorithm for sorting a singly linked list is a bit more complex than sorting an array, primarily because you cannot access elements by index in a singly linked list. Instead, you'll need to split the list into smaller sublists, sort those sublists recursively, and then merge them back together.

DIVIDE AND CONQUER ALGORITHM:

STEP 1: If the linked list contains zero or one element, it is already sorted, so return the list.

STEP 2: Split the input linked list into two (roughly in case of odd number of elements) equal-sized sublists. You can do this by traversing the list with two pointers: one that moves one step at a time, and another that moves two steps at a time. When the faster pointer reaches the end of the list, the slower pointer will be at the midpoint, allowing you to split the list into two halves.

STEP 3: Recursively break the lists into sublists and sort both halves of the list using the merge sort algorithm.

STEP 4: Merge the two sorted sublists back into a single sorted list. This is similar to the merge step in the standard merge sort algorithm for arrays, but you'll need to traverse the lists using pointers since you can't access elements by index in a singly linked list.

Now, let's discuss the worst-case runtime complexity of this algorithm:

Splitting: Splitting the list into two halves takes $O(n)$ time in the worst case, where n is the number of elements in the list.

Recursion: Recursively sorting each half takes $T(n/2)$ time for each half. Since there are two halves, this step contributes $2T(n/2)$ to the runtime.

Merging: Merging two sorted halves into one sorted list takes $O(n)$ time, as you need to traverse each element in both halves to merge them.

Now, let's write the recurrence relation for the runtime:

$$T(n) = 2T(n/2) + O(n) + O(n)$$

Using the Master Theorem, we can analyze it:

$a = 2$ (Two recursive calls, one for each half)

$b = 2$ (Each time the input size is halved)

The Master Theorem provides us with the following case:

$$T(n) = aT(n/b) + f(n)$$

$=> T(n) = 2T(n/2) + O(n)$

$a = 2, b = 2, and\ f(n) = O(n)$ (Merging step) + $O(n)$ (Breaking step)

=> $f(n) \approx$ Asymptotically $O(n)$

$c = log_b^a$

=> $c = log_2^2 = 1$

i.e., **Case 2** where the complexities of the leaves and $f(n)$ are asymptotically equal and hence, the $\boldsymbol{T(n) = \theta(n\ log\ n)}$ for the given recurrence equation.

Therefore, the worst-case runtime complexity of the mergesort algorithm for singly linked lists is $\boldsymbol{T(n) = \theta(n\ log\ n)}$.

8. (**Divide-and-Conquer**) Imagine you are responsible for organizing a music festival in a large field, and you need to create a visual representation of the stage setup, accounting for the various stage structures. These stages come in different shapes and sizes, and they are positioned on a flat surface. Each stage is represented as a tuple (L, H, R), where L and R are the left and right boundaries of the stage, and H is the height of the stage.

Your task is to create a skyline of these stages, which represents the outline of all the stages as seen from a distance. The skyline is essentially a list of positions (x-coordinates) and heights, ordered from left to right, showing the varying heights of the stages.

Take Fig. 1 as an example: Consider the festival setup with the following stages: (2, 5, 10), (8, 3, 16), (5, 9, 12), (14, 7, 19). The skyline for this festival setup would be represented as: (2, 5, 5, 9, 12, 3, 14, 7, 19), with the x-coordinates sorted in ascending order.

(a) Given the skyline information of n stages for one part of the festival and the skyline information of m stages for another part of the festival, demonstrate how to compute the combined skyline for all $m+n$ stages efficiently, in $O(m+n)$ steps.

(b) Assuming you've successfully solved part (a), propose a Divide-and-Conquer algorithm for computing the skyline of a given set of n stages in the festival. Your algorithm should run in $O(n \log n)$ steps.

ANS:

Given that each stage is represented by a tuple (L, H, R) where L, R are the left and right boundaries of the stage and H is the height of the stage.

Any stage of the form (L, H, R) can also be represented (L, H, R, 0).

(a): Required to find the Combined skyline view of $m+n$ stages in $O(m+n)$.

Given Festival1 has skyline of m stages.

Festival1 is of the form $(x_1, h_1, x_2, h_2, x_3, \dots, x_m)$.

Given Festival2 has skyline of n stages.

Festival2 is of the form $(x'_1, h'_1, x'_2, h'_2, x'_3, \dots, x'_n)$.

ALGORITHM:

1. Let's take 2 arrays leftarr and rightarr.
2. Initialize an array called result to store the resultant.
3. Iterate the left list and right list by keeping a track of leftH and rightH.
4. Initialize leftH=0 and rightH=0.
5. Create two pointers leftPointer = 0 and rightPointer = 0, pointing to the first elements of the corresponding lists.

6. Check (leftPointer < length(leftarr) or rightPointer < length(rightarr)) and the minimum(leftarr[leftPointer], rightarr[rightPointer]). If (leftPointer < length(leftarr) or rightPointer < length(rightarr)) is false, return the result and stop the algorithm. Else go to Step 7.
7. If minimum is from the leftarr, pop 2 elements (x1, h1) from the leftarr. Add x1 to the result. Update leftH = h1. Go to Step 8.
   Else go to Step 9.
8. Compare leftH and rightH and push maximum (leftH, rightH) into result. Increment the leftPointer such that it points to next available x-coordinate in leftarr (given that leftarr is not empty).
9. If minimum is from the rightarr, pop 2 elements (x1, h1) from the rightarr. Add x1 to the result. Update rightH = h1. Go to Step 10.
   Else go to Step 6.
10. Compare leftH and rightH and push maximum (leftH, rightH) into result. Increment the rightPointer such that it points to next available x-coordinate in rightarr, if rightarr is not empty. Go to Step 6.

Therefore, using the two pointers with some constraints, we can achieve combined skyline view of 2 lists of sizes m and n in $O(m+n)$ time complexity.

(b) <u>DIVIDE AND CONQUER ALGORITHM</u>:

1. If there is one building, return building.
2. If there are n buildings, recursively split the n buildings into two lists leftarr and rightarr of size $n/2$ each.
3. Merge those 2 buildings of size $n/2$ each and return result. As We have done in part (a)

Recurrence relation:

At each level 1 list of 'n' buildings is split into 2 lists of $n/2$ buildings. This can be done in $O(1)$ time complexity.

At each level we are merging lists of different sizes $n_1, n_2 \ldots, n_n$, such that Time complexity at each level is $O(n_1 + n_2 + \cdots + n_n) \approx O(n)$.

The recurrence relation can be written as follows,

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

According to the master's theorem, we have $a = 2, b = 2$ and $f(n) = n$ and we get $c = 1$. This falls into Case 2 of master's theorem.

Therefore, $T(n) = \theta(nlog(n))$. We have $\log(n)$ levels and each level have time complexity of $O(n)$.

9. (**Dynamic Programming**) Imagine you are organizing a charity event

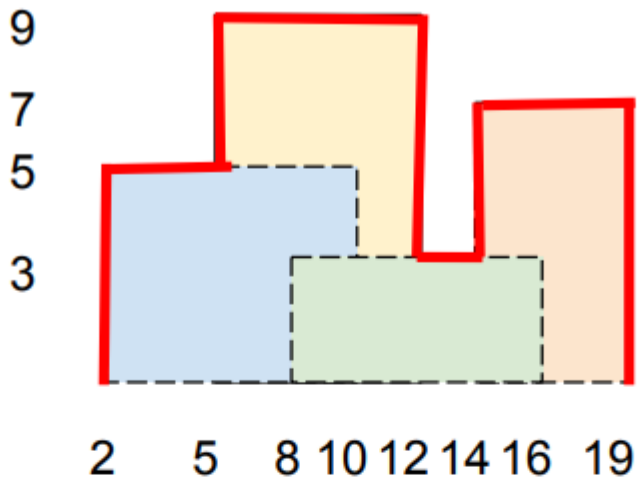

Figure 1: Example of festival stage setup.

to raise funds for a cause you care deeply about. To reach your fundraising goal, you plan to sell two types of tickets.

You have a total fundraising target of n dollars. Each time someone contributes, they can choose to buy either a 1-dollar ticket or a 2-dollar ticket. Use **Dynamic Programming** to find the number of distinct combinations of ticket sales you can use to reach your fundraising goal of n dollars?

For example, if your fundraising target is 2 dollars, there are two ways to reach it: 1) sell two 1-dollar tickets; 2) sell one 2-dollar ticket.

(a) Define (in plain English) subproblems to be solved.

(b) Write a recurrence relation for the subproblems

(c) Using the recurrence formula in part b, write pseudocode using iteration to compute the number of distinct combinations of ticket sales to reach fundraising goal of n dollars.

(d) Make sure you specify base cases and their values; where the final answer can be found.

(e) What is the runtime complexity of your solution? Explain your answer.

ANS:

(a) Subproblems:

Using dynamic programming to find the number of distinct combinations of ticket sales to reach a fundraising goal of n dollars

we can define the following subproblems:

How many distinct combinations are there to reach a fundraising goal of 0 dollars?

How many distinct combinations are there to reach a fundraising goal of 1 dollar?

How many distinct combinations are there to reach a fundraising goal of 2 dollars?

… How many distinct combinations are there to reach a fundraising goal of n dollars?

(b) Recurrence relation:

Let $dp[i]$ represent the number of distinct combinations to reach the fundraising goal of i dollars. We can formulate the recurrence relation as follows:

$$dp[i] = dp[i-1] + dp[i-2]$$

This recurrence relation means that the number of distinct combinations to reach the fundraising goal of i dollars is equal to the sum of the number of distinct combinations to reach the goal of $(i-1)$ dollars and the number of distinct combinations to reach the goal of $(i-2)$ dollars.

(c) Pseudo Code:

```
def find_Combinations(n):
    if n <= 1:
        return n #If condition to break the recurrence

    dp = [0] * (n + 1) # Create an array to store the number of
    distinct combinations for each goal from 0 to n.
    dp[0] = 1 # Base Case 1 - There's one way to reach the goal of
    0 dollars (no tickets sold).
    dp[1] = 1 # Base Case 2 - There's one way to reach the goal
    of 1 dollar (one 1-dollar ticket sold).

    for i in range (2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
        # Iteration to find the distinct combinations to raise i-
        1 and i-2 dollars respectively.
```

```
    # The final answer is stored in dp[n], which represents the
    number of distinct combinations to reach the fundraising goal
    of n dollars.
    return dp[n]
```

(d) Base Cases:

- $dp[0] = 1$: There is one distinct combination to reach the fundraising goal of 0 dollars (no tickets sold).
- $dp[1] = 1$: There is one distinct combination to reach the fundraising goal of 1 dollar (one 1-dollar ticket sold).
- The final answer is stored in $dp[n]$, which represents the number of distinct combinations to reach the fundraising goal of n dollars.

(e) Runtime Complexity:

The pseudocode provided above uses a bottom-up dynamic programming approach. It calculates the number of distinct combinations iteratively from 2 to n based on the recurrence relation. Since each calculation depends on the results of the two previous calculations, the algorithm has a time complexity of $\boldsymbol{O(n)}$.

Therefore, the runtime complexity of this solution is **linear** $\boldsymbol{O(n)}$ in terms of the fundraising goal, making it efficient for calculating the number of distinct combinations to reach a given fundraising target.

10. (**Dynamic Programming**) Assume a truck with capacity W is loading. There are n packages with different weights, i.e. $(w_1, w_2, \ldots w_n)$, and all the weights are integers. The company's rule requires that the truck needs to take packages with exactly weight W to maximize profit, but the workers like to save their energies for after work activities and want to load as few packages as possible. Assuming that there are combinations of packages that add up to weight W, design an algorithm to find out the minimum number of packages the workers need to load.

    (a) Define (in plain English) subproblems to be solved.

    (b) Write a recurrence relation for the subproblems

    (c) Using the recurrence formula in part b, write pseudocode using iteration to compute the minimum number of packages to meet the objective.

    (d) Make sure you specify base cases and their values; where the final answer can be found.

    (e) What is the worst case runtime complexity? Explain your answer.

ANS:

(a) Subproblems:

To find the minimum number of packages needed to meet the weight requirement W, we can define the following subproblems:

What is the minimum number of packages needed to reach a weight of 0?

What is the minimum number of packages needed to reach a weight of 1?

What is the minimum number of packages needed to reach a weight of 2?

… What is the minimum number of packages needed to reach a weight of W?

(b) Recurrence relation:

Let $dp[i]$ represent the minimum number of packages needed to reach a weight of i. We can formulate the recurrence relation as follows:

$$dp[i] = min(dp[i - w_j] + 1) \; for \; all \; j \; such \; that \; w_j <= i.$$

This recurrence relation means that to find the minimum number of packages needed to reach a weight of i, we consider all packages with weights less than or equal to i and calculate the minimum number of packages needed by adding one package of weight w_j to the minimum number of packages needed to reach a weight of $(i - w_j)$.

(b) Pseudo Code:

```
def minPackages(weights, W):
      n = len(weights)
      dp = [float('inf')] * (W + 1)
      dp [0] = 0 # Base Case 1 - There's one way to reach the goal
of 0 weight.
      for i in range(1, W + 1):
            for j in range(n):
                  if weights[j] <= i:
                      dp[i] = min(dp[i], dp [i - weights[j]] + 1)
      # Computing the minimum number of items with given weights
      so that we can reach W).
     return dp[W]
```

NOTE: Given that there's a combination of packages that exists which adds up exactly to W, the algorithm gives correct output.

(d) Base Cases:

$dp[0] = 0$: To reach a weight of 0, no packages are needed (Base Case).

All other elements in dp are initialized to positive infinity (float('inf')) to ensure that they are larger than any possible number of packages required.

The final answer is found in $dp[W]$, which represents the minimum number of packages needed to reach the weight requirement W.

(e) Worst-Case runtime complexity:

The pseudocode provided above uses dynamic programming with two nested loops. The outer loop runs from 1 to W, and the inner loop runs for each package weight. In the worst case, the inner loop iterates through all n packages for each value of i from 1 to W. Therefore, the worst-case runtime complexity is $O(n * W)$, where n is the number of packages and W is the weight requirement. Runtime complexity is pseudo polynomial and is dependent on size of weights.

It's important to note that the time complexity depends on both the number of packages (n) and the weight requirement (W). In practice, if W is very large compared to n, the algorithm may become less efficient due to its dependency on W.

********** **THE END** **********