CSCI 570 Homework 3

Due Date: Oct. 1, 2023 at 11:59 P.M.

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.

   - Create a min-heap and insert all lengths into the min-heap.
   - Do following while the number of elements in min-heap is greater than one.
     - Extract the minimum and second minimum from min-heap
     - Add the above two extracted values and insert the added value to the min-heap.
     - Maintain a variable for total cost and keep incrementing it by the sum of extracted values.

   Return the value of total cost.

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.

$m = 2, N = 3, D = (1, 2, 3), A = (10, 20, 30)$

Algorithm will give below day to performance mapping

day 1: Performance 1: turnout = 10

day 2: Performance 2: turnout = 20

total turnout 30

Optimal solution (Performance 2, Performance 3) 30+20 = 50.

(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.

$m = 2, N = 3, D = (1, 2, 3), A = (20, 10, 30)$

Algorithm will give below day to performance mapping

day 1 : Performance 3: turnout = 30

day 2 : can't schedule Performance 1 as its deadline is gone, hence schedule Performance 2: turnout = 10

total turnout = 40

1 optimal solution (Performance 1, Performance 3) 20+30 = 50.

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

Sort all performances in decreasing order of expected turnouts.

Iteratively assign the Performance i to the last available day before or on its deadline. This day is now unavailable for other performances. If no slot is available. Skip this performance.

Other solutions may exist

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?

   For T(n), setting $0 < \epsilon < \log_2 7 - 2 \simeq 0.81$ we have $n^2 = O(n^{log_2(7\ -\epsilon)})$. Hence, $T(n) = \Theta(n^{\log_2 7})$. For ALG', if $log_4 a > 2$ then setting $0 < \delta < \log_4\ a - 2$ implies $n^2 \log n = O(n^{\log_4 a - \delta})$ and thus, $T'(n) = \Theta(n^{\log_4 a})$. For $T'(n)$ to be asymptotically faster than $T(n)$, we need $T'(n) = O(T(n))$. Thus, $n^{\log_4 a} = O(n^{\log_2 7})$. Hence $log_4 a < \log_2 7$, thus, $a \leq 49$.

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 $T(n)$.

   Suppose that A has n elements. Then steps (c), (d) and (f) can take up to O(n) time, step (a) takes constant time and step (e) takes 2T(n/2) time. Step (b), when implemented in the way described above, takes $\Theta(n^2)$ time since each scan of A takes $\Theta(n)$ time and the scan is repeated for each of the n elements of A. Counting times taken for all the steps, we have

$$T(n) = 2T(n/2) + O(n) + \Theta(n^2) + O(1) \tag{1}$$

And thus, using the Master theorem, we have $T(n) = \Theta(n^2)$

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

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

   $n^{log_2^1} = n^0 = 1$

   Case 3 Master Theorem: $f(n) = \Omega(n \log_2^{1+\epsilon}), 2^{n-1} \leq c2^n \Rightarrow T(n) = \Theta(2^n)$

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

   $a = 5, b = 5, f(n) = n \log n – 1000n$

   $n^{\log_5^5} = n^1 = n$

   General Case 2: $f(n) = \Theta(n^{\log_5^5} \log^1 n) = \Theta(n \log n) \Rightarrow T(n) = \Theta(n \log^2 n)$

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

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

   $n^{\log_2^2} = n^1 = n$

   Case 1: $\log^2 n = O(n \log_2^{2-\epsilon}) \Rightarrow T(n) = \Theta(n)$

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

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

   Not applicable

   The problem is not a correct form to apply Master theorem: $T(n) = aT(n/b) + f(n)$. The prerequisite regarding the positive $f(n)$ is not satisfied.

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

   To solve the recurrence using the Master Theorem, we need to compare the given recurrence with the general form:

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

   where:

   - $a \geq 1$ is the number of subproblems.

- $b > 1$ is the factor by which the problem size decreases.
- $f(n)$ is the cost of the work done outside the recursive calls.

From our recurrence, $a = 3$, $b = 4$, and $f(n) = n \log n$.
Comparing the given $f(n)$ with the general formula, we have:

$$n^{\log_b a} = n^{\log_4 3}$$

Thus:

$$f(n) = \Omega(n^{\log_b a})$$

.
Thus, using the third case of the Master Theorem, we can conclude that:

$$T(n) = \Theta(n \log n)$$

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.

   A sorted singly linked list doesn't provide constant-time access to its middle element, so getting the middle element takes O(n) time. Because of this, we can't achieve O(logn) time complexity as in binary search on arrays, where we can access the middle element in constant time. However, we can still design a divide-and-conquer algorithm with a time complexity of O(n) by traversing the linked list to find the middle element.

   Here's an outline of the algorithm:

   (a) Traverse the list to find the middle node. Let's call it mid.
   (b) Compare the data of mid with the target value.
   (c) If they match, return mid.
   (d) If the target is less than mid$\rightarrow$data, repeat the process for the left half of the list.
   (e) If the target is greater than mid$\rightarrow$data, repeat the process for the right half of the list.

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.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as $A[p \ldots r]$.

- **Divide** If q is the half-way point between p and r, then we can split the subarray $A[p \ldots r]$ into two arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$.

  Details: we split the nodes of the given list into front and back halves, and return the two lists using the reference parameters.
  We use the fast/slow pointer strategy; every time, we advance 'fast' two nodes, and advance 'slow' one node; when the 'fast' node reaches the end of list, we stop and use the 'slow' node as q and its next node as $q + 1$. Below is the pseudo-code for implementation:

```
FrontBackSplit(Node* source, Node** frontRef, Node** backRef)
{
   Node* fast;
   Node* slow;
   slow = source;
   fast = source→next;
   while (fast != NULL) {
      fast = fast→next;
      if (fast != NULL) {
         slow = slow→next;
         fast = fast→next;
      }
   }
   *frontRef = source; // for mergeSort(A, p, q)
   *backRef = slow→ next; // for mergeSort(A, q+1, r)
```

```
    slow→ next = NULL;
}
```

- **Conquer** In the conquer step, we try to sort both the subarrays $A[p \dots q]$ and $A[q+1 \dots r]$. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

- **Combine** When the conquer step reaches the base step and we get two sorted subarrays $A[p \dots q]$ and $A[q+1 \dots r]$ for array $A[p \dots r]$, we combine the results by creating a sorted array $A[p \dots r]$ from two sorted subarrays $A[p \dots q]$ and $A[q+1 \dots r]$.

  Details: The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array:

  Have we reached the end of any of the arrays?

  - No: Compare current elements of both arrays Copy smaller element into sorted array Move pointer of element containing smaller element
  - Yes: Copy all remaining elements of non-empty array

Pseudo code:

```
MergeSort(A, p, r):
  if $p > r$
    return
  $q = \lfloor \frac{p+r}{2} \rfloor$
  mergeSort(A, p, q)
  mergeSort(A, q+1, r)
  merge(A, p, q, r)
```

To sort an entire array, we need to call MergeSort(A, 0, length(A)-1).

Time complexity: $O(n \log n)$

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.

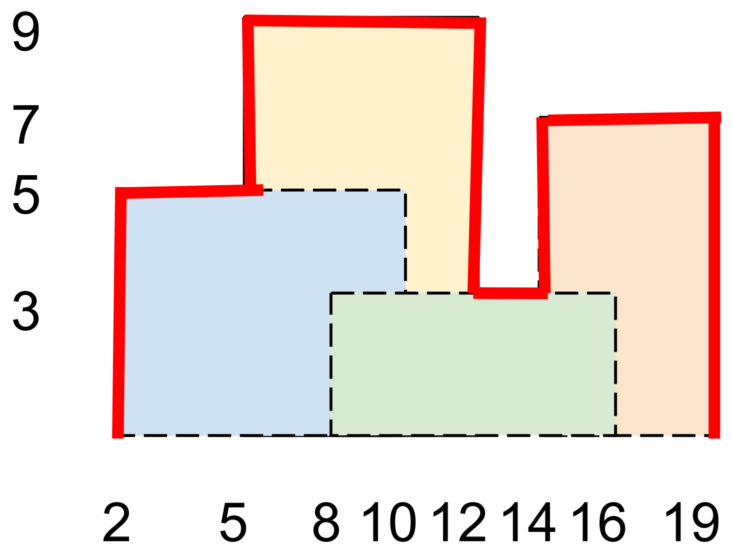

Figure 1: Example of festival stage setup.

(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.

Suppose we have skyline $(x_1, h_1, x_2, h_2, ...x_n)$ for n stages and skyline $(x'_1, h'_1, x'_2, h'_2, \ldots, x'_m)$ for m stages.

Merge the "left" list and the "right" list iteratively by keeping track of the current left height (initially 0), the current right height (initially 0), finding the lowest next x-coordinate in either list; we assume it is the left list.

We remove the first two elements, x and h, and set the current left height to h, and output x and the maximum of the current left and right heights.

(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.

If there is one stage, output it.

Otherwise, split the stages into two equal groups, recursively compute skylines, output the result of merging them using part (a).

Algorithm:

getSkyline for n stages:

```
i. If $n == 0$: return an empty list.
ii. If $n == 1$: return the skyline for one stage (it's straightforward).
iii. leftSkyline = getSkyline for the first $n/2$ stages.
iv. rightSkyline = getSkyline for the last $n/2$ stages.
v. Merge leftSkyline and rightSkyline.
```

Recurrence Relation: $T(n) \leq 2T(n/2) + O(n)$

Time Complexity:

The runtime is bounded by the recurrence $T(n) \leq 2T(n/2) + O(n)$, $a = 2, b = 2, k = 1$, which implies that $T(n) = O(n \log n)$ by Master theorem.

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

dp[i]- number of ways to reach the ith step

(b) Write a recurrence relation for the subproblems

dp[i] = dp[i - 1] + dp[i - 2]- taking a step from (i - 1), or taking a step of 2 from (i - 2)

(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.

- Initialize for base cases (mentioned below for clarity)
- For $i = 0$ to n

  If not base case: call recurrence
- Return ans (mentioned below for clarity)

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

There are two possible base cases:

- dp[0] = 0; dp[1] = 1; dp[2] = 2;
- or dp[0]=1, dp[1]=1

the final answer - dp[n]

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

Time complexity: O(n). Single loop up to n.

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.

OPT(w,k) = min packages needed to make up a capacity of exactly w, by considering only the first k packages

OR

Same except, considering packages k onwards up to n (add details below)

(b) Write a recurrence relation for the subproblems

If $w \geq w_k$

$\text{OPT}(w,k) = \min\{1 + \text{OPT}(w - w_k, k-1), \text{OPT}(w, k-1)\}$

Else

$\text{OPT}(w,k) = \text{OPT}(w, k-1)$

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

```
• Initialize for base cases (mentioned below for clarity)
• For w = 0 to W
      For k = 0 to n //can switch the inner-outer loops
         If not base case: call recurrence
• Return ans (mentioned below for clarity)
```

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

OPT(w,0) = inf for all $w < 0$ (up to -W)

OPT(0,0) = 0

OPT(w,k) = infinity for $w < 0$ and all k. (Not required if the recurrence has the second case to ensure w never takes negative values)

where the final answer can be found (e.g. opt(n), or opt(0,n), etc.)

OPT[W, n] in the first definition

OPT[W, 1] in the alternate one

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

$O(Wn)$