CSCI570 Fall2023 HW3

Vinit Pitamber Motwani (USC ID: 8187180925)

October 2023

1. **(Greedy)** Given n rods of lengths $L1, L2, ..., Ln$, 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.

   Solution :

   (a) Make a min-heap containing the lengths of all the ropes.

   (b) While there is more than one element in the min-heap

   (c) Extract the two smallest elements from the min-heap

   (d) Add their length and put the added length back into the min-heap

   (e) At the end only one element is left in min-heap which is the minimum cost of connecting all the ropes.

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.

Solution :

(a) **Counterexample for Greedy Algorithm Based on Increasing Deadlines:**

i. Consider there are three days with three performances and four artists with the following details :

A. Artist 1: $D_1 = 3$ (deadline on day 3), $A_1 = 50$ (expected turnout)

B. Artist 2: $D_2 = 2$ (deadline on day 2), $A_2 = 20$ (expected turnout)

C. Artist 3: $D_3 = 1$ (deadline on day 1), $A_3 = 70$ (expected turnout)

D. Artist 4: $D_4 = 3$ (deadline on day 3), $A_4 = 100$ (expected turnout)

ii. Apply the greedy algorithm based on increasing deadlines:

A. Schedule Artist 3 on day 1 ($A_3 = 70$).

B. Schedule Artist 2 on day 2 ($A_2 = 20$).

C. Schedule Artist 1 on day 3 ($A_1 = 50$).

D. We won't be able to schedule Artist 4 as it's deadline is on day 3 and we already scheduled performances on all three days

So the total turnouts are $70 + 20 + 50 = 140$ but it is not the optimal solution as if we would have scheduled Artist 4 on day 2 our turnouts would be $70 + 100 + 50 = 220$

(b) **Counterexample for Greedy Algorithm Based on Decreasing Turnouts:**

i. Consider the same example from point (a)

ii. Apply the greedy algorithm based on decreasing turnouts:

A. Schedule Artist 4 on day 1 ($A_4 = 100$).

B. We won't be able to schedule Artist 3 because it has a deadline of day 1.

C. Schedule Artist 1 on day 2 ($A_1 = 50$).

D. We won't be able to schedule Artist 2 because it has a deadline of day 2.

So the total turnouts are $100 + 50 = 150$ but it not the optimal solution as if would have scheduled Artist 3 on day 1 and then Artist 1 and Artist 4 on day 2 and day 3 our turnouts would be $70 + 100 + 50 = 220$

(c) **Efficient Greedy Algorithm**

i. Sort the artists in descending order of their expected audience turnout (A_i)

ii. Initialize an empty list to track schedule.

iii. For each artist in the sorted list:

A. Find the latest available day (D_i) on which the artist can perform without exceeding their deadline.

B. If there is a day available on or before D_i, schedule the artist on that day and update the list schedule.

C. Else skip the artist

iv. After the loop terminates schedule list contains the most efficient greedy schedule.

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 $T'(n) = aT'(n/4) + n^2 logn$ What is the largest value of a such that ALG' is asymptotically faster than ALG?

Solution :

Using Master's Theorem to evaluate the running time complexity of algorithms

(a) For $T(n) = 7T(n/2) + n^2$ we have $a = 7$ and $b = 2$, So after applying master's theorem Case 1 we get $T(n) = \theta(n^{log_2(7)})$

(b) For $T'(n) = aT'(n/4) + n^2 logn$, if $log_4(a) > 2$ which means $a > 16$ then from master's theorem Case 1 we will have $T'(n) = \theta(n^{log_4(a)})$

To have $T'(n)$ asymptotically faster than $T(n)$, we need $T'(n) = \mathcal{O}(T(n))$, which implies $n^{log_4(a)} = \mathcal{O}(n^{log_2(7)})$, so we get

$$\begin{aligned} \log_4(a) &\leq log_2(7) \\ 2^{log_4(a)} &\leq 7 \\ 2^{\frac{log_2(a)}{log_2(4)}} &\leq 7 \\ \mathrm{a}^{\frac{1}{2}} &\leq 7 \\ \mathrm{a} &\leq 49 \end{aligned}$$

So the largest value of a is 49 for ALG' to be asymptotically 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 $O(.)$ bound on $T(n)$.

   Solution :

   To formulate a recurrence relation for the running time T(n) of the Strange-Sort algorithm, we can analyze the time complexity of each step in the algorithm:

   (a) **Base Case :** If n is less than or equal to the algorithm returns A unchanged. This takes constant time

   (b) **Counting Step (b):** This step is done for each of the n items in A. The worst-case time complexity of this step is $\mathcal{O}(n^2)$ as we need to compare each element with every other element.

   (c) **Splitting Step (c,d,e):** In this step, the algorithm puts items with a count less than $n/2$ into list B and the others into list C.Here we are splitting the original list to size $n/2$ and $n/2$ recursively.

   (d) **Merging Step (f):** Finally, the algorithm appends the sorted list C to the sorted list B, which takes $\mathcal{O}(n)$ time because we need to iterate through both lists to merge them.

   Now, the recurrence relation for $T(n)$:

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

   Solving the recurrence relation using master's theorem we get $a = 2$,$b = 2$ and $c = log_2(2) = 1$. So Case 3 of master's theorem applies therefore $T(n) = \mathcal{O}(n^2)$

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) $T(n) = 5T(n/5) + nlogn$–$1000n$

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

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

(e) $T(n) = 3T(n/4) + n\ log\ n$

Solution :

(a) Here we have $a = 1$,$b = 2$ and $f(n) = 2^n$. So $c = log_2(1) = 0$ that is Case 3 of master's theorem applies as $f(n) = \Omega(n^{c+\varepsilon}) = \Omega(n)$ for $\varepsilon = 1$ then $T(n) = \theta(2^n)$

(b) Here we have $a = 5$,$b = 5$ and $f(n) = nlogn$–$1000n$. So $c = log_5(5) = 1$ that is Case 2 applies then $T(n) = \theta(n\ log^2 n)$

(c) Here we have $a = 2$,$b = 2$ and $f(n) = log^2 n$. So $c = log_2(2) = 1$ that is Case 1 applies then $T(n) = \theta(n)$

(d) Here $f(n) = -n^2 log(n)^2$ that means $f(n) < 0$. So masters theorem **cannot be applied** as for masters theorem $f(n)$ should be a positive function

(e) Here we have $a = 3$,$b = 4$ and $f(n) = n\ log\ n$. So $c = log_4(3)$ that is Case 3 applies then $T(n) = \theta(n\ log\ n)$

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.

Solution :

(a) First divide the sorted singly linked list into two equal halves using a slow pointer that advances one node at a time and a fast pointer that advances two nodes at a time. When the fast pointer reaches the end of the list, the slow pointer will be at the middle (or close to the middle) of the list.

(b) Compare the value at the middle node with the target value you are searching for.

(c) If the value at the middle node is equal to the target value, return the middle node as the result.

(d) If the value at the middle node is less than the target value, that means our result lies on right so we discard the first half of the list (including the middle node).

(e) If the value at the middle node is greater than the target value, that means our result lies on left so we discard the second half of the list (including the middle node).

(f) Repeat the process until you either find the target value or the list becomes empty (indicating that the target value is not in the list).

Time Complexity : Recurrence relation for this algorithm is

$T(n) = T(n/2)+\mathcal{O}(n)(as\ traversal\ to\ find\ middle\ element\ takes\ linear\ time)$. So the worst case time complexity of this algorithm is $\mathcal{O}(n)$

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.

   Solution :

   mergeSort():

   (a) If the size of the linked list is 1 then return the head

   (b) Find mid using The Tortoise and The Hare Approach

   (c) Store the next of mid in head2 i.e. the right sub-linked list.

   (d) Now Make the next midpoint null.

   (e) Recursively call mergeSort() on both left and right sub-linked list and store the new head of the left and right linked list.

   (f) Call merge() given the arguments new heads of left and right sub-linked lists and store the final head returned after merging.

   (g) Return the final head of the merged linkedlist.

   merge(head1, head2):

   (a) Take a pointer say merged to store the merged list in it and store a dummy node in it.

   (b) Take a pointer temp and assign merge to it.

   (c) If the data of head1 is less than the data of head2, then, store head1 in next of temp and move head1 to the next of head1.

   (d) Else store head2 in next of temp and move head2 to the next of head2.

   (e) Move temp to the next of temp.

   (f) Repeat steps 3, 4 and 5 until head1 is not equal to null and head2 is not equal to null.

   (g) Now add any remaining nodes of the first or the second linked list to the merged linked list.

   (h) Return the next of merged(that will ignore the dummy and return the head of the final merged linked list)

   Time Complexity : Finding the middle element takes $\mathcal{O}(n)$ time as we need to traverse the linked list then we just keep dividing the linked list in two $n/2$ linked list just like merge sort and merging two linked list also takes $\mathcal{O}(n)$ time. So the recurrence relation becomes

   $T(n) = 2T(n/2) + \mathcal{O}(n)$ which can be solved using Case 2 of master's theorem so time complexity is $T(n) = \mathcal{O}(n\ log\ n)$

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.

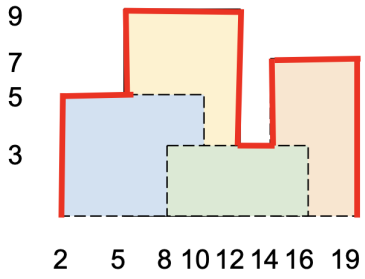

Figure 1: Example of festival stage setup.

   (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.

   (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 $O(nlogn)$ steps.

Solution :

   (a) Given skyline information of n stages and m stages in the form $(x1, h1, x2, h2, ..., xn)$ and $(x'1, h'1, x'2, h'2, ..., x'm)$ respectively we can combine them using following steps:

   Merge the left list and the right list by iterating through them and keeping track of current left height (initially 0) and the current right height (initially 0). After that we find the lowest next x-coordinate in either list, we assume it is in left list. We remove the first two

elements x and h and set the current left height to h and output x and maximum of current left and right heights.

(b) If there is one building output it Otherwise split the buildings into two equal groups and recursively compute skylines and output the result of merging them using point (a).

Algorithm:

getSkyline(n):

i. If n=0 : return empty list
ii. If n=1 : return the skyLine for one building
iii. leftSkyLine = getSkyline for first n/2 buildings
iv. rightSkyLine = getSkyline for last n/2 buildings
v. Merge leftSkyLine and rightSkyLine

Recurrence relation for this algorithm is : $T(n) \leq 2T(n/2) + \mathcal{O}(n)$
Time Complexity : From the recurrence relation we get $a = 2$, $b = 2$ and $c = log_2(2) = 1$. So Case 2 of master's theorem applies than $T(n) = \mathcal{O}(n\ log\ n)$

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.
   (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.

   Solution :

   (a) $OPT[K]$ be the number of distinct combinations of tickets to reach the fundraising goal of K
   (b) Recurrence relation is $OPT[k] = OPT[k-1] + OPT[k-2]$ where $OPT[k-1]$ is if we buy 1-dollar ticket and $OPT[k-2]$ is if we buy 2-dollar ticket

(c) **Pseudocode**

```
int[] FundGoal(int n)
{
    int Opt [n+1];
    Opt[0] = Opt[1] = 1;
    for(int k = 2; k < n; k++)
        Opt[k]=Opt[k-1]+Opt[k-2];

    return Opt[n];
}
```

(d) Base Cases

i. $OPT[0] = 1$ (There is one way to reach 0 dollars no tickets sold)

ii. $OPT[1] = 1$ (There is one way to reach 1 dollar by selling one 1-dollar ticket)

(e) The runtime complexity of this solution is $\mathcal{O}(n)$ because it uses a single loop to compute the number of combinations for each amount from 2 to $'n'$. The algorithm iterates through all possible fundraising targets once, making it a linear-time solution.

10. **(Dynamic Programming)** Assume a truck with capacity W is loading. There are n packages with different weights, i.e. $(w_1, w_2, ...w_n)$, 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.

Solution :

(a) $OPT[K, X]$ be the minimum count of packages achievable using a truck with capacity of X with K packages

(b) If a package is chosen $OPT[K, X] = 1 + OPT[K-1, X-w_k]$
If a package is not chosen $OPT[K, X] = OPT[K-1, X]$
So the recurrence relation is
$OPT[K, X] = MIN(1 + OPT[K-1, X-w_k], OPT[K-1, X])$

(c) **Pseudocode**

```
int countPackages(int W, int w[], int n) {
    int Opt[n+1][W+1];
    for (k = 0; k <= n; k++) {
      for (x = 0; x <= W; x++) {
        if (k==0 || x==0) Opt[k][x] = 0;
        if (w[k] > x) Opt[k][x] = Opt[k-1][x];
        else
        Opt[k][x] = min( 1 + Opt[k-1][x - w[k]], Opt[k-1][x] );
      }
    }
    return Opt[n][W];
}
```

(d) Base Cases

i. $OPT[0, X] = 0$ (No packages to load)

ii. $OPT[K, 0] = 0$ (No capacity is left in truck)

iii. If $w_k > X$ than $OPT[K, X] = OPT[K - 1, X]$

(e) As there are two loop running till n and W respectively worst case run time complexity is $\mathcal{O}(n.W)$

But actual run time complexity is $\mathcal{O}(n.2^{input\ size\ of\ W})$ as in input size W is $(log\ W)$ in bits