CPSC 320 2024W1: Assignment 3

This assignment is due **Friday, October 25 at 7 PM**. Late submissions will not be accepted. All the submission and formatting rules for Assignment 1 apply to this assignment as well.

1 List of names of group members (as listed on Canvas)

Provide the list here. This is worth 1 mark. Include student numbers as a secondary failsafe if you wish.

- Hansel Poe - 82673292

2 Statement on collaboration and use of resources

To develop good practices in doing homeworks, citing resources and acknowledging input from others, please complete the following. This question is worth 2 marks.

1. All group members have read and followed the guidelines for groupwork on assignments given in the Syllabus).

   [✓ Yes] No

2. We used the following resources (list books, online sources, etc. that you consulted):

3. One or more of us consulted with course staff during office hours.

   Yes [✓ No]

4. One or more of us collaborated with other CPSC 320 students; none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   Yes [✓ No]

   If yes, please list their name(s) here:

5. One or more of us collaborated with or consulted others outside of CPSC 320; none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   Yes [✓ No]

   If yes, please list their name(s) here:

3 More Heat!

Recall the worked example we discussed in class at the beginning of the greedy algorithms unit:

You are an air conditioner mechanic, and your normally temperate coastal city is expecting record-breaking heat this summer. You have agreed to install or repair an air conditioning system for n clients, and each job will take one day to complete. You must decide the order in which to complete the jobs.

None of your clients want to be hot and uncomfortable in the coming heat wave, so they would all prefer to have their project completed earlier rather than later. In particular, after n days the uncomfortable heat is expected to be over, so nobody is going to be especially happy with being the last client on your list. More precisely, if you complete the job for client i at the end of day j, then client i's satisfaction is $s_i = n - j$.

Some clients are more important to you than others (perhaps they are paying you more, are more likely to recommend you to others, or are more likely to be repeat customers in the future). You assign each client i a weight w_i. You want to schedule your clients across the n days such that you maximize the weighed sum of satisfaction, that is, $\sum_{1\le i\le n} w_i s_i$.

We saw in the worked example that, in this scenario, an optimal greedy algorithm is to sort the jobs in order of w_i.

1. (3 points) Instead of assuming that each job takes exactly 1 day, suppose that it takes t_i days to complete the job i (here t_i must be greater than 0, but is not necessarily an integer). Assume that $\sum_{1\le i\le n} t_i = n$. Again, you want to maximize the weighted sum of satisfaction $\sum_{1\le i\le n} w_i s_i$, where $s_i = n - c_i$. and c_i is the time to completion of the project. For example, if project i is completed first, then $c_i = t_i$. If a project i' is completed second (after project i), then $c_{i'} = t_i + t_{i'}$.

   Give and briefly explain a counterexample to show that the greedy strategy of completing projects in decreasing order of w_i is not optimal for this variant of the problem.

   Consider the case:
   $n = 3$ and we have jobs:

   - $w_1 = 7$, $t_1 = 2.25$
   - $w_2 = 5$, $t_2 = 0.5$
   - $w_3 = 3$, $t_3 = 0.25$

   where job i has the weight w_i and takes t_i days to complete :

   Decreasing order of weight, w_1, w_2, w_3, would give us the total weighted satisfaction of $7 * 0.75 + 5 \times 0.25 + 3 \times 0 = 6.5$.
   but the ordering, w_3, w_2, w_1 gives the weighted satisfaction of $2.75 \times 3 + 2.25 \times 5 + 7 \times 0 = 19.5 > 6.5$.
   so our greedy solution is not optimal.

2. (2 points) Describe a greedy strategy that is optimal on the problem variant described in part 1. A one-sentence description is sufficient (e.g., describe the order that you would use to sort the projects).

   We want to order our jobs such that the jobs that have high weights and low completion time come first. We can take the ratio $\frac{w_i}{t_i}$ and order the jobs in decreasing order of this ratio.

3. (5 points) Prove that your greedy strategy from part 2 is optimal.

Let O be the optimal solution with ratio ordering $r_1, r_2, \ldots, r_n$. Let i, j where $i < j$ be consecutive jobs in O that defines an inversion relative to G, that is, in G, j comes first than i. Note that if i and j are not consecutive , we can find consecutive elements between i and j and consider that instead (this is proven in asymptotic analysis worksheet).

We must have that $\frac{w_j}{t_j} \geq \frac{w_i}{t_i}$ which implies $w_j t_i - w_i t_j \geq 0$

Now, we will swap i and j in O. The sum of weighted satisfaction of all jobs before i and j and after them remains unaffected. Let C be the total time of completion of all jobs before i and j. We will analyze the weighted satisfaction of i and j before and after the swap:

Before:

- $w_i(n - c_i) = w_i n - w_i(C + t_i)$
- $w_j(n - c_j) = w_j n - w_i(C + t_i + t_j)$

After:

- $w_i(n - c_i) = w_i n - w_i(C + t_j + t_i)$
- $w_j(n - c_j) = w_j n - w_j(C + t_j)$

If we take the difference between the sum of weighted satisfaction between O after and before the swap, we will get $w_j t_i - w_i t_j \geq 0$ This implies the swap produces an ordering that is at least as good as before the swap. We can continue swapping inversions until O is transformed into G, and thus our G solution is optimal. $Q.E.D$

4 Weekly Meeting Logistics

You're the manager of an animal shelter, which is run by a few full-time staff members and a group of n volunteers. Each of the volunteers is scheduled to work one shift during the week. There are different jobs associated with these shifts (such as caring for the animals, interacting with visitors to the shelter, handling administrative tasks, etc.), but each shift is a single contiguous interval of time. Shifts cannot span more than one week (e.g., we cannot have a shift from 10 PM Saturday to 6 AM Sunday). There can be multiple shifts going on at once.

You'd like to arrange a weekly meeting with your staff and volunteers, but you have too many volunteers to be able to find a meeting time that works for everyone. Instead, you'd like to identify a suitable subset of volunteers to instead attend the weekly meeting. You'd like for every volunteer to have a shift that overlaps (at least partially) with a volunteer who is attending the meeting. Your thinking is that you can't personally meet with every single volunteer, but you would like to at least meet with people who have been working with every volunteer (and may be able to let you know if a volunteer is disgruntled or having any difficulties with their performance, etc.). Because your volunteers are busy people, you want to accomplish this with the fewest possible volunteers.

Your volunteer shifts are given as an input list V, and each volunteer shift v_i is defined by a start and finish time (s_i, f_i). Your goal is to choose a subset of volunteer shifts of minimum size, such that every shift in V overlaps with at least one of the chosen shifts. A shift $v_i = (1, 4)$ overlaps with the shift $v_j = (3, 5)$ but not with the shift $v_k = (4, 6)$.

1. (2 points) Your co-worker proposes the following greedy algorithm to select volunteers for your meeting:

   > Select the shift v that overlaps with the most other shifts, discard all shifts that overlap with v, and recurse on the remaining shifts.

   Give and briefly explain a counterexample to prove that this greedy strategy is not optimal.

   Consider the case with shifts: $\{(1,3),(2,3),(3,4),(2,5),(4,12),(6,7),(7,8)\}$ The optimal solution is $\{(2,5),(4,12)\}$ but our greedy algorithm outputs $\{(2,5),(6,7),(7,8)\}$

2. (3 points) Give a greedy algorithm to solve this problem. Give an unambiguous specification of your algorithm using a **brief, plain English description**. Do not write pseudocode or worry about implementation details yet. (You may do this in part 3 if you feel that it's necessary to achieve a particular runtime.)

   Choose a shift, find the shift that overlaps with this shift and has most overlaps with other shifts, put in our solution set. Choose a shift that has not been overlapped. Find a shift that overlaps with this shift that has most overlaps with shifts not yet overlapped, put that in our solution set. recurse.

3. (4 points) Briefly justify a good asymptotic bound on the runtime of your algorithm. If you prefer to present pseudo-code to help track the runtime incurred, you may do so.

   We could use a disjoint set to track which shifts have been overlapped, for a given input set n at worst case,we may have $\in O(\log n)$ joins (how many times we pick a new shift). Now looking for overlapped shifts will cost $\in O(n^2)$ as we are searching through the whole array input and for each shift if they overlap, we need to iterate the array again to find number of overlaps. Overall the algorithm will incur a cost of $\in O(n^2 \log n)$.

5 Fun with Recurrences

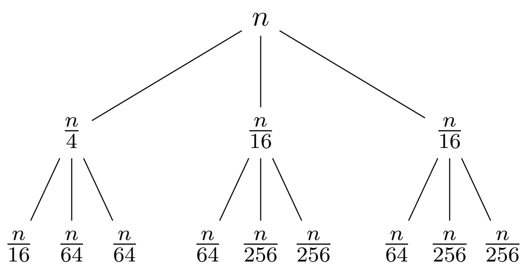
Give an asymptotic solution (which should be a Θ-bound) to each of the recurrences below. You may use whatever solution method you wish (drawing out the tree, unrolling the recurrence, proof by induction, Master Theorem, etc.), but make sure you fully justify your answer.

1. (3 points) $T(n) = 6T\left(\frac{n}{2}\right) + 2^n$ for $n > 1$, $T(1) = 1$.

   $T(n) \in \theta(2^n)$
   We can use case 3 the of master theorem. Let $a = 6$ and $b = 2$. Notice that $2^n \in \omega(n^c)$ for $c > \log_2 6$ and we can let $k = 1/2$ such that $af(\frac{n}{b}) \leq kf(n)$ for all $n \geq 2log_2 12$.

2. (3 points) $T(n) = T\left(\frac{n}{4}\right) + 2T\left(\frac{n}{16}\right) + \sqrt{n}$ for $n > 16$, $T(n) = 1$ for $n \leq 16$.

   Drawing a few levels of the recursion tree, we get:



   The amount of work done per level is $\sqrt{n}$. The shortest leaf of the tree is $\frac{\log_4 n}{2} - 1$, so our recurrence is $\in \Omega(\sqrt{n}\log n)$. And the deepest leaf is $\log_4 n - 2$, giving us an upper bound of $\in O(\sqrt{n}\log n)$. Our recurrence is therefore $\in \Theta(\sqrt{n}\log n)$.

3. (4 points) $T(n) = mT(n-1) + 1$ for $n > 1$, $T(1) = 1$. Assume that m is an integer value greater than or equal to 2.

   For each level l, the tree will have m^l nodes and each node does a constant amount of work. So each level does m^l amount of work. Now, going down one level we reduce the input size by one and the base case is $n = 1$ so we will have $n - 1$ levels. Total work done is $\sum_{i=0}^{n-1} m^i = \frac{1-m^n}{1-m} \in \Theta(m^n)$.

4. (4 points) $T(n) = mT(\frac{n}{4}) + n^2$ for $n \geq 4$; $T(n) = 1$ for $n < 4$. Assume that m is greater than or equal to 1 (but not necessarily an integer).

   Let $a = m, b = 4$ and $k = 2$. Using the Master Theorem Colorally, we have 3 possible cases:

   - $m > 16$ in which case, $T(n) \in \Theta(n^{log_4 m})$
   - $m = 16$ in which case, $T(n) \in \Theta(n^2 \log n)$
   - $m < 16$ in which case, $T(n) \in \Theta(n^2)$

6 D+C = Profit

You own an online sales company called DCAuctions.com that sells goods both on auction and on a fixed-price basis. You want to use historical auction data to investigate your fixed price choices.

Over n minutes, you have a good's price in each minute of the auction. You want to find the largest *price-over-time stretch* in your data. That is, given an array A of n price points, you want to find the largest possible value of

$$f(i,d) = d \cdot \min(A[i], A[i+1], \ldots, A[i+d-1]),$$

where i is the index of the left end of a stretch of minutes, d is the duration (number of minutes) of the stretch, and the function f computes the duration times the minimum price over that period. (Prices are positive, $d \geq 0$, and for all values of i, $f(i,0) = 0$ and $f(i,1) = A[i]$.)

For example, the best stretch is underlined in the following price array: $[8, 2, \underline{9, 5, 6, 5}, 3, 1]$. Using 1-based indexing, the value for this optimal stretch starting at index 3 and running for 4 minutes is $f(3,4) = 4 \cdot \min(9,5,6,5) = 4 \cdot 5 = 20$.

1. (3 points) Describe a polynomial-time brute force algorithm to solve this problem.

   We could generate all possible Intervals and calculate their $f(i,d)$, then take the interval which produces the maximum value.

2. (3 points) Write a function `MidHelper` (consider the name to be a hint for the next question!) that, given an array A, an index $1 \leq i \leq n-1$ and length $k \leq n$, finds the best stretch of length less than or equal to k that includes both $A[i]$ and $A[i+1]$. (Another hint for the next question: you will want to do this in $O(k)$ time.)

```
procedure MidHelper($A, i, k$)
    $l = 2$;
    $stretch = 2$;
    $min = A[i]$;
    $val = 0$;
    $maxVal = 0$;
    for $j$ from $i+1$ to $i+k-1$ do
        if $A[j] < min$ then
            $min = A[j]$
        end if
        $val = l \times min$;
        if $maxVal < val$ then
            $maxVal = val$;
            $stretch = l$;
        end if
        $l = l + 1$;
    end for
return stretch;
end procedure
```

3. (5 points) Write a divide and conquer algorithm algorithm to efficiently find the best price stretch. Your algorithm should use the `MidHelper` function described in part 2.

```
procedure FindStretch(A)
    return FindStretchHelper(A,1,A.length());
end procedure

procedure FindStretchHelper(()A, i, j)
```

```
        **if** $i == j$ **then return** $(A[i], 1)$;
        **end if**
        $mid = (i + j)/2$;
        $(leftI, leftL) = FindStretchHelper(A, i, mid)$;
        $(rightI, rightL) = FindStretchHelper(A, mid + 1, j)$;
        $midL = MidHelper(A, left_i, j)$
         **return** first and last index of the following $max(f(leftI, leftL), f(rightI, rightL), f(mid, midL)$;
    **end procedure**
```

4. (2 points) Give and briefly justify a good asymptotic bound on the runtime of your algorithm.

Our Function divides the input into half, and each recurrence performs $\in O(n)$ work, we have $\in Olog(n)$ levels in our recurrence tree so or algorithm runs for $\in Onlog(n)$.