CPSC 320 2024W1: Assignment 3 Solutions

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, $\sum_{1 \leq i \leq n} w_i s_i$.

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 $\sum_{1 \leq i \leq n} t_i = n$. Again, you want to maximize the weighted sum of satisfaction $\sum_{1 \leq i \leq n} w_i s_i$, 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 example

i	w_i	t_i
1	5	2
2	3	1
3	2	0.5
4	1	0.5

   The greedy strategy would complete the projects in the order $[1, 2, 3, 4]$, with a total weighted satisfaction of

   $$5 * (4 - 2) + 3 * (4 - 3) + 2 * (4 - 3.5) + 1 * (4 - 4) = 13.$$

   A better strategy would be the ordering $[3, 2, 1, 4]$, which gives a total weighted satisfaction of

   $$2 * (4 - 0.5) + 3 * (4 - 1.5) + 5 * (4 - 3.5) + 1 * (4 - 4) = 17.$$

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).

   An optimal greedy strategy is to perform the jobs in decreasing order of w_i/t_i.

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

As in 4.1, let $\mathcal{O}$ denote an optimal schedule that completes jobs in the order $o_1, o_2, \ldots o_n$ (with respective weights $w_1, \ldots, w_n$ and completion times $t_1, \ldots, t_n$). And let p, q, where $p < q$ denote the indices of two jobs that appear in a different order in $\mathcal{O}$ than in the greedy ordering $\mathcal{G}$. For simplicity here in reasoning about the s_i values, we'll assume that p and q constitute a *neighbouring* inversion – that is, $q = p + 1$, and o_p and o_q appear in the opposite order in the greedy schedule. It's not hard to show that as long as a schedule has an inversion, it has a neighbouring inversion (in fact, we essentially **did** prove this in the asymptotic analysis worksheet). We can easily show this by contradiction: if there are no neighbouring inversions in $\mathcal{O}$ – i.e., no pairs of consecutive elements that appear in a different order in $\mathcal{O}$ than in $\mathcal{G}$ – then it must mean there are no inversions.

The weighted sum of satisfaction for $\mathcal{O}$ is

$$S(\mathcal{O}) = w_1 s_1 + w_2 s_2 + \ldots + w_p s_p + w_q s_q + \ldots + w_n s_n,$$

where $s_j = n - \sum_{i=1}^{j} t_i$. Let $\mathcal{O}'$ denote the schedule obtained by swapping jobs o_p and o_q in $\mathcal{O}$. The weighted satisfaction for $\mathcal{O}'$ is

$$S(\mathcal{O}') = w_1 s_1 + w_2 s_2 + \ldots + w_q s'_q + w_p s'_p + \ldots + w_n s_n,$$

where

$$s'_q = n - (t_1 + t_2 + \ldots + t_{p-1} + t_q) = s_p + t_p - t_q$$

and

$$s'_p = n - (t_1 + t_2 + \ldots + t_{p-1} + t_q + t_p) = s_q.$$

Therefore,

$$\begin{aligned} S(\mathcal{O}') - S(\mathcal{O}) &= w_q s'_q + w_p s'_p - w_p s_p - w_q s_q \\ &= w_q s_p + w_q t_p - w_q t_q + w_p s_q - w_p s_p - w_q s_q \\ &= w_q(s_p - s_q + t_p - t_q) + w_p(s_q - s_p) \\ &= w_q t_p - w_p t_q, \end{aligned}$$

where we obtained the last equality by using $s_q = s_p - t_q$.

Because the greedy algorithm orders jobs in descending order by w_i/t_i, and jobs p and q appear in the opposite order in $\mathcal{G}$, we must have

$$\frac{w_q}{t_q} \geq \frac{w_p}{t_p},$$

which implies (by multiplying both sides by $t_p t_q$) that

$$w_q t_p \geq w_p t_q.$$

Thus, $S(\mathcal{O}') - S(\mathcal{O}) \geq 0$, and the weighted sum of satisfaction has not decreased by swapping elements p and q.

As mentioned earlier, as long as $\mathcal{O}$ has an inversion relative to the order of $\mathcal{G}$, it has a neighbouring inversion (i.e., a pair of two consecutive elements that appear in opposite order from $\mathcal{G}$). Thus, we can repeatedly swap neighbouring inversions until we have converted $\mathcal{O}$ to $\mathcal{G}$ without decreasing the weighted sum of satisfaction. From this we conclude that our greedy strategy yields an optimal schedule.

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.

   Suppose our shifts are:

```
-------- v1
    -------- v2
         --- v3
         ------- v4
             --- v5
             ------- v6
                   -------- v7
```

   The optimal solution is to select v_2 and v_6. In terms of overlap: v_1 overlaps with 1 shift, v_2 with 3 shifts, v_3 with 2 shifts, v_4 with 4 shifts, v_5 with 2 shifts, v_6 with 3 shifts, and v_7 with 1 shift. Thus, the greedy strategy will select v_1, v_4 and v_7.

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.)

   For a simple statement of the algorithm without worrying about efficiency. We'll say that an shift is "covered" if it overlaps with some shift currently in our solution:

   > Mark all shifts as uncovered. While not all shifts are covered, consider the uncovered shift v' with the earliest end time. Of the shifts that intersect with v', add to the solution the shift v with the latest end time. Mark as covered all shifts that intersect with v.

Additional Remarks: Above is the complete solution. However, we add a couple of important details to address common mistakes here.

Common mistake 1: One is that we do need to consider v' with the earliest end time, and **not** the earliest start time. For an example to show why this is the case, consider the example:

```
-------------------------
   --  --  --  --  --
                       ----------
```

The optimal solution is to choose only the shift in the first row. Let's see what an algorithm that looks at the earliest-starting shift as v' would do. The shift in the top row has the earliest start time, so it would become v'. From there we would choose the shift in the third row as v to be the first shift in our solution, but this is an incorrect choice, since (depending on the exact statement of your algorithm) we would then either inappropriately terminate without having all shifts covered, or we would select v as our first shift, notice that some shifts are not covered, and then select more shifts. The correct algorithm takes the first shift in the second row as v', selects the shift in the first row as v, and then terminates because all shifts are covered.

Common Mistake 2: Another common mistake is to assume (perhaps implicitly) that covered shifts won't be part of the solution – e.g., a solution might select a shift v, discard all shifts that overlap with v, and recurse on the remaining shifts. But, sometimes a previously covered shift **does** need to be part of the solution – as in, e.g., $V = [(1,2),(1,6),(2,3),(4,12),(6,7),(8,9)]$, where the first step of the algorithm will select $v = (1,6)$, but then the correct choice for the next interval is the (already covered) shift $(4,12)$.

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 introduce pseudocode below to help discuss the runtimes. We keep track of uncovered events in an array called `U`, and the events that could still be part of the solution (which includes all uncovered events, in addition to all covered events that end later than the previously selected v) in an array called `S`.

```
Algorithm FindEvents(V):
   Sort V by increasing start time
   U ← V   % uncovered events
   S ← V   % events that could be part of the solution
   Soln ← []

   While U is not empty:
      % assume 1-based indexing
      Let EarlyU ← the set of events in U with s < U[1].f
      Let v' ← the event in EarlyU with the earliest finish time
      Let SBeforeVp ← the set of events in S with s < v'.f

      Let v ← the event in SBeforeVp with the latest finish time
      Add v to Soln
      Let UCoveredByV ← the set of events in U with s < v.f
      Remove from U all events in UCoveredByV
      SBeforeV ← the set of events in S with s < v.f
```

```
        Let NoLongerInS ← the set of events in SBeforeV with f ≤ v.f
        Remove from S all events in NoLongerInS

    return Soln
```

Sorting V by increasing start time takes $O(n \log n)$ time. The while loop analysis is trickier, but let's consider how many times a particular event can be considered in the iterations through U and S. Let's start with U. The elements of U we iterate through at each step of the while loop are those in `EarlyU` and those in `UCoveredByV`. However, `EarlyU` is clearly a subset of `UCoveredByV` (because v could always be $U[1]$, and if it isn't it must mean that v ends later), and everything in `UCoveredByV` is removed at the end of the loop. Thus, no element of U will be accessed in more than two loop iterations (we will need to access the first element *after* `UCoveredByV` to make sure we have all the elements in that set), and all work within the loop takes at most linear time in the number of elements accessed (assuming that each array deletion takes constant time), so the iterations through U take linear time total.

With S, the case is less straightforward because not all elements of S that we access in forming `SBeforeVp` or `SBeforeV` will be deleted at the end of the loop. The set `SBeforeVp` is a subset of `SBeforeV`, because the end time of v is later than or equal to that of v' (because v could be equal to v', so if it isn't it must mean v ends later). There may be some elements of `SBeforeV` (those with $f > v.f$) that are not deleted at the end of the loop iteration. However, these will need to be deleted by the end of the **next** loop iteration (unless all events are covered, in which case the algorithm will simply terminate): a non-deleted element of `SBeforeV` needs to start before the next v' (because the next v' has to start after the current $v.f$, and in order for an element to be in `SBeforeV` it must start before $v.f$). This means that, at the next iteration, a non-deleted element of `SBeforeV` either becomes the next selected v, or it ends earlier than the next selected v – in either case, at the next iteration it will be in the set `NoLongerInS`, and will therefore be deleted from S. Thus, no element of S will be accessed in more than three loop iterations (again, at each step we will need to access the first element after `SBeforeV`), and all work within the loop takes at most linear time in the number of elements accessed, meaning the iterations through S will also take linear time total.

Putting all this together, the sorting takes $O(n \log n)$ and the while loop takes $O(n)$, so the total running time of the algorithm is $O(n \log n)$. Note that you can earn full credit here without your explanation being quite so thorough as ours (additionally, if you came up with a different algorithm, the runtime analysis may have been quite straightforward).

A naïve implementation of the "simple statement of the algorithm" we gave in question 1 would run in something like $\Theta(n^2)$ time. Without pre-sorting by start time, we would need to iterate through the entire list to find all events that intersect with a given one, which will lead to an overall $\Theta(n^2)$ runtime. Additionally, if you didn't come up with some way to remove earlier events from consideration in choosing the next event, you would also get a $\Theta(n^2)$ runtime (because you would potentially need to search all earlier-starting events to correctly find the one with the latest end time). If your answer to 4.1 uses any kind of repeated "intersects with" calculation **without** first appropriately sorting the array **and** removing earlier-sorted elements from consideration at each step, we will likely consider it to be a $\Theta(n^2)$ (and therefore not optimally efficient) algorithm.

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 $n > 1$, $T(1) = 1$.

   We can solve this recurrence using the Master Theorem. Since $2^n \in \Omega(n^c)$ for any constant c, we are in case 3, provided we can satisfy the regularity condition. To check this, we need to verify that there exists a constant k such that

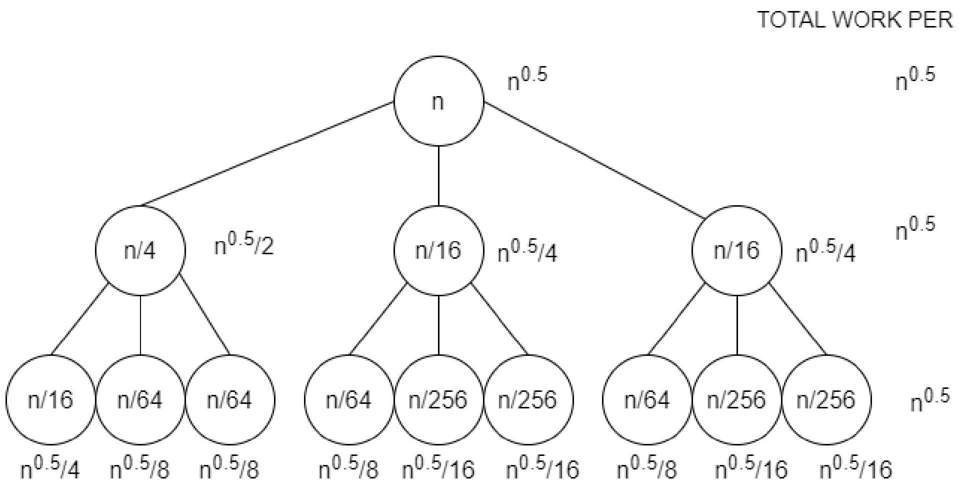
   $$af(n/b) = 6 \cdot 2^{n/2} \leq k \cdot 2^n,$$

   for all sufficiently large n. Since $2^{n/2} = \sqrt{2^n} \in o(2^n)$, we know that we can satisfy this for any positive k.

   Therefore, case 3 of the Master Theorem applies here, and thus $T(n) \in \Theta(2^n)$.

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 $n \leq 16$.

   We've drawn the tree below:



   The work per level is $\sqrt{n}$. Similarly to how we analyzed the QuickSort tree in class, we can argue that the shallowest leaf and deepest leaf of the tree are both at depth $\Theta(\log n)$. Therefore, we can give a bound of $\Theta(\sqrt{n} \log n)$.

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.

   Analyzing the tree, unrolling the recurrence, and proof by induction would all work for this question (Master Theorem would not, because the subproblem is not of the form n/b).

   Since it's tough to draw a tree with arbitrarily many branches, we'll unroll the recurrence. We can write:

   $$\begin{aligned} T(n) &= 1 + mT(n-1) \\ &= 1 + m + m^2T(n-2) \\ &= 1 + m + m^2 + m^3T(n-3) \\ &\dots \\ &= 1 + m + m^2 + \ldots + m^{n-1}. \end{aligned}$$

This is a geometric series with $a = 1$ and $r = m$, we can write the sum as

$$\frac{m^n - 1}{m - 1} \in \Theta(m^{n-1}).$$

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).

   For a given value of m, we can solve this recurrence using the Master Theorem. The critical value, c_{crit} (in the language of the Wikipedia article on the Master Theorem) is $\log_4 m$, and the power on n in $f(n)$ is 2. The solution to the recurrence has three separate cases, depending on the value of m.

   - We're in case 1 of the Master Theorem if $2 < \log_4 m$ – i.e., if $m > 16$. When $m > 16$, then $T(n) \in \Theta(n^{\log_4 m})$.
   - We're in case 2 of the Master Theorem if $2 = \log_4 m$ – i.e., if $m = 16$. When $m = 16$, then $T(n) \in \Theta(n^2 \log n)$.
   - We're in case 3 of the Master Theorem if $2 > \log_4 m$ – i.e., if $1 \leq m < 16$. Since our $f(n)$ term here is n^2 we can use the Master Theorem Corollary and skip the regularity check. Therefore, when $m < 16$, $T(n) \in \Theta(n^2)$.

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.

   Our brute force algorithm is to check the value of $f(i, d)$ for all admissible values of i and d (that is, for i from 1 to n and d from 1 to $(n - i + 1)$) and return the values of i and d that give us the maximum value. This will take $O(n^2)$ (and therefore polynomial) time.

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.)

   Let's consider the problem of finding the best stretch containing $A[i]$ and $A[i+1]$ with length **exactly equal** to k. We know the the length of this is k, so the best choice is the one that maximizes the minimum value in the interval (since the value of the stretch is k times the minimum value in the stretch).

   This means we can start with the length-2 stretch $A[i], A[i+1]$ and get the best length-3 stretch by extending to either $A[i-1]$ or $A[i+2]$ – whichever value is larger. Similarly, we can get the best length-4 stretch by taking our best length-3 stretch and extending it to its larger neighbour. To find the best stretch with length up to k, we just do this "extending" strategy to find the best stretch of length 2, 3, …, k, and return the one with the maximum value.

   In pseudocode:

```
function MidHelper(A, i, k):

    currLength = 2 // start with length-2 stretch A[i], A[i+1]
    currStart = i // first index of current stretch
    currEnd = i+1 // last index of current stretch

    // keep track of the minimum value in stretch
    // (this is to help keep the algorithm linear time)
    minInStretch = min(A[i], A[i+1])

    // keep track of best stretch (of any length) found so far.
    // store as {i, d, value} tuple
    bestSoFar = {i, 2, 2*minInStretch}

    define A[0] = -inf and A[n+1] = -inf. This will stop our loop below from
    extending the stretch out of bounds (without our needing to
    write extra if statements).

    // loop to increase size of interval. We expand the current interval to include the
    // max-valued neighbour.
    while currLength <= k:
        if A[currStart-1] > A[currEnd+1]:
           // element to our immediate left is bigger, so extend left
           currStart = currStart-1
           minInStretch = min(A[currStart], minInStretch)
        else:
            // element to our immediate right is bigger, so extend right
            currEnd = currEnd+1
            minInStretch = min(A[currEnd], minInStretch)

        currLength = currLength + 1
        currValue = currLength*minInStretch

        if currValue > value of bestSoFar:
            bestSoFar = {currStart, currLength, currValue}

    return bestSoFar
```

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.

For our divide and conquer algorithm, we'll split our array into two equal halves. The best stretch is either entirely in the left half, entirely in the right half, or spans both halves. We'll use recursion to find the best stretch in the left and right halves, and `MidHelper` to find the best stretch that goes across both halves. And we'll return whichever one of these has the maximum value. As in `MidHelper` we'll return our answer as a tuple `{ i, d, value }`.

Pseudocode:

```
function BestStretch(A):
    n = length(A)
    if n = 1:
        return {1, 1, A[1]}

    mid = floor(n/2)

    // i, d, value on left
    {iL, dL, vL} = BestStretch(A[1:mid])
    // i, d, value on right
    {iR, dR, vR} = BestStretch(A[mid+1:n])
    // i, d, value across middle
    {iM, dM, vM} = MidHelper(A, mid, n)

    return the tuple with highest value (third element).
```

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

Our algorithm recurses on two subproblems of size $n/2$, and our combine step takes $O(n)$ time ($O(n)$ for the call to `MidHelper` and constant time for all other steps). So our recurrence is

$$T(n) = 2T(n/2) + cn,$$

which gives a runtime of $O(n \log n)$.