CPSC 320 2017W1: Midterm 2 Sample Solution

January 26, 2018

1 O'd to a Pair of Runtimes [4 marks]

You are working on algorithms that operate on two strings. You are guaranteed that the first string, of length s, is shorter than the second, of length t. Each string has at least 2 letters. The pairs below represent runtimes for different algorithms. For each pair, fill in the circle next to the best choice of:

LEFT: the left function is big-O of the right, i.e., left $\in$ O(right)

RIGHT: the right function is big-O of the left, i.e., right $\in$ O(left)

SAME: the two functions are Θ of each other, i.e., left $\in$ Θ(right)

INCOMPARABLE: none of the previous relationships holds for all allowed values of s and t.

Do not choose **LEFT** or **RIGHT** if **SAME** is true. The first one is filled in for you.

Left Function	Right Function	Answer
s	s^2	**LEFT**
s^{t+s}	s^t	○ LEFT [✓ RIGHT] ○ SAME ○ INCOMPARABLE
$(s+t)^2$	t^2	○ LEFT ○ RIGHT [✓ SAME] ○ INCOMPARABLE
2^s	$3t^3+s^2$	○ LEFT ○ RIGHT ○ SAME [✓ INCOMPARABLE]
$\frac{s^2+t}{t}$	$t \lg t$	[✓ LEFT] ○ RIGHT ○ SAME ○ INCOMPARABLE

SOLUTION EXPLANATIONS:

1. Let's use the trick of taking the ratio of the two functions: $\frac{s^{t+s}}{s^t} = s^s$. So, if we hold s constant, then the two functions are Θ of each other. Otherwise, the left one dominates, and the **right** is smaller/"faster".

2. We'll just multiply the left one out: $(s+t)^2 = s^2+2st+t^2$. Clearly, the right is $\leq$ to the left. So, the answer is one of "right" or "same". Note that we know $s < t$. So, $(s+t)^2 < (t+t)^2 = (2t)^2 = 4t^2$. But, $4t^2 \in O(t^2)$. So, they're Θ of each other, and the answer is **same**.

3. This one proved tricky. After all, exponentials grow fast! But, what if $t = 2^s$, for example? In that case, the right hand side becomes $3(2^s)^3 + s^2 = 3(2^{3s}) + s^2 = 3(8^s) + s^2$. 8^s dominates 2^s. In other words, depending on the relationship between s and t, the left or right could dominate (or they could be Θ of each other). Thus, the answer is **incomparable**.

4. It helps to rewrite the left side: $\frac{s^2+t}{t} = \frac{s^2}{t} + 1$. That, in turn, is definitely smaller than $s+1$. On the right side, $t \lg t > s \lg s$. But, $s+1 \in o(s \lg s)$. So, the right side definitely dominates, and the **left** side is smaller/"faster".

2 Fine Dining [16 marks]

A common problem when friends get together is, "where shall we go to eat?" Suppose a group of friends have decided to vote on a restaurant for the evening. They will only choose a restaurant if more than half of them agree on the choice.

We parameterize the *Fine Dining (FD)* problem by the size of the group n, and the (unordered) list of suggestions they produce, S. Each element of S is a vote—a positive integer code for the restauarant one person chose (e.g., 1: Mr. Red, 2: Darcy's, 3: Maenam, 4: Tojo's, ...). $FD(n, S)$ returns a restaurant code, r, if there are at least $\lfloor \frac{n}{2} \rfloor + 1$ elements with value r in S, and "None" otherwise.

Select and fill in the best completions to the design discussion below of one possible algorithm for solving the problem.

1. If there is a restaurant code favored by the majority, it will be the MIN [✓ MEDIAN] MAX valued suggestion in S. We can find that code, r, in [✓ $\Theta(n)$] $\Theta(n \log n)$ $\Theta(n^2)$ time by letting r be the result of calling the QuickSort [✓ DeterministicSelect] LongestCommonSubsequence algorithm with input(s) $\boxed{\text{S, } \lfloor \frac{n}{2} \rfloor}$. **[5 marks]**

   SOLUTION NOTES: Imagine that we were to sort the list of restaurant recommendations. Now, imagine that some restaurant r is chosen by the majority. There will be a big stretch of r values in the sorted array. Since more than half the array is r values, that stretch must cross the middle. I.e., r will be the median element. (Figuring that out requires a great deal of insight for the exam... but figuring out that "max" and "min" are both wrong answers doesn't require nearly as much insight.)

   Deterministic select finds the i^{th} largest element of an array in worst-case linear time. As we saw in class, we can use it to find the median (or min or max, though it's overkill in that case). We need to pass in the appropriate value to get the i^{th} largest. That will either be the floor or ceiling of $\frac{n}{2}$. (Either will work equally well.)

2. Suppose r is the restaurant code from the previous part. We can check whether or not r was suggested by the majority of the group by $\boxed{\text{counting the r values in a linear scan}}$ in running time [✓ $\Theta(n)$] $\Theta(n \log n)$ $\Theta(n^2)$. **[2 marks]**

3. Good news! The group of friends has chosen a restaurant, Jam's Cafe, and they're lined up outside, waiting for a table. Each person in line either faces the door of the restaurant or faces away from the door, looking toward the people behind them in line. When two neighboring people are facing one another, we say that they are a "conversing pair". We want to design an algorithm to find a pair of conversing people.

   We represent the line as a list of arrows, A, indicating the direction each person is facing. $A[1]$ is at the door of the restaurant, and $A[n]$ is the end of the line. Assume that $A[1] = \rightarrow$ and $A[n] = \leftarrow$.

   Fill in the blanks in the pseudocode below to complete an efficient algorithm to find a conversing pair. `ConversingPair` should take an array of arrows as input, and should return the 1-based index of the $\rightarrow$ from the conversing pair. Given this example: $[\rightarrow, \rightarrow, \rightarrow, \leftarrow]$, your algorithm should return 3. **[8 marks]**

   SOLUTION is below. You may, however, wonder **why** this works. We know initially that the outer arrows face in (the left arrow faces right, the right arrow faces left). Thus, **some** neighboring pair must face each other. (If there are only two elements, then they are a conversing pair. Otherwise, either the left-hand element and its right neighbor are a conversing pair or else the left-hand element's right neighbor and the rightmost element form another pair that face each other across a smaller array. We'll assume (an informal IH) that this smaller array has a conversing pair.)

Now, how fast can we "narrow down" to a conversing pair? Well, we could go "one step at a time" from left to right as the parenthetical proof above suggests. But... any element at all between the left and right—such as the middle element—must either face toward the left element or the right one. Those are its only options. If it faces left, then the left half must contain a conversing pair. If it faces right, then the right half must contain a conversing pair. (In either case, the "half" goes from that middle element to the one we know it faces. If we exclude it, we may end up with everything facing the same direction!)

That's the insight we need for an algorithm!

```
// preconditions: A is a list of arrows; n >= 2;
//                A[1] = ->, A[n] = <-; Indexing is 1-based.
ConversingPair(A, n):
        return CPHelper(A, 1, n)

CPHelper(A, lo, hi):

        If (hi - lo) <= 1:

           return _lo_  // The array is length 2; so, this IS a conversing pair

        Else:

           mid = _floor((lo+hi)/2)_  // ceiling works as well

           If A[mid] _faces_ A[hi]:  // not equal (or equal, with lines below swapped) works.
                                     // However, < and > are meaningless
              return _CPHelper(A, mid, hi)_  // it was mid and hi that faced each other

           Else:

              return _CPHelper(A, lo, mid)_  // it was lo and mid that faced each other
```

4. What is the running time of an efficient algorithm for finding a conversing pair? **[1 marks]**

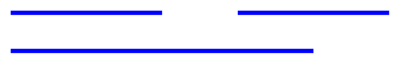
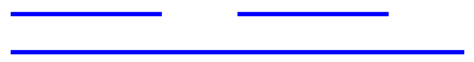
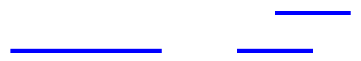
   - ○ $O(1)$
   - [✓ $O(\log n)$]
   - ○ $O(n)$
   - ○ $O(n \log n)$
   - ○ $O(n^2)$
   - ○ We don't have enough information to answer the question.

SOLUTION NOTES: Nothing exciting is happening here. This is, essentially, binary search. The recursive case of a recurrence describing the runtime is roughly $T(n) = T(n/2) + 1$.

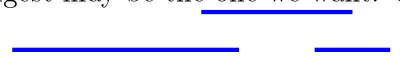
3 Preparing for Sasquatch! [7 marks]

A *performance* is represented by a pair (s, f) where s is its start time and f is its finish time (relative to the start of the festival). There are n performances over the three days, hosted across many stages. Your goal is to **maximize the number of non-overlapping performances in your festival itinerary**.

NEW TEXT: For each of the following greedy algorithms, answer "Yes" if the algorithm *always* constructs an optimal schedule, and "No" otherwise. Your counterexamples may not rely on tie-breaking behaviour. The first problem is answered for you.

	Yes	No	Algorithm:
	○	●	Choose the performance p with the longest duration, discard performances that conflict with p, and recurse on the remaining performances. Clear, simple counterexample, if your answer is “No”: 
1	○	●	Choose the performance p that ends last, discard performances that conflict with p, and recurse on the remaining performances. Clear, simple counterexample, if your answer is “No”: 
2	○	●	If no performances conflict, choose them all. Otherwise, discard the performance with longest duration and recurse on the remaining performances. Clear, simple counterexample, if your answer is “No”: 
3	●	○	If no performances conflict, choose them all. Otherwise, let p be the performance with the earliest start time, and let q be the performance with the second earliest start time: (1) If p and q are disjoint, choose p and recurse on everything but p. (2) If p completely contains q, discard p and recurse. (3) Otherwise, discard q and recurse. Clear, simple counterexample, if your answer is “No”:

SOLUTION NOTES: For the first algorithm, we just have to make the performance that ends last the wrong one. We do that by having it conflict with two separate performances that we could otherwise choose. For the second one, it has two conceptual problems. First, in a conflicting set of performances, the longest may be the one we want. Consider this other counterexample:



Here, the leftmost performance is the longest, but it and the rightmost performance are the optimal solution.

Our example above illustrates the other problem. This algorithm can choose a performance that conflicts with **nothing** and throw it away! Oops :P

Finally, the third algorithm is actually just the "pick the earliest finish time and discard conflicts" algorithm. Let j be the performance with the earlist finish time. We want the algorithm above to select it and discard all conflicts. We prove that it does this by cases:

1. If j also has the earliest start time, the algorithm above picks it as p. The performance with the next earliest start time either doesn't exist (in which case no conflicts exist and the algorithm selects all performances), doesn't conflict with j (in which case the algorithm above picks them both), or both starts and finishes after j (because j has both the earliest start and finish times) and so it is discarded and the algorithm recurses. Thus, we discard conflicting jobs until we can select p.

2. If some other performance i has an earlier start time, then it must completely contain j. (As the earliest starting performance, it starts before j, but since j is the earliest ending performance, it ends after j.) So, we won't fall into case (1) above. Whether we fall into case (2) or (3), we discard that other performance i and recurse. By (an entirely informal argument by) induction, we'll eventually get to i having the earliest start time as well.

4 Greed is Part of our DNA [22 marks]

We define a "correspondence" between a string A and a string B as: (1) a way to space out and line up the letters of A beneath matching letters of B, or equivalently (2) a list, for each letter of A, of an index it matches in B, where the indexes are strictly increasing. For example, here is a correspondence between `TCCG` and `AAGTACGCG`:

B:	A	A	G	T	A	C	G	C	G
A:				T		C		C	G
indexes				4		6		8	9

Now, consider the DNA Subsequence Test problem (DST): Given two DNA sequences A and B (i.e., two strings containing only the letters A, T, C, and G) of length n and m respectively—where $n \leq m$—determine whether a correspondence exists between A and B.

1. **Solve the instances of DST in the table** below. Then **answer the question below the table**. In any row whose answer is "Yes", also give the list of 1-based indexes for a correspondence between A and B. The first DST instance is solved for you. **[4 marks]**

A	B	Solution	Correspondence (if any)
TCCG	AAGTACGCG	[✓ Yes] No	4, 6, 8, 9
ACT	ACT	[✓ Yes] No	1, 2, 3
ACT	CATGAT	Yes [✓ No]	
ATAT	ACATAGT	[✓ Yes] No	1, 4, 5, 7 or 3, 4, 5, 7

Does any instance in the table have more than one correspondence between A and B?
[✓ Yes] No

2. Complete this reduction from DST to the Longest Common Subsequence problem (LCS). Recall that LCS takes two strings and produces the longest string that is a subsequence of both input strings. **[3 marks]**

Note that a solution to DST is a `Yes` or `No` answer, not the correspondence itself.

Given an instance of DST: A, B (where $\text{len}(A) = n$ and $\text{len}(B) = m$).

Produce an instance of LCS: $\boxed{\text{A, B}}$

Given a solution S to that instance of LCS, produce `Yes` as the solution to DST exactly when: $\boxed{\text{S = A or } \text{len}(S) = n}$.

SOLUTION NOTES: In general, the longest the LCS of A and B can be is the full length of the shorter of the strings. In this case, A is the shorter string. If the LCS of A and B is as long as A, then it is precisely A (since it must be a subsequence of A). In that case (and only in that case), A is also a subsequence of B, and that's what we need!

3. Complete the following correct, recursive pseudocode for an efficient greedy algorithm to solve DST. The algorithm produces just `Yes` or `No`, but to correctly solve the problems below, you should understand how it could be adapted to discover a correspondence. **[5 marks]**

   NOTE: The line numbers on the left are for use in a subsequent problem.

```
    // preconditions: A and B are strings composed of only the letters A, T, C, and G
    //                len(A) <= len(B). Indexing is 1-based.
    IsSubsequence(A, B):
        // A helper function that does all the work.
        Helper(i, j):
1           If i > len(A):

2              return _Yes_

3           Else if j > len(B):

4              return _No_

5           Else if A[i] = B[j]:

6              return _Helper(i+1, j+1)_

7           Else:

8              return _Helper(i, j+1)_

        // Initial call to the helper:
9       return Helper(1, 1)
```

   SOLUTION NOTES: If we've "run out" of string A, we've found matches for all its characters. That's good, even if we've also run out of B at the same time. Otherwise, if we've run out of B (and not A), that's bad because we lack matches for the remainder of A. Otherwise, there's at least one more character to test in B for possible matches with A. If there's a match, we greedily accept it as part of the correspondence and move on to the next letter to match from A and the next candidate match in B. Otherwise, we still need to match the A character, but we're done with this B character.

4. Give a good big-O bound on the runtime of an efficient greedy algorithm for DST in terms of n and/or m. **[2 marks]**

   Bound: $\boxed{\text{O(m)}}$.

   SOLUTION NOTES: Every recursive call increases `j`. So, the worst we can do is run through all of the longer string, B.

5. A memoized version of `IsSubsequence` (or, more precisely, its helper function) does not make sense for which of the following reasons? Fill in the boxes next to **ALL** correct answers. **[2 marks]**

   - [✓] No subproblem is solved more than once in any call to `IsSubsequence`.
   - [✓] Memoization would not improve the running time.
   - ☐ The function already uses O(mn) memory.
   - ☐ The parameters are not values suitable for memoization.
   - ☐ The solution to `IsSubsequence` is just "Yes" or "No".

   SOLUTION NOTES: Indeed we never solve any subproblem more than once. It's a greedy algorithm; we make a locally optimal decision and then move on, never looking back. As a result, memoization cannot help with the runtime (but can waste memory). However, the function definitely does *not* use $O(mn)$ memory (and if it did, it'd be a reason that memoization would be asymptotically cheap, not a reason not to use memoization). The parameters to `Helper` particularl (and even to `IsSubsequence`, if need be) are fine for memoization. And finally, we can memoize a Yes-or-No answer just as easily as we can memoize any other answer.

6. Fill in the blanks in the following **PARTIAL** proof of the correctness of `IsSubsequence`. **[6 marks]**

   Assume for an arbitrary instance A, B of DST that a correspondence—a 1-based list of strictly increasing indexes $C_{\mathcal{O}} = [c_1, c_2, \ldots, c_n]$ of length $n = \text{len}(A)$—between A and B exists.

   We can think of `IsSubsequence` as discovering the next element of its own correspondence $C_{\mathcal{G}}$ between A and B each time that it reaches line $\boxed{6}$. At that line, it discovers that $C_{\mathcal{G}}[\boxed{\text{i}}] = \boxed{\text{j}}$.

   We now reason about the correspondenc $C_{\mathcal{G}}$ formed this way.

   We prove by induction that at any given index $1 \leq i \leq n$, $C_{\mathcal{O}}[i] \geq C_{\mathcal{G}}[i]$.

   Base case: When $i = 1$, *EXCLUDED FROM THIS PARTIAL PROOF*. Thus $C_{\mathcal{O}}[i] \geq C_{\mathcal{G}}[i]$.

   Inductive case: Consider an arbitrary integer i where $\boxed{2} \leq i \leq n$. Assume that $C_{\mathcal{O}}[\boxed{\text{i-1}}] \geq C_{\mathcal{G}}[\boxed{\text{i-1}}]$. Then, it must be that $C_{\mathcal{O}}[i] \geq C_{\mathcal{G}}[i]$ because

   > $C_{\mathcal{O}}[i]$ and $C_{\mathcal{G}}[i]$ both refer to $A[i]$. The greedy algorithm searches for $A[i]$ in B starting at $C_{\mathcal{G}}[i-1]$: before $C_{\mathcal{O}}[i-1]$ (by the IH) and thus before $C_{\mathcal{O}}[i]$. So, it will find $A[i]$ at $B[C_{\mathcal{O}}[i]]$ if not before.

   This completes our inductive proof that at any given index $1 \leq i \leq n$, $C_{\mathcal{O}}[i] \geq C_{\mathcal{G}}[i]$.

5 Making Every Second Count [11 marks]

NEW TEXT: Aconcagua's new "easy pricing" (E for short) customers buy storage over a period of n seconds (numbered 1 through n), and Aconcagua ensures that they pay the lowest price for that period achievable under the following rules:

> At each successive second i with auction price for that second $p[i]$, choose either (1) to pay $p[i] + 10$, (2) to fix your price for that second and the next two at $p[i] + 20$, or (3) to fix your price at $p[i] + 50$ from that second up to and including a later second $j \leq n$. Note that option (2) is not available for $i + 2 > n$.

For example, the following illustrates price data over a period of 12 seconds and the pricing scheme chosen for E over this period of the options above (1), (2), and (3):

time in seconds (i)	1	2	3	4	5	6	7	8	9	10	11	12
$p[i]$	80	85	103	98	37	41	145	80	200	39	21	29
option:	(1)	(2)			(1)	(3)				(1)	(1)	(1)
price:	90	105	105	105	47	91	91	91	91	49	31	39

with total cost 935. (A small change can alter strategies. E.g., decreasing $p[8]$ to 75 changes the strategy at seconds 5–10, inclusive.)

1. Fill in the blanks to complete the following recurrence to compute E prices given the array p of prices for the period $1, \ldots, n$. **[7 marks]**

   Implementation notes: All indexes are 1-based. $p[i] = \infty$ for $i \leq 0$. The price of a 0 second period is 0. The minimum over an empty set of options is ∞.

$$E(n) = \begin{cases} \infty & \text{when } n < 0 \\ \boxed{0} & \text{when } n = 0 \\ \min \begin{cases} \boxed{\text{E(n-1) + p[n] + 10}}, \\ E(n-3) + (p[n-2] + 20) * 3, \\ \min\limits_{1<i\leq n} \{ E(\boxed{\text{n-i}}) + (p[n-i+1] + 50) * i \} \end{cases} & \text{when } n > 0 \end{cases}$$

   SOLUTION NOTES: Each of these is a translation "looking backward from the end" of one of the pricing policies described above. For example, we can freeze a price for a particular second and the two after it. Looking back from index n, then, we can freeze n, $n-1$, and $n-2$ at $p[n-2] + 20$, which leaves the subproblem $E(n-3)$.

2. Imagine that we write a direct, recursive, memoized implementation of this recurrence that records the solutions to all subproblems except the base cases in a table after it solves each subproblem the first time. **[4 marks]**

(a) Exactly (not asymptotically), how many subproblems will this implementation memoize when solving $E(100)$? $\boxed{100}$.

SOLUTION NOTES: 100 absolutely *is* a subproblem (just as an entire binary tree is a subtree of itself and an entire set a subset of itself). The comments above say base cases aren't memoized, and the whole point of memoization is that we memoize each subproblem at most once. So, we memoize all of $1, 2, 3, \ldots, 100$.

(b) Imagine the memoized implementation was called initially on a problem of size n. We'd like to bound the runtime of the recursive case of this memiozed implementation as it solves a (possibly smaller) subproblem i, **not counting the runtime of recursive calls to solve subproblems of i.**

Fill in the circle next to the best bound on this runtime.

- ○ $O(1)$
- [✓ ● $O(i)$]
- ○ $O(n)$
- ○ $O(2^i)$
- ○ $O(2^n)$

SOLUTION NOTES: It's somewhat coincidental that subproblems are numbers here, but since they are, we can use them in bounds. The first two recursive cases in solving a non-base-case subproblem $E(i)$ take constant time each. However, the third refers to every subproblem in the range from $i - 2$ down to 0. That's $O(i)$ subproblems, with computation on each taking constant time. Thus, $O(i)$. (Of course, $O(i) \subseteq O(n)$ since $i \leq n$, and 2^i and 2^n are even larger. However, the *best* answer is $O(i)$.)

(c) Give a good overall big-O bound on the runtime of an efficient memoized implementation on a problem of size n: $\boxed{\mathrm{O(n^2)}}$

Roughly speaking, we can think of ourselves as spending 1 unit of time on $E(1)$, 2 on $E(2)$, 3 on $E(3)$, and so on up to n on $E(n)$. I.e., $1 + 2 + 3 + \ldots + n \in O(n^2)$.

6 BONUS: From the Cutting Room Floor [3 BONUS marks]

DO complete the **first** of these bonus problems!

Bonus marks add to your exam and course bonus mark total but are **not** required. **WARNING**: These questions are too hard for their point values. We are free to mark these questions harshly. Finish the rest of the exam before attempting these questions. Do not **taunt** these questions.

1. Give a recurrence relation that describes how to maximize your favorite things about CPSC 320 and minimize your least favorite. No creativity is required for marks, but you **must** at least define a function of one parameter. The best/funniest/most profound **may** just earn 1 additional course bonus point (but no more exam points). ☺

2. Follow the rules for the "Preparing for Sasquatch!" problem on this slightly different greedy algorithm:

Yes	No	Algorithm:
◯	◯	If no performances conflict, choose them all. Otherwise, let p be the performance with the earliest start time, and let q be the performance with the second earliest start time: (1) If p and q are disjoint, choose p and recurse on everything but p. (2) If p completely contains q, discard p and recurse. (3) Otherwise, discard q, choose p, and recurse. Clear, simple counterexample, if your answer is "No":

Furthermore: If your answer is "Yes", sketch the key points in a brief proof of the correctness of this algorithm. If your answer is "No", you already gave a counterexample, but clearly indicate both the optimal solution, the (suboptimal) greedy algorithm's solution, and why the greedy algorithm generates that solution.

3. Give and briefly justify the correctness and performance of a linear-time, constant-memory solution to the fine dining (FD) problem.

This page intentionally left (almost) blank.

If you write answers here, you must CLEARLY indicate on this page what question they belong with AND on the problem's page that you have answers here.