Homework #2 Solutions

CS 344: Design and Analysis of Computer Algorithms (Fall 2022)

Problem 1

For all the recursive formula problems below, you can assume that $T(1) = T(2) = 1$.

(1) Consider an algorithm that solves a problem of size n by dividing it into 6 pieces of size $n/6$, recursively solving each piece, and then combining the solutions in $O(n^3)$ time. What is the recurrence formula for this algorithm? Use a recursion tree to figure out the running time of the algorithm (in big-O notation, as always) AND then use a proof by induction to show that your result is correct.

The recurrence relation for $T(n)$ is

$$T(n) = 6T(n/6) + n^3.$$

The recursion tree is then summarized as follows:

level	number of problems	problem size	non-recursive work
0	1	n	$1 \cdot (n)^3$
1	6	$n/6$	$6 \cdot (n/6)^3$
2	6^2	$n/6^2$	$6^2 \cdot (n/6^2)^3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
ℓ	6^ℓ	$n/6^\ell$	$6^\ell \cdot (n/6^\ell)^3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_6 n$	$6^{\log_6 n}$	$n/6^{\log_6 n}$	$6^{\log_6 n} \cdot (n/6^{\log_6 n})^3$

The ℓth level of the recursion tree sums to $6^\ell \cdot (n/6^\ell)^3 = n^3/36^\ell$ non-recursive work done (number of problems times non-recursive work done per problem sized at that level), and there are $\log_6 n$ levels. Summing across non-recursive work done on all levels of the recursion tree gives

$$\begin{aligned}
\sum_{\ell=0}^{\log_6 n} n^3/36^\ell &= n^3 \sum_{\ell=0}^{\log_6 n} (1/36)^\ell \\
&\leq n^3 \sum_{\ell=0}^{\infty} (1/36)^\ell \\
&= n^3 \frac{1}{1 - 1/36} \qquad \text{(Geometric sum formula when radius} < 1\text{)} \\
&= \frac{36}{35} n^3.
\end{aligned}$$

Thus, $T(n) = O(n^3)$. This is a top-heavy recurrence.

We will now prove this formally by showing $T(n) \leq \frac{36}{35} n^3$ by induction.

Base case: By assumption, $T(k) \leq \frac{36}{35} k^3$ when $k < 6$. These are the atomic problem sizes for which no more division into smaller sub-problems are done

(i.e. the recursion ends).

Inductive hypothesis: $T(k) \leq \frac{36}{35}k^3$ for all $k < n$.

We want to prove: $T(n) \leq \frac{36}{35}n^3$.

Proof / Inductive Step:

$$\begin{aligned} T(n) &= 6T(n/6) + n^3 \\ &\leq 6 \cdot (\frac{36}{35} \cdot (n/6)^3) + n^3 \qquad \text{(Inductive Hypothesis applied to } T(n/6)) \\ &= n^3/35 + n^3 \\ &= \frac{36}{35}n^3, \end{aligned}$$

as desired.

(2) Say that you have an algorithm with recurrence formula $T(n) = 16T(n/4) + n^2$. Use a recursion tree to figure out the running time of your algorithm. *No need for an induction proof of this one.*

The recursion tree is summarized as follows:

level	number of problems	problem size	non-recursive work
0	1	n	$1 \cdot (n)^2$
1	16	$n/4$	$16 \cdot (n/4)^2$
2	16^2	$n/4^2$	$16^2 \cdot (n/4^2)^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
ℓ	16^ℓ	$n/4^\ell$	$16^\ell \cdot (n/4^\ell)^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_4 n$	$16^{\log_4 n}$	$n/4^{\log_4 n}$	$16^{\log_4 n} \cdot (n/4^{\log_4 n})^2$

The ℓth level of the recursion tree sums to $16^\ell \cdot (n/4^\ell)^2 = n^2$ non-recursive work done (number of problems times non-recursive work done per problem sized at that level), and there are $\log_4 n$ levels. Summing across non-recursive work done on all levels of the recursion tree gives

$$\begin{aligned} \sum_{\ell=0}^{\log_4 n} n^2 &= \sum_{\ell=0}^{\log_4 n} n^2 \cdot 1 \\ &= n^2 \sum_{\ell=0}^{\log_4 n} 1 \\ &= n^2(\log_4 n + 1). \end{aligned}$$

Thus, $T(n) = O(n^2 \log n)$. This is a balanced recurrence.

(3) Say that you have an algorithm with recurrence formula $T(n) = 8T(n/5) + n$. Use a recursion tree to figure out the running time of your algorithm. *No need for an induction proof on this one.*

The recursion tree is summarized as follows:

level	number of problems	problem size	non-recursive work
0	1	n	$1 \cdot n$
1	8	$n/5$	$8 \cdot n/5$
2	8^2	$n/5^2$	$8^2 \cdot n/5^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
ℓ	8^ℓ	$n/5^\ell$	$8^\ell \cdot n/5^\ell$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_5 n$	$8^{\log_5 n}$	$n/5^{\log_5 n}$	$8^{\log_5 n} \cdot n/5^{\log_5 n}$

The ℓth level of the recursion tree sums to $8^\ell \cdot n/5^\ell = n \cdot (8/5)^\ell$ non-recursive work done (number of problems times non-recursive work done per problem sized at that level), and there are $\log_5 n$ levels. Summing across non-recursive work done on all levels of the recursion tree gives

$$\begin{aligned}
\sum_{\ell=0}^{\log_5 n} n \cdot (8/5)^\ell &= n \sum_{\ell=0}^{\log_5 n} (8/5)^\ell \\
&= n \cdot \frac{(8/5)^{\log_5 n + 1} - 1}{3/5} && \text{(Geometric sum formula when radius } > 1) \\
&\leq n \cdot \frac{(8/5)^{\log_5 n + 1}}{3/5} \\
&= (8/3) \cdot n \cdot (8/5)^{\log_5 n} && \text{(Pulling a 8/5 out; } \tfrac{8/5}{3/5} = 8/3) \\
&= (8/3) \cdot n \cdot n^{\log_5 8/5} && (x^{\log_b y} = y^{\log_b x}) \\
&= (8/3) \cdot n^{1 + \log_5 8/5} \\
&= (8/3) \cdot n^{\log_5 5 + \log_5 8/5} && (1 = \log_b b) \\
&= (8/3) \cdot n^{\log_5 8}. && (\log_b x + \log_b y = \log_b xy)
\end{aligned}$$

Thus, $T(n) = O(n^{\log_5 8})$, where $\log_5 8 \approx 1.29$. This is a bottom-heavy recurrence.

Problem 2

For this problem, we are assuming that arithmetic operations take $O(1)$ time. So addition, subtraction, division, and multiplication, are done in $O(1)$ time no matter how big the numbers are. (This is not exactly true in practice, but it's a reasonable approximation to reality.)

Now consider the following function SQRT(n)

- Input: a positive integer n
- Output: an integer k such that $k^2 = n$, or "No Solution" if none exists

Example: SQRT(9) should output 3, while SQRT(10) should output "no solution"

The Problem Write pseudocode for a recursive algorithm that computes SQRT(n) in time $O(\log(n))$. In addition to pseudocode, you should state the recursion formula for your algorithm. So all you need is pseudocode and the recursion formula, nothing else. You do NOT need to prove that this recursion formula solves to $T(n) = O(\log(n))$

NOTE Don't use tools from outside this class. For example, some of you know Newton's method, but you should not use that. I'm looking for a simple algorithm that only uses ideas we've covered in class so far.

Naively, we can test if $1^2 \stackrel{?}{\geq} n$, then $2^2 \stackrel{?}{\geq} n$, ..., until $n^2 \stackrel{?}{\geq} n$. That's n tests, which is far too many to meet our $\log n$ target.

Notice, however, that the results of these tests, in the order they are done, are monotone: a sequence of falses followed by a sequence of trues. If there is indeed a solution $k^2 = n$, it would be at the first occurrence of a true. In view of this, we can use binary search!

```
Sqrt(n):
    return SqrtBinarySearch(n, 0, n)
```

```
SqrtBinarySearch(n, ℓ, r):
    if ℓ > r
        return No Solution
    k ← ⌊(l + r)/2⌋
    if k^2 > n
        return SqrtBinarySearch(n, ℓ, k − 1)
    else if k^2 < n
        return SqrtBinarySearch(n, k + 1, r)
    else
        return k
```

Let $t(k, n)$ be the runtime of the test computed at each recursive call of binary search. In this case, the test is essentially $k^2 \stackrel{?}{\geq} n$ and so $t(k, n) = O(1)$. We know that binary search halves its search range and does $t(k, n) = O(1)$ work at every recursive call and so $T(n) = T(n/2) + O(1)$. Moreover, we know (from knowing about binary search) that $T(n) = T(n/2) + O(1)$ solves to $T(n) = O(\log n)$.

Problem 3

Given an array A, we say that elements $A[i]$ and $A[j]$ are swapped if $j > i$ but $A[j] < A[i]$. For example, if $A = [8, 5, 9, 7]$, then there are a total of 3 swapped pairs, namely 8 and 5; 8 and 7; and 9 and 7.

The goal for this problem is to write a recursive algorithm that given an array A determines the number of swapped pairs in the array in $O(n \log(n))$ time. The algorithm to do this is extremely similar to merge sort; in fact, the algorithm does all of merge sort (verbatim), plus a few extra things to keep track of the number of swaps. In particular, one side effect of the algorithm is to sort the array A.

what you need to do: In order to save you the trouble of retyping the pseudocode for merge sort,I have written the pseudocode for most of the algorithm. Your job is simply to fill in two lines. These are the key lines that keep track of the number of swaps.

Note that the algorithm below has an outer function MergeSortAndCountSwaps which I wrote entirely; nothing to fill in. But MergeSortAndCountSwaps then calls MergeAndCountSwapsBetween as a subroutine; in that subroutine you will have to fill in two lines.

The Algorithm MergeSortAndCountSwaps(A)
INPUT: array A with unique elements (no duplicates)
OUTPUT: pair (SortedA, S), where SortedA is the array A sorted in increasing order and S is the total number of swapped pairs in A.

- A1 $\leftarrow$ first half of A. So $A1 = A[0], ..., A[\frac{n}{2} - 1]$
- A2 $\leftarrow$ second half of A. So $A2 = A[\frac{n}{2}], ..., A[n-1]$
- (SortedA1,S_1) $\leftarrow$ MergeSortAndCountSwaps(A1)
- (SortedA2,S_2) $\leftarrow$ MergeSortAndCountSwaps(A2)
- (SortedA, S_3) $\leftarrow$ MergeAndCountSwapsBetween(SortedA1,SortedA2)
- Return (SortedA, $S_1 + S_2 + S_3$).

We now need to define MergeAndCountSwapsBetween(A, B). This is the part where I will have you fill in a few lines. Note that MergeAndCountSwapsBetween(A, B) is very similar to Merge(A,B) from class, and in particular it is NOT a recursive function.
MergeAndCountSwapsBetween(A,B):

(a) Initialize array C of size $2n$

(b) $i \leftarrow 0$

(c) $j \leftarrow 0$

(d) $S' \leftarrow 0$ Comment: S' will count swaps; the answer lines below which you have to add will be responsible for changing S'.

(e) While (i < n OR j < n)

(i) if $j = n$ or A[i] < B[j] :

- **YOUR ANSWER 1 GOES HERE**
- set next element of C to $A[i]$

- (formally, $C[i+j] \leftarrow A[i]$)
- i ← i+1

(ii) if i = n or A[i] > B[j]:

- **YOUR ANSWER 2 GOES HERE**
- set next element of C to $B[j]$
- j ← j+1

(f) output pair (C, S').

Questions you need to answer:

- What goes in Answer 1? *your answer should be a single line.*
- What goes in Answer 2? *your answer should be a single line.*

(Answer 1) Do nothing

(Answer 2) $S' \leftarrow S' + (n - i)$

Explanation MergeAndCountSwapsBetween counts the number of swaps between elements in A and elements in B (intra-A and intra-B swaps have already been counted by recursive calls).

We can count these swaps by computing, for every element in B, the number of swaps with A it is involved in. Note that at any step, the remaining elements of B are larger than the elements in C; they are thus not involved in any swaps with elements in C that have come from A. This is an invariant the algorithm maintains. The remaining elements of B can, however, be involved in swaps with the remaining elements of A.

When $A[i] < B[j]$, none of the remaining elements of B are involved in a swap with $A[i]$. Moving $A[i]$ to C thus maintains our invariant.

When $A[i] > B[j]$, we know that

- The original index of $B[j]$ is larger than that of $A[i], A[i+1], \ldots, A[n-1]$.
- $B[j] < A[i] < A[i+1] < \cdots < A[n-1]$.

$B[j]$ is thus involved in swaps with all of $A[i], A[i+1], \ldots, A[n-1]$, of which there are $n - i$ many.

Problem 4

Given an array A of length n, we say that x is a *frequent* element of A if x appears at least $n/4$ times in A. Note that an array can have anywhere between 0 and 4 frequent elements.

Now, Say that your array A consists of incomparable objects, where you can ask whether two objects are equal (i.e. check if $A[i] == A[j]$), but there is NOT a notion of one object being bigger than another. In particular, we cannot sort A or find the median of A. (Think of e.g. an array of colors.)

(1) Write pseudocode for an algorithm IsFrequent(A,x) which takes as input the array A of incomporable objects and a single element x, and returns TRUE if x is frequent in A and FALSE otherwise. Your algorithm should run in $O(n)$ time, where n is the length of array A.

```
IsFrequent(A[0..n − 1], x):
  k ← 0
  for y ∈ A
      if x = y
          k ← k + 1
  return 4k ≥ n
```

(2) Given an array A of n incomparable objects, write pseudocode for an algorithm that finds all the frequent elements of A, or returns "no solution" if none exists. Your algorithm should run in $O(n \log(n))$ time.

The idea is simple: find the frequent elements of the left- and right-halves, then test which of the elements are frequent with respect to the entire list.

```
FindFrequentElements(A[0..n − 1]):
  if A is empty
      return No Solution
  if n ≤ 4
      return deduplicated elements of A
  i ← ⌊n/2⌋
  F[0], F[1], F[2], F[3] ← FindFrequentElements(A[0..i − 1])
  F[4], F[5], F[6], F[7] ← FindFrequentElements(A[i..n])
  for f ∈ F
      if not IsFrequent(A, f)
          remove f from F
  if F is empty
      return No Solution
  else
      return deduplicated elements of F
```

If $T(n)$ is the running time of FindFrequentElements on a list of length n, then, ignoring floors and ceilings, $T(n) = 2T(n/2) + O(n)$ since each problem spawns two subproblems with half the size and calls IsFrequent which takes $O(n)$ time. Asymptotically, we have $T(n) = O(n \log n)$; this is the same recurrence relation as MergeSort from lecture.

To justify the algorithm, we must argue that the frequent elements of A are

frequent in either the left- or the right-half of A. We show the contrapositive: if x is neither frequent in the left-half of A nor the right-half of A, then x is not frequent in A. Indeed, if x is not frequent in either half, then x appears less than $\lfloor n/2 \rfloor \cdot 1/4$ times in the left-half and less than $\lceil n/2 \rceil \cdot 1/4$ times in the right-half. Across all of A, then, x appears no more than $n/8 - 1 + n/8 = n/4 - 1$ times. Hence, x is not frequent in A.

Problem 5

Give an $O(n)$ algorithm for problem 4. *Your algorithm does not need to be recursive.* (The simplest solution I can think of of is indeed not recursive, though there exists a recursive solution as well.)

For this problem, you need to justify both correctness and running time. As always, "justify" means that a reader who doesn't know the solution should be able to read your solution and be convinced that it is correct.

Consider the following algorithm FindFrequentElements. The algorithm passes over A once to find candidate frequent elements $F[0], F[1], F[2], F[3]$. Once candidates have been found (the array has been passed over), the algorithm checks that each candidate is indeed frequent, and outputs the ones that are (or NoSolution if none are).

Becoming a candidate. If there is some candidate-vacancy (some $F[j] = \perp$) during the pass of A, then an element can become a candidate and its level $L[j]$ is set to 1.

Staying a candidate. Every time $F[j]$ is seen, its level $L[j]$ increases by 1. If, on the other hand, an element is seen that is not currently a candidate and, moreover, there are no candidate-vacancies (no $F[j] = \perp$), the levels of all candidates decrease by 1. If any candidate's level reaches 0, it ceases to be a candidate (but can become a candidate again later).

```
FindFrequentElements(A[0..n − 1]):
  // j ranges from 0 (inclusive) to 3 (inclusive).
  F[j] ← ⊥ for all j
  L[j] ← 0 for all j
  for i ← 0 to n − 1
      if some F[j] = A[i]
          L[j] ← L[j] + 1
      else if some F[j] = ⊥
          F[j] ← A[i]
          L[j] ← 1
      else
          L[j] ← L[j] − 1 for all j
          for all L[j] = 0 set F[j] ← ⊥
  for f ∈ F
      if not IsFrequent(A, f)
          remove f from F
  if F is empty
      return No Solution
  else
      return deduplicated elements of F
```

Call an index i from 0 to $n-1$ a *diminisher* if $A[i]$ is not a candidate in the ith step of the main loop, and there are no vacant candidates; put otherwise, the else-block in the main loop is executed. Observe that there can be no more than $n/5$ diminishers. Why? Every diminisher discounts a total of 4 previously seen elements

from L, in addition to itself. This makes a total discount of 5 per diminisher. If there were more than $n/5$ diminishers, then there would be more than n discounts; this is not possible (we can only discount that which has already been counted or which is doing the discounting).

Now consider a frequent element f in A, should there be any. Among the at least $n/4$ times f appears in A, at most $n/5$ of them can be discounted. There must thus be some $F[j] = f$ with $L[j] \geq n/4 - n/5 > 0$.