Exam #1 Solutions

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

Problem 1

(1) Say you had an algorithm with recurrence formula $T(n) \leq 27(n/3) + n^2$. Use a recursion tree (or recursion table) to solve for $T(n)$.

The recursion tree is then summarized as follows:

level	number of problems	problem size	non-recursive work
0	1	n	$1 \cdot (n)^2$
1	27	$n/3$	$27 \cdot (n/3)^2$
2	27^2	$n/3^2$	$27^2 \cdot (n/3^2)^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
ℓ	27^ℓ	$n/3^\ell$	$27^\ell \cdot (n/3^\ell)^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_3 n$	$27^{\log_3 n}$	$n/3^{\log_3 n}$	$27^{\log_3 n} \cdot (n/3^{\log_3 n})^2$

The ℓth level of the recursion tree sums to $27^\ell \cdot (n/3^\ell)^2 = 3^\ell \cdot n^2$ non-recursive work done (number of problems times non-recursive work done per problem sized at that level), and there are $\log_3 n$ levels. Summing across non-recursive work done on all levels of the recursion tree gives

$$\begin{aligned}
\sum_{\ell=0}^{\log_3 n} 3^\ell \cdot n^2 &= n^2 \sum_{\ell=0}^{\log_3 n} 3^\ell \\
&= n^2 \cdot \frac{3^{\log_3 n+1} - 1}{2} && \text{(Geometric sum formula)} \\
&\leq n^2 \cdot \frac{3^{\log_3 n+1}}{2} \\
&= 3n^3/2.
\end{aligned}$$

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

(2) Say you had an algorithm with recurrence formula $T(n) \leq 4T(n/3) + 5T(n/4) + n^4$. Use *induction* to show that $T(n) = O(n^4)$.

We will show that $T(n) \leq 2n^4$, and consequently $T(n) = O(n^4)$.
Base case: By assumption, $T(i) = 1 \leq 2 \cdot i^4$ for $i = 1, 2, 3$. These are the atomic problem sizes for which no more division into smaller sub-problems are done (i.e. the recursion ends).
Inductive hypothesis: $T(k) \leq 2k^4$ for all $k < n$.
We want to prove: $T(n) \leq 2n^4$.

Proof / Inductive Step:

$$\begin{aligned}
T(n) &\le 4T(n/3) + 5T(n/4) + n^4 && \text{(Recurrence)} \\
&\le 4 \cdot 2(n/3)^4 + 5 \cdot 2(n/4)^4 + n^4 && \text{(I.H. on } T(n/3), T(n/4)\text{)} \\
&= n^4(8/81 + 10/256 + 1) \\
&\le 2n^4.
\end{aligned}$$

Problem 2

Recall the Select(A,k) from class where we broke the array into chunks of size 5, found the median of each chunk, recursively computed the median of the medians and so on.

Let us say that we use a different algorithm where we break the array into chunks of size 13 rather than size 5. Other than that the algorithm is the same. So the algorithm is:

(a) Break A into chunks of size 13

(b) Compute the median of each chunk

(c) Recursively compute the median of the medians (call it m)

(d) Partition around m

(e) Recurse to one side

What is the recurrence formula for this algorithm? That is, with chunks of size 5 we had recurrence formula $T(n) \leq T(n/5) + T(7n/10) + O(n)$. So what is the recurrence formula if we use chunks of size 13 instead? Make sure to explain why your recurrence formula is the correct one. As in class, you can assume that all elements of the array A are distinct (no duplicates).

Consider the median of chunk medians m. It is larger than 1/2 of the chunk medians. Since there are $n/13$ chunk medians, m is therefore larger than $n/26$ chunk medians. Each of these medians is in turn larger than 6 other elements. Thus, m is larger than at least $(n/13) * (1/2) * 7 = 7n/26$ elements. Symmetrically, m is smaller than at least $7n/26$ elements. Hence, the size of the array recursed on has size at most $19n/26$.

The algorithm recurses once on the $n/13$ chunk medians and once on one side of the array, which has size at most $19n/26$. Thus, the recurrence satisfies $T(n) \leq T(n/13) + T(19n/26) + O(n)$.

Problem 3

Consider the following problem. We are given an array $A[0...n-1]$ of positive numbers with $len(A) = n$ (duplicates allowed). We say that a subarray $A[i]...A[j]$ is *special* if $j-i+1 = n/5$ (i.e. the length of the subarray is $n/5$) and also there is some number x such that x appears at least 100 times in the subarray $A[i]...A[j]$.

Now consider the following problem:

FindSpecialSubarray(A)

- INPUT: an array $A[0...n-1]$ of positive numbers (duplicates allowed).
- OUTPUT: indices i, j such that subarray $A[i...j]$ is special. Or “no solution" if no special subarray exists. If there exists multiple special subarrays $A[i...j]$ you only have to return one of them.

Write pseudocode for an algorithm that solves FindSpecial(A) in $O(n)$ time.

We use a sliding window approach.

```
FindSpecialSubarray($A[0..n-1]$):
  $D \leftarrow$ BuildDictionary()
  for $j \leftarrow 0$ to $n-1$
      $i \leftarrow j - n/5 + 1$
      $c \leftarrow D.search(A[j])$
      $D.add(A[j], c+1$ if $c \neq$ Nil and 1 otherwise)
      if $i < 0$
          continue
      if $D.search(A[i]) \geq 100$ or $D.search(A[j]) \geq 100$
          return $(i, j)$
      $D.add(A[i], D.search(A[i]) - 1)$
  return NoSolution
```

Correctness. Let S be a special subarray. Then by definition there would be an element x in S that occurs at least 100 times in S. Either the last occurring x must enter a sliding window of length $n/5$ (from the right side of the window), or the first occurring x must leave a sliding window of length $n/5$ (from the left side of the window), and thus the correctness of FindSpecialSubarray follows.

To see why either the last x enters or the first x exits our sliding window, observe that placing a window starting from the first x and placing another window ending at the last x covers at most $2n/5$ indices of A. The remaining indices of A must be covered at some point in time by a window, which means one of the events must occur as our window slides across A.

Time complexity. The loop has $O(n)$ iterations, each which takes $O(1)$ time in expectation. The total running time is thus $O(n)$ in expectation.

Problem 4

Consider the following problem
FindCutoff(A)

- INPUT: an array $A[0...n-1]$ of length n. You can assume that all numbers in A are positive and unique (i.e. no duplicates). You can NOT assume that A is sorted.
- OUTPUT: find the *largest* number x such that A contains at least x elements that are $\geq 10x$.

(1) Write pseudocode for an algorithm that solves FindCutoff(A) in $O(n \log(n))$ time.

```
FindCutoff($A[0..n-1]$):
    $A \leftarrow$ sort($A$)
    for $x \leftarrow n$ to 1
        if $A[n-x] \geq 10x$
            return $x$
    return 0
```

Correctness. Suppose x is the cutoff. Then, after sorting A, we have $A[n-1] \geq A[n-2] \geq .. \geq A[n-x] \geq 10x$. We need only check $A[n-x] \geq 10x$ since it implies the rest.

Time complexity. The running time is bounded by sorting, which takes $O(n \log(n))$ time.

(2) Write pseudocode for an algorithm that solves FindCutoff(A) in $O(n)$ time.

```
FindCutoff($A[0..n-1]$):
    return FindCutoff($A, 0$)
```

```
FindCutoff($A[0..n-1], \mathit{offset}$):
    if $n = 0$
        return 0
    $m \leftarrow$ Select($A, \lfloor n/2 \rfloor + 1$)
    $L, R \leftarrow$ Partition($A, m$)
    if $m \geq 10 \cdot (1 + \mathrm{len}(R) + \mathit{offset})$
        return FindCutoff($L, 1 + \mathrm{len}(R) + \mathit{offset}) + 1 + \mathrm{len}(R)$
    else
        return FindCutoff($R, \mathit{offset}$)
```

Correctness. The meaning of FindCutoff(A, *offset*) is to find the largest x such that the largest x elements of A are at least as large $10 \cdot (x + \mathit{offset})$. Then it is clear that FindCutoff(A, 0) gives our answer, and it remains to show the correctness of FindCutoff(A, *offset*). Correctness follows from an inductive argument on the length of A, which we provide a sketch of with some details omitted.

- **Base case**: When $n = 0$, correctness is clear.
- **Inductive hypothesis**: $\textsc{FindCutoff}(A[0..i-1], \mathit{offset})$ is correct for all $i < n$.
- **We want to prove**: $\textsc{FindCutoff}(A[0..n-1], \mathit{offset})$ is correct.
- **Proof / Inductive Step**: Let x be the largest number such that the largest x elements of A are at least as large $10 \cdot (x + \mathit{offset})$. Let m be the median of A, and L be the list of elements in A smaller than m, and R be the list of elements in A larger than m. We examine two cases.
  - $(x \geq 1 + \text{len}(R))$
    In this case we can express $x = x' + 1 + \text{len}(R)$ where $x' \geq 0$ counts the number of elements in L which are at least $10 \cdot (x + \mathit{offset})$. Expressed in terms of x', the largest x' elements in L are at least $10 \cdot (x' + (1 + \text{len}(R) + \mathit{offset}))$ and so the new offset should be set to $1 + \text{len}(R) + \mathit{offset}$. From the induction hypothesis we can then assert that calling $\textsc{FindCutoff}(L, 1 + \text{len}(R) + \mathit{offset}) + 1 + \text{len}(R)$ finds $x' + (1 + \text{len}(R)) = x$.

    The algorithm correctly recurses into a call to $\textsc{FindCutoff}(L, 1 + \text{len}(R) + \mathit{offset}) + 1 + \text{len}(R)$ since $m \geq 10 \cdot (x + \mathit{offset}) \geq 10 \cdot (1 + \text{len}(R) + \mathit{offset})$ in this case.
  - $(x < 1 + \text{len}(R))$
    This case is easier to reason about; we leave it to the curious reader.

Time complexity. The algorithm spends $O(n)$ time non-recursively, and then recurses on a list of size at most $n/2$. Thus, the recurrence satisfies $T(n) = T(n/2) + O(n)$ which solves to $T(n) = O(n)$ as we have seen before.

Problem 5 (Extra Credit)

Consider the following problem:

- INPUT: An array A of length n.
- OUTPUT: A pair of indices (i, j), with $j \geq i$ such that $\sum_{k=i}^{j} = A[i] + A[i+1] + ... + A[j] = 100$, or “no solution" if no such pair of indices exists. (If there exist many such pair of indices i, j, you only have to return one of them.)

Write pseudocode for an $O(n)$ time algorithm for this problem.

Note that for $i \leq j$, we have

$$\textsc{sum}(A[i..j]) = \textsc{sum}(A[0..j]) - \textsc{sum}(A[0..i-1])$$

where $\textsc{sum}(A[0..-1])$ is taken to be 0. Thus, finding a pair (i, j) such that $\textsc{sum}(A[i..j]) = 100$ is equivalent to finding a pair (i, j) such that

$$\textsc{sum}(A[0..j]) - 100 = \textsc{sum}(A[0..i-1]).$$

We can use dictionaries keyed on $\textsc{sum}(A[0..p])$, for all $p < j$ at time j, to efficiently check if $\textsc{sum}(A[0..j]) - 100$ had been seen (and, if so, get the index for which it had been seen).

```
Sum100($A[0..n-1]$):
    $D \leftarrow$ BuildDictionary()
    $D.add(0, 0)$
    $s \leftarrow 0$
    for $j \leftarrow 0$ to $n-1$
        $s \leftarrow s + A[j]$
        $D.add(s, j+1)$
        $i \leftarrow D.search(s - 100)$
        if $i \neq$ Nil
            return $(i, j)$
    return NoSolution
```

Correctness. Correctness follows from the discussion above.

Time complexity. The algorithm performs a loop with n iterations, with each iteration performing $O(1)$ operations in expectation. Thus, the total running time is $O(n)$ in expectation.