Math 590 Homework 5 Solutions Fall 2018

Due: Thursday, October 25

1. **Recurrence Relations**

You've decided to accept a job as an algorithm analyst at the social media company FacePage. One day, your boss Clark Hackerberg stops by with a problem: he has three different algorithms that all perform the same task, and he needs to choose which one to use.

- Algorithm A solves a problem of size n by dividing it into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time $O(n)$.
- Algorithm B solves a problem of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time $O(1)$.
- Algorithm C solves a problem of size n by dividing it into nine subproblems of size $n/3$, recursively solving each subproblem, then combining the solutions in quadratic time $O(n^2)$.

For each of these algorithms you should:

(a) Write down a recurrence relation for the runtime.

(b) Derive the runtime of the algorithm.

Which algorithm should your boss implement?

For algorithm A, the recurrence relation is

$$T(n) = 5T(n/2) + O(n).$$

This means that the table for our work tree is

node size	# of nodes	work/node	total work
$s = n$	1	$c \cdot s$	$c \cdot n$
$s = n/2$	5	$c \cdot s$	$5 \cdot c \cdot \frac{n}{2}$
$s = n/4 = \frac{n}{2^2}$	$25 = 5^2$	$c \cdot s$	$5^2 \cdot c \cdot \frac{n}{2^2}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

So our total work done is (assume that $n = 2^k$)

$$cn\left(1 + \frac{5}{2} + \left(\frac{5}{2}\right)^2 + \cdots + \left(\frac{5}{2}\right)^k\right) = cn\frac{\frac{5}{2}\left(\frac{5}{2}\right)^k - 1}{\frac{5}{2} - 1} = \frac{2cn}{3}\left(\frac{5}{2}\left(\frac{5}{2}\right)^{\log_2 n} - 1\right)$$

$$= \frac{2cn}{3}\left(\frac{5n^{\log_2 5}}{2n} - 1\right) = \frac{5cn^{\log_2 5}}{3} - \frac{2cv}{3} \in O(n^{\log_2 5}) \sim O(n^{2.32}).$$

For algorithm B, the recurrence relation is

$$T(n) = 2T(n-1) + O(1).$$

This means that the table for our work tree is

node size	# of nodes	work/node	total work
$s = n$	1	c	c
$s = n-1$	2	c	$2 \cdot c$
$s = n-2$	$4 = 2^2$	c	$2^2 \cdot c$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

So our total work done is

$$c\left(1 + 2 + 2^2 + \cdots + 2^{n-1}\right) = c\left(2^n - 1\right) \in O(2^n).$$

For algorithm C, the recurrence relation is

$$T(n) = 9T(n/3) + O(n^2).$$

This means that the table for our work tree is

node size	# of nodes	work/node	total work
$s = n$	1	$c \cdot s^2$	$c \cdot n^2$
$s = n/3$	9	$c \cdot s^2$	$9 \cdot c \cdot \left(\frac{n}{3}\right)^2$
$s = n/9 = \frac{n}{3^2}$	$81 = 9^2$	$c \cdot s^2$	$9^2 \cdot c \cdot \left(\frac{n}{3^2}\right)^2$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

We now make an interesting note about the work done at the $m-1$ level of the work tree:

$$9^m \cdot c \cdot \left(\frac{n}{3^m}\right)^2 = 9^m \cdot c \cdot \left(\frac{n^2}{(3^m)^2}\right) = 9^m \cdot c \cdot \left(\frac{n^2}{(3^2)^m}\right) = 9^m \cdot c \cdot \left(\frac{n^2}{9^m}\right) = c \cdot n^2.$$

Now if we assume that n is a power of 3, so that $n = 3^k$, then there will be k levels of the work tree, each requiring cn^2 work, for a total runtime

$$ckn^2 = c(\log_3 n)n^2 \in O(n^2 \log n).$$

Now we need to determine which algorithm is better. Clearly algorithm B is terrible because it has exponential running time. So, we are left to compare algorithm A that runs in $\sim O(n^{2.32})$, and algorithm C that runs in $O(n^2 \log n)$.

Which is larger, $n^{2.32}$ or $n^2 \log n$? Well, we can divide both by n^2, leaving us to compare $n^{0.32}$ and $\log n$. But, we know from the last homework that $\log n$ is dominated by **all** polynomial functions of n, i.e., $\log n \in o(n^p)$ for **any** positive value of p. Thus, the $\log n$ will result in a faster runtime.

Therefore algorithm C is the best with a runtime of $O(n^2 \log n)$.

2. **The Index Problem**

Let A be an array of n distinct integers where A is **already sorted** in ascending order. Our problem is to find an index i, $1 \leq i \leq n$, such that $A[i] = i$ or determine that no such i exists.

Describe an algorithm for this problem with $O(\log n)$ worst case running time. You should give the algorithm (in clear English or in clear high-level pseudo-code) and briefly explain why the running time is $O(\log n)$ in the worst case.

We first make two observations:

- If at an index k we see that $A[k] > k$, then we know that $A[m] > m$ for all $m > k$. This is because the integers in the array are distinct and sorted, so $A[k+1] \geq A[k] + 1 > k + 1$. So, it would only be possible for $A[i] = i$ if $i < k$.
- If at an index k we see that $A[k] < k$, then we know that $A[m] < m$ for all $m < k$. This is because the integers in the array are distinct and sorted, so $A[k-1] \leq A[k] - 1 < k - 1$. So, it would only be possible for $A[i] = i$ if $i > k$.

With these observations in hand, we create our algorithm similar to binary search:

- Compute the index k of the middle of the array (rounding down if odd length).
- If $A[k] = k$, then we have succeeded, return k.
- If $A[k] < k$, then recursively search the upper half of the array.
- If $A[k] > k$, then recursively search the lower half of the array.

Clearly at each level of recursion, we do at most 1 arithmetic operation (dividing length by 2), and 3 conditional checks, which is constant time. Furthermore, we make only one recursive call of size $n/2$, for a runtime $T(n) = T(n/2) + O(1)$. In the worst case, where we do not find an index i such that $A[i] = i$ (i.e., our algorithm is not able to stop early with a success), we will have at most $\log_2 n$ recursive calls (because we are dividing the array in 2 each time), where each recursive call is constant work. So our overall runtime is $O(\log n)$.

```
def findIndex(A, offset):
    if (length(A) == 1) AND (A[0] ≠ offset): return -1 # Failure
    k = ⌊length(A)/2⌋ # Compute half the length
    if A[k] == k + offset:
        return k+offset # Success
    if A[k] < k + offset:
        return findIndex(A[k+1:end], k+1+offset) # Search upper half of array
    if A[k] > k + offset:
        return findIndex(A[0:k-1], offset) # Search lower half of array
```

Note that this code keeps track of an offset. This is because the recursive calls only keep half of the array. So our index into the old array is the index in the new array plus the index of wherever the new array had come from in the old array (i.e., the offset). We make the call findIndex(A,0).

3. **Recursive Calls**[1]

Professor Mae Trix has devised an algorithm for computing the Trixian function on two $n \times n$ matrices. (Nevermind what that function does - we only care about the algorithm!)

(a) Prof. Trix's first attempt at her algorithm has a worst-case running time described by the recurrence relation:

$$\begin{aligned} T(1) &= c \\ T(n) &= 8T(n/2) + cn^4 \end{aligned}$$

What is the big-O asymptotic runtime of this algorithm as a function of n? Show your work.

The table for our work tree is

node size	# of nodes	work/node	total work
$s = n$	1	$c \cdot s^4$	$c \cdot n^4$
$s = n/2$	8	$c \cdot s^4$	$8 \cdot c \cdot \left(\frac{n}{2}\right)^4$
$s = n/4 = \frac{n}{2^2}$	$64 = 8^2$	$c \cdot s^4$	$8^2 \cdot c \cdot \left(\frac{n}{2^2}\right)^4$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

We now make an interesting note about the work done at the $m-1$ level of the work tree:

$$8^m \cdot c \cdot \left(\frac{n}{2^m}\right)^4 = 8^m \cdot c \cdot \left(\frac{n^4}{(2^m)^4}\right) = 8^m \cdot c \cdot \left(\frac{n^4}{(2^4)^m}\right) = 8^m \cdot c \cdot \left(\frac{n^4}{8^m \cdot 2^m}\right) = \frac{c \cdot n^4}{2^m}.$$

Now if we assume that n is a power of 2, so that $n = 2^k$, then the total runtime will be

$$cn^4\left(1 + \frac{1}{2} + \cdots + \frac{1}{2^k}\right) = cn^4 \frac{\frac{1}{2^{k+1}} - 1}{\frac{1}{2} - 1} = 2cn^4\left(1 - \frac{1}{2}\frac{1}{2^k}\right)$$

$$= 2cn^4\left(1 - \frac{1}{2n}\right) \in O(n^4).$$

(b) By using some clever tricks, Prof. Trix has removed one of the recursive calls and now has an algorithm with a worst-case runtime described by the recurrence relation:

$$\begin{aligned} T(1) &= d \\ T(n) &= 7T(n/2) + dn^4 \end{aligned}$$

What is the big-O asymptotic runtime of this algorithm as a function of n? Show your work.

The table for our work tree is

node size	# of nodes	work/node	total work
$s = n$	1	$d \cdot s^4$	$d \cdot n^4$
$s = n/2$	7	$d \cdot s^4$	$7 \cdot d \cdot \left(\frac{n}{2}\right)^4$
$s = n/4 = \frac{n}{2^2}$	$49 = 7^2$	$d \cdot s^4$	$7^2 \cdot d \cdot \left(\frac{n}{2^2}\right)^4$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

We now simply note that:

$$\left(\frac{1}{2^m}\right)^4 = \frac{1}{16^m}.$$

Now if we assume that n is a power of 2, so that $n = 2^k$, then the total runtime will be

$$dn^4\left(1 + \frac{7}{16} + \cdots + \frac{7^k}{16^k}\right) = dn^4 \frac{\left(\frac{7}{16}\right)^{k+1} - 1}{\frac{7}{16} - 1} = \frac{16dn^4}{9}\left(1 - \frac{7}{16}\left(\frac{7}{16}\right)^k\right)$$

$$= \frac{16dn^4}{9}\left(1 - \frac{7n^{\log_2 7}}{16n^{\log_2 16}}\right) \sim \frac{16dn^4}{9}\left(1 - \frac{7}{16n^{1.19}}\right) \in O(n^4).$$

(c) Is the second algorithm asymptotically better than the first in this case? Briefly, what do you think is the reason for this outcome?

Both algorithms have the same asymptotic runtime, $O(n^4)$. So neither is asymptotically better. Decreasing the number of recursive calls did not help with the runtime. This is because the bottleneck in the code was not the recursive calls. Instead, the bottleneck was the work done aside form the recursive calls. Note that for the single node of size n, the amount of work required was $O(n^4)$, even without counting any work done by the recursive calls. So it would have been impossible for the algorithm to run any faster than $O(n^4)$.

4. **Stooge Sort**[1]

Professors Curly, Mo, and Larry have proposed the following sorting algorithm:

- First sort the first two-thirds of the elements in the array.
- Next sort the last two-thirds of the elements in the array.
- Finally, sort the first two-thirds again.

The code is given below. Notice that the floor function, $\lfloor x \rfloor$, simply rounds down to the nearest integer. This is just used to compute the appropriate two-thirds and round to an integer so that we don't use non-integer indices into our array!

```
def Stooge-Sort(A, i, j):

    if A[i] > A[j]:
        swap A[i] and A[j]

    if i+1 >= j:
        return

    k = ⌊(j-i+1)/3⌋
    Stooge-Sort(A, i, j-k) # Sort the first two-thirds.
    Stooge-Sort(A, i+k, j) # Sort the last two-thirds.
    Stooge-Sort(A, i, j-k) # Sort the first two-thirds again!
```

(a) Give an informal but convincing explanation (not a rigorous proof by induction) of why the approach of sorting the first two-thirds of the array, then sorting the last two-thirds of the array, and then sorting again the first two-thirds of the array yields a sorted array. A few well-chosen sentences should suffice here.

Consider what happens when we sort the first two-thirds of the array. If an element k ended up in the first third of the array after this sort occurred, this means that k was smaller than at least one third of the array (because there were at least one third of the elements that were placed above it). This is important because it means that k could not possibly belong in the final third of the array (since there was at least a third of the array larger than it). But that means that after the first sort, all of the elements that belong in the final third of the array will have been placed somewhere in the last two-thirds of the array. When the second sort occurs, these elements then get properly placed into sorted order in the final third of the array, leaving only the first two-thirds potentially out of order. The final sort takes care of those elements, yielding a sorted array.

[1] Adapted from problem sets created by Harvey Mudd College CS Professor Ran Libeskind-Hadas.

(b) Find a recurrence relation for the worst-case runtime of Stooge-Sort. To simplify your recurrence relation, you may assume each of the recursive calls is on a portion of the array that is *exactly* two-thirds the length of the original array.

$$T(n) = 3T(2n/3) + c, \quad T(1) = c.$$

(c) Next, solve the recurrence relation using the work tree method. **Show all of your work**. In your analysis, it will be convenient to choose n to be a^k for some fixed constant a. (For example, we used $a = 2$ when analyzing the multiplication problem or Mergesort in class. Here you will want to use a different value of a. The value of a that you choose might not even be an integer! As we've seen in class, this is valid and allows us to significantly simplify the analysis.)

The table for our work tree is

node size	# of nodes	work/node	total work
$s = n$	1	c	c
$s = 2n/3$	3	c	$3c$
$s = 4n/9 = n\,(2/3)^2$	$9 = 3^2$	c	3^2c
$\cdots$	$\cdots$	$\cdots$	$\cdots$

We will now make the assumption that n is a power of $3/2$ to help with our analysis, i.e., we are assuming that $n = \left(\frac{3}{2}\right)^k$. This means that the work tree will have $k + 1$ levels, and so the total work done is

$$c(1 + 3 + 3^2 + \cdots + 3^k) = c\frac{3^{k+1} - 1}{3 - 1} = \frac{c}{2}(3 \cdot 3^k - 1) = \frac{c}{2}(3 \cdot 3^{\log_{3/2} n} - 1)$$

$$= \frac{c}{2}(3 \cdot n^{\log_{3/2} 3} - 1) \in O(n^{\log_{3/2} 3}) \sim O(n^{2.71}).$$

(d) How does the worst-case runtime of Stooge-Sort compare with the worst-case runtime of the other sorting algorithms that we've seen so far?

The very naive Bubble sort, the worst of the sorting algorithms we've seen so far, had a runtime of $O(n^2)$. Stooge-sort is even worse, and is therefore a bad algorithm for sorting.