Homework 3

ComS 311

Summer 2019

1 Warm-ups

1. This question is about properties of the following sorting algorithm.

```
while Array is not sorted do
    for j from 1 to n-1 do
        if $A[j] < A[j+1]$ then
            Swap $A[j]$ and $A[j+1]$
        end if
    end for
end while
```

   (a) How long do the subroutine calls take, in big-O terms? Specifically, how long does swapping two array indices take? How long does it take to check if an array is sorted?

   Solution: Swapping can be done in constant time: $temp = A[j], A[j] = A[j+1], A[j+1] = temp$. Checking if an array is sorted can be done with a linear scan through the array.

   (b) What is the fastest possible runtime for this algorithm? What input gives the fastest runtime?

   Solution: If the array is already sorted, the algorithm will check if it's sorted ($O(n)$) and then not enter the while loop.

   (c) What is the slowest possible runtime? What input gives the slowest runtime?

   Solution: Any array in which the smallest element is in the last position will take $O(n^2)$ time, because each pass through the array will move that element one space earlier in the array. So, to get it to $A[1]$, it will take n passes, and each takes $O(n)$ time.

2. Here's a recursive algorithm and a walkthrough of how recurrences can be used to analyze the runtime of recursive algorithms.

```
function DEMO(integer $n$)
    if $n = 1$ then
        return
    else
        Waste $n^2$ time
        Demo($n/2$)
        Demo($n/2$)
        Demo($n/3$)
    end if
end function
```

Writing out a recurrence just involves counting how long the program takes on a particular input, how many recursive calls it makes, how big the input to each recursive call is, and how long the base case is.

We use $T(n)$ to represent the time taken when the input is size n. In this case, that's the time required to waste $O(n^2)$ time, then make three recursive calls with smaller inputs. We don't know how long those recursive calls take, but we can write each of them using T.

(a) Write $T(n)$ for this algorithm.
Solution: $T(n) = 2T(\frac{n}{2}) + T(n/3) + n^2$

(b) We also need to count the time taken by the base case. What input size is the base case? How much time does the base case take?
Solution: Base case is when $n = 1$. The time taken is constant. $T(1) = 1$. (Or $T(1) = c$ - it usually doesn't matter if you use 1 or c.)

2 Recurrences

Solve the following recurrences.

1. $T(n) = 2T(n/2) + 2n$, and $T(1) = 1$

Answer 1: $T(n/2) = 2T(n/4) + 2(n/2)$. Substitute this in: $T(n) = 2(2T(n/4) + 2(n/2)) + 2n = 4T(n/4) + 2n + 2n$. Keep doing these substitutions and you'll get $T(n) = 2^i T(n/2^i) + i * 2n$. When $i = \log n$, this is $T(n) = nT(n/n) + 2n\log n = nT(1) + 2n\log n = n + 2n\log n = O(n\log n)$.

Answer 2: Guess $T(n) \leq cn\log n$(for some constant c)

- Base case: we need to show that our guess holds for some base case (when n is small) and $T(n) = O(n\log n)$ holds when n=2.
- Assume it holds for $n/2$, that is $T(n/2) \leq c\frac{n}{2}\log\frac{n}{2}$. Prove it holds for n, i.e., $T(n) \leq cn\log n$

$$\begin{aligned}
T(n) &= 2T(n/2) + 2n \\
&\leq 2(c\frac{n}{2}\log\frac{n}{2}) + 2n \\
&= cn\log\frac{n}{2} + 2n \\
&= cn\log n - cn + 2n
\end{aligned}$$

It holds when $c \geq 2$.

Therefore $T(n) = O(n\log n)$.

2. $T(n) = T(n/3) + T(n/4) + 1$, and $T(1) = 1$

   Answer: $T(n/3) > T(n/4)$ (we won't prove this, just trust me.) So $T(n) < T(n/3) + T(n/3) + 1 = 2T(n/3) + 1$. Solve $T(n) = 2T(n/3) + 1$ and you get $T(n) = 2^i T(n/3^i) + 1 * i$. When $i = \log_3(n)$, we get $T(n) = 2^{\log_3(n)} T(1) + \log_3(n) = n^{\log_3(2)} + \log_3(n)$.

   This is not a tight upper bound - the tight upper bound is somewhere between $n^{\log_3(n)}$ and $n^{\log_4(n)}$.

3. $T(n) = T(n/3) + T(n/4) + 1$, and $T(1) = n$

   Follow the same derivation, but instead of replacing $T(1)$ with 1, replace $T(1)$ with n. You get $T(n) = 2^{\log_3(n)} n + \log_3(n) = n^{\log_3(2)+1} + \log_3(n)$.

4. **Answer:** Expanding out the recurrence gives a pattern that has two terms: $T(n) = T(n/2) + 2T(n/4) + \log n$
   $T(n/2) = T(n/4) + 2T(n/8) + \log(n/2)$
   $T(n/4) = T(n/8) + 2T(n/16) + \log(n/4)$
   $T(n/8) = T(n/16) + 2T(n/32) + \log(n/8)$
   Substituting (and using $\log(n/2) = \log n - 1)$:
   $T(n) = 3T(n/4) + 2T(n/8) + \log n + \log n - 1$
   $T(n) = 5T(n/8) + 6T(n/16) + 3\log n - 1 - 2$
   $T(n) = 11T(n/8) + 10T(n/32) + 3\log n - 1 - 2$
   Call the sequence of coefficients a_i and b_i. $a_1 = 1$ and $b_1 = 2$. The pattern is $b_i = 2a_{i-1}$ and $a_i = a_{i-1} + b_{i-1}$. Combine these into a function for just a_i : $b_{i-1} = 2a_{i-2}$, so $a_i = a_{i-1} + 2a_{i-2}$. There's a closed form solution to this recurrence: $a_i = (1/3)((-1)^i + 2^{i+1})$.

   So we get $T(n) = a_i T(n/2^i) + b_i T(n/2^{i+1}) + a_{i-1}(\log(n) - i) + a_{i-2}(\log(n) - (i-1)) + \cdots + a_1 \log(n)$. Since $b_i = 2a_{i-1}$, $T(n) = a_i T(n/2^i) + a_{i-1} T(n/2^{i+1}) + a_{i-1}(\log(n) - i) + a_{i-2}(\log(n) - (i-1)) + \cdots + a_1 \log(n)$

   When $i = \log n - 1$, we get
   $T(n) = a_{\log n - 1} T(2) + a_{\log n} T(1) + a_{\log n - 2}(\log n - (\log n - 1)) + a_{\log n - 3}(\log(n) - (\log n - 1 - 1)) + \cdots + a_1 \log(n)$
   $= a_{\log n - 1} T(2) + a_{\log n} T(1) + a_{\log n - 2} * 1 + a_{\log n - 3} * 2) + \cdots + a_1 * \log(n)$

   a_i is $2^i * 2/3 \pm 1/3$, so we can write the sum $\sum_{j=1}^{\log n - 1} a_{\log n - j - 1} * j$ as roughly $2/3 \sum_{j=1}^{\log n} j 2^j$, which is $O(\log n \log\log n)$, a fact which I won't prove because I'm so done with this problem by now.

5. $T(n) = T(n-1) + 1$, and $T(1) = 1$

Answer:

$$\begin{aligned} T(n) &= T(n-1)+1 \\ &= (T(n-2)+1)+1 \\ &= T(n-2)+2 \\ &= T(n-k)+k \end{aligned}$$

Set $k = n-1$, then we have $T(n) = T(1)+n-1 = n$. Hence $T(n) = O(n)$.

6. $T(n) = 2T(n-1)+1$, and $T(1) = 1$

   Answer:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2(2T(n-2)+1)+1 \\ &= 2(2(2T(n-3)+1)+1)+1 \\ &= \dots \\ &= 2^rT(n-r)+\sum_{i=0}^{r-1} 2^i \\ &= 2^rT(n-r)+2^r-1 \end{aligned}$$

   Setting $r = n-1$, we can have the following

$$\begin{aligned} T(n) &= 2^{n-1}T(1)+2^{n-1}-1 \\ &= O(2^n) \end{aligned}$$

7. $T(n) = 2T(\sqrt{n}) + \log n$, and $T(2) = \log n$

   Answer: Write $\sqrt{n}$ as $n^{\frac{1}{2}}$. $T(n^{\frac{1}{2}}) = 2T(n^{\frac{1}{4}})+\log(n^{\frac{1}{2}})$. $T(n^{\frac{1}{4}}) = 2T(n^{\frac{1}{8}})+\log(n^{\frac{1}{4}})$. So,

$$\begin{aligned} T(n) &= 2T(n^{\frac{1}{2}}) + \log n \\ &= 2\left(2T(n^{\frac{1}{4}}) + \frac{1}{2}\log n\right) + \log n \\ &= 4T(n^{\frac{1}{4}}) + 2\log n \\ &= 4\left(2T(n^{\frac{1}{8}}) + \frac{1}{4}\log n\right) + 2\log n \\ &= 8T(n^{\frac{1}{8}}) + 3\log n \end{aligned}$$

   The pattern is $T(n) = 2^iT(n^{\frac{1}{2^i}}) + i\log n$. Now we need to figure out which value of i gets $T(n^{\frac{1}{2^i}})$ to reach the base case. That is, for which i is $n^{\frac{1}{2^i}} = 2$.

Take the log of both sides: $\frac{1}{2^i}\mathrm{log}n = 1$

Multiply by 2^i : $\mathrm{log}n = 2^i$

Take logs again: $\mathrm{loglog}n = i$.

So when $i = \mathrm{loglog}n$, we have $T(n) = 2^{\mathrm{loglog}n}T(n^{\frac{1}{2^{\mathrm{loglog}n}}} + \mathrm{loglog}n\mathrm{log}n = \mathrm{log}nT(2) + \mathrm{loglog}n\mathrm{log}n = \mathrm{log}^2n + \mathrm{log}n\mathrm{loglog}n$. The larger term is log^2n, so this is $O(\mathrm{log}^2n)$.

3 Algorithm design

For each problem, write pseudocode, derive the recurrence, analyze the runtime, and prove correctness.

1. Input: a binary tree implemented as a linked list (you start with a pointer to the root node, and each node has 0, 1, or 2 children; your only way to interact with the tree is to follow the child pointers). Design a divide-and-conquer algorithm that counts the number of nodes in the tree. Target runtime: $O(n)$

Algorithm

```
TreeSize(T, r)
{
Input: a binary tree T, a pointer r to the root node of T
Output: the number of nodes in T
If r == null
        return 0;
else
        return 1 + TreeSize(T, r.leftChild) + TreeSize(T, r.rightChild);
}
```

Runtime Analysis:
The algorithm is very similar to a pre-order traversal. $T(0) = T(1) = 1$ as the base case where the tree is empty takes constant time.
The recurrence relation for this algorithm is $T(n) = T(|L|) + T(|R|) + 1$ with $|L| + |R| + 1 = n$ as the conquer part solves two subproblems and addition takes constant time.
The algorithm visits the root node, every node in left subtree, and every node in right subtree, ant it visits each node exactly once by following the link to reach the node, resulting in runtime of $O(n)$.

Proof of correctness:

Proof. Let T be a binary tree with r as the root node and L as the left subtree and R as the right subtree. We use structural induction to prove that the algorithm output the correct number of nodes in T. Let n be the

number of nodes in T.
Base case: When T is empty, the number of nodes in T should be 0. This is the same as the output of the algorithm in the *if* part.
Inductive Hypothesis: For all $n \geq 1$, assume that the algorithm correctly computed the number of nodes in L(denote as L(n)) and in R(denote as R(n)).
Inductive Step: We show that the algorithm correctly computed the number of nodes in T for all $n \geq 1$. T consists of the root node, the nodes in the left subtree L, and the nodes in the right subtree R. So the number of nodes in T should be $1 + L(n) + R(n)$,by IH, which is the same as the output of the algorithm in the recursive part.
Thus the algorithm correctly computed the number of nodes in T for all $n \geq 0$. □

2. Input: a sorted array of integers, and a target number k. Design a divide and conquer algorithm that outputs true if k is in the array, and false if not. Target runtime: $O(\log n)$

Algorithm

```
BinarySearch(A, k):
{
Input: a sorted array A of n integers($n \geq 0$), a target number k
Output: True if k is in A, false otherwise
//base case
If n == 0:
        return False;
//recursive part
int middle = $\frac{n}{2}$;
If A[middle] == $k$
        return True;
Else if (A[middle] $> k$)
        return BinarySearch(A[0, middle - 1], k);
Else:
        return BinarySearch(A[middle + 1, n - 1], k);
}
```

Runtime Analysis:
The recurrence relation for this algorithm is $T(n) = T(\frac{n}{2}) + 1$ as the conquer part solves only one subproblem(only one of the else part is executed) and comparison of A[middle] and k takes constant time, and $T(0) = T(1) = 1$ as the base case where the array is empty takes constant time. Solving the recurrence gives $T(n) = O(\log n)$ as the runtime of the algorithm.

Proof of correctness:

Proof. Let n be the number of integers in A.
Base case: When A is empty, whatever value of k is, k is not in A. This is the same as the output of the algorithm in the first *if* part.
Inductive Hypothesis: For all $n \geq 1$, assume that the algorithm correctly computed whether k is the middle element of A, or whether k is in the first half of A, or whether k is in the second half of A.
Inductive Step: We show that the algorithm correctly computed whether k is in A for all $n \geq 1$. A consists of the middle element, the first half of A(all elements from A[0] to A[middle - 1]), and the second half of A(all elements from A[middle + 1] to A[n - 1]). If $A[middle] = k$ then k is in A and the algorithm correctly returned true. Since A is sorted, if $A[middle] > k$, then k is not possible in the second half of A, and is only possible in the first half of A. By IH, the algorithm correctly computed whether k is in the first half of A. On the other hand, if $A[middle] < k$, then k is not possible in the first half of A, and is only possible in the second half of A. By IH, the algorithm correctly computed whether k is in the second half of A.
Thus the algorithm correctly computed whether k is in A for all $n \geq 0$. □

3. Input: an array of integer pairs $< a, b >$. Design a divide-and-conquer algorithm that computes, for each pair $< x, y >$, the number of pairs $< u, v >$ where $u < x$ and $v < y$. Target runtime: $O(n \log n)$

Algorithm: Before we run the algorithm, we sort the points by x coordinate, in increasing order.

```
function Rank(array $A$ of size $n$)
    if $n = 1$ then
        $A[1].rank = 0$ return $A$
    end if
    Left = Rank($A[1 \dots n/2]$)
    Right = Rank($A[n/2 + 1 \dots n]$)
      ▷ Left and Right are sorted by $y$, since we return lists sorted by $y$
    $i = 1$
    $j = 1$
    while $i < n/2$ and $j < n/2$ do
        if $Left[j].y \geq Right[i].y$ then
            $Right[i].rank += j - 1$
            $i++$
        else
            $j++$
        end if
    end while
    while $i < n/2$ do      ▷ all remaining elements are larger than all of Left
```

```
        $Right[i].rank+ = n/2$
    **end while**
    Merge Left and Right by $y-$coordinate in increasing order (using
the combine function from mergesort) **return** the merged array
**end function**
```

Runtime: Two recursive calls with half the input size. The additional work is scanning through the list ($O(n)$), doing a constant time comparison and some arithmetic for each element ($O(1) * n$), and then merging the two lists ($O(n)$). The recurrence is $T(n) = 2T(n/2) + O(n)$, which is the same as the mergesort recurrence, so $T(n) = O(nlogn)$.

Proof of correctness: First, we know that the arrays Left and Right are correctly sorted by y coordinate, because we're using the combine algorithm from mergesort, which we already proved correct.

Now we prove that the ranks are correctly computed for each subarray. Base case is an array of size 1. There's only one pair, and it has rank 0 because there's no other pairs to compare to. Assume as our IH that the ranks are correctly computed within the Left and Right subarrays. We need to prove that we correctly update the ranks for the combined array.

First, we don't have to update any pairs in Right, because all pairs in Right have smaller x coordinates than the pairs in Left. (This is because of how we split the arrays.)

Second, when we update a pair in Left, we need to know how many pairs have smaller y coordinates. We walk through Right until we reach a pair with larger y coordinate than the current Left pair. Since we return arrays sorted by y coordinate, all previous pairs in Right have smaller y coordinate, so we can just increase the rank by the current index of j (minus 1 because the current pair doesn't count). Since Left is also sorted by y, all the Right pairs less than the current pair are also less than the next pair. Therefore, this correctly computes the rank.

4. Input: two really big integers a and b, so big that arithmetic operations are no longer constant time. (You may assume addition is $O(m)$, where a and b are $\leq m$ bits.) Design a divide-and-conquer algorithm that computes $a * b$.

   Lots of hints: start with two-digit numbers. Use the following ideas: you can write e.g. 16 as 1 * 10 + 6, or 93 as 9 * 10 + 3. If you write a as $10x_1+x_2$ and b as $10y_1+y_2$, what is $a*b$? Try to write it as $100z_1+10z_2+z_3$. Can you compute z_1, z_2, and z_3 using only 3 multiplications (and maybe some addition)?

```
**function** MULTIPLY($n$-digit integers $x, y$)
    **if** $n = 1$ **then**
        Do basic integer multiplication
    **end if**
    Write $x$ as $10^{n/2}a + c$
    Write $y$ as $10^{n/2}b + d$
    I = Multiply$(a, b)$
    J = Multiply$(c, d)$
    K = Multiply$(a + c, b + d)$
    **return** $I * 10^n + (K - J - I) * 10^{n/2} + J$
**end function**
```

First a note: for computers, the 10 would be replaced with a 2, because multiplying by powers of 2 can be done by bitshifting. For humans, powers of 10 are easy to bitshift. So if you're trying to follow this algorithm, use the 10s, if you're trying to implement it use the 2s.

Three recursive calls with half the input size. Other work: a bunch of addition and bitshifting, both of which take $O(n)$ work. Recurrence: $T(n) = 3T(n/2) + n$. Single-digit integer multiplication is $O(1)$, so $T(1) = 1$. Solving this gets you $O(n^{\log_2(3)})$.

Correctness: Base case is obviously correct. Now assume the recursive calls return the correct results: $I = ab$, $J = cd$, and $K = ab + ad + bc + cd$ so $K - J - I = ad + bc$. We want to compute $(10^{n/2}a + c) * (10^{n/2}b + d)$. FOIL this and you get $10^n ab + 10^{n/2}ad + 10^{n/2}bc + cd$, which is exactly what we return.