Section 07: Solutions

Section Problems

1. Sorting

(a) Demonstrate how you would use quick sort to sort the following array of integers. Use the first index as the pivot; show each partition and swap.

[6, 3, 2, 5, 1, 7, 4, 0]

Solution:

[Solutions omitted]

(b) Show how you would use merge sort to sort the same array of integers.

Solution:

[Solutions omitted]

(c) Show how you would use selection sort to sort the same array of integers.

Solution:

[Solutions omitted]

(d) Suppose we have an array where we expect the majority of elements to be sorted "almost in order". What would be a good sorting algorithm to use?

Solution:

Merge sort and quick sort are always predictable standbys, but we may be able to get better results if we try using something like insertion sort, which is $\mathcal{O}(n)$ in the best case.

Alternatively, we could try using an adaptive sort such as Timsort, which is specifically designed to handle almost-sorted inputs efficiently while still having a worst-case $\mathcal{O}(n \log(n))$ runtime.

2. In-Depth Recurrence

Consider the following recurrence: $A(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 3A(n/6) + n & \text{otherwise} \end{cases}$

We want to find an *exact* closed form of this equation by using the tree method.

(a) Suppose we draw out the total work done by this method as a tree, as discussed in lecture. Including the base case, how many levels in total are there?

Your answer should be a mathematical expression which uses the variable n, which represents the original number we passed into $A(n)$.

Solution:

Counting the base level, there are $\log_6(n) + 1$ levels in total.

(b) How many nodes are there on any given level i? Your answer should be a mathematical expression that uses the variable i.

Note: let $i = 0$ indicate the level corresponding to the root node. So, when $i = 0$, your expression should be equal to 1.

Solution:

There are 3^i nodes on level i.

(c) How much work is done on the i-th *recursive* level? Your answer should be a mathematical expression that uses the variables i and n.

Solution:

We do $3^i \cdot \frac{n}{6^i} = (\frac{3}{6})^i n$ work on level i.

Note: I know $\frac{3}{6} = \frac{1}{2}$. Hold your horses. It will be nicer for us if we leave it this way for now.

(d) How much work is done on the final, *non-recursive* level? Your answer should be a mathematical expression that uses the variable n.

Solution:

We do $1 \cdot 3^{\log_6(n)} = n^{log_6(3)}$ total work at the base case level.

(e) What is the closed form of this recurrence? Be sure to show your work.

Note: you do not need to simplify your answer, once you found the closed form. Hint: You should use the finite geometric series identity somewhere while finding a closed form.

Solution:

We combine all the pieces and simplify:

$$A(n) = \left(\sum_{i=0}^{\log_6(n)-1} (\frac{3}{6})^i n \right) + n^{\log_6(3)}$$

$$\sum_{i=0}^{\log_6(n)-1} (\frac{3}{6})^i n = n \sum_{i=0}^{log_6(n)-1} (\frac{1}{2})^i$$

Side note: Yes, it says $\log_6(n) - 1$ and not $\log_6(n) + 1$. This is because we pulled out one iteration for the base case, and started at 0, so we have to "subtract 2" from the upper bound.

After applying the finite geometric series identity, we get:

$$A(n) = n \cdot \frac{\left(\frac{3}{6}\right)^{\log_6(n)} - 1}{\frac{1}{2} - 1} + n^{\log_6(3)}$$

You don't have to simplify further, but if you were to simplify, you would get:

$$\begin{aligned}
A(n) &= n \cdot \frac{\left(\frac{3}{6}\right)^{\log_6(n)} - 1}{\frac{1}{2} - 1} + n^{\log_6(3)} \\
&= -2n \cdot \left(\left(\frac{3}{6} \right)^{\log_6(n)} - 1 \right) + n^{\log_6(3)} \\
&= -2n \cdot \left(n^{\log_6(3/6)} - 1 \right) + n^{\log_6(3)} \\
n^{\log_6(3/6)} &= n^{\log_6(3) - \log_6(6)} \\
&= n^{\log_6(3) - 1} \\
&= \frac{n^{\log_6(3)}}{n} \\
A(n) &= -2n^{\log_6(3)} + 2n + n^{\log_6(3)} \\
&= 2n - n^{\log_6(3)}
\end{aligned}$$

(f) Use the master theorem to find a big-Θ bound of $A(n)$.

Solution:

$T(n) \in \Theta(n)$.

3. Recurrences

For each of the following recurrences, find their closed form using the tree method. Then, check your answer using the master method (if applicable). It may be a useful guide to use the steps from part 2 of this handout to help you with all the parts of solving a recurrence problem fully.

(a) $J(k) = \begin{cases} 1 & \text{if } k = 1 \\ 5J(k/5) + k^3 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 5^i
- Total work done per level: $5^i \cdot \frac{k^3}{5^{3i}} = 5^i \cdot \frac{k^3}{125^i}$
- Total number of *recursive* levels: $\log_5(k)$
- Total work done in base case: $5^{\log_5(k)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_5(k)-1} 5^i \cdot \frac{k^3}{125^i}\right) + 5^{\log_5(k)}$$

We apply the finite geometric series to get:

$$k^3 \frac{\left(\frac{5}{125}\right)^{\log_5(k)} - 1}{\frac{5}{125} - 1} + 5^{\log_5(k)}$$

If we wanted to simplify, we'd get:

$$\frac{25k^3}{24} - \frac{k}{24}$$

We can also apply the master theorem here. Note that $\log_b(a) = \log_5(5) = 1 < 3 = c$, so we know $J(k) \in \Theta\left(k^3\right)$.

(b) $S(q) = \begin{cases} 1 & \text{if } q = 1 \\ 2S(q-1) + 1 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 2^i
- Total work done per level: $2^i \cdot 1$
- Total number of *recursive* levels: $q - 1$
- Total work done in base case: 2^{q-1}

Note that these expressions look a little different from the ones we've seen up above. This is because we aren't *dividing* our terms by some constant factor – instead, we're *subtracting* them.

So we get the expression:

$$\left(\sum_{i=0}^{q-1-1} 2^i \right) + 2^{q-1}$$

We apply the finite geometric series to get:

$$\frac{2^{q-1} - 1}{2 - 1} + 2^{q-1}$$

If we wanted to simplify, we'd get:

$$2^q - 1$$

Note that we may NOT apply the master theorem here – our original recurrence doesn't match the form given in the theorem.

(c) $Z(x) = \begin{cases} \log(x) & \text{if } x = 7 \\ 3Z(x/3) + 1 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 3^i
- Total work done per level: 3^i
- Total number of *recursive* levels: $\log_3(x) - 6$
- Total work done in base case: $\log_2(7) \cdot 3^{\log_3(x)-6}$

Note that the height here is different, since the recursive function hits the base case when $i = 7$.

So we get the expression:

$$\left(\sum_{i=0}^{\log_3(x)-6-1} 3^i \right) + 3^{\log_3(x)-6}$$

We apply the finite geometric series to get:

$$\frac{3^{\log_3(x)-6} - 1}{3 - 1} + 3^{\log_3(x)-6}$$

If we wanted to simplify, we'd get:

$$\frac{x}{2 \cdot 3^5} - \frac{1}{2}$$

However, our closed form only holds when $x > 7$ – our recurrence doesn't define what happens if x is less then that.

Initially, it doesn't seem like we can apply the master theorem because the base case doesn't match the exact form of the theorem.

However, since $x = 7$ in the base case, $\log(x)$ always ends up being a constant, so it actually works out.

Note that $\log_b(a) = \log_3(3) = 1 > 0 = c$, so we know $Z(x) \in \Theta\left(n^{\log_b(a)}\right)$. This ends up being $Z(x) \in \Theta\left(n^{\log_3(3)}\right)$ which simplifies to just $Z(x) \in \Theta\left(n\right)$.

(d) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 3 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: $1^i = 1$
- Total work done per level: $1 \cdot 3 = 3$
- Total number of *recursive* levels: $\log_2(n)$
- Total work done in base case: $1^{\log_3(n)} = 1$

So we get the expression:

$$\left(\sum_{i=0}^{\log_2(n)-1} 3 \right) + 1$$

Using the summation of a constant identity, we get:

$$3\log_2(n) + 1$$

We can apply the master thereom here. Note that $\log_b(a) = \log_2(1) = 0 = c$, which means that $T(n) \in \Theta\left(n^c \log(n)\right)$ which is $T(n) \in \Theta\left(n^0 \log(n)\right)$ which further simplifies to $T(n) \in \Theta\left(\log(n)\right)$.

This agrees with our simplified form.

4. Divide and conquer

(a) Suppose we have an array of sorted integers that has been *circularly shifted* k positions to the right. For example, $[35, 42, 5, 10, 20, 30]$ is a sorted array that has been circularly shifted $k = 2$ positions, while $[27, 29, 35, 42, 5, 9]$ is a sorted array that has been shifted $k = 4$ positions.

Now, suppose you are given a sorted array that has been shifted an unknown number of times – we do not know what k is.

Describe how you would implement an algorithm to find k in $\mathcal{O}(\log(n))$ time.

Solution:

First, notice that the smallest element in our array will tell us precisely how many times the array has been shifted, since the smallest element should go in index 0. This is equivalent to looking for an element in our array which is smaller than its neighbor to the left. Additionally, notice that given two elements on the circular sorted array, the smallest element in the array (if it isn't one of the selected elements) cannot be to the right of the smaller element and left of the larger. Therefore, we know it must exist on the other part of the array.

Algorithm:

Begin by choosing an arbitrary element of the list - for the purpose of this example, we'll pick the first element. Then, if we did not find the smallest element (by checking the element to the left of it), we can check the $\frac{n}{2}$th element in the list. If that element is greater than the first element, then we know that the smallest element must be in the second-half of the array. If that element is less than the first element, then we know that the smallest element in the array must be in the first-half of the array. In any case, we know that the half-array must still be a circular sorted array, since we only remove elements, which cannot break the sortedness of an array.

Therefore, we can recurse on this half array, guaranteeing that the smallest element overall is on the half array. When the array size is 1, we certainly must have found the smallest element.

(b) Suppose we have some Java method `double foo(int n)`. This function is *monotonically decreasing* – this means that as we keep plugging in larger and larger values of `n`, the `foo(...)` method will keep returning smaller and smaller numbers.

More specifically, for any integer `i`, it is always true that `foo(i) > foo(i + 1)`.

We want to find the smallest value of `n` that when plugged in will make `foo(...)` return a negative number.

Describe how you would implement a $\mathcal{O}(\log(n))$ algorithm to do this (where n is the final answer).

Solution:

Algorithm:

Check `foo(`2^k`)`, incrementing k until the output of the function is negative.

This gives us bounds on n, specifically that $2^{c-1} < n \leq 2^c$ for whichever c we end up on (which is also approximately $\log(n)$). Finally, we can perform a binary search over the elements between 2^{c-1} and 2^c to find the exact value of n.

Ultimately, we first do c checks to find the c, then do $c-1$ more checks binary search over the $2^c - 2^{c-1} = 2^{c-1}$ remaining elements on the range. This results in $\mathcal{O}(c)$ checks overall, which is on order of $\mathcal{O}(\log(n))$ as desired.

(c) Describe how you would modify merge sort so that it can sort a singly linked list in $\mathcal{O}(n \log(n))$ time. Your algorithm should modify the linked list in place, without needed extra data structures.

Solution:

We first split the array by obtaining a pointer to the middle of the linked list, then splitting it in two.

We can find the middle in one of two ways. One way is to have the linked list always maintain its size and loop size/2 times. The other way is to have two pointers. Both pointers begin at the start, but one moves two spaces per iteration and the other moves only one. Once the "move-two-spaces" pointer reaches the end of the linked list, we know the first pointer must be in the middle.

Either way, this takes $\mathcal{O}(n)$ time since we need to examine $\frac{n}{2}$ nodes. Once we have the middle pointer, split (which takes $\mathcal{O}(1)$ time) and recurse.

We can then merge using nearly the identical algorithm we had for arrays in $\mathcal{O}(n)$ time.

We end up with the exact same recurrence as the original version of merge sort, which means the runtime must be $\mathcal{O}(n \log(n))$.

(d) Describe how you would modify your answer from the previous question to randomly shuffle a linked list in $\mathcal{O}(n \log(n))$ time. As before, your algorithm should modify the linked list in place, again without needing any extra data structures.

Solution:

Modify the merge step so that instead of moving the smallest item from the two linked lists to the combined one, pick them at random.

So, instead of combining two sorted lists into a bigger sorted list, our `merge(a, b)` algorithm will instead take two (randomized) lists and produce another randomized list.

Challenge Problems

5. Recurrences

For each of the following recurrences, find their closed form using the tree method. Then, check your answer using the master method (if applicable).

(a) $Y(q) = \begin{cases} 1 & \text{if } q = 1 \\ 8T(q/2) + q^3 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 8^i
- Total work done per level: $8^i \cdot \frac{q^3}{2^{3i}} = 8^i \cdot \frac{q^3}{8^i}$
- Total number of *recursive* levels: $\log_2(q)$
- Total work done in base case: $8^{\log_2(q)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_2(q)-1} 8^i \cdot \frac{q^3}{8^i} \right) + 8^{\log_2(q)}$$

While we could apply the finite geometric series identity here, there's actually a simpler approach. Notice that the 8^i term cancels itself out. So, we're left with:

$$\left(\sum_{i=0}^{\log_2(q)-1} q^3 \right) + 8^{\log_2(q)}$$

We can then apply the "summation of a constant" identity to get:

$$q^3 \log_2(q) + 8^{\log_2(q)}$$

We can also apply the master theorem here. Note that $\log_b(a) = \log_2(8) = 3 = c$, so we know $Y(q) \in \Theta\left(q^3 \log(q)\right)$.

(b) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 8T(n/2) + 4n^2 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 8^i
- Total work done per level: $8^i \cdot 4\left(\frac{n}{2^i}\right)^2 = 8^i \cdot 4 \cdot \frac{n^2}{4^i}$
- Total number of *recursive* levels: $\log_2(n)$
- Total work done in base case: $8^{\log_2(n)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_2(n)-1} 8^i \cdot 4 \cdot \frac{n^2}{4^i}\right) + 8^{\log_2(n)}$$

We can simplify by pulling the $4n^2$ out of the summation:

$$4n^2 \left(\sum_{i=0}^{\log_2(n)-1} \frac{8^i}{4^i}\right) + 8^{\log_2(n)}$$

This further simplifies to:

$$4n^2 \left(\sum_{i=0}^{\log_2(n)-1} 2^i\right) + 8^{\log_2(n)}$$

After applying the finite geometric series identity, we get:

$$4n^2 \cdot \frac{2^{\log_2(n)} - 1}{2 - 1} + 8^{\log_2(n)}$$

This is a closed form so we could stop, but if we want a tidy solution, we can continue simplifying:

$$\begin{aligned} T(n) &= 4n^2 \cdot \frac{2^{\log_2(n)} - 1}{2 - 1} + 8^{\log_2 2(n)} \\ &= 4n^2 \cdot \left(2^{\log_2(n)} - 1\right) + 8^{\log_2(n)} \\ &= 4n^2 \cdot \left(n^{\log_2(2)} - 1\right) + n^{\log_2(8)} \\ &= 4n^2 \cdot (n - 1) + n^3 \\ &= 5n^3 - 4n^2 \end{aligned}$$

We can apply the master thereom here. Note that $log_b(a) = \log_2(8) = 3 > 2 = c$, which means that $T(n) \in \Theta\left(n^{\log_b(a)}\right)$ which is $T(n) \in \Theta\left(n^{\log_2 8}\right)$ which in turn simplifies to $T(n) \in \Theta\left(n^3\right)$.

This agrees with our simplified form.

(c) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7T(n/3) + 18n^2 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 7^i
- Total work done per level: $7^i \cdot 18\left(\frac{n}{3^i}\right)^2 = 7^i \cdot 18 \cdot \frac{n^2}{9^i}$
- Total number of *recursive* levels: $\log_3(n)$
- Total work done in base case: $7^{\log_3(n)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_3(n)-1} 7^i \cdot 18 \cdot \frac{n^2}{9^i}\right) + 7^{\log_3(n)}$$

We can simplify by pulling the $18n^2$ out of the summation:

$$18n^2 \left(\sum_{i=0}^{\log_3(n)-1} \frac{7^i}{9^i}\right) + 7^{\log_3(n)}$$

This is equivalent to:

$$18n^2 \left(\sum_{i=0}^{\log_3(n)-1} \left(\frac{7}{9}\right)^i\right) + 7^{\log_3(n)}$$

After applying the finite geometric series identity, we get:

$$18n^2 \cdot \frac{\left(\frac{7}{9}\right)^{\log_3(n)} - 1}{\frac{7}{9} - 1} + 7^{\log_3(n)}$$

This is a closed form so we could stop, but if we want a tidy solution, we can continue simplifying:

$$\begin{aligned}
T(n) &= 18n^2 \cdot \frac{\left(\frac{7}{9}\right)^{\log_3(n)} - 1}{\frac{7}{9} - 1} + 7^{\log_3(n)} \\
&= 18n^2 \cdot \frac{\left(\frac{7}{9}\right)^{\log_3(n)} - 1}{-\frac{2}{9}} + 7^{\log_3(n)} \\
&= -81n^2 \cdot \left(\left(\frac{7}{9}\right)^{\log_3(n)} - 1\right) + 7^{\log_3(n)} \\
&= -81n^2 \cdot \left(n^{\log_3(7/9)} - 1\right) + n^{\log_3(7)} \\
&= -81n^2 \cdot \left(n^{\log_3(7)-2} - 1\right) + n^{\log_3(7)} \\
&= -81n^2 n^{\log_3(7)-2} + 81n^2 + n^{\log_3(7)} \\
&= -80n^{\log_3(7)} + 81n^2
\end{aligned}$$

We can apply the master thereom here. Note that $\log_b(a) = \log_3(7) < 2 = c$, which means that $T(n) \in \Theta\left(n^c\right)$ which is $T(n) \in \Theta\left(n^2\right)$

This agrees with our simplified form.

6. Divide and conquer

Given an array containing elements of type `E` design an algorithm that finds the **majority element** – that is, an element that appears more then $n/2$ times. If no majority element exists, return `null`.

Your algorithm should run in $\mathcal{O}(n \log(n))$ time (and use only $\mathcal{O}(1)$ extra memory).

Note: the items in the array do **NOT** implement `compareTo`. This means you cannot sort the array!

Challenge: can you find the majority in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ extra memory?

Solution:

The $\mathcal{O}(n \log(n))$ solution works by first splitting the array into two halves. We recurse on both halves and receive back the majority elements for the two halves (if they exists).

Once we finish recursing, there are four different scenarios:

(a) The two subarrays have the same majority element.

This means, by definition, that element must also be the majority of the full array.

Why is this? Suppose that there are n elements in the overall array. If A is the majority of the left half, then that means that by definition, there must be $> \frac{n}{4}$ occurrences of A on the left. Similarly, if A is the majority on the right, there must be $> \frac{n}{4}$ occurrences there.

Therefore, there must be $> \frac{n}{2}$ occurrences of A overall. So we can just return A without needing to check anything else.

(b) The two subarrays have different majority elements.

In that case, we need to figure out which one is the true majority. We take the majority element from the left and loop over the entire array to figure out how many times it appears. We do the same thing with the majority element from the right. This will take $\mathcal{O}(2n) = \mathcal{O}(n)$ time.

If either of them appear more then $\frac{n}{2}$ time, return that element as the majority. If neither of them appear enough times, return null (or whatever else we're using to indicate that there's no majority).

(c) Only one subarray has a majority; the other doesn't.

We do the same sort of looping thing as before, again in $\mathcal{O}(n)$ time.

(d) Neither subarrays have a majority.

We can automatically give up here, for basically the same reason why we could automatically return in case 1.

We end up doing $2T(n) + n$ work in the worst case in the recursive case, which results in $\mathcal{O}(n \log(n))$.

The $\mathcal{O}(n)$ solution is called "Boyer-Moore majority vote algorithm." The Wikipedia page has a good overview of how the algorithm works: `https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_majority_vote_algorithm`