Section 07: Sorting, divide-and-conquer

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]

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

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

(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?

2. In-Depth Recurrence

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

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 $A(n)$.

(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.

(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.

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

(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.

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

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

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

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

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

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.

(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).

(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.

(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.

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

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

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

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?