Section 06: Heaps, sorting, divide-and-conquer

1. Heaps

(a) Insert the following sequence of numbers into a *min heap*:

[10, 7, 15, 17, 12, 20, 6, 32]

(b) Now, insert the same values into a *max heap*.

(c) Now, insert the same values into a *min heap*, but use Floyd's `buildHeap` algorithm.

(d) Insert 1, 0, 1, 1, 0 into a *min heap*.

(e) Call `removeMin` three times on the min heap stored as the following array: $[1, 5, 10, 6, 7, 13, 12, 8, 15, 9]$

2. 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) 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?

3. Master Theorem

For each of the recurrences below, use the Master Theorem to find the big-Θ of the closed form or explain why Master Theorem doesn't apply.

Master Theorem:
Given a recurrence of the following form:

$$T(n) = \begin{cases} d & \text{if } n \leq \text{ some constant} \\ aT(n/b) + n^c & \text{otherwise} \end{cases}$$

with a, b, c are constants.
If $\log_b(a) < c$ then $T(n)$ is $\Theta(n^c)$
If $\log_b(a) = c$ then $T(n)$ is $\Theta(n^c \log n)$
If $\log_b(a) > c$ then $T(n)$ is $\Theta\left(n^{\log_b(a)}\right)$

(a) $T(n) = \begin{cases} 18 & \text{if } n \leq 5 \\ 3T(n/4) + n^2 & \text{otherwise} \end{cases}$

(b) $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 9T(n/3) + n^2 & \text{otherwise} \end{cases}$

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

(d) $T(n) = \begin{cases} 1 & \text{if } n \leq 19 \\ 4T(n/3) + n & \text{otherwise} \end{cases}$

(e) $T(n) = \begin{cases} 5 & \text{if } n \leq 24 \\ 2T(n-2) + 5n^3 & \text{otherwise} \end{cases}$

4. Analyzing recurrences, redux

Consider the following recurrences. For each recurrence, (a) find a closed form using the tree method and (b) check your answer using the master theorem.

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

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

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

5. 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.

6. Divide and conquer: challenge questions

(a) Design an algorithm that accepts an unsorted array of integers and finds the subarray with the maximum possible sum.

For example, consider the array `[2, -4, 1, 9, -6, 7, -3]`. The maximum subarray would be `[1, 9, -6, 7, -3]`, which sums to 11.

A naive solution that considers every possible subarray would take $\mathcal{O}(n^2)$ time. Design a more efficient algorithm that uses divide and conquer and runs in $\mathcal{O}(n \log(n))$ time.

Challenge: Can you design an algorithm that runs in $\mathcal{O}(n)$ time?

(b) 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?

(c) Suppose we have two polynomials represented as two int arrays, where the i-th item represents the i-th coefficient. So, the array `[5, 10, 0, 2, -3]` would represent the polynomial $5 + 10x + 2x^3 - 3x^4$.

Design an algorithm that accepts two of these arrays and returns a new one representing the product of the two. You may assume both input arrays both have length n. A naive implementation using nested loops will have $\mathcal{O}(n^2)$ work; your algorithm must be asymptotically better.

Hint: Note that a polynomial A can be written as $A_0 + A_1 x^{n/2}$, where A_0 is the first $n/2$ terms and A_1 is the latter $n/2$ terms. This means that $A \cdot B = (A_0 + A_1 x^{n/2}) \cdot (B_0 + B_1 x^{n/2})$. With some algebra, we can simplify to obtain:

$$A \cdot B = A_0 B_0 + ((A_0 + A_1)(B_0 + B_1) - A_0 B_0 - A_1 B_1) x^{n/2} + A_1 B_1 x^{n/2}$$

This means that computing the product of A and B requires you to multiply polynomials exactly three times. (Note: not 5 times – why?). You should exploit this property when implementing your algorithm.

(d) Suppose you are trying to write a video game containing thousands of different moving elements and want to check if two elements have collided or overlapped.

A naive way of implementing this would be to use two nested loops and check every pair of elements. This often ends up being too inefficient for most video games, even for only a few thousand elements (especially games that require a high degree of responsiveness).

Describe how you would design a data structure to store these points in a way that lets you more efficiently check whether two elements are colliding.

For the sake of simplicity, you may assume that each element is a circle and has a relatively small radius. You may also assume that the elements are moving on a 2d plane – you don't need to worry about collisions in 3d.

As a hint: think about recursively subdividing the 2d plane.