Week 7: Lab

Collaboration level 0 (no restrictions). Open notes.

Overview: We saw two examples of divide-and-conquer algorithms that are widely used: large integer multiplication and matrix multiplication.

1. **Integer multiplication:** Download the notebook *karatsuba.ipynb*, and extend your understanding of this problem by running through it step by step.

2. **The maximum sub-array problem**[1] Given an array A of n integers, find values of i and j with $0 \leq i \leq j < n$ such that the sum

$$A[i] + A[i+1] + ... + A[j] = \sum_{k=i}^{j} A[k]$$

is maximized. Sometimes this is called the *maximum partial sum problem.*

Example: Consider the array $A = [4, -5, 6, 7, 8, -10, 5]$. What is the MPS?
Answer: the solution to MPS is $i = 2$ and $j = 4$ $(6 + 7 + 8 = 21)$.

Note that if all values are positive, then the problem is trivial ($i = 0$ and $j = n - 1$). The problem is non-trivial only if there are negative numbers.

Applications: This problem might be encountered in financial analysis, and one textbook offers a possible scenario. Basically, think of a value in the array as a relative gain/loss of a stock (wrt to some fixed initial value). Finding the maximum sub-array would (retro-actively) tell you when you should have purchased and sold a stock in order to maximize gain. Today we'll come up with an $O(n \lg n)$ solution via divide-and-conquer. In a couple of weeks we'll come back and give an $O(n)$ solution for this problem using dynamic programming (Kadane's algorithm).

(a) Consider the following array:

$$A = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]$$

Find the MPS and the corresponding i, j.

[1] This problem is frequently asked in interviews. LeetCode #53.

(b) One thought that comes to mind is whether the MPS can include negative numbers. A negative number will afterall decrese the MPS, so it feels that they should be skipped. Or do they? Examine this issue and give an argument one way or the other.

(c) Describe a straightfoward algorithm to find the MPS and analyze its running time. We'll refer to this as the simple algorithm, the straightforward algorithm, or the brute-force.

As always, the question is: Can we do better? For e.g., can we solve MPS in $O(n \lg n)$ time? As it turns out, a neat $O(n \lg n)$ algorithm for MPS is possible via divide-and-conquer. The subsequent questions guide you towards the solution.

(d) Suppose you know the MPS of A and its i and j indices. List the three posibilities that can happen based on how i and j are located compared to the middle index $\lfloor n/2 \rfloor$.

(e) You can use this insight to set up the correctness grounds for recursion. Fill in the blanks:

Claim: The MPS of array A[0.., n-1] is either (a) the MPS of .....; or (b) the MPS of ..... or (c) .....

(f) To find the MPS we need to find the two indices i and j that mark its start and end.

Now consider the one-dimensional version of this problem, Namely, consider that the left index of the MPS l is given and you want to find the index j ($l \leq j < n$) such that $A[l] + A[l+1] + ... + A[j]$ is maximized. (the "find the MPS that starts at a given index" problem).

How fast can you find index j in this case? Remember that l is given.

(g) Similarly, assume that the right index r of the MPS is given, and you want to find the index i such that $A[i] + A[i+1] + ... + A[r]$ is maximized. (the "find the MPS that ends at a given index" problem).

How would you find index j and how fast? Brief answer ok.

(h) So if *you knew* the start or end of the MPS, you could find the other one in $O(n)$ time. Right? What if an oracle told you that a certain index k is part of the MPS; would that help? how would you use that to find the MPS in linear time?

(i) Describe an $O(n \log n)$ divide-and-conquer algorithm for solving MPS problem.

"Reduce-and-conquer" algorithms

The algorithms we saw so far divide the problem into several sub-problems and solve them recursively: For e.g. we start from a problem of size n, then solve two or more subproblems of size $n/2$, then four (or more) subproblems of size $n/4$, and so on.

Now let's consider Binary search: To search an array of size n, we search either the left half or the right half, recursively. So we start with an array and recurse on one of its halves. We are still *dividing* the problem, but we only solve one of the subproblems. This can also be considered a divide-and-conquer-type algorithm and it's sometimes refered to as reduce-and-conquer—whatever the name, it does not really matter. The next problem is an example of this type.

Local min[2]: Suppose we are given an array $A[1..n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x \ \leq A[x+1]$. For example. there are six local minima in the following array:

$$A = [9, 7, 7, 2, 1, 3, 7, 5, 4, 7, 3, 3, 4, 8, 6, 9]$$

.

We can obviously find a a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\lg n)$ time. (*Hint: with the given boundary conditions, the array must have at least one local minimum. Can you see why?*)

We expect: pseudocode and a brief English description of your algorithm; why is it correct, i.e. why i can't miss the minimum value; (3) analysis of its running time.

[2] LeetCode #162