Section 4: Divide and Conquer

In this section, we'll design a Divide and Conquer algorithm together for the maximum subarray sum problem.

1. Maximum Subarray Sum

Input: An array of `ints` (possibly a mix of positive, negative or 0)
Output: The largest possible sum of a (contiguous) subarray $A[i] + A[i+1] + \cdots + A[j]$.

A single element counts as a subarray (the sum is the value of that element). No elements counts as a subarray (the sum is 0).

1.1. Read and understand the problem

Read the problem and answer the usual quick-check-questions

- What is the input type?
- What is the output type?
- Are there any technical terms in the problem you should pay attention to?
- What is a clear English definition of the return value from the recursive calls?

1.2. Generate Examples

Generate at least two examples along with their correct answers. When you're in the brainstorming step, make sure at least one of these examples has the closest pair in a recursive call and at least one has the answer found in the "conquer" step.

1.3. Come up with a Baseline

What is the first algorithm that comes to mind for the problem? What would it's running-time be? (Don't try to do divide and conquer *yet*).

1.4. Brainstorm!

Now, let's try divide and conquer. How do you want to split up the problem? Imagine you have the answers from those recursive calls? What is there still to handle?

1.5. Write an algorithm

The key to a good divide and conquer algorithm is making sure your conquer step is **not** just brute force. Find an algorithm that is faster than your baseline.

1.6. Show your algorithm is correct

Write a proof that your algorithm is correct. You probably want a proof by induction. For simplicity, in your proof you may assume there are no ties (i.e., there is only one possible subarray with the maximum sum).

1.7. Optimize and Analyze the running time

Analyze the running time.

More Problems!

2. Counting Inversions

In this problem, we'll design a divide and conquer algorithm for counting inversions. An "inversion" in an array A is a pair of indices i, j such that $i < j$ but $A[i] > A[j]$. Intuitively, they're elements that are "not in sorted order." For simplicity, assume all elements of your array are distinct in this problem.

For example, in the array: $[8, 2, 91, 22, 57]$
There are three inversions: 8 with 2, 91 with 22 and 91 with 57.

(a) Write a preliminary algorithm that *looks like* a divide and conquer algorithm for counting inversions. Try to do the most obvious choice for counting (this will most likely still be brute force). After designing your algorithm look at the solution given and convince yourself why this on the same level as brute force.

(b) Now imagine that after you make the recursive calls, you sort only the right subarray. How can you update your conquer step? How much faster is it?

(c) Now imagine that after you make the recursive calls, you sort both subarrays (separately). How can you update your conquer step now? How much faster is it?

(d) Does the conquer step feel familiar? Maybe like a step from `mergesort`? Update your code to do both the (merge-)sorting and the inversion counting in the same recursive structure.

(e) If sorting was so helpful, why didn't we just sort the whole array at the start? Wouldn't that have been way easier?

3. Binary Search Variant

Let $A[1..n]$ be an array of `ints`. Call an array a **mountain** if there exists an index i called "the peak", such that:
$\forall 1 \leq j < i (A[j] < A[j+1])$
$\forall i \leq j < n (A[j] > A[j+1])$

Intuitively, the array increases to the "peak" index i, and then decreases. Note that either of these conditions could be vacuous if the peak is index 1 or n (e.g., a decreasing array is still a mountain).

(a) Given an array `A[1..n]` that you are promised is a mountain, find the index peak index in better than $\mathcal{O}(n)$ time.

(b) Can you design an algorithm with the same running time that also **determines** whether a given array is a mountain (and if it is, finds the peak)?

4. `QuickSelect` Running Time

In class we derived a recurrence for `QuickSelect`. Here's a particular version of that recurrence, with some constants filled in, and some floors/ceilings added as appropriate.

$$T(n) = \begin{cases} 10 & \text{if } n \leq 100 \\ T\left(\lceil \frac{7n}{10} \rceil\right) + T\left(\lceil \frac{n}{5} \rceil\right) + 5n & \text{otherwise} \end{cases}$$

Prove, via induction, that $T(n) \leq 80 \cdot n$ for all $n \geq 1$. In this problem, we really care about ceilings/floors and off-by-one errors (the goal here is to double-check the hand-waving from lecture really works. It doesn't help much

to replace the other hand-waving with more hand-waving!). In particular, be careful that $T(\cdot)$ takes only integers as inputs.