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 integers $\texttt{A} = a_1, \dots, a_n$ (possibly both positive and negative)
Expected output: The largest sum of any contiguous subarray $\texttt{A}[i\,..\,j]$.

Notation: Denote $\texttt{A}[i\,..\,j]$ the subarray $a_i, a_{i+1}, \dots, a_j$.

Note that the list of no elements is a valid subarray (the sum is 0). The expected output is the sum, not the actual subarray.

1.1. Generating ideas

For problems that can be solved with divide an conquer, there will almost always be an easy but slow baseline idea that you can try first.

(a) Let's come up with an easy baseline solution (no divide and conquer yet).

(i) What is the simplest idea that you can try? What is the running time?

(ii) Are there any inefficiencies with this idea that can be easily fixed (still no divide and conquer)? If so, what is the running time after fixing?

Thus, we would be satisfied to get a solution around $O(n \log n)$.

(b) Here are some basic questions to always ask yourself for divide and conquer:

(i) How do you want to split up the problem?

(ii) What is returned from the recursive calls?

(iii) Up to how much work is allowed in each call, in order to get the running time you want? (here $O(n \log n)$)

(c) Solve these examples by hand, as well as the two recursive subproblems in each example (just one level of recursion). Then, think about the following to get ideas:

"How can I use the two answers to the subproblems to get the final answer?"

Remember how much work you are allowed to do.

(i) $2, -10, -5, 8, -1, 7$

(ii) $6, -3, -4, 4, 2, 1, -7, 5$

(iii) $-3, 2, 4, -1, 3, -10, 6, -4$

(Again, in a real problem without our help, you would come up with these examples by yourself.) Continue trying more examples until you have an idea for the full solution.

1.2. Writing up your idea

Some reminders for divide and conquer pseudocode:

- Always give your function a name, since you will need to call it recursively.
- In pseudocode, our default will be that function parameters pass by value.
  - If you pass arrays by value, you automatically use $O(n)$ time.
  - To achieve sub-$O(n)$, you must use references, pointers, global variables (or generally variables scoped outside the function), or the equivalent in your favorite programming language. (These solutions use global variables, since they are a bit more language-agnostic, but it's subjective.)
  - Not relevant for this problem since we use $O(n)$ time anyways.

(d) Write the pseudocode for your solution.

Some reminders for divide and conquer proofs:

- Always use strong induction. Your IH should be:

  "My main function outputs its expected output for all inputs of size $\leq k$."
- The structure can be inspired by your code, which already has a "base case" and "recursive (inductive) step". Also, if your code branches on anything (if, max, min, etc.), your proof should have cases based on what kinds of inputs end up at each branch.
- You should explain why your output is the expected output as usual, but also, just a special reminder that if the input to your problem is "X such that Y holds", make sure to explain why Y holds for recursive calls, too.

(e) Write the proof that your pseudocode works.

(f) Analyze the running time of your code by solving a recurrence.

The following problems will not be covered in section, but may be useful to think about.
We recommend trying them by yourself first. Solutions will be posted in the evening.

2. Counting inversions

In this problem, we'll design a divide and conquer algorithm for counting inversions. You've seen them in lecture, but to review, an "inversion" in an array `A` $= a_1, \ldots, a_n$ 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 $(3, 5)$.

Input: An array `A` $= a_1, \ldots, a_n$ of distinct integers
Expected output: The number of inversions in `A`

(a) Describe a simple $O(n^2)$ algorithm for this problem that does not involve divide and conquer.

(b) Write an algorithm that *looks like* a divide and conquer algorithm (by using answers from two subproblems), but still takes $O(n^2)$ time per recursive call. What is the running time of such an algorithm?

(c) Now imagine that after you make the recursive calls, you sort the right subarray. Show that you can now find an $O(n \log^2 n)$ algorithm.

Hint: A more general version of the Master Theorem includes the following case: Suppose $T(n) = aT(n/b) + O(n^k \log^c(n))$. If $a = b^k$, then $T(n) = O(n^k \log^{c+1}(n))$.

(d) Now imagine that after you make the recursive calls, you sort both subarrays (separately). How can you update your the code now?

Note: You will not get an overall big-O improvement from this step. However, the entirety of the for loop(s) from previous parts should be able to be replaced with something just $O(n)$ (where it used to be $O(n^2)$ in part (a), and then $O(n \log n)$ in part (b)).

Hint: Think about how merge sort handles two already sorted arrays.

(e) Now, update your code to do both merge sort and inversion counting at the same time. Show how this leads to an $O(n \log n)$ algorithm for inversion counting. Lastly, if you were to prove the correctness of this algorithm, what should the inductive hypothesis be?

(f) 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. Peak performance

Let `A` $= a_1, \ldots, a_n$ be an array of integers. Call `A` a *mountain* if there exists an index i called the "peak", such that $a_1 < \cdots < a_{i-1} < a_i$ and $a_i > a_{i+1} > \cdots > a_n$. We allow the peak to be at index 1 or n (that is, a strictly increasing or strictly decreasing array is still a mountain). For simplicity, we are not allowing $a_j = a_{j+1}$ for any j. More formally, mountains satisfy:

- For all $1 \leq j < i$, we have $a_j < a_{j+1}$, and
- For all $i \leq j < n$, we have $a_j > a_{j+1}$.

(a) Given a mountain A, describe an algorithm to find the index of the peak in $O(\log n)$ time — very fast!

(b) Prove that there is no deterministic algorithm for deciding in the same $O(\log n)$ running time: Given an array, whether or not it is a mountain.

(Deterministic means that we don't use randomness, like almost all algorithm we study in this class. In other words, the algorithm always has the same output when viewing the same input.)