Section 4: Solutions

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 $\texttt{A} = a_1, \ldots, a_n$ (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 $a_i, a_{i+1}, \ldots, a_j$.

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?

Solution:

Check the sum of every possible subarray $\texttt{A}[i, j]$. There are $O(n^2)$ different subarrays (pick i and j), and sum takes $O(n)$ time per subarray, for a total of $O(n^3)$.

(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?

Solution:

To compute the sum of $\texttt{A}[i \,..\, j+1]$, you don't need to spend $O(n)$ time, just use $O(1)$ time to add a_{j+1} to the sum of $\texttt{A}[i \,..\, j]$, which you already computed. Now it's $O(n^2)$.

Now, we know that $O(n^2)$ is easy. 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?

Solution:

Two halves, $\texttt{A}[1 \,..\, m]$ and $\texttt{A}[m+1 \,..\, n]$, where $m = \lfloor \frac{n}{2} \rfloor$. (Need to call both.)

(ii) What is returned from the recursive calls?

Solution:

The largest sum of any contiguous subarray in each half.

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

Solution:

Up to $O(n)$ work per recursive call gets $O(n \log n)$, like in merge sort.

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

Solution:

Full solution: $[8, -1, 7]$ with sum 14.
Left half: $[2]$ with sum 2.
Right half: $[8, -1, 7]$ with sum 14.
How to combine: We took the solution from the right half.

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

Solution:

Full solution: $[4, 2, 1]$ with sum 7.
Left half: $[6]$ with sum 6.
Right half: $[5]$ with sum 5.
How to combine: Both answers to subproblems were smaller than the full solution, which crossed the boundary.

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

Solution:

Full solution: $[2, 4, -1, 3]$ with sum 8.
Left half: $[2, 4]$ with sum 6.
Right half: $[6]$ with sum 6.
How to combine: Both answers to subproblems were smaller than the full solution, which crossed the boundary.

(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. **Solution:**

The largest subarray sum is either in the left or right half, or crosses the boundary. To find the largest subarray sum that crosses the boundary in $O(n)$ time: From the middle, search down for the largest sum of all arrays of the form $\texttt{A}[i \,..\, m]$ (where $1 \leq i \leq m$) and similarly search up for arrays of the form $\texttt{A}[m{+}1 \,..\, j]$ (where $m + 1 \leq j \leq n$), then put them together.

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.

Solution:

The following is the core idea of the algorithm:

1: **function** MAXSUBARRAYSUM(A$[1\,..\,n]$)
2: **if** $n = 1$ **then**
3: **return** A$[1]$ if it is positive, or 0 otherwise
4: $m \leftarrow \lfloor \frac{n}{2} \rfloor$
5: maxSumToMiddle $\leftarrow$ largest sum of any subarray of type A$[i\,..\,m]$
6: maxSumFromMiddle $\leftarrow$ largest sum of any subarray of type A$[m+1\,..\,j]$
7: crossSum $\leftarrow$ maxSumToMiddle $+$ maxSumFromMiddle
8: **return** max(crossSum, MAXSUBARRAYSUM(A$[1\,..\,m]$), MAXSUBARRAYSUM(A$[m+1\,..\,n]$))

In order to analyze the running time of lines 5 and 6, we need to be a bit more specific about how we find these sums. Here is pseudocode that explains line 5 (and line 6 is similar). In your solutions, you can either do this, or include an English summary somewhere in the running time analysis, along the lines of: "To compute line 5 efficiently, compute partial sums $a_i + \cdots + a_m$ by starting with a_m and iterating downwards."

1: partialSum $\leftarrow$ A$[m]$
2: maxSumToMiddle $\leftarrow$ partialSum
3: **for** $i \leftarrow m-1$ down to 1 **do**
4: partialSum $\leftarrow$ partialSum $+$ A$[i]$
5: **if** partialSum $>$ maxSumToMiddle **then**
6: maxSumToMiddle $\leftarrow$ partialSum

We mentioned above that sometimes, you may want to use global variables to improve the efficiency of your solution. That didn't matter here, since pass by value takes $O(n)$ time to copy the input, but we spend $O(n)$ per recursive call already. But just for your reference, here is how we would rewrite the above pseudocode to use global variables:

1: **global** A$[1\,..\,n]$
2: **function** MAXSUBARRAYSUM(a, b)
3: **if** $a = b$ **then**
4: **return** A$[a]$ if it is positive, or 0 otherwise
5: $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
6: maxSumToMiddle $\leftarrow$ largest sum of any subarray of type A$[i\,..\,m]$, where $i \geq a$
7: maxSumFromMiddle $\leftarrow$ largest sum of any subarray of type A$[m+1\,..\,j]$, where $j \leq b$
8: crossSum $\leftarrow$ maxSumToMiddle $+$ maxSumFromMiddle
9: **return** max(crossSum, MAXSUBARRAYSUM(a, m), MAXSUBARRAYSUM$(m+1, b)$)
10: **output** MAXSUBARRAYSUM$(1, n)$

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.

Solution:

BC: The largest subarray sum of a length 1 array is itself if positive, or 0 if negative.

IH: MAXSUBARRAYSUM returns the maximum subarray sum for all arrays of length $\leq k$.

IS: Let `A` be an array of length $k+1$.

Case 1: The maximum subarray is entirely in the left or right subarray. By IH, the recursive calls in line 8 find this subarray and return it.

Case 2: The maximum subarray crosses from the left to the right. All subarrays $\texttt{A}[i\,..\,j]$ that cross both halves can be divided into $\texttt{A}[i\,..\,m]$ and $\texttt{A}[m+1\,..\,j]$, where $m = \lfloor \frac{n}{2} \rfloor$ as in the code. But we know that $\texttt{maxSumToMiddle} \geq \texttt{sum}(\texttt{A}[i\,..\,m])$ and similarly $\texttt{maxSumFromMiddle} \geq \texttt{sum}(\texttt{A}[m+1\,..\,j])$. Adding these, we get that $\texttt{crossSum} \geq \texttt{sum}(\texttt{A}[i\,..\,j])$ for all subarrays $\texttt{A}[i\,..\,j]$ that cross both halves, and `crossSum` certainly represents *some* array, so it is the maximum subarray sum.

Solution:

Remark: Note how the cases here are only inspired by code, not regurgitating code. The cases are high-level: what kinds of inputs end up at each branch? They are not the specific criteria that the code checks.

It's technically possible to write a correct proof if you use cases based on the specific criteria in the code, like "*Case 1*: `crossSum` is larger than MAXSUBARRAYSUM($\texttt{A}[1\,..\,m]$) and MAXSUBARRAYSUM($\texttt{A}[m+1\,..\,n]$)". But it's unnecessarily more complicated. The proof would have to go something like:

"By IH, we know that `crossSum` must be larger than all subarrays entirely in the left or right half. It remains to show that `crossSum` is at least as large as all subarrays that cross from the left to the right." Then, write exactly the proof as Case 2 above.

See how it's just longer for no reason?

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

Solution:

Lines 5 and 6 take $O(n)$ time each. Since we make a recursive call on each half, we have the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

By the Master Theorem, this means $T(n) = O(n \log n)$.

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 $\texttt{A} = a_1, \ldots, a_n$ 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 $\texttt{A} = a_1, \ldots, a_n$ of distinct integers
Expected output: The number of inversions in $\texttt{A}$

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

Solution:

Loop over all pairs (i, j) with $i < j$ and increment a counter whenever $a_i > a_j$.

(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?

Solution:

```
1:  global A
2:  function numInversions(a, b)
3:      if a = b then
4:          return 0
5:      m ← ⌊(a+b)/2⌋
6:      invCount ← numInversions(a, m) + numInversions(m + 1, b)
7:      for i ← a to m do
8:          for j ← m + 1 to b do
9:              if a_i > a_j then
10:                 invCount ← invCount + 1
11:     return invCount
12: output numInversions(1, n)
```

The recurrence is $T(n) = 2T(n/2) + O(n^2)$. The Master Theorem says that if $T(n) = aT(n/b) + O(n^k)$ and $a < b^k$, then $T(n) = O(n^k)$. Since $2 < 2^2$, we have $T(n) = O(n^2)$.

(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 $T(n) = O(n^k \log^{c+1}(n))$.

Solution:

```
1:  global A
2:  function numInversions(a, b)
3:      if a = b then
4:          return 0
5:      m ← ⌊(a+b)/2⌋
6:      invCount ← numInversions(a, m) + numInversions(m + 1, b)
7:      Sort A[m + 1 .. b] in place, in increasing order.
8:      for i ← a to m do
9:          Binary search in A[m + 1 .. b] to find the largest k ∈ {m + 1, ..., b} such that a_k < a_i.
10:         if k exists then
11:             invCount ← invCount + k − m        ▷ elements a_{m+1}, ..., a_k form inversions with a_i
12:     return invCount
13: output numInversions(1, n)
```

Line 7 takes $O(n \log n)$ time. The loop in line 8 occurs of $O(n)$ iterations and takes $O(\log n)$ time per iteration, again for $O(n \log n)$ time. Thus the recurrence is $T(n) = 2T(n/2) + O(n \log n)$. By the more general Master Theorem, we get $T(n) = O(n \log^2 n)$.

(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.

Solution:

```
1: global A
2: function NUMINVERSIONS(a, b)
3:     if a = b then
4:         return 0
5:     m ← ⌊(a+b)/2⌋
6:     invCount ← NUMINVERSIONS(a, m) + NUMINVERSIONS(m + 1, b)
7:     Separately sort A[a..m] and A[m + 1..b] in place, in increasing order.
8:     i ← a
9:     j ← m + 1
10:    while i ≤ m and j ≤ b do
11:        if a_i < a_j then                    ▷ i forms no inversions with anything ≥ j
12:            i ← i + 1
13:        else                                 ▷ j forms an inversion with everything ≥ i
14:            invCount ← invCount + m − i + 1
15:            j ← j + 1
16:    return invCount
17: output NUMINVERSIONS(1, n)
```

(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?

Solution:

```
1: global A

   Input: Indices a and b
   Output: The number of inversions in A[a..b]
   Ensures: A[a..b] is sorted in increasing order
2: function NUMINVERSIONS(a, b)
3:     if a = b then
4:         return 0
5:     m ← ⌊(a+b)/2⌋
6:     invCount ← NUMINVERSIONS(a, m) + NUMINVERSIONS(m + 1, b)

7:     Let sorted be a new empty list.
8:     i ← a
9:     j ← m + 1
10:    while i ≤ m and j ≤ b do
11:        if a_i < a_j then                    ▷ i forms no inversions with anything ≥ j
12:            Append a_i to sorted.
13:            i ← i + 1
14:        else                                 ▷ j forms an inversion with everything ≥ i
15:            invCount ← invCount + m − i + 1
16:            Append a_i to sorted.
17:            j ← j + 1
```

```
    while i ≤ m do
        Append a_i to sorted.
        i ← i + 1
    while j ≤ b do
        Append a_j to sorted.
        j ← j + 1
    Replace A[a..b] with sorted.
    return invCount
output NUMINVERSIONS(1, n)
```

Since we now get sorting by induction (recursion) rather than having to do it repeatedly every iteration, every call is now just $O(n)$. By the Master Theorem, $T(n) = 2T(n/2) + O(n)$ resolves to $T(n) = O(n \log n)$.

The IH is: "For all $1 \leq a \leq b \leq n$ with $b - a \leq k$, we have that NUMINVERSIONS(a, b) returns the number of inversions in $\texttt{A}[a..b]$ and sorts $\texttt{A}[a..b]$ in increasing order."

Note that it is extremely important here to include the sorting in the inductive hypothesis, because we are relying on this fact to accurately count the number of inversions. This is despite the fact that the original problem didn't ask us to sort the array. So, this is an example of a problem where it was helpful to compute more than we were asked to, and when this happens, the IH should reflect what you actually compute, not what you were asked to.

(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?

Solution:

Sorting can destroy inversions! We're only allowed to swap two elements if we have already counted any inversions we destroy when we move them.

3. Peak performance

Let $\texttt{A} = a_1, \ldots, a_n$ 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.)

Solution:

(a) **Key idea**: We adapt binary search — by looking at consecutive elements, we can see if we're on the "upward" or "downward" slope and find the peak.

```
1:  global A

    Input: Indices i and j such that A[i..j] contains the peak of A.
    Output: The peak of A
2:  function peakFinder(i, j)
3:      if i = j then
4:          return i
5:      m ← ⌊(i+j)/2⌋
6:      if a_{m+1} exists and a_m < a_{m+1} and m + 1 ≤ j then
7:          return peakFinder(m + 1, j)
8:      else if a_{m−1} exists and a_{m−1} > a_m and i ≤ m − 1 then
9:          return peakFinder(i, m − 1)
10:     else
11:         return m
12: output peakFinder(1, n).
```

The running time is the same as binary search, which is $O(\log n)$.

We will show by strong induction on k that for all k, given indices i and j such that $k = |\{i, \ldots, j\}|$ and $\texttt{A}[i\,..\,j]$ contains the peak of A, the function peakFinder(i, j) returns the peak of A.

BC: When $k = 1$, the only way for $\texttt{A}[i\,..\,j]$ to contain the peak of A is for $i = j$ to be the peak. In this case, we return i as intended.

IH: Suppose that the claim above is true for all $k \leq \ell$.

IS: We will show the claim for $k = \ell + 1$. Take the following three cases, corresponding to the three cases in the code.

Case 1: The peak is in $\texttt{A}[m + 1\,..\,j]$. Then a_{m+1} certainly exists, and a_m is in the increasing part of the mountain, and certainly $m + 1 \leq j$. Thus, the code will enter the first "if" branch on line 6. The case hypothesis allows us to apply the IH, which tells us that peakFinder$(m+1, j)$ returns the peak, as desired.

Case 2: The peak is in $\texttt{A}[i\,..\,m - 1]$. Similar to Case 1.

Case 3: The peak is a_m. Then $a_m > a_{m+1}$ or a_{m+1} doesn't exist, and $a_{m-1} < a_m$ or a_{m-1} doesn't exist. Thus, the code will hit the "else" case and return m, as desired.

(b) The main idea is that you need to examine *every* element of the array to know if it's a mountain, which requires at least $O(n)$ time.

To see why, suppose for contradiction that there exists an algorithm for this problem that looks at only $\leq n - 1$ entries. Let A be a mountain. By correctness of the algorithm, it outputs true on input A. Let a_k be an array element that the algorithm did not look at. Let $\texttt{A}'$ be an array identical to A on all elements except a_k, which we set to make $\texttt{A}'$ not a mountain (e.g. by lowering a_k below both of its neighbors, or above its neighbor if k is an endpoint). Since the algorithm did not look at a_k, its execution on A must be identical to its execution on $\texttt{A}'$. So the algorithm will return true on $\texttt{A}'$, contradicting its correctness.

Solution:

Remark: It is very easy to make a mistake in the IH here. The following are not correct IH's:

- For all arrays of length $\leq k$, the function peakFinder(i, j) returns the peak.

  (Many issues, not least of which is that peakFinder doesn't even take an array as input.)

- For all arrays of length $\leq k$, the outer function (the one which returns $\textsc{peakFinder}(1, n)$) returns the peak of `A`.

  (Although the claim is true, this is not a useful IH. You are never calling this outer function, so you have no opportunity to use the IH in your proof.)

- For all indices i, j, the function $\textsc{peakFinder}(i, j)$ returns the peak.

  (Not specific enough—the peak of what? If you wrote this, we would assume you meant the peak of `A`. But that's wrong. If you call $\textsc{peakFinder}(i, j)$ when $\texttt{A}[i\,..\,j]$ does not contain the peak of `A`, it will not return the peak of `A`.)

Another correct IH would be:

"For all indices i, j, the function $\textsc{peakFinder}(i, j)$ returns the peak of $\texttt{A}[i\,..\,j]$."

The proof would be slightly more complicated, but also correct. You would need to notice (and maybe prove) that every subarray of a mountain is a mountain.