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 `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?

Solution:

- Input type is `int[]`
- Output type is `int` (the largest **sum** of a subarray)
- "subarray" is a technical term. A subarray must be contiguous (elements 1,3,4 are not a subarray without element 2). The edge cases (one element, no elements) are defined in the problem.
- Each recursive call of the form `SubarraySumDC`$(A[], i, j)$ returns the largest sum of a contiguous subarray of $A[]$, with all elements of the subarray occurring between i and j.

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. **Solution:**

$[-3, -7, -2, -10]$ has a maximum subarray of [] with a sum of 0.
$[2, -100, 50, 3, -10, 17]$ has a maximum subarray of $[50, 3, -10, 17]$ with a sum of 60.
$[1, 2, 3, 4]$ has a maximum subarray of $[1, 2, 3, 4]$ with a sum of 10.
$[16, 20, -10, 4, 1, 0]$ has a maximum subarray of $[16, 20]$ with a sum of 36.

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*). **Solution:**

The main idea here is to check all possible subarrays. There are n possible "start" indices and n possible "end" indices (about half of which correspond to empty arrays). The following algorithm runs in $\mathcal{O}\left(n^3\right)$ time, because each subarray can be described by a start and end index, so $\mathcal{O}\left(n^2\right)$ possible subarrays, and computing each sum is $\mathcal{O}\left(n\right)$.

```
function NaiveBaseline(A[1..n])
    bestSum ← −∞
    for i from 1 to n do                          ▷ i represents start index
        for j from i to n do                      ▷ j represents end index
            sum ← 0
            for k from i to j do                  ▷ Find sum for A[i]+...+A[j]
                sum += A[k]
            if sum > bestSum then
                bestSum ← sum
    if bestSum < 0 then                           ▷ handle all negative entries case
        return 0                                  ▷ empty subarray is best if we've found only negative so far
    return bestSum
```

By keeping track of the partial sum you can get the running time down to $\mathcal{O}\left(n^2\right)$.

```
function BetterBaseline(A[1..n])
    bestSum ← −∞
    for i from 1 to n do                          ▷ i represents start index
        sum ← 0
        for j from i to n do                      ▷ j represents end index
            sum += A[j]
            if sum > bestSum then
                bestSum ← sum
    if bestSum < 0 then                           ▷ handle all negative entries case
        return 0                                  ▷ empty subarray is best if we've found only negative so far
    return bestSum
```

Our baseline is $\mathcal{O}\left(n^2\right)$.

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? **Solution:**

Let's just split the array in half! Make two recursive calls, one for each half (we don't know where the subarray is, so we'll have to make both).

What's still to handle? If the subarray "crosses" from one side to the other (i.e., has at least one element in both subarrays), then we need to discover and check those.

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. **Solution:**

Just taking the brute force code and updating it a bit is **not** a more efficient algorithm.

So what can we do? Well we're only interested in "crossing" subarrays, that is a subarray where $i < n/2$ and $j > n/2$. We know that $A[n/2]$ is included and we know the subarray is contiguous. How do these help us? It makes the choice of i and j independent of each other! In fancy math notation:

$$\max_{i,j:i<n/2<j} \sum_{k=i}^{j} A[k] = \max_{i \le n/2} \sum_{k=i}^{n/2} A[k] + \sum_{k=n/2+1}^{j} A[k]$$

In English: Since we know $n/2$ is included, i no longer affects j: in the original problem, when $i < j$, the indices

are invalid, and we define the sum to be 0. But since we know $A[n/2]$ must be included, we know $i \leq n/2$ and i is now independent of j. For all values of i and j, we'll have $i < j$. And so changing i can be done independently of j. So optimizing i can be done independently of optimizing j.

```
function SubarraySumDC(A[1..n])
    if n < 100 then
        Run the baseline algorithm                    ▷ or any other brute force
    bestRecursiveSum ← max{SubarraySumDC(A[1..n/2]), SubarraySumDC(A[n/2+1..n])}
    if bestRecursiveSum < 0 then
        bestRecursiveSum ← 0
    bestLeftSum ← −∞; leftSum ← 0
    for i from n/2 down to 1 do
        leftSum += A[i]
        if leftSum > bestLeftSum then
            bestLeftSum ← leftSum
            bestLeftIndex ← i
    bestRightSum ← −∞; rightSum ← 0
    for j from n/2 + 1 to n do
        rightSum += A[j]
        if rightSum > bestRightSum then
            bestRightSum ← rightSum
            bestRightIndex ← j
    crossSum ← bestRightSum + bestLeftSum
    if crossSum > bestRecursiveSum then
        return crossSum
    return bestRecursiveSum
```

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). **Solution:**

We show that `subarraySumDC`$(i..j)$ returns the maximum subarray sum by induction on n, the length of the interval $i..j$.

Base Case: If $n < 10$, we run a brute force algorithm that checks every interval and returns the largest. Since every interval is checked, we return the largest.

IH: Suppose that `subarraySumDC` returns the largest subarray sum for all intervals of length $1, 2, ...k$, $k \geq 10$.

IS: Consider an array of length $k + 1$. By the bound on k, we will hit our recursive case in the code. We divide into cases, based on what the maximum subarray is:

Case 1: The maximum subarray is entirely in the left or right subarray
By IH, the recursive calls will return the sum of largest subarray in each half, so `bestRecursiveSum` will hold our desired final answer. We will return `bestRecursiveSum` unless `crossSum` is larger, but since `crossSum` always contains the sum of some subarray, it will not be larger in this case. Thus we return the sum of the maximum subarray.

Case 2: The maximum subarray crosses from the left to the right
Let the maximum subarray be from index i to index j. By the assumption for this case, $i \leq n/2$ and $j \geq n/2+1$. Let i' be the value of `bestLeftIndex` and j' be the value of `bestRightIndex` at the end of the two loops computing them. The code ensures that these give the largest value of $A[i'] + \cdots + A[n/2]$ so $A[i'] + \cdots + A[n/2] \geq A[i] + \cdots + A[n/2]$ and the largest value of $A[n/2+1] + \cdots + A[j']$ so $A[n/2+1] + \cdots + A[j'] \geq A[n/2+1] + \cdots + A[j]$ and hence

$$\texttt{crossSum} = A[i'] + \cdots + A[n/2] + A[n/2+1] + \cdots + A[j'] \leq A[i] + \cdots + A[n/2] + A[n/2+1] + \cdots + A[j],$$

Both of these are sums of crossing subarrays and the subarray from i to j is maximum so the values must be the same[a] Therefore `crossSum` is the sum of the maximum crossing subarray. By IH, the recursive calls contain sums of the maximum subarrays on the left and right. By the assumption for this case, those are less than `crossSum`, so we return the sum $A[i] + \cdots + A[j]$, as required.

[a]Note that this argument does not necessarily imply that $i = i'$ and $j = j'$ since it is possible to have more than one subarray that achieves the maximum; for example, one might have elements $+1, -1$ next to each other.

1.7. Optimize and Analyze the running time

Analyze the running time. **Solution:**

Note that the recursive case has two loops, each with $\mathcal{O}(n)$ iterations, doing constant work per iteration. Since we make recursive calls on each half, we have the recurrence:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{if } n < 100 \\ 2T(n/2) + \mathcal{O}(n) & \text{otherwise} \end{cases}$$

We have seen in class that this recurrence has the closed form $\mathcal{O}(n \log n)$.

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 an algorithm that *looks like* a divide and conquer algorithm for counting inversions, but is actually just brute force.

Solution:

```
function InversionsDC(()A, i,j)
    if j-i ≤2 then
        Count inversions by brute force and return.
    mid ← i + ⌊(j−i)/2⌋
    invCount ← 0
    invCount+ = InversionsDC(A, i, Mid)
    invCount+ = InversionsDC(A, Mid, j)
    for lo from i to mid do
        for hi from mid + 1 to j do
            if A[lo] > A[hi] then
                invCount++
```

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

```
for lo from i to mid do
    binary search in mid + 1 to j for the value A[lo].
    Let k be the first index in the sorted array with A[k] > A[lo] or hi+1 if there is no such element.
    invCount += k-(mid+1)                          ▷ num elements in right array less than A[lo]
```

(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? **Solution:**

```
left ← lo; right ← mid + 1
while left ≤ mid ∧ right ≤ hi do
    if A[left] < A[right] then                ▷ left forms no inversions with anything right or later
        left++
    else    ▷ right forms an inversion with left and everything after left. Count those, and we're done with right
        invCount+ = (mid − left) + 1
        right++
```

(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. **Solution:**

```
function countInversions(A, lo, hi)
    if lo ≥ hi then return 0
    midpoint ← (hi − lo)/2 + lo
    inversions ← 0
    invCount+ = CountInversions(A, lo, midpoint)
    inversions+ = CountInversions(A, midpoint + 1, hi)
    invsersions+ = mergeAndCount(A, lo, hi)
    return inversions
function mergeAndCount(A, lo, hi)
    invCount ← 0
    temp ← new int[A.length]
    left ← lo; right ← mid + 1
    curr ← start
    while left ≤ mid ∧ right ≤ hi do
        if A[left] < A[right] then            ▷ left forms no inversions with anything right or later
            temp[curr++] ← A[left++]
        else▷ right forms an inversion with left and everything after left. Count those, and we're done with right
            invCount+ = (mid − left) + 1
            temp[curr++] ← A[right++]
                                              ▷ Finish moving leftover elements from unfinished array.
    while left ≤ mid do
        temp[curr++] ←temp[left++]
    while right ≤ hi do
        temp[curr++] ←temp[right++]
    Copy temp into A[lo]toA[hi]
```

(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? **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. 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.

(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)?

Solution:

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

```
function PeakFinder(A, i, j)
    if j − i ≤ 2 then
        For each i ≤ k ≤ j, check if A[k] satisfies the definition of peak in the range i..j.
        return the first element that does.
    Mid ← i + ⌊(j−i)/2⌋
    if A[Mid − 1] < A[Mid] ∧ A[Mid] < A[Mid + 1] then
        return PeakFinder(A, Mid, j)
    else if A[Mid − 1] > A[Mid] ∧ A[Mid] > A[Mid + 1] then
        return PeakFinder(A, i, Mid)
    else
        return Mid
```

For correctness, observe that $A[\texttt{Mid}-1] > A[\texttt{Mid}] \wedge A[\texttt{Mid}] < A[\texttt{Mid}+1]$ is impossible in a mountain array, so in the "else" branch, $A[\texttt{Mid}]$ is greater than both $A[\texttt{Mid}+1]$ and $A[\texttt{Mid}-1]$. We will argue by induction that if the array $A[i..j]$ is a mountain, then the return value of `PeakFinder` is the peak. For the base case, we do a brute force search, so the

IH: Suppose for all arrays where $j - i < k$ and $A[i..j]$ is a mountain that `PeakFinder`(A, i, j) returns the peak. ($k \geq 3$)

IS: Let i, j be integers such that $j - i = k$ A be an array such that $A[i..j]$ is a mountain. Since $j - i = k \geq 3$, the code goes to the recursive case. If `Mid` is the peak, then we hit the else case, and return `Mid` as required. Otherwise, we have two cases:

Case 1: `Mid` is before the peak
Then since $A[i..j]$ is a mountain, $A[\texttt{Mid}-1] < A[\texttt{Mid}] \wedge A[\texttt{Mid}] < A[\texttt{Mid}+1]$. Thus we make a recursive all on `Mid`, j. By the assumption for this case, the peak is still in the range chosen by the recursive call. Thus, the remaining array is still a mountain with the peak in the desired range. Furthermore, since $k \geq 3$, `Mid` and i are different indices, so the subarray is smaller. By IH, the result of the recursive call is therefore the peak of the subarray (and thus also of $A[i..j]$), as required.

Case 2: `Mid` is after the peak
Is symmetric to case 1, with the code making the recusive call on i..`Mid`, where the peak will be.

In both cases, we have completed the inductive step.

Running Time: We do constant work (calculating `Mid`, checking inequalities, and setting up a recursive call) before making a recursive call. The recursive call is (up to rounding) $1/2$ the size of the original array.

Thus the running time has the recurrence $T(n) = \begin{cases} T(n/2) + O(1) & \text{if } n \geq 3 \\ O(1) & \text{otherwise} \end{cases}$ which (by recognizing it as the binary-search recurrence or solving) has a closed form of $\mathcal{O}(\log n)$.

(b) No. You need to examine **every** element of the array to see if it's a mountain. To see why, suppose that you have examined all elements except for the one at index u (the "unknown" element). For simplicity, assume

that $u \neq 1$ and $u \neq n$. Furthermore, suppose that so far it is consistent with being a mountain. That is there is an index i such that $\forall 1 \leq j < i(A[j] \leq A[j+1] \vee j = u)$ and $\forall i \leq j < n(A[j] \geq A[j+1] \vee j = u)$.

We will consider two cases; in every case we show that depending on the value of $A[u]$, the array may or may not be a mountain.

Case 1: The index i is u
If $A[u]$ is set to be $\max\{A[u-1], A[u+1]\} + 1$, then all conditions will be met, as we've made u a peak (index u also satisfies $A[u] < A[u+1]$ and $A[u] > A[u+1]$, so the condition is met without needing the $j = u$ option).

If $A[u]$ is set to be $\min\{A[u-1], A[u+1]\} - 1$, then we will not satisfy the mountain property. u cannot be a peak, as $A[u-1] \not\leq A[u]$. No $i < u$ can be a peak, as $A[u] < A[u+1]$ violates the peak condition. Similarly, no $i > u$ can be a peak as $A[u-1] > A[u]$, which violates the first condition.

Case 2: The index $i \neq u$
Observe that there is only one such index i: for some i and i', with $i < i'$, for i to be a peak, $A[i]$ $A[i']$; for i' to be a peak, $A[i'] > A[i]$, and only one of these can be true.

If $u < i$, we can make A not a mountain by setting $A[u] = A[u+1] + 1$. Then $A[u] > A[u+1]$ and i is not a peak. Setting $A[u] = A[u+1]$ guarantees that u satisfies the conditions and makes it a peak. For $u > i$, setting $A[u] = A[u-1] + 1$ or $A[u] = A[u-1]$ gives a symmetric argument to the $u < i$ case.

In all cases, until we examine $A[u]$ we cannot determine whether $A[]$ is a mountain or not. Thus, we will need at least $\Omega(n)$ time to determine if the array is a mountain.

4. BFPRT Median Algorithm Running Time

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

$$T(n) = \begin{cases} 100 & \text{if } n \leq 100 \\ T\left(\lceil \frac{3n}{4} \rceil\right) + T\left(\lceil \frac{n}{5} \rceil\right) + 4n & \text{otherwise} \end{cases}$$

Prove, via induction, that $T(n) \leq 100 \cdot n$ 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.

Solution:

Base Case: For $1 \leq n \leq 200$. Since $n \geq 1$, we have $T(n) = 100 \leq 100n$ as required

IH: Suppose that $T(n) \leq 100n$ for $n = 1, 2, ..., k$ for $k \geq 200$.

IS:

$$
\begin{aligned}
T(k+1) &= T\left(\left\lceil \frac{k+1}{5} \right\rceil\right) + T\left(\left\lceil \frac{3(k+1)}{4} \right\rceil\right) + 4(k+1) \\
&\leq 100 \cdot (\left\lceil \frac{k+1}{5} \right\rceil + 100 \cdot \left\lceil \frac{3(k+1)}{4} \right\rceil + 4(k+1) && \text{by IH} \\
&\leq 100\left(\frac{k+1}{5} + 1\right) + 100\left(\frac{3(k+1)}{4} + 1\right) + 4(k+1) && \lceil x \rceil \leq x + 1 \\
&= (20(k+1) + 100) + (75(k+1) + 100) + 4(k+1) \\
&= 99(k+1) + 200 \\
&\leq 100(k+1) && \text{since } k+1 \geq k \geq 200,
\end{aligned}
$$

as required.