CSE 421 Section 4

Divide and Conquer

Administrivia

Announcements & Reminders

- HW3
  - Due Yesterday, 1/24 @ 11:59pm

- HW4
  - Due Wednesday 1/31 @ 11:59pm

Writing a Divide and Conquer Algo

Divide and Conquer

1. **Divide** instance into subparts
2. **Solve** the parts recursively
3. **Conquer** by combining the answers

The keys to this strategy:

- Come up with a baseline!
- Once you have your algo, write a recurrence for the runtime
  - Your d&c runtime should be BETTER than the baseline runtime

The Strategy (hint: it's the same as last week!)

1. Read and Understand the Problem
2. Generate Examples
3. Produce a Baseline
4. Brainstorm and Analyze Possible Algorithms
5. Write an Algorithm
6. Show Your Algorithm is Correct
7. Optimize and Analyze the Run Time

Problem 1 – Maximum Subarray Sum

Input: An array of `ints` (possibly both positive and negative)
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. Read and Understand the Problem

Reminder of the Questions to Ask:

- What is the **input type**? (Array? Graph? Integer? Something else?)

- What is your **return type**? (Integer? List?)

- Are there any **technical terms** in the problem you should pay attention to?

Reminder of the Questions to Ask:

- What is the **input type**? (Array? Graph? Integer? Something else?)

    `int[]`

- What is your **return type**? (Integer? List?)

- Are there any **technical terms** in the problem you should pay attention to?

Reminder of the Questions to Ask:

- What is the **input type**? (Array? Graph? Integer? Something else?)

  `int[]`

- What is your **return type**? (Integer? List?)

  `int` (the largest sum of any subarray)

- Are there any **technical terms** in the problem you should pay attention to?

Reminder of the Questions to Ask:

- What is the **input type**? (Array? Graph? Integer? Something else?)

  `int[]`

- What is your **return type**? (Integer? List?)

  `int` (the largest sum of any subarray)

- Are there any **technical terms** in the problem you should pay attention to?

  “subarray” means contiguous elements of the array

Key Idea with Divide and Conquer (and other recursive algorithms)

- If you identify that you want to use a recursive algorithm paradigm like Divide and Conquer, it’s not enough to just answer those key questions on the previous slide

- Since you know you will have recursive calls, you need to be explicit about what those recursive calls are giving you that, when combined together, gives you the solution you’re looking for

- You should be able to state a **clear English definition** of the return value you want to get from the recursive calls, keeping in mind the return type, the optimality, and the range & other parameters.

Problem 1.1 – Maximum Subarray Sum

What is a **clear English definition** of the return value from the recursive calls?

Work through this problem with the people around you, and then we'll go over it together!

Problem 1.1 – Maximum Subarray Sum

What is a **clear English definition** of the return value from the recursive calls?

Problem 1.1 – Maximum Subarray Sum

What is a **clear English definition** of the return value from the recursive calls?

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`.

2. Generate Examples

Good examples help with understanding now and testing later!

- You should generate two or three sample instances and the correct associated outputs.

- It’s a good idea to have some “abnormal” examples – consecutive negative numbers, very large negative numbers, only positive numbers, etc.

- *Note*: You should not think of these examples as debugging examples – null or the empty list is not a good example for this step. You can worry about edge cases at the end, once you have the main algorithm idea. You should be focused on the “typical” (not edge) case.

Problem 1.2 – Maximum Subarray Sum

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

Work through generating some examples, and then we'll go over it together!

Problem 1.2 – Maximum Subarray Sum

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

Problem 1.2 – Maximum Subarray Sum

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

$[-3, -7, -2, -10]$ 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

3. Come Up with a Baseline

Inefficient (non Divide and Conquer) First Attempt

- Review: In a time-constrained setting (like a **technical interview** or an **exam**) you often want a "baseline" algorithm. This should be an algorithm that you can implement and will give you the right answer, **even if it might be slow**.

- When you're pretty sure you want to use a Divide and Conquer algorithm, this step is **extremely** important! You need a (brute force) non Divide and Conquer baseline (with a quick runtime analysis) so you can see whether all the recursive steps of your Divide and Conquer algo are actually saving you any time!

Problem 1.3 – Maximum Subarray Sum

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).

Problem 1.3 – Maximum Subarray Sum

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).

Key idea: just check the sum of every possible subarray

Problem 1.3 – Maximum Subarray Sum

```
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 must be best here
   return bestSum
```

Problem 1.3 – Maximum Subarray Sum

```
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 must be best here
    return bestSum
```

Run-time: three for-loops going through the indices of the array, algo runs in $\mathcal{O}(n^3)$

Problem 1.3 – Maximum Subarray Sum

```
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 must be best here
   return bestSum
```

Problem 1.3 – Maximum Subarray Sum

```
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 must be best here
   return bestSum
```

Run-time: by keeping track of the partial sum, we only need two for-loops going through the indices of the array, so the algo runs in $\mathcal{O}(n^2)$

4. Brainstorm and Analyze Possible Algorithms

Think about How to Divide and Conquer

- Questions to help you brainstorm out your Divide and Conquer algo:
    - How do you want to split up the problem?
    - What is returned from the recursive calls? (hint: look back at part 1)
    - Imagine you have the answers from those recursive calls;
      what is there still to handle?

- When you have time, it’s a good idea to try to run through your idea with some of the examples you came up with earlier, and see whether you get the correct output (especially as you try to transition from your brainstorming to formalizing your algorithm)

Problem 1.4 – Maximum Subarray Sum

For each call SubarraySumDC(A[1..n]) answer these questions:
How do you want to split up the problem?

What is returned from the recursive calls?

Imagine you have the answers from those recursive calls; what is there still to handle?

Problem 1.4 – Maximum Subarray Sum

For each call SubarraySumDC(A[1..n]) answer these questions:

How do you want to split up the problem?

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 is returned from the recursive calls?

Imagine you have the answers from those recursive calls; what is there still to handle?

Problem 1.4 – Maximum Subarray Sum

For each call SubarraySumDC(A[1..n]) answer these questions:

How do you want to split up the problem?

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 is returned from the recursive calls?

Each recursive call will return the sum of the largest subarray among half the elements, either 1,…n/2 or n/2+1,…n.

Imagine you have the answers from those recursive calls; what is there still to handle?

Problem 1.4 – Maximum Subarray Sum

For each call SubarraySumDC(A[1..n]) answer these questions:

How do you want to split up the problem?

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 is returned from the recursive calls?

Each recursive call will return the sum of the largest subarray among half the elements, either 1,…n/2 or n/2+1,…n.

Imagine you have the answers from those recursive calls; what is there still to handle?

If the subarray "crosses" from one side to the other (i.e., includes both n/2 and n/2+1), it hasn't been checked yet. We still need to discover and check those.

5. Write an Algorithm

Translate the brainstorm into an algorithm!

- We need to take those ideas we were just noodling on and write them into an algorithm!
- We can start with formalizing our ideas from earlier, but then we still need to figure out how to deal with those subarrays that cross from one half to the other…

Translate the brainstorm into an algorithm!

- We need to take those ideas we were just noodling on and write them into an algorithm!
- We can start with formalizing our ideas from earlier, but then we still need to figure out how to deal with those subarrays that cross from one half to the other…

Key idea: If we know n/2 and n/2+1 are both included, then we know that $i <= n/2$ and $j >= n/2 + 1$. SO, i and j are "independent" of each other, and we can optimize for them separately. Now, we can have two separate loops instead of nested loops!

Problem 1.5 – Maximum Subarray Sum

```
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
```

6. Show Your Algorithm is Correct

Problem 1.6 – Maximum Subarray Sum

Write a proof of correctness.

Work on this proof with the people around you, and then we'll go over it together!

Problem 1.6 – Maximum Subarray Sum

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 $n < 100$, 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 $1, 2, \dots k$, $k \geq 99$.

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:

<u>Case 1</u>: 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.

Problem 1.6 – Maximum Subarray Sum

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, $i < n/2 < j$. We claim that i will be stored in `bestLeftIndex` by the end of the loop. Suppose, for the sake of contradiction, that some other index i' were stored. Then it must have been that $A[i'] + \cdots + A[n/2]$ is greater than $A[i] + \cdots + A[n/2]$. But then $A[i'] + \cdots + A[n/2] + A[n/2 + 1] + \cdots + A[j] > A[i] + \cdots + A[n/2] + A[n/2 + 1] + \cdots + A[j]$, which would make i' ... j the maximum subarray (but we assumed that i ... j was the maximum), a contradiction. Thus i is stored in `bestLeftIndex`. A symmetric argument will show that `bestRightIndex` holds j.

We then will compare `crossSum`, which contains the sum from i to j, to the values from the recursive calls. 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.

7. Optimize and Analyze the Run Time

Problem 1.7 – Maximum Subarray Sum

Write the big-O of your code and justify the running time with a few sentences.

Problem 1.7 – Maximum Subarray Sum

Write the big-O of your code and justify the running time with a few sentences.

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

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

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

That’s All, Folks!

Thanks for coming to section this week!
Any questions?