CSE 421 Section 4

Divide and Conquer

Administrivia

Announcements & Reminders

- **HW2**
  - Regrade requests are open
  - Answer keys available on Ed

- **HW3**
  - Was due yesterday, 10/16
  - Remember the **late problems** policy (NOT assignments)
    - Total of up to **10 late problem days**
    - At most **2 late days per problem**

- **HW4**
  - Due Wednesday 10/23 @ 11:59pm

Ideas for divide and conquer

Problem solving strategy overview

Read and **summarize** the problem

↓

Decide to use **known algorithm** or **techniques from scratch**

not covered this section

no idea

have idea

Solve examples to get ideas ⇄ Check that idea **isn't easily falsified** or **slow**

↓

Write pseudocode, proof, and running time analysis

…in a **"divide and conquer"** mindset

Problem 1 – Maximum subarray sum

Input: An array of integers A $= a_1, \ldots, a_n$ (possibly both positive and negative)
Expected output: The largest sum of any contiguous subarray `A[i..j]`

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

Notes:

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

For divide and conquer word problems: Summary is extremely important, because recursion demands that you understand exactly what the input and output are.

Problem solving strategy overview

Read and **summarize** the problem

Decide to use **known algorithm** or **techniques from scratch**

not covered this section

no idea

have idea

Solve examples to get ideas

Check that idea **isn't easily falsified** or **slow**

…in a **"divide and conquer"** mindset

Write pseudocode, proof, and running time analysis

Problem 1 – Maximum subarray sum

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

Input: An array of integers $a_1, \dots, a_n$ (possibly both positive and negative)
Expected output: The largest sum of any contiguous subarray A`[i..j]`

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?

Feel free to work with the people around you!

Solution

Problem 1 – Maximum subarray sum

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

Input: An array of integers $a_1, \dots, a_n$ (possibly both positive and negative)
Expected output: The largest sum of any contiguous subarray A`[i..j]`

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?

Check the sum of every possible subarray 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 $O(n^3)$.

Problem 1 – Maximum subarray sum

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?

Previous Solution

Problem 1 – Maximum subarray sum

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?

Check the sum of every possible subarray `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?

Feel free to work with the people around you!

Solution

Problem 1 – Maximum subarray sum

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?

Check the sum of every possible subarray `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?

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

Problem solving strategy overview

Read and **summarize** the problem

Decide to use **known algorithm** or **techniques from scratch**

not covered this section

no idea

have idea

Solve examples to get ideas

Check that idea **isn't easily falsified** or **slow**

Write pseudocode, proof, and running time analysis

…in a **"divide and conquer"** mindset

Problem 1 – Maximum subarray sum

Now, we know that $\boldsymbol{O(n^2)}$ **is easy**. Thus, we should **aim around $\boldsymbol{O(n \log n)}$**.

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. How much work can you do in each call, in order to get $O(n \log n)$?

Feel free to work with the people around you!

Solution

Problem 1 – Maximum subarray sum

Now, we know that $\boldsymbol{O(n^2)}$ **is easy**. Thus, we should **aim around** $\boldsymbol{O(n \log n)}$.

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

i. How do you want to split up the problem?

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

ii. What is returned from the recursive calls?

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 $O(n \log n)$?

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

Problem 1 – Maximum subarray sum

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

Feel free to work with the people around you!

Continue trying more examples until you have an idea.

Solution

Problem 1 – Maximum subarray sum

i. 2, -10, -5, 8, -1, 7

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.

Solution

Problem 1 – Maximum subarray sum

ii. 6, -3, -4, 4, 2, 1, -7, 5

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.

Solution

Problem 1 – Maximum subarray sum

iii. -3, 2, 4, -1, 3, -10, 6, -4

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.

Previous Solution

Problem 1 – Maximum subarray sum

The largest subarray sum is either in the left or right half, or crosses the boundary.

So, can we find the largest subarray sum that crosses the boundary in $O(n)$ time?

If you haven't gotten it yet, take a moment to think.
Feel free to work with the people around you!

Solution

Problem 1 – Maximum subarray sum

The largest subarray sum is either in the left or right half, or crosses the boundary.

So, can we find the largest subarray sum that crosses the boundary in $O(n)$ time?

Yes! From the middle, search down for the largest sum of all arrays of the form `A[i..m]` (where $1 \leq i \leq m$), and similarly search up for arrays of the form `A[m+1..j]` (where $m + 1 \leq j \leq n$), then put them together.

Writing about divide and conquer

Divide and conquer pseudocode

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 other equivalents.
        - These slides use global variables, but it's subjective.
    - Not relevant for this problem since we use $O(n)$ time anyways.

Problem 1 – Maximum subarray sum

d) Write the pseudocode for your solution.

Previous Solution

Problem 1 – Maximum subarray sum

The largest subarray sum is either in the left or right half, or crosses the boundary.

For crossing the boundary, from the middle, search down for the largest sum of all arrays of the form `A[i..m]` (where $1 \leq i \leq m$), and similarly search up for arrays of the form `A[m+1..j]` (where $m + 1 \leq j \leq n$), then put them together.

d) Write the pseudocode for your solution.

Feel free to work with the people around you!

Solution

Problem 1 – Maximum subarray sum

function MAXSUBARRAYSUM($\texttt{A}[1\,..\,n]$)
  if $n = 1$ **then**
    return $\texttt{A}[1]$ if it is positive, or 0 otherwise

  $m \leftarrow \lfloor \frac{n}{2} \rfloor$
  $\texttt{maxSumToMiddle} \leftarrow$ largest sum of any subarray of type $\texttt{A}[i\,..\,m]$
  $\texttt{maxSumFromMiddle} \leftarrow$ largest sum of any subarray of type $\texttt{A}[m+1\,..\,j]$
  $\texttt{crossSum} \leftarrow \texttt{maxSumToMiddle} + \texttt{maxSumFromMiddle}$

  return $\max(\texttt{crossSum}, \text{MAXSUBARRAYSUM}(\texttt{A}[1\,..\,m]), \text{MAXSUBARRAYSUM}(\texttt{A}[m+1\,..\,n]))$

Solution

Problem 1 – Maximum subarray sum

```
function MaxSubarraySum(A[1..n])
    if n = 1 then
        return A[1] if it is positive, or 0 otherw

    m ← ⌊n/2⌋
    maxSumToMiddle ← largest sum of any subarray of type A[i..m]
    maxSumFromMiddle ← largest sum of any subarray of type A[m+1..j]
    crossSum ← maxSumToMiddle + maxSumFromMiddle

    return max(crossSum, MaxSubarraySum(A[1..m]), MaxSubarraySum(A[m+1..n]))
```

```
partialSum ← A[m]
maxSumToMiddle ← partialSum
for i ← m − 1 down to 1 do
    partialSum ← partialSum + A[i]
    if partialSum > maxSumToMiddle then
        maxSumToMiddle ← partialSum
```

Solution

Problem 1 – Maximum subarray sum

function MAXSUBARRAYSUM(A$[1\,..\,n]$)
  if $n = 1$ **then**
    return A$[1]$ if it is positive, or 0 otherwise

  $m \leftarrow \lfloor \frac{n}{2} \rfloor$
  maxSumToMiddle $\leftarrow$ largest sum of any subarray of type A$[i\,..\,m]$
  maxSumFromMiddle $\leftarrow$ largest sum of any subarray of type A$[m+1\,..\,j]$
  crossSum $\leftarrow$ maxSumToMiddle $+$ maxSumFromMiddle

  return max(crossSum, MAXSUBARRAYSUM(A$[1\,..\,m]$), MAXSUBARRAYSUM(A$[m+1\,..\,n]$))

This passes the array by value!
Fine this time because we are doing $O(n)$ work per iteration anyway, but would be bad if we trying to be faster.

Solution

Problem 1 – Maximum subarray sum

global $\texttt{A}[1..n]$
function $\textsc{MaxSubarraySum}(a, b)$
    if $a = b$ **then**
        return $\texttt{A}[a]$ if it is positive, or 0 otherwise

    $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
    $\texttt{maxSumToMiddle} \leftarrow$ largest sum of any subarray of type $\texttt{A}[i..m]$, where $i \geq a$
    $\texttt{maxSumFromMiddle} \leftarrow$ largest sum of any subarray of type $\texttt{A}[m+1..j]$, where $j \leq b$
    $\texttt{crossSum} \leftarrow \texttt{maxSumToMiddle} + \texttt{maxSumFromMiddle}$

    return $\max(\texttt{crossSum}, \textsc{MaxSubarraySum}(a, m), \textsc{MaxSubarraySum}(m+1, b))$
output $\textsc{MaxSubarraySum}(1, n)$

For future reference, we can fix it by using a global variable and passing indices instead, which takes $O(1)$ time!

Divide and conquer proofs

Reminders for divide and conquer proofs:

- Always use **strong induction**. Your IH should be:
  "My core 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, AND
  - If the input is "X such that Y holds", explain why Y holds for recursive calls.

Problem 1 – Maximum subarray sum

e) Write the proof that your pseudocode works.

Previous Solution

Problem 1 – Maximum subarray sum

e) Write the proof that your pseudocode works.

function $\textsc{maxSubarraySum}(\mathtt{A}[1\,..\,n])$
    if $n = 1$ **then**
        return $\mathtt{A}[1]$ if it is positive, or 0 otherwise

    $m \leftarrow \lfloor \frac{n}{2} \rfloor$
    $\mathtt{maxSumToMiddle} \leftarrow$ largest sum of any subarray of type $\mathtt{A}[i\,..\,m]$
    $\mathtt{maxSumFromMiddle} \leftarrow$ largest sum of any subarray of type $\mathtt{A}[m+1\,..\,j]$
    $\mathtt{crossSum} \leftarrow \mathtt{maxSumToMiddle} + \mathtt{maxSumFromMiddle}$

    return $\max(\mathtt{crossSum}, \textsc{maxSubarraySum}(\mathtt{A}[1\,..\,m]), \textsc{maxSubarraySum}(\mathtt{A}[m+1\,..\,n]))$

Feel free to work with the people around you!

Solution

Problem 1 – Maximum subarray sum

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, we find this subarray and return it.

Case 2: The maximum subarray crosses from the left to the right.

- All subarrays `A[i..j]` that cross can be divided into `A[i..m]` and `A[m+1..j]`.
- But we know that `maxSumToMiddle` ≥ `sum(A[i..m])` and similarly `maxSumFromMiddle` ≥ `sum(A[m+1..j])`.
- Adding these, `crossSum` ≥ `sum(A[i..j])` for all subarrays `A[i..j]` that cross, and it certainly represents *some* subarray, so it is the max subarray sum.

Solution

Problem 1 – Maximum subarray sum

BC: Th… ve.

IH: MA… $\leq k$.

IS: Let …

Note how cases are only inspired by code, **not regurgitating** code.

The **cases are high-level**: what kinds of inputs end up at each branch? NOT necessarily the specific criteria you check in code.

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

Case 2: The maximum subarray crosses from the left to the right.

- All subarrays `A[i..j]` that cross can be divided into `A[i..m]` and `A[m+1..j]`.
- But we know that `maxSumToMiddle` $\geq$ `sum(A[i..m])` and similarly `maxSumFromMiddle` $\geq$ `sum(A[m+1..j])`.
- Adding these, `crossSum` $\geq$ `sum(A[i..j])` for all subarrays `A[i..j]` that cross, and it certainly represents *some* subarray, so it is the max subarray sum.

Solution

Problem 1 – Maximum subarray sum

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

Here is a sample BAD proof:

aximum subarray sum for all arrays of length $\leq k$.

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

We compute `maxSumToMiddle` and `maxSumFromMiddle`, the largest sum of any subarray of type `A[i..m]` and `A[m+1..j]`, respectively. Then, we add them together to get `crossSum`, and return the biggest between `crossSum`, MAXSUBARRAYSUM`(A[1..m])`, and MAXSUBARRAYSUM`(A[m+1..n])`.

Case 1: `crossSum` was the biggest.
Since we were asked to return the maximum, we returned it, which was correct.

Case 2: MAXSUBARRAYSUM`(A[1..m])` was the biggest. …

Solution

Problem 1 – Maximum subarray sum

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

Here is a sample BAD proof:

aximum sub ... $\leq k$.

Unnecessarily regurgitates the pseudocode

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

We compute `maxSumToMiddle` and `maxSumFromMiddle`, the largest sum of any subarray of type `A[i..m]` and `A[m+1..j]`, respectively. Then, we add them together to get `crossSum`, and return the biggest between `crossSum`, MAXSUBARRAYSUM`(A[1..m])`, and MAXSUBARRAYSUM`(A[m+1..n])`.

Case 1: `crossSum` was the biggest.
Since we were asked to return the maximum, we returned it, which was correct.

Case 2: MAXSUBARRAYSUM`(A[1..m])` was the biggest. …

Solution

Problem 1 – Maximum subarray sum

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

Here is a sample BAD proof:

aximum subarray sum for all arrays of length $\leq k$.

Doesn't explain why `crossSum` is bigger than all other subarray sums (only that it's bigger than the recursive calls).

ma

1 .

h th

Also, these cases are **low-level** criteria copied from the code. It's possible to make them work, but they will be **much wordier**.

m of any

em

`MAXSUBARRAYSUM(A[1..m])`, and `MAXSUBARRAYSUM(A[m+1..n])`.

<u>Case 1</u>: `crossSum` was the biggest.
Since we were asked to return the maximum, we returned it, which was correct.

<u>Case 2</u>: `MAXSUBARRAYSUM(A[1..m])` was the biggest. …

Problem 1 – Maximum subarray sum

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

Previous Solution

Problem 1 – Maximum subarray sum

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

function MAXSUBARRAYSUM($\texttt{A}[1\,..\,n]$)
    if $n = 1$ **then**
        return $\texttt{A}[1]$ if it is positive, or 0 otherwise

    $m \leftarrow \lfloor \frac{n}{2} \rfloor$
    `maxSumToMiddle` $\leftarrow$ largest sum of any subarray of type $\texttt{A}[i\,..\,m]$
    `maxSumFromMiddle` $\leftarrow$ largest sum of any subarray of type $\texttt{A}[m+1\,..\,j]$
    `crossSum` $\leftarrow$ `maxSumToMiddle` + `maxSumFromMiddle`

    return max(`crossSum`, MAXSUBARRAYSUM($\texttt{A}[1\,..\,m]$), MAXSUBARRAYSUM($\texttt{A}[m+1\,..\,n]$))

Feel free to work with the people around you!

Solution

Problem 1 – Maximum subarray sum

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

Lines 5 and 6 take $O(n)$ time each.

Since we then 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 $T(n) = O(n \log n)$.

Final thoughts

- How to choose between divide and conquer vs. greedy?
  - Try easy algorithms first, like baselines or greedy.
  - If easy ones are slow and subproblems seem useful, try divide and conquer.
- Sometimes, it will be useful to **compute more than what's asked for**.
  - Examples:
    - Problem 2 in your section packet
    - Problem 2 on your homework: today's problem in $O(n)$! It will guide you.
  - In this case, your **IH should reflect what you actually compute**, not what you were asked to compute.
  - **Try the usual thing first**, only compute more if it doesn't work/is too slow.

Summary

- First, try an easy but slow **baseline** algorithm.
  - Use this to estimate how much time you can take per recursive call, in order to still get an improvement.
- Ask yourself: **How can I use answers to subproblems to find the full answer?**
- Keep in mind the cost of copying arrays, and avoid this with global variables.
- Prove using **strong induction**.

Thanks for coming to section this week!