Solution Set

CIS 1210—Data Structures and Algorithms—Fall 2023

Divide & Conquer—Tuesday, September 19th / Wednesday, September 20th

Readings

- Lecture Notes Chapter 7: Divide & Conquer and Recurrence Relations

Review: Divide & Conquer and Recurrence Relations

What does it mean to have a "Divide & Conquer" algorithm?

Divide the original problem into subproblems that are smaller-sized instances of the same problem.
Conquer the subproblems by solving them with recursion.
Combine the solutions to the subproblems into the solution for the original problem.

How do you recognize problems where "Divide & Conquer" might work? A natural first question is "Can I break this down into subproblems equivalent to the original problem?" You can then ask "How can I solve these subproblems and *combine* these solutions to reach a solution for my original problem?" In order to better illustrate the "Divide & Conquer" paradigm, we will do an in-depth study on a familiar algorithm: **Mergesort**.

Applying Divide & Conquer to Mergesort

We can apply the principles of Divide & Conquer when thinking about the problem of sorting an array:

Divide Can we divide this problem into equivalent subproblems? Yes, we can divide the array into two halves, each half a subarray of size $\frac{n}{2}$. Thus, each half of the array is an equivalent subproblem.

Conquer How can we solve our subproblems? That is, how can we sort both halves of the array? Since each half of the array is an equivalent subproblem, we can sort each half by recursively calling Mergesort on each half of the array until we hit our base case of a subarray with a single element, which we know by definition is already sorted. Note that when applying "Divide & Conquer," our base case must also be an equivalent subproblem.

Combine How can I combine the solutions to my subproblems into a solution for the original problem? After Mergesort has finished recursing on both halves of the array, we have two sorted subarrays of size $\frac{n}{2}$. We can leverage how both halves of the array are already sorted, combining them into a sorted array to solve the original problem by interleaving the two halves in $O(n)$ time.

Runtime Analysis of Mergesort

As you have seen in lecture, **recurrence relations** are equations used to describe the running time of a recursive algorithm. To derive recurrence relations for "Divide & Conquer" algorithms, we usually need to consider the size of subproblem(s) recursed on; the number of recursive calls made; the amount of work done per "level" to combine the solutions to our subproblems; and the amount of work done at the base case.

Let $T(n)$ represent the time Mergesort takes on an input of size n. The algorithm divides the original problem into 2 subproblems of size $\frac{n}{2}$ — each of which takes $T(\frac{n}{2})$ time to solve since each is half the size of the original problem — and makes 2 recursive calls to Mergesort, one call to solve each subproblem. At each level, we perform linear work by merging the two sorted halves of the array; at the base case, we perform constant work by returning a singleton array. Thus, the recurrence relation for Mergesort can be expressed as

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 2 \\ 1 & n = 1 \end{cases}$$

Solving this yields a running time of $\Theta(n \log n)$.

Problems

Problem 1: Element-Index Matching

You are given a sorted array of n *distinct* integers A$[1...n]$. Design an $O(\lg n)$ time algorithm that either outputs an index i such that A$[i] = i$ or correctly states that no such index i exists.

Solution

A brute force solution is to iterate through the array, checking if $A[i] = i$ at each index, but this runs in $O(n)$ time. Because we have a sorted array, the intuition here is that we can improve our runtime by leveraging a modified binary search approach (since binary search requires a sorted array and runs in $O(\lg n)$ time).

Algorithm: Consider an array $A[\text{lo...hi}]$. If lo = hi, then the subarray is of size 1, so return whether $A[\text{lo}] = \text{lo}$. Otherwise, find the (lower) middle element of the array at index $m = \lfloor \frac{\text{lo+hi}}{2} \rfloor$. If $A[m] = m$, then return m. If not, if $A[m] < m$, recurse on $A[m + 1...\text{hi}]$; otherwise, recurse on $A[\text{lo}...m - 1]$.

Proof of Correctness: The correctness of our algorithm follows via strong induction.

Base Case: If the subarray is of size 1, we correctly return whether its only element is equal to its index.

Induction Hypothesis: Assume that our algorithm holds for any subarray of size j for $1 \leq j \leq k$.

Induction Step: We want to show that our algorithm holds for an array $A[\text{lo...hi}]$ of size $k+1$. Let m be the (lower) middle element index. If $A[m] = m$, then we correctly return m. Otherwise, since $A[m] \neq m$ and the array is sorted with distinct integers, we show that we correctly narrow down where a potential solution would be in the array.

WLOG, let $A[m] < m$ (the case where $A[m] > m$ follows similar logic). It is impossible to find an index $i < m$, where $i = m - \ell$ (for some arbitrary positive ℓ) such that $A[i] = i$ because for all ℓ,

$$A[m - \ell] \leq A[m] - \ell < m - \ell.$$

Note that the $A[m - \ell] \leq A[m] - \ell$ comes from the logic that if an element is ℓ spots to the left of index m, the value $A[m - \ell]$ must also be less than $A[m]$ by at least ℓ since the elements are distinct in the array. Because $A[m] < m$, we also know that $A[m] - \ell < m - \ell$, so putting together these two inequalities yields the above math. Thus, none of the elements before index m can be index-matching elements, so we can safely reduce this to an equivalent subproblem by recursing on the other half of the array. By applying IH to $A[m + 1...\text{hi}]$, we know our algorithm correctly returns a solution or determines that one does not exist through recursion.

Runtime Analysis: The algorithm divides the original problem into 1 subproblem of size $\frac{n}{2}$ which takes $T(\frac{n}{2})$ time to solve since it is half the size of the original problem. At each level, we perform constant work by checking whether $A[m] = m$; at the base case, we perform constant work by returning a singleton array if $A[\text{lo}] = \text{lo}$. Therefore, the running time of our algorithm is given by the following recurrence.

$$T(n) = \begin{cases} T(\frac{n}{2}) + c & n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Solving this recurrence yields a running time of $O(\lg n)$.

Problem 2: Local Maximum

You are given an integer array $A[1..n]$ with the following properties:

- Integers in adjacent positions are different
- $A[1] < A[2]$
- $A[n-1] > A[n]$

A position i is referred to as a local maximum if $A[i-1] < A[i]$ and $A[i] > A[i+1]$. You may assume $n > 2$.

Example: You have an array $[0, 1, 5, 3, 6, 3, 2]$. There are multiple local maxes at 5 and 6.

Design an $O(\lg n)$ algorithm to find a local maximum and return its index.

Solution

A brute force solution is to iterate through the array and check whether each element satisfies the definition of a local maximum until we find a possible solution, but this runs in $O(n)$ time. The intuition here is that **any faster algorithm cannot check every element in the array**, making a Divide & Conquer strategy useful here.

Algorithm: Consider an array $A[\text{lo}...\text{hi}]$. If the subarray is of size 3, return the middle element index. Otherwise, find the (lower) middle element of the array at index $m = \lfloor \frac{\text{lo}+\text{hi}}{2} \rfloor$. If that element satisfies the properties of a local maximum, return m. If not, this element must have a larger neighbor. If its right neighbor is larger, then recurse on the subarray $A[m...\text{hi}]$; otherwise, recurse on the subarray $A[\text{lo}...m]$.

Proof of Correctness: Given the properties of the array, observe that there *must* be a local maximum. Intuitively, consider a maximum element in the array. Since neighbors must be distinct and endpoints cannot be maximum, we are guaranteed that such an element will be a local maximum. (It is also a good follow-up exercise to think about how to prove this rigorously!) Now, the correctness of our algorithm follows via strong induction.

Base Case: If the subarray is of size 3, then the middle element index must be a local maximum, since it is an equivalent subproblem and invariants state that endpoints cannot be maximum. Thus, the base case holds.

Induction Hypothesis: Assume that our algorithm holds for any subarray of size j for $3 \le j \le k$.

Induction Step: We want to show that our algorithm holds for an array $A[\text{lo}...\text{hi}]$ of size $k+1$. Let m be the (lower) middle element index. If m is a local maximum, then we correctly return its index. Otherwise, since elements in adjacent positions are different and since m is not a local maximum, one of its neighbors is larger. WLOG, let its right neighbor be larger (the case where its left neighbor is larger is similar). Consider $A[m...\text{hi}]$. Because its endpoints are smaller than their neighbors and every pair of adjacent elements are distinct, it satisfies the invariants of the input array and is an equivalent subproblem, so it must have a local maximum. By applying IH to $A[m...\text{hi}]$, we know our algorithm correctly finds a local maximum via recursion.

Runtime Analysis: The running time of our algorithm is given by the following recurrence.

$$T(n) = \begin{cases} T(\frac{n}{2}) + c & n > 3 \\ 1 & \text{otherwise} \end{cases}$$

Solving this recurrence yields a running time of $O(\lg n)$.

Problem 3: Largest Continuous Sum

You are given an integer array, with both positive and negative elements. Design an $O(n \lg n)$ algorithm to return the sum of the continuous subarray with the maximum sum.

Solution

A brute force solution uses two for-loops. The outer loop runs through the elements in the array, while the inner loop finds the maximum continuous subarray sum starting at the current outer loop element. If this sum is bigger than the current best maximum, we update a global variable, but this runs in $O(n^2)$ time.

Algorithm: Consider an array A[lo...hi]. If lo = hi and the subarray is of size 1, return A[lo]. Otherwise, find the (lower) middle element index $m = \lfloor \frac{\text{lo}+\text{hi}}{2} \rfloor$. Recurse on A[lo...m]; recurse on $A[m+1$...hi]; and find the largest continuous subarray sum that spans both halves of the array by combining the maximum sum directly to the left of m and the maximum sum directly to the right of m. Finally, return the maximum of the three values computed. Formally, the pseudocode is as follows:

```
LargestContinuousSum($A$, lo, hi)
if lo = hi then
   return $A$[lo]

mid ← $\lfloor \frac{\text{lo + hi}}{2} \rfloor$
leftSum ← LargestContinuousSum($A$, lo, mid)
rightSum ← LargestContinuousSum($A$, mid+1, hi)

// Find max sum directly to right of middle index
sum ← 0
middleRightSum ← $-\infty$
for ($i$ = mid+1;  $i \leq$ hi; i++)
     sum = sum + $A[i]$
     middleRightSum = max(sum, middleRightSum)

// Find max sum directly to left of middle index
sum = ← 0
middleLeftSum ← $-\infty$
for ($i$ = mid;  $i \geq$ lo; $i--$)
     sum = sum + $A[i]$
     middleLeftSum = max(sum, middleLeftSum)

middleSum ← middleLeftSum + middleRightSum
return max(leftSum, rightSum, middleSum)
```

Proof of Correctness: The correctness of our algorithm follows via strong induction.

Base Case: If the subarray is of size 1, then its largest continuous subarray sum is just its only element, which we correctly return.

Induction Hypothesis: Assume that our algorithm holds for any subarray of size $1 \leq j \leq k$.

Induction Step: We want to show that our algorithm holds for an array A[lo...hi] of size $k+1$. Observe that there are only 3 cases for where the largest continuous subarray sum can lie: it either lies completely in the left half of the array; lies completely in the right half of the array; or spans across both halves of the array. Hence, the maximum of the continuous subarray sums in each of these 3 regions will be our solution.

We correctly find the largest continuous subarray sum in the left half of the array via recursion by applying IH to $A[\text{lo}...m]$, and we correctly find the largest continous subarray sum in the right half of the array via recursion by applying IH to $A[m+1...\text{hi}]$. We find the largest continuous subarray sum in the left half by starting from the middle element index and iterating left and the largest continuous subarray sum in the right half by starting from the middle element index and iterating right; concatenating both subarray sums yields the largest continuous subarray sum that spans across both halves of the array. Therefore, returning the maximum of these 3 computed values yields the correct solution.

Runtime Analysis: The running time of our algorithm is given by the following recurrence.

$$T(n) = \begin{cases} 2 \cdot T(\frac{n}{2}) + cn & n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Solving this recurrence yields a running time of $O(n \lg n)$.