CS381 Homework 0 Problem 1

Connor Couetil

January 22, 2026

1 Exercise 1.3.1

You can tell where you are in a rotated array by comparing against the first element of the array. That way, you tell if you are in the rotated portion or merely the shifted portion.

Recursive Algorithm

RotationIndex(*array*, *start*, *end*)

1. If $start == end$
   A. Return 1
2. $midpoint = \lceil (end - start + 1)/2 \rceil + start$
3. $neighbor = (midpoint - 1) \bmod \text{Length}(array)$
4. If $array[midpoint] < array[neighbor]$
   A. Return *midpoint*
5. If $array[midpoint] > array[1]$
   A. Return RotationIndex(*array*, *midpoint*, *end*)
6. Return RotationIndex(*array*, *start*, *midpoint*)

Runtime Analysis O(log n) as we are using binary search to move through the array. The recurrence relation $\approx S(n) = S(n/2) + c$, $S(1) = C$, n approaches the base cases logarithmically.

2 Exercise 1.3.2

For every point in a binary search, we check the neighbors of the midpoint for a peak, otherwise we follow the upslope to the peak of the mountain.

Recursive Algorithm

MountainIndex(*array*, *start*, *end*)

1. If $start == end$
   A. Return 1
2. $midpoint = \lceil (end - start + 1)/2 \rceil + start$
3. $left = (midpoint - 1) \bmod \text{Length}(array)$

4. $right = (midpoint - 1) \bmod \text{Length}(array)$
5. If $array[midpoint] > array[left]$ and $array[midpoint] > array[right]$
   A. Return $midpoint$
6. If $array[midpoint] > array[left]$
   A. Return MountainIndex($array$, $midpoint$, end)
7. Return MountainIndex($array$, $start$, $midpoint$)

Runtime Analysis O(log n) as we are using binary search to move through the array. The recurrence relation $\approx S(n) = S(n/2) + C$, $S(1) = C$, n approaches the base cases logarithmically, it's halved at each step of the recursion.

3 Exercise 1.3.3

Recursive Algorithm The heuristic is to follow a downslope or unslope until the local minimum. If we follow the slope, we either meet a boundary condition, or we must meet a point at which the slope turns.

Assuming the array has a non-zero length.

`LocalMinimum(`$array$`, `$start$`, `$length$`)`

1. If $length == 1$
   A. Return $start$
2. If $array[start] < array[start + 1]$
   A. Return $start$
3. If $array[start + length - 1] < array[start + length - 2]$
   A. Return $start + length - 1$
4. return Search($array$, $start$, $length$) `// into recursion`

`Search(`$array$`, `$start$`, `$length$`)`

1. $midpoint = start + length/2$
2. $array[midpoint] < array[midpoint - 1]$ and $array[midpoint] < array[midpoint + 1]$
   A. Return $midpoint$
3. If $array[midpoint] > array[midpoint + 1]$ `// downslopes to right`
   A. Return Search($array$, $midpoint$, $length/2$)
4. Return Search($array$, $start$, $length/2$) `// upslopes to right`

Runtime Analysis O(log n) as we are using binary search to move through the array. The recurrence relation $\approx S(n) = S(n/2) + C$, $S(1) = C$, n approaches the base cases logarithmically, it's halved at each step of the recursion.

4 Exercise 1.3.4

When you realize any array can be split and the sub array will exhibit the same odd gap property, the solution becomes a recursive binary search where you follow the odd arrays to the final odd gap.

Recursive Algorithm

OddGap(*array, start, length*)

1. If $length == 2$
   A. Return *start*
2. $midpoint = start + length/2$
3. $lparity = array[midpoint] - array[start]$
4. If $lparity \bmod 2 == 1$
   A. Return OddGap(*array*, *start*, $length/2$)
5. Return OddGap(*array*, $start + length/2$, $length/2$)

Runtime Analysis O(log n) as we are using binary search to move through the array. The recurrence relation $\approx S(n) = S(n/2) + C$, $S(2) = C$, n approaches the base cases logarithmically, it's halved at each step of the recursion.

5 Exercise 1.3.5

Much like the previous problem, any subarrays must have the same property hold. So, we examine candidate jump pairs recursively, following the subarray where the final element is greater than the first, until we settle at the final jump pair.

Recursive Algorithm

Jump(*array, start, length*)

1. If $length == 2$
   A. Return *start*
2. $midpoint = start + length/2$
3. $jump = array[midpoint] - array[start]$
4. If $jump > 0$
   A. Return Jump(*array*, *start*, $length/2$)
5. Return Jump(*array*, $start + length/2$, $length/2$)

Runtime Analysis O(log n) as we are using binary search to move through the array. The recurrence relation $\approx S(n) = S(n/2) + C$, $S(2) = C$, n approaches the base cases logarithmically, it's halved at each step of the recursion.

6 Exercise 1.3.6

A comparison query has 2 possible outcomes. We have n elements sorted, which means the number of possible starting points for a search is n. So, we must have at least 2^k comparisons for n elements. This is expressed as $2^k \geq n$, which may be represented as $k \geq log(n)$, meaning the smallest value for k is $log(n)$, or that at least $log(n)$ comparisons must be made for n elements, and that $\Omega(log(n))$ is thus the lower bound for search on a sorted list.