Pset 4

Darwin Do

March 8, 2022

(a) Darwin Do

(b) 919941748

(c) Collaborators:

(d) I have followed the academic integrity and collaboration policy

(e) Hours:

1 Finding Element

(a) We can solve this problem using binary search. We split our input of size n into 2 groups of size $n/2$ with a "partition" element dividing between the two groups. If the partition element equals the target, we return the index of the partition. If the partition is less than the target, we recurse on the right group. Otherwise, the partition is greater than the target and we recurse on the left group.

This algorithm works as it relies on the fact that successive elements can only change by 1 and $p \leq t \leq q$. Let's define the partition element as x. If $x \leq t$, then $x \leq q$ and t has to appear somewhere in between x and q as every value between x and q will be reached as we increase by 1 to get to q. The same logic applies for the $x \geq t$ case, t would have to appear between p and x.

(b) $\star$ Note that I'm using array inclusive array range notation (i.e. $A[1:x]$) means all elements in the array A from the first element to element x, including element x.

function FindElement(A, t)
  $n =$ length(A)
  $x = A[\lceil \frac{n}{2} \rceil]$
  if $x == t$ **then return** $\frac{n}{2}$
  else if $x > t$ **then return** FindElement($A[1 : \frac{n}{2} - 1], t$)
  else
   return FindElement($A[\frac{n}{2} + 1 : n], t$)

(c) $T(n) = T(\frac{n}{2}) + 1$

(d)

$$A = 1, B = 2, D = 0$$
$$R = \frac{A}{B^D} = \frac{1}{2^0} = 1$$

So according to the Master Theorem: $T(n) = O(n^D \log n) = O(\log n)$

2 Maximum Sum

(a) We use a divide and conquer algorithm that divides each input into 2 parts with size $n/2$. The algorithm keeps doing this until we reach the base case where $n = 1$. In this case, we simply return the value of the element. Otherwise, for the "conquering" step of the algorithm, we let L equal the solution for the left part and R equal the solution for the right part as returned by our recursive calls. We also define LS as the largest sum of the left part starting from $A[\lceil n/2 \rceil]$ to $A[0]$ and RS as the largest sum of the right part starting from $A[\lceil n/2 \rceil + 1]$ to $A[n]$. We then return the maximum of the following options, L, R, or $LS + RS$.

This algorithm works as each recursive call will always return the biggest contiguous sum within its input, whether that be the sum from the left side, the sum from the right side, or a sum that spans between the two sides.

(b)
function MaxSum(A)
  $n =$ length(A)
  if $n == 1$ **then return** $A[1]$
  $M = \lceil \frac{n}{2} \rceil$
  $L =$ MaxSum($A[1 : M]$)
  $R =$ MaxSum($A[M + 1 : n]$)
  $LS = 0; RS = 0;$
  $tempTotal = 0$
  for i from M to 1 **do**
    $tempTotal = tempTotal + A[i]$
    if $tempTotal > LS$ **then**
      $LS = tempTotal$
  $tempTotal = 0$
  for i from $M + 1$ to n **do**
    $tempTotal = tempTotal + A[i]$
    if $tempTotal > RS$ **then**
      $RS = tempTotal$
  return max($L, R, LS + RS$)

(c) $T(n) = 2T(\frac{n}{2}) + O(n)$

(d)

$$A = 2, B = 2, D = 1$$
$$R = \frac{A}{B^D} = \frac{2}{2^1} = 1$$

So according to the Master Theorem: $T(n) = O(n^D \log n) = O(n \log n)$

3 Longest Common Substring

(a)

4 Infinitely Many Rods

(a)

5 At Most One Rod

(a)