Abraham Cardenas

CMPS 102 — Winter 2019 – Homework 1 revised 1/12

Four problems, 36 points, due Wednesday Jan. 23rd, see the *Homework Guidelines*

1. (10 pts)

```
1:  function G-SORT($A$, $n$)            ▷ takes as input an array of $n$ numbers, $A[0..n-1]$
2:      G-sort-recurse($A$, 0, $n-1$)
3:  end function
4:  function G-SORT-RECURSE($A$, $\ell$, $u$)
5:      if $u - \ell \leq 0$ then
6:          return                        ▷ 1 or fewer elements already sorted
7:      else if $u - \ell = 1$ then       ▷ 2 elements
8:          if $A[u] < A[\ell]$ then      ▷ swap values
9:              temp ← $A[u]$
10:             $A[u] \leftarrow A[\ell]$
11:             $A[\ell]$ ← temp
12:         end if
13:     else                              ▷ 3 or more elements
14:         size ← $u - \ell + 1$
15:         twothirds ← $\lceil(2 * \text{size})/3\rceil$
16:         G-sort-recurse($A, \ell, \ell +$ twothirds $- 1$)
17:         G-sort-recurse($A, u -$ twothirds $+ 1, u$)
18:         G-sort-recurse($A, \ell, \ell +$ twothirds $- 1$)
19:     end if
20: end function
```

Claim: The algorithm correctly sorts the numbers in an array A of size n (in ascending order).

Induction Proof:

Base Case: First, our base case occurs when we have an array A of size ≤ 1 or size $= 2$ (handled in the algorithm on lines 5-6 and 7-12 respectively). The first base case is handled by simply returning because by definition, an array of size ≤ 1 is already sorted. The second base case is handled by swapping $A[u]$ and $A[\ell]$ if $A[u] < A[\ell]$. As a result of this swapping, the algorithm sorts all values in ascending order. So therefore, in both cases, the algorithm works.

Inductive Step: We need to show that the algorithm works for an array A of size ≥ 3. Let's assume that the algorithm works for any array A of size $\leq n-1$. Let us prove that the algorithm works for an array A of size n.

After line 16 is executed, $A[\ell \text{ .... } (\ell + twothirds - 1)]$ is sorted. This implies that

(1) $A[(u - twothirds + 1) \text{ .... } (\ell + twothirds - 1)] \geq A[\ell \text{ .... } (u - twothirds)]$

(2) $A[(u - twothirds + 1) \text{ .... } u]$ has at least $u - \ell - (2 * twothirds) + 1$ number of elements in where each is no smaller than every element in $A[\ell \text{ .... } (u - twothirds)]$.

After line 17 is executed, $A[(u - twothirds + 1) \text{ .... } u]$ is sorted. This implies

(3) $A[(\ell + twothirds) \text{ .... } u]$ is sorted, and that

(4) $A[(\ell + twothirds) \;....\; u] \geq A[(u - twothirds + 1) \;....\; (\ell + twothirds - 1)]$

Using (2) and the fact that the size of $A[(\ell + twothirds) \;....\; u] \leq u - \ell - (2 * twothirds) + 1$, we can conclude that

(5) $A[(\ell + twothirds) \;....\; u] \geq A[\ell \;....\; (u - twothirds)]$

Using (4) and (5), we can conclude that

(6) $A[(\ell + twothirds) \;....\; u] \geq A[\ell \;....\; (\ell + twothirds - 1)]$

After line 18 is executed, $A[\ell \;....\; (\ell + twothirds - 1)]$ is sorted. Using (3) and (6), we can conclude that $A[\ell \;....\; u]$ is sorted. And since the algorithm lets $\ell = 0$ and $u = n - 1$, we can see that $A[0 \;....\; n-1]$ is sorted and has size n. $\square$

Recurrence relation:

From lines 9-12, our algorithm performs comparisons which only take a constant amount of time. From lines 14-18, our algorithm performs recursion 3 times, where each time, the array is $\frac{2}{3}$ the size of the original array. We can express this as the recurrence:

$$\boxed{T(n) = 3T(\tfrac{2}{3}n) + \Theta(1)}$$

Upper bound:

We will use the master theorem to find a good upper bound for our algorithm.

Recall:

If

$$T(n) = aT(\tfrac{n}{b}) + \Theta(n^d) \text{ where } a > 0,\ b > 1, \text{ and } d \geq 0$$

then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

$$\Rightarrow \quad T(n) = 3T(\tfrac{2}{3}n) + \Theta(1)$$
$$= 3T\left(\frac{n}{\frac{3}{2}}\right) + \Theta(n^0) \text{ where } a = 3,\ b = \tfrac{3}{2}, \text{ and } d = 0$$

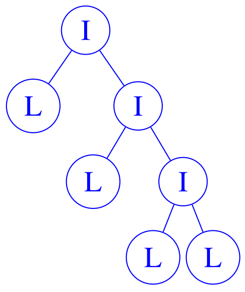
Since $0 < \log_{\frac{3}{2}} 3$,

$$\Rightarrow \quad T(n) = \Theta(n^{\log_{\frac{3}{2}} 3}) \approx \boxed{\Theta(n^{2.71})}$$

2. Induction Proof correctness (10 pts)

Claim 2 counter-example:

Suppose we had the following full binary tree. (I = internal node and L = leaf node)



This full binary tree contains 3 internal nodes and 4 leaf nodes. According to the claim, the height of the full binary tree is 3 - 1 = 2. But that's incorrect, since the height is 3. Hence, the claim is false.

Proof flaw for Claim 2:
The flaw in the proof for Claim 2 is that it only considers full binary trees that have two children leaves. This doesn't cover all cases for all full binary trees.

Proof flaw for Claim 1:
The proof for Claim 1 does have a flaw. Like the flaw in the proof for Claim 2, the proof for Claim 1 doesn't cover all cases for all full binary trees.

3. Asymptotic notation (6 pts)

Recall:

$$\lim_{x\to\infty}\frac{f(x)}{g(x)} = L \begin{cases} \text{if } 0 \le L < \infty \text{ then} & f(x) \in O(g(x)) \\ \text{if } 0 < L < \infty \text{ then} & f(x) \in \Theta(g(x)) \\ \text{if } 0 < L \le \infty \text{ then} & f(x) \in \Omega(g(x)) \end{cases}$$

a) Counter-example:

Suppose $f(n) = \frac{1}{n}$ and $g(n) = \frac{1}{n} + 1$,

$\Rightarrow$

$$\lim_{n\to\infty}\frac{\log_2(\frac{1}{n})}{\log_2(\frac{1}{n}+1)} = \lim_{n\to\infty}\frac{\frac{\ln(n^{-1})}{\ln 2}}{\frac{\ln(n^{-1}+1)}{\ln 2}}$$
$$= \lim_{n\to\infty}\frac{-\ln n}{\ln(1+\frac{1}{n})} = \frac{-\infty}{0} = -\infty$$

Hence,

$$\log_2(n^{-1}) \notin O(\log_2(n^{-1}+1))$$

b) Counter-example:

Suppose $f(n) = \log_2(\frac{1}{n}+1)$ and $g(n) = \log_2(\frac{1}{n})$,

$\Rightarrow$

$$\lim_{n\to\infty}\frac{2^{(\log_2(\frac{1}{n}+1))}}{2^{(\log_2(\frac{1}{n}))}} = \lim_{n\to\infty}\frac{\frac{1}{n}+1}{\frac{1}{n}} = \lim_{n\to\infty} n+1 = \infty$$

Hence,

$$2^{(\log_2(\frac{1}{n}+1))} \notin O(2^{(\log_2(\frac{1}{n}))})$$

c) Proof:

Suppose there exists $N \in \mathbb{N}$ and $c \in \mathbb{R} > 0$ such that $\forall n \in \mathbb{N}$ with $n \geq N$ then,

$$0 \leq f(n) \leq cg(n)$$

$$\Rightarrow \quad 0^2 \leq f(n)^2 \leq (cg(n))^2$$

$$0 \leq f(n)^2 \leq c^2 g(n)^2$$

Therefore, $f(n)^2 \in O(g(n)^2)$ □

4. Submarine Hiding (10 pts)

Given: An array A of size n where $A[0] = 0$, $A[n-1] = 0$ and all elements in $A[1 \; .... \; n-2]$ are positive integers, we need to find a number at some index m in A such that $A[m-1] \leq A[m]$ and $A[m+1] \leq A[m]$.

a) To accomplish $O(\log n)$, we will use a variation of the binary search algorithm to look at half of the array.

```
function FIND-VALLEY($A$, $n$)
    Find-Valley-Rec($A$, 0, $n-1$)
end function

function FIND-VALLEY-REC($A$, $\ell$, $u$)
    $mid \leftarrow \lfloor (u+\ell)/2 \rfloor$
    if $A[mid-1] \leq A[mid]$ and $A[mid+1] \leq A[mid]$ then
        return $mid$
    else if $mid > 0$ and $A[mid-1] > A[mid]$ then
        return Find-Valley-Rec($A, \ell, m-1$)
    else
        return Find-Valley-Rec($A, m+1, u$)
    end if
end function
```

Claim: Given an array A of size n, the algorithm always returns an index m such that $A[m-1] \leq A[m]$ and $A[m+1] \leq A[m]$.

Induction Proof:

Base case: Given that the problem states that the array must hold the depth of both ports and have some distance in between the ports, our base case occurs when we have an array A of size 3. In this case, $A[0] = 0$, $A[1] =$ some positive integer and $A[2] = 0$. Clearly the algorithm works since it will return index 1.

Inductive Step: We need to show the algorithm works for any array A of size > 3. Let's assume the algorithm works for any array A of size $\leq k-1$. Let us prove that the algorithm works for an array A of size k.

We have three cases:
(1) When $A[mid-1] \leq A[mid]$ and $A[mid+1] \leq A[mid]$, the algorithm clearly works.

(2) When $mid > 0$ and $A[mid-1] > A[mid]$, we recurse through half of the array. Namely the left half side of the array $A[\ell$ .... $m-1]$. The size of the array of this half is $n = mid - \ell - 1 = \lfloor(\ell+u)/2\rfloor - \ell - 1$. If $\ell + u$ is odd, then $n = (\ell+u-1)/2 - \ell - 1 = (u-\ell)/2 - 1$ which is smaller than $k = u - \ell$. Conversely, if $\ell + u$ is even, then $n = (\ell+u)/2 - \ell - 1 = (u-\ell)/2$, which is smaller than $k = u - \ell$. Hence, the recursive call must be between 0 and $k-1$, and is correct by the induction hypothesis.

(3) When $A[mid-1] \leq A[mid]$ or $mid \leq 0$, we recurse through the right half of the array $A[m+1$ .... $u]$. The size of this half is $n = u - (m+1) - 1 = u - \lfloor(\ell+u)/2\rfloor - 1$. If $\ell + u$ is even, then $n = (u-\ell)/2 - 1$, which is less than $k = u - \ell$. On the other hand, if $\ell + u$ is odd, then $n = u - (\ell+u-1)/2 - 1 = (u-\ell)/2 - 1/2$, which is also less than $k = u - \ell$. Hence, the recursive call uses a smaller range of values within array A and is thus, correct by our induction hypothesis.

Therefore, since each case is correct, we can conclude that the algorithm will work on an array A of size k. $\square$

Recurrence relation:

From lines 6-8, the algorithm does a constant amount of work. Then, from lines 9-13, the algorithm does recursion 1 time on an array that's $\frac{1}{2}$ smaller than the original array. This can be expressed as the following recurrence:

$$\boxed{T(n) = T(\tfrac{n}{2}) + \Theta(1)}$$

Upper Bound:

Finding a good upper bound for this recurrence is fairly straight forward if we use the master theorem.

$\Rightarrow$ $T(n) = T\left(\frac{n}{2}\right) + \Theta(n^0)$ where $a = 1$, $b = 2$, and $d = 0$

Since $0 = \log_2 1$,

$\Rightarrow$ $T(n) = \Theta(n^0 \log n) = \boxed{\Theta(\log n)}$

b) If the submarine started from the beginning (at the first port), then in the worst-case, there will be a sea valley at the last possible location. That is, the location just before getting to the second port ($A[n-2]$). Since we'd have to traverse through all possible locations between the ports, our upper bound would be linear in time. So the upper bound would be $\boxed{O(n)}$.