CMPS 102 — Winter 2019 – Homework 1 Solution Sketches

1. (10 pts) A grad student comes up with the following algorithm to sort an array $A[1..n]$ that works by first sorting the first 2/3rds of the array, then sorting the last 2/3rds of the (resulting) array, and finally sorting the first 2/3rds of the new array.

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

- First (5 pts), prove that the algorithm correctly sorts the numbers in the array (in increasing order). After showing that it correctly sorts 1 and 2 element intervals, you may make the (incorrect) assumption that the number of elements being passed to *G-sort-recurse* is always a multiple of 3 to simplify the notation (and drop the floors/ceilings).

  Solution:

  Proof. For all $n \geq 1$, let IH(n) be the statement: "Given an array A, u, and ℓ where A has at least u elements, $0 < \ell < u$, and $u - \ell + 1 = n$, *G-sort-recurse*(A, u, ℓ) correctly sorts the slice $A[u, ...\ell]$ (in increasing order) and preserves the other array elements.
  Base Case: IH(1) When $n = 1$, $u = \ell$ and the slice has one element. Since the slice has only one element, it is already sorted. *G-sort-recurse*(A, ℓ, u) will execute lines 5-6 and thus terminate the algorithm without altering A. Hence, IH(1) is true.
  Base Case: IH(2) Consider $n = 2$ so $u = \ell + 1$ and the array slice has 2 elements. *G-sort-recurse*(A, ℓ, u) will execute lines 5, 7, and 8. On line 8, it will compare the only two elements in the slice and, if the elements are not in increasing order already, it will execute lines 9-11 and thus swap the elements (in-place) so that they are in increasing order. Thus, the slice is sorted and the rest of A is unchanged and IH(2) is true.
  Inductive Step: Assume $n > 2$ and IH(k) holds for $1 \leq k < n$ to show that IH(n) also holds. Primarily for notational simplicity, we have the additional assumption that n is a multiple of 3 (although this assumption would not be made is a strictly formal proof).

Consider an arbitrary call *G-sort-recurse(*A, ℓ, u*)* with $u - \ell + 1 = n$ and array A with at least u elements. Assume n is a multiple of 3 such that $n = 3m$. Thus $u = \ell + 3m - 1$. Since $n \geq 3$ we have $m \geq 1$. Also, let:

- A_1 be the m-element slice (i.e. $A[\ell..\ell + m - 1]$),
- A_2 be the next m elements in A (i.e. $A[\ell + m..\ell + 2m - 1]$), and
- A_3 be the last m elements in the slice (i.e. $A[\ell + 2m..u]$).

Since $n > 2$, *G-sort-recurse(*A, ℓ, u*)* will execute lines 5, 7, and 13 thru 19. On lines 13 thru 19, it will recursively sort A (in-place) as follows:

(a) First, it will sort the first two-thirds of the array by calling: *G-sort-recurse(*$A, \ell, \ell+2m-1$*)*.
   - Note, there are $k = 2m$ elements in the first two-thirds of the array and $k < n$ when $n = 3m$ and $m \geq 1$.
   - By the inductive hypothesis, IH(k) is true for $k = 2m$ when $n = 3m$ and $m \geq 1$.
   - Thus, after (1a) the first $2m$ elements in the slice (i.e. positions A_1 thru A_2) will be sorted in increasing order such that all m elements in A_2 are larger than (or equal to) all of the m elements in A_1.

(b) Next, it will sort the last two-thirds of the slice (i.e. array sections A_2 and A_3) by calling: *G-sort-recurse(*$A, \ell + m, u$*)*.
   - Note, there are also $k = 3m - (m + 1) + 1 = 2m$ elements in the last two-thirds of the slice and $k < n$ when $n = 3m$ and $m \geq 1$.
   - By the inductive hypothesis, IH(k) is true for $k = 2m$ when $n = 3m$ and $m \geq 1$. Thus, after (1b) the last $2m$ elements in the resulting array A (i.e. A_2 thru A_3) will be sorted in increasing order such that all the m elements in A_3 are larger than (or equal to) all the m elements in A_2.
   - In addition, after (1b) all the m elements in A_3 will also be larger than (or equal to) all the m elements in A_1. To prove this we will assume the contrary:

     Proof. **By contradiction:** Assume to the contrary that after (1b) there exists an element e in A_3 such that e is **not** larger than (or equal to) the elements in A_1.

     * Since all m elements in A_3 should be greater than (or equal to) the m elements in A_2 after (1b), there will be **at least** $m + 1$ elements (all the m elements in A_2 plus e in A_3) in the last two-thirds of the resulting slice that will be smaller than the elements in A_1.
     * As noted before, after (1a), all m elements in A_2 will be larger than (or equal to) all m elements in A_1. This means that, even after (1b), there should be **at least** m elements in the last two-thirds of the resulting slice which are larger than (or equal to) all m elements in A_1.
     * Hence, after (1b), there can only be **at most** m elements in the last two-thirds of the slice which are smaller than the elements in A_1.
     * Thus proving that, after (1b), there cannot be an element e in A_3 such that e is not larger than (or equal to) the elements in A_1. In other words, after (1b), all the elements in A_3 will be larger than (or equal to) all the elements in A_1.

     □

(c) Finally, the algorithm will sort the first two-thirds of the slice by calling: *G-sort-recurse(*$A, \ell, \ell + 2m - 1$*)*.
   - As in 1a, note that there are $k = 2m$ elements in the first two-thirds of the array and that $k < n$ when $n = 3m$ and $m \geq 1$.
   - By the inductive hypothesis, IH(k) is true for $k = 2m$ when $n = 3m$ and $m \geq 1$.
   - This implies that after (1c) the first $2m$ elements in the resulting array A (i.e. A_1 thru A_2) will be sorted in increasing order such that all the m elements in A_2 are larger than (or equal to) all the m elements in A_1.
   - Recall that after (1b), the elements in A_3 are:
     * sorted in increasing order too, and
     * larger than (or equal to) the elements in A_1 and A_2.
   - Hence, after (1c) all the elements in A are sorted in increasing order.

In conclusion, if IH(k) holds for $k < n$ when $n = 3m$ and $m \geq 1$, then IH(n) holds for $n > 2$. Thus proving that *G-sort(*n*)* sorts the numbers in an array of n elements in increasing order. □

- Next (1 pts), Derive a recurrence for the algorithm's running time (or number of comparisons made).

  Solution:
  For some constant c,
  $$T(n) \leq \begin{cases} c, & \text{if } n = 1, 2. \\ 3T(\frac{2n}{3}), & \text{if } n > 2. \end{cases} \quad (1)$$

- Finally (4 pts), obtain a good asymptotic upper bound (big-O) for your recurrence.

  Solution:
  Note the recurrence fits the Master Theorem with $a = 3$, $b = 3/2$ and $f(n) = 0$. Also, $f(n)$ is $O(n^c)$, where $c < \log_{3/2} 3 \approx 2.7$. Then it follows from the first case of the Master Theorem in CLR that
  $$T(n) = O(n^{\log_{3/2} 3})$$
  This can also be shows with recursion trees: The number of subproblems up by a factor of 3 as you go down a level, while the problem size goes down by a factor of $2/3$. After $\log_{3/2}(n)$ levels, the problem sizes will reach 1. Meanwhile the number of problems on the bottom level will be $3^{\log_{3/2} n} = n^{\log_{3/2}(3)}$. Noting that the split-combine work is just a constant, the work on the various levels is a geometric series that sums to a constant times the largest term, or $O(n^{\log_{3/2}(3)})$

2. Induction Proof correctness (10 pts).

Recall that a full binary tree contains (A) just a single leaf node, or (B) an internal node (the root) connected to two disjoint subtrees, which are themselves full binary trees.

First consider the following claim and proof. Think about if the theorem is true or not, and if the proof is correct or not. (These preliminary thoughts need not be included in your answer.)

Claim 1. *In any full binary tree, the number of leaf nodes is one greater than the number of internal nodes.*

Proof. (??) By induction on number of internal nodes.

For all $n \geq 0$, let IH(n) be the statement: "all full binary trees having exactly n internal nodes have $n + 1$ leaf nodes."

Base Case: Show IH(0). Every full binary tree with zero internal nodes is formed by case (A) of the definition, and thus consists of just a single leaf node. Therefore, every full binary tree with 0 internal nodes has exactly 1 leaf node, and IH(0) is true.

Inductive Step: Assume $k > 0$ and IH(k) holds to show that IH$(k + 1)$ also holds.

Consider an arbitrary full binary tree T with k internal nodes. By the inductive hypothesis T has $k + 1$ external nodes. Create a $k + 1$ internal node tree T' by removing a bottom leaf node in T and replacing it with an internal node connected to two children that are leaves. T' has one more internal node than T, and $2 - 1 = 1$ more external node than T. Therefore T' has $k + 1$ internal nodes and $(k + 1) + 1 = k + 2$ external nodes, proving $P(k + 1)$. □

Now consider the following claim.

(Recall that the height of a binary tree is the length of the longest root-to-leaf path).

Claim 2. *For all n, all full binary trees with n internal nodes have height $n - 1$.*

We have the following "proof" of the claim.

Proof. ?? By induction on n. For each $n \geq 1$, let IH(n) be the statement "all full binary trees with n internal nodes have height $n - 1$".

Base Case: Show IH(0). Every full binary tree with zero internal nodes is formed by case (A) of the definition, and thus consists of just a single leaf node. Therefore, every full binary tree with 0 internal nodes has only one leaf (which is also the root). Thus the longest root-to-leaf path has length 0, and IH(0) is true.

Inductive Step: Let $k \geq 1$ and show that IH(k) implies IH$(k + 1)$. Let T be an arbitrary full binary tree with k internal nodes. Create the binary tree T' having $k+1$ internal nodes by removing a bottom leaf node in T and replacing it with an internal node connected to two children that are leaves. The longest root-to-leaf path in T' is thus one greater than the longest root-to-leaf path in T, so the height of T' is the height of T plus 1. Furthermore, by the inductive hypothesis IH(k), the height of T is $k - 1$. Therefore T' has $k + 1$ internal nodes and height $(k - 1) + 1 = k = (k + 1) - 1$, showing IH$(k + 1)$. □

- Give a counter-example showing that the claim is false (1 pt).

  Counter-example:

  With the typo in the problem we can take $n = 1$. By case **(B)** of the definition, a full binary tree with **one** internal node will be connected to two disjoint subtrees, which are themselves full binary trees. In addition, by Claim 1, a full binary tree with $n = 1$ internal nodes has $n + 1 = 1 + 1 = 2$ leaf nodes. Hence, the length of the path between the root and the leaf nodes is 1. This implies that the height of the tree is 1 and not equal to $n - 1 = 1 - 1 = 0$ as suggested by Claim 2. Therefore, there exists an n ($n = 1$) such that a full binary tree with n internal nodes does not have height $n - 1$. Thereby proving Claim 2 is false.

- Identify and clearly describe the flaw in this proof (4 pts).

  Solution:

  The hypothesis is that for $n \geq 1$ all full binary trees with n internal nodes have height $n - 1$. As shown above by counter-example, IH(1) does not hold.

This would lead students to the incorrect conclusion that the first proof is correct.

A slightly different version of the second claim:

Theorem?: For all n, all n-node binary trees have height $n - 1$.

Proof??: By induction on n. For each $n \geq 1$, let IH(n) be the statement "all n node binary trees have height $n - 1$".

Base Case: $n = 1$. Every 1-node binary tree consists of just a root, and has height 0. Therefore IH(1) is true.

Inductive Step: Let $n \geq 1$ and show that IH(n) implies IH$(n+1)$. Let T be an arbitrary n-node binary tree. Create the $n+1$-node binary tree T' by adding a new leaf to T hanging off the bottom level. The height of T' is thus one greater than the height of T. Furthermore, by the inductive hypothesis, the height of T is $n-1$. Therefore T' has $n+1$ nodes and height $n-1+1 = n$, showing IH$(n+1)$. $\square$

The claim is clearly false, as a 3-node binary tree (with root and two leaves) has height 1.

The flaw in the above proof is not easy to find. The problem is that IH$(n+1)$ says that *all* $n+1$-node binary trees have height n. The inductive step constructs only one $n+1$-node binary tree. The correct approach for the inductive step is to start with an arbitrary $n+1$-node binary tree, find a smaller tree inside of it, and apply the inductive hypothesis on the smaller tree.

This flaw is also in the first proof – it relies on the unproven (and false) statement that growing trees in the way described (extended from the lowest level) produces all possible trees in the class.

3. Asymptotic notation (6 pts): Exercise 5 of Chaper 2 (prove or disprove 3 asymptotic implications).

   Assume you have functions f and g such that f(n) is O(g(n)). For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

   (a) $\log_2 f(n)$ is $O(\log_2 g(n))$. (2 points)

   There is a subtle issue that needs to be pointed out here. This is true if $g(n) \geq 2$, $\forall\ n \geq n_0$, since $f(n) \leq cg(n)$, which means $\log_2 f(n) \leq \log_2 g(n) + \log_2 c \leq (1 + \log_2 c)(\log_2 g(n))$.

However, in the general case, since we can say that $g(n) = 1$ and $f(n) = 2, \forall n$, then $\log_2 g(n) = 0$ and the inequality $\log_2 f(n) \leq c \log_2 g(n)$ is false for every $c > 0$. A similar problem arises if $g()$ is decreasing towards 1, say $g(n) = 1 + \frac{1}{n}$.

(b) $2^{f(n)}$ is $O(2^{g(n)})$. (2 points)

This is false. Let $f(n) = 2n$ and $g(n) = n$. Then $2^{f(n)} = 4^n$, but $2^{g(n)} = 2^n$.

(c) $f(n)^2$ is $O(g(n)^2)$. (2 points)

This is true. Since $f(n) \leq cg(n), \forall\, n \geq n_0$, this means that $(f(n))^2 \leq c^2(g(n))^2, \forall\, n \geq n_0$.

4. Submarine Hiding (10 pts).

   A submarine is ordered to patrol between two ports. Assume the locations along the patrol route are identified by consecutive integers between 0 (for the first port) and n (for the other port). One way for the submarine to hide is to settle on the bottom in a "sea valley" where the depth at that location is greater than the depths of both adjacent locations. The submarine may determine the depth at its current location by performing a "depth survey" that is somewhat expensive. The returned depth for the location will be a non-negative number, with larger numbers indicating deeper depths. Since the ports have adjacent locations on only one side, they do not qualify as "sea valleys". Furthermore, assume:

   - the depth at both ports is 0
   - the depth at the other locations is positive
   - adjacent locations always have different depths

   First (9 pts): Create a divide and conquer algorithm for finding a "sea valley" (a location whose depth is greater than the adjacent locations) that requires only $O(\log n)$ depth surveys in the worst case (any sea valley will do, it need not be the first or deepest). State your algorithm, argue that it is correct, derive the recurrence for the number of depth surveys needed, and then bound the recurrence. Measure the cost of your algorithm in "depth surveys" and ignore the time to sail between locations.

Solution:
One assumption that is implied in the problem is that there is always at least one location on the patrol route in the ocean.

The depths of the sea floor and ports can be represented as an array, A, of size $n + 1$. This array has the property that the first two elements are increasing in value, and the last two are decreasing (as the ports are valued at 0, and every other depth is positive). Furthermore, besides the ports at the end, no two adjacent elements have the same value.

First, we need to show that the given problem always has a solution. As there is an increase in depth leaving the first port, and a decrease in depth entering the second, and all consecutive elements are either increasing or decreasing, there must be at least one place where the increasing depth changes to decreasing depth, and this place corresponds to a sea valley.

The algorithm is as follows. Let $left = 0, right = n$, and the array that our algorithm operates on be $A[left, right]$. For any instance of the problem with $n = 2$, the middle element must be a sea valley. For $n > 2$, we will first check the middle index of A, $A[mid] = A[\lfloor \frac{n+1}{2} \rfloor]$. If it is larger than the two elements to its left and right, then it is a sea valley.

If $A[mid]$ is smaller than the left element, then from the first element, $A[left]$, to the middle element we have a subproblem with identical structure to the original problem, the depth starts increasing and finishes by decreasing. As we have shown above, that guarantees that the subarray will have a local maximum. As a result, we can disregard all of the elements to the right of the middle element, and recursively call our algorithm on the left subarray, $A[left, mid]$.

Otherwise, if the middle element is smaller than the element to the right, then we have a subarray with identical structure to the original problem between the middle element and the last element, $A[right]$. Thus, we can disregard all elements to the left of the middle element, and recursively call our algorithm on the right subarray, $A[mid, right]$.

This algorithm will never reduce A to less than three elements, as in the only situation where that is possible, $|A[left, right]| = 4$, the middle element, here $A[mid]$, must be greater than $A[left]$. If $A[mid + 1] < A[mid]$, $A[mid]$ is a sea valley. Else, the algorithm will recurse on $A[mid, right]$, which has size 3, and thus has a sea valley at the middle element. For $|A[left, right]| > 4$, the algorithm will not reduce the array to a subarray of size less than 3, as all subarrays include the middle element.

This algorithm terminates when the middle element is a sea valley. As the algorithm recursively divides the array in half until the array size is 3, the maximum number of divisions is in $O(\log_2 n)$. Each time the array is divided, three depth surveys are done, which means the total number of surveys possible is in $3 * O(\log_2 n) = O(\log n)$.

For $n = 2, T(n) = 1$.

For $n > 2$, $T(n) = T(n/2) + 3$ as the problem is recursively split in half, only one subproblem is examined, and 3 surveys are done each split. By the Master Theorem, $T(n) = O(logn)$.

Second (1 pt): Assume that the submarine starts out at location 0, and runs your algorithm by sailing to each requested depth survey location in turn, and the locations are evenly spaced at one unit apart. Get a good asymptotic bound on how far the submarine might have to sail (as a function of n) when using your algorithm.

Solution:
A quick way to establish a lower bound on the number of spaces travelled in the worst case is the

situation where the only maxima is the location right before the second port. Here, the submarine would have to travel to every location between the ports to make a depth survey of the sea valley.

Examining the worst case scenario in general, the algorithm runs until the subproblem is of size 3. Each step of the algorithm , the submarine must travel to the new middle of the subarray, and examine the middle and the locations to each side. For ease of calculation, assume this takes an extra two spaces moved (one adjacent location is passed on the way to the middle - then it is one space out to the other adjacent location, and one space back).

Each time the array is split, the required travel distance to the middle is also split in half. Thus, the total distance covered travelling to the middles is $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... + 1$. This can be approximated (upper-bounded) by an infinite geometric series, which can be simplified to n. The number of splits is in $O(\log n)$, so the number of extra spaces moved is $2 * O(\log n)$ Thus, the number of spaces the submarine must travel in the worst case is in $O(n) + O(\log n) = O(n)$.