Isai Lopez Rodas
1605542, ilopezro@ucsc.edu

CMPS 102 — Spring 2020 – Homework 1

Updated Ver. 5(04-08)
Three problems, 17 points, due Friday April 10 (11:59 PM) on Gradescope.

Before you begin the assignment, please read the following carefully.

- Read the *Homework Guidelines*.
- Every part of each question begins on a new page. Do not change this.
- This does not mean that you should write a full page for every question. Your answers should be short and precise. Lengthy and wordy answers will lose points.
- Do not change the format of this document. Simply type your answers as directed.
- You are **not** allowed to work in teams.

I have read and agree to the collaboration policy. – Isai Lopez, ilopezro@ucsc.edu
Collaborators:

---

1. (Total: 8 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
```

a (4 pts). First, 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.

I will provide a proof using proof by cases.

1. *Array length = 1: the array is already in order.*
2. *Array length = 2:* G-sort-recurse() *will check to see if a[0] $>$ a[1] and switch them if it is true. The resulting array will also be in order at the end of that function call.*
3. *Array length = 3:* G-sort-recurse() *will first run on the first two elements of the array. According to case 2, if the first element is greater than the second, they will be flipped. This initial function call will return an ordered subarray. The second function call will be on the last two elements in the array. Again, using case 2, we know that the returned array will be in order. The last function call will ensure that the first two elements are in order.*
   a) *Initial array = [100, 50, 1]*
   b) G-sort-recurse([100,50]) *will return [50,100] according to case 2*
   c) *Resultant array = [50, 100, 1]*
   d) G-sort-recurse([100,1]) *will return [1, 100] according to case 2*
   e) *Resultant array = [50, 1, 100]*
   f) G-sort-recurse([50,1]) *will return [1, 50] according to case 2*
   g) *Final array = [1, 50, 100] is in order.*
4. *Array length $>$ 3: will rely on cases 2 & 3 to succesfully place each element in the array in ascending order.*

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

Solution.

The first function being called, G-sort()*, is called only once and it calls* G-sort-recurse() *on the entire array. When you're in* G-sort-recurse()*, the first comparison is made on line 5, then on line 7, and if neither of these conditions are met, you enter the else statement in line 13. Once in this else statement, you split up the array into two thirds until you're left with either 1 or 2 elements in the array. Line 16 therefore takes* $T(\frac{2n}{3})$*. Line 18 does the same, so therefore you've got another* $T(\frac{2n}{3})$*. Line 17 does the same with the last two-thirds of the array so that also has time* $T(\frac{2n}{3})$*. All other comparisons take constant time,* $\mathcal{O}(1)$*). The final recurrence for the algorithm's running time is:* $T(n) = 3T(\frac{2n}{3}) + \mathcal{O}(1)$.

c (3 pts). Finally, obtain a good asymptotic upper bound (big-O) for your recurrence.

Solution. [1]

In its worst case, the algorithm will run on an array that is in descending order.

[1] I referenced this link from StackOverflow to formulate my answer.

"I have read and agree to the collaboration policy." – FirstName LastName, email@ucsc.edu
Collaborators:

---

2. (Total: 6 pts) Asymptotic notation: Exercise 5 of Chaper 2: Prove or disprove 3 asymptotic implications: If $f(n)$ is in $O(g(n))$ then is it always true that:
a (2pts). $\log_2(f(n))$ is in $O((\log_2(g(n))$.

Solution.

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

Solution.

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

Solution.

"I have read and agree to the collaboration policy." – FirstName LastName, email@ucsc.edu
Collaborators:

---

3. (3 pts) Least counterexample: Give a proof by contradiction using the *least counterexample* method that:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \qquad \text{for all } n \geq 0.$$

Solution.

Recommended exercises (not to be turned in)

1. The solved exercises in Chapters 1, 2, and 5 (Divide and Conquer).
2. Exercises 1 and 2 in Chapter 1 of the text.
3. Exercises 3 and 4 in Chapter 2 of the text.