Computer Science C73 September 29, 2021
Scarborough Campus University of Toronto

Solutions for Homework Assignment #3

Answer to Question 1.

a. The basic reasoning is as follows: Let A_1, A_2, and A_3 be the first third, middle third, and last third of A. After the first recursive call (which sorts A_1 and A_2) A_2 contains elements that are greater than or equal to the elements in A_1. Thus, after the second recursive call (which sorts A_2 and A_3), A_3 contains elements that are greater than or equal to the elements in A_2 and therefore also in A_1 — i.e., it contains the largest one-third of the elements of A in sorted order. The third recursive call sorts the remaining elements in A_1 and A_2, all of which are smaller than the elements in A_3.

We can make this argument more rigorous by using complete induction on the length n of $A[\ell..r]$, i.e. $n = r - \ell + 1$, to prove that the algorithm correctly sorts subarrays of length n, for every $n \geq 1$. (As suggested, you may assume for simplicity that n is a power of 3, though we will not do so in this proof.)

Suppose the algorithm sorts $A[\ell..r]$ when the length of $A[\ell..r]$ is strictly less than n. We will prove that it also sorts $A[\ell..r]$ when the length of $A[\ell..r]$ is equal to n.

If $n = 1$ or $n = 2$ this is immediate from lines 1–2 or lines 3–4, respectively.

If $n \geq 3$, let $t = \lceil 2(r - \ell + 1)/3 \rceil$, $A_1 = A[\ell..r - t]$ (the first third of A), $A_2 = A[r - t + 1..\ell + t - 1]$ (the middle third of A), and $A_3 = A[\ell + t..r]$ (the final third of A). Also, denote by A^1, A^2, and A^3 the content of A after the first, second, and third recursive call respectively (line 8, 9, and 10). So A_i^j denotes the content of the i-th third of A after the j-th recursive call.

It is easy to see that the lengths of the subarrays $A[\ell..\ell + t - 1]$ and $A[r - t + 1..r]$ are strictly less than n, and so by the induction hypothesis, the three recursive calls correctly sort the relevant subarrays.

By the properties of sorting, it is obvious that:

$$\text{for all } x \text{ in } A_2^1 \text{ and all } y \text{ in } A_1^1,\ x \geq y \tag{1}$$

$$\text{for all } x \text{ in } A_3^2 \text{ and all } y \text{ in } A_2^2,\ x \geq y \tag{2}$$

Now we show that

$$\text{for all } x \text{ in } A_3^2 \text{ and all } y \text{ in } A_1^2,\ x \geq y \tag{3}$$

To see this, let x be in A_3^2 and y be in A_1^2. Since the second recursive call does not affect the first third of A, y is also in A_1^1. There are two cases:

Case 1. No z in A_2^1 is in A_2^2. (In other words, the sorting done by the second recursive call caused every element in the middle third of A to move to the last third.) Therefore, x is one of the values that were in A_2^1. Thus, by (1), $x \geq y$.

Case 2. Some z in A_2^1 is in A_2^2. (In other words, the sorting done by the second recursive call caused some element in the middle third of A to remain there.) By (2), $x \geq z$ (since z is in A_2^2). By (1), $z \geq y$ (since z is also in A_2^1). Therefore $x \geq y$.

By (2) and (3), after the second recursive call, A_3 contains the largest third of the elements of A, in sorted order. Thus, the third recursive call, which does not affect A_3, sorts the remaining two-thirds of the elements of A in A_1 and A_2. Thus, after the third recursive call all of A is sorted.

b. The recurrence equation that describes the running time of WEIRDSORT(A, ℓ, r) when $r - \ell + 1 = n$ is $T(n) = 3T(2n/3) + c$, since there are three recursive calls, each on input $2/3$ the size of the original, and a constant amount additional work. In terms of the parameters of the Master Theorem, we have $a = 3$, $b = 3/2$, and $d = 0$. Since $a > b^d$, the third case of the Master Theorem applies, and we have $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.71})$ ($\log_b a = \log_{1.5} 3 = \ln 3 / \ln 1.5 \approx 2.71$).

c. The running time of Bubblesort is $\Theta(n^2)$. So, Weirdsort is much worse than even the simple quadratic sorting algorithms such as Bubblesort — let alone the $\Theta(n \log n)$ ones such as Mergesort or Heapsort. Bad idea!

Answer to Question 2. Suppose $A[1..n]$ is an array of distinct integers. Our algorithm for finding a local minimum of $A[1..n]$ is based on the following:

Lemma 1 *For any k such that $1 \leq k < n$,*
(a) if none of the integers in $A[1..k]$ is a local minimum of the array $A[1..n]$, then $A[1..k+1]$ is sorted in decreasing order;
(b) if none of the integers in $A[k+1..n]$ is a local minimum of the array $A[1..n]$, then $A[k..n]$ is sorted in increasing order.

PROOF. We use induction on k to prove (a); the proof of (b) is similar.

For the basis, when $k = 1$, we note that if $A[1] < A[2]$ then $A[1]$ is a local minimum. (The case $k = 1$ is simply the contrapositive of this statement.)

For the induction step, assume that the claim holds for $k \geq 1$, and suppose that $k + 1 < n$ and that none of the integers in $A[1..k+1]$ is a local minimum of $A[1..n]$. We must show that $A[1..k+2]$ is sorted in decreasing order. By the induction hypothesis, since none of the integers in $A[1..k]$ is a local minimum, $A[1..k+1]$ is sorted in decreasing order. It remains to show that $A[k+1] > A[k+2]$. Suppose, for contradiction, that $A[k+1] < A[k+2]$. Since $A[1..k+1]$ is sorted in decreasing order, $A[k] > A[k+1]$. But then $A[k+1]$ is a local minimum of $A[1..n]$, contrary to the assumption that none of the integers in $A[1..k+1]$ are. $\square$

If $A[k] < A[k+1]$ then $A[1..k+1]$ is certainly not sorted in decreasing order; similarly, if $A[k] > A[k+1]$ then $A[k..n]$ is not sorted in increasing order. Thus, the (contrapositive of the) above lemma implies:

Corollary 2 *For any k such that $1 \leq k < n$,*
(a) if $A[k] < A[k+1]$ then some integer in $A[1..k]$ is a local minimum of the array $A[1..n]$;
(b) if $A[k] > A[k+1]$ then some integer in $A[k+1..n]$ is a local minimum of the array $A[1..n]$.

This leads us to the following divide-and-conquer algorithm, for finding a local minimum in $A[\ell, r]$, where $1 \leq \ell \leq r \leq n$.

```
FindLocalMin(A, ℓ, r)
if ℓ = r then return A[ℓ]
m := ⌊(ℓ + r)/2⌋
if A[m] < A[m+1] then return FindLocalMin(A, ℓ, m)
else return FindLocalMin(A, m+1, r)
```

The fact that this algorithm is correct, i.e., it returns a local minimum of $A[\ell, r]$ follows by a straightforward (complete) induction, using the above Corollary. To find a local minimum of the entire array $A[1..n]$, we simply call FindLocalMin$(A, 1, n)$.

The running time $T(k)$ of FindLocalMin(A, ℓ, r), where $k = \ell - r + 1$, is given by the following recurrence (for simplicity we assume that k is a power of 2):

$$T(k) = \begin{cases} T(k/2) + 1, & \text{if } k > 1 \\ 1, & \text{if } k = 1 \end{cases}$$

In terms of the parameters of the Master Theorem, here we have $a = 1$, $b = 2$, and $d = 0$, so $a = b^d$, and the applicable case of the theorem yields $T(k) = \Theta(k^d \log k) = \Theta(\log k)$. Therefore, the running time for the call FindLocalMin$(A, 1, n)$ is $\Theta(\log n)$, as wanted.