CMSC 441: Unit 2 Homework Solutions

October 4, 2017

(6.1-6) To check whether the array

$$\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$$

is a max-heap, convert it to tree form and check that each parent node is greater than or equal to each of its children. Alternatively, use the formula for the parent node, $\textsc{Parent}(i) = \lfloor i/2 \rfloor$, to check that the parent is always greater than or equal to the child. Either way, you will find that it is *not* a max-heap because $A[9] = 7$ and its parent is $A[4] = 6$.

(6.2-3) There is *no* effect calling $\textsc{Max-Heapify}(A, i)$ when $A[i]$ is larger than its children. Lines 1 – 7 simply determine which vertex is larger, $A[i]$ or one of its children. If $A[i]$ is the largest, then lines 9 – 10 do not execute.

(6.3-2) Max-Heapify assumes the left- and right-subtrees are already max-heaps. This would not be the case if the loop increased from 1 to $\lfloor A.length/2 \rfloor$.

(6.4-2) At the start of the loop, $i = A.length$, so $A[1\,..\,i]$ is the entire heap. Since Build-Max-Heap was called immediately before the loop, the invariant holds trivially.

To show that the invariant is maintained, assume it holds a the beginning of an iteration. Since $A[1\,..\,i]$ contains the i smallest elements and is a max-heap, $A[1]$ is the largest of the i smallest elements. Thus $A[1]$ is smaller than any element of $A[i+1\,..\,n]$ but larger than any other element of $A[1\,..\,i]$, so swapping $A[1]$ with $A[i]$ ensures that $A[i\,..\,n]$ contains the $n - i + 1$ largest elements in sorted order. Also, the elements of $A[1\,..\,i-1]$ are the $i - 1$ smallest elements, and the call to Max-Heapify on line 5 ensures that $A[1\,..\,i-1]$ is a max-heap.

Finally, the loop terminates when $i = 2$, and at the end of the iteration, $A[1]$ contains the smallest element, and $A[2\,..\,n]$ contains the remaining $n - 1$ elements in order.

(6.4-3) Suppose A is already sorted in increasing order. You may be tempted to think that this will reduce the cost of Max-Heapify, but once the array has been converted to a max-heap by the call to Build-Max-Heap, there is really nothing very special about it. The calls to Max-Heapify will still need

to exchange values of A and recurse, so there is no savings. The algorithm is still $O(n \lg n)$.

Similarly, if A is sorted in decreasing order, it is already a max-heap, so the call to BUILD-MAX-HEAP is not necessary, but that is not an asymptotic savings. Once exchanges occur (line 3 of HEAPSORT), the calls to MAX-HEAPIFY are still likely to recurse. The algorithm is still $O(n \lg n)$.

(7.2-2) If all the elements of the array are equal, the the condition on line 6 of PARTITION will always be true. Therefore, i will be incremented at every iteration and the loop will terminate with $i = r - 1$, and the function will return the value r for the partition point. Thus the partitions will be maximally imbalanced and the running time will be $\Theta(n^2)$.

(7.2-4) Because the checks will be nearly in order by check number, the PARTITION procedure will tend produce the worst case, completely imbalanced partition (since $A[r]$ will be largest element in A). Even when it does not produce the worst case, it will be nearly the worst case since there will only be a small number of check numbers in A for wich $A[i] \geq A[r]$. Therefore, the run-time will be close to the worst case, $O(n^2)$. On the other hand, INSERTION-SORT will be close to $O(n)$ time since the loop on line 5 will typically have zero, or a small number, of iterations.

(8.2-3) No, the algorithm is no longer stable. Consider Figure 8.2 in the textbook. If the order of the loop were reversed, the first iteration would place the *first* 2 from A into position 4 in B; later, the *second* 2 would be placed in position 3 in B. Therefore, the repeated values would not be in the same order in B as they were in A.

(8.3-3) We will use induction on the number of digits, d, in the numbers to be sorted. If $d = 1$, then radix sort is just counting sort with $k = 9$, so it correctly sorts the numbers. Now, suppose radix sort works for $d - 1$ digits; we need to show that it works for d digits. The first $d - 1$ iterations will correctly sort the integers on their $d-1$ low digits, by assumption. On the d^{th} iteration, counting sort will correctly sort on the high digit, grouping the integers with leading '0' together, followed by the integers with leading '1', etc., and since counting sort is stable, the integers will remain ordered on their low $d - 1$ digits within each group. Therefore the integers will be sorted correctly.

(8-3) (a) The usual radix sort algorithm will not solve this problem in the required time bound. Assume without loss of generality that all the integers are positive and have no leading zeros. (If there are negative integers or 0, deal with the positive numbers, negative numbers, and 0 separately.) Under this assumption, we can observe that integers with more digits are always greater than integers with fewer digits. Thus, we can first sort the integers by number

of digits (using counting sort), and then use radix sort to sort each group of integers with the same length. Noting that each integer has between 1 and n digits, let m_i be the number of integers with i digits, for $i = 1, 2, \ldots, n$. Since there are n digits altogether, we have $\sum_{i=1}^{n} i \cdot m_i = n$.

It takes $O(n)$ time to compute how many digits all the integers have and, once the numbers of digits have been computed, it takes $O(m+n) = O(n)$ time to group the integers by number of digits. To sort the group with m_i digits by radix sort takes $\Theta(i \cdot m_i)$ time. The time to sort all groups, therefore, is

$$\sum_{i=1}^{n} \Theta(i \cdot m_i) = \Theta\left(\sum_{i=1}^{n} i \cdot m_i\right) = \Theta(n).$$

(b) To solve the problem in $O(n)$ time, we use the property that, if the first letter of string x is lexicographically less that the first letter of string y, then x is lexicographically less than y, regardless of the lengths of the two strings. We take advantage of this property by sorting the strings on the first letter, using counting sort. We take an empty string as a special case and put it first. We gather together all strings with the same first letter as a group. Then we recurse, within each group, based on each string with the first letter removed.

The correctness of this algorithm is straightforward. Analyzing the running time is a bit trickier. Let us count the number of times that each string is sorted by a call of counting sort. Suppose that the ith string, s_i, has length l_i. Then s_i is sorted by at most $l_i + 1$ counting sorts. (The "+1" is because it may have to be sorted as an empty string at some point; for example, ab and a end up in the same group in the first pass and are then ordered based on b and the empty string in the second pass. The string a is sorted its length, 1, time plus one more time.) A call of counting sort on t strings takes $\Theta(t)$ time (remembering that the number of different characters on which we are sorting is a constant.) Thus, the total time for all calls of counting sort is

$$O\left(\sum_{i=1}^{m}(l_i + 1)\right) = O\left(\sum_{i=1}^{m} l_i + m\right) = O(n + m) = O(n)$$

where the second equality follows from $\sum_{i=1}^{m} l_i = n$, and the last equality is because $m \leq n$.