CS624: Analysis of Algorithms

Assignment 2 – Solution

1. Argue the correctness of HEAPSORT using the following loop invariant:

   At the start of each iteration of the for loop of lines 2–5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$ and the subarray $A[i+1..n]$ contains the n-i largest elements of $A[1..n]$ sorted.

   Solution:

   - **Initialization:** In the beginning i=n, so the subarray $A[i+1..n]$ is empty, therefore the property is trivially true.
   - **Maintenance:** Let us assume it is true in the beginning of the iteration for $1 \leq i \leq n$. The elements A[1..i] form a heap by construction - we start with a heap of size n, and by the beginning of every iteration we reduce the heap size by 1 so in the beginning of iteration i, the heap size is i. and so $A[1]$ is the largest element in the range $A[1..i]$. By inductive hypothesis, $A[1..i]$ are the smallest elements, and $A[i+1..n]$ contains the n-i largest elements of $A[1..n]$ sorted. Therefore, when moving the largest element in $A[1..i]$ to position i and decreasing the heap size by one, makes it so $A[i..n]$ contains the n-(i-1) largest elements sorted and $A[i..i-1]$ still has the smallest elements. Heapify in line 5 makes sure that $A[i..i-1]$ is now a heap, and so in the beginning of the next iteration we can rely on A[1] being the largest again.
   - **Termination:** In the last loop i=2. After the loop terminates, $A[2..n]$ contains the n-1 largest elements sorted and $A[1]$ contains the remaining element, being the smallest per the inductive hypothesis, therefore the entire array is sorted.

2. Describe an $O(n \log k)$ algorithm for merging k sorted lists into one sorted list, where n is the total number of elements. **Hint:** Think of the merging part of MergeSort and extend it to multiple lists. Remember that the lists are not necessarily the same size.

   Solution: Maintain a min-heap of size k that contains the smallest element of each list that has not yet been processed.

   - In the beginning, build a heap from the smallest elements of each list. Each element has its numerical value as key but also a pointer to the original list – O(k).
   - At every step, extract-min the heap. Let us say it came from Heap i. Insert the next element on heap i if it exists – O(log k) – because the heap is always at most of size k. If the list is empty, do nothing (the lists don't have to be the same size, and sometimes a list runs out before others).
   - You repeat it until all the n elements are processed – so the total number of heap operations take O(n log k).

3. The procedure `Max-Heap-Delete(A,i)` deletes the item in node i from heap A. Give an implementation of `Max-Heap-Delete` that runs in $O(\log n)$ for an n-element Max-Heap. Assume that the heap elements are mapped into indices, so you have access to the i^{th} node. Note that

the problem asks you to give an algorithm that runs in $O(\log n)$ time. So you not only have to give the algorithm, you also have to show that it really does run in $O(\log n)$ time.

Solution: One way to do it is the following:

- Swap $A[i]$ with $A[n]$ (the last node in the heap) – O(1)
- heapsize(A) = heapsize(A) - 1 (reduce the size by 1), thus removing that element – O(1).
- Now the previous $A[n]$ is in position i. It may or may not be in the right place, so we need to heapify it – O(log n).
- It may not be enough, though. Heapify only "sinks" a node and does not float it up. There is a boundary case where the previous $A[n]$ may be bigger than its parent. See for example the heap on the right side of slide 14 in slide set 3. Say we want to delete the rightmost node, whose key is 7. If we swap it with $A[n]$ whose value is 9, we will have a situation where a child is bigger than its parent (8). To prevent this, we also have to check the parent and "float up" if needed. This also costs at most O(log n).

4. Suppose you start with a rectangular array of numbers. Perform the following operations:

   - First sort each row (smallest to largest).
   - Then sort each column (smallest to largest).

   Show that after sorting the columns, each row is still sorted. (Hint: Prove by contradiction).

   Solution: This can probably be shown in more than one way, here is the one I found easiest (there are certainly more elegant ways though): Let's look at the first row a after the sorting of rows and then columns. Let us look at any two indices i, j such that $i < j$. The element at position i at row a after the double sorting was originally the i^{th} smallest element, following row sorting and before column sorting, of some row l – let us call this element l_i, and the element at position j was, after row sorting and before column sorting, the j^{th} smallest element at some row m, denoted m_j. Since this is the first row, then l_i is the smallest element in column i, and in particular - $l_i \leq m_i$ where m_i was the element that was originally the i^{th} smallest at row m after row sorting. Since $i < j$, then after row sorting $m_i \leq m_j$ and therefore $m_j \geq l_i$. This is true for any i, j and therefore the top row is sorted. We can remove it from consideration and apply the same logic to the second row which is the smallest remaining etc.

5. Exercise 3.1 in the Lecture 3 handout (on page 7 of the handout). Don't be sloppy here! I'm looking for a precise explanation.

   Answer: Proof by induction on the heap property.

   - Base case: i is a leaf (level 0). It has no children, so trivially it is a heap and no action is taken in the code.
   - Inductive hypothesis: We heapify a node at height $0 \leq k < h$. That is – the two children of node i, if they exist, are heaps but i may be smaller than one or two of its children. Lines 3–10 in the code select the largest of i and its children. If i is the largest (or has no children) we do nothing. By inductive hypothesis the two children are the root of heaps and so A[i], the largest, is the root of a heap that contains as children the heaps rooted at its children, if they exist.
   - If i is not the largest, we swap it with the largest of its two children – call it $A[largest]$. By inductive hypothesis, $A[L]$ and $A[R]$ are the largest of their subtrees, and so $A[largest]$ is the largest overall (including both subtrees and $A[i]$), so putting it at the root of the overall heap maintains the heap property at that level. Notice that by "sinking" A[i] down one level it may not be the largest of its now subtree, but we can use the inductive hypothesis at any level below k until we reach a leaf.

6. Exercise 6.1 in the Lecture 3 handout (on page 13 of the handout).

   Answer:

   - Build a min-heap – O(n).
   - run the first k stages of Heapsort (adapted to min-heap) – O(log n) each.

7. Exercise 3.1 from the Lecture 4 handout (page 7).

   Answer: The costs come from the total overhead of partition. At the top level, it is cn. at the second level – $0.9cn + 0.1cn = cn$ (remember that the cost of partition is proportionate to the size of the array). At the third level there are four branches – $0.01cn + 2 * 0.09cn + 0.81cn = cn$ In general, every full level contains the combinations of multiples of 0.1 and 0.9 that behave like the binomial formula: $cn * \sum_{k=0}^{n} \binom{n}{k} 0.1^k 0.9^{n-k} = cn * (0.9 + 0.1)^n = cn$. The partial levels occur when some short branches end before longer branches, so cn is the upper bound.

8. Based on exercises 8.2-1 - 8.2-3 in the 4^{th} edition (slightly edited).

   (a) Show the run of counting-Sort on the array $A = [6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2]$. No need to draw out every stage. Just show the array C after line 5, C after line 8 and the first three stages of the final sorting (similar to figure 8.2)

   Here is the input array and the frequency count:

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	2	2	2	1	0	2

   Here is the input array and the relative order count:

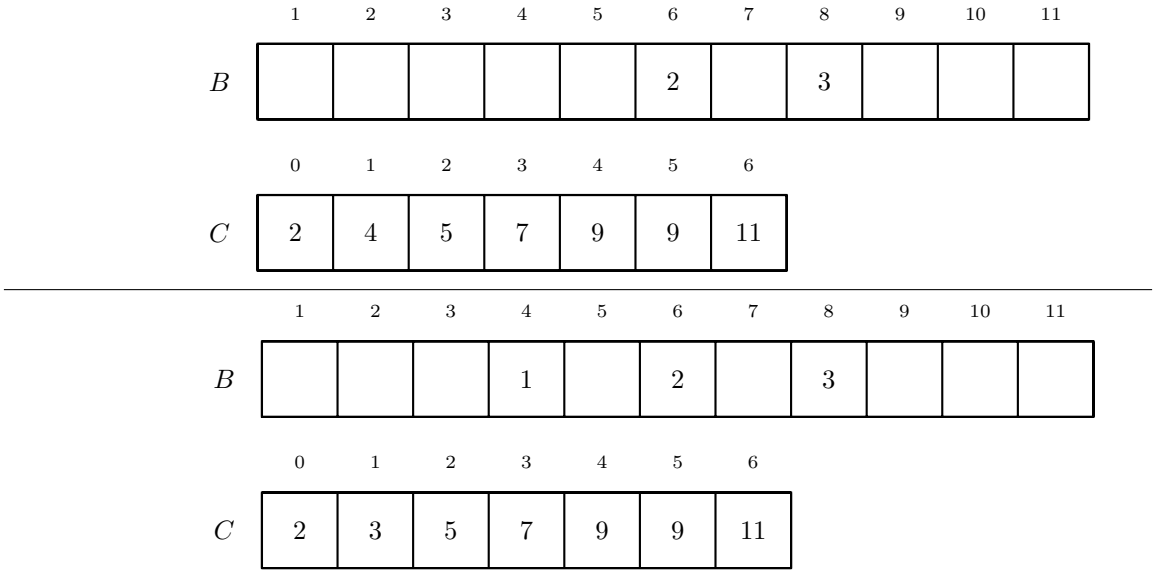
	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	4	6	8	9	9	11

   And the first three stages of the sorting:

	1	2	3	4	5	6	7	8	9	10	11
B						2					

	0	1	2	3	4	5	6
C	2	4	5	8	9	9	11



(b) Prove that Counting-Sort is stable. In other words, equal elements appear in the sorted array in the same order as they were in the original input – if we have two equal elements $A[i]$ and $A[j]$ such that $i < j$, $A[i]$ will appear before $A[j]$ in the sorted array.

Answer: Say we have two equal elements $A[i]$ and $A[j]$ such that $i < j$. Since we traverse A from end to start, we will look at $A[j]$ before $A[i]$. It will be placed at the index indicated by $C[A[j]]$, which is then decreased. When we get to $A[i]$, it will be placed by $C[A[j]]$ which now has a smaller value, therefore it will appear before $A[j]$.

(c) Show that if we rewrite the for loop header in line 11 of the Counting-Sort pseudo code as `for i=1 to n` (going from the start to the end), the algorithm still works properly, but it is not stable. Then rewrite the pseudocode for counting sort so that elements with the same value are written into the output array in order of increasing index and the algorithm is stable.

Solution: If the only thing we change is the loop header in line 11, then we traverse A from start to end, and we will look at $A[i]$ before $A[j]$. However, since we still put A[i] at the index indicated by $C[A[i]]$, and then decrease it, then when later we look at $A[j]$ it will be placed before $A[i]$, so the order is reversed.

9. Problem 8-5 (a-d only) (page 207 in 3rd edition, 221 in 4th edition).

(a) Being 1-sorted means the array is sorted. If we substitute $k = 1$ in the formula above we get $\sum_{j=i}^{i} A[j] \leq \sum_{j=i+1}^{i+1} A[j] \rightarrow A[i] \leq A[i+1]$ for each i, which is the definition of sorting.

(b) For example, $\{2, 1, 3, 4, 5, 6, 7, 8, 9, 10\}$ is 2-sorted. For the first pair, $\frac{\sum_{j=1}^{2} A[j]}{2} = 1.5$, and $\frac{\sum_{j=2}^{3} A[j]}{2} = 2$, so the condition holds for the first three indices. For the other consecutive pairs it's obviously true because the rest of the array is sorted.

(c) We can show that the definition of k-sorting is equivalent to the condition in the question. By definition, $\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$, which means $\frac{\sum_{j=i}^{i+k-1} A[j]}{k} - \frac{\sum_{j=i+1}^{i+k} A[j]}{k} \leq 0$ or equivalently, $\sum_{j=i}^{i+k-1} A[j] - \sum_{j=i+1}^{i+k} A[j] \leq 0$ (after multiplying both sides by k). These two sums have most

terms in common except the first of the first and the last of the last, so when subtracting one from the other most terms cancel out and we get $A[i] - A[i+k] \leq 0$ or $A[i] \leq A[i+k]$

(d) According to (c) above, an array is k-sorted iff $A[i] \leq A[i+k]$ for every i. Therefore, to k-sort an array we only need to make sure that every element $A[i]$ is only sorted with respect to its $i + dk$ neighbors where d=1,2... . So, we divide the array into k subsets of size $\frac{n}{k}$ and sort them using merge-sort or heap-sort. Each sorting is $O(\frac{n}{k}\log(\frac{n}{k}))$, and we do k such sorting operations, one per subset. Then we write down the smallest of each element. Next the second smallest etc. This way we make sure every $A[i] \leq A[i+k]$ because they are from a sorted set. This only adds an $O(n)$. Overall the runtime is $O(k\frac{n}{k}\log(\frac{n}{k})) = O(n\log(\frac{n}{k}))$.