Problem Set 2 Solutions

Problem 2-1. Sorting can be boring!

A comparison sort is boring if, for infinitely many $n$, when sorting some sequence of $n$ items, at least one item (or any copy of it) gets compared $\Omega(n)$ times. In other words, a comparison sort is boring if there exists a positive constant $c > 0$ such that, for infinitely many values of $n$, there is a sequence of $n$ items on which the sorting algorithm compares one item at least $cn$ times. Assume throughout this question that all $n$ items must be distinct.

(a) Is Mergesort boring? Prove your answer.

**Solution:** Yes. If the initial sequence is already sorted, the first item in the second half of the sequence is compared $n/2$ times, once with each element in the first half of the sequence.

(b) Is Quicksort boring? Prove your answer.

**Solution:** Yes. In the first “pass” that partitions the $n$ items, each item is compared once with the pivot, which is therefore compared $\Omega(n)$ times.

(c) Is Heapsort boring? Prove your answer.

**Solution:** Yes. Consider an array holding a max-heap of $n$ items, where all items in the left subtree (of the root) are smaller than all items in the right subtree. Let us sort the array with Heapsort - note that the initial call to BUILD-MAX-HEAP moves no items, since the array already contains a max-heap. Clearly, after the initial root is extracted, each element in the initial right subtree must be extracted from the heap before any element in the left subtree can be extracted. Now observe that, until all elements of the (initial) right subtree are extracted, the item that is initially the root of the left subtree is not moved, since it is never a leaf and never “wins” a comparison with the root (i.e. the largest element) of the right subtree. Since there is one such comparison for every item that is extracted from the heap (after the initial root), and the number of elements in the (initial) right subtree is clearly $\Omega(n)$, the item that is initially the root of the left subtree is compared $\Omega(n)$ times.

(d) **Bonus Part:** This part is optional. However, solving this part does not exempt you from solving the other parts of this question. Modify any one of the three sorting algorithms above that you find boring, in such a way that it is no longer boring (but keep the time to sort $n$ items $O(n \log n)$—in expectation if randomized).
Solution: Consider the following INTERESTINGMERGE algorithm that merges two sorted sequences of 0 or more items:

If either of the two sequences contains no items, output the other sequence; otherwise let the two sequences be \(a_1, \ldots, a_p\) and \(b_1, \ldots, b_q\) (with \(p, q \geq 1\)), and:

1. Compare \(a_1\) and \(b_1\), and assume, without loss of generality, that \(a_1 \leq b_1\).
2. Find the smallest \(i\) such that \(b_1 \geq a_{2i-1}\) and either \(2^i > q\) or \(a_{2i} > b_1\); start with \(i = 1\) and increase \(i\) by 1 until \(2^i > q\) or \(a_{2i} > b_1\) (a “doubling trick”).
3. Find the largest \(j\) such that \(a_{j-1} \leq b_1 < a_j\) by performing a binary search in the interval \([a_{2i-1}, a_{2i}]\) (or \([a_{2i-1}, a_q]\) if \(2^i > q\)).
4. Output \(a_1, \ldots, a_{j-1}\), followed by \(b_1\), followed by the INTERESTINGMERGE of the two remaining subsequences.

One can prove that, when merging two sorted sequences with a total of \(n\) items through INTERESTINGMERGE, each item is compared \(O(\lg n)\) times. Note that from step 1 to step 3 the only item that is compared more than once is \(b_1\), which is compared \(O(\lg n)\) times and is then placed into the sorted sequence. Any other item that is compared, is compared only once, and then either it is placed into the sorted sequence (if it was no larger than \(b_1\)) or at least half the items preceding it are placed into the sorted subsequence. Thus, an item will be compared \(O(\lg n)\) times before every other item preceding it has been placed into the sorted sequence, and then \(O(\lg n)\) more times before it is itself placed into the sorted sequence.

It is also easy to see that INTERESTINGMERGE outputs a sorted sequence of \(n\) items (by induction on the number of items in the smaller of the two sequences to be merged - the base case being 0 items), and that it runs in time \(O(n)\): each item is moved only to be placed into the sorted sequence, and the number of comparisons necessary to perform the “doubling trick” in step 2 and the binary search in step 3 is logarithmic in the number of elements placed into the sorted sequence before INTERESTINGMERGE is called again in step 4.

An implementation of Mergesort that sorts \(n\) elements by repeatedly merging sorted sequences through INTERESTINGMERGE takes time \(O(n \lg n)\); and because every item is involved in \(O(\lg n)\) merging passes, every item is compared \(O((\lg n)^2) = o(n)\) times—so it is not boring!

Of course, one could instead modify Quicksort or Heapsort...
must be at least $\log(n!) = \Omega(n \log n)$. Because the depth of the decision tree is a lower bound on the worst-case sorting time, the worst-case running time of Radix Sort with a radix of 2 must be $\Omega(n \log n)$.

On the other hand, we know that Radix Sort with a radix of 2 sorts $b$-bit integers in $O(nb)$ worst-case time. This time bound is $o(n \log n)$ if $b = o(\log n)$. So the decision-tree lower bound cannot give an $\Omega(n \log n)$ worst-case lower bound in this case.

(a) Explain why the $\Omega(n \log n)$ decision-tree lower bound fails when $b = o(\log n)$.

**Solution:** The decision-tree lower bound argues that every permutation must appear as a leaf in the decision tree because there is an input that has just that permutation as its sorting permutation. However, this assumes that every permutation on $n$ items can be represented as input. If $b < \log n$, the number of distinct representable integers is $2^b < n$. Hence, in this case, the input integers are not all distinct, so they cannot represent an arbitrary permutation.

(b) What lower bound can you derive on sorting $n$ $b$-bit integers, using the decision-tree model described above? State your lower bound using $\Omega$ notation in terms of both $n$ and $b$.

**Solution:** There are actually two views of the sorting problem. It is fine to consider either one; here we consider both.

In the first view, the goal of the sorting problem is to find a permutation of the numbers that results in sorted order. This view is particularly helpful, for example, if there is auxiliary data associated with each integer key, and the auxiliary data needs to be permuted along with the key. In this case, we proceed as follows.

To derive a worst-case lower bound, it suffices to consider a particular family of examples. Consider an input containing $n/2^b$ copies of each integer from 0 to $2^b - 1$. Let $N_{n,b}$ denote the number of distinct permutations of such integers. Because the permutation made by the sorting algorithm on all of these different permutations must be different, we have that the number of leaves of the decision tree must be at least $N_{n,b}$. Thus the depth of the decision tree, and hence the worst-case sorting time, must be at least $\log N_{n,b}$.

It remains to prove a lower bound on $N_{n,b}$. Consider the number of permutations of $n$ elements where the first $2^b$ elements form a permutation of the set $\{0, \ldots, 2^b - 1\}$, the next $2^b$ elements form a permutation of the same set, and so on. The number of such permutations is at least $(2^b!)^{n/2^b}$. We conclude that $N_{n,b} \geq (2^b!)^{n/2^b}$. Using the approximation that $\log k! = \Omega(k \log k)$, we have $\log N_{n,b} \geq (n/2^b) \log (2^b!) = \Omega((n/2^b)2^b \log 2^b) = \Omega(n \log 2^b) = \Omega(nb)$. This bound is the best possible because Radix Sort with a radix of 2 attains a matching upper bound.

In the second view, the goal of the sorting problem is to output a sorted list of the input numbers. In this case, the examples considered above do not suffice, because
the sorted output is always the same: $n/2^b$ copies of each integer in order. Generating inputs with different sorted outputs is easy: we just need to choose different frequencies for each integer from 0 to $2^b - 1$. There are $N'_{n,b} = \binom{n}{2^b}$ such choices, and therefore the number of leaves in the decision tree is at least $N'_{n,b}$: there must be at least one leaf per possible output. Hence the height of the decision tree is at least 

$$\log N'_{n,b} = \log \left( \binom{n}{2^b} \right) \approx n \log n - 2^b b - (n - 2^b) \log(n - 2^b) \geq 2^b(\log(n - 2^b) - b),$$

which is $\Omega(2^b \log n)$ in the case $n > 3 \times 2^b$. This bound is quite weak, but we need to use different lower bound tools than decision trees for a tight bound in this setting.

Note that all of these lower bounds assume that the algorithm simply reads single bits of the input at a time. This is called the \textit{bit-probe model}. In this model, it is not surprising that all $nb$ bits of the input need to be read. In contrast, outside the bit-probe model, we could use radix sort with a radix of $n$ and an auxiliary sort of counting sort to sort $n b$-bit integers in $O(n + nb / \log n)$ time, which is usually better than our $\Omega(nb)$ lower bounds.

On a very fast planar chip, the time to access the $i$th memory location is bounded by the speed of light, and is thus proportional to the distance between the CPU and that memory location. If a conceptually linear memory is arranged in a spiral around the CPU, the time to access the $i$th location is then $\Theta(\sqrt{i})$. In the next three problem parts, you will consider the \textit{HMM}_\sqrt{x} model of computation, where it costs $\sqrt{x}$ “time” to access memory location $x$, for any integer $x \geq 1$. (HMM stands for Hierarchical Memory Model.) In this model, we assume that the cost of performing a computation (e.g., a comparison) is included in the cost of accessing its inputs.

\textbf{(c)} On the \textit{HMM}_\sqrt{x}, what is the expected running time of \textsc{Randomized-QuickSort}? (Give your answer using $\Theta$ notation.) Assume that the elements of the array $A[1..n]$ occupy memory locations 1 through $n$, respectively, and that the $\Theta(1)$ index variables $(i, j, p, q, r)$ can be accessed for free.

\textbf{Solution:} $\Theta(n^{3/2} \log n)$.

Let us first prove that the time taken is, in expectation, $O(n^{3/2} \log n)$. Because Quick-sort sorts in place, it only uses the first $\Theta(n)$ memory locations. The cost of accessing each element is $O(\sqrt{n})$. Thus the total cost is $O(\sqrt{n})$ times the standard running time. By linearity of expectation, the expected running time on the HMM is $O((n \log n) \sqrt{n}) = O(n^{3/2} \log n)$.

Now let us prove that the expected running time is $\Omega(n^{3/2} \log n)$. There is a constant $p > 0$ probability that, when partitioning the $n$ items into two subarrays, both subarrays have size at least $n/10$. In this case, to recursively sort one of the two partitions, one has to sort $n/10$ items located in a memory area where each access costs $\Omega(\sqrt{n})$. (Remember that an item, once placed by QuickSort into a subarray, never leaves that subarray.) Thus the expected time taken is $\Omega((n \log n) \sqrt{n}) = \Omega(n^{3/2} \log n)$. 

\begin{problem}

\textbf{(c)} On the \textit{HMM}_\sqrt{x}, what is the expected running time of \textsc{Randomized-QuickSort}? (Give your answer using $\Theta$ notation.) Assume that the elements of the array $A[1..n]$ occupy memory locations 1 through $n$, respectively, and that the $\Theta(1)$ index variables $(i, j, p, q, r)$ can be accessed for free.

\textbf{Solution:} $\Theta(n^{3/2} \log n)$.

Let us first prove that the time taken is, in expectation, $O(n^{3/2} \log n)$. Because Quick-sort sorts in place, it only uses the first $\Theta(n)$ memory locations. The cost of accessing each element is $O(\sqrt{n})$. Thus the total cost is $O(\sqrt{n})$ times the standard running time. By linearity of expectation, the expected running time on the HMM is $O((n \log n) \sqrt{n}) = O(n^{3/2} \log n)$.

Now let us prove that the expected running time is $\Omega(n^{3/2} \log n)$. There is a constant $p > 0$ probability that, when partitioning the $n$ items into two subarrays, both subarrays have size at least $n/10$. In this case, to recursively sort one of the two partitions, one has to sort $n/10$ items located in a memory area where each access costs $\Omega(\sqrt{n})$. (Remember that an item, once placed by QuickSort into a subarray, never leaves that subarray.) Thus the expected time taken is $\Omega((n \log n) \sqrt{n}) = \Omega(n^{3/2} \log n)$.

\end{problem}
(d) Modify \textsc{Randomized-Quicksort} so that it sorts \( n \) items in \( O(n^{3/2}) \) expected time on the HMM. Analyze the expected time taken by your algorithm. Assume that the \( n \) inputs are initially in memory locations 1 through \( n \), and that the outputs must be placed in nondecreasing order into locations 1 through \( n \).

\textit{Hint:} One way to analyze this algorithm is to modify the analysis from Lecture 4 of \textsc{Randomized-Quicksort}. This yields a recurrence of the form

\[ T(n) \leq \frac{2}{n} \sum_{k=1}^{n-1} T(k) + f(n). \]

As in lecture, this recurrence can be solved by the method of substitution, now using the guess that, for some positive constants \( c \) and \( n_0 \), \( T(n) \leq c n^{3/2} \) for all \( n \geq n_0 \). Of course, you should verify the guess by induction. When evaluating tricky sums, you might find it helpful to use inequality (A.11) on page 1067 of CLRS.

\textbf{Solution:} Consider the following (minor) modification of \textsc{Randomized-Quicksort}:

If \( n > 1 \):

1. Partition the input array into two subarrays as in standard Quicksort, but with the items larger than the pivot placed into memory locations 1, \ldots, \( k - 1 \), the pivot into location \( k \), and all remaining items (no larger than the pivot) into locations \( k + 1, \ldots, n \), for some \( k \leq n \).

2. Recursively sort the set of “large” items in locations 1, \ldots, \( k - 1 \) in \textit{non-increasing} order.

3. Reverse the array of memory locations 1, \ldots, \( n \), by swapping the item in location \( i \) with that in location \( n - i + 1 \), for \( 1 \leq i \leq \lfloor n/2 \rfloor \). This leaves all items larger than the pivot in \textit{non-decreasing} order in memory locations \( n - k + 2, \ldots, n \), the pivot in location \( k + 1 \), and remaining items in locations 1, \ldots, \( n - k \).

4. Recursively sort the first \( n - k \) locations (in non-decreasing order).

It is clear that our algorithm correctly sorts in place, because all items larger than the pivot end up in non-decreasing order after all other items, which are also sorted in non-decreasing order, and no location beyond the \( n \)th is accessed.

The number of memory accesses performed in Steps 1 and 3 is \( O(n) \), and each memory access costs \( O(\sqrt{n}) \); therefore the total time spent in these steps is at most \( c n^{3/2} \) for some constant \( c > 1 \). Assuming a random choice of the pivot, the algorithm obeys the recurrence:

\[ T(n) \leq \frac{2}{n} \sum_{k=1}^{n-1} T(k) + c n^{3/2}, \]

with a base case of \( T(1) = c \).

We now prove by induction on \( n \) that \( T(n) \leq 5c n^{3/2} \) for all \( n \geq 1 \). This claim is clearly true in the base case \( n = 1 \) because \( T(1) = c \leq 5c 1^{3/2} \). Now suppose that
$T(k) \leq 5c k^{3/2}$ for all $0 < k < n$. Then

$$T(n) \leq \frac{1}{n} \sum_{k=1}^{n-1} T(k) + c n^{3/2}$$

$$\leq \frac{2}{n} \sum_{k=1}^{n-1} 5c n^{3/2} + c n^{3/2}$$

$$\leq \frac{2}{n} \int_0^n 5c x^{3/2} \, dx + c n^{3/2}$$

$$\leq \frac{2}{n} 5c \frac{2}{5} n^{5/2} + c n^{3/2}$$

$$= 5c n^{3/2}.$$

(e) Is there an algorithm that sorts $n$ distinct items in $o(n^{3/2})$ expected time on the HMM $\sqrt{x}$?

**Solution:** No. Just reading the items initially placed in the $\lfloor n/2 \rfloor$ most expensive locations, $\lceil n/2 + 1 \rceil, \ldots, n$, takes time no less than $\lfloor n/2 \rfloor (n/2)^{1/2} = \Omega(n^{3/2})$. 