CSCI 3333 Homework: Sorting (with Solutions)

1 Algorithm Analysis

Problem 1. (Based on Book Exercise 7.1) Show the step-by-step result of performing insertion sort on the following array: 3, 5, 2, 1, 5, 6, 0.

Solution 1. See Figure 1.

Problem 2. (Based on Book Exercise 7.2) Give the worst-case asymptotic running time of the following sorting algorithms for input array of $n$ identical items:

- Insertion sort.
• Mergesort.
• Quicksort (assume pivot is first element).

Solution 2.
• Insertion sort: $\Theta(n)$.
• Mergesort: $\Theta(n \log(n))$.
• Quicksort: $\Theta(n^2)$.

Problem 3. Mergesort (as seen in class) is actually just one algorithm in a family of merge-sort algorithms that partition the input list into 2, 3, 4, etc. equal-sized sublists; these algorithms are called 2-way, 3-way, 4-way, etc. mergesort. Analyze the running time of 3-way mergesort (give the recurrence relation, find a closed form, prove the closed form correct).

Solution 3. The recurrence relation is $T(n) = 3T(n/3) + cn$, $T(1) = d$.
First, finding a closed form via repeated substitution:

\[
T(n) = 3T(n/3) + cn
= 3(3T((n/3)/3) + c(n/3)) + cn
= 3^{2}T(n/3^{2}) + cn + cn
= 3^{2}T(n/3^{2}) + 2cn
= 3^{2}(3T((n/3^{2})/3) + c(n/3^{2})) + 2cn
= 3^{3}T(n/3^{3}) + cn + 2cn
= 3^{3}T(n/3^{3}) + 3cn
\]

So we guess that $T(n) = 3^{k}T(n/3^{k}) + kcn$. Let $k = \log_{3}(n)$. Then $T(n) = nT(1) + \log_{3}(n)cn = dn + cn\log_{3}(n)$.

Now prove it by induction:
Base case: $T(1) = d = d + c \cdot 1 \cdot 0 = d \cdot 1 + c \cdot 1 \cdot \log_{3}(1)$.
Inductive step: Assume $T(n) = dn + cn\log_{3}(n)$. Then

\[
T(3n) = 3T((3n)/3) + c(3n)
= 3T(n) + 3cn
= 3(dn + cn\log_{3}(n)) + 3cn
= 3dn + 3cn\log_{3}(n) + 3cn
= 3dn + 3cn + 3cn\log_{3}(n)
= 3dn + 3cn(1 + \log_{3}(n))
= 3dn + 3cn(\log_{3}(3) + \log_{3}(n))
= 3dn + 3cn\log_{3}(3n)
= d(3n) + 3cn\log_{3}(3n)
\]
**Proved!**

So \( T(n) = dn + cn \log_3(n) = \Theta(n \log(n)) \) is the running time of 3-way mergesort.

**Problem 4.** Suppose you will run quicksort **only** on inputs that are almost sorted (i.e., \( A[i] \leq A[j] \) for all but at most \( O(1) \) index pairs \( i, j \) with \( i < j \)). For these inputs, give the worst-case running time of the following sorting algorithms:

- Insertion sort.
- Mergesort.
- Quicksort (selecting the first element as pivot).

**Solution 4.**

- *Insertion sort:* \( O(n) \) time.
- *Mergesort:* \( O(n \log(n)) \) time.
- *Quicksort:* \( O(n \log(n)) \) time.

## 2 Algorithm Design

A sequence of numbers \( A_1, A_2, \ldots, A_n \) is **sign-sorted** provided that for every pair of elements \( A_i, A_j \) with \( i < j \):

- \( A_i, A_j > 0 \Rightarrow A_i \leq A_j \).
- \( A_i, A_j < 0 \Rightarrow A_i \leq A_j \).

**Problem 5.** Give a \( O(n) \)-time algorithm that determines whether an array \( A \) (of length \( n \)) is sign-sorted.

**Solution 5.** Scan \( A \) from front to back, maintaining two variables \( lp \) and \( ln \) whose values sorted the last positive and negative values seen. For each element \( A[j] \):

- If \( A[j] > 0 \) and \( A[j] < lp \), return \text{false}. Let \( lp = A[j] \).
- If \( A[j] < 0 \) and \( A[j] < ln \), return \text{false}. Let \( ln = A[j] \).

Running time: The algorithm runs in \( O(n) \)-time, since it scans through the array once and performs \( O(1) \)-time operations of comparing pairs of floats and assigning a float variable.

Correctness: If the array is not sign-sorted, then there must exist a pair of positive or negative elements \( A[i], A[j] \) that do not obey \( A[i] \leq A[j] \), i.e. have \( A[i] > A[j] \). By transitivity, if \( A[i] > A[j] \), then there are two consecutive positive or negative elements \( A[i'], A[j'] \) that have \( A[i'] > A[j'] \). The algorithm works because it checks all consecutive pairs for this property, finding such a pair if one exists.

**Problem 6.** Give a \( O(n) \)-time algorithm that sorts a sign-sorted array \( A \) (of length \( n \)).
Solution 6. Create a new array $B$ of length $n$. Scan through $A$ from front to back, copying each negative element encountered into the next available location in $B$. Repeat the process for zero elements. Repeat the process for positive elements. Copy the contents of $B$ into $A$ and erase $A$.

Running time: Creating $B$ (and later copying its contents and deleting it) takes $O(n)$ time. The three scans of $A$ each spend $O(1)$ time per loop iteration and iterate $n$ times. Thus the total time spent is $O(n) + 3 \cdot O(n) = O(n)$.

Correctness: The resulting contents of the array, from front to back, are the negative, zero, and positive elements of $A$. Within each group, consecutive elements are larger, since they are copied in the order they appear in $A$ and $A$ is sign-sorted.

A sequence of numbers $A_1, A_2, \ldots, A_n$ is half-sorted provided that for at least half of the consecutive pairs $A_i, A_{i+1}, A_i \leq A_{i+1}$.

Problem 7. Give an algorithm for the following “half-sorting” problem:

- Input: an array $A$ of length $n$.
- Output: an array containing a permutation of the items in $A$ such that at most half of the indices have the property that $A[i] > A[i+1]$.

Now try to give a $O(n)$-time algorithm for the problem. (Hint: if a sequence of things has no two consecutive “bad” things, then at most half of the things are “bad”).

Solution 7. Scan through the array from front to back. If any sequence of three elements have the property that $A[i] > A[i+1] > A[i+2]$, then reverse them. Here’s the code:

```c
void half_sort(float* A, int n)
{
    for (int i = 0; i < n-2; ++i)
        {
            float tmp = A[i+1];
            A[i+1] = A[i+2];
            A[i+2] = tmp;
        }
}
```

Running time: This algorithm consists of a single loop through the array, performing $O(1)$ comparison and assignment operations per loop iteration.

3 Lower Bounds

Problem 8. (Based on Book Exercise 7.52) Prove that any comparison-based sorting algorithm has $\Omega(n \log(n))$ average-case running time.

Solution 8. This is equivalent to proving that the average depth of a node in any binary tree with $n!$ nodes is $\Omega(n \log(n))$.

Proof: Recall that the maximum number of vertices in a binary tree of height $k$ is $2^k$. So in any tree, there are at most $2^k$ vertices in with depth at most $k$.

Recall from the book that $\log(n!) \geq n/2 \log(n/2)$. So there are at most $2^{n/2 \log(n/2)} = (n^2)^{n/2}$ vertices of height at most $n/2 \log(n/2)$ and at least $n! - (n^2)^{n/2}$ vertices of depth more than $n/2 \log(n/2)$. So the average depth is at least

$$n/2 \log(n/2)(n! - (n^2)^{n/2}/n! = n/2 \log(n/2) - (n^2)^{n/2}/n!$$

$$\geq n/2 \log(n/2) - (n^2)^{n/2}/((n^2)^{n/2} \cdot (n/2 - 1)!))$$

$$\geq n/2 \log(n/2) - 1/(n/2 - 1)!$$

$$\geq n/2 \log(n/2) - 1$$

$$= \Omega(n \log(n))$$