Homework 3 Solutions

Qiyam Tung

June 29, 2014

Problem 1

Brief description/reminders of insertion sort:

- When adding new element to list, $|list|$-1 is already sorted.
- Insertion sort will compare and swap until it reaches an element that is equal or less than the new element.

Proof (Inductive):

Basis: $|list| = 1$

By definition, a single element is already sorted. Therefore,

Inductive: if $|list|$ is sorted, then adding a new element e to $list$ (using insertion sort) will produce a sorted array.

Insertion sort will insert the new element in the last slot first.

It will compare with element before it. If the adjacenet element is less than the new element, then we are done because we assumed that all elements in $list$ are sorted.

If it is greater than the new element, then we swap the two values. Repeat these steps until it satisifes the previous condition. Now, all elements after our new element will be sorted because this swap merely moves shifts the previous elements down a box and we know by our inductive hypothesis that this sequence is already sorted.

QED

Problem 2

It would change the worst/avg case's asymptotic running time for comparisons to $O(nlogn)$. However, it would not change the swaps as you would still need to shift the same number of elements down the list. Nothing else would change.

Problem 3

Add a condition before you swap to check to see whether the ith record is already in position. That is, whether the minIndex is the same as the index to be swapped.

While this reduces the number of swaps, it doesn't necessarily decrease the total running time.

It makes very little difference in the average case. It took around 9730 milliseconds to run the original code on 20 random large (32 million elements) arrays whereas it took 9729 milliseconds to run the code with the check on the same arrays.

The avoidance of the swap would reduce the number of swaps to zero in the best case, but increases the number of comparisons by that same amount. So, even in the best case, the average time it takes to sort a sorted array is 0.4483s in the original and 0.4479s in the modified code, which means it runs 0.001% faster, a negligible difference.

The reason is that swap, as an operation, is not terribly costly. Furthermore, the check adds an additional conditional operation. This could lead to more operations than the original in the worst case, as we see in the average times, which are 0.4812s and 0.4984s on the original code and the code with checks, respectively. The code with checks runs 3.6% *slower* than the code without checks due to the fact that every swap is necessary and therefore the conditional checks does not help us avoid the swaps.

Problem 4

With Quicksort, it depends on how you pick the pivot. If it is a median-of-three, it will not work. If the pivot is picked as the first element, then it should be fine so long as when recombining the two halves you are sure to put it in the correct position.

Shellsort is not stable because the sublists can move the keys out of position. For example, given a list like 5 3_1 3_2 7 8 2, if there were two sublists, then sorting it would make the penultimate list be 3_2 2 5 3_1 8 7, and Insertion sort over the entire list will keep this list in its original order.

To make Selection Sort stable, insert the minimum to the first position rather than swap. If you use a linked list, then this does not change the "swap" big-O running time (O(n)). Using an array, however, would make it $O(n^2)$

The rest of the sorts are stable. Insertion sort and Bubble sort swap adjacent element only if it is greater than (or less than, depending on your implementation). Likewise, in Mergesort, when merging two elements, simply pick the one that came before (the left half of the merge) when there it is comparing duplicate keys.

Problem 5

1. Best case: 1 2 3 4 5
2. $F(n) = 2F(n-1) + c$, $F(1) = c_0$

$$F(n) = 2F(n-1) + c$$

$$F(n-1) = 2F(n-2) + c$$

$$F(n-2) = 2F(n-3) + c$$

$$F(n-1) = 2(2F(n-3) + c) + c$$

$$F(n-1) = 2^2F(n-3) + 2c + c$$

$$F(n-1) = 2^2F(n-3) + 3c$$

$$F(n) = 2(2^2F(n-3) + 3c) + c$$

$$F(n) = 2^3F(n-3) + 6c + c$$

$$F(n) = 2^3F(n-3) + 7c$$

$$F(n) = 2^kF(n-k) + (2^k - 1)c$$

Let $k = n - 1$

$$F(n) = 2^{n-1}c_0 + (2^{n-1} - 1)c$$

$$F(n) = 2^{n-1}(c_0 + c) - c$$

3. 5 4 3 2 1
4. I assumed c instead of 1. Just substitute in 1

Proof (Inductive):

Basis: n = 1

$F(1) = c_0$ and $F(1) = 2^0(c_0 + c) - c = c_0$

Inductive: $F(n) = 2^{n-1}(c_0 + c) - c \to F(n+1) = 2^n(c_0 + c) - c$

$F(n+1) = 2F(n) + c$

By IH, $F(n+1) = 2(2^{n-1}(c_0 + c) - c) + c = 2^n(c_0 + c) - c$

QED

5.

n	Comparison (w)	w_i/w_{i-1}
10	129	–
20	1159	8.984
40	9919	8.558
80	82239	8.291
160	670079	8.147
320	5410559	8.074
640	43486719	8.037
1280	348707839	8.018

Problem 6

Run the basic counting sort algorithm (where each element is the count, not the sum of previous counts). Have a counter set to 0 and go from the end of the count array and move down until you find an element with a non-zero count. For every element found, append these elements and increment your counter each time you do (and decrement the value in the count array). Once the count is equal to k, take the output array and reverse it.

In all these operations (counting, outputting, reversing), it will only take $O(n)$. Specifically, for counting and outputting, it will be $O(n + r)$, where r is the range of the values. Outputting takes $O(n + k)$ because the worst case is that all the largest elements are the smallest element, meaning that the program will have to move down all the way down to the first element and then output k elements. Reversing a list takes $O(n)$ time.