EC330 HW4 Solution

Emre Ateş

October 19, 2016

1. (a) The numbers given in reverse order is the worst case for insertion sort. It does 10 comparisons for the first number until it reaches the appropriate location, 9 for the next etc. which adds up to $\Omega(n^2)$ comparisons.

$$\{9, 8, 7, 6, 5, 4, 3, 2, 1, 0\}$$

   (b) The worst case running time is when almost all (the order of n) elements are in the same bucket. In this case, bucket just gives all of the elements to the next sorting algorithm, which runs in $O(n^2)$ time.

   If we modify the sorting algorithm to a faster algorithm that runs in $O(n \log n)$ time, like quick sort or merge sort, then the worst case running time will be reduced, and the linear time will be preserved for cases where the elements are still appropriately split into buckets.

2. (a) The running time is $\Theta(n \log n)$. At every iteration of quick sort, the numbers will be perfectly split into two groups, so there will be $\log n$ iterations that each take $\Theta(n)$ time.

   (b) We can take a sorted array duplicate it at the end at the reverse order. Quick sort would always pick the largest number in this case.

$$\{1, 2, 3, 4, 5, \textcircled{5}, 4, 3, 2, 1\}$$

$$\{1, 2, 3, 4, \textcircled{4}, 3, 2, 1, 5, 5\}$$

$$\{1, 2, 3, \textcircled{3}, 2, 1, 4, 4, 5, 5\}$$

$$\{1, 2, \textcircled{2}, 1, 3, 3, 4, 4, 5, 5\}$$

$$\{1, \textcircled{1}, 2, 2, 3, 3, 4, 4, 5, 5\}$$

3.
   - *Insertion sort:* It is stable. If two elements have the same magnitude, the sorting algorithm wouldn't swap them. If there is an equal element at another place in the array, it would stop when it reaches one of them, and it would not swap with any of them.
   - *Merge sort:* It is stable. Equal elements would again not be swapped, and merging is done in order.

- *Heap sort:* It is not stable. Some elements might be swapped during sorting.
- *Quick sort:* It is not stable. If an element is picked as pivot, it might be swapped with another element that is the same value as the current element.
- *Scheme:* We can use the element's original key in the array to break ties, and we can append this key to the element in $\Theta(n)$ time.

  This approach requires $\Theta(n)$ extra space to keep the key of every element with itself, and $\Theta(n)$ extra time to append the keys. Looking up the keys is constant time.

4. We can sort the array using a $\Theta(n \log n)$ sorting algorithm. After that, we can go over the array and delete any duplicates.
5. For brevity, it is assumed that all array accesses are wrapped with if statements that prevent accessing elements that are out of the range of the array. The algorithm is in the following page.

```
Function Left(int i):
    return $2i$;
Function Right(int i):
    return $2i + 1$;
Function MaxHeapify(int A[], int n):
    if A[Left(n)] > A[n] then
        largest = Left(n);
    else
        largest = n;
    end
    if A[Right(n)] > A[largest] then
        largest = Right(n);
    end
    if largest $\neq$ n then
        Swap(n, largest);
                /* The swap function is assumed to be already implemented */
        MaxHeapify(A, largest);
    end
    return;
Function Pop(int A[]):
    head = A[0];
    A[0] = A[n];
    delete A[n];                                   /* We reduce the array size by 1 */
    MaxHeapify(A, n);
    return head;
Function Main(int A[]):
    x = A[0];
    for $i = 0 \to k$ do
        temp = Pop(A);
        if temp $\neq$ x then
            return False;   /* We found an element that is different from x */
        end
    end
    return True;             /* All of the first k elements are equal to x */
```