

CS231 Fall 2013 Lecture 24

Stephanie R Taylor

October 30, 2013

1 Administrative Topics

- I return quizzes.

2 Sorting

Before we return to priority queues (which we will use to implement a sorting algorithm), let's return to two sorting algorithms we briefly analyzed a few weeks ago. Selection sort and insertion sort.

2.1 Selection sort

The strategy taken in a selection sort is to find the smallest item and put it in the first position (by swapping the current item 0 with the newly found smallest item). Then to find the second small item and put it in the second position, etc. In other words, it fills in each slot in the list, each time doing a sequential search through the unsorted remainder of the list.

```
public static void selectionSort(int[] a) {  
    for (int i=0; i< a.length; i++) {  
        int j = findIndexOfSmallestInt(a, i, a.length-1);  
        swapIntegers(a, i, j);  
    }  
}
```

```

private static int findIndexOfSmallestInt(int[] a, int low, int high) {
    int winner = low;
    for (int i=low+1; i<=high; i++) {
        if( a[i] < a[winner] )
            winner = i;
    }
    return winner;
}

private static void swapIntegers(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

```

The time complexity of a selection sort is $O(n^2)$.

2.2 Insertion sort

The insertion sort, like the selection sort, builds the list one at a time. But instead of searching for the lowest, then second to lowest items. It simply puts the items in order as the items are encountered.

```

public static void insertionSort(int[] a) {
    for (int i=1; i< a.length; i++)
        insert(i, a);
}

private static void insert(int i, int[] a) {
    int temp = a[i];
    while( i > 0 && temp < a[i-1] ) {
        a[i] = a[i-1];
        i--;
    }
    a[i] = temp;
}

```

The time complexity of an insertion sort is $O(n^2)$.

3 Using Priority Queues to sort

You can use PQs for a new sorting algorithm!!

Suppose you create a class PQ that implements the PriorityQueue interface.

```
public static void pqSortA(int[] a, Comparator<Integer> comparator) {
    PQ<Integer> pq = new PQ<Integer>(comparator);

    // add all integers in A to pq
    for( int i = 0; i < a.length; i++ ) {
        pq.add( a[i] );
    }

    // remove all the items from the PQ; they should be in order
    for( int i = 0; i < a.length; i++ ) {
        a[i] = pq.remove();
    }
}
```

Let's think about the performance of this sort using the implementations we talked about on Monday.

3.1 Performance analysis of PQ implementations

In order for pqSort to be a good sorting algorithm, we need to have fast efficient add and remove methods. Lets try to implement our PQ class so that they are both fast.

- (a) Implementation of PQs using an unsorted linked list:
Adding is $O(1)$ (we can just add to one end of the list). Removing is $O(n)$ because we need to examine every element to determine which one has the highest priority.
- (b) Implementation of PQs using a sorted linked list:
Adding requires a list traversal, which takes time proportional to the number of elements. On average, it takes $n/2$ traversal steps, although it is easy to construct a worst case where every add takes n . Remove is constant time.

- (c) Implementation of PQs using an unsorted array:

Actually we need more details, such as whether we are storing the items in the first part of the array and how often we have to copy. But, in general, adding is constant time, except that every once in a while we have to copy the whole array. Removing takes $O(n)$ time.

- (d) Implementation of PQs using a sorted array:

Adding is $O(n)$ because we will likely need to insert into the middle of the array, and will have to shift the rest of the elements over, to make room. Removing is constant time.

Using any of these PQ implementations leads to a sorting method of $O(n^2)$. Why? Because we call both add and remove n times. As long as one of them is $O(n)$, the sort is (n^2) .

None of them have the speed we want, namely something better than $O(n^2)$! However, there is an additional problem with all of them: They take an additional $O(n)$ space. Can we do pqSort with only $O(1)$ additional space? If so, the PQ would need to reuse the same array that contained the original data to be sorted.

[Show how reusing the array in implementation (c) is just selection sort, (d) is just insertion sort.]