

CS231 Fall 2013 Lecture 15

Stephanie R Taylor

October 7, 2013

1 Administrative Topics

- If you missed classes last week, please get the notes from someone in the class. I don't have detailed notes of Bruce's lecture.
- Now that we are into the data structures and algorithm part of the course, that means the free on-line textbook is relevant. I highly recommend it.

2 Writing test code

This course is much different from CS151 in that visual inspection of your final output is not sufficient to test your code. Instead, you need to write good test functions. And you need to test each method separately (insofar as possible). Further, you want to keep your test code so that you can re-run it if you make an adjustment.

When writing test code for the implementation of an abstract data type, you will want to take into consideration the same cases you considered when writing the code. For example, when you write an insert method, test it when the list is empty, which the insert is at the beginning of the list, when the insert is in the middle of the list, and when the insert is at the end of the list.

If you test your code as you write it, you will have a far less painful process debugging your simulations. So, write test code that you are proud of. And make us smile on you when we grade it :).

3 Time complexity analysis

Why are we talking about time complexity analysis? Does it matter? Yes, because most programs can run rather quickly on small amounts of data. But if we move to large amounts, some programs will slow down more than others. We need to understand the rate at which that slow-down will happen.

Loop Example 1: Let's consider some simple for-loops. Suppose they operate on an int array a of length N .

```
1 int sum1 = 0;
2 for (int i=0; i < a.length; i++) {
3     sum1 += a[i];
4 }
```

The time complexity is $O(N)$ because the loop body executes N times and is itself of $O(1)$.

Loop Example 2: What about more summing?

```
1 int sum2 = 0;
2 for (int i=0; i < a.length; i++) {
3     for (int j=0; j < a.length; j++) {
4         sum2 += a[i]*a[j];
5     }
6 }
```

To assess the time complexity of this code, we can look at the line that is executed most often (line 4), determine its complexity, and then determine how many times it is executed. Its complexity is $O(1)$ and it is executed N^2 times. Therefore, the time complexity of this snippet is $O(N^2)$

Loop Example 3: Now for even more summing, with a subtle difference:

```
1 int sum3 = 0;
2 for (int i=0; i < a.length; i++) {
3     for (int j=0; j < i; j++) {
4         sum3 += a[i]*a[j];
5     }
6 }
```

6 }

This has the same summing line as above, but the inner loop executes only i times instead of N times. So the inner loop is executed once for $i=0$, twice for $i=1$, and N times for $i=N-1$. This means line 4 is executed

$$1 + 2 + 3 + \dots + N$$

times. Which is $\frac{N(N+1)}{2}$ times.

Now we will look at several methods used when sorting data (and array of length N) and will determine their time-complexity. They are all in a class called `Sorter`, which I show piece by piece below.

First, we have the methods we need to perform a 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 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 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 void swapIntegers(int [] A, int i, int j)
{
    int temp = A[i];
    A[i] = A[j];
```

```

    A[j] = temp;
}

```

- The time complexity of `selectionSort` depends on the complexity of the other two, so let's start with them.
- The time complexity of `swapIntegers` is $O(1)$ because it does not depend on the size of the list.
- The time complexity of `findIndexOfSmallestInt` is $O(N)$. It is called on segments of the array ranging from length N down to 1, so that averages out to arrays of length $N/2$.
- The time complexity of `selectionSort` is $O(N^2)$. Notice that this method is similar to loop example 3 above. i.e. there are N calls to `findIndexOfSmallestInt` and they take time $N/2$ on average and so the time is $N * N/2$, but we remove constants from Big-Oh notation, so we say it is $O(N^2)$.

Second, we have the methods needed for an 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 void insertionSort(int [] A)
{
    for (int i=1; i< A.length; i++)
        insert(i, A);
}

private 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;
}

```

- `insert` is $O(N)$. It is called on segments of the array ranging from length 1 up to N , so that averages out to arrays of length $N/2$.

- `insertionSort` is $O(N^2)$ because it uses a for loop with N iterations, that calls a method of $O(N)$. (And, like `selectionSort`, the details are similar to that of loop example 3).

Finally, we consider bubble sort.

```
public void bubbleSort(int [] A) {  
    for(int i = A.length-1; i > 0; i--)  
        for(int j = 0; j < i; j++)  
            if(A[j] > A[j+1])  
                swapIntegers(A,j,j+1);  
}
```

This also is of $O(N^2)$ for the same reasons as loop example 3.