

CS231 Fall 2013 Lecture 20

Stephanie R Taylor

October 21, 2013

1 Administrative Topics

- Return the quizzes.

2 Binary search

Suppose you have a sorted list of integers. How could you find a particular integer in that list? What about a binary search? The way a binary search works is you look at the middle element and determine whether the item would be in the first half or the second half of the list. Then search the appropriate half, recursively.

Let's write the code:

```
private static boolean binarySearch( int[] a, int start, int stop, int goal ) {
    if (start == stop) {
        return a[start] == goal;
    }

    int half_dist = (stop-start)/2;
    if (a[start+half_dist] >= goal)
        return binarySearch( a, start, start+half_dist, goal );
    else
        return binarySearch( a, start+half_dist+1, stop, goal );
}
```

```

public static boolean binarySearch( int[] a, int goal ) {
    if (a.length == 0)
        return false;
    return binarySearch( a, 0, a.length-1, goal );
}

```

Note that if we use `start == stop` to detect an array of length 1, then we have to use another way to detect an array of length 0. In the above code, I chose to put that test for an empty array in the public method.

Analysis: There are approximately $\log n$ recursive calls with constant work done in each call. Therefore, the algorithm is $\Theta(\log n)$.

3 Merge sort

The strategy of a merge sort is to recursively split the list into smaller pieces, sort the small pieces, then merge the pieces together until you have one large sorted list.

Here is the code:

```

// sorts (in place) a[start]..a[end]
private static void mergeSort(int[] a, int start, int end) {
    if( start >= end ) return;

    mergeSort(a, start, (start+end)/2);
    mergeSort(a, (start+end)/2+1, end);
    merge(a, start, (start+end)/2, end);
}

public static void mergeSort(int[] a) {
    mergeSort(a, 0, a.length-1);
}

```

3.1 One way to analyze it

Visualize the calls. Instead of looking at it like a stack, I am going to draw it as a tree. (shown in class).

Suppose we have an array of length n .

No comparisons get done on the way down.

At the bottom level of the tree, we have n calls to sort lists of length 1.

It takes $n/2$ comparisons to merge them. (For each pair, we need to determine which item goes first).

At the next level up, we have $n/2$ calls to sort lists of length 2. The work done by each call is merging two lists of length 2. This takes 3 comparisons each, which is $3 * n/2$ total comparisons.

At the next level up, we have $n/4$ calls to sort lists of length 4. The work done by each call is merging two lists of length 4. This takes 7 comparisons each, which is $7 * n/4$ total comparisons.

Then $15 * n/8$

then $31 * n/16$

Ok, we need to simplify this!

First, recognize that the numerator is pretty close to a power of two. We'll round everything up, so our analysis will be a little pessimistic.

$2 * n$

$4 * n/2$

$8 * n/4$

Then $16 * n/8$

then $32 * n/16$

Now we see the pattern!

Each level involves (a little less than) $2n$ comparisons.

How many levels are there? $\log n$. So the total number of comparisons is (a little less than) $2n \log n$. And the algorithm is $\Theta(n \log n)$.

3.2 Comparison to the example in class last Wed

The most complicated example in class last Wed, was actually modeled after the merge sort algorithm. Our analysis in class on Wed was correct. But

because we didn't have a particular algorithm in mind, it was more challenging.

We determined how many recursive calls were made to the function (n) and the average amount of work done at each function (which we estimated to be $\log n$).

In the context of the merge sort algorithm, we instead think about how many layers of calls there are (e.g. the height of the call stack at the bottom of the recursion ($\log n$)). And how much work needed to be done at each layer (n).

This is two ways of looking at the same tree of calls.