22C:21 Lecture Notes
Running time of Binary Search

Sept 3rd, 2008

The last time we met (Friday, 8/29), we discussed *binary search*. This was in the context of the `RecordDB` class. Here is simplified "generic" version of that code. The input parameters are a sorted `int` array called `list` and an `int` value called `key` that is being searched for.

```
public static boolean binarySearch(int[] list, int key)
{
        int first = 0;
        int last = list.length-1;
        int mid;
        while(first <= last)
        {
                mid = (first + last)/2;
                if(list[mid] == key)
                        return true;
                else if(list[mid] < key)
                        last = mid - 1;
                else if(list[mid] > key)
                        first = mid + 1;
        }
        return false;
}
```

We will first figure out what the maximum number of iterations of the `while`-loop in the above code is, as a function of n, the size of the input-array. Note that the portion of the array `list` that is yet to be examined is always between indices `first` and `last`, inclusive of elements `list[first]` and `list[last]`. Thus the size of the portion of the array `list` that is yet to be examined is $\texttt{last} - \texttt{first} + 1$. The following table shows how this quantity decreases as a function of the number of iterations of the `while`-loop.

Number of times the while loop has executed	size of array yet to be examined ($\texttt{last} - \texttt{first} + 1$)
0	n
1	$n/2$
2	$n/4$
.	.
.	.
.	.
i	$n/2^i$

The `while`-loop is executed maximum number of times if `key` is not found. When the function exits the `while`-loop, it does so because `first` has exceeded `last` and the size of the array "yet-to-be-examined" has become 0. Suppose that this happens after t iterations. This means that after $t-1$ iterations, this size must be 1. Note that by consulting the above table, we see that the after $t-1$ iterations, the size of the "yet-to-be-examined" array is $n/2^{t-1}$. For this to be 1, it must be the case that $2^{t-1} = n$, and this happens when $t - 1 = \log_2(n)$. Therefore, the maximum number of iterations of the `while`-loop is $t = \log_2(n) + 1$.

Logarithmic functions. If $a^b = x$, then $b = \log_a(x)$. In other words, $\log_a(x)$ is the quantity to which a has to be raised to get to x. Therefore, if $2^i = n$, then $i = \log_2(n)$. The function $\log_2(n)$ grows *very* slowly as compared to the linear function, n. For illustration, consider this table.

n	$\log_2(n)$
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20

Even when n exceeds a million, $\log_2(n)$ is still at 20. This means that even for a million element array, binary search examines (in the worst case) about 21 elements!

We will now introduce the notion of the *running time* of an algorithm (or a function or a program) and talk about how the running time of an algorithm may be computed. In this class, when we talk about "running time" we don't mean an actual time in milliseconds or microseconds that the program took to run. Instead, we mean a quantity that is a function of the *input size*. For example, if the input to a function `foo` is an array of length n, then we would like to figure out how long `foo` takes to complete as a function of n. We will see many examples of this throughout the semester.

Define a *basic operation* as a line of code (or pseudocode) that runs in *constant time*, i.e., time independent of the input size. It is easy to see that all 12 lines in the `binary search` function are basic operations. For example, consider the comparison (Line 6)

```
if(list[mid] == key)
```

This is a basic operation because:

- Arrays are `random access` data structures, i.e., given any index i, the time it takes to access slot i in the array does not depend on the size of the array or the value of i. Thus `list[mid]` is accessed in constant time. Later we will see many data structures that are not random access, e.g., linked lists, trees, etc. in which the time to reach an item will be a function of the "distance" to that item from some access point to that data structure.
- The value of `key` is obtained in constant time.
- The comparison runs in constant time.

Define the *running time* of an algorithm (or a function or a program) as the number of basic operations executed by the algorithm, when provided input of size n. To see an example of how the running time of a function is calculated, consider the `binarySearch` function above. Suppose that the body of the `while`-loop is executed K times.

- There are 3 basic operations before the `while`-loop, each of which is executed once.
- There is one basic operation after the `while`-loop, that is executed at most once.
- Each time the body of the `while`-loop executes, at most 7 basic operations are executed. Therefore, the body of the `while`-loop contributes at most $7K$ basic operation executions.
- Line 4 is a basic operation that is executed $K+1$ times.

Adding all of these up, we see that at most

$$3 + 1 + 7K + (K + 1) = 8K + 5$$

basic operations are executed. From our discussion earlier on the number `while`-loop iterations executed, we see that $K \leq \log_2 n$. Therefore, the running time of the `binarySearch` function is at most $8 \log_2 n + 5$. Since our calculations were rather rough, we will drop the constant coefficient, i.e., 8, as well as the lower order term, i.e., 5, to claim that the running time of the `binarySearch` function is $\log_2 n$.

It is worth noting that the analysis above is *worst case analysis* because it uses the worst case value for K, i.e., $\log_2 n$. It is quite possible for K to be much smaller, e.g., if the element we are looking for is exactly in the middle of the array, it will take just one iteration of the `while`-loop to find it. We will typically do *worst case analysis* in this class.
