22C:21 Lecture Notes

Jan 20th, 2006

In Wednesday's class we mentioned *binary search*. Let me start by describing the main idea.

Binary search: idea The input is a sorted array and a key we are searching for. Since the array is known to be sorted, we can do much better than search by doing a linear scan of the array. Here is how. Go to the middle of the array and compare the given key with the element in the middle. Note that since an array permits random access, going to the middle or in fact going to any slot in the array takes constant time. The comparison can yield one of three possible results:

- The key equals the element in the middle. In this case, we have found the key and we can exit the function.
- The key is smaller than the element in the middle. In this case, it is guaranteed that if the key is present in the array, it will only be present among elements before the middle. So we just have to search the first half of the array.
- The key is larger than the element in the middle. This is similar to the above case, except that we have to search the second half of the array.

Binary search: implementation As one can see from the above description, the binary search algorithm keeps track of the portion of the array in which the key has the potential to be found. Let us use two integer variables, `first` and `last` to keep track of the two ends of this portion of the array. So in each step we compare key with the middle element of `array[first..last]` and depending on the result of the comparison, we shrink this portion to about half its original size, by adjusting `first` or `last`. Here is the code.

```
public static boolean binarySearch(int[] list, int n, int key)
{
        int first = 0; int last = n-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;
}
```

Example. Consider the sorted array:

```
2 6 11 12 12 17 19 20 30
```

Suppose we are looking for the key 18. Then the successive values of `first`, `last`, and `mid` after each iteration are:

```
first           last            mid
0               8               4
5               8               6
5               5               5
6               5
```

Informally speaking, the reason why binary search is so fast as compared to linear search is this. In linear search, each comparison reduces the size of the "yet-to-be-searchead" array by 1. In contrast, each comparison in binary search reduces the size of the "yet-to-be-searched" array to half its previous size. So the array of the "yet-to-be-searched" array shrinks from n to $n/2$ to $n/4$ and so on and rapidly approaches 1.

We will show in subsequent classes that the running time of binary search is proportional to $\log_2 n$ and this quantity is very small compared to n, for large values of n. Before we do the running time analysis of binary search we will look at some simpler examples.

Running time analysis. The goal of *running time analysis* is to obtain a "pen and paper," machine independent estimate of how efficient an algorithm or a program or a data structure is as a function of the size of the input. The focus of running time analysis is not on how long a program or an algorithm takes for a particular input or input of a particular size. Instead, running time analysis focuses on trends — it tries to give a sense of how the running time of the algorithm or program grows as the input size increases.

Example 1.

```
for(i = 0; i < n; i++)
      sum = sum + i;
```

Here n is the input size. We want to estimate the running time of the above code as a function of n. The above code can be expanded to

```
1.  i = 0
2.  if i >= n then goto line 6
3.  sum = sum + i
4.  i++
5.  goto line 2
6.
```

Now note that on any machine, each of the above 5 lines of code will run in time that does not depend on n. So assume that on some hypothetical machine, line i takes c_i units of time, for $i = 1, 2, \ldots, 5$. Now note that Line 1 executes once, Line 2 executes $(n + 1)$ times, and Lines 3, 4, and 5 each execute n times. So the the total running time of the above code equals

$$c_1 + c_2 \cdot (n + 1) + c_3 \cdot n + c_4 \cdot n + c_5 \cdot n = n \cdot (c_2 + c_3 + c_4 + c_5) + (c_1 + c_2).$$

Note that this has the form $An + B$, where $A = c_2 + c_3 + c_4 + c_5$ and $B = c_1 + c_2$. Note that these are constants (i.e.,independent of n). Any function of the form $An + B$ is called a *linear function*. We say that this code fragment has linear running time. The constants A and B are machine dependent and we ignore them. Instead we focus on the fact that the running time grows linearly with respect to n.
