22C:21 Lecture Notes
Run-time analysis and the "Big Oh" notation

Aug 26, 2005

Example 4. Linear Search.

```
i = 0;
while((key != data[i]) && (i < n))
      i++;
```

Here n is the number of slots in the array data that we wish to search. It therefore represents the input size. The boolean condition and the increment statement each take constant time. Therefore the running time of the code fragment is linear (in n) in the worst case. Note that the running time is not always linear in n; *key* may be in the first slot all the time and in such cases the running time is just constant. Therefore, it is important to add the qualifier "in worst case."

Example 5. Binary Search.
Often we can assume that the array of items we have been asked to search is sorted in some way. We can take advantage of this assumption to come up with a much more efficient search.

The main idea is that we look for key in the middle of the array. If we find it there we are done. If key is less than the middle we only need to search the first half. If key is more than the middle we only need to search the second half.

```
first = 0;
last = n-1;
found = false;
while (first <= last) && (!found)
{
      mid = (first + last)/2;
      if (key == data[mid])
            found = true;
      if (key < data[mid])
            last = mid-1;
      if (key > data[mid])
            first = mid+1;
}
```

We will do a worst case analysis of the code. In other words, we will assume that *key* is not to be found and the while-loop terminates only when ($first > last$). The code fragment before the while-loop runs in constant time. Also, the code fragment inside the loop runs in constant time. Therefore the worst case running time of the code fragment is

$$A + B \cdot f(n),$$

where A and B are constants independent of n and $f(n)$ is the number of times the while-loop executes, in the worst case. The fact that this is a function of n is explicitly denoted by its form. To compute $f(n)$ consider the following table.

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

When does the size of the array to be examined become 1? When $n/2^i = 1$, that is when $2^i = n$, that is when $i = log_2(n)$. After $last - first + 1 = 1$, it takes just one more execution of the loop to get to $first > last$. So the worst case running time of the code fragment is logarithmic.

Logarithmic functions (review) If $a^b = x$, then $b = \log_a(x)$. So, as in the previous example, 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 20 elements!

"Big Oh" notation

Our run-time analysis aims to ignore machine-dependent aspects of the running time. For example, when we showed that the running time of a code fragment was $A \cdot n + B$, we don't care about the constants A or B because these depend on the machine. We simply focus on the fact that the shape of the function $A \cdot n + B$ is linear. In other words, we "approximate" $A \cdot n + B$ by n. The "Big Oh" notation permits a mathematically precise way of doing this.

Definition: Let $f(n)$ and $g(n)$ be functions defined on the set of natural numbers. A function $f(n)$ is said to be $O(g(n))$ if there exists positive constants c and n_0 such that for all $n >= n_0$, $f(n) <= c \cdot g(n)$.

Informally speaking, $f(n)$ is $O(g(n))$ if there is a multiple of $g(n)$ that eventually overtakes $f(n)$.

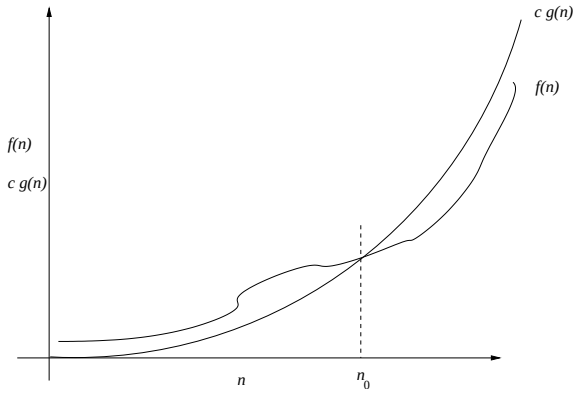

Figure 1: $f(n) = O(g(n))$ because there are positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Example 1. Show that $5n + 20 = O(n)$.
To see this, let $c = 6$. At what point does $6n$ overtake $5n + 20$? This happens at $6n = 5n + 20$. In other words, at $n = 20$. So for all $n >= 20$, $6n >= 5n + 20$.

Example 2. Let A and B be arbitrary constants, with $A > 0$. Show that $An + B = O(n)$.
Let $c = A + 1$. Then, we observe that $(A + 1) \cdot n >= An + B$, for all $n >= B$. This example is telling us that whenever the running time of an algorithm has the form An + B, we can simply say the running time is O(n).

Example 3. Show that $8n^2 + 10n + 25 = O(n^2)$.
As in the previous examples, let us select $c = 9$. We need to ask, when does $9n^2$ start overtaking $8n^2 + 10n + 25$?

$$\begin{aligned} 9n^2 &\geq 8n^2 + 10n + 25 \\ n^2 &\geq 10n + 25 \\ (n - 10) \cdot n &\geq 25 \end{aligned}$$

Now note that at $n = 12$, the left hand side (LHS) = 12 and the above inequality is not satisfied. However, at $n = 13$, the LHS = 39 and the inequality is satisfied. Furthermore, LHS is an increasing function of n and therefore the inequality continues to be satisfied for all larger n as well. In summary, we can set $c = 9$ and $n_0 = 13$.

Example 4. Show that $8n^2 + 10n + 25$ is not $O(n)$.
To obtain a contradiction suppose there are constants c and n_0 such that

$$8n^2 + 10n + 25 <= cn \text{ for all n } \geq n_0.$$

Clearly, c has to be larger than 10. So let us assume this. Then, the above inequality implies

$$8n^2 <= (c - 10)n - 25 \text{ for all n } \geq n_0.$$

Now pick $n = k(c - 10)$ where k is a natural number such that $k(c - 10) \geq n_0$. Then the $LHS = 8k^2(c - 10)^2$ and the $RHS = k(c - 10)^2 - 25$. Clearly, the LHS is larger than the RHS - a contradiction.