22C:21 Lecture Notes

Jan 23rd, 2006

Example 2.

```
for(i = 0; i < n; i++)
      for(j = 0; j < n; j++)
              System.out.println("Hello");
```

Here n is the input size. We want to estimate the running time of the above code as a function of n. Just as in Example 1, the above code can be expanded to

```
1.  i = 0
2.  if i >= n then goto line 10
3.  j = 0
4.  if j >= n then goto line 8
5.  Output "Hello"
6.  j++
7.  goto 4
8   i++
9   goto 2
10
```

Now note that on any machine, each of the above 9 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 $i = 1, 2, \ldots, 9$. Let us now figure out how many times each line executes. Clearly, Line 1 executes once. Line 2 executes once for each value of $i = 0, 1, 2, \ldots, n$, for a total of $(n + 1)$ times. For values of $i = 0, 1, \ldots, n - 1$, the condition in Line 2 evaluates to false and we just drop down to Line 3. Therefore, Line 3 executes n times, once for each value of $i = 0, 1, 2, \ldots, n - 1$. When $i = 0$, Line 4 is executed $(n + 1)$ times, once for each value of $j = 0, 1, \ldots, n$. Similarly, when $i = 1$, Line 4 is executed $(n + 1)$ times, once for each value of $j = 0, 1, \ldots, n$. Continuing in this manner we see that Line 4 is executed $(n + 1)$ times, for each value of $i = 0, 1, \ldots, n - 1$. Note that we don't get to Line 4 when $i = n$. Thus, Line 4 is executed $n(n+1)$ times. The calculation of how many times Line 5 is executed is very similar. For each value of i, Line 5 is executed n times, once for each value of $j = 0, 1, \ldots, n - 1$. This gives a total of n^2 times. Lines 6 and 7 are executed exactly as many times as Line 5, that is, n^2 times. Line 8 is outside the inner loop and is executed once for each value of $i = 0, 1, \ldots, n - 1$ for a total of n times. Similarly, Line 9 is exeecuted n times.

The following table summarizes our calculations.

Line 1	1 time
Line 2	$(n + 1)$ times
Line 3	n times
Line 4	$n(n + 1)$ times
Line 5	n^2 times
Line 6	n^2 times
Line 7	n^2 times
Line 8	n times
Line 9	n times

The total running time of Line i for $i = 1, 2, \ldots, 9$ is c_i times the number of times Line i is executed. The total running time of the code fragment is the sum of the total running time of each line. We get that the total running time of the code fragment is:

$$c_1 + c_2 \cdot (n + 1) + c_3 \cdot n + c_4 \cdot n(n + 1) + c_5 \cdot n^2 + c_6 \cdot n^2 + c_7 \cdot n^2 + c_8 \cdot n + c_9 \cdot n.$$

We separate out terms that contain n^2, terms that contain n, and terms that are just constants, to get the following expression:

$$n^2 \cdot (c_4 + c_5 + c_6 + c_7) + n \cdot (c_2 + c_3 + c_4 + c_8 + c_9) + (c_1 + c_2).$$

This can be written as $An^2 + Bn + C$, where A, B, and C are constants independent of n.

Any expression of the form $An^2 + Bn + C$, where A, B, and C are constants independent of n, is called a *quadratic function of* n. We say that the running time of the above code fragment is *quadratic in* n or just *quadratic*, if it is clear from the context what quantity the running time is a function of.

Quadratic functions grow faster than linear functions and will eventually catch up, no matter what the slope of the linear function is. As a result, for large enough n, a code fragment that runs in linear time will be faster than a code fragment that runs in quadratic time.
