Shortcuts in determining complexity

Finding the tightest complexity class

Thus far, in determining whether a function `f(n)` is in the set O(`g (n)`), we have asked for a proof. However, once you understand this stuff, in many cases you can find the smallest set O(`g(n)`) in which `f(n)` lies just by looking at `f(n)` and identifying the dominant term as `n` gets large. Here are examples.

Example 1. $f(n) = 20n^2 + 100(n \log n) + 1000n$. The dominant term is $20n^2$. Discarding the constant 20, we can see that the smallest class in which `f(n)` resides is $O(n^2)$.

Example 2. $f(n) = n(n–1)/2 + 1000n$. This expands to $n^2/2 – n /2 + 1000n$. The dominant term is $n^2/2$, so the smallest class in which `f(n)` resides is $O(n^2)$.

Example 3. $f(n) = n(n–1)/2 + 1000n^3$. This expands to $n^2/2 – n /2 + 1000n^3$. The dominant term is $1000n^3$, so the smallest class in which `f(n)` resides is $O(n^3)$.

Shortcutting the counting of basic steps

In determining the complexity class in which some algorithm lies, we have asked you to count basic steps in a careful manner. Once you understand how this works, there is no need to come up with an exact number of basic steps —as long as you arrive at the correct complexity class. We give some examples.

Example 1. In the algorithm to the right, we see that the loop iterates `n` times. We also see that the repetend is a basic step, as is every other statement or expression. Therefore, we say without further ado that this algorithm, takes times O(`n`). There is no need to count the basic steps.

```
// Store in s the sum of 1..n.
int s= 0;
for (int k= 1; k <= n; k++)
    s= s + k;
```

Example 2. To the right is insertion sort. To simplify, write `n = b.length`. We see that all the statements (except the two while-loops) are basic statements. We also see that execution of the outer loop requires `n` iterations. In fact, we can look only at the number of swaps performed in order to determine the time the method takes.

In the best case, the array is already sorted, so no swaps are executed at all —0 iterations of the inner loop are executed. Thus, in the best case, this algorithm takes time O(`n`).

In the worst case, the array is in descending order. Iteration 0 requires 0 swaps, iteration 1 requires 1 swap, …, iteration `n`–1 requires `n`–1 swaps. That's a total of

$$0 + 1 + 2 + \ldots + (n–1) \;=\; (n–1)n/2$$

swaps. Therefore, the worst-case time is $O(n^2)$.

```
k= 0;
// Invariant P: b[0..k-1] is sorted
while (k != b.length) {
   //Push b[k]down to its sorted
   //        position in b[0..k];
   j= k;
   while (0 < j  &&  b[j-1] > b[j]) {
       swap b[j-1] and b[j];
        j= j–1;
   }

   k= k+1;
}
```

What about the average or expected case? We can guess at this. On the average, each iteration will push `b[k]` half way down in `b[0..k]`. Therefore, the number of swaps is expected to be half of what it is in the worst case. That's still $O(n^2)$.

Example 3. The assignment `s= s + 'c';` takes time proportional to the length of `s`. The length increases by 1 with each iteration of the loop. Thus, in total, the time taken by that assignment is proportional to:

$$0 + 1 + 2 + \ldots + (n–1) = (n–1)n/2$$

so this algorithm takes time $O(n^2)$. It would be better to use the second algorithm, which takes time O(`n`) by first building an array of chars.

```
// Store n copies of 'c' in s
String s= "";
for (int k= 1; k <= n; k++)
    s= s + 'c ';
```

```
// Store n copies of 'c' in s
char[] ch= new char[n];
for (int k= 0; k < n; k++)
    ch[k]= 'c';
String s= new String(ch);
```