CSC165H, Mathematical expression and reasoning for computer science
week 11

4th August 2005

Gary Baumgartner and Danny Heap
heap@cs.toronto.edu
SF4306A
416-978-5899
http://www.cs.toronto.edu/~heap/165/S2005/index.shtml

Insertion sort example

Here is an intuitive,[1] sorting algorithm

```
// A is an array of comparable elements
// that will be rearranged (sorted) in non-decreasing order
IS(A)
 i = 1;
 while (i < A.length) {
         t = A[i];
         j = i;
         while (j>0 && A[j-1]>5) {
                 A[j] = A[j-1];
                 j = j-1;
         }
         A[j] = t;
         i = i+1;
}
```

Since we last computed running time, we got lazier. We could use the list from last week of the number of steps, and we'd find that there are between 3 and 11 steps for the lines in the program above. Since we are interested in big-O comparisons, that four-fold difference in steps will be absorbed into our multiplicative constants, so a better use of our time would be to count each line as one step.

Let's find an upper bound for $T_{IS}(n)$, the maximum number of steps to InsertionSort an array of size n. We'll use the proof format to prove and find the bound simultaneously — during the course of the proof we can fill in the necessary values for c and B.

Proof that $T_{IS}(n) \in O(n^2)$ (where $n = A.length$).
Let $c =??$. Let $B =???$.

> Then $c \in \mathbb{R}^+$ and $B \in \mathbb{N}$.
>
> Let $n \in \mathbb{N}$, and let A be an array of length n, and assume $n \geq B$.
>
> > So lines 5–7 execute at most n times, for n steps, plus 1 step for the last loop test.
> >
> > So lines 2–11 take no more than $n^2 + 5n + 1$ steps.
> >
> > So $n^2 + 5n + 1 \leq cn^2$ (fill in the values of c and B that makes this so — $c = B = 6$ should do).
>
> So $n \geq B \Rightarrow T_{IS}(n) \leq cn^2$.
>
> Since n is the length of an arbitrary array A and a natural number, $\mathbb{N}$, $\forall n \in \mathbb{N}, n \geq B \Rightarrow T_I S n \leq cn^2$ (so long as $B \geq 1$).

Since c is a positive real number and B is a natural number,
$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow T_{IS}(n) \leq cn^2$.
So $T_{IS} \in O(n)^2$. (by definition of $O(n^2)$).

Similarly, we prove a lower bound

$T_{IS} \in \Omega(n^2)$
Let $c =?$. Let $B =??$.

> Then $c \in \mathbb{R}^+$ and $B \in \mathbb{N}$.
>
> Let $n \in \mathbb{N}$, and let $A = [n-1, \ldots, 1, 0]$ (notice that this means $n \geq 1$). Assume $n \geq B$.
>
> > Note that at any point during the outside loop, $A[0..(i-1)]$ contains the same elements as before but sorted (i.e., no element from $A[(i+1)..(n-1)]$ has been examined yet). Since the value A[i] is less than all the values A[0..(i-1)], by construction of the array, the inner while loop makes i iterations, at a cost of 3 steps per iteration, plus 1 for the final loop check. This is strictly greater

than $2i+1$, so (since the outer loop varies from $i = 1..i = n-1$ and we have $n-1$ iterations of lines 3 and 4, plus one iteration of line 1), we have that $t_{IS}(n) \geq 1+3+5+\cdots+2n-1+2n+1 = n^2$ (the sum of the first n odd numbers.

So $n \geq B \Rightarrow T_{is}(n) \geq cn^2$ ($B = c = 1$ will do).

So there is some array A of size n such that $t_{IS}(A) \geq cn^2$.

Since n was an arbitrary natural number, $\forall n \in \mathbb{N}, n \geq B \Rightarrow T_{IS}(n) \geq cn^2$

Since $c \in \mathbb{R}^+$ and B is a natural number,
$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow T_{IS}(n) \geq cn^2$.
So $T_{IS} \in \Omega(n^2)$ (definition of $\Omega(n^2)$).

Floating-point systems

We can't represent every real number on a computer. We use "floating-point system" instead — given a fixed β, fixed number of digits t, and a range $[e_{\min}, e_{\max}]$ of exponents (integers), we can represent only numbers of the form:

$$\pm d_0 d_1 \ldots d_{t-1} \times \beta^e,$$

...where the $d_i \in [0, \beta - 1]$ are called the digits (and the sequence of digits is called the mantissa), and $e \in [e_{\max}, e_{\min}]$ is the exponent. (there's also a sign, costing at least a bit).

Here's an example. If $\beta = 10$, $t = 3$, $e_{\min} = -4$, and $e_{\max} = +4$, then you can represent $1/4$ as $+0.25 \times 10^0$ or $+2.5 \times 10^{-1}$. You can represent $1/3$ as $+3.33 \times 10^{-1}$. (Note that $+0.33 \times 10^0$ loses one digit of precision). Notice that there are multiple representations, so we agree on a NORMALIZED mantissa: require that the first digit $d_0 \neq 0$ unless we are representing 0 itself.

Using this normalized floating-point system:

- The smallest positive number is $+1.00 \times 10^{-4} = 0.0001$.
- The largest positive number is $+9.99 \times 10^4 = 99900$.

Another example. Suppose $\beta = 2$, $t = 3$, $e_{\min} = -2$, $e_{\max} = +3$. Numbers (other than 0) have the form

$$\pm 1.d_1 d_2 \times 2^e.$$

- Smallest positive number: $(1.00)_2 \times 2^{-2} = 1/4$.
- Largest positive number: $(1.11)_2 \times 2^3 = 14$.

Draw these out on a number line, and note that the larger numbers are spaced further apart, since a difference of 1 in the last digit represents a larger magnitude when the exponent is larger). For example, $(1.01)_2 \times 2^{-1} - (1.00)_2 \times 2^{-1} = 1/8$, versus $(1.01)_2 \times 2^2 - (1.00)_2 \times 2^2 = 1$. However, the percentage remains constant:

$$\frac{1/8}{2^{-1}} = 1/4 = \frac{1}{2^2}.$$

Rounding

Most numbers are not exactly representable in a floating-point system using a given base β. For example, when $\beta = 10$, you cannot represent $1/3$ exactly (no matter how large t is), so we used 3.33×10^{-1} when $t = 3$. What should we do with something like the base of natural logarithms, $e = 2.718281828\ldots$? Two approaches are used:

- Round to nearest: 2.72×10^0.
- Truncate to zero: 2.71×10^0.

Overflow

There is no way to represent a number larger than the largest floating-point number. In our first example, there is no way to represent 99901 or greater.

Underflow

There is no way to represent a positive number smaller than the smallest positive floating-point number. In our first example, there is no way to represent 0.00001 (or a smaller positive number).

Absolute rounding error

We can calculate the difference between the true value we're trying to represent and the value of its floating-point representation. For example, the absolute error in our representation of e is $|2.71 - 2.718281828\ldots| = |0.008281828\ldots|$.

Relative error

100 and 100.1 are "closer" than 1 and 1.1, even though the absolute difference is 0.1 in both cases. Look at the size of the error in terms of the size of the value being represented.

Relative error: For $x \neq 0$, the relative error between the approximate value x' and the "real" value x is

$$\frac{|x - x'|}{|x|}$$

For example, $|1.1 - 1|/|1.1| = 0.0909 \approx 9\%$. However, $|100.1 - 100|/|100| = 0.000999 \approx 0.09\%$.

Relative error in round-to-nearest

When we round numbers to represent them in a floating-point system, can we bound relative error for positive numbers with no overflow or underflow? (In fact, with overflow or underflow, the error can be arbitrarily large).

Notes

[1]but not particularly efficient...