CSCA48 Winter 2015

Week 9 - Worst Case Complexiy

Anna Bretscher

March 9, 2015

What is Complexity?

- A *measure* of how efficient an algorithm is.

Q. How should we *evaluate* the efficiency of an algorithm? should we code the algorithm in *Python* and *time* it for different values of *n*?

A. No.

- This is *machine* dependent.
- Why choose *Python*? why not *C* or *Java*?
- Implementation details can alter the timing.
- Want a method that allows us to compare different algorithms for large input sizes *without* a computer.

Worst-Case complexity

- For an *algorithm A*, let $t(x)$ be the number of steps A takes on input x.
- Then, the *worst-case time complexity* of A on input of size n is

$$T_{wc}(n) \stackrel{d}{=} \max_{|x|=n}\{t(x)\}$$

- **In words:** Look at all the inputs of size n and take the time of the one that is the *slowest*.

What is a step?

There are many conventions - we will use the following:

- **method call** 1 + steps to evaluate *each argument* + steps to *execute method*
- **return statement** 1 + steps to evaluate *return value*
- **if statement**, **while statement** (not the entire loop) 1 + steps to evaluate *exit condition*
- **assignment statement** 1 + steps to evaluate *each side*
- **arithmetic, comparison, boolean operators** 1 + steps to evaluate each *operand*
- **array access** 1 + steps to evaluate *index*
- **member access** 2 steps
- **constants, variables** 1 step

Often, we will just focus on operations or variable accesses.

Precondition. L is an array of integers.
Postcondition L sorted in non-decreasing order.

```
def insertion_sort (L):
  i = 1                               // 1:       steps
  while (i < len(L)):                 // 2:       steps
     t = L[i]                         // 3:       steps
     j = i                            // 4:       steps
     while (j > 0 and L[j-1] > t):    // 5:       steps
        L[j] = L[j-1]                 // 6:       steps
        j = j-1                       // 7:       steps
     L[j] = t                         // 8:       steps
     i = i+1                          // 9:       steps
```

Notation.

- $t_{IS}(L)$ is the *number of steps* or *time* for `insertion_sort` to run on a *specific input L*.
- $T_{IS}(n)$ is the *worst case* time for any input of *size n*.

Bounding $T_{IS}(n)$

Q. Why might we prefer $T_{IS}(n)$ to $t_{IS}(L)$?

A. It is much more general.

Q. Why might computing $T_{IS}(n)$ be *difficult*?

A. We have to consider all possible inputs of size n.

$\rightarrow$ We find *upper* and *lower bounds* for $T_{IS}(n)$.

Upper and Lower Bounds

Q. What do we mean by an *upper bound* for $T_{IS}(n)$?

A. The max number of steps that the code can make.

Q. What do we mean by a *lower bound* for $T_{IS}(n)$?

A. We want the maximum number of steps that an input will force.

$\rightarrow$ No *input* can take *more* steps than the *upper bound*.

$\rightarrow$ A *lower bound* cannot take *less* steps than any input.

FINDING AN UPPER BOUND

```
def insertion_sort (L):
  i = 1                                  // 1:  3    steps
  while (i < len(L)):                    // 2:  5    steps
     t = L[i]                            // 3:  5    steps
     j = i                               // 4:  3    steps
     while (j > 0 and L[j-1] > t):       // 5:  12   steps
        L[j] = L[j-1]                    // 6:  9    steps
        j = j-1                          // 7:  5    steps
     A[j] = t                            // 8:  5    steps
     i = i+1                             // 9:  5    steps
```

Q. At *most* how many *steps* do lines 5-7 execute?

A. At most len(L) times the number of steps, so n(9+5+12) + the loop test 12.

Q. At *most* how many *steps* do lines 2-9 take? line 1?

A. Loops at most n times, so $n(n26+12+23)+5+3=26n^2+35n+8$.

Big Oh - The Upper Bound

Idea. Want a *function* $g(n)$ such that for

- *BIG* enough n, i.e., $n > b$,
- $T(n) \leq c \cdot g(n)$
- Where the constant $c \in \mathbb{R}^+$ and $b \in \mathbb{N}$

Definition: Big Oh

- Let $g \in \mathcal{F}$. $\mathcal{O}(g)$ is the *set* of functions $f \in \mathcal{F}$ such that

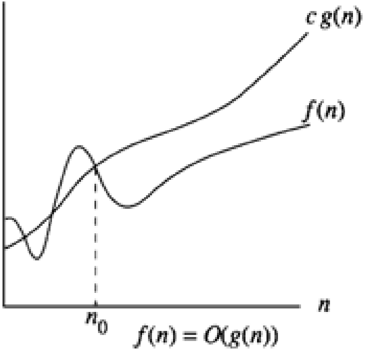
$$\exists c \in \mathbb{R}^+, \exists b \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq b \rightarrow f(n) \leq c \cdot g(n)$$

- Where $\mathcal{F}$ is the set of *functions*, $f : \mathbb{N}_k \rightarrow \mathbb{R}_0^+$.

Big Oh

- Let $g \in \mathcal{F}$. $\mathcal{O}(g)$ is the *set* of functions $f \in \mathcal{F}$ such that

$$\exists c \in \mathbb{R}^+, \exists b \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq b \rightarrow f(n) \leq c \cdot g(n)$$



INSERTION SORT

Recall that we discovered the number of steps `insertion_sort` does on a list of size n is at most

$$T_{IS}(n) \leq 26n^2 + 35n + 8 \forall n \geq 1$$

Q. For which *function* $g(n)$ does $T_{IS}(n) \in \mathcal{O}(g(n))$?

A. $g(n) = n^2$

Q. Why?

A. Notice that $26n^2 + 35n + 8 \leq 26n^2 + 35n^2 + 8n^2 = 69n^2$ for $n \geq 1$. Therefore, let $c = 69$ and $b = 1$.

Then $\exists c \in \mathbb{R}^+, \exists b \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq b \rightarrow T_{IS}(n) \leq cn^2$.