“Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.”

- Edsger Dijkstra

ASYMPTOTIC COMPLEXITY

Lecture

CS2110 – Summer 2019

"I didn't have time to write a short letter, so I wrote a long one instead."

- Mark Twin

ASYMPTOTIC COMPLEXITY

Lecture
CS2110 – Summer 2019

What Makes a Good Algorithm?

Suppose you have two possible algorithms that do the same thing; which is *better*?

What do we mean by *better*?

- Faster?
- Less space?
- Simpler?
- Easier to code?
- Easier to maintain?
- Required for homework?

FIRST, Aim for simplicity, ease of understanding, correctness.

SECOND, Worry about efficiency only when it is needed.

How do we measure speed of an algorithm?

Basic Step: one "constant time" operation

Constant time operation: its time doesn't depend on the size or length of anything. Always roughly the same. Time is bounded above by some number

Basic step:

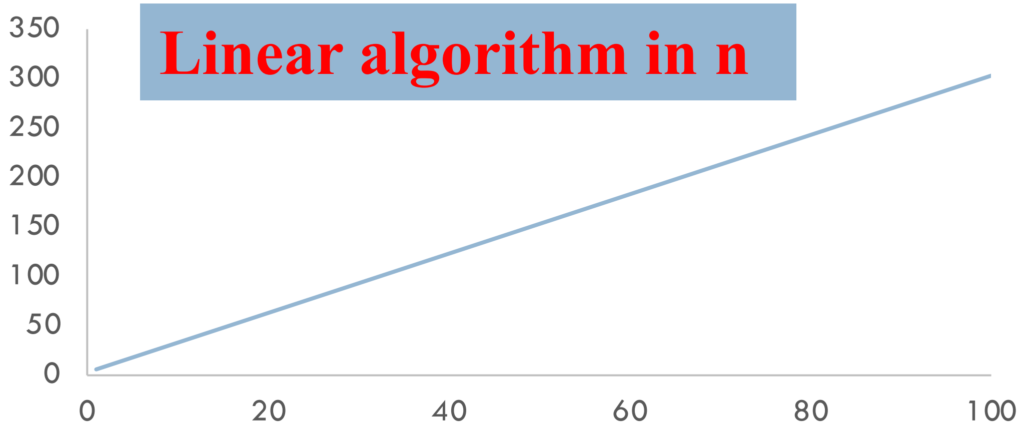
- Input/output of a number
- Access value of primitive-type variable, array element, or object field
- assign to variable, array element, or object field ***
- do one arithmetic or logical operation
- method call (not counting arg evaluation and execution of method body)

Counting Steps


```
// Store sum of 1..n in sum
sum= 0;
// inv: sum = sum of 1..(k-1)
for (int k= 1; k <= n; k= k+1){
    sum= sum + k;
}
```

Statement:	# times done
sum= 0;	1
k= 1;	1
k <= n	n+1
k= k+1;	n
sum= sum + k;	n
Total steps:	3n + 3

All basic steps take time 1.
There are n loop iterations.
Therefore, takes time proportional to n.



Not all operations are basic steps

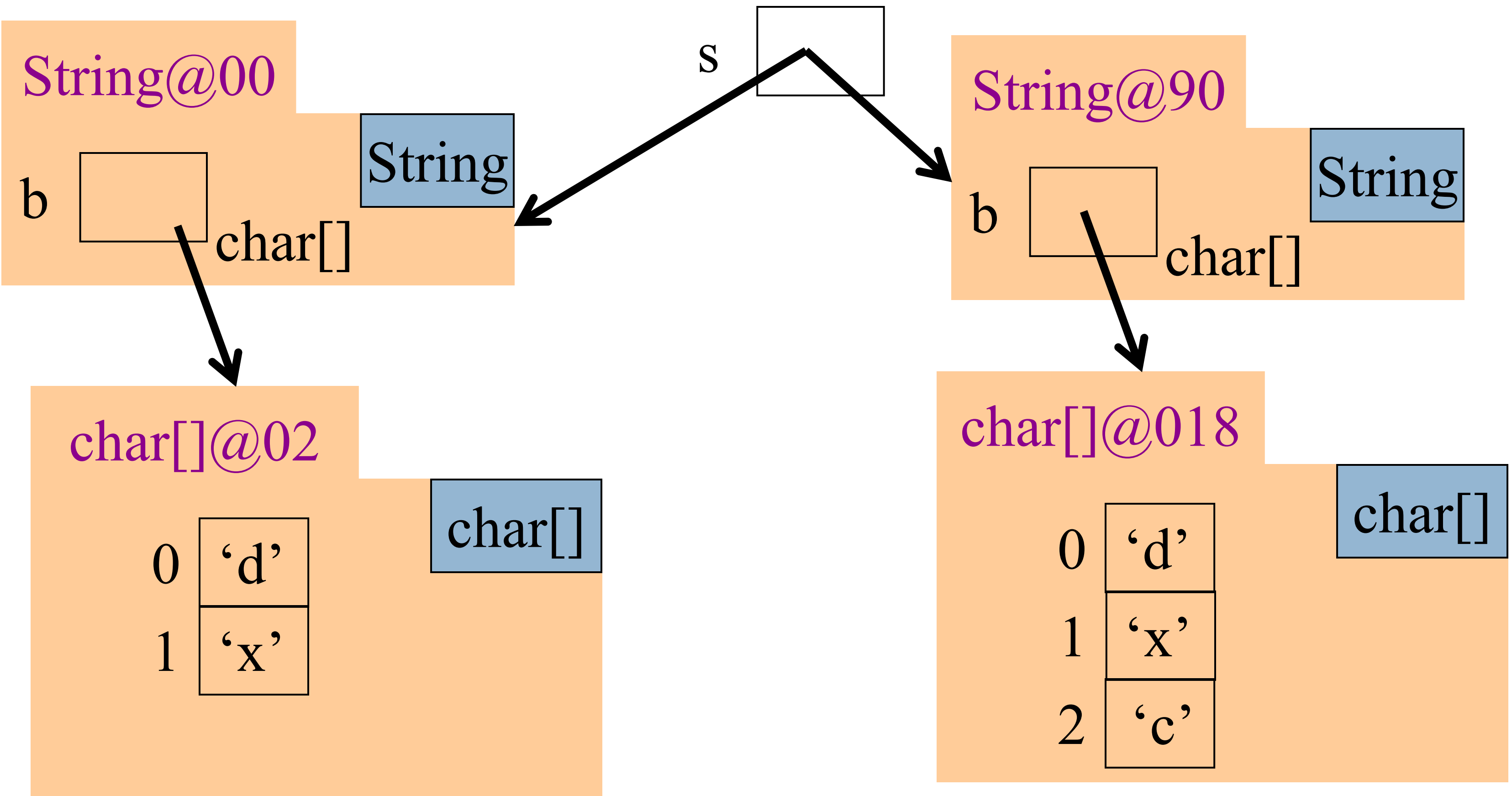

```
// Store n copies of 'c' in s
s= "";
// inv: s contains k-1 copies of 'c'
for (int k= 1; k <= n; k= k+1){
    s=  s + 'c';
}
```

Statement:	# times done
s= "";	1
k= 1;	1
k <= n	n+1
k= k+1;	n
s= s + 'c';	n
Total steps:	3n + 3

Catenation is not a basic step. For each k, catenation creates and fills k array elements.

String Catenation

s= s + “c”;   is NOT constant time.
It takes time proportional to 1 + length of s



Basic steps executed in s= s + ‘c’;

s= s + 'c’; // Suppose length of s is k

1. Create new String object, say C basic steps.
2. Copy k chars from object s to the new object: k basic steps
3. Place char ‘c’ into the new object: 1 basic step.
4. Store pointer to new object into s: 1 basic step.

Total of (C+2) + k basic steps.

In the algorithm, s= s + ‘c’; is executed n times:

s= s + ‘c’; with length of s = 0

s= s + ‘c’; with length of s = 1

…

s= s + ‘c’; with length of s = n-1

Total of n*(C+2) + (0 + 1 + 2 + … n-1) basic steps

Basic steps executed in s= s + 'c';

s= s + 'c'; // Suppose length of s is k

In the algorithm, s= s + 'c'; is executed as follows:

s= s + 'c'; with length of s = 0

s= s + 'c'; with length of s = 1

…

s= s + 'c'; with length of s = n-1

Total of n*(C+2) + (0 + 1 + 2 + … n-1) basic steps

0 + 1 + 2 + … n-1 = n(n-1) / 2. Gauss figured this out in the 1700's

$= n^2/2 - n/2.$

mathcentral.uregina.ca/qq/database/qq.02.06/jo1.html

Basic steps executed in s= s + 'c';

s= s + 'c'; // Suppose length of s is k

In the algorithm, s= s + 'c'; is executed as follows:

s= s + 'c'; with length of s = 0

s= s + 'c'; with length of s = 1

…

s= s + 'c'; with length of s = n-1

Total of n*(C+2) + (0 + 1 + 2 + … n-1) basic steps

Total of n*(C+2) + $n^2/2 - n/2$ basic steps

Total of n*(C+2) + $\mathbf{n^2}/2 - n/2$ basic steps. Quadratic in n.

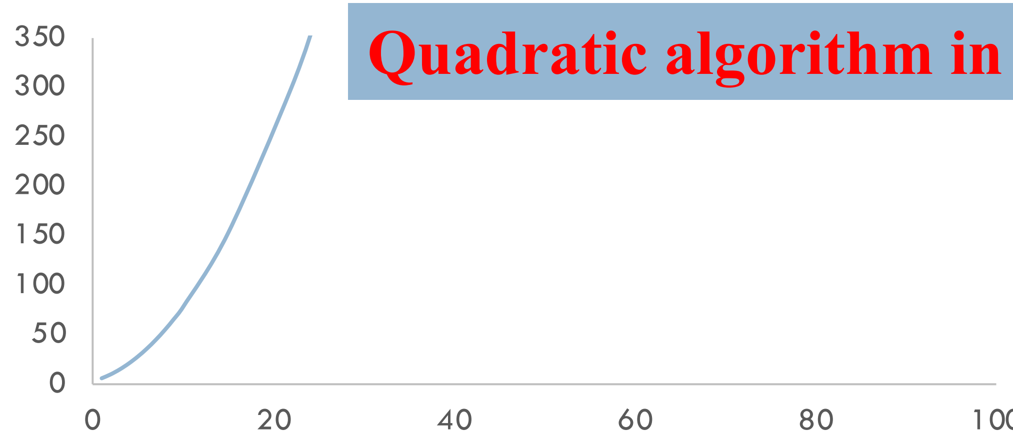
Not all operations are basic steps


```
// Store n copies of 'c' in s
s= "";
// inv: s contains k-1 copies of 'c'
for (int k= 1; k <= n; k= k+1){
    s=  s + 'c';
}
```

Statement:	# times	# steps
s= "";	1	1
k= 1;	1	1
k <= n	n+1	1
k= k+1;	n	1
s= s + 'c';	see to left	
Total steps:	…	

Total steps:
$2n + 3 +$
$n*(C+2) + \mathbf{n^2}/2 - n/2$

for s= s + 'c';



Quadratic algorithm in n

Linear versus quadractic


```
// Store sum of 1..n in sum
sum= 0;
// inv: sum = sum of 1..(k-1)
for (int k= 1; k <= n; k= k+1)
    sum= sum + n
```

Linear algorithm

```
// Store n copies of 'c' in s
s= "";
// inv: s contains k-1 copies of 'c'
for (int k= 1; k = n; k= k+1)
    s=  s + 'c';
```

Quadratic algorithm

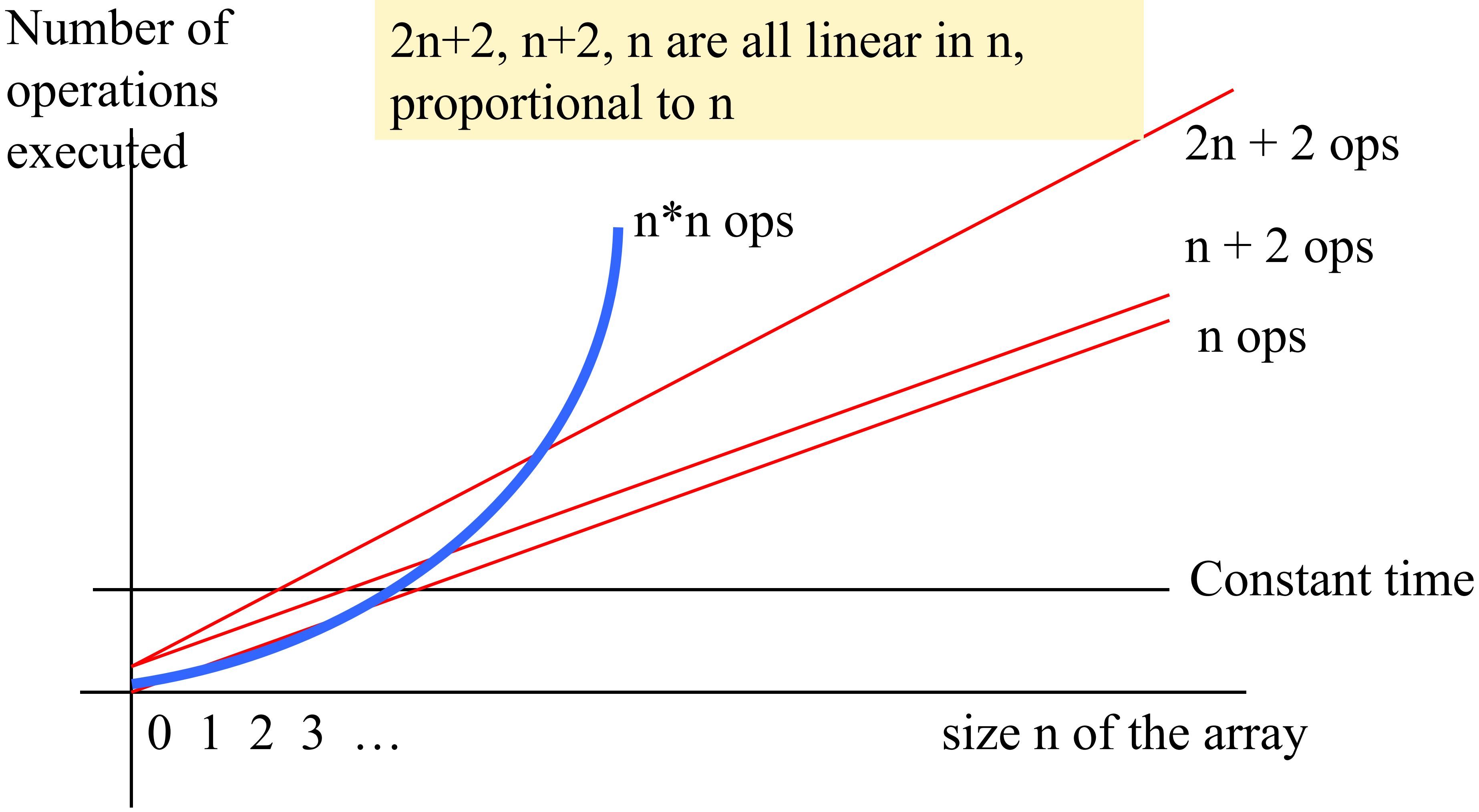
In comparing the runtimes of these algorithms, the exact number of basic steps is not important. What's important is that

One is linear in n —takes time proportional to n

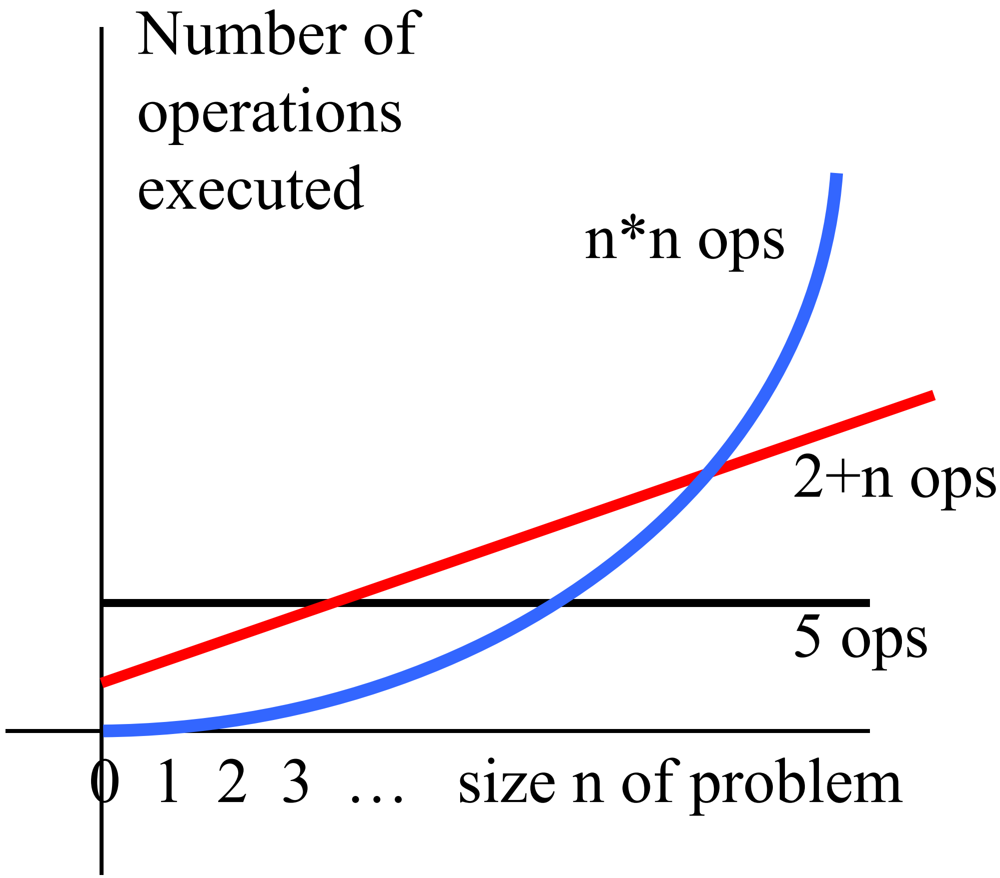
One is quadratic in n —takes time proportional to n^2

Looking at execution speed

2n+2, n+2, n are all linear in n, proportional to n



What do we want from a definition of “runtime complexity”?



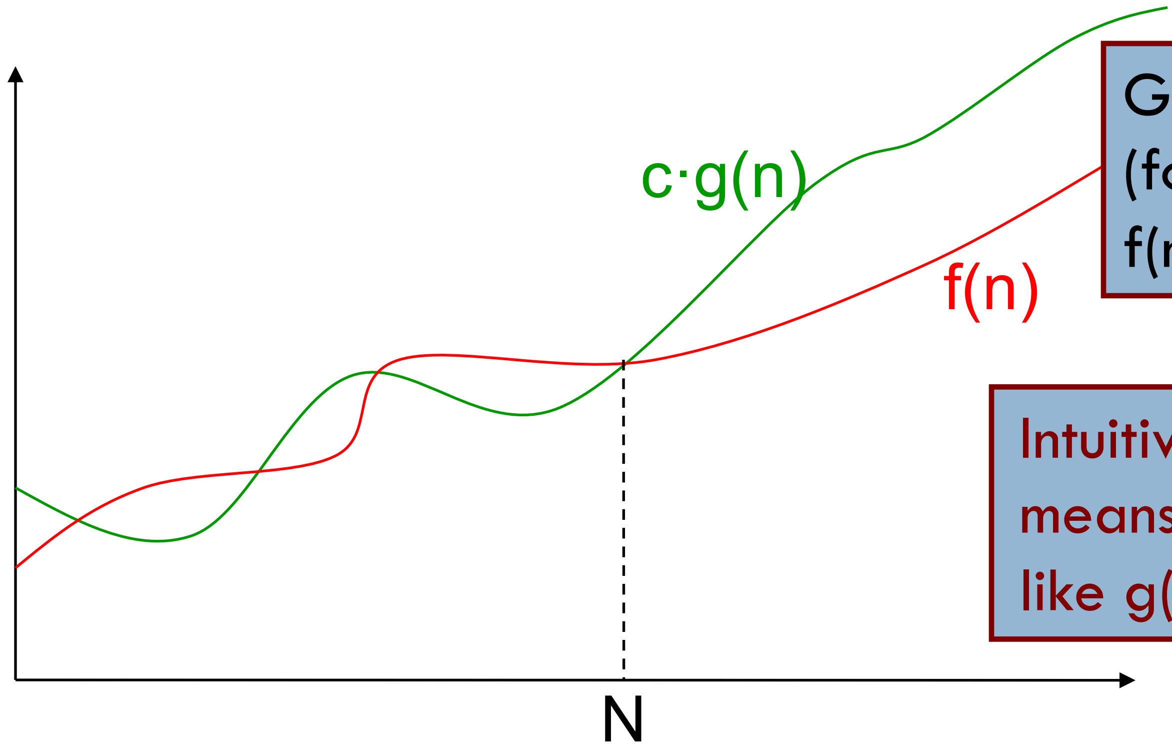
1. Distinguish among cases for large n, not small n

2. Distinguish among important cases, like
- n*n basic operations
- n basic operations
- log n basic operations
- 5 basic operations

3. Don’t distinguish among trivially different cases.
•5 or 50 operations
•n, n+2, or 4n operations

"Big O" Notation

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$



Get out far enough (for $n \geq N$) $f(n)$ is at most $c \cdot g(n)$

Intuitively, $f(n)$ is $O(g(n))$ means that $f(n)$ grows like $g(n)$ or slower

Prove that $(2n^2 + n)$ is $O(n^2)$

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Example: Prove that $(2n^2 + n)$ is $O(n^2)$

Methodology:

Start with $f(n)$ and slowly transform into $c \cdot g(n)$:

- Use = and <= and < steps
- At appropriate point, can choose N to help calculation
- At appropriate point, can choose c to help calculation

Prove that $(2n^2 + n)$ is $O(n^2)$

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Example: Prove that $(2n^2 + n)$ is $O(n^2)$

$f(n)$

= <definition of f(n)>

$2n^2 + n$

<= <for $n \geq 1$, $n \leq n^2$>

$2n^2 + n^2$

= <arith>

$3*n^2$

= <definition of $g(n) = n^2$>

$3*g(n)$

Transform $f(n)$ into $c \cdot g(n)$:

- Use =, <= , < steps
- Choose N to help calc.
- Choose c to help calc

Choose $N = 1$ and $c = 3$

Prove that 100 n + log n is O(n)

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

f(n)

= <put in what f(n) is>

100 n + log n

<= <We know log n ≤ n for n ≥ 1>

100 n + n

= <arith>

101 n

= <g(n) = n>

101 g(n)

Choose
N = 1 and c = 101

But what's origin of complexity?

- Computing a theory of all knowledge
- Some of my own thoughts