Object-oriented programming and data-structures

CS/ENGRD 2110 SUMMER 2018

Lecture 7: Complexity

http://courses.cs.cornell.edu/cs2110/2018su

Lecture 6 Recap

- Introduced the notion of recursion and backtracking recursion
- Discussed a number of problems that could be solved using recursions
- Hinted that recursion could be expensive.
  - What does expensive mean?

This lecture

- Formalise the notion of “expensive”
- Introduce Big-O notation
- Proofs of Big-O
- Applying Big-O to datastructures

What Makes a Good Algorithm?

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

Ex: is retrieving an element from **LinkedList** better than from **ArrayList?**

What Makes a Good Algorithm?

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

Ex: is retrieving an element from **LinkedList** better than from **ArrayList?**

What do we mean by *better*?

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

What Makes a Good Algorithm?

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

Ex: is retrieving an element from **LinkedList** better than from **ArrayList?**

What do we mean by *better*?

- Faster?
- Less space?
- 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.

What Makes a Good Algorithm?

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

Ex: is retrieving an element from **LinkedList** better than from **ArrayList?**

What do we mean by *better*?

- Faster?
- Less space?
- 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 consecutive integers in sum
sum= 0;
// inv: sum = sum of 1..(k-1)
for (int k= 1; k <= n; k= k+1){
    sum= sum + k;
}
```

All basic steps take time 1.

Counting Steps


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

All basic steps take time 1.

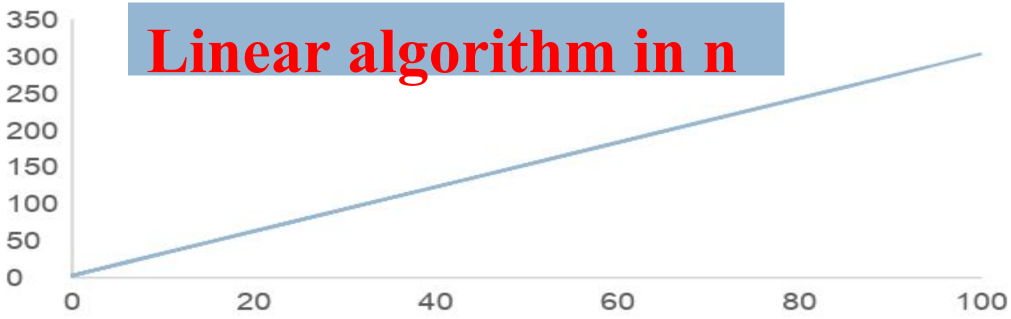
Statement:	# times done
sum= 0;	1
k= 1;	1
k <= n	n+1
k= k+1;	n
sum= sum + k;	n
Total steps:	3n + 3

Counting Steps


```
// Store sum of 1..n consecutive integers 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

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';	
Total steps:	3

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

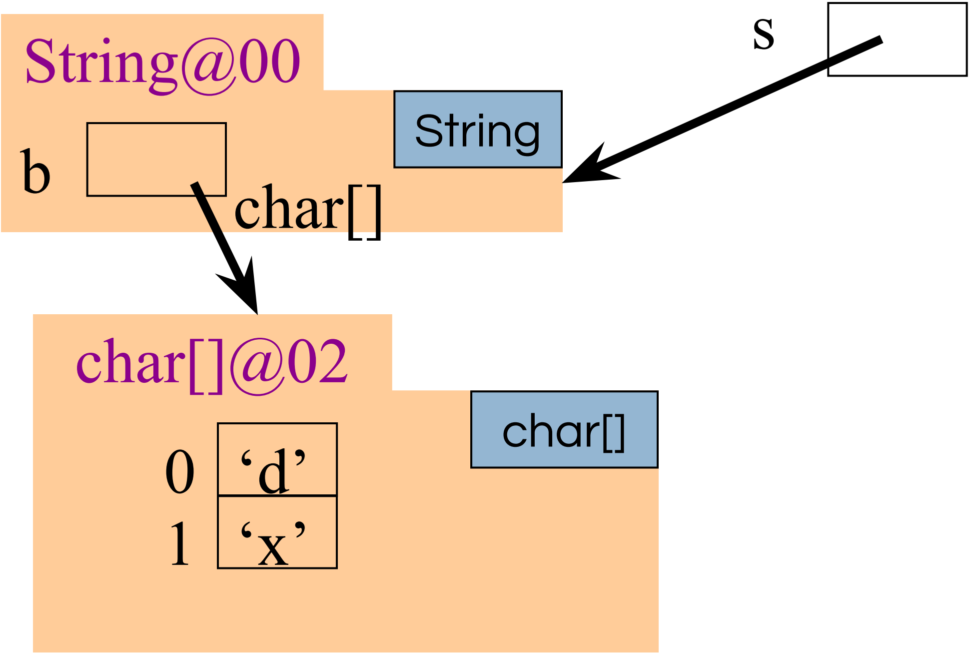
String Concatenation

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

s

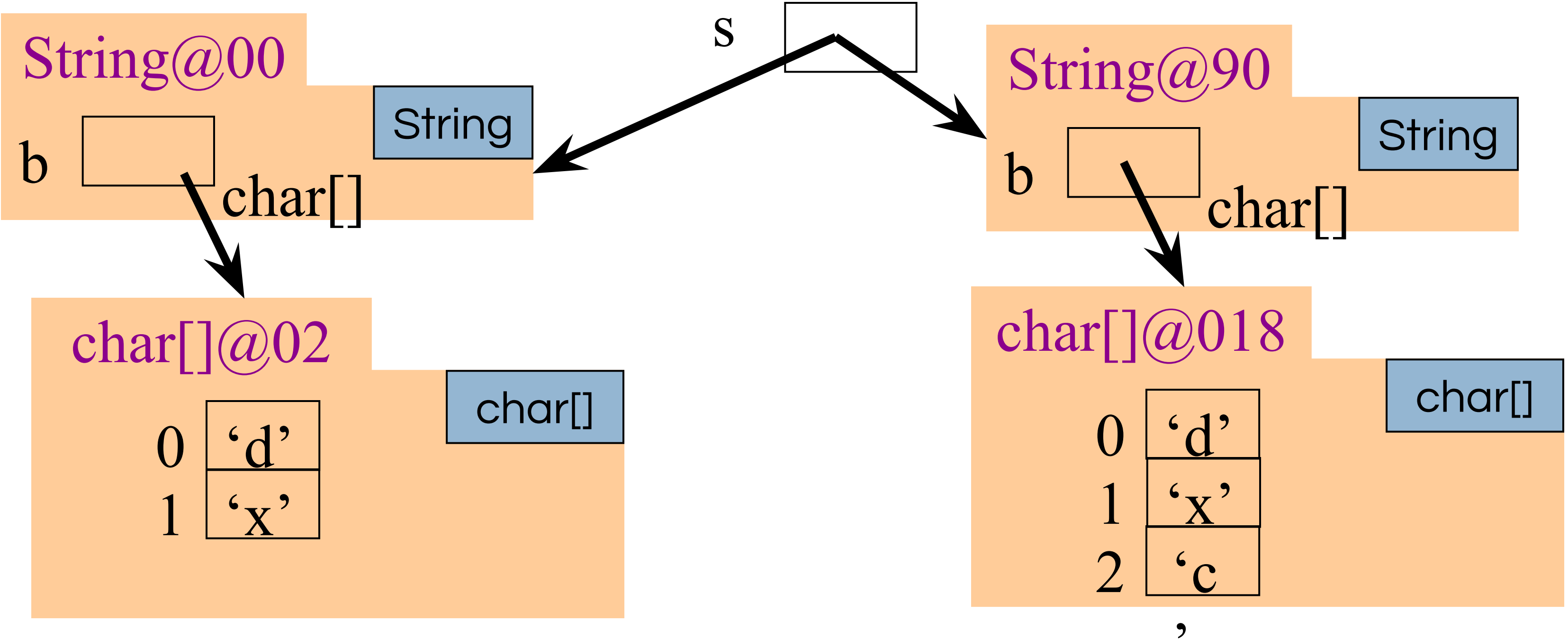
String Concatenation

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



String Concatenation

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



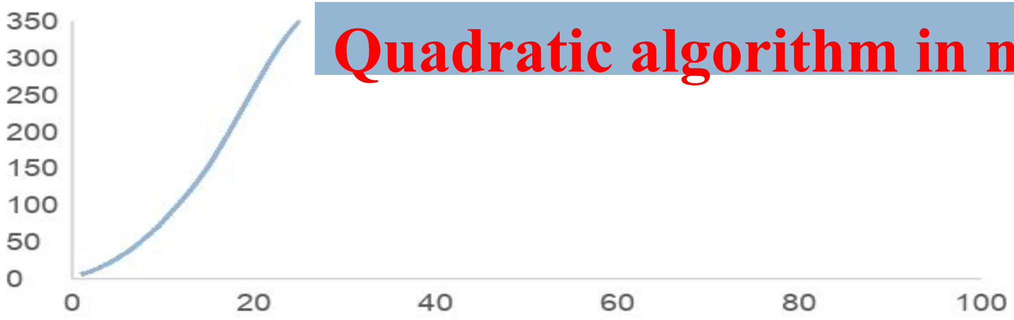
Not all operations are basic steps


```
// 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';	n	k

Total steps: n*(n+1)/2 + 2n + 3

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



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
```

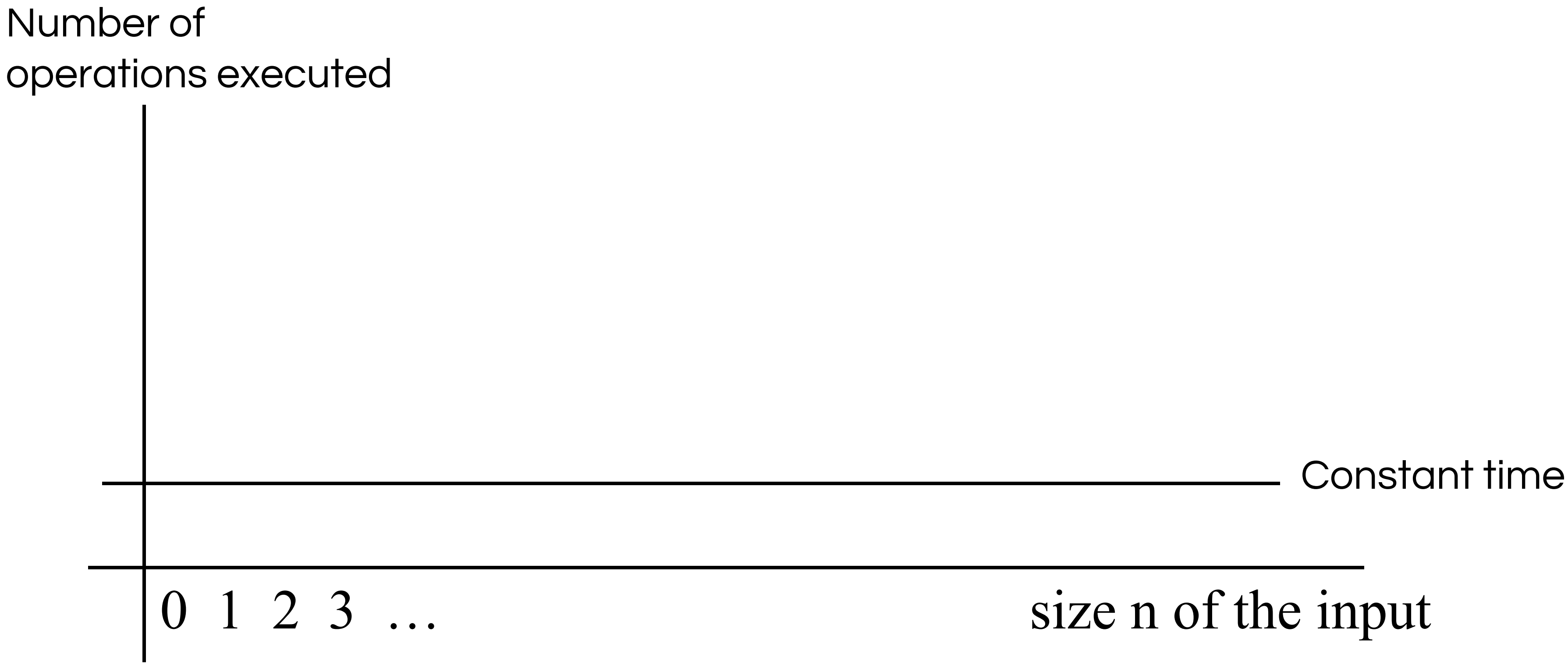
Linear algorithm

```
// 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’;
```

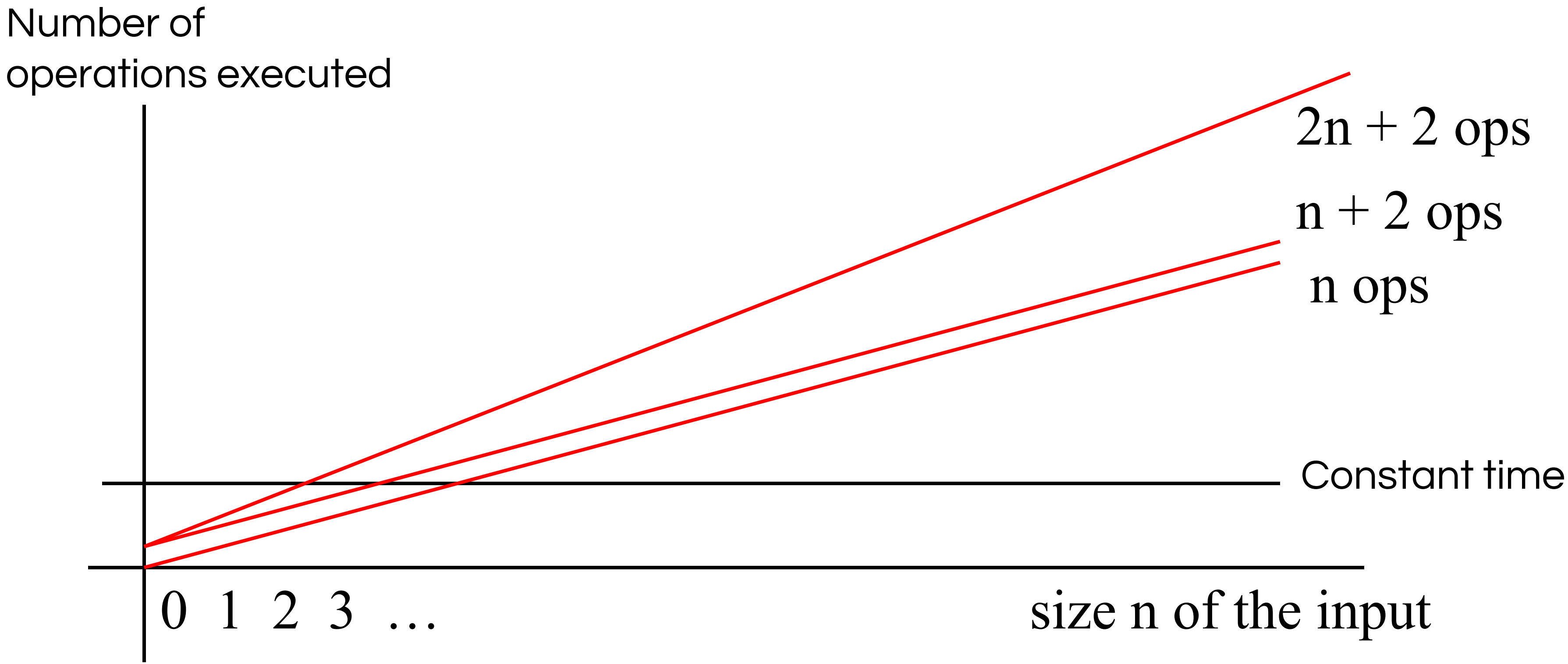
Quadratic algorithm

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

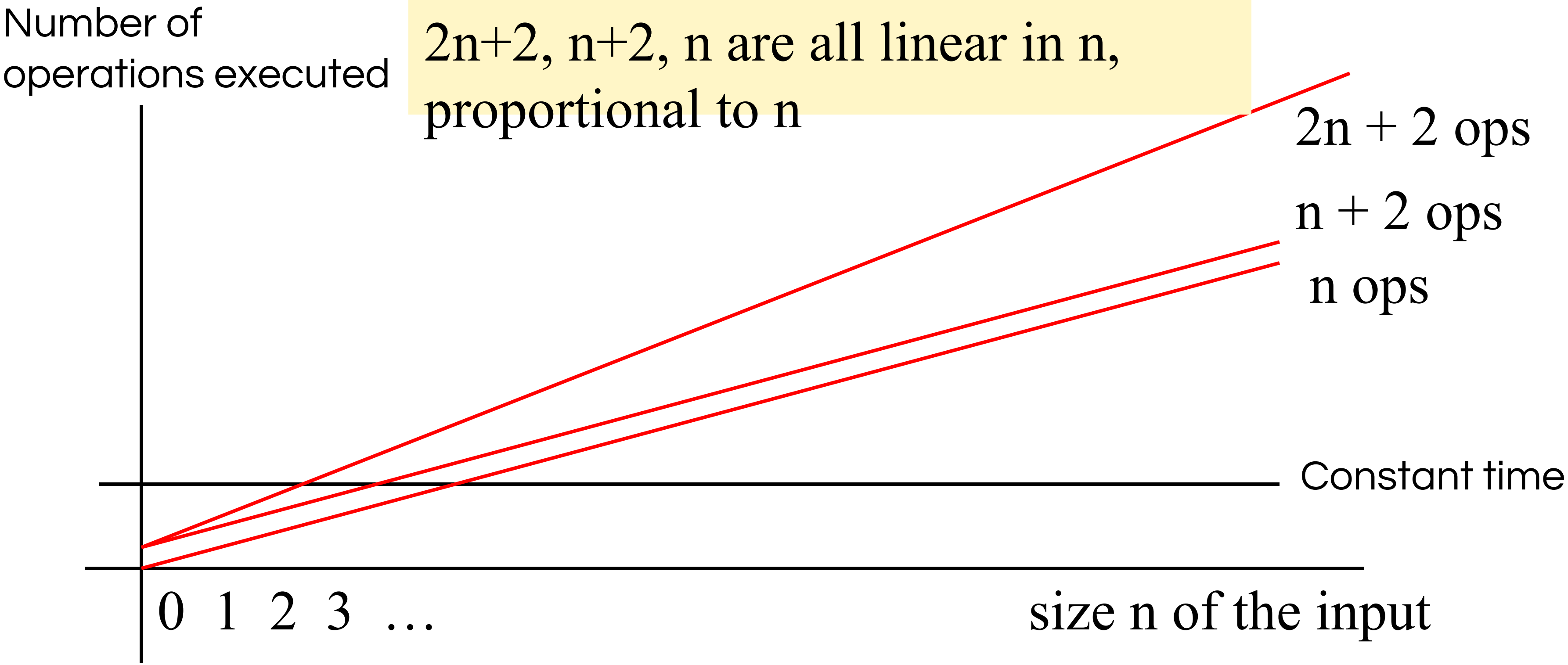


Looking at execution speed

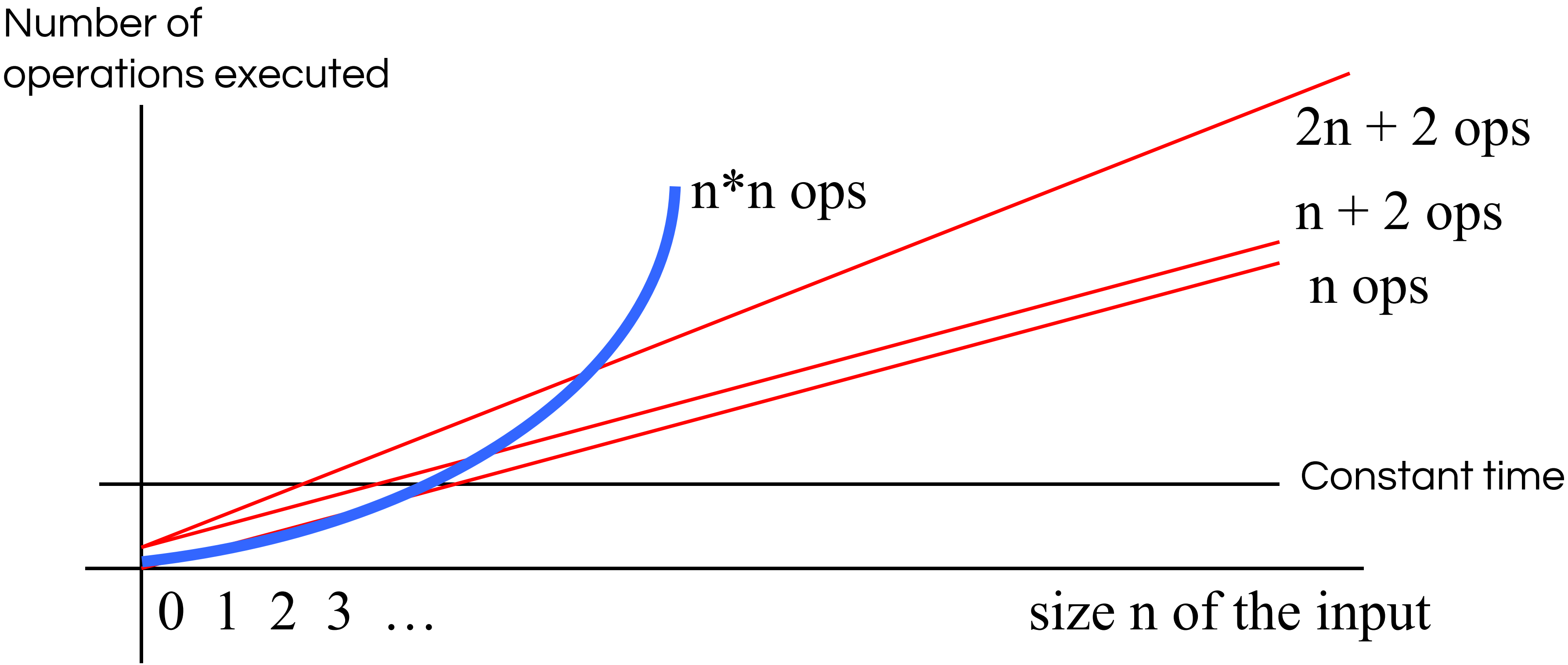


Looking at execution speed

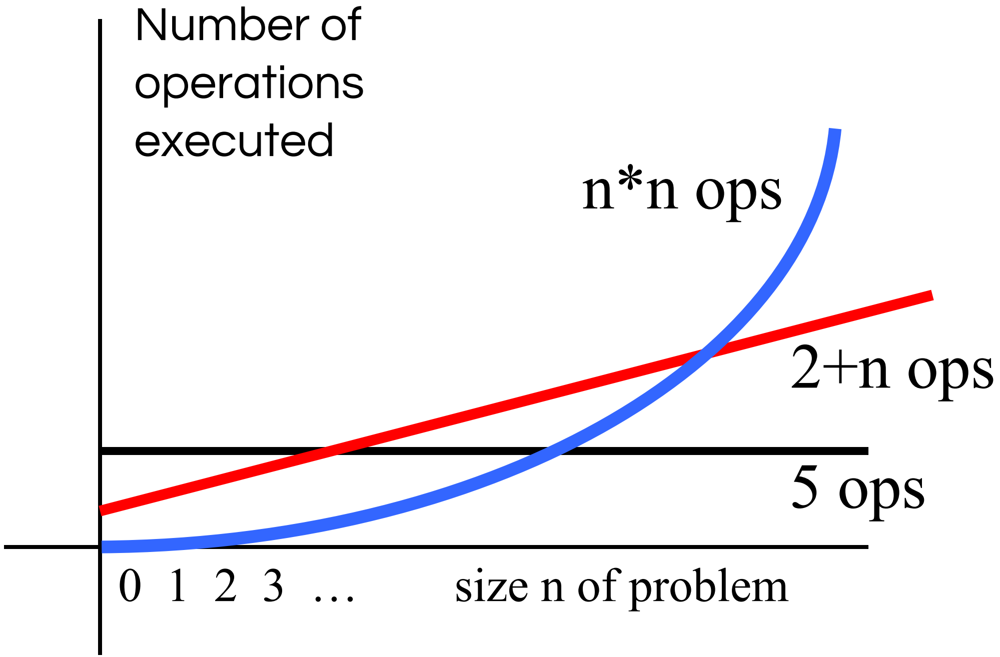
2n+2, n+2, n are all linear in n, proportional to n



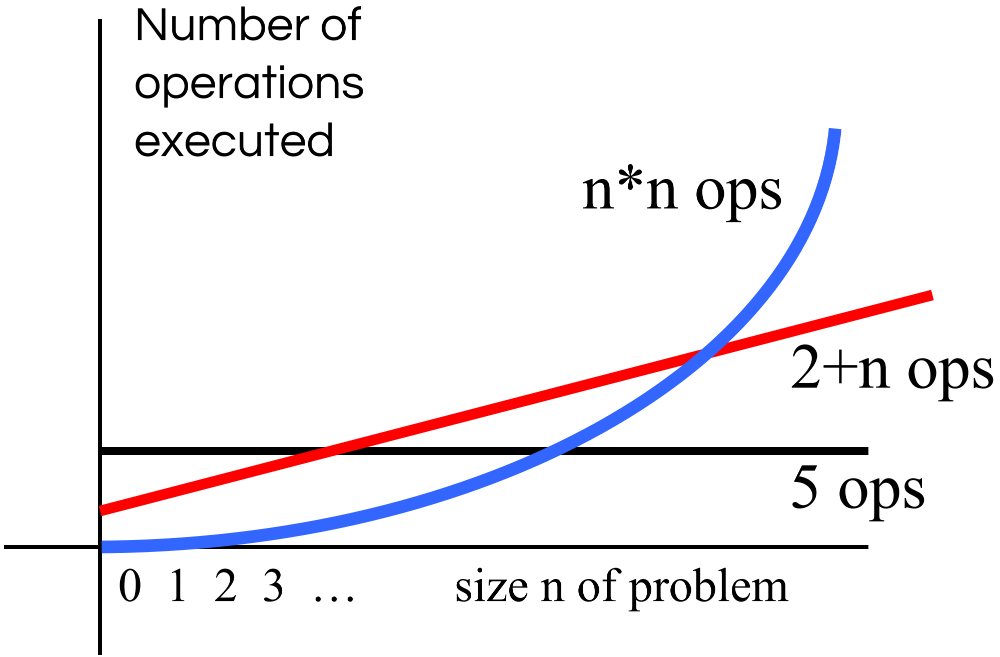
Looking at execution speed



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

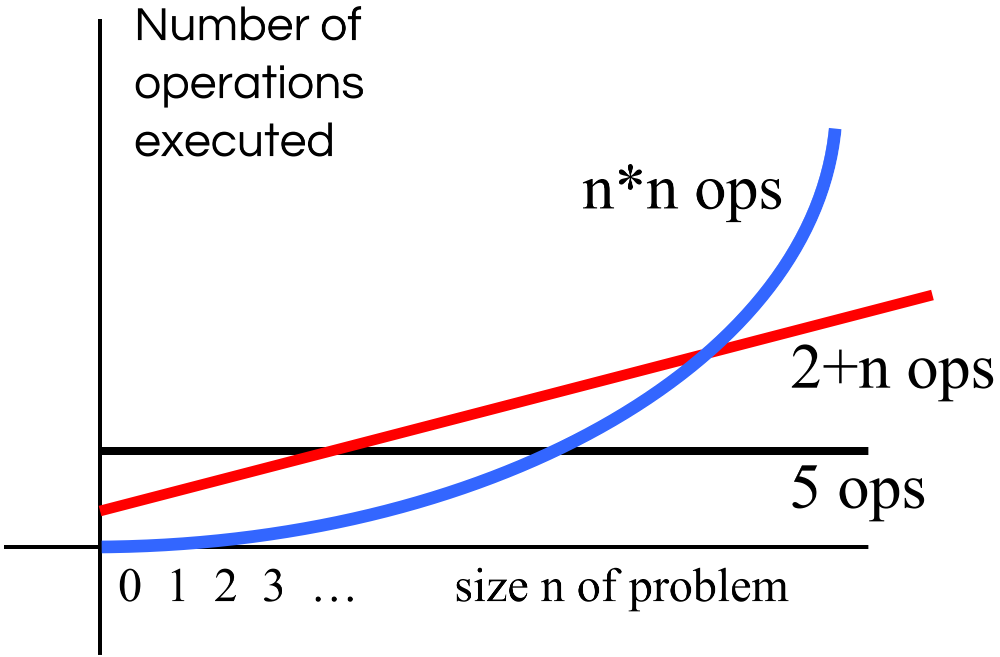


What do we want from a definition of "runtime complexity"?



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

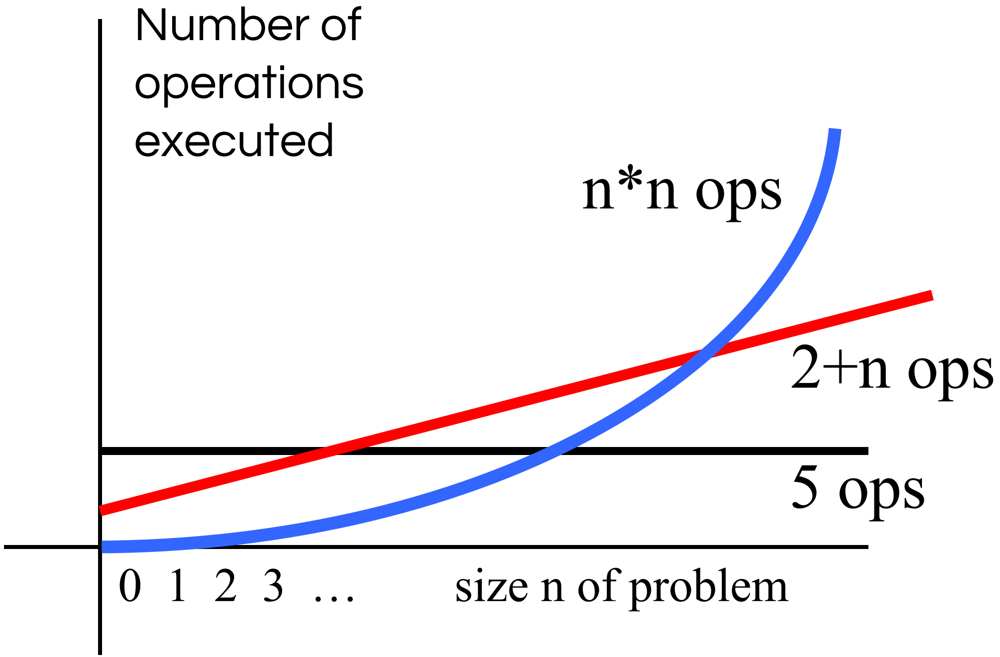
What do we want from a definition of “runtime complexity”?



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

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



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

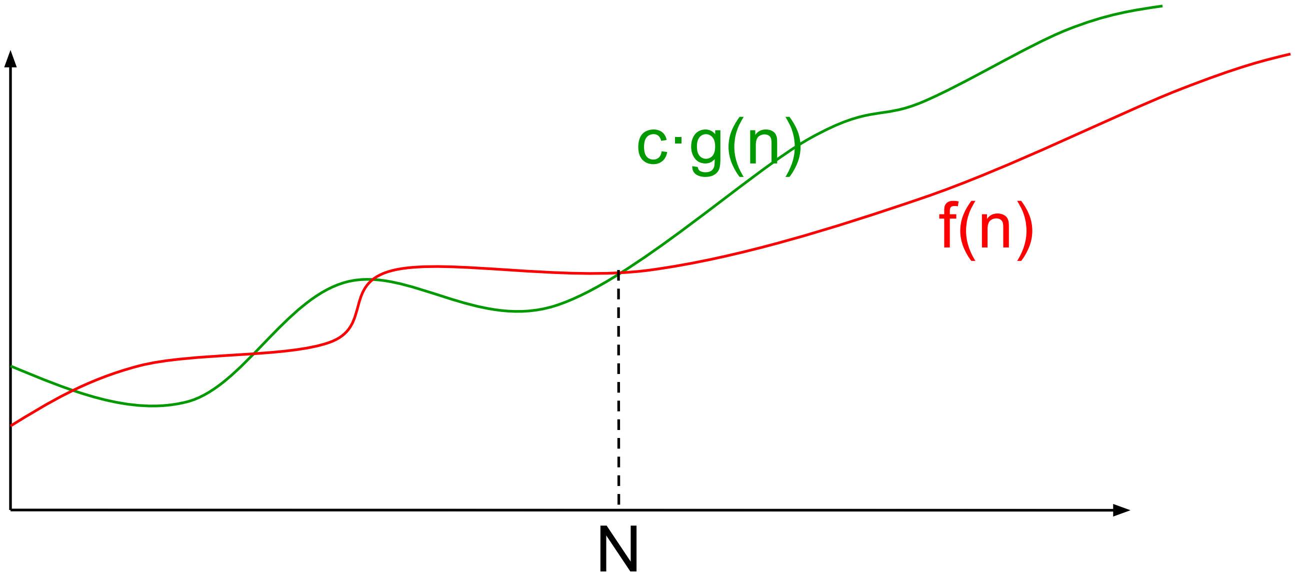
2. Distinguish among important cases, like
   - n*n basic operations
   - n basic operations
   - log n basic operations
   - 5 basic operations

3. Don’t distinguish among trivially different cases.

- 5 or 50 operations
- n, n+2, or 4n operations

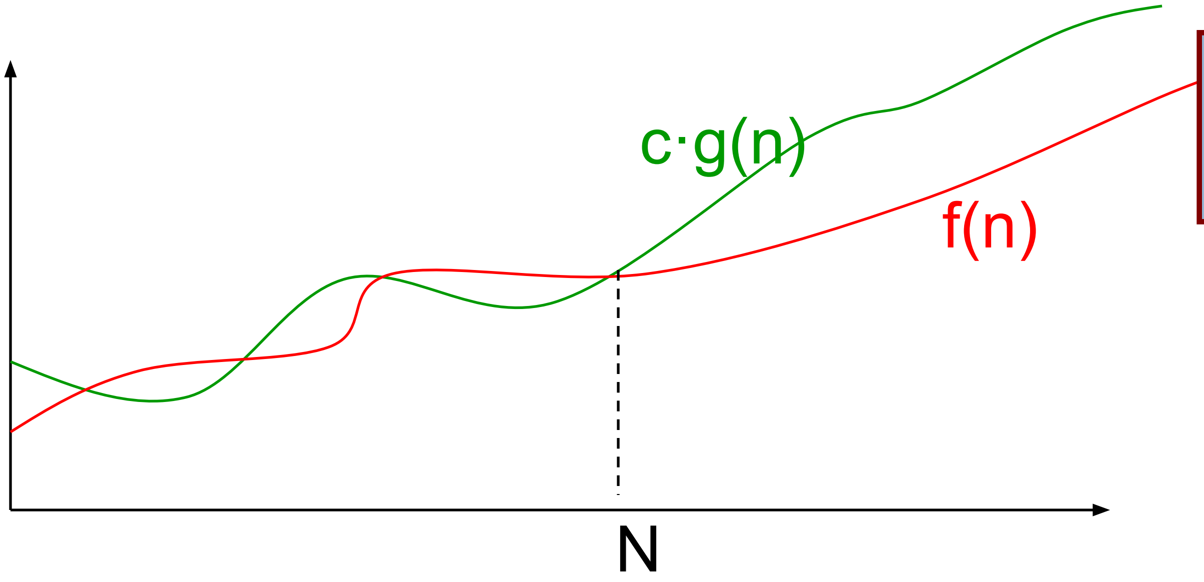
"Big O" Notation

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



"Big O" Notation

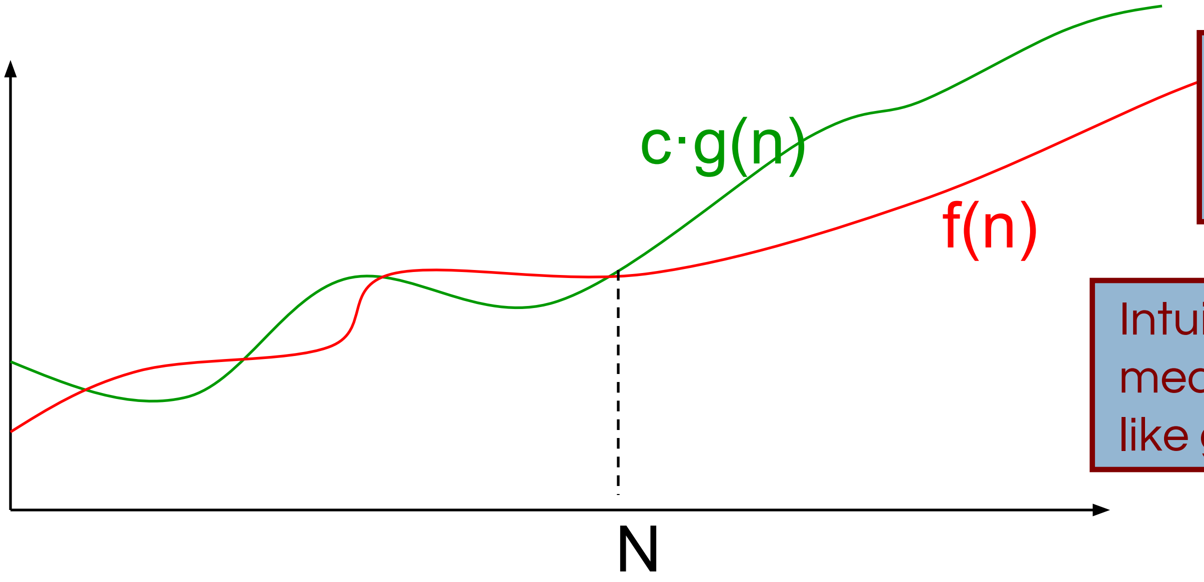
Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$



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

"Big O" Notation

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

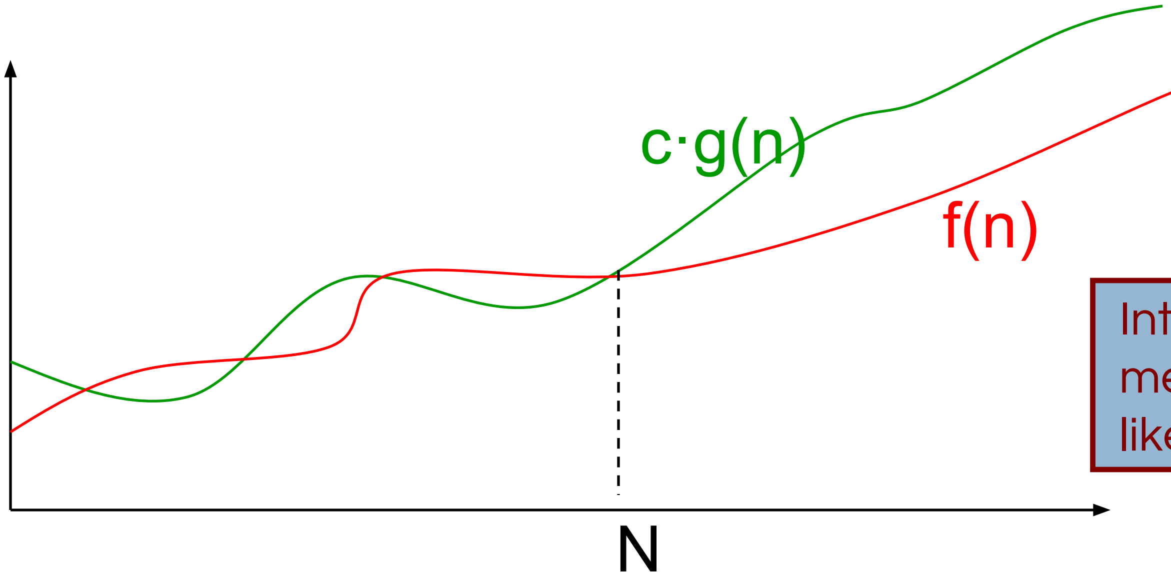


Get out far enough (for $n \ge N$)
f(n) is at most c·g(n)

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

"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 ≥ N)
f(n) is at most c·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)>

$2n^2 + n$

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

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

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 $n \geq 1$, $n \leq n^2$>

$2n^2 + n^2$

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

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

Choose

$N = 1$

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>

$3 * n^2$

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

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

Choose
$N = 1$

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

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

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

$f(n)$

= <definition of f(n)>

$2n^2 + n$

$<=$ <for $n \ge 1$, $n \le n^2$>

$2n^2 + n^2$

= <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

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

O(…) Examples

Let $f(n) = 3n^2 + 6n - 7$

- $f(n)$ is $O(n^2)$
- $f(n)$ is $O(n^3)$
- $f(n)$ is $O(n^4)$

$p(n) = 4\, n \log n + 34\, n - 89$

- $p(n)$ is $O(n \log n)$
- $p(n)$ is $O(n^2)$

$h(n) = 20 \cdot 2^n + 40n$

$h(n)$ is $O(2^n)$

$a(n) = 34$

- $a(n)$ is $O(1)$

O(...) Examples

Let $f(n) = 3n^2 + 6n - 7$

- $f(n)$ is $O(n^2)$
- $f(n)$ is $O(n^3)$
- $f(n)$ is $O(n^4)$

$p(n) = 4\ n \log n + 34\ n - 89$

- $p(n)$ is $O(n \log n)$
- $p(n)$ is $O(n^2)$

$h(n) = 20 \cdot 2^n + 40n$

$h(n)$ is $O(2^n)$

$a(n) = 34$

- $a(n)$ is $O(1)$

Only the *leading* term (the term that grows most rapidly) matters

If it's $O(n^2)$, it's also $O(n^3)$ etc! However, we always use the smallest one

Do NOT say or write f(n) = O(g(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) = O(g(n)) is simply WRONG. Mathematically, it is a disaster.
You see it sometimes, even in textbooks. Don't read such things.

Do NOT say or write $f(n) = O(g(n))$

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

$f(n) = O(g(n))$ is simply WRONG. Mathematically, it is a disaster.
You see it sometimes, even in textbooks. Don't read such things.

Here's an example to show what happens when we use = this way.

We know that $n+2$ is $O(n)$ and $n+3$ is $O(n)$. Suppose we use =

$$n+2 = O(n)$$
$$n+3 = O(n)$$

But then, by transitivity of equality, we have $n+2 = n+3$.
We have proved something that is false. Not good.

Problem-size examples

- Suppose a computer can execute 1000 operations per second; how large a problem can we solve?

operations	1 second	1 minute	1 hour
n	1000	60,000	3,600,000
$n \log n$	140	4893	200,000
n^2	31	244	1897
$3n^2$	18	144	1096
n^3	10	39	153
2^n	9	15	21

Big-O notation is not just for time

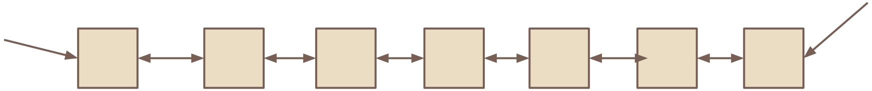
- Applies to both **time complexity** and **space complexity**
- Same reasoning in both cases
- In this class, we'll focus primarily on time complexity

A more formal look at datastructures

- Recall the two types of List in Java Collections (<List>)
  - ArrayList
  - LinkedList

- ArrayList is backed by an underlying array

- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element



Array Lists

- ArrayList is backed by an underlying array

- Arrays allow direct access to each element
  - What is the cost of accessing the ith element of the array?

O(1)

Array Lists

- ArrayList is backed by an underlying array

- Arrays allow direct access to each element
  - What is the cost of accessing the ith element of the array?
- What is the cost of inserting an element
  - May need to allocate a new array and copy all the previous elements into new array

Array Lists

- ArrayList is backed by an underlying array

- Arrays allow direct access to each element
  - What is the cost of accessing the ith element of the array?
- What is the cost of inserting an element
  - May need to allocate a new array and copy all the p nts into new array

Amortised
O(1)/O(n)

Array Lists

- ArrayList is backed by an underlying array

- Arrays allow direct access to each element
  - What is the cost of accessing the ith element of the array?
- What is the cost of inserting an element
  - May need to allocate a new array and copy all the previous elements into new array
- What is the cost of deleting the ith element
  - When delete an element, have to shift all the remaining elements to the left

Array Lists

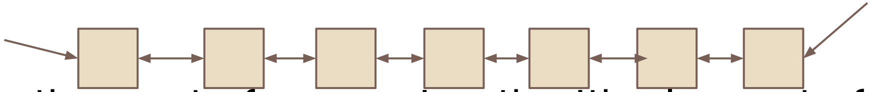
- ArrayList is backed by an underlying array

- Arrays allow direct access to each element
  - What is the cost of accessing the ith element of the array?
- What is the cost of inserting an element
  - May need to allocate a new array and copy all the previous elements into new array
- What is the cost of deleting the ith element
  - When delete an element, have to shift all the remaining ents to the left

O(n)

Linked Lists

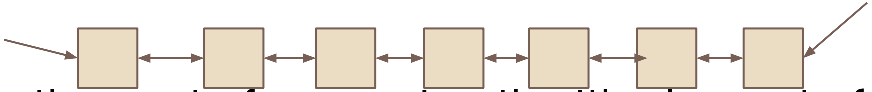
- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element



- What is the cost of accessing the ith element of the array?
- What is the cost of inserting an element to the head
- What is the cost of deleting the ith element

Linked Lists

- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element



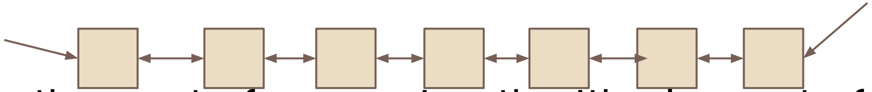
- What is the cost of accessing the ith element of the array?
  - Need to start from the head and follow pointers
- What is the cost of inserting an element to the head

O(n)

- What is the cost of deleting the ith element

Linked Lists

- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element



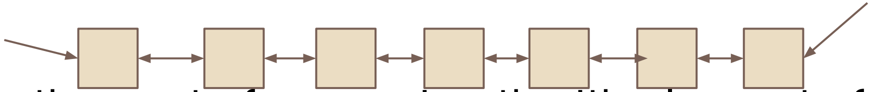
- What is the cost of accessing the ith element of the array?
  - Need to start from the head and follow pointers
- What is the cost of inserting an element to the head
  - Direct access through head pointer
- What is the cost of deleting the ith element

O(n)

O(1)

Linked Lists

- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element



- What is the cost of accessing the ith element of the array?
  - Need to start from the head and follow pointers
- What is the cost of inserting an element to the head
  - Direct access through head pointer
- What is the cost of deleting the ith element
  - Need to find the ith element first

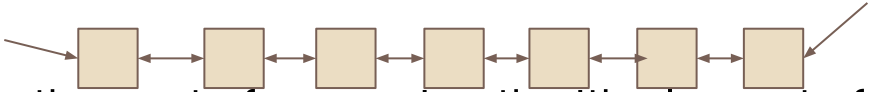
O(n)

O(1)

O(n)

Linked Lists

- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element



- What is the cost of accessing the ith element of the array?
  - Need to start from the head and follow pointers
- What is the cost of inserting an element to the head
  - Direct access through head pointer
- What is the cost of deleting the ith element
  - Need to find the ith element first

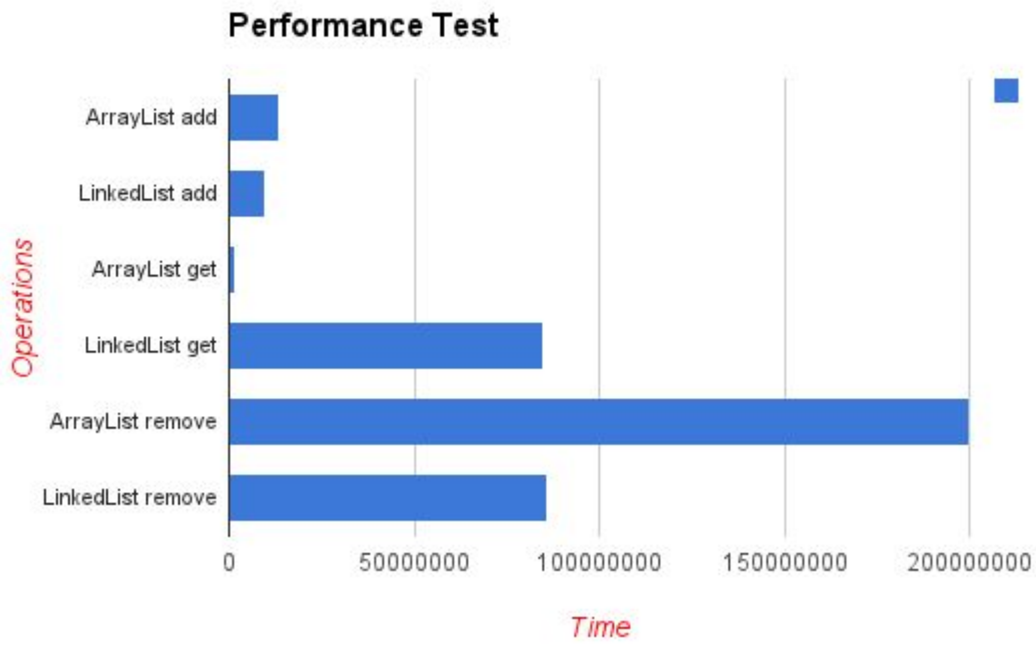
O(n)

O(1)

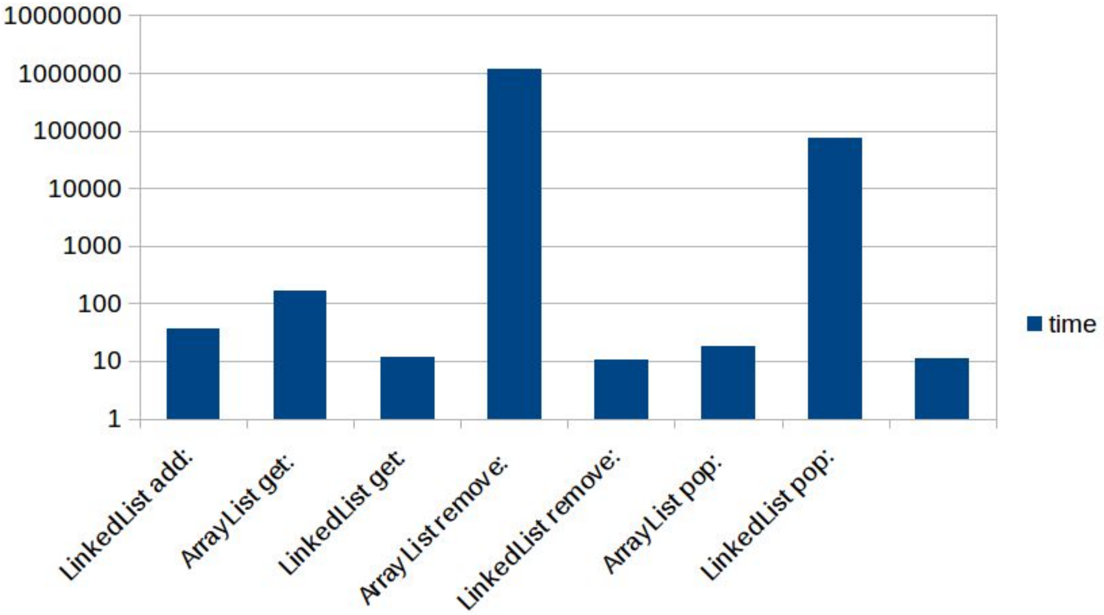
O(n)

What about deleting the head/tail element?

Do the performance numbers match up?



Only tell half the story …



- On my machine, ArrayList add is 5 times faster than LinkedList add
- Underlying reason is memory allocation is much more efficient for arrays than linked list: arrays can allocate large blocks of memory at once while you have to allocate individual nodes for a linked list