

Complexity

Cost

Final Code for search

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
           || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    //@loop_invariant !is_in(x, A, 0, i);
    {
      if (A[i] == x) return i;
    }
  return -1;
}
```

- **How long** does it take to run?
 - *with contract-checking off*

What do we mean by “How long”?

- First idea: wall-clock time
 - Time the code takes to run on a benchmark

The diagram illustrates the process of measuring wall-clock time for a benchmark. It features a terminal window with the following content:

```
# cc0 search.c0 search-time.c0
# time ./a.out -r 200 -n 1000000
0
real 0.83
user 0.82
sys 0.00
```

Callouts provide context for each part of the terminal output:

- Compiling with contracts disabled**: Points to the `cc0` command.
- The `search` function**: Points to the `search.c0` file.
- `main` that generates random tests and runs them**: Points to the `search-time.c0` file.
- Runs a.out 200 times on arrays of size 1000000**: Points to the `time ./a.out -r 200 -n 1000000` command.
- Unix utility that times execution**: Points to the `time` command.
- Execution took 0.82 seconds**: Points to the `user 0.82` output line, which is circled in red.

What do we mean by “How long”?

- Wall-clock time
 - Gives different results depending on
 - what else is running on computer
 - what specific computer it is running on
 - Is this a useful notion of “how long”?

```
Linux Terminal
# cc0 search.c0 search-time.c0
# time ./a.out -r 200 -n 1000000
0
real 0.83
user 0.82
sys 0.00
# time ./a.out -r 200 -n 1000000
0
real 0.91
user 0.90
sys 0.00
# time ./a.out -r 200 -n 1000000
0
real 0.81
user 0.80
sys 0.00
```

Different runs of the **same** code take **different times!**

What do we mean by “How long”?

- Wall-clock time

- Is this a useful notion of “how long”?

- Time is *about double* when we double the length of the array

- not exactly double though

```
Linux Terminal
# cc0 search.c0 search-time.c0
# time ./a.out -r 200 -n 1000000
0
real 0.83
user 0.82
sys 0.00
# time ./a.out -r 200 -n 2000000
0
real 1.62
user 1.61
sys 0.00
# time ./a.out -r 200 -n 4000000
0
real 3.17
user 3.15
sys 0.01
```

What do we mean by “How long”?

Can we do better than wall-clock time?

What are we looking for? A measure that is

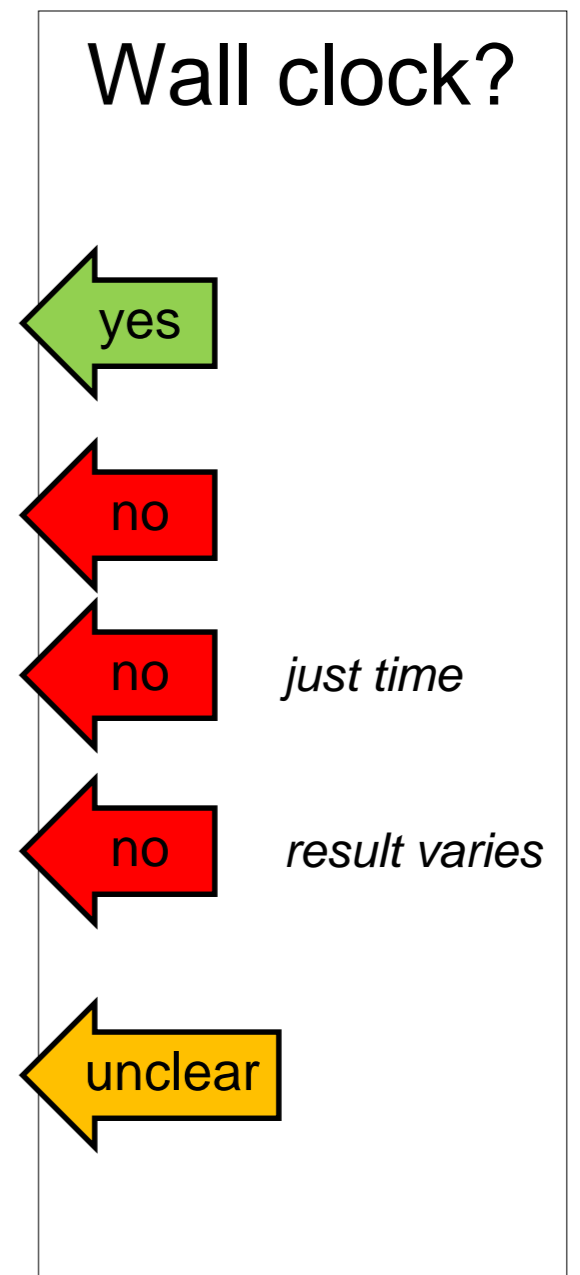
- **general**

- applicable to a large class of programs (and algorithms)
- independent of particular hardware
- applicable to many types of resources
 - time, space, energy, ...

- **mathematically rigorous**

- **useful**


- help us select among various algorithms for the same problem
 - e.g., POW vs. mystery function for exponentiation



What do we mean by “How long”?

- Second idea: count the number of execution steps
 - How many operations do we do to search an n -element array?

```
int search(int x, int[] A, int n) {  
    for (int i = 0; i < n; i++) {  
        if (A[i] == x) return i;  
    }  
    return -1;  
}
```



○ $i = 0$	1 step	} $3n + 2$ steps
○ loop	n times	
➤ $i < n$	1 step	
➤ $\text{if } (A[i] == x)$	1 step	
➤ $i++$	1 step	
○ $\text{return } -1$	1 step	

What do we mean by “How long”?

Step count

- $3n + 2$ steps to search an n -element array
- Always?
 - only if element is not found
- This is a **worst-case analysis**
 - Gives an **upper bound** on the number of steps

```
int search(int x, int[] A, int n) {  
    for (int i = 0; i < n; i++) {  
        if (A[i] == x) return i;  
    }  
    return -1;  
}
```

What do we mean by “How long”?

Step count

- $3n + 2$ steps to search an n -element array

- Depends only on n
 - value of x doesn't matter
 - contents of A doesn't matter
 - other than its length

```
int search(int x, int[] A, int n) {  
    for (int i = 0; i < n; i++) {  
        if (A[i] == x) return i;  
    }  
    return -1;  
}
```

- n is a **measure of the input** of the function
- Let's call the (upper bound on the) number of steps $T(n)$

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

What do we mean by “How long”?

Step count

- *What is a step?*

- is `i++` one step? 2 steps? 3 steps?
- what about `if (A[i] == x)` ?
- ... this gets complicated

- Each instruction takes a *constant* number of steps

- exact number is tricky to tell, but it's constant

- In the worst case, `search` makes

- a constant *b* number of steps outside the loop
- a constant *a* number of steps in each iteration of the loop

- So,

$$T(n) = an + b$$

```
int search(int x, int[] A, int n) {  
    for (int i = 0; i < n; i++) {  
        if (A[i] == x) return i;  
    }  
    return -1;  
}
```

- `i = 0`
- `return -1`
- `i < n`

OOPS!!!
loop guard
runs $n+1$
times!

- `i < n`
- `if (A[i] == x)`
- `i++`

What do we mean by “How long”?

- Step count tells us “how long” a function takes to run
 - Time (here, number of steps) is a type of **resource**
- Other resources of interest
 - Space: how much memory does the function use?
 - Energy: how much energy does running it consume?
 - Connectivity: how many network connection does it make?
 - ...

In this course, we will be mainly interested in execution time

- The amount of resources a function uses is called its **cost**
 - $T(n) = an + b$ is a **cost function**

Argument is
a **measure** of
the input

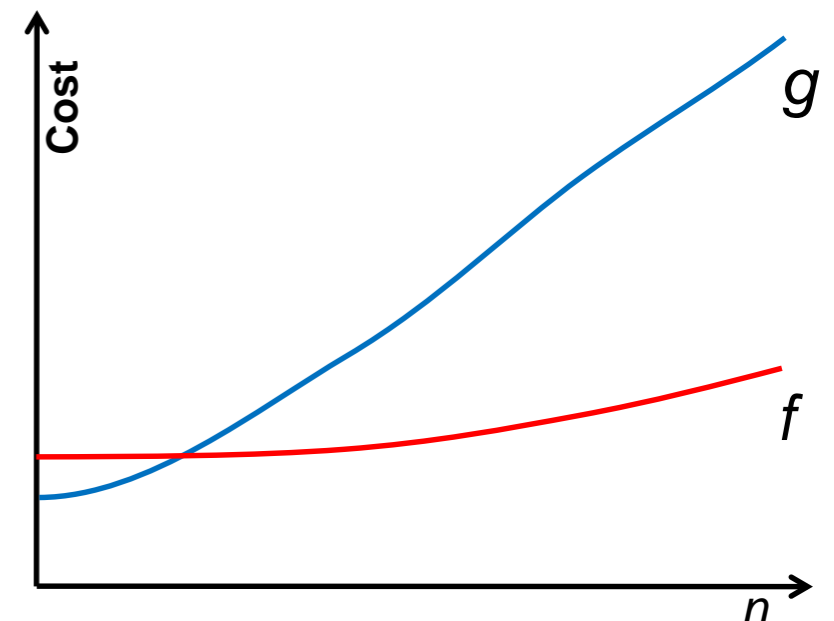
Computes the **cost**
of executing it on
an input of size n

Comparing Cost

Comparing Cost

- Given two $C0$ functions that solve the same problem
 - F has cost $f(n)$
 - G has cost $g(n)$we want to answer the question “is F better than G ?”
- We will do so by answering the question
“is f better than g ?”
 - **How do we define this?**
- f and g are functions that
 - take a natural number as input
 - return a natural number as output

a non-negative integer

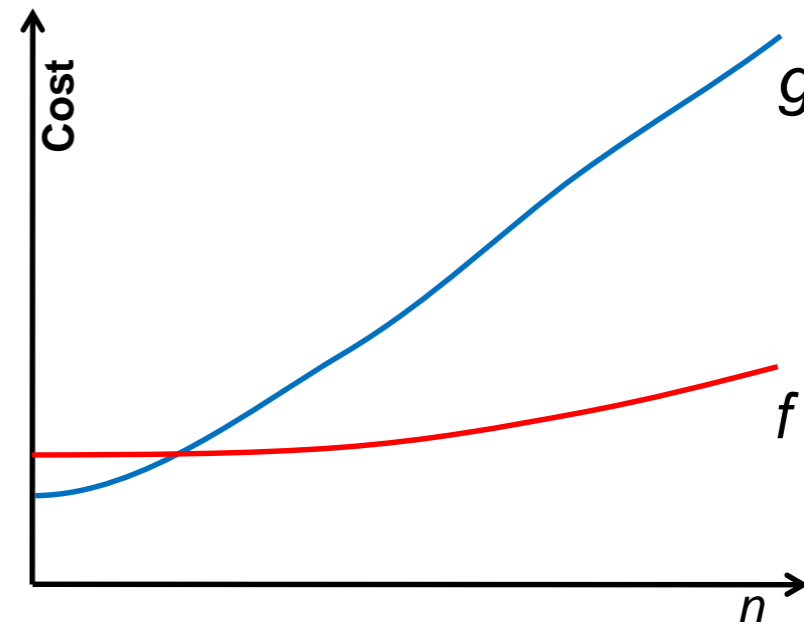
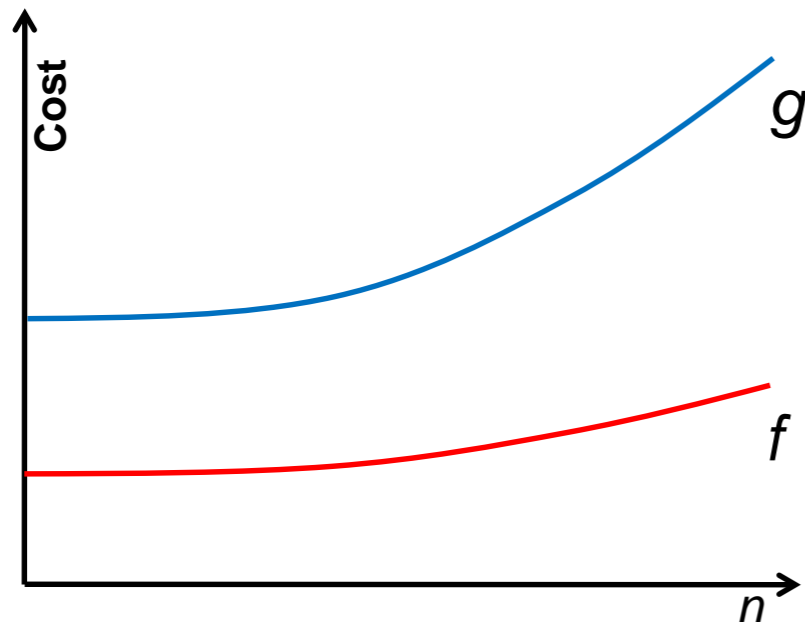


“Is f better than g ?”

- Attempt #1:

“ f is better than g ” if
for all n , $f(n) \leq g(n)$

It's Ok if $f(n) = g(n)$ for some (or all) n



“Is f better than g ?”

- Attempt #1:

“ f is better than g ” if

for all n , $f(n) \leq g(n)$

- But is this useful?

- f is initially worse than g

- but f is better beyond a certain point

- For small inputs, both costs are low

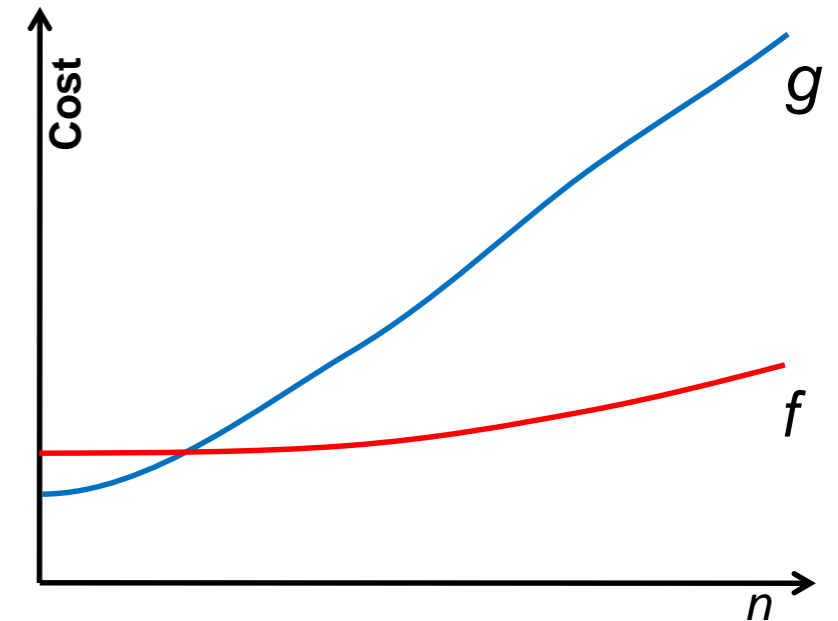
- 0.12 ms vs. 0.23 ms doesn't matter for most applications

- For large inputs, we want lower cost

- 1.35 ms vs. 200 years matters for all applications

- **Solution:** ignore small inputs

- **Asymptotic** notion of cost

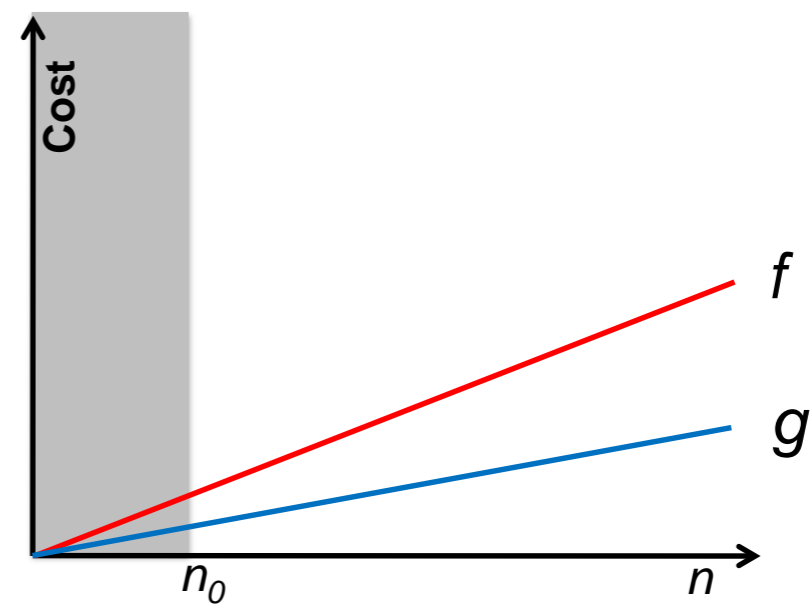
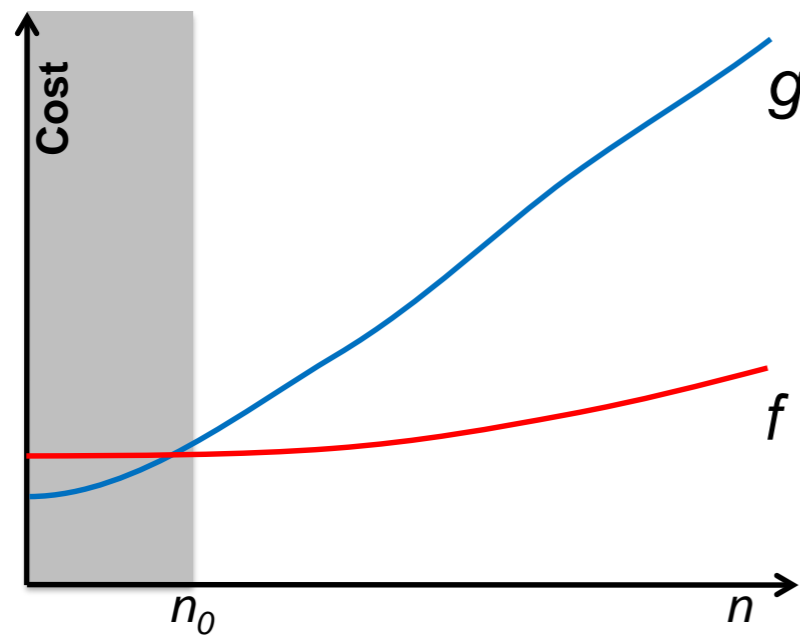


“Is f better than g ?”

- Attempt #2:

“ f is better than g ” if

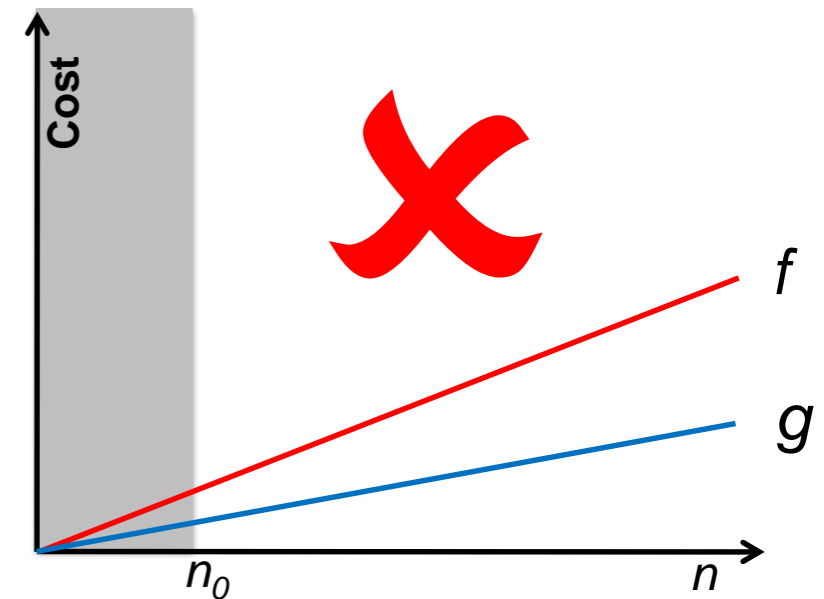
there exists a natural number n_0 such that
for all $n \geq n_0$, $f(n) \leq g(n)$



“Is f better than g ?”

- Attempt #2:

“ f is better than g ” if
there exists a natural number n_0 s.t.
for all $n \geq n_0$, $f(n) \leq g(n)$



- But is this useful?

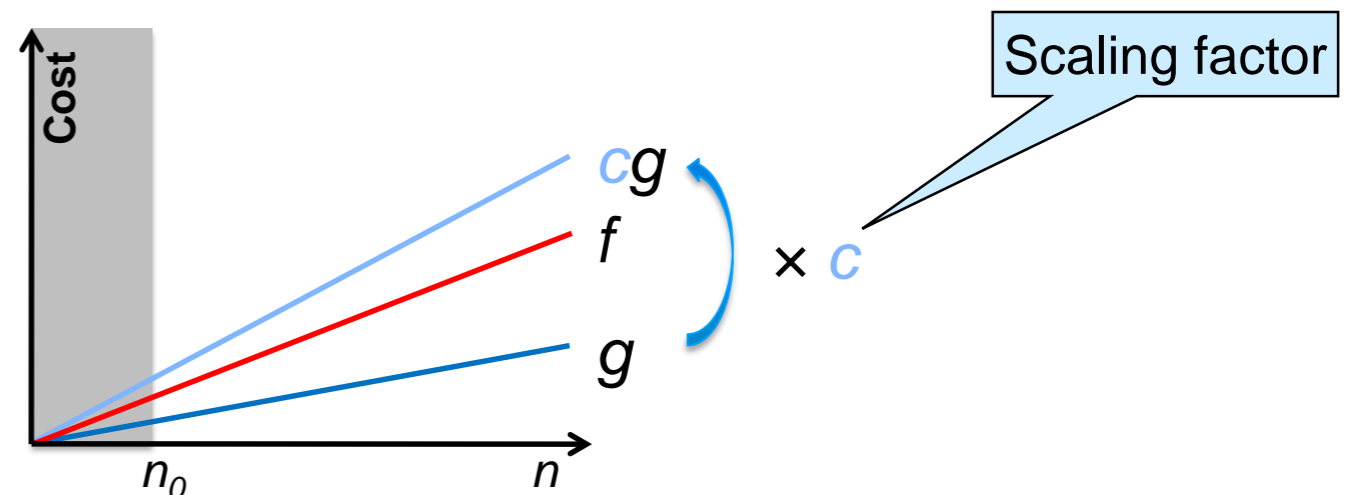
- f and g are both linear functions

- $f(n) = a_1n$ and $g(n) = a_2n$

- a_1 and a_2 summarize unknown instruction-level step constants

- we can't easily know if $a_1 > a_2$ or $a_1 < a_2$ or even $a_1 = a_2$

- **Solution:** scale g

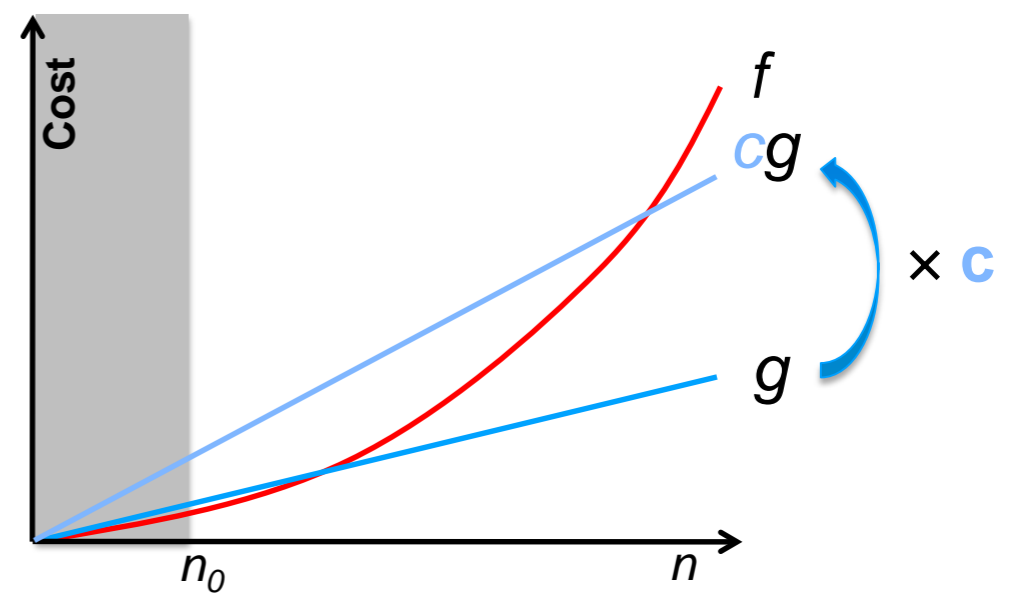
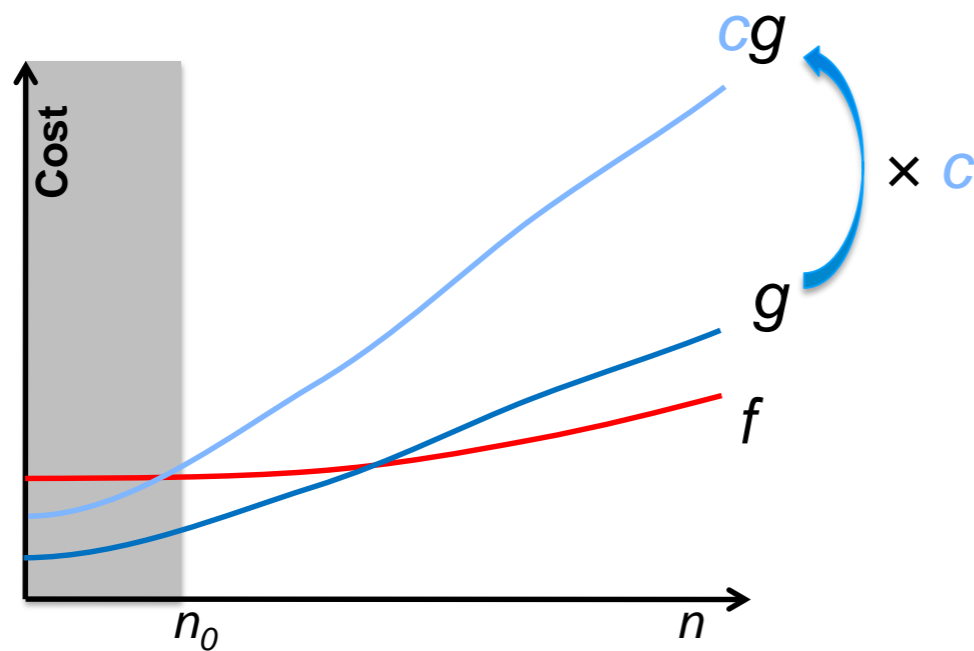


“Is f better than g ?”

- Final attempt:

“ f is better than g ” if

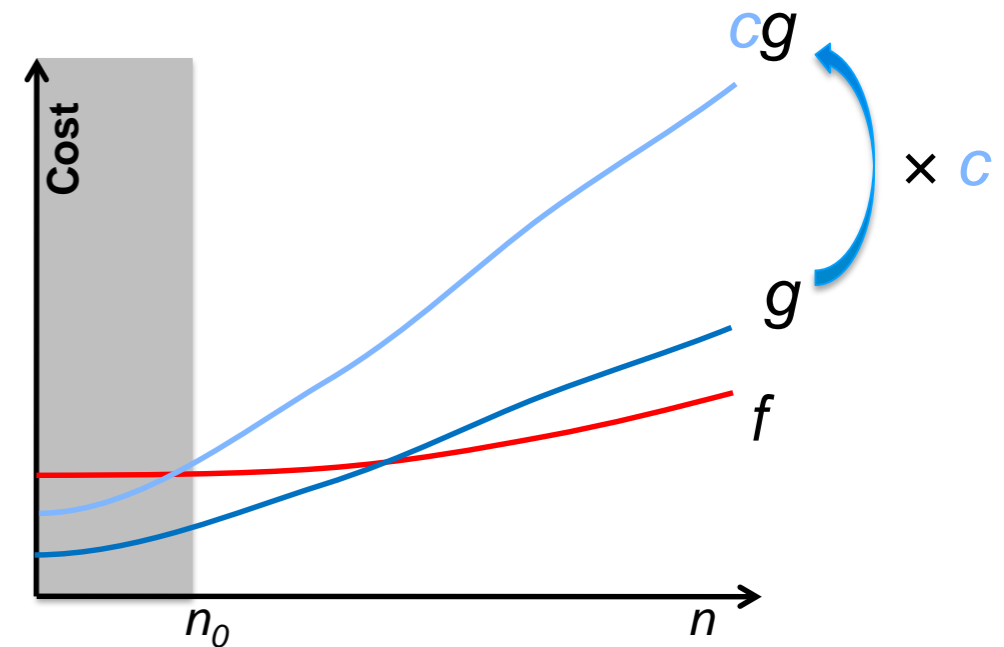
there exists a natural number n_0 and a real $c > 0$ s.t.
for all $n \geq n_0$, $f(n) \leq c g(n)$



Big O

Big-O

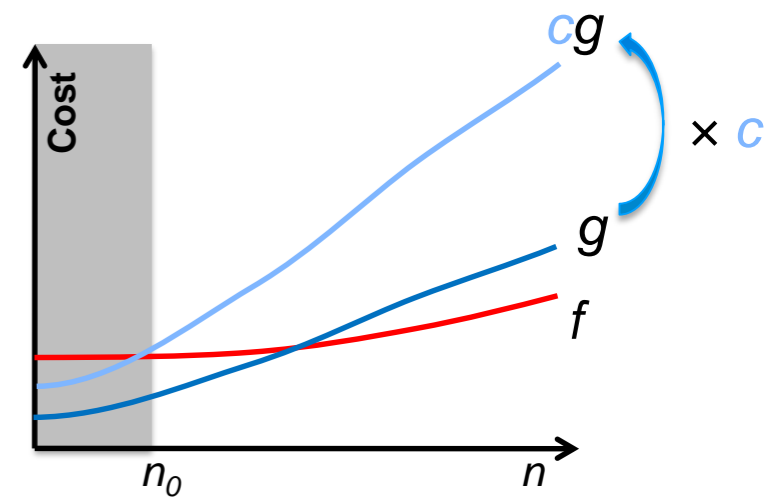
- Rather than “ f is better than g ”, we say $f \in O(g)$
 - “ f is in big-O of g ”



$f \in O(g)$ if
there exists a natural number n_0 and a real $c > 0$ s.t.
for all $n \geq n_0$, $f(n) \leq c g(n)$

- $O(g)$ is a **set**:
 $O(g) = \{ f \text{ s.t. there exists a natural number } n_0$
 $\text{and a real } c > 0 \text{ s.t.}$
 $\text{for all } n \geq n_0, f(n) \leq c g(n) \}$

Big-O



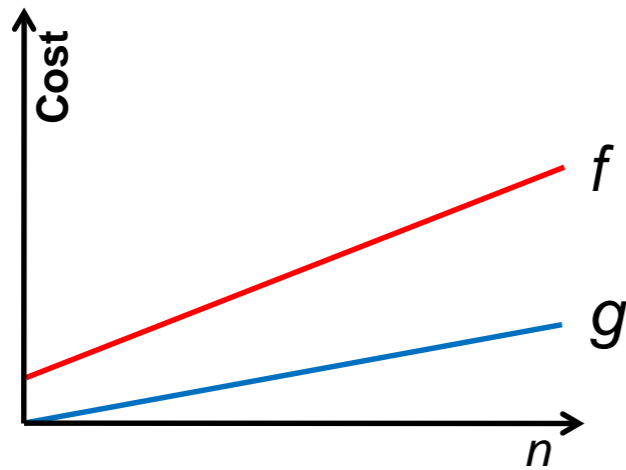
- Given two concrete functions f and g , how to tell if $f \in O(g)$?
 - do the math
 - find n_0 and c and show that the definition holds
 - or show that the definition cannot hold for any n_0 or c
 - recall what you learned in your calculus classes
 - enough for most of this course

Big-O

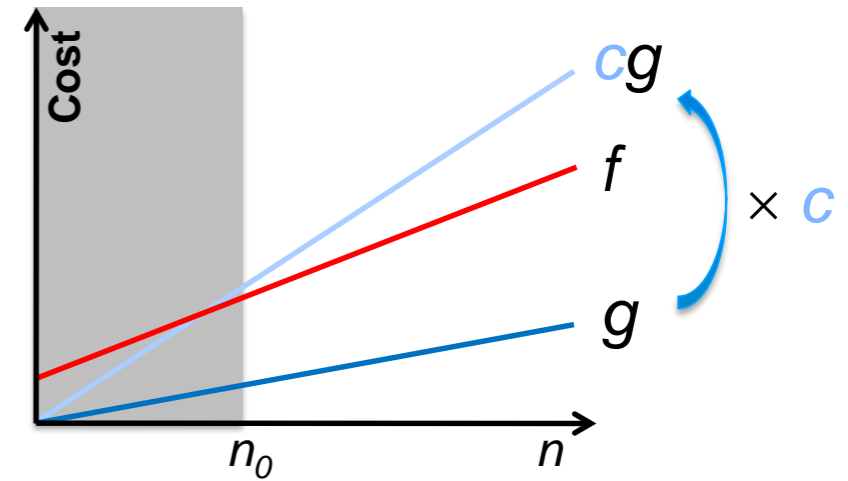
- Is $3n + 2 \in O(n)$?

$f(n) = 3n + 2$

$g(n) = n$



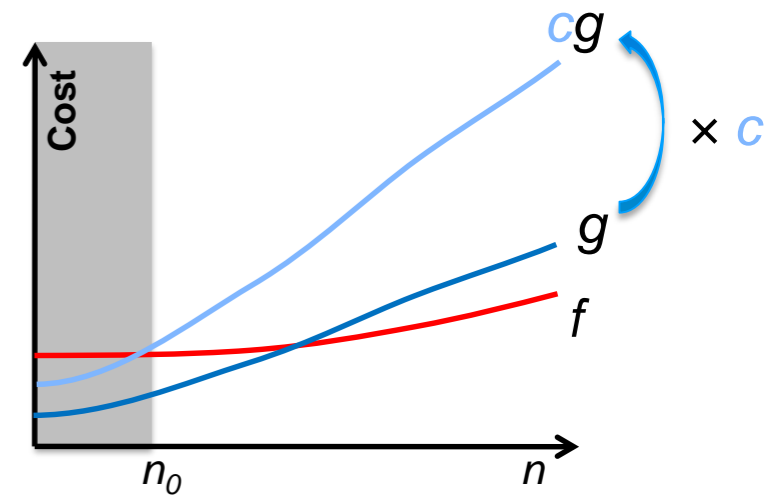
Take $c = 3.5$
and $n_0 = 4$



Any bigger values
work too!



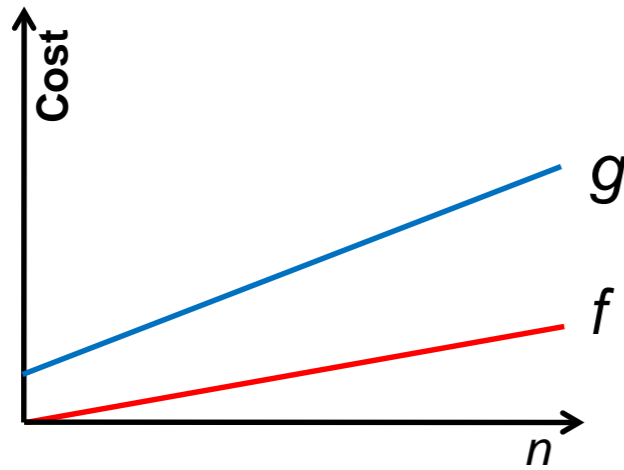
Big-O



● Is $n \in O(3n + 2)$?

$f(n) = n$

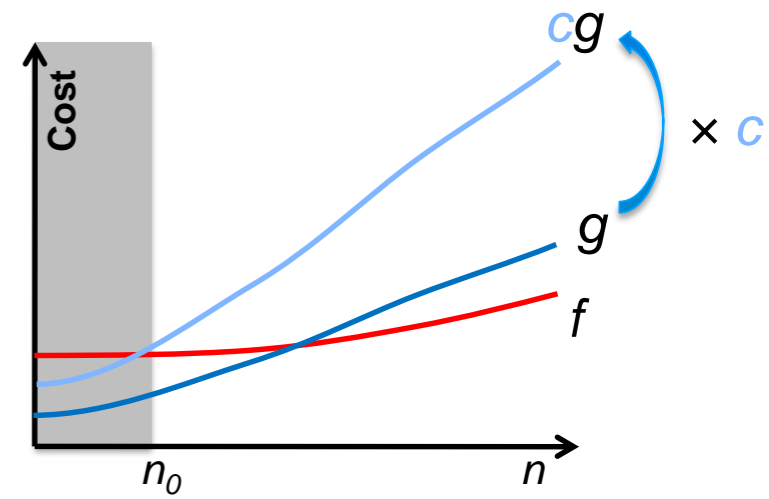
$g(n) = 3n + 2$



Take $c = 1.0$
and $n_0 = 0$



Big-O



- Every **linear cost function** is in $O(n)$
 - as long as the coefficient of n is not 0
- Every linear cost function is also in $O(3n + 2)$
- As sets, $O(n) = O(3n + 2)$
 - $O(n)$ is simpler however
 - $g(n) = n$ is the **simplest** linear function
- We describe a cost function is linear by saying that it is in $O(n)$

Big-O

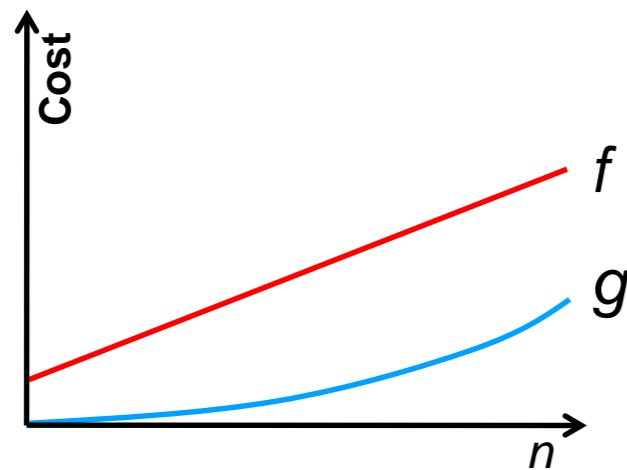
- Similarly, every **quadratic cost function** is in $O(n^2)$
 - as long as the coefficient of n^2 is not 0
 - the other coefficients don't matter
 - $g(n) = n^2$ is the **simplest** quadratic function
- We describe a cost function is quadratic by saying that it is in $O(n^2)$
- In general, every **polynomial cost function of degree p** is in $O(n^p)$
 - as long as the lead coefficient of n^p is not 0
 - $g(n) = n^p$ is the **simplest** polynomial of degree p
- We describe a cost function is polynomial of degree p by saying that it is in $O(n^p)$

Big-O

- Is $3n + 2 \in O(n^2)$?

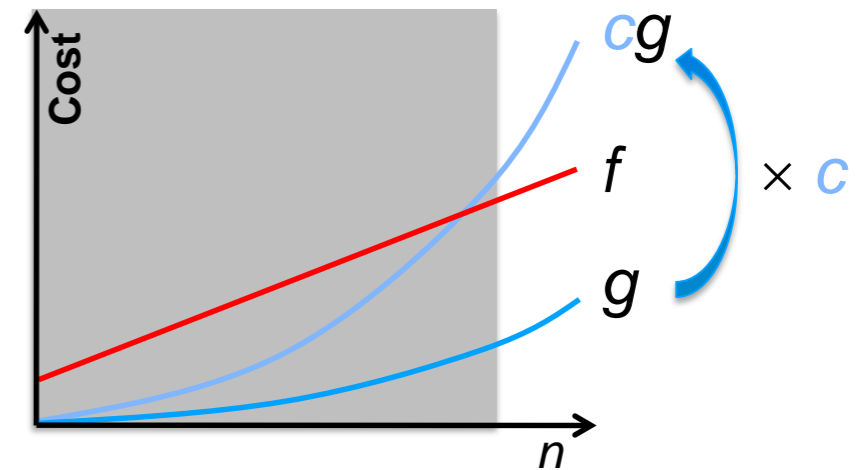
$$f(n) = 3n + 2$$

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



Take $c = 1.5$
and $n_0 = 3$

Any bigger values
work too!



Big-O

- $3n + 2 \in O(n)$ and $3n + 2 \in O(n^2)$
 - $O(n)$ is **tighter** however
- Every linear function is in $O(n^2)$
 - $O(n) \subseteq O(n^2)$
- In general, if $p \leq q$
 - $O(n^p) \subseteq O(n^q)$

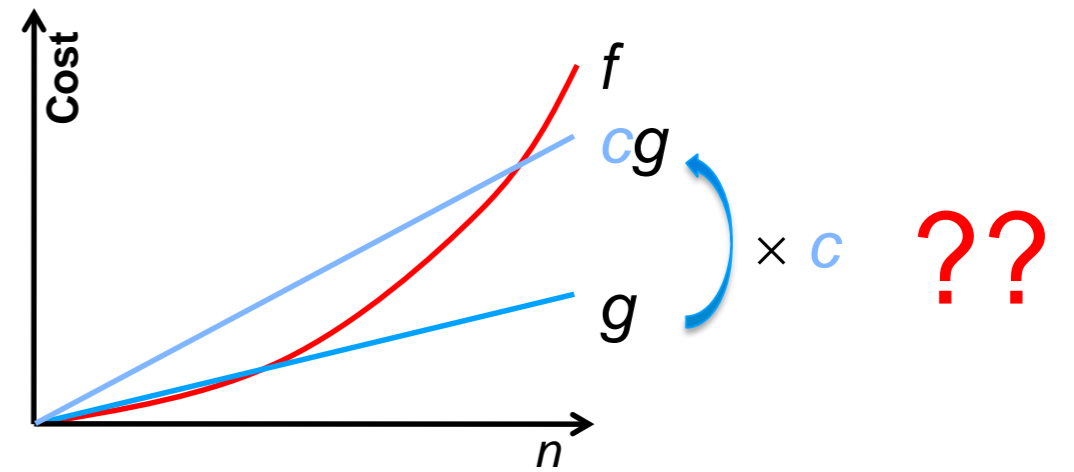
Big-O

- Is $n^2 \in O(3n + 2)$?

$$f(n) = n^2$$

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

- n^2 eventually dominates $c(3n+2)$ no matter the scaling factor c
- $n^2 \notin O(3n + 2)$



- Not every quadratic function is in $O(n)$
 - $O(n^2) \not\subset O(n)$
 - $O(n) \subset O(n^2)$

Complexity Classes

Complexity Classes

- We learned that $O(n) \subset O(n^2)$
 - $O(n) \subseteq O(n^2)$
 - but $O(n^2) \not\subseteq O(n)$
- $O(n)$ and $O(n^2)$ are called **complexity classes**
 - Simplest and tightest expressions for sets of cost functions

Complexity Classes

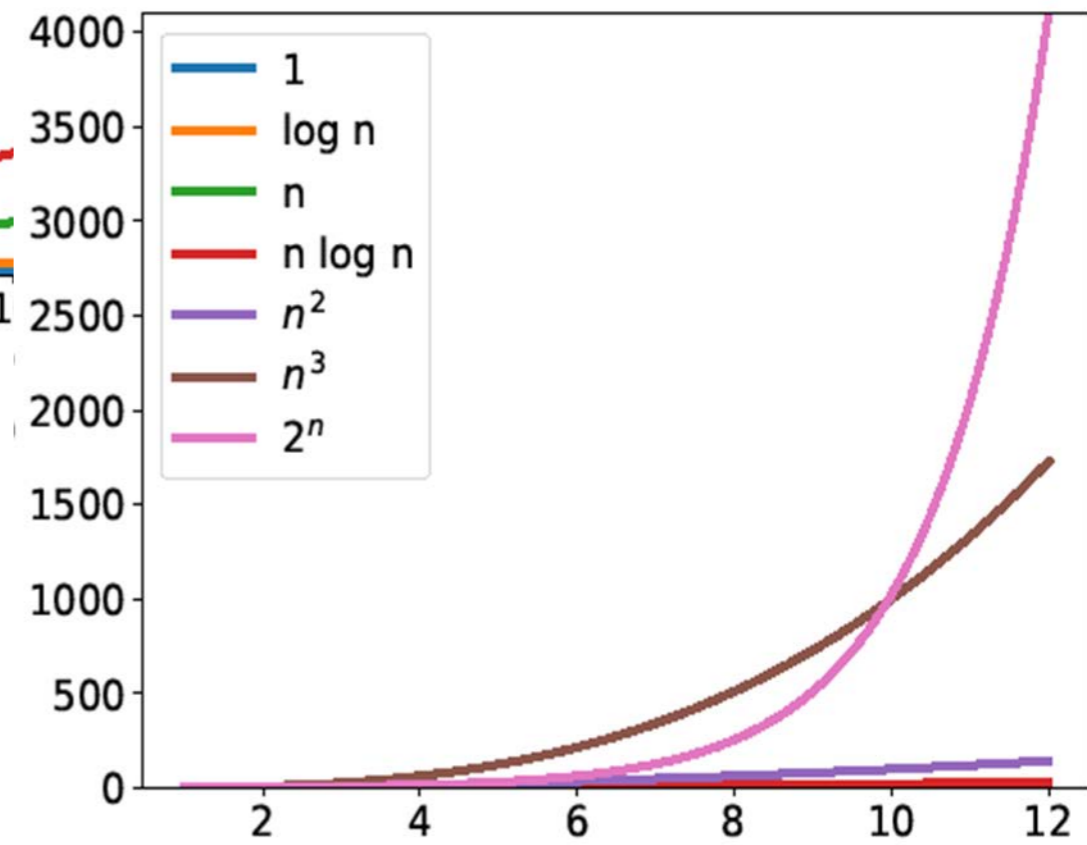
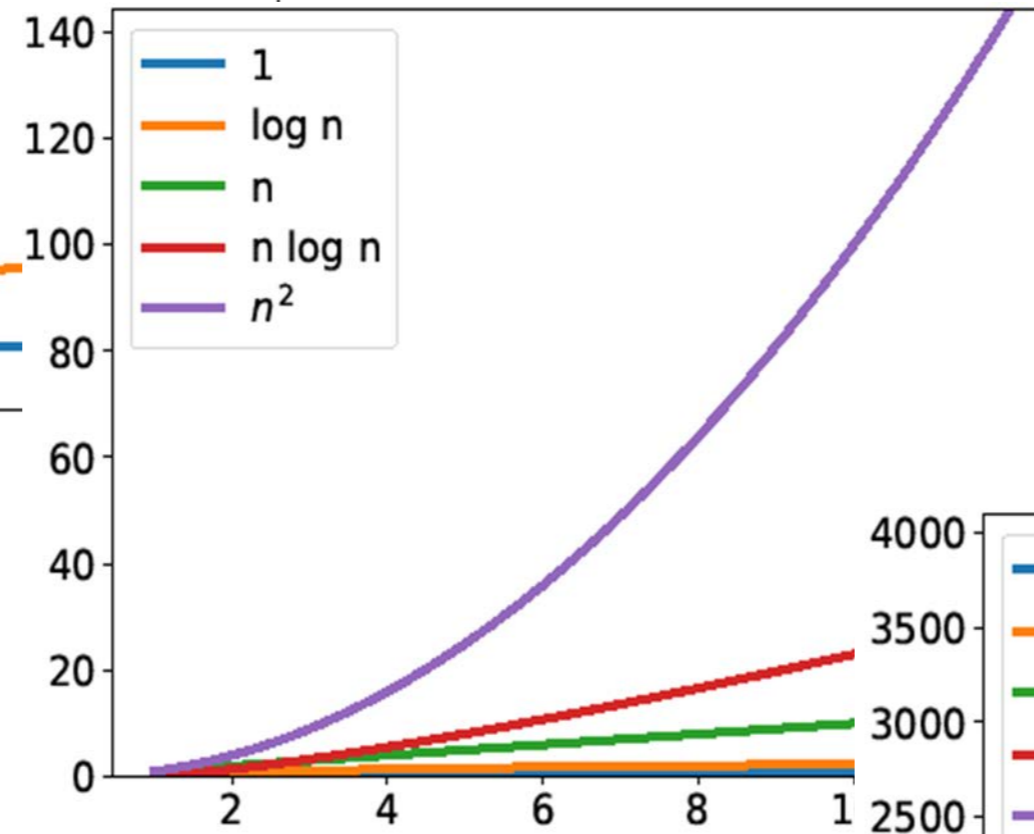
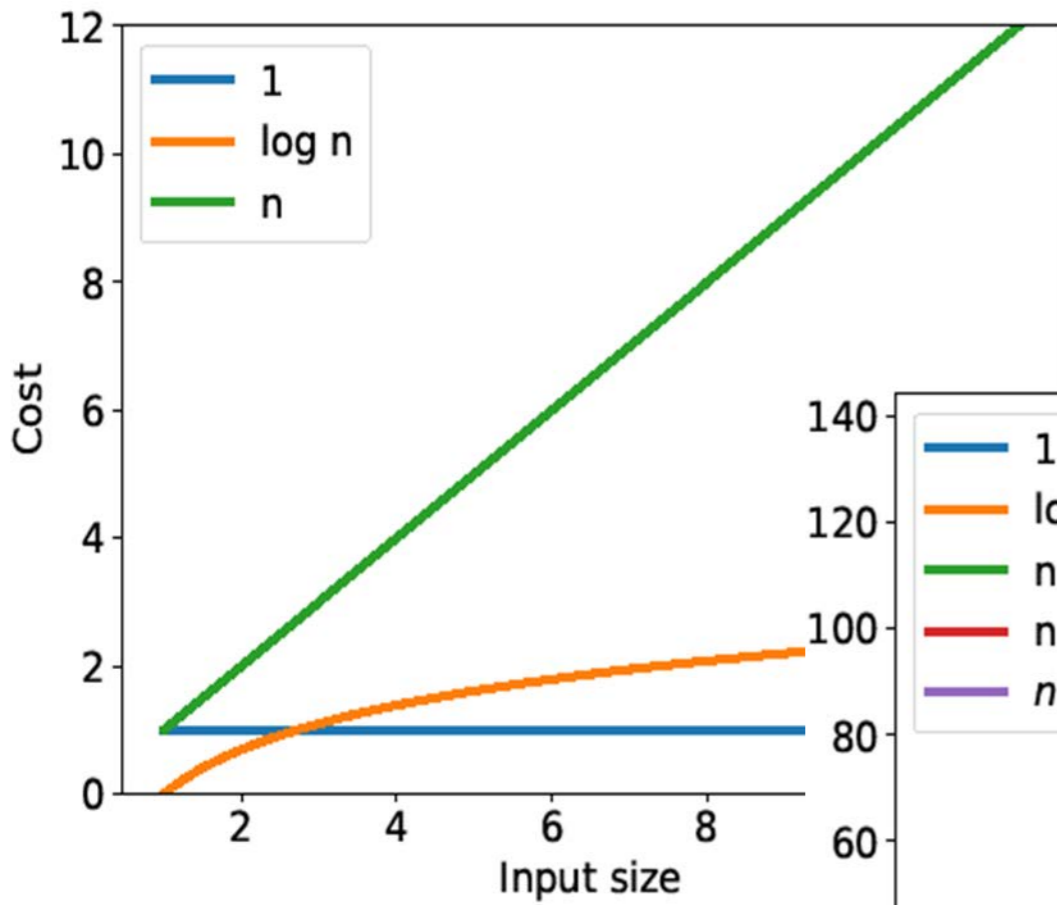
Complexity classes we will use in this course

Complexity class	Common name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	(just “ $n \log n$ ”)
$O(n^2)$	Quadratic
$O(2^n)$	Exponential

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(2^n)$$

- There are many more, though

Complexity Classes



Complexity of Linear Search

What do we mean by “How long”?

- $T(n) = 3n + 2$

- So $T(n) \in O(n)$

Simplest and tightest
class

```
int search(int x, int[] A, int n) {  
    for (int i = 0; i < n; i++) {  
        if (A[i] == x) return i;  
    }  
    return -1;  
}
```

- Linear search has **worst case complexity** $O(n)$

- It has **linear complexity** in the size n of its input

- That's why it is called *linear* search

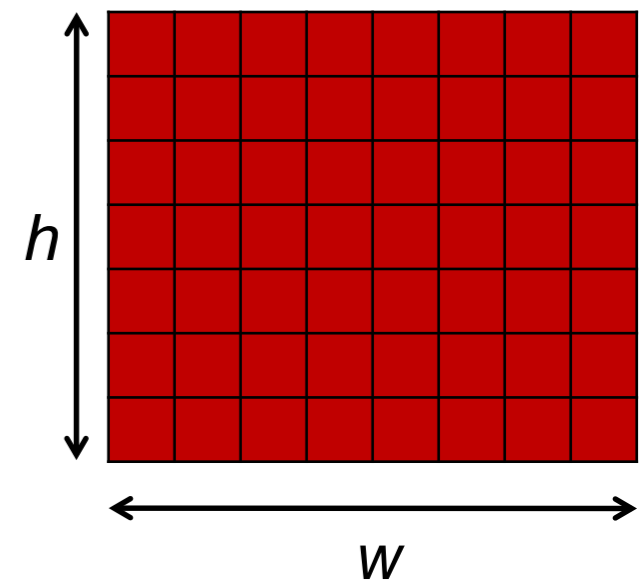
Big-O

- There is nothing special about “ n ”
 - If we call the size of the input k ,
 - then this function has cost $O(k)$
 - still linear

```
int search(int x, int[] A, int k) {  
    for (int i = 0; i < k; ++i) {  
        if (A[i] == x) return i;  
    }  
    return -1;  
}
```

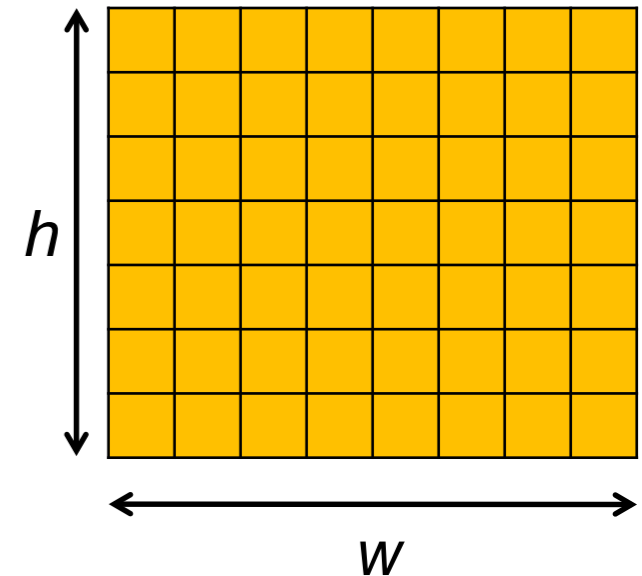
Big-O

An input may have multiple size parameters

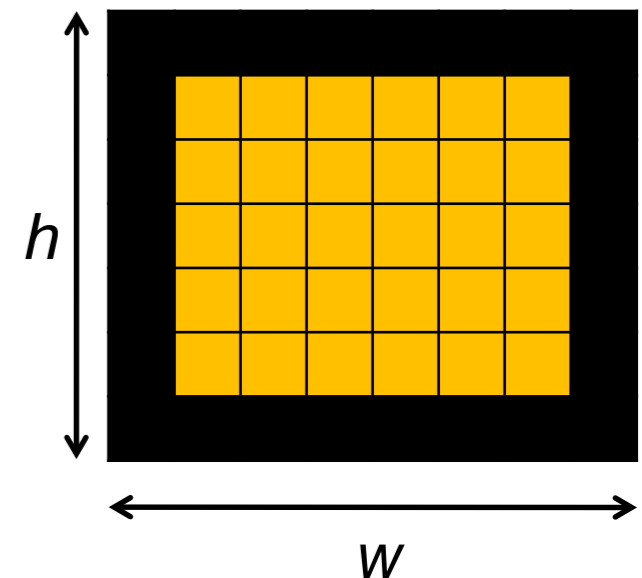


- For example, an image has a width w and a height h

- Brightening an image: $O(wh)$
 - We change every one of its $w \times h$ pixels



- Putting a border: $O(w+h)$
 - we touch about $2(w + h)$ pixels



Towards a Better Search

Algorithms vs. Problems

- Linear search has cost $O(n)$

```
int search(int x, int[] A, int n) {  
    for (int i = 0; i < n; i++) {  
        if (A[i] == x) return i;  
    }  
    return -1;  
}
```

- But this is only one of the many algorithms to search an array

- Can a different algorithm find an element faster?

- **No**: the **problem** of searching a *generic* n-element array has complexity $O(n)$

- some algorithms have worse complexity

- but no algorithm has better complexity

- unless we radically change what we mean by “step”

- Can we do better if we make (reasonable) assumptions?

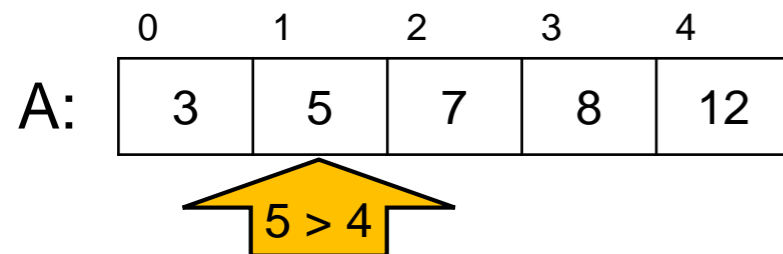
- Say, the array is **sorted**

Searching Sorted Arrays

A is sorted

- the segment $A[0,n)$ is sorted
 - (useful for later)

x: 4



● Idea:

- If we find an element larger than x, we can stop searching
 - All elements behind it will also be larger than x
 - That's because A is sorted

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
           || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    //@loop_invariant !is_in(x, A, 0, i);
    {
      if (A[i] == x) return i;
    }
  return -1;
}
```


Searching Sorted Arrays

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
           || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    //@loop_invariant !is_in(x, A, 0, i);
    {
      if (A[i] == x) return i;
      if (x < A[i]) return -1;
      //@assert A[i] < x;
    }
  return -1;
}
```

If $A[i]$ is not equal to x and not larger than x then it must be smaller than x

Searching Sorted Arrays

- What is the cost of this **search**?
- Still $O(n)$
 - **worst case** is when searching an element bigger than anything in A
- But that's just *one* algorithm for searching in a sorted array
 - Can we do better?
 - ... next lecture ...

```
int search(int x, int[] A, int n)
//@requires is_sorted(A, 0, n);
{
    for (int i = 0; i < n; i++) {
        if (A[i] == x) return i;
        if (x < A[i]) return -1;
        //@assert A[i] < x;
    }
    return -1;
}
```

Searching Sorted Arrays

- Is this code safe?

- Yes, no new array accesses

- Is it correct?

- Original argument still holds

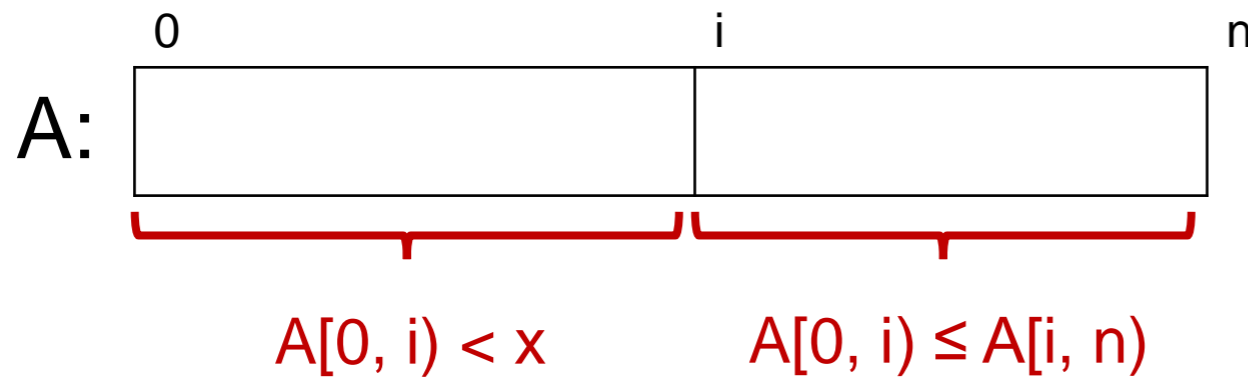
- but we have a new place where the function returns

- Is it correct there?
- no good argument
- we need new insight

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
           || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    //@loop_invariant !is_in(x, A, 0, i);
    {
      if (A[i] == x) return i;
      if (x < A[i]) return -1;
      //@assert A[i] < x;
    }
  return -1;
}
```

Searching Sorted Arrays

- What do we know at iteration i ?
 - Let's draw pictures!



- $A[0, i) < x$: every element in segment $A[0, i)$ is less than x
- $A[0, i) \leq A[i, n)$: everything in $A[0, i)$ is \leq everything in $A[i, n)$

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
           || (0 <= \result && \result < n &&
              A[\result] == x);          @*/
{
  for (int i = 0; i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    //@loop_invariant !is_in(x, A, 0, i);
    {
      if (A[i] == x) return i;
      if (x < A[i]) return -1;
      //@assert A[i] < x;
    }
  return -1;
}
```

Reasoning about Array Segments

- $A[0, i) < x$, etc are useful to reason about array segments
- Implement them into *specification* functions: **arrayutil**

➤ $A[lo, hi)$ is sorted

➤ $x \in A[lo, hi)$

➤ $x < A[lo, hi)$

➤ $x \leq A[lo, hi)$

➤ $x > A[lo, hi)$

➤ $x \geq A[lo, hi)$

➤ $A[lo_1, hi_1) < B[lo_2, hi_2)$

➤ $A[lo_1, hi_1) \leq B[lo_2, hi_2)$

➤ $A[lo_1, hi_1) > B[lo_2, hi_2)$

➤ $A[lo_1, hi_1) \geq B[lo_2, hi_2)$

`is_sorted(A, lo, hi)`

`is_in(x, A, lo, hi)`

`lt_seg(x, A, lo, hi)`

`le_seg(x, A, lo, hi)`

`gt_seg(x, A, lo, hi)`

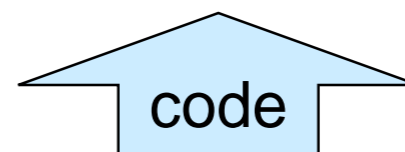
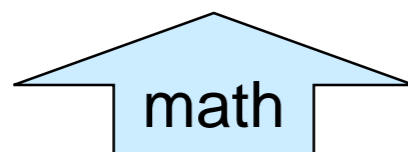
`ge_seg(x, A, lo, hi)`

`lt_segs(A, lo1, hi1, B, lo2, hi2)`

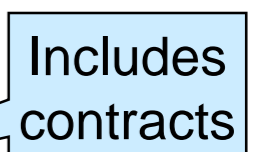
`le_segs(A, lo1, hi1, B, lo2, hi2)`

`gt_segs(A, lo1, hi1, B, lo2, hi2)`

`ge_segs(A, lo1, hi1, B, lo2, hi2)`

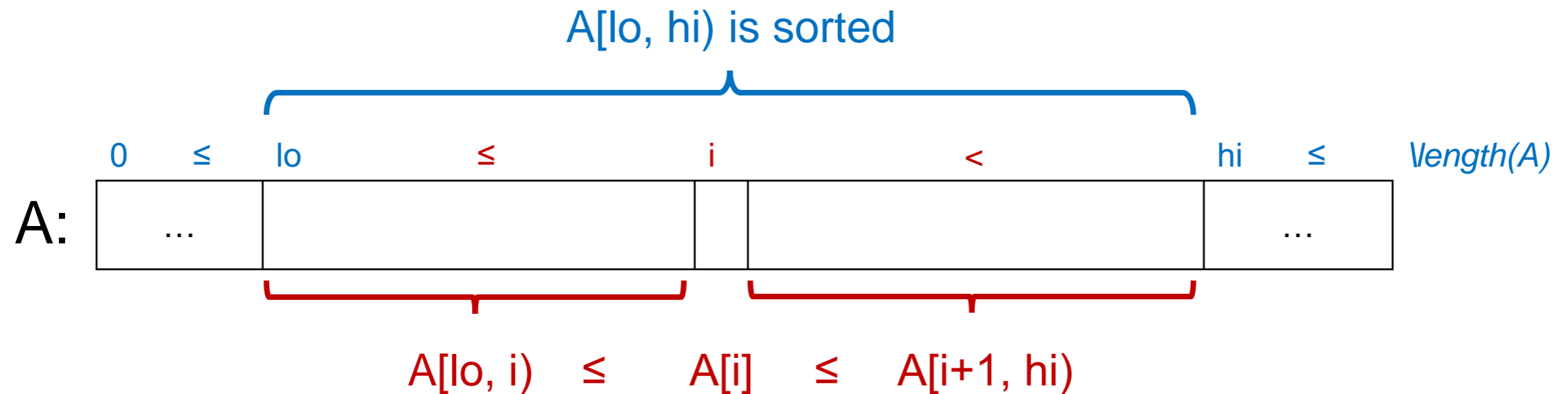


See **arrayutil.c0** file



Reasoning about Sorted Arrays

- If an array (segment) is sorted, what do we know?



- for every element $A[i]$

- $lo \leq i < hi$

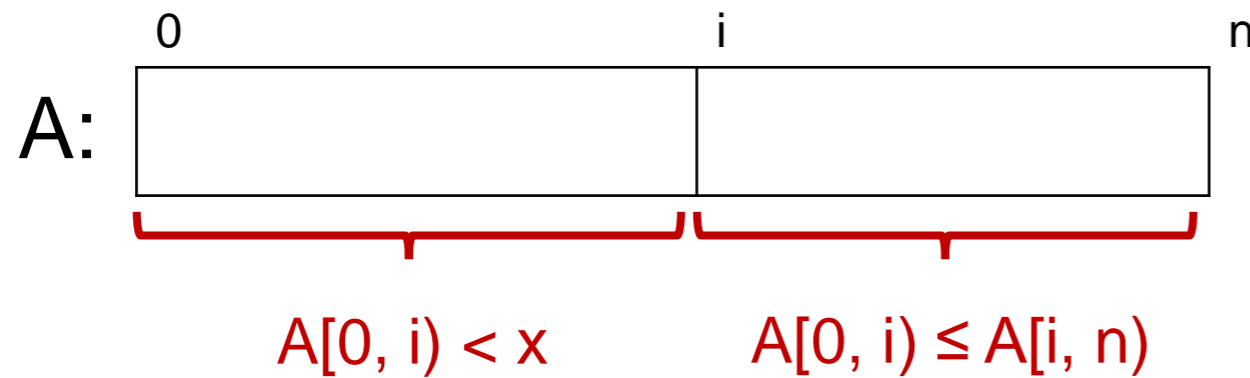
$A[lo, hi)$ can't be empty

- $A[lo, i) \leq A[i] \leq A[i+1, hi)$

- $A[lo, i)$ and $A[i+1, hi)$ are sorted

Searching Sorted Arrays

- What do we know at iteration i ?
 - Let's draw pictures!



- Candidate loop invariants
 - $gt_seg(x, A, 0, i)$: that's $A[0, i) < x$
 - This implies $!is_in(x, A, 0, i)$, that's $x \notin A[0, i)$
 - $le_segs(A, 0, i, A, i, n)$: that's $A[0, i) \leq A[i, n)$

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
           || (0 <= \result && \result < n &&
              A[\result] == x);          @*/
{
  for (int i = 0; i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    //@loop_invariant gt_seg(x, A, 0, i);
    //@loop_invariant le_segs(A, 0, i, A, i, n):
    {
      if (A[i] == x) return i;
      if (x < A[i]) return -1;
      //@assert A[i] < x;
    }
  return -1;
}
```

Searching Sorted Arrays

Is this code correct?

➤ **To show:** $\text{!is_in}(x, A, lo, hi)$
(assuming invariants are valid)

- A. $A[0, i) < x$ by line 10 (LI 2)
- B. $x \notin A[0, i)$ by math on A,
- C. $x < A[i]$ by line 14 (conditional)
- D. $x < A[i, n)$ by math on C and line 3 (precondition)
- E. $x \notin A[0, n)$ by math on B, D

```
1. int search(int x, int[] A, int n)
2. //@requires n == \length(A);
3. //@requires is_sorted(A, 0, n);
4. /*@ensures (\result == -1 && !is_in(x, A, 0, n))
5.           || (0 <= \result && \result < n &&
6.             A[\result] == x);          @*/
7. {
8.   for (int i = 0; i < n; i++)
9.     //@loop_invariant 0 <= i && i <= n;
10.    //@loop_invariant gt_seg(x, A, 0, i);
11.    //@loop_invariant le_segs(A, 0, i, A, i, n);
12.    {
13.      if (A[i] == x) return i;
14.      if (x < A[i]) return -1;
15.      //@assert A[i] < x;
16.    }
17.   return -1;
18. }
```


Searching Sorted Arrays

$x > A[0, i)$ is a **valid** loop invariant

INIT

➤ **To show:** $x > A[0, i)$ initially

- A. $i = 0$ by line 7
- B. $x > A[0, 0)$ by definition of **gt_seg**
 - $A[0,0)$ is the empty array segment
 - Nothing is in it

PRES

➤ **To show:** if $x > A[0, i)$, then $x > A[0, i')$

- A. $x > A[0, i)$ by assumption
- B. $x > A[i]$ by line 15 (math on lines 13 and 14)
- C. $x > A[0, i+1)$ by math on A and B
- D. $i' = i+1$ by line 8
- E. $x > A[0, i')$ by math on C and D

```
1. int search(int x, int[] A, int n)
2. //@requires n == \length(A);
3. //@requires is_sorted(A, 0, n);
4. /*@ensures (\result == -1 && !is_in(x, A, 0, n))
5.           || (0 <= \result && \result < n &&
6.             A[\result] == x);          @*/
7. {
8.   for (int i = 0; i < n; i++)
9.     //@loop_invariant 0 <= i && i <= n;
10.    //@loop_invariant gt_seg(x, A, 0, i);
11.    //@loop_invariant le_segs(A, 0, i, A, i, n);
12.    {
13.      if (A[i] == x) return i;
14.      if (x < A[i]) return -1;
15.      //@assert A[i] < x;
16.    }
17.   return -1;
18. }
```

Searching Sorted Arrays

$A[0,i) \leq A[i,n)$ is a **valid** loop invariant

INIT

➤ **To show:** $A[0, i) \leq A[i,n)$ initially

- A. $i = 0$ by line 7
- B. $A[0, 0) \leq A[i,n)$ by definition of **le_segs**
 - $A[0,0)$ is the empty array segment
 - Nothing is in it

PRES

➤ **To show:** if $A[0, i) \leq A[i, n)$, then $A[0, i') \leq A[i', n)$

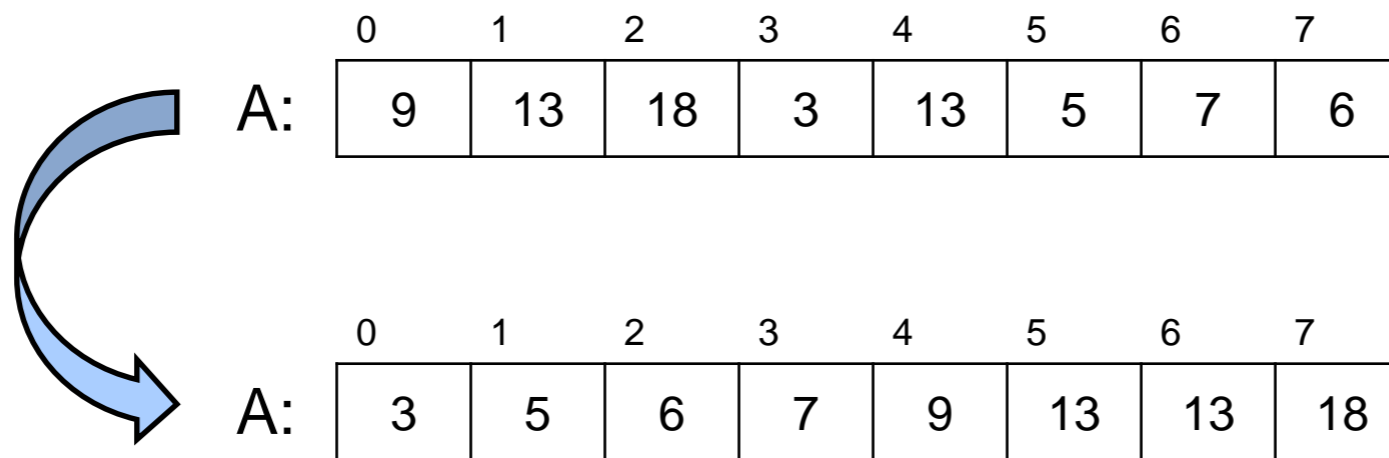
- A. $A[0, i) \leq A[i, n)$ by assumption
- B. $A[0, n)$ sorted by line 3 (precondition)
- C. $A[0, i+1) \leq A[i+1, n)$ by math on B
- D. $i' = i+1$ by line 8
- E. $A[0, i') \leq A[i', n)$ by math on C and D

```
1. int search(int x, int[] A, int n)
2. //@requires n == \length(A);
3. //@requires is_sorted(A, 0, n);
4. /*@ensures (\result == -1 && !is_in(x, A, 0, n))
5.           || (0 <= \result && \result < n &&
6.              A[\result] == x);          @*/
7. {
8.   for (int i = 0; i < n; i++)
9.     //@loop_invariant 0 <= i && i <= n;
10.    //@loop_invariant gt_seg(x, A, 0, i);
11.    //@loop_invariant le_segs(A, 0, i, A, i, n);
12.    {
13.     if (A[i] == x) return i;
14.     if (x < A[i]) return -1;
15.     //@assert A[i] < x;
16.    }
17.   return -1;
18. }
```

Selection Sort

Sorting an Array

- Reorder the elements to put them in increasing order
 - Duplicate elements are allowed



- There are many algorithms to sort arrays

Selection Sort


- Find element that shall go in $A[0]$
 - Smallest element in $A[0, n)$
- Swap it with $A[0]$
- Find element that shall go in $A[1]$
 - Smallest element in $A[1, n)$
- Swap it with $A[1]$
- ... carry on ...
- Stop when A is entirely sorted

A:

0	1	2	3	4	5	6	7
9	13	18	3	13	5	7	6

A:

0	1	2	3	4	5	6	7
3	13	18	9	13	5	7	6




A:

0	1	2	3	4	5	6	7
3	13	18	9	13	5	7	6

A:

0	1	2	3	4	5	6	7
3	5	18	9	13	13	7	6



A:

0	1	2	3	4	5	6	7
3	5	6	7	9	13	13	18

Selection Sort

We need two operations

- find the minimum of an array segment $A[lo, hi)$
 - and return its index

$A[lo,hi)$ can't be empty

```
int find_min(int[] A, int lo, int hi)
//@requires 0 <= lo && lo < hi && hi <= \length(A);
/* @ensures lo <= \result && \result < hi
    && le_seg(A[\result], A, lo, hi); @*/ ;
```

That's $A[\text{result}] \leq A[lo, hi)$

- swap two elements of an array (given their indices)

```
// swaps A[i] and A[j]; all other elements are unchanged
void swap(int[] A, int i, int j)
//@requires 0 <= i && i < \length(A);
//@requires 0 <= j && j < \length(A);
```

returns no value
swap is **in-place**

We can't say this as
a postcondition.
We use a comment instead

Implementation left as exercise

Selection Sort

- Let's capture our intuition about how it works in code
 - Generalization: sort array segment $A[lo, hi)$

sort is in-place

$A[lo, hi)$ can be empty

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  for (int i = lo; i < hi; i++)
  {
    int min = find_min(A, i, hi);
    swap(A, i, min);
  }
}
```

for every index i
from lo to hi

find the minimum
of $A[i, hi)$

swap it with
 $A[i]$

Selection Sort

```
int find_min(int[] A, int lo, int hi)
//@requires 0 <= lo && lo < hi && hi <= \length(A);
/*@ensures ... @*/;
```

Is this code safe?

● find_min(A, i, hi)

➤ To show: $0 \leq i < hi \leq \text{length}(A)$

A. $hi \leq \text{length}(A)$ by line 2

B. $i < hi$ by line 5

C. $0 \leq i$ *oops! we need the usual loop invariant*

➤ // @loop_invariant $lo \leq i$; then

a. $0 \leq lo$ by line 2

b. $lo \leq i$ by new LI

c. $0 \leq i$ by math



```
1. void sort(int[] A, int lo, int hi)
2. //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3. //@ensures is_sorted(A, lo, hi);
4. {
5.   for (int i = lo; i < hi; i++)
6.   {
7.     int min = find_min(A, i, hi);
8.     swap(A, i, min);
9.   }
10. }
```


Selection Sort

Is this code safe?

● swap(A, i, min)

➤ To show: $0 \leq \text{min} < \text{length}(A)$

A. $0 \leq i$ by lines 2 and 6

B. $i \leq \text{min}$ by line postconditions of find_min

C. $\text{min} < \text{hi}$ by line postconditions of find_min

D. $\text{hi} \leq \text{length}(A)$
by line 2

➤ To show: $0 \leq i$
&& $i < \text{length}(A)$

(similar)



```
int find_min(int[] A, int lo, int hi)
//@requires 0 <= lo && lo < hi && hi <= \length(A);
/*@ensures lo <= \result && \result < hi
    && le_seg(A[\result], A, lo, hi); @*/;
```

```
void swap(int[] A, int i, int j)
//@requires 0 <= i && i < \length(A);
//@requires 0 <= j && j < \length(A);
```

} $0 \leq \text{min}$ by math

```
1. void sort(int[] A, int lo, int hi)
2. //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3. //@ensures is_sorted(A, lo, hi);
4. {
5.   for (int i = lo; i < hi; i++)
6.     //@loop_invariant lo <= i;
7.     {
8.       int min = find_min(A, i, hi);
9.       swap(A, i, min);
10.    }
11. }
```

Added

Selection Sort

Is this code correct?

- To show: $\text{is_sorted}(A, \text{lo}, \text{hi})$
- What do we know at iteration i ?
 - Let's draw pictures!

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi && hi <= \length(A);
//@ensures is_sorted(A, lo, hi):
{
  for (int i = lo; i < hi; i++)
    //@loop_invariant lo <= i;
    {
      int min = find_min(A, i, hi);
      swap(A, i, min);
    }
}
```

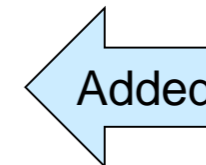


- Candidate loop invariants
 - $\text{lo} \leq i \ \&\& \ i \leq \text{hi}$
 - $\text{is_sorted}(A, \text{lo}, i)$
 - $\text{le_segs}(A, \text{lo}, i, A, i, \text{hi})$

Selection Sort

- Resulting code

```
1. void sort(int[] A, int lo, int hi)
2. // @requires 0 <= lo && lo <= hi && hi <= \length(A);
3. // @ensures is_sorted(A, lo, hi);
4. {
5.   for (int i = lo; i < hi; i++)
6.     // @loop_invariant lo <= i && i <= hi;
7.     // @loop_invariant is_sorted(A, lo, i);
8.     // @loop_invariant le_segs(A, lo, i, A, i, hi);
9.     {
10.      int min = find_min(A, i, hi);
11.      swap(A, i, min);
12.    }
13. }
```



- We need to prove that the added invariants are valid

Selection Sort

Is this code correct?

- **To show:** `is_sorted(A, lo, hi)`
(assuming invariants are valid)

```
1. void sort(int[] A, int lo, int hi)
2. //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3. //@ensures is_sorted(A, lo, hi);
4. {
5.     for (int i = lo; i < hi; i++)
6.         //@loop_invariant lo <= i && i <= hi;
7.         //@loop_invariant le_segs(A, lo, i, A, i, hi);
8.         //@loop_invariant is_sorted(A, lo, i);
9.         {
10.            int min = find_min(A, i, hi);
11.            swap(A, i, min);
12.        }
13. }
```

- A. `i ≥ hi` by line 5 (negation of loop guard)
- B. `i ≤ hi` by line 6 (LI 1)
- C. `i = hi` by math on A, B
- D. `is_sorted(A, lo, hi)` by line 8 (LI 2) and C

○ This is a standard EXIT argument

- But are the loop invariants valid?

We didn't need LI 3
`A[lo, i] ≤ A[i, hi]`

Selection Sort

Are the loop invariants valid?

INIT

- To show: $lo \leq lo$ by math
- To show: $lo \leq hi$ by line 2 (preconditions)
- To show: $A[lo, lo)$ sorted by math (empty interval)
- To show: $A[lo, lo) \leq A[lo, hi)$ by math (empty interval)

PRES

- To show: if $lo \leq i \leq hi$, then $lo \leq i' \leq hi$

Proof left as exercise

```
1. void sort(int[] A, int lo, int hi)
2. //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3. //@ensures is_sorted(A, lo, hi);
4. {
5.   for (int i = lo; i < hi; i++)
6.     //@loop_invariant lo <= i && i <= hi;
7.     //@loop_invariant is_sorted(A, lo, i);
8.     //@loop_invariant le_segs(A, lo, i, A, i, hi);
9.   {
10.    int min = find_min(A, i, hi);
11.    swap(A, i, min);
12.   }
13. }
```

Selection Sort

Are the loop invariants valid?

PRES

➤ **To show:** if $A[lo, i)$ is sorted,
then $A[lo, i')$ is sorted

```
1. void sort(int[] A, int lo, int hi)
2. //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3. //@ensures is_sorted(A, lo, hi);
4. {
5.   for (int i = lo; i < hi; i++)
6.     //@loop_invariant lo <= i && i <= hi;
7.     //@loop_invariant is_sorted(A, lo, i);
8.     //@loop_invariant le_segs(A, lo, i, A, i, hi);
9.     {
10.      int min = find_min(A, i, hi);
11.      swap(A, i, min);
12.    }
13. }
```

A. $A[lo, i)$ is sorted

B. $A[lo, i) \leq A[i, hi)$

C. $A[lo, i) \leq A[i]$

D. $A[lo, i+1)$ is sorted

E. $i' = i+1$

assumption

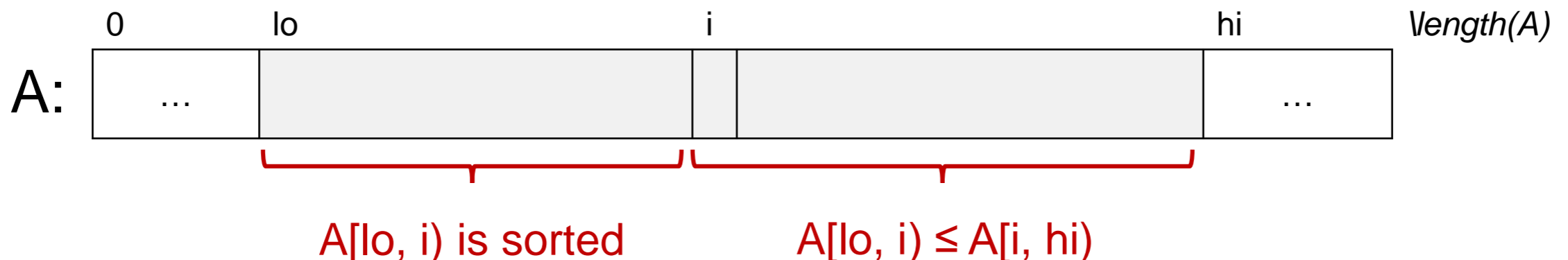
by line 8 (LI 3)

by math on B

by math on C

by line 5

This is where
we need LI 3!



Selection Sort

Are the loop invariants valid?

PRES

➤ **To show:** if $A[lo, i) \leq A[i, hi)$,
then $A[lo, i') \leq A[i', hi)$

A. $A[lo, i) \leq A[i, hi)$

B. $A[min] \leq A[i, hi)$

C. $A[i] \leq A[i, hi)$

D. $A[i] \leq A[i+1, hi)$

E. $A[lo, i) \leq A[i]$

F. $A[lo, i+1) \leq A[i+1, hi)$

G. $i' = i+1$

assumption

by postcondition of `find_min`

after swap by definition (in comment)

by math

by math on A and definition of `swap`

by math on E and D

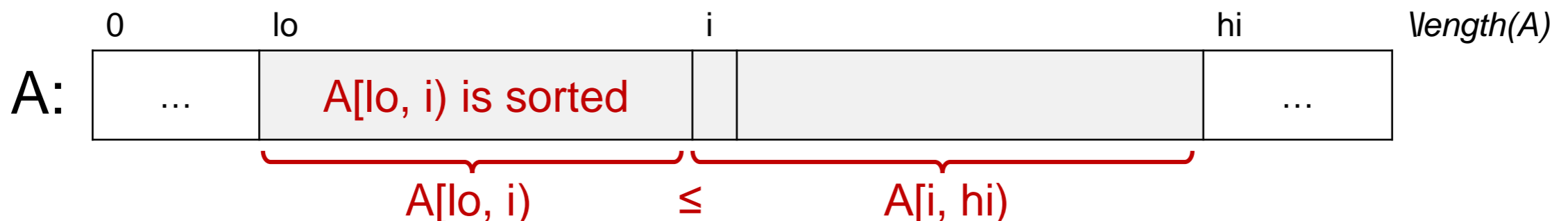
by line 5

after
swap

```

1. void sort(int[] A, int lo, int hi)
2. //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3. //@ensures is_sorted(A, lo, hi);
4. {
5.   for (int i = lo; i < hi; i++)
6.     //@loop_invariant lo <= i && i <= hi;
7.     //@loop_invariant is_sorted(A, lo, i);
8.     //@loop_invariant le_segs(A, lo, i, A, i, hi);
9.     {
10.      int min = find_min(A, i, hi);
11.      swap(A, i, min);
12.    }
13. }

```



Selection Sort

```
1. void sort(int[] A, int lo, int hi)
2. //@requires 0 <= lo && lo <= hi && hi <= \length(A);
3. //@ensures is_sorted(A, lo, hi);
4. {
5.   for (int i = lo; i < hi; i++)
6.     //@loop_invariant lo <= i && i <= hi;
7.     //@loop_invariant is_sorted(A, lo, i);
8.     //@loop_invariant le_segs(A, lo, i, A, i, hi);
9.     {
10.      int min = find_min(A, i, hi);
11.      swap(A, i, min);
12.     }
13. }
```

- We have proved it correct



Cost of Selection Sort

```
void sort(int[] A, int lo, int hi) {  
    for (int i = lo; i < hi; i++) {  
        int min = find_min(A, i, hi);  
        swap(A, i, min);  
    }  
}
```

Contracts
omitted

- Assume array segment $A[lo, hi)$ has length n
- The loop runs n time
- At each iteration,
 - call `find_min` on segment of length $j = hi - i$
 - j ranges over $1, \dots, n$
 - `find_min` needs to scan the entire segment once: cost $\mathbf{O}(j)$
 - in particular, it makes j comparisons
 - number of comparisons is a convenient proxy for our unit of cost
 - call `swap`
 - simply swaps values at two indices: cost $\mathbf{O}(1)$
- Number of comparisons to sort an n -element array segment

Carl Friedrich Gauss
came up with
this formula
when he was
9 years old



$$n + (n-1) + \dots + 1 = \sum_{j=0}^n j = n(n+1)/2$$

Cost of Selection Sort

```
void sort(int[] A, int lo, int hi) {  
    for (int i = lo; i < hi; i++) {  
        int min = find_min(A, i, hi);  
        swap(A, i, min);  
    }  
}
```

- Assume array segment $A[lo, hi)$ has length n
- Number of comparisons to sort an n -element array segment

$$n(n+1)/2$$

- $n(n+1)/2 \in O(n^2)$

Selection sort has cost is in **$O(n^2)$**