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

Compiling with contracts disabled

The search function

main that generates random tests and runs them

Runs a.out 200 times on arrays of size 1000000

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
```

Unix utility that times execution

Execution took 0.82 seconds

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

Wall clock?

- general: yes
- applicable to a large class of programs: yes
- independent of particular hardware: no
- applicable to many types of resources: no — *just time*
- mathematically rigorous: no — *result varies*
- useful: unclear

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;
}
```

Omitting contracts

- i = 0 — *1* step
- loop — *n* times
  - i < n — *1* step
  - if (A[i] == x) — *1* step
  - i++ — *1* step
- return -1 — *1* step

3n + *2* steps

What do we mean by “How long”?

Step count

- *3n + 2* steps 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;
}
```

- Always?
  - only if element is not found
- This is a **worst-case analysis**
  - Gives an **upper bound** on the number of steps

What do we mean by “How long”?

Step count

- *3n + 2* steps 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;
}
```

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

- *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"?

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

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,

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

Note that *a* and *b* make it hard to plot exactly

- 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
  - **about** “how long”
    - we can’t easily learn the constants a, b, …
  - **at most** “how long”
    - in the worst case

- We need to develop math that deals with
  - “about”
  - “at most”

  to reason about “how long” a function takes to run

That’s big-O

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

We will often keep the constants implicit:
- 3n+2 when we really mean
- an+b

Our new math will need to deal with this

Comparing Cost

Comparing Cost

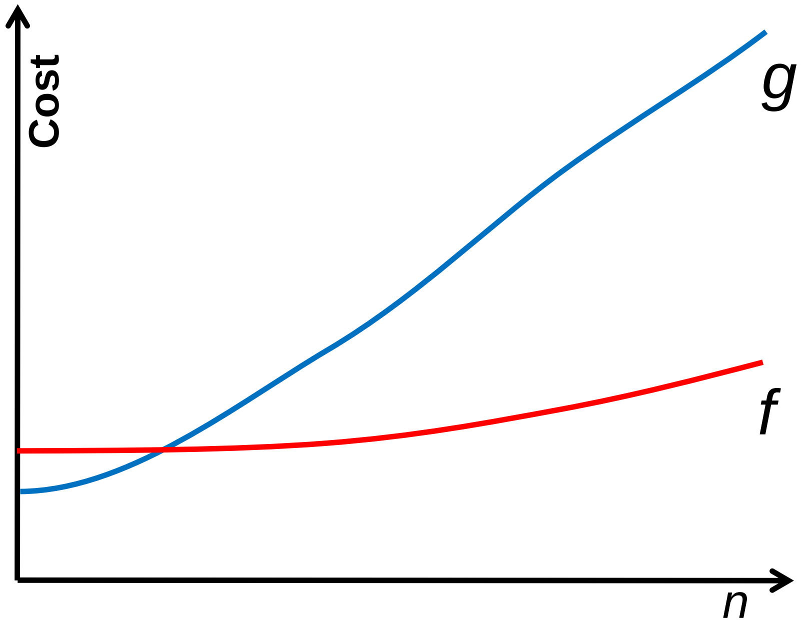
- Given two C0 functions that solve the same problem
  - F has cost *f(n)*
  - G has cost *g(n)*

  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

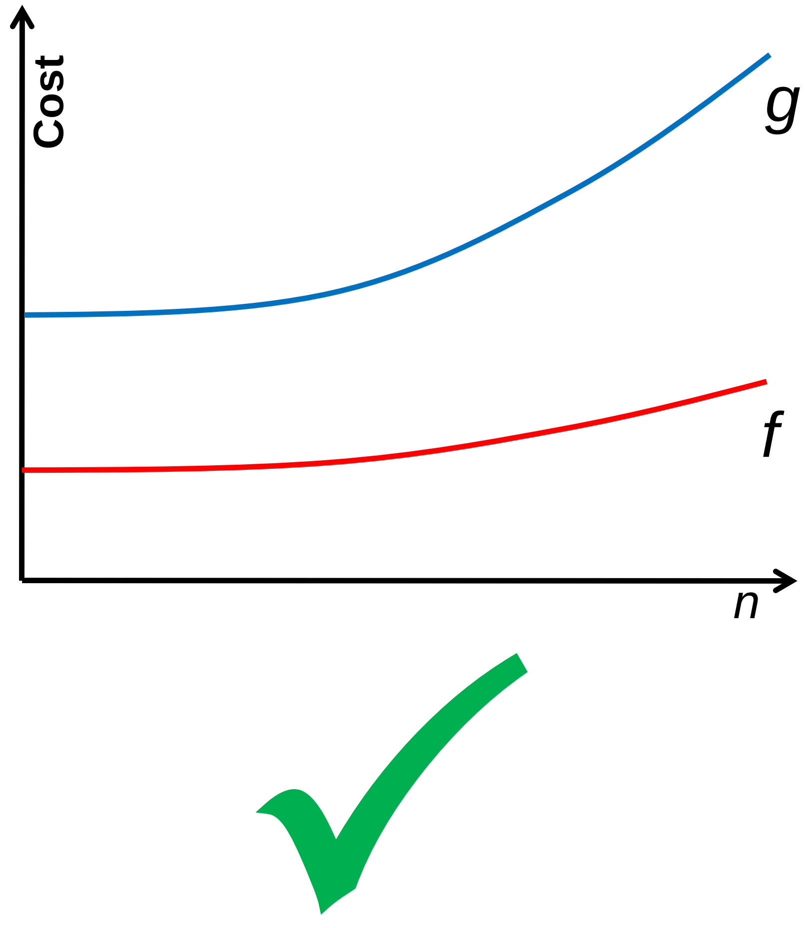
“Is *f* better than *g*?”

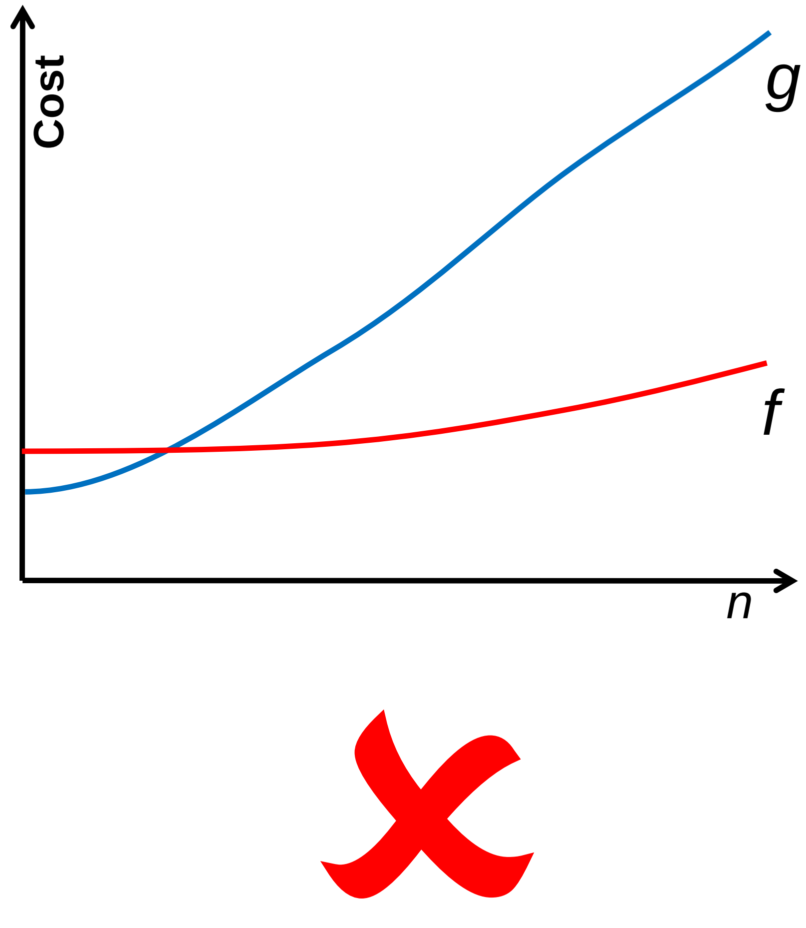
- Attempt #1:

  “*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



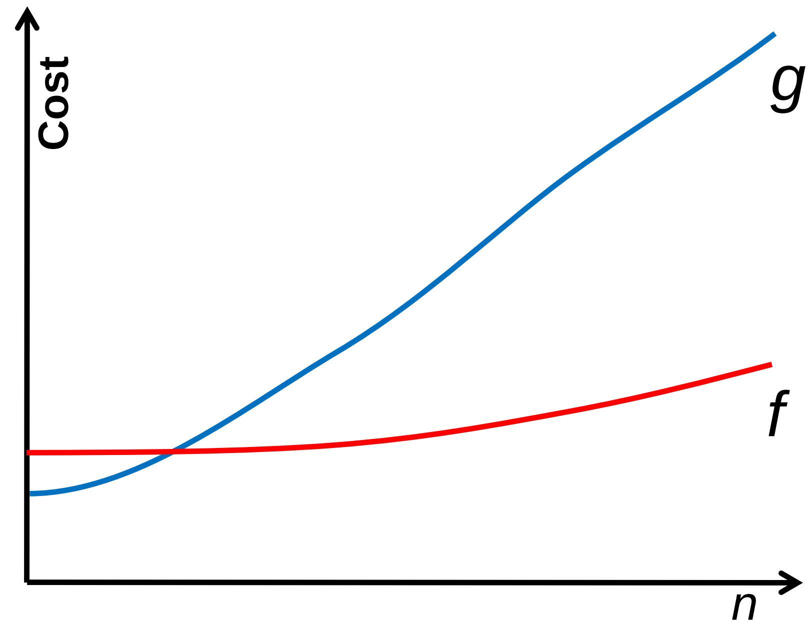


"Is f better than g?"

- Attempt #1:

  "f is better than g" if

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



- But is this useful?
  - This 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

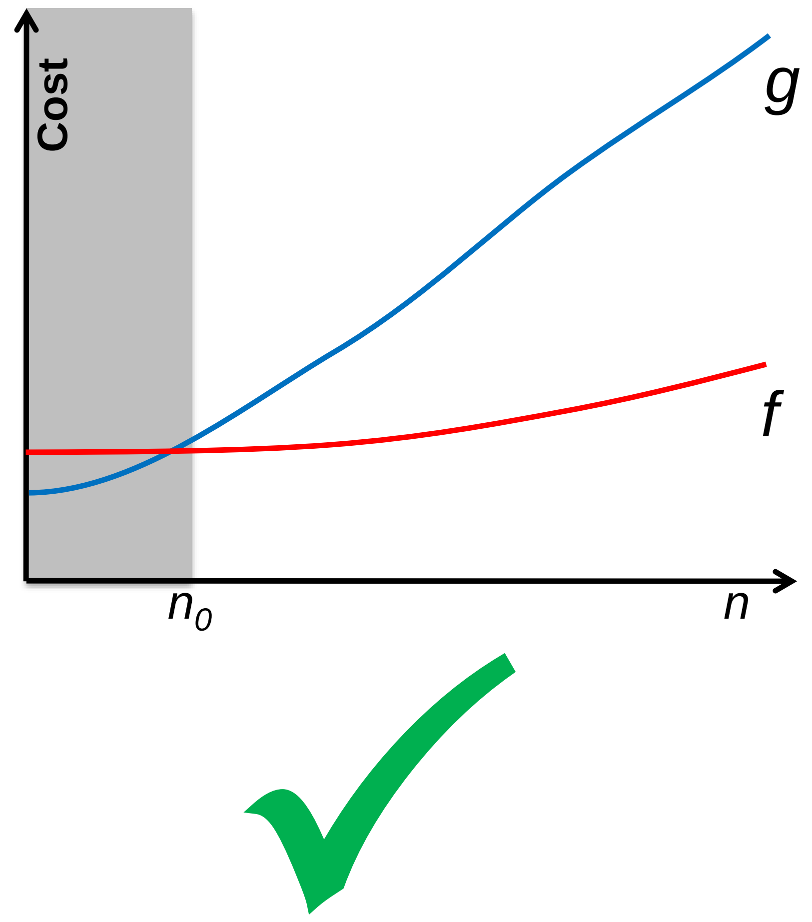
“Is f better than g?”

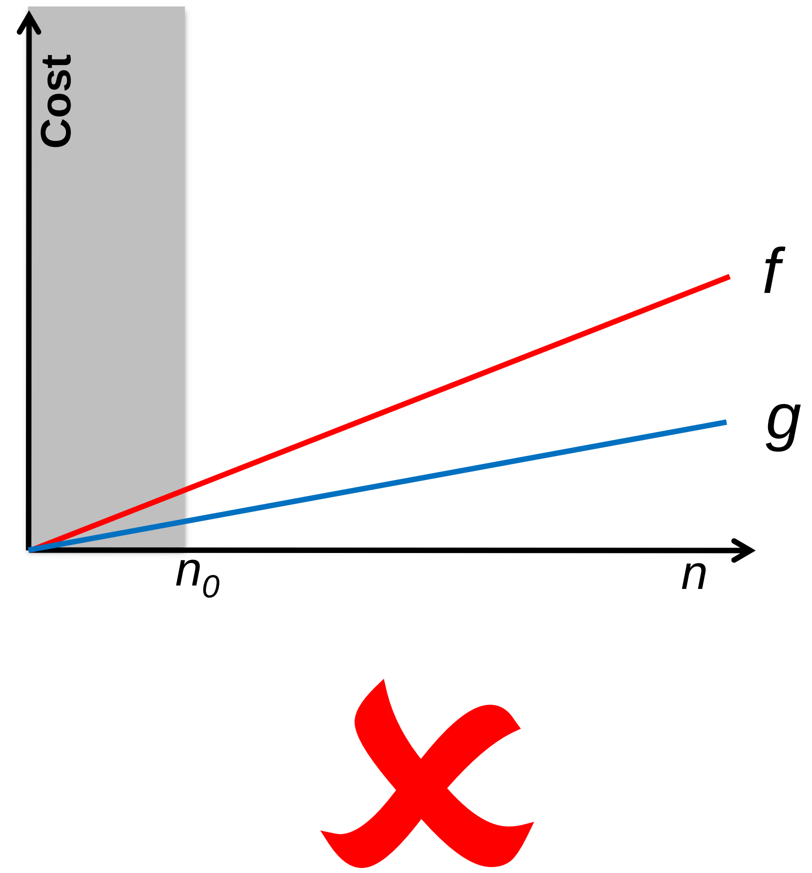
- Attempt #2:

  “f is better than g” if

  there exists a natural number n_0 such that

  for all $n \geq n_0$, $f(n) \leq g(n)$





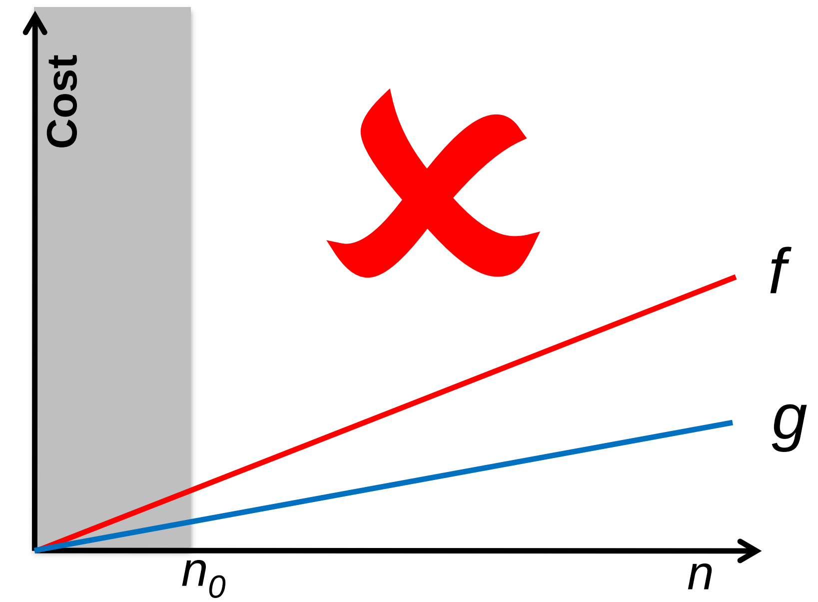
“Is f better than g?”

- Attempt #2:

  “f is better than g” if

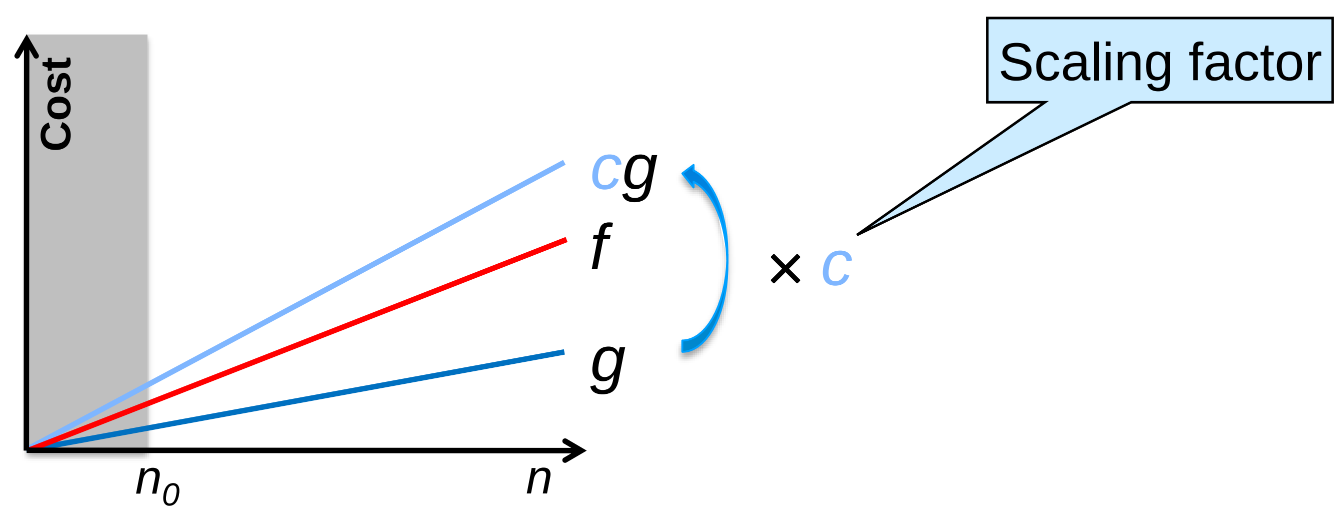
  there exists a natural number n_0 s.t.

  for all $n \geq n_0$, $f(n) \leq g(n)$



- But is this useful?
  - These f and g are both linear functions
    - $f(n) = a_1 n$ and $g(n) = a_2 n$
  - 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



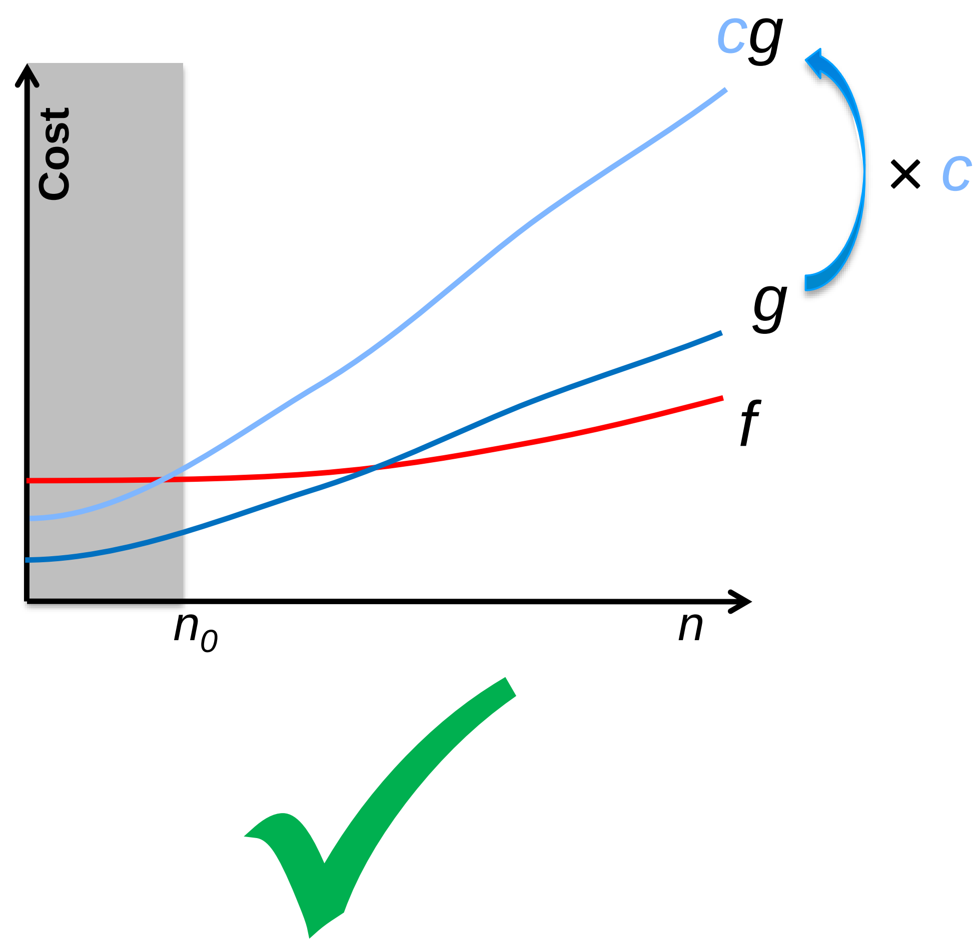
"Is f better than g?"

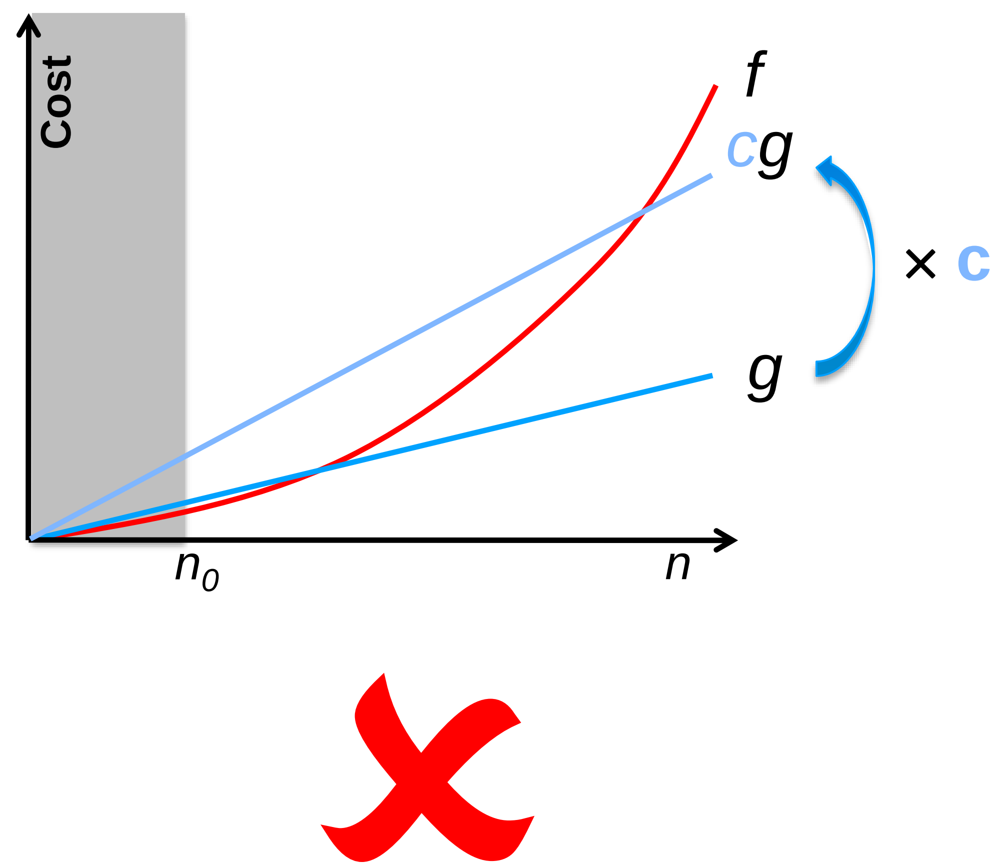
- Final attempt:

  "*f* is better than *g*" if

  there exists a natural number n_0 *and a real $c > 0$* s.t.

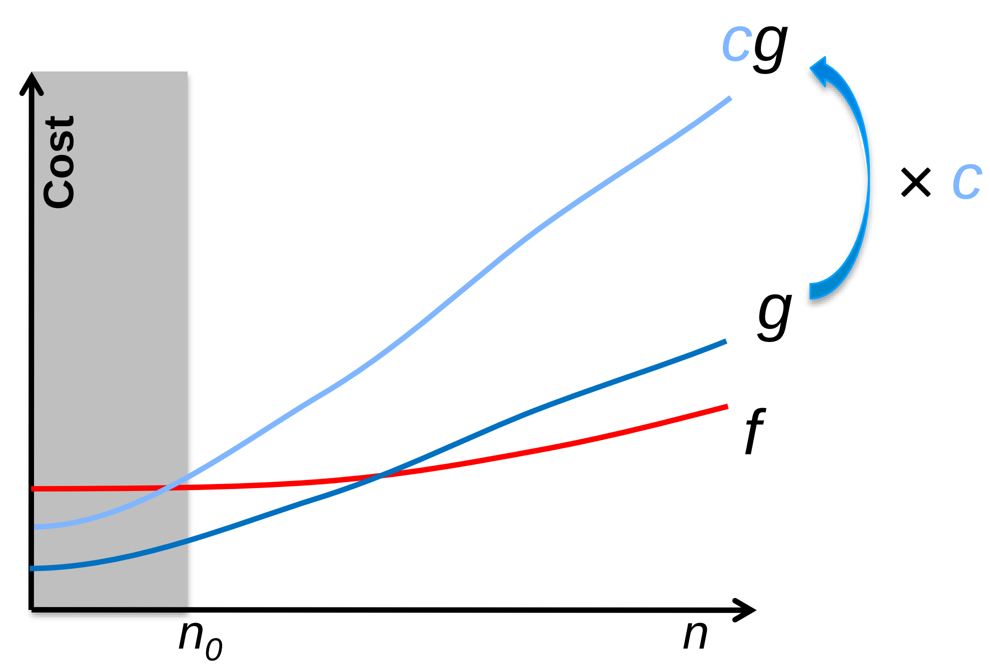
  for all $n \geq n_0$, $f(n) \leq c\, g(n)$





Big O

Big-O



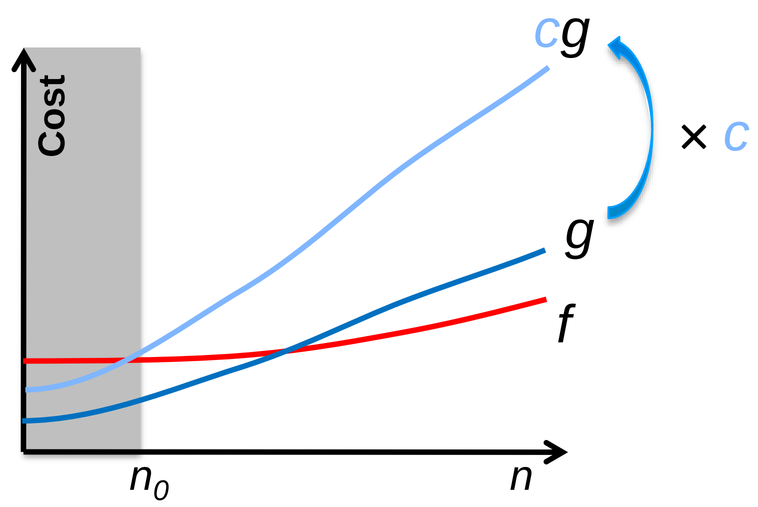
- Rather than "*f* is better than *g*", we say $f \in O(g)$
  - "*f* is in big-O of *g*"

> $f \in O(g)$ 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* s.t. 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



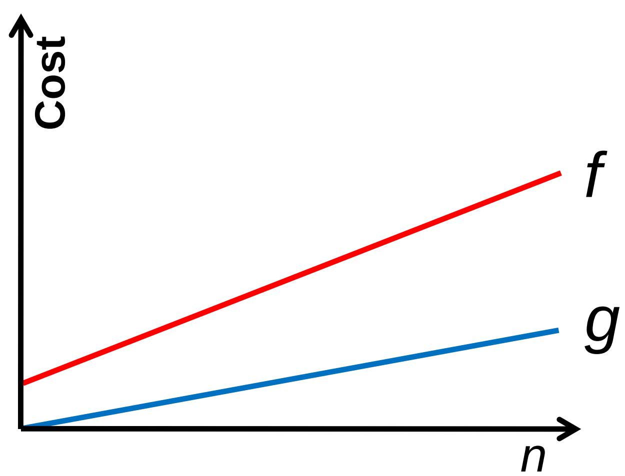
- Given two concrete functions *f* and *g*, how to tell if $f \in O(g)$?
  - do the math
    - find n_0 and c and show that the definition holds
    - or show that the definition cannot hold for any n_0 or c
  - recall what you learned in your calculus classes
    - enough for most of this course

Big-O

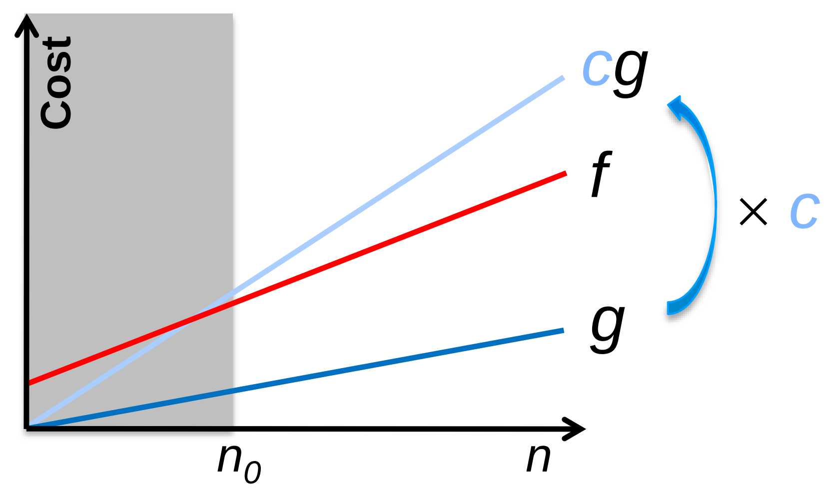
- Is $3n + 2 \in O(n)$?

$f(n) = 3n+2$

$g(n) = n$



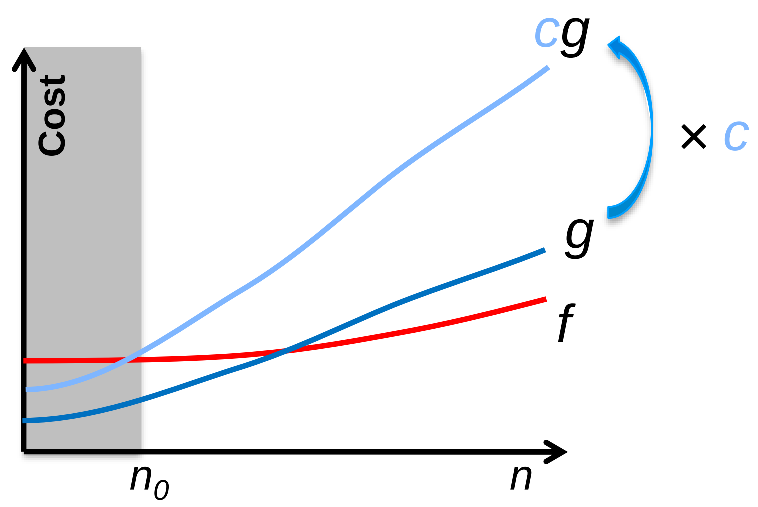
Take $c = 3.5$
and $n_0 = 4$



Any bigger values work too!

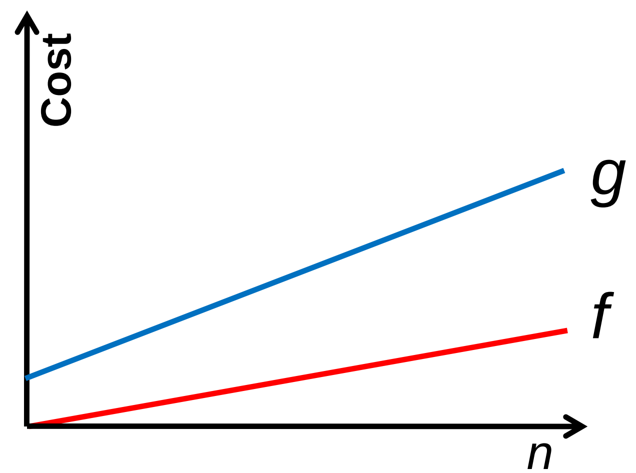
✓

Big-O



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

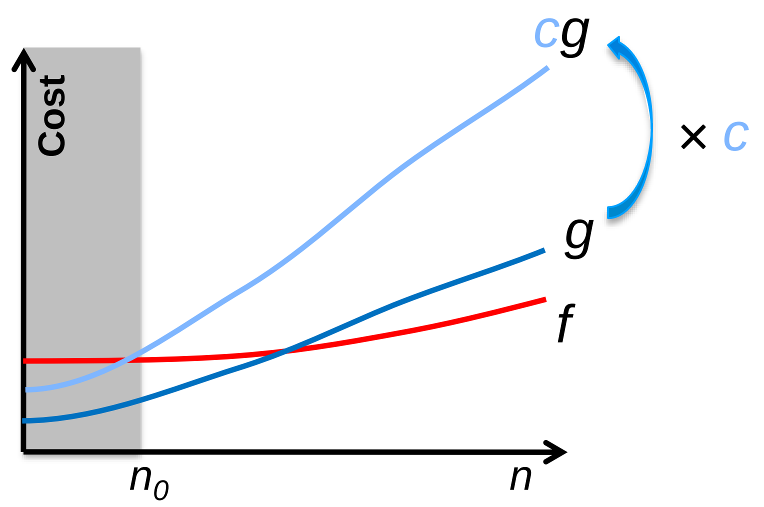
$f(n) = n$ $\quad$ $g(n) = 3n+2$



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

✓

Big-O



- Every **linear cost function** is in *O(n)*
- 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 that is linear by saying that it is in *O(n)*

Big-O

- Similarly, every **quadratic cost function** is in $O(n^2)$
  - $g(n) = n^2$ is the **simplest** quadratic function

- We describe a cost function that is quadratic by saying that it is in $O(n^2)$

- In general, every **polynomial cost function of degree *p*** is in $O(n^p)$
  - $g(n) = n^p$ is the **simplest** polynomial of degree p
    - We can ignore the terms with a smaller exponent

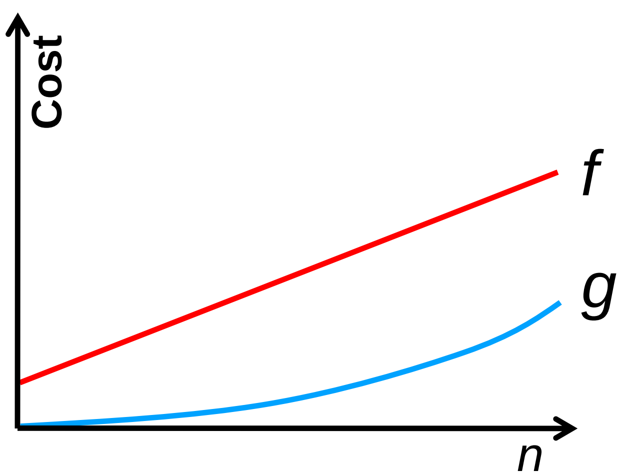
- We describe a cost function that is polynomial of degree p by saying that it is in $O(n^p)$

Big-O

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

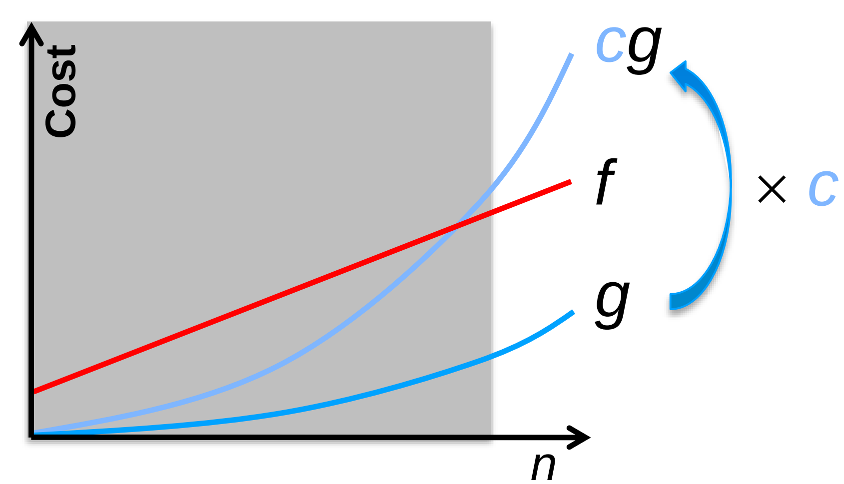
$f(n) = 3n + 2$

$g(n) = n^2$



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

Any bigger values work too!



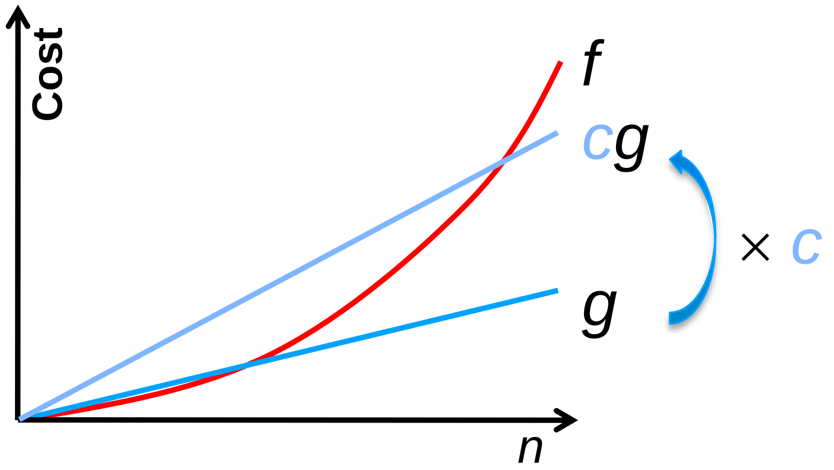
Big-O

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

- Quadratic functions are **not** 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 $O(n^2) \not\subset O(n)$

- $O(n)$ and $O(n^2)$ are called **complexity classes**
  - Simplest and tightest expressions for sets of cost functions

We always write complexity class in **simplest** and **tightest** form

Complexity Classes

Main complexity classes we will use in this course

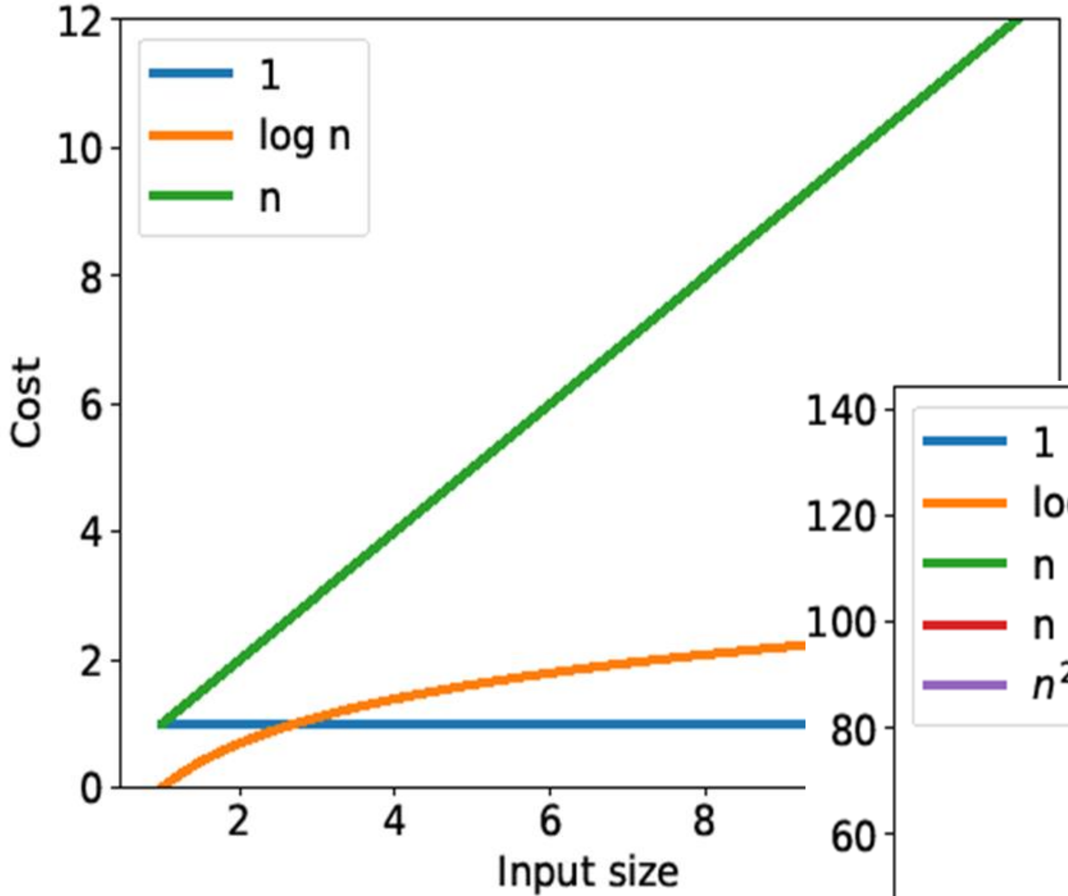
Complexity class	Common name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	(just "*n log n*")
$O(n^2)$	Quadratic
$O(2^n)$	Exponential

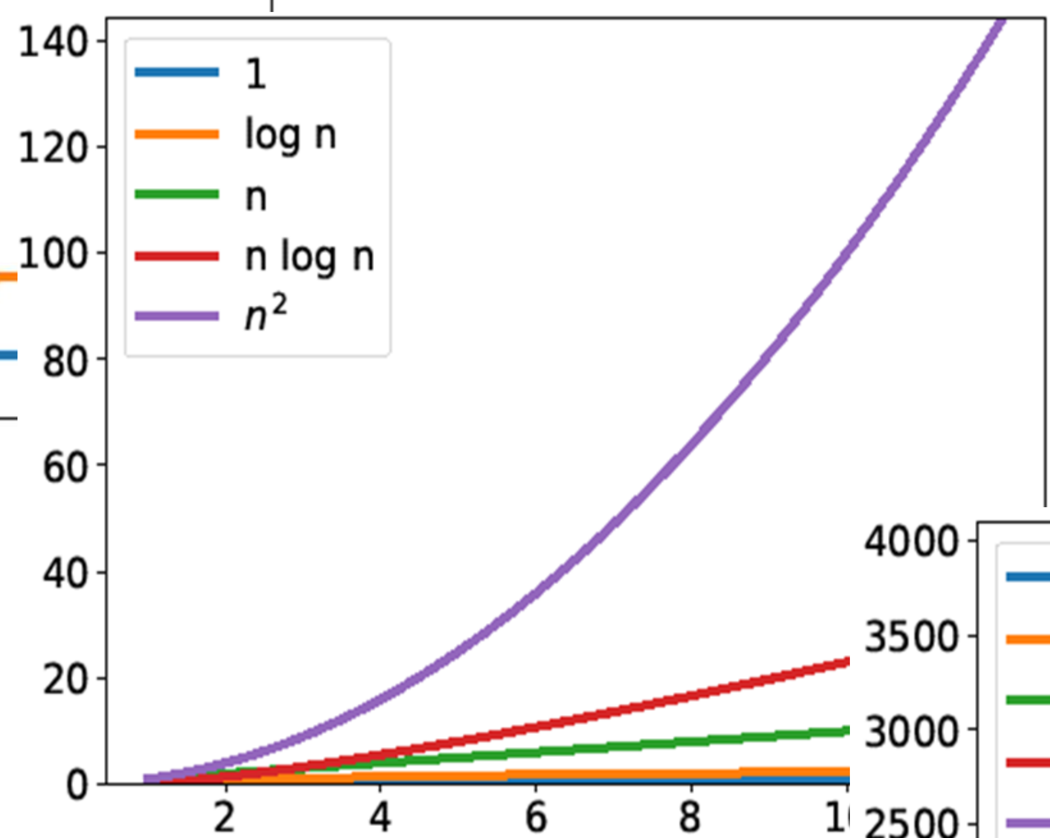
We do not write the base of the logarithm!

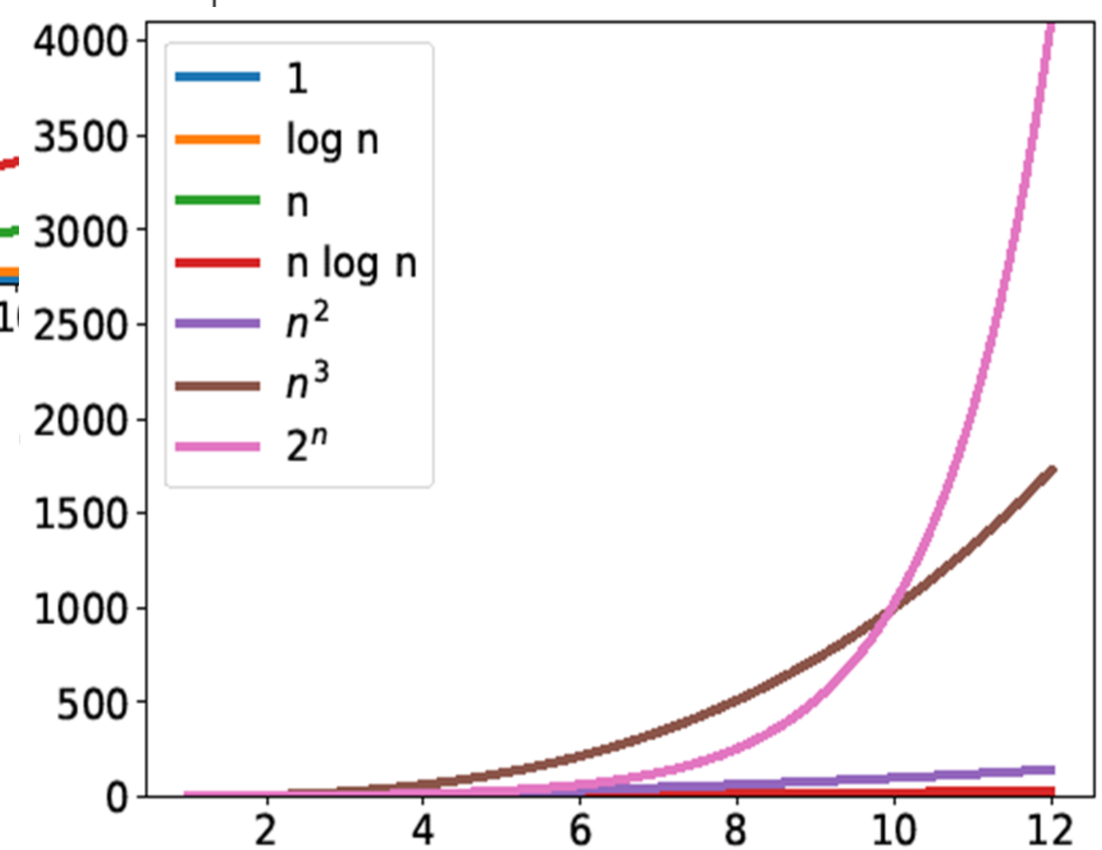
$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(2^n)$$

- There are many more, though

Complexity Classes







Complexity of Linear Search

What do we mean by "How long"?

- $T(n) = an + b$
  - 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 caller *linear* search

What do we mean by “How long”?

- We can often determine the big-O class of a function without writing down its cost function
  - For each operation, note
    - its cost
    - how many times it is executed

    Add up

```
                                                 Cost         Tally
int search(int x, int[] A, int n) {
  // int i = 0 happens before the loop           O(1)         O(1)
  for (int i = 0; i < n; i++)  { // at most      n times
    // i < n happens before each iteration         O(1)       O(n)
    if (A[i] == x)                                 O(1)       O(n)
      return i;                                    O(1)       O(n)
    // i++ happens last in the body                O(1)       O(n)
  }
 return -1;                                      O(1)         O(n)
}
```

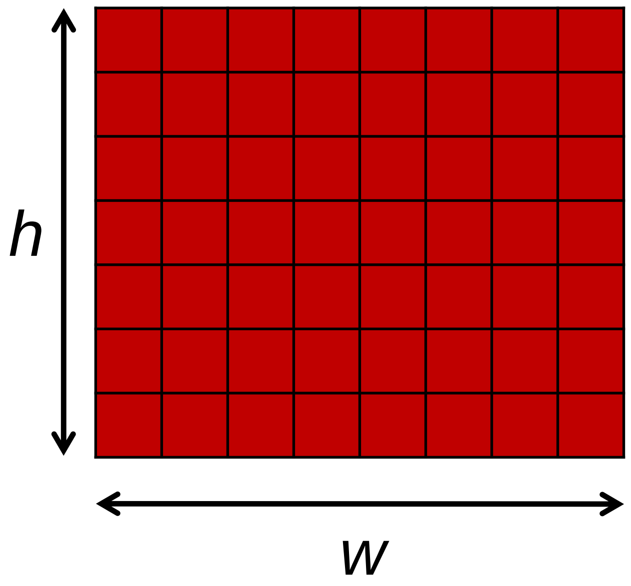
Complexity of search(x, A, n)

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

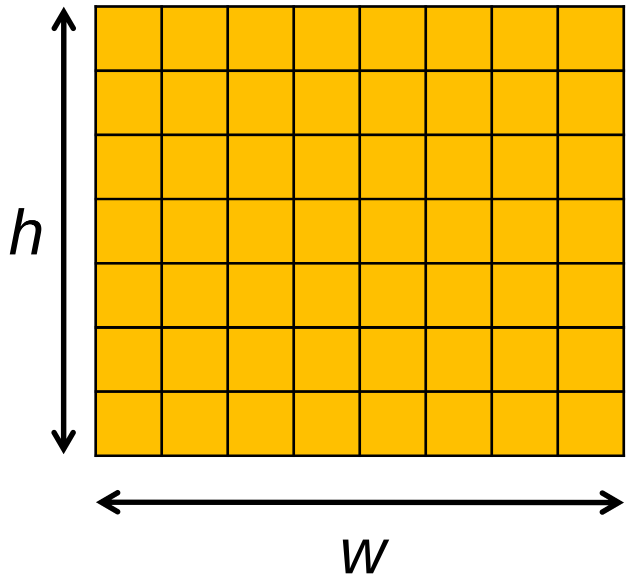
```
int search(int x, int[] A, int k) {
  for (int i = 0; i < k; i++)  {
    if (A[i] == x)  return i;
  }
 return -1;
}
```

Big-O

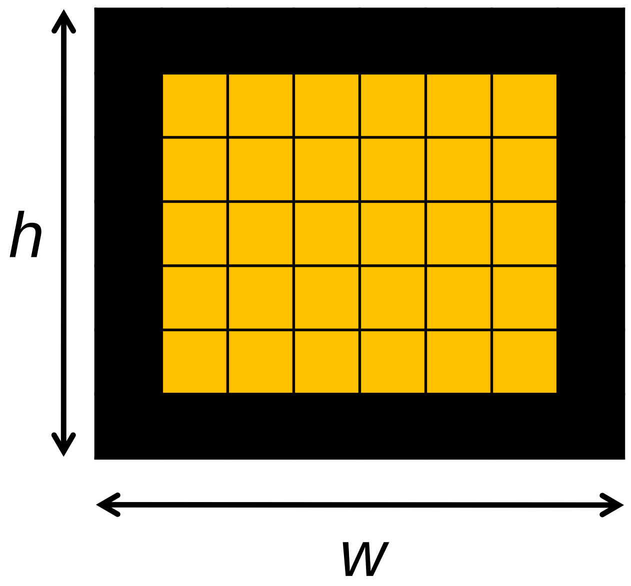


An input may have multiple size parameters

- For example, an image has a width *w* and a height *h*
- Brightening an image: *O(wh)*
  - We change every one of its *w* × *h* pixels



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



Big-O

- Here's the definition of big-O with 2 variables

  $f(x,y) \in O(g(x,y))$ if
  there exist natural numbers x_0, y_0 and a real $c > 0$ s.t.
  for all $x \geq x_0$ and $y \geq y_0$, $f(x,y) \leq c\, g(x,y)$

  - There are other definitions, but this is the one we will use

  - It is asymptotic in both x and y
    - no need for special cases for small values of x or y
      - e.g., if x=0 and y grows very big

- This generalizes to any number of variables

Towards a Better Search

Algorithms vs. Problems

```
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 cost *O(n)*
- 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 for arrays with common characteristics?
  - Say, the array is **sorted**

Searching Sorted Arrays

A is sorted

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

x: 4

	0	1	2	3	4
A:	3	5	7	8	12

5 > 4

```
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;
}
```

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

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

```
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;
}
```

- But that's just *one* algorithm for searching in a sorted array
  - Can we do better?
  - … next lecture …

Searching Sorted Arrays

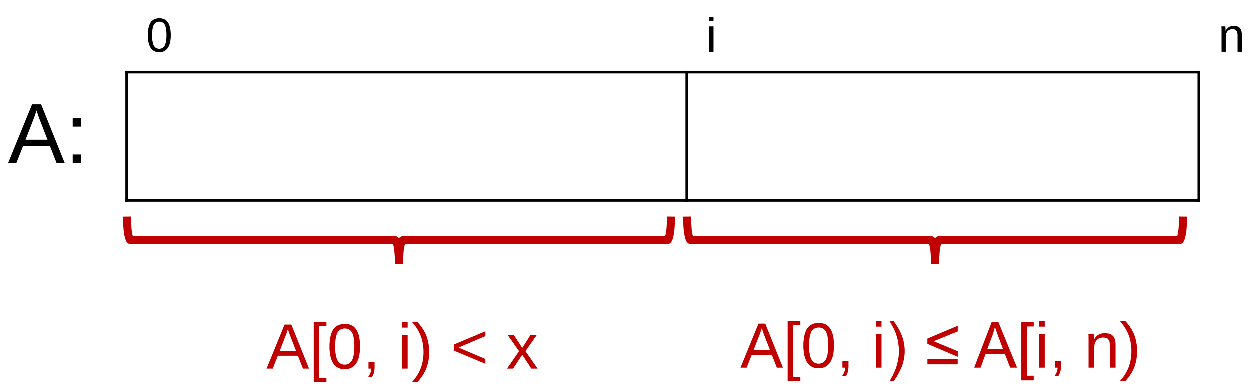
- Is this code safe?
  - Yes, no new array accesses
- Is it correct?
  - Original argument still holds
  - but we have a new place where the function returns
    - Is it correct there?
    - no good argument
    - we need new insight

```
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

```
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;
}
```

- 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
    - because we would have returned otherwise
  - A[0, i) ≤ A[i, n): everything in A[0,i) is ≤ everything in A[i, n)
    - because A is sorted

Reasoning about Array Segments

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

math	code
➢ A[lo, hi) is sorted	is_sorted(A, lo, hi)
➢ $x \in$ A[lo, hi)	is_in(x, A, lo, hi)
➢ x < A[lo, hi)	lt_seg(x, A, lo, hi)
➢ x ≤ A[lo, hi)	le_seg(x, A, lo, hi)
➢ x > A[lo, hi)	gt_seg(x, A, lo, hi)
➢ x ≥ A[lo, hi)	ge_seg(x, A, lo, hi)
➢ $A[lo_1, hi_1) < B[lo_2, hi_2)$	lt_segs(A, lo1, hi1, B, lo2, hi2)
➢ $A[lo_1, hi_1) \leq B[lo_2, hi_2)$	le_segs(A, lo1, hi1, B, lo2, hi2)
➢ $A[lo_1, hi_1) > B[lo_2, hi_2)$	gt_segs(A, lo1, hi1, B, lo2, hi2)
➢ $A[lo_1, hi_1) \geq B[lo_2, hi_2)$	ge_segs(A, lo1, hi1, B, lo2, hi2)

math

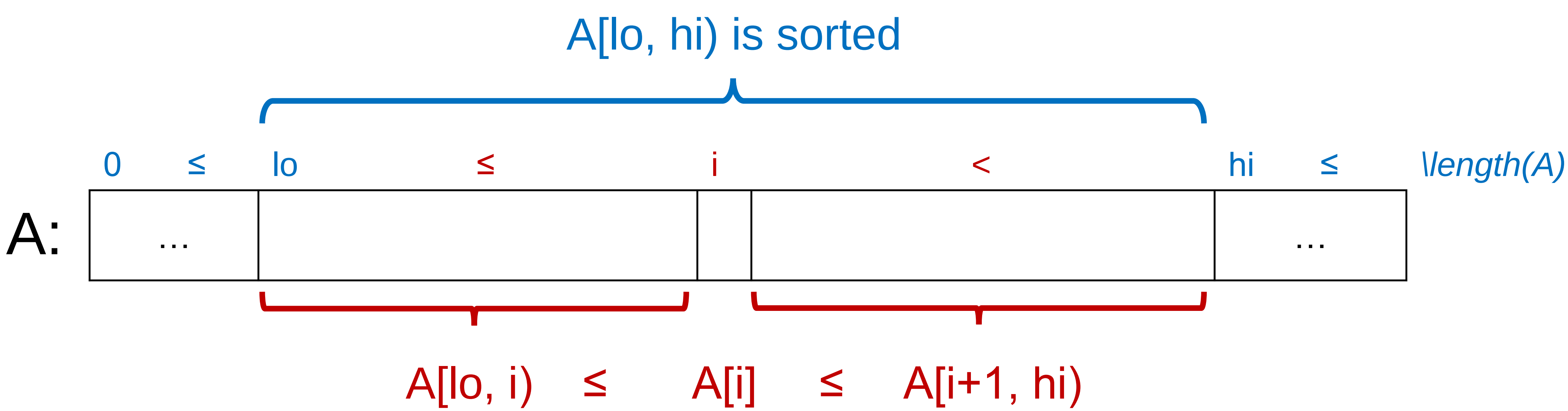
code

Includes contracts

See **arrayutil.c0** file

Reasoning about Sorted Arrays

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



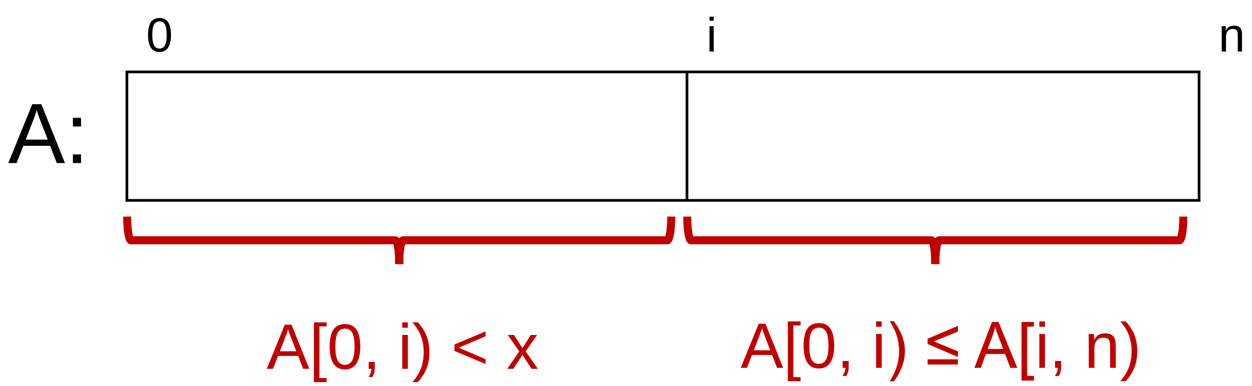
- for every element A[i]
  - lo ≤ i < hi

    A[lo, hi) can't be empty
  - A[lo, i) ≤ A[i] ≤ A[i+1, hi)
  - A[lo, i) and A[i+1, hi) are sorted

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 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;
}
```

- 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 ∉ A[0, i)
  - le_segs(A, 0, i, A, i, n): that's A[0, i) ≤ A[i, n)

Searching Sorted Arrays

```
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. }
```

Is this code correct?

- **To show**: !is_in(x, A, 0, n)
  (assuming invariants are valid)

A. A[0, i) < x	by line 10 (LI 2)
B. x ∉ 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 ∉A[0, n)	by math on B, D

Searching Sorted Arrays

```
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. }
```

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. i' = i+1 — by line 8

B. x > A[0, i) — by assumption

C. x > A[i] — by line 15 (math on lines 13 and 14)

D. x > A[0, i+1) — by math on B and C

E. x > A[0, i') — by math on A and D

Searching Sorted Arrays

```
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. }
```

A[0,i) ≤ A[i,n) is a **valid** loop invariant

INIT

- **To show**: A[0, i) ≤ A[i,n) initially

A. i = 0 — by line 7

B. A[0, 0) ≤ A[i,n) — by definition of le_segs

- A[0,0) is the empty array segment
  - All the (**zero**) things in A[0,0) are ≤ everything in A[i,n)

PRES

- **To show**: if A[0, i) ≤ A[i, n), then A[0, i’) ≤ A[i’, n)

A. i’ = i+1 — by line 8

B. A[0, i) ≤ A[i, n) — by assumption

C. A[0, n) sorted — by line 3 (precondition)

D. A[0, i+1) ≤ A[i+1, n) — by math on C

E. A[0, i’) ≤ A[i’, n) — by math on A and D

We actually don’t need this loop invariant to prove correctness

Selection Sort

Sorting an Array

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

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

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

- There are many algorithms to sort arrays

Selection Sort

- Find element that shall go in A[0]
  - Smallest element in A[0, n)

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

- Swap it with A[0]

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

- Find element that shall go in A[1]
  - Smallest element in A[1, n)

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

- Swap it with A[1]

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

- … carry on …

- Stop when A is entirely sorted

	0	1	2	3	4	5	6	7
A:	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[\result] ≤ 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 modifies
input array

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 does not return anything either

but it modifies the input array

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);
  }
}
```

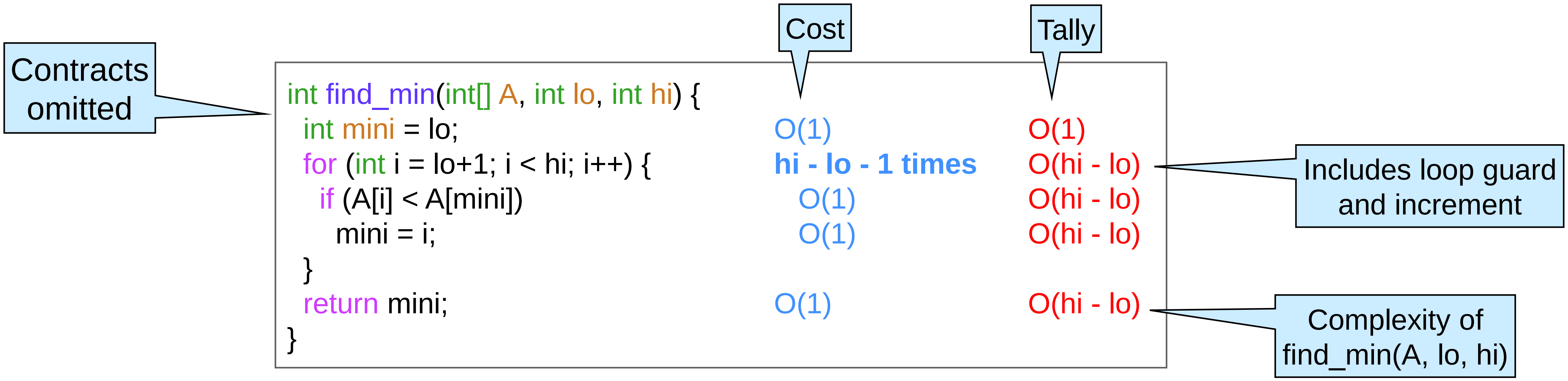
for every index *i* from *lo* to *hi*

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

swap it with *A[i]*

Cost of Selection Sort

- find_min(A, lo, hi)
  - ➢ finds the minimum of an array segment A[lo, hi)
  - ➢ and returns its index
  - ○ it scans the entire segment once



Cost: ***O(hi – lo)***

- ○ note that it makes *hi - lo - 1* comparisons
  - ➢ the number of comparisons is a convenient proxy for our unit of cost

Cost of Selection Sort

- swap(A, i, j)
  - simply swaps values at two indices:

Contracts omitted

Code	Cost	Tally
`void swap(int[] A, int i, int j) {`		
`  int tmp = A[i];`	O(1)	O(1)
`  A[i] = A[j];`	O(1)	O(1)
`  A[j] = tmp;`	O(1)	O(1)
`}`		

Complexity of swap(A, i, j)

Cost: ***O(1)***

Selection Sort

- sort(A, lo, hi)

Cost

Tally

Contracts omitted

```
void sort(int[] A, int lo, int hi) {
  for (int i = lo; i < hi; i++)              hi - lo times     O(hi - lo)
  {
    int min = find_min(A, i, hi);             O(hi - i)         O(∑_{i=lo}^{hi} (hi – i))
    swap(A, i, min);                          O(1)              O(∑_{i=lo}^{hi} (hi – i))
  }
}
```

Includes loop guard and increment

Complexity of sort(A, lo, hi)

But not in simplest and tightest form!

- Let n = hi - lo
  - the length of the segment A[lo,hi)

  then $\sum_{i=lo}^{hi} (hi - i) = \sum_{j=0}^{n} j$

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

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

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 the 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 in $\boldsymbol{O(n^2)}$

That's $O((hi-lo)^2)$ in terms of lo and hi

Selection Sort

```
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);
  }
}
```

```
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 ≤ i < hi ≤ \length(A)

A. hi ≤ \length(A) by line 2

B. i < hi by line 5

C. 0 ≤ i *oops! we need the usual loop invariant*

- //@loop_invariant lo <= i; then

a. 0 ≤ lo by line 2

b. lo ≤ i by new LI

c. 0 ≤ 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. }
```

Is this Code Safe?

```
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);
```

- swap(A, i, min)
  - **To show**: 0 ≤ min < \length(A)

A. 0 ≤ i by lines 2 and 6
B. i ≤ min by postconditions of find_min
C. min < hi by postconditions of find_min
D. hi ≤ \length(A) by line 2

(A, B) 0 ≤ min by math

(C, D) min < \length(A) 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

- **To show**: 0 ≤ i
  && i < \length(A)

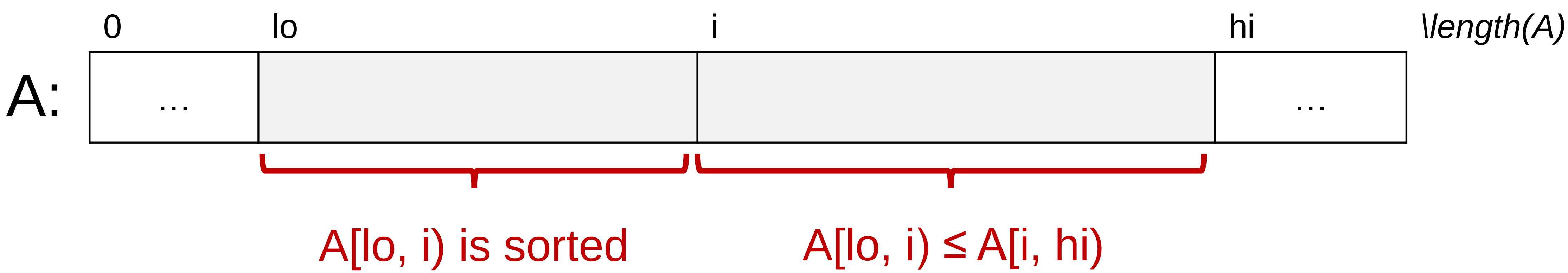
(just proved for find_min)

✓

Is this Code Correct?

```
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);
  }
}
```

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



- Candidate loop invariants
  - lo <= i && i <= hi
  - is_sorted(A, lo, i)
  - le_segs(A, lo, i, A, i, hi)

Selection Sort

- Resulting code

```
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 && i <= hi;
  //@loop_invariant is_sorted(A, lo, i);
  //@loop_invariant le_segs(A, lo, i, A, i, hi);
  {
    int min = find_min(A, i, hi);
    swap(A, i, min);
  }
}
```

Added

  - We will need to prove that the added invariants are valid

Correctness

```
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;   )
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.
11.
12.   }
13. }
```

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

A. $i \geq hi$ — by line 5 (negation of loop guard)

B. $i \leq hi$ — 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

```
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. }
```

Are the loop invariants valid?

INIT

- **To show**: lo ≤ lo — by math
- **To show**: lo ≤ hi — by line 2 (preconditions)
- **To show**: A[lo, lo) sorted — by math (empty interval)
- **To show**: A[lo, lo) ≤ A[lo, hi) — by math (empty interval)

PRES

- **To show**: if lo ≤ i ≤ hi, then lo ≤ i’ ≤ hi

Proof left as exercise

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. }
```

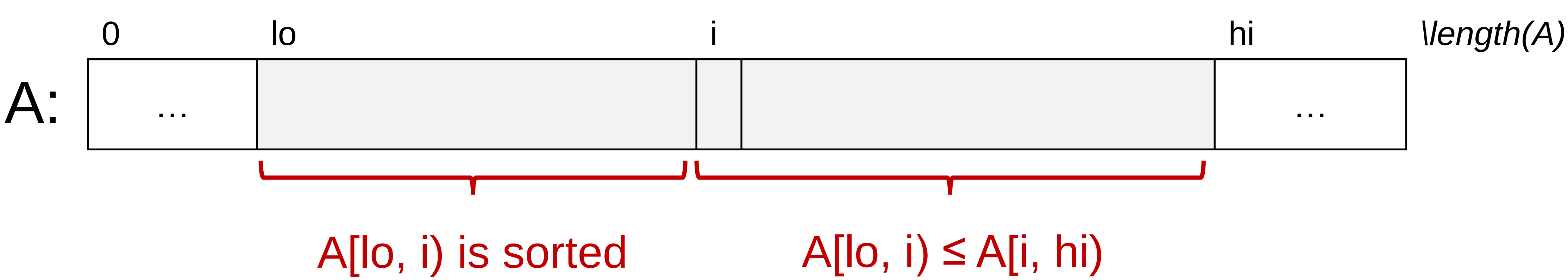
Are the loop invariants valid?

PRES

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

A.	i' = i+1	by line 5 (step)
B.	A[lo, i) is sorted	assumption
C.	A[lo, i) ≤ A[i, hi)	by line 8 (LI 3)
D.	A[lo, i) ≤ A[i]	by math on C and line 5 (loop guard)
E.	A[lo, i') is sorted	by math on A and D

This is where we need LI 3!



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. }
```

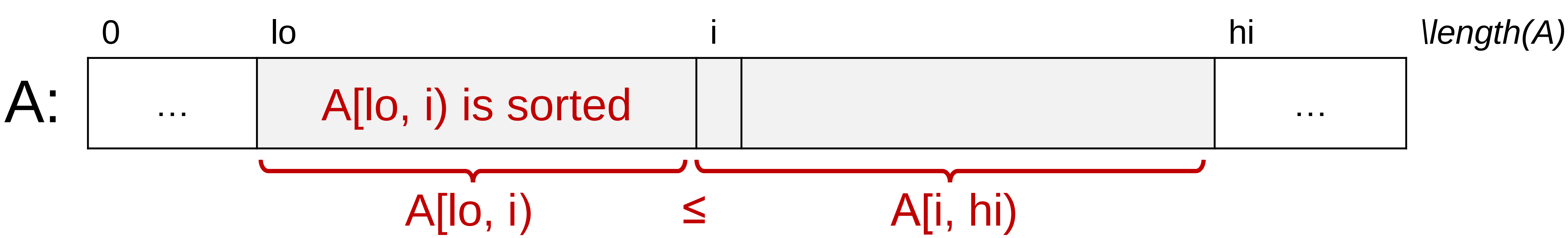
Are the loop invariants valid?

PRES

- **To show**: if A[lo, i) ≤ A[i, hi),
  then A[lo, i') ≤ A[i', hi)

A.	i' = i+1	by line 5
B.	A[lo, i) ≤ A[i, hi)	assumption
C.	A[min] ≤ A[i, hi)	by postcondition of find_min
D.	A[i] ≤ A[i, hi)	**after** swap by definition (in comment)
E.	A[i] ≤ A[i+1, hi)	by math
F.	A[lo, i) ≤ A[i]	by math on B and definition of swap
G.	A[lo, i') ≤ A[i', hi)	by math on A, F and E

after **swap**



Selection Sort

```
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 && i <= hi;
  //@loop_invariant is_sorted(A, lo, i);
  //@loop_invariant le_segs(A, lo, i, A, i, hi);
  {
    int min = find_min(A, i, hi);
    swap(A, i, min);
  }
}
```

- We have proved it correct ✓