CS11212 - **Spring** 2022

Data Structures & Introduction to **Algorithms**

Analysis of Algorithms

part 1: Counting Operations

Ibrahim Albluwi

What is an Algorithm?

A sequence of steps to solve a problem.

What is an Algorithm?

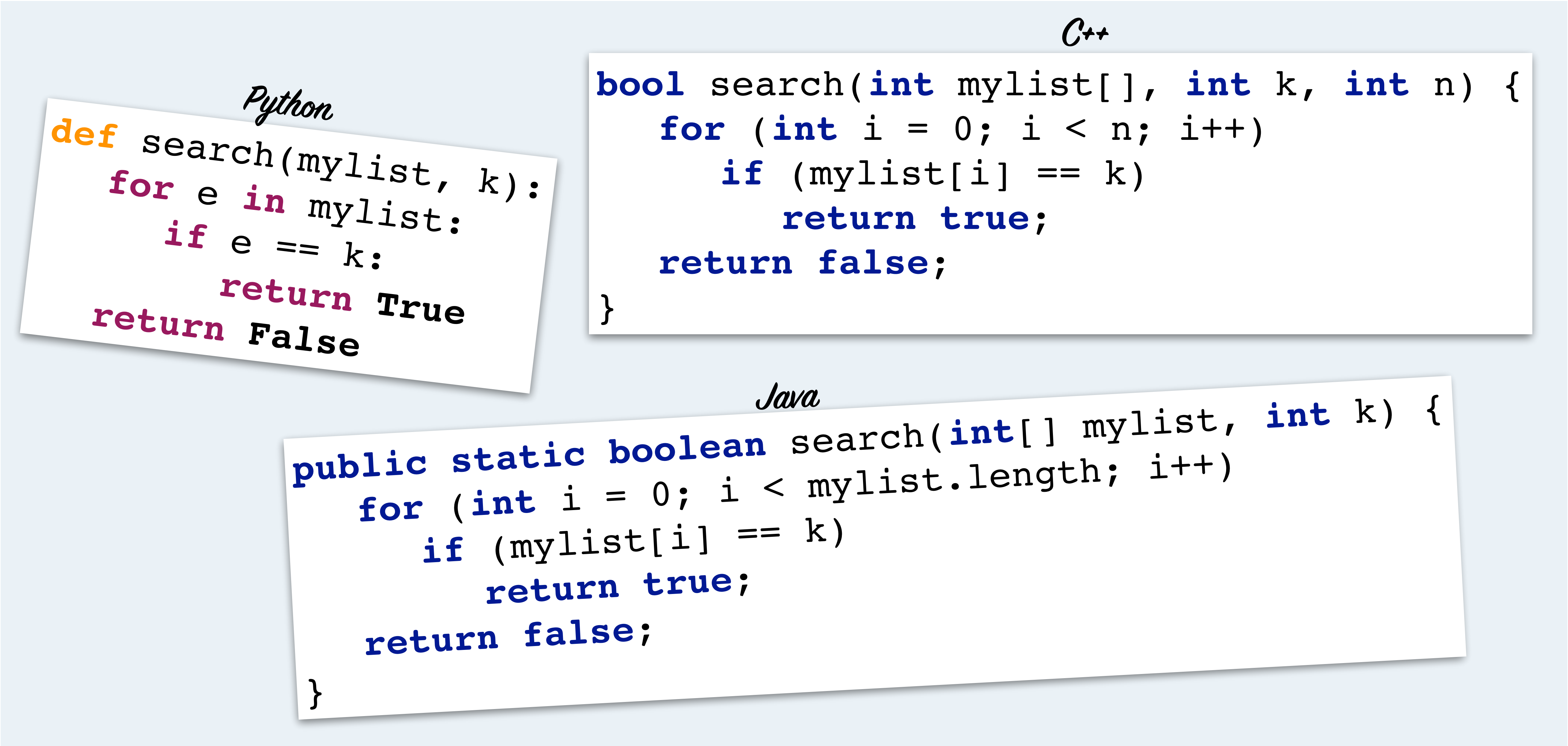
A sequence of steps to solve a problem.

Example. Sequential Search is an algorithm for searching for an element in an array, which goes through all the elements one-by-one.

What is an Algorithm?

A sequence of steps to solve a problem.

Example. Sequential Search is an algorithm for searching for an element in an array, which goes through all the elements one-by-one.



Python

```python
def search(mylist, k):
    for e in mylist:
        if e == k:
            return True
    return False
```

C++

```cpp
bool search(int mylist[], int k, int n) {
  for (int i = 0; i < n; i++)
    if (mylist[i] == k)
      return true;
  return false;
}
```

Java

```java
public static boolean search(int[] mylist, int k) {
  for (int i = 0; i < mylist.length; i++)
    if (mylist[i] == k)
      return true;
  return false;
}
```

The same algorithm implemented in different languages

Comparing Algorithms

Q. Given two algorithms **A** and **B**, how do we know which is faster?

Comparing Algorithms

Q. Given two algorithms **A** and **B**, how do we know which is faster?

A. Implement and run both and compare the time each takes!

Experimental Analysis

To compare two algorithms, we can implement them, run them and compare their running times.

Challenges.

Experimental Analysis

To compare two algorithms, we can implement them, run them and compare their running times.

Challenges.

- The running time of a program is *hardware and software dependent*.
  We need to run both algorithms on the same machine (or on machines with the same specs), using the same programming language, the same compiler, etc.

Experimental Analysis

To compare two algorithms, we can implement them, run them and compare their running times.

Challenges.

- The running time of a program is *hardware and software dependent*.
  We need to run both algorithms on the same machine (or on machines with the same specs), using the same programming language, the same compiler, etc.

- The running time of a program depends on the *input size* and on the *input type*.
  We need to run the programs as many times as needed to cover all possible input sizes and types that might affect the behavior of the programs.

Experimental Analysis

To compare two algorithms, we can implement them, run them and compare their running times.

Challenges.

- The running time of a program is *hardware and software dependent*.
  We need to run both algorithms on the same machine (or on machines with the same specs), using the same programming language, the same compiler, etc.

- The running time of a program depends on the *input size* and on the *input type*.
  We need to run the programs as many times as needed to cover all possible input sizes and types that might affect the behavior of the programs.

- Running the programs *might take a long time*!
  Takes as long as the fastest of the two programs requires.

Which program runs faster?

Program A:

```
x = 1;
y = 2;
sum = x + y;
```

Program B:

```
x = 1;
y = 2;
z = 3;
k = 4;
m = 5;
n = 6;
x = x + y;
x = x + z;
x = x + k;
x = x + m;
X = x + n;
```

Which program runs faster?

Program A:

```
x = 1;
y = 2;
sum = x + y;
```

4 operations

Program B:

```
x = 1;
y = 2;
z = 3;
k = 4;
m = 5;
n = 6;
x = x + y;
x = x + z;
x = x + k;
x = x + m;
X = x + n;
```

16 operations

Theoretical Analysis

To compare two algorithms, *count* the number of operations each one performs.

Theoretical Analysis

To compare two algorithms, *count* the number of operations each one performs.

Problem. Sometimes it is very difficult to count the number of operations or come up with a model for that.

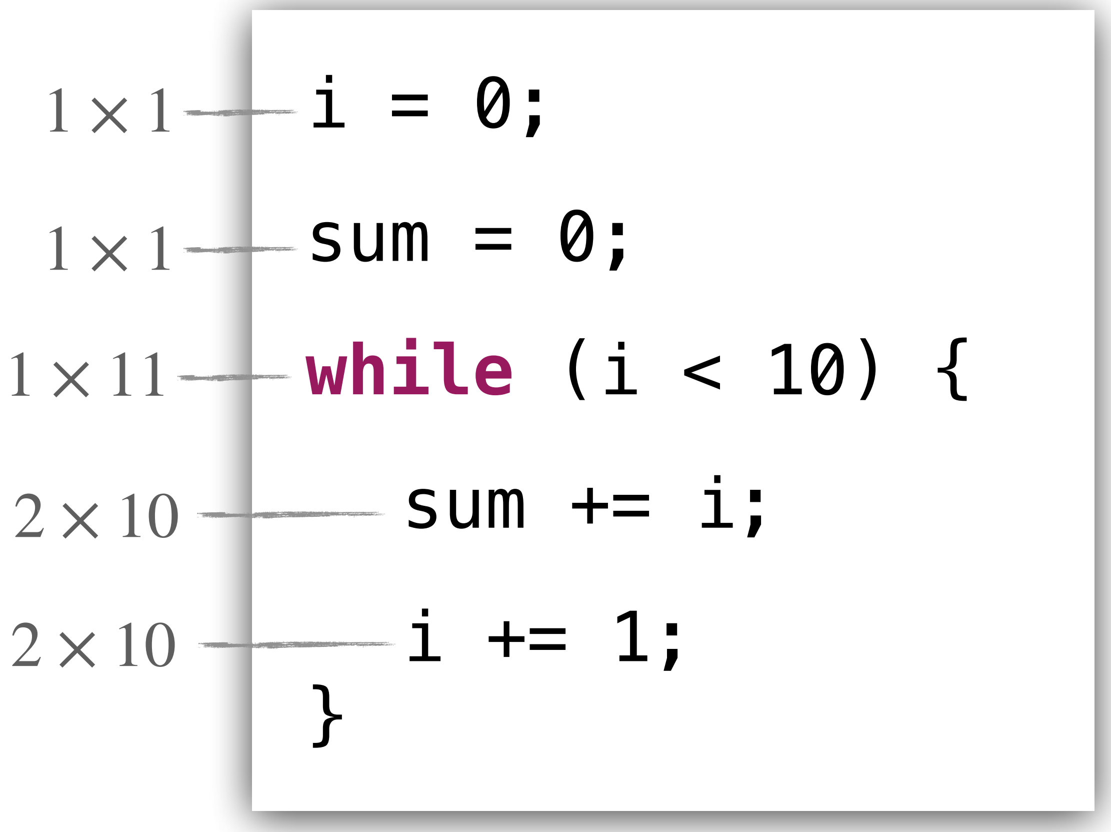
Solution. Perform experimental analysis!

How Many Operations?

```
i = 0;
sum = 0;
while (i < 10) {
   sum += i;
   i += 1;
}
```

```
i = 0;
sum = 0;
while (i < 20) {
   sum += i;
   i += 1;
}
```

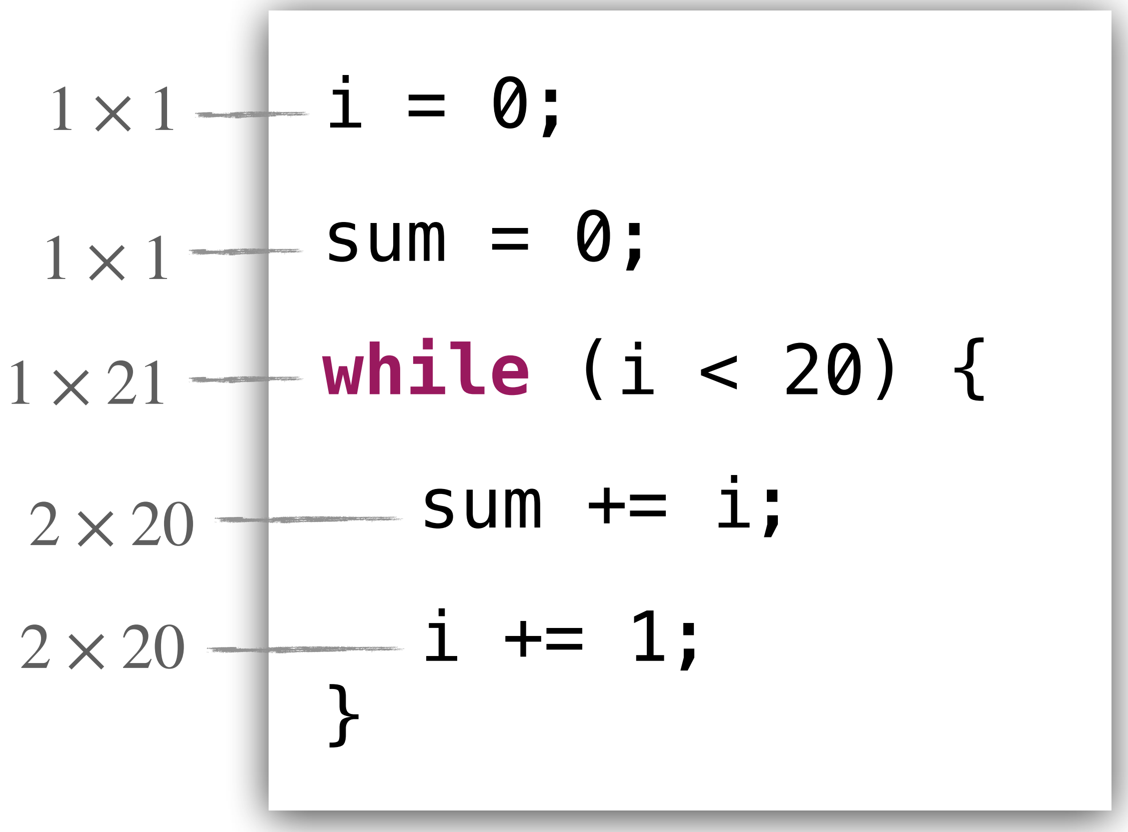
How Many Operations?



```
1 × 1  ——  i = 0;
1 × 1  ——  sum = 0;
1 × 11 ——  while (i < 10) {
2 × 10 ——      sum += i;
2 × 10 ——      i += 1;
           }
```

$2 + (1 \times 11) + (4 \times 10) =$

53 operations



```
1 × 1  ——  i = 0;
1 × 1  ——  sum = 0;
1 × 21 ——  while (i < 20) {
2 × 20 ——      sum += i;
2 × 20 ——      i += 1;
           }
```

$2 + (1 \times 21) + (4 \times 20) =$

103 operations

For simplicity, we will say:

- the left code performed the `sum += i` operation 10 times.
- the right code performed the `sum += i` operation 20 times.

We will always pick a certain operation to be the basis for our cost model.

How Many Operations?

How many times does `sum += i` get executed?

```
i = 0;
sum = 0;
while (i<5) {
   sum += i;
   i += 1;
}
```

5 times

```
i = 10;
sum = 0;
while (i>0) {
   sum += i;
   i -= 1;
}
```

10 times

```
i = 0;
sum = 0;
while (i<n) {
   sum += i;
   i += 1;
}
```

n times

Note: In all of the examples, *n* is assumed to be positive

How Many Operations?

How many times does `op()` get called?

```
i = 100;
while (i<n) {
   op();
   i += 1;
}
```

$n - 100$ times

for all $n > 100$ and 0 otherwise

```
i = 0;
while (i<n) {
   op();
   i += 5;
}
```

$\lceil n \;/\; 5 \rceil$ times

```
i = 100;
while (i<n) {
   op();
   i += 5;
}
```

$\lceil (n - 100) \;/\; 5 \rceil$ times

for all $n > 100$ and 0 otherwise

How Many Operations?

How many times does `op()` get called?

```
for (int i=0; i<n; i++)
   op();
```

n

```
for (int i=0; i<n; i+=5)
   op();
```

$\lceil n/5 \rceil$

How Many Operations?

How many times does `op()` get called?

```
for (int i=0; i<n; i++) {
   op();
   op();
}
```

$2n$

```
for (int i=0; i<n; i+=3) {
   op();
   op();
   op();
}
```

n

assuming n is a multiple of 3. If not, then the answer is: $\lceil n/3 \rceil \times 3$

How Many Operations?

How many times does `op()` get called?

```
for (int i=0; i<n; i++) {
   for (int j=0; j<n; j++)
      op();
}
```

n^2

```
for (int i=0; i<n; i++)
      op();
for (int j=0; j<n; j++)
      op();
```

$2n$

How Many Operations?

How many times does `op()` get called? (assuming n is a multiple of 2)

```
for (int i = 10; i < n; i++) {
   for (int j = 5; j < n; j += 2)
      op();
}
```

$$(n - 10) \times \frac{1}{2}(n - 5)$$

for all $n > 10$, 0 otherwise

How Many Operations?

How many times does `op()` get called? (assuming n is a multiple of 2)

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j += 2)
     op();

  for (int j = 0; j < n; j += 2)
     op();
}
```

$$n \times (\tfrac{1}{2}n \ + \ \tfrac{1}{2}n) = n^2$$

If n is not a multiple of 2, the answer is: $n \times (\lceil \tfrac{1}{2}n \rceil \ + \ \lceil \tfrac{1}{2}n \rceil)$

How Many Operations?

How many times does `op()` get called? (assuming n is a multiple of 2)

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j += 2)
    for (int k = 10; k < n; k++)
      op();
```

$$n \times \frac{1}{2}n \times (n - 10) = \frac{1}{2}n^3 - 5n^2$$

for all $n > 10$, 0 otherwise

How Many Operations?

How many times does `op()` get called? (assuming *n* is a multiple of 2)

```
for (int i = 0; i < n*n; i++)
    op();

for (int i = 0; i < n; i += 2)
  for (int j = 0; j < n; j += 2)
    op();
```

$$n^2 + (\tfrac{1}{2}n \times \tfrac{1}{2}n)$$

$$= n^2 + \tfrac{1}{4}n^2$$

$$= \tfrac{5}{4}n^2$$

How Many Operations?

How many times does `op()` get called?

```
for (int i = 0; i < n; i++)
  for (int j = i; j < i + 7; j++)
    op();
```

$7n$

(the inner loop always repeats 7 times, regardless of what the value of *i* is)

```
for (int i = 0; i*i < n; i++)
    op();
```

$\sqrt{n}$

(the loop stops when $i^2 = n$ i.e. when $i = \sqrt{n}$)

How Many Operations?

How many times does `op()` get called? (assuming n is a power of 2)

```
for (int i = 1; i <= n; i *= 2)
    op();
```

$$i = 1,\quad 2,\quad 4,\quad 8,\quad \ldots,\quad \tfrac{1}{2}n,\quad n$$

$$= 2^0,\quad 2^1,\quad 2^2,\quad 2^3,\quad \ldots,\quad 2^{k-1},\quad 2^k$$

These are $k+1$ steps, where $2^k = n$ i.e. $k = \log_2(n)$

Total number of times `op()` is called = $\log_2(n) + 1$

```
for (int i = n; i >= 1; i /= 2)
    op();
```

$$i = n,\quad \tfrac{1}{2}n,\quad \tfrac{1}{4}n,\quad \ldots, 8,\ 4,\ 2,\ 1$$

$$= 2^k,\ 2^{k-1}, 2^{k-2},\ \ldots, 2^3, 2^2, 2^1, 2^0$$

These are $k+1$ steps, where $2^k = n$ i.e. $k = \log_2(n)$

Total number of times `op()` is called = $\log_2(n) + 1$

How Many Operations?

How many times does `op()` get called? (assuming n is a power of 3)

```
for (int i = 1; i <= n; i *= 3)
    op();
```

$$i = 1, \quad 3, \quad 9, \quad 27, \quad \ldots, \quad n$$

$$= 3^0, \quad 3^1, \quad 3^2, \quad 3^3, \quad \ldots, \quad 3^k$$

These are $k + 1$ steps, where $3^k = n$ i.e. $k = \log_3(n)$

Total number of times `op()` is called = $\log_3(n) + 1$

! In general:

```
for (i = 1; i <= whatever; i *= b)
    op();
```

$\lfloor \log_b(\text{whatever}) \rfloor + 1$

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

X If the nested loops are *dependent*, we can't analyze each loop separately and then multiply them!

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

X If the nested loops are *dependent*, we can't analyze each loop separately and then multiply them!

1 Trace

`i`	`j`	*number of* `op()` *calls*
1		
2		
3		
...		
n		

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

X If the nested loops are *dependent*, we can't analyze each loop separately and then multiply them!

1 Trace

i	j	*number of* op() *calls*
1	[1]	1
2		
3		
...		
n		

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

X If the nested loops are *dependent*, we can't analyze each loop separately and then multiply them!

1 Trace

i	j	*number of* op() *calls*
1	[1]	1
2	[1, 2]	2
3		
...		
n		

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

X If the nested loops are *dependent*, we can't analyze each loop separately and then multiply them!

1 Trace

`i`	`j`	*number of* `op()` *calls*
1	[1]	1
2	[1, 2]	2
3	[1, 2, 3]	3
...		
n		

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

X If the nested loops are *dependent*, we can't analyze each loop separately and then multiply them!

1 Trace

`i`	`j`	*number of* `op()` *calls*
1	[1]	1
2	[1, 2]	2
3	[1, 2, 3]	3
...	...	...
n	[1, 2, 3, …, n]	n

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

X If the nested loops are *dependent*, we can't analyze each loop separately and then multiply them!

1 Trace

i	j	*number of* op() *calls*
1	[1]	1
2	[1, 2]	2
3	[1, 2, 3]	3
...	...	...
n	[1, 2, 3, ..., n]	n

Formulate a sum 2 Total $= 1 + 2 + 3 + \ldots + n$

$$= \sum_{i=0}^{n} i$$

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

X If the nested loops are *dependent*, we can't analyze each loop separately and then multiply them!

1 Trace

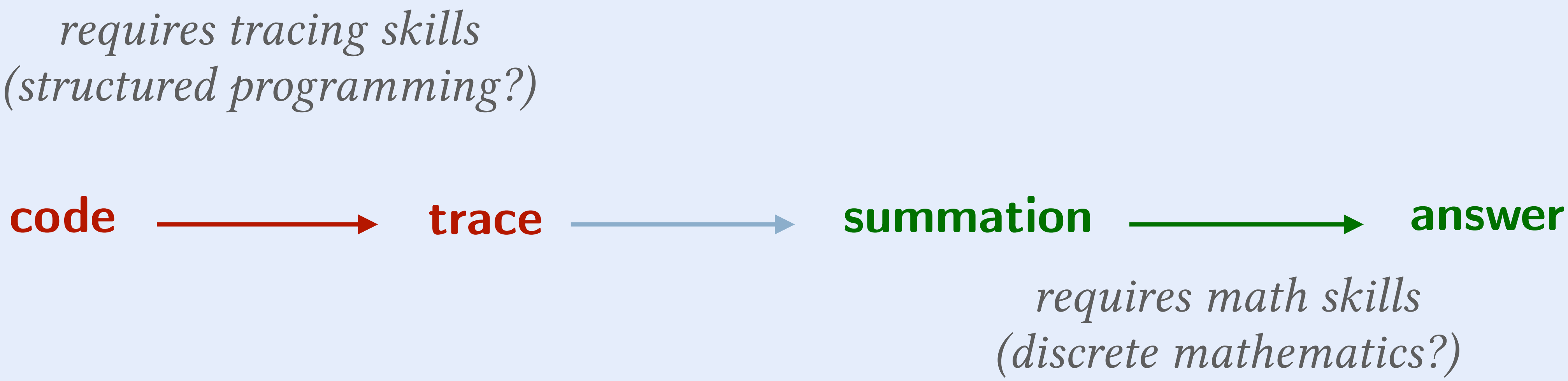
i	j	*number of* `op()` *calls*
1	[1]	1
2	[1, 2]	2
3	[1, 2, 3]	3
...	...	...
n	[1, 2, 3, …, n]	n

Formulate a sum 2 Total $= 1 + 2 + 3 + \ldots + n$

$$= \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

3 Solve the sum

Runtime Analysis Procedure



requires tracing skills
(structured programming?)

code → trace → summation → answer

requires math skills
(discrete mathematics?)

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n*n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n*n; i++)
  for (int j = 1; j <= i; j++)
    op();
```

i	j	*number of* op() *calls*
1	[1]	1
2	[1, 2]	2
3	[1, 2, 3]	3
...	...	...
n∗n	[1, 2, 3, …, n∗n]	n∗n

Total $= 1 + 2 + 3 + \ldots + n^2$

$$= \sum_{i=0}^{n^2} i = \frac{n^2(n^2+1)}{2}$$

! A very frequently encountered sum:

$$\sum_{i=0}^{\star} i = \frac{\star\ (\star + 1)}{2}$$

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    for (int k = 1; k <= i; k++)
      op();
```

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    for (int k = 1; k <= i; k++)
      op();
```

i	*number of* op() *calls*
1	1 x 1
2	2 x 2
3	3 x 3
...	...
n	n x n

Total $= 1^2 + 2^2 + 3^2 + \ldots + n^2$

$$= \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

← see the math cheatsheet

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j *= 2)
    op();
```

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j *= 2)
    op();
```

`i`	*number of* `op()` *calls*
1	$\log_2(1) + 1$
2	$\log_2(2) + 1$
3	$\log_2(3) + 1$
…	…
n	$\log_2(n) + 1$

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j *= 2)
    op();
```

`i`	*number of* `op()` *calls*
1	$\log_2(1) + 1$
2	$\log_2(2) + 1$
3	$\log_2(3) + 1$
…	…
n	$\log_2(n) + 1$

Total $= \log_2(1) + \log_2(2) + \log_2(3) + \ \dots\ + \log_2(n) + (n \times 1)$

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j *= 2)
    op();
```

`i`	*number of* `op()` *calls*
1	$\log_2(1) + 1$
2	$\log_2(2) + 1$
3	$\log_2(3) + 1$
…	…
n	$\log_2(n) + 1$

Total $= \log_2(1) + \log_2(2) + \log_2(3) + \; \dots \; + \log_2(n) + (n \times 1)$

$= \log_2(1 \times 2 \times 3 \times \dots \times n) + (n \times 1) = \log_2(n!) + n$

How Many Operations?

How many times does `op()` get called?

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j *= 2)
    op();
```

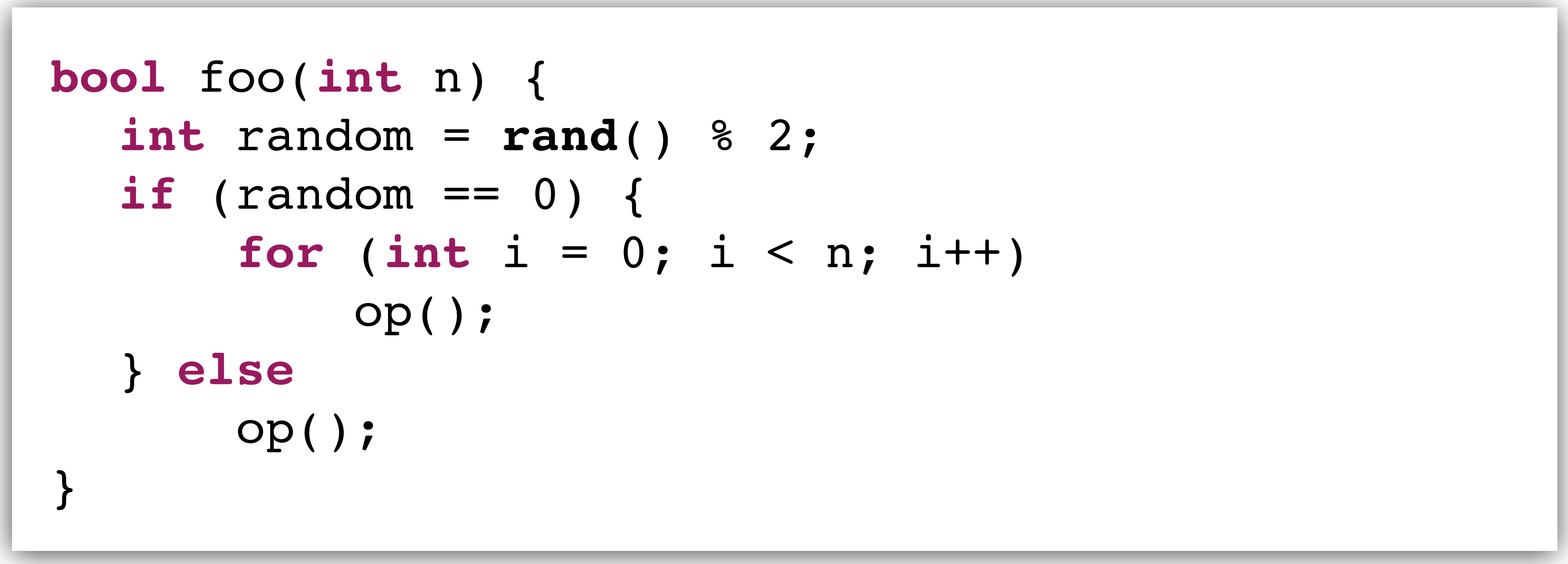
`i`	*number of* `op()` *calls*
1	$\log_2(1) + 1$
2	$\log_2(2) + 1$
3	$\log_2(3) + 1$
…	…
n	$\log_2(n) + 1$

$$\text{Total} = \log_2(1) + \log_2(2) + \log_2(3) + \ \dots\ + \log_2(n) + (n \times 1)$$

$$= \log_2(1 \times 2 \times 3 \times \dots \times n) + (n \times 1) = \log_2(n!) + n$$

$$\sim n \log_2(n)$$ ← Stirling's Approximation (see the math cheatsheet)

How Many Operations?



```
bool foo(int n) {
   int random = rand() % 2;
   if (random == 0) {
       for (int i = 0; i < n; i++)
           op();
   } else
       op();
}
```

How Many Operations?

```
bool foo(int n) {
   int random = rand() % 2;
   if (random == 0) {
        for (int i = 0; i < n; i++)
             op();
   } else
        op();
}
```

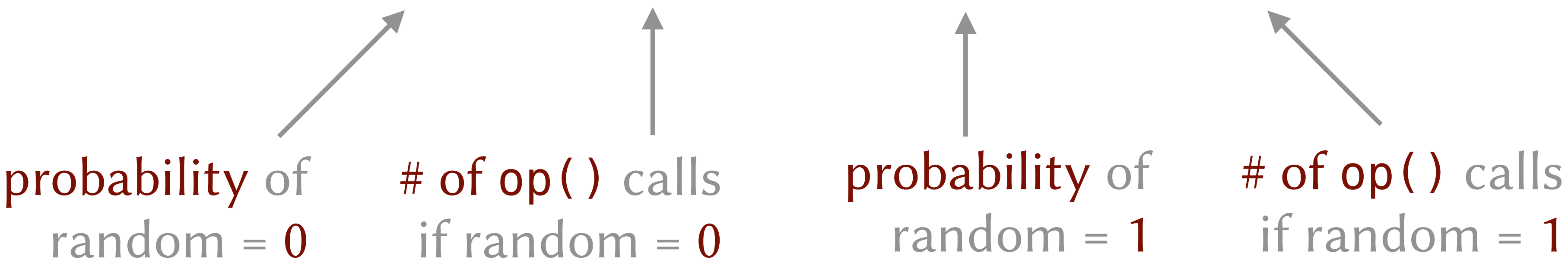
Best Case: `op()` is called 1 time (if random = 1)
Worst Case: `op()` is called *n* times (if random = 0).

How Many Operations?

```
bool foo(int n) {
   int random = rand() % 2;
   if (random == 0) {
      for (int i = 0; i < n; i++)
         op();
   } else
      op();
}
```

Best Case: `op()` is called 1 time (if random = 1)
Worst Case: `op()` is called *n* times (if random = 0).

Average Case: P(0) × cost(0) + P(1) × cost(1)



probability of random = 0

# of `op()` calls if random = 0

probability of random = 1

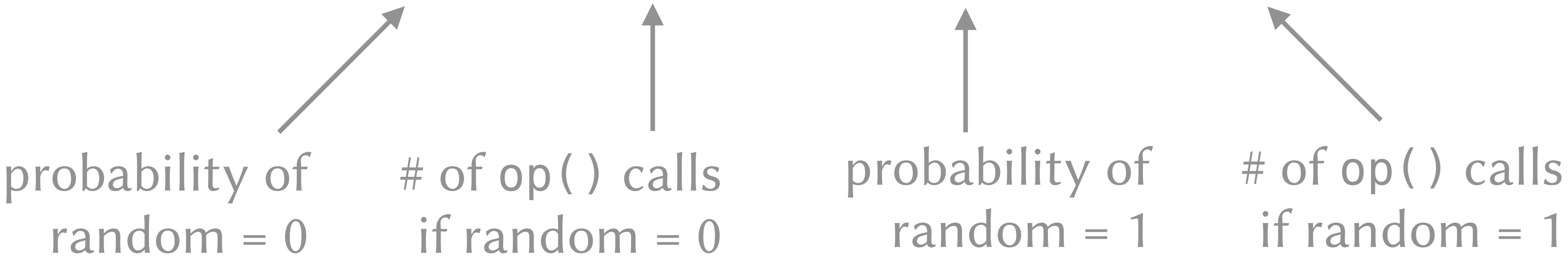
# of `op()` calls if random = 1

How Many Operations?

```
bool foo(int n) {
   int random = rand() % 2;
   if (random == 0) {
        for (int i = 0; i < n; i++)
             op();
   } else
        op();
}
```

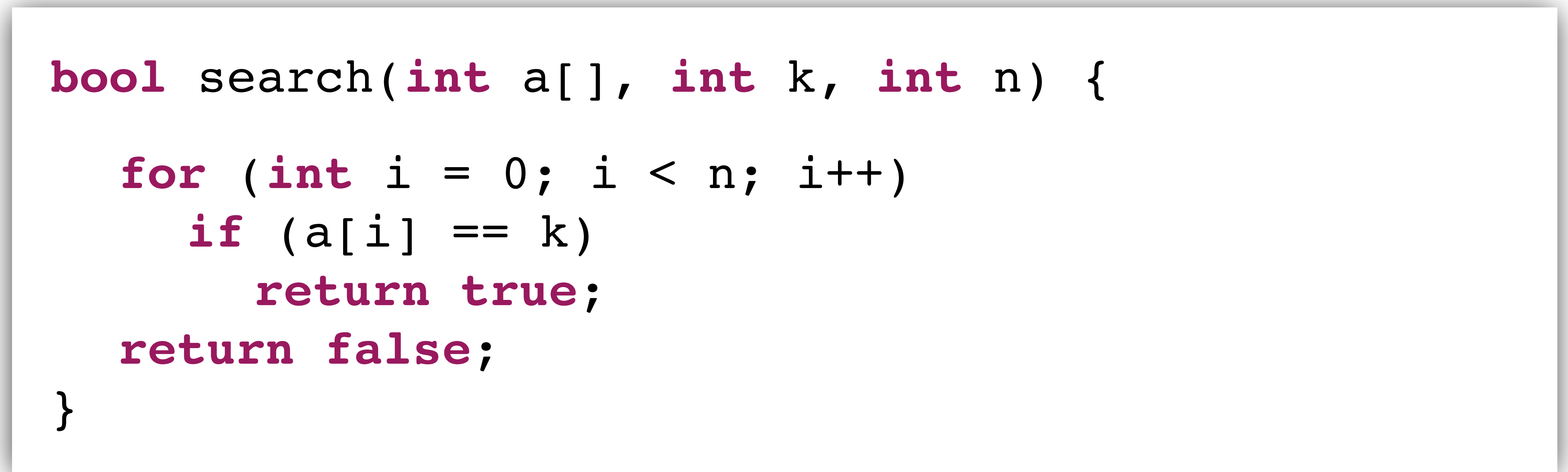
Best Case: `op()` is called 1 time (if random = 1)
Worst Case: `op()` is called n times (if random = 0).

Average Case: P(0) × cost(0) + P(1) × cost(1)


probability of random = 0
# of op( ) calls if random = 0
probability of random = 1
# of op( ) calls if random = 1

$$\frac{1}{2} \times n + \frac{1}{2} \times 1 = \frac{1}{2}n+\frac{1}{2} = \frac{1}{2}(n+1)$$

How Many Operations?



```
bool search(int a[], int k, int n) {

  for (int i = 0; i < n; i++)
    if (a[i] == k)
      return true;
  return false;
}
```

Let's consider *comparisons with k* as the basis for our analysis.

Best Case: 1 comparison (*k* is the first element in the list).
Worst Case: *n* comparisons (*k* is not in the list).

How Many Operations?

```
bool search(int a[], int k, int n) {

   for (int i = 0; i < n; i++)
      if (a[i] == k)
         return true;
   return false;
}
```

Let’s consider *comparisons with k* as the basis for our analysis.

Best Case: 1 comparison (k is the first element in the list).
Worst Case: n comparisons (k is not in the list).

Average Case: $$\sum_{i=0}^{n-1} \underset{\uparrow}{\mathrm{P}(i)} \times \underset{\uparrow}{\mathrm{cost}(i)}$$

probability of finding *k* at index *i*

number of operations if *k* is found at index *i*

How Many Operations?

```
bool search(int a[], int k, int n) {

   for (int i = 0; i < n; i++)
      if (a[i] == k)
         return true;
   return false;
}
```

Let’s consider *comparisons with k* as the basis for our analysis.

Best Case: 1 comparison (k is the first element in the list).
Worst Case: n comparisons (k is not in the list).

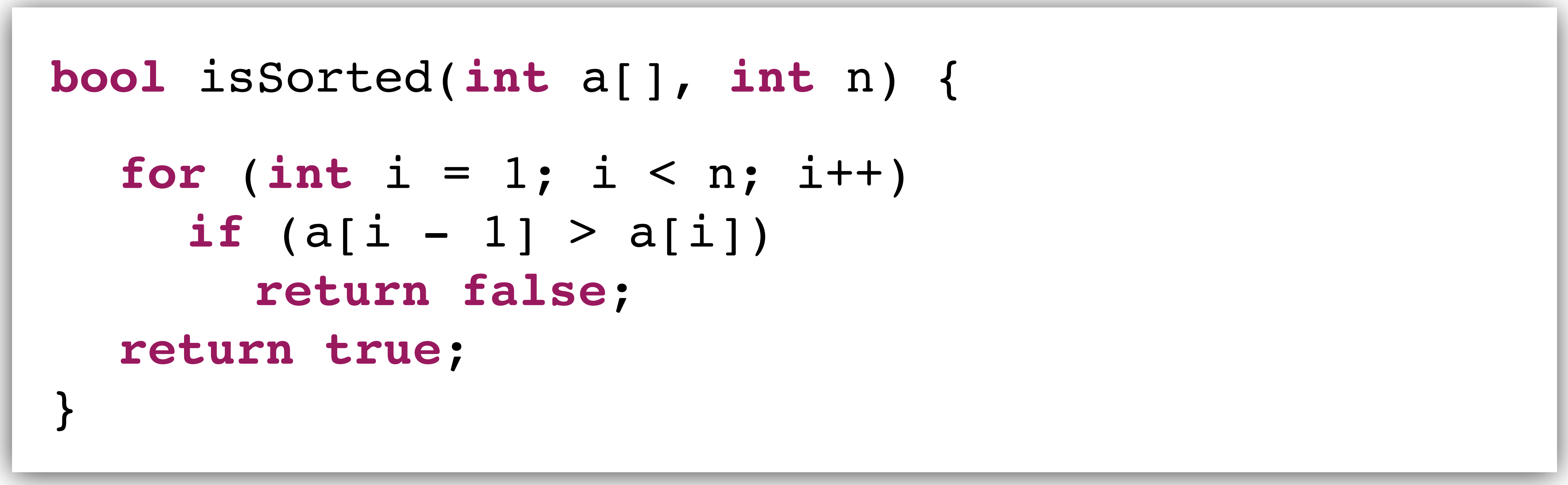
Average Case: $$\sum_{i=0}^{n-1} \underbrace{\mathrm{P}(i)}_{\text{probability of finding } k \text{ at index } i} \times \underbrace{\mathrm{cost}(i)}_{\text{number of operations if } k \text{ is found at index } i}$$

Assuming k is equally likely to appear at any index:

$$= \left(\frac{1}{n} \times 1\right) + \left(\frac{1}{n} \times 2\right) + \ldots + \left(\frac{1}{n} \times n\right)$$

$$= \frac{1}{n} \times \left(\frac{n(n+1)}{2}\right) = \frac{1}{2}(n+1)$$

How Many Operations?



```
bool isSorted(int a[], int n) {

  for (int i = 1; i < n; i++)
    if (a[i - 1] > a[i])
      return false;
  return true;
}
```

Let’s consider *comparisons between array elements* as the basis for our analysis.

Best Case: 1 comparison (first two elements are not in order).
Worst Case: *n* - 1 comparisons (list is in order).

Average Case: Not straightforward!

! We will focus on *best case* and *worst case* analysis in this course.

CS11212 - **Spring** 2022

Data Structures & Introduction to **Algorithms**

Analysis of Algorithms

part 2: Asymptotic Analysis

Ibrahim Albluwi

Which is better?

A

```
for (int i=0; i < 50 * n; i++)
   op();
```

$50n$

B

```
for (int i=0; i < n * n; i++)
   op();
```

n^2

Which is better?

A

```
for (int i=0; i < 50 * n; i++)
   op();
```

$50n$

B

```
for (int i=0; i < n * n; i++)
   op();
```

n^2

We expressed the number of operations performed by each program as $T_A(n) = 50n$ and $T_B(n) = n^2$, which are two functions that have different values depending on the value of the input size n.

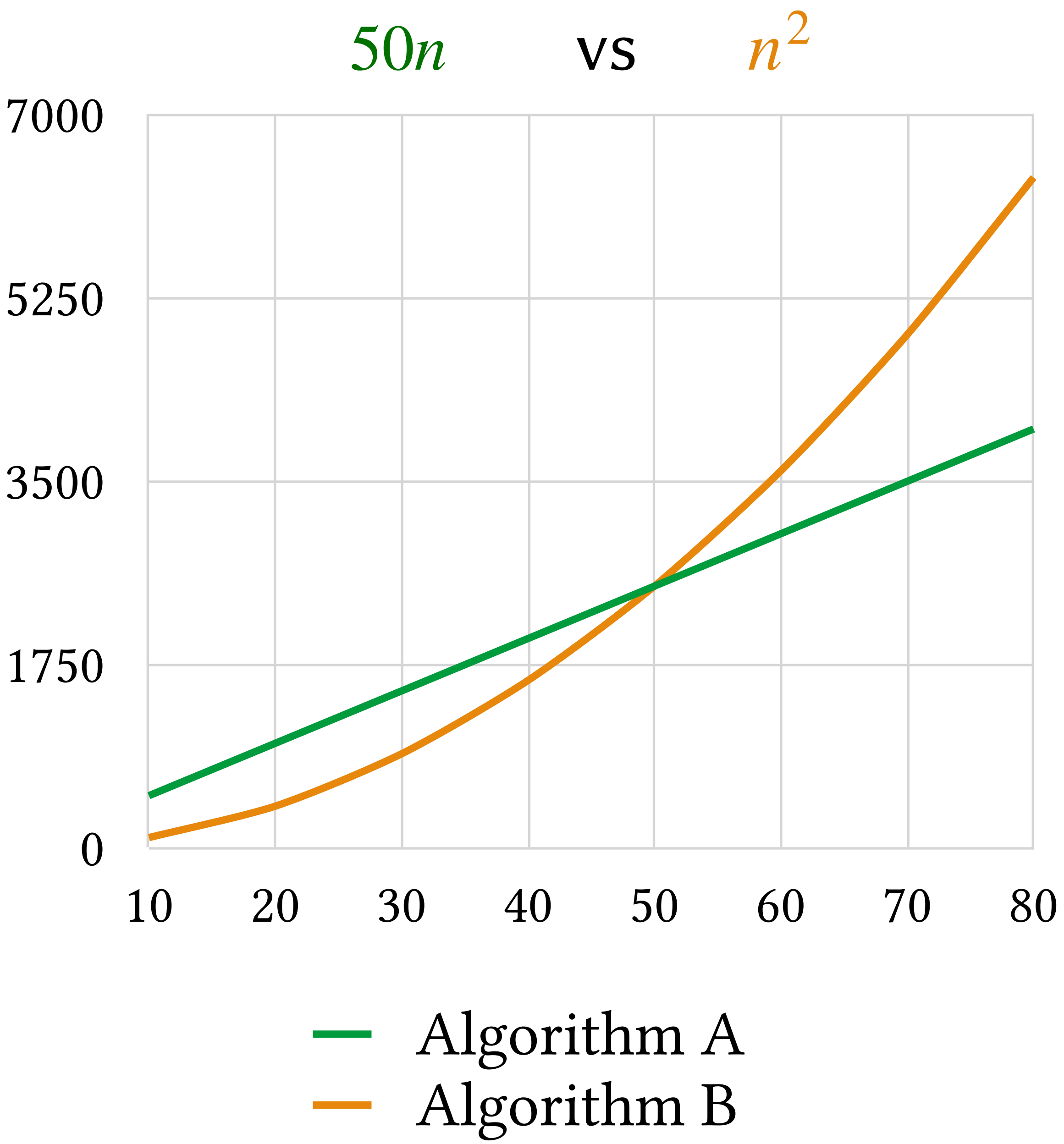
? Which function represents a better running time (less performed operations)?

Which is better?

	$50n$	n^2
n	Algorithm A	Algorithm B
10	500	100

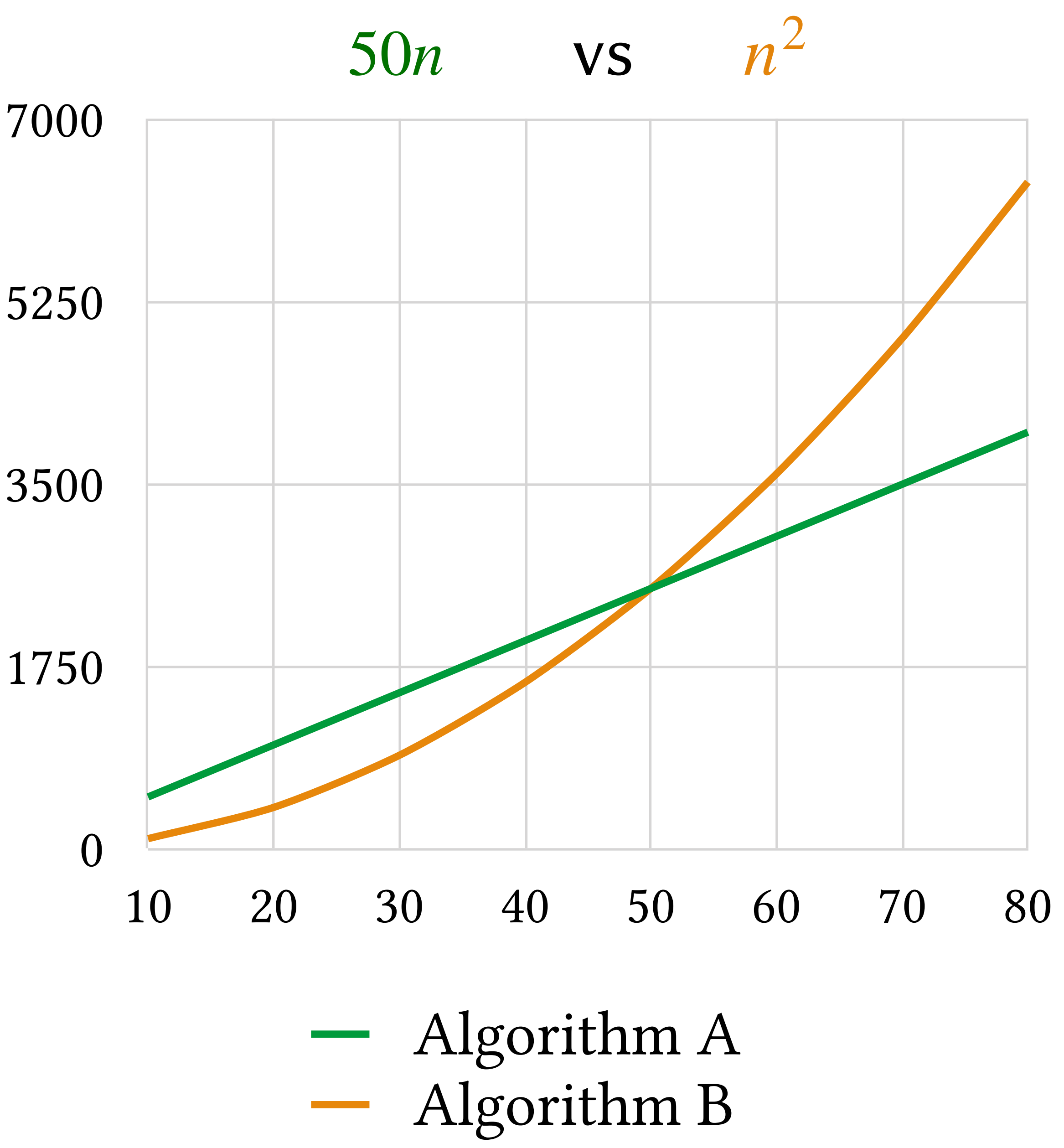
Which is better?

		$50n$	n^2
	n	Algorithm A	Algorithm B
$50n$ is worse	10	500	100
	20	1000	400
	30	1500	900
	40	2000	1600
	50	2500	2500
n^2 is worse	60	3000	3600
	70	3500	4900
	80	4000	6400
	90	4500	8100



Which is better?

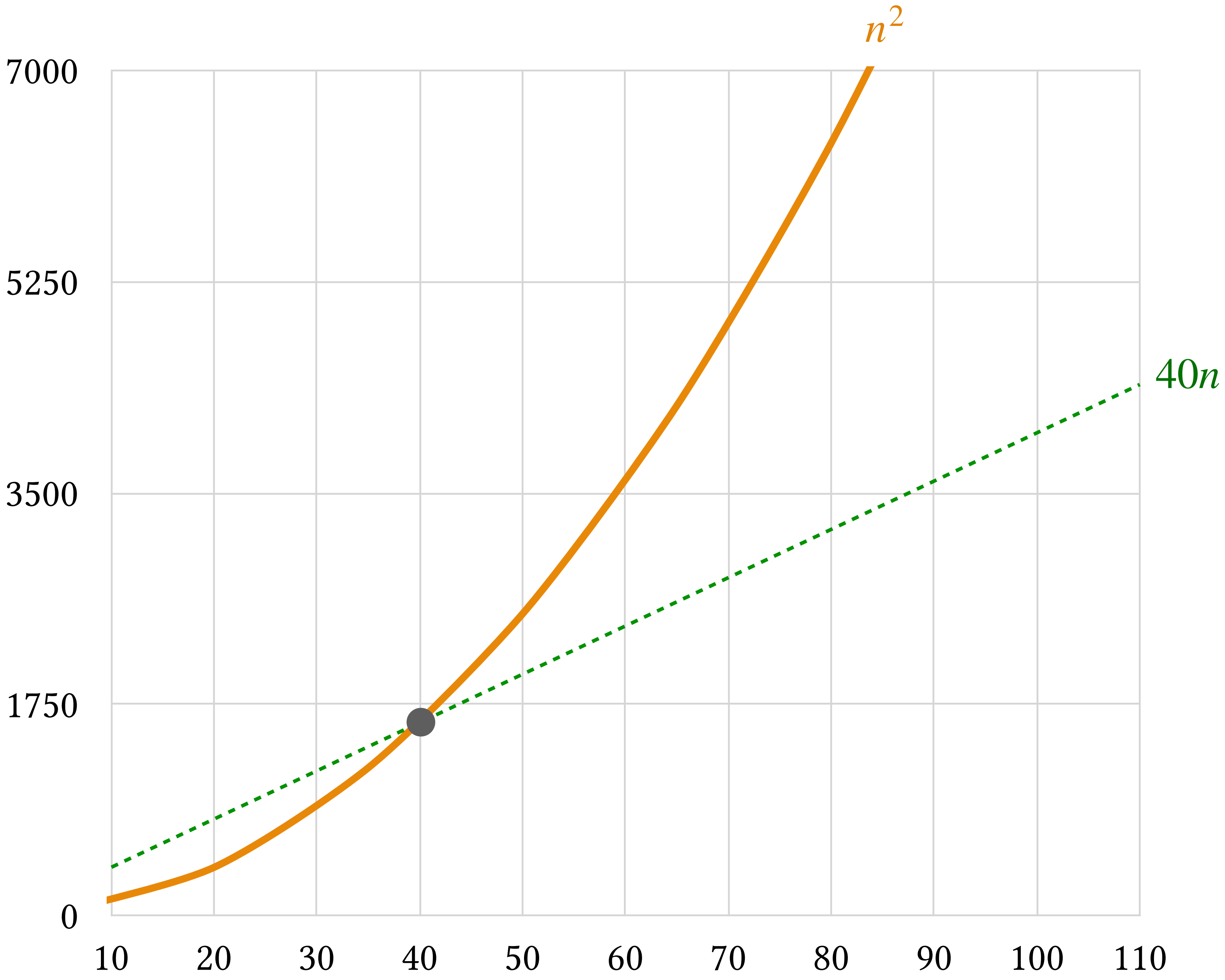
		$50n$	n^2
	n	Algorithm *A*	Algorithm *B*
$50n$ is worse	10	500	100
	20	1000	400
	30	1500	900
	40	2000	1600
	50	2500	2500
n^2 is worse	60	3000	3600
	70	3500	4900
	80	4000	6400
	90	4500	8100



! n^2 grows faster than $50n$.

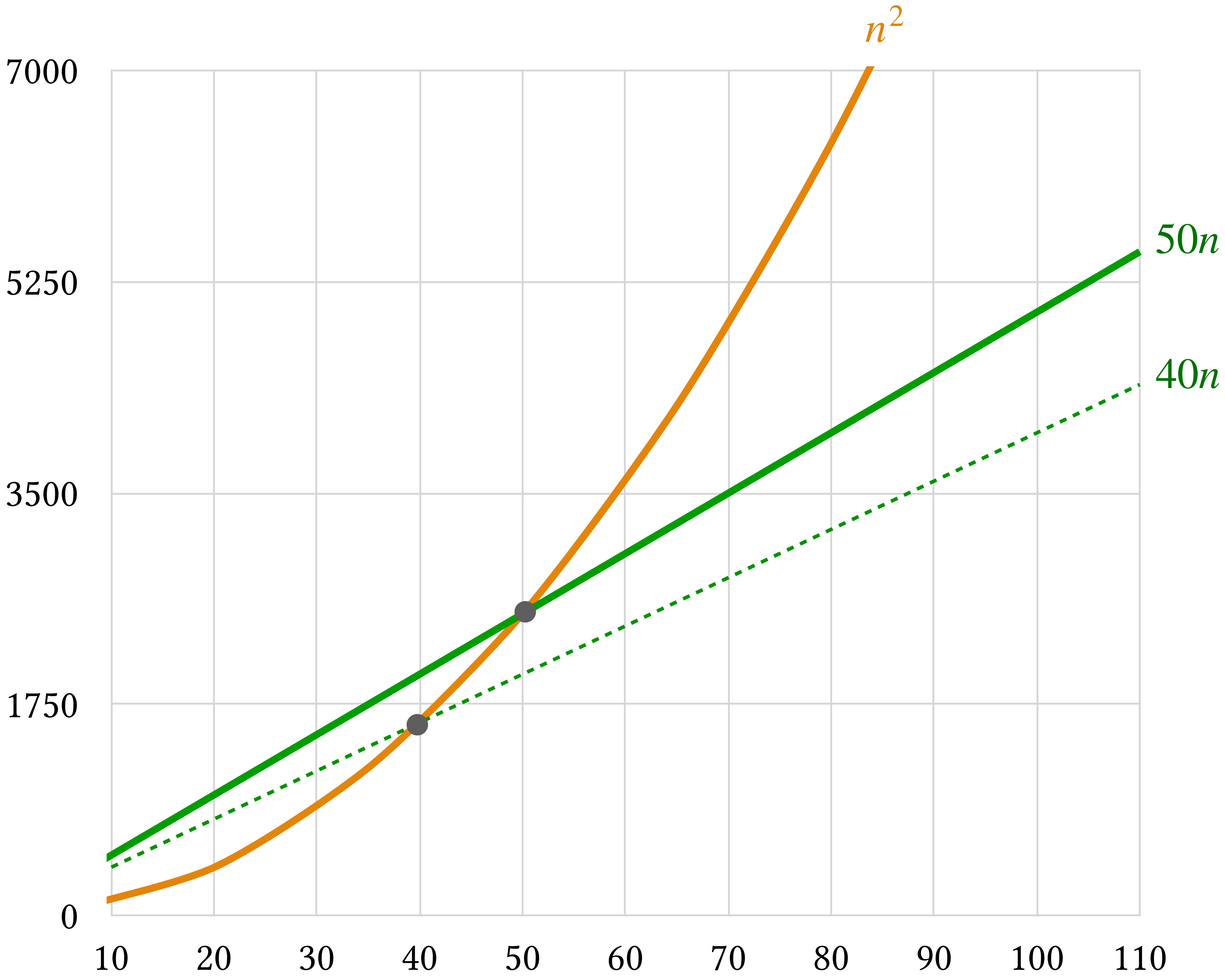
n^2 must at some point become worse (perform more operations) than $50n$ forever (when n > 50 in this case)

Orders of Growth



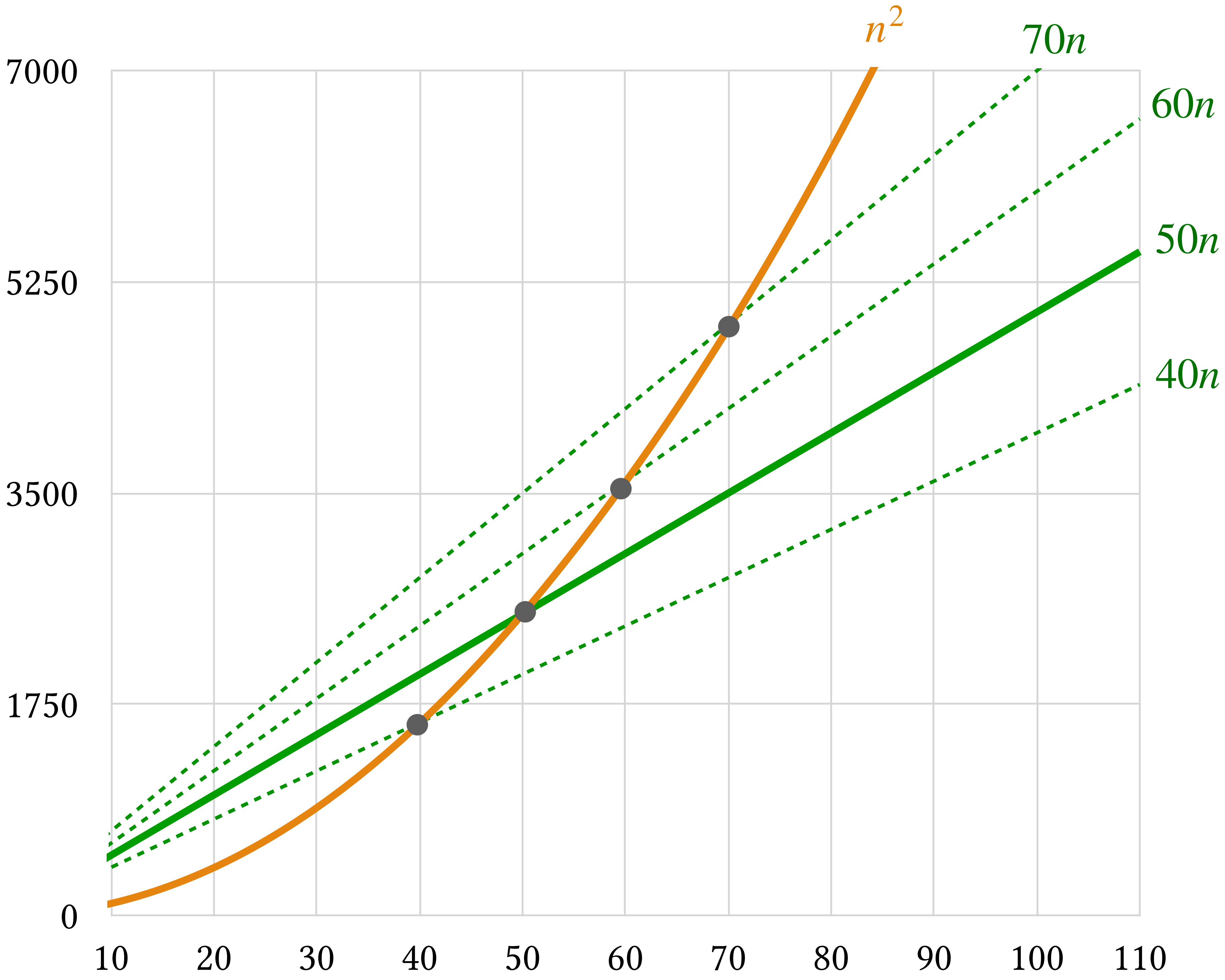
! n^2 will at some point exceed cn regardless of what the value of c is.

Orders of Growth



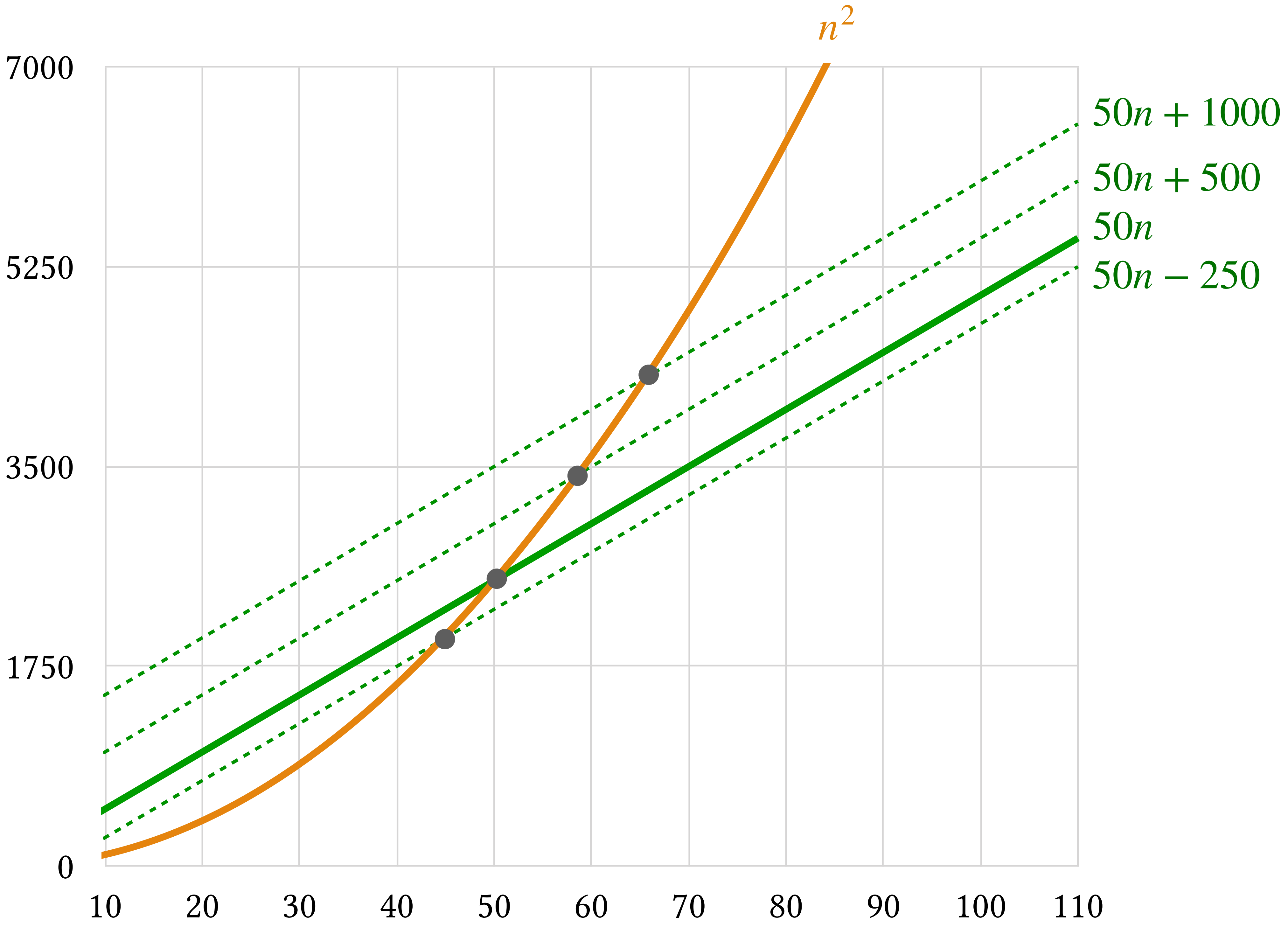
! n^2 will at some point exceed cn regardless of what the value of c is.

Orders of Growth



! n^2 will at some point exceed cn regardless of what the value of c is.

Orders of Growth



! n^2 will at some point exceed $cn + a$ regardless of what the values of c and a are.

Orders of Growth

Example. Assume $n \geq 10$ is the size of an array and we are interested in counting the number of array accesses an algorithm performs.

? How quickly does the number operations performed grows when the input size grows (when the array size grows)?

```
for (i=0; i<10; i++)
   sum += a[0];
```

```
for (i=0; i<n; i++)
   sum += a[i];
```

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
     sum += a[j];
```

Orders of Growth

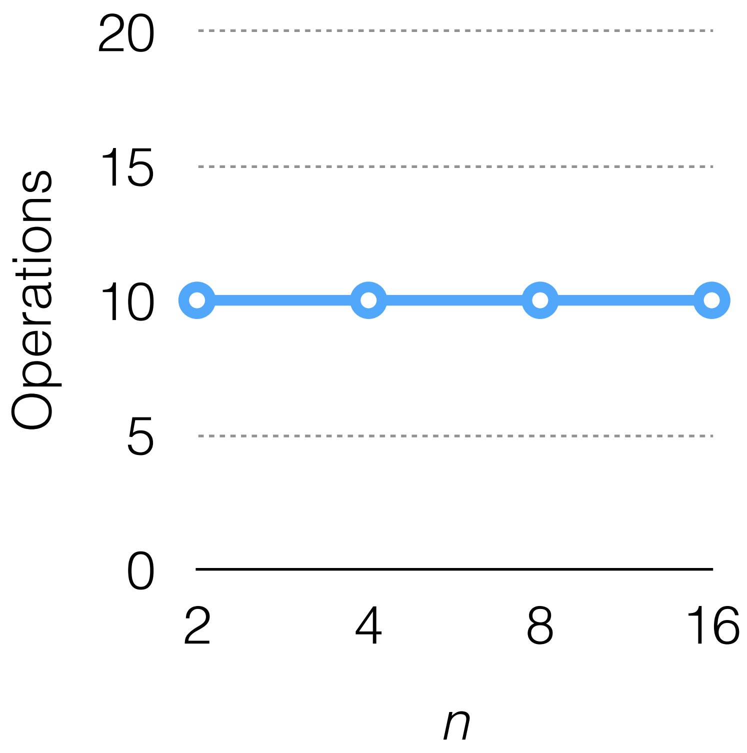
Example. Assume $n \geq 10$ is the size of an array and we are interested in counting the number of array accesses an algorithm performs.

? How quickly does the number operations performed grows when the input size grows (when the array size grows)?

```
for (i=0; i<10; i++)
   sum += a[0];
```

```
for (i=0; i<n; i++)
   sum += a[i];
```

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    sum += a[j];
```



No growth!

Always 10, regardless of the array size

Orders of Growth

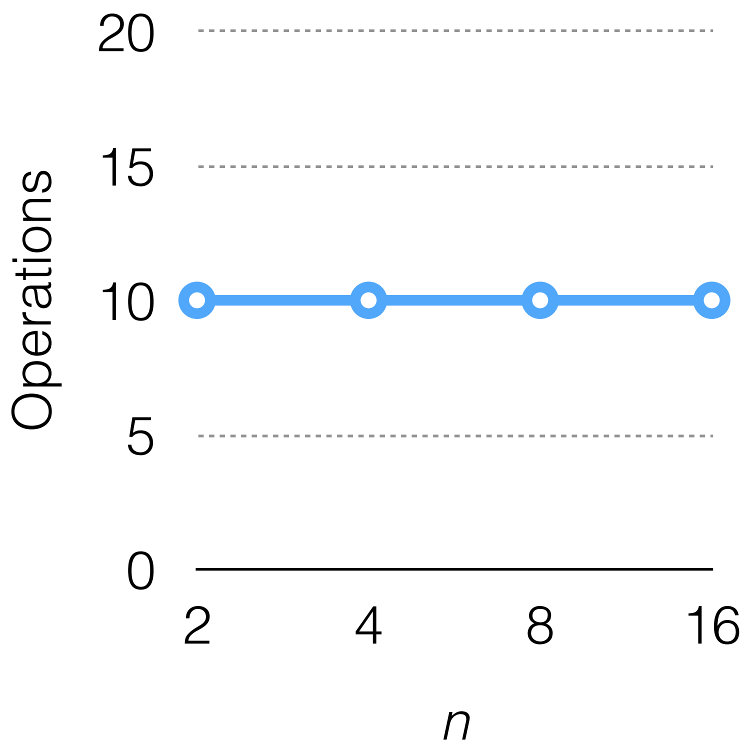
Example. Assume $n \geq 10$ is the size of an array and we are interested in counting the number of array accesses an algorithm performs.

? How quickly does the number operations performed grows when the input size grows (when the array size grows)?

```
for (i=0; i<10; i++)
   sum += a[0];
```

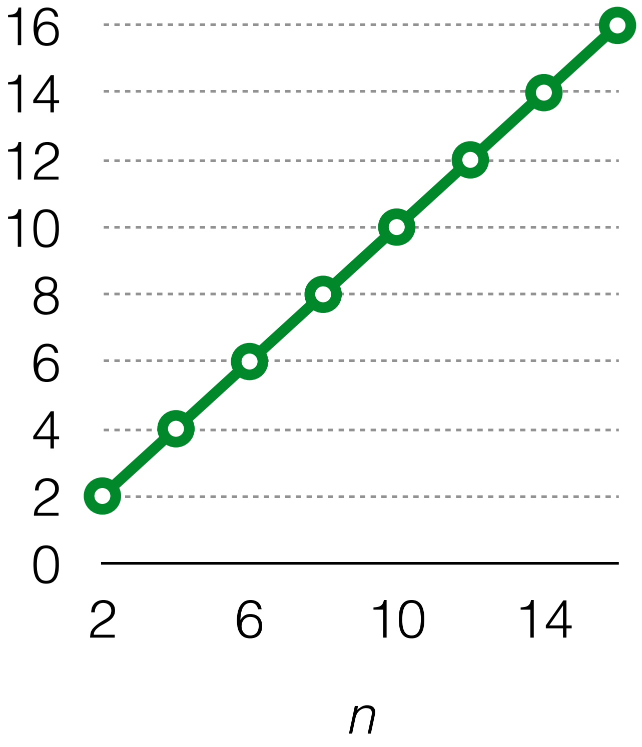
```
for (i=0; i<n; i++)
   sum += a[i];
```

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    sum += a[j];
```



No growth!

Always 10, regardless of the array size



Linear growth!

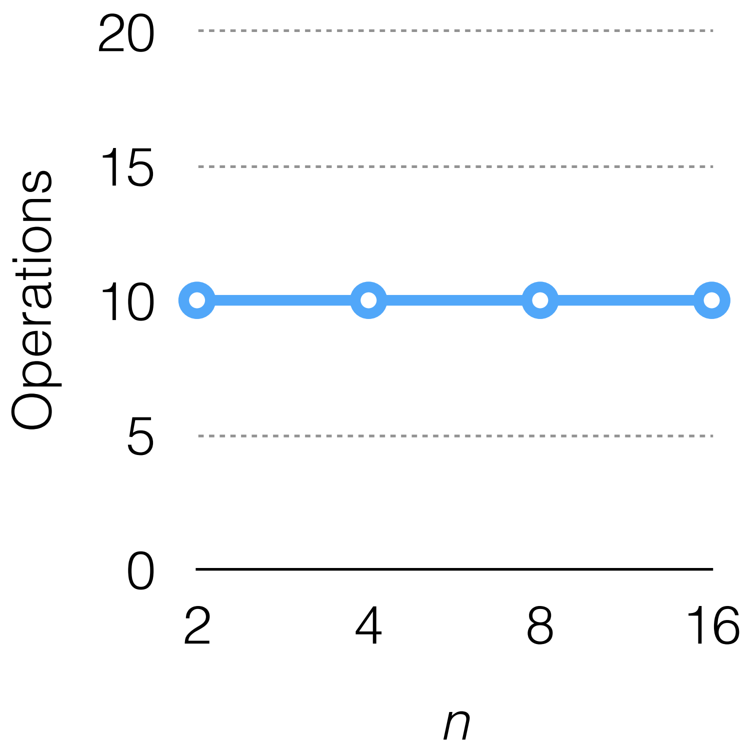
operations double when the array size doubles

Orders of Growth

Example. Assume $n \geq 10$ is the size of an array and we are interested in counting the number of array accesses an algorithm performs.

? How quickly does the number operations performed grows when the input size grows (when the array size grows)?

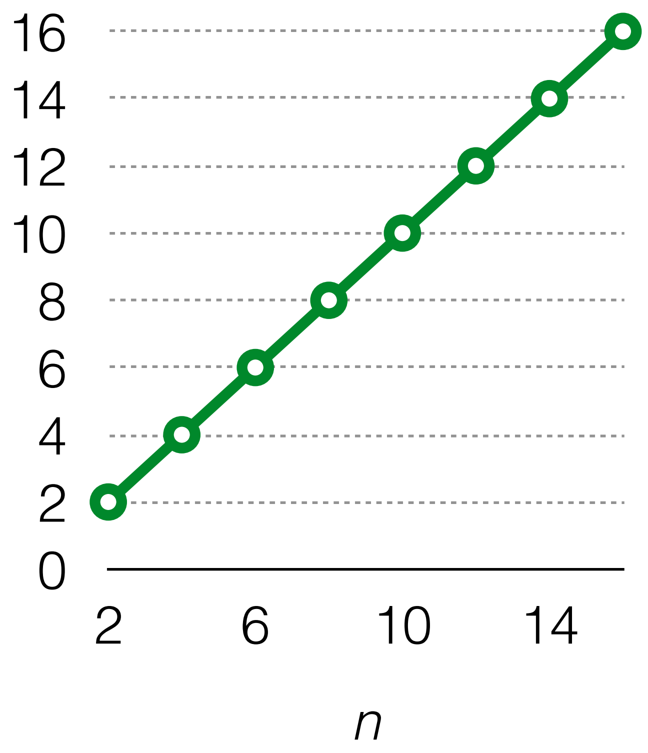
```
for (i=0; i<10; i++)
   sum += a[0];
```



No growth!

Always 10, regardless of the array size

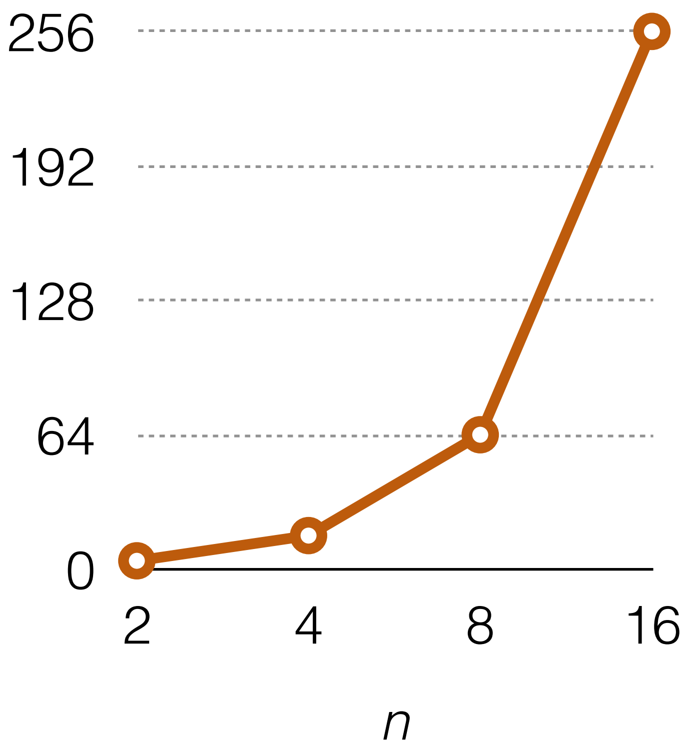
```
for (i=0; i<n; i++)
   sum += a[i];
```



Linear growth!

operations double when the array size doubles

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      sum += a[j];
```



Quadratic growth!

operations quadruple when the array size doubles

Lesson # 1

Look at the running time ***growth rate***!

Classify algorithms based on the order of growth of their running time (ignoring the coefficients).

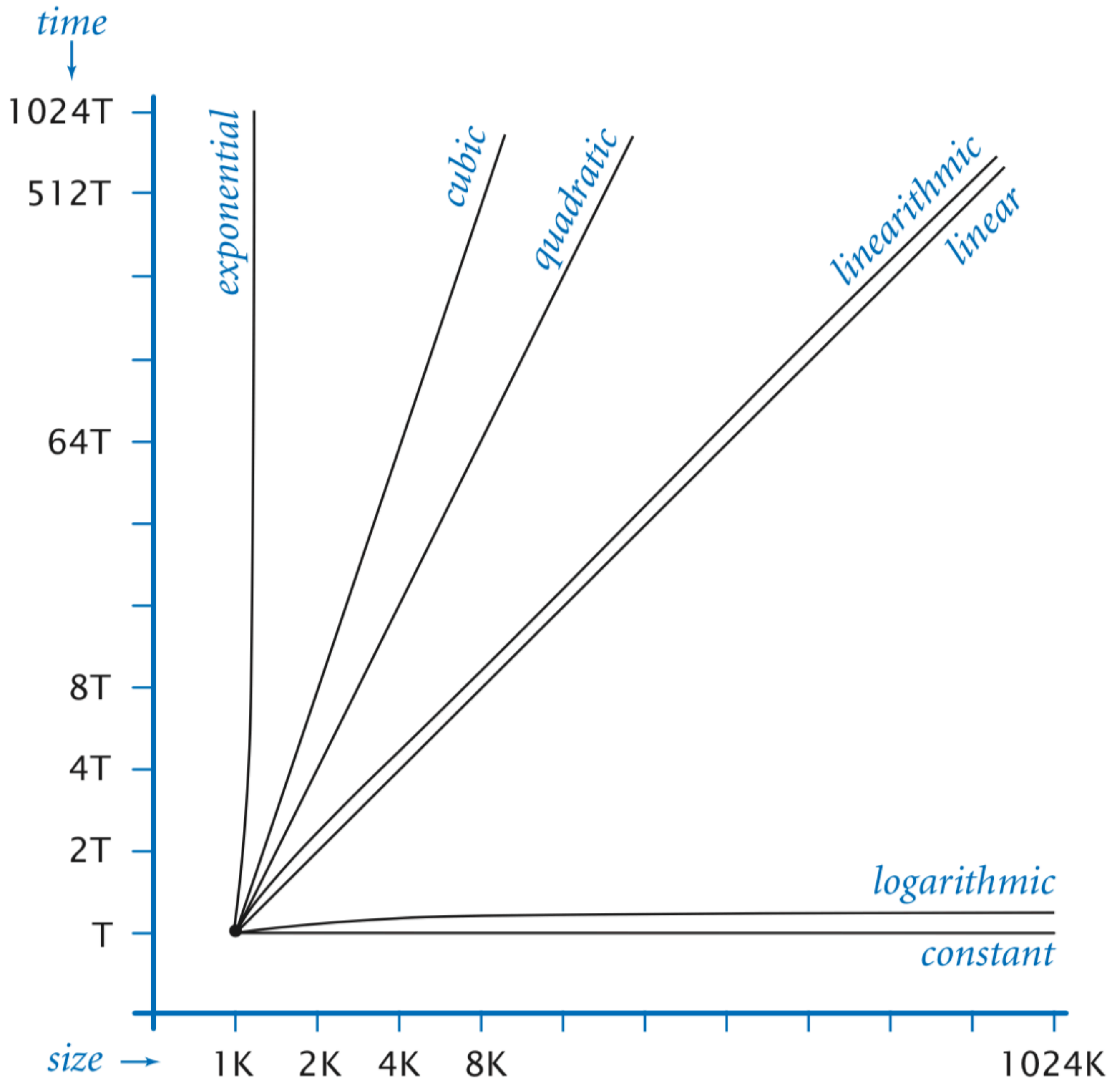
Example:

$5n^2, \quad 30n^2, \quad 7n^2, \quad \text{etc}\,.$	have a *quadratic* order of growth.
$7n, \quad 87n, \quad 3n, \quad \text{etc}\,.$	have a *linear* order of growth.
$3\log_2 n, \quad 2\ln n, \quad 10\log_{10} n, \quad \text{etc}\,.$	have a *logarithmic* order of growth.

Examples of Growth Rates

graph by Kevin Wayne and Robert Sedgewick

order of growth name	order of growth function
constant	1
logarithmic	$\log(n)$
	$\sqrt{n}$
linear	n
linearithmic	$n\log(n)$
	$n\sqrt{n}$
quadratic	n^2
cubic	n^3
exponential	2^n
exponential	3^n
factorial	$n!$

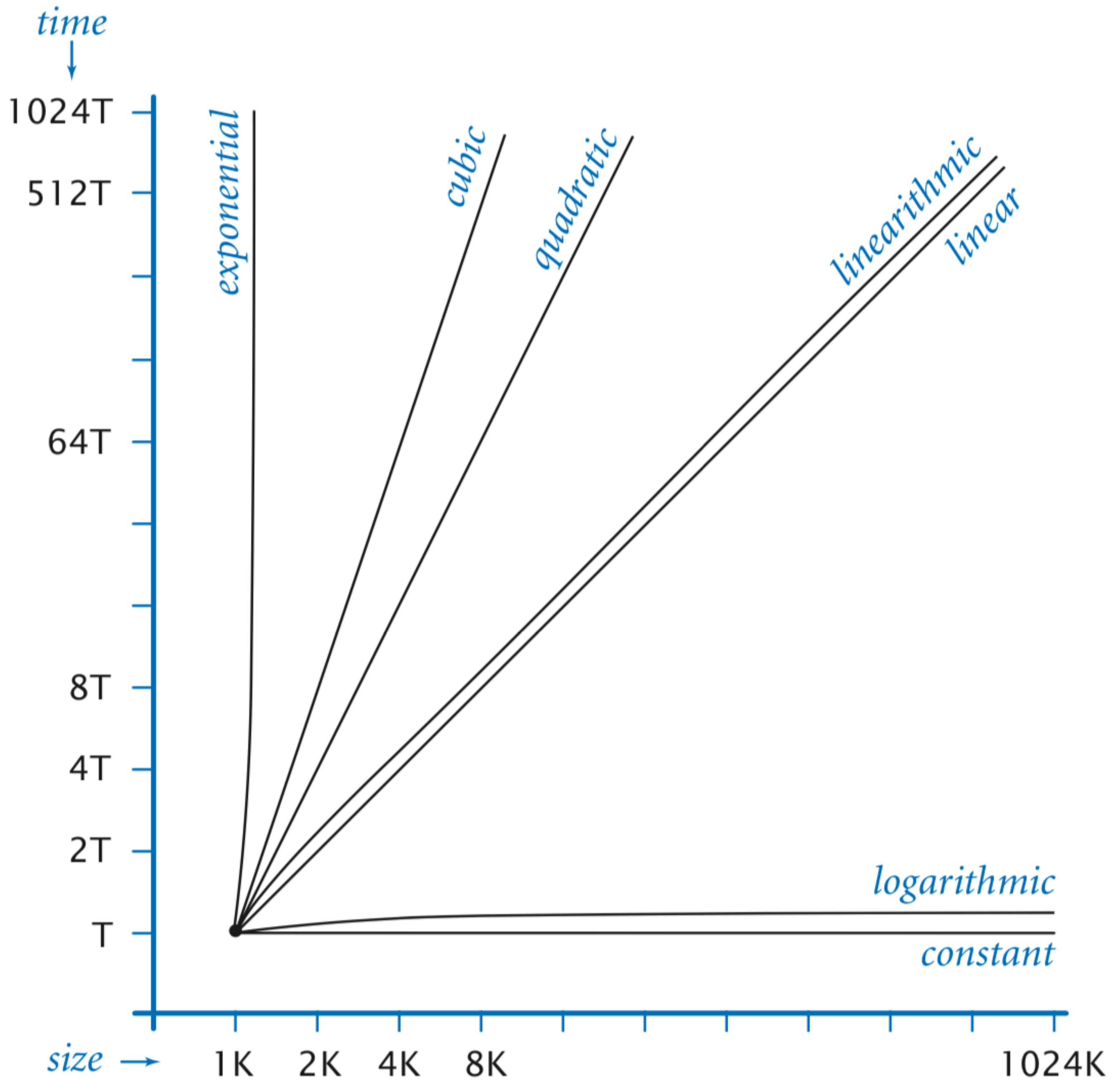


Orders of growth (log–log plot)

Examples of Growth Rates

graph by Kevin Wayne and Robert Sedgewick

order of growth	
name	function
constant	1
logarithmic	$\log(n)$
	$\sqrt{n}$
linear	n
linearithmic	$n \log(n)$
	$n\sqrt{n}$
quadratic	n^2
cubic	n^3
exponential	2^n
exponential	3^n
factorial	$n!$



Orders of growth (log–log plot)

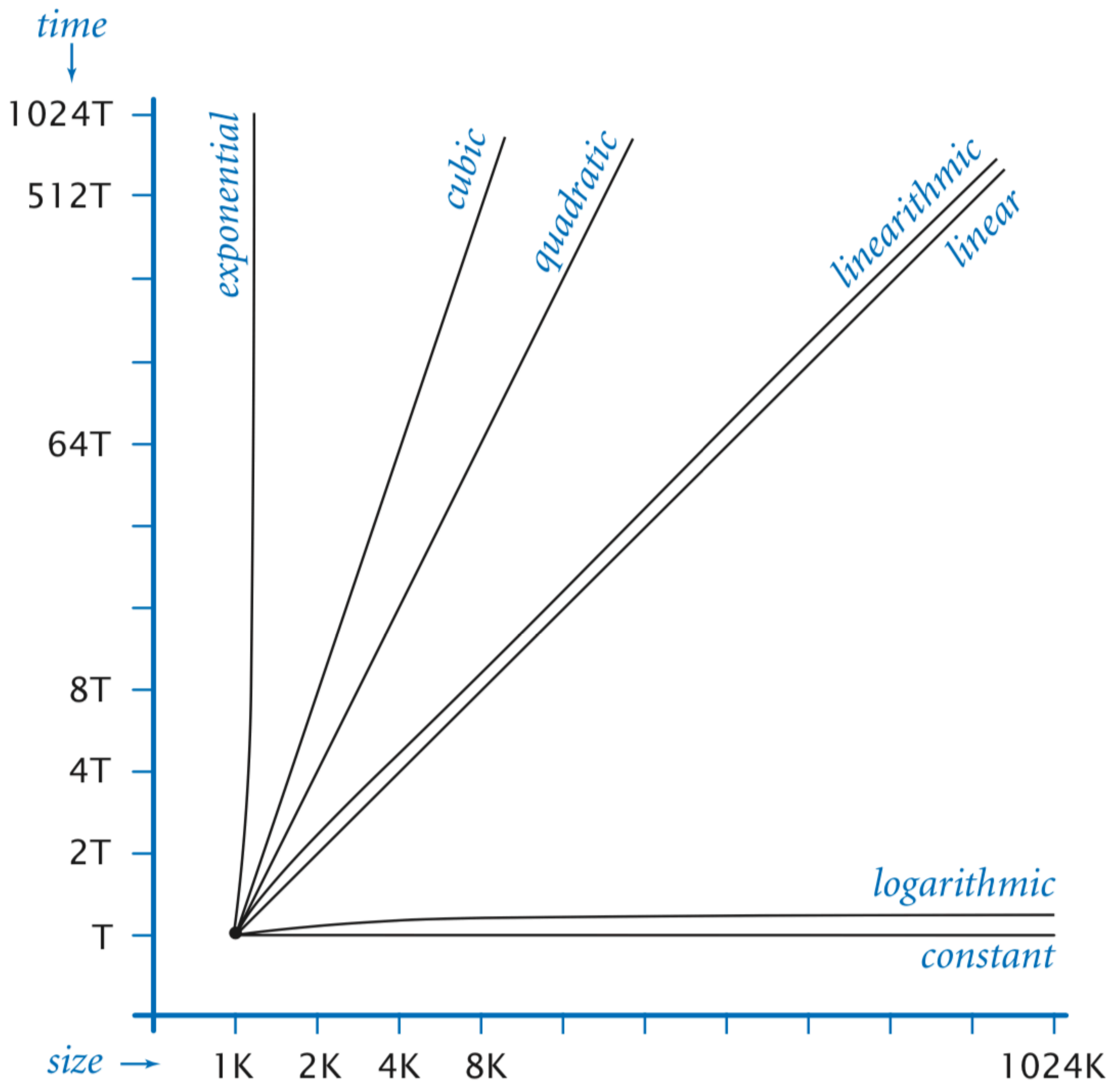
! constant < logarithmic < polynomial < exponential < factorial < n^n

$\log_b(n)$ n^c $(c > 0)$ c^n $(c > 1)$

Examples of Growth Rates

graph by Kevin Wayne and Robert Sedgewick

	order of growth	
	name	function
	constant	1
good	logarithmic	$\log(n)$
		$\sqrt{n}$
fine	linear	n
	linearithmic	$n\log(n)$
		$n\sqrt{n}$
bad	quadratic	n^2
	cubic	n^3
horrible	exponential	2^n
	exponential	3^n
	factorial	$n!$



Orders of growth (log–log plot)

! constant < logarithmic < polynomial < exponential < factorial < n^n

$\log_b(n)$ $n^c\ (c > 0)$ $c^n\ (c > 1)$

Lower Order Terms? Really?

If the running time of an algorithm is given by:

$$n^2 \quad + \quad 100n \quad + \quad \log_{10}(n) \quad + \quad 1000$$

What is the *most* important term?

Lower Order Terms? Really?

If the running time of an algorithm is given by:

$$n^2 \quad + \quad 100n \quad + \quad \log_{10}(n) \quad + \quad 1000$$

What is the *most* important term?

value of n	n^2	$100n$	$\log_{10}(n)$	1000
1	1	10^2	0	$\boxed{10^3}$

Lower Order Terms? Really?

If the running time of an algorithm is given by:

$$n^2 \quad + \quad 100n \quad + \quad \log_{10}(n) \quad + \quad 1000$$

What is the *most* important term?

value of n	n^2	$100n$	$\log_{10}(n)$	1000
1	1	10^2	0	$\boxed{10^3}$
10	10^2	$\boxed{10^3}$	1	$\boxed{10^3}$

Lower Order Terms? Really?

If the running time of an algorithm is given by:

$$n^2 \quad + \quad 100n \quad + \quad \log_{10}(n) \quad + \quad 1000$$

What is the *most* important term?

value of n	n^2	$100n$	$\log_{10}(n)$	1000
1	1	10^2	0	$\boxed{10^3}$
10	10^2	$\boxed{10^3}$	1	$\boxed{10^3}$
100	$\boxed{10^4}$	$\boxed{10^4}$	2	10^3

Lower Order Terms? Really?

If the running time of an algorithm is given by:

$$n^2 \quad + \quad 100n \quad + \quad \log_{10}(n) \quad + \quad 1000$$

What is the *most* important term?

value of n	n^2	$100n$	$\log_{10}(n)$	1000
1	1	10^2	0	$\boxed{10^3}$
10	10^2	$\boxed{10^3}$	1	$\boxed{10^3}$
100	$\boxed{10^4}$	$\boxed{10^4}$	2	10^3
1000	$\boxed{10^6}$	10^5	3	10^3

Lower Order Terms? Really?

If the running time of an algorithm is given by:

$$n^2 \quad + \quad 100n \quad + \quad \log_{10}(n) \quad + \quad 1000$$

What is the *most* important term?

value of n	n^2	$100n$	$\log_{10}(n)$	1000
1	1	10^2	0	$\boxed{10^3}$
10	10^2	$\boxed{10^3}$	1	$\boxed{10^3}$
100	$\boxed{10^4}$	$\boxed{10^4}$	2	10^3
1000	$\boxed{10^6}$	10^5	3	10^3
100000	$\boxed{10^{10}}$	10^7	5	10^3

Lower Order Terms? Really?

If the running time of an algorithm is given by:

$$n^2 \quad + \quad 100n \quad + \quad \log_{10}(n) \quad + \quad 1000$$

What is the *most* important term?

value of n	n^2	$100n$	$\log_{10}(n)$	1000
1	1	10^2	0	$\boxed{10^3}$
10	10^2	$\boxed{10^3}$	1	$\boxed{10^3}$
100	$\boxed{10^4}$	$\boxed{10^4}$	2	10^3
1000	$\boxed{10^6}$	10^5	3	10^3
100000	$\boxed{10^{10}}$	10^7	5	10^3

Assume
1 op requires
10^{-6} seconds:

Lower Order Terms? Really?

If the running time of an algorithm is given by:

$$n^2 \quad + \quad 100n \quad + \quad \log_{10}(n) \quad + \quad 1000$$

What is the *most* important term?

value of n	n^2	$100n$	$\log_{10}(n)$	1000
1	1	10^2	0	$\boxed{10^3}$
10	10^2	$\boxed{10^3}$	1	$\boxed{10^3}$
100	$\boxed{10^4}$	$\boxed{10^4}$	2	10^3
1000	$\boxed{10^6}$	10^5	3	10^3
100000	$\boxed{10^{10}}$	10^7	5	10^3

Assume
1 op requires
10^{-6} seconds:

$10^{10} \times 10^{-6}$
$= 10^4$ seconds
2.78 Hours +

$10^7 \times 10^{-6}$
10 sec +

5×10^{-6}
0.000005 sec +

$10^3 \times 10^{-6}$
0.001 sec

Lower Order Terms? Really?

If the running time of an algorithm is given by:

$$n^2 \quad + \quad 100n \quad + \quad \log_{10}(n) \quad + \quad 1000$$

What is the *most* important term?

value of n	n^2	$100n$	$\log_{10}(n)$	1000
1	1	10^2	0	$\boxed{10^3}$
10	10^2	$\boxed{10^3}$	1	$\boxed{10^3}$
100	$\boxed{10^4}$	$\boxed{10^4}$	2	10^3
1000	$\boxed{10^6}$	10^5	3	10^3
100000	$\boxed{10^{10}}$	10^7	5	10^3

n^2 dominates
when the input
size is large!

$10^{10} \times 10^{-6}$
$=10^4$ seconds
2.78 Hours

+

$10^7 \times 10^{-6}$
10 sec

+

5×10^{-6}
0.000005 sec

+

$10^3 \times 10^{-6}$
0.001 sec

Lower Order Terms? Really?

Running time in seconds assuming each operation takes 10^{-6} seconds to execute.

order of growth	input size: 10	10^2	10^3	10^4	10^5
$\log_2(n)$	3.3×10^{-6}	6.6×10^{-6}	10^{-5}	1.3×10^{-5}	1.7×10^{-5}
$\sqrt{n}$	3.2×10^{-6}	10^{-5}	3.1×10^{-5}	10^{-4}	3.2×10^{-4}
n					
n^2					
n^3					
2^n					
$n!$					
n^n					

Lower Order Terms? Really?

Running time in seconds assuming each operation takes 10^{-6} seconds to execute.

input size

order of growth

	10	10^2	10^3	10^4	10^5
$\log_2(n)$	3.3×10^{-6}	6.6×10^{-6}	10^{-5}	1.3×10^{-5}	1.7×10^{-5}
$\sqrt{n}$	3.2×10^{-6}	10^{-5}	3.1×10^{-5}	10^{-4}	3.2×10^{-4}
n	10^{-5}	10^{-4}	0.001	0.01	0.1
n^2					
n^3					
2^n					
$n!$					
n^n					

Lower Order Terms? Really?

Running time in seconds assuming each operation takes 10^{-6} seconds to execute.

input size

order of growth

	10	10^2	10^3	10^4	10^5
$\log_2(n)$	3.3 x 10^{-6}	6.6 x 10^{-6}	10^{-5}	1.3 x 10^{-5}	1.7 x 10^{-5}
$\sqrt{n}$	3.2 x 10^{-6}	10^{-5}	3.1 x 10^{-5}	10^{-4}	3.2 x 10^{-4}
n	10^{-5}	10^{-4}	0.001	0.01	0.1
n^2	10^{-4}	0.01	1 *sec*	1.67 *sec*	2.78 *hr*
n^3					
2^n					
$n!$					
n^n					

Lower Order Terms? Really?

Running time in seconds assuming each operation takes 10^{-6} seconds to execute.

input size

order of growth

	10	10^2	10^3	10^4	10^5
$\log_2(n)$	3.3 x 10^{-6}	6.6 x 10^{-6}	10^{-5}	1.3 x 10^{-5}	1.7 x 10^{-5}
$\sqrt{n}$	3.2 x 10^{-6}	10^{-5}	3.1 x 10^{-5}	10^{-4}	3.2 x 10^{-4}
n	10^{-5}	10^{-4}	0.001	0.01	0.1
n^2	10^{-4}	0.01	1 *sec*	1.67 *sec*	2.78 *hr*
n^3	0.001	1 *sec*	16.7 *min*	11.6 *days*	31.7 *years*
2^n					
$n!$					
n^n					

Lower Order Terms? Really?

Running time in seconds assuming each operation takes 10^{-6} seconds to execute.

input size

order of growth

	10	10^2	10^3	10^4	10^5
$\log_2(n)$	3.3×10^{-6}	6.6×10^{-6}	10^{-5}	1.3×10^{-5}	1.7×10^{-5}
$\sqrt{n}$	3.2×10^{-6}	10^{-5}	3.1×10^{-5}	10^{-4}	3.2×10^{-4}
n	10^{-5}	10^{-4}	0.001	0.01	0.1
n^2	10^{-4}	0.01	1 *sec*	1.67 *sec*	2.78 *hr*
n^3	0.001	1 *sec*	16.7 *min*	11.6 *days*	31.7 *years*
2^n	0.001	4×10^{16} *years*	!!	!!	!!
$n!$	2.78 *hr*	3×10^{144} *years*	!!	!!	!!
n^n	42 *days*	!!	!!	!!	!!

Lesson # 2

When working with large input sizes, consider only the ***highest order term***.

Lesson # 2

When working with large input sizes, consider only the ***highest order term***.

! Drop all lower order terms and coefficients and express the running time using Big-O notation:

Example: $T(n) = 5n^2 + 3n + 1 \longrightarrow O(n^2)$

Example: $T(n) = 3n^3 + n\log_2(n) \longrightarrow O(n^3)$

Technically (not for this course):
The running time of the algorithm (as a function of the input size n) is bounded above (after some point) by a constant multiplied by n^2.

Lesson # 2

When working with large input sizes, consider only the ***highest order term***.

Informally (in this course):
The running time (as a function of the input size n) has cn^2 as the highest order term ($c > 0$ is a constant).

! Drop all lower order terms and coefficients and express the running time using Big-O notation:

Example: $T(n) = 5n^2 + 3n + 1 \longrightarrow O(n^2)$

Example: $T(n) = 3n^3 + n\log_2(n) \longrightarrow O(n^3)$

What is the order of growth as a function of *n*?

```
for (int i = 0; i < 100; i += 5)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < 2 * n; k++)
         op();
```

What is the order of growth as a function of *n*?

```
for (int i = 0; i < 100; i += 5)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < 2 * n; k++)
         op();
```

$O(1)$

$O(n)$

$O(n)$

$O(1) \times O(n) \times O(n) = O(n^2)$

What is the order of growth as a function of *n*?

```
for (int i = 0; i < 100; i += 5)              O(1)
   for (int j = 0; j < n; j++)                O(n)
      for (int k = 0; k < 2 * n; k++)         O(n)
         op();
```

$O(1) \times O(n) \times O(n) = O(n^2)$

```
for (int i = 0; i < 100; i += 5) {
   for (int j = 1; j < n; j += 2)
      op();
   for (int k = 0; k < 2 * n; k++)
      op();
   op();
}
```

What is the order of growth as a function of *n*?

```
for (int i = 0; i < 100; i += 5)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < 2 * n; k++)
         op();
```

$O(1)$

$O(n)$

$O(n)$

$O(1) \times O(n) \times O(n) = O(n^2)$

```
for (int i = 0; i < 100; i += 5) {
   for (int j = 1; j < n; j += 2)
      op();
   for (int k = 0; k < 2 * n; k++)
      op();
   op();
}
```

$O(1)$

$O(n)$

$O(n)$

$O(1) \times (O(n) + O(n) + O(1)) = O(n)$

What is the order of growth as a function of *n*?

```
for (int i = 0; i <= n; i += 2)
  for (int j = 1; j <= i; j++)
       op();
```

What is the order of growth as a function of *n*?

```
for (int i = 0; i <= n; i += 2)
   for (int j = 1; j <= i; j++)
         op();
```

i	j	*number of* `op()` *calls*
0	–	0
2	[1, 2]	2
4	[1 ⟶ 4]	4
6	[1 ⟶ 6]	6
...	...	...
n	[1 ⟶ n]	n

What is the order of growth as a function of *n*?

```
for (int i = 0; i <= n; i += 2)
   for (int j = 1; j <= i; j++)
         op();
```

i	j	*number of* op() *calls*
0	–	0
2	[1, 2]	2
4	[1 ⟶ 4]	4
6	[1 ⟶ 6]	6
...	...	...
n	[1 ⟶ n]	n

Total $= 0 + 2 + 4 + 6 + \ldots + n$

What is the order of growth as a function of *n*?

```
for (int i = 0; i <= n; i += 2)
   for (int j = 1; j <= i; j++)
         op();
```

i	j	*number of* op() *calls*
0	–	0
2	[1, 2]	2
4	[1 ⟶ 4]	4
6	[1 ⟶ 6]	6
...	...	...
n	[1 ⟶ n]	n

$$\text{Total} = 0 + 2 + 4 + 6 + \ldots + n$$

$$= 2 \times (0 + 1 + 2 + 3 + \ldots + \tfrac{1}{2}n)$$

What is the order of growth as a function of *n*?

```
for (int i = 0; i <= n; i += 2)
   for (int j = 1; j <= i; j++)
         op();
```

i	j	*number of* op() *calls*
0	–	0
2	[1, 2]	2
4	[1 ⟶ 4]	4
6	[1 ⟶ 6]	6
...	...	...
n	[1 ⟶ n]	n

$$\text{Total} = 0 + 2 + 4 + 6 + \ldots + n$$

$$= 2 \times (0 + 1 + 2 + 3 + \ldots + \tfrac{1}{2}n) \;=\; 2 \times \sum_{i=0}^{n/2} i \;=$$

What is the order of growth as a function of *n*?

```
for (int i = 0; i <= n; i += 2)
   for (int j = 1; j <= i; j++)
         op();
```

i	j	*number of* op() *calls*
0	–	0
2	[1, 2]	2
4	[1 ⟶ 4]	4
6	[1 ⟶ 6]	6
...	...	...
n	[1 ⟶ n]	n

$$\text{Total} = 0 + 2 + 4 + 6 + \ldots + n$$

$$= 2 \times (0 + 1 + 2 + 3 + \ldots + \tfrac{1}{2}n) = 2 \times \sum_{i=0}^{n/2} i = 2 \times \frac{\frac{1}{2}n(\frac{1}{2}n + 1)}{2}$$

What is the order of growth as a function of *n*?

```
for (int i = 0; i <= n; i += 2)
   for (int j = 1; j <= i; j++)
         op();
```

i	j	*number of* op() *calls*
0	–	0
2	[1, 2]	2
4	[1 ⟶ 4]	4
6	[1 ⟶ 6]	6
...	...	...
n	[1 ⟶ n]	n

$$\text{Total} = 0 + 2 + 4 + 6 + \ldots + n$$

$$= 2 \times (0 + 1 + 2 + 3 + \ldots + \tfrac{1}{2}n) = 2 \times \sum_{i=0}^{n/2} i = 2 \times \frac{\frac{1}{2}n(\frac{1}{2}n + 1)}{2} = O(n^2)$$

Which function grows faster?

$T(n) = n^5$

$H(n) = n^5 + n^4 + n^3 + n^2 + n$

Which function grows faster?

$T(n) = n^5$ same $H(n) = n^5 + n^4 + n^3 + n^2 + n$

Which function grows faster?

$T(n) = n^5$	same	$H(n) = n^5 + n^4 + n^3 + n^2 + n$
$T(n) = 2^n$		$H(n) = 2^{n+1}$

Which function grows faster?

$T(n) = n^5$	same	$H(n) = n^5 + n^4 + n^3 + n^2 + n$
$T(n) = 2^n$	same	$H(n) = 2^{n+1} = 2^1 \times 2^n = O(2^n)$

Which function grows faster?

$T(n) = n^5$	same	$H(n) = n^5 + n^4 + n^3 + n^2 + n$
$T(n) = 2^n$	same	$H(n) = 2^{n+1}$
$T(n) = \log_2(n^2)$		$H(n) = (\log_2(n))^2$

Which function grows faster?

$T(n) = n^5$	same	$H(n) = n^5 + n^4 + n^3 + n^2 + n$
$T(n) = 2^n$	same	$H(n) = 2^{n+1}$
$T(n) = \log_2(n^2) = 2\log_2(n)$		$H(n) = (\log_2(n))^2$

Which function grows faster?

$T(n) = n^5$	same	$H(n) = n^5 + n^4 + n^3 + n^2 + n$
$T(n) = 2^n$	same	$H(n) = 2^{n+1}$
$T(n) = \log_2(n^2)$		$H(n) = (\log_2(n))^2$
$T(n) = \log_2(n)$		$H(n) = \log_{10}(n)$

Which function grows faster?

$T(n) = n^5$	same	$H(n) = n^5 + n^4 + n^3 + n^2 + n$
$T(n) = 2^n$	same	$H(n) = 2^{n+1}$
$T(n) = \log_2(n^2)$		$H(n) = (\log_2(n))^2$
$T(n) = \log_2(n)$	same	$H(n) = \log_{10}(n) = \frac{\log_2(n)}{\log_2(10)}$

Which function grows faster?

$T(n) = n^5$	same	$H(n) = n^5 + n^4 + n^3 + n^2 + n$
$T(n) = 2^n$	same	$H(n) = 2^{n+1}$
$T(n) = \log_2(n^2)$		$H(n) = (\log_2(n))^2$
$T(n) = \log_2(n)$	same	$H(n) = \log_{10}(n)$
$T(n) = \log_2(n)$		$H(n) = \sqrt{n}$

Which function grows faster?

$T(n) = n^5$	same	$H(n) = n^5 + n^4 + n^3 + n^2 + n$
$T(n) = 2^n$	same	$H(n) = 2^{n+1}$
$T(n) = \log_2(n^2)$		$H(n) = (\log_2(n))^2$
$T(n) = \log_2(n)$	same	$H(n) = \log_{10}(n)$
$T(n) = \log_2(n)$		$H(n) = \sqrt{n} = n^{0.5}$

Do the math!

? The speed of a machine is 10^6 operations per second. Given an algorithm that performs $\sim n \lg n$ operations, how much time will this algorithm (roughly) require if we run it on an input size of $n = 1000$?

Do the math!

? The speed of a machine is 10^6 operations per second. Given an algorithm that performs $\sim n \lg n$ operations, how much time will this algorithm (roughly) require if we run it on an input size of $n = 1000$?

$$\text{speed} = \frac{\text{number of operations}}{\text{time}}$$

$$10^6 = \frac{1000 \times \lg(1000)}{\text{time}}$$

Do the math!

? The speed of a machine is 10^6 operations per second. Given an algorithm that performs $\sim n \lg n$ operations, how much time will this algorithm (roughly) require if we run it on an input size of $n = 1000$?

$$\text{speed} = \frac{\text{number of operations}}{\text{time}}$$

$$10^6 = \frac{1000 \times \lg(1000)}{\text{time}}$$

$$\text{time} = \frac{1000 \times \lg(1000)}{10^6}$$

Do the math!

? The speed of a machine is 10^6 operations per second. Given an algorithm that performs $\sim n \lg n$ operations, how much time will this algorithm (roughly) require if we run it on an input size of $n = 1000$?

$$\text{speed} = \frac{\text{number of operations}}{\text{time}}$$

$$10^6 = \frac{1000 \times \lg(1000)}{\text{time}}$$

$$\text{time} = \frac{1000 \times \lg(1000)}{10^6} = \frac{\lg(1000)}{10^3} \approx 0.01 \text{ sec}$$

Do the math!

An algorithm that performs $\sim\sqrt{n}$ operations takes 10^{-4} seconds to run with an input of size $n = 10^4$. How long is this algorithm expected to take if the input size is $n = 10^8$?

Do the math!

? An algorithm that performs $\sim\sqrt{n}$ operations takes 10^{-4} seconds to run with an input of size $n = 10^4$. How long is this algorithm expected to take if the input size is $n = 10^8$?

$$\text{speed} = \frac{\text{number of operations}}{\text{time}}$$

$$\text{speed} = \frac{\sqrt{10^4}}{10^{-4}} = 10^6 \text{ operations per second}$$

Do the math!

?

An algorithm that performs $\sim \sqrt{n}$ operations takes 10^{-4} seconds to run with an input of size $n = 10^4$. How long is this algorithm expected to take if the input size is $n = 10^8$?

$$\text{speed} = \frac{\text{number of operations}}{\text{time}}$$

$$\text{speed} = \frac{\sqrt{10^4}}{10^{-4}} = 10^6 \text{ operations per second}$$

$$10^6 = \frac{\sqrt{10^8}}{\text{time}}$$

Do the math!

? An algorithm that performs $\sim\sqrt{n}$ operations takes 10^{-4} seconds to run with an input of size $n = 10^4$. How long is this algorithm expected to take if the input size is $n = 10^8$?

$$\text{speed} = \frac{\text{number of operations}}{\text{time}}$$

$$\text{speed} = \frac{\sqrt{10^4}}{10^{-4}} = 10^6 \text{ operations per second}$$

$$10^6 = \frac{\sqrt{10^8}}{\text{time}} \quad \longrightarrow \quad \text{time} = \frac{\sqrt{10^8}}{10^6} = 0.01 \text{ sec}$$

Do the math!

?

An algorithm that performs $\sim n^2$ operations required 10 seconds to run on a machine that performs 10^7 operations per second. How much time is an algorithm that performs $\sim n^3$ operations expected to take when run on the same machine and the same input?

$$\text{speed} = \frac{\text{number of operations}}{\text{time}}$$

Do the math!

? An algorithm that performs $\sim n^2$ operations required 10 seconds to run on a machine that performs 10^7 operations per second. How much time is an algorithm that performs $\sim n^3$ operations expected to take when run on the same machine and the same input?

$$\text{speed} = \frac{\text{number of operations}}{\text{time}}$$

$$10^7 = \frac{n^2}{10}$$

find the input size on which the n^2 algorithm took 10 seconds

Do the math!

? An algorithm that performs $\sim n^2$ operations required 10 seconds to run on a machine that performs 10^7 operations per second. How much time is an algorithm that performs $\sim n^3$ operations expected to take when run on the same machine and the same input?

$$\text{speed} = \frac{\text{number of operations}}{\text{time}}$$

$$10^7 = \frac{n^2}{10} \longrightarrow n^2 = 10 \times 10^7 \longrightarrow n = 10^4$$

find the input size on which the n^2 algorithm took 10 seconds

Do the math!

? An algorithm that performs $\sim n^2$ operations required 10 seconds to run on a machine that performs 10^7 operations per second. How much time is an algorithm that performs $\sim n^3$ operations expected to take when run on the same machine and the same input?

$$\text{speed} = \frac{\text{number of operations}}{\text{time}}$$

$$10^7 = \frac{n^2}{10} \longrightarrow n^2 = 10 \times 10^7 \longrightarrow n = 10^4$$

find the input size on which the n^2 algorithm took 10 seconds

$$10^7 = \frac{n^3}{\text{time}} \longrightarrow \text{time} = \frac{(10^4)^3}{10^7} = 10^5 \text{ sec} \approx 27.8 \text{ hours}$$

use the computed input size to find the time taken by the n^3 algorithm