MONASH College

MCD4710

Introduction to algorithms and programming

Lecture 11

Computational Complexity

Different algorithms can solve the same problem but at vastly different costs

```
def gcd_brute_force(m, n):
    """Input : integers m and n such that not n==m==0
       Output: the greatest common divisor of m and n
    """
    x = min(m, n)
    while not (m % x == 0 and n % x == 0):
        x = x - 1
    return x
```

```
def gcd_euclid(m, n):
    """
    Input : integers m and n such that not n==m==0
    Output: the greatest common divisor of m and n
    """
    while n != 0:
        m, n = n, m % n
    return m
```

Objectives

Objectives of this lecture are to get familiar with:

1. The concept of computational complexity
2. Asymptotic analysis of order of growth: Big-Oh Notation
3. Complexity analysis of simple algorithms
4. The computational complexity of Selection Sort and Insertion Sort

This covers learning outcomes:

- 5 – Determine the computational cost and limitations of algorithms

Overview

1. Computational Complexity
2. Asymptotic Analysis (Big-Oh notation)
3. Application to Sorting Algorithms

How long does a program run?

Depends on:

1. the machine used to execute the program
2. the input (runs longer for larger inputs)
3. the computational complexity of the algorithm

Definition

The computational (time) complexity of an algorithm is the *number of elementary steps* $T(n)$ needed for computing its output for an input *of a size* n.

What is an *elementary step*?

Counting elementary steps

Fix *cost model* for set of basic instructions

`x = 8 % 2`	1
`a = [10, 2, 10] + [5, 2, 27, 18]`	7
`len(a)`	0
`x == 0`	1

Cost model for basic instructions

Instruction	Examples	Cost
assignment	x = 1, return x	0
int/float/bool creation*	1, -3.0, True	0
range creation*	range(i,j,s)	0
list/string creation	list(seq)	len(seq)

*objects with fixed (in memory) size

Instruction	Examples	Cost
numerical arithmetic	1.0 + 324, -1**231	1
Boolean operation	True or False	1
numerical comparison	10 >= -3.5	1

Instruction	Example	Cost
list/string slicing	a[i:j:s]	len(a[i:j:s])
list/string concatenation	a + b	len(a) + len(b)
list augmentation	a+=b	len(b)
getting size of str/list/set/range	len(a)	0

Counting elementary steps

Fix cost model for set of basic instructions

`x = 8 % 2`	1
`a = [10, 2, 10] + [5, 2, 27, 18]`	7
`len(a)`	0
`x == 0`	1

Break down complex instructions to determine their cost

`not (18 % 3 == 0 and 10 % 3 == 0)`	6
`price_with_gst(27+3)`	3

Determine total cost of program line by analysing how often and in what states it is executed:

`a = []`	0

```
while len(a)<5:
    a = a + [len(a)]
```

Counting elementary steps

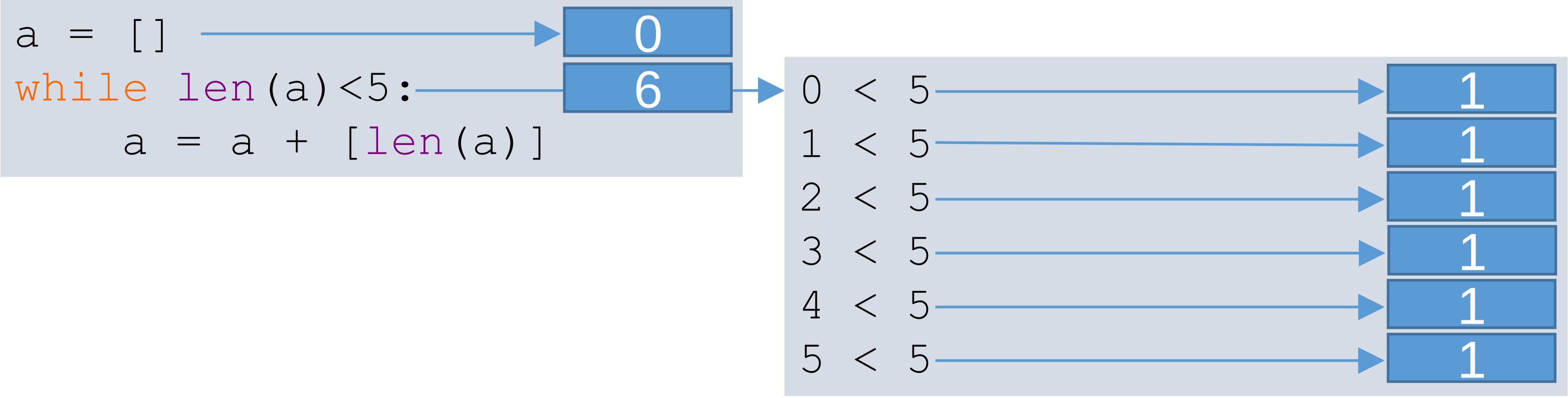
Model cost of elementary instructions

`x = 8 % 2`	1
`a = [10, 2, 10] + [5, 2, 27, 18]`	7
`len(a)`	0
`x == 0`	1

Break down complex instructions to determine their cost

`not (18 % 3 == 0 and 10 % 3 == 0)`	6
`price_with_gst(27+3)`	3

Determine total cost of program line by analysing how often and in what states it is executed:



```
a = []
while len(a)<5:
    a = a + [len(a)]
```

`a = []`	0
`while len(a)<5:`	6

`0 < 5`	1
`1 < 5`	1
`2 < 5`	1
`3 < 5`	1
`4 < 5`	1
`5 < 5`	1

Counting elementary steps

Model cost of elementary instructions

`x = 8 % 2`	1
`a = [10, 2, 10] + [5, 2, 27, 18]`	7
`len(a)`	0
`x == 0`	1

Break down complex instructions to determine their cost

`not (18 % 3 == 0 and 10 % 3 == 0)`	6
`price_with_gst(27+3)`	3

Determine total cost of program line by analysing how often and in what states it is executed:

`a = []`	0
`while len(a)<5:`	6
`    a = a + [len(a)]`	15

`a = [] + [0]`	1
`a = [0] + [1]`	2
`a = [0,1] + [2]`	3
`a = [0,1,2] + [3]`	4
`a = [0,1,2,3] + [4]`	5

In general number of steps depend on input

```
def power(x, n):
    """I: number x and pos. integer n
       O: x to the power of n"""
    value = 1
    k = 1
    while k <= n:
        value *= x
        k += 1
    return value
```

According to our model, what is the number of steps for power(2, 5)?

A. 16

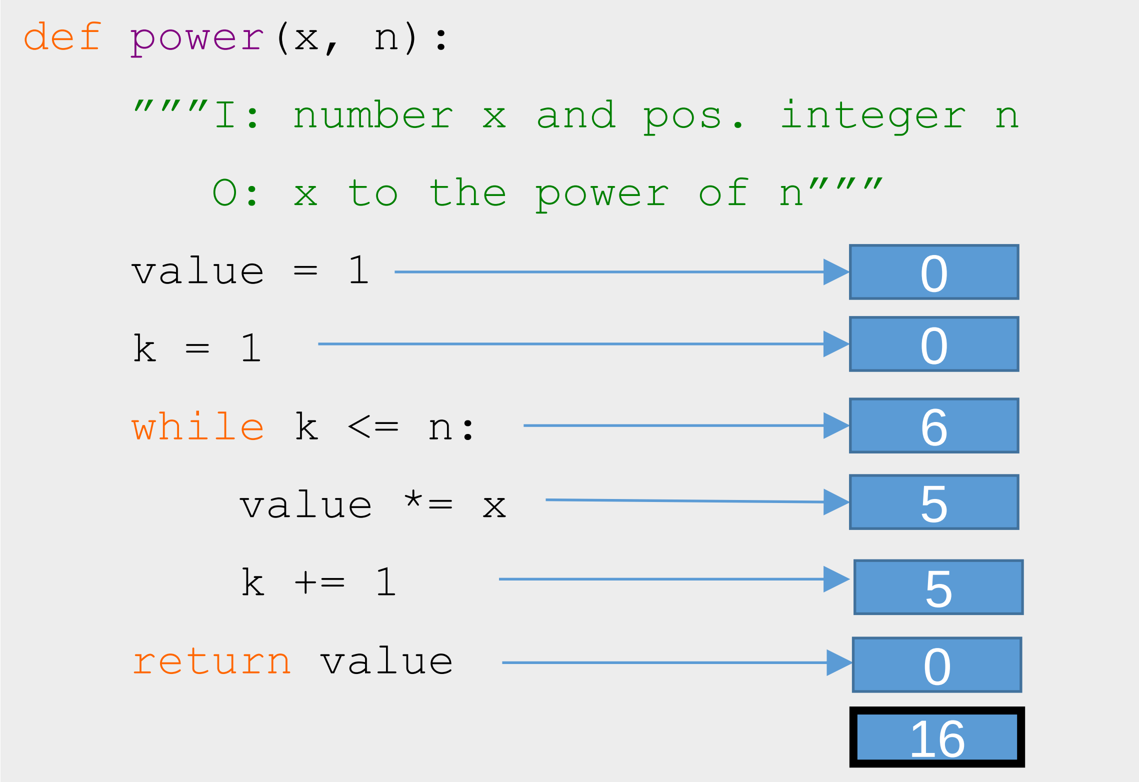
B. 15

C. 6

D. 19

1. Visit https://flux.qa
2. Log in using your Authcate details (not required if you're already logged in to Monash)
3. Touch the + symbol and enter the code: UF7BD9
4. Answer questions when they pop up.

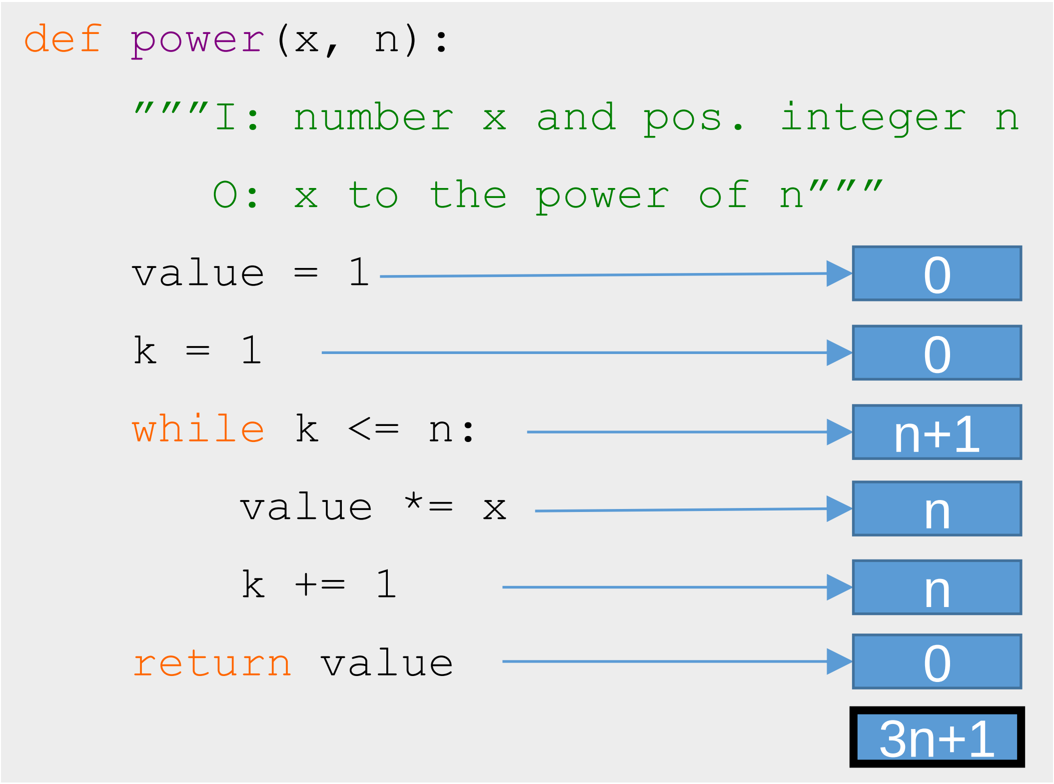
In general number of steps depend on input



```
def power(x, n):
    ”””I: number x and pos. integer n
       O: x to the power of n”””
    value = 1
    k = 1
    while k <= n:
        value *= x
        k += 1
    return value
```

According to model, what is the number of steps for power(2, 5)?

In general number of steps depend on input



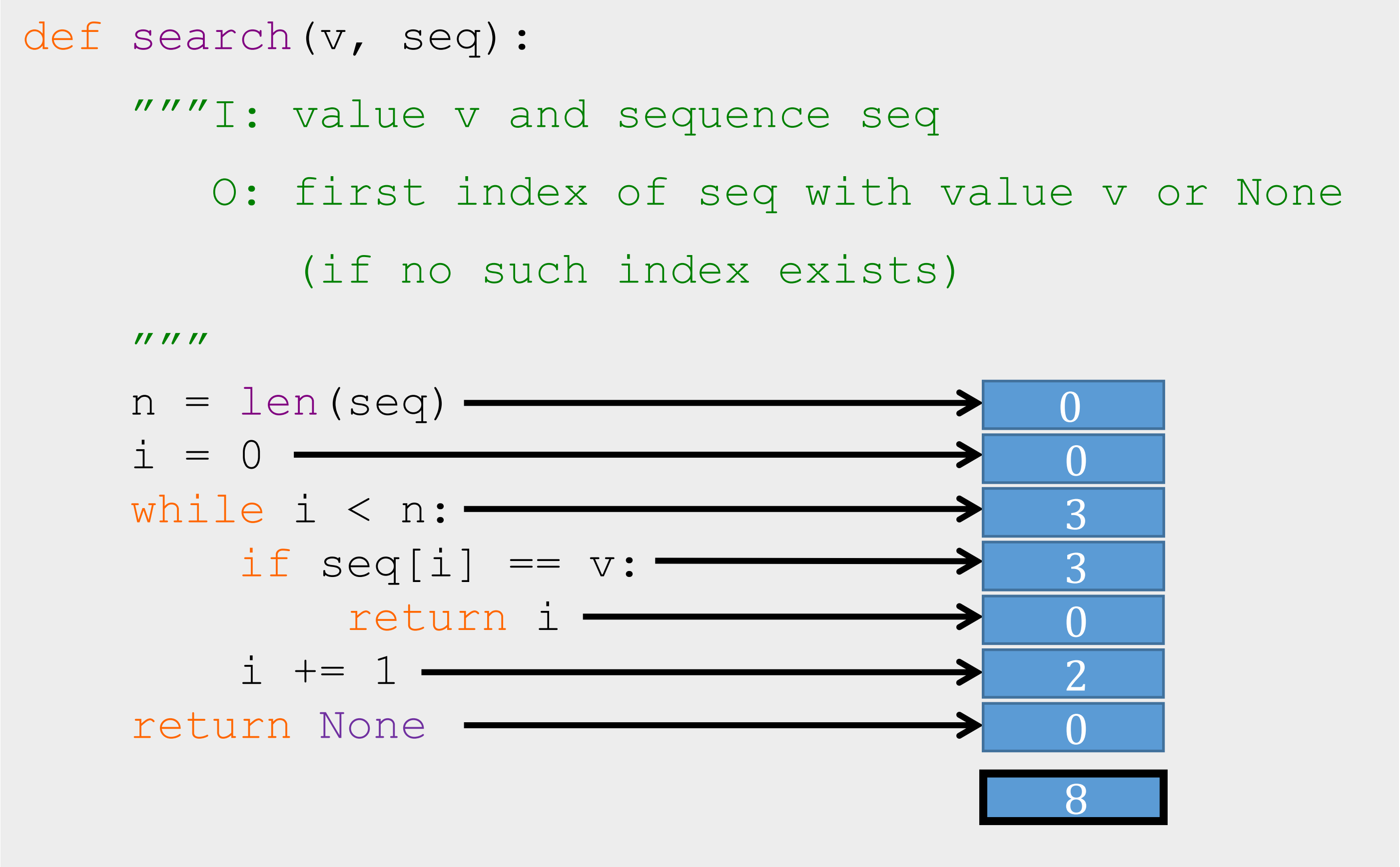
```
def power(x, n):
    """I: number x and pos. integer n
       O: x to the power of n"""
    value = 1
    k = 1
    while k <= n:
        value *= x
        k += 1
    return value
```

What is the number of steps in general for power(x, n)?

Time complexity of power: $T(n) = 3n + 1$

Disclaimer: here, we regarded parameter n as input size for sake of simplicity (usually other choices for numeric algorithms; see FIT1008)

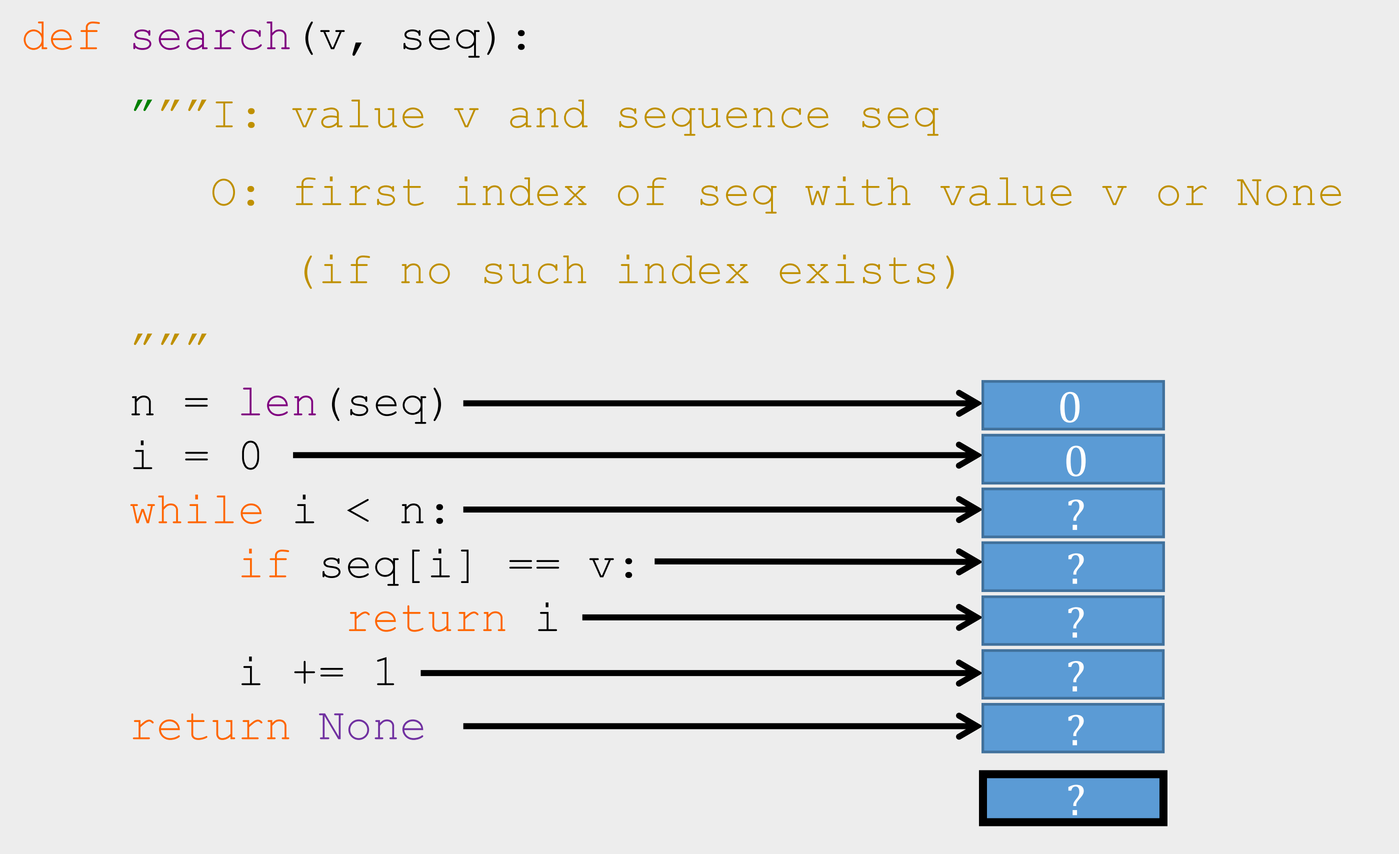
Next example: sequential search



```
def search(v, seq):
    """I: value v and sequence seq
       O: first index of seq with value v or None
          (if no such index exists)
    """
    n = len(seq)
    i = 0
    while i < n:
        if seq[i] == v:
            return i
        i += 1
    return None
```

What is the number of steps for `search(2,range(5))`?

Next example: sequential search



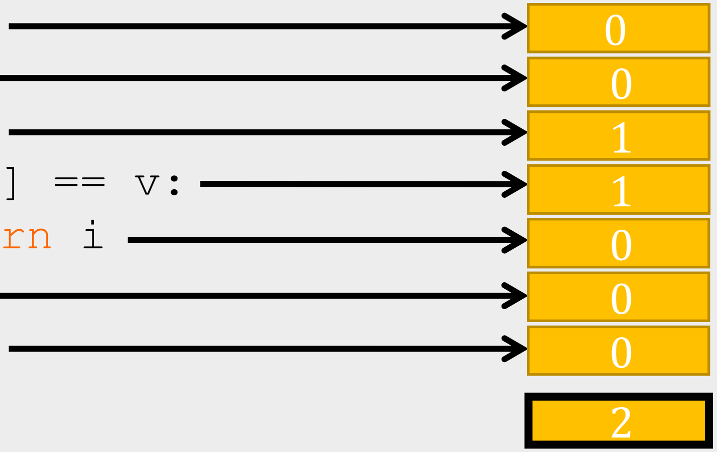
```
def search(v, seq):
    """I: value v and sequence seq
       O: first index of seq with value v or None
          (if no such index exists)
    """
    n = len(seq)
    i = 0
    while i < n:
        if seq[i] == v:
            return i
        i += 1
    return None
```

What is the number of steps in general for input `v`, `seq`?
(consider input size $n = \mathrm{len}(\mathrm{seq})$)

Number of steps not equal for all inputs of size n !!

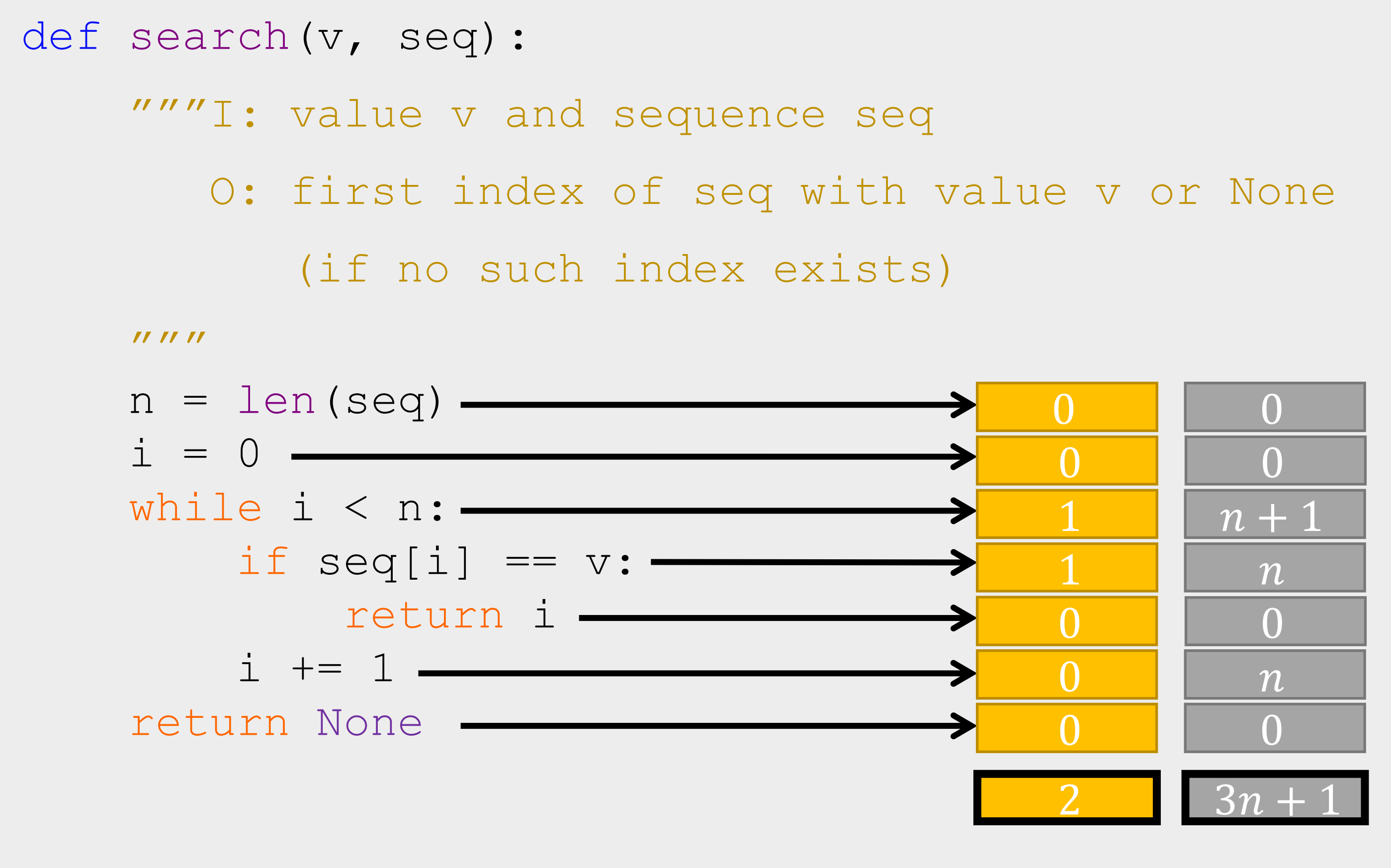
Best case analysis

```
def search(v, seq):
    """I: value v and sequence seq
       O: first index of seq with value v or None
          (if no such index exists)
    """
    n = len(seq)
    i = 0
    while i < n:
        if seq[i] == v:
            return i
        i += 1
    return None
```



Best case (fewest steps): `v==seq[0]`

Worst case analysis



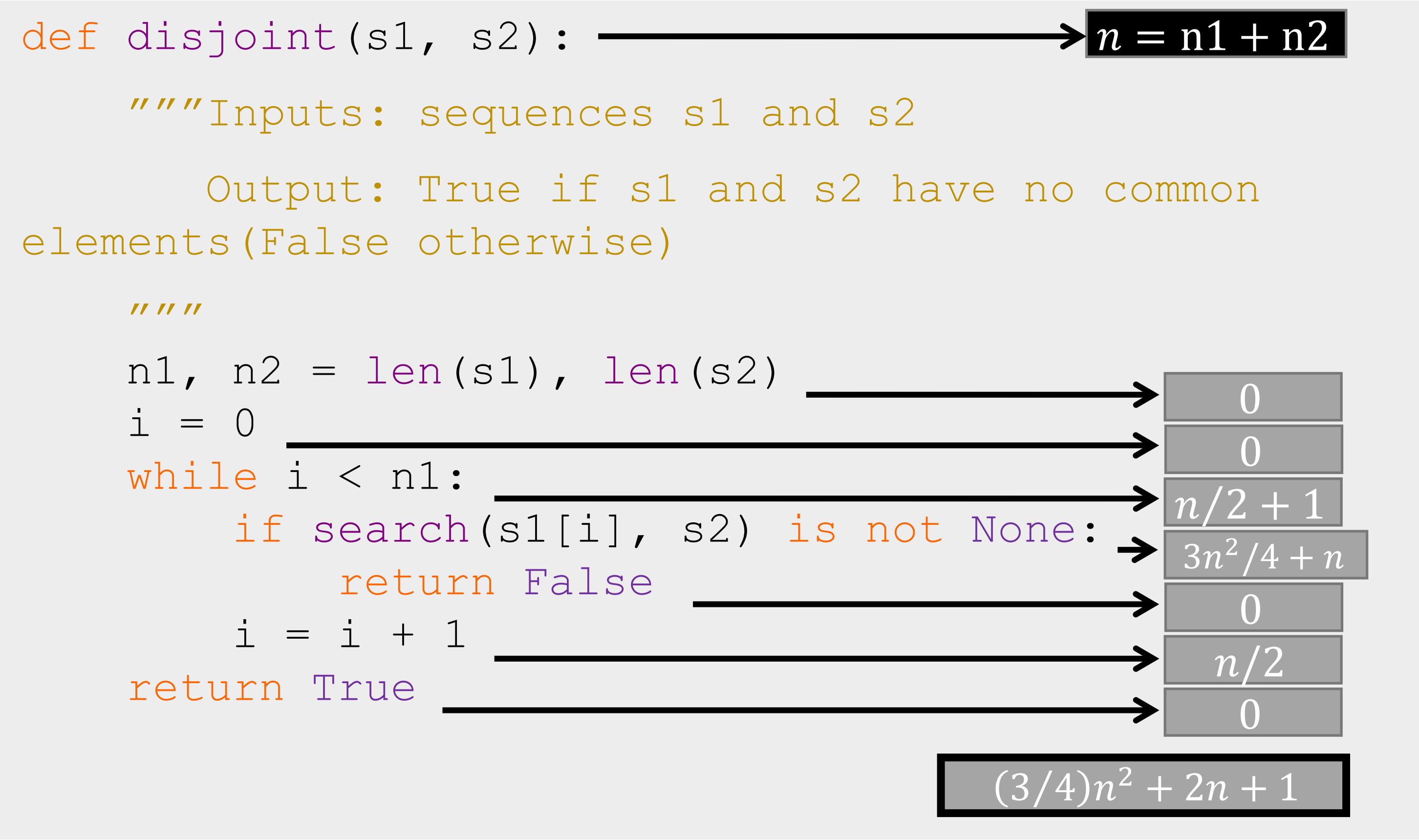
```
def search(v, seq):
    """I: value v and sequence seq
       O: first index of seq with value v or None
          (if no such index exists)
    """
    n = len(seq)
    i = 0
    while i < n:
        if seq[i] == v:
            return i
        i += 1
    return None
```

`n = len(seq)`	0	0
`i = 0`	0	0
`while i < n:`	1	$n+1$
`if seq[i] == v:`	1	n
`return i`	0	0
`i += 1`	0	n
`return None`	0	0
	2	$3n+1$

Best case (fewest steps): `v==seq[0]`

Worst case (most steps): `v not in seq`

Another example: repeated search



```
def disjoint(s1, s2):
    """Inputs: sequences s1 and s2
       Output: True if s1 and s2 have no common
elements(False otherwise)
    """
    n1, n2 = len(s1), len(s2)
    i = 0
    while i < n1:
        if search(s1[i], s2) is not None:
            return False
        i = i + 1
    return True
```

$n = \mathrm{n1} + \mathrm{n2}$

0

0

$n/2+1$

$3n^2/4+n$

0

$n/2$

0

$(3/4)n^2+2n+1$

What is input size?

Worst case: no common element and `n1=n2=n/2`

Total cost of search: $\frac{n}{2} T_{\text{search}}\left(\frac{n}{2}\right) = \frac{n}{2}\left(\frac{3}{2}n+1+1\right) = \frac{3}{4}n^2+n$

Intermediate Summary

Computational complexity

- Express number of computational steps as a function of input size (independent of computer and specific input)
- Need to decide how to measure input size (what is n?)
- Have to decide whether to analyse worst case or best case

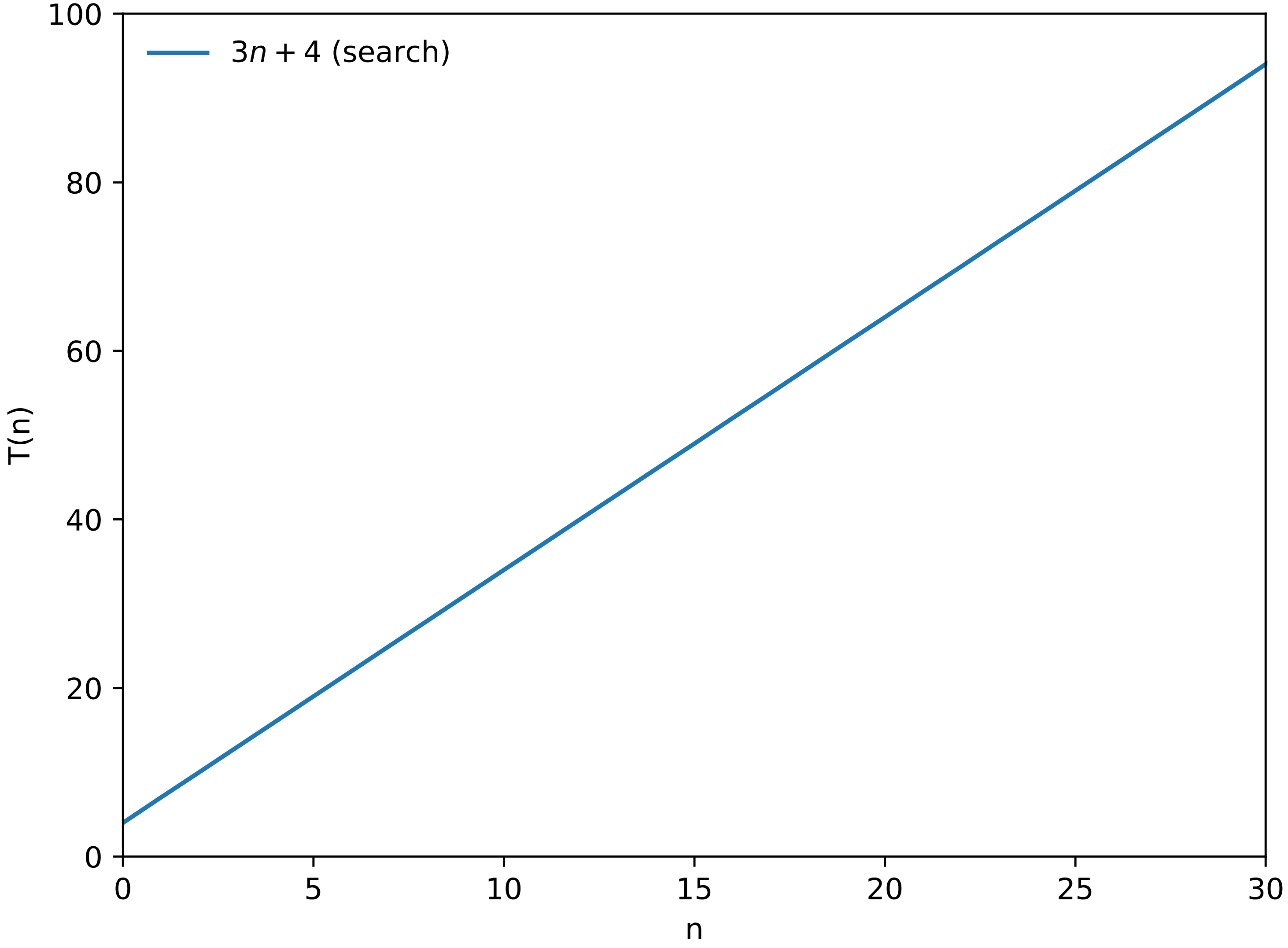
Observations

- Model assumptions of what constitutes single step are either a bit complicated or a bit arbitrary
- Even with simplifying assumptions: analysis is quite tedious

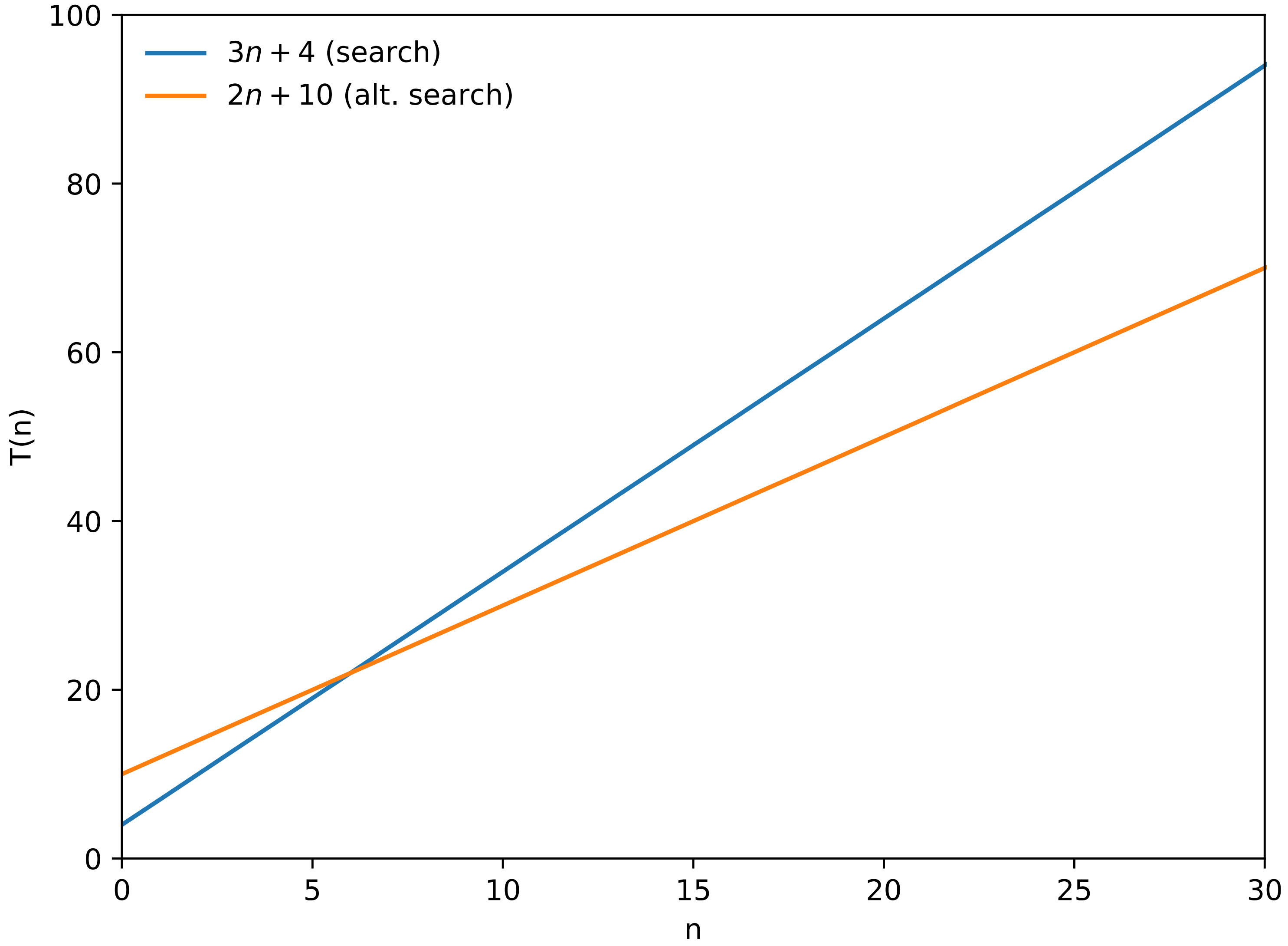
Overview

1. Computational Complexity
2. Asymptotic Analysis (Big-Oh notation)
3. Application to Sorting Algorithms

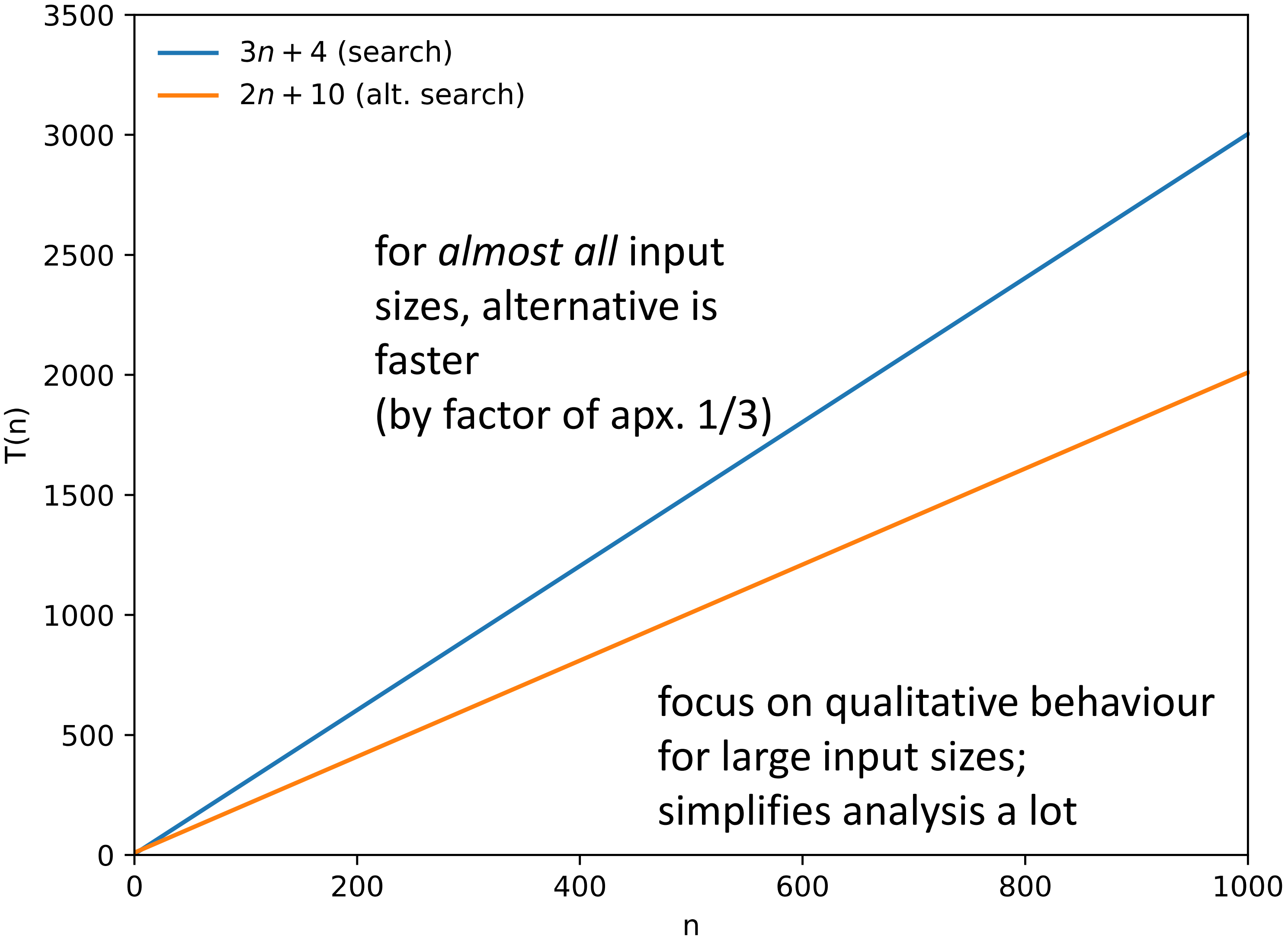
Let’s plot time complexity of `search` function



Assume alternative algorithm



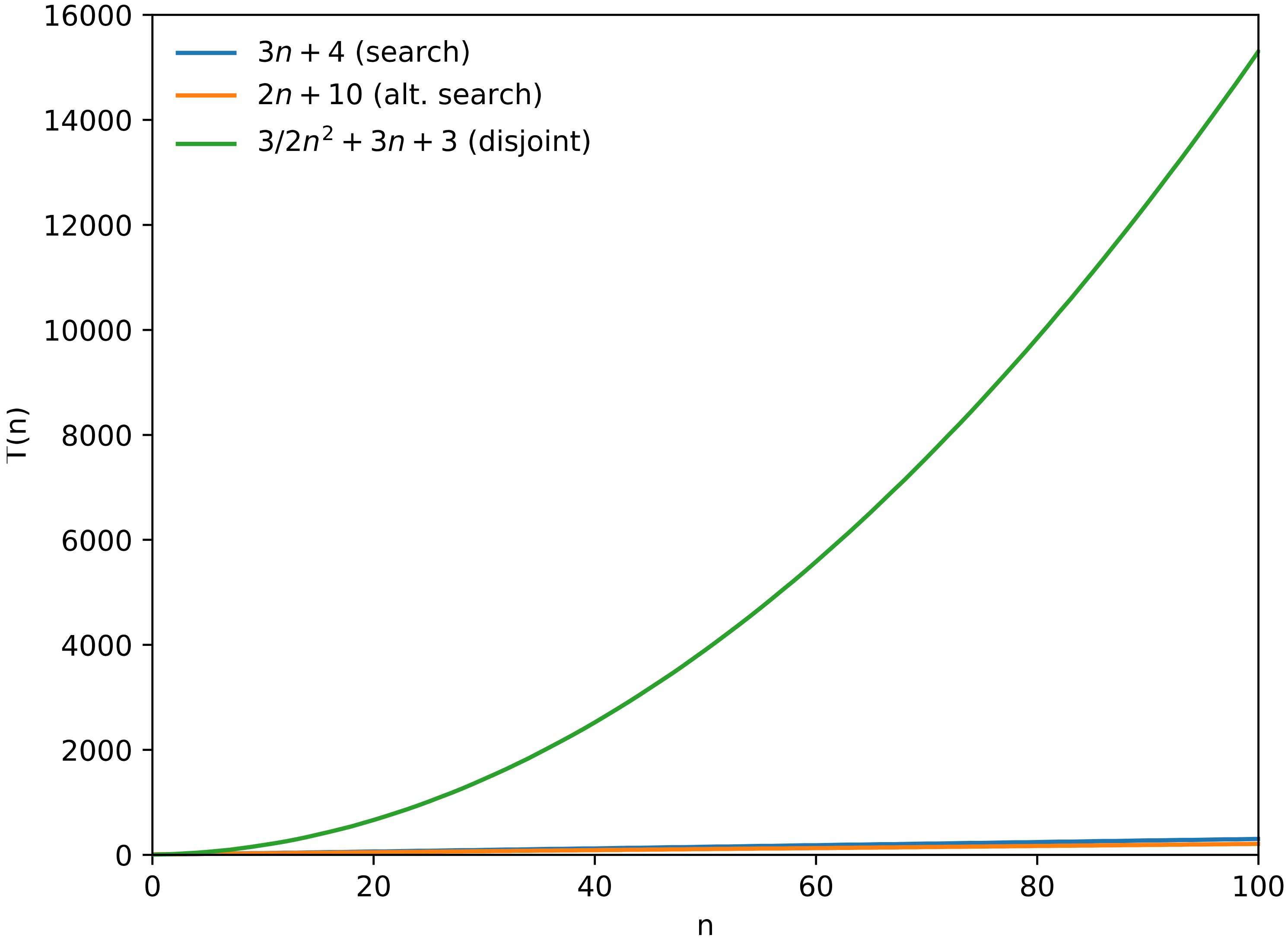
Assume alternative algorithm



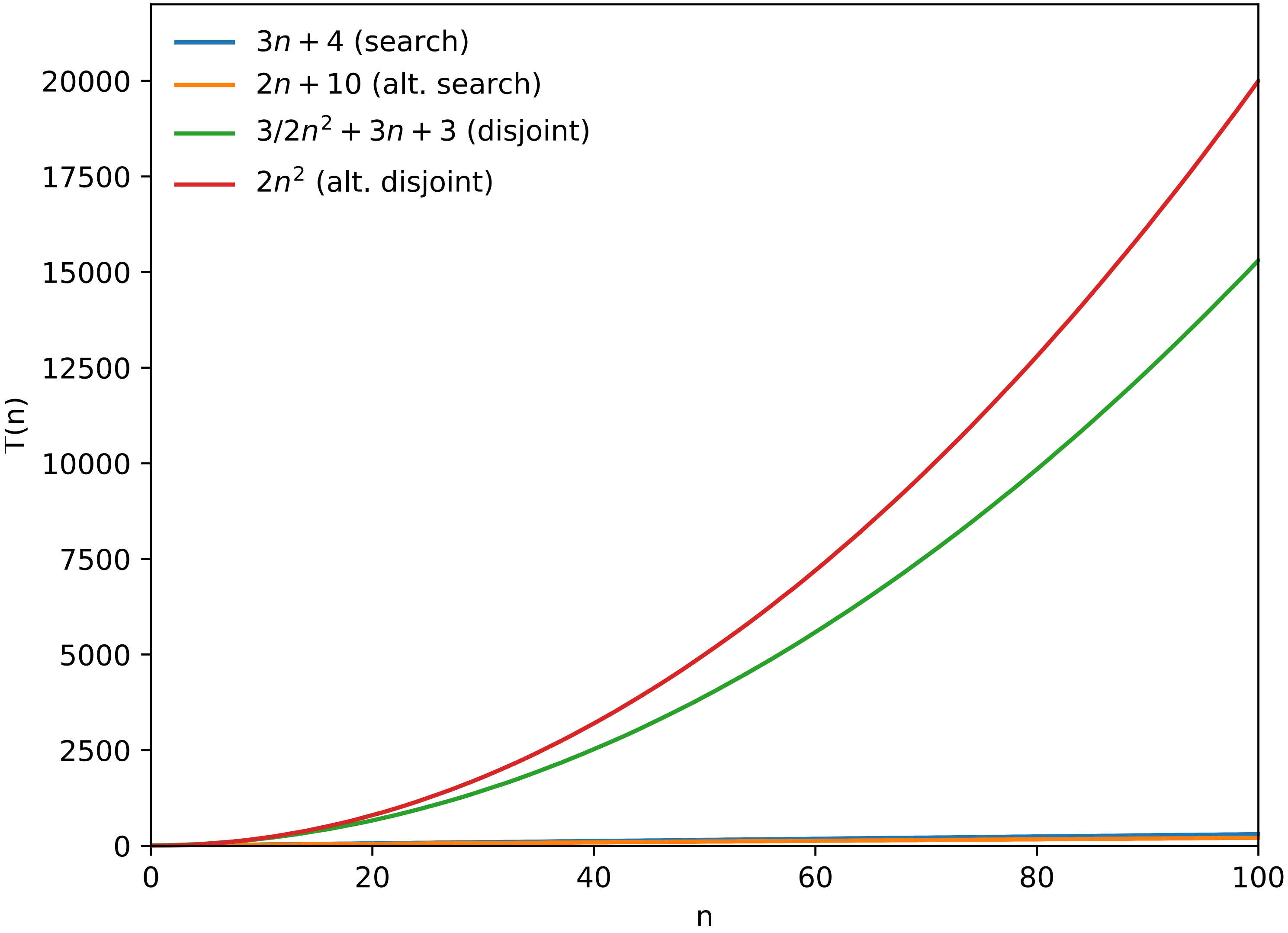
for *almost all* input sizes, alternative is faster
(by factor of apx. 1/3)

focus on qualitative behaviour for large input sizes; simplifies analysis a lot

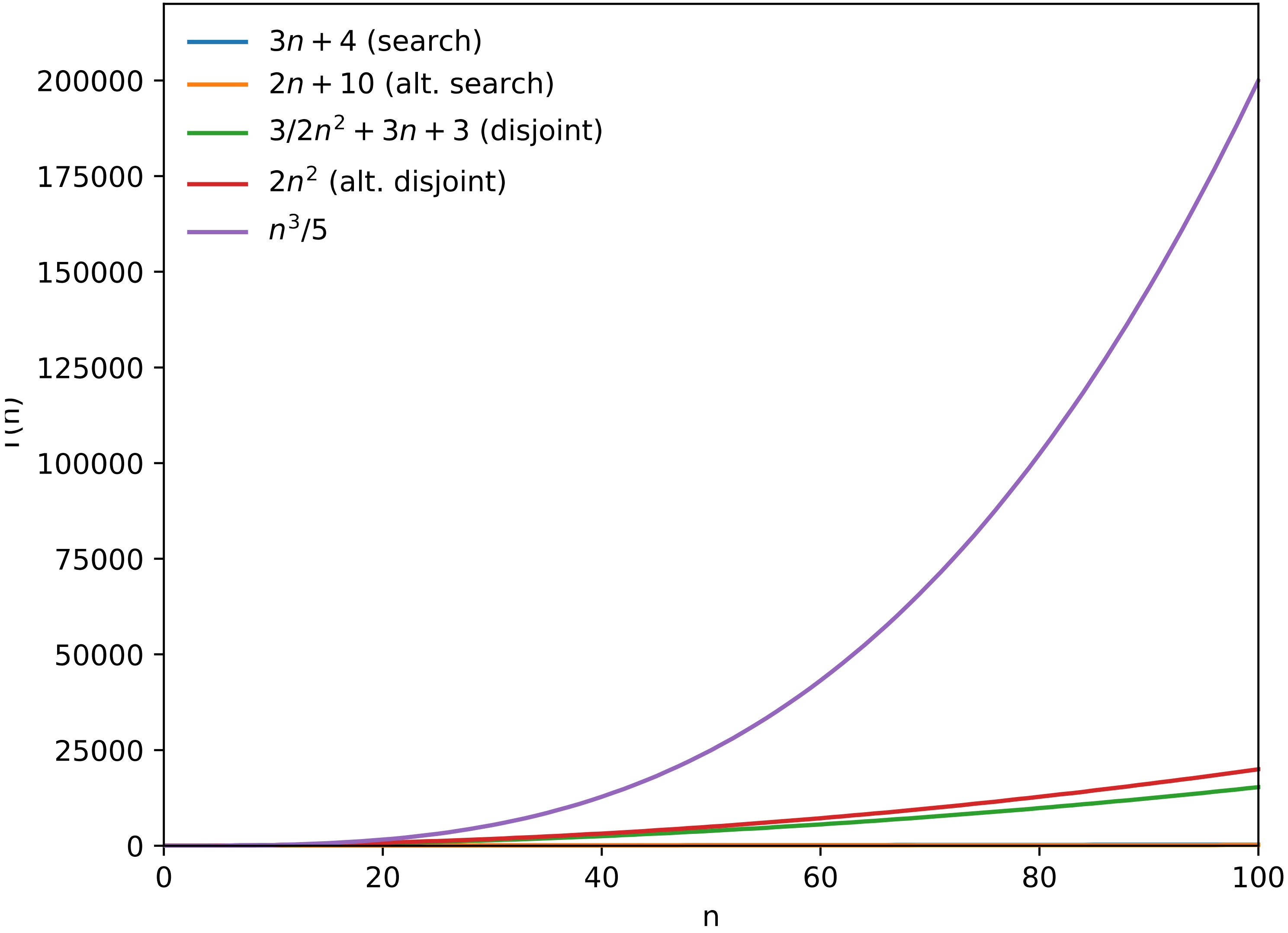
Let’s compare both to `disjoint`



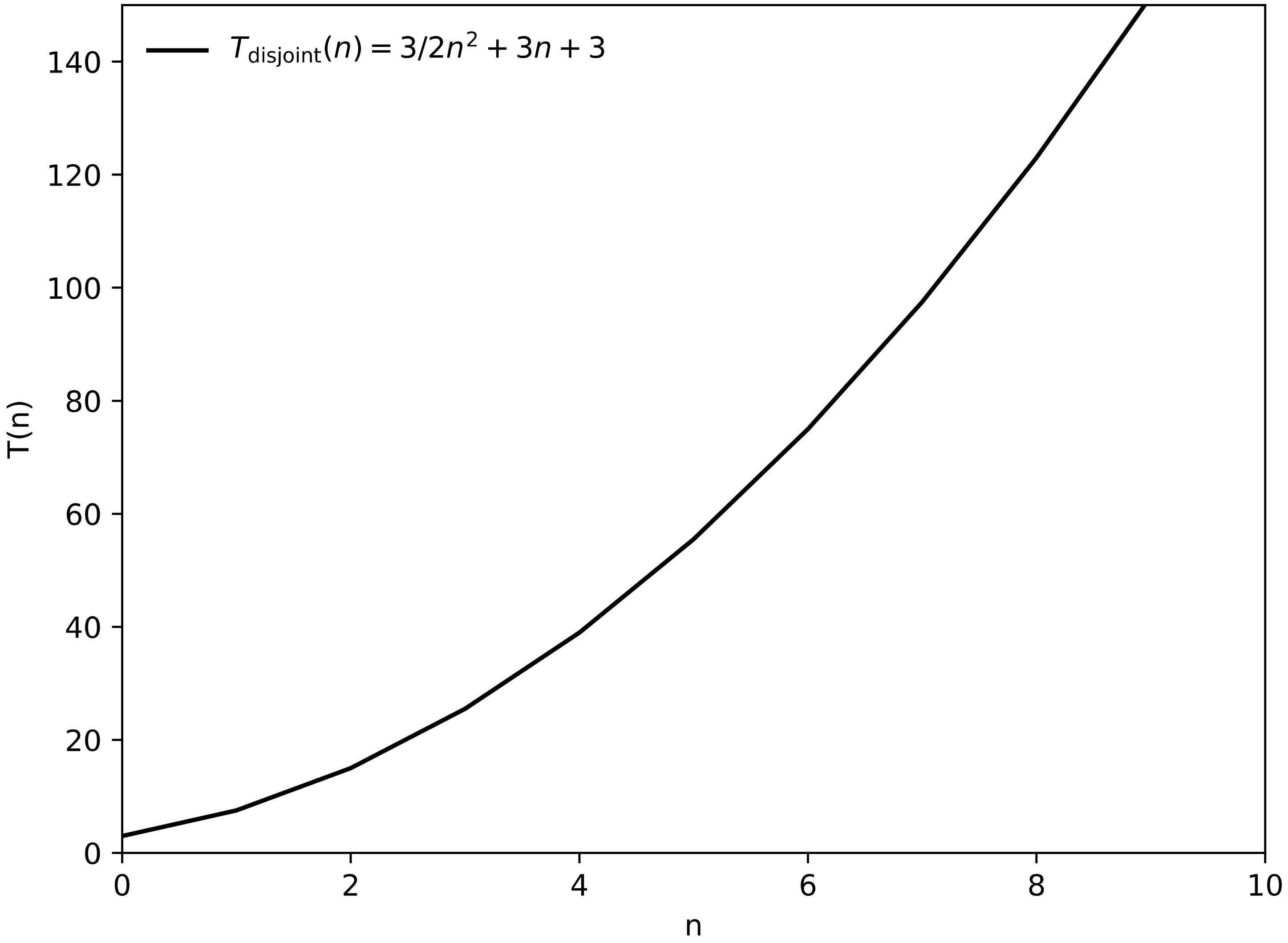
Only coefficient of largest growth term is relevant for large input sizes



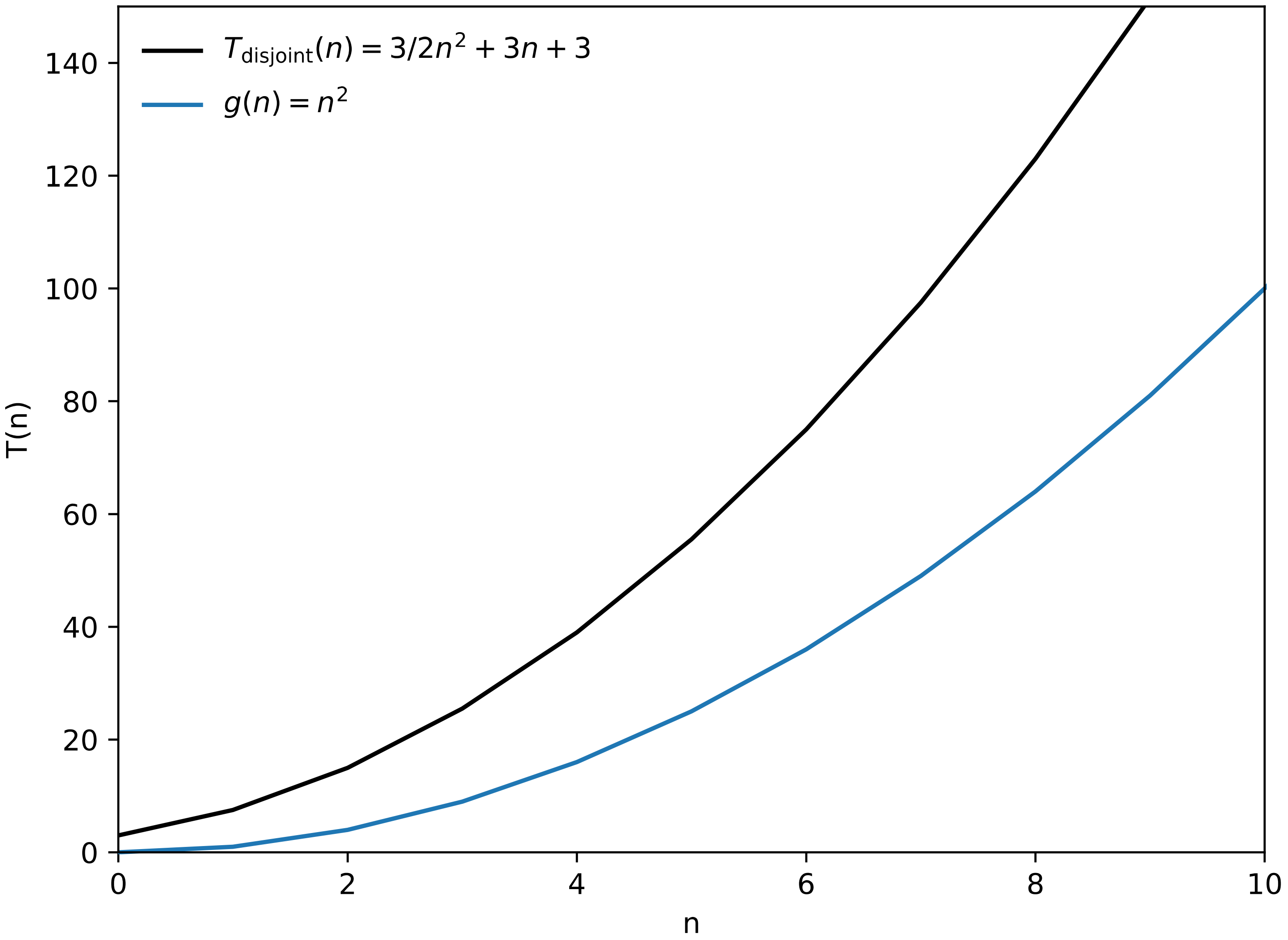
…but also negligible when comparing to higher *order of growth*



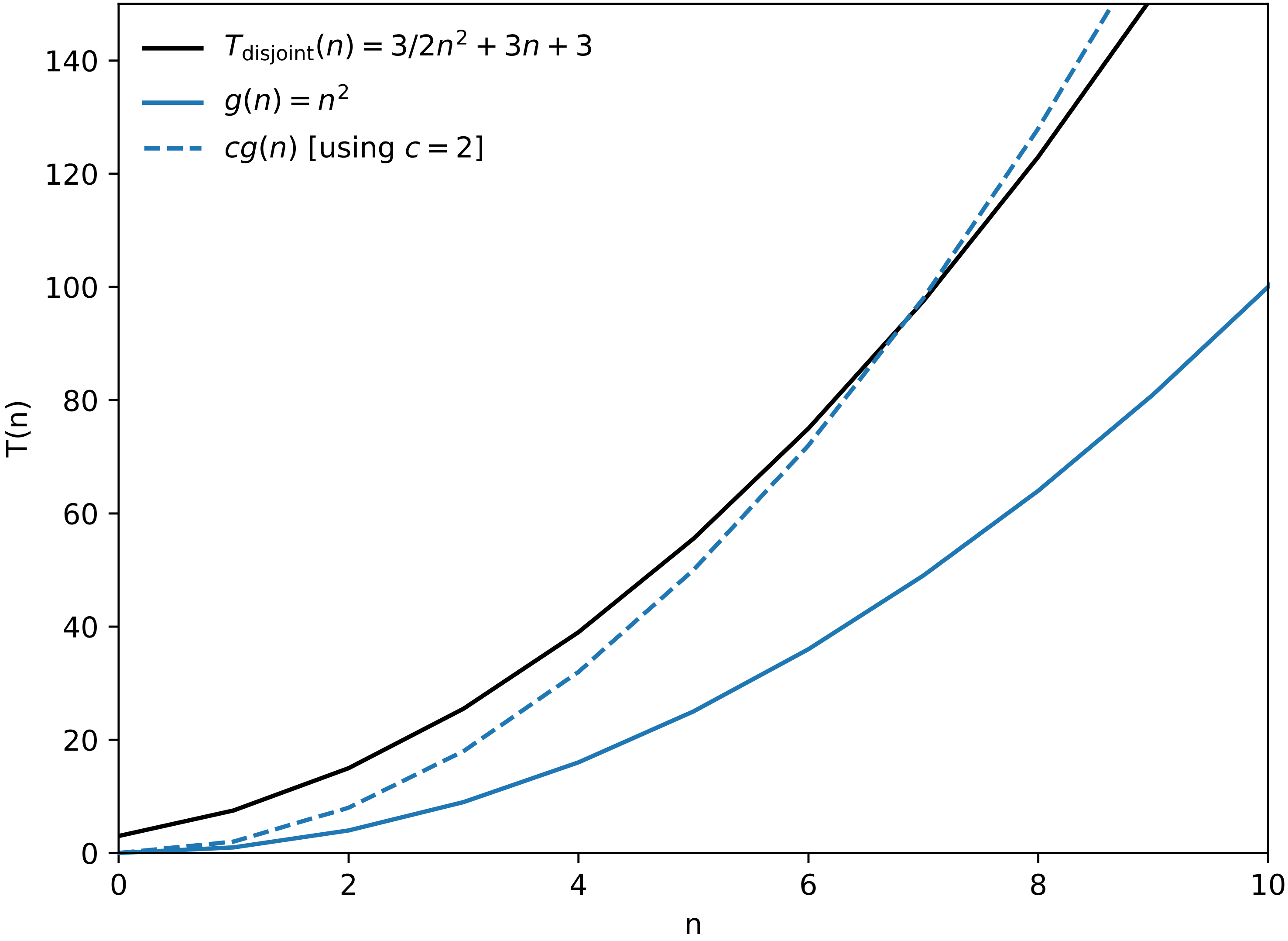
Let’s try to narrow down what we mean by “order of growth”



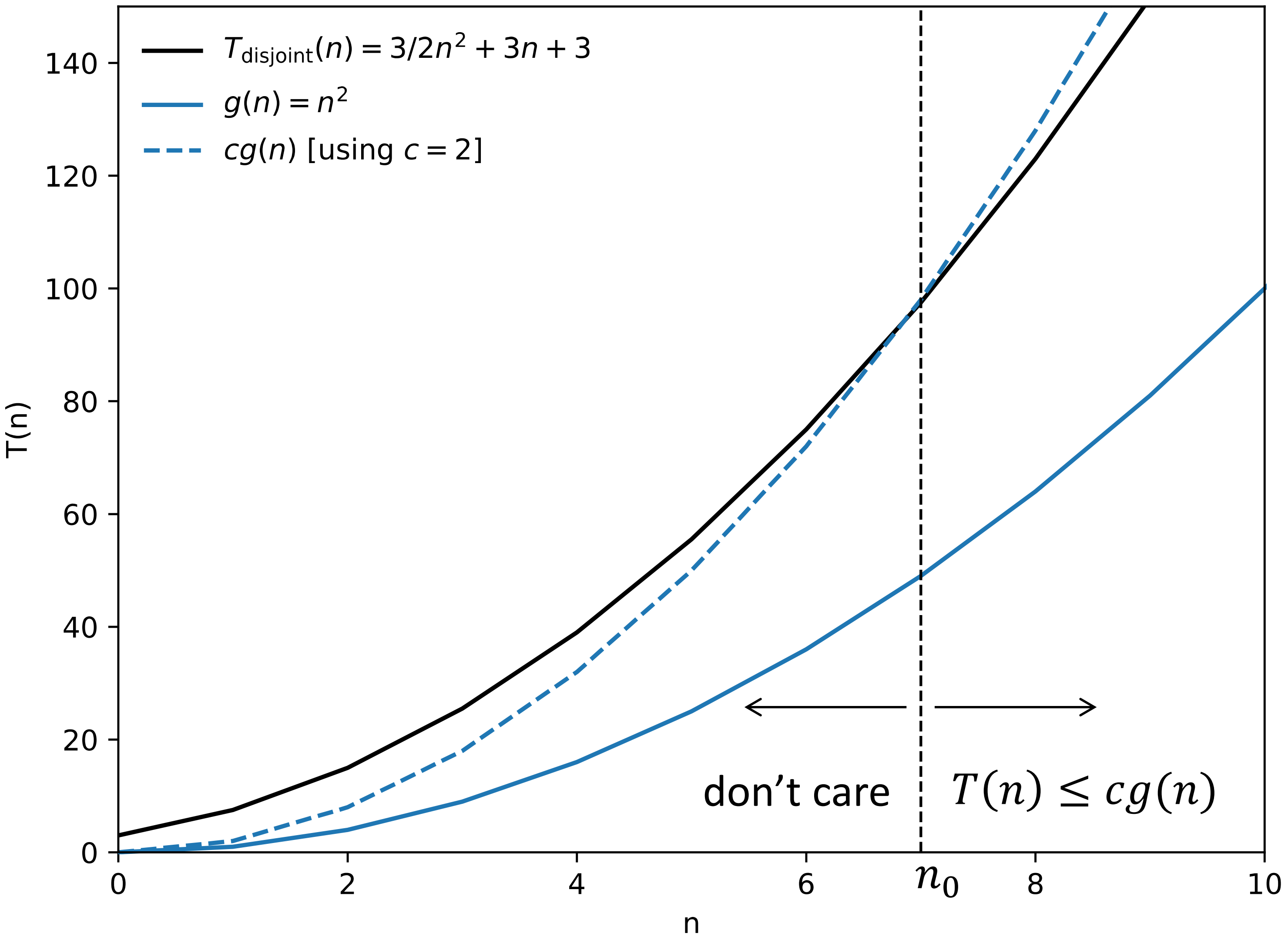
Intuitively, simple function g(n) has same order of growth as T(n). Why?



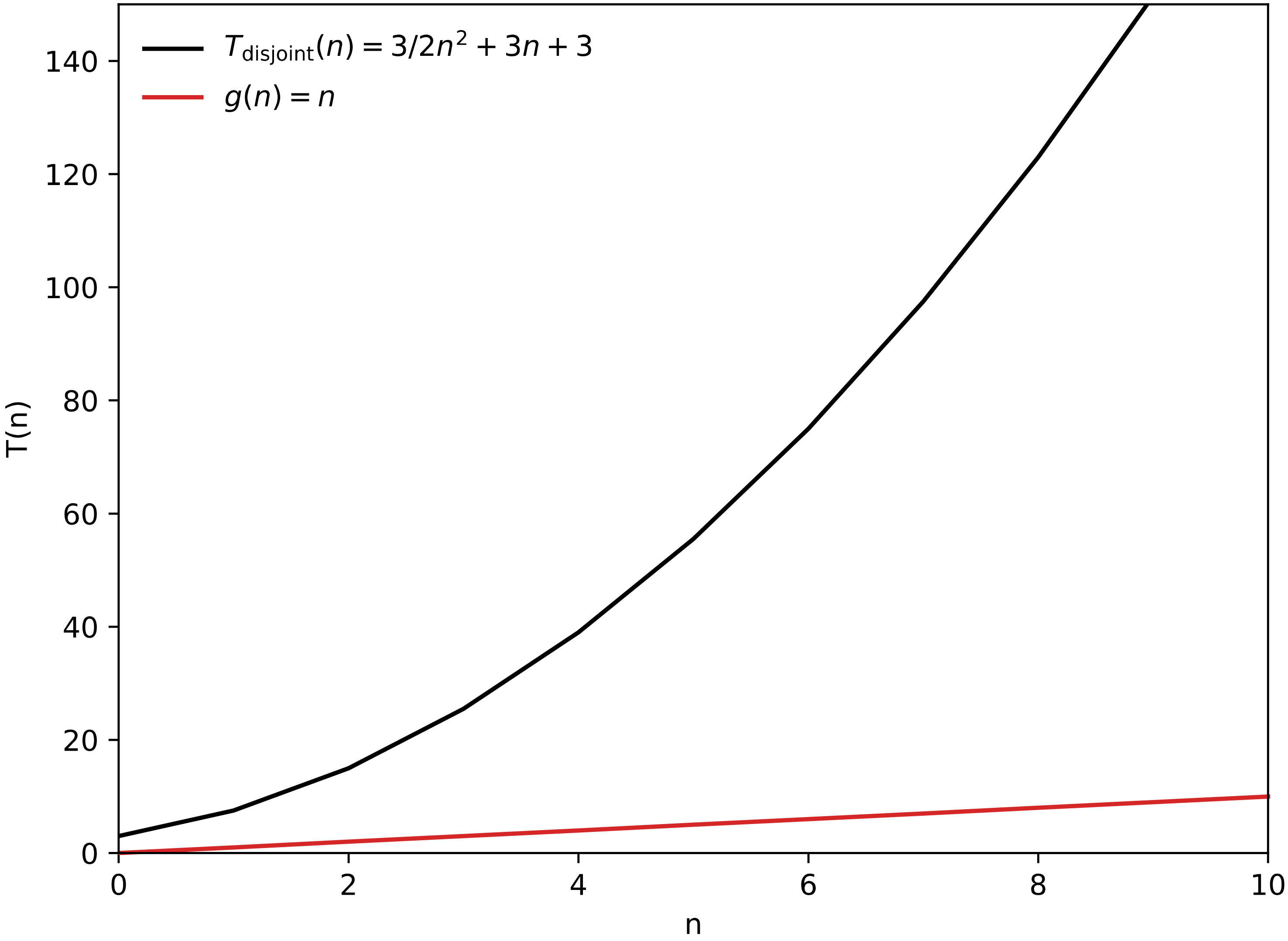
We can simply "scale it up" by a constant *c*...



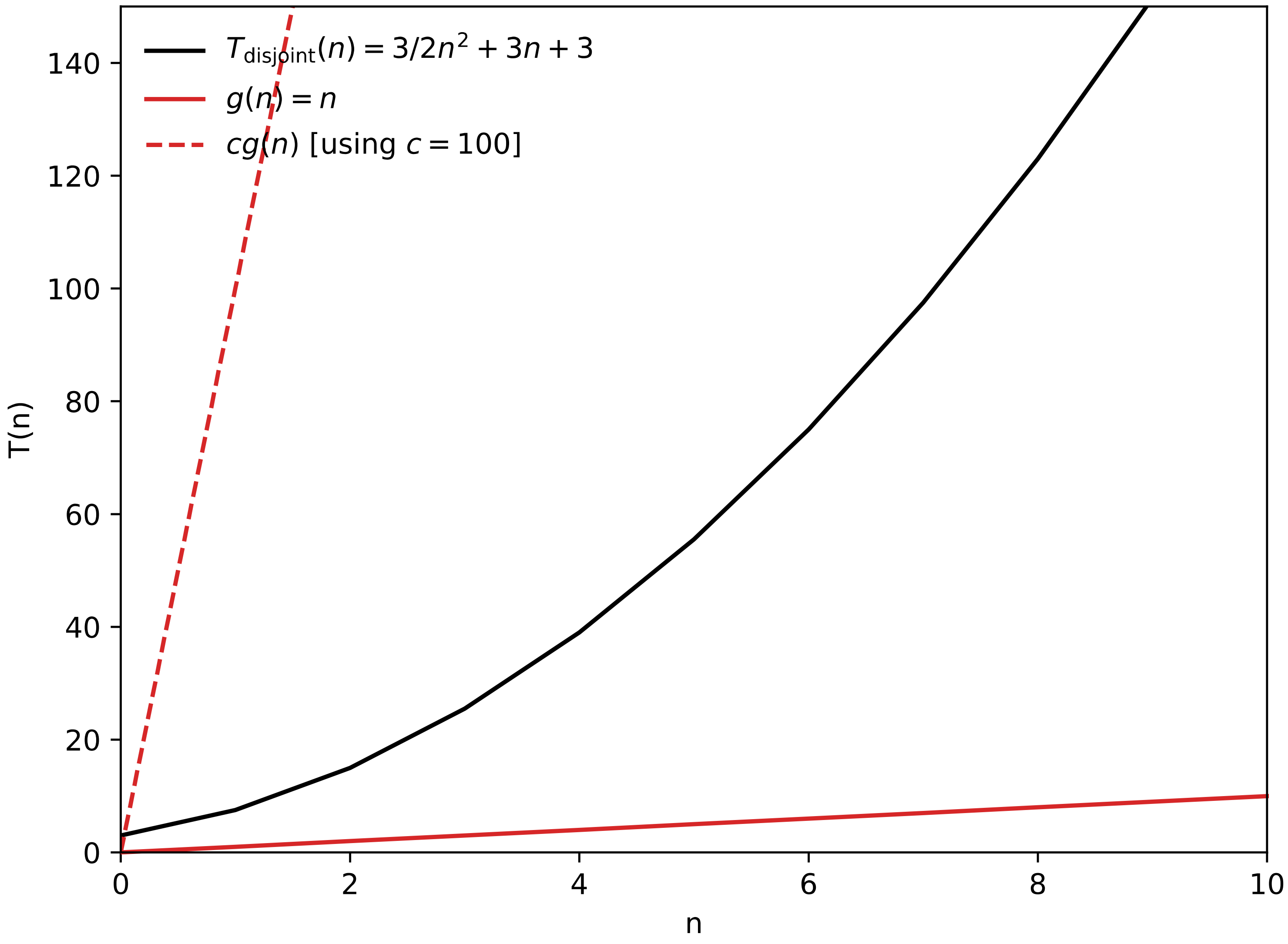
…and it dominates T(n) for all but a finite prefix of input sizes!



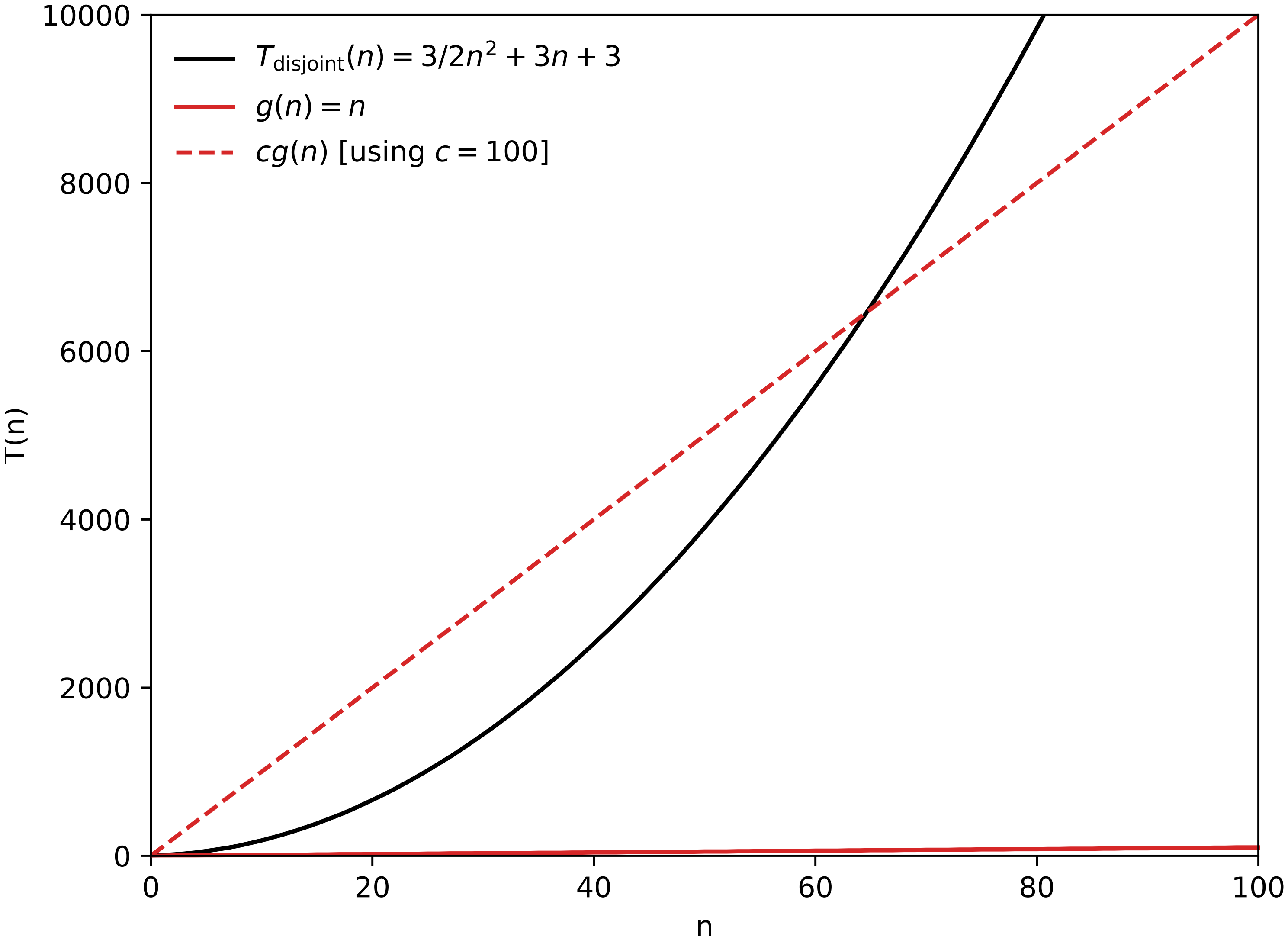
Could we also have used linear function as upper bound?



Could we also have used linear function as upper bound?



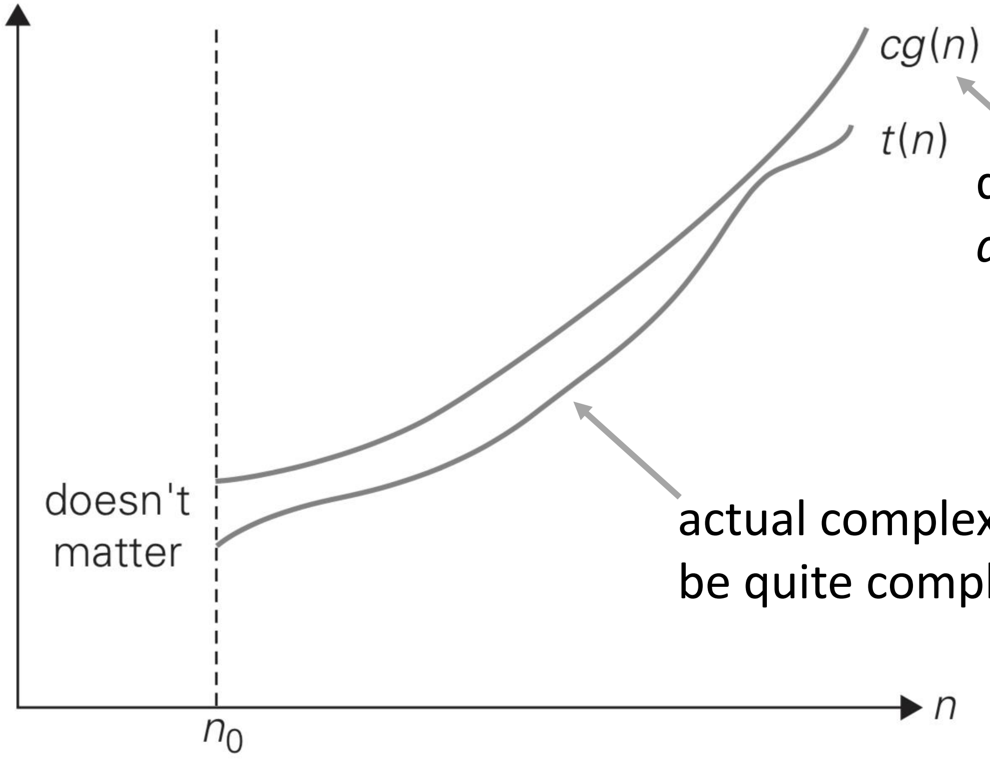
No! Eventually quadratic term of original function is “winning”



Order of growth: “Big O”-Notation

Definition [Levitin, p. 53]

A function $T(n)$ is in $O(g(n))$ if there are positive numbers c and n_0 such for all $n \geq n_0$ it holds that $T(n) \leq cg(n)$.

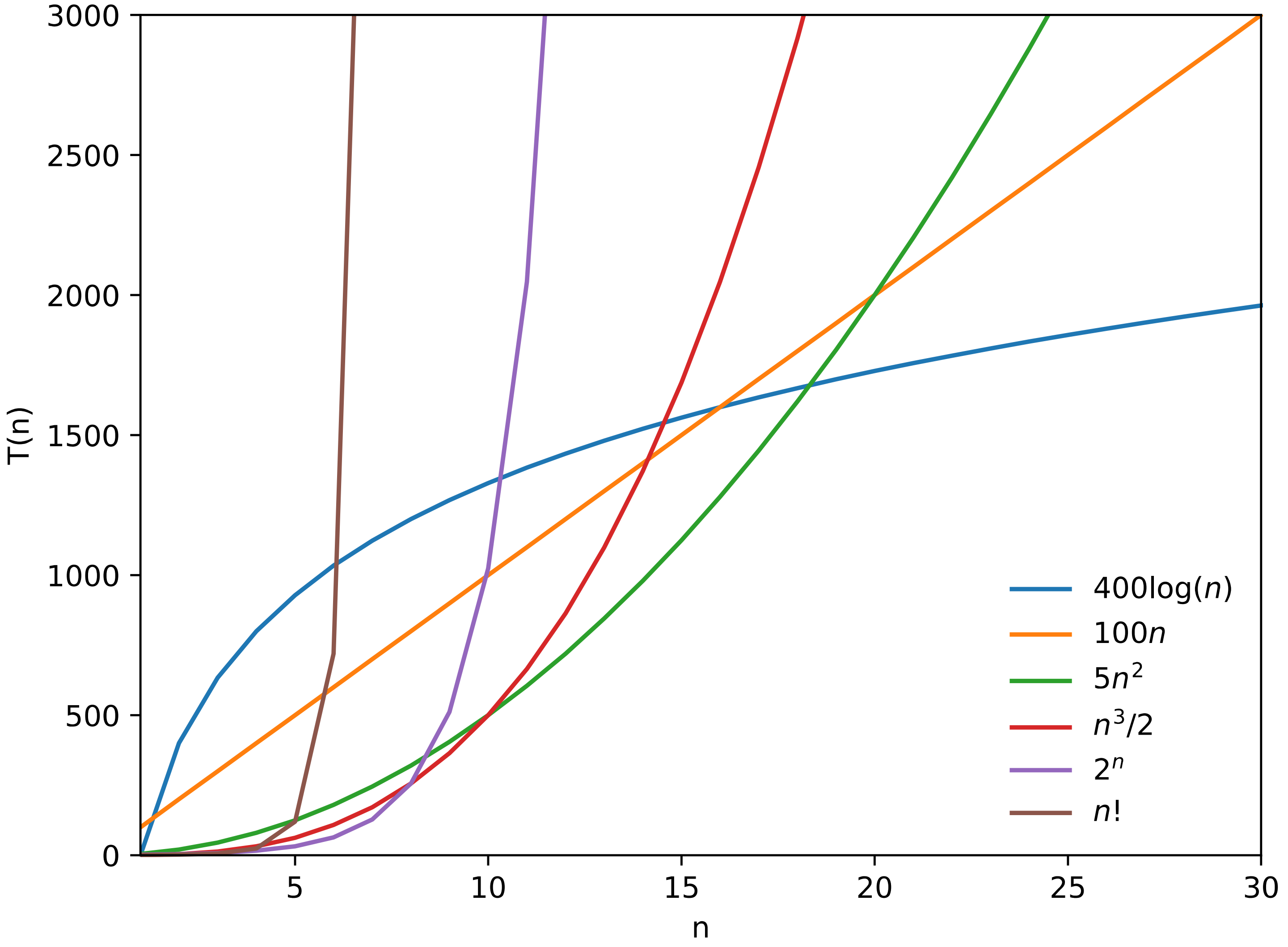


describe by simple *asymptotic* upper bound

actual complexity can be quite complicated

illustration: [Levitin, p. 54]

Different growth rates



Different growth rates

n	log(n)	n	Nlog(n)	n^2	2^n	n!
10	0.003 µs	0.01 µs	0.033 µs	0.1 µs	1 µs	3.63 ms
20	0.004 µs	0.02 µs	0.086 µs	0.4 µs	1 ms	77.1 years
30	0.005 µs	0.03 µs	0.147 µs	0.9 µs	1 sec	$8.4\text{x}10^{15}$ years
40	0.005 µs	0.04 µs	0.213 µs	1.6 µs	18.3 min	
50	0.006 µs	0.05 µs	0.282 µs	2.5 µs	13 days	
100	0.007 µs	0.1 µs	0.644 µs	10 µs	$4\text{x}10^{13}$ years	
1,000	0.010 µs	1 µs	9.966 µs	1 ms		
10,000	0.013 µs	10 µs	130 µs	100 ms		
100,000	0.017 µs	100 µs	1.67 ms	10 sec		
1,000,000	0.020 µs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 µs	10 ms	0.23 sec	1.16 days		
100,000,000	0.027 µs	0.1 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 µs	1 sec	29.90 sec	31.7 years		

Measured in nanoseconds (10^{-9} secs)

Slide based on material by Maria Garcia de la Banda

Different growth rates

n	log(n)	n	Nlog(n)	n^2	2^n	n!
10	0.003 µs	0.01 µs	0.033 µs	0.1 µs	1 µs	3.63 ms
20	0.004 µs	0.02 µs	0.086 µs	0.4 µs	1 ms	77.1 years
30	0.005 µs	0.03 µs	0.147 µs	0.9 µs	1 sec	8.4x10^{15} years
40	0.005 µs	0.04 µs	0.213 µs	1.6 µs	18.3 min	
50	0.006 µs	0.05 µs	0.282 µs	2.5 µs	13 days	
100	0.007 µs	0.1 µs	0.644 µs	10 µs	4x10^{13} years	
1,000	0.010 µs	1 µs	9.966 µs	1 ms		
10,000	0.013 µs	10 µs	130 µs	100 ms		
100,000	0.017 µs	100 µs	1.67 ms	10 sec		
1,000,000	0.020 µs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 µs	10 ms	0.23 sec	1.16 days		
100,000,000	0.027 µs	0.1 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 µs	1 sec	29.90 sec	31.7 years		

Our universe is only 13.8x10^9 years old.

Measured in nanoseconds (10^{-9} secs)

Slide based on material by Maria Garcia de la Banda

Points to keep in mind

- The input that produces the worst case may be very unlikely to occur in practice.
- Big-O ignores constants, which in practice may be very large.
- If a program is used only a few times, then the actual running time may not be a big factor in the overall costs.
- If a program is only used on small inputs, the growth rate of the running time may be less important than other factors.
- A complicated but efficient algorithm may be less desirable than a simpler algorithm.
- In numerical algorithms, accuracy and stability are just as important as efficiency.
- The average case complexity is always between the best and the worst cases.

Overview

1. Computational Complexity
2. Asymptotic Analysis (Big-Oh notation)
3. Application to Sorting Algorithms

Complexity of Selection Sort

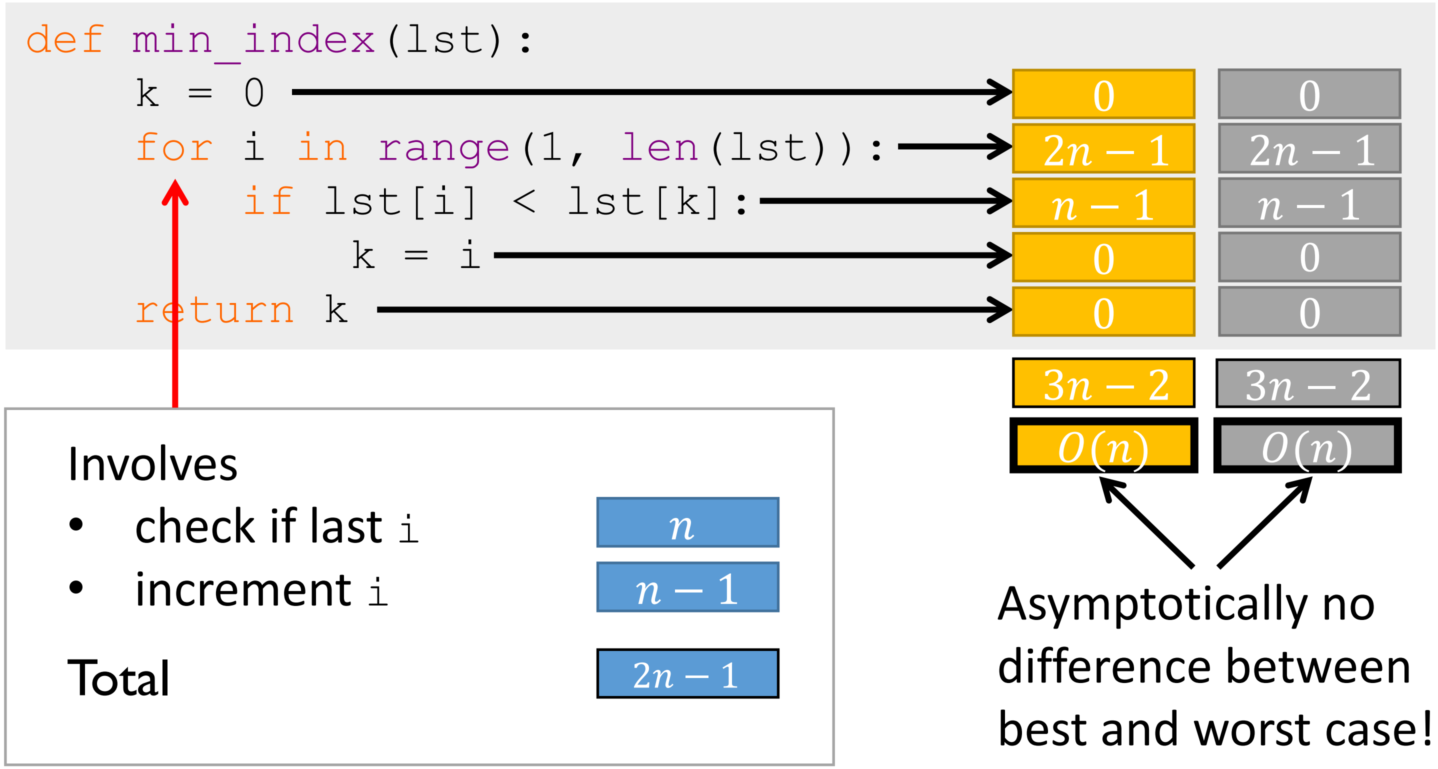
```
def min_index(lst):
    k = 0
    for i in range(1, len(lst)):
        if lst[i] < lst[k]:
            k = i
    return k
```

```
def selection_sort(lst):
    for i in range(len(lst)):
        j = min_index(lst[i:]) + i
        lst[i], lst[j] = lst[j], lst[i]
```

another advantage of decomposition:

can reason about complexity per function!

Complexity of `min_index`



```
def min_index(lst):
    k = 0
    for i in range(1, len(lst)):
        if lst[i] < lst[k]:
            k = i
    return k
```

`k = 0`	0	0
`for i in range(1, len(lst)):`	$2n-1$	$2n-1$
`if lst[i] < lst[k]:`	$n-1$	$n-1$
`k = i`	0	0
`return k`	0	0
	$3n-2$	$3n-2$
	$O(n)$	$O(n)$

Involves

- check if last `i`	n
- increment `i`	$n-1$
Total	$2n-1$

Asymptotically no difference between best and worst case!

Simpler way to arrive at the same conclusion

```
def min_index(lst):
    k = 0
    for i in range(1, len(lst)):
        if lst[i] < lst[k]:
            k = i
    return k
```

`k = 0`	$O(0)$	$O(0)$
`for i in range(1, len(lst)):`	$O(n)$	$O(n)$
`if lst[i] < lst[k]:`	$O(n)$	$O(n)$
`k = i`	$O(0)$	$O(0)$
`return k`	$O(0)$	$O(0)$
	$3n-2$	$3n-2$
	$O(n)$	$O(n)$

Involves

- check if last `i`	$O(n)$
- increment `i`	$O(n)$
Total	$O(n)$

Asymptotically no difference between best and worst case!

General Observation

If T_1 is in $O(g_1(n))$ and T_2 is in $O(g_2(n))$ then sum $T_1(n) + T_2(n)$ is in $O(\max\{g_1(n), g_2(n)\})$

Overall complexity is determined by step of highest complexity!!!

Complexity of overall Selection Sort

```
def selection_sort(lst):
    for i in range(len(lst)):
        j = min_index(lst[i:]) + i
        lst[i], lst[j] = lst[j], lst[i]
```

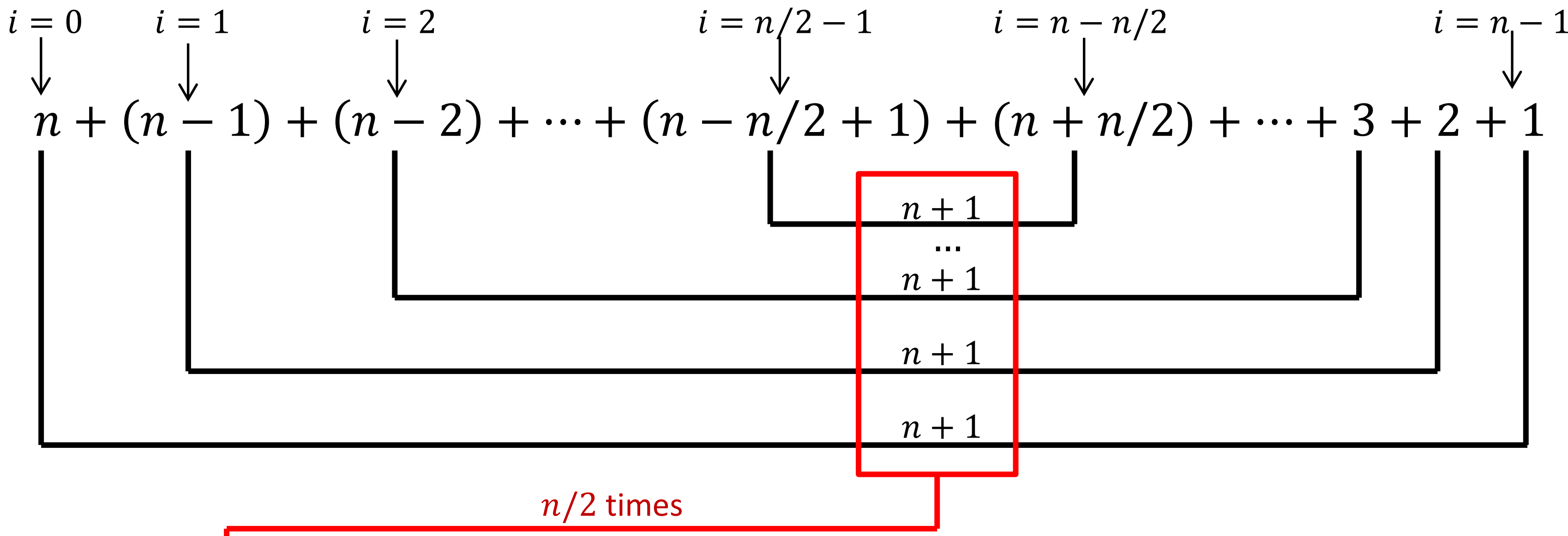
$O(n)$

$O(n^2)$

$O(n)$

$\boxed{O(n^2)}$

Total cost of `min_index` calls



$$n + (n - 1) + (n - 2) + \cdots + (n - n/2 + 1) + (n + n/2) + \cdots + 3 + 2 + 1$$

$$= (n + 1)n/2 = (n^2 + n)/2 = O(n^2)$$

Let’s analyse Insertion Sort

```
def insert(i, lst):
    temp = lst[i]
    j = i-1
    while j >= 0 and lst[j] > temp:
        lst[j+1] = lst[j]
        j = j - 1
    lst[j+1] = temp
```

$n = \text{i}$	
$O(0)$	$O(0)$
$O(0)$	$O(0)$
$O(1)$	$O(n)$
$O(0)$	$O(n)$
$O(0)$	$O(n)$
$O(0)$	$O(0)$
$\boxed{O(1)}$	$\boxed{O(n)}$

```
def insertion_sort(lst):
    for i in range(1, len(lst)):
        insert(i, lst)
```

$n = \text{len(lst)}$	
$O(n)$	$O(n)$
$O(n)$	$O(n^2)$
$\boxed{O(n)}$	$\boxed{O(n^2)}$

Total cost of `insert` calls

- best case (sorted input): $\overbrace{1 + 1 + 1 + \cdots + 1}^{n \text{ times}} = O(n)$
- worst case (inversely sorted input): $\underbrace{1 + 2 + \cdots + (n-1) + n}_{= \frac{(n+1)n}{2} \text{ as previously}} = O(n^2)$

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        for k in range(n):
                for j in range(n):
                        L[k][j]=k*j
        print(L)
```

a. $O(n)$
b. $O(\log n)$
c. $O(n \log n)$
d. $O(n^2)$
e. $O(2^n)$
f. $O(n!)$
g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        for k in range(n):
                for j in range(k):
                        L[k][j]=k*j
        print(L)
```

a. *O(n)*
b. *O(log n)*
c. *O(n log n)*
d. $O(n^2)$
e. $O(2^n)$
f. *O(n!)*
g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        for k in range(n):
                for j in range(n):
                        L[k][j]=k*j

        for x in range(n):
                print(L[x])
        print(L)
```

a. *$O(n)$*
b. *$O(\log n)$*
c. *$O(n \log n)$*
d. *$O(n^2)$*
e. *$O(2^n)$*
f. *$O(n!)$*
g. None of the above

What is the time complexity for this algorithm?

```
def pairs(n):
    for k in range(n):
        for j in range(k+1, n):
            print([k, j])
```

a. $O(1)$

b. $O(n)$

c. $O(n^2)$

d. None of the above

1. Visit https://flux.qa
2. Log in using your Authcate details (not required if you're already logged in to Monash)
3. Touch the + symbol and enter the code: UF7BD9
4. Answer questions when they pop up.

What is the time complexity for this algorithm?

```
def power(x, n):
    ’computes x to the power of n’
    value = 1
    if n > 0:
        value = power(x, n//2)
        if n % 2 == 0:
            value = value*value
        else:
            value = value*value*x
    return value
```

a. *$O(n)$*

b. *$O(\log n)$*

c. *$O(n^2)$*

d. None of the above

1. Visit https://flux.qa
2. Log in using your Authcate details (not required if you’re already logged in to Monash)
3. Touch the + symbol and enter the code: UF7BD9
4. Answer questions when they pop up.

Exercise

Find the complexity of the following algorithms

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        d = 500
        for k in range(n):
                for j in range(d):
                        L[k][j]=k*j
```

- *a.* *O(n)*
- *b.* *O(log n)*
- *c.* *O(n log n)*
- *d.* *O(n^2)*
- *e.* *O(2^n)*
- *f.* *O(n!)*
- g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        d = 500
        for k in range(n):
                for j in range(d):
                        L[k][j]=k*j

        for x in range(n):
                print(L[x])
        print(L)
```

a. *$O(n)$*

b. *$O(\log n)$*

c. *$O(n \log n)$*

d. *$O(n^2)$*

e. *$O(2^n)$*

f. *$O(n!)$*

g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        d = 500
        for k in range(n):
                for j in range(2*n):
                        L[k][j]=k*j
```

- *a.* $O(n)$
- *b.* $O(\log n)$
- *c.* $O(n \log n)$
- *d.* $O(n^2)$
- *e.* $O(2^n)$
- *f.* $O(n!)$
- g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        d = 500
        for k in range(2,n,4):
                for j in range(2*n):
                        L[k][j]=k*j
```

- a. $O(n)$
- b. $O(\log n)$
- c. $O(n \log n)$
- d. $O(n^2)$
- e. $O(2^n)$
- f. $O(n!)$
- g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        d = 500
        for k in range(2,n,4):
                for j in range(5,2*n):
                        L[k][j]=k*j
```

- a. $O(n)$
- b. $O(\log n)$
- c. $O(n \log n)$
- d. $O(n^2)$
- e. $O(2^n)$
- f. $O(n!)$
- g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        d = 500
        for k in range(2,n,4):
                for j in range(5,n):
                        L[k][j]=k*j
```

a. *$O(n)$*
b. *$O(\log n)$*
c. *$O(n \log n)$*
d. *$O(n^2)$*
e. *$O(2^n)$*
f. *$O(n!)$*
g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        d = 500
        for k in range(2,n,4):
                for j in range(-50,1,2):
                        L[k][j]=k*j
```

a. $O(n)$
b. $O(\log n)$
c. $O(n \log n)$
d. $O(n^2)$
e. $O(2^n)$
f. $O(n!)$
g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        s = 5
        for k in range(2,n,2):
                s = s + n
        for j in range(2*n):
                L.append(s)
```

a. $O(n)$

b. $O(\log n)$

c. $O(n \log n)$

d. $O(n^2)$

e. $O(2^n)$

f. $O(n!)$

g. None of the above

What is the time complexity for this algorithm?

```
def func(d):
        while d > 0:
                d = d - 1
                print("Hello ",d)
```

a. $O(n)$

b. $O(\log n)$

c. $O(n \log n)$

d. $O(n^2)$

e. $O(2^n)$

f. $O(n!)$

g. None of the above

What is the time complexity for this algorithm?

```
def func(d):
		if d > 0:
				d = d - 1
				print("Hello ",d)
```

a. $O(n)$

b. $O(\log n)$

c. $O(n \log n)$

d. $O(n^2)$

e. $O(2^n)$

f. $O(n!)$

g. None of the above

What is the time complexity for this algorithm?

```
def func(d):
        while d > 0:
                d = d + 1
                d = d // 4
                print("Hello ",d)
        return d
```

a. $O(n)$

b. $O(\log n)$

c. $O(n \log n)$

d. $O(n^2)$

e. $O(2^n)$

f. $O(n!)$

g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        L=[]
        for k in range(n):
                for j in range(n):
                        L[k][j]=k*j

        for x in range(n):
                print(L[x])
        print(L)

        while n > 0:
                n = n + 1
                n = n // 4
                print("Hello ",n)
        return n
```

a. *O(n)*
b. *O(log n)*
c. *O(n log n)*
d. $O(n^2)$
e. $O(2^n)$
f. *O(n!)*
g. None of the above

What is the time complexity for this algorithm?

```
def func(n):
        total = 0
        while n > 0:
                for k in range(n):
                        total = total + 5
                if total > 100:
                        print("Big total")
                n = n // 4
                print("Hello ",n)
        return n
```

a. *O(n)*
b. *O(log n)*
c. *O(n log n)*
d. *$O(n^2)$*
e. *$O(2^n)$*
f. *O(n!)*
g. None of the above

Summary

Computational complexity of algorithm number of steps for input size (i.e., it is a function)

- needs to choose best-case or worst-case
- details depends on some computational model

Usually we are interested in order of growth of complexity only (Big-O notation)

Application to sorting algorithms

- Both Selection Sort and Insertion Sort have quadratic worst-case complexity
- Insertion Sort is adaptive (linear best-case)

Before Next Lecture

Read: The Design & Analysis of Algorithms, L. Perkovic,

Chapter 2.2 – Asymptotic Notations & Efficiency Classes

Log onto the MCD4710 Moodle site

Watch (again) the following video:

Big O Notation

Before Next Lecture

Read

“Introduction Design and Analysis of Algorithms”

A. Levitin

Chapter 2.2: Big-O Notation section