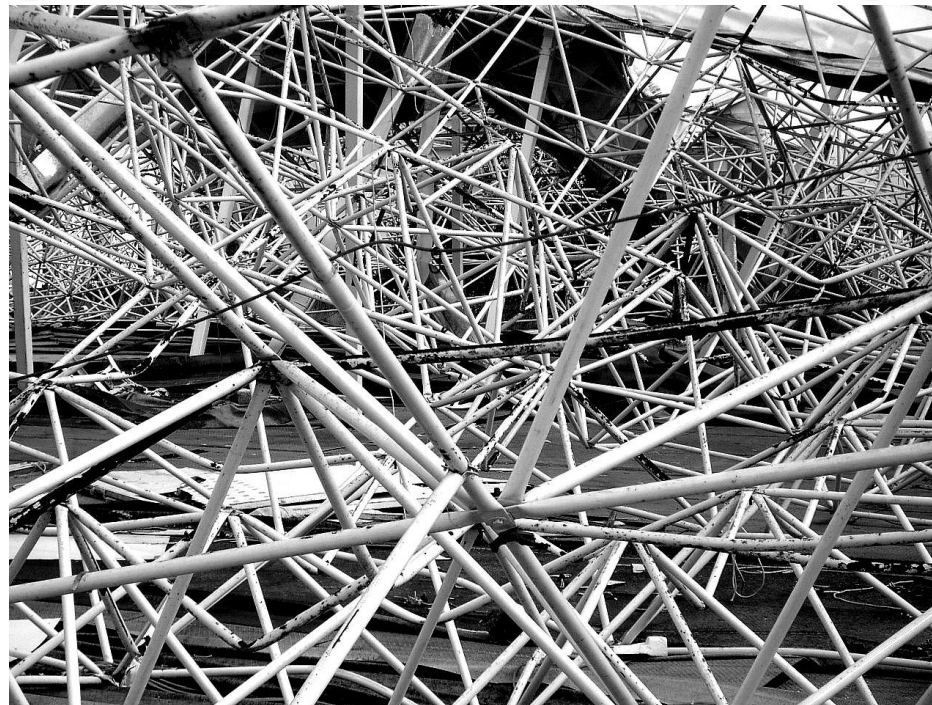


FIT1045: Algorithms and Programming Fundamentals in Python

Lecture 11

Computational Complexity



COMMONWEALTH OF AUSTRALIA
Copyright Regulations 1969
WARNING

This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act). The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Assignment Part I

- Assignment due this Friday!
- Only part I (part 2 is due in week II)
- Changes in submission:
 - Do **not** submit zip with regression.py
 - Instead, submit regression.py
 - This would greatly help the marking team
 - (you won't be grilled if you submit a .zip, but please try to make it easy for your tutors)

Different algorithms can solve 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

<code>x = 8 % 2</code>	→	1
<code>a = [10, 2, 10] + [5, 2, 27, 18]</code>	→	7
<code>len(a)</code>	→	0
<code>x == 0</code>	→	1

Cost model for basic instructions

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

*objects with fixed (in memory) size

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

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

Counting elementary steps

Fix cost model for set of **basic instructions**

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

Break down **complex instructions** to determine their cost

<code>not (18 % 3 == 0 and 10 % 3 == 0)</code>	6
<code>price_with_gst(27+3)</code>	3

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

<code>a = []</code>	0
<code>while len(a) < 5:</code>	
<code>a = a + [len(a)]</code>	

Quiz time (<https://flux.qa>)

Clayton: AXXULH

Malaysia: LWERDE

Counting elementary steps

Model cost of elementary instructions

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

Break down complex instructions to determine their cost

<code>not (18 % 3 == 0 and 10 % 3 == 0)</code>	6
<code>price_with_gst(27+3)</code>	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)]
```

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

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

Break down complex instructions to determine their cost

<code>not (18 % 3 == 0 and 10 % 3 == 0)</code>	6
<code>price_with_gst(27+3)</code>	3

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

<code>a = []</code>	0	
<code>while len(a) < 5:</code>	6	
<code> a = a + [len(a)]</code>	15	
→		
<code>a = [] + [0]</code>	1	
<code>a = [0] + [1]</code>	2	
<code>a = [0, 1] + [2]</code>	3	
<code>a = [0, 1, 2] + [3]</code>	4	
<code>a = [0, 1, 2, 3] + [4]</code>	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
```

Quiz time (<https://flux.qa>)

Clayton: AXXULH

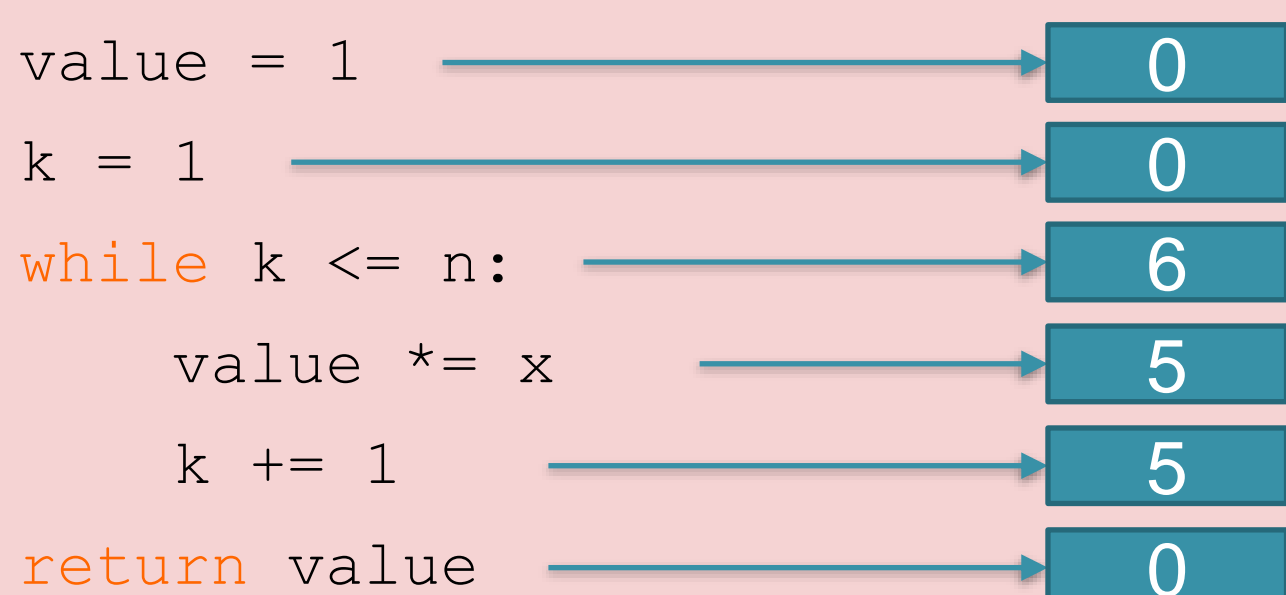
Malaysia: LWERDE

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

- A. 6
- B. 15
- C. 16
- D. 19

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
```



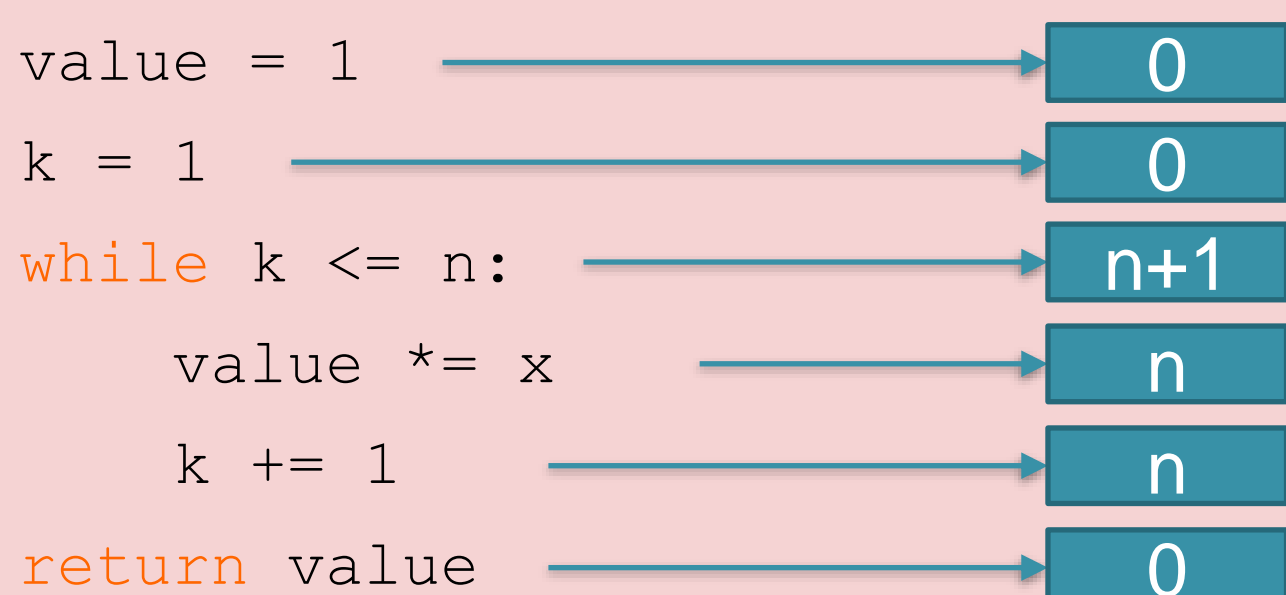
Code Line	Value
value = 1	0
k = 1	0
while k <= n:	6
value *= x	5
k += 1	5
return value	0

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

16

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)`?

3n+1

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
```

0
0
3
3
0
2
0
8

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) → 

|   |
|---|
| 0 |
| 0 |
| ? |
| ? |
| ? |
| ? |
| ? |

  
    i = 0 → 

|   |
|---|
| 0 |
| 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 $n = \text{len}(seq)$)

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) → 0  
    i = 0 → 0  
    while i < n: → 1  
        if seq[i] == v: → 1  
            return i → 0  
        i += 1 → 0  
    return None → 0  
→ 2
```

Best case (fewest steps): $v == \text{seq}[0]$

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
```

Code Line	Green Box (Left)	Red Box (Right)
<code>n = len(seq)</code>	0	0
<code>i = 0</code>	0	0
<code>while i < n:</code>	1	$n + 1$
<code> if seq[i] == v:</code>	1	n
<code> return i</code>	0	0
<code> i += 1</code>	0	n
<code>return None</code>	0	0
Total	2	$3n + 1$

Best case (fewest steps): $v == \text{seq}[0]$

Worst case (most steps): $v \text{ not in seq}$

Another example: repeated search

```
def disjoint(s1, s2): n = n1 + n2
    """I: sequences s1 and 2
       O: True is s1 and s2 have no common element
          (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
```

0
0
$n/2 + 1$
$3n^2/2 + n/2$
0
$n/2$
0

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

What is input size?

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

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

Intermediate Summary

Computational complexity

- Express number of computational steps as **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**

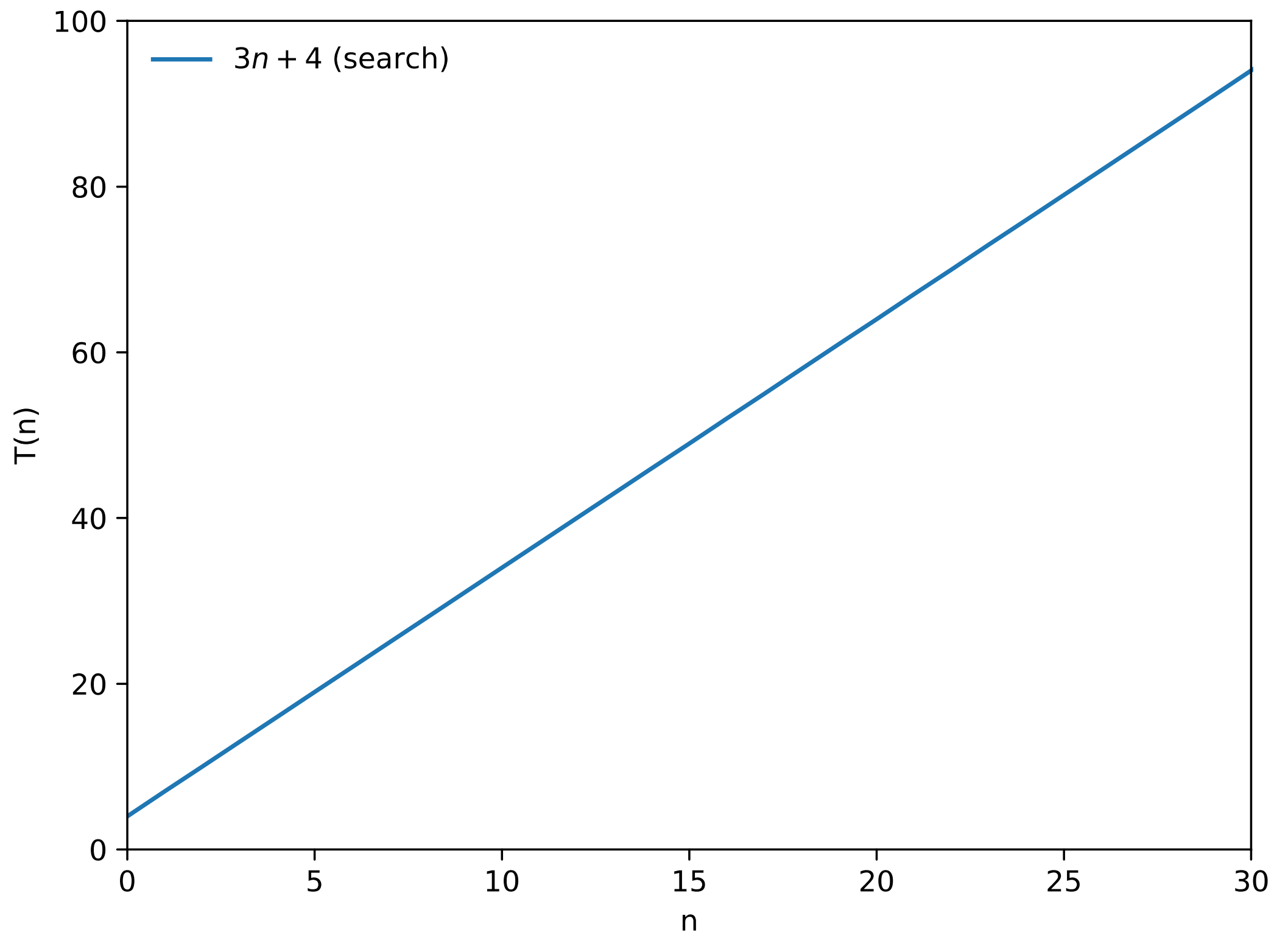
Observations

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

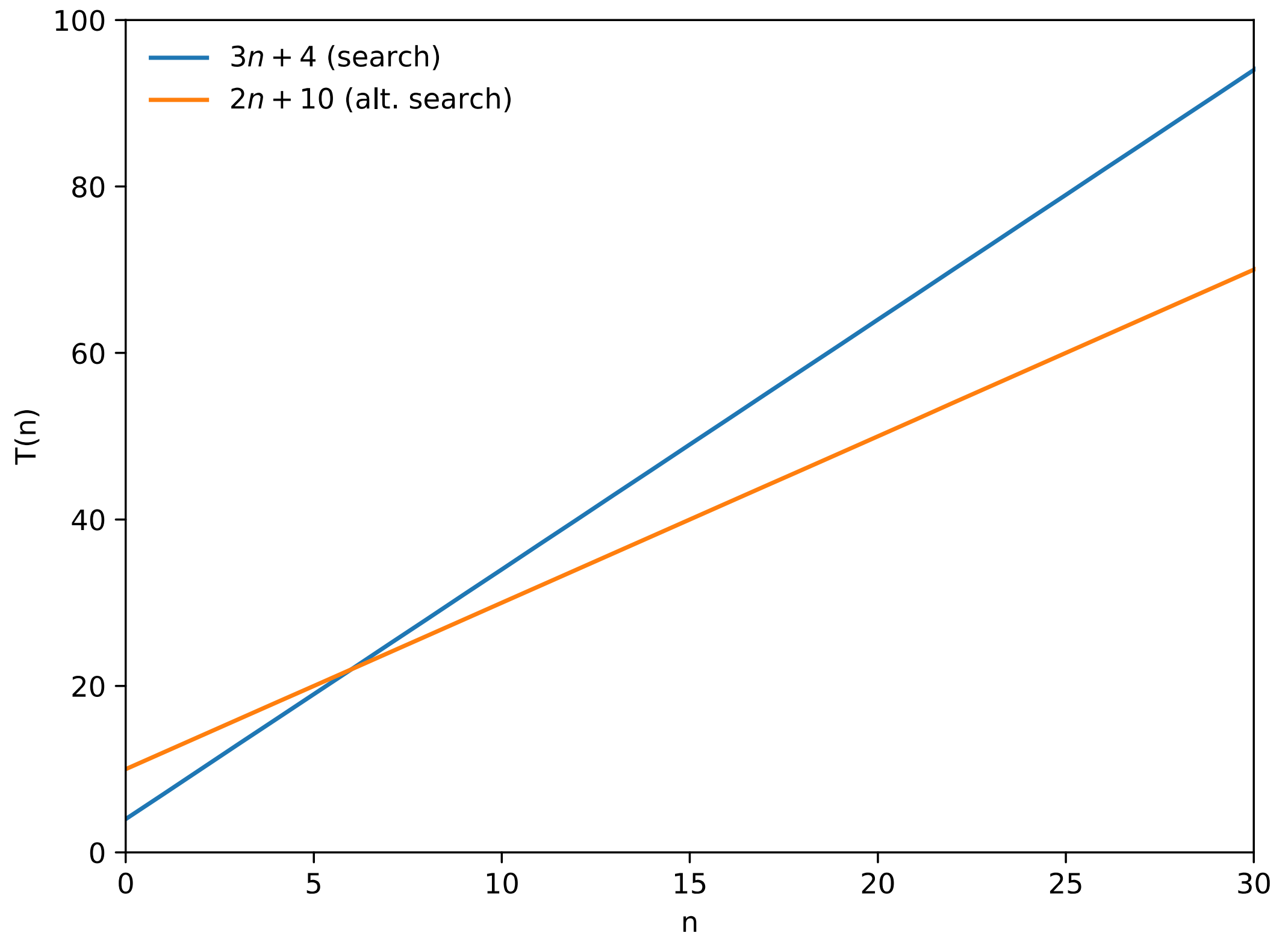
Overview

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

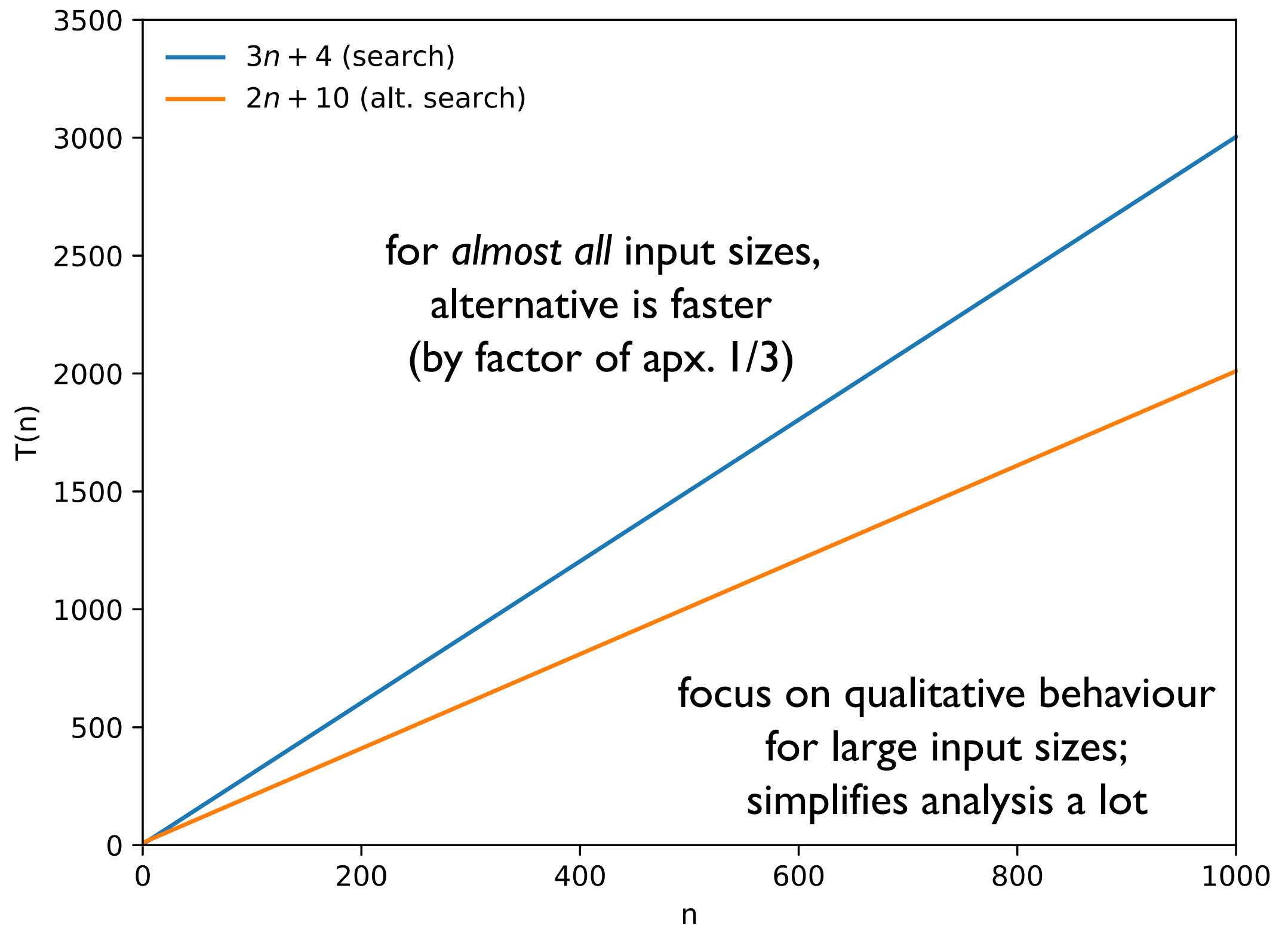
Let's plot time complexity of search function



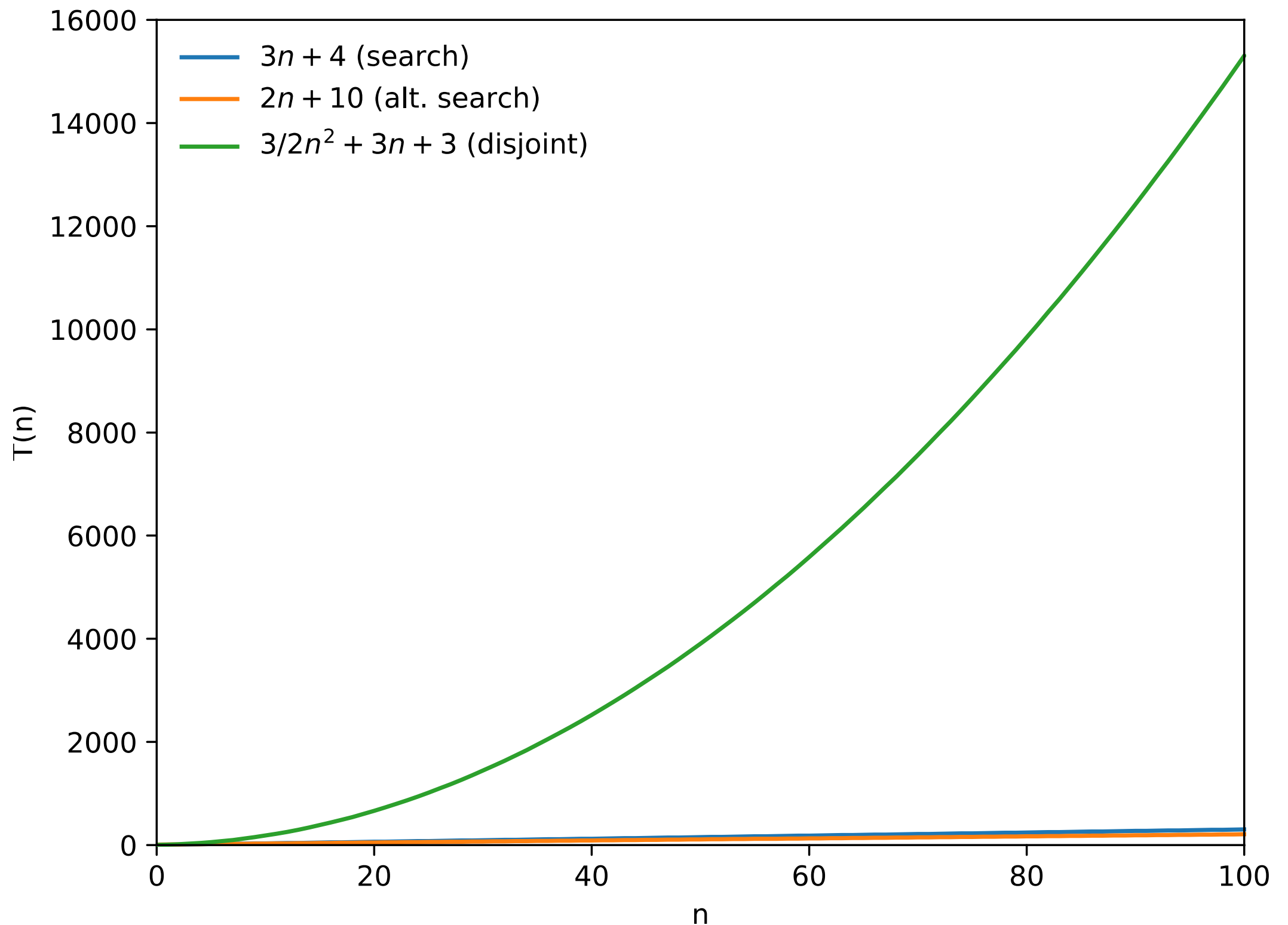
Assume alternative algorithm



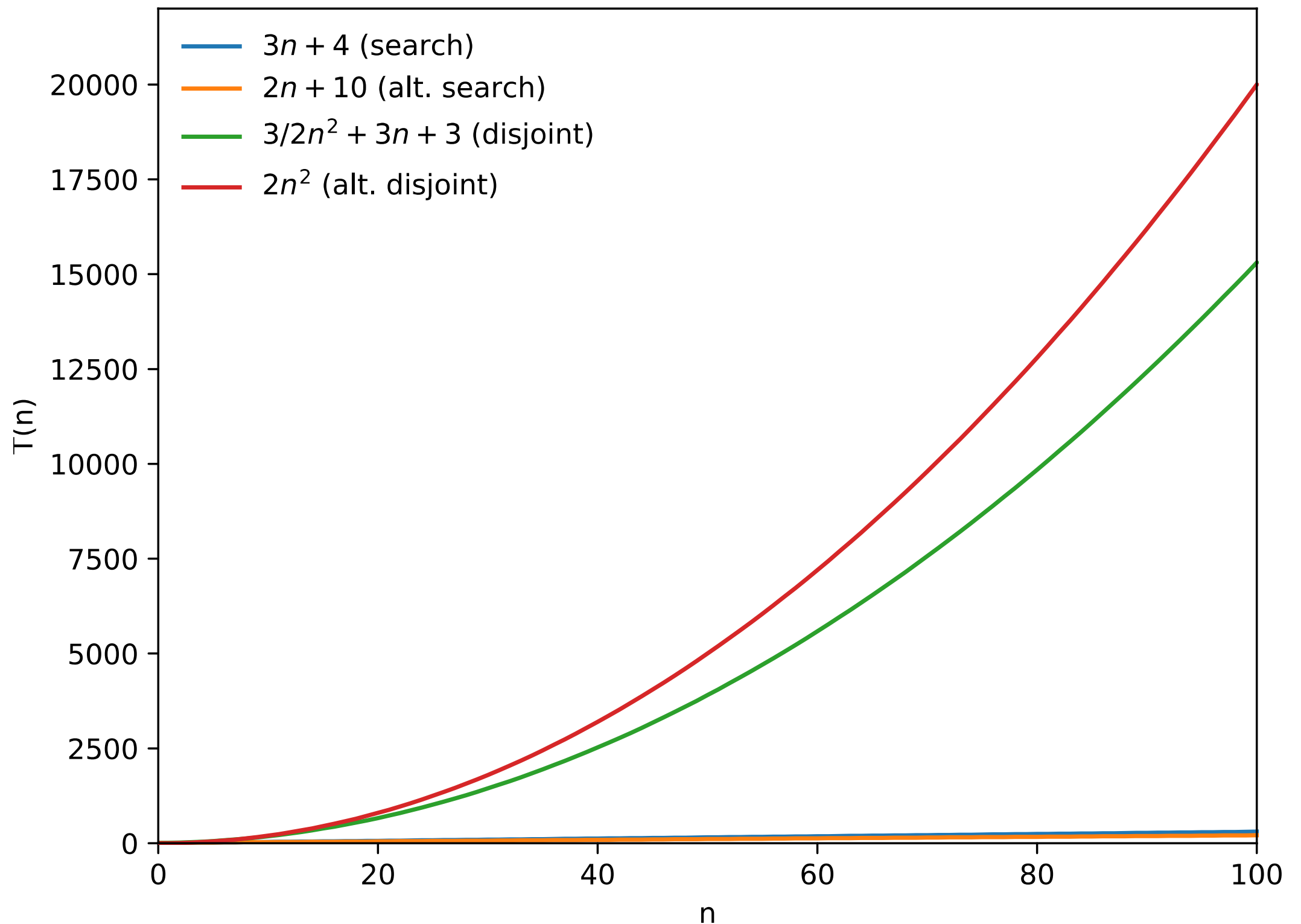
Assume alternative algorithm



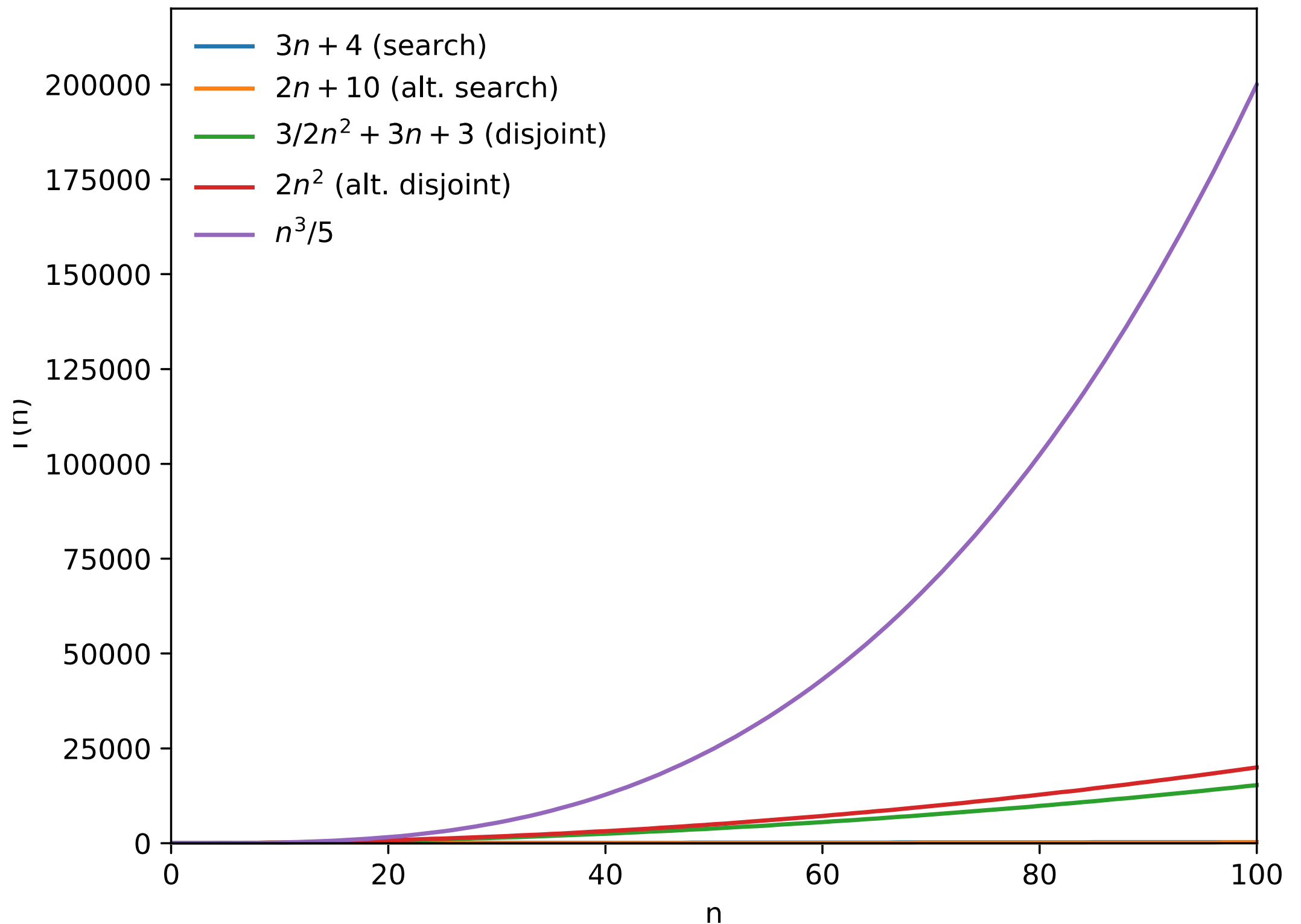
Let's compare both to disjoint



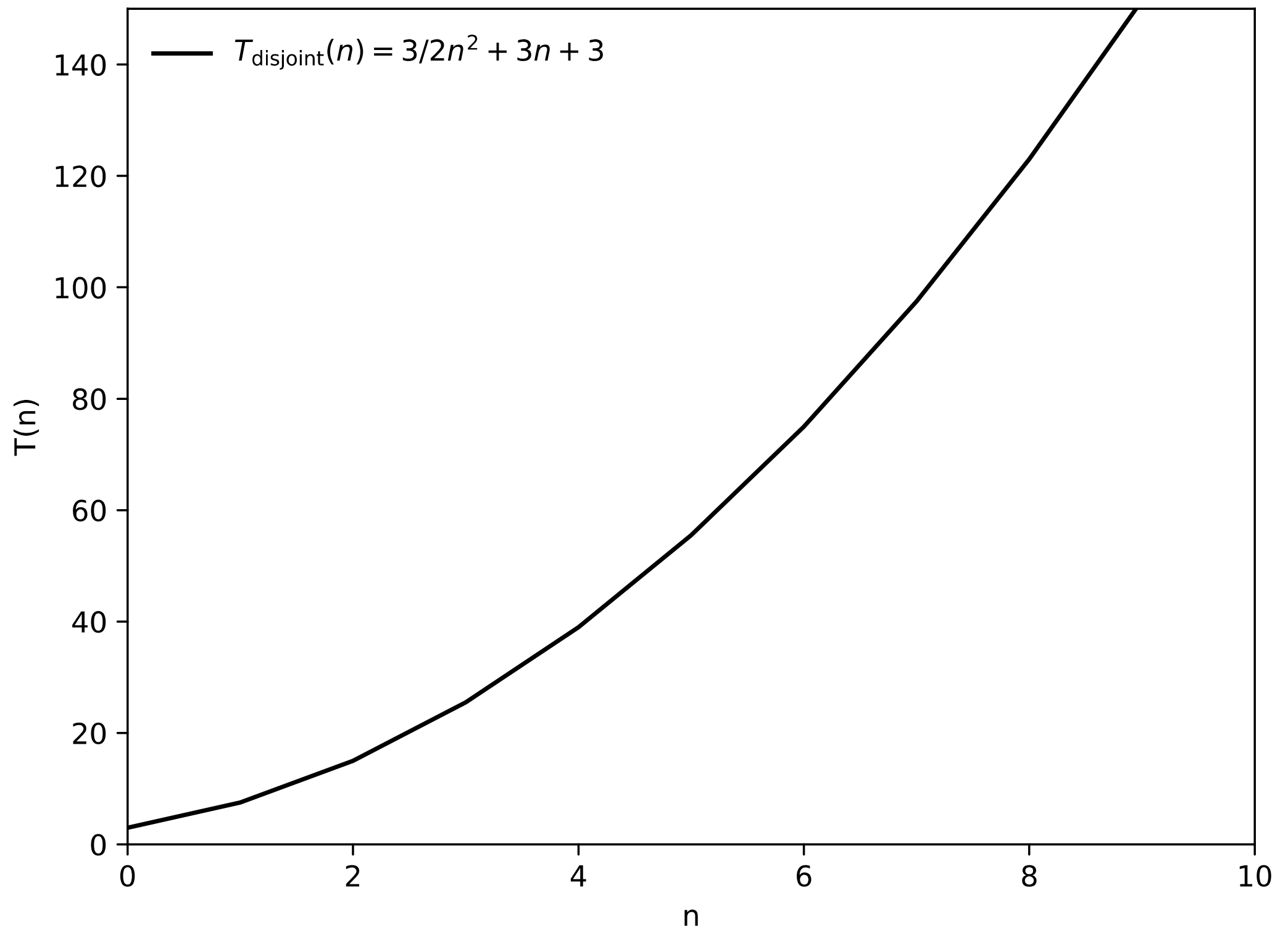
Only coefficient of largest growth term is relevant for large input sizes



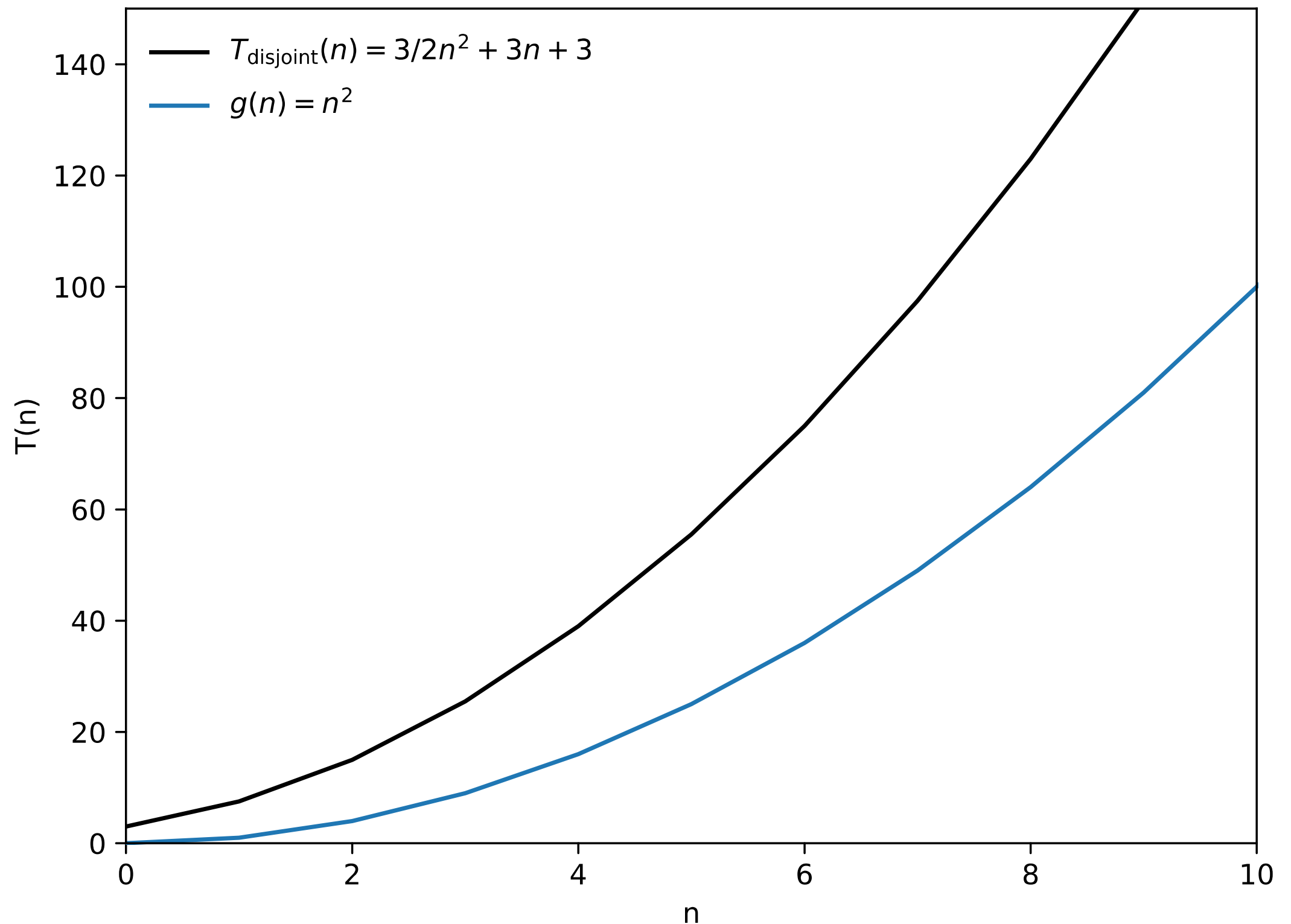
...but also negligible when comparing
to higher *order of growth*



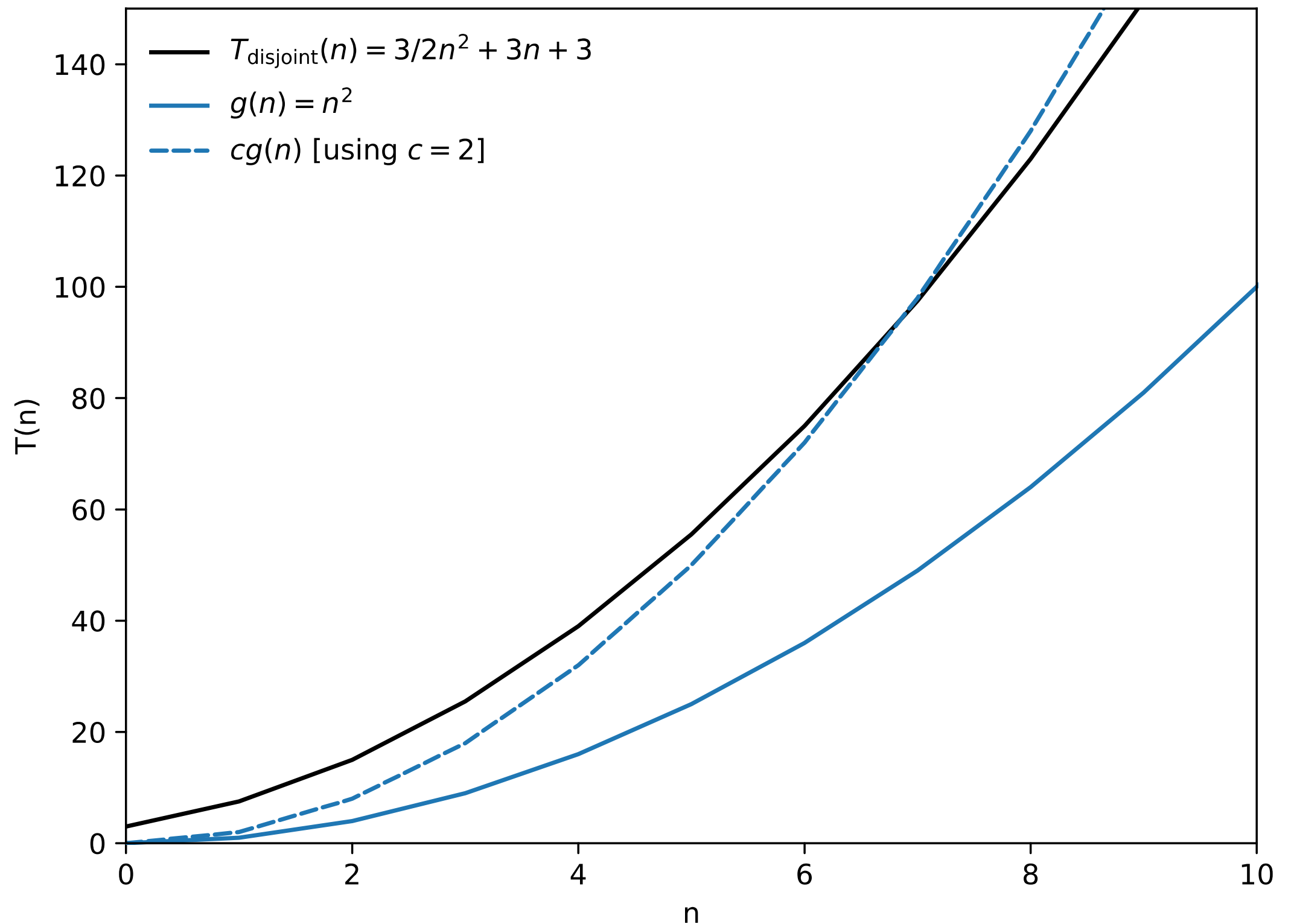
Let's try to narrow down what we mean by “order of growth”



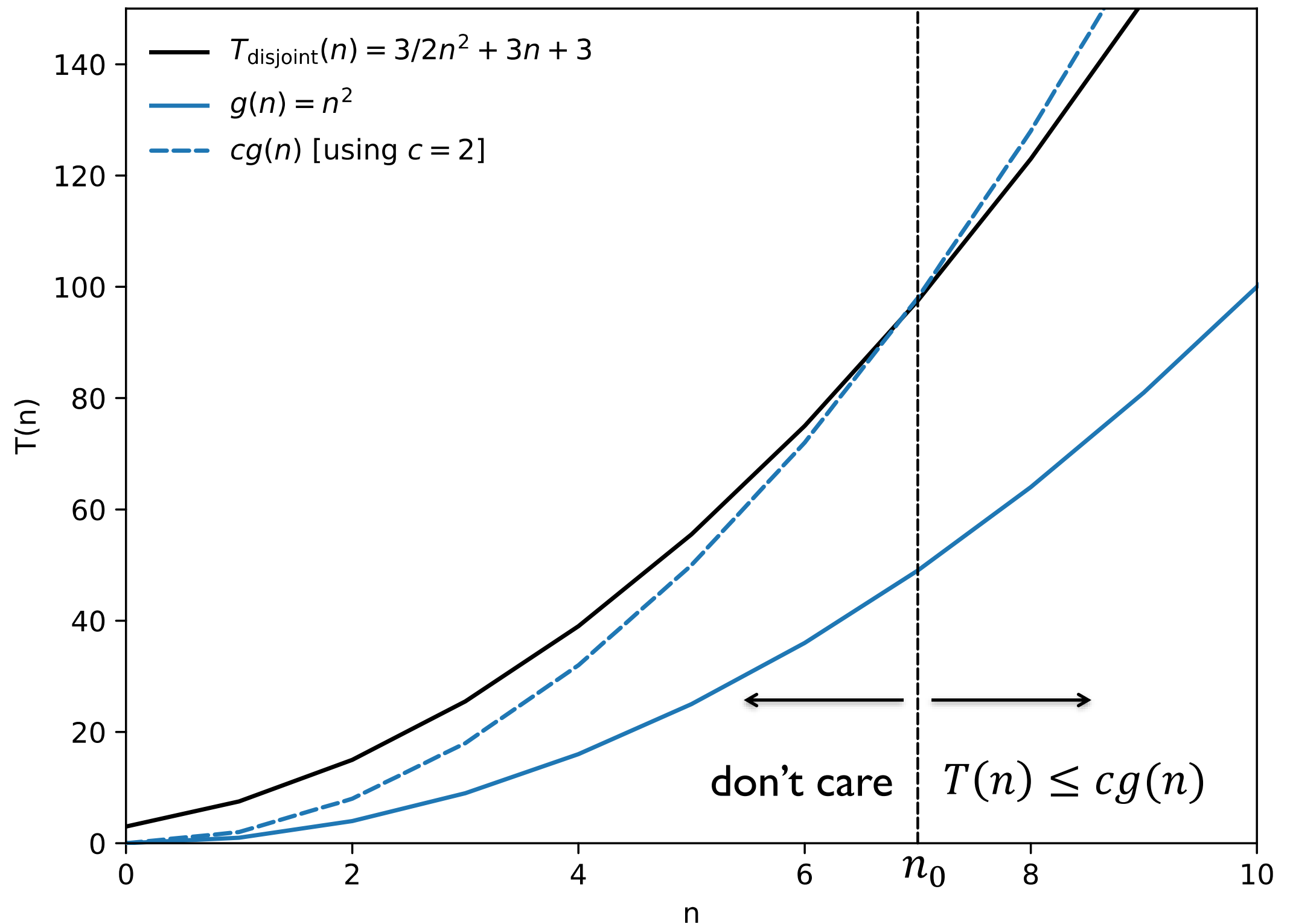
Intuitively, simple function $g(n)$ has same order of growth as $T(n)$. Why?



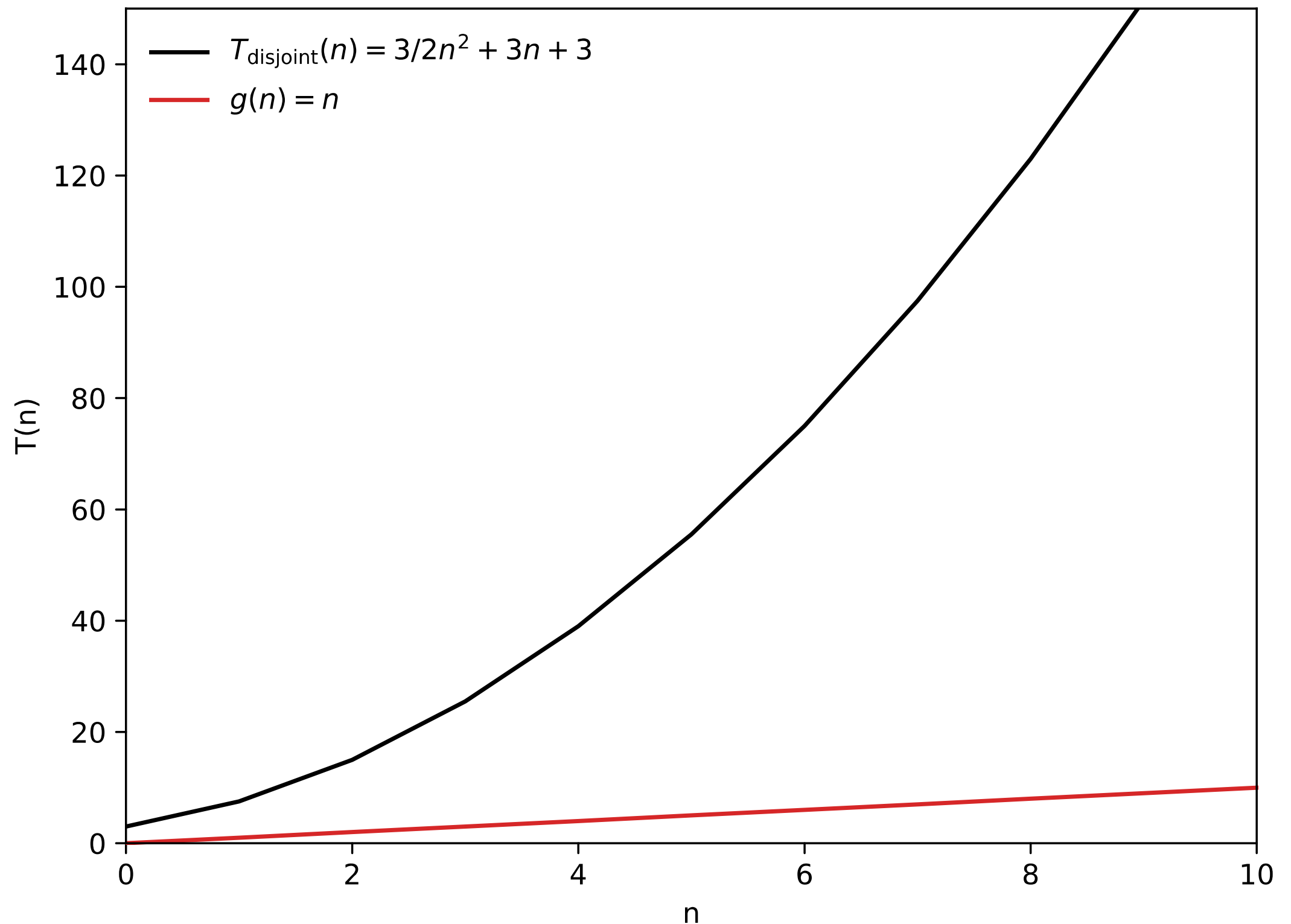
We can simply “scale it up” by a constant c ...



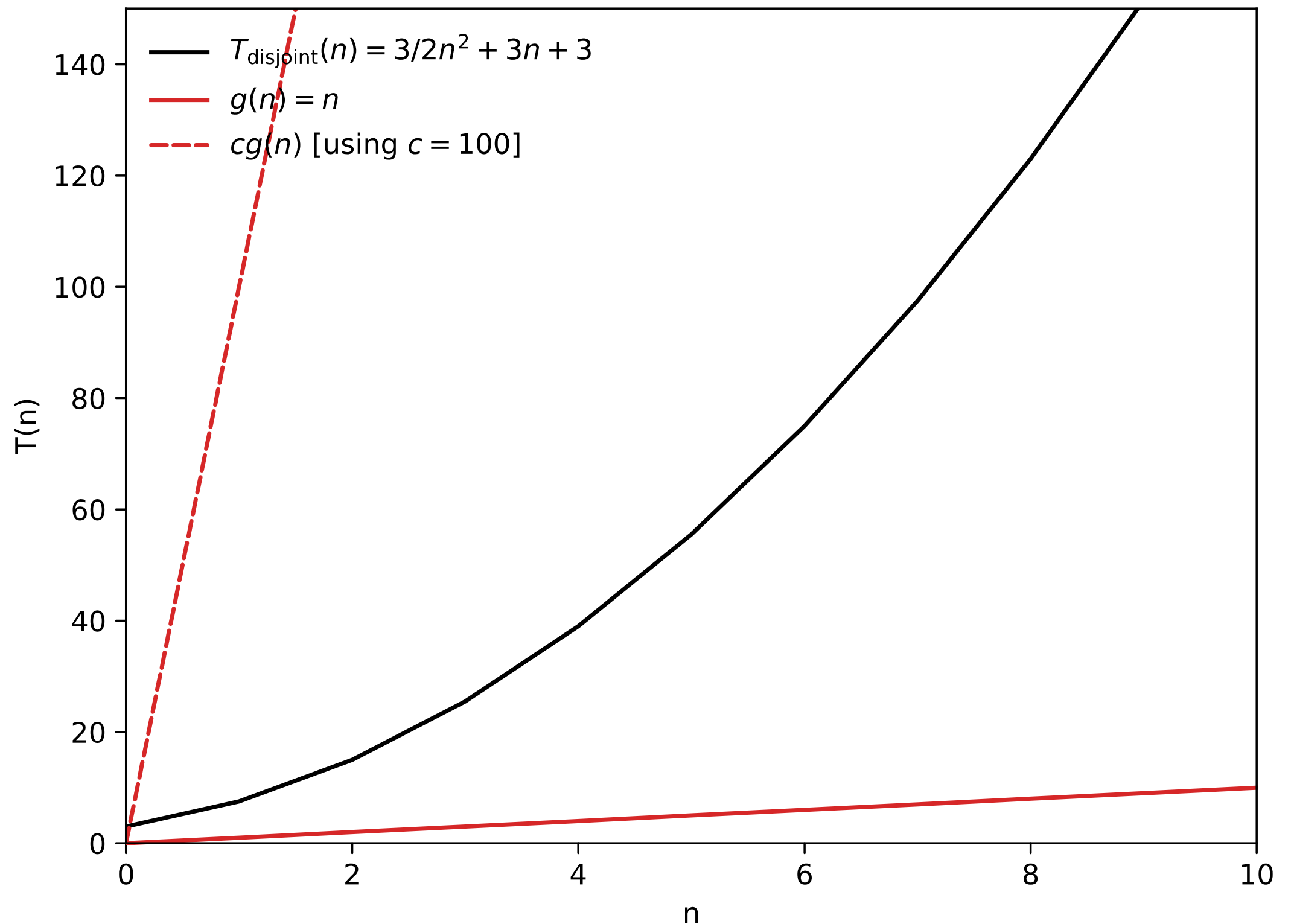
...and it dominates $T(n)$ for all but a finite prefix of input sizes!



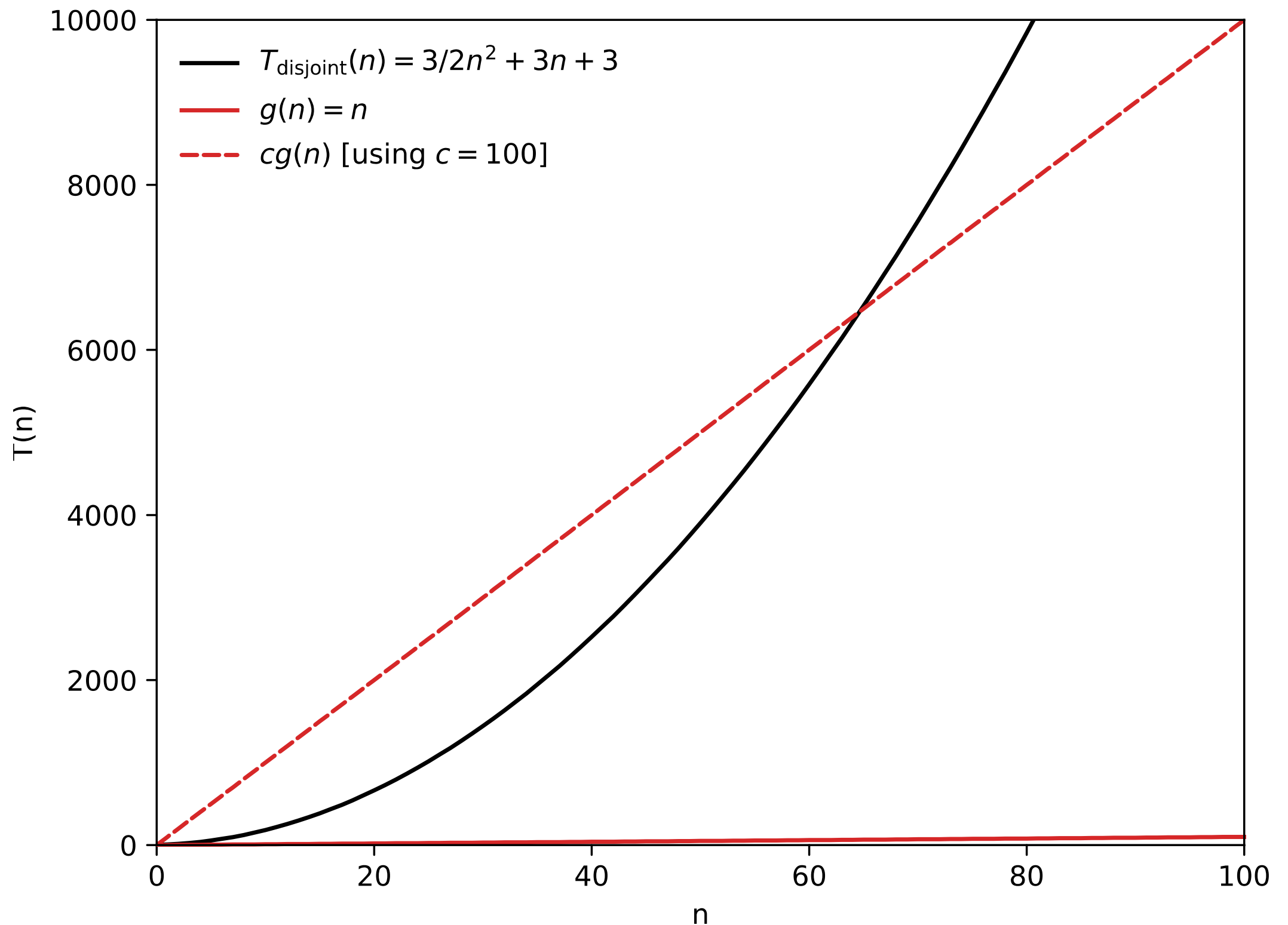
Could we also have used linear function as upper bound?



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 $T(n) \leq cg(n)$.

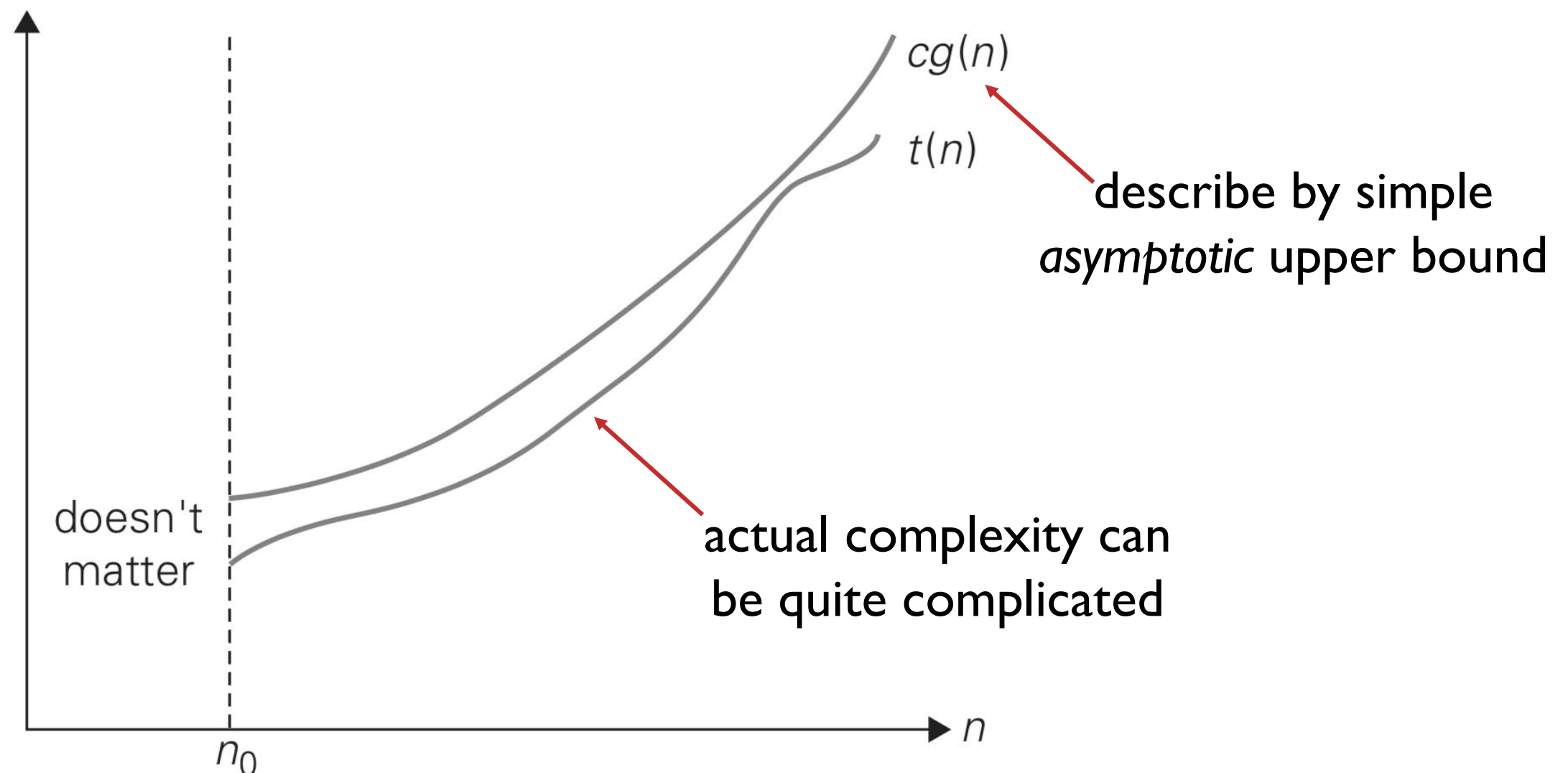
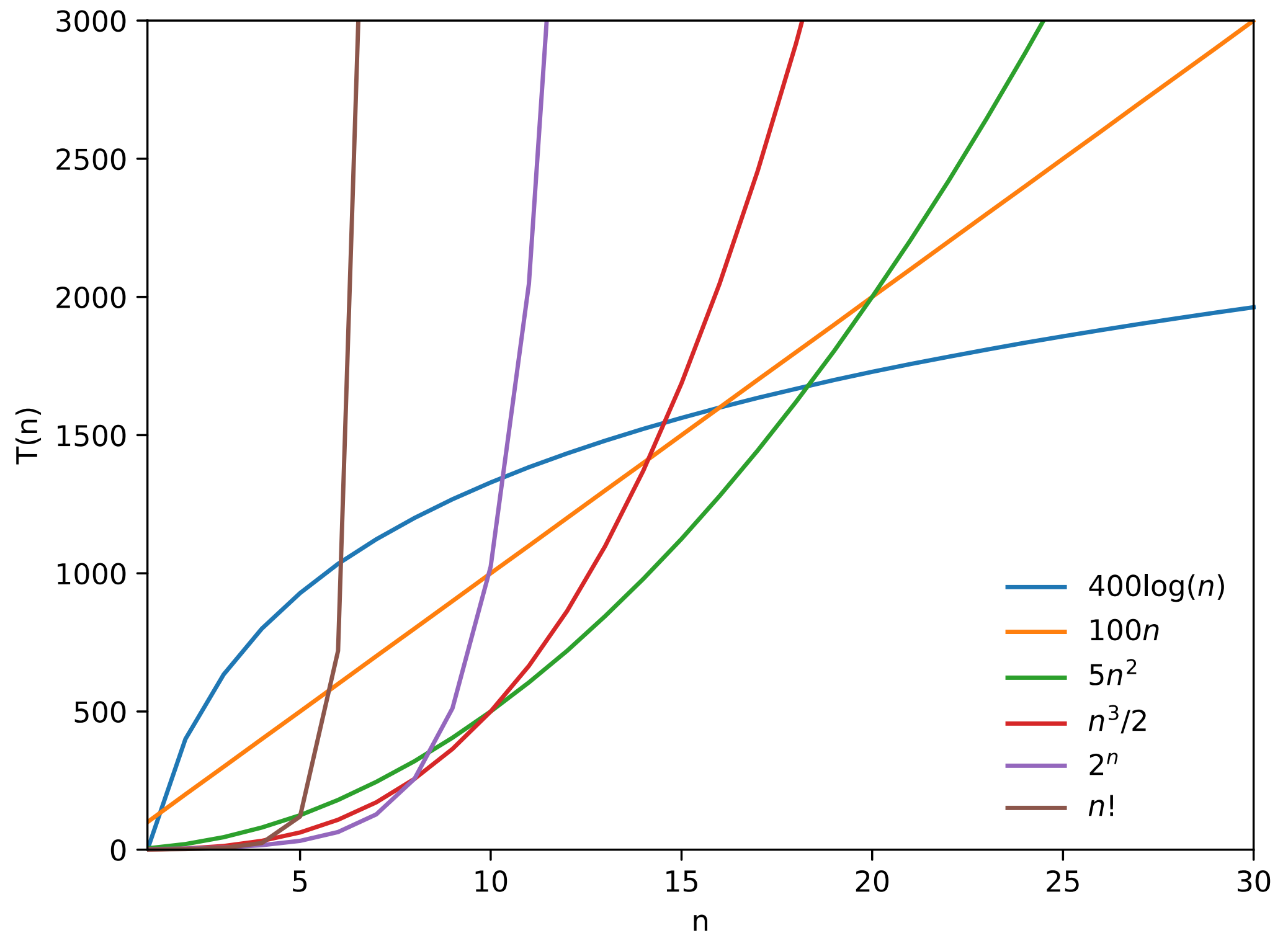


illustration: [Levitin, p. 54]



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		

Measured in nanoseconds (10⁻⁹ secs)

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
13.8x10⁹ years old.

Measured in nanoseconds (10⁻⁹ secs)

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
```

0	0
$2n - 1$	$2n - 1$
$n - 1$	$n - 1$
0	$n - 1$
0	0

$3n - 2$	$4n - 3$
$O(n)$	$O(n)$

Involves

- check if last i
- increment i

n

$n - 1$

Total

$2n - 1$

Asymptotically no
difference between
best and worst case!

Simpler way to arrive at same conclusion

```
def min_index(lst):
```

```
    k = 0
```

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

```
        if lst[i] < lst[k]:
```

```
            k = i
```

```
    return k
```

$O(0)$

$O(0)$

$O(n)$

$O(n)$

$O(n)$

$O(n)$

$O(0)$

$O(n)$

$O(0)$

$O(0)$

$3n - 2$

$4n - 3$

$O(n)$

$O(n)$

Involves

- check if last i
- increment i

$O(n)$

$O(n)$

Total

$O(n)$

Asymptotically no
difference between
best and worst case!

General Observation

If T_1 in $O(g_1(n))$ and T_2 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]
```

$O(n)$

$O(n^2)$

$O(n)$

$O(n^2)$

Total cost of min_index calls

$i = 0$ $i = 1$ $i = 2$ $i = n/2 - 1$ $i = n - n/2$ $i = n - 1$

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

$n + 1$
 \dots
 $n + 1$
 $n + 1$
 $n + 1$

$n/2$ times

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

Let's analyse Insertion Sort

<code>def insert(i, lst):</code>	→	<code>n = i</code>	
<code> j = i</code>	→	<code>O(0)</code>	<code>O(0)</code>
<code> while j > 0 and lst[j-1] > lst[j]:</code>	→	<code>O(1)</code>	<code>O(n)</code>
<code> lst[j-1], lst[j] = lst[j], lst[j-1]</code>	→	<code>O(0)</code>	<code>O(n)</code>
<code> j = j - 1</code>	→	<code>O(0)</code>	<code>O(n)</code>
		<code>O(1)</code>	<code>O(n)</code>

<code>def insertion_sort(lst):</code>	→	<code>n = len(lst)</code>	
<code> for i in range(1, len(lst)):</code>	→	<code>O(n)</code>	<code>O(n)</code>
<code> insert(i, lst)</code>	→	<code>O(n)</code>	<code>O(n²)</code>
		<code>O(n)</code>	<code>O(n²)</code>

Total cost of insert calls

- **best case** (sorted input): $1 + 1 + 1 + \dots + 1 \quad \overset{n \text{ times}}{=} O(n)$
- **worst case** (inversely sorted input): $1 + 2 + \dots + (n - 1) + n \quad \overset{= (n + 1)n/2 \text{ as previously}}{=} O(n^2)$

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)

Coming Up

- Logarithmic complexity and the decrease-and-conquer paradigm
- More efficient search
- Analysis of Euclid's Algorithm