Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 10
Complexity

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

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 o
communication of this material by you may be the subject of copyright protection under the Act.
Do not remove this notice.

Running Time and RAM

Insertion sort

Binary Search

Big O

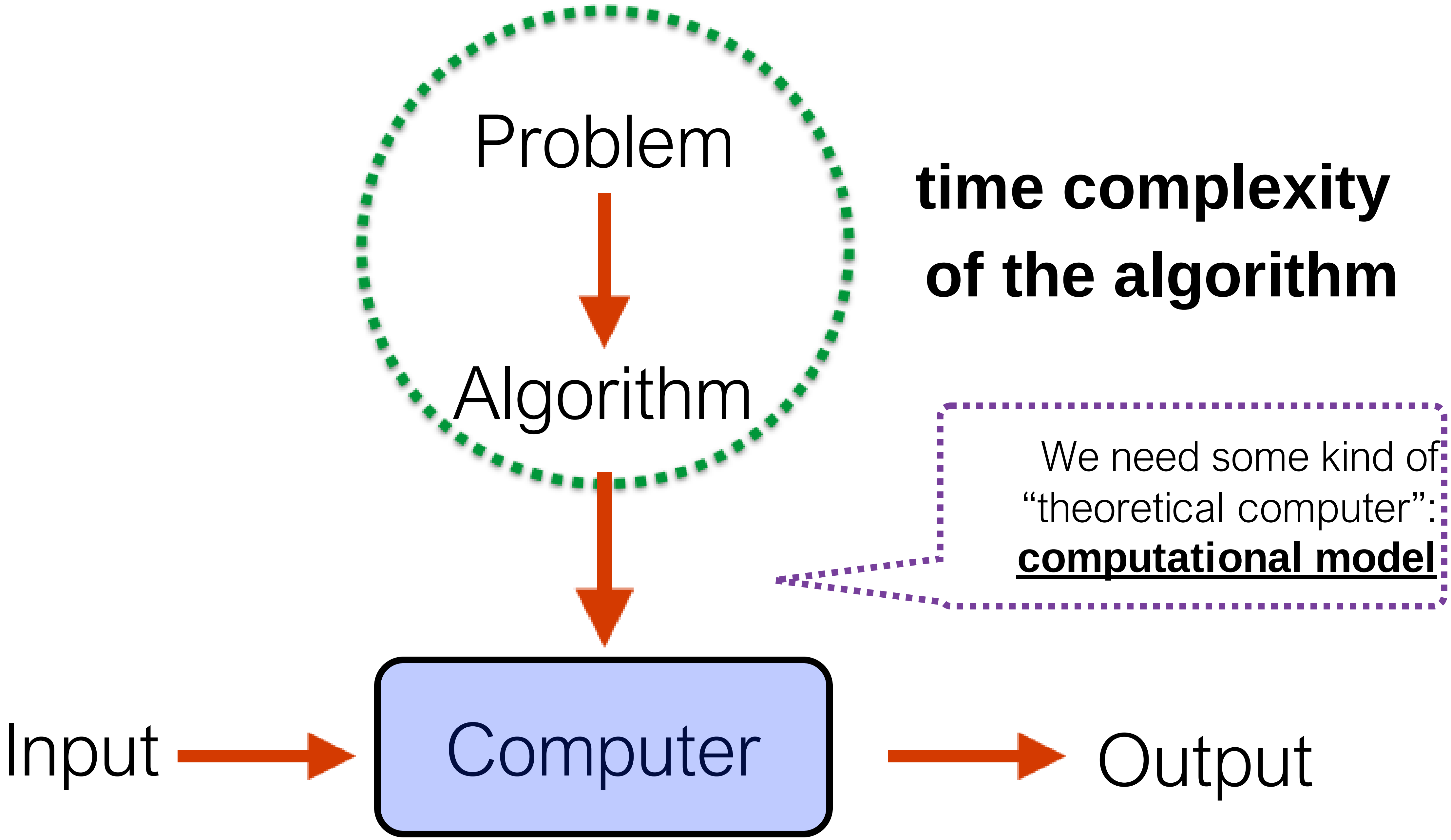
Growth rates

Running Time

Depends on a number of factors including:

- The **input**
- The quality of the code generated by the **compiler**
- The nature and **speed** of the instructions on the **machine** used to execute the program
- The **time complexity of the algorithm**

Jamaica's Usain Bolt celebrating after winning the final of the men's 100 metres athletics event at the 2015 IAAF World Championships in Beijing. AFP PHOTO / PEDRO UGARTE


Problem
Algorithm
time complexity
of the algorithm
We need some kind of
“theoretical computer”:
computational model
Input
Computer
Output

Simple computation model

- Each simple operation takes one step
  (e.g., **assignment**, **print** or **return** statement).
- Each **comparison** takes one time step.
- Running time of **a sequence of statements** = **Sum** of the running time of the **statements**.
- **Loops** and **modules**
  - Composition of many simple operations,
    and their running time
  - Depends on how many times each of these simple operations are performed.

RAM model = abstract machine

Algorithm FindMin(L[0..n-1])

Finds minimum element in a list

Input: A list L[0, n-1] of real numbers

Output: A list sorted in ascending order.

```
min ⟵ list[0]                          } 2 assignments
k ⟵ 0
while (k < n) do {  1 comparison
      if list[k] < min do {  1 comparison
           min ⟵ list[k]       1 assignment
      }
      k ⟵ k+1  1 assignment
}
return min  1 return
```

n-1 times

Does running time depend only on n?

If it always enters the if: 2 + 4(n-1) + 1 +1 = **4 + 3(n-1)**

If it never enters the if: 2 + 3(n-1) + 1 +1 = **4 + 2(n-1)**

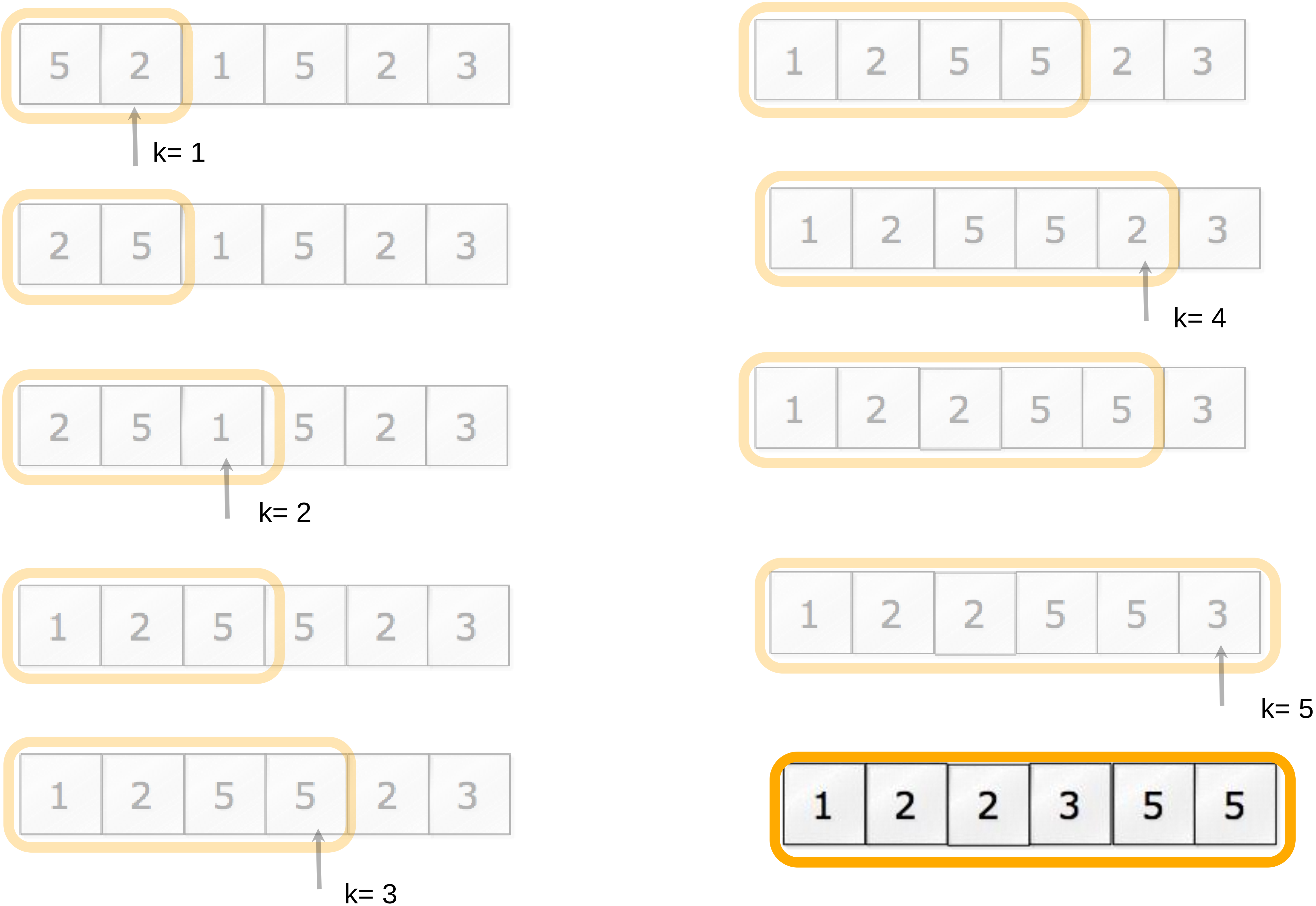
This difference is unimportant, when considering the big picture

(we will discuss this again at the end)

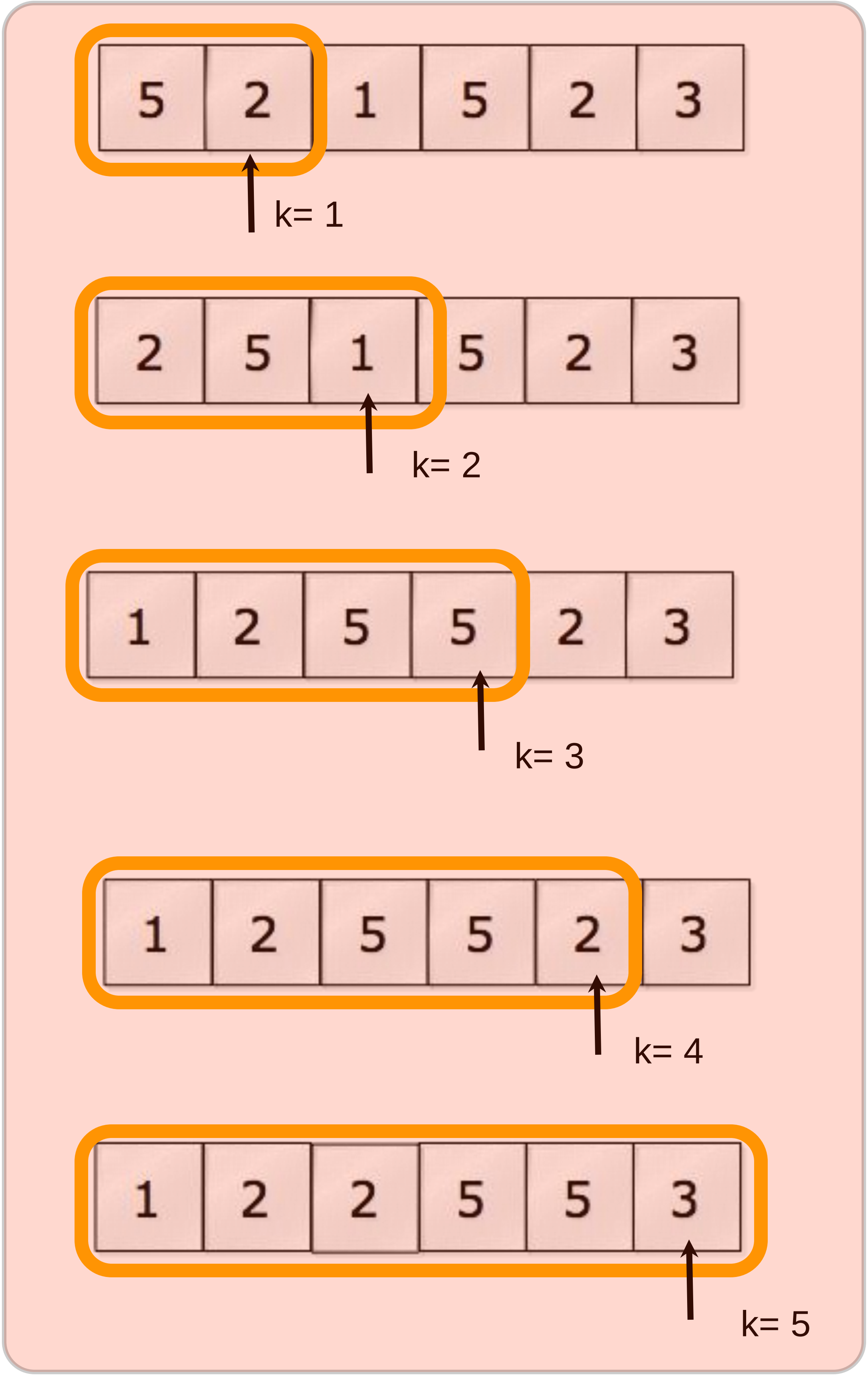
Insertion sort

(take last, put slowly in correct position, enlarge)

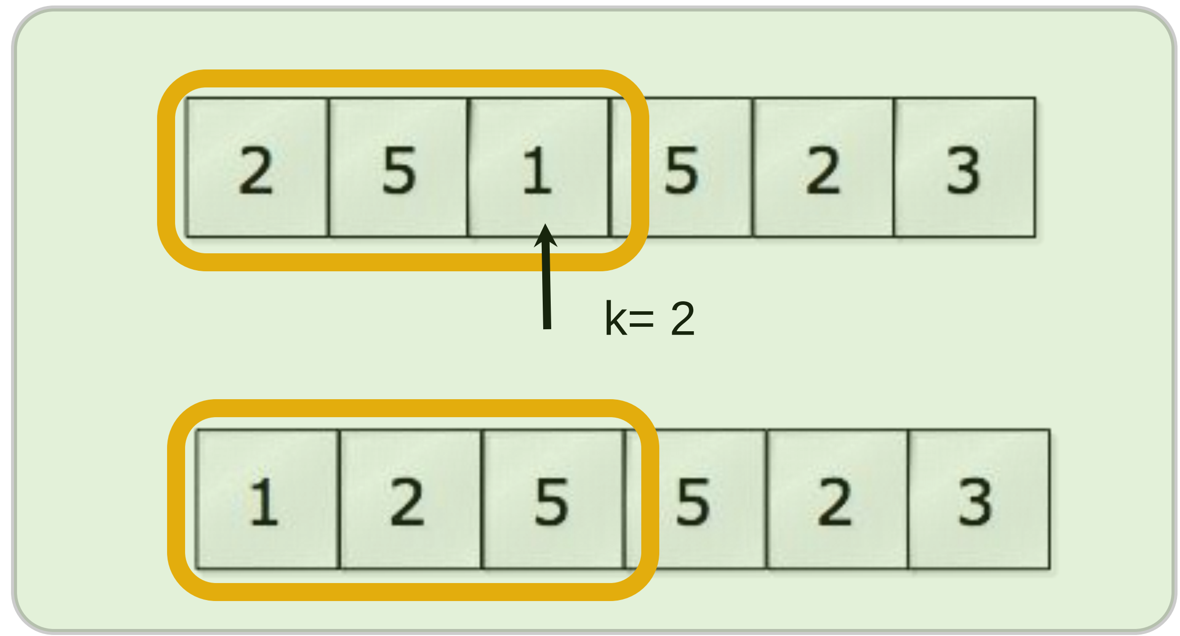
L



Loop 1 – I bring another into our home



Loop 2 – I find each item a bed to stay in



Algorithm InsertionSort(L[0..n-1])

Sorts a list using insertion sort.

Input: A list L[0, n-1] of real numbers

Output: A list sorted in ascending order.

```
k ⟵ 1                                          1 assignment
while (k < n) do {                             1 comparison
      tmp ⟵ list[k]                            2 assignments
      j ⟵ k-1
      while (j ≧ 0 and tmp < list[j]) do {     2 comparisons
          list[j+1] ⟵ list[j]                  2 assignments
          j ⟵ j – 1
      }

      list[j+1] ⟵ tmp                          2 assignments
      k ⟵ k + 1
   }
}
```

At most k times $k < n$

n-1 times

Running time does not depend on **n** only

Best and Worst Case

Insertion Sort: Time complexity

```
k ⟵ 1
while (k < n) do {
      tmp ⟵ list[k]
      j ⟵ k-1
      while (j ≧ 0 and tmp < list[j]) do {
         list[j+1] ⟵ list[j]
         j ⟵ j – 1
      }

      list[j+1] ⟵ tmp
      k ⟵ k + 1
   }
}
```

- Can we stop any of the two loops early?
  - Yes, the second one, when tmp ≧ list[j]
  - Best and worst cases are going to be different
  - The average case lies in between them.
- Best case?
  - [1, 2, 3, 4]
- Worst case?
  - [4, 3, 2, 1]

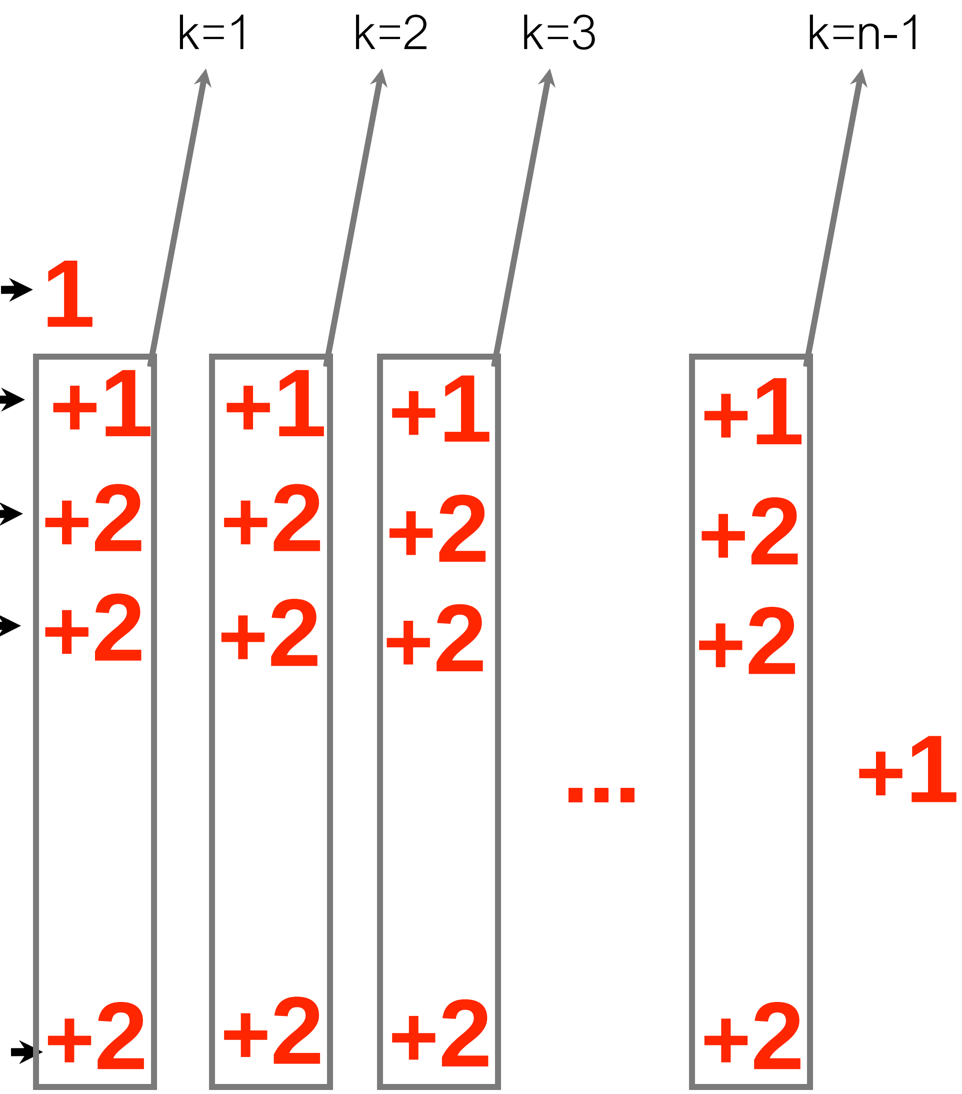
We often refer to best AND worst case complexities

Best case

```
k ← 1                                          1
while (k < n) do {                             1
        tmp ← list[k]
        j ← k-1                                2
        while (j ≧ 0 and tmp < list[j]) do {   2
            list[j+1] ← list[j]
            j ← j-1                            2
        }
        list[j+1] ← tmp
        k ← k + 1                              2
    }
}
```

Not used

n-1 times



$1 + 7(n-1) + 1$

$\boxed{7n - 5}$

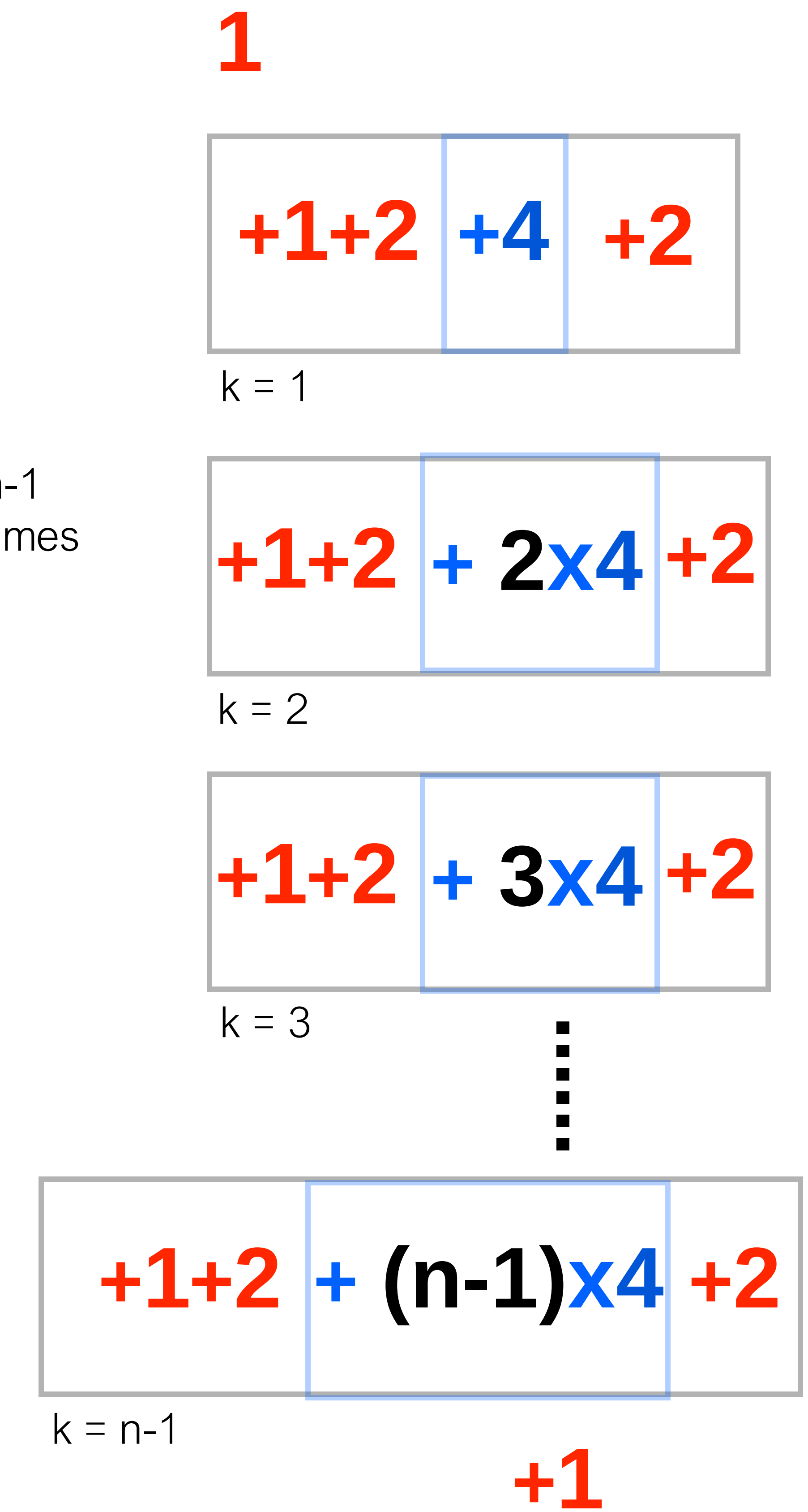
Worst case

```
k ⟵ 1                                          1
while (k < n) do {                             1
      tmp ⟵ list[k]                            
      j ⟵ k-1                                  2
      while (j ≧ 0 and tmp < list[j]) do {
          list[j+1] ⟵ list[j]                  4
          j ⟵ j – 1
      }

      list[j+1] ⟵ tmp
      k ⟵ k + 1                                2
   }
}
```

k times max

n-1 times



Worst case

```
k ⟵ 1                                          1
while (k < n) do {                             1
        tmp ⟵ list[k]
        j ⟵ k-1                                2
        while (j ≧ 0 and tmp < list[j]) do {
            list[j+1] ⟵ list[j]                4
            j ⟵ j – 1
        }

        list[j+1] ⟵ tmp
        k ⟵ k + 1                              2
    }
}
```

k
times
max

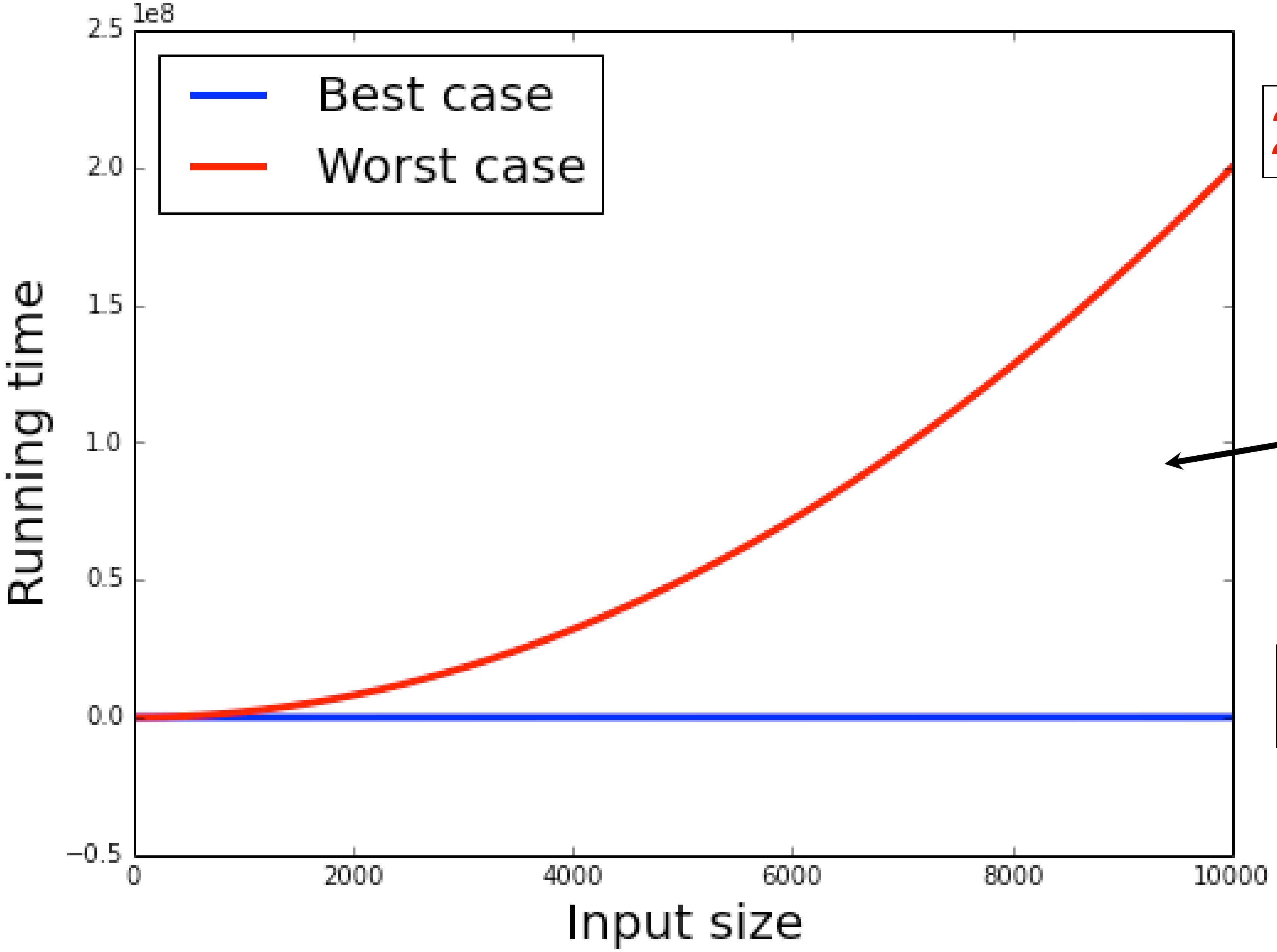
n-1
times

$$5(n-1) + (1 \times 4 + 2 \times 4 + 3 \times 4 + \ldots + (n-1) \times 4) \quad \mathbf{+2}$$

$$5(n-1) + (2n(n-1)) + 2$$

$$\boxed{2n^2 + 3n - 3}$$

Insertion Sort running time



$2n^2 + 3n - 3$

average case must be somewhere in the middle

$7n - 5$

- Select **how to measure the input size**.
- If running time depends only on the input size, then that's great.

- If running time depends on input size **and other characteristics** of the input:
  - Analyse best case separately *(can I leave any loops early).*
  - Analyse worst case separately.
  - Together best and worst case are informative.

Insertion Sort: Code

```
k ⟵ 1
while (k < n) do {
      tmp ⟵ list[k]
      j ⟵ k-1
      while (j ≥ 0 and tmp < list[j]) do {
         list[j+1] ⟵ list[j]
         j ⟵ j – 1
      }

      list[j+1] ⟵ tmp
      k ⟵ k + 1
   }
}
```

```python
def insertion_sort(the_list):
    n = len(the_list)
    for k in range(1, n):
        temp = the_list[k]
        i = k - 1
        while i >= 0 and the_list[i] > temp:
            the_list[i + 1] = the_list[i]
            i -= 1
        the_list[i + 1] = temp
```

Binary Search Assumptions

- The list is sorted

- We can random access the list
  (you can get the value of any position in the list)

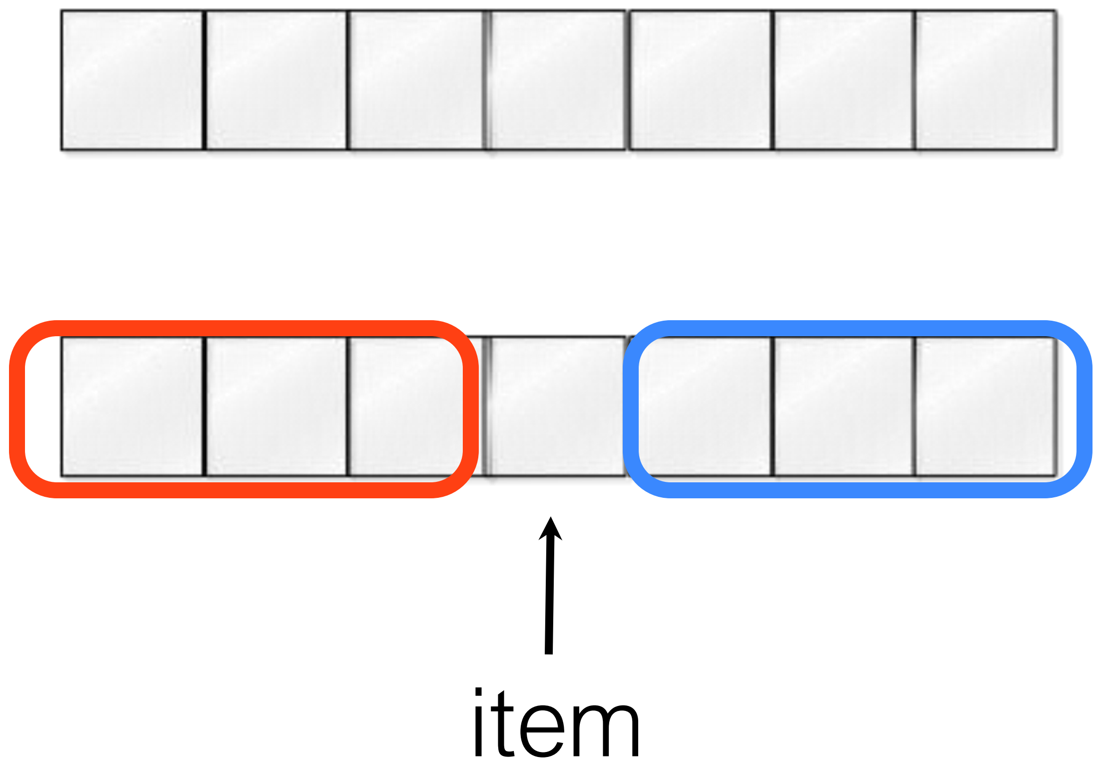
Binary Search

```
item ⟵ the item in the middle of the list

if (item = target)
{
        return index of item
}

if (target < item)
{
        search the first part of the list
}

if (target > item)
{
        search the second part of the list
}
```



item < target　　item = target　　item > target

Binary Search

```
 Algorithm BinarySearch(target,  L[0..n-1])
// Find the index such that L[index] = target
// Input:  target and list L[0..n-1]
// Output:  If target is in L,  return the index of the first
// item with that value. Otherwise return -1.

lower ⟵ 0
upper ⟵ n-1
while (lower ≤ upper) do {
      mid =  ⌊ (lower + upper)/2 ⌋
      if (target == L[mid])
           return  mid
      if (target < L[mid])
            upper = mid – 1
      if (target > L[mid])
            lower = mid + 1
}
return -1
```

Worst case

$\boxed{6 \log_2(n) + 4}$

$2 + \log_2(n)\ (1 + 1 + 1 + 3) + 1 + 1$

```
lower ⟵ 0
upper ⟵ n-1
while (lower ≤ upper) {
    mid ⟵ ⌊(lower + upper)/2⌋
    if (target = L[mid])
        return mid
    if (target < L[mid])
        upper ⟵ mid – 1
    if (target > L[mid])
        lower ⟵ mid + 1
}
return -1
```

2 assignments

1 comparison

1 assignment

1 comparison

1 return

3 operations

1 return

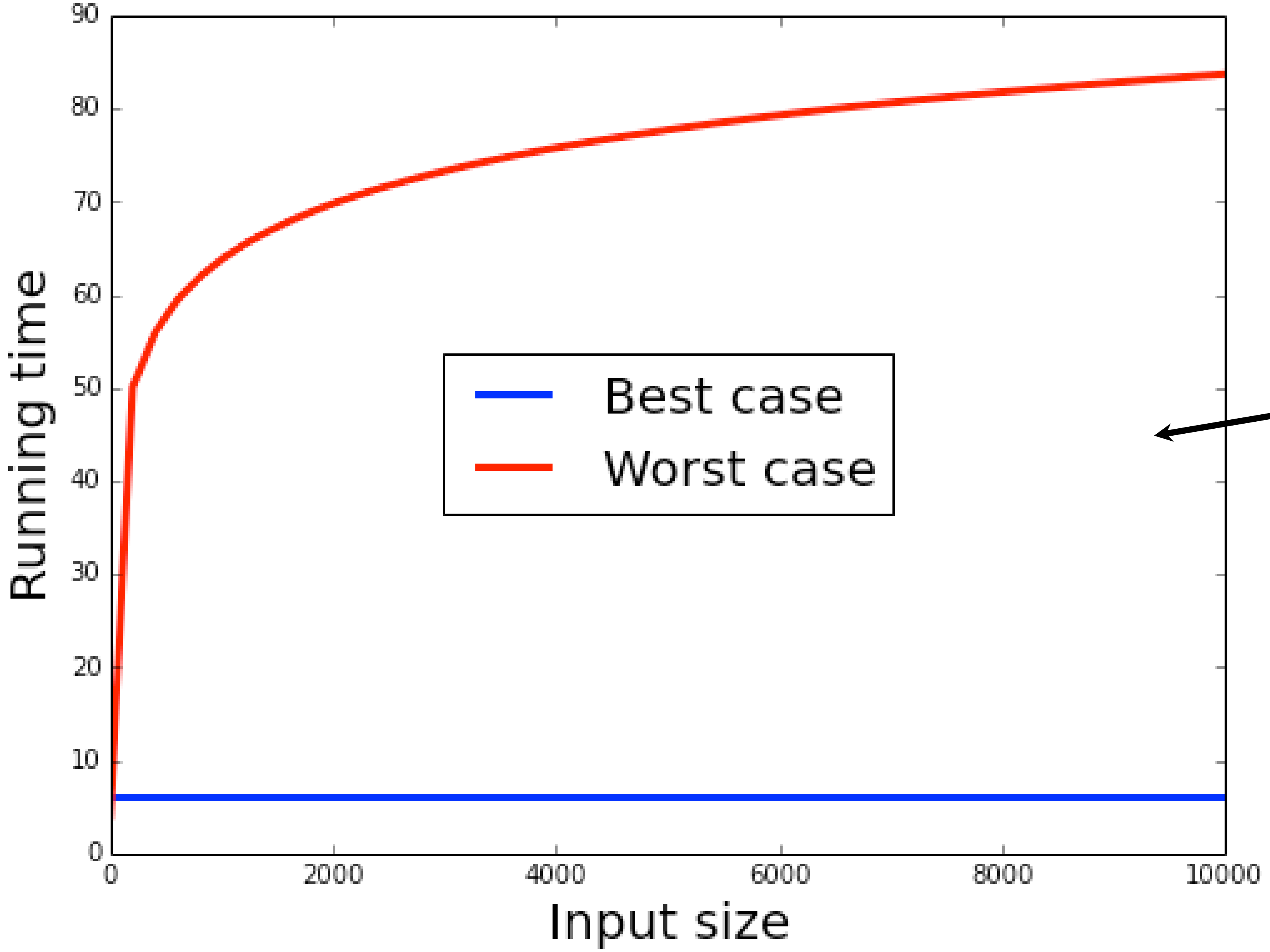
Target not in List

at most
$\log_2(n)$
times

Big O

Focus on the big picture

Binary Search running time



$6 \log_2(n) + 4$

average case must be somewhere in the middle

6

$6 \log_2(n) + 4$

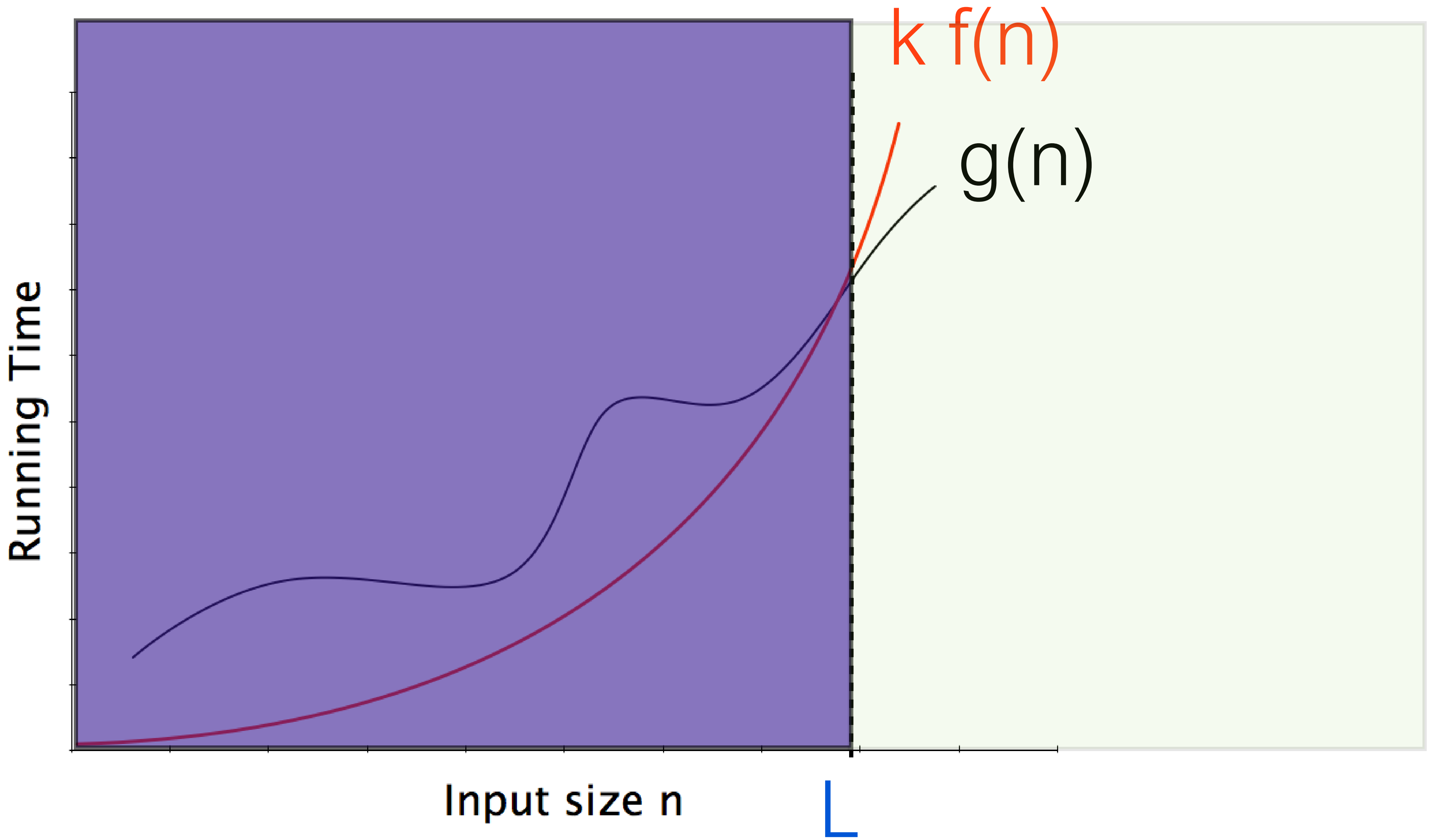
```
n = 1, 4.0 = 0.0 + 4.0
n = 2, 10.0 = 6.0 + 4.0
n = 3, 13.5 = 9.5 + 4.0
n = 5, 17.9 = 13.9 + 4.0
n = 10, 23.9 = 19.9 + 4.0
n = 100, 43.9 = 39.9 + 4.0
n = 1000, 63.8 = 59.8 + 4.0
n = 10000, 83.7 = 79.7 + 4.0
n = 100000, 103.7 = 99.7 + 4.0
n = 1000000, 123.6 = 119.6 + 4.0
```

Ignore parts that do not contribute significantly, when the input is large

Big O notation

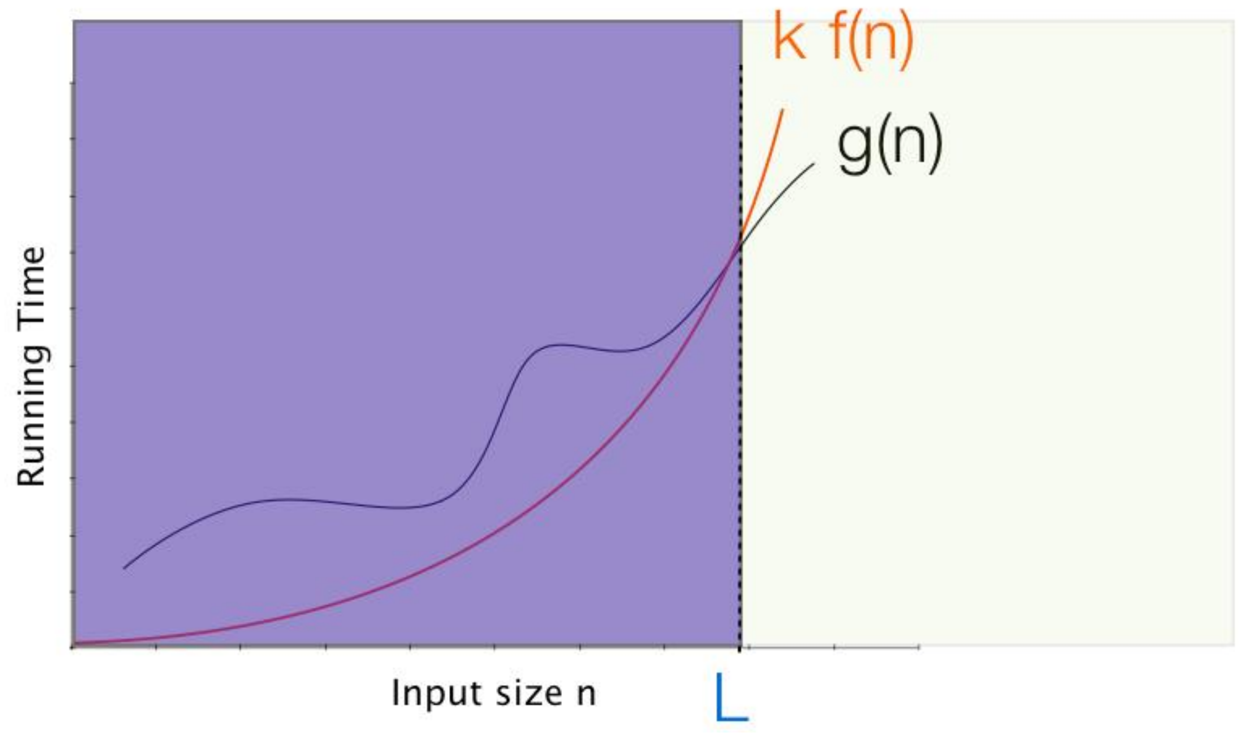
Function **g(n)** is said to be **O(f(n))** if there exist constants **k** and **L** such that:

$$g(n) < k*f(n) \text{ for all } n > L$$



Big O notation

g(n) is **O(f(n))**



- Intuitively:
  f(n) gives an **upper bound** to running time g(n), which:
- ignores parts of the algorithm that do not contribute significantly to the total running time
- bounds the error made when ignoring small terms in g

Big O gives us an idea of g(n)'s behaviour for **large inputs**. Simple but formal.

Big O notation

Ignore constants

Ignore parts that do not contribute significantly

Algorithm FindMin(L[0..n-1])

Finds minimum element in a list

Input: A list L[0, n-1] of real numbers

Output: A list sorted in ascending order.

```
min ← list[0]                          } 2 assignments
k ← 0
while (k < n) do {   1 comparison
    if list[k] < min do {   1 comparison
        min ← list[k]       1 assignment
    }
    k ← k+1   1 assignment
}                                       n-1 times
return min   1 return
```

In Big O best = worst O(n)

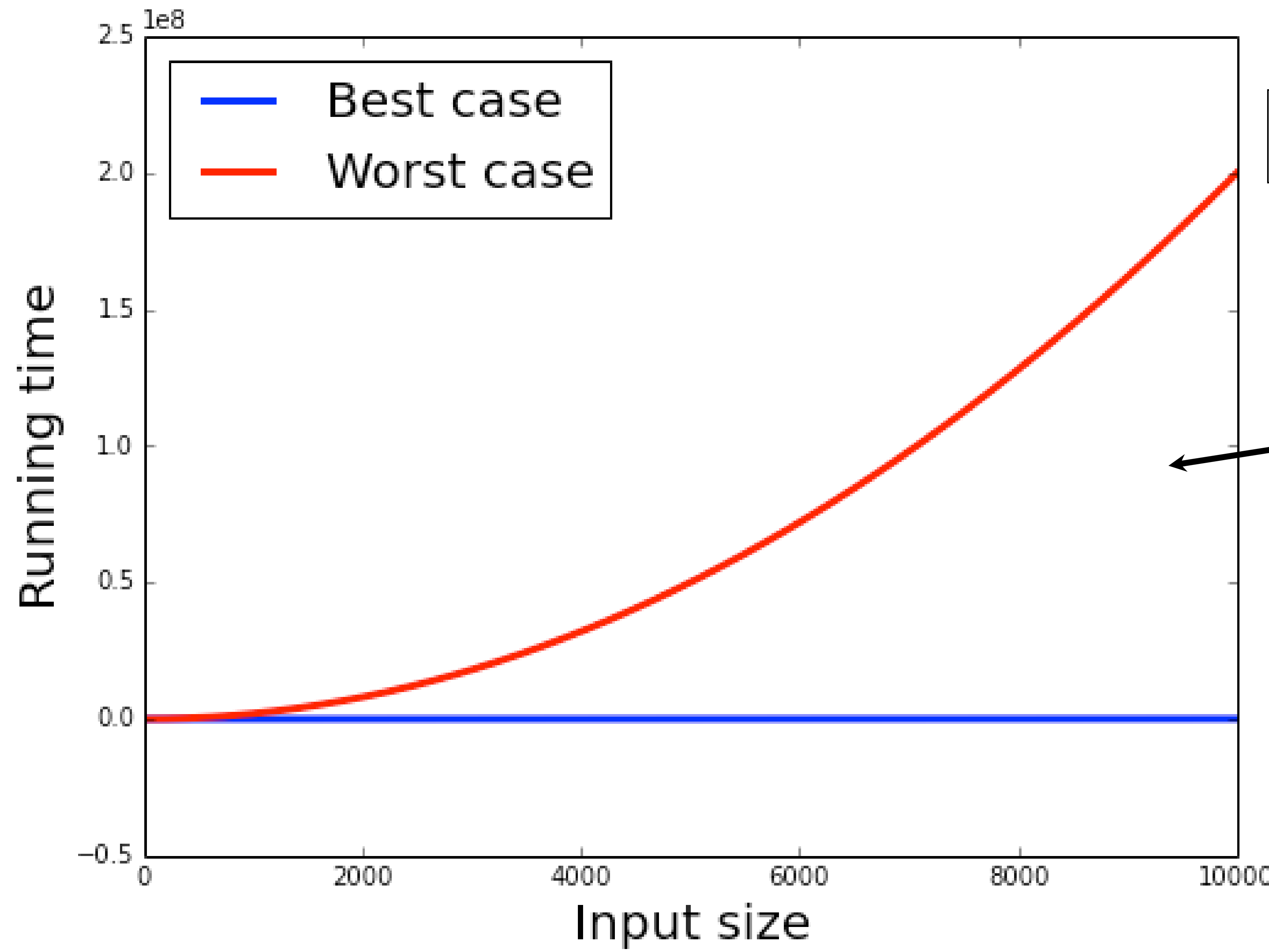
If it always enters the if: $2 + 3(n-1) + 1 + 1 = \mathbf{4 + 3(n-1)}$

If it never enters the if: $2 + 2(n-1) + 1 + 1 = \mathbf{4 + 2(n-1)}$

This difference is unimportant, when considering the big picture

(we will discuss this again at the end)

Insertion Sort running time



$2n^2 + 3n - 3$

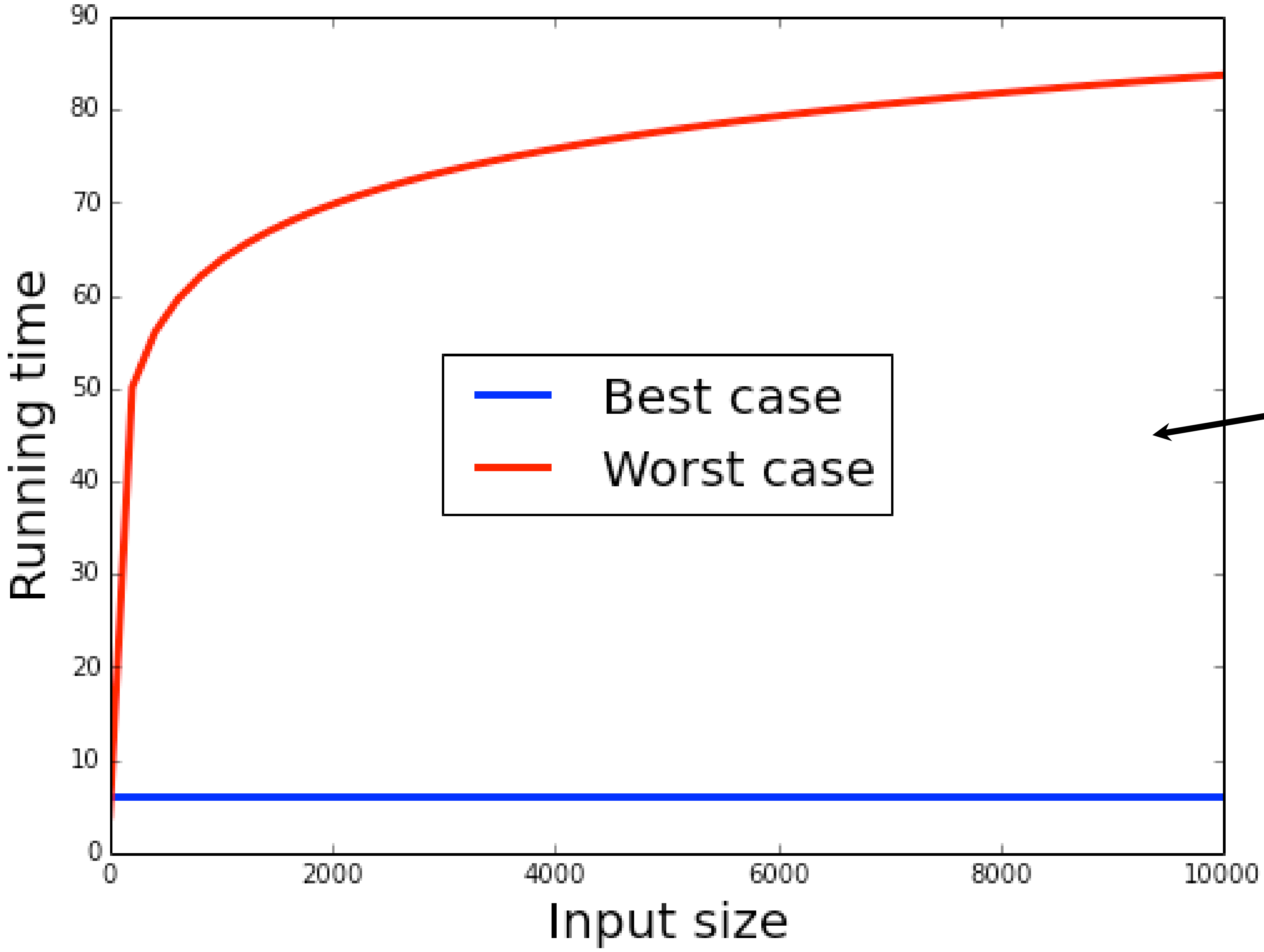
average case must be somewhere in the middle

$7n - 5$

Best case O(n)

Worst case O(n^2)

Binary Search running time



$6 \log_2(n) + 4$

average case must be somewhere in the middle

6

Best case O(1)

Worst case O(log n)

Basic efficiency classes

In order of increasing time complexity:

• Constant	$O(1)$	😎
• Logarithmic	$O(\log N)$	😆
• Linear	$O(N)$	😃
• Superlinear	$O(N \log N)$	🙂
• Quadratic	$O(N^2)$	😒
• Exponential	$O(2^N)$	😰
• Factorial	$O(N!)$	😱
		💩

Constant	O(1)	Running time does not depend on N	N doubles, T remains constant
Logarithmic	O(log N)	Problem is broken up into smaller problems and solved independently. Each step cuts the size by a constant factor.	If N doubles, running time T gets slightly slower
Linear	O(N)	Each element requires a certain (fixed) amount of processing	If N doubles, running time T doubles (2*T)
Superlinear	O(N log N)	Problem is broken up in sub-problems. Each step cuts the size by a constant factor and the final solution is obtained by combining the solutions.	If N doubles, running time T gets slightly bigger than double (2*T and a bit)
Quadratic	$O(N^2)$	Processes pairs of data items. Often occurs when you have double nested loop	If N doubles, running time T increases four times (4*T)
Exponential	$O(2^N)$	Combinatorial explosion (think about a family tree)	If N doubles, running time T squares (T*T)
Factorial	O(N!)	Finding all the permutations of N items	

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	8.4×10^{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	4×10^{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)

Points to keep in mind

- Big-O gives an **upper bound**, which may be much larger than the actual value.
- The input that produces the **worst case may be very unlikely** to occur.
- 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.
- **Other criteria**: 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.