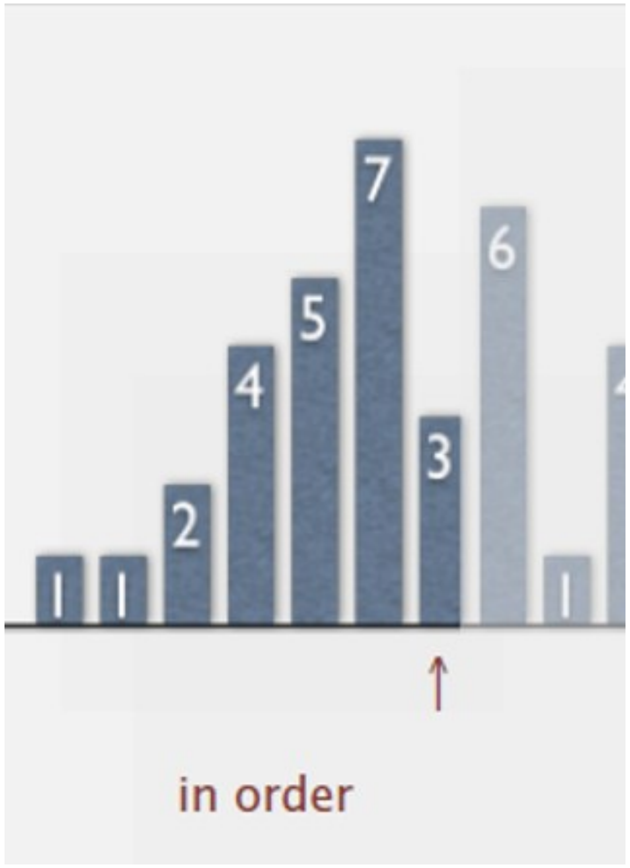
Insertion Sort: Idea

- Strategy: The list will have a sorted portion followed by an unsorted portion; initially the sorted portion contains one list element
- The basic operation is to take the first element of the unsorted portion of the list and place it in its correct position within the sorted portion of the list
- Procedure: Perform the basic operation successively at list positions 2 through n

Insertion Sort: Idea

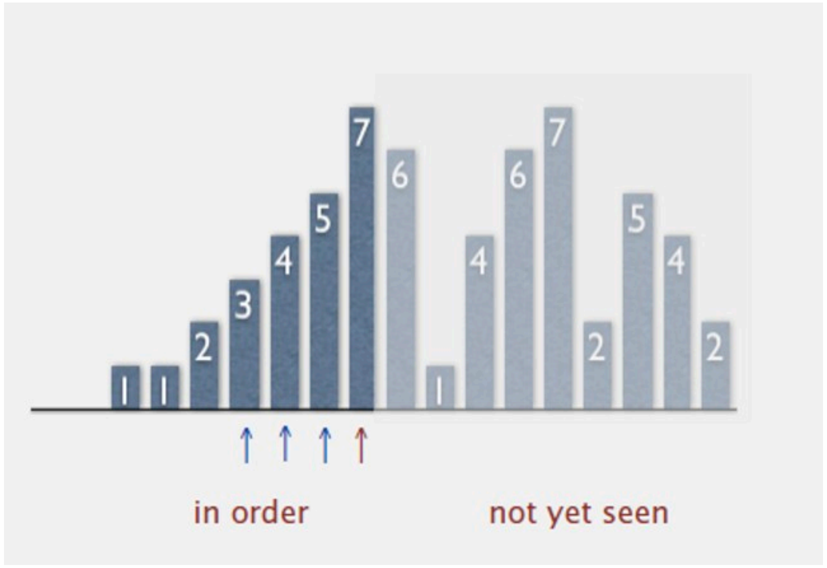
Scan data/array from left to right (scan is at ↑)

- Entries to the left of ↑ are in non-decreasing order
- Entries to the right of ↑ are in the original order



Insertion Sort: Idea

Maintain the invariant condition as ↑ moves one position to the right



Move 3 left to correct position

Insertion Sort: Idea

[3, 1, 2]

[1, 3, 2]

[1, 2, 3]

Insertion Sort (version in KR text)

```
Procedure Insertion Sort($a_1$, …, $a_n$)
  for $j$:= 2 to $n$ {$a_1$,…,$a_{j-1}$ are now sorted}
    $m$:= $a_j$        {set aside $a_j$ to insert later}
    $i$:= 1
    while $a_j > a_i$ do   {find insert position}
        $i$:= $i$+1
    for $k$:=0 to $j-i-1$     {open insert position}
        $a_{j-k}$:= $a_{j-k-1}$      {by moving elements >$a_j$}
    $a_i$:= $m$            {insert $a_j$ in position $i$}
  endfor {$a_1$,$a_2$, …,$a_n$ are now sorted}
```

What is compared when and how does data move?

Insertion Sort (version in KR text)

```
Procedure Insertion Sort($a_1$, …, $a_n$)
  for $j$:= 2 to $n$ {$a_1$,…,$a_{j-1}$ are now sorted}
    $m$:= $a_j$        {set aside $a_j$ to insert later}
    $i$:= 1
    while $a_j > a_i$ do   {find insert position}
        $i$:= $i$+1
    for $k$:=0 to $j-i-1$     {open insert position}
        $a_{j-k}$:=  $a_{j-k-1}$       {by moving elements $>a_j$}
    $a_i$:= $m$            {insert $a_j$ in position $i$}
  endfor {$a_1$,$a_2$, …,$a_n$ are now sorted}
```

Insertion Sort (version in KR text)

```
Procedure Insertion Sort($a_1$, …, $a_n$)
  for $j$:= 2 to $n$ {$a_1$,…,$a_{j-1}$ are now sorted}
    $m$:= $a_j$      {set aside $a_j$ to insert later}
    $i$:= 1
    while $a_j > a_i$ do   {find insert position}
        $i$:= $i$+1
    for $k$:=0 to $j-i-1$     {open insert position}
        $a_{j-k}$:=  $a_{j-k-1}$      {by moving elements >$a_j$}
    $a_i$:= $m$            {insert $a_j$ in position $i$}
  endfor {$a_1$,$a_2$, …,$a_n$ are now sorted}
```

Insertion Sort (version in KR text)

```
Procedure Insertion Sort($a_1$, …, $a_n$)
  for $j$:= 2 to $n$ {$a_1$,…,$a_{j-1}$ are now sorted}
     $m$:= $a_j$        {set aside $a_j$ to insert later}
     $i$:= 1
     while $a_j$ > $a_i$ do   {find insert position}
          $i$:= $i$+1
     for $k$:=0 to $j$-$i$-1     {open insert position}
          $a_{j-k}$:=  $a_{j-k-1}$       {by moving elements >$a_j$}
     $a_i$:= $m$             {insert $a_j$ in position $i$}
  endfor {$a_1$,$a_2$, …,$a_n$ are now sorted}
```

Insertion Sort (version in KR text)

```
Procedure Insertion Sort($a_1$, …, $a_n$)
  for $j$:= 2 to $n$ {$a_1$,…,$a_{j-1}$ are now sorted}
    $m$:= $a_j$        {set aside $a_j$ to insert later}
    $i$:= 1
    while $a_j > a_i$ do   {find insert position}
        $i$:= $i$+1
    for $k$:=0 to $j-i-1$      {open insert position}
        $a_{j-k}$:=   $a_{j-k-1}$        {by moving elements >$a_j$}
    $a_i$:= $m$              {insert $a_j$ in position $i$}
  endfor {$a_1$,$a_2$, …,$a_n$ are now sorted}
```

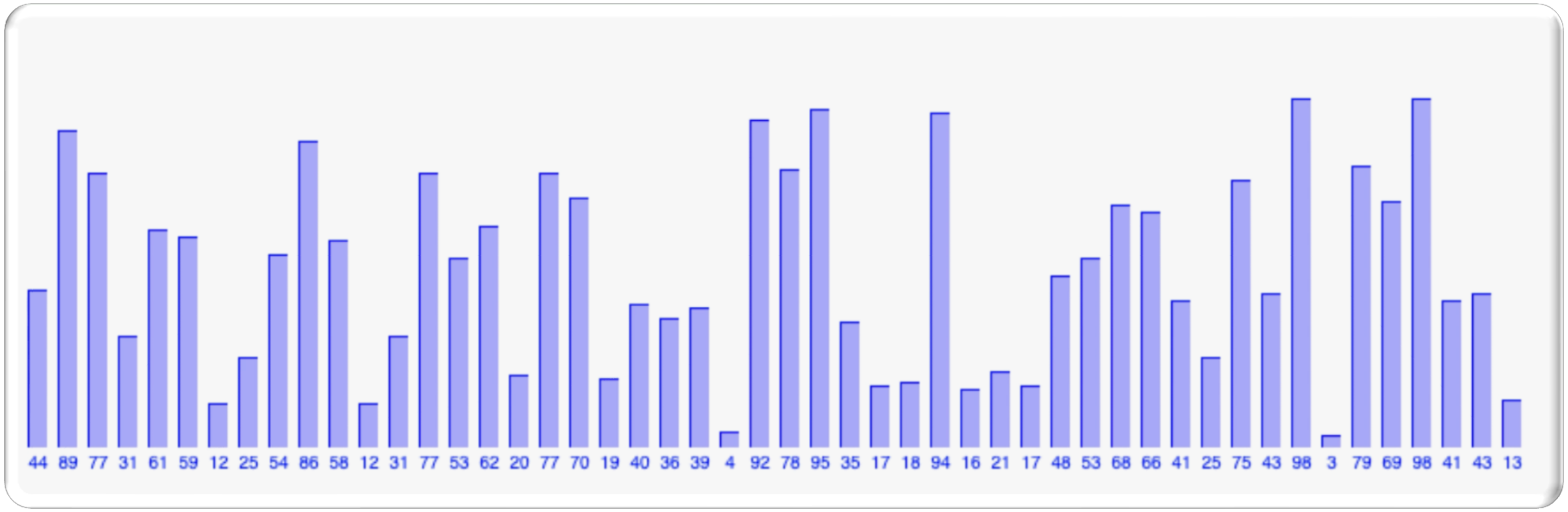
Insertion Sort (version in KR text)

```
Procedure Insertion Sort($a_1$, …, $a_n$)
  for $j$:= 2 to $n$ {$a_1$,…,$a_{j-1}$ are now sorted}
     $m$:= $a_j$        {set aside $a_j$ to insert later}
     $i$:= 1
     while $a_j > a_i$ do   {find insert position}
          $i$:= $i$+1
     for $k$:=0 to $j-i-1$     {open insert position}
          $a_{j-k}$:=  $a_{j-k-1}$      {by moving elements >$a_j$}
     $a_i$:= $m$            {insert $a_j$ in position $i$}
  endfor {$a_1$,$a_2$, …,$a_n$ are now sorted}
```

Insertion Sort (version in KR text)

```
Procedure Insertion Sort($a_1$, …, $a_n$)
  for $j$:= 2 to $n$ {$a_1$,…,$a_{j-1}$ are now sorted}
    $m$:= $a_j$        {set aside $a_j$ to insert later}
    $i$:= 1
    while $a_j > a_i$ do   {find insert position}
        $i$:= $i$+1
    for $k$:=0 to $j-i-1$     {open insert position}
        $a_{j-k}$:=  $a_{j-k-1}$       {by moving elements $>a_j$}
    $a_i$:= $m$           {insert $a_j$ in position $i$}
  endfor {$a_1$, $a_2$, …, $a_n$ are now sorted}
```

Insertion Sort



Slightly different implementation of Insertion Sort:

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Insertion Sort

```
Procedure Insertion Sort($a_1$, …, $a_n$)
   for $j$:= 2 to $n$
      $m$:= $a_j$
            $i$:= 1
      while $a_j > a_i$ do
            $i$:= $i$+1
      for $k$:=0 to $j$-$i$-1
            $a_{j-k}$:=  $a_{j-k-1}$
      $a_i$:= $m$
   endfor
```

How many comparisons are done in the worst case?

Insertion Sort: [4, 3, 2, 1]

```
Procedure Insertion Sort($a_1$, …, $a_n$)
  for $j$:= 2 to $n$
    $m$:= $a_j$
        $i$:= 1
    while $a_j > a_i$ do
        $i$:= $i$+1
    for $k$:=0 to $j$-$i$-1
        $a_{j-k}$:=  $a_{j-k-1}$
    $a_i$:= $m$
  endfor
```

How many comparisons are done in the worst case?

Insertion Sort

4, **3**, 2, 1   2 comparisons

3, 4, **2**, 1   3 comparisons

2, 3, 4, **1**   4 comparisons

1, 2, 3, 4

```
Procedure Insertion Sort($a_1$,…,$a_n$)
    for $j$:= 2 to $n$
        $m$:= $a_j$
        $i$:= 1
        while $a_j > a_i$ do
                $i$:= $i$+1
        for $k$:=0 to $j$-$i$-1
                $a_{j-k}$:=  $a_{j-k-1}$
        $a_i$:= $m$
    endfor
```

How many comparisons are done in the worst case?

$$\sum_{j=2}^{n-1} j = \frac{n(n+1)}{2} - 1$$

Growth of Functions and Complexity (KR 3.2, 3.3)

The Growth of Functions

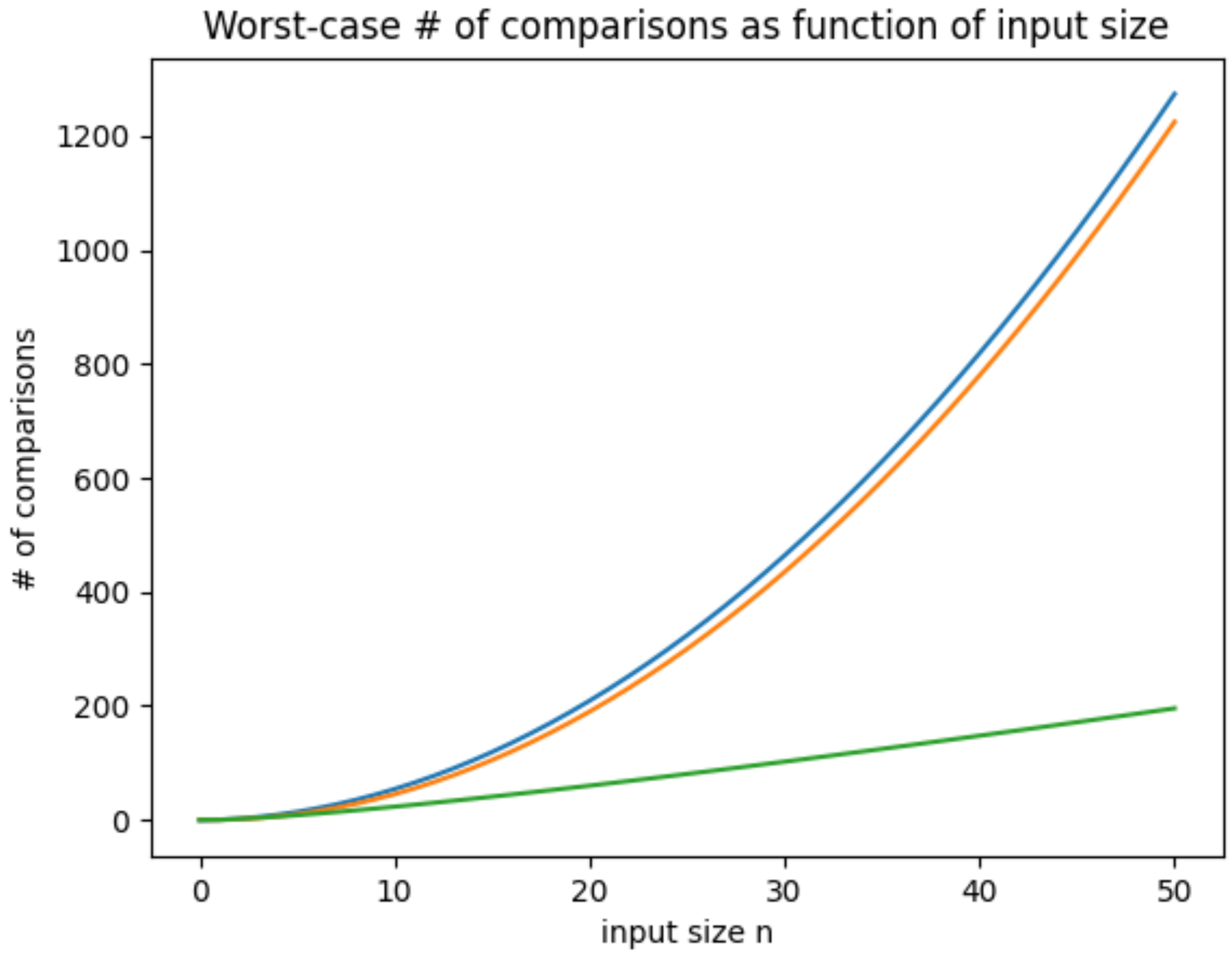
The time required to solve a problem is determined by:

- Number of operations executed
  - Depends on the size of the input
- Speed of the hardware and software
  - Does not depend on the size of the input
  - Can be accounted for by constant multiplier

Big-O notation estimates the growth of functions representing the number of operations executed.

How to Analyze Algorithms?

- Compare execution times? No
- Count the number of statements executed? No
- Express running time as a function of the input size *n* (i.e., *f(n)*)? ***Yes***
    - Analysis is independent of machine speed, programming style, programming language, etc.


Worst-case # of comparisons as function of input size
# of comparisons
input size n
0
200
400
600
800
1000
1200
0
10
20
30
40
50

Complexity of an Algorithm

- The **complexity** of an algorithm refers to the amount of time and space required to execute the algorithm.

- Computing the amount of time and space used without having the actual program requires one to focus on the essential features that affect performance.

Example (revisited):

find the maximum element in a sequence $s_1, s_2, \ldots, s_n$

```
procedure findMax(s, n)
    myMax = $s_1$
    i = 2
    while  i ≤  n  do
        if  $s_i$ > myMax  then myMax =  $s_i$  endif
        i =  i + 1
    endwhile
    return(myMax)
end findMax
```

Analyzing Algorithm findMax

Time of execution depends on the number of iterations of the while loop.

Performance does not generally depend on the values of the elements.

How many iterations are executed? **n – 1**

How much work is done in each iteration? **constant**

The time needed is **linearly proportional to n**.

Example: Given a sequence $s_1, s_2, \ldots, s_n$

```
for i := 1 to n do
        for j:=1 to n do
                $s_i := s_i + s_j$
```

Number of iterations executed: n^2

Total time needed:

- Assignment statement takes constant time
- Time is proportional to n^2

Optional exercise: What does this code actually do?

Big-*O* Notation

- Estimate the growth of a function without worrying about constant multipliers or smaller order terms.
  - Do not need to worry about hardware or software used
  - Big-O notation analyses asymptotic performance independent of an implementation
- Assume that different operations take the same time.
  - Addition is actually faster than division, but for the purposes of analysis we assume they take the same time.
- Used to express worst-case time complexity

Big-O: Definition

Let f and g be functions from $\mathbb{N}$ to $\mathbb{R}$.

We say $f(x)$ is $O\big(g(X)\big)$ if there exist constants C and k such that $|f(x)| \leq C|g(x)|$ for all $x > k$.

Notational Variations

$f(x)$ is $O\big(g(x)\big)$

$f(x) = O\big(g(x)\big)$

$f(x) \in O\big(g(x)\big)$

Example 1

Suppose an algorithm is known to terminate using at most $60n^2 + 5n + 1$ units of time for inputs of size n.

Show that the time complexity of the algorithm is $O(n^2)$.

Note that when $n > 1$:

$60n^2 + 5n + 1 \ \leq \ 60n^2 + 5n^2 + n^2 \ \leq \ 66n^2$

So, choose $C = 66$ and $k = 1$.

Then, $60n^2 + 5n + 1$ is $O(n^2)$.

We say $f(x)$ is $\boldsymbol{O(g(x))}$ if there are constants C and k such that $|f(x)| \ \leq \ C|g(x)|$ for all $x > k$.

Example 2: Show that n^2 is not $O(n)$.

Assume n^2 is $O(n)$.

Then $\exists C \; \exists k \; \forall n > k \quad n^2 \leq Cn$

Need $C \geq n$

But no constant is bigger than every n.

Hence, we have a contradiction.

We say $f(x)$ is $\boldsymbol{O(g(x))}$ if there are constants C and k such that $|f(x)| \leq C|g(x)|$ for all $x > k$.

Example 3: Show that $10n^3 - 5n^2 + 2560$ is $O(n^3)$.

For sufficiently large n,

$10n^3 - 5n^2 + 2560 \leq 10n^3 + 2560 \leq 10n^3 + n^3 \leq 11n^3$.

How large must n be?

$$n^3 \geq 2560$$

$$n \geq 14 > \sqrt[3]{2560}$$

If we choose $C = 11$ and $k = 14$, we have that

$|10n^3 - 5n^2 + 2560| \leq C|n^3|$ for all integers $n > k$.

In other words, $10n^3 - 5n^2 + 2560$ is $O(n^3)$.

We say $f(x)$ is $\boldsymbol{O(g(x))}$ if there are constants C and k such that $|f(x)| \leq C|g(x)|$ for all $x > k$.

Example 3: Show that $10n^3 - 5n^2 + 2560$ is $O(n^3)$.

For sufficiently large n,

$10n^3 - 5n^2 + 2560 \leq 10n^3 + 2560 \leq 10n^3 + 5n^3 \leq 15n^3$.

How large must n be?

$5n^3 \geq 2560$

$n^3 \geq 512$

$n \geq 8$

If we choose $C = 15$ and $k = 8$, we have that

$|10n^3 - 5n^2 + 2560| \leq C|n^3|$ for all integers $n > k$.

In other words, $10n^3 - 5n^2 + 2560$ is $O(n^3)$.

We say $f(x)$ is $\boldsymbol{O(g(x))}$ if there are constants C and k such that $|f(x)| \leq C|g(x)|$ for all $x > k$.

Example 3: Show that $10n^3 - 5n^2 + 2560$ is $O(n^3)$.

For sufficiently large n,

$10n^3 - 5n^2 + 2560 \leq 10n^3 + 2560 \leq 10n^3 + 40n^3 \leq 50n^3$.

How large must n be?

$$40n^3 \geq 2560$$

$$n^3 \geq 64$$

$$n \geq 4$$

If we choose $C = 50$ and $k = 4$, we have that

$|10n^3 - 5n^2 + 2560| \leq C|n^3|$ for all integers $n > k$.

In other words, $10n^3 - 5n^2 + 2560$ is $O(n^3)$.

We say $f(x)$ is $\boldsymbol{O(g(x))}$ if there are constants C and k such that $|f(x)| \leq C|g(x)|$ for all $x > k$.

Theorem: Let $a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$ be a polynomial in n of degree d. Then, $a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$ is $O(n^d)$.

Example: $2n^8 + 55n^5 - 6n^2 + 4856 = O(n^8)$

Claim: $a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$ is $O(n^d)$.

Proof: Let a be the ceiling of the largest $|a_i|$, let $C = a(d+1)$, and let $k = 1$. Then for $n > k$:

$a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$

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

$\leq \sum_{i=0}^{d} a n^i$

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

$\leq a \sum_{i=0}^{d} n^d$

$= a(d+1) n^d$

$= C n^d$

Example

```
for i := 1 to n do
    for j := 1 to i do
        x := x + 1
```

How many times is x := x + 1 executed?

```
for i := 1 to n do
    for j := 1 to i do
        x := x + 1
```

How many times is x := x+1 executed?

When i = 1 it is executed once

i = 2 it is executed 2 times

…

i = n it is executed n times

The total number executions is

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

From the previous theorem we have that

$1 + 2 + \cdots + n$ is $O(n^2)$

Example

Give a big-O estimate for the factorial function $f(n) = n!$

1. $n!$ is $O(n)$
2. $n!$ is $O(n^2)$
3. $n!$ is $O(n^n)$

Example

Give a big-O estimate for the factorial function $f(n) = n!$

1. $n!$ is $O(n)$
2. $n!$ is $O(n^2)$
3. **$\boldsymbol{n!}$ is $\boldsymbol{O(n^n)}$**

Solution

$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \leq n \cdot n \cdot n \cdot \ldots \cdot n = n^n$

$n!$ is $O(n^n)$

Example

```
i = n
while  i ≥ 1  do
    x = x + 1
    i = ⌊ i / 2 ⌋
endwhile
```

How many times is x := x + 1 executed?

If $n = 2^k$, it is executed k times.

If $2^{k-1} < n \leq 2^k$, it is executed k times.

number of times x = x + 1 is executed is proportional to k, which is O(log n)

Note: O(n) is not incorrect, but an overestimation.

We want the tightest bounds!

Example

Let $f(n) = 6n^2 + 5n\log(n)$

Which statement(s) are true?

1. f is $O(n)$
2. f is $O(n^2)$
3. f is $O(n^3)$

Example

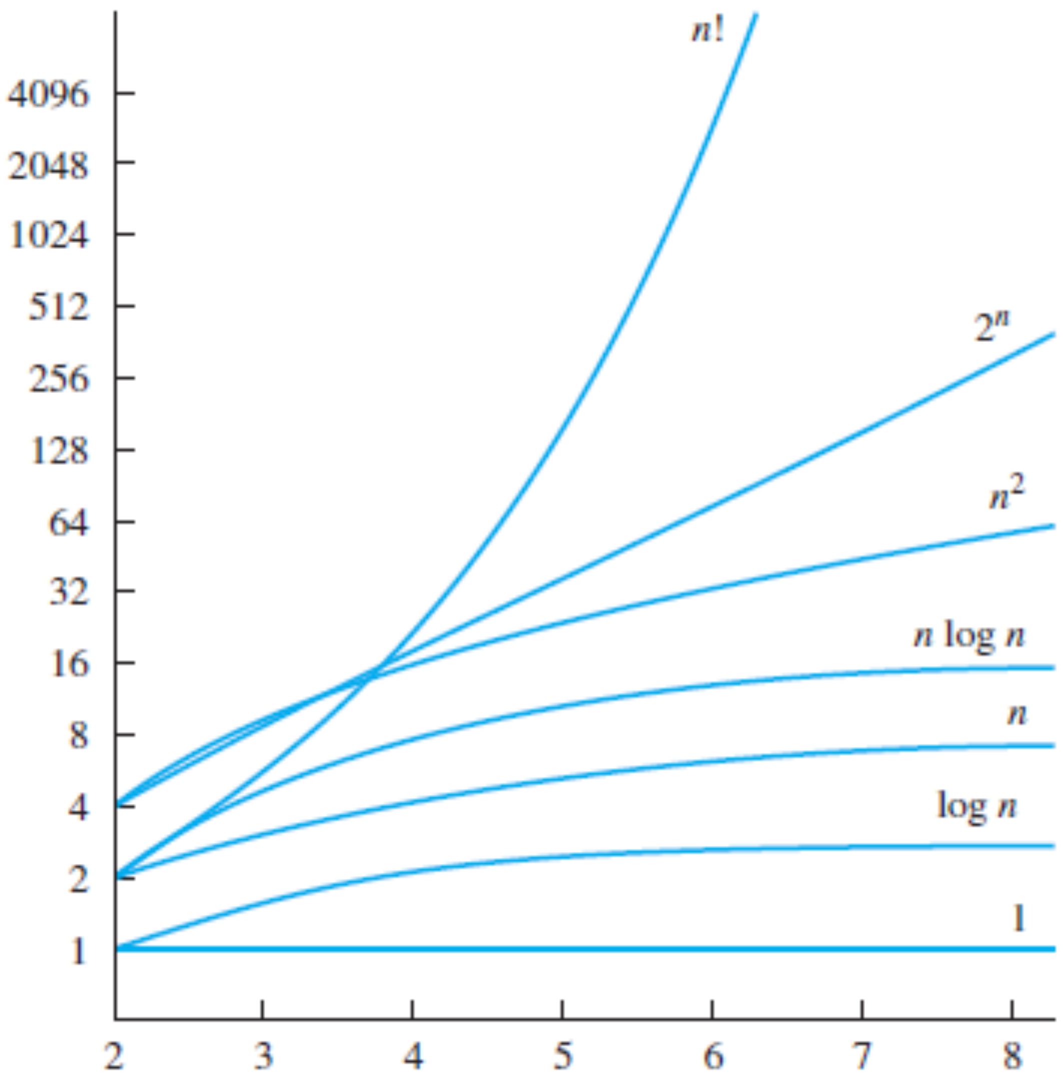
Let $f(n) = 6n^2 + 5n\log(n)$

Which statement(s) are true?

1. f is $O(n)$	False
2. f is $O(n^2)$	**True**
3. f is $O(n^3)$	**True** but not tight!

Note: To show that (2) and (3) are true,
choose $k = 1$ and $C = 11$.

Common Growth Functions



Factorial O(n!)

Exponential O(c^n)

Polynomial O(n^2)

Linearithmic O(n log n)

Linear O(n)

Logarithmic O(log n)

Constant O(1)

Big-Omega

[Definition] Let f and g be functions from $\mathbb{N}$ to $\mathbb{R}$.

$\boldsymbol{f(x)}$ **is** $\boldsymbol{\Omega\big(g(x)\big)}$ if there exist constants $C > 0$ and k such that $|\boldsymbol{f(x)}| \geq \boldsymbol{C}|\boldsymbol{g(x)}|$ for all integers $x > k$.

Recall that $\boldsymbol{f(x)}$ is $\boldsymbol{O(g(x))}$ if there exist constants C and k such that $|\boldsymbol{f(x)}| \leq \boldsymbol{C}|\boldsymbol{g(x)}|$ for all $x > k$.

Big-Omega

[Definition] Let f and g be functions from $\mathbb{N}$ to $\mathbb{R}$.

$\boldsymbol{f(x)}$ **is** $\boldsymbol{\Omega\big(g(x)\big)}$ if there exist constants $C > 0$ and k such that $|\boldsymbol{f(x)}| \geq \boldsymbol{C}|\boldsymbol{g(x)}|$ for all integers $x > k$.

[Big-O & Big-Omega Connection]

$f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $O\big(f(x)\big)$.

Example

An algorithm terminates using $\boldsymbol{6n^2 + 5n\log(n)}$ units of time for inputs of size $\boldsymbol{n}$. Show that the time complexity of the algorithm is $\boldsymbol{\Omega(n^2)}$.

Recall that $\boldsymbol{f(x)}$ is $\boldsymbol{\Omega(g(x))}$ if there exist constants $C > 0$ and k such that $|\boldsymbol{f(x)}| \geq \boldsymbol{C}|\boldsymbol{g(x)}|$ for all integers $x > k$.

Example

Recall that $\boldsymbol{f(x)}$ is $\boldsymbol{\Omega\big(g(x)\big)}$ if there exist constants $C > 0$ and k such that $|\boldsymbol{f(x)}| \geq \boldsymbol{C}|\boldsymbol{g(x)}|$ for all integers $x > k$.

Recall that $\log(1) = 0$ and that $\log(n)$ is an increasing function. Thus, $\log(n) \geq 0$ for all $n > 1$.

Therefore, when $n > 1$, we have:

$6n^2 + 5n\log(n) \geq 6n^2$

So, choose $C = 6$ and $k = 1$.

Big-Theta

[Definition] Let f and g be functions from $\mathbb{N}$ to $\mathbb{R}$.
$f(x)$ is $\boldsymbol{\Theta}\big(g(x)\big)$ if $f(x)$ is $O\big(g(x)\big)$ and $f(x)$ is $\Omega\big(g(x)\big)$.

When $f(x)$ is $\Theta\big(g(x)\big)$, we say that

- $f(x)$ is of order $g(x)$
- $f(x)$ and $g(x)$ are of the same order
- $g(x)$ is also $\Theta\big(f(x)\big)$

Example

An algorithm terminates using $\boldsymbol{6n^2 + 5n\log(n)}$ units of time for inputs of size $\boldsymbol{n}$. Show that the time complexity of the algorithm is $\Theta(\boldsymbol{n^2})$.

Recall that $\boldsymbol{f(x)}$ is $\boldsymbol{\Theta(g(x))}$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$.

Example

$6n^2 + 5n\log(n) \leq 11n^2$ for all $n > 1$, so
$6n^2 + 5n\log(n)$ is $O(n^2)$.

$6n^2 + 5n\log(n) \geq 6n^2$ for all $n > 1$, so
$6n^2 + 5n\log(n)$ is $\Omega(n^2)$.

Therefore, $6n^2 + 5n\log(n)$ is $\Theta(n^2)$.

Example

```
i = 0
x = 1
while  x < n  do
    i :=  i + 1
    x :=  4 x
```

$$O(\log_4 n) = O(\log_2 n)$$

$$\log_4 n = \frac{\log_2 n}{\log_2 4} = \frac{\log_2 n}{2}$$

What are the values of i and x when the loop terminates?

i = 0 1 2 3 4 5 ...

x = 1 4 16 4^3 4^4 4^5 . . .

When is $4^i \geq n$? When $i \geq \log_4 n$

Common growth functions

• constant	$O(1)$	
• logarithmic	$O(\log n)$	
• linear	$O(n)$	
• linearithmic	$O(n \log n)$	
• quadratic	$O(n^2)$	
• cubic	$O(n^3)$	
• polynomial	$O(n^k)$,	k is a constant
• exponential	$O(2^n)$,	$O(c^n)$, c is a constant
• factorial	$O(n!)$	

Important big-O relationships

$\log_2 n = O(n)$, reverse not true

$\log_2 n = O(\sqrt{n})$, reverse not true

$\log_b n = O(\log n)$ where $b > 1$

$\log_2 n = O(n^{1/8})$, reverse not true

$(\log_2 n)^4 = O(\sqrt{n})$, reverse not true

$n \log n = O(n^2)$, reverse not true

Designing an Efficient Algorithm for the Maximum Subarray Problem

Maximum Subarray

https://leetcode.com/problems/maximum-subarray/

Given n integers stored in array S of size n, find i and j, ($1 \leq i \leq j \leq n$) such that $S[i] + \cdots + S[j]$ is maximized.

Note: In algorithm examples, array indices often start at 1

Example:

$n = 12$

$S = [4, -7, 4, 0, 5, -8, 13, -2, 4, 6, -6, 2]$

Maximum sum of 22 for $i = 3$ and $j = 10$.

Note: We will solve the slightly easier problem of finding the maximum sum. We will only return the sum and not the indices.

Maximum Subarray

https://leetcode.com/problems/maximum-subarray/

Given n integers stored in array S of size n, find i and j, $(1 \le i \le j \le n)$ such that $S[i] + \cdots + S[j]$ is maximized.

Note: In algorithm examples, array indices often start at 1

Compute maximum subarray sum:

1	2	3	4	5
-10	3	2	-1	5

Maximum Subarray

https://leetcode.com/problems/maximum-subarray/

Given n integers stored in array S of size n, find i and j, $(1 \leq i \leq j \leq n)$ such that $S[i] + \cdots + S[j]$ is maximized.

Note: In algorithm examples, array indices often start at 1

Compute maximum subarray sum:

1	2	3	4	5
-10	3	2	-1	5

i = 2 and j = 5 gives maximum sum of 9

Solution 1: Find all possible solutions

```
procedure MaxSum1(S,n)
  max := 0
  for j := 1 to n do
     for i := 1 to j do
          sum := 0
          for k := i to j do  sum := sum + S(k)
          if sum > max then max := sum
  return(max)
```

Solution 1: Find all possible solutions

```
procedure MaxSum1(S,n)
  max := 0
  for j := 1 to n do
    for i := 1 to j do
      sum := 0
      for k := i to j do sum := sum + S(k)
      if sum > max then max := sum
  return(max)
```

3 nested for-loops

Innermost loop adds terms for the sum from S(i) to S(j).

$O(n^3)$ time.

Can we do better?

Solution 2: Don’t re-compute

```
procedure MaxSum2(S,n)
  max := 0
  for j := 1 to n do
      for i := 1 to j −1 do
              sum(i,j) := sum(i,j−1) + S(j)
              if sum(i,j) > max then max := sum(i,j)
      sum(j,j) := S(j)
      if sum(j,j) > max then max := sum(j,j)
  return(max)
```

Solution 2: Don’t re-compute

```
procedure MaxSum2(S,n)
  max := 0
  for j := 1 to n do
    for i := 1 to j −1 do
        sum(i,j) := sum(i,j−1) + S(j)
        if sum(i,j) > max then max := sum(i,j)
    sum(j,j) := S(j)
    if sum(j,j) > max then max := sum(j,j)
  return(max)
```

Solution 2: Don’t re-compute

```
procedure MaxSum2(S,n)
  max := 0
  for j := 1 to n do
      for i := 1 to j −1 do
          sum(i,j) := sum(i,j−1) + S(j)
          if sum(i,j) > max then max := sum(i,j)
      sum(j,j) := S(j)
      if sum(j,j) > max then max := sum(j,j)
  return(max)
```

Solution 2: Don’t re-compute

```
procedure MaxSum2(S,n)
  max := 0
  for j := 1 to n do
      for i := 1 to j −1 do
              sum(i,j) := sum(i,j−1) + S(j)
              if sum(i,j) > max then max := sum(i,j)
      sum(j,j) := S(j)
      if sum(j,j) > max then max := sum(j,j)
  return(max)
```

Solution 2: Don’t re-compute

```
procedure MaxSum2(S,n)
  max := 0
  for j := 1 to n do
      for i := 1 to j −1 do
              sum(i,j) := sum(i,j−1) + S(j)
              if sum(i,j) > max then max := sum(i,j)
      sum(j,j) := S(j)
      if sum(j,j) > max then max := sum(j,j)
  return(max)
```

Solution 2: Don’t re-compute

```
procedure MaxSum2(S,n)
  max := 0
  for j := 1 to n do
     for i := 1 to j −1 do
           sum(i,j) := sum(i,j−1) + S(j)
           if sum(i,j) > max then max := sum(i,j)
     sum(j,j) := S(j)
     if sum(j,j) > max then max := sum(j,j)
  return(max)
```

2 nested for-loops

$O(n^2)$ time and $O(n^2)$ space.

Can we do better?

Solution 3: Fast updating

```
procedure MaxSum3(S,n)
  max := S(1);
  sum := S(1);
  for i := 2 to n do
        if sum > 0 then sum := sum + S(i)
          else sum := S(i)
        if sum > max then max:= sum
  return(max)
```

$O(n)$ time algorithm (one for-loop).

Dynamic Programming solution.

Correctness is not obvious.

CS 381 covers this in detail.

Problem-Solving Tips

- Fully understand the problem.
- Understand how to solve it “by hand.”
- Develop a first, possibly straightforward algorithm.
- Take a close and critical look at your solution and ask: “How could it be improved?”
- Solutions that repeat or duplicate computations are often not efficient and can be improved.

Recommended Exercises

Implement the algorithms discussed in class using the programming language of your choice

KR 3.1: 1, 9, 13, 37

Recommended Exercises:

KR 3.2

- 1, 3, 5, 7, 9
- 11, 13, 15, 17, 19
- 21, 23, 25, 27, 29,
- 31, 33, 35, 37, 39,