

CS141: Intermediate Data Structures and Algorithms

Analysis of Algorithms

Amr Magdy

Analyzing Algorithms

1. Algorithm Correctness

- a. Termination
- b. Produces the correct output for all possible input.

2. Algorithm Performance

- a. Either runtime analysis,
- b. or storage (memory) space analysis
- c. or both

Algorithm Correctness



- › **Sorting problem**
 - › Input: an array A of n numbers
 - › Output: the same array in ascending sorted order (smallest number in $A[1]$ and largest in $A[n]$)

Algorithm Correctness



- ▶ Sorting problem
 - ▶ Input: an array A of n numbers
 - ▶ Output: the same array in ascending sorted order (smallest number in $A[1]$ and largest in $A[n]$)
- ▶ Insertion Sort

INSERTION-SORT(A, n)

for $j = 2$ **to** n

$key = A[j]$

 // Insert $A[j]$ into the sorted sequence $A[1 .. j - 1]$.

$i = j - 1$

while $i > 0$ and $A[i] > key$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = key$

Algorithm Correctness

- › How does insertion sort work?



Algorithm Correctness

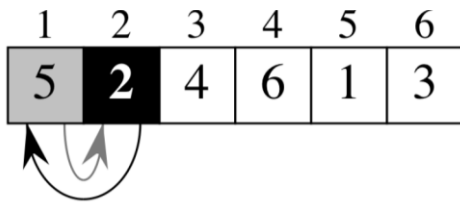


5	2	4	6	1	3
---	---	---	---	---	---

Algorithm Correctness

5	2	4	6	1	3
---	---	---	---	---	---


1	2	3	4	5	6
5	2	4	6	1	3

A diagram illustrating a swap operation in an array. The array is shown as a row of six cells containing the values 5, 2, 4, 6, 1, and 3. Above each cell is its index from 1 to 6. The cell containing '5' at index 1 is shaded light gray, and the cell containing '2' at index 2 is shaded black. Two curved arrows originate from the bottom of these cells: one arrow points from the '5' cell to the '2' cell, and the other points from the '2' cell to the '5' cell, indicating the exchange of their positions.


Algorithm Correctness

5	2	4	6	1	3
---	---	---	---	---	---

1	2	3	4	5	6
5	2	4	6	1	3




1	2	3	4	5	6
2	5	4	6	1	3




Algorithm Correctness

5	2	4	6	1	3
---	---	---	---	---	---

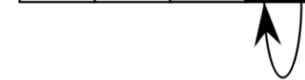
1	2	3	4	5	6
5	2	4	6	1	3



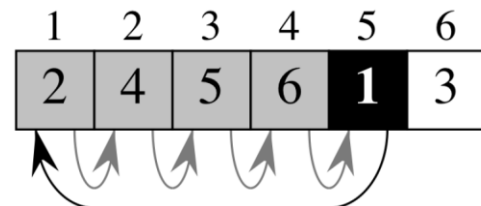
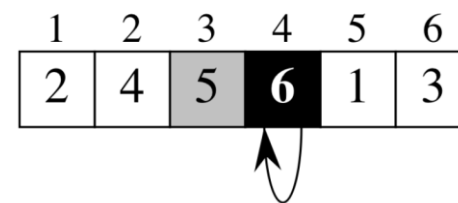
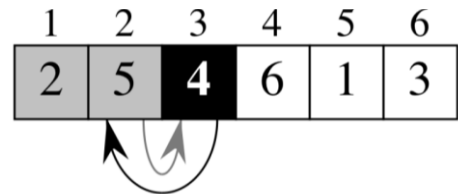
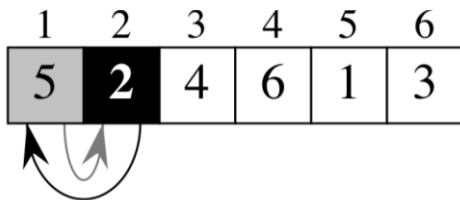
1	2	3	4	5	6
2	5	4	6	1	3



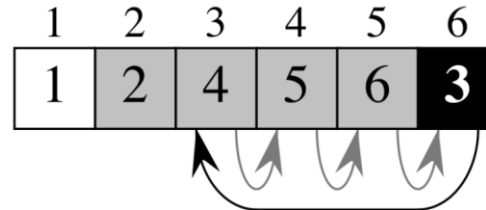
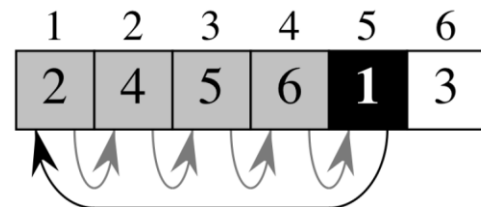
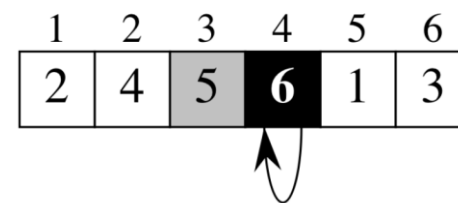
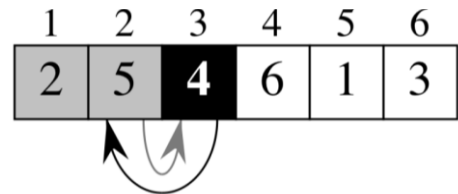
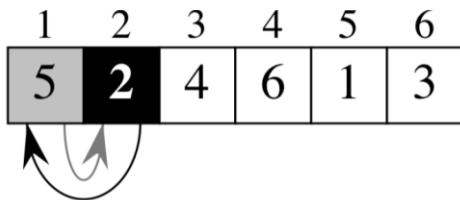
1	2	3	4	5	6
2	4	5	6	1	3



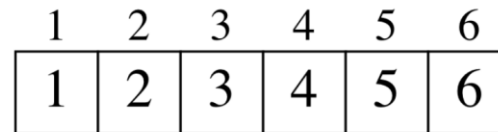
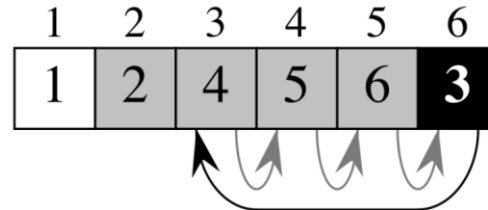
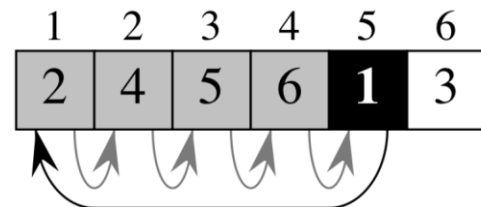
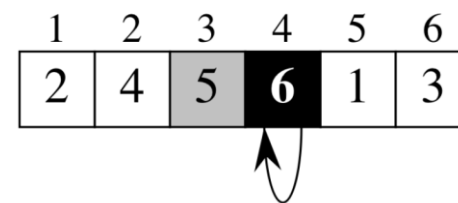
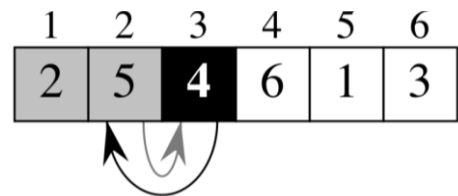
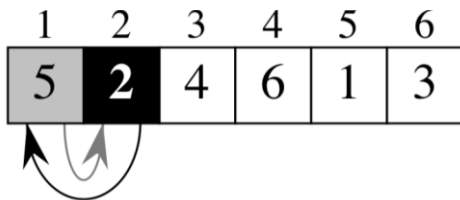
Algorithm Correctness



Algorithm Correctness



Algorithm Correctness



Algorithm Correctness



- › Is insertion sort a correct algorithm?

Algorithm Correctness

- › Is insertion sort a correct algorithm?
 - › Does it halt?
 - › Does it produce correct output for all possible input?

Algorithm Correctness

- › Is insertion sort a correct algorithm?
 - › Does it halt? Yes
 - › Two deterministically bounded loops, no infinite loops involved
 - › Does it produce correct output for all possible input?

INSERTION-SORT(A, n)

for $j = 2$ **to** n

$key = A[j]$

 // Insert $A[j]$ into the sorted sequence $A[1 .. j - 1]$.

$i = j - 1$

while $i > 0$ and $A[i] > key$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = key$

Algorithm Correctness

- › Is insertion sort a correct algorithm?
 - › Does it halt? Yes
 - › Does it produce correct output for all possible input?
 - › Will check through loop invariants

Algorithm Correctness

- › Is insertion sort a correct algorithm?
- › Loop invariant:
 - › It is a property that is true before and after each loop iteration.

Algorithm Correctness

- › Is insertion sort a correct algorithm?
- › Loop invariant:
 - › It is a property that is true before and after each loop iteration.
- › Insertion sort loop invariant (ISLI):
 - › The first $(j-1)$ array elements $A[1..j-1]$ are:
(a) the original $(j-1)$ elements, and (b) sorted.

INSERTION-SORT(A, n)

for $j = 2$ **to** n

$key = A[j]$

 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.

$i = j - 1$

while $i > 0$ and $A[i] > key$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = key$

Algorithm Correctness

- ▶ Is insertion sort a correct algorithm?
 - ▶ If ISLI correct, then insertion sort is correct
 - ▶ How?
 - ▶ Halts and produces the correct output after $(n-1)$ iterations

Algorithm Correctness

- › Is insertion sort a correct algorithm?
 - › If ISLI correct, then insertion sort is correct
 - › How?
 - › Halts and produces the correct output after $(n-1)$ iterations
- › Loop invariant (LI) correctness
 1. Initialization:
 - LI is true prior to the 1st iteration.
 2. Maintenance:
 - If LI true before the iteration, it remains true before the next iteration
 3. Termination:
 - After the loop terminates, the output is correct.

Algorithm Correctness

- ISLI: The first $(j-1)$ array elements $A[1..j-1]$ are:
(a) the original $(j-1)$ elements, and (b) sorted.

1. Initialization:

Prior to the 1st iteration, $j=2$, the first $(2-1)=1$ elements is sorted.

2. Maintenance:

The $(j-1)$ th iteration inserts the j th element in a sorted order, so after the iteration, the first $(j-1)$ elements remains the same and sorted.

3. Termination:

The loop terminates after $(n-1)$ iterations, $j=n+1$, so the first n elements are sorted, then the output is correct.

INSERTION-SORT(A, n)

for $j = 2$ **to** n

$key = A[j]$

 // Insert $A[j]$ into the sorted sequence $A[1 .. j - 1]$.

$i = j - 1$

while $i > 0$ and $A[i] > key$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = key$

Algorithm Correctness

- ISLI: The first $(j-1)$ array elements $A[1..j-1]$ are:
 - the original $(j-1)$ elements, and
 - sorted.

- Initialization:

Prior to the 1st iteration, $j=2$, the first $(2-1)=1$ elements is sorted.

- Maintenance:

The $(j-1)$ th iteration inserts the j th element in a sorted order, so after the iteration, the first $(j-1)$ elements remains the same and sorted.

- Termination:

The loop terminates after $(n-1)$ iterations, $j=n+1$, so the first n elements are sorted, then the output is correct.

INSERTION-SORT(A, n)

for $j = 2$ **to** n

$key = A[j]$

 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.

$i = j - 1$

while $i > 0$ and $A[i] > key$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = key$

Correct

Analyzing Algorithms

1. Algorithm Correctness

- a. Termination
- b. Produces the correct output for all possible input.



2. Algorithm Performance

- a. Either runtime analysis,
- b. or storage (memory) space analysis
- c. or both

Algorithms Performance Analysis



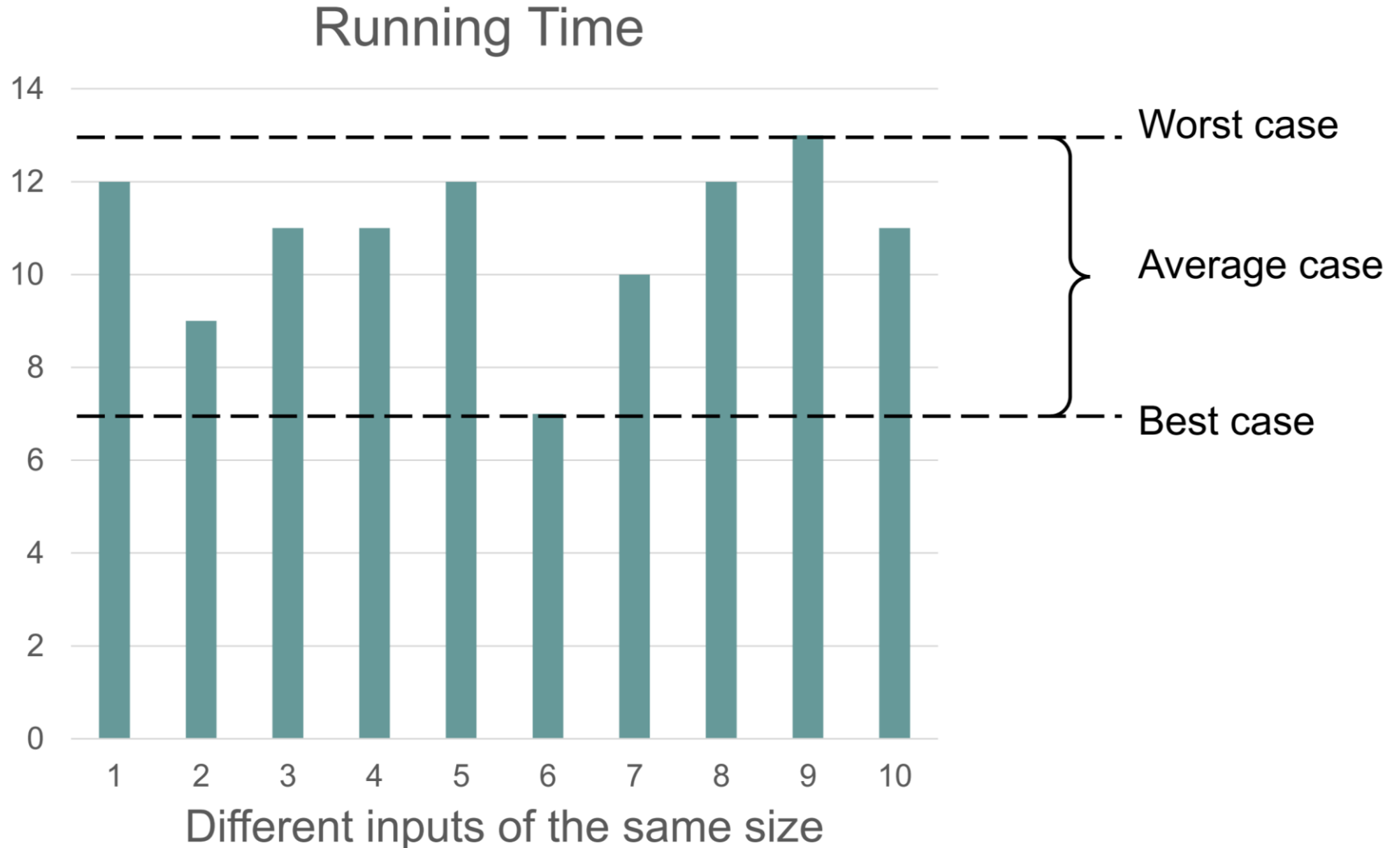
- › Which criteria should be taken into account?
- › Running time
- › Memory footprint
- › Disk IO
- › Network bandwidth
- › Power consumption
- › Lines of codes
- › ...

Algorithms Performance Analysis

› Which criteria should be taken into account?

- › Running time
- › Memory footprint
- › Disk IO
- › Network bandwidth
- › Power consumption
- › Lines of codes
- › ...

Average Case vs. Worst Case

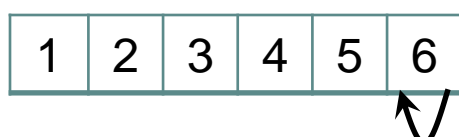
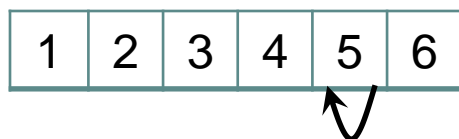
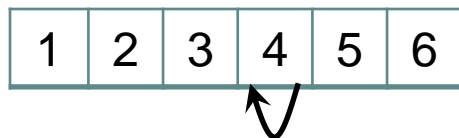
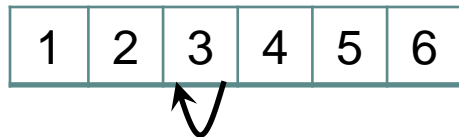
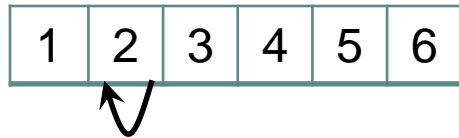


Insertion Sort Best Case



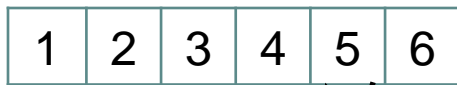
Insertion Sort Best Case

- › Input array is sorted



Insertion Sort Best Case

- Input array is sorted



INSERTION-SORT(A, n)

```

for  $j = 2$  to  $n$  .....c1
     $key = A[j]$  .....c2
    // Insert  $A[j]$  into the sorted sequence  $A[1 .. j - 1]$  .....0
     $i = j - 1$  .....c3
    while  $i > 0$  and  $A[i] > key$  .....c4
         $A[i + 1] = A[i]$ 
         $i = i - 1$ 
        } do not execute ..... 0 } 1
     $A[i + 1] = key$  .....c5
    
```

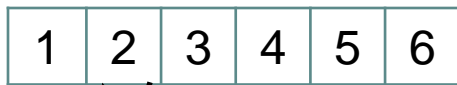
(n-1)

Insertion Sort Best Case

- Input array is sorted



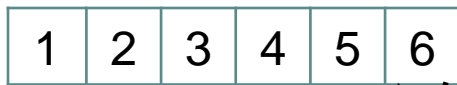
INSERTION-SORT(A, n)



```

for j = 2 to n .....c1
  key = A[j] .....c2
  // Insert A[j] into the sorted sequence A[1 .. j - 1] .....0
  i = j - 1 .....c3
  while i > 0 and A[i] > key .....c4
    A[i + 1] = A[i] } do not execute ..... 0 } 1
    i = i - 1
  A[i + 1] = key .....c5
    
```

(n-1)



$$T(n) = (n-1) * (c1+c2+0+c3+1*(c4+0)+c5)$$

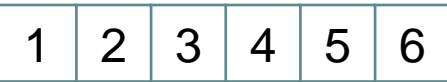
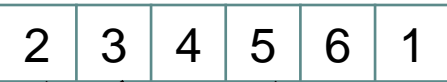
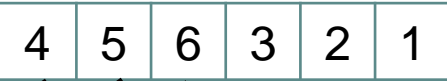
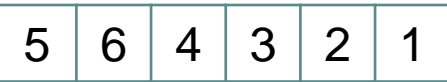
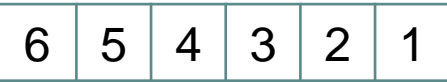
$$T(n) = cn-c, \quad \text{const } c=c1+c2+c3+c4+c5$$

Insertion Sort Worst Case



Insertion Sort Worst Case

- › Input array is reversed



Insertion Sort Worst Case

- Input array is reversed

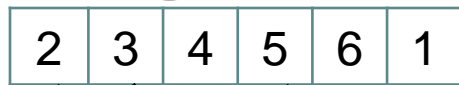
INSERTION-SORT(A, n)

```

for  $j = 2$  to  $n$  .....c1
     $key = A[j]$  .....c2
    // Insert  $A[j]$  into the sorted sequence  $A[1..j - 1]$  .....0
     $i = j - 1$  .....c3
    while  $i > 0$  and  $A[i] > key$  .....c4
         $A[i + 1] = A[i]$  .....c5
         $i = i - 1$  .....c6
     $A[i + 1] = key$  .....c7
    
```

(n-1)

i

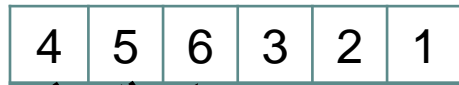


Insertion Sort Worst Case

- Input array is reversed



INSERTION-SORT(A, n)



```

for j = 2 to n .....c1
  key = A[j] .....c2
  // Insert A[j] into the sorted sequence A[1 .. j - 1] .....0
  i = j - 1 .....c3
  while i > 0 and A[i] > key .....c4
    A[i + 1] = A[i] .....c5
    i = i - 1 .....c6
  A[i + 1] = key .....c7
    
```

(n-1) }
i }

$$T(n) = (n-1) * (c1+c2+0+c3+i*(c4+c5+c6))+c7$$

$$T(n) = (n-1)*(c1+c2+0+c3+c7) + \sum i*(c4+c5+c6), \text{ for all } 1 \leq i < n$$

$$T(n) = (cn-c) + \sum i*d, \text{ c \& d are constants}$$

$$\sum i*d = 1*d+2*d+3*d+\dots+(n-1)*d = d*(1+2+3+\dots+(n-1)) = d*n(n-1)/2$$

$$T(n) = (cn-c) + dn^2/2-dn/2 = d*n^2+c11*n+c12, \text{ c's \& d are consts}$$

Insertion Sort Average Case

- › Average = (Best + Worst)/2
- › $T(n) = cn^2 + dn + e$, c, d, e are consts

Which case we consider?



Which case we consider?



- › The worst case

Which case we consider?



- ▶ The worst case
 - ▶ Why?

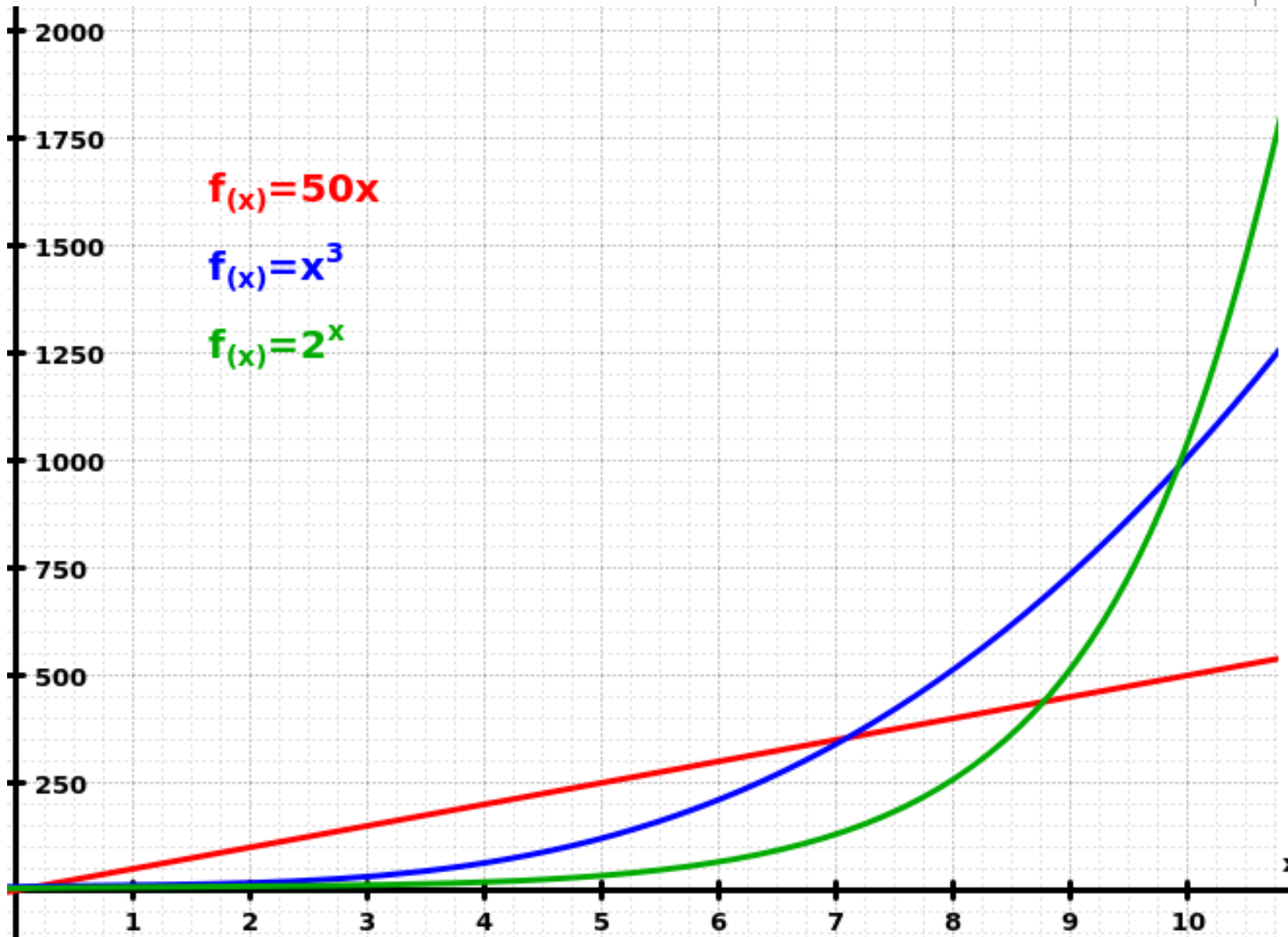
Which case we consider?

- ▶ The worst case
 - ▶ Why?
 - ▶ It gives guarantees on the upper bound performance

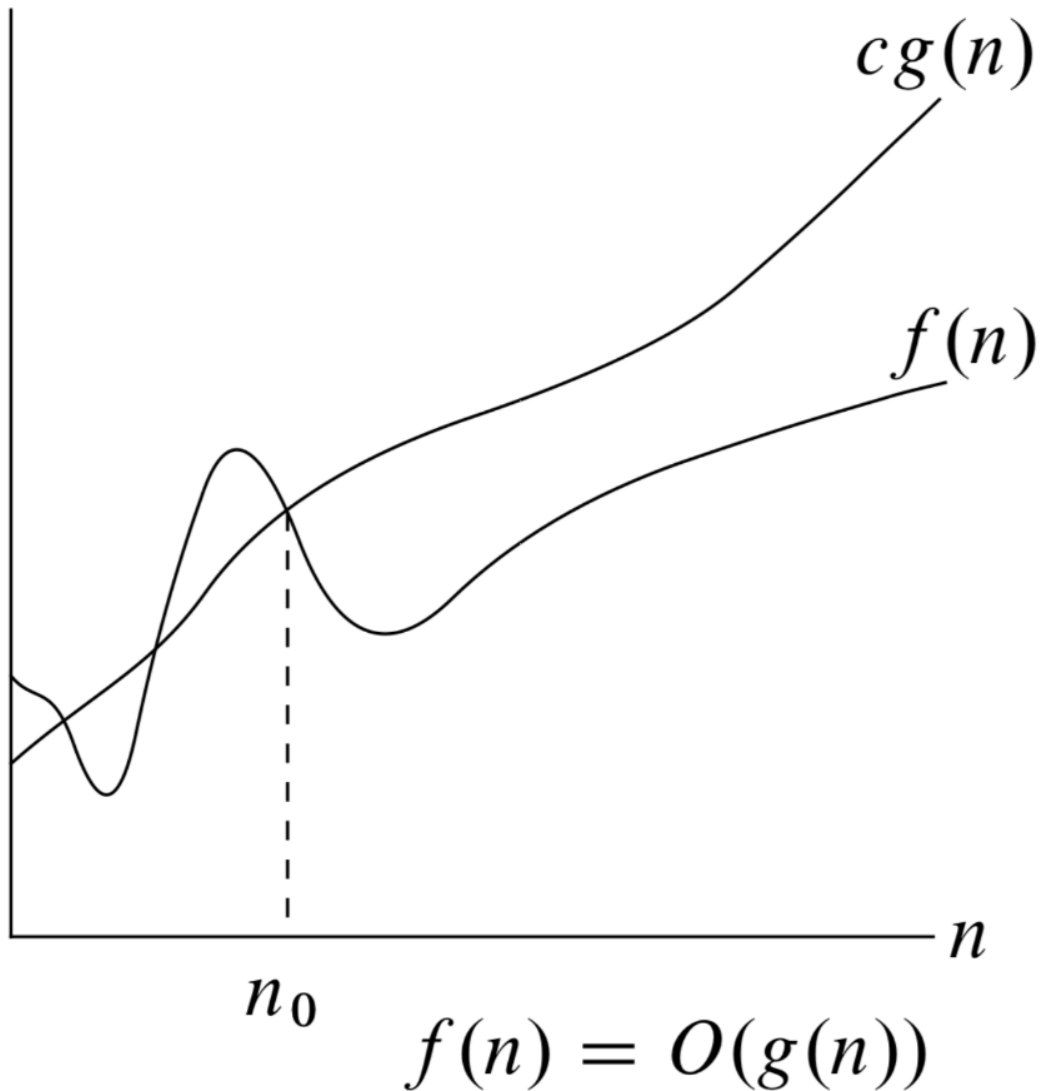
Growth of Functions

- › It is hard to compute the actual running time for more complex algorithms
- › The cost of the worst-case is a good measure
- › The growth of the cost function is what interests us (when input size is large)
- › We are more concerned with comparing two cost functions, i.e., two algorithms.

Growth of Functions



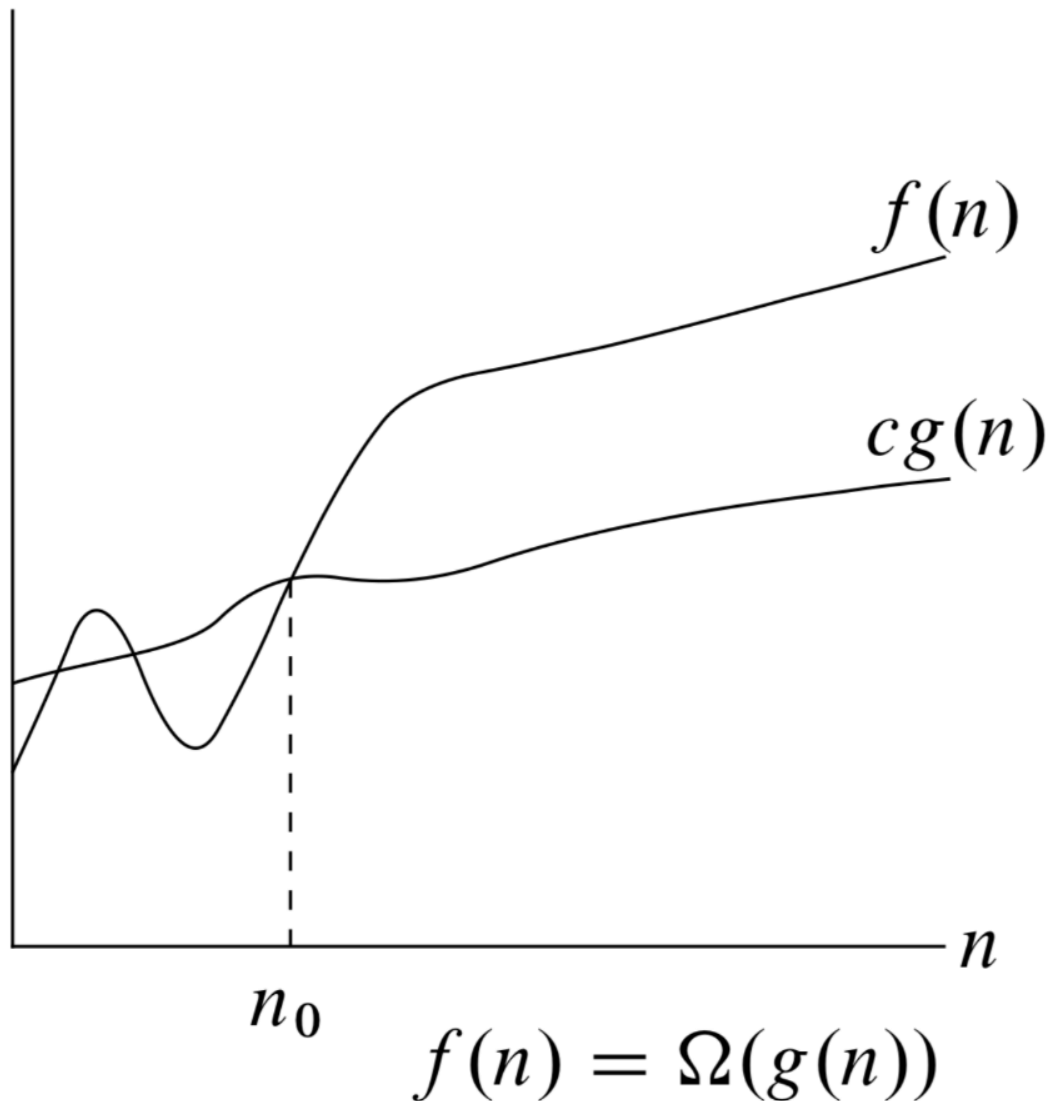
O-notation



$$\begin{aligned} \exists c > 0, n_0 > 0 \\ 0 \leq f(n) \leq cg(n) \\ n \geq n_0 \end{aligned}$$

$g(n)$ is an asymptotic **upper-bound** for $f(n)$

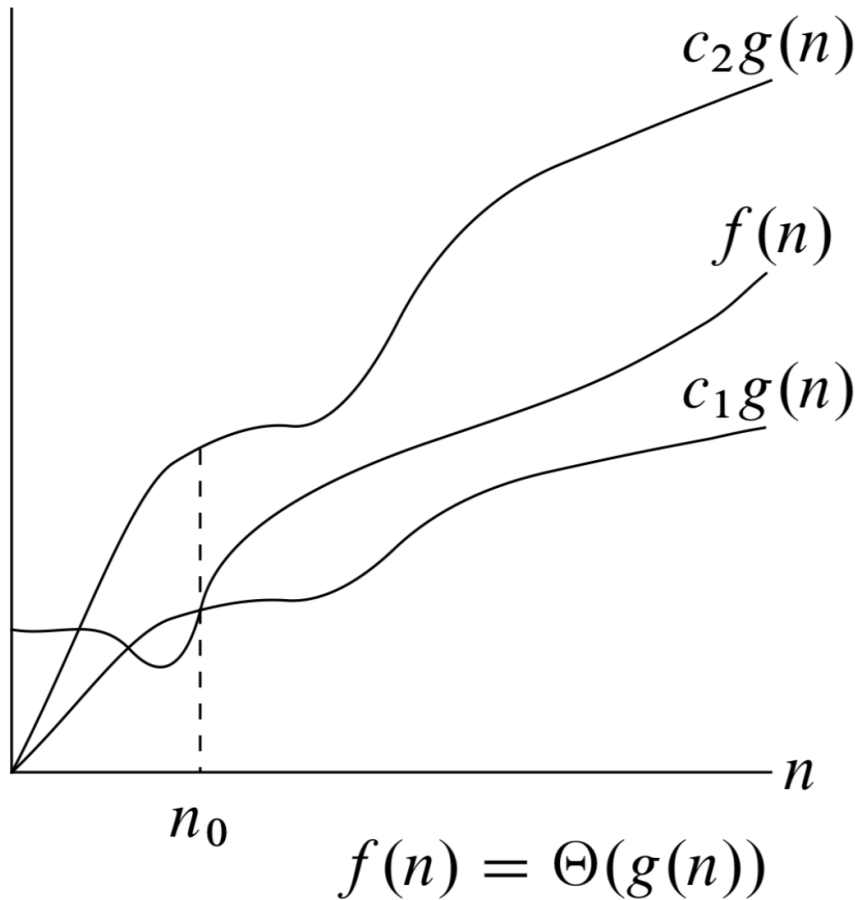
Ω -notation



$$\begin{aligned} \exists c > 0, n_0 > 0 \\ 0 \leq cg(n) \leq f(n) \\ n \geq n_0 \end{aligned}$$

$g(n)$ is an asymptotic **lower**-bound for $f(n)$

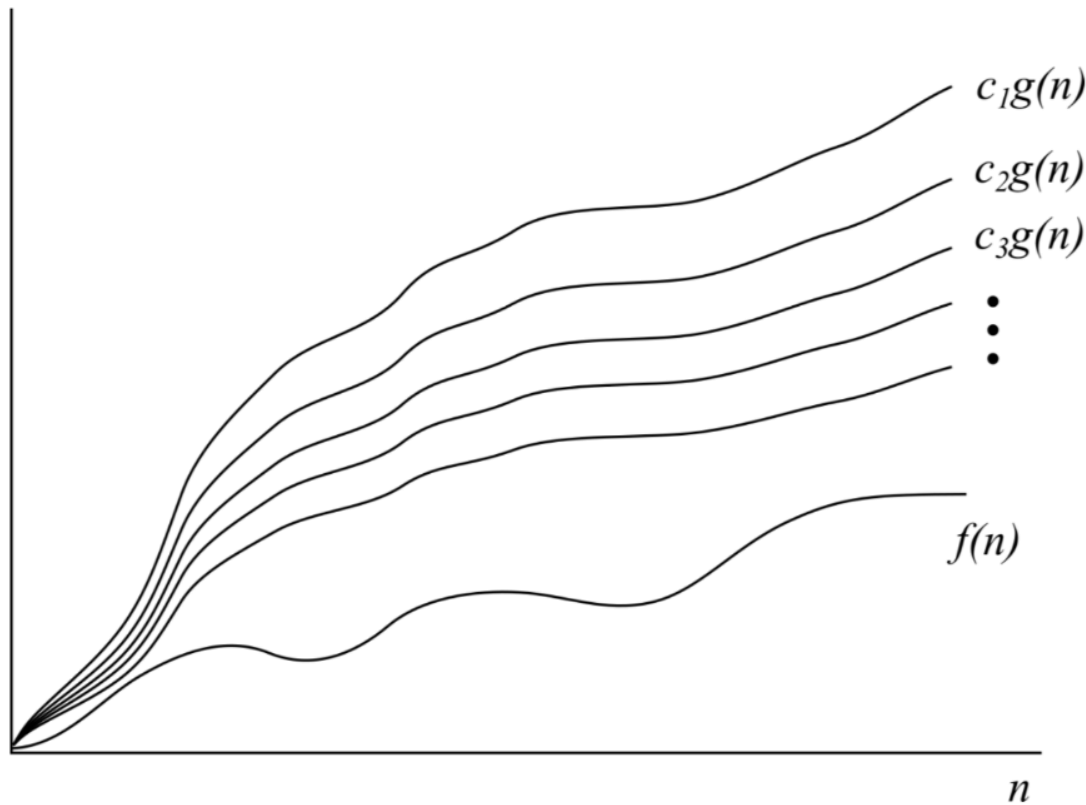
Θ -notation



$$\begin{aligned} &\exists c_1, c_2 > 0, n_0 > 0 \\ &0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \\ &n \geq n_0 \end{aligned}$$

$g(n)$ is an asymptotic **tight**-bound for $f(n)$

o-notation



$$f(n) = o(g(n))$$

$$\forall c > 0$$

$$\exists n_0 > 0$$

$$0 \leq f(n) \leq cg(n)$$

$$n \geq n_0$$

$g(n)$ is a **non-tight**
asymptotic **upper-**
bound for $f(n)$

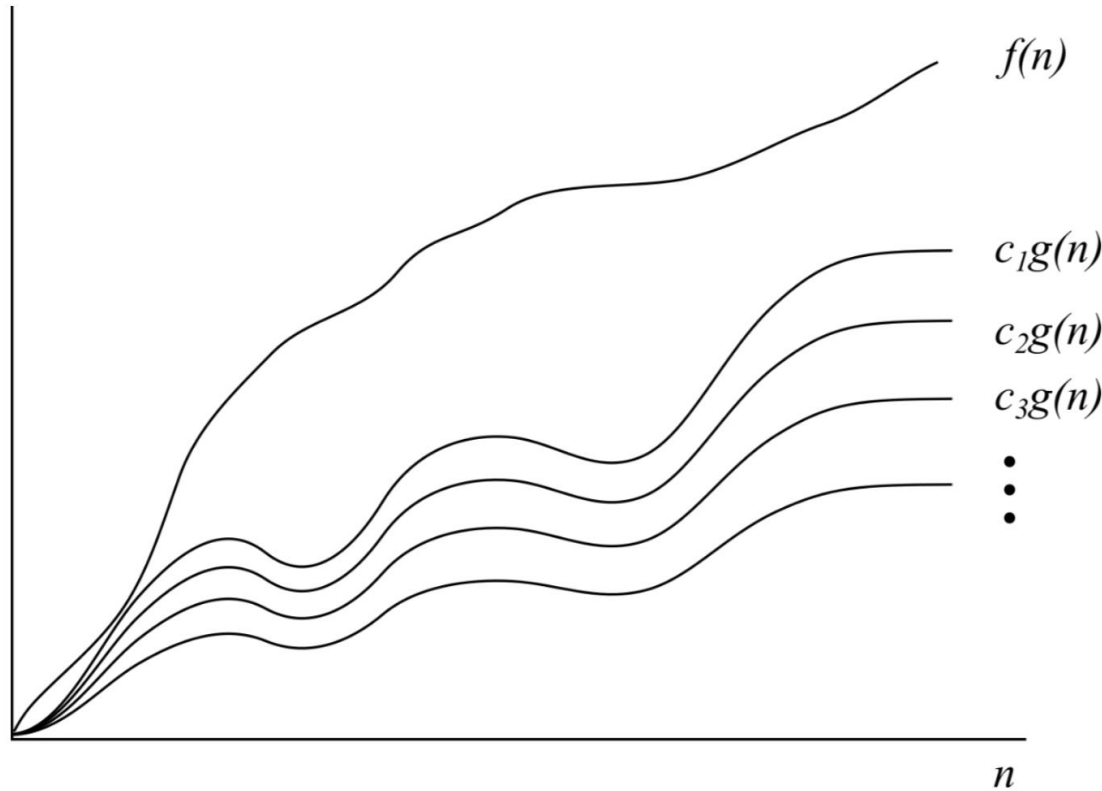
ω -notation

$$\forall c > 0$$

$$\exists n_0 > 0$$

$$0 \leq cgn(n) \leq f(n)$$

$$n \geq n_0$$



$$f(n) = \omega(g(n))$$

$g(n)$ is a **non-tight** asymptotic **lower-bound** for $f(n)$

Comparing Two Functions

- > $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$
- > 0: $f(n) = o(g(n))$
- > $c > 0$: $f(n) = \Theta(g(n))$
- > ∞ : $f(n) = \omega(g(n))$

Analogy to Real Numbers

Functions	Real numbers
$f(n) = O(g(n))$	$a \leq b$
$f(n) = \Omega(g(n))$	$a \geq b$
$f(n) = \Theta(g(n))$	$a = b$
$f(n) = o(g(n))$	$a < b$
$f(n) = \omega(g(n))$	$a > b$

Simple Rules

- › We can omit constants
- › We can omit lower order terms
- › $\Theta(an^2+bn+c)$ becomes $\Theta(n^2)$, a , b , c are constants
- › $\Theta(c1)$ and $\Theta(c2)$ become $\Theta(1)$, c 's are constants
- › $\Theta(\log_{k1}n)$ and $\Theta(\log_{k2}n)$ become $\Theta(\log n)$, k 's are constants
- › $\Theta(\log(n^k))$ becomes $\Theta(\log n)$, k is constant

Popular Classes of Functions

Constant: $f(n) = \Theta(1)$

Logarithmic: $f(n) = \Theta(\lg(n))$

Sublinear: $f(n) = o(n)$

Linear: $f(n) = \Theta(n)$

Super-linear: $f(n) = \omega(n)$

Quadratic: $f(n) = \Theta(n^2)$

Polynomial: $f(n) = \Theta(n^k)$; k is a constant

Exponential: $f(n) = \Theta(k^n)$; k is a constant

Insertion Sort Worst Case (Revisit)

- Input array is reversed

INSERTION-SORT(A, n)

for $j = 2$ **to** n

$key = A[j]$

 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.

$i = j - 1$

while $i > 0$ and $A[i] > key$

$A[i + 1] = A[i]$

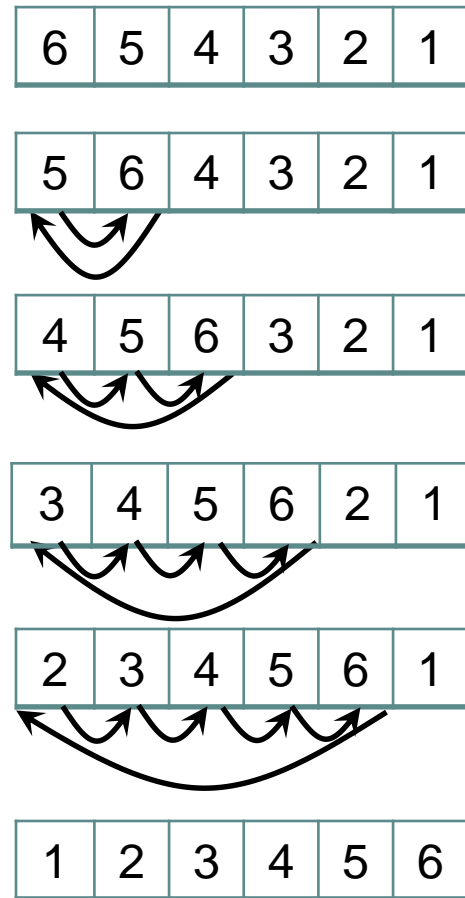
$i = i - 1$

$A[i + 1] = key$

max n

(n-1)

$T(n) = (n-1)*n = O(n^2)$



Comparing two algorithms

- › $T1(n) = 2n + 10000000$
- › $T2(n) = 200n + 1000$
- › Which is better? Why?
 - › In terms of order of growth?

Comparing two algorithms

- › $T1(n) = 2n + 10000000$
- › $T2(n) = 200n + 1000$
- › Which is better? Why?
 - › In terms of order of growth? Same

Comparing two algorithms

- › $T1(n) = 2n + 10000000$
- › $T2(n) = 200n + 1000$
- › Which is better? Why?
 - › In terms of order of growth? Same
 - › In terms of actual runtime?

Comparing two algorithms

- › $T1(n) = 2n + 10000000$
- › $T2(n) = 200n + 1000$
- › Which is better? Why?
 - › In terms of order of growth? Same
 - › In terms of actual runtime?
 - › For $n \leq 5045$, T2 is faster, otherwise T1 is faster

Comparing two algorithms

- › $T1(n) = 2n + 10000000$
- › $T2(n) = 200n + 1000$
- › Which is better? Why?
 - › In terms of order of growth? Same
 - › In terms of actual runtime?
 - › For $n \leq 5045$, T2 is faster, otherwise T1 is faster
- › What is the main usage of asymptotic notation analysis?

Analyzing Algorithms



```
› Algorithm 1
  for i = 1 to n
    j = 2*i
  for j = 1 to n/2
    print j
```

Analyzing Algorithms

- › Algorithm 2

```
for i = 1 to n/2 {  
    print i  
    for j = 1 to n, step j = j*2  
        print i*j  
}
```

Analyzing Algorithms

- › Algorithm 3

```
    for i = 1 to n/2
```

```
        print i
```

```
    for j = 1 to n, step j = j*2
```

```
        print i*j
```

Analyzing Algorithms



› Algorithm 4

input x (+ve integer)

while $x > 0$

 print x

$x = \lfloor x/5 \rfloor$

Credits & Book Readings

- › Book Readings
 - › 2.1, 2.2, 3.1, 3.2
- › Credits
 - › Prof. Ahmed Eldawy notes
 - › <http://www.cs.ucr.edu/~eldawy/17WCS141/slides/CS141-1-09-17.pdf>
 - › Online websites
 - › <https://commons.wikimedia.org/wiki/File:Exponential.svg>