Insertion Sort and Asymptotic Analysis

Pontus Ekberg

Uppsala University

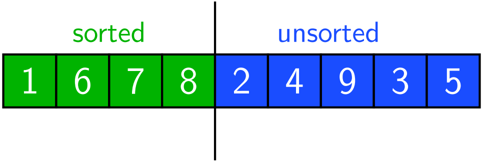
(Based on previous material by Mohamed Faouzi Atig and Parosh Aziz Abdulla)

Sorting: Insertion Sort

We want to sort an array A of n elements in non-decreasing order.

The Insertion sort algorithm divides A into two sub-arrays:

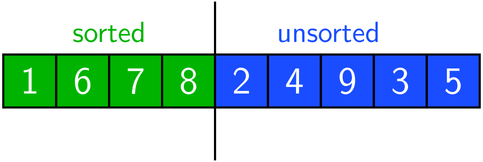
- The sorted section (usually occupying the lower positions).
- The unsorted section (not treated yet).



Sorting: Insertion Sort

The Insertion sort works as follows:

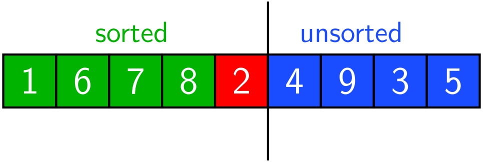
- Initially, all the array is unsorted.
- While the border line is not at the end of the array:
  - Move the line one step to the right.
  - While the newly added element is strictly less than its left neighbor:
    - Swap the newly added element with its left neighbor.



Sorting: Insertion Sort

The Insertion sort works as follows:

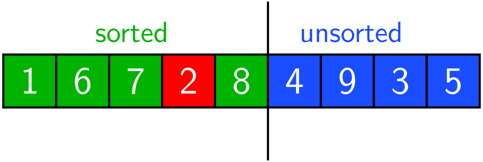
- Initially, all the array is unsorted.
- While the border line is not at the end of the array:
  - Move the line one step to the right.
  - While the newly added element is strictly less than its left neighbor:
    - Swap the newly added element with its left neighbor.



Sorting: Insertion Sort

The Insertion sort works as follows:

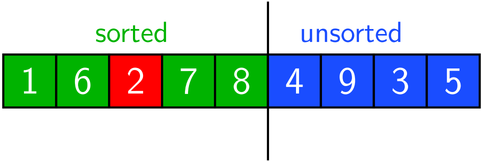
- Initially, all the array is unsorted.
- While the border line is not at the end of the array:
  - Move the line one step to the right.
  - While the newly added element is strictly less than its left neighbor:
    - Swap the newly added element with its left neighbor.



Sorting: Insertion Sort

The Insertion sort works as follows:

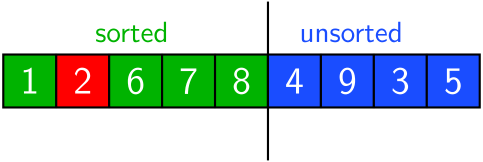
- Initially, all the array is unsorted.
- While the border line is not at the end of the array:
    - Move the line one step to the right.
    - While the newly added element is strictly less than its left neighbor:
        - Swap the newly added element with its left neighbor.



Sorting: Insertion Sort

The Insertion sort works as follows:

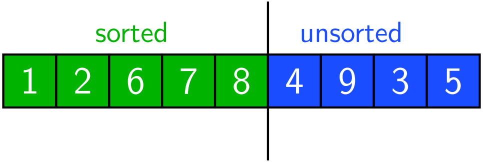
- Initially, all the array is unsorted.
- While the border line is not at the end of the array:
  - Move the line one step to the right.
  - While the newly added element is strictly less than its left neighbor:
    - Swap the newly added element with its left neighbor.



Sorting: Insertion Sort

The Insertion sort works as follows:

- Initially, all the array is unsorted.
- While the border line is not at the end of the array:
    - Move the line one step to the right.
    - While the newly added element is strictly less than its left neighbor:
        - Swap the newly added element with its left neighbor.



Insertion Sort

```
INSERTION-SORT(A)
for j ← 2 to A.length
     do key ← A[j]
        ▷ Insert A[j] into A[1 . . j − 1].
        i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

6	3	5	1	2	4

Insertion Sort

INSERTION-SORT(A)

```
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

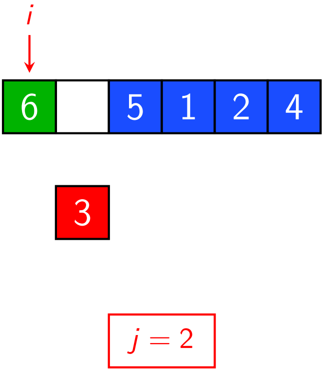
6	3	5	1	2	4

$\boxed{j = 2}$

Insertion Sort

INSERTION-SORT(A)

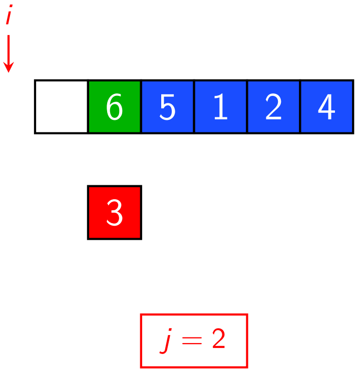
for $j \leftarrow 2$ **to** $A.\,length$
  do $key \leftarrow A[j]$
    $\triangleright$ Insert $A[j]$ into $A[1\,.\,.\,j-1]$.
    $i \leftarrow j - 1$
    while $i > 0$ and $A[i] > key$
      do $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] \leftarrow key$



Insertion Sort

INSERTION-SORT(A)

for $j \leftarrow 2$ **to** $A.\,length$
  do $key \leftarrow A[j]$
    $\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.
    $i \leftarrow j - 1$
    while $i > 0$ and $A[i] > key$
      do $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] \leftarrow key$



Insertion Sort

INSERTION-SORT(A)

```
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

3	6	5	1	2	4

$\boxed{j = 2}$

Insertion Sort

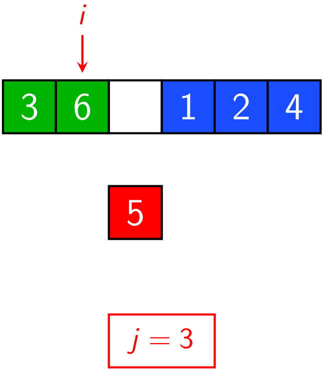
```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

3	6	5	1	2	4

$\boxed{j = 3}$

Insertion Sort

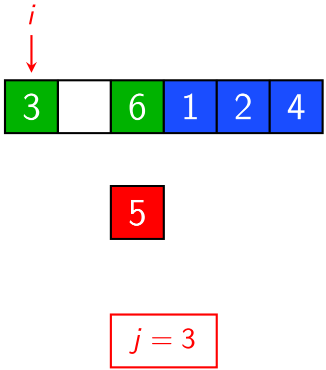
```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort

INSERTION-SORT(A)

```
for j ← 2 to A.length
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
              do A[i + 1] ← A[i]
                 i ← i − 1
         A[i + 1] ← key
```



Insertion Sort

```
INSERTION-SORT(A)
for j ← 2 to A.length
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
         A[i + 1] ← key
```

3	5	6	1	2	4

$\boxed{j = 3}$

Insertion Sort

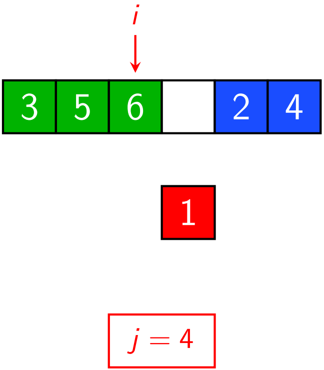
```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

3	5	6	1	2	4

$\boxed{j = 4}$

Insertion Sort

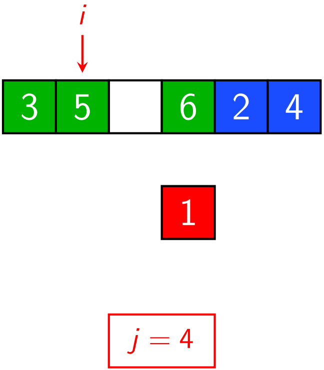
```
INSERTION-SORT(A)
for j ← 2 to A.length
     do key ← A[j]
          ▷ Insert A[j] into A[1 . . j − 1].
          i ← j − 1
          while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
          A[i + 1] ← key
```



Insertion Sort

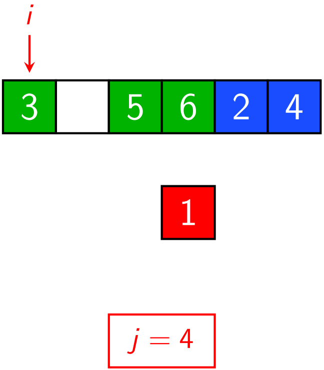
Insertion-Sort(A)

```
for j ← 2 to A.length
        do key ← A[j]
           ▷ Insert A[j] into A[1 . . j − 1].
           i ← j − 1
           while i > 0 and A[i] > key
                do A[i + 1] ← A[i]
                   i ← i − 1
           A[i + 1] ← key
```



Insertion Sort

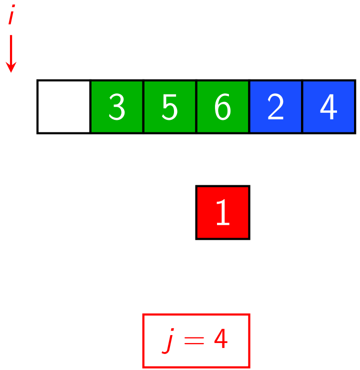
```
INSERTION-SORT(A)
for j ← 2 to A.length
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
         A[i + 1] ← key
```



Insertion Sort

INSERTION-SORT(A)

```
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort

INSERTION-SORT(A)

```
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

1	3	5	6	2	4

$\boxed{j = 4}$

Insertion Sort

INSERTION-SORT(A)

```
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

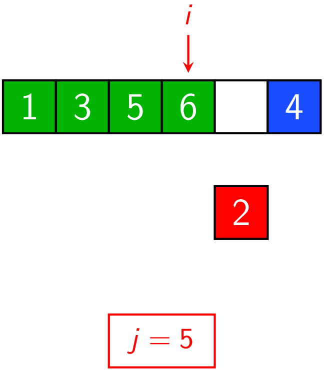
1	3	5	6	2	4

$\boxed{j = 5}$

Insertion Sort

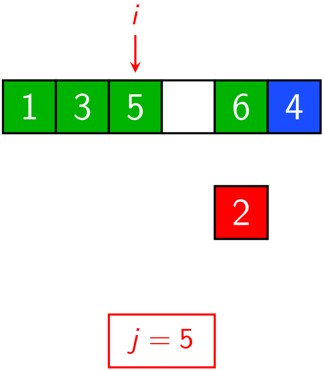
INSERTION-SORT(A)

for $j \leftarrow 2$ **to** $A.\,length$
  do $key \leftarrow A[j]$
    ▷ Insert $A[j]$ into $A[1\,..\,j-1]$.
    $i \leftarrow j - 1$
    while $i > 0$ and $A[i] > key$
      do $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] \leftarrow key$



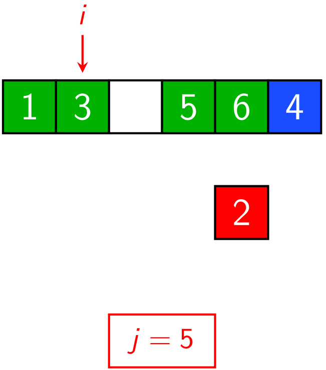
Insertion Sort

```
INSERTION-SORT(A)
for j ← 2 to A.length
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
              do A[i + 1] ← A[i]
                 i ← i − 1
         A[i + 1] ← key
```



Insertion Sort

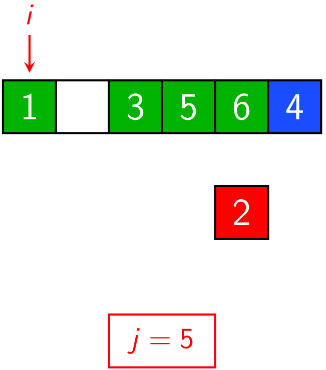
```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort

INSERTION-SORT(A)

```
for j ← 2 to A.length
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
         A[i + 1] ← key
```



```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

1	2	3	5	6	4

$\boxed{j = 5}$

Insertion Sort

```
INSERTION-SORT(A)
for j ← 2 to A.length
        do key ← A[j]
            ▷ Insert A[j] into A[1 . . j − 1].
            i ← j − 1
            while i > 0 and A[i] > key
                do A[i + 1] ← A[i]
                    i ← i − 1
            A[i + 1] ← key
```

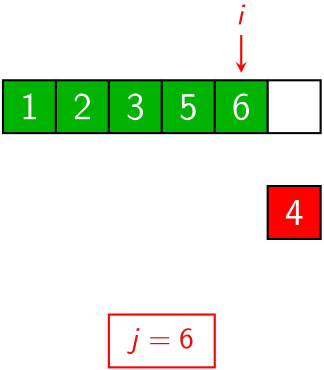
1	2	3	5	6	4

$\boxed{j = 6}$

Insertion Sort

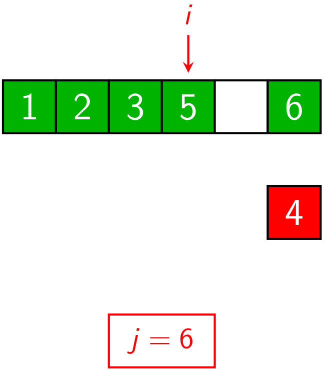
INSERTION-SORT(A)

```
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort

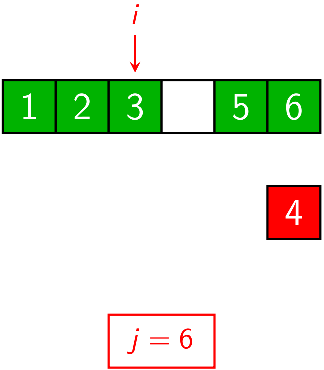
```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort

INSERTION-SORT(A)

for $j \leftarrow 2$ **to** $A.\,length$
  do $key \leftarrow A[j]$
    $\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.
    $i \leftarrow j - 1$
    while $i > 0$ and $A[i] > key$
      do $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] \leftarrow key$



Insertion Sort

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

1	2	3	4	5	6

$\boxed{j = 6}$

Insertion Sort

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
        ▷ Insert A[j] into A[1 . . j − 1].
        i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

1	2	3	4	5	6

$\boxed{j = 6}$

Correctness – Loop Invariants

Loop Invariant:

- Property which remains true throughout the execution of a loop.
- It should be a useful property for showing correctness of the algorithm.

How to Show a Loop Invariant?

- Initialization: The invariant holds prior to the first iteration of the loop.
- Maintenance: The invariant is preserved by each iteration of the loop. In other words, if it holds before the next iteration, it will also hold after performing that iteration. This continues until (and including) the point of termination of the loop.
- Termination: At the point of termination, the invariant implies a "useful property", which in turn can be used for proving correctness of the program.

Example – Insertion Sort

Loop Invariant (of the outer loop): The subarray $A[1 \mathinner{.\,.} j-1]$ is a sorted permutation of the the original subarray $A[1 \mathinner{.\,.} j-1]$.

We show the loop invariant as follows:

Example – Insertion Sort

Loop Invariant (of the outer loop): The subarray $A[1\,..\,j-1]$ is a sorted permutation of the the original subarray $A[1\,..\,j-1]$.

We show the loop invariant as follows:

- Initialization: $j = 2$, and $A[1\,..\,1] = A[1]$ is a sorted permutation of $A[1]$.

Example – Insertion Sort

Loop Invariant (of the outer loop): The subarray $A[1\,..\,j-1]$ is a sorted permutation of the the original subarray $A[1\,..\,j-1]$.

We show the loop invariant as follows:

- Initialization: $j = 2$, and $A[1\,..\,1] = A[1]$ is a sorted permutation of $A[1]$.
- Maintenance: The loop of lines $5-7$ finds the smallest k such that $1 \le k < j$ and $A[j] < A[k]$. It shifts the subarray $A[k\,..\,j-1]$ one step to the right, and inserts $A[j]$ in the resulting hole, i.e., in position k. Therefore the invariant is maintained.

Example – Insertion Sort

Loop Invariant (of the outer loop): The subarray $A[1\,..\,j-1]$ is a sorted permutation of the the original subarray $A[1\,..\,j-1]$.

We show the loop invariant as follows:

- Initialization: $j = 2$, and $A[1\,..\,1] = A[1]$ is a sorted permutation of $A[1]$.
- Maintenance: The loop of lines $5 - 7$ finds the smallest k such that $1 \leq k < j$ and $A[j] < A[k]$. It shifts the subarray $A[k\,..\,j-1]$ one step to the right, and inserts $A[j]$ in the resulting hole, i.e., in position k. Therefore the invariant is maintained.
- Termination: When the loop terminates, we have $j = n+1$. The invariant then implies that the array $A[1\,..\,n]$ is a sorted permutation of the the original array $A[1\,..\,n]$.

Insertion Sort - Cost Analysis

```
INSERTION-SORT(A)                                  cost      times
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 .. j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

Insertion Sort - Cost Analysis

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.length$	c_1	
do $key \leftarrow A[j]$	c_2	
$\triangleright$ Insert $A[j]$ into $A[1 \,..\, j-1]$.	0	
$i \leftarrow j - 1$	c_4	
while $i > 0$ and $A[i] > key$	c_5	
do $A[i+1] \leftarrow A[i]$	c_6	
$i \leftarrow i - 1$	c_7	
$A[i+1] \leftarrow key$	c_8	

Insertion Sort - Cost Analysis

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\,length$	c_1	n
do $key \leftarrow A[j]$	c_2	
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	
$i \leftarrow j - 1$	c_4	
while $i > 0$ and $A[i] > key$	c_5	
do $A[i+1] \leftarrow A[i]$	c_6	
$i \leftarrow i - 1$	c_7	
$A[i+1] \leftarrow key$	c_8	

Insertion Sort - Cost Analysis

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\,length$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	
$i \leftarrow j-1$	c_4	
while $i > 0$ and $A[i] > key$	c_5	
do $A[i+1] \leftarrow A[i]$	c_6	
$i \leftarrow i-1$	c_7	
$A[i+1] \leftarrow key$	c_8	

Insertion Sort - Cost Analysis

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\,length$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	
while $i > 0$ and $A[i] > key$	c_5	
do $A[i+1] \leftarrow A[i]$	c_6	
$i \leftarrow i-1$	c_7	
$A[i+1] \leftarrow key$	c_8	

Insertion Sort - Cost Analysis

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\,length$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	
do $A[i+1] \leftarrow A[i]$	c_6	
$i \leftarrow i-1$	c_7	
$A[i+1] \leftarrow key$	c_8	

Insertion Sort - Cost Analysis

	Insertion-Sort(A)	cost	times
1	**for** $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
2	**do** $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > \mathit{key}$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	
7	$i \leftarrow i-1$	c_7	
8	$A[i+1] \leftarrow \mathit{key}$	c_8	

- t_j denotes the number of times the control part of the **while** loop in line 5 is executed during the j^{th} iteration.
- Observe that t_j depends on the value of the input.

Insertion Sort - Cost Analysis

	INSERTION-SORT(A)	cost	times
1	**for** $j \leftarrow 2$ **to** $A.length$	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	
8	$A[i+1] \leftarrow key$	c_8	

- t_j denotes the number of times the control part of the **while** loop in line 5 is executed during the j^{th} iteration.
- Observe that t_j depends on the value of the input.

Insertion Sort - Cost Analysis

	INSERTION-SORT(A)	cost	times
1	**for** $j \leftarrow 2$ **to** $A.\,length$	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow key$	c_8	

- t_j denotes the number of times the control part of the **while** loop in line 5 is executed during the j^{th} iteration.
- Observe that t_j depends on the value of the input.

Insertion Sort - Cost Analysis

	INSERTION-SORT(A)	cost	times
1	**for** $j \leftarrow 2$ **to** $A.length$	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

- t_j denotes the number of times the control part of the **while** loop in line 5 is executed during the j^{th} iteration.
- Observe that t_j depends on the value of the input.

Insertion Sort - Best Case

What is the best case?

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
      do key ← A[j]
          ▷ Insert A[j] into A[1 . . j − 1].
          i ← j − 1
          while i > 0 and A[i] > key
                do A[i + 1] ← A[i]
                    i ← i − 1
          A[i + 1] ← key
```

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

INSERTION-SORT(A)

for $j \leftarrow 2$ **to** $A.\,length$
    do $key \leftarrow A[j]$
        $\triangleright$ Insert $A[j]$ into $A[1\,.\,.\,j-1]$.
        $i \leftarrow j-1$
        while $i > 0$ and $A[i] > key$
            do $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$
        $A[i+1] \leftarrow key$

1	2	3	4	5	6

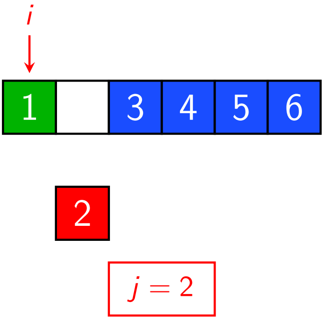
$\boxed{j = 2}$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1..j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

1	2	3	4	5	6

$\boxed{j = 2}$

$\boxed{t_j = 1}$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

1	2	3	4	5	6

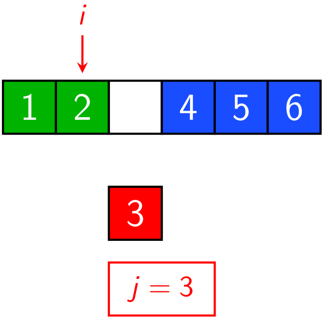
$\boxed{j = 3}$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
        ▷ Insert A[j] into A[1 . . j − 1].
        i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

1	2	3	4	5	6

$\boxed{j = 3}$

$\boxed{t_j = 1}$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \le key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
        do key ← A[j]
            ▷ Insert A[j] into A[1 . . j − 1].
            i ← j − 1
            while i > 0 and A[i] > key
                  do A[i + 1] ← A[i]
                     i ← i − 1
            A[i + 1] ← key
```

1	2	3	4	5	6

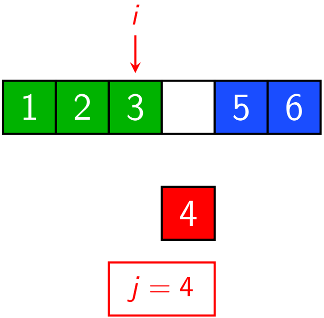
$\boxed{j = 4}$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
     do key ← A[j]
          ▷ Insert A[j] into A[1 . . j − 1].
          i ← j − 1
          while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
          A[i + 1] ← key
```



Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

INSERTION-SORT(A)

```
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 .. j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

1	2	3	4	5	6

$\boxed{j = 4}$

$\boxed{t_j = 1}$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 .. j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

1	2	3	4	5	6

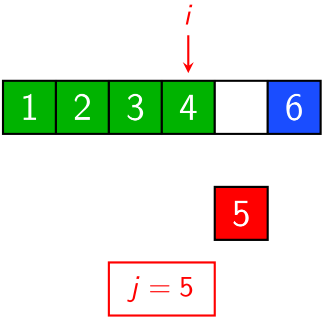
$\boxed{j = 5}$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
        ▷ Insert A[j] into A[1 . . j − 1].
        i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

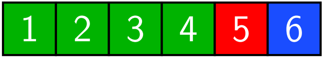


Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



$\boxed{j = 5}$

$\boxed{t_j = 1}$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

1	2	3	4	5	6

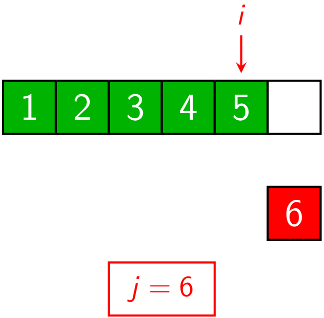
$\boxed{j = 6}$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

- Always find $A[i] \leq key$ upon the first time the **while** loop is tested.

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
        ▷ Insert A[j] into A[1 . . j − 1].
        i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

1	2	3	4	5	6

$\boxed{j = 6}$

$\boxed{t_j = 1}$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\,length$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} 1$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (0)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (0)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.length$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$n-1$
do $A[i+1] \leftarrow A[i]$	c_6	0
$i \leftarrow i-1$	c_7	0
$A[i+1] \leftarrow key$	c_8	$n-1$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > \mathit{key}$	c_5	$n-1$
do $A[i+1] \leftarrow A[i]$	c_6	0
$i \leftarrow i-1$	c_7	0
$A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

$$T(n) = c_1 + (c_1 + c_2 + c_4 + c_5 + c_8)(n-1)$$

Insertion Sort - Best Case

Best Case: The array is already sorted: $t_j = 1$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > \mathit{key}$	c_5	$n-1$
do $A[i+1] \leftarrow A[i]$	c_6	0
$i \leftarrow i-1$	c_7	0
$A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

$$T(n) = c_1 + (c_1 + c_2 + c_4 + c_5 + c_8)(n-1)$$

In the best case, insertion sort runs in linear time ($T(n)$ is of the form $an + b$)

Insertion Sort - Worst Case

What is the worst case?

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

6	5	4	3	2	1

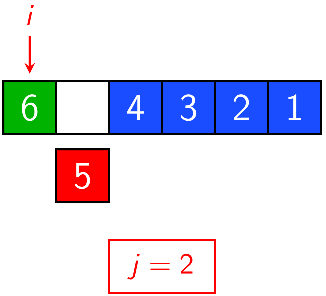
$\boxed{j = 2}$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
             do A[i + 1] ← A[i]
                i ← i − 1
         A[i + 1] ← key
```

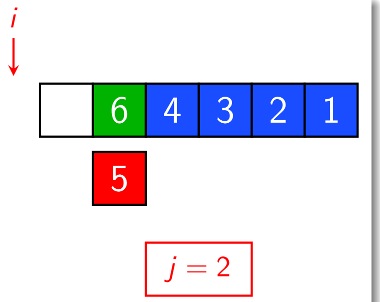


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
         A[i + 1] ← key
```

5	6	4	3	2	1

$\boxed{j = 2}$

$\boxed{t_j = 2}$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

5	6	4	3	2	1

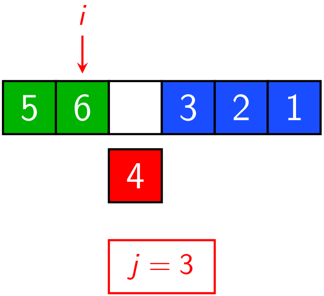
$\boxed{j = 3}$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

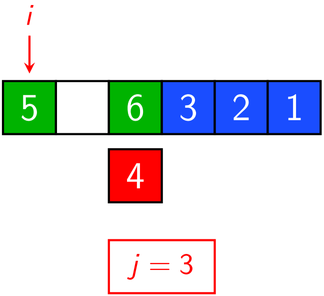


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
         A[i + 1] ← key
```

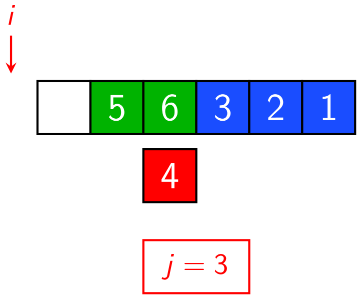


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
             do A[i + 1] ← A[i]
                i ← i − 1
         A[i + 1] ← key
```

4	5	6	3	2	1

$\boxed{j = 2}$

$\boxed{t_j = 3}$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

4	5	6	3	2	1

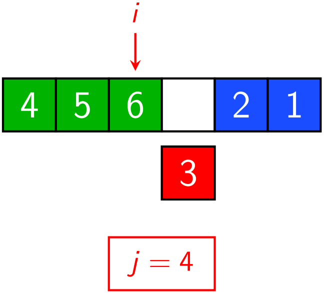
$\boxed{j = 4}$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

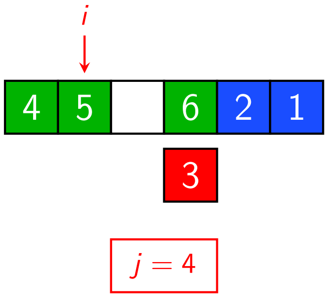


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

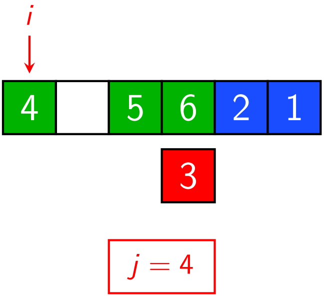


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

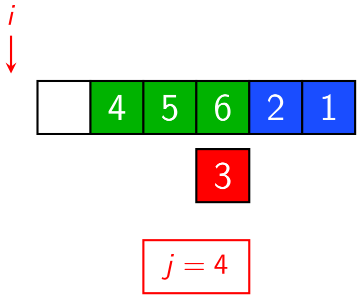


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

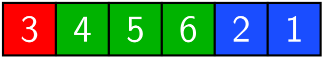


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



$\boxed{j = 4}$

$\boxed{t_j = 4}$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
     do key ← A[j]
        ▷ Insert A[j] into A[1 . . j − 1].
        i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

3	4	5	6	2	1

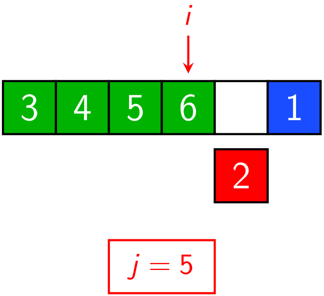
$\boxed{j = 5}$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

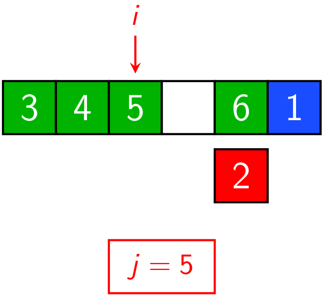


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
             do A[i + 1] ← A[i]
                i ← i − 1
         A[i + 1] ← key
```

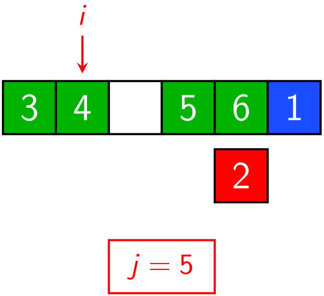


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
         A[i + 1] ← key
```

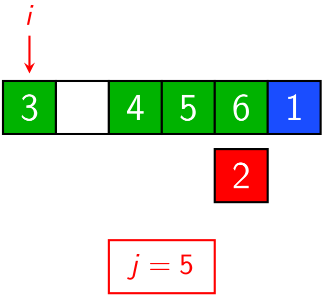


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
     do key ← A[j]
        ▷ Insert A[j] into A[1 . . j − 1].
        i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

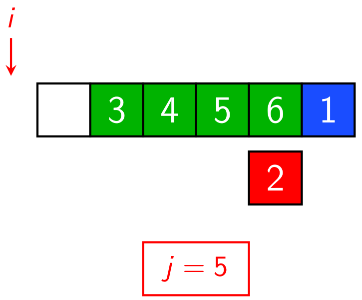


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

2	3	4	5	6	1

$\boxed{j = 5}$

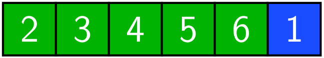
$\boxed{t_j = 5}$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
          ▷ Insert A[j] into A[1 . . j − 1].
          i ← j − 1
          while i > 0 and A[i] > key
                do A[i + 1] ← A[i]
                    i ← i − 1
          A[i + 1] ← key
```



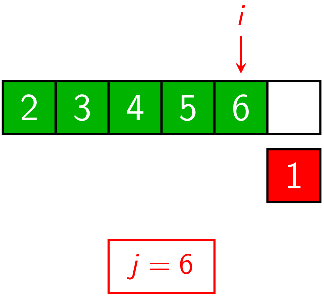
$\boxed{j = 6}$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

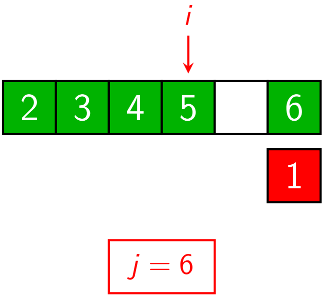


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
          ▷ Insert A[j] into A[1 . . j − 1].
          i ← j − 1
          while i > 0 and A[i] > key
              do A[i + 1] ← A[i]
                  i ← i − 1
          A[i + 1] ← key
```

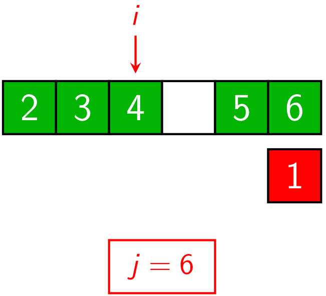


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

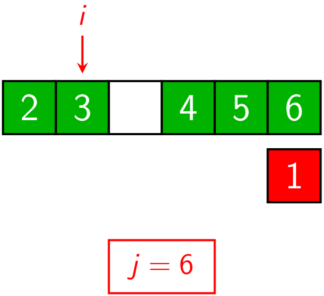


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

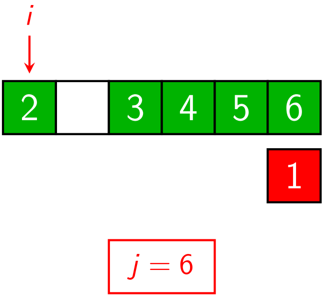


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
     do key ← A[j]
        ▷ Insert A[j] into A[1 . . j − 1].
        i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

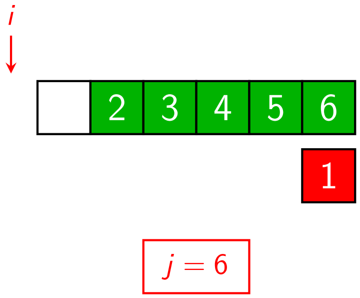


Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

- Always find $A[i] > key$ in **while** loop test.
- Have to compare *key* with all elements to the left of the j^{th} position

INSERTION-SORT(A)

```
for j ← 2 to length[A]
      do key ← A[j]
         ▷ Insert A[j] into A[1 . . j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
              do A[i + 1] ← A[i]
                 i ← i − 1
         A[i + 1] ← key
```

1	2	3	4	5	6

$\boxed{j = 6}$

$\boxed{t_j = 6}$

Insertion Sort - Worst Case

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,.\,.\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > \mathit{key}$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > \mathit{key}$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

Insertion-Sort(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1 \mathinner{.\,.} j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > \mathit{key}$	c_5	$\sum_{j=2}^{n} j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (j-1)$
$A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > \mathit{key}$	c_5	$\sum_{j=2}^{n} j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (j-1)$
$A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

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

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.length$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\frac{n(n+1)}{2} - 1$
do $A[i+1] \leftarrow A[i]$	c_6	$\frac{n(n-1)}{2}$
$i \leftarrow i-1$	c_7	$\frac{n(n-1)}{2}$
$A[i+1] \leftarrow key$	c_8	$n-1$

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

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.length$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
▷ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\frac{n(n+1)}{2} - 1$
do $A[i+1] \leftarrow A[i]$	c_6	$\frac{n(n-1)}{2}$
$i \leftarrow i-1$	c_7	$\frac{n(n-1)}{2}$
$A[i+1] \leftarrow key$	c_8	$n-1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7)\left(\frac{n(n-1)}{2}\right)$$

Insertion Sort - Worst Case

Worst Case: The array is reverse sorted: $t_j = j$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.length$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1..j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\frac{n(n+1)}{2} - 1$
do $A[i+1] \leftarrow A[i]$	c_6	$\frac{n(n-1)}{2}$
$i \leftarrow i-1$	c_7	$\frac{n(n-1)}{2}$
$A[i+1] \leftarrow key$	c_8	$n-1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7)\left(\frac{n(n-1)}{2}\right)$$

In the worst case, insertion sort runs in quadratic time
($T(n)$ is of the form $an^2 + bn + c$)

Insertion Sort - Average Case

What is the average case?

Insertion Sort - Average Case

What is the average case?

- It depends on the probability distribution of inputs!

Insertion Sort - Average Case

What is the average case?

- It depends on the probability distribution of inputs!
- Let's assume uniformly random permutations in the inputs.

Insertion Sort - Average Case

What is the average case?

- It depends on the probability distribution of inputs!
- Let's assume uniformly random permutations in the inputs.
- In other words, when given input arrays of length n, we assume that all the $n!$ possible initial permutations/orderings are equally likely.

Insertion Sort - Average Case

Average Case: $t_j \simeq \frac{j}{2}$

- On average, $A[j]$ is smaller than half of the elements in $A[1..j-1]$
- On average, the **while** loop has to look halfway through the sorted subarray $A[1..j-1]$ to decide where to put *key*

INSERTION-SORT(A)

```
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

1	3	6	8	8	9	7	6

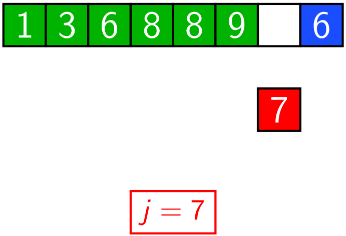
$\boxed{j = 7}$

Insertion Sort - Average Case

Average Case: $t_j \simeq \frac{j}{2}$

- On average, $A[j]$ is smaller than half of the elements in $A[1..j-1]$
- On average, the **while** loop has to look halfway through the sorted subarray $A[1..j-1]$ to decide where to put *key*

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```



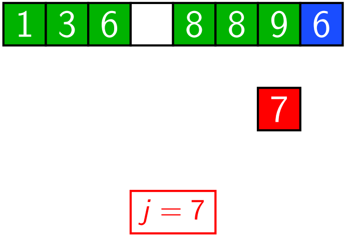
Insertion Sort - Average Case

Average Case: $t_j \simeq \frac{j}{2}$

- On average, $A[j]$ is smaller than half of the elements in $A[1..j-1]$
- On average, the **while** loop has to look halfway through the sorted subarray $A[1..j-1]$ to decide where to put *key*

INSERTION-SORT(A)

```
for j ← 2 to A.length
      do key ← A[j]
          ▷ Insert A[j] into A[1 . . j − 1].
          i ← j − 1
          while i > 0 and A[i] > key
              do A[i + 1] ← A[i]
                  i ← i − 1
          A[i + 1] ← key
```



Insertion Sort - Average Case

Average Case: $t_j \simeq \frac{j}{2}$

- On average, $A[j]$ is smaller than half of the elements in $A[1..j-1]$
- On average, the **while** loop has to look halfway through the sorted subarray $A[1..j-1]$ to decide where to put *key*

```
INSERTION-SORT(A)
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

1	3	6	7	8	8	9	6

$\boxed{j = 7}$

$\boxed{t_j \approx \frac{j}{2}}$

Insertion Sort - Average Case

Average Case: $t_j = \frac{j}{2}$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,.\,.\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > \mathit{key}$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

Insertion Sort - Average Case

Average Case: $t_j = \frac{j}{2}$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > \mathit{key}$	c_5	$\sum_{j=2}^{n} \frac{j}{2}$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} \left(\frac{j}{2} - 1\right)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} \left(\frac{j}{2} - 1\right)$
$A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

Insertion Sort - Average Case

Average Case: $t_j = \frac{j}{2}$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > \mathit{key}$	c_5	$\sum_{j=2}^{n} \frac{j}{2}$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} \left(\frac{j}{2} - 1\right)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} \left(\frac{j}{2} - 1\right)$
$A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

$\sum_{j=2}^{n} \frac{j}{2} = \frac{n(n+1)}{4} - \frac{1}{2}$ and $\sum_{j=2}^{n}(\frac{j}{2} - 1) = \frac{n(n-3)}{4} + \frac{1}{2}$

Insertion Sort - Average Case

Average Case: $t_j = \frac{j}{2}$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.length$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,.\,.\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\frac{n(n+1)}{4} - \frac{1}{2}$
do $A[i+1] \leftarrow A[i]$	c_6	$\frac{n(n-3)}{4} + \frac{1}{2}$
$i \leftarrow i-1$	c_7	$\frac{n(n-3)}{4} + \frac{1}{2}$
$A[i+1] \leftarrow key$	c_8	$n-1$

$\sum_{j=2}^{n} \frac{j}{2} = \frac{n(n+1)}{4} - \frac{1}{2}$ and $\sum_{j=2}^{n} (\frac{j}{2} - 1) = \frac{n(n-3)}{4} + \frac{1}{2}$

Insertion Sort - Average Case

Average Case: $t_j = \frac{j}{2}$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\mathit{length}$	c_1	n
do $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > \mathit{key}$	c_5	$\frac{n(n+1)}{4} - \frac{1}{2}$
do $A[i+1] \leftarrow A[i]$	c_6	$\frac{n(n-3)}{4} + \frac{1}{2}$
$i \leftarrow i-1$	c_7	$\frac{n(n-3)}{4} + \frac{1}{2}$
$A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5\left(\frac{n(n+1)}{4} - \frac{1}{2}\right) + (c_6 + c_7)\left(\frac{n(n-3)}{4} + \frac{1}{2}\right)$$

Insertion Sort - Average Case

Average Case: $t_j = \frac{j}{2}$

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.\,length$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\frac{n(n+1)}{4} - \frac{1}{2}$
do $A[i+1] \leftarrow A[i]$	c_6	$\frac{n(n-3)}{4} + \frac{1}{2}$
$i \leftarrow i-1$	c_7	$\frac{n(n-3)}{4} + \frac{1}{2}$
$A[i+1] \leftarrow key$	c_8	$n-1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5\left(\frac{n(n+1)}{4} - \frac{1}{2}\right) + (c_6 + c_7)\left(\frac{n(n-3)}{4} + \frac{1}{2}\right)$$

This gives quadratic time complexity
The average case is often similar to the worst case

Back to Algorithm Analysis

- $T(n) = an^2 + bn + c$ is useful, but approximate, predictor of the worst-case runtime of the Insertion Sort algorithm.

Back to Algorithm Analysis

- $T(n) = an^2 + bn + c$ is useful, but approximate, predictor of the worst-case runtime of the Insertion Sort algorithm.

- a, b, and c can vary with every change in the hardware or software environment (Not our main interest)

Back to Algorithm Analysis

- $T(n) = an^2 + bn + c$ is useful, but approximate, predictor of the worst-case runtime of the Insertion Sort algorithm.
- a, b, and c can vary with every change in the hardware or software environment (Not our main interest)
- We are interested in the order of growth of the runtime.

Back to Algorithm Analysis

- $T(n) = an^2 + bn + c$ is useful, but approximate, predictor of the worst-case runtime of the Insertion Sort algorithm.
- a, b, and c can vary with every change in the hardware or software environment (Not our main interest)
- We are interested in the order of growth of the runtime.
  - Calling Insertion Sort algorithm with an array twice as along will approximately quadruple the runtime.

Back to Algorithm Analysis

- $T(n) = an^2 + bn + c$ is useful, but approximate, predictor of the worst-case runtime of the Insertion Sort algorithm.
- a, b, and c can vary with every change in the hardware or software environment (Not our main interest)
- We are interested in the order of growth of the runtime.
  - Calling Insertion Sort algorithm with an array twice as along will approximately quadruple the runtime.
  - Almost all algorithms are fast for small values of n, so our focus is on how they scale as n grows big.

Asymptotic Analysis

We simplify the runtime $T(n)$ by:

- Keeping only the leading term of $T(n)$

Asymptotic Analysis

We simplify the runtime $T(n)$ by:

- Keeping only the leading term of $T(n)$
    - Lower-order terms are relatively insignificant for large values of n

Asymptotic Analysis

We simplify the runtime $T(n)$ by:

- Keeping only the leading term of $T(n)$
    - Lower-order terms are relatively insignificant for large values of n
    - If $T(n) = 10n^3 + 2n^2 + 1000$ then we keep only $10n^3$

Asymptotic Analysis

We simplify the runtime $T(n)$ by:

- Keeping only the leading term of $T(n)$
  - Lower-order terms are relatively insignificant for large values of n
  - If $T(n) = 10n^3 + 2n^2 + 1000$ then we keep only $10n^3$
    - $T(1000) = 10002001000$ and $10(1000)^3 = 10000000000$

Asymptotic Analysis

We simplify the runtime $T(n)$ by:

- Keeping only the leading term of $T(n)$
  - Lower-order terms are relatively insignificant for large values of n
  - If $T(n) = 10n^3 + 2n^2 + 1000$ then we keep only $10n^3$
    - $T(1000) = 10002001000$ and $10(1000)^3 = 10000000000$
- Ignoring the leading term's constant

Asymptotic Analysis

We simplify the runtime $T(n)$ by:

- Keeping only the leading term of $T(n)$
  - Lower-order terms are relatively insignificant for large values of n
  - If $T(n) = 10n^3 + 2n^2 + 1000$ then we keep only $10n^3$
    - $T(1000) = 10002001000$ and $10(1000)^3 = 10000000000$
- Ignoring the leading term's constant
  - If $T(n) = 10n^3 + 2n^2 + 1000$ then we keep only n^3

Asymptotic Analysis

We simplify the runtime $T(n)$ by:

- Keeping only the leading term of $T(n)$
  - Lower-order terms are relatively insignificant for large values of n
  - If $T(n) = 10n^3 + 2n^2 + 1000$ then we keep only $10n^3$
    - $T(1000) = 10002001000$ and $10(1000)^3 = 10000000000$
- Ignoring the leading term's constant
  - If $T(n) = 10n^3 + 2n^2 + 1000$ then we keep only n^3

If $T(n) = 10n^3 + 2n^2 + 1000$ then we say that $T(n)$ is $\Theta(n^3)$

Asymptotic Notations

Asymptotic Notations

- Characterize the order of growth of the runtimes of algorithms.
- For functions $f(n)$ and $g(n)$, we will define three notations:
    - O-notation: $f(n) \in O(g(n))$

Asymptotic Notations

- Characterize the order of growth of the runtimes of algorithms.
- For functions $f(n)$ and $g(n)$, we will define three notations:
  - O-notation: $f(n) \in O(g(n))$
    $\approx$ "$f(n)$ grows no faster than $g(n)$"

Asymptotic Notations

- Characterize the order of growth of the runtimes of algorithms.
- For functions $f(n)$ and $g(n)$, we will define three notations:
  - O-notation: $f(n) \in O(g(n))$

    $\approx$ "$f(n)$ grows no faster than $g(n)$"
  - Ω-notation: $f(n) \in \Omega(g(n))$

    $\approx$ "$f(n)$ grows no slower than $g(n)$"

Asymptotic Notations

- Characterize the order of growth of the runtimes of algorithms.
- For functions $f(n)$ and $g(n)$, we will define three notations:
  - O-notation: $f(n) \in O(g(n))$
    $\approx$ "$f(n)$ grows no faster than $g(n)$"
  - Ω-notation: $f(n) \in \Omega(g(n))$
    $\approx$ "$f(n)$ grows no slower than $g(n)$"
  - Θ-notation: $f(n) \in \Theta(g(n))$
    $\approx$ "$f(n)$ grows the same as $g(n)$"

The O-Notation

- For a given function $g(n)$, we denote by $O(g(n))$ the set of functions $f(n)$ such that there are a constant $c > 0$ and $n_0 > 0$ such that:

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

The O-Notation

- For a given function $g(n)$, we denote by $O(g(n))$ the set of functions $f(n)$ such that there are a constant $c > 0$ and $n_0 > 0$ such that:

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

- If $f(n) \in O(g(n))$ then $g(n)$ is an asymptotically upper bound for $f(n)$

The O-Notation

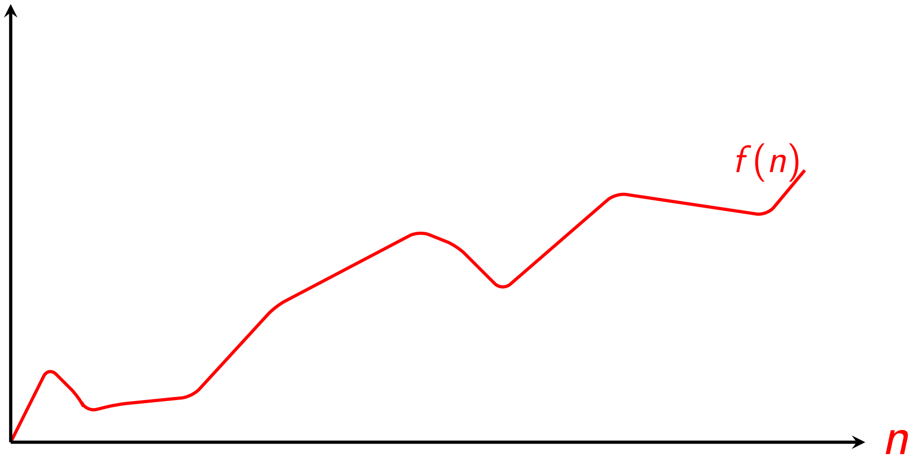
- For a given function $g(n)$, we denote by $O(g(n))$ the set of functions $f(n)$ such that there are a constant $c > 0$ and $n_0 > 0$ such that:

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

- If $f(n) \in O(g(n))$ then $g(n)$ is an asymptotically upper bound for $f(n)$

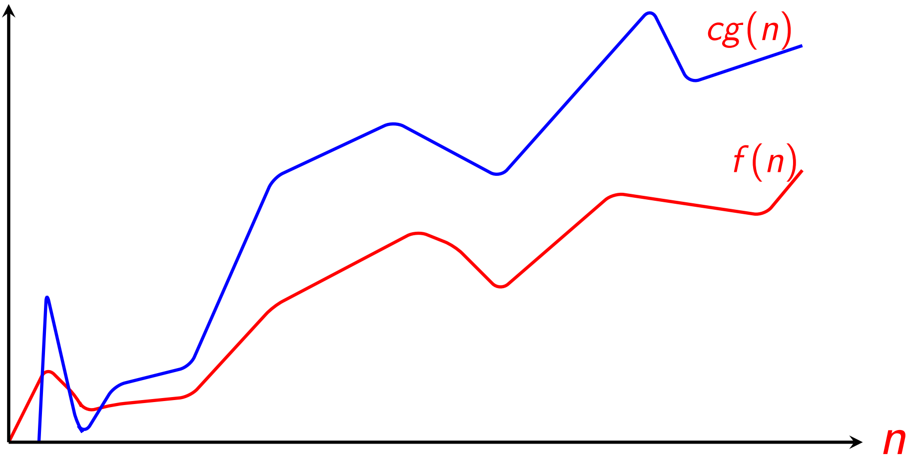
- We often write $f(n) = O(g(n))$ to denote that $f(n) \in O(g(n))$ (Should be read "$f(n)$ is $O(g(n))$".)

O-Notation



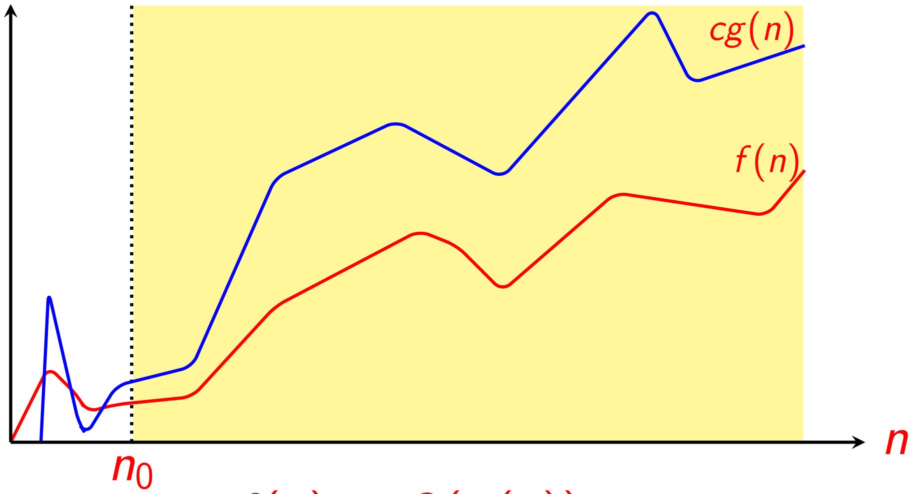
$$f(n) = O(g(n))$$

O-Notation



$$f(n) = O(g(n))$$

O-Notation



$$f(n) = O(g(n))$$

Examples

- $5n^2 + 3n + 6 = O(n^2)$
- $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0 = O(n^\ell)$ for any $\ell \geq k$
- $log(n) = O(n)$
- $n = O(2^n)$

The Ω-Notation

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions $f(n)$ such that there are a constant $c > 0$ and $n_0 > 0$ such that:

$$0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0$$

The Ω-Notation

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions $f(n)$ such that there are a constant $c > 0$ and $n_0 > 0$ such that:

$$0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0$$

- If $f(n) \in \Omega(g(n))$ then $g(n)$ is an asymptotically lower bound for $f(n)$

The Ω-Notation

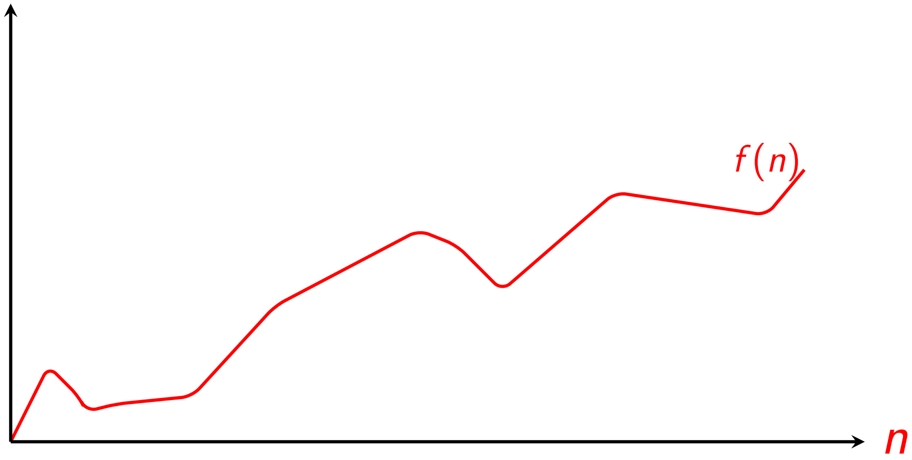
- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions $f(n)$ such that there are a constant $c > 0$ and $n_0 > 0$ such that:

$$0 \le c \cdot g(n) \le f(n) \text{ for all } n \ge n_0$$

- If $f(n) \in \Omega(g(n))$ then $g(n)$ is an asymptotically lower bound for $f(n)$

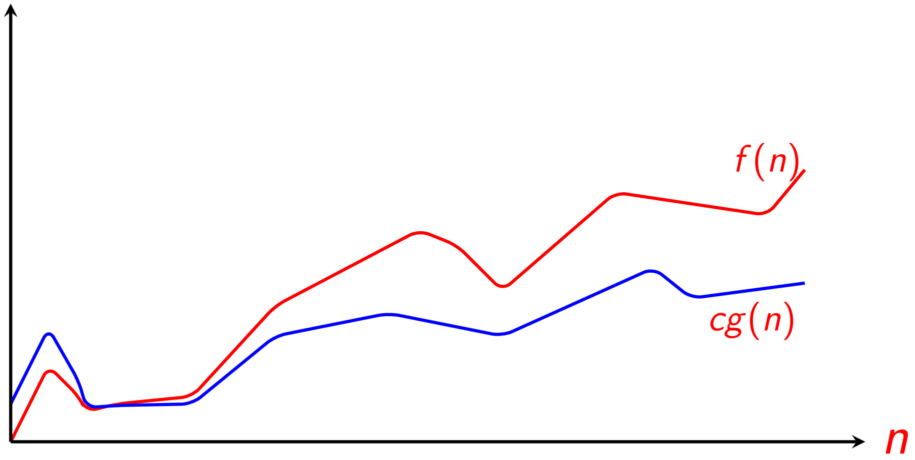
- We often write $f(n) = \Omega(g(n))$ to denote that $f(n) \in \Omega(g(n))$

Ω-Notation



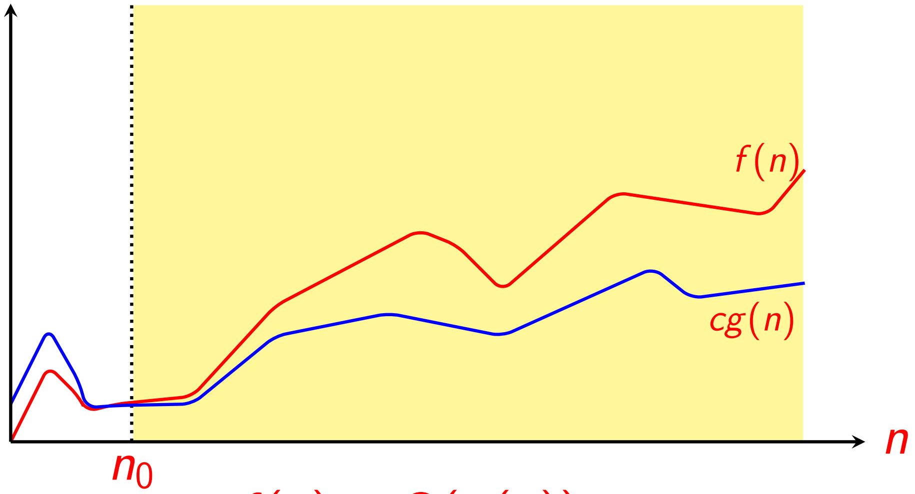
$$f(n) = \Omega(g(n))$$

Ω-Notation



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

Ω-Notation



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

Examples

- $5n^2 - 3n - 6 = \Omega(n^2)$
- $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0 = \Omega(n^\ell)$ for any $\ell \leq k$
- $n = \Omega(log(n))$
- $2^n = \Omega(n)$

The Θ-Notation

- For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions $f(n)$ such that there are constants $c_1 > 0, c_2 > 0$, and $n_0 > 0$ such that

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0$$

The Θ-Notation

- For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions $f(n)$ such that there are constants $c_1 > 0, c_2 > 0$, and $n_0 > 0$ such that

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0$$

- If $f(n) \in \Theta(g(n))$ then $g(n)$ is an asymptotically tight bound for $f(n)$

The Θ-Notation

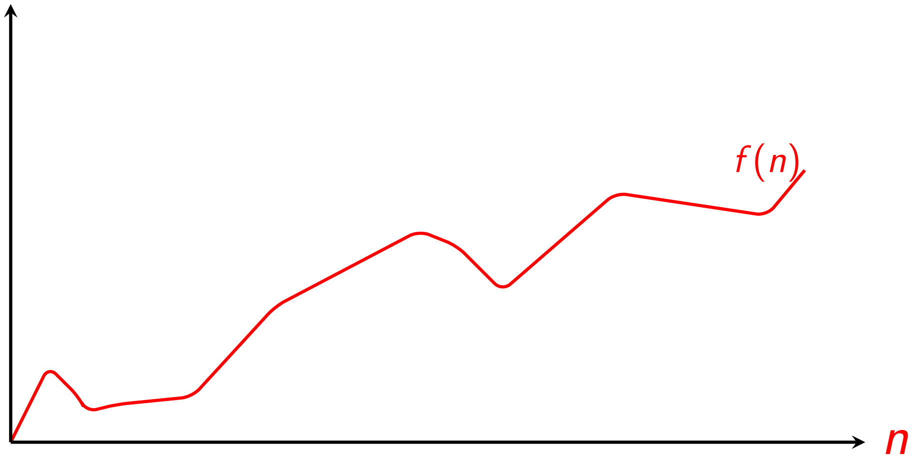
- For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions $f(n)$ such that there are constants $c_1 > 0, c_2 > 0$, and $n_0 > 0$ such that

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0$$

- If $f(n) \in \Theta(g(n))$ then $g(n)$ is an asymptotically tight bound for $f(n)$

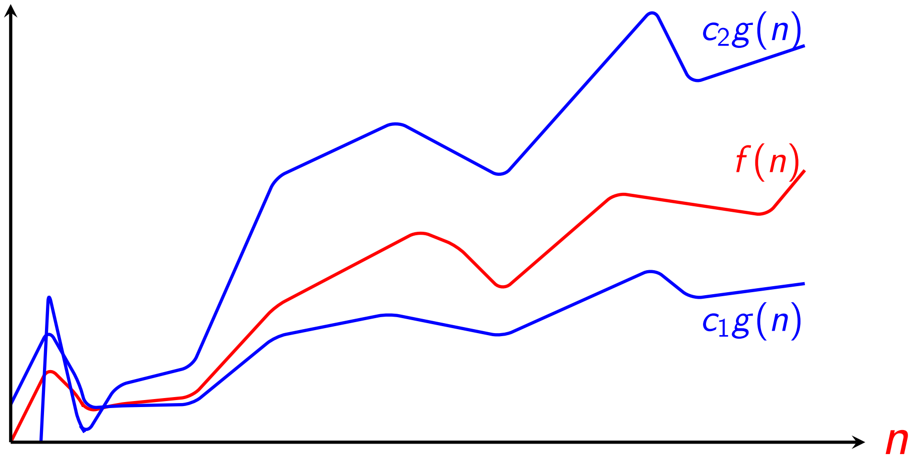
- We often write $f(n) = \Theta(g(n))$ to denote that $f(n) \in \Theta(g(n))$

Θ – *Notation*

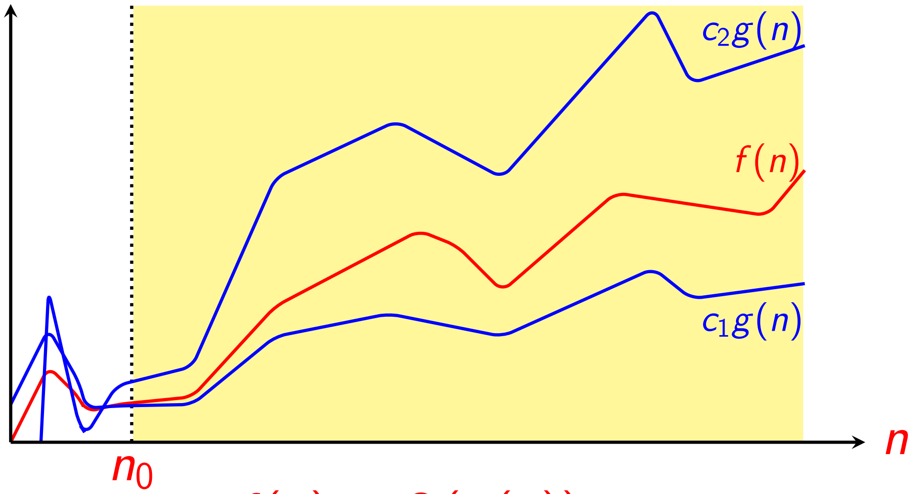


$$f(n) = \Theta(g(n))$$

Θ – *Notation*



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



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

Examples

- $5n^2 - 3n - 6 = \Theta(n^2)$
- $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0 = \Theta(n^k)$
- For any constant function $f(n)$ (i.e., there is $c > 0$ such that $f(n) = c$ for all n), we have $f(n) = \Theta(1)$
- $log(n) \neq \Theta(n)$
- $n \neq \Theta(2^n)$
- The Worst-Case complexity of Insertion Sort algorithm is $\Theta(n^2)$
- The Best-case complexity of Insertion Sort algorithm is $\Theta(n)$

Some Properties

Let $X \in \{O, \Theta, \Omega\}$.

- If $f(n) \in X(g(n))$ and $g(n) \in X(h(n))$ then $f(n) \in X(h(n))$
- $f(n) = X(f(n))$
- $f(n) \in \Omega(g(n))$ iff $g(n) \in O(f(n))$
- $f(n) \in \Theta(g(n))$ iff $g(n) \in \Theta(f(n))$
- $f(n) \in \Theta(g(n))$ iff $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

Algorithmic Complexity

- We use the asymptotic notations to characterize the complexity of algorithms.
- Always specify on which complexity we talk: Best, Worst or Average-Case complexity
- The O-notation is the most used one.

Complexity of a Problem

- Asymptotic notations can characterize problem complexity:
  - A problem is $O(g(n))$ if there is an algorithm $O(g(n))$ to solve it.
  - A problem is $\Omega(g(n))$ if any algorithm solving it is $\Omega(g(n))$.
  - A problem is $\Theta(g(n))$ if it is $O(g(n))$ and $\Omega(g(n))$.

Different Complexity Classes

The O-notation groups functions into set of classes, for example:

- Sub-linear functions:
  - Constant functions: $O(1)$
  - Logarithmic functions: $O(log(n))$
- Polynomial functions:
  - Linear functions: $O(n)$
  - Super-linear functions: $O(nlog(n))$
  - quadratic functions: $O(n^2)$
- Exponential functions: $O(c^n)$ for a given constant $c > 1$
- Factorial functions: $O(n!)$