

COMP20007 Design of Algorithms

Sorting - Part 1

Daniel Beck

Lecture 11

Semester 1, 2020

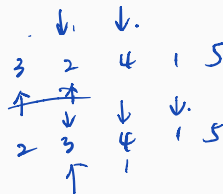
Insertion Sort

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $A[j+1] < A[j]$  do  
      SWAP( $A[j+1], A[j]$ )  
       $j \leftarrow j-1$ 
```

Insertion Sort

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $A[j+1] < A[j]$  do  
      SWAP( $A[j+1], A[j]$ )  
       $j \leftarrow j-1$ 
```

- Decrease-And-Conquer algorithm.



Insertion Sort

```
function INSERTIONSORT( $A[0..n - 1]$ )  
  for  $i \leftarrow 1$  to  $n - 1$  do  
     $j \leftarrow i - 1$   
    while  $j \geq 0$  and  $A[j + 1] < A[j]$  do  
      SWAP( $A[j + 1], A[j]$ )  
       $j \leftarrow j - 1$ 
```

- Decrease-And-Conquer algorithm.
- The idea behind Insertion Sort is recursive, but the code given here, using iteration, is preferable to the recursive version.

Insertion Sort - Properties

Questions!

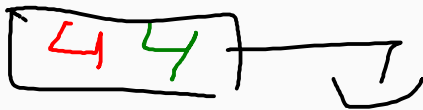
Insertion Sort - Properties

Questions!

- In-place? (does it require extra memory?)

Insertion Sort - Properties

Questions!



- In-place? (does it require extra memory?)
- Stable? (preserves original order of inputs?)

44

Insertion Sort - Properties

Questions!

- In-place? (does it require extra memory?)
- Stable? (preserves original order of inputs?)

```
function INSERTIONSORT( $A[0..n - 1]$ )  
  for  $i \leftarrow 1$  to  $n - 1$  do  
     $j \leftarrow i - 1$   
    while  $j \geq 0$  and  $A[j + 1] < A[j]$  do  
      SWAP( $A[j + 1], A[j]$ )  
       $j \leftarrow j - 1$ 
```


Insertion Sort - Properties

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $A[j+1] < A[j]$  do  
      SWAP( $A[j+1], A[j]$ )  
       $j \leftarrow j-1$ 
```

Insertion Sort - Properties

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $A[j+1] < A[j]$  do  
      SWAP( $A[j+1], A[j]$ )  
       $j \leftarrow j-1$ 
```

- In-place?

Insertion Sort - Properties

function INSERTIONSORT($A[0..n-1]$)

for $i \leftarrow 1$ **to** $n-1$ **do**

$j \leftarrow i-1$

while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**

 SWAP($A[j+1], A[j]$) $\rightarrow v \leftarrow A[j]$

$j \leftarrow j-1$

$A[j] \leftarrow A[j+1]$

$A[j+1] \leftarrow v$

- In-place? Yes! (may need additional $O(1)$ memory)

Insertion Sort - Properties

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $A[j+1] < A[j]$  do  
      SWAP( $A[j+1], A[j]$ )  
       $j \leftarrow j-1$ 
```

- In-place? Yes! (may need additional $O(1)$ memory)
- Stable?

Insertion Sort - Properties

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $j \leftarrow i-1$   
    while  $j > 0$  and  $A[j+1] < A[j]$  do  
      SWAP( $A[j+1], A[j]$ )  
       $j \leftarrow j-1$ 
```

- In-place? Yes! (may need additional $O(1)$ memory)
- Stable? Yes! (**local**, adjacent swaps ensure stability)

Insertion Sort - Properties

```
function INSERTIONSORT( $A[0..n - 1]$ )  
  for  $i \leftarrow 1$  to  $n - 1$  do  
     $j \leftarrow i - 1$   
    while  $j \geq 0$  and  $A[j + 1] < A[j]$  do  
      SWAP( $A[j + 1]$ ,  $A[j]$ )  
       $j \leftarrow j - 1$ 
```

- In-place? Yes! (may need additional $O(1)$ memory)
- Stable? Yes! (**local**, adjacent swaps ensure stability)

Compare with Selection Sort:

- Also in-place.
- **Not stable.** (swaps are not **local**)

Insertion Sort - Complexity

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $A[j+1] < A[j]$  do  
      SWAP( $A[j+1], A[j]$ )  
       $j \leftarrow j-1$ 
```

Insertion Sort - Complexity

```
function INSERTIONSORT( $A[0..n - 1]$ )  
  for  $i \leftarrow 1$  to  $n - 1$  do  
     $j \leftarrow i - 1$   
    while  $j \geq 0$  and  $A[j + 1] < A[j]$  do  
      SWAP( $A[j + 1], A[j]$ )  
       $j \leftarrow j - 1$ 
```

- Worst case?
- Best case?

Insertion Sort - Worst case

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $A[j+1] < A[j]$  do  
      SWAP( $A[j+1], A[j]$ )  
       $j \leftarrow j-1$ 
```

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$$

Insertion Sort - Best case

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $A[j+1] < A[j]$  do  
      SWAP( $A[j+1], A[j]$ )  
       $j \leftarrow j-1$ 
```

$$\sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

Insertion Sort - Average case

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $A[j+1] < A[j]$  do  
      SWAP( $A[j+1], A[j]$ )  
       $j \leftarrow j-1$ 
```

$\Theta(n+v)$
 $v: \# \text{ INV. PAIRS}$
 $\rightarrow \text{INV. PAIRS}$

• BEST: $0 \text{ INV. PAIRS} = \Theta(n)$

• WORST: $\frac{n(n-1)}{2} \text{ INV. PAIRS} = \Theta(n + \frac{n(n-1)}{2}) = \Theta(n^2)$

• AVG: $\frac{n(n-1)}{4} \rightarrow \Theta(n + \frac{n(n-1)}{4}) = \Theta(n^2)$

Insertion Sort - Complexity

- Worst case: $\Theta(n^2)$
- Best case: $\Theta(n)$
- Average case: $\Theta(n^2)$

Insertion Sort - Complexity

- Worst case: $\Theta(n^2)$
- Best case: $\Theta(n)$
- Average case: $\Theta(n^2)$

Compare with Selection Sort, which is input-insensitive: best, average and worst case complexity is $\Theta(n^2)$

Insertion Sort - Complexity

- Worst case: $\Theta(n^2)$
- Best case: $\Theta(n)$
- Average case: $\Theta(n^2)$

Compare with Selection Sort, which is input-insensitive: best, average and worst case complexity is $\Theta(n^2)$

Insight

In many cases, real-world data is **already partially sorted**.

This makes Insertion Sort a powerful sorting algorithm in practice, particularly useful for small arrays (up to hundreds of elements).

Insertion Sort - A faster version

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $v \leftarrow A[i]$   
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $v < A[j]$  do  
       $A[j+1] \leftarrow A[j]$   
       $j \leftarrow j-1$   
     $A[j+1] \leftarrow v$ 
```

Insertion Sort - A faster version

```
function INSERTIONSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 1$  to  $n-1$  do  
     $v \leftarrow A[i]$   
     $j \leftarrow i-1$   
    while  $j \geq 0$  and  $v < A[j]$  do  
       $A[j+1] \leftarrow A[j]$   
       $j \leftarrow j-1$   
     $A[j+1] \leftarrow v$ 
```

This is the version presented in the Levitin book.

Insertion Sort - Sentinels

- Assume the domain is bounded from below.

Insertion Sort - Sentinels

- Assume the domain is bounded from below.
- There is a **minimal** element *min*.

Insertion Sort - Sentinels

- Assume the domain is bounded from below.
- There is a **minimal** element *min*.
- Assume a **free cell** to the left of $A[0]$

Insertion Sort - Sentinels

- Assume the domain is bounded from below.
- There is a **minimal** element *min*.
- Assume a **free cell** to the left of $A[0]$

Insertion Sort can be made faster by using a *min sentinel* in that cell ($A[-1]$) and change the test from

$$~~j \geq 0~~ \text{ and } v < A[j]$$

to just

$$v < A[j]$$

Insertion Sort - Sentinels

- Assume the domain is bounded from below.
- There is a **minimal** element *min*.
- Assume a **free cell** to the left of $A[0]$

Insertion Sort can be made faster by using a *min sentinel* in that cell ($A[-1]$) and change the test from

$$j \geq 0 \text{ and } v < A[j]$$

to just

$$v < A[j]$$

For this reason, extreme array cells (such as $A[0]$ in C, and/or $A[n + 1]$) are sometimes left free deliberately, so that they can be used to hold sentinels; only $A[1]$ to $A[n]$ hold proper data.

Sorting - Practical Implementations

- C - Quicksort (fastest)
- C++ - Introsort (a variant of Quicksort)
- Javascript/Mozilla: Mergesort (stable)
- Python: Timsort (very roughly, a mix of Mergesort and Insertion Sort, stable)
- Linux Kernel: Heapsort (low memory consumption, guaranteed $\Theta(n \log n)$ worst case performance: important for security reasons)