COMP20007 Design of Algorithms

Sorting - Part 1

Daniel Beck

Lecture 11

Semester 1, 2020

Insertion Sort

function $\textsc{InsertionSort}(A[0..n-1])$

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

    $j \leftarrow i - 1$

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

      $\text{SWAP}(A[j+1], A[j])$

      $j \leftarrow j - 1$

Insertion Sort

function $\textsc{InsertionSort}(A[0..n-1])$

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

    $j \leftarrow i - 1$

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

      $\mathrm{SWAP}(A[j+1], A[j])$

      $j \leftarrow j - 1$

- Decrease-And-Conquer algorithm.

3 2 4 1 5

2 3 4 1 5

1

Insertion Sort

function $\textsc{InsertionSort}(A[0..n-1])$
    for $i \leftarrow 1$ **to** $n-1$ **do**
        $j \leftarrow i - 1$
        while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
            $\text{SWAP}(A[j+1], A[j])$
            $j \leftarrow j - 1$

- 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!

4 4

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

4 4

Insertion Sort - Properties

Questions!

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

function $\textsc{InsertionSort}(A[0..n-1])$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i - 1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      $\text{SWAP}(A[j+1], A[j])$
      $j \leftarrow j - 1$

Insertion Sort - Properties

```
function InsertionSort(A[0..n − 1])
    for i ← 1 to n − 1 do
        j ← i − 1
        while j ≥ 0 and A[j + 1] < A[j] do
            SWAP(A[j + 1], A[j])
            j ← j − 1
```

Insertion Sort - Properties

function $\textsc{InsertionSort}(A[0..n-1])$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      $\mathrm{SWAP}(A[j+1], A[j])$
      $j \leftarrow j-1$

- 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 $\textsc{InsertionSort}(A[0..n-1])$
    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 $\textsc{InsertionSort}(A[0..n-1])$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i - 1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      $\text{SWAP}(A[j+1], A[j])$
      $j \leftarrow j - 1$

- 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 $\textsc{InsertionSort}(A[0..n-1])$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      $\text{SWAP}(A[j+1], A[j])$
      $j \leftarrow j-1$

Insertion Sort - Complexity

function $\textsc{InsertionSort}(A[0..n-1])$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      $\text{SWAP}(A[j+1], A[j])$
      $j \leftarrow j-1$

- Worst case?
- Best case?

Insertion Sort - Worst case

function $\textsc{InsertionSort}(A[0..n-1])$

  $\boxed{\textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do}}$

    $j \leftarrow i - 1$

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

      $\text{SWAP}(A[j+1], A[j])$

      $j \leftarrow j - 1$

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

Insertion Sort - Best case

function $\textsc{InsertionSort}(A[0..n-1])$

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

        ~~$j \leftarrow i - 1$~~

        while $j \geq 0$ **and** $\boxed{A[j+1] < A[j]}$ **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**
      $\boxed{\text{SWAP}(A[j+1], A[j])}$ → INV. PAIRS
      $j \leftarrow j - 1$

$\Theta(n + v)$

v: # INV. PAIRS

- BEST: 0 INV. PAIRS $= \boxed{\Theta(n)}$
- WORST: $\frac{n(n-1)}{2}$ INV. PAIRS $= \Theta\left(n + \frac{n(n-1)}{2}\right) = \Theta(n^2)$
- AVG: $\frac{n(n-1)}{4} = \Theta\left(n + \frac{n(n-1)}{4}\right) = \Theta(n^2)$

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-insensitve: 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-insensitve: 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 ← 1 to n − 1 do
        v ← A[i]
        j ← i − 1
        while j ≥ 0 and v < A[j] do
            A[j + 1] ← A[j]
            j ← j − 1
        A[j + 1] ← 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

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

to just

$$\boxed{v < A[j]}$$

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

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

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 $\Theta(nlogn)$ worst case performance: important for security reasons)