How can we sort a deck of cards?

wikipedia

InsertionSort

- Video: http://www.youtube.com/watch?v=cFoLbjGUKWs

- What is the algorithm?

6 5 3 1 8 7 2 4

wikipedia

InsertionSort – Specification

First let's make sure we know what we want to do:

- The **specification** of the algorithm
- AKA the **problem we are trying to solve**

Input: sequence of **n** numbers

$$A=(a_1, \dots a_n)$$

Output: a permutation (reordering) of the input

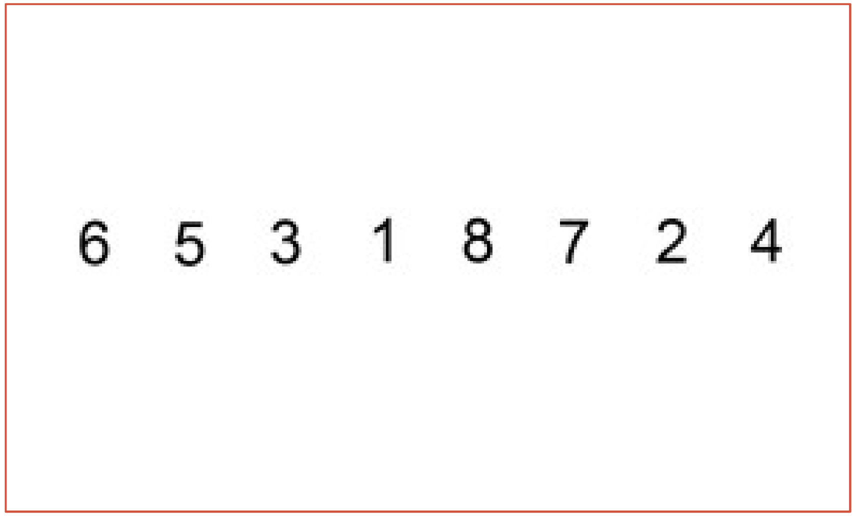
$$(a'_1, \dots a'_n)$$

Such that

$$a'_1 \le a'_2 \le \dots \le a'_n$$

InsertionSort

- Algorithm (in English)
  1. Start from 1st element of the array (optimisation: start from 2nd)
  2. Shift element back until you find a smaller element – maintain the array from 0 to (current position) sorted.
  3. Continue to next element
  4. Repeat (2) and (3) until the end of the array



InsertionSort

- Algorithm (in pseudocode)

```
for (j = 1; j<A.length; j++) {
 //shift A[j] into the sorted A[0..j-1]
  i=j-1
  while i>=0 and A[i]>A[i+1] {
   swap A[i], A[i+1]
   i=i-1
}}
return A
```

6 5 3 1 8 7 2 4

InsertionSort

- Algorithm (in Java)
  - *left as an exercise.*

We are skipping the first number A[0]

A.length == n

```
for (j = 1; j<A.length; j++) {
 //shift A[j] into the sorted A[0..j-1]
  i=j-1
  while i>=0 and A[i]>A[i+1] {
   swap A[i], A[i+1]
   i=i-1
}}
return A
```

```
for (j = 1; j<A.length; j++) {
 //shift A[j] into the sorted A[0..j-1]
  i=j-1
  while i>=0 and A[i]>A[i+1] {
   swap A[i], A[i+1]
   i=i-1
}}
return A
```

How to calculate runtime performance?

Calculating Approximate Performance

- Assume basic operations cost 1
- Then, choose one:
  - Count all operations (if uncertain, it's the safer choice)
  - Count only some operations (which ones?)
- Then, keep only highest-order terms (~ notation)

		cost	no of times
for (j = 1; j<A.length; j++) {	1		
//shift A[j] into the sorted A[0..j-1]			
i=j-1		1	
while i>=0 **and** A[i]>A[i+1] {	1		
swap A[i], A[i+1]	1		
i=i-1		1	
}}			
return A		1	

Method of Calculations

- Assume basic operations cost 1
- Then, choose one:
  - Count all operations (if uncertain, it's the safer choice)
  - Count only some operations (which ones?)
    - We will count only **array comparisons** and **array swaps**
    - It is equivalent to counting **array accesses**
- Then, keep only highest-order terms (~ notation)

	cost	no of times
for (j = 1; j<A.length; j++) {		
//shift A[j] into the sorted A[0..j-1]		
i=j-1		
while i>=0 **and** A[i]>A[i+1] {	1	
swap A[i], A[i+1]	1	
i=i-1		
}}		
return A		

```
                                                    cost    no of times
for (j = 1; j<A.length; j++) {
 //shift A[j] into the sorted A[0..j-1]
 i=j-1
 while i>=0 and A[i]>A[i+1] {                 1
  swap A[i], A[i+1]                           1
  i=i-1
}}
return A
```

The input is A, an array of size N.
Besides the size N, is the “no of times” dependent of the **actual** ints in A?

Best Case

	cost	no of times
`for (j = 1; j<A.length; j++) {`		
`//shift A[j] into the sorted A[0..j-1]`		
`i=j-1`		
`while i>=0 and A[i]>A[i+1] {`	1	
`swap A[i], A[i+1]`	1	
`i=i-1`		
`}}`		
`return A`		

In the best case the array is **already sorted**.

Best Case

```
for (j = 1; j<A.length; j++) {
 //shift A[j] into the sorted A[0..j-1]
 i=j-1
 while i>=0 and A[i]>A[i+1] {
  swap A[i], A[i+1]
  i=i-1
}}
return A
```

	cost	no of times
while i>=0 and A[i]>A[i+1] {	1	$(1+1+\ldots+1)_{n-1 \text{ times}}$
swap A[i], A[i+1]	1	0

In the best case the array is already sorted.

The time (as a function of the input size n):

$$T(n) = n - 1$$

Worst Case

	cost	no of times
`for (j = 1; j<A.length; j++) {`		
`//shift A[j] into the sorted A[0..j-1]`		
`i=j-1`		
`while i>=0 and A[i]>A[i+1] {`	1	
`swap A[i], A[i+1]`	1	
`i=i-1`		
`}}`		
`return A`		

Worst Case

	cost	no of times
`for (j = 1; j<A.length; j++) {`		
`//shift A[j] into the sorted A[0..j-1]`		
`i=j-1`		
`while i>=0 and A[i]>A[i+1] {`	1	
`swap A[i], A[i+1]`	1	
`i=i-1`		
`}}`		
`return A`		

In the worst case the array is in <u>reverse sorted order</u>.

Worst Case

	cost	no of times
`for (j = 1; j<A.length; j++) {`		
`//shift A[j] into the sorted A[0..j-1]`		
`i=j-1`		
`while i>=0 and A[i]>A[i+1] {`	1	2+…+n
`swap A[i], A[i+1]`	1	1+…+(n-1)
`i=i-1`		
`}}`		
`return A`		

In the worst case the array is in <u>reverse sorted order</u>.

$$T(n) = \Sigma_{i=2..n}(i) + \Sigma_{i=1..n-1}(i) \quad = \quad \Sigma_{i=1..n}(i) - 1 + \Sigma_{i=1..n-1}(i)$$

$$= (n(n+1)/2 \; - \; 1) + 2n(n-1)/2$$

$$\sim (5/2)n^2$$

Closed-form Expressions for Some Commonly Encountered Series

$\sum_{n=0}^{N-1} a^n = \frac{1-a^N}{1-a}$	$\sum_{n=0}^{\infty} a^n = \frac{1}{1-a} \qquad \lvert a\rvert < 1$
$\sum_{n=0}^{N-1} na^n = \frac{(N-1)a^{N+1} - Na^N + a}{(1-a)^2}$	$\sum_{n=0}^{\infty} na^n = \frac{a}{(1-a)^2} \qquad \lvert a\rvert < 1$
$\sum_{n=0}^{N-1} n = \frac{1}{2}N(N-1)$	$\sum_{n=0}^{N-1} n^2 = \frac{1}{6}N(N-1)(2N-1)$

Average Case

	cost	no of times
`for (j = 1; j<A.length; j++) {`		
`//shift A[j] into the sorted A[0..j-1]`		
`i=j-1`		
`while i>=0 and A[i]>A[i+1] {`	1	(2+…+n)/2
`swap A[i], A[i+1]`	1	(1+…+(n-1))/2
`i=i-1`		
`}}`		
`return A`		

In the worst case the array is in reverse sorted order.

$T(n) = \ldots$(same calculations)$\ldots \sim (5/4)n^2$

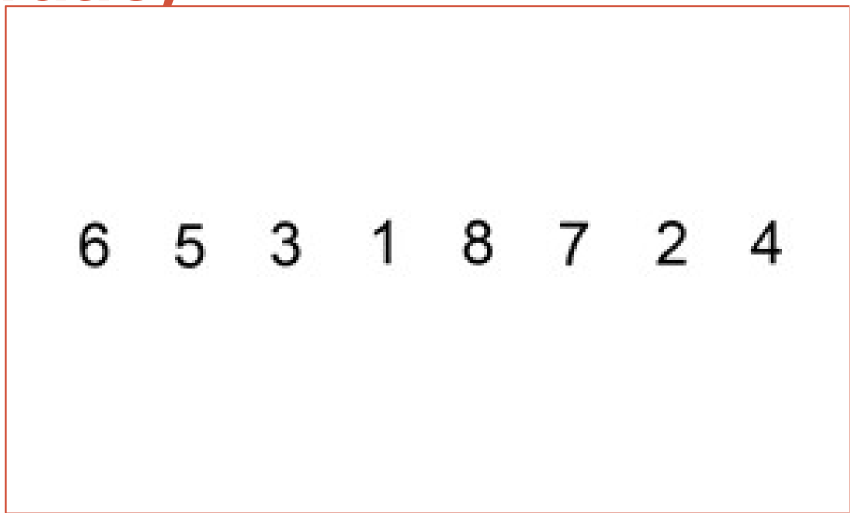
InsertionSort – three types of analyses

- With best case input of size n:
  - $T(n) \sim 3\,n$
- with the worst case input of size n:
  - $T(n) \sim (5/2)n^2$
- with average input of size n:
  - $T(n) \sim (5/4)n^2$

6 5 3 1 8 7 2 4

InsertionSort – three types of analyses

- With best case input of size n:
  - $T(n) \sim 3\,n$
  - Order of growth: **n (linear)**
- with the worst case input of size n:
  - $T(n) \sim (5/2)n^2$
  - Order of growth: **n^2 (quadratic)**
- with average input of size n:
  - $T(n) \sim (3/4)n^2$
  - Order of growth: **n^2**



InsertionSort

Which case analysis would you pick?

- With best case input of size n:
  - T(n) ~ 3 n
  - Order of growth: **n (linear)**
- with the worst case input of size n:
  - T(n) ~ (5/2)n^2
  - Order of growth: **n^2 (quadratic)**
- with average input of size n:
  - T(n) ~ (3/4)n^2
  - Order of growth: **n^2**

6 5 3 1 8 7 2 4

InsertionSort

- With best case input of size n:
  - $T(n) \sim 3\,n$
  - Order of growth: **n (linear)**
- with the worst case input of size n:
  - $T(n) \sim (5/2)n^2$
  - Order of growth: **n^2 (quadratic)**
- with average input of size n:
  - $T(n) \sim (3/4)n^2$
  - Order of growth: **n^2**

We will focus on **Worst Case:**
"InsertionSort is quadratic"
=
"The worst case running time of InsertionSort is quadratic"
=
"InsertionSort with the worst case input of size N runs in $\Theta(N^2)$ time"

6 5 3 1 8 7 2 4

InsertionSort

- With best case input of size n:
  - T(n) ~ 3 n
  - Order of growth: **n (linear)**
- with the worst case input of size n:
  - T(n) ~ $(5/2)n^2$
  - Order of growth: **n^2 (quadratic)**
- with average input of size n:
  - T(n) ~ $(3/4)n^2$
  - Order of growth: **n^2**

Sometimes we will talk about on **Average Case:**
"The average case running time of InsertionSort is quadratic"
=
"InsertionSort with the average case input of size N runs in $\Theta(N^2)$ time"