Data Sorting: Insertion sort

Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures

1. Ordering

2. Data sorting

3. Efficiency of comparison-based sorting

4. Insertion sort

Relations, Partial Order, and Linear Order

A **relation on a set** S is a set R of ordered pairs of elements of S, i.e. a subset, $R \subseteq S \times S$ of the set, $S \times S$, of all pairs of these elements.

- An ordered pair $(x, y) \in R$ means the element y relates to x.
  - The relation is denoted sometimes as yRx.
- An important type of relation: a **partial order**, which is **reflexive**, **antisymmetric**, and **transitive**.

Main features of the partial order:

Reflexivity: xRx for every $x \in S$.

Antisymmetry: If xRy and yRx then $x = y$ for every $x, y \in S$.

Transitivity: If xRy and yRz then xRz for every $x, y, z \in S$.

- A **linear order** (or a **total order**) is a partial order, such that every pair of elements is related (i.e. $R = S \times S$).

Examples of Linear Order Relations

1. S – the set of real numbers; R – the usual "less than or equal to" relation, $x \leq y$, for all pairs of numbers.
   - For every $x \in S$, $x \leq x$.
   - For every $x, y \in S$, if $x \leq y$ and $y \leq x$ then $x = y$.
   - For every $x, y, z \in S$, if $x \leq y$ and $y \leq z$ then $x \leq z$.
2. S – the set of Latin letters:

$$S = \{q, w, e, r, t, y, u, i, o, p, a, s, d, f, g, h, j, k, l, z, x, c, v, b, n, m\}$$

and R – the alphabetic relation for all pairs of letters:

$$R = \left\{ \begin{array}{ccccccccc} (a,a) & (a,b) & (a,c) & (a,d) & (a,e) & (a,f) & \ldots & (a,y) & (a,z) \\ & (b,b) & (b,c) & (b,d) & (b,e) & (b,f) & \ldots & (b,y) & (b,z) \\ & & (c,c) & (c,d) & (c,e) & (c,f) & \ldots & (c,y) & (c,z) \\ & & & & & & \ldots & & \\ & & & & & & & (y,y) & (y,z) \\ & & & & & & & & (z,z) \end{array} \right\}$$

Data Ordering: Numerical Order

Ordering relation places each pair α, β of countable data items in a fixed order denoted as (α, β) or $\langle \alpha, \beta \rangle$.

- **Order notation**: $\alpha \leq \beta$ (*less than or equal to*).
- **Countable item**: labelled by a specific **integer key**.

Comparable objects in Java and Python: if an object can be **less than**, **equal to**, or **greater than** other object:

Java:	`object.compareTo( other_object )`	$< 0, = 0, > 0$
Python:	`object.__cmp__(self,other)`	$< 0, = 0, > 0$

Numerical order - on any set of numbers by values of elements:

$$5 \leq 5 \leq 6.45 \leq 22.79 \leq \ldots \leq 1056.32$$

Alphabetical and Lexicographic Orders

Alphabetical order - on a set of letters by their position in an alphabet:

$$a \le b \le c \le d \le e \le f \le g \le h \le i \le \ldots \le y \le z$$

Such ordering depends on the alphabet used: look into any bilingual dictionary...

Lexicographic order - on a set of strings (e.g. multi-digit numbers or words) by the first differing character in the strings:

$$\begin{array}{ccccccccc} 5456 & \le & 5457 & \le & 5500 & \le & 6100 & \le & \ldots \\ \text{pork} & \le & \text{ward} & \le & \text{word} & \le & \text{work} & \le & \ldots \end{array}$$

The characters are compared in line with their numerical or alphabetical order: look into any dictionary or thesaurus...

The Problem of Sorting

Rearrange an input list of **keys**, which can be compared using a total order $\leq$, into an output list such that key i precedes key j in the output list if $i \leq j$.

The key is often a data field in a larger object: rather than move such objects, a pointer from the key to the object is to be kept.

Sorting algorithm is **comparison-based** if the total order can be checked only by comparing the order $\leq$ of a pair of elements at a time.

- Sorting is **stable** if any two objects, having the same key in the input, appear in the same order in the output.
- Sorting is **in-place** if only a fixed additional memory space is required independently of the input size.

Efficiency of Comparison-Based Sorting

No other information about the keys, except of only their order relation, can be used.

The running time of sorting is usually dominated by two elementary operations: a **comparison** of two items and a **move** of an item.

Every sorting algorithm in this course makes at most constant number of moves for each comparison.

- Asymptotic running time in terms of elementary operations is determined by the number of comparisons.
- Time for a data move depends on the list implementation.
- Sorting makes sense only for linear data structures.

The efficiency of a particular sorting algorithm depends on the number of items to be sorted; place of sorting (fast internal or slow external memory); to what extent data items are presorted, etc.

Sorting with Insertion Sort

Insertion sort (the same scheme also in Selection Sort and Bubble Sort)

- Split an array into a unordered and ordered parts:

Head (ordered) Tail (unordered)

$$\boxed{a_0, a_1, \ldots, a_{i-1}}\boxed{a_i, a_{i+1}, \ldots, a_{n-1}}$$

- Sequentially contract the unordered part, one element per stage:

At the beginning of each stage $i = 1, \ldots, n-1$:

i ordered and $n - i$ unordered elements.

i	The array to be sorted							C_i	M_i
	44	13	35	18	15	10	20		
1	13	44	35	18	15	10	20	1	1
2	13	35	44	18	15	10	20	2	1
3	13	18	35	44	15	10	20	3	2

C_i and M_i – numbers of comparisons and moves at stage i, respectively.

Python Code of Insertion Sort

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheInsertionSort.html

```
# Insertion sort of an input array a of size n
# Each leftmost unordered a[i] is compared right-to-left to the already
# ordered elements a[i-1],...,a[0], being right-shifted to free place
# between them for insertion of the element a[i]

def insertionSort( a )
  for i in range (1, len( a ) ) :
    tmp = a[ i ]                                  # pick a[i]
    k = i
    while k > 0 and tmp < a[ k - 1 ] :            # compare to a[k]
      a[ k ] = a[ k - 1]                          # shift a[k] right
      k = k - 1

    a[ k ] = tmp                                  # insert a[i]
```

Java Code of Insertion Sort

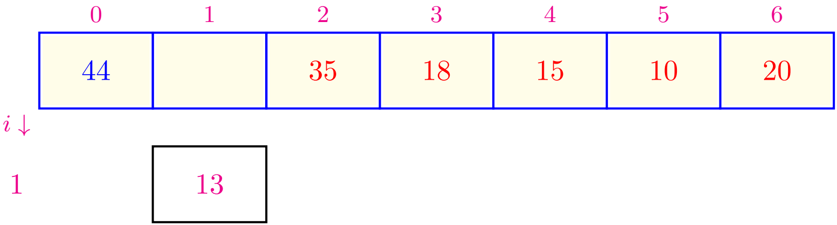
```
// Insertion sort of an input array a of size n
//
// Each leftmost unordered a[i] is compared right-to-left to the already
// ordered elements a[i-1],...,a[0], being right-shifted to free place
// between them for insertion of the element a[i]

public static void insertionSort( int [ ] a ) {
  for ( int i = 1; i < a.length; i++ ) {
    int tmp = a[ i ];                              // pick a[i]
    int k = i - 1;
    while ( k >= 0 && tmp < a[ k ] ) {   // compare to a[k]
      a[ k + 1 ] = a[ k ];                   // shift a[k] right
      k--;
    }
    a[ k + 1 ] = tmp;                            // insert a[i]
  }
}
```

Insertion Sort: Stages $i = 1, 2, 3$

0	1	2	3	4	5	6
44	13	35	18	15	10	20

$i \downarrow$

Insertion Sort: Stages $i = 1, 2, 3$



0	1	2	3	4	5	6
44		35	18	15	10	20

$i \downarrow$

1 13

Insertion Sort: Stages $i = 1, 2, 3$

	0	1	2	3	4	5	6
	44		35	18	15	10	20
$i \downarrow$							
1		13					

$13 < 44?$ → Comparison 1 for $i = 1$

Insertion Sort: Stages $i = 1, 2, 3$

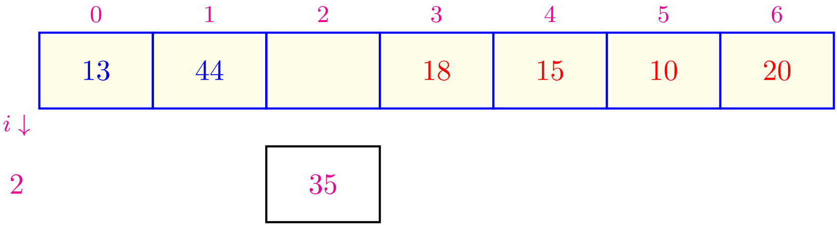
	0	1	2	3	4	5	6
		44	35	18	15	10	20
$i \downarrow$							
1		13					

Insertion Sort: Stages $i = 1, 2, 3$

0	1	2	3	4	5	6
13	44	35	18	15	10	20

$i \downarrow$

Insertion Sort: Stages $i = 1, 2, 3$



0	1	2	3	4	5	6
13	44		18	15	10	20

$i \downarrow$

2 — 35

Insertion Sort: Stages $i = 1, 2, 3$

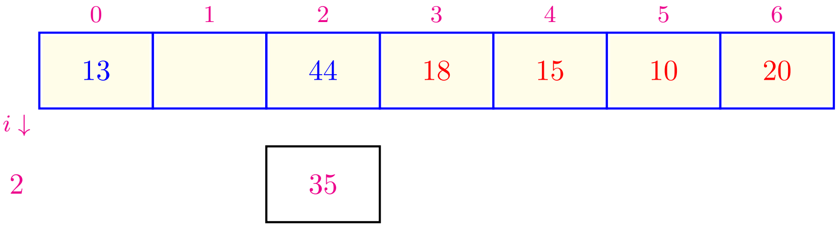
0	1	2	3	4	5	6
13	44		18	15	10	20

$i \downarrow$

2			35

$35 < 44?$ → Comparison 1 for $i = 2$

Insertion Sort: Stages $i = 1, 2, 3$

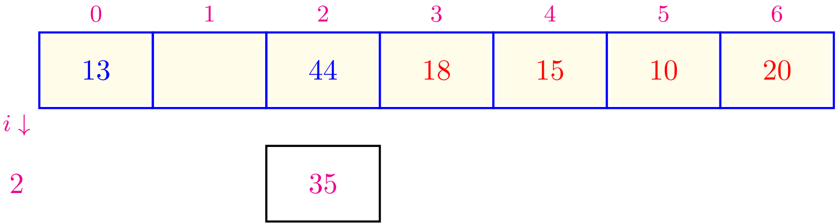


0	1	2	3	4	5	6
13		44	18	15	10	20

$i \downarrow$

i	0	1	2	3	4	5	6
2			35				

Insertion Sort: Stages $i = 1, 2, 3$



0	1	2	3	4	5	6
13		44	18	15	10	20

$35 < 13?$ → Comparison 2 for $i = 2$

Insertion Sort: Stages $i = 1, 2, 3$

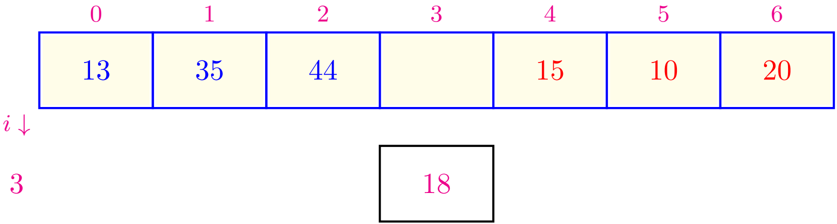
0	1	2	3	4	5	6
13	35	44	18	15	10	20

$i \downarrow$

Insertion Sort: Stages $i = 1, 2, 3$

	0	1	2	3	4	5	6
	13	35	44		15	10	20
$i\downarrow$							
3				18			

Insertion Sort: Stages $i = 1, 2, 3$



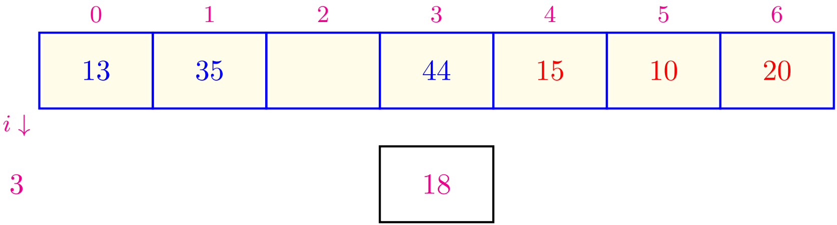
0	1	2	3	4	5	6
13	35	44		15	10	20

$i \downarrow$

3 18

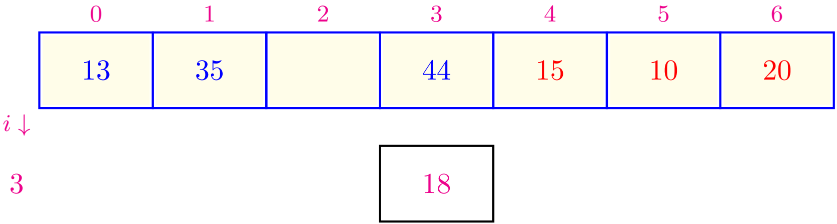
$18 < 44?$ → Comparison 1 for $i = 3$

Insertion Sort: Stages $i = 1, 2, 3$



	0	1	2	3	4	5	6
	13	35		44	15	10	20
$i \downarrow$							
3				18			

Insertion Sort: Stages $i = 1, 2, 3$



0	1	2	3	4	5	6
13	35		44	15	10	20

$i \downarrow$

3 18

$18 < 35?$ → Comparison 2 for $i = 3$

Insertion Sort: Stages $i = 1, 2, 3$

	0	1	2	3	4	5	6
	13		35	44	15	10	20
$i \downarrow$							
3				18			

Insertion Sort: Stages $i = 1, 2, 3$

0	1	2	3	4	5	6
13		35	44	15	10	20

$i \downarrow$

3 $\boxed{18}$

$18 < 13?$ → Comparison 3 for $i = 3$

Insertion Sort: Stages $i = 1, 2, 3$

0	1	2	3	4	5	6
13	18	35	44	15	10	20

i	C_i	M_i
1	1	1
2	2	1
3	3	2

Insertion Sort: Stage $i = 4$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.

3	13	18	35	44	15	10	20	*3*	*2*
4				*15*	44			$<$	$\rightarrow$
			15	35				$<$	$\rightarrow$
		15	18					$<$	$\rightarrow$
	15							$\geq$	
4	13	15	18	35	44	10	20	*4*	*3*
i								C_i	M_i

Insertion Sort: Stage $i = 4$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.

3	13	18	35	44	15	10	20	*3*	*2*
4				*15*	44			$<$	$\rightarrow$
			15	35				$<$	$\rightarrow$
		15	18					$<$	$\rightarrow$
	15							$\geq$	
4	13	15	18	35	44	10	20	*4*	*3*
i								C_i	M_i

Insertion Sort: Stage $i = 4$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.

3	13	18	35	44	15	10	20	*3*	*2*	
4				*15*	44			$<$	$\rightarrow$	●
			15	35				$<$	$\rightarrow$	●
		15	18					$<$	$\rightarrow$	●
	15							$\geq$		●
4	13	15	18	35	44	10	20	*4*	*3*	
i								C_i	M_i	

Insertion Sort: Stage $i = 4$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.

3	13	18	35	44	15	10	20	*3*	*2*	
4				*15*	44			$<$	$\rightarrow$	●
			15	35				$<$	$\rightarrow$	●
		15	18					$<$	$\rightarrow$	●
	15							$\geq$		●
4	13	15	18	35	44	10	20	*4*	*3*	
i								C_i	M_i	

Insertion Sort: Stage $i = 4$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.

3	13	18	35	44	15	10	20	*3*	*2*	
4				*15*	44			$<$	$\rightarrow$	●
			15	35				$<$	$\rightarrow$	●
		15	18					$<$	$\rightarrow$	●
	15							$\geq$		●
4	13	15	18	35	44	10	20	*4*	*3*	
i								C_i	M_i	

Insertion Sort : Stage $i = 5$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.

4	13	15	18	35	44	10	20	*4*	*3*
5					*10*	44		$<$	$\rightarrow$
				10	35			$<$	$\rightarrow$
			10	18				$<$	$\rightarrow$
		10	15					$<$	$\rightarrow$
	10	13						$<$	$\rightarrow$
5	10	13	15	18	35	44	20	*5*	*5*
i								C_i	M_i

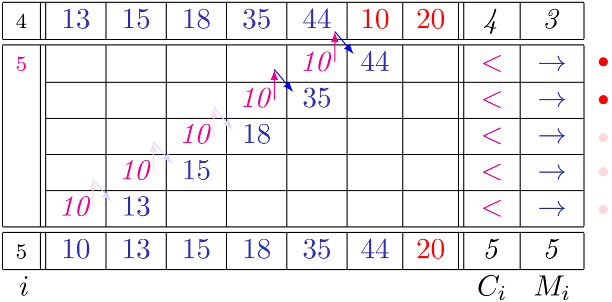
Insertion Sort : Stage $i = 5$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.

4	13	15	18	35	44	10	20	*4*	*3*
5					*10*	44		$<$	$\rightarrow$
				10	35			$<$	$\rightarrow$
			10	18				$<$	$\rightarrow$
		10	15					$<$	$\rightarrow$
	10	13						$<$	$\rightarrow$
5	10	13	15	18	35	44	20	*5*	*5*
i								C_i	M_i

Insertion Sort : Stage $i = 5$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.



i								C_i	M_i
4	13	15	18	35	44	10	20	4	3
5					10	44		$<$	$\rightarrow$
				10	35			$<$	$\rightarrow$
			10	18				$<$	$\rightarrow$
		10	15					$<$	$\rightarrow$
	10	13						$<$	$\rightarrow$
5	10	13	15	18	35	44	20	5	5

Insertion Sort : Stage $i = 5$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.

4	13	15	18	35	44	10	20	4	3	
5					10	44		$<$	$\rightarrow$	●
				10	35			$<$	$\rightarrow$	●
			10	18				$<$	$\rightarrow$	●
		10	15					$<$	$\rightarrow$	●
	10	13						$<$	$\rightarrow$	●
5	10	13	15	18	35	44	20	5	5	
i								C_i	M_i	

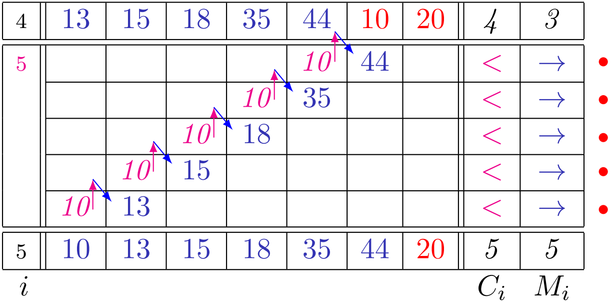
Insertion Sort : Stage $i = 5$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.

4	13	15	18	35	44	10	20	*4*	*3*	
5					*10*	44		$<$	$\rightarrow$	●
				10	35			$<$	$\rightarrow$	●
			10	18				$<$	$\rightarrow$	●
		10	15					$<$	$\rightarrow$	●
	10	13						$<$	$\rightarrow$	●
5	10	13	15	18	35	44	20	*5*	*5*	
i								C_i	M_i	

Insertion Sort : Stage $i = 5$

- C_i – the number of comparisons at stage i.
- M_i – the number of moves at stage i.



4	13	15	18	35	44	10	20	4	3	
5					10	44		<	→	●
				10	35			<	→	●
			10	18				<	→	●
		10	15					<	→	●
	10	13						<	→	●
5	10	13	15	18	35	44	20	5	5	
i								C_i	M_i	

Insertion Sort : Stage $i = 6$

- C_i – the number of comparisons per insertion
- M_i – the number of moves per insertion

5	10	13	15	18	35	44	20	*5*	*5*
5						*20*	44	$<$	$\rightarrow$
					20	35		$<$	$\rightarrow$
				20				$\geq$	
6	10	13	15	18	20	35	44	*3*	*2*
i								C_i	M_i

Insertion Sort : Stage $i = 6$

- C_i – the number of comparisons per insertion
- M_i – the number of moves per insertion

5	10	13	15	18	35	44	20	5	5
5						20	44	<	→
					20	35		<	→
				20				≥	
6	10	13	15	18	20	35	44	3	2
i								C_i	M_i

Insertion Sort : Stage $i = 6$

- C_i – the number of comparisons per insertion
- M_i – the number of moves per insertion

5	10	13	15	18	35	44	20	*5*	*5*	
5						*20*	44	$<$	$\rightarrow$	●
					20	35		$<$	$\rightarrow$	●
				20				$\geq$		●
6	10	13	15	18	20	35	44	*3*	*2*	
i								C_i	M_i	

Insertion Sort : Stage $i = 6$

- C_i – the number of comparisons per insertion
- M_i – the number of moves per insertion

5	10	13	15	18	35	44	20	*5*	*5*	
5						*20*	44	$<$	$\rightarrow$	●
					20	35		$<$	$\rightarrow$	●
				20				$\geq$		●
6	10	13	15	18	20	35	44	*3*	*2*	
i								C_i	M_i	

Total Number of Moves and Comparisons

Insertion sort:

$\{44, 13, 35, 18, 15, 10, 20\} \longrightarrow \{10, 13, 15, 18, 20, 35, 44\}\}$

Stage i	1	2	3	4	5	6	Total
Comparisons C_i	*1*	*2*	*3*	*4*	*5*	*3*	**18**
Moves M_i	*1*	*1*	*2*	*3*	*5*	*2*	**14**

- The best case – an already sorted array, e.g. $\{10, 13, 15, 18, 20, 35, 44\}$:
  - 1 comparison and 0 moves per each stage $i = 1, \ldots, n-1$.
  - In total, 0 moves and $n-1$ comparisons for the already sorted array of size n.
- The worst case – a reversely sorted array. e.g. $\{44, 35, 20, 18, 15, 13, 10\}$:
  - i comparisons and i moves per each stage $i = 1, \ldots, n-1$.
  - In total, $1 + \ldots + (n-1) = \frac{(n-1)n}{2}$ moves and $\frac{(n-1)n}{2}$ comparisons for the reversely sorted array of size n.