Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.

- Any computer output is generally arranged in some sorted order so that it can be interpreted.

- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.

- Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.

The Sorting Problem

- **Input:**
  - A sequence of n numbers $a_1, a_2, \ldots, a_n$
- **Output:**
  - A permutation (reordering) $a_1', a_2', \ldots, a_n'$ of the input sequence such that $a_1' \leq a_2' \leq \cdots \leq a_n'$

Stable sort algorithms

- A stable sort keeps equal elements in the same order
- This may matter when you are sorting data according to some characteristic
- Example: sorting students by test scores

original array		stably sorted	
Ann	98	Ann	98
Bob	90	Joe	98
Dan	75	Bob	90
Joe	98	Sam	90
Pat	86	Pat	86
Sam	90	Zöe	86
Zöe	86	Dan	75

Why Study Sorting Algorithms?

- There are a variety of situations that we can encounter
  - Do we have randomly ordered elements?
  - Are all elements distinct?
  - How large is the set of elements to be ordered?
  - Need guaranteed performance?

- Various algorithms are better suited to some of these situations

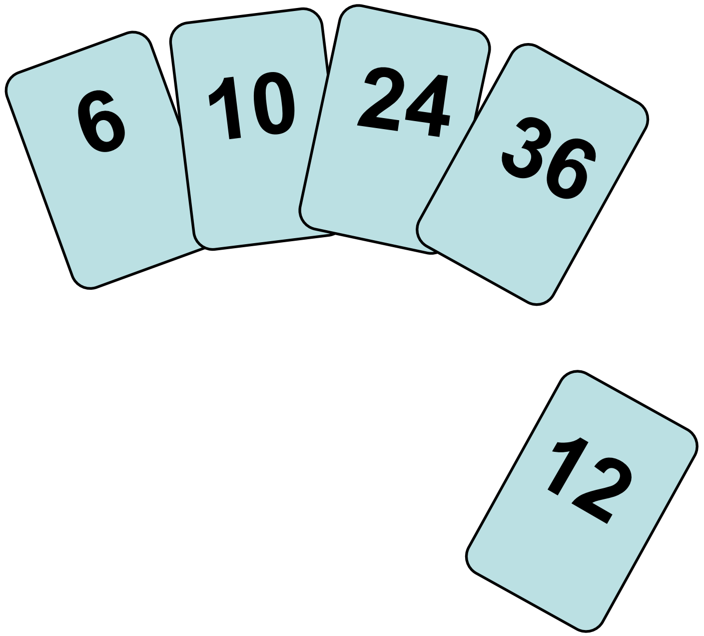
Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

Insertion Sort

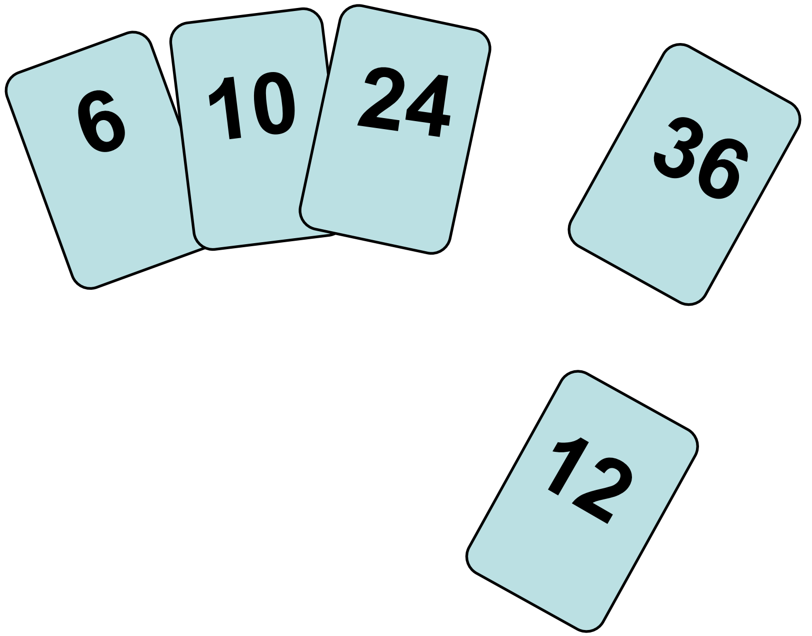
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

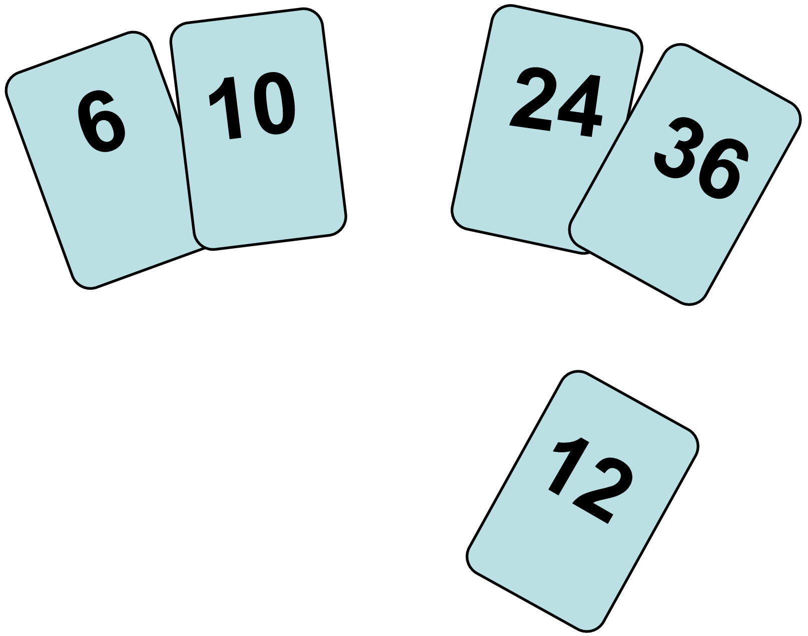


To insert 12, we need to make room for it by moving first 36 and then 24.

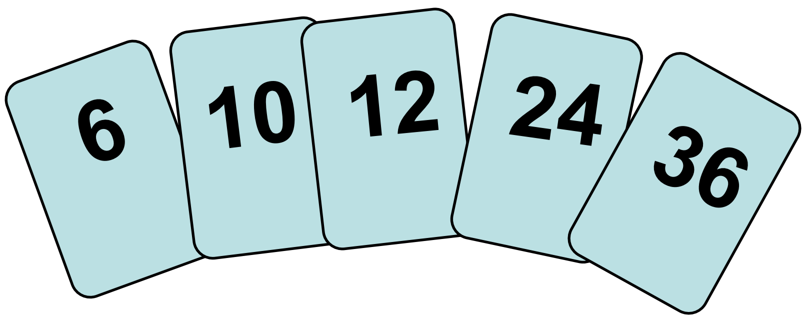
Insertion Sort



Insertion Sort



Insertion Sort



Insertion Sort

input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

left sub-array

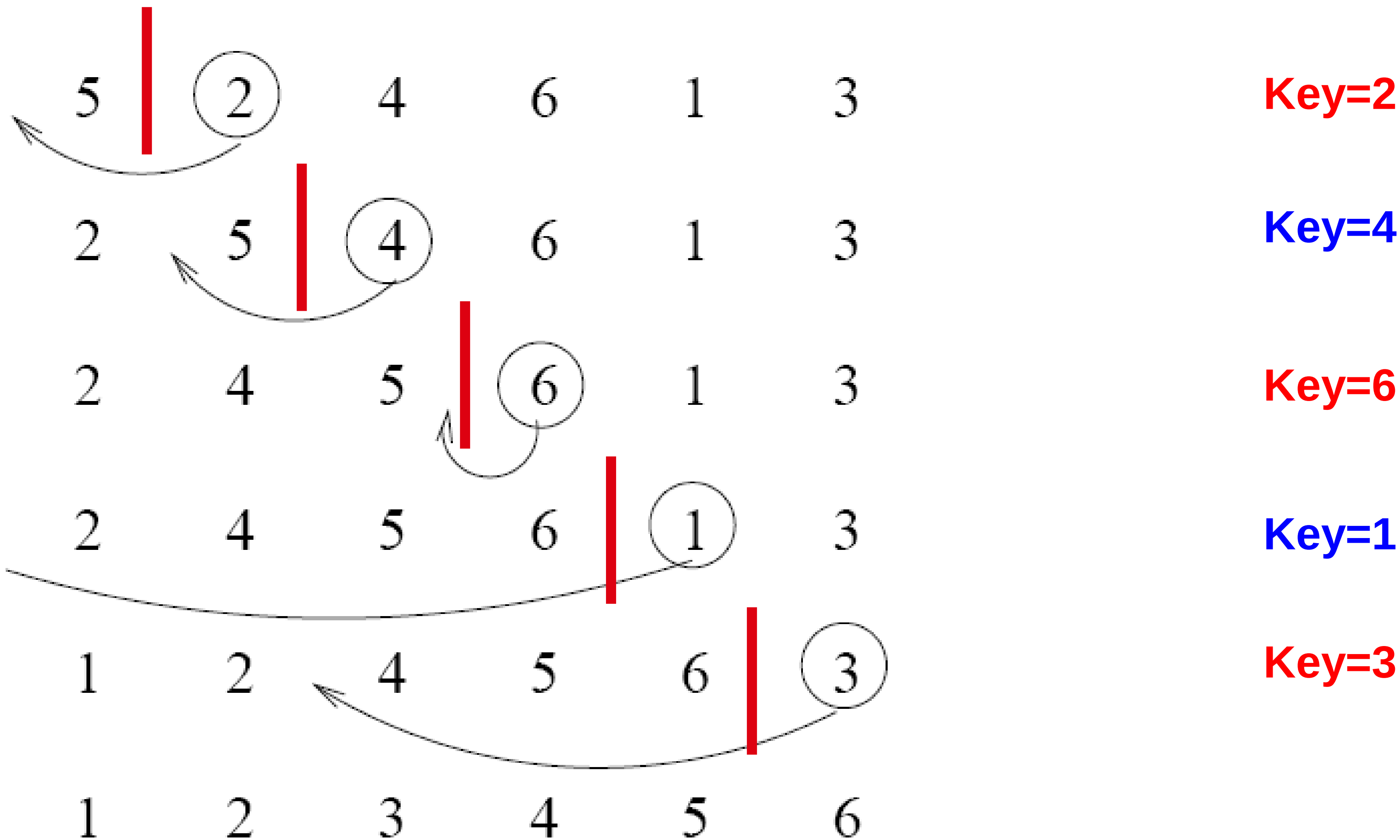
right sub-array

2 5 | 4 6 1 3

sorted

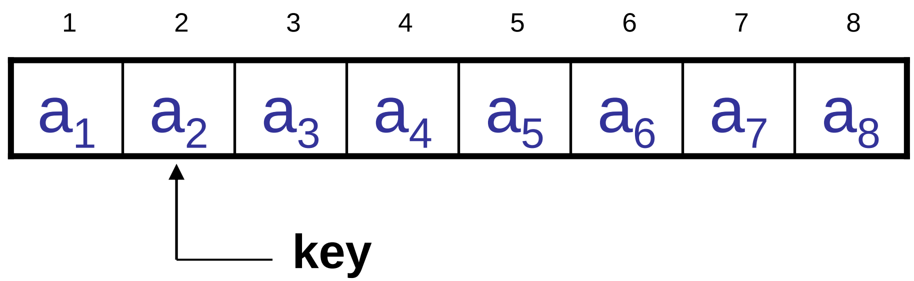
unsorted

Insertion Sort



INSERTION-SORT

Alg.: INSERTION-SORT(*A*)



for j ← 2 **to** n

  do key ← A[j]

    Insert A[j] into the sorted sequence 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 – sorts the elements <u>in place</u>

Analysis of Insertion Sort

INSERTION-SORT*(A)*

	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	$n-1$
▷Insert A[j] into the sorted sequence A[1 . . j -1]	0	$n-1$
i ← 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] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	$n-1$

t_j: # of times the while statement is executed at iteration j

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

Best Case Analysis

- The array is already sorted **"while** i > 0 and A[i] > key"
  - A[i] ≤ key upon the first time the **while** loop test is run (when $i = j - 1$)
  - $t_j = 1$
- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$

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

$= an - b$ **$= O(n)$**

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Worst Case Analysis

- The array is in reverse sorted order **"while** i > 0 and A[i] > key"
  - Always A[i] > key in **while** loop test
  - Have to compare key with all elements to the left of the j-th position ⇒ compare with j-1 elements ⇒ $t_j = j$

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \Rightarrow \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ we have:

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

$$= an^2 + bn + c$$ a quadratic function of n

- $T(n) = O(n^2)$ order of growth in n^2

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Comparisons and Exchanges in Insertion Sort

INSERTION-SORT(*A*)		cost	times
for j ← 2 **to** n		c_1	n
do key ← A[j]		c_2	$n-1$
Insert A[j] into the sorted sequence A[1 . . j -1]		0	$n-1$
i ← j - 1	$\approx n^2/2$ comparisons	c_4	$n-1$
while i > 0 and A[i] > key		c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]		c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	$\approx n^2/2$ exchanges	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key		c_8	$n-1$

```
public void insertionSort(int[] list)
{    int temp, j;
     for(int j = 1; j < list.length; j++)
     {    temp = list[j];
          i = j-1;
          while( i > -1 && temp < list[i])
          {    // swap elements
               list[i+1] = list[i];
               i--;
          }
          list[i + 1] = temp;
     }
}
```

Insertion Sort - Summary

- Advantages
  - Good running time for “almost sorted” arrays O(n)
- Disadvantages
  - $O(n^2)$ running time in worst and average case
  - $\approx n^2/2$ comparisons and exchanges

Comparison of Sorting Algorithms

Sorting Algorithm	Best Case time	Worst Case time	Avg. Case time
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$

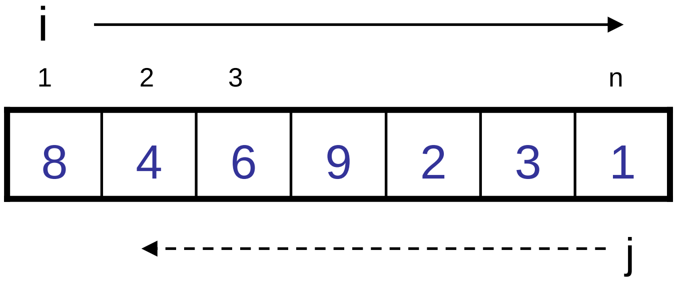
Internal Sort	Yes
External Sort	No
Stable Sort	Yes
In Place	Yes

Insertion Sort

5	2	4	6	1	3	Key=2
2	5	4	6	1	3	Key=4
2	4	5	6	1	3	Key=6
2	4	5	6	1	3	Key=1
1	2	4	5	6	3	Key=3
1	2	3	4	5	6	

Bubble Sort

- Idea:
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



- Easier to implement, but slower than Insertion sort

Example

8	4	6	9	2	3	1

i = 1 ← j

8	4	6	9	2	1	3

i = 1 ← j

8	4	6	9	1	2	3

i = 1 ← j

8	4	6	1	9	2	3

i = 1 ← j

8	4	1	6	9	2	3

i = 1 ← j

8	1	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 2 j

1	2	8	4	6	9	3

i = 3 j

1	2	3	8	4	6	9

i = 4 j

1	2	3	4	8	6	9

i = 5 j

1	2	3	4	6	8	9

i = 6 j

1	2	3	4	6	8	9

i = 7

j

Bubble Sort

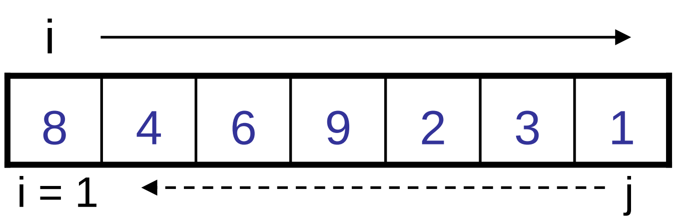
Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A]

  do for j ← length[A] **downto** i + 1

    do if A[j] < A[j -1]

      then exchange A[j] ↔ A[j-1]



Bubble-Sort Running Time

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A] c_1

do for j ← length[A] **downto** i + 1 c_2

Comparisons: ≈ $n^2/2$ **do if** A[j] < A[j -1] c_3

Exchanges: ≈ $n^2/2$ **then** exchange A[j] ↔ A[j-1] c_4

$$T(n) = c_1(n+1) + c_2\sum_{i=1}^{n}(n-i+1) + c_3\sum_{i=1}^{n}(n-i) + c_4\sum_{i=1}^{n}(n-i)$$

$$= O(n) + (c_2 + c_2 + c_4)\sum_{i=1}^{n}(n-i)$$

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

Thus, $T(n) = O(n^2)$

Comparison of Sorting Algorithms

Sorting Algorithm	**Best Case time**	**Worst Case time**	**Avg. Case time**
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$

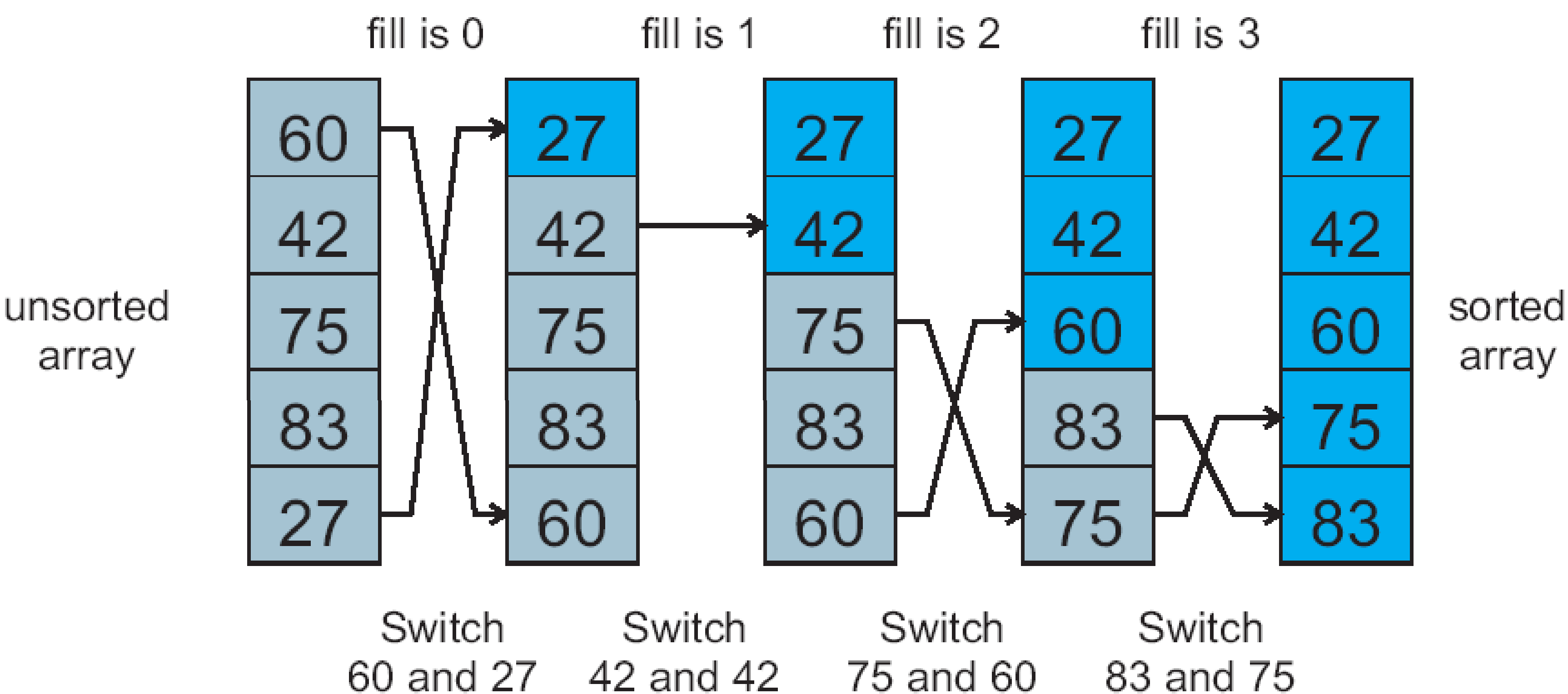
Internal Sort Yes
External Sort No
Stable Sort Yes
In Place Yes

Selection Sort

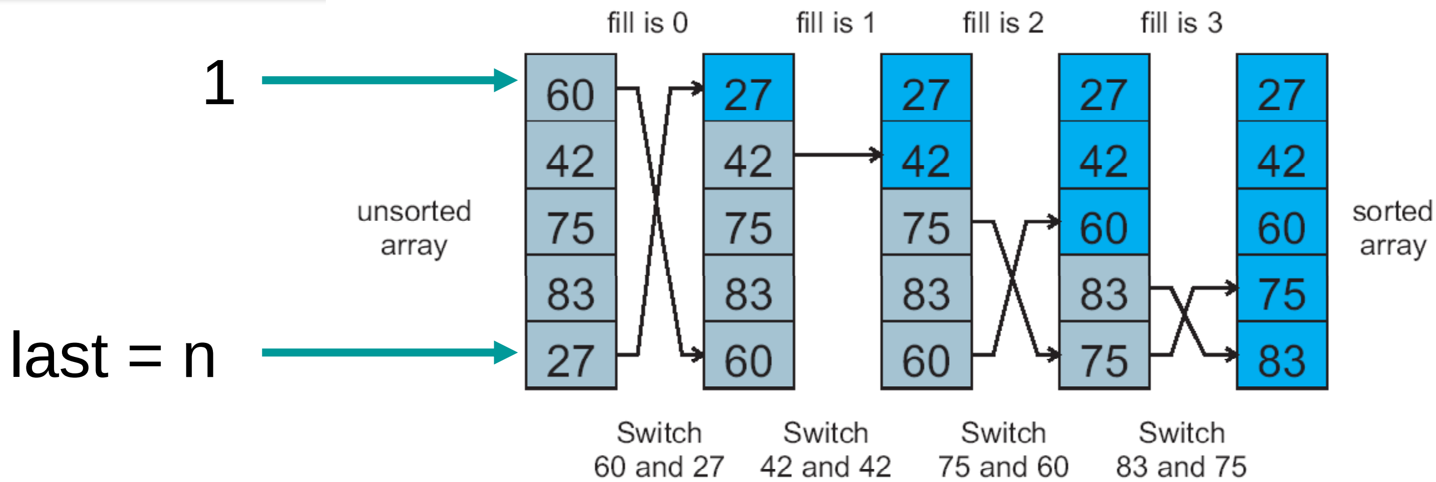
- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- Disadvantage:
  - Running time depends only slightly on the amount of order in the file

Selection Sort: Example

- The Brute Force Method: Selection Sort



Selection Sort: Algorithm



Algorithm:

- For j=1 .. last
- find smallest element M in subarray j .. last
- if M != element at j: swap elements

Selection Sort

Alg.: SELECTION-SORT(*A*)

```
n ← length[A]
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
    exchange A[j] ↔ A[smallest]
```

8	4	6	9	2	3	1

Analysis of Selection Sort

Alg.: SELECTION-SORT*(A)*

	cost	times
n ← length[A]	c_1	1
for j ← 1 **to** n - 1	c_2	n
do smallest ← j	c_3	n-1
for i ← j + 1 **to** n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] ↔ A[smallest]	c_7	n-1

≈$n^2/2$ comparisons

≈n exchanges

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$

Analysis of Selection Sort

Number of comparisons:

- $(n-1) + (n-2) + ... + 3 + 2 + 1 =$
- $n * (n-1)/2 =$
- $(n^2 - n)/2$
- $\rightarrow O(n^2)$

Number of exchanges (worst case):

- $n - 1$

$\rightarrow O(n)$

Overall (worst case) $O(n) + O(n^2) = O(n^2)$ (‘quadratic sort‘)

```
public static void selectionSort(int[] list)
{     int min;
      int temp;
      for(int j=1; i<=list.length -1; j++)
      {
            min = j;
            for(int i=j+1; i <= list.length; i++)
                  if( list[i] < list[min] )
                        min = i;
            temp = list[j];
            list[j] = list[min];
            list[min] = temp;
       }
}
```

Comparison of Sorting Algorithms

Sorting Algorithm	Best Case time	Worst Case time	Avg. Case time
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$

Internal Sort Yes
External Sort No
Stable Sort No
In Place Yes

Can be made stable?