Data Structure and Algorithm (CS 102)

Sambit Bakshi
Dept. of Computer Science and Engineering
National Institute of Technology Rourkela, India

Sorting

Sorting refers to the operation of arranging data in some order such as increasing or decreasing with numerical data or alphabetically with character data

Complexity of Sorting Algorithm

The complexity of a sorting algorithm measures the running time as a function of the number on n of items to be sorted

Complexity of Sorting Algorithm

Each sorting algorithm S will be made up of the following operations, where $A_{1,}$ A_2 , ……, A_n contain the items to be sorted and B is an auxiliary location:

(a) Comparisons, which test whether $A_i < A_j$ or test $A_i < B$

(b) Interchange, which switch the content of A_i and A_j or of A_i and B

(c) Assignments, which set $B := A_i$ and then set $A_j := B$ or Set $A_j := A_i$

Sorting

Algorithms are divided into two categories:

Internal Sorts

and

External sorts.

Sorting

Internal Sort:

Any sort algorithm which uses main memory exclusively during the sort.

This assumes high-speed random access to all memory.

Sorting

External Sort:

Any sort algorithm which uses external memory, such as tape or disk, during the sort.

Sorting

Note:

Algorithms may read the initial values from magnetic tape or write sorted values to disk, but this is not using external memory during the sort. Note that even though virtual memory may mask the use of disk, sorting sets of data much larger than main memory may be much faster using an explicit external sort.

Sorting

Sort Stable

A sort algorithm is said to be "stable" if multiple items which compare as equal will stay in the same order they were in after a sort.

Internal Sorting

Bubble Sort

The oldest and simplest sort in use.

Unfortunately, also the slowest.

Works by comparing each item in the list with the item next to it, and swapping them if required.

Bubble Sort

The algorithm repeats this process until it makes a pass all the way through the list without swapping any items.

This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

Bubble Sort

Suppose the list of number A[1], [2], A[3], …, A[N] is in memory. Algorithm works as follows.

[1] Compare A[1] and A[2], arrange them in the desired order so that A[1] < A[2]. Then Compare A[2] and A[3], arrange them in the desired order so that A[2] < A[3]. Continue until A[N-1] is compared with A[N], arrange them so that A[N-1] < A[N].

Bubble Sort

[2] Repeat Step 1, Now stop after comparing and re-arranging A[N-2] and A[N-1].

[3] Repeat Step 3, Now stop after comparing and re-arranging A[N-3] and A[N-2].

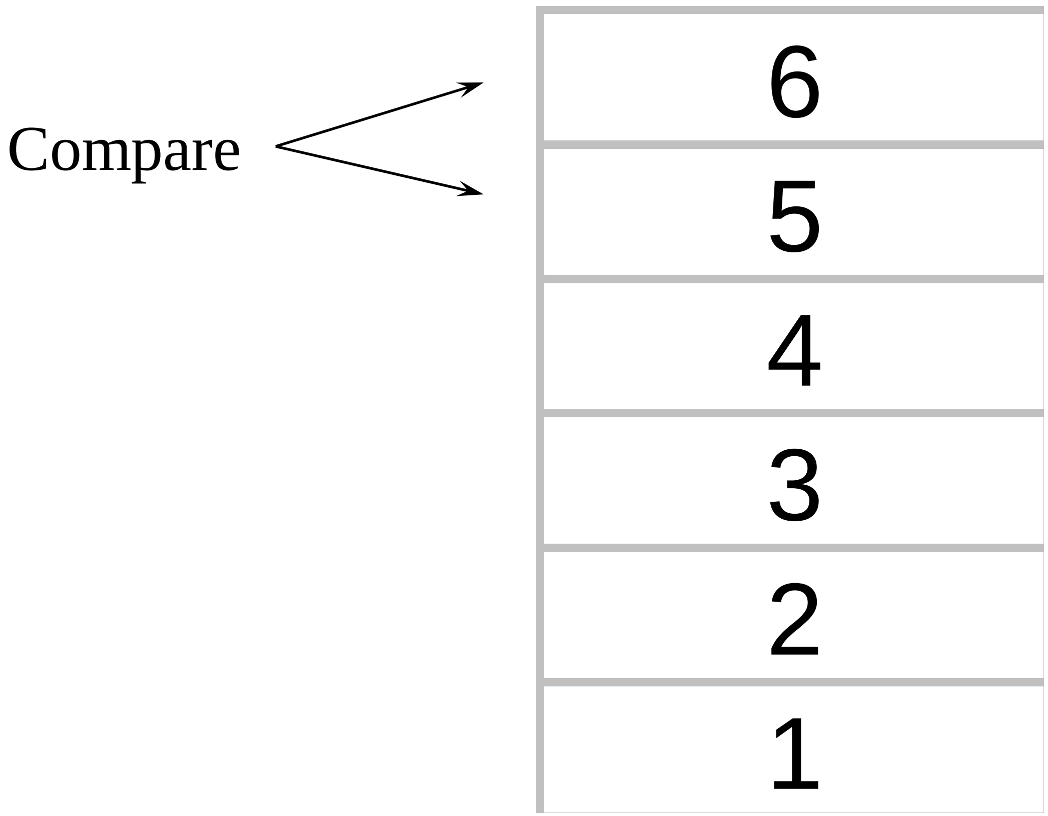
.

.

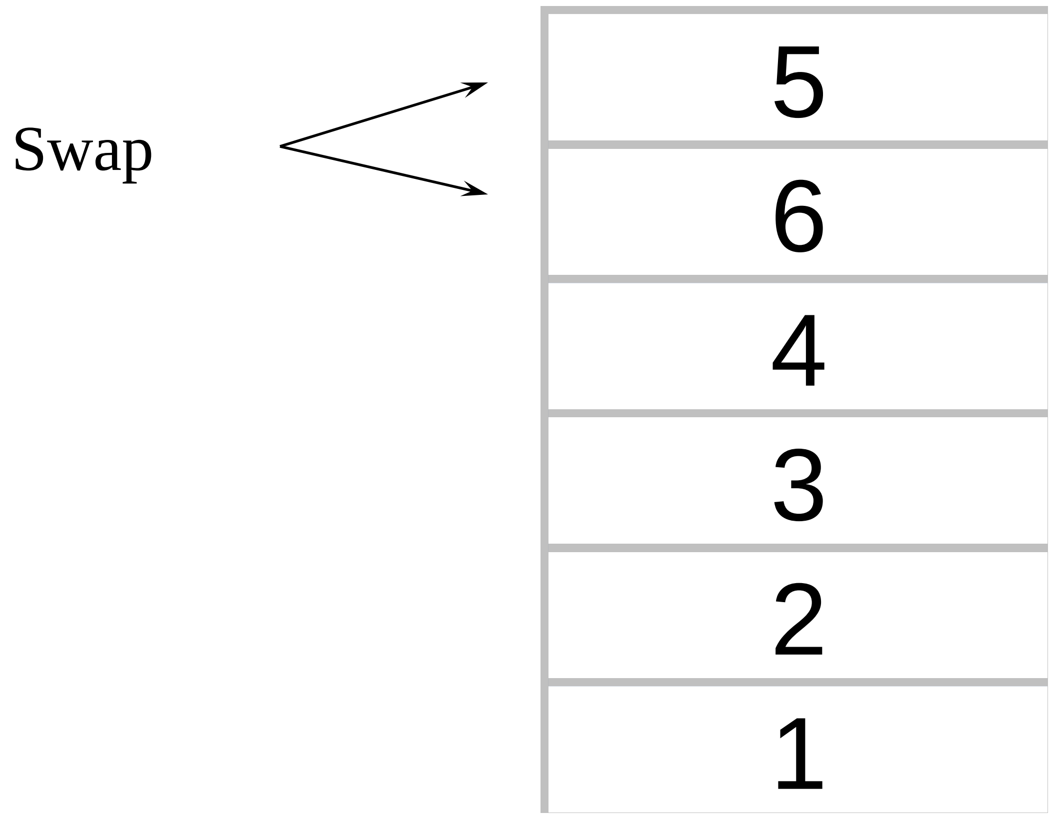
[N-1] Compare A[1] and A[2] and arrange them in sorted order so that A[1] < A[2].

After N-1 steps the list will be sorted in increasing order.

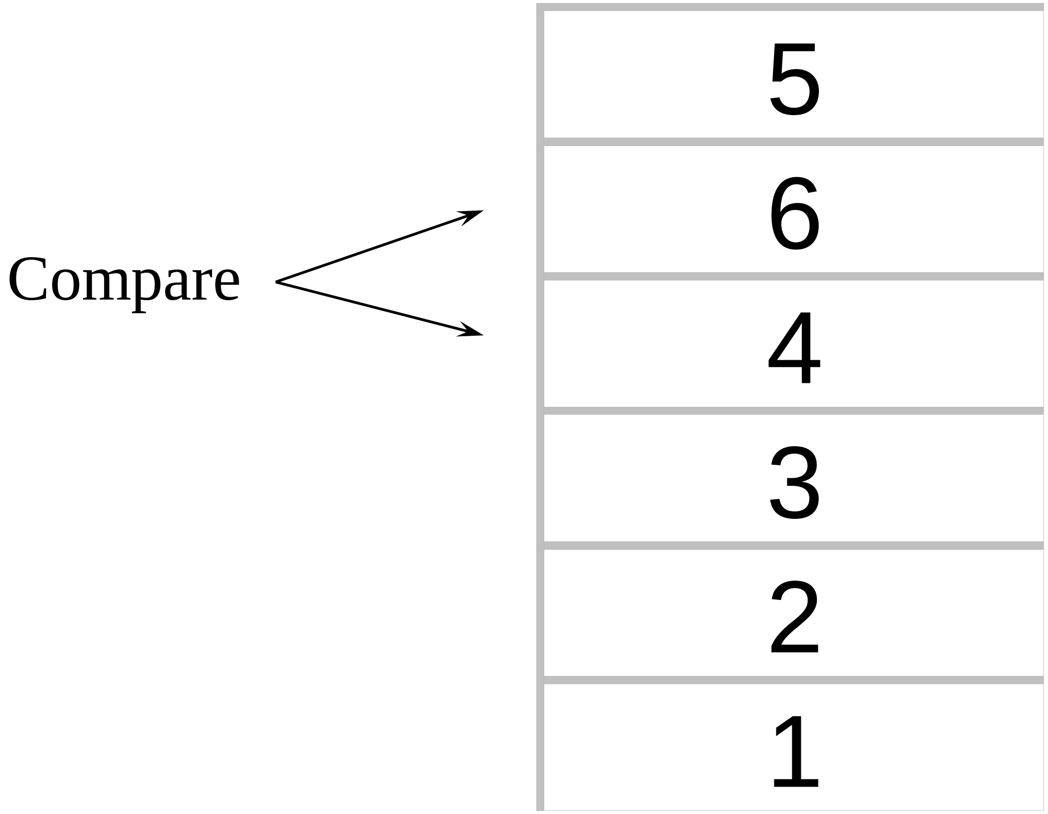
A Bubble Sort Example



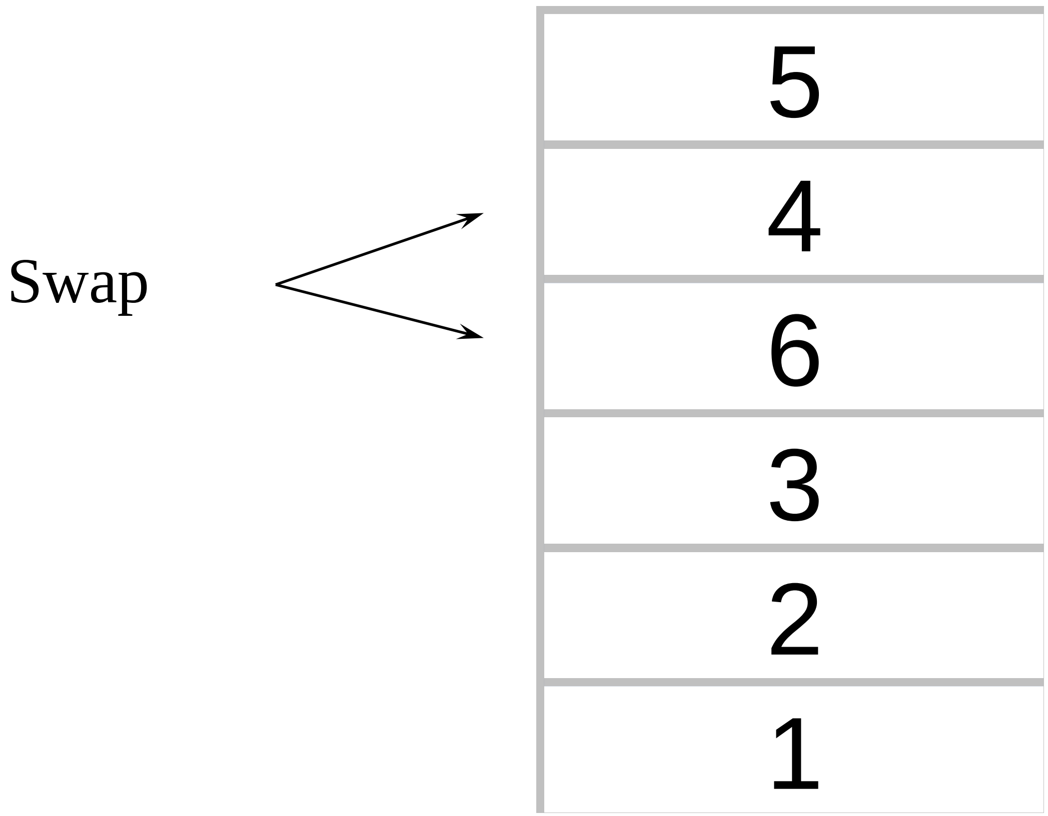
A Bubble Sort Example



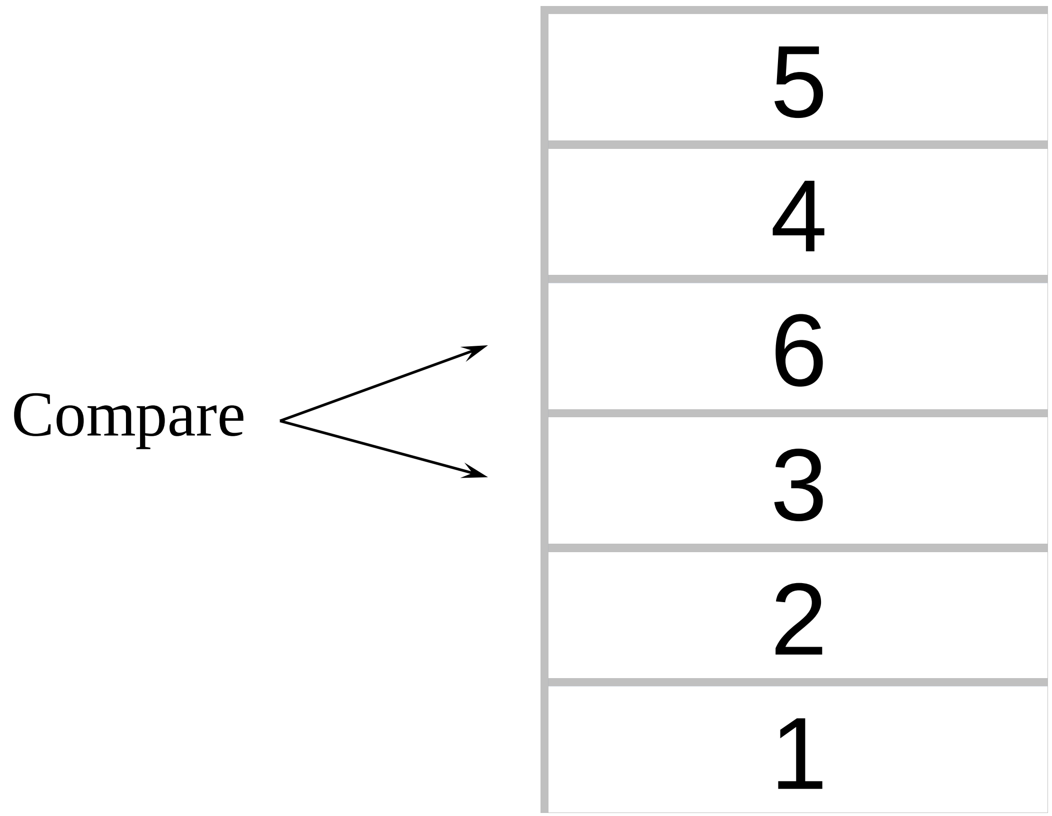
A Bubble Sort Example



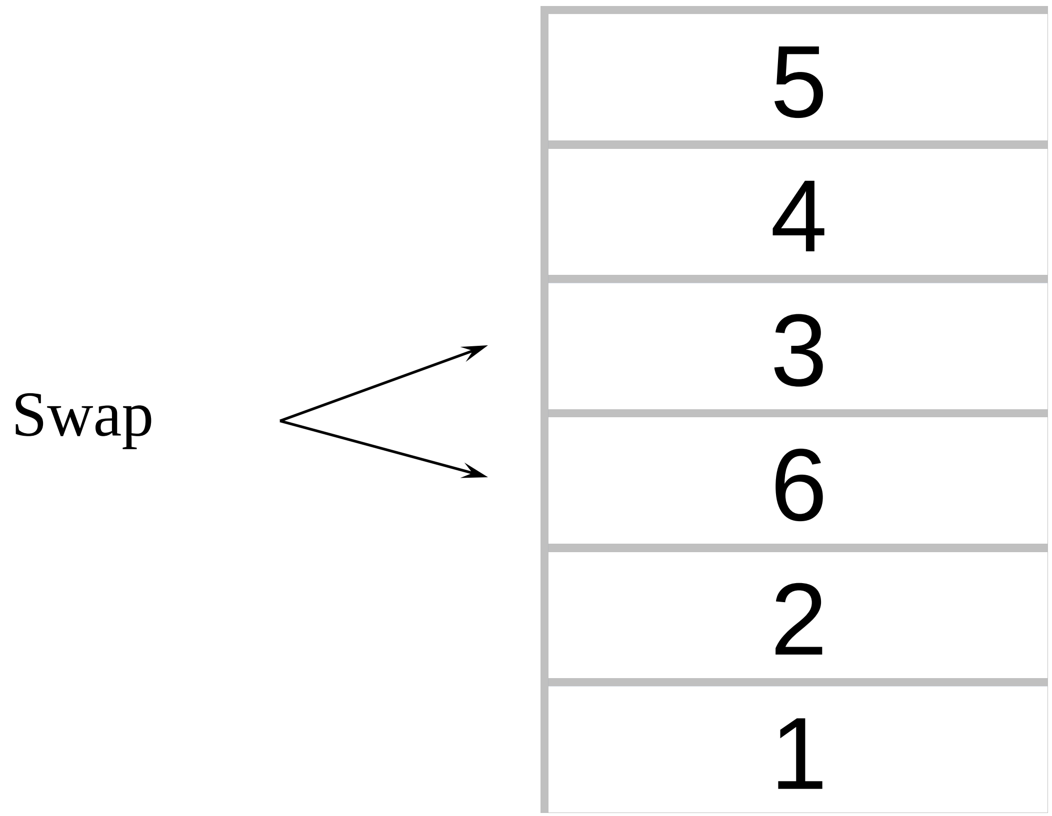
A Bubble Sort Example



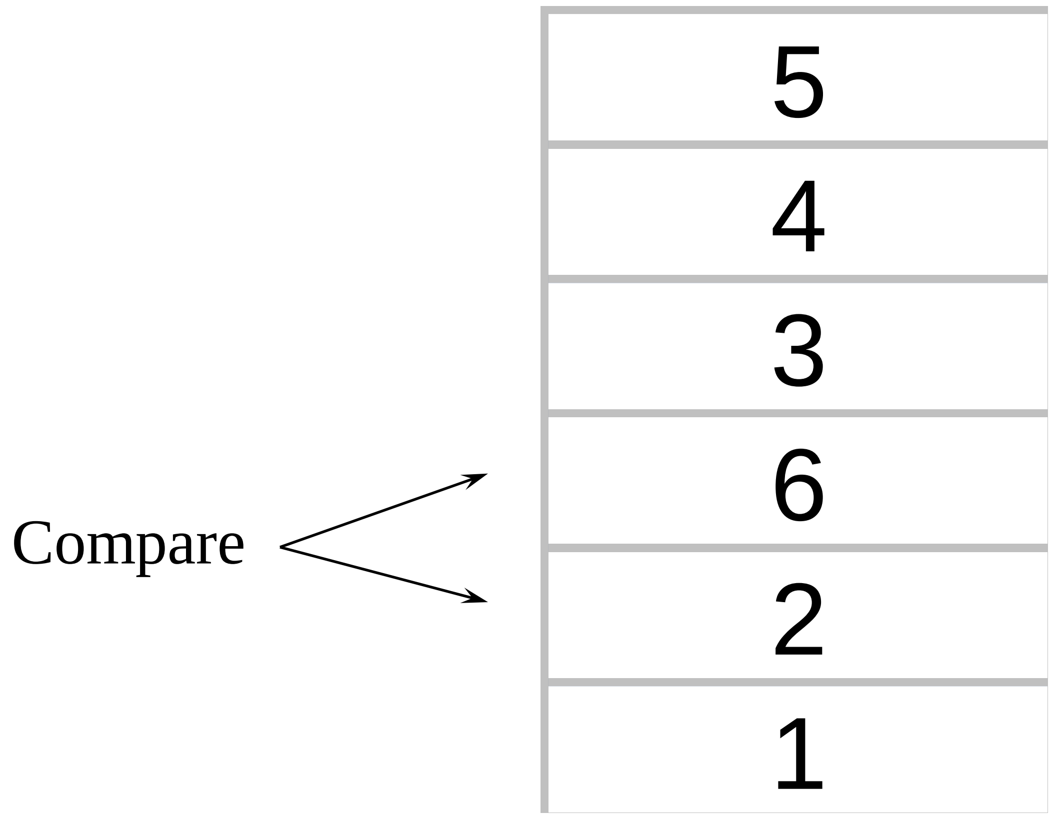
A Bubble Sort Example



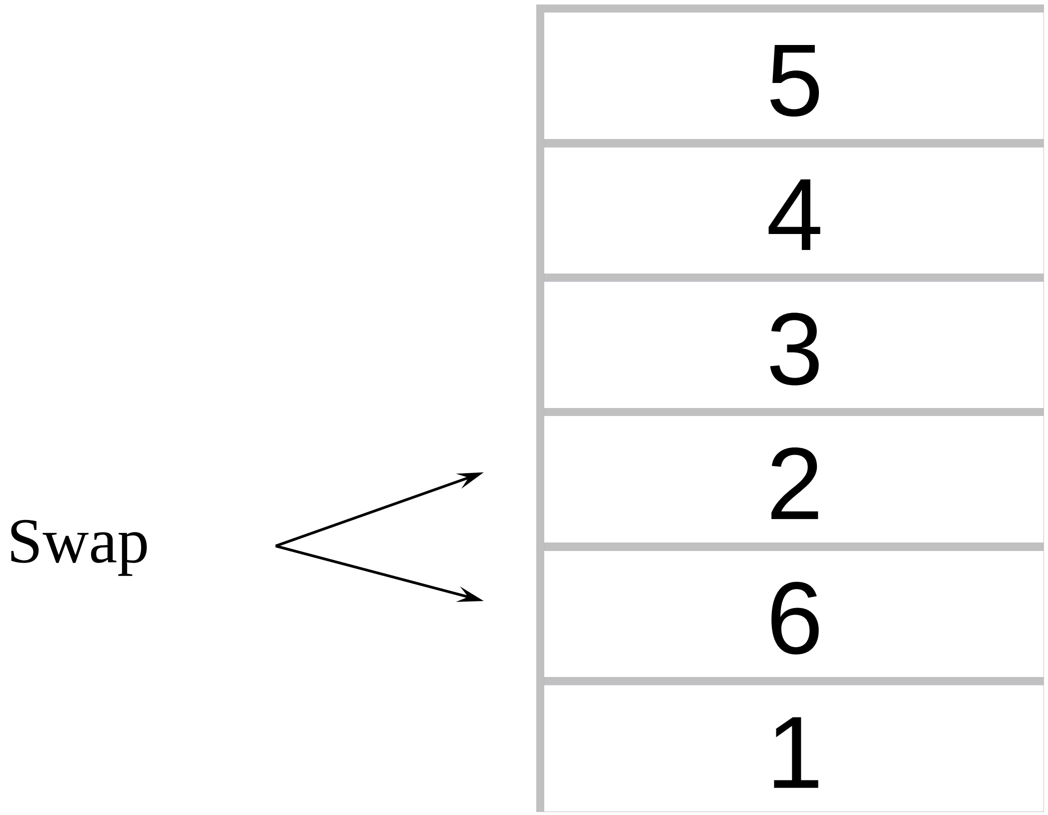
A Bubble Sort Example



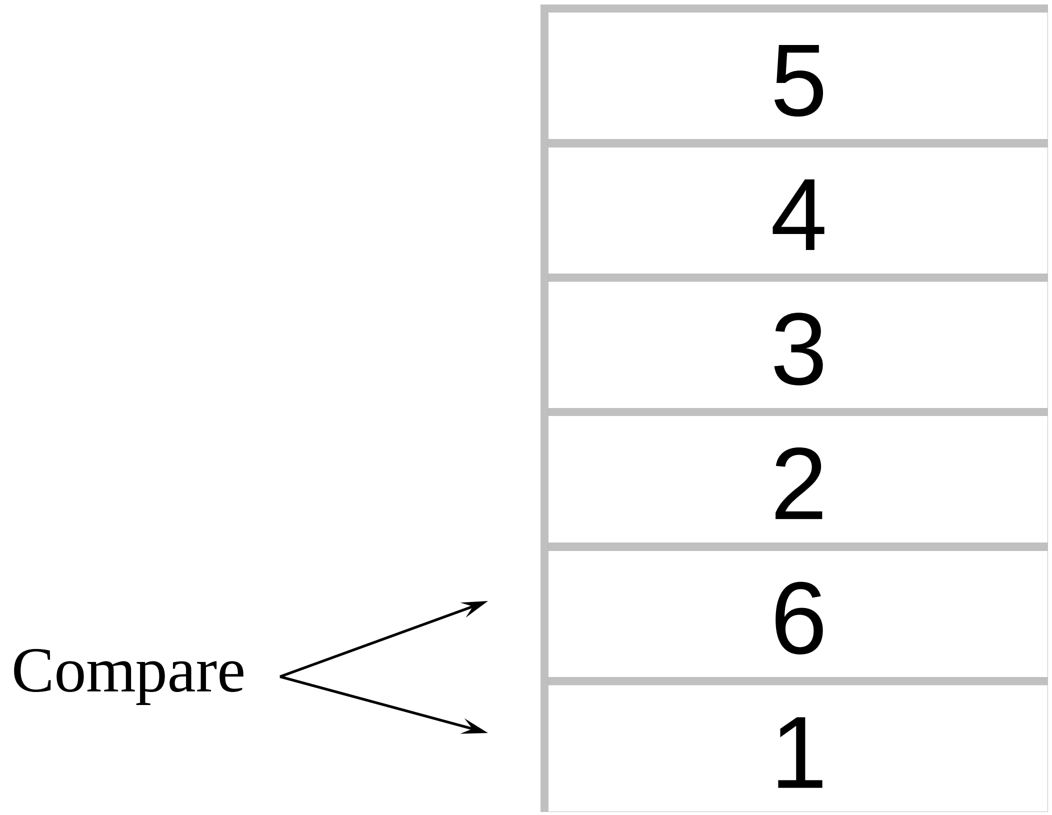
A Bubble Sort Example



A Bubble Sort Example

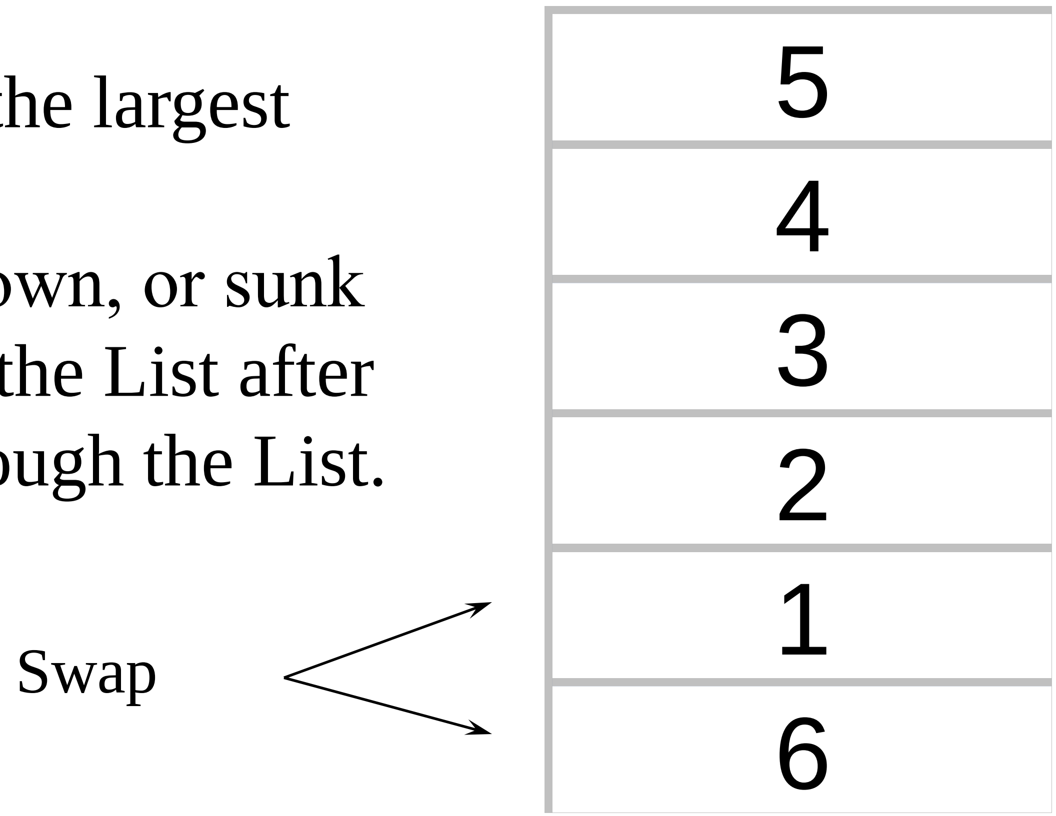


A Bubble Sort Example

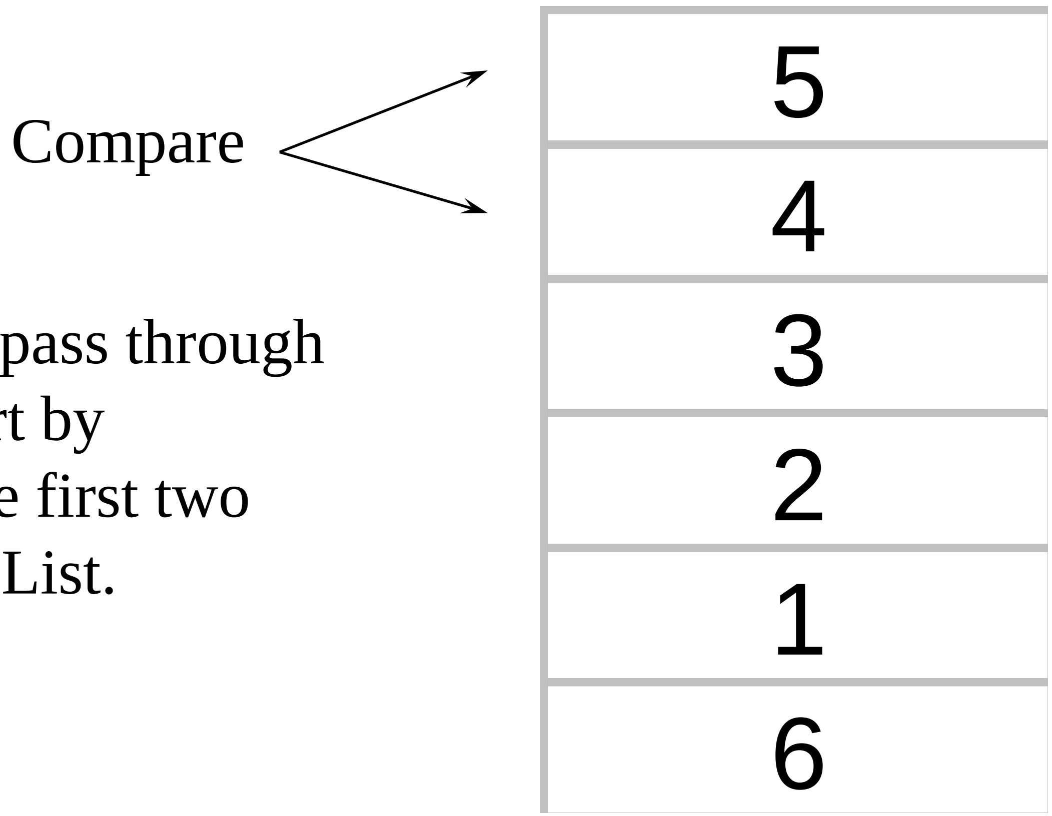


A Bubble Sort Example

As you can see, the largest number
has "bubbled" down, or sunk
to the bottom of the List after
the first pass through the List.

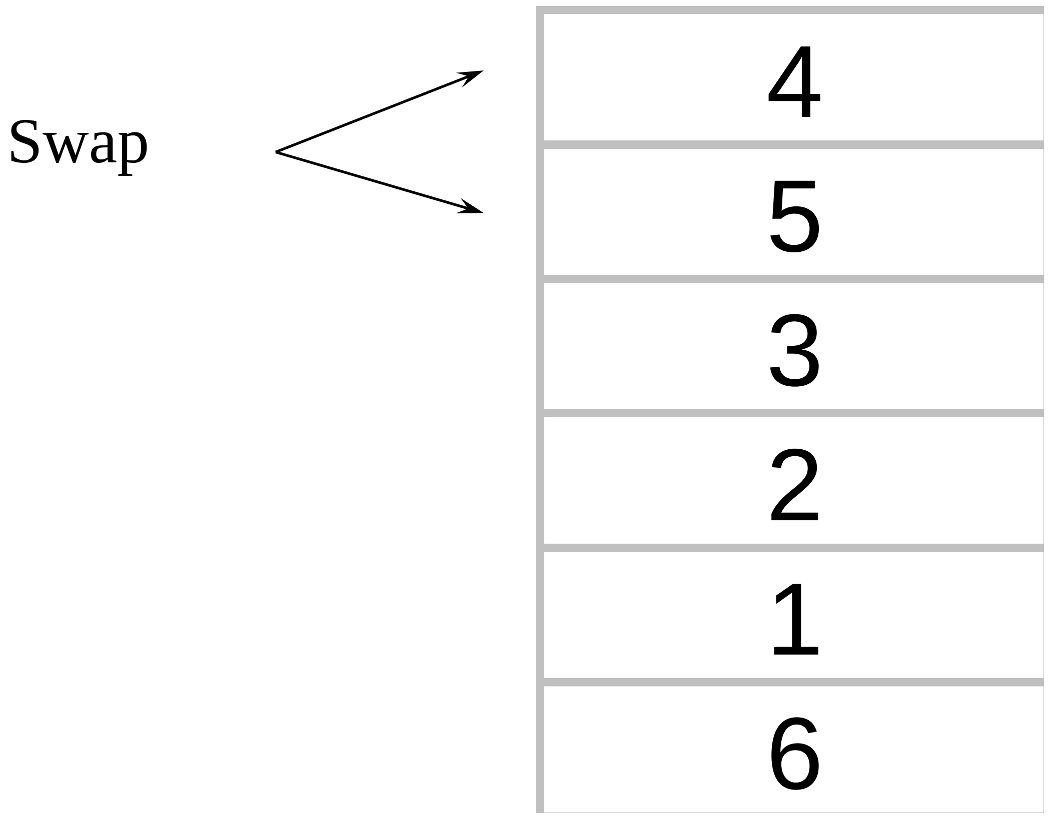


A Bubble Sort Example

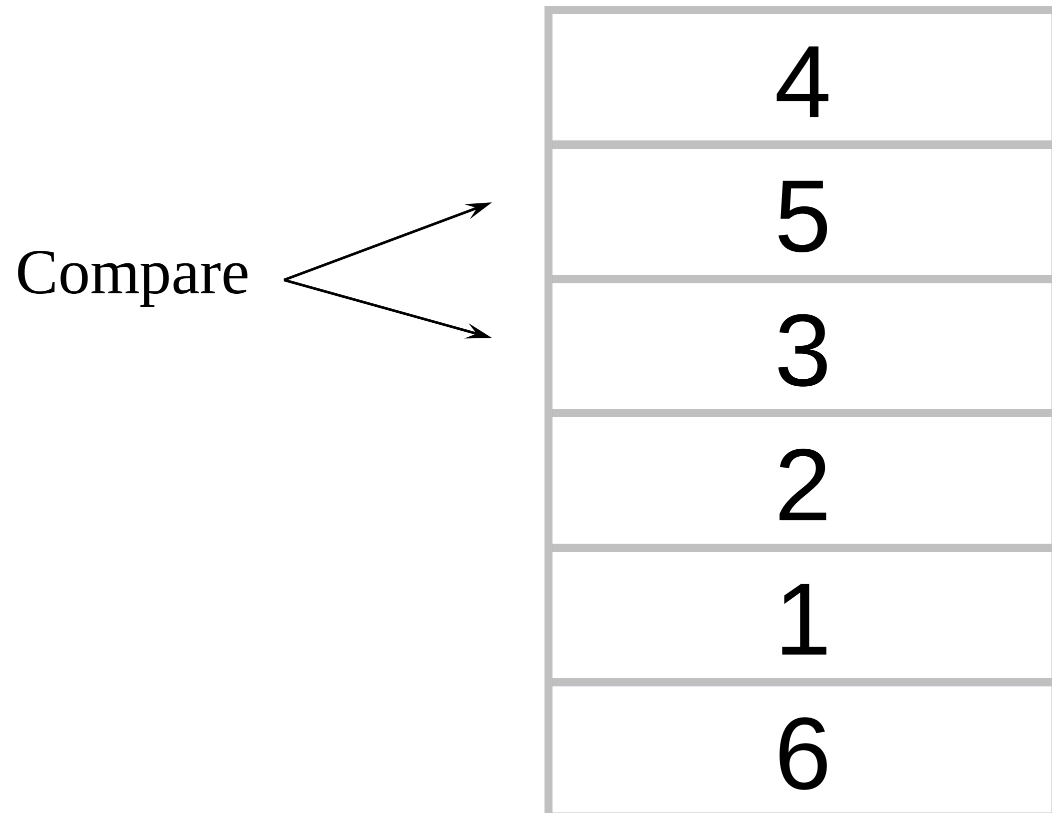


For our <u>second</u> pass through the List, we start by comparing these first two elements in the List.

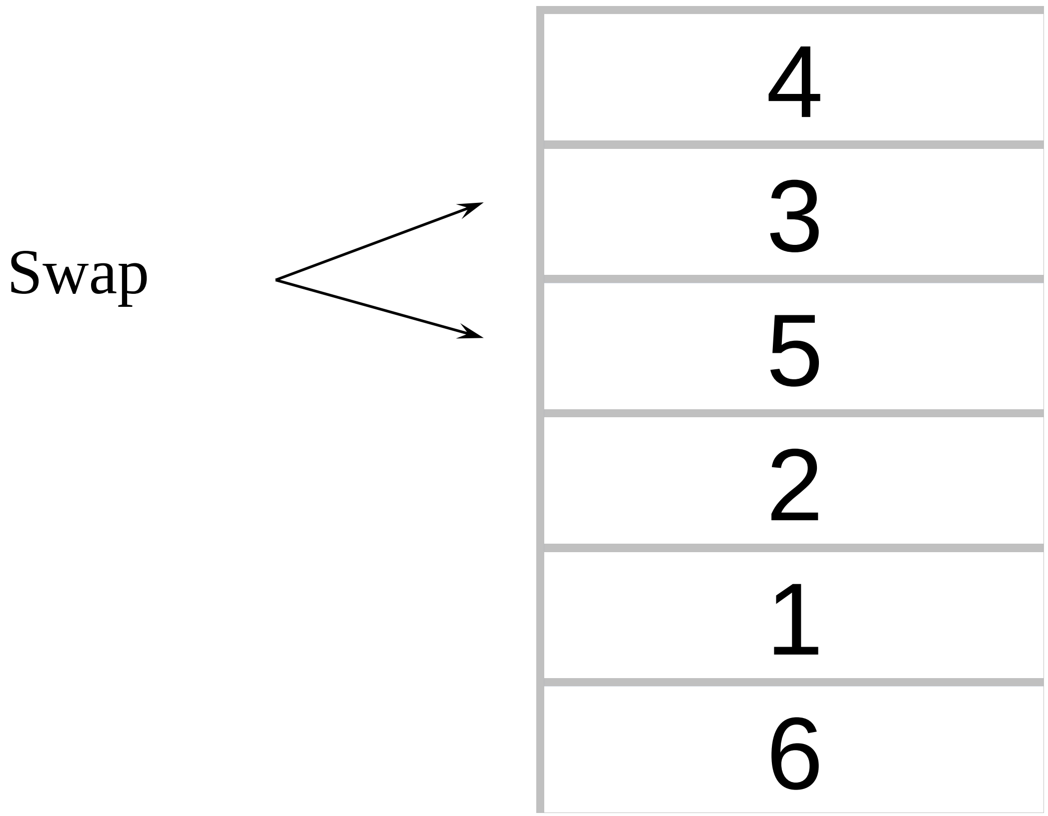
A Bubble Sort Example



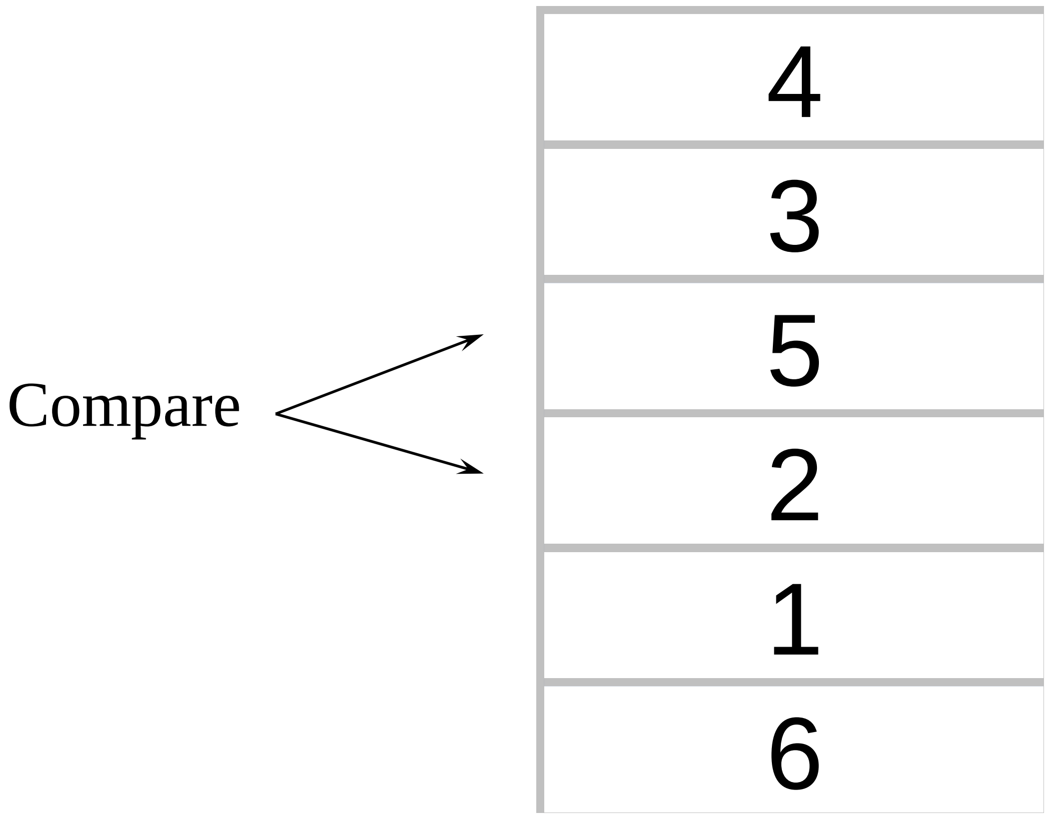
A Bubble Sort Example



A Bubble Sort Example



A Bubble Sort Example



A Bubble Sort Example

Swap

4
3
2
5
1
6

A Bubble Sort Example

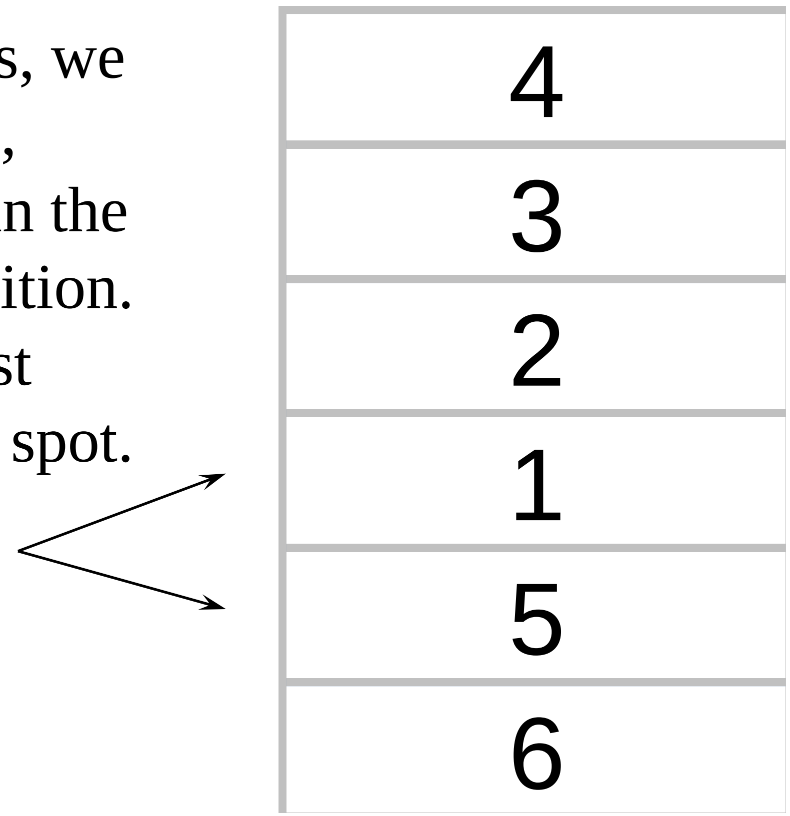
Compare

4
3
2
5
1
6

A Bubble Sort Example

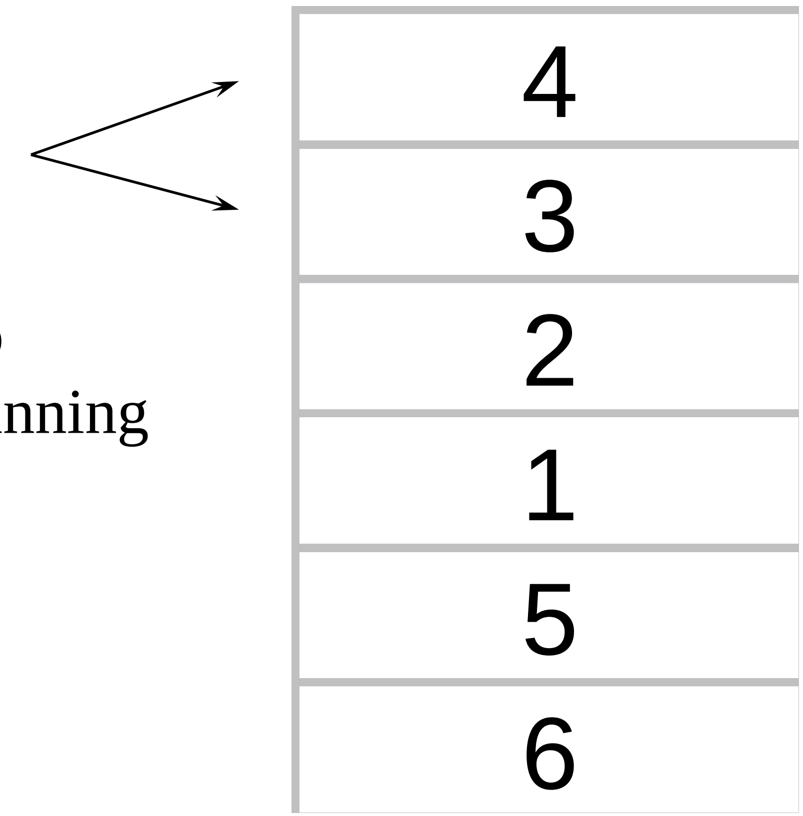
At the end of the second pass, we stop at element number n - 1, because the largest element in the List is already in the last position. This places the second largest element in the second to last spot.

Swap



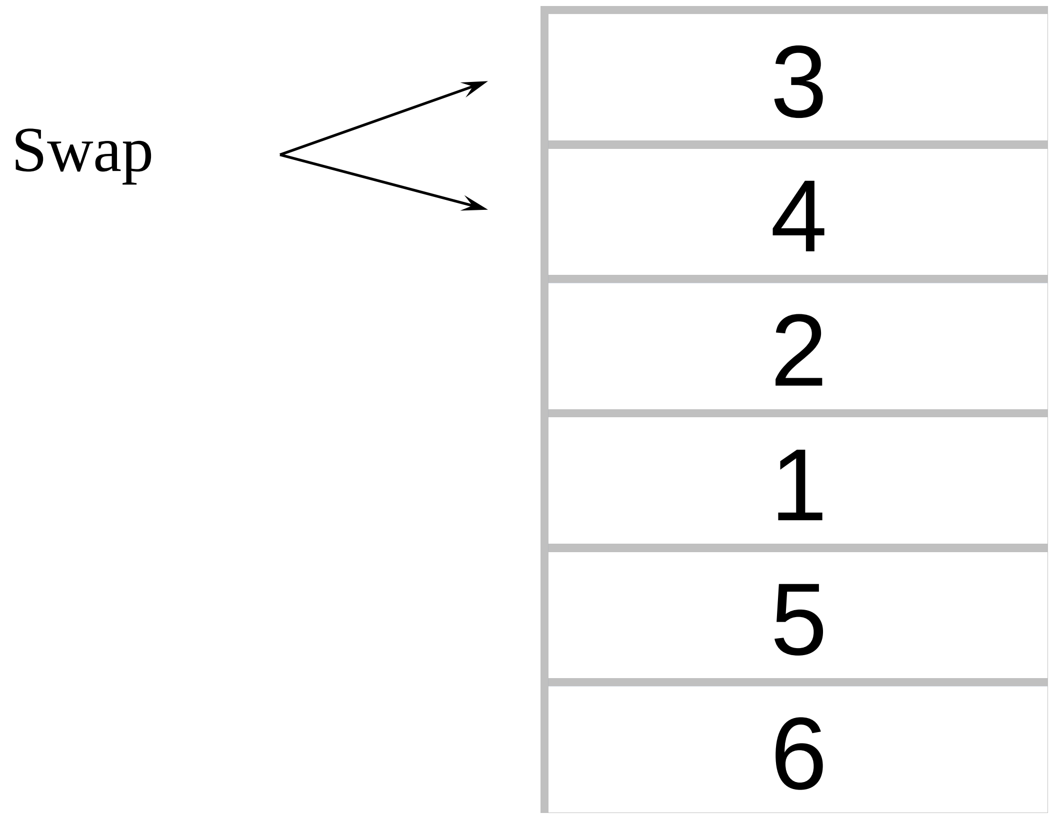
A Bubble Sort Example

Compare

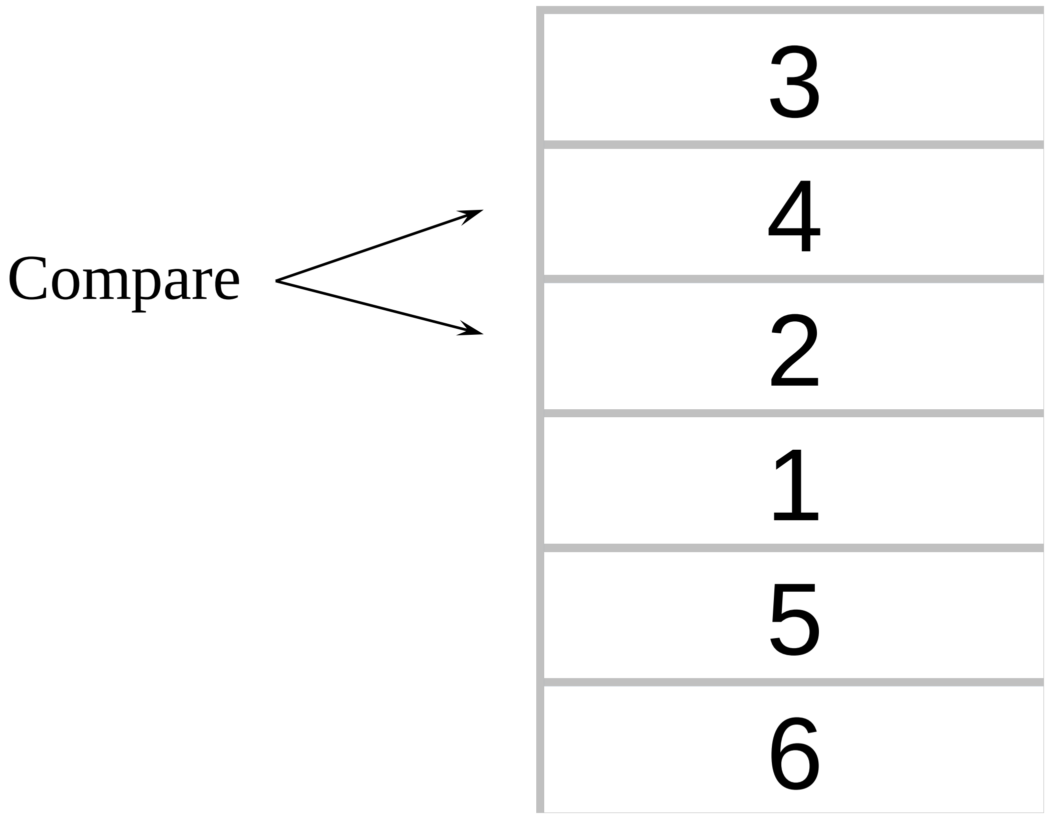


We start with the first two elements again at the beginning of the <u>third</u> pass.

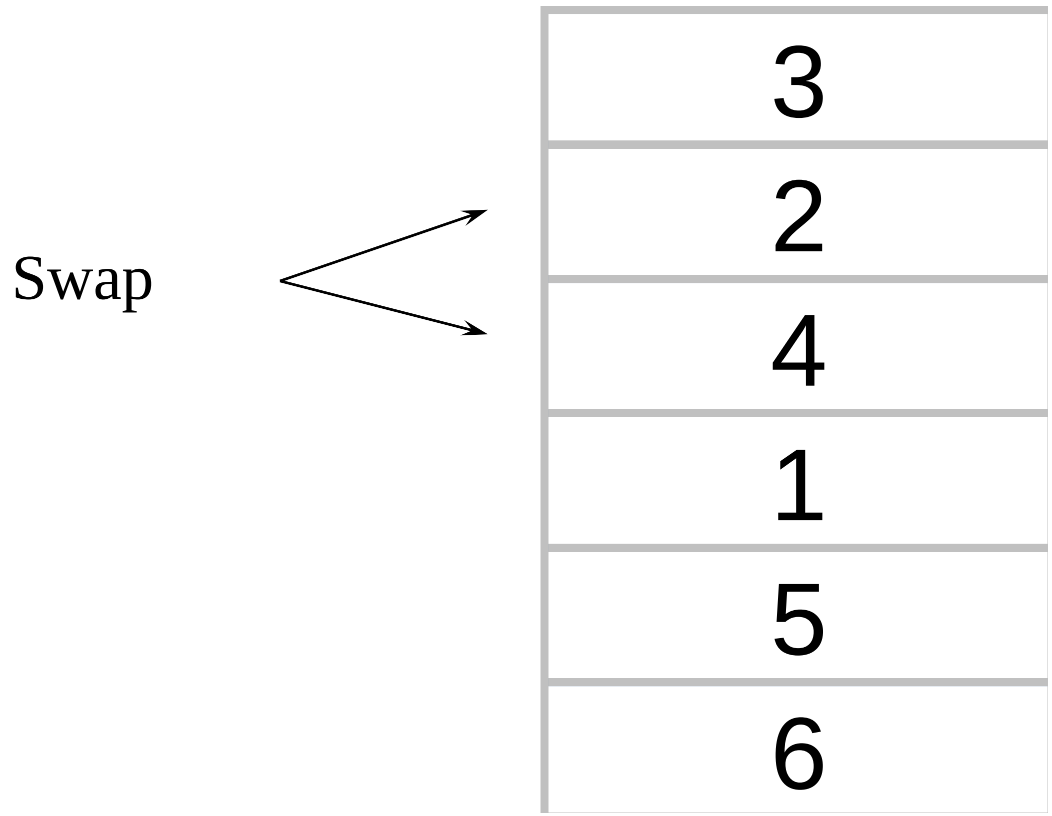
A Bubble Sort Example



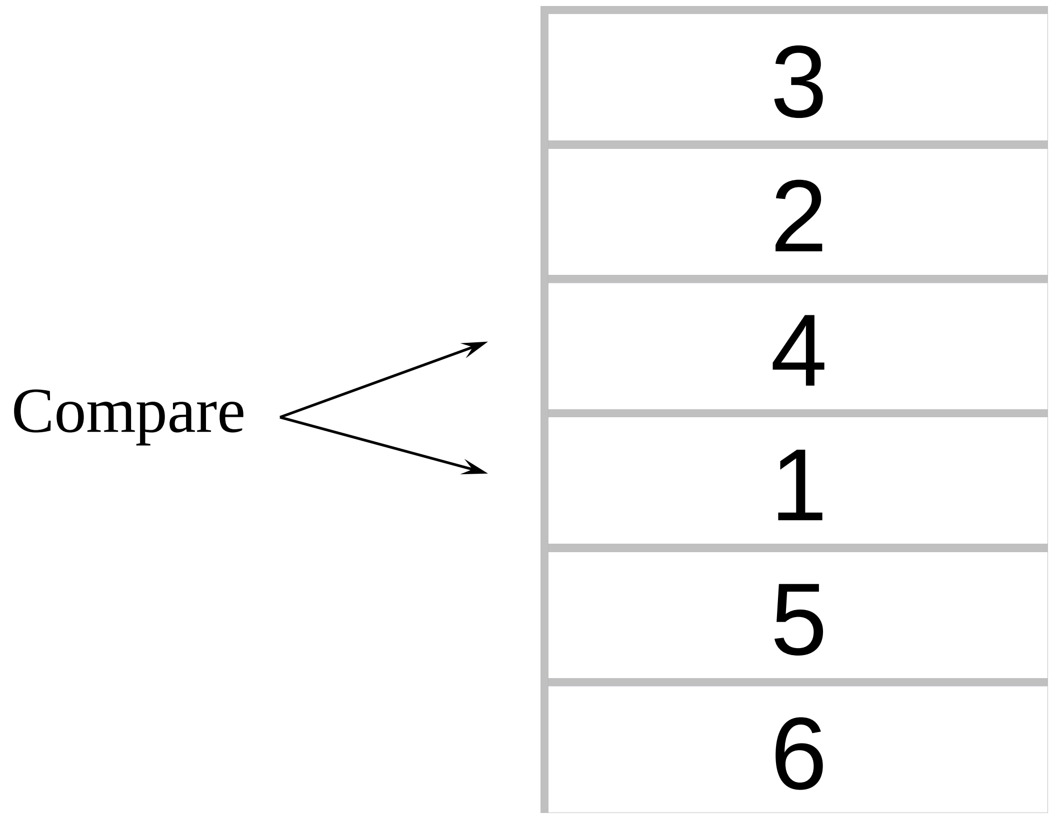
A Bubble Sort Example



A Bubble Sort Example

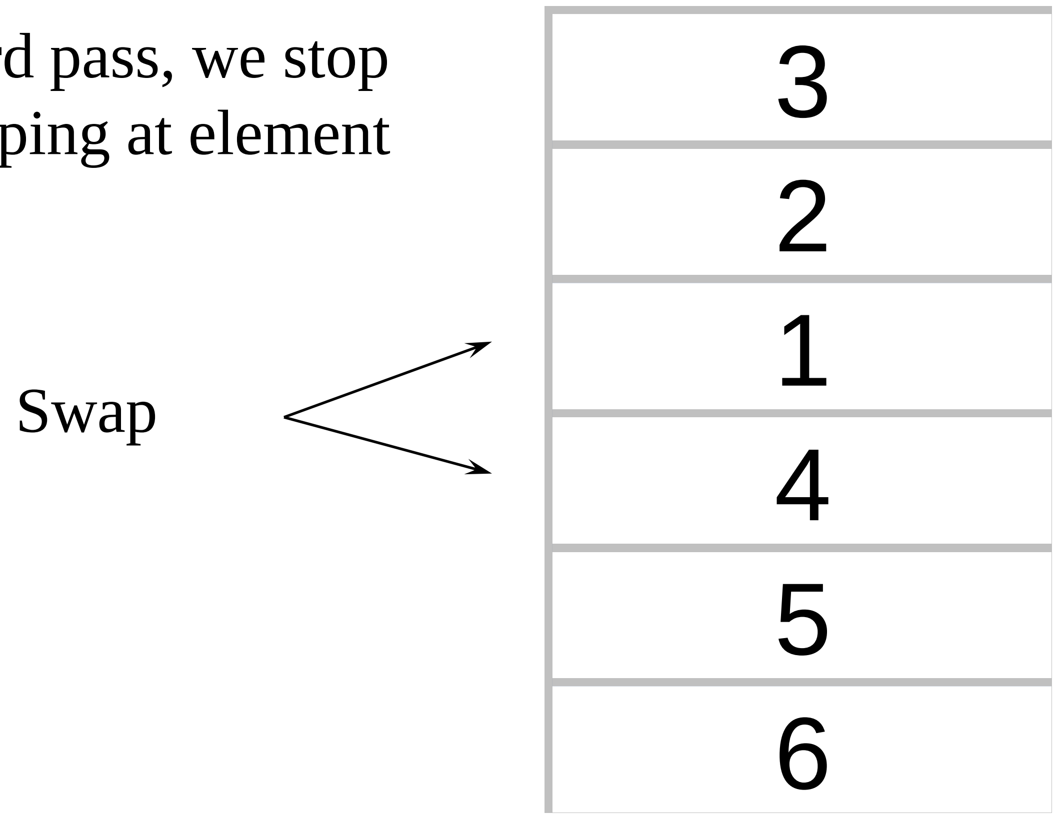


A Bubble Sort Example



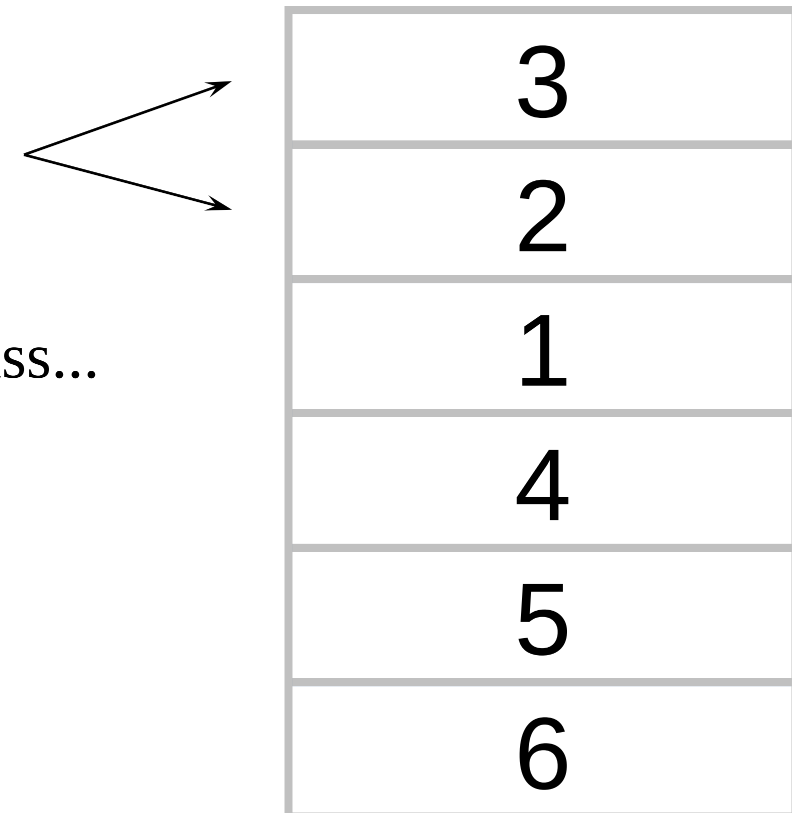
A Bubble Sort Example

At the end of the third pass, we stop comparing and swapping at element number n - 2.



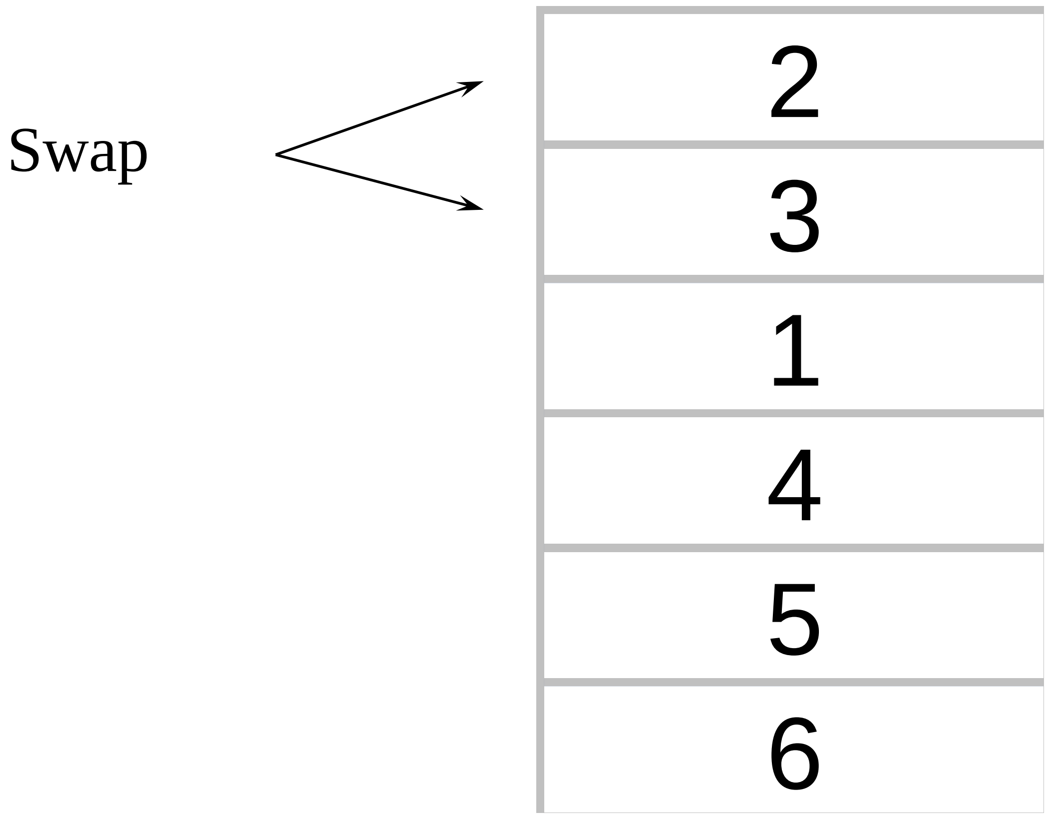
A Bubble Sort Example

Compare

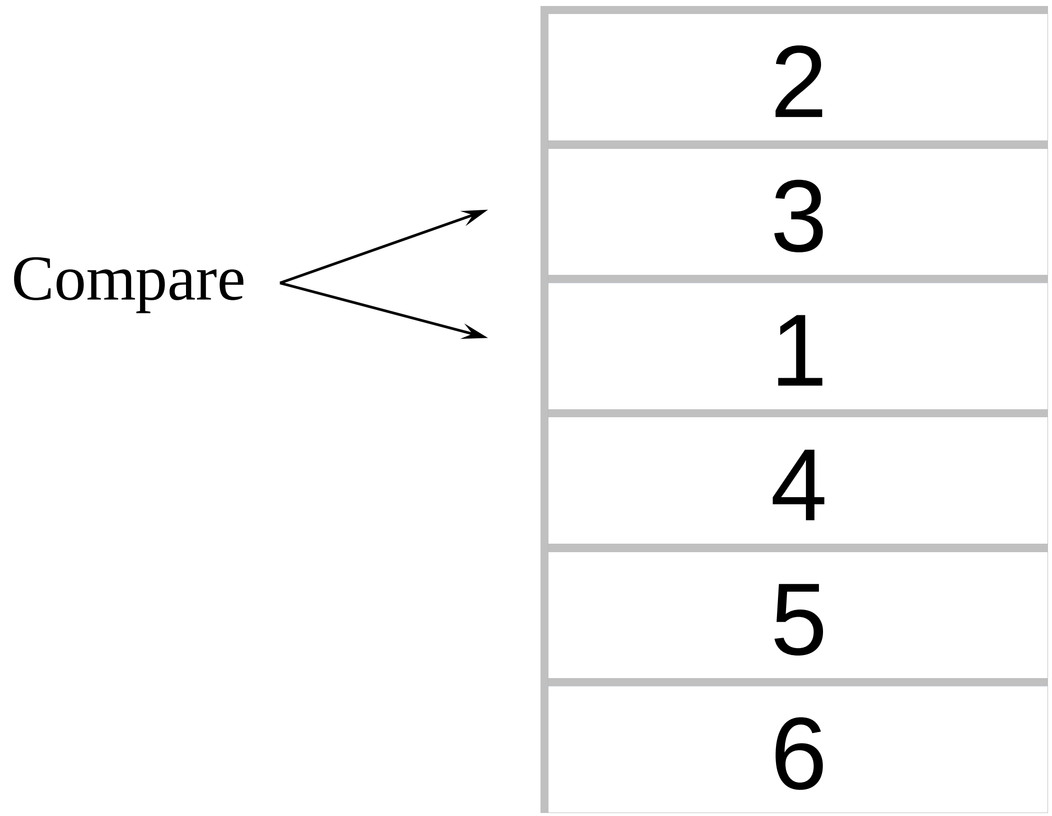


The beginning of the fourth pass...

A Bubble Sort Example

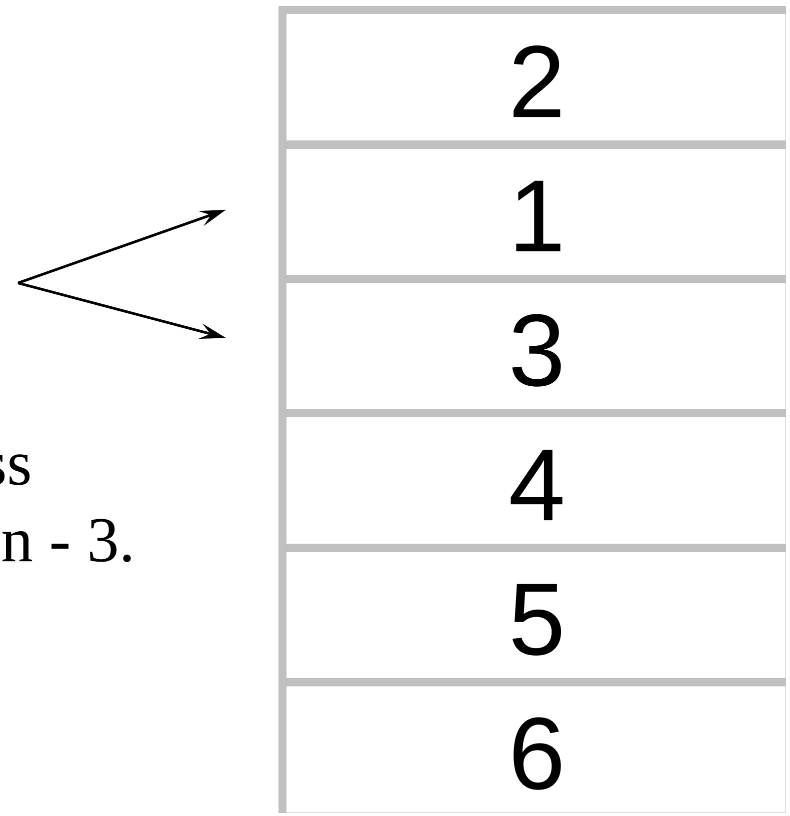


A Bubble Sort Example



A Bubble Sort Example

Swap



The end of the fourth pass stops at element number n - 3.

A Bubble Sort Example

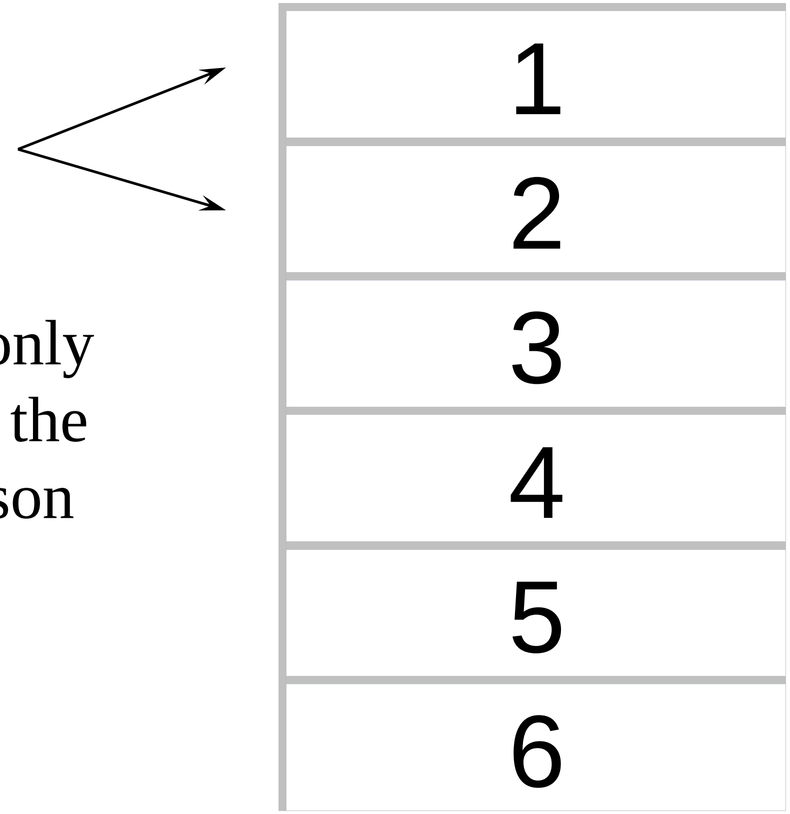
Compare

2
1
3
4
5
6

The beginning of the fifth pass...

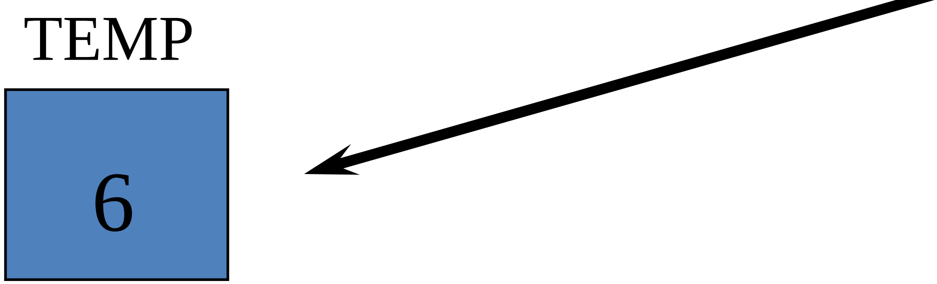
A Bubble Sort Example

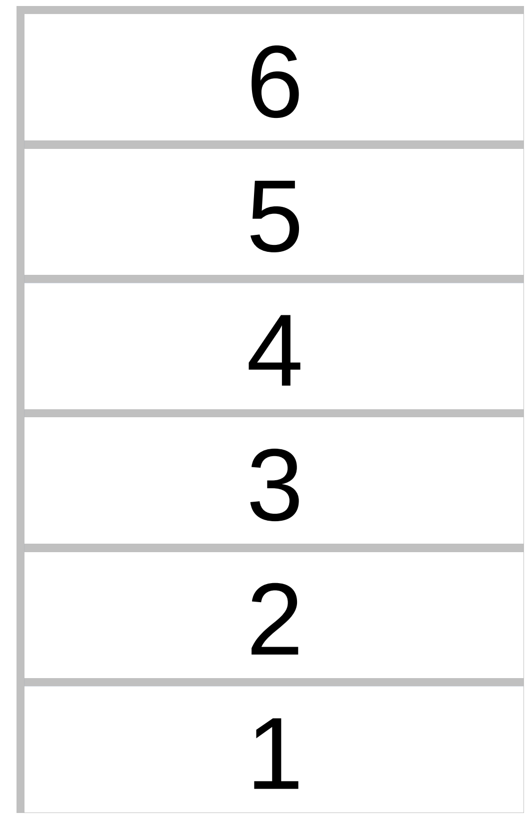
Swap



The <u>last pass</u> compares only the first two elements of the List. After this comparison and possible swap, the smallest element has “bubbled” to the top.

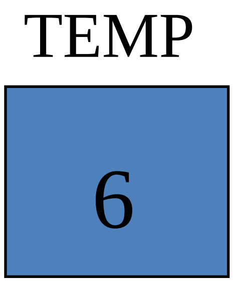
What “Swapping” Means



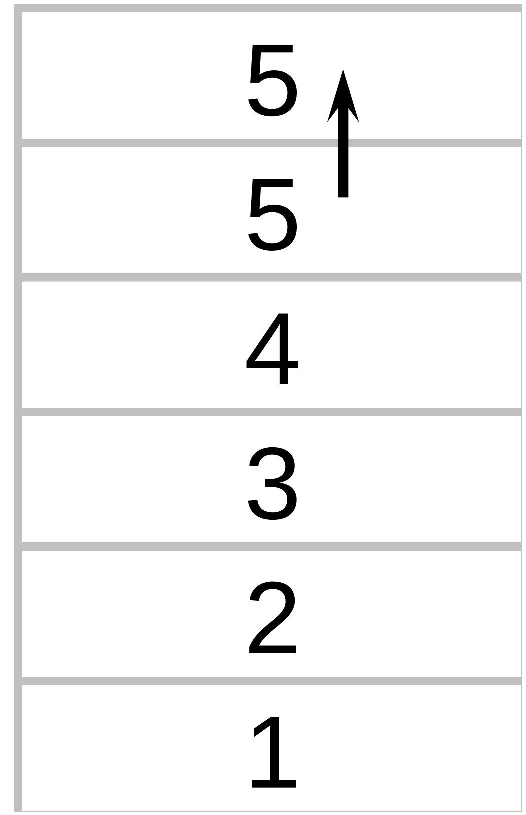


Place the first element into the Temporary Variable.

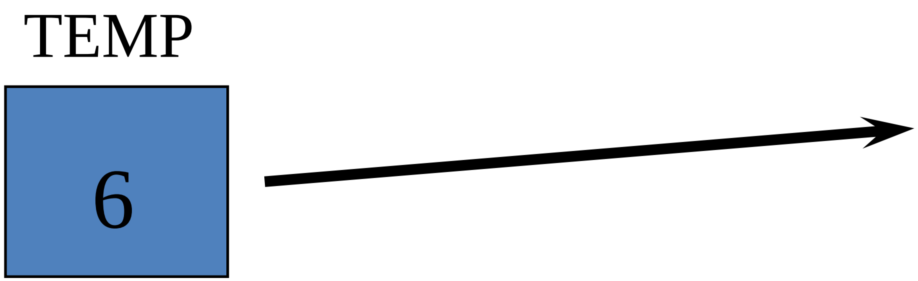
What “Swapping” Means

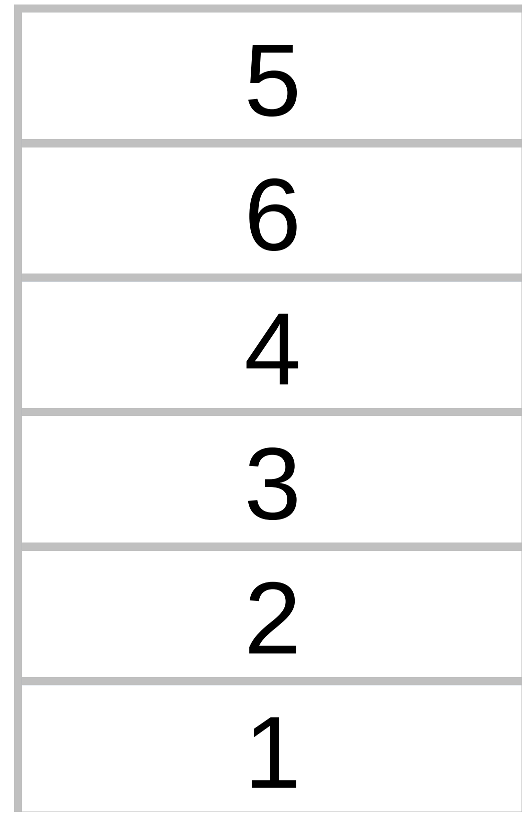


Replace the first element with the second element.



What "Swapping" Means





Replace the second element
with the Temporary Variable.

Bubble Sort

DATA is an array with N elements

[1] Repeat Step 2 and 3 for K =1 to N-1

[2] Set PTR :=1

[3] Repeat While PTR <= N –K

(a) If DATA[PTR] > DATA[PTR+1]

Interchange DATA[PTR] and DATA[PTR + 1]

(b) Set PTR = PTR + 1

[4] Exit

Analysis of Bubble Sort

$$f(n) = (n-1) + (n-2) + \dots + 2+1 = n(n-1)/2$$
$$= O(n^2)$$

Insertion Sort

An array A with N elements A[1], A[2] … A[N] is in memory

Insertion Sort scan A from A[1] to A[N] inserting each elements A[k] into its proper position in the previously sorted subarray A[1], A[2], …. A[K-1]

Pass 1: A[1] by itself is trivially sorted

Pass 2: A[2] is inserted either before or after A[1] so that A[1], A[2] is sorted

Pass 3: A[3] is inserted in its proper place in A[1], A[2], that is before A[1], between A[1] and A[2] or after A[2] so that A[1], A[2], A[3] is sorted.

Pass 4: A[4] is inserted in its proper place in A[1], A[2], A[3] so that A[1], A[2], A[3], A[4] is sorted.

.

.

Pass N: A[N] is inserted in its proper place in A[1], A[2], A[3], .. A[N-1] so that A[1], A[2], A[3], A[4] , A[N] is sorted.

Insertion Sort Example

Sort an array A with 8 elements

77, 33, 44, 11, 88, 66, 55

Insertion Sort

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K=1	-∞	77	33	44	11	88	22	66	55
K=2	-∞	77	33	44	11	88	22	66	55
K=3	-∞	33	77	44	11	88	22	66	55
K=4	-∞	33	44	77	11	88	22	66	55
K=5	-∞	11	33	44	77	88	22	66	55
K=6	-∞	11	33	44	77	88	22	66	55
K=7	-∞	11	22	33	44	77	88	66	55
K=8	-∞	11	22	33	44	66	77	88	55
Sorted	-∞	11	22	33	44	55	66	77	88

Insertion Algorithm

This algorithm sort an array with N elements

[1] Set A[0] = $-\infty$ [Initialize a delimiter]

[2] Repeat Steps 3 to 5 for K = 2, 3, …, N

[3] Set TEMP = A[K] and PTR = K-1

[4] Repeat while TEMP < A[PTR]

(a) Set A[PTR+1] = A[PTR]

(b) Set PTR = PTR -1

[5] Set A[PTR+1] = TEMP

[6] Exit

Complexity of Insertion Sort

Worst Case: $O(n^2)$

Average Case: $O(n^2)$

Properties

- Stable
- O(1) extra space
- $O(n^2)$ comparisons and swaps
- Adaptive: O(n) time when nearly sorted
- Very low overhead

Discussion

- Although it is one of the elementary sorting algorithms with $O(n^2)$ worst-case time, insertion sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead).
- For these reasons, and because it is also stable, insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

Selection Sort

Suppose an array A with N elements is in memory. Selection sort works as follows

First find the smallest element in the list and put it in the first position. Then, find the second smallest element in the list and put it in the second position and so on.

Pass 1: Find the location LOC of the smallest element in the list A[1], A[2], ... A[N]. Then interchange A[LOC] and A[1]. Then: A[1] is sorted

Pass 2: Find the location LOC of the smallest element in the sublist A[2], A[3], ... A[N]. Then interchange A[LOC] and A[2]. Then: A[1],A[2] is sorted since A[1] <= A[2].

Pass 3: Find the location LOC of the smallest element in the sublist A[3], A[4], ... A[N]. Then interchange A[LOC] and A[3]. Then: A[1], A[2],A[3] is sorted, since A[2] <= A[3].

Pass N-1: Find the location LOC of the smallest element in the sublist A[N-1], A[N]. Then interchange A[LOC] and A[N-1]. Then: A[1], A[2], ….. , A[N] is sorted, since A[N-1] <= A[N].

A is sorted after N-1 pass.

Selection Sort

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K=1 LOC=4	77	33	44	11	88	22	66	55
K=2 LOC=6	11	33	44	77	88	22	66	55
K=3 LOC=6	11	22	44	77	88	33	66	55
K=4 LOC=6	11	22	33	77	88	44	66	55
K=5 LOC=8	11	22	33	44	88	77	66	55
K=6 LOC=7	11	22	33	44	55	77	66	88
K=7 LOC=4	11	22	33	44	55	66	77	88

Sorted	11	22	33	44	55	66	77	88

Complexity

$$f(n) = (n-1) + (n-2) + \ldots\ldots + 2 + 1 = n(n-1)/2$$

$$= O(n^2)$$

Properties

- Not stable
- $O(1)$ extra space
- $\Theta(n^2)$ comparisons
- $\Theta(n)$ swaps
- Not adaptive

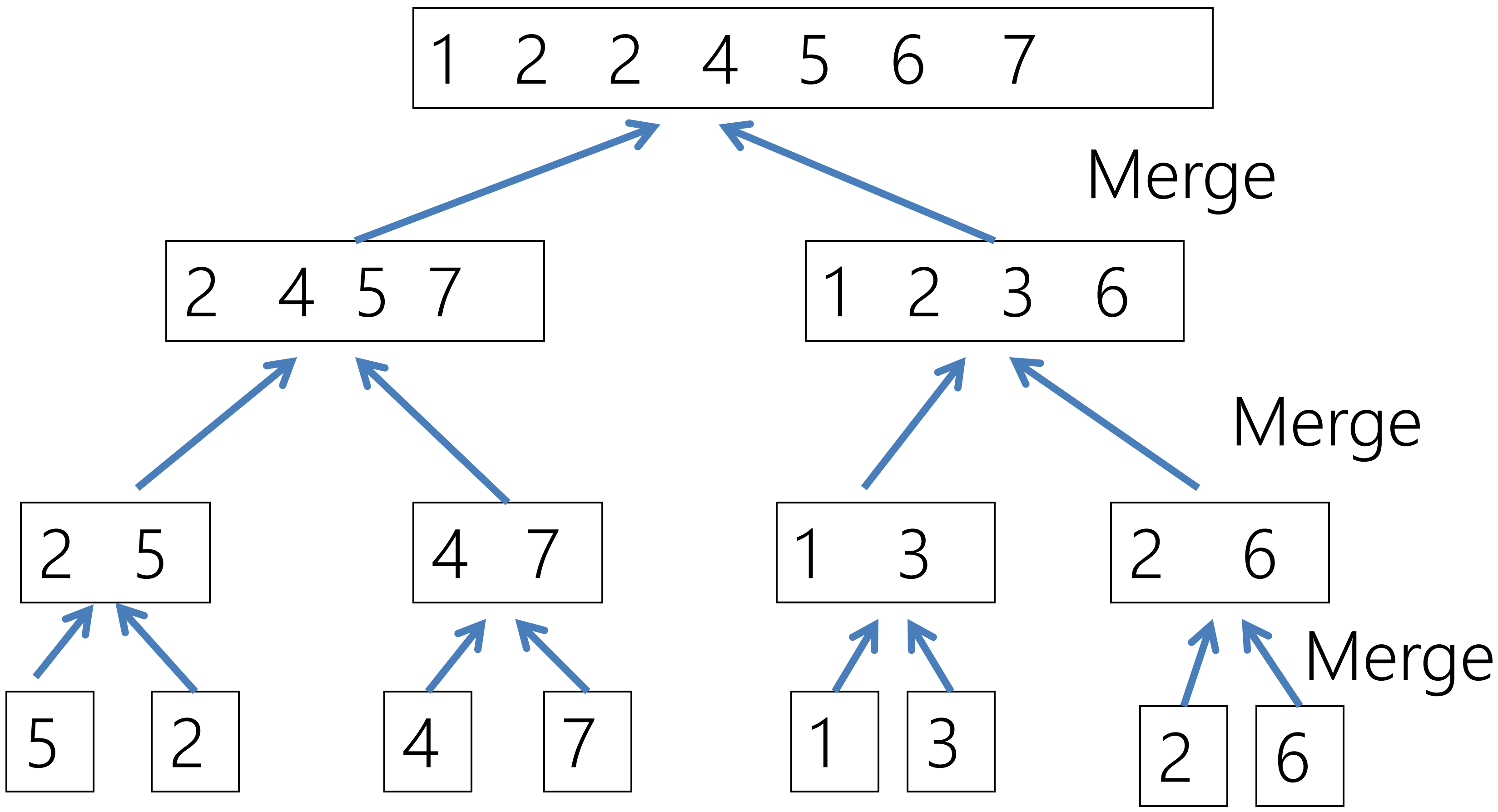
Discussion

- From the comparisons presented here, one might conclude that selection sort should never be used. It does not adapt to the data in any way (notice that the four animations above run in lock step), so its runtime is always quadratic.
- However, selection sort has the property of minimizing the number of swaps. In applications where the cost of swapping items is high, selection sort very well may be the algorithm of choice.

Merge Sort

Suppose the array A containing 8 elements

5, 2, 4, 7, 1, 3, 2, 6



1 2 2 4 5 6 7
Merge
2 4 5 7
1 2 3 6
Merge
2 5
4 7
1 3
2 6
Merge
5
2
4
7
1
3
2
6

Time complexity = $\Theta(n \log n)$

Quick Sort

To select the pivot

Select the first number FIRST in the list,

beginning with the last number in the list, scan from right to left, comparing with each number and stopping at the first number less than FIRST.

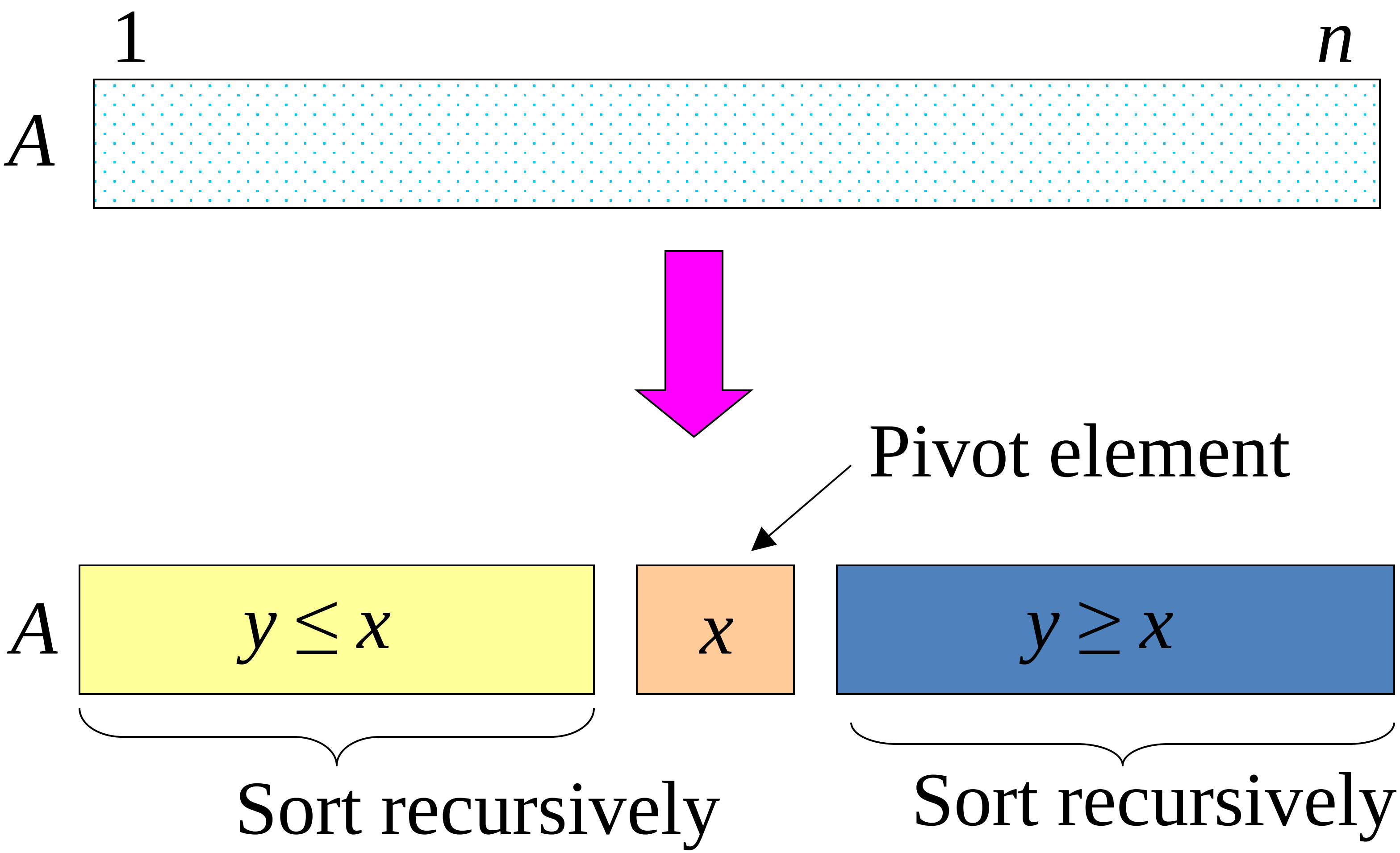
Then interchange the two number.

Quick Sort

Quick sort is an algorithm of the divide-and-conquer type

The problem of sorting a set is reduced to the problem of sorting two smaller sets.

Quick Sort Approach



44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

22, 33, 11, 55, 77, 90, 40, 60, 99, 44, 88, 66

22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66

22, 33, 11, 40, 77, 90, 44, 60, 99, 55, 88, 66

22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

$$\underbrace{22,\ 33,\ 11,\ 40}_{\text{First Sublist}},\ 44,\ \underbrace{90,\ 77,\ 60,\ 99,\ 55,\ 88,\ 66}_{\text{Second SubList}}$$

Quick Sort Algorithm

Input: Unsorted sub-array A[first..last]

Output: Sorted sub-array A[first..last]

QUICKSORT (A, first, Last)

    if first < last

        then **loc**← **PARTITION**(*A*, first, *last*)

            QUICKSORT (A, first, loc-1)

            QUICKSORT (A, loc+1, last)

Partition Algorithm

Input: Sub-array A[first..last]

Output: Sub-array A[first..loc] where each element of A[first..loc-1] is ≤ to each element of $A[(q+1)..r]$; returns the index ***loc***

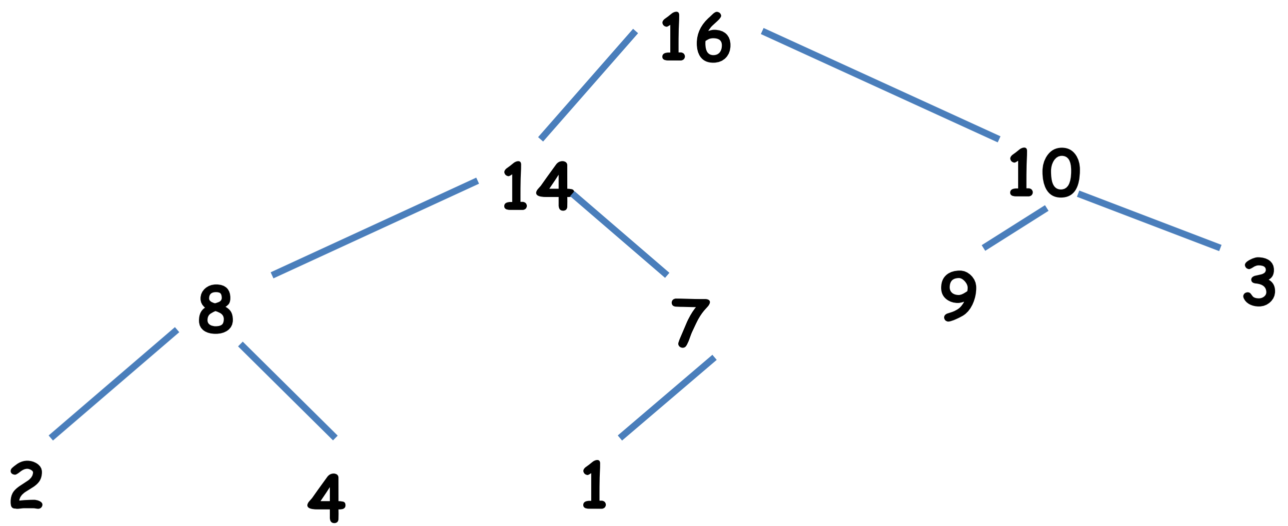
PARTITION (A, first, last)

```
x ← A[first]
i ← first -1
j ← last +1
while TRUE
        repeat j ← j - 1
            until A[j] ≤ x
        repeat i ← i + 1
            until A[i] ≥ x
        if i < j
            then exchange A[i] ↔ A[j]
            else return j
```

Heap Sort

4, 1, 3, 2, 16, 9, 10, 14, 8, 7

Heap



Heap