Sorting

Sorting Algorithms

"Inefficient" Algorithms:

- Bubble Sort
- Insertion Sort
- Selection Sort

"Efficient Algorithms"

- Merge Sort
- Quick Sort
- Radix Sort
- Tim Sort
- …

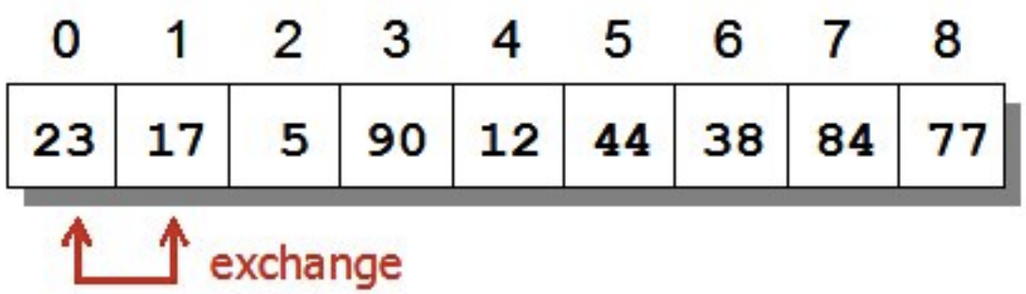
Stability:

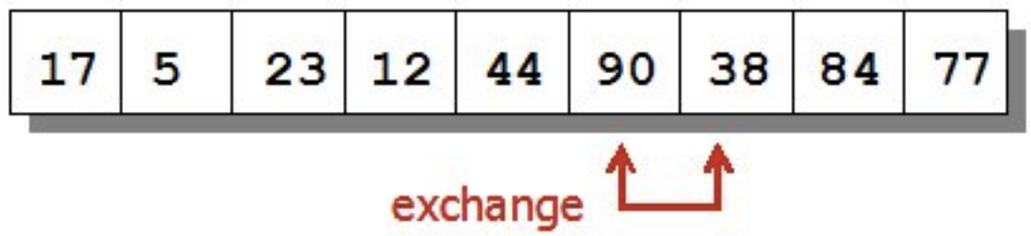
A sorting algorithm is stable if it maintains the relative position of equal objects.

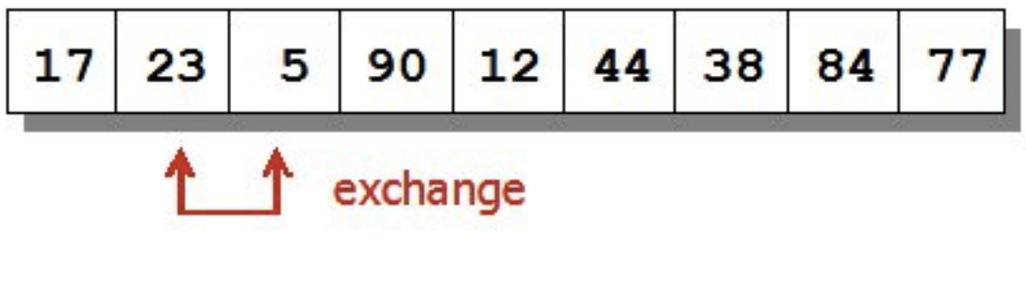
Refer to Chp. 11

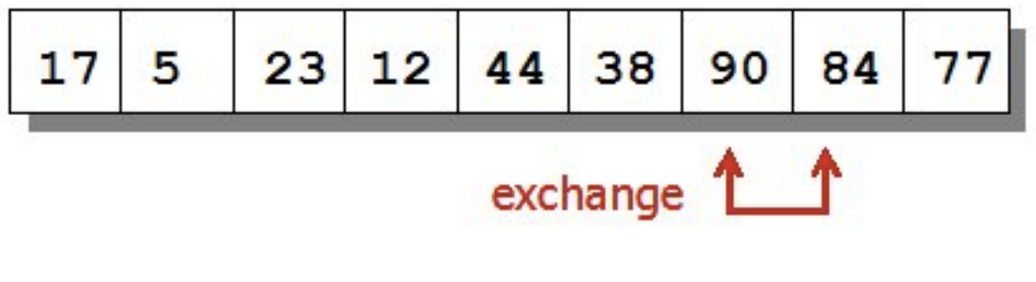
Bubble Sort

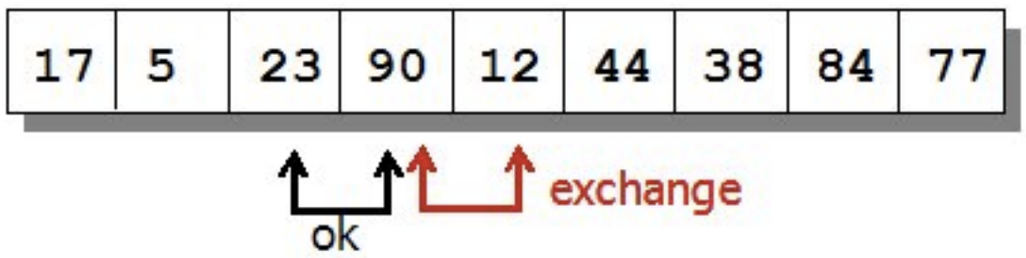
Bubble sort is based on comparison among adjacent objects:

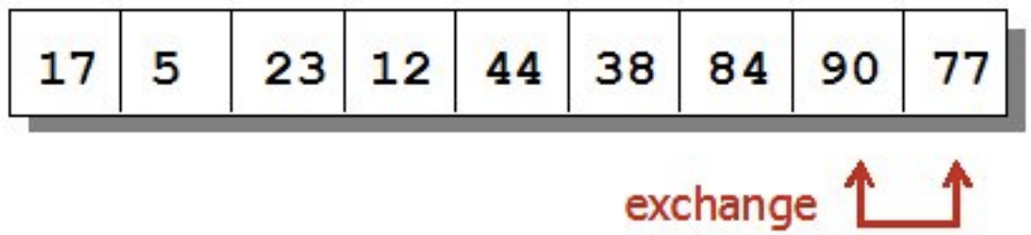


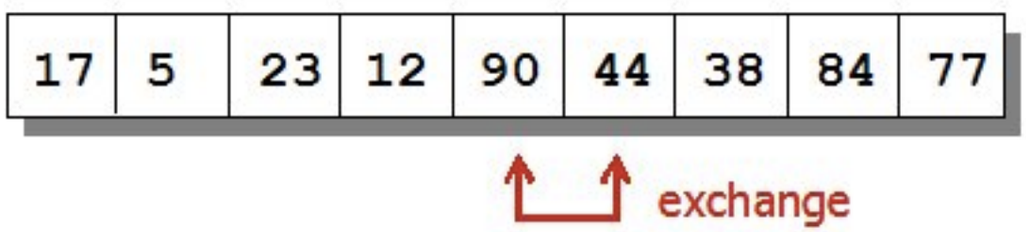


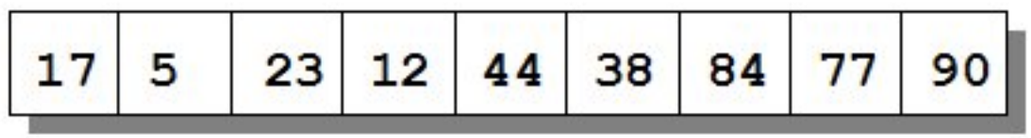






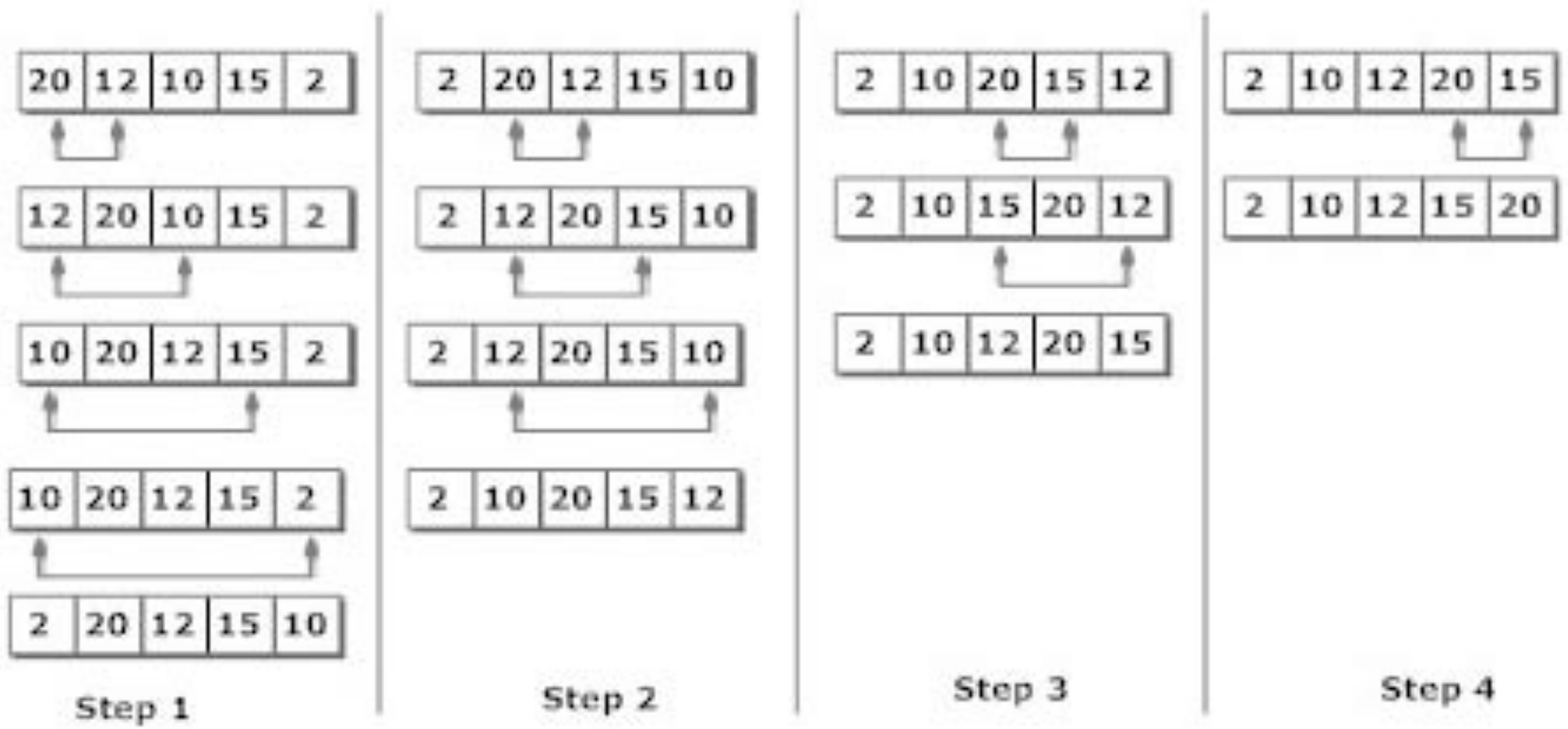






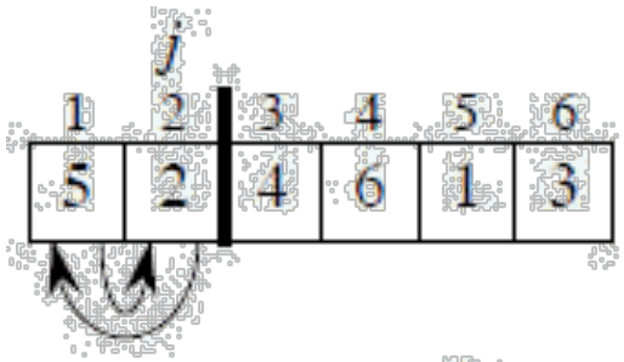
Selection Sort

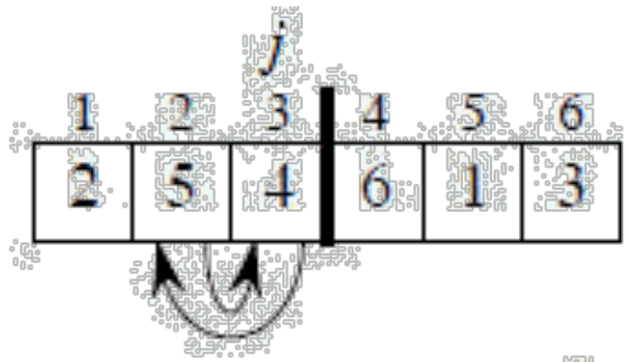
Similar to sorting a hand of cards: find the smallest element and swap it with the first element. Find the next smallest element and swap it with the second, …

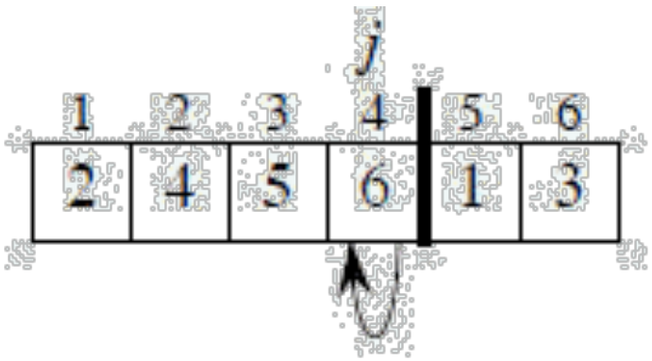


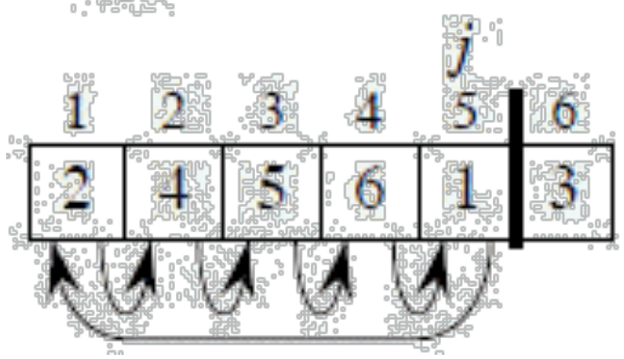
Insertion Sort

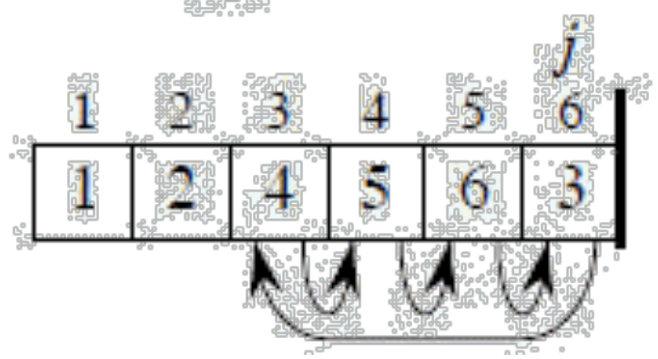
Partition the array in sorted and unsorted regions.
At each step, pick an element of the unsorted region and insert it at the right place in the sorted region.

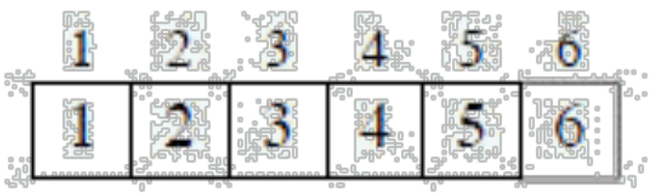










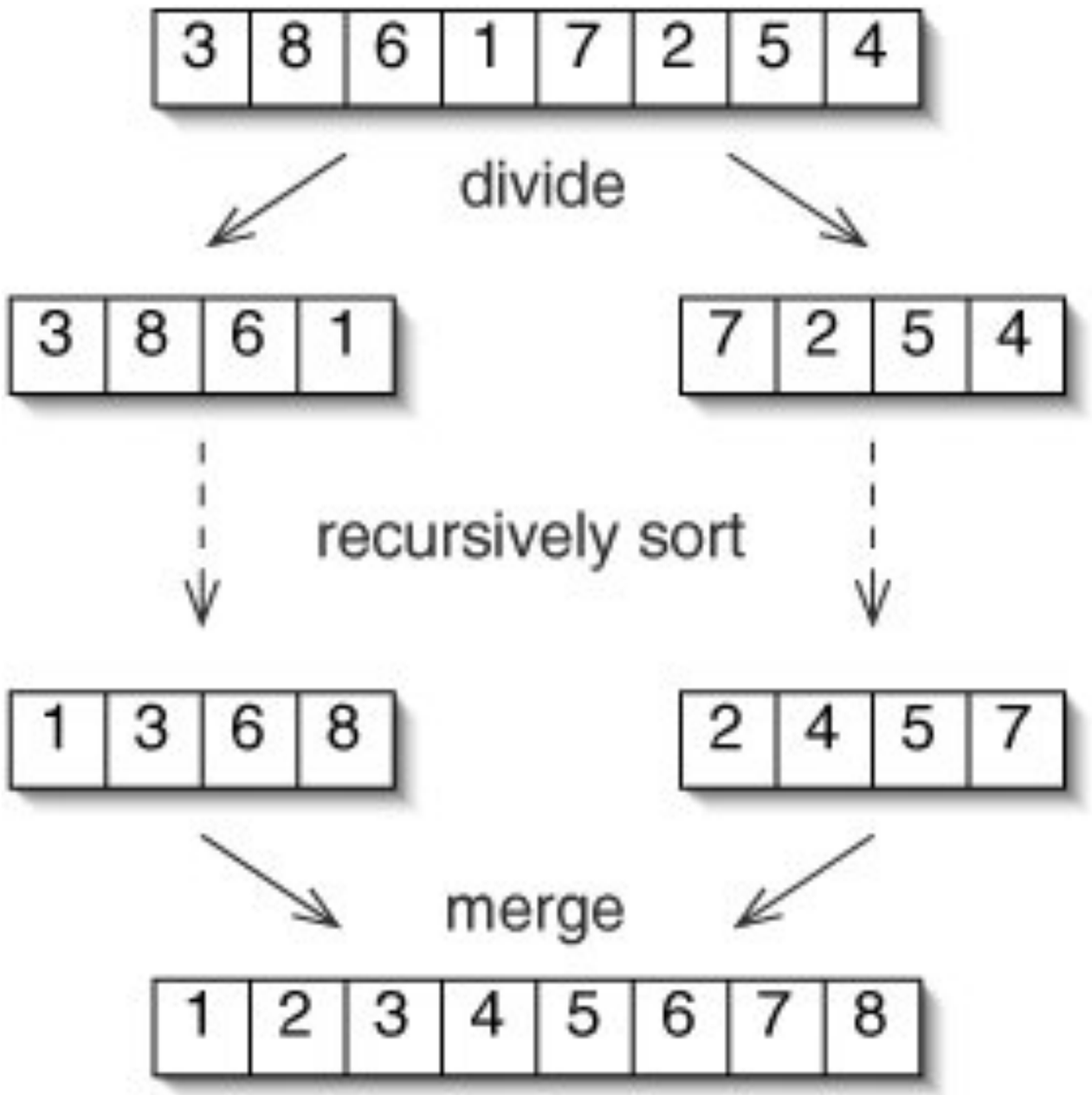


Merge Sort

<u>Divide-and-conquer algorithm:</u>
Divide the array in two and recursively sort and merge the results:

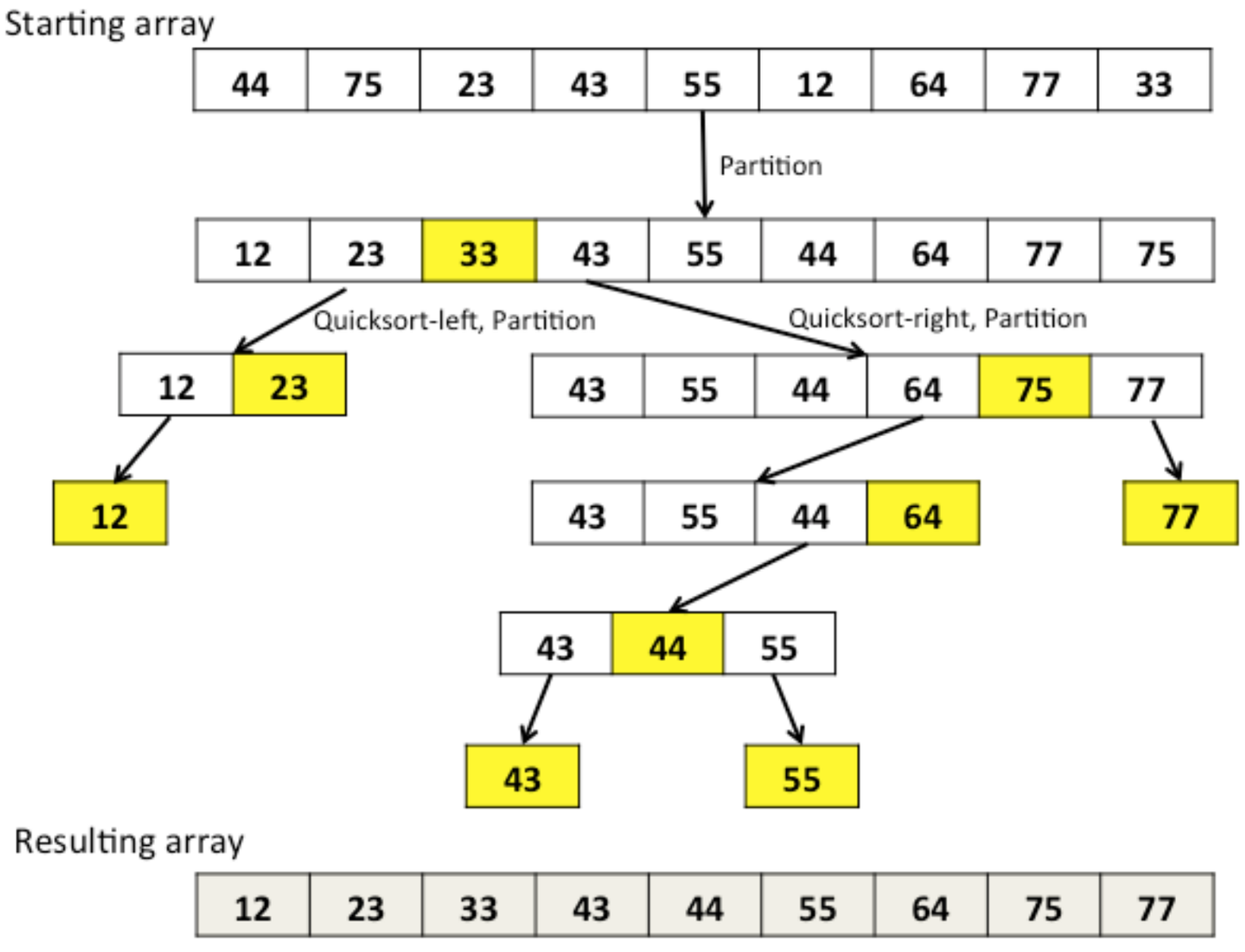
1) Divide in halves
2) Sort the halves (recursively!)
3) Merge the halves in a tmp vector
4) Copy the tmp in the original

Recursive implementation.



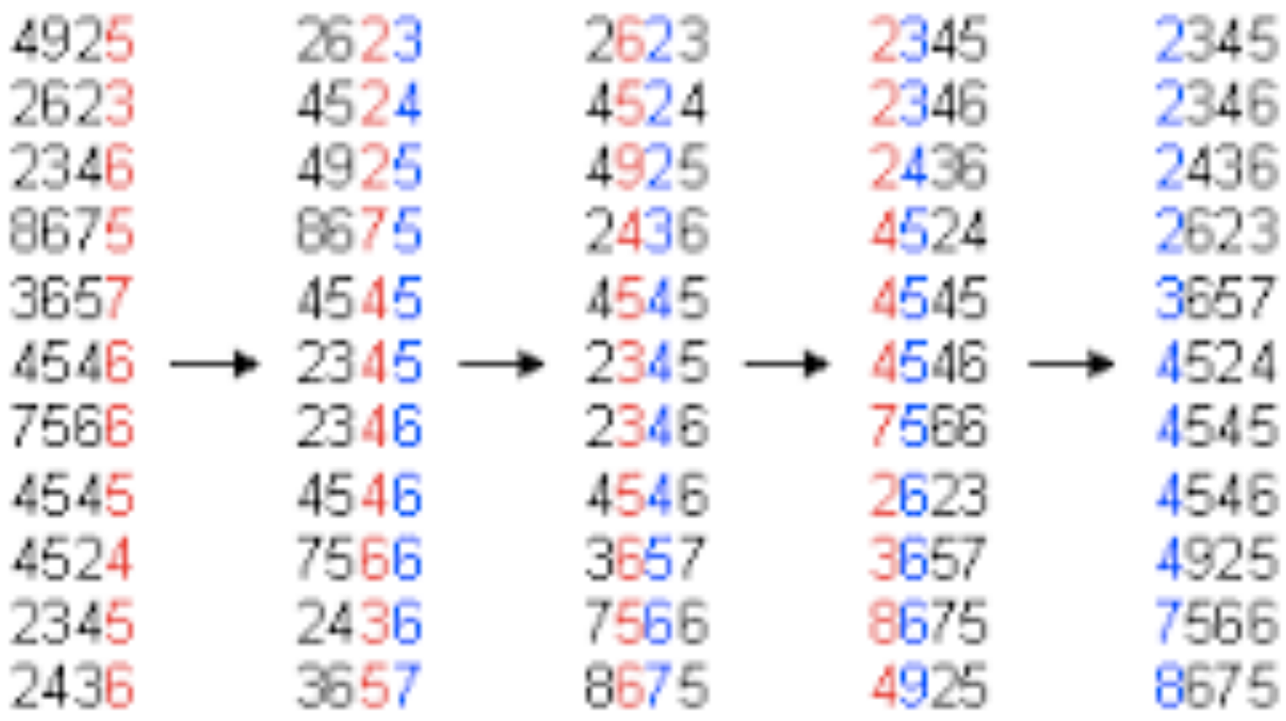
Quick Sort

Divide-and-conquer algorithm:



Radix Sort

Considers the elements as strings. First it orders according to the last digit, then the second to last, etc..



Tim Sort

Tim-Sort (Tim Peters, 2002) is an hybrid (combines more methods) adaptive (chooses the method dynamically) sorting algorithm which is used e.g. as standard sorting routine in the Python language.

It is stable and combines merge sort with insertion sort.
It is designed for good performance on real-world data.

The idea behind it is to look for “natural runs”, which are short sub-sequences in the data which are already ordered.

The algorithms then merges intelligently (making use also of binary searches) the ordered runs.

A good simple reference is Wikipedia https://en.wikipedia.org/wiki/Timsort

Computational Complexity

Sorting Algorithm	Average Time	Best Time	Worst Time	Extra Space (in-place)	Stability
Bubble Sort	O(n^2)	O(n)	O(n^2)	O(1)	Stable
Insertion Sort	O(n^2)	O(n)	O(n^2)	O(1)	Stable
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Unstable
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(n)	Unstable
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Stable
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	Unstable
Radix Sort	O(n)	O(n)	O(n)	O(n)	Stable
Counting Sort	O(n)	O(n)	O(n)	O(n+k)	Stable
Tim Sort	O(n log n)	O(n)	O(n log n)	O(n) In the worst case	Stable

Measuring the time of a routine

```
#include <time.h>

clock_t start,diff;

start = clock(); //read the clock

//do something

diff = clock() - start; //re-read and calc. time difference

double elapsed_time = diff / (double)CLOCKS_PER_SEC; //time in s
```

Generating Random Numbers

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

`srand ((unsigned)time(NULL))` : sets the random sequence's seed.

`rand()`: generates a random number between 0 and RAND_MAX

`rand()/(double)RAND_MAX` : generates a random number between 0 and 1

`rand()%N` : generates an integer random number between 0 and N.

Computational Complexity: Try to estimate it!

1) Choose a “slow” and a “fast” sorting algorithms.

2) Implement them in a single C++ program

3) Create an array filled with random numbers

4) Sort with the two algorithms the array increasing its length N up to a “big” number.

5) Measure the time taken by the sorting algorithms

6) Plot on a chart the time (y) as a function of N (x).

7) Do the obtained curves respect the expected theoretical scaling?