Sorting

What is Sorting?

Sorting: an operation that segregates items into groups according to specified criterion.

A = { 3 1 6 2 1 3 4 5 9 0 }

A = { 0 1 1 2 3 3 4 5 6 9 }

Why Sort and Examples

Consider:

- Sorting Books in Library (Dewey system)
- Sorting Individuals by Height (Feet and Inches)
- Sorting Movies in Blockbuster (Alphabetical)
- Sorting Numbers (Sequential)

Types of Sorting Algorithms

There are many, many different types of sorting algorithms, but the primary ones are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Radix Sort
- Swap Sort

Review of Complexity

Most of the primary sorting algorithms run on different space and time complexity.

Time Complexity is defined to be the time the computer takes to run a program (or algorithm in our case).

Space complexity is defined to be the amount of memory the computer needs to run a program.

Time Efficiency

- How do we improve the time efficiency of a program?

- **The 90/10 Rule**

  90% of the execution time of a program is spent in executing 10% of the code

- So, how do we locate the **critical 10%**?

  - software metrics tools
  - global counters to locate bottlenecks (loop executions, function calls)

Time Efficiency Improvements

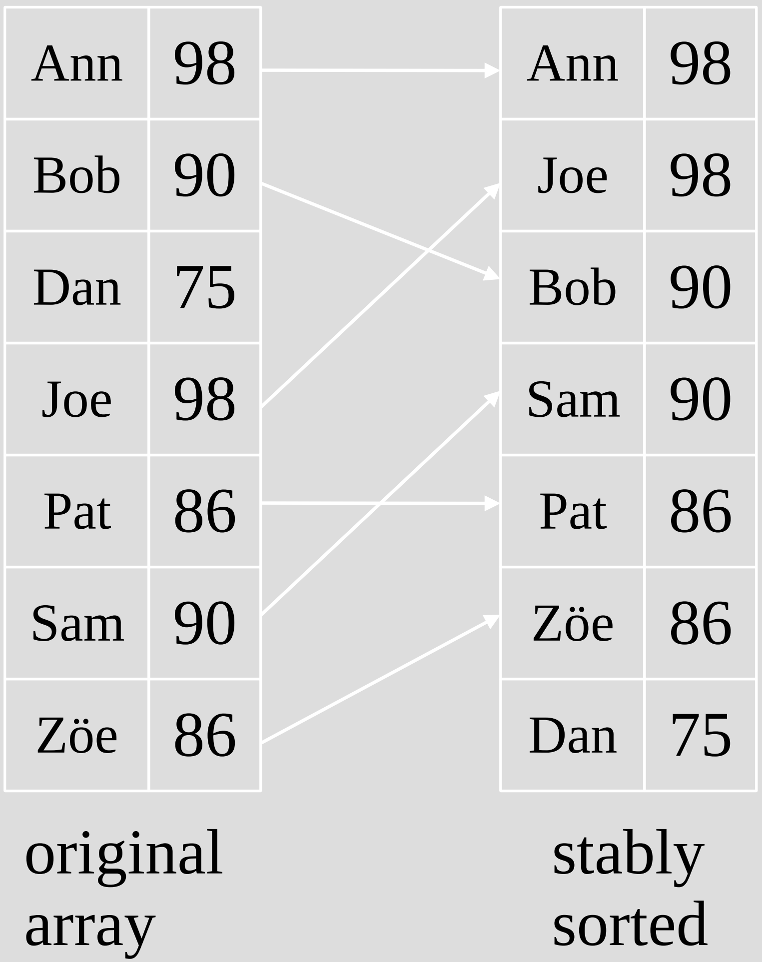
Possibilities (some better than others!)

- Move code out of loops that does not belong there (just good programming!)
- Remove any unnecessary I/O operations (I/O operations are expensive time-wise)
- Code so that the compiled code is more efficient

Moral - Choose the most appropriate algorithm(s) BEFORE program implementation

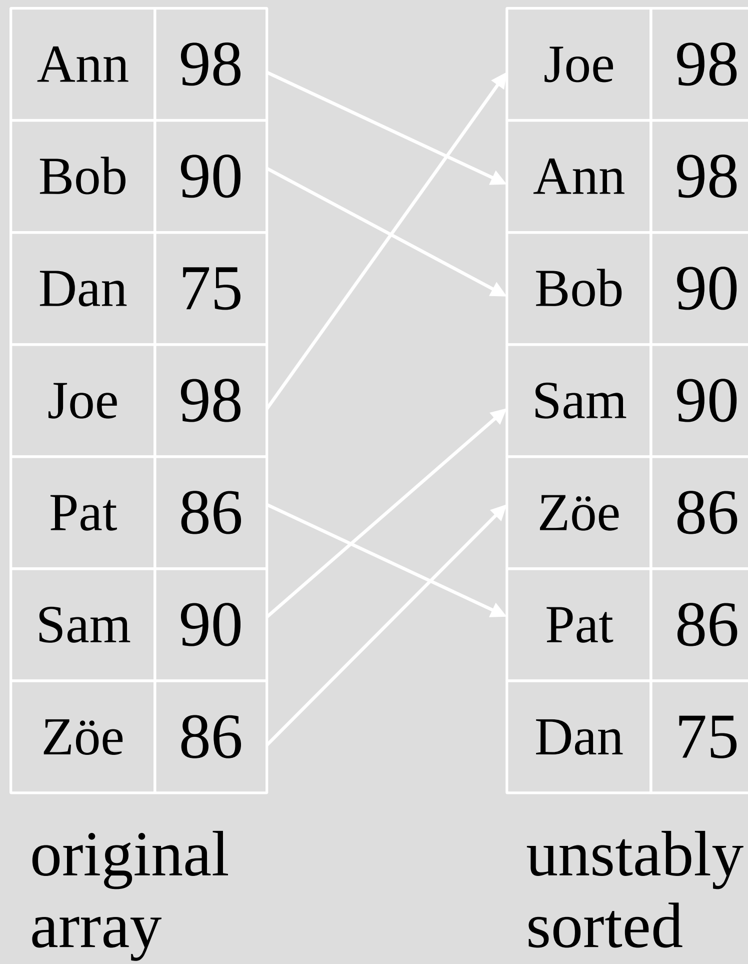
Stable sort algorithms

- 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



Unstable sort algorithms

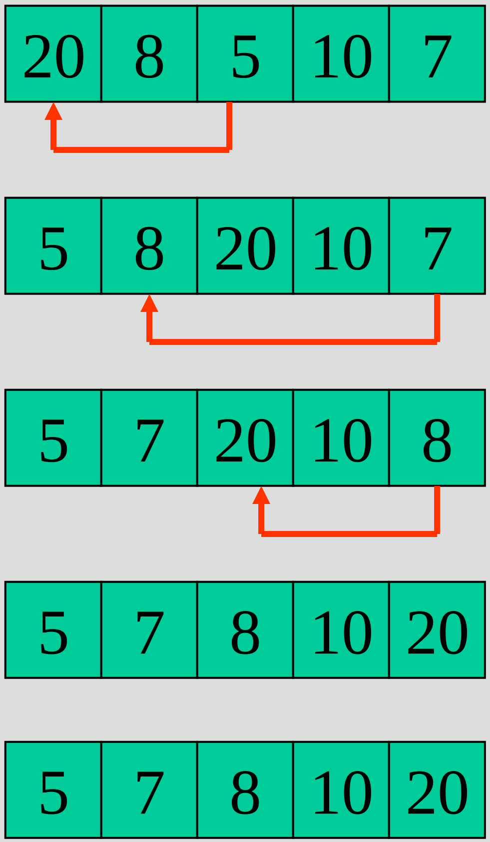
- An unstable sort may or may not keep equal elements in the same order
- Stability is usually not important, but sometimes it is important



Selection Sorting

Step:

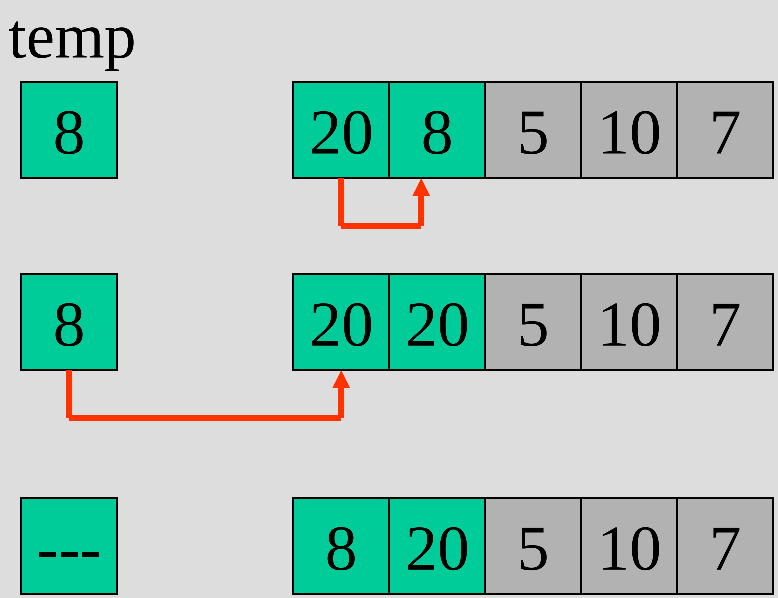
- 1. select the smallest element
- among *data[i]~ data[data.length-1];*
- *2.* swap it with *data[i]*;
- 3. if not finishing, repeat 1&2



Insertion Sorting

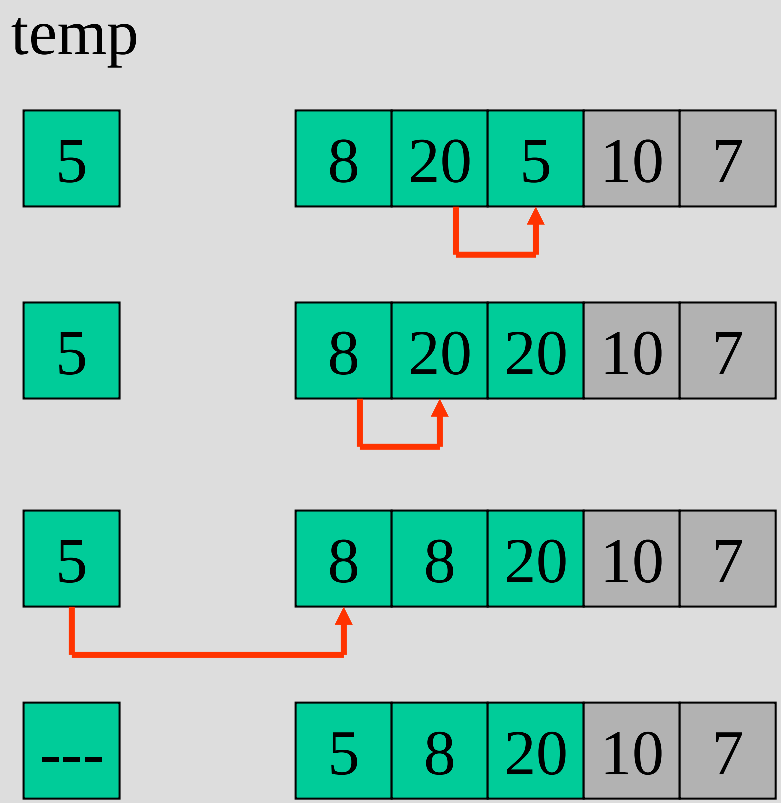
- Place i*th* item in proper position:
  - *temp = data[i]*
  - shift those elements *data[j]* which greater than *temp* to right by one position
  - place *temp* in its proper position

Insert Action: i=1



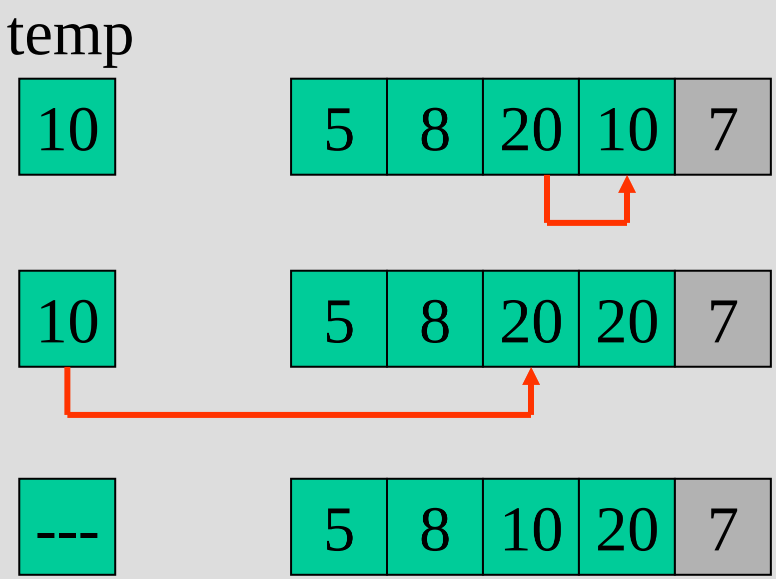
i = 1, first iteration

Insert Action: i=2



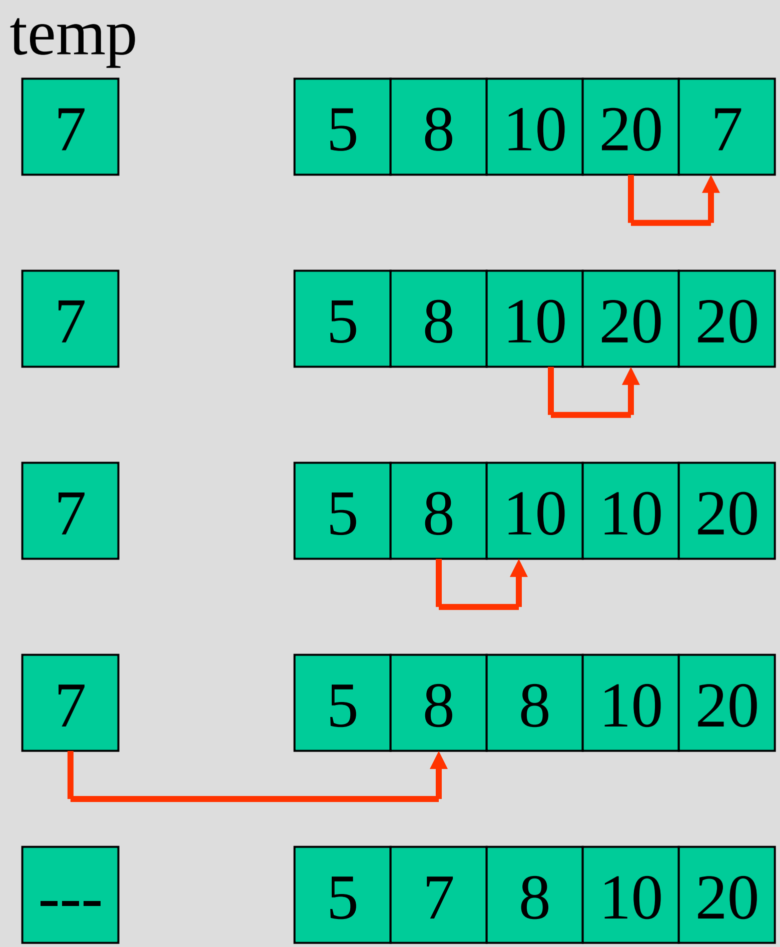
i = 2, second iteration

Insert Action: i=3



i = 3, third iteration

Insert Action: i=4



i = 4, forth iteration