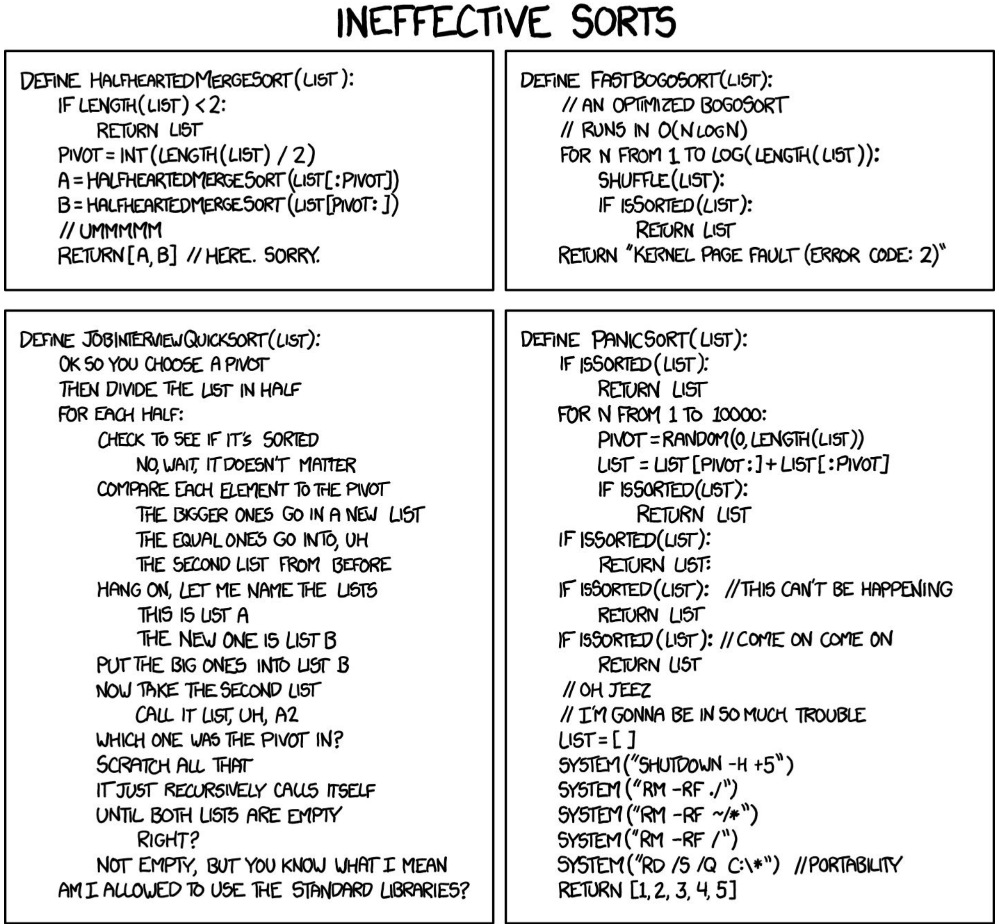
University of Colorado **Boulder**

CSCI 1300: Starting Computing

Spring 2019 Tony Wong

Lecture 33: Sorting



INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

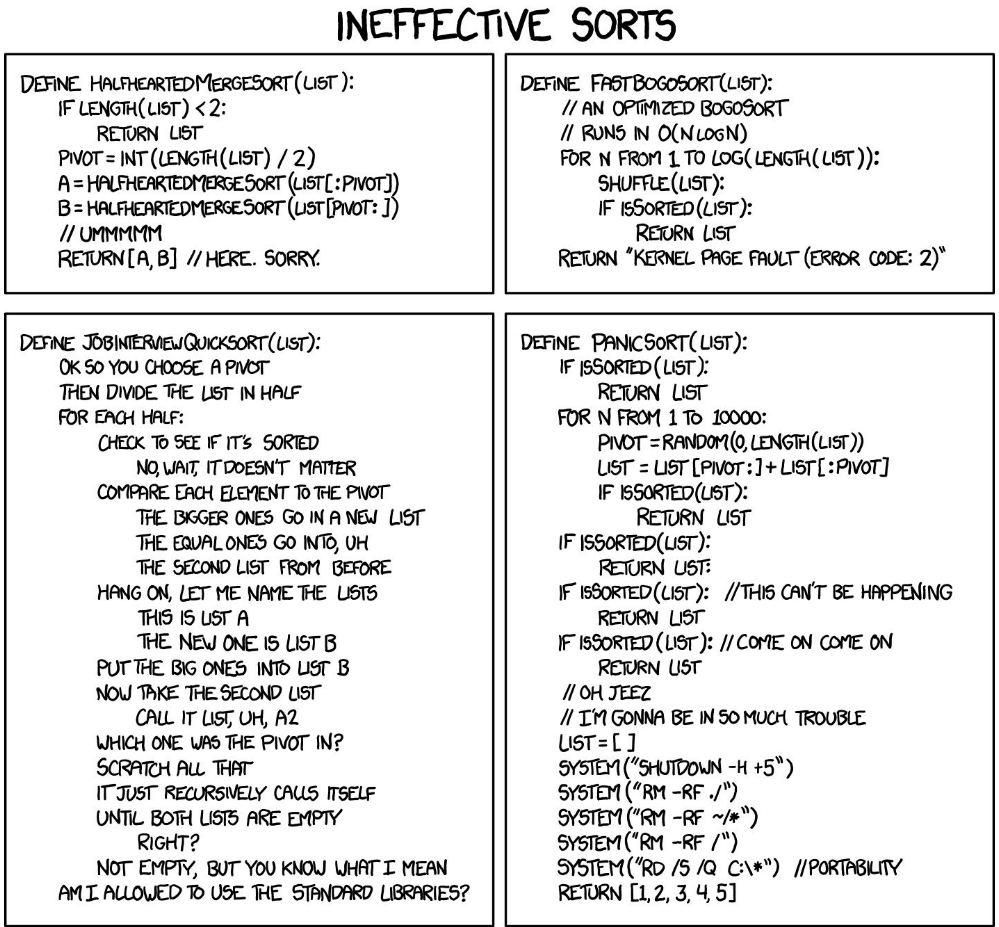
```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): //COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

Announcements and reminders

- Project 3 TA/CA design meeting
  (sign up on Moodle!)
  → **due Wednesday (10 Apr) by 6 PM**
- … & submit classes and code skeleton
  → **due Wednesday (10 Apr) by 11 PM**
- Project 2 interview grading
  (sign up on Moodle!)
- Homework 8
  → **due Saturday (6 Apr) by 6 PM**



```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOGN)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): //COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

Project 3 design meeting

Question: What do I need?

Answer: @1130 on Piazza:

Game of Thrones:

- Class diagrams: on paper, or electronically
- Classes skeleton -- data members, functions, pseudocode, game flow
- Main class
  - functionality of the game
  - initializers
  - game flow
  - endOfGame function
  - etc...

Choose Your Own:

- Your idea: what's the story (typed up)
- Class diagrams: on paper, or electronically
- Classes skeleton -- data members, functions, pseudocode
- Main class:
  - functionality of the game
  - game flow
  - endOfGame function
  - etc...
- How will you meet requirements:
  - what will you read from a file?
  - Where does your data coming from?
  - what will you write to a file?

Last time on *Intro Computing…*

We looked at **what to do when we're handed a big project**

- Identify what are the key **structures**
- … and how those structures **relate** to one another
- Identify what are the key **functions**
- … and how these functions are related to our structures

One of the key helper functions: **sorting** a vector

Last time: **selection sort** (Special Topic 6.2)

Input: **X** = [13, 3, 9, 5, 1]

Output: The **sorted** version of **X**, in increasing order: [1, 3, 5, 9, 13]

Step 1: Find the smallest element out of X[0 - end].
Swap X[0] and smallest element.

Step 2: Find the smallest element out of X[1 - end].
Swap X[1] and smallest element.

Step 3: Find the smallest element out of X[2 - end].
Swap X[2] and smallest element.

And so on…

This time: more sorts of sorting!

Sorting is a key active area of study in computer science.

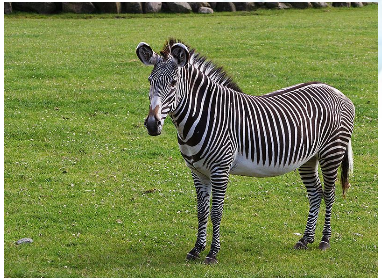
Task: Given some unordered list of elements, organize them according to some notion of “order” (e.g., increasing numbers, alphabetizing, etc…)

Applications: Sort mail by location along route.
Alphabetizing CSCI 1300 students by name.
Sorting a list of household incomes

Goals: Sometimes the whole point is just to sort the lists.

Examples:

- do a binary search
- find duplicates in a list
- **compute the median**



This time: more sorts of sorting!

We will look a handful of common algorithms.

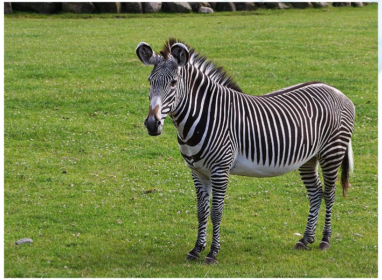
… some we'll look at in detail (actually code!) … and others we will just talk through.

The point: give you a feel for what's out there.

→ You'll study sorting algorithms more deeply in … well… *Algorithms.*

Overview:

1) Bubble sort
2) Cocktail sort
3) Merge sort
4) Quick sort

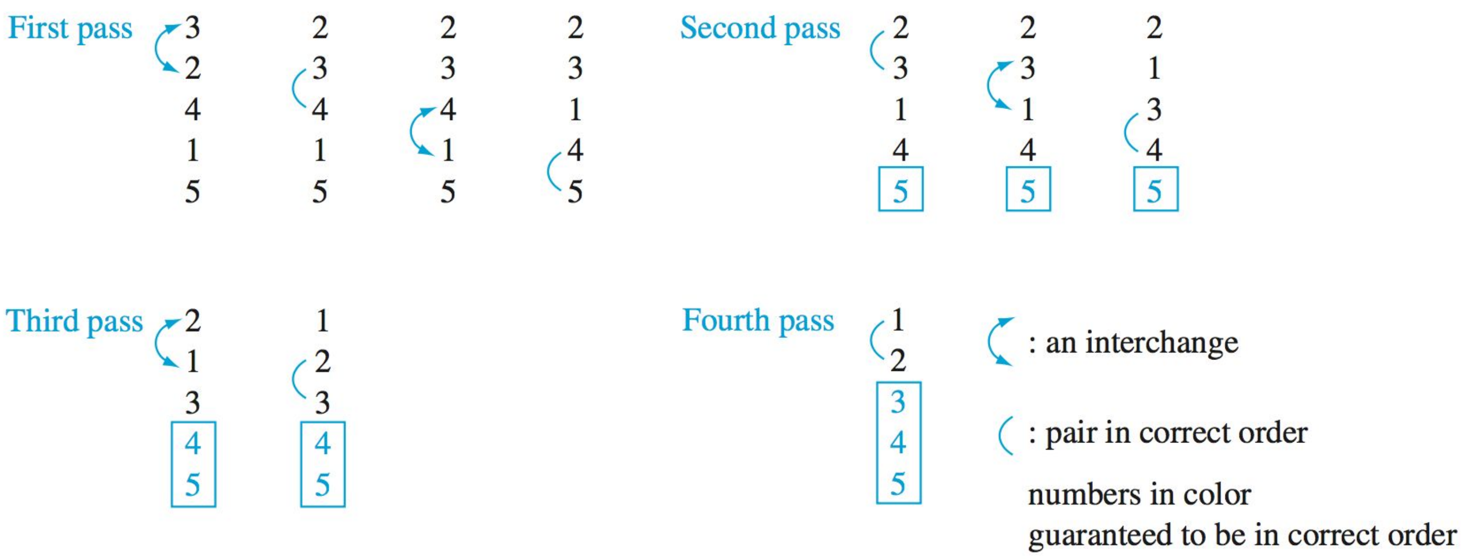


Bubble sort

https://www.geeksforgeeks.org/bubble-sort/

Make passes through the list; whenever you encounter two elements that are out of order, swap them. Repeat until the list is sorted.

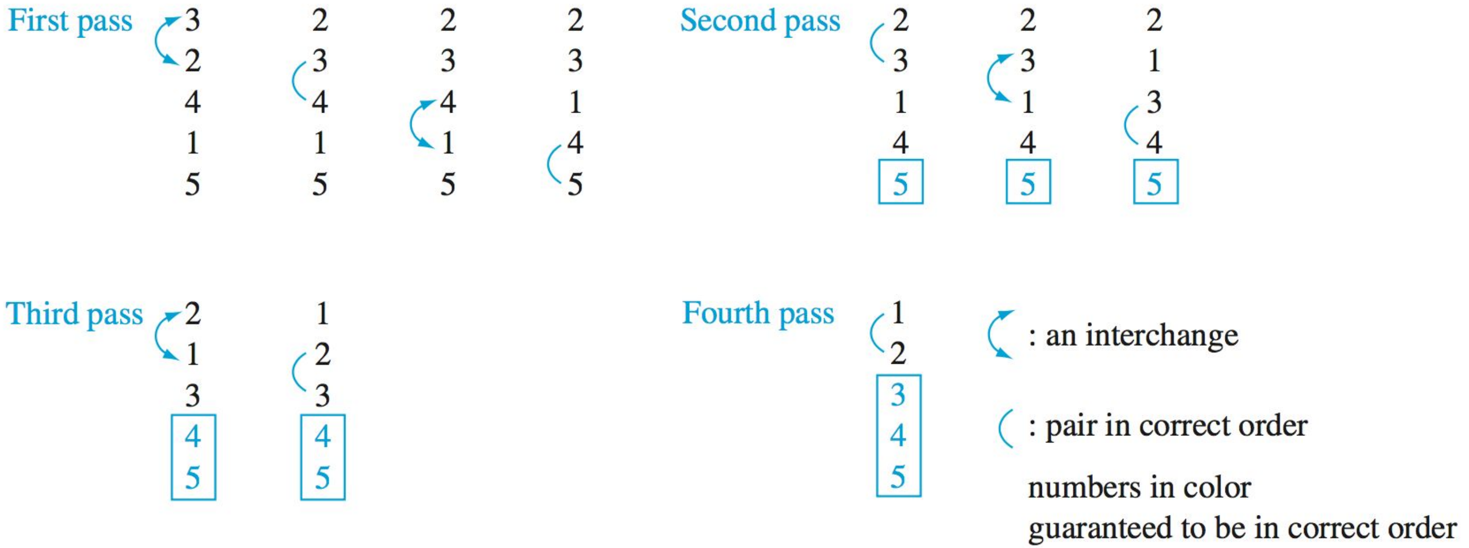
Example: S’pose we want to sort the list {3, 2, 4, 1, 5}.



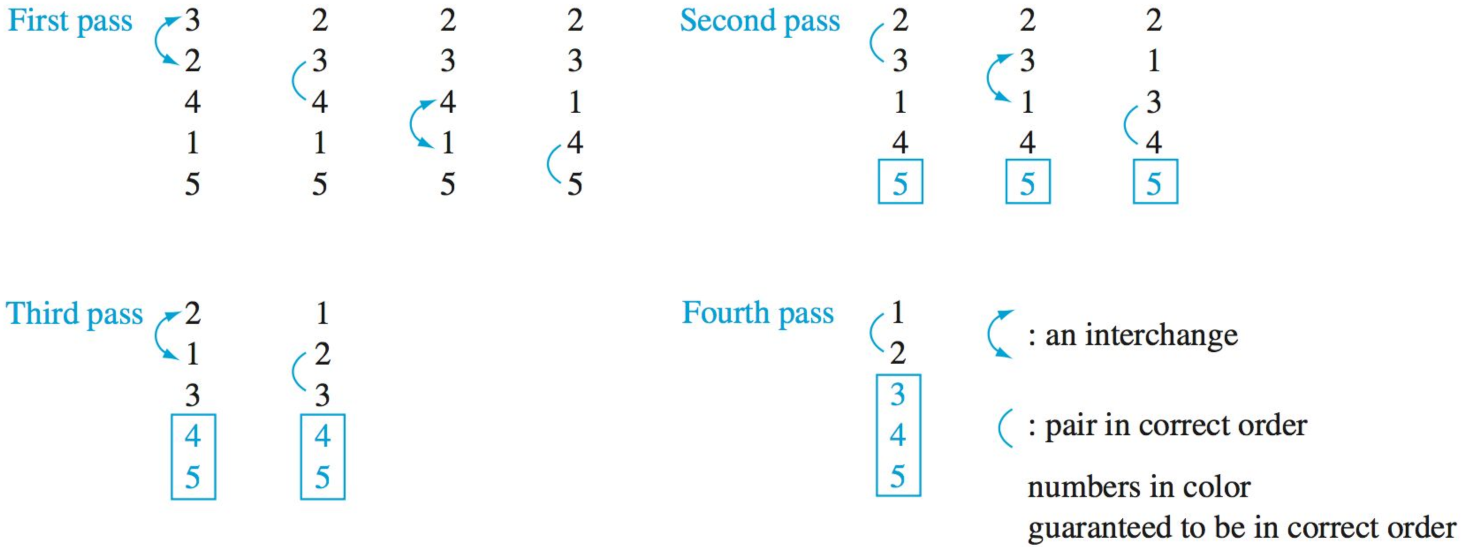
Bubble sort

Make passes through the list; whenever you encounter two elements that are out of order, swap them. Repeat until the list is sorted.

- After the **first pass**, the largest element has **sunk** to the bottom
- After the **second pass**, the second-largest element has **sunk** to the bottom
- And so on…



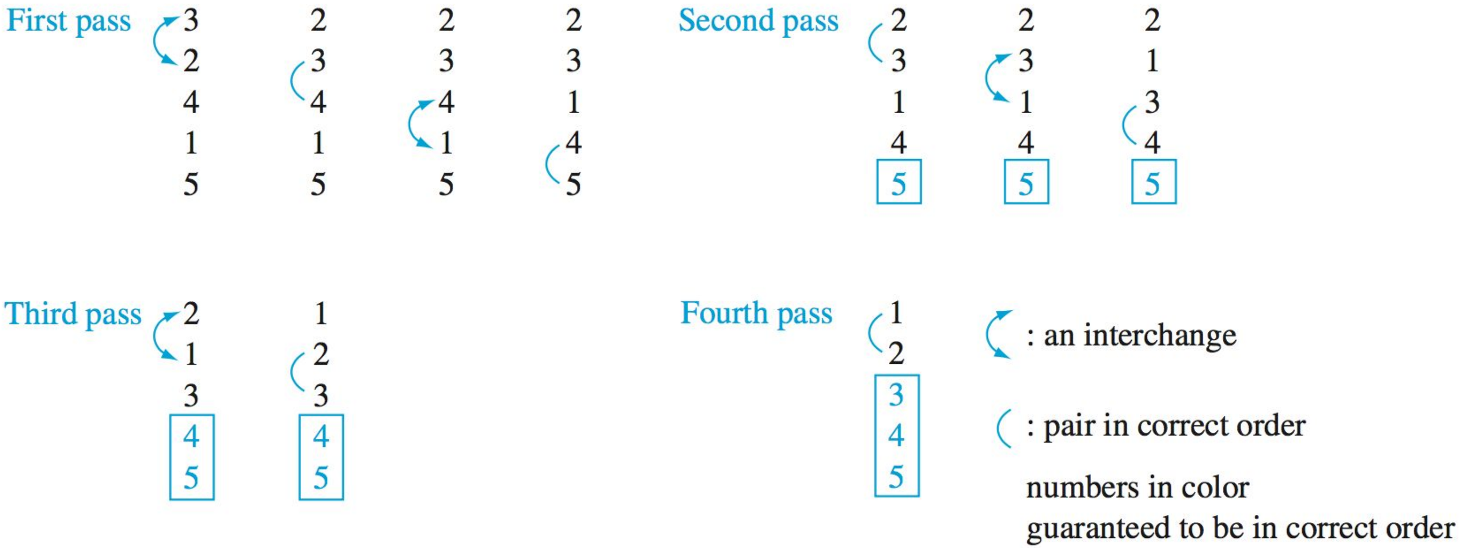
Bubble sort -- let’s code!



Bubble sort -- now, let’s *count!*

How many comparisons does our bubble sort need to make?

- … in the **best-case scenario?**
- … in the **worst-case scenario?**



: an interchange

: pair in correct order

numbers in color
guaranteed to be in correct order

https://www.geeksforgeeks.org/cocktail-sort/

Cocktail sort

Like bubble sort but make passes in **both directions** through the list; whenever you encounter two elements that are out of order, swap them. Repeat until the list is sorted.

- So when we go front → back, the **largest element sinks to the bottom**
- But when we go back → front, the **smallest element *bubbles* to the top!**

Example: S'pose we want to sort the list {5 1 4 2 8 0 2}.

(**5 1** 4 2 8 0 2) → (**1 5** 4 2 8 0 2), Swap since 5 > 1

(1 **5 4** 2 8 0 2) → (1 **4 5** 2 8 0 2), Swap since 5 > 4

(1 4 **5 2** 8 0 2) → (1 4 **2 5** 8 0 2), Swap since 5 > 2

(1 4 2 **5 8** 0 2) → (1 4 2 **5 8** 0 2)

(1 4 2 5 **8 0** 2) → (1 4 2 5 **0 8** 2), Swap since 8 > 0

(1 4 2 5 0 **8 2**) → (1 4 2 5 0 **2 8**), Swap since 8 > 2

Cocktail sort

Like bubble sort but make passes in **both directions** through the list; whenever you encounter two elements that are out of order, swap them. Repeat until the list is sorted.

- So when we go front → back, the **largest element sinks to the bottom**
- But when we go back → front, the **smallest element *bubbles* to the top!**

Example: S’pose we want to sort the list {5 1 4 2 8 0 2}.

First pass **forwards:**

(**5 1** 4 2 8 0 2) → (**1 5** 4 2 8 0 2), Swap since 5 > 1

(1 **5 4** 2 8 0 2) → (1 **4 5** 2 8 0 2), Swap since 5 > 4

(1 4 **5 2** 8 0 2) → (1 4 **2 5** 8 0 2), Swap since 5 > 2

(1 4 2 **5 8** 0 2) → (1 4 2 **5 8** 0 2)

(1 4 2 5 **8 0** 2) → (1 4 2 5 **0 8** 2), Swap since 8 > 0

(1 4 2 5 0 **8 2**) → (1 4 2 5 0 **2** 8), Swap since 8 > 2

Cocktail sort

Like bubble sort but make passes in **both directions** through the list; whenever you encounter two elements that are out of order, swap them. Repeat until the list is sorted.

- So when we go front → back, the **largest element sinks to the bottom**
- But when we go back → front, the **smallest element *bubbles* to the top!**

Example: S'pose we want to sort the list {5 1 4 2 8 0 2}.

First pass **backwards:**

(1 4 2 5 **0 2** 8) → (1 4 2 5 **0 2** 8)

(1 4 2 **5 0** 2 8) → (1 4 2 **0 5** 2 8), Swap since 5 > 0

(1 4 **2 0** 5 2 8) → (1 4 **0 2** 5 2 8), Swap since 2 > 0

(1 **4 0** 2 5 2 8) → (1 **0 4** 2 5 2 8), Swap since 4 > 0

(**1 0** 4 2 5 2 8) → (**0 1** 4 2 5 2 8), Swap since 1 > 0

Cocktail sort

Like bubble sort but make passes in **both directions** through the list; whenever you encounter two elements that are out of order, swap them. Repeat until the list is sorted.

- So when we go front → back, the **largest element sinks to the bottom**
- But when we go back → front, the **smallest element *bubbles* to the top!**

Example: S’pose we want to sort the list {5 1 4 2 8 0 2}.

Second pass **forwards:**

(0 **1 4** 2 5 2 8) → (0 **1 4** 2 5 2 8)

(0 1 **4 2** 5 2 8) → (0 1 **2 4** 5 2 8), Swap since 4 > 2

(0 1 2 **4 5** 2 8) → (0 1 2 **4 5** 2 8)

(0 1 2 4 **5 2** 8) → (0 1 2 4 **2 5** 8), Swap since 5 > 2

and so on...

Cocktail sort -- (pseudo)coding!

https://www.geeksforgeeks.org/merge-sort/

Merge sort

A type of **divide and conquer algorithm**

- ***Divides*** the original input into two halves, and
- calls itself on each smaller list (***conquers***)

S'pose we have a function **merge** that takes **in** two small lists and sorts them together to **return** a combined list that is sorted.

Example: merge({1, 5}, {4, 2}) = {1, 2, 4, 5}

- Then we call **merge** on the sorted smaller lists, and put everything back together.

Example: S'pose we want to sort the list

{38, 27, 43, 3, 9, 82, 10}

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

https://www.geeksforgeeks.org/merge-sort/

Merge sort

A type of **divide and conquer algorithm**

- ***Divides*** the original input into two halves, and
- calls itself on each smaller list (***conquers***)

S'pose we have a function **merge** that takes **in** two small lists and sorts them together to **return** a combined list that is sorted.

Example: merge({1, 5}, {4, 2}) = {1, 2, 4, 5}

- Then we call **merge** on the sorted smaller lists, and put everything back together.

Example: S'pose we want to sort the list

{38, 27, 43, 3, 9, 82, 10}

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

Merge sort

https://www.geeksforgeeks.org/merge-sort/

A type of **divide and conquer algorithm**

- ***Divides*** the original input into two halves, and
- calls itself on each smaller list (***conquers***)

S’pose we have a function **merge** that takes **in** two small lists and sorts them together to **return** a combined list that is sorted.

Example: merge({1, 5}, {4, 2}) = {1, 2, 4, 5}

- Then we call **merge** on the sorted smaller lists, and put everything back together.

Example: S’pose we want to sort the list

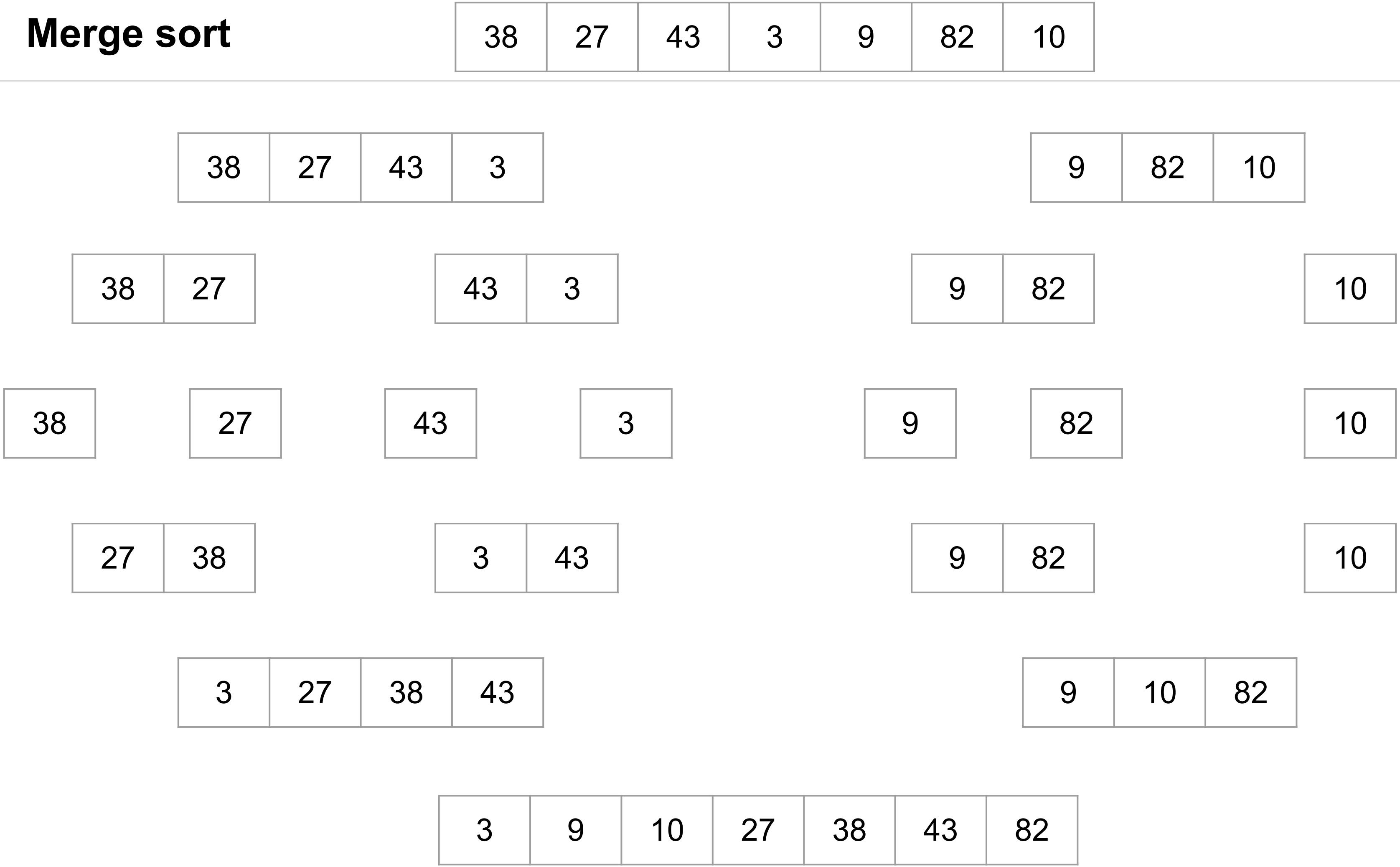
{38, 27, 43, 3, 9, 82, 10}

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

Merge sort

38	27	43	3	9	82	10

Merge sort



38 27 43 3 9 82 10

38 27 43 3 | 9 82 10

38 27 | 43 3 | 9 82 | 10

38 | 27 | 43 | 3 | 9 | 82 | 10

27 38 | 3 43 | 9 82 | 10

3 27 38 43 | 9 10 82

3 9 10 27 38 43 82

Merge sort -- (pseudo)coding!

In the wild, you will want to use **arrays**. You can use **indices with the main array** to keep track of the beginning/end of each partition.

```
void mergesort(int[] values, int size) {
    … ?




}
```

https://www.geeksforgeeks.org/quick-sort/

Quick sort

Like **merge sort**, this is a **divide and conquer algorithm**

- Pick one of the elements of the list to sort as the **pivot**
- ***Divides*** the original list into parts <= pivot, and > pivot…
- … and calls itself on those smaller parts to sort them (***conquers***)

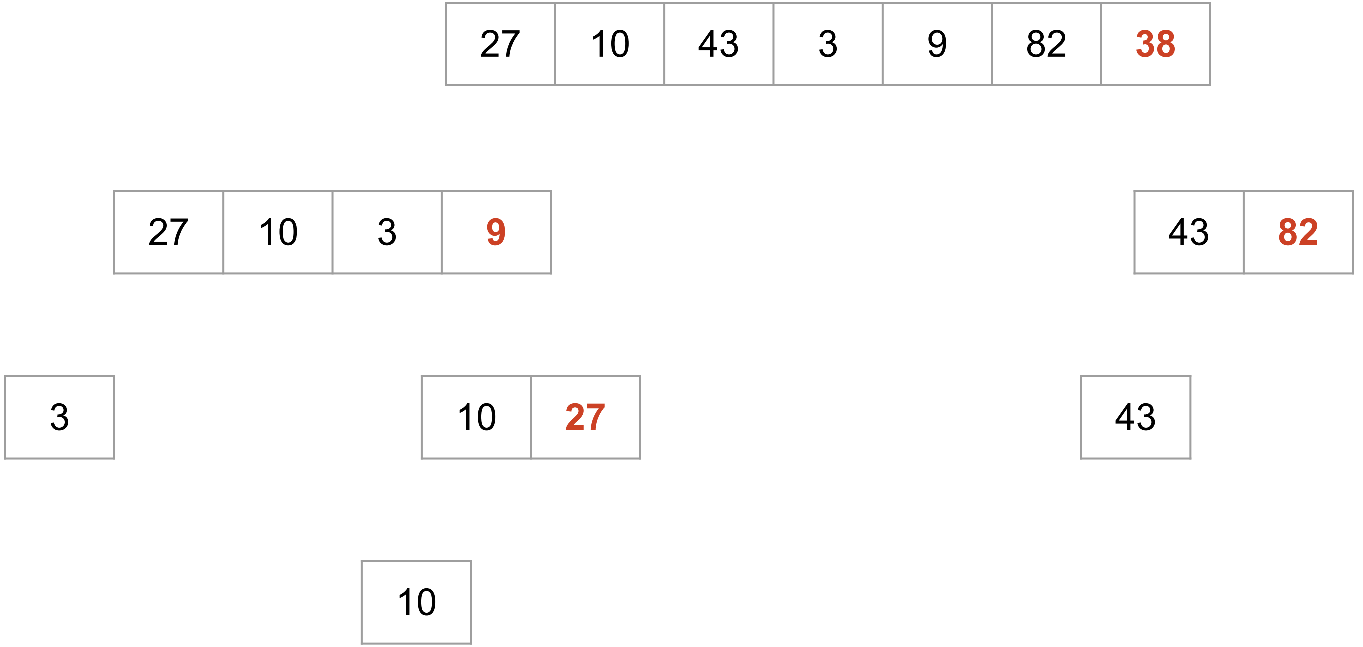
Example: S'pose we want to sort the list {27, 10, 43, 3, 9, 82, 38}.

- Let's arbitrarily pick the **last element** as the pivot, always (different versions do different things)

Quick sort

27	10	43	3	9	82	**38**

Quick sort



27 10 43 3 9 82 **38**

27 10 3 **9**

43 **82**

3

10 **27**

43

10

Quick sort -- (pseudo)coding!

Similarly to **merge sort**, in the wild, you will want to use **arrays**.

For clarity here, we will use **vectors**, and always use the **last element** of each partition as pivot.

```
// sorts the input vector vec and returns a sorted version
vector<int> quicksort(vector<int> vec) {
    … ?



    return sorted_vector;
}
```

What just happened?!

We just **sorted!**

Useful links:

https://visualgo.net/en/sorting

Bonus material: bubble sort demonstration through a folk dance!
(you'll also notice there are other algorithms demonstrated through dance linked from that video too, if you're interested)

Bonus challenge: Write a function **merge()** that takes two **sorted** vectors of integers in, and returns the two input vectors **combined** into a **sorted vector**