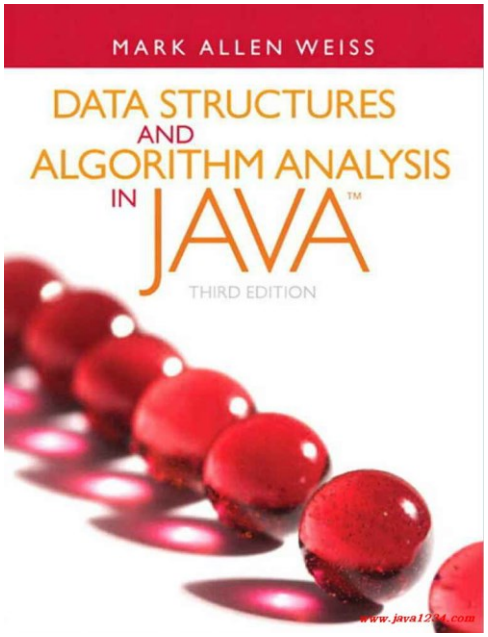
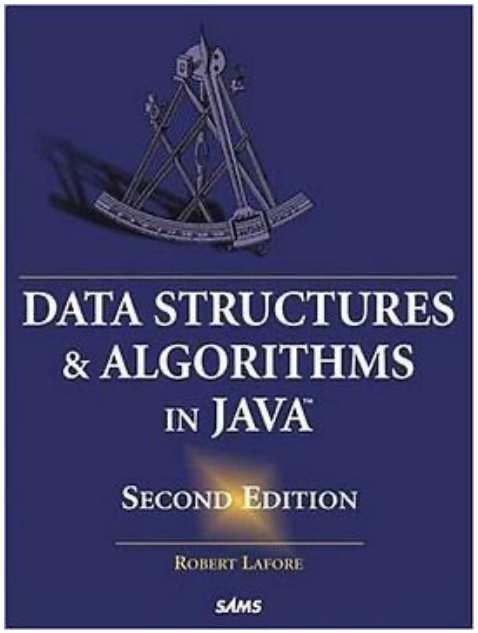
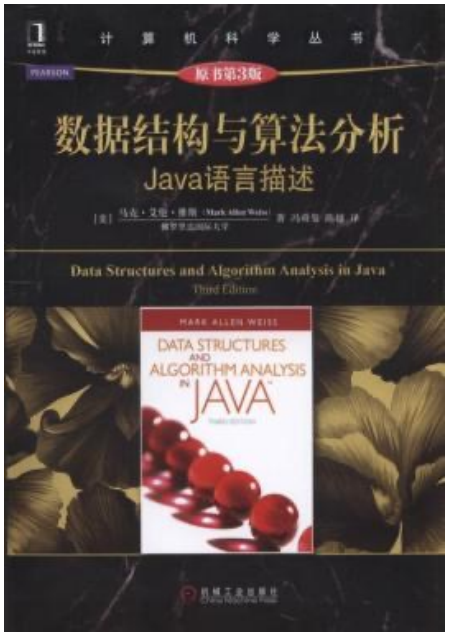
Topic 6 – Sorting algorithms







Topics

- Introduction
- Programming Revision
- Methods and Objects
- Arrays and Array Algorithms
- Big O Notation
- **Sorting Algorithms**
- Stacks and Queues
- Linked Lists
- Recursion
- Bit Manipulation

Outline

- **Introduction to Sorting**
- Bubble Sort
- Selection Sort
- Insertion Sort
- Wrap up

Sorting

- Put the data according to the rules
- Example.
  - Input:

$$[7\ \ 6\ \ 0\ \ 3\ \ 8\ \ 9\ \ 4\ \ 1]$$

  - Output:
    - from small to large

$$[0\ \ 1\ \ 3\ \ 4\ \ 6\ \ 7\ \ 8\ \ 9]$$

    - from large to small

$$[9\ \ 8\ \ 7\ \ 6\ \ 4\ \ 3\ \ 1\ \ 0]$$

    - decrease first and then increase

$$[9\ \ 8\ \ 7\ \ 0\ \ 1\ \ 3\ \ 4\ \ 6]$$

Sorting

- To be useful, data should be sorted so that it is easy to search through

- Often, we will get an array of data that is in a random order, and we will need to sort it

- We need to develop some sorting algorithms to sort arrays - the more efficient the sorting algorithm, the better

Sorting

There are many different sorting algorithms and the correct one to choose depends on the following:

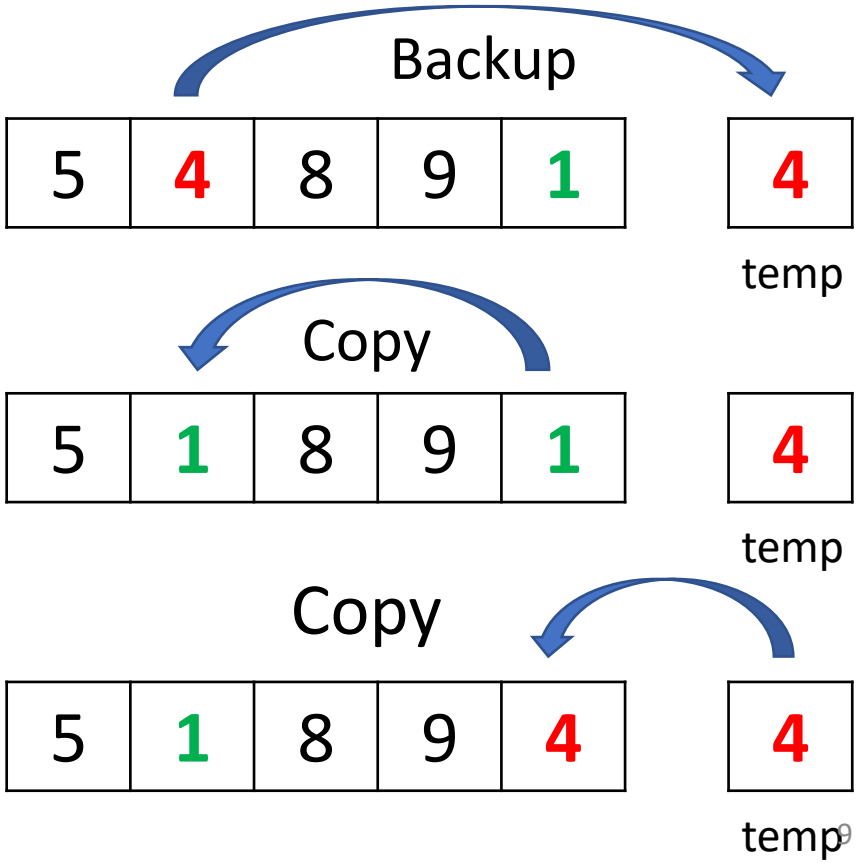
- The amount of data that needs to be sorted
- The amount of memory that is available
- The amount of time that is available
- The way the data is distributed

Sorting

- For starters, we will consider 3 simple sorting algorithms
  - **bubble sort**
  - **selection sort**
  - **insertion sort**
- All of these algorithms have a complexity of $\boldsymbol{O(n^2)}$
- This means that as the size of the array *n* increases, the time taken to sort it will increase by that amount squared
  - An array with size of *2n* will take 4 times longer to sort
  - An array that is 3 times bigger will take 9 times longer to sort
- For that reason, these algorithms are only used to sort small amounts of data!

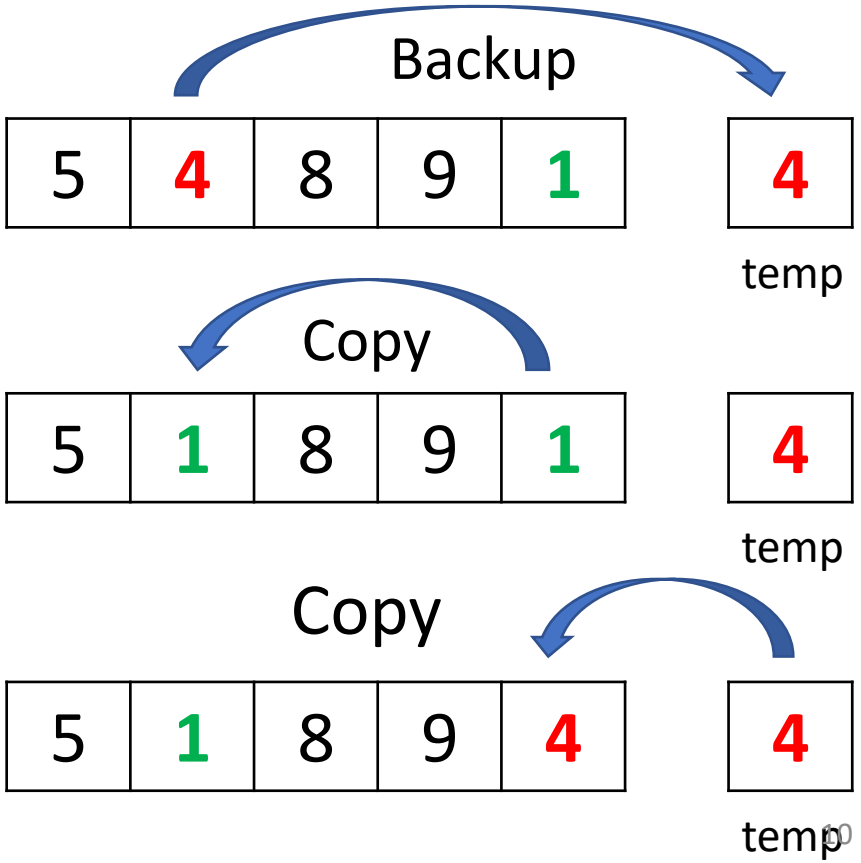
Swapping

- Swapping elements in an array is crucial to be able to sort it
- Let's swap slot 2 with slot 5
  1. Backup slot 2 into temp
  2. Copy slot 5 into slot 2
  3. Copy temp into slot 5



Swapping

- Swapping elements in an array is crucial to be able to sort it
- Lets swap slot 2 with slot 5
  1. Backup slot 2 into temp
     - **int temp = arr[1];**
  2. Copy slot 5 into slot 2
     - **arr[1] = arr[4];**
  3. Copy temp into slot 5
     - **arr[4] = temp;**



Swapping

- In order to swap one variable with another in an array
  - Back-up slot **A** (the one that will be overwritten first) into a temporary variable

    int **temp** = array[**A**];
  - Overwrite slot **A** with the value of slot **B**

    array[**A**] = array[b];
  - Use the temporary variable to overwrite the value of slot **B** with the original value of slot **A**

    array[**B**] = **temp**;

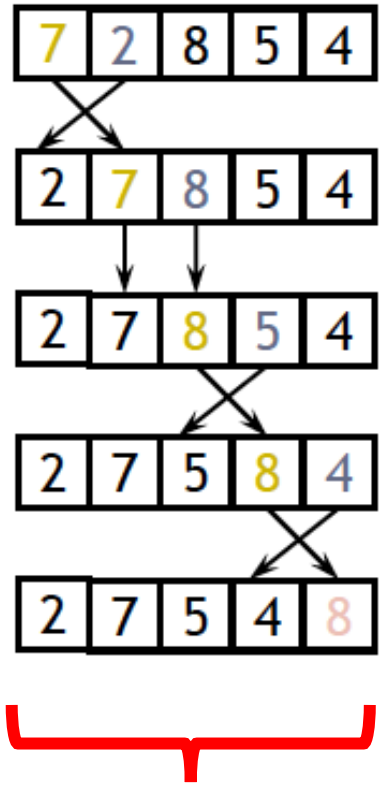
Outline

- Introduction to Sorting
- **Bubble Sort**
- Selection Sort
- Insertion Sort
- Wrap up

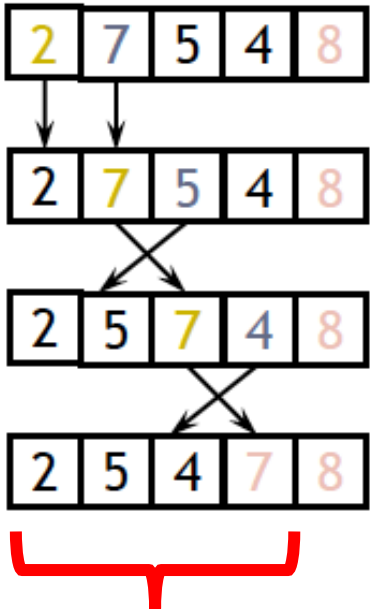
Bubble Sort

- The **bubble sort** algorithm works like this:
  - Start at the beginning of the array
  - Compare the first two numbers
  - If the one to the left is bigger, swap it with the one on the right
  - Move one position to the right and check the next two
  - Keep doing this until you reach the end of the array
  - **The biggest number** has now been 'bubbled' to the top
  - Go back to the beginning, do the same thing and bubble the next biggest number up
  - Stop short of the top part of the array where the '**bubbles**' have arrived

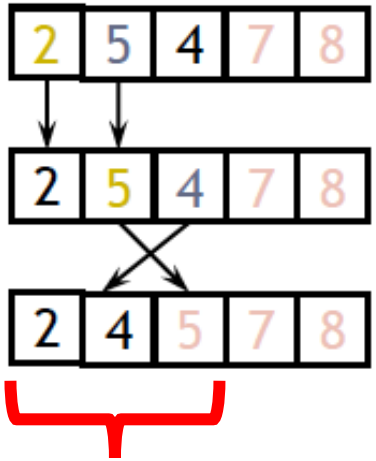
Example of Bubble Sort



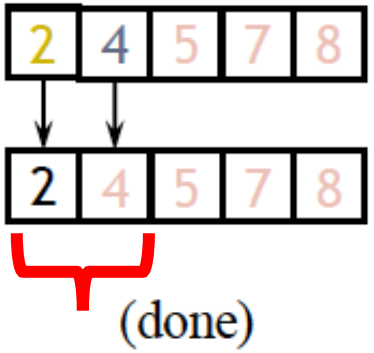
Do this from array[0] to array[4]



Do this from array[0] to array[3]



Do this from array[0] to array[2]



6 5 3 1 8 7 2 4

Example of Bubble Sort

- Red pair for current comparison
- Black squares for fixed ones

WIKIPEDIA
The Free Encyclopedia

Implementing bubble sort

- How would you write this in Java?

- Because of all the swapping, it makes sense to write a swap() method

```
public void swap(int first, int second) {
    int temp = array[first];
    array[first] = array[second];
    array[second] = temp;
}
    //Here, we write the function for swap first.
```

Main bubble sort method

- Pseudo code might look something like this

  outer 'bubbling' loop running from end of array backwards, bubbling biggest element to the top each time it runs

  {

  inner 'swapping' loop running from start of array up to last unsorted element, swapping two elements at a time

  {

  check if element[i] > element[i+1]

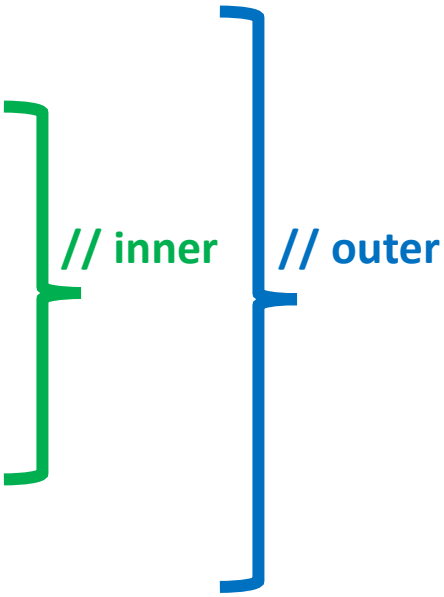
  if so, swap them

  }

  }

Java Implementation

```
public void bubblesort() {
    int outer, inner;
    for(outer = nElems-1; outer > 0; outer--) {
        for(inner = 0;inner < outer; inner++) {
            if( array[inner] > array[inner+1]) {
                swap(inner,inner+1);
            }
        }
    }
}
```



Exercise

- Write a function to sort an array using bubble sort
  - Input: array to be sorted

Algorithm complexity

- The complexity of a sorting algorithm is the relationship between the size of the array to be sorted and the length of time it takes to sort
- This depends on the number of comparisons and swaps
- A **comparison** involves a memory read because you are checking the difference between two numbers

```
if (a[inner] < a[min])
```

- A **swap** involves a memory write because you are changing the arrangement of data in memory

```
swap(inner,inner+1);
```

- Swaps take far longer to execute than comparisons because memory writes take longer than memory reads

Complexity of bubble sort

- Assume there are 10 numbers to be sorted (i.e. n is 10)
- Because the outer loop decreases by one each time the number of comparisons performed by the inner loop will be

$$9 + 8 + 7 + \cdots + 1 = 45$$

- In general this can be expressed as

$$(n-1) + (n-2) + \cdots + 1 = (n-1) \times \frac{n}{2}$$

number of comparisons

Complexity of bubble sort

- On average, the algorithm will do a swap half of the time: $\frac{(n-1)\times n}{4}$ in total
  - In the **BEST CASE SCENARIO** it won't have to do any swaps
  - In the **WORST CASE SCENARIO** it will have to perform a swap after every comparison: $(n-1)\times\frac{n}{2}$ in total
  - As the size of the array n increases, the number of comparisons and swaps increases by a factor of n^2
  - Therefore, bubble sort is $O(n^2)$ ➔ very inefficient

Outline

- Introduction to Sorting
- Bubble Sort
- **Selection Sort**
- Insertion Sort
- Wrap up

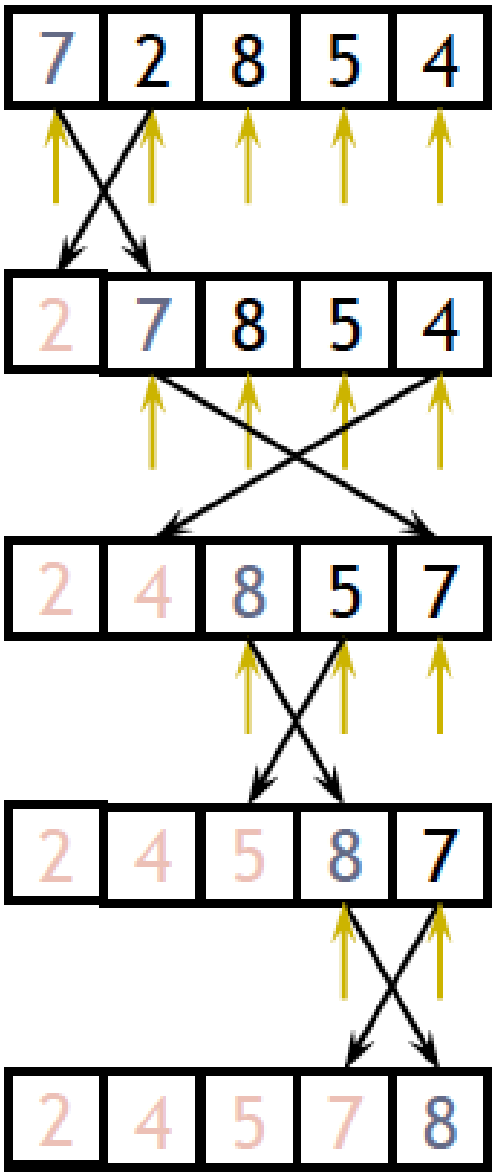
Selection Sort

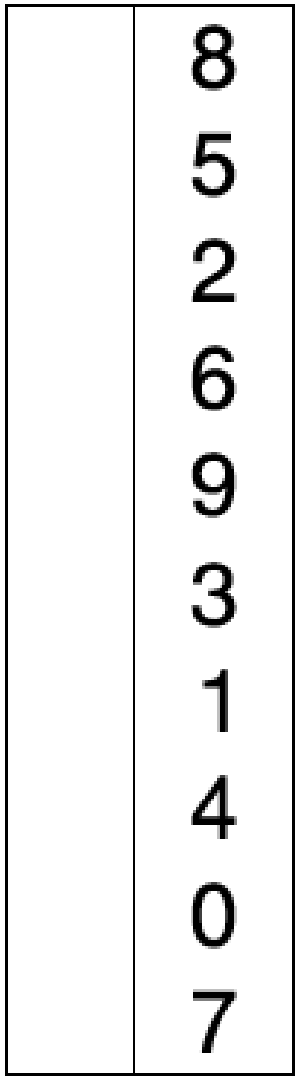
- Selection sort improves on bubble sort by reducing number of swaps from $O(n^2)$ to $O(n)$
- Number of comparisons stays $O(n^2)$ but comparisons are shorter than swaps so this can be an important improvement
- Rather than continuously swapping to 'bubble' a number to the top, we find the smallest number and swap it into place

Selection Sort

- Given an array of length n
  - Search elements 0 through $n-1$ and select the smallest
    - Swap it with the element in location 0
  - Search elements 1 through $n-1$ and select the smallest
    - Swap it with the element in location 1
  - Search elements 2 through $n-1$ and select the smallest
    - Swap it with the element in location 2
  - Search elements 3 through $n-1$ and select the smallest
    - Swap it with the element in location 3
  - Continue in this fashion until there's nothing left to search

Selection Sort





Selection Sort

- Red is current min.
- Yellow is sorted list.
- Blue is current item

Main selection sort method

- Pseudo code might look something like this

outer loop running through each place in the array looking for the correct element to swap into that place – starts at the beginning

{

    inner loop which always looks for the smallest remaining unsorted item

    {

        find minimum

        swap array[outer] with array[minimum]

    }

}

Code for Selection Sort

```
public static void selectionSort() {
        int min;
        for (int outer = 0; outer < array.length-1; outer++) {
        // outer is the point where the unsorted numbers start
                min = outer;
                // min's default value is the first slot to be checked
                for (int i = outer + 1; i < array.length; i++) {
                // inner loop checks through the unsorted numbers
                        if (array[i] < array[min]) {
                                min = i; //inner loop finds the minimum
                        }
                } // min always refers to the min found so far
                swap(outer, min);
        } // all items with slot numbers less than or equal to outer are sorted
}
```

Exercise

- Write a function to sort an array using selection sort
  - Input: array to be sorted

Analysis of selection sort

- The outer loop executes $\boldsymbol{n-1}$ times (i.e. we have to find the smallest number $\boldsymbol{n-1}$ times)
- The inner loop executes about $\frac{\boldsymbol{n}}{\boldsymbol{2}}$ times on average (it executes number $\boldsymbol{n-1}$ times the first time and only once for the final time)

- Number of comparisons required is roughly

$$(\boldsymbol{n-1}) \times \frac{\boldsymbol{n}}{\boldsymbol{2}} = \frac{\boldsymbol{1}}{\boldsymbol{2}}\boldsymbol{n^2} - \frac{\boldsymbol{1}}{\boldsymbol{2}}\boldsymbol{n}$$

- The algorithm is therefore $\boldsymbol{O(n^2)}$

Complexity of Selection Sort

- Selection sort performs the same number of comparisons as bubble sort :

$$(n-1) \times \frac{n}{2}$$

- However, we have minimized the number of swaps required – we only swap something into its correct location once – a total of $n-1$ swaps
- The algorithm is still $O(n^2)$ but it is a faster $O(n^2)$ than bubble sort since a swap takes far longer than a comparison and there are less swaps involved

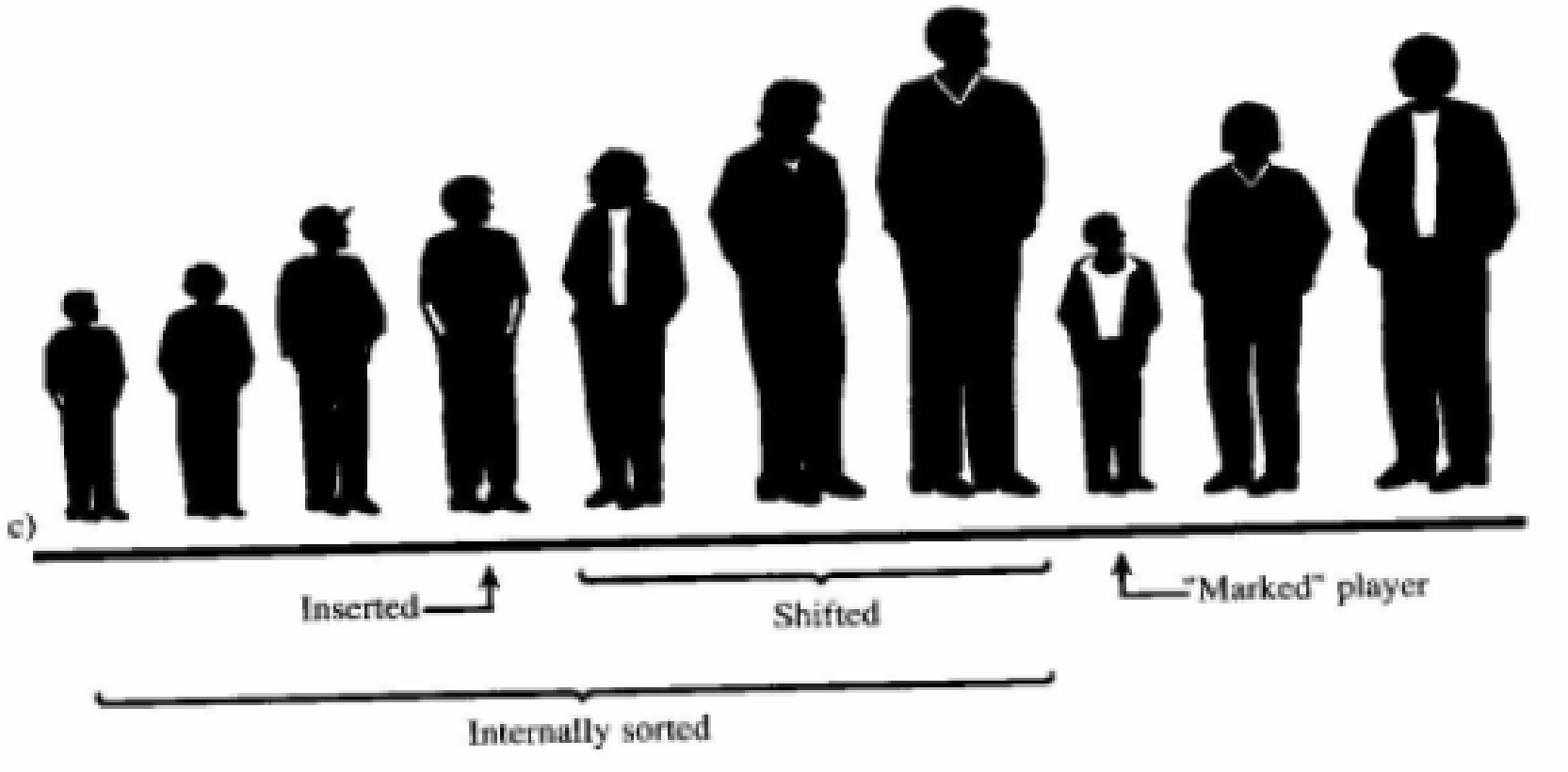
Outline

- Introduction to Sorting
- Bubble Sort
- Selection Sort
- **Insertion Sort**
- Wrap up

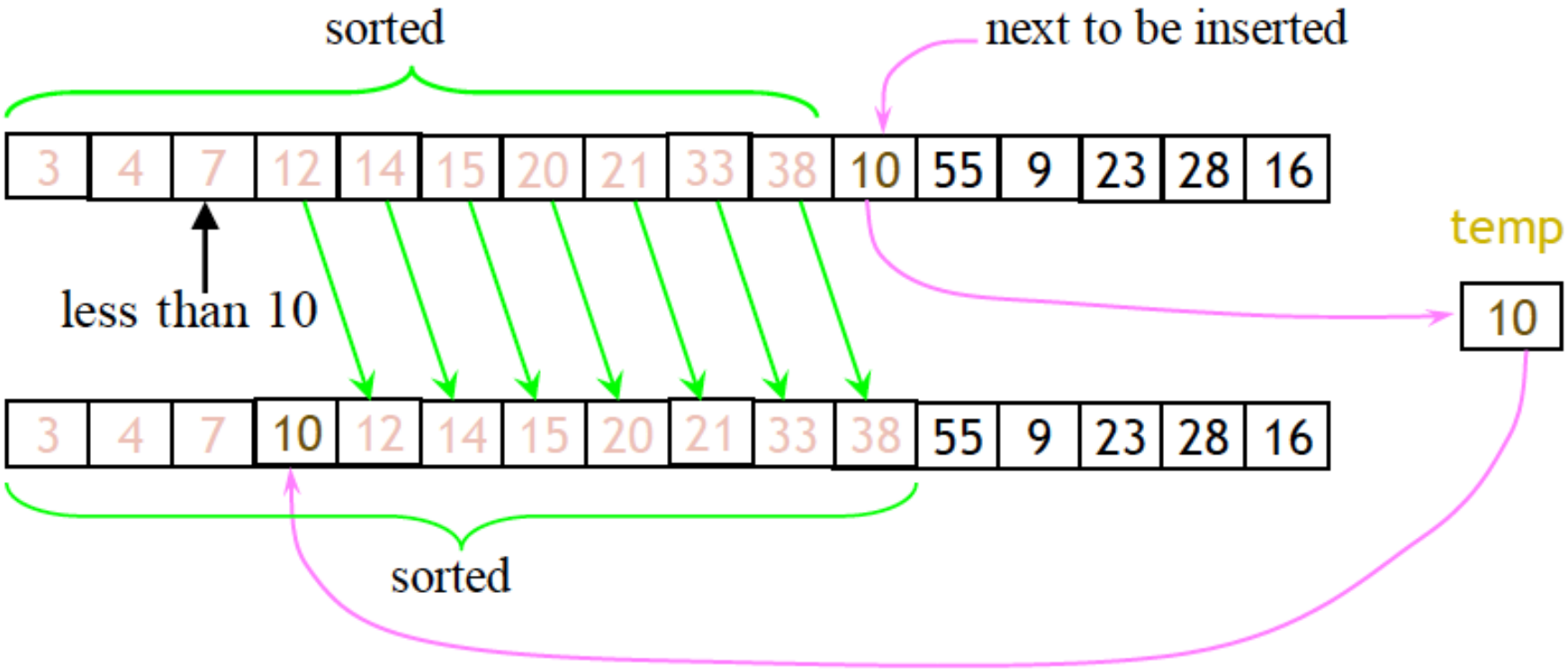
Insertion Sort

- Still $O(n^2)$ but can be faster than bubble and selection sort
- Usually used as the final stage of more sophisticated sorts, like quicksort
- Doesn't use a swap – instead shifts elements up to make space
- The idea is that the elements on the left of a marker are already sorted
- You make space to slot in the new unsorted element by moving all the other elements up one
- Exactly like insertion into an ordered array

Insertion Sort



Insertion Sort



6 5 3 1 8 7 2 4

Insertion Sort

- The partial sorted list (**black**) initially contains only the first element in the list.
- With each iteration one element (**red**) is removed from the "not yet checked for order" input data and inserted in-place into the sorted list.

WIKIPEDIA
The Free Encyclopedia

Code for Insertion Sort

```
public static void insertionSort() {
    for (int outer = 1; outer < array.length; outer++) {
    // outer is the next element to be sorted
        int temp = array[outer]; //back it up
        int inner = outer; // inner used to track shifts
        while (inner > 0 && array[inner - 1] >= temp) {
            array[inner] = array[inner - 1]; // swap
            inner--;
        } //shift them all right until one is smaller
        array[inner] = temp;
    }
}
```

Exercise

- Write a function to sort an array using insertion sort
  - Input: array to be sorted

Analysis of insertion sort

- We run once through the outer loop, inserting each of $\boldsymbol{n}$ elements
- On average, there are $\frac{\boldsymbol{n}}{\boldsymbol{2}}$ elements already sorted
  - The inner loop checks and moves half of these
  - This gives a second factor of $\frac{\boldsymbol{n}}{\boldsymbol{4}}$
- Hence, the time required for an insertion sort of an array of n elements is proportional to $\frac{\boldsymbol{n^2}}{\boldsymbol{4}}$
- Discarding constants, we find that insertion sort is $\boldsymbol{O(n^2)}$
- For already sorted data, runs in $\boldsymbol{O(n)}$ time
- For data arranged in inverse order, runs no faster than bubble sort

Outline

- Introduction to Sorting
- Bubble Sort
- Selection Sort
- Insertion Sort
- **Wrap up**

Remembering

- Bubble Sort
  - Bubbles the biggest to the end by swapping every time
- Selection Sort
  - Selects the min and swaps that element to the beginning
- Insertion Sort
  - Finds where the element should go in the sorted part and moves all the elements up one to make space

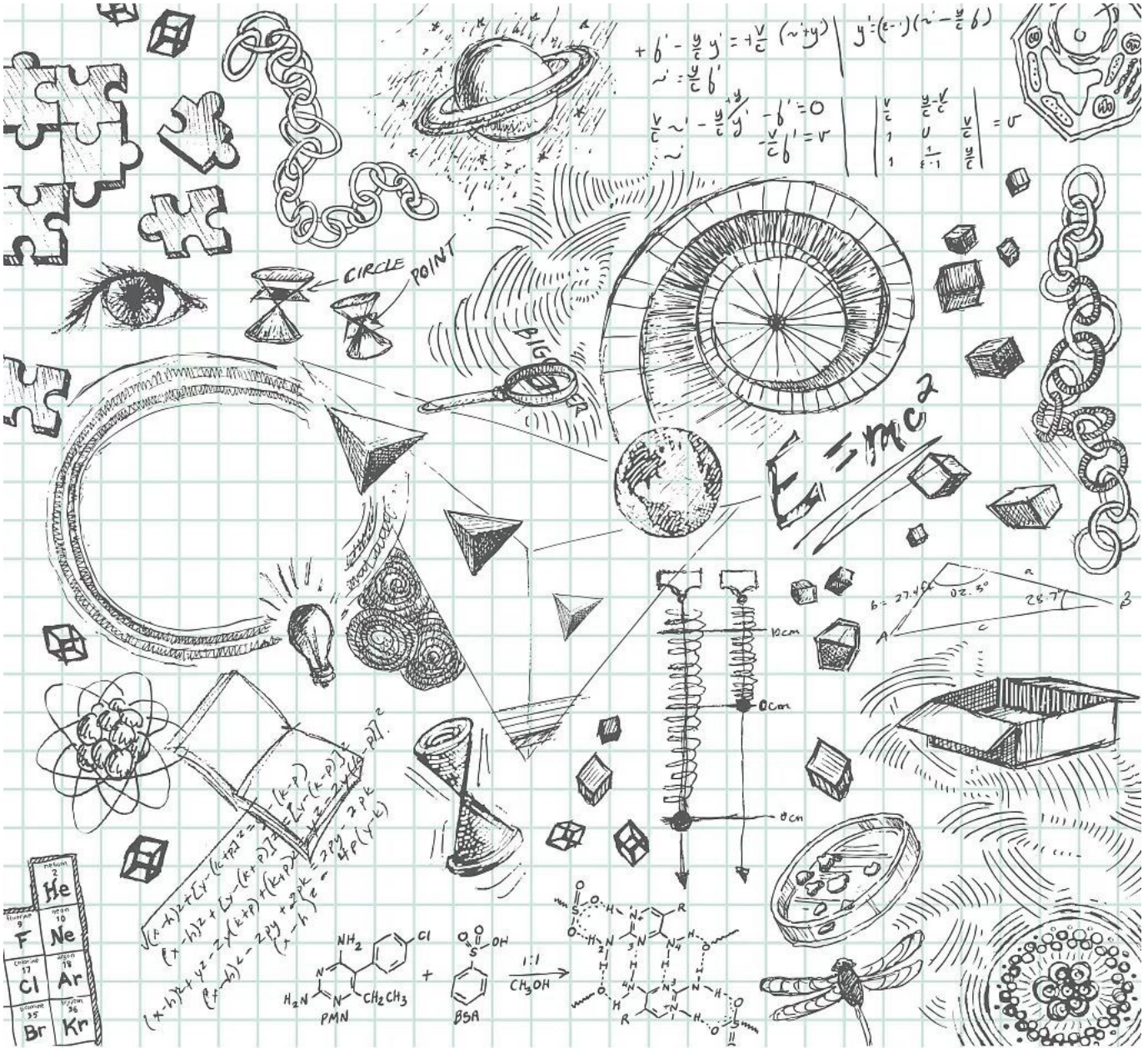
- All three algorithms are considered $\boldsymbol{O(n^2)}$

Comparison

- **Bubble sort** is useful for **small** amounts of data because it is easy to code
  - Comparisons: Always $O(n^2)$
  - Swaps: Depends on how sorted the list is ➔ varies from 0 to $O(n^2)$

- **Selection sort** can be used when amount of data is small and memory writing is far more time consuming than memory reading
  - Comparisons: Always $O(n^2)$
  - Swaps: Always $n - 1$

- **Insertion sort** performs well when the list is already partially sorted
  - Comparisons: Depends on how sorted the list is ➔ varies from $O(n)$ to $O(n^2)$
  - Swaps: Depends on how sorted the list is ➔ varies from $O(n)$ to $O(n^2)$


CIRCLE
POINT
BIGGER
$E=mc^2$
10cm
0cm
PMN
BSA
1:1
CH_3OH