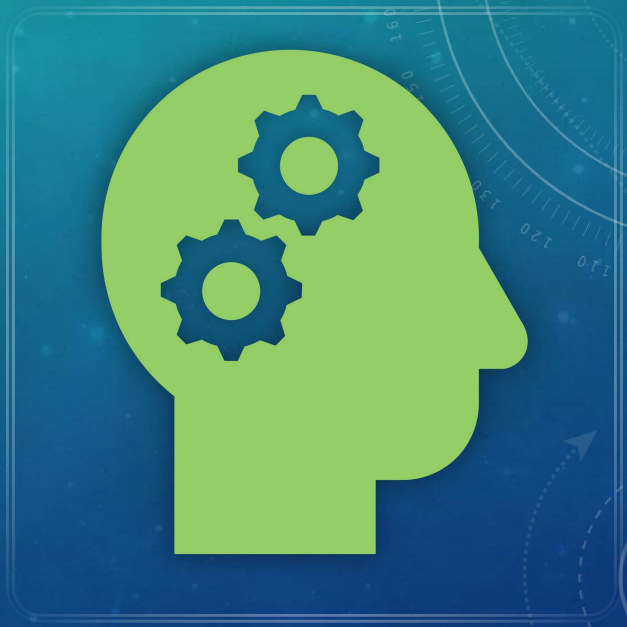
CS210 - DATA STRUCTURES AND ALGORITHMS 1

DR. NATALIE CULLIGAN

NATALIE.CULLIGAN@MU.IE

SORTING

- Often, for data to be useful to us as computer scientists we need our data to be sorted so that it is easy to search through
- If we have an array of data in random order, we may want to sort it
- How can we approach this problem?



SORTING ALGORITHMS

- There are algorithms we can use to procedurally sort our algorithms
- The more efficient the algorithms are, the better!

WHICH ALGORITHM TO USE?

There are a lot of sorting algorithms, some of which we'll talk about in this module

How do we choose which one we should use?

We need to ask:

- How much data has to be sorted?
- How much memory is available?
- How much time is available?

SORTING ALGORITHMS

Bubble Sort

Selection Sort

Insertion Sort

BUBBLE, SELECTION, INSERTION

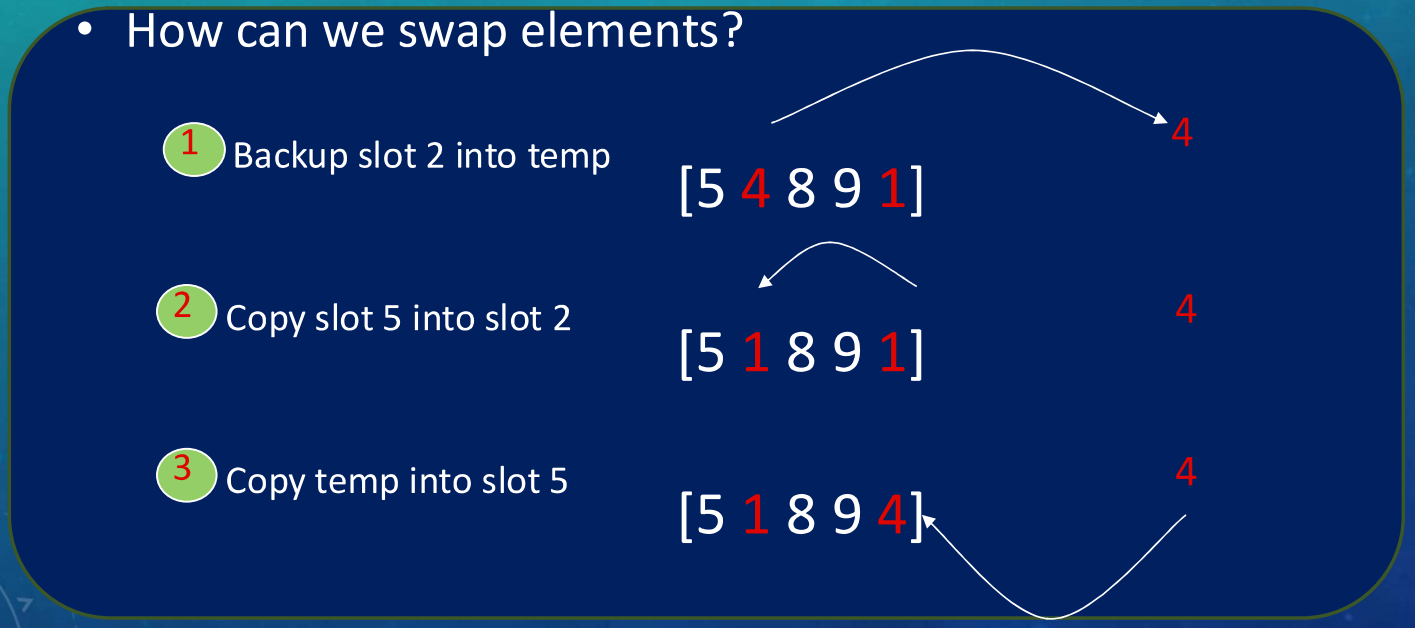
- These algorithms all have a complexity of $O(n^2)$
- So, as the size of the array increases the time taken to sort the array increases by that amount squared
  - An array that is twice as big will take …… times longer to sort ?
  - An array that is three times bigger will take …… times longer to sort?
- For that reason, these algorithms are only used to sort small amounts of data!

Four

Nine

SWAPPING

- When sorting an array, we will need to swap its elements around
- How can we swap elements?



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
  - Overwrite slot A with the value of slot B
  - Use the temporary variable to overwrite the value of slot B with the original value of slot A

```
int temp = array[a];
array[a]=array[b];
array[b]=temp;
```


BUBBLE SORT

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

LET’S ARRANGE THESE BUBBLES FROM SMALLEST TO LARGEST, USING BUBBLESORT

START AT THE BEGINNING OF THE ARRAY

COMPARE THE FIRST TWO NUMBERS

- The bubble in position 0 is bigger than the one in position 1
- So, we have to swap them
- The bubble containing 7 ”bubbles up”

COMPARE THE FIRST TWO NUMBERS

- The bubble in position 0 is bigger than the one in position 1
- So, we have to swap them
- The bubble containing 7 ”bubbles up”

HOW WOULD YOU WRITE THIS IN JAVA?

- 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;
}
```

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);
        }
    }
   }
```

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
         }
   }
```

BUBBLE SORT METHOD

```
myBubbleSortMethod(elements[]){
for(i = elements.length;i>0;i--)
    {
        for(j=0;j<i;j++)
        {

                if (elements[i] > elements[i+1]){
                    swap(elements[i],elements[i+1])
            }
         }
      }
    }
```

WHAT IS THE COMPLEXITY OF BUBBLESORT?

The complexity of an algorithm is the relationship between the input and the time it takes to run on that input

So, 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

COMPLEXITY OF A SORTING ALGORITHM

- Time taken 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

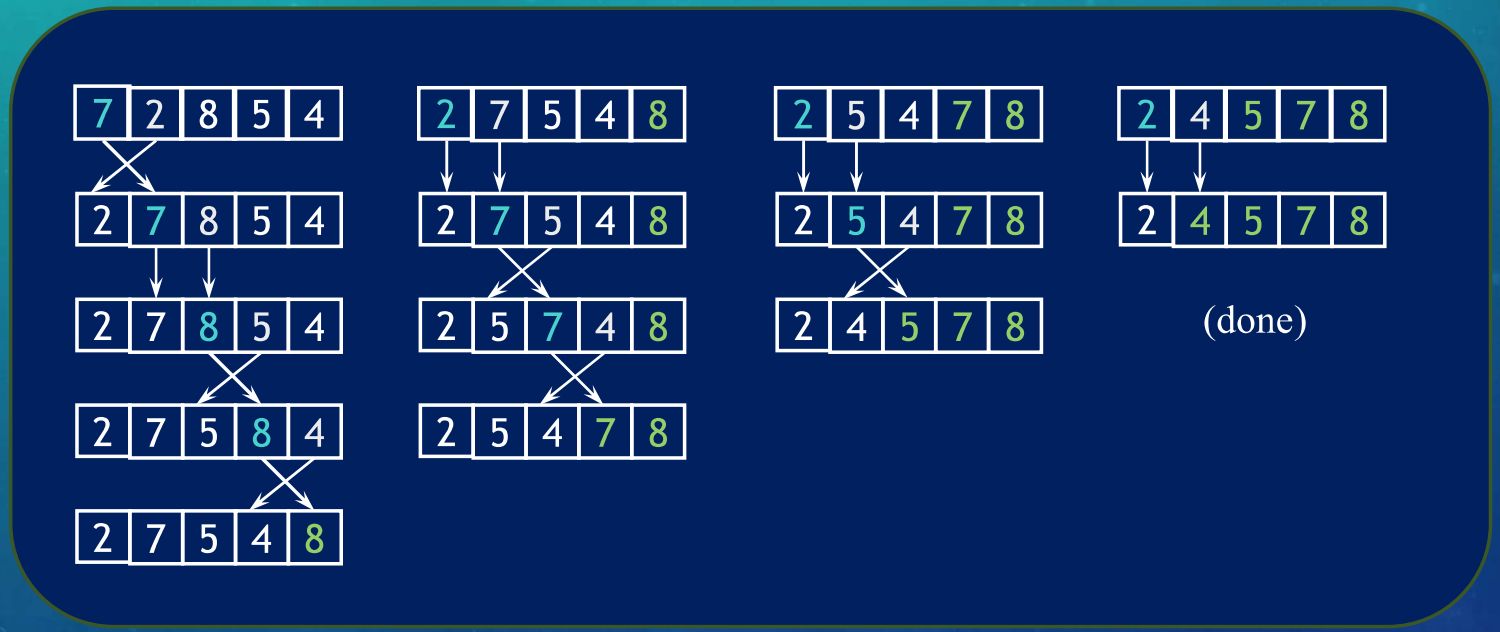
COMPLEXITY OF BUBBLE SORT

- 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 + 6 + 5 + 4 + 3 + 2 + 1 = 45
- In general this can be expressed as
  - (n -1) + (n -2) + (n – 3) + … + 1
  - or **(n-1)*(n / 2)** number of comparisons

COMPLEXITY OF BUBBLE SORT

- On average, the algorithm will do a swap half of the time: **n*(n-1) / 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)*(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

EXAMPLE OF BUBBLE SORT



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

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

- 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

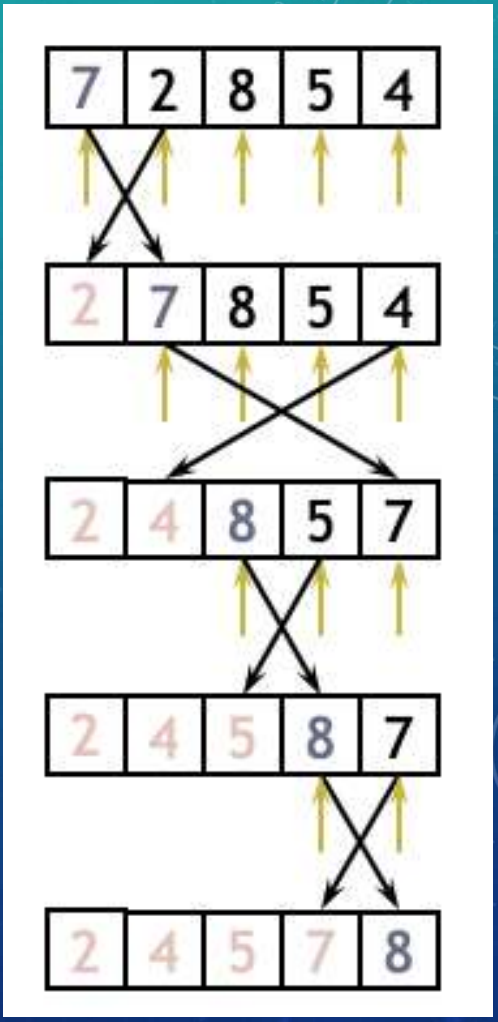
{

find minimum

swap array[outer] with array[minimum]

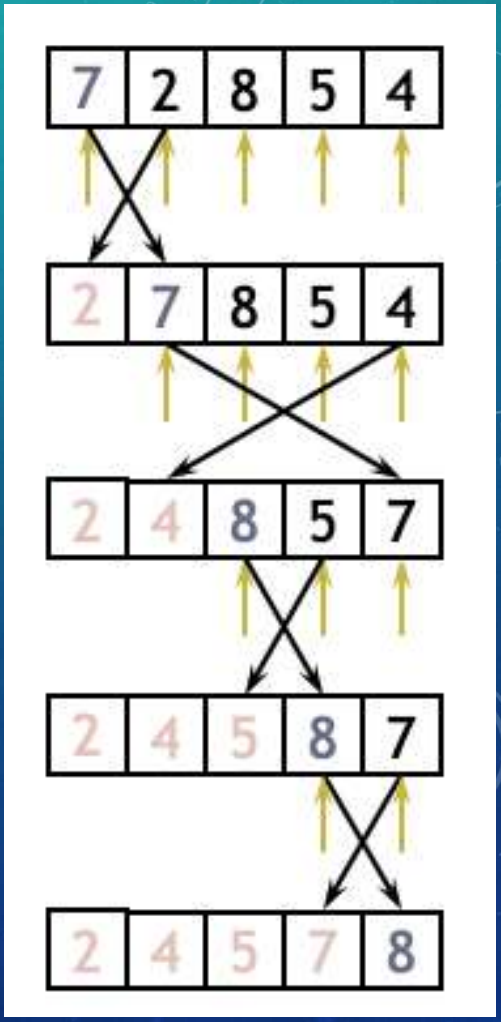
}

}



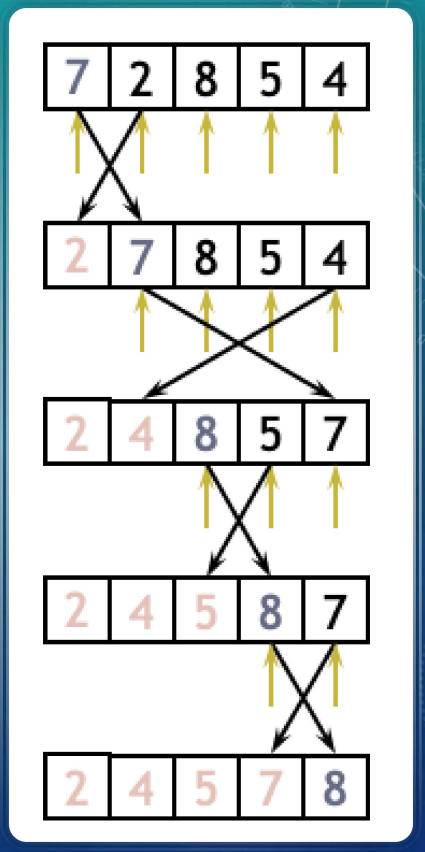
SELECTION SORT

```
selectionSortMethod(inputArray[])
{ // outer is where the unsorted numbers start
for(int outer =0; outer < inputArrayLength; i++)
    {// inner loop checks through the unsorted numbers
        min = outer;
        for(int inner = outer+1, inner < inputArray.length; inner ++)
            {
            if(inputArray[inner]<inputArray[min])
                min = inner;
            }
        Swap(array[j], array[min])
    }
}
```



ANALYSIS OF SELECTION SORT

- The outer loop executes n-1 times (i.e. we have to find the smallest number n-1 times)
- The inner loop executes about n/2 times on average (it executes n-1 times the first time and only once for the final time)
- Number of comparisons required is roughly (n-1)*(n/2)
- The algorithm is therefore $O(n^2)$



COMPLEXITY OF SELECTION SORT

Selection sort performs the same number of comparisons as bubble sort : **n*(n-1)/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

INSERTION SORT

INSERTION SORT

Still $O(n^2)$ but can be faster than bubble sort and selection sort

(Often used as the final stage of more sophisticated sorts, like quicksort)

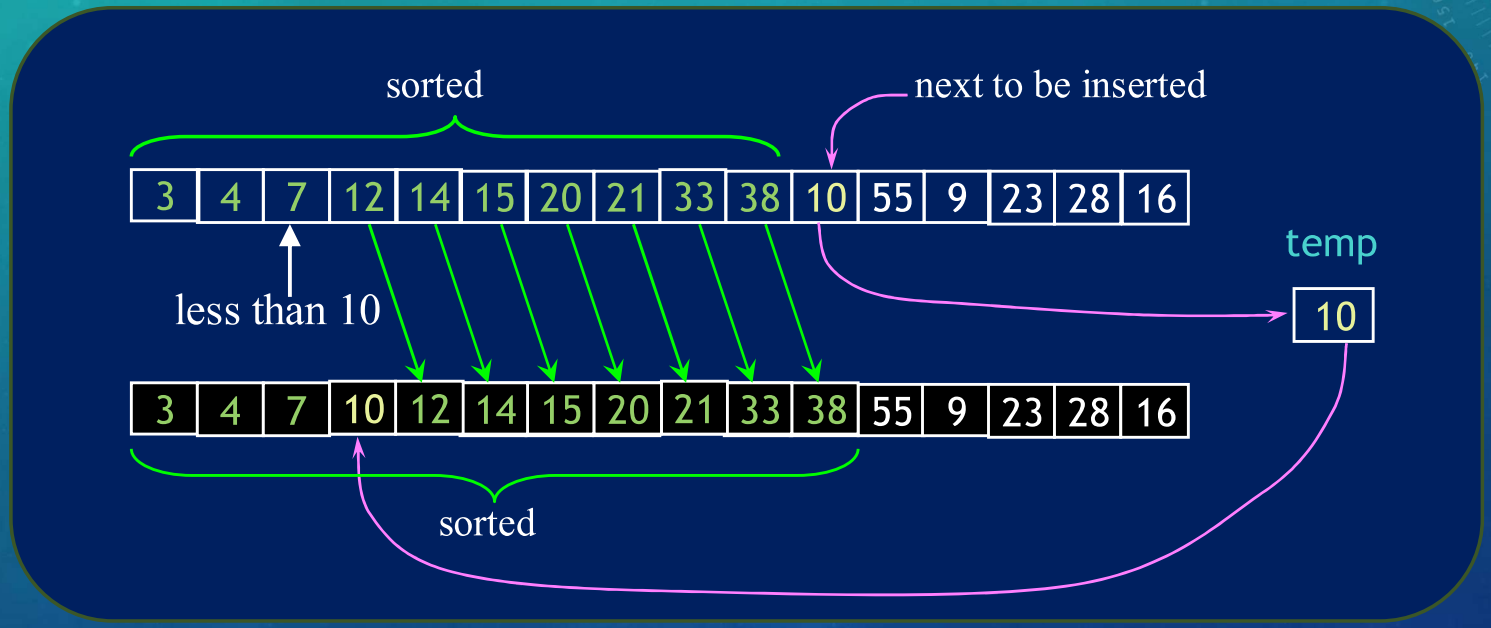
Doesn't use a swap – instead *shifts* elements up to make space

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



ANALYSIS OF INSERTION SORT

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

REMEMBER!

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 **$O(n^2)$**

WHICH SHOULD WE USE?

Bubble?

Selection?

Insertion?

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 **$O(n^2)$**

COMPARISON

- **Selection sort** can be used when amount of data is small and we are concerned about the time taken to write to memory
  - Comparisons: Always **$O(n^2)$**
  - Swaps: Always **n-1**

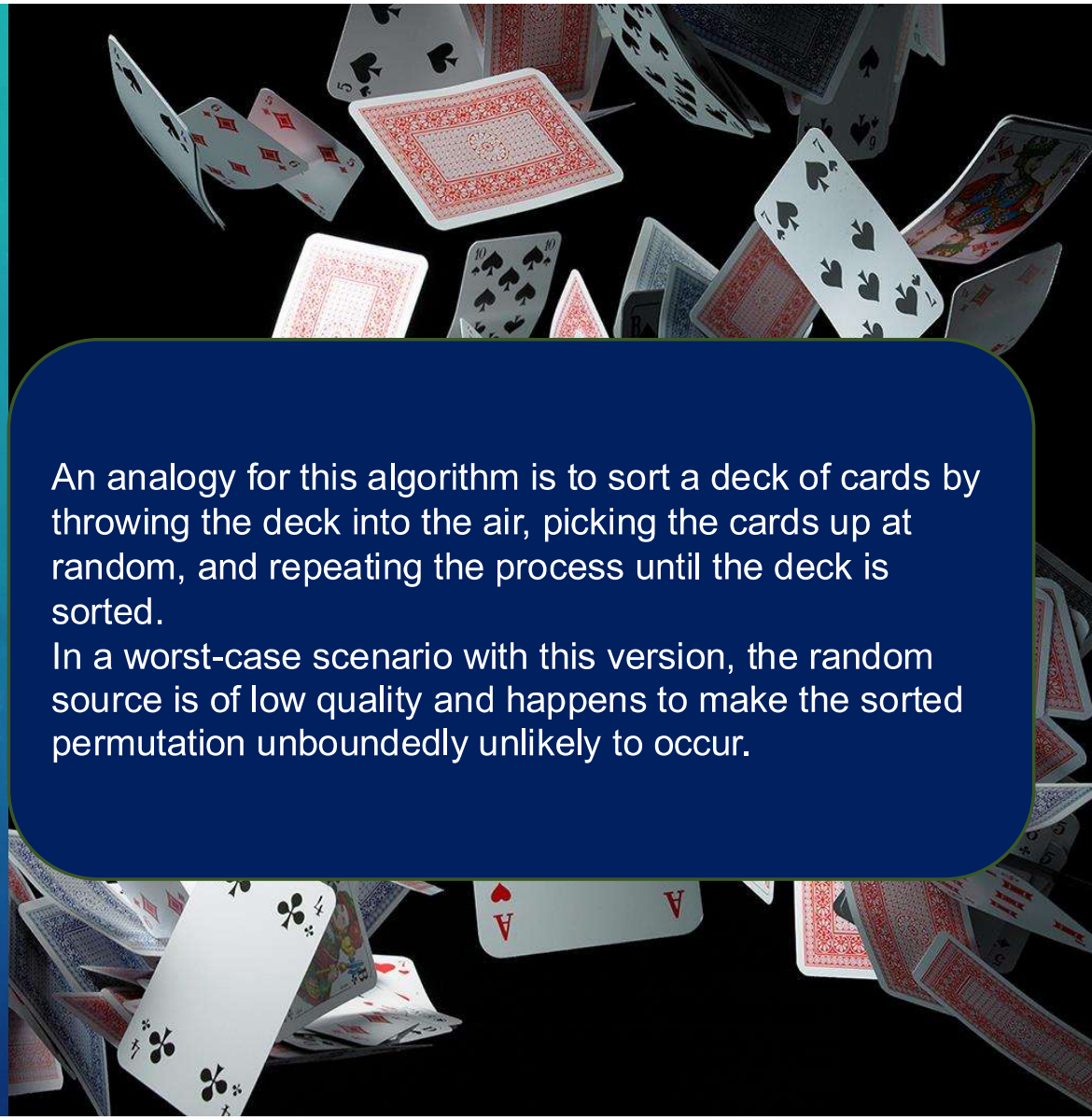
COMPARISON

- **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 **$O(n^2)$**

BONUS: BOGO SORT (THE SCARIEST SORTING ALGORITHM)

```
while not sorted(deck):
    shuffle(deck)
```

What is the Big O notation of this algorithm?



An analogy for this algorithm is to sort a deck of cards by throwing the deck into the air, picking the cards up at random, and repeating the process until the deck is sorted.
In a worst-case scenario with this version, the random source is of low quality and happens to make the sorted permutation unboundedly unlikely to occur.

MID TERM

- No classes or labs next week
- Take this time to look over the slides, make sure you understand the code from labs
- Start looking at exam papers
- Make sure you understand how to write methods
- Have a fun (and safe!) Halloween!