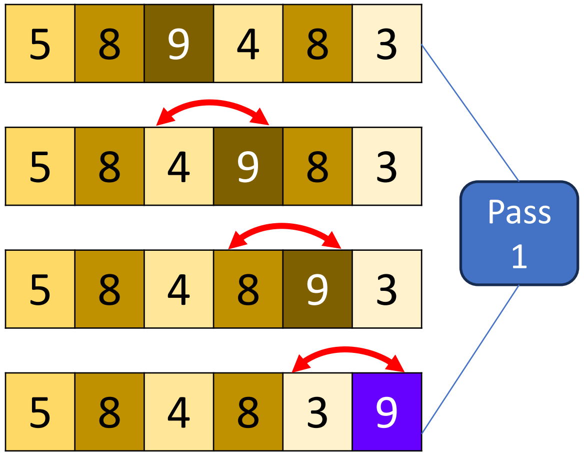
# Review

for Exam 2

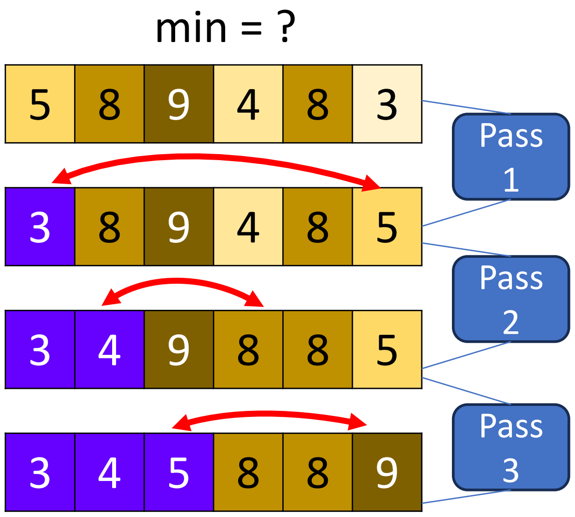
## Classification of sorting algorithms

We are sorting elements of array A of size n

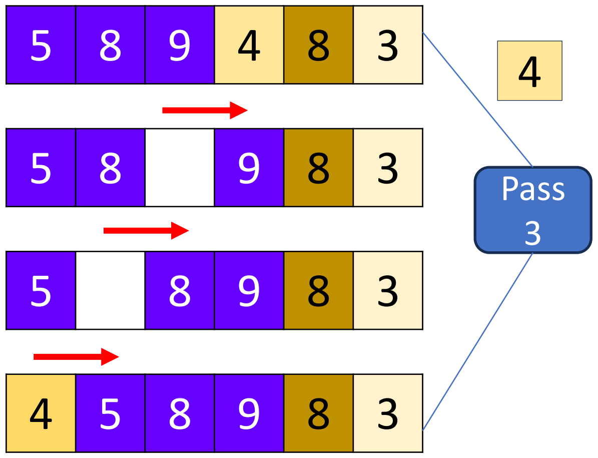
# Sorting algorithms: 1/2



Bubble sort

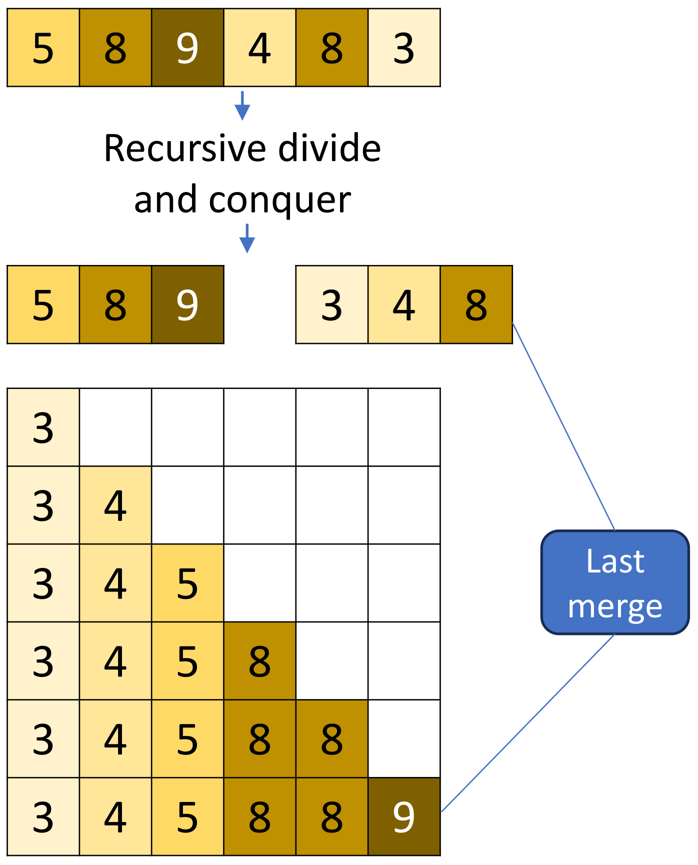


Selection sort

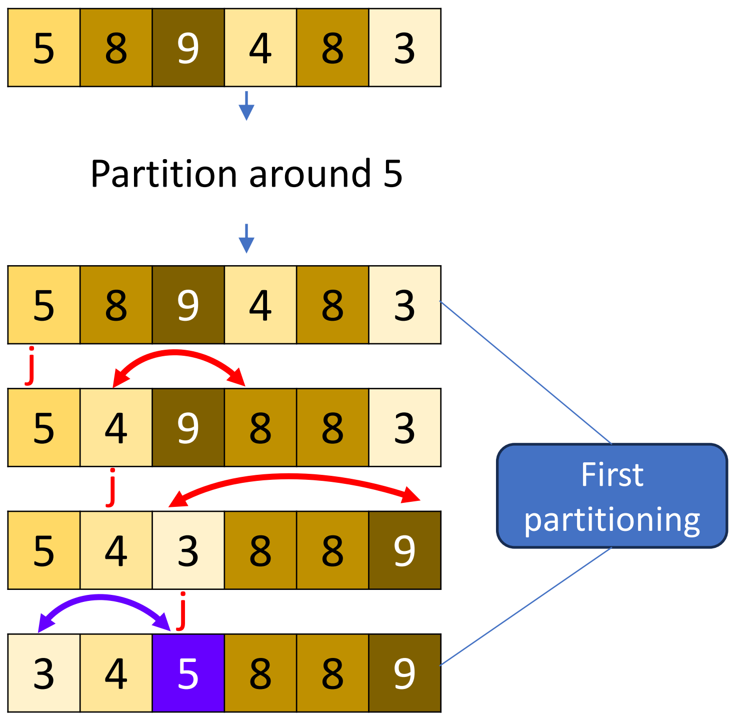


Insertion sort

# Sorting algorithms: 2/2



Merge sort



Quick sort

# Classifying by number of comparisons

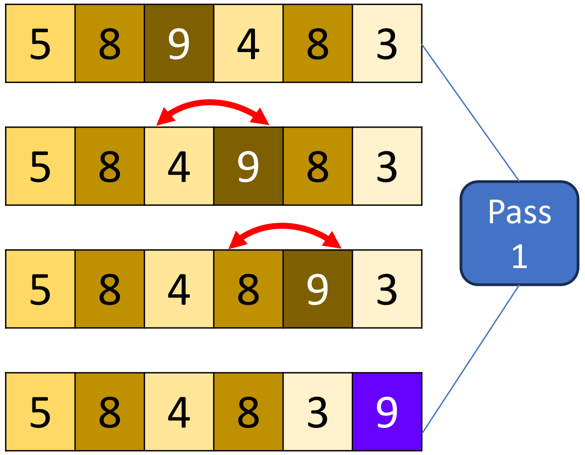
- Comparison-based algorithms:
  - O($n^2$)
  - O($n^{1.5}$)
  - O(n log n)

- No comparisons:
  - O(n)

## Classifying by additional memory used

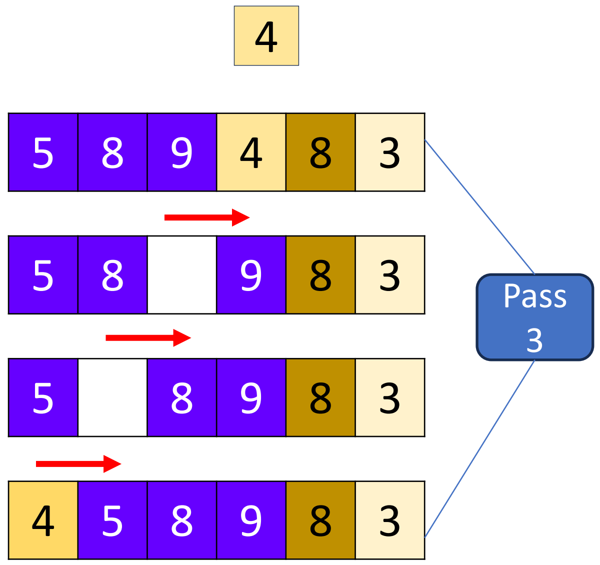
- Sorting “in place”: needs O(1) additional temporary memory: We rearrange elements inside the array itself
- Sorting “not in place”: needs >=O(n) of additional memory. We move data between A and temp arrays

# Bubble sort?



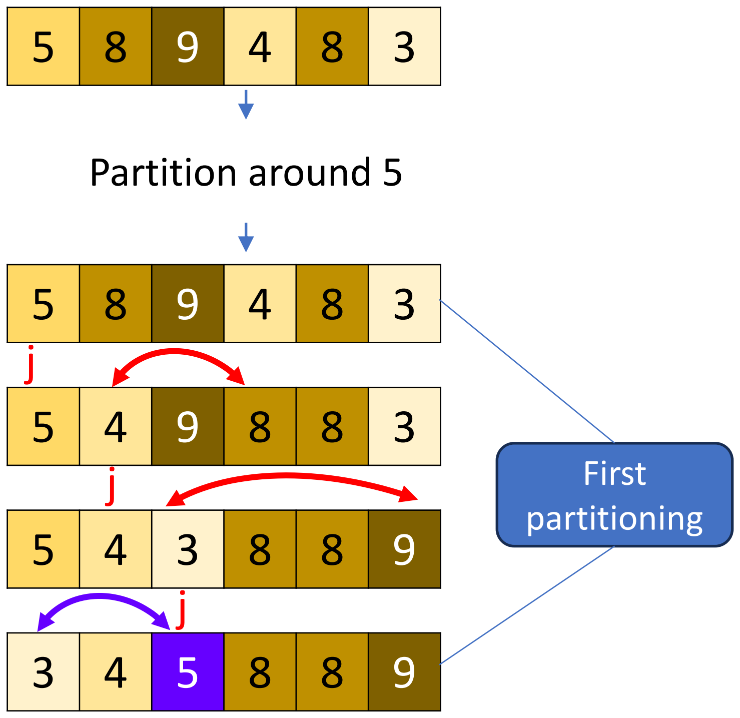
In place ✓

# Insertion sort?



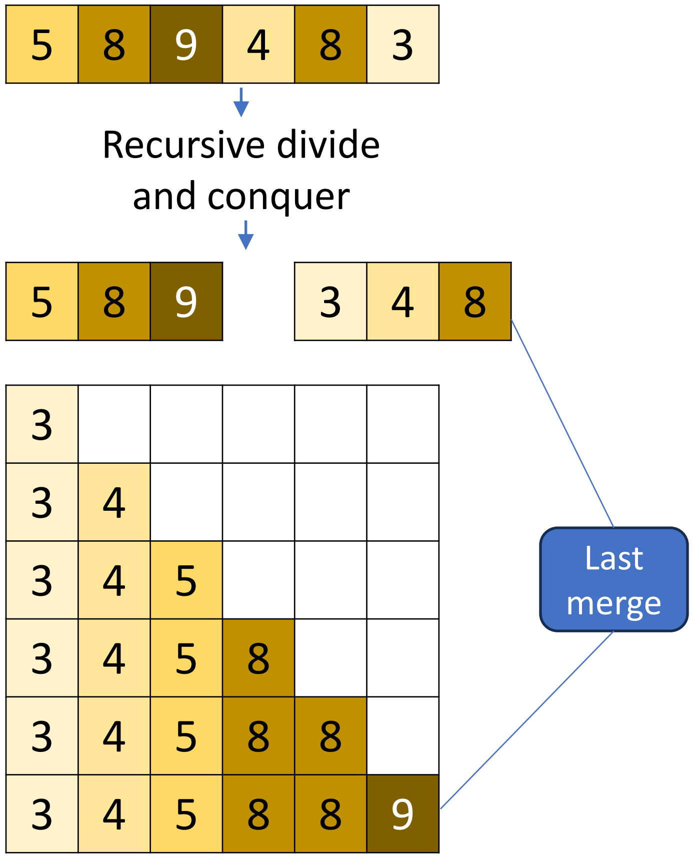
In place ✓

# Quick sort?



In place ✓

# Merge sort?



Not in place ✗

## By stability

**Definition:**

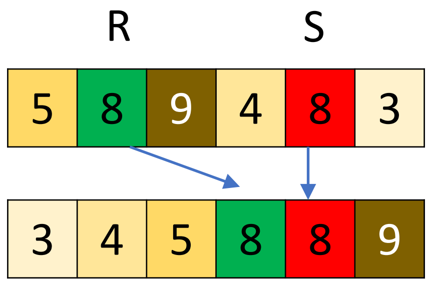
Sorting algorithm is ***stable*** if for all indexes *i* and *j* such that the A[*i*] = A[*j*], if element A[*i*] precedes element A[*j*] in the original array, then element A[*i*] also precedes element A[*j*] in the sorted array.

- In other words, sorting algorithms that maintain the relative order of elements with equal keys (equivalent elements retain their relative positions even after sorting) are called *stable sorting algorithms*
- This is an important property: for example, we preserve the order of insertions (time stamp) even after sorting

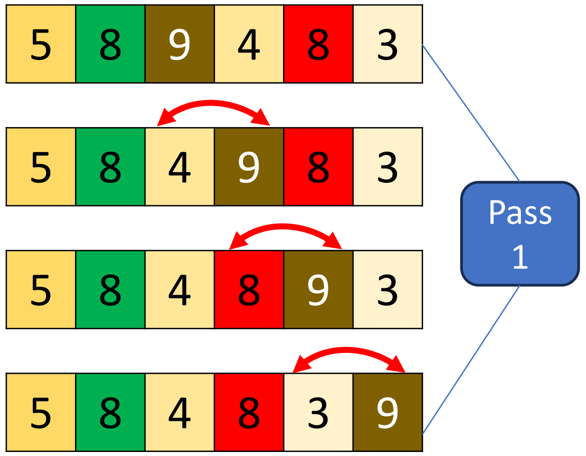
## Which algorithms are stable?

To answer this:

- A is the array to be sorted
- Let R and S be two elements of A with the same key and R appears earlier in the array than S:
    - R is at position *i*, and S at position *j*, s.t. *i*<*j*
- To show that algorithm is stable make sure that in the sorted output R cannot appear after S

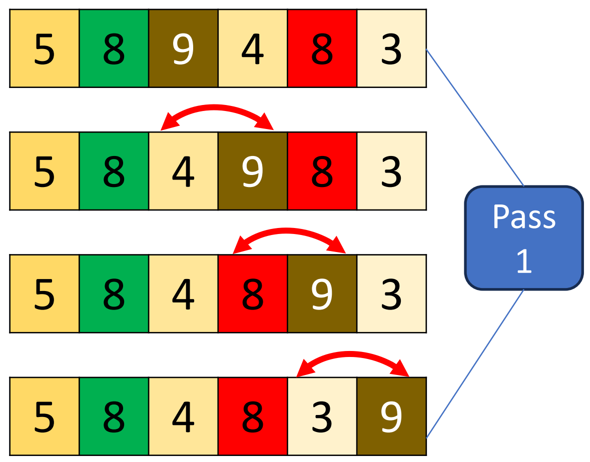


# Bubble Sort?



Is it possible that after sorting green 8 will be AFTER red 8?

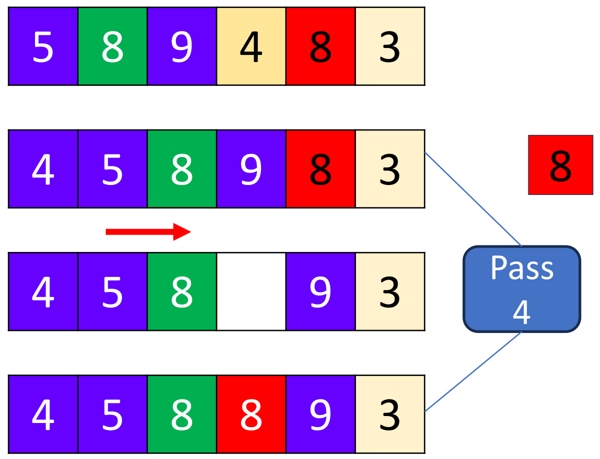
# Bubble Sort?



Elements change order only when a smaller element follows a larger. Since red 8 is not smaller than green 8 their relative order cannot change.

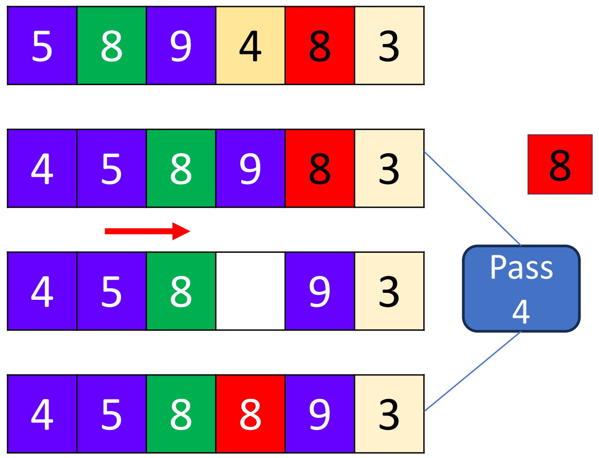
Stable ✓

# Insertion Sort?



Is it possible that after sorting green 8 will be AFTER red 8?

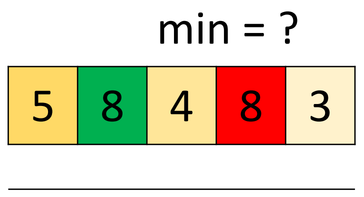
## Insertion Sort?



When red 8 is to be inserted into sorted subarray A[0..j – 1], only elements larger than it are shifted. Thus green 8 would not be shifted during red 8's insertion and hence would always precede it.

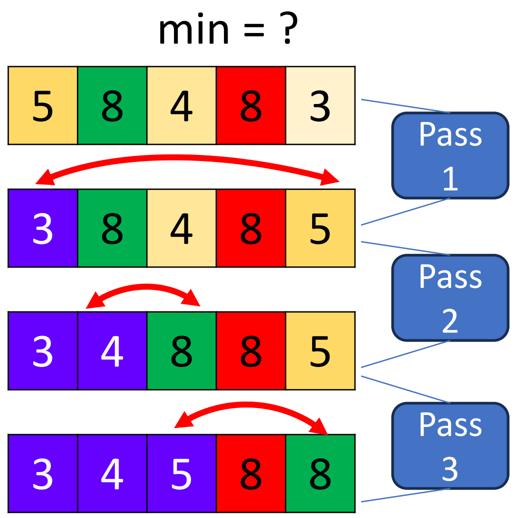
Stable ✓

# Selection Sort?



Is it possible that after sorting green 8 will be AFTER red 8?

# Selection Sort?

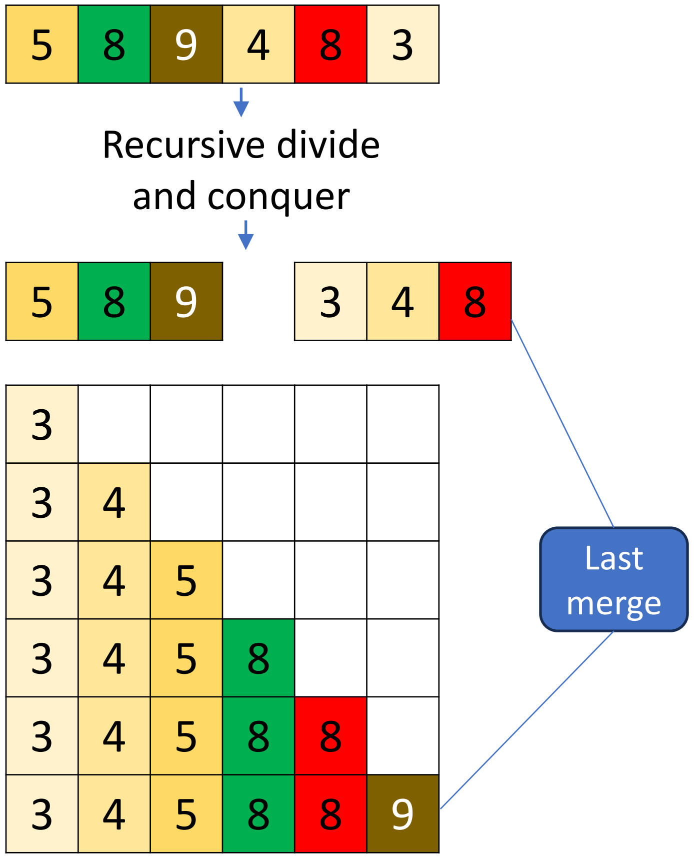


Selection sort

Not stable ✗

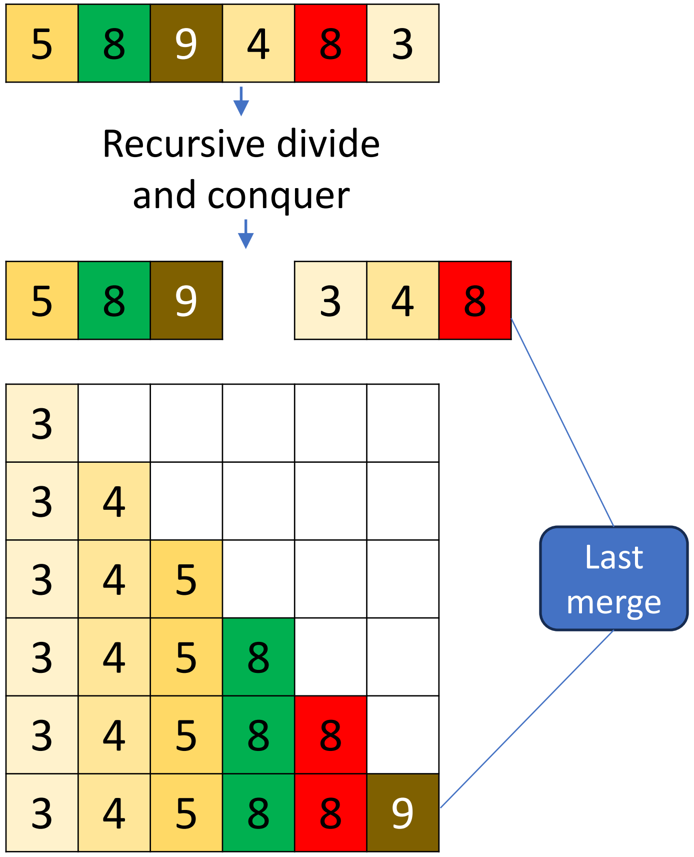
We divide the array into sorted and unsorted portions and iteratively find the minimum values in the unsorted portion. After finding a minimum x, we swap x into the unsorted portion of A: the element swapped could be green 8 which then could be moved behind red 8.
If we were shifting instead of swapping, we could have made it stable but the cost in running time would be very significant.

# Merge Sort?



Is it possible that after sorting green 8 will be AFTER red 8?

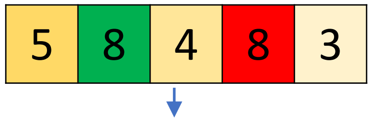
# Merge Sort?



Stable ✓

In the case of equal elements, the element in the left subarray is moved to the output first. Those are the elements that came first in the unsorted array. As a result, they will precede later elements with the same key.

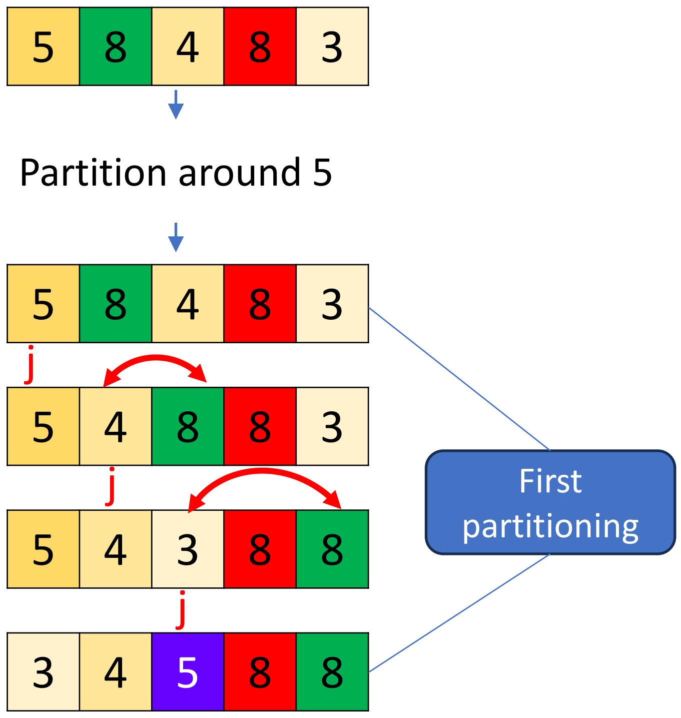
# Quick Sort?



Partition around 5

Is it possible that after partitioning green 8 will be AFTER red 8?

# Quick Sort?



Not stable ✗

The partitioning step can swap the location of elements many times, and thus two elements with equal keys could swap positions in the final output.

## Recursion and tail recursion

## Tail recursion

- In *tail-recursive algorithms* the recursive call is the last statement that is executed: So basically nothing is left to execute after the recursive call.
- That means there is no need to keep the previous stack frame and the recursion can be easily rewritten as iteration: each recursive call can be one step of an iterative algorithm

## Example 1: Sum of values in the list

```
Algorithm sum (List lst)
      if len(lst) == 0:
            return 0
      return lst[0] + sum(lst[1:])
```

Is this a tail-recursive algorithm?

## Sum of values in the list

```
Algorithm sum (List lst)
      if len(lst) == 0:
            return 0
      return lst[0] + sum(lst[1:])
```

Is this a tail-recursive algorithm?

No, because we need to wait for the return of *sum* and only then add the value of the first element of the list

# Sum of values in the list

```
Algorithm sum (List lst)
     if len(lst) == 0:
          return 0
     return lst[0] + sum(lst[1:])
```

Can we convert it into a tail-recursive algorithm?

# Sum of values in the list

```
Algorithm sum (List lst)
      if len(lst) == 0:
            return 0
      return lst[0] + sum(lst[1:])
```

Can we convert it into a tail-recursive algorithm?

↓

```
Algorithm sumTR (List lst, int sumSoFar)
      if len(lst) == 0:
            return sumSoFar
      sumSoFar += lst[0]
      return sumTR(lst[1:], sumSoFar)
```

We call it: `sumTR (list, 0)`

# Example 2. Tail recursion?

```
public void printHello (int N) {
    if (N > 1) {
      System.out.println("Hello");
      printHello (N - 1);
    } else
      System.out.println("Hello");
}
```

- Is this a tail-recursive method?        Yes

## Example 3: Tail recursion?

```
public void printReverse (String s) {
    if (s.length() > 0) {
       printReverse(s.substring(1));
       System.out.print(s.charAt(0));
   }
}
```

- Is this a tail-recursive method? No
- Can we convert it into tail-recursive method?

# Print reverse TR

```
public void printReverse (String s) {
    if (s.length() > 0) {
        printReverse(s.substring(1));
        System.out.print(s.charAt(0));
    }
}
```

↓

```
public void printReverseTR (String s, reverseSoFar) {
    if (s.length() > 0) {
        reverseSoFar = s.charAt(0) + reverseSoFar;
        printReverseTR(s.substring(1), reverseSoFar );
    }
    else {
         System.out.println(reverseSofar);
    }
}
```

## Memoization

Optimizing recursive algorithms

## Memoization

- ***Memoization*** is a term describing an optimization technique where you cache previously computed results, and return the cached result when the same computation is needed again

- It is based on the assumption that calculations take long time, lookups are faster – so we save the results of calculations for future lookup

  Which data structures can be used for efficient lookup?

# Example 1: Fibonacci numbers

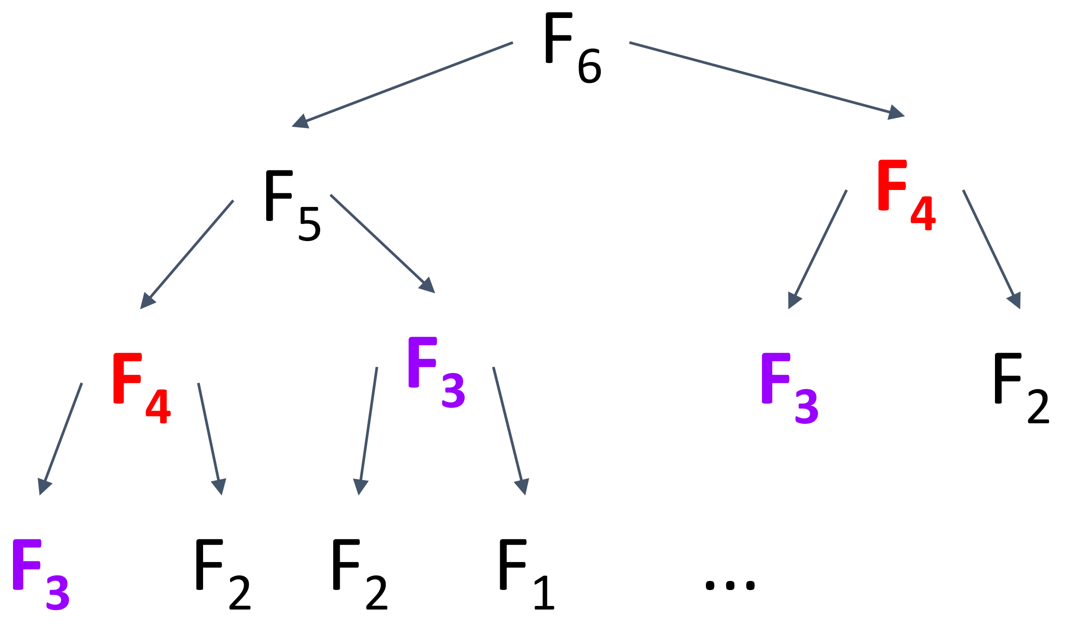
n-th Fibonacci number

$$F_n = \begin{cases} 0, & \text{if } n=0 \\ 1, & \text{if } n=1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$$

```
Algorithm Fib_recurs($n$)
if $n \le 1$:
    return $n$
else:
   return Fib_recurs($n - 1$) + Fib_recurs($n - 2$)
```

# Fibonacci numbers: problems

```
Algorithm Fib_recurs($n$)
if $n \le 1$:
    return $n$
else:
   return Fib_recurs($n - 1$) + Fib_recurs($n - 2$)
```



Running time: $O(2^n)$

Note the repeating calls with the same arguments

We can store the results of computation for each $F_i$ in a position $i$ of an array

In this case for each $i$ $F_i$ will be computed only once

# Fibonacci numbers: with memoization

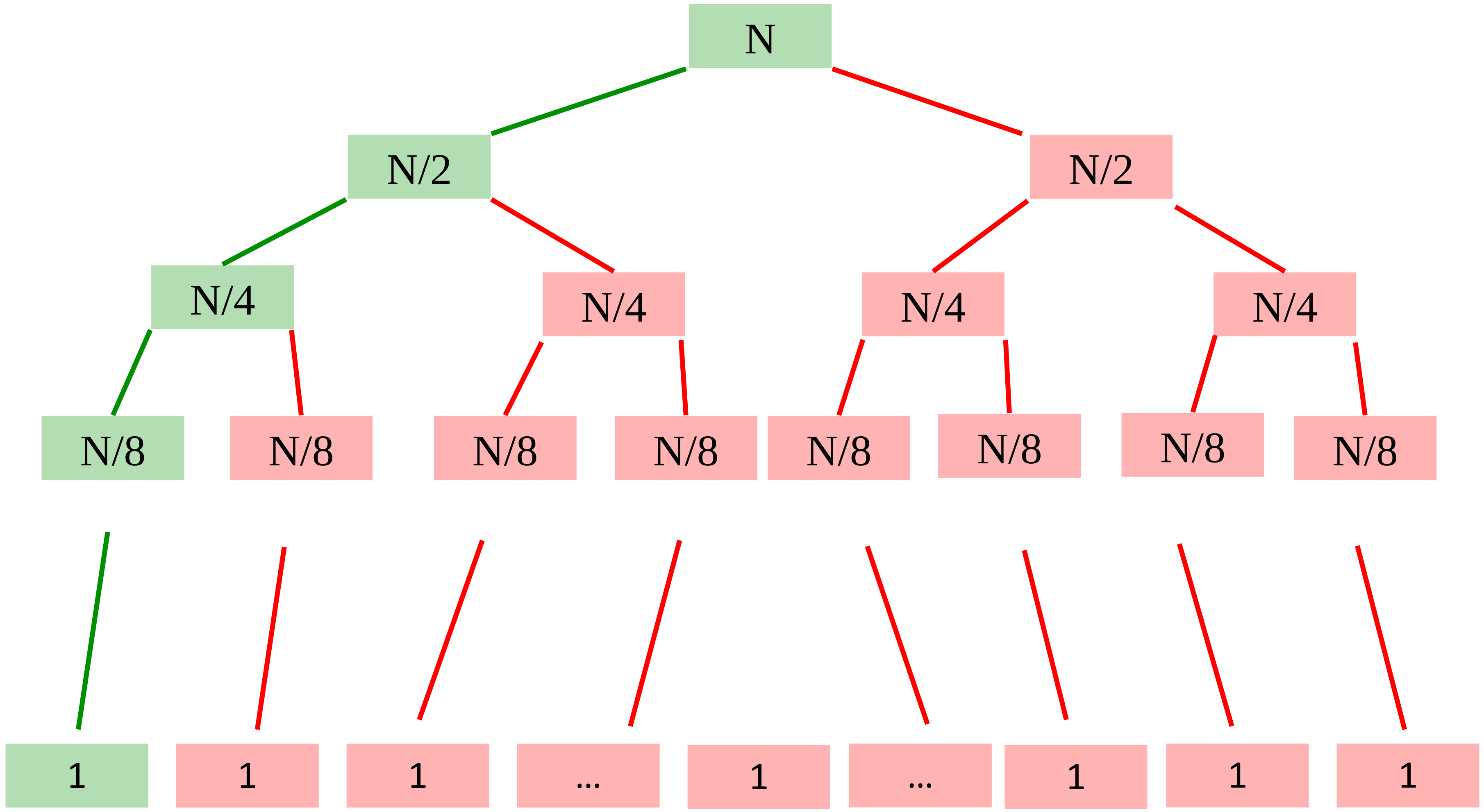
```
Algorithm Fib_recurs_memo(n, FibArray of size n)
  if n ≤ 1:
     FibArray[n] = n
     return n
  else:
    if FibArray[n - 1] is null
      FibArray[n - 1]  = Fib_recurs_memo (n - 1)
    if  FibArray[n -2] is null
      FibArray[n - 2]  = Fib_recurs_memo (n - 2)

  return FibArray[n - 1] + FibArray[n - 2]
```

## Example 2: *power* $X^N$

```
Algorithm power (int X, int N)
  if (N == 0):
    return 1

  if (N % 2 == 0) // N is even
    return power(X, N/2)* power(X, N/2)
  else
    return X* power(X, N/2)* power(X, N/2)
```

What is wrong with this implementation?

# Bad implementation of *power* X$^N$



*power* will be called with the same parameters multiple times

## power $X^N$ with memoization

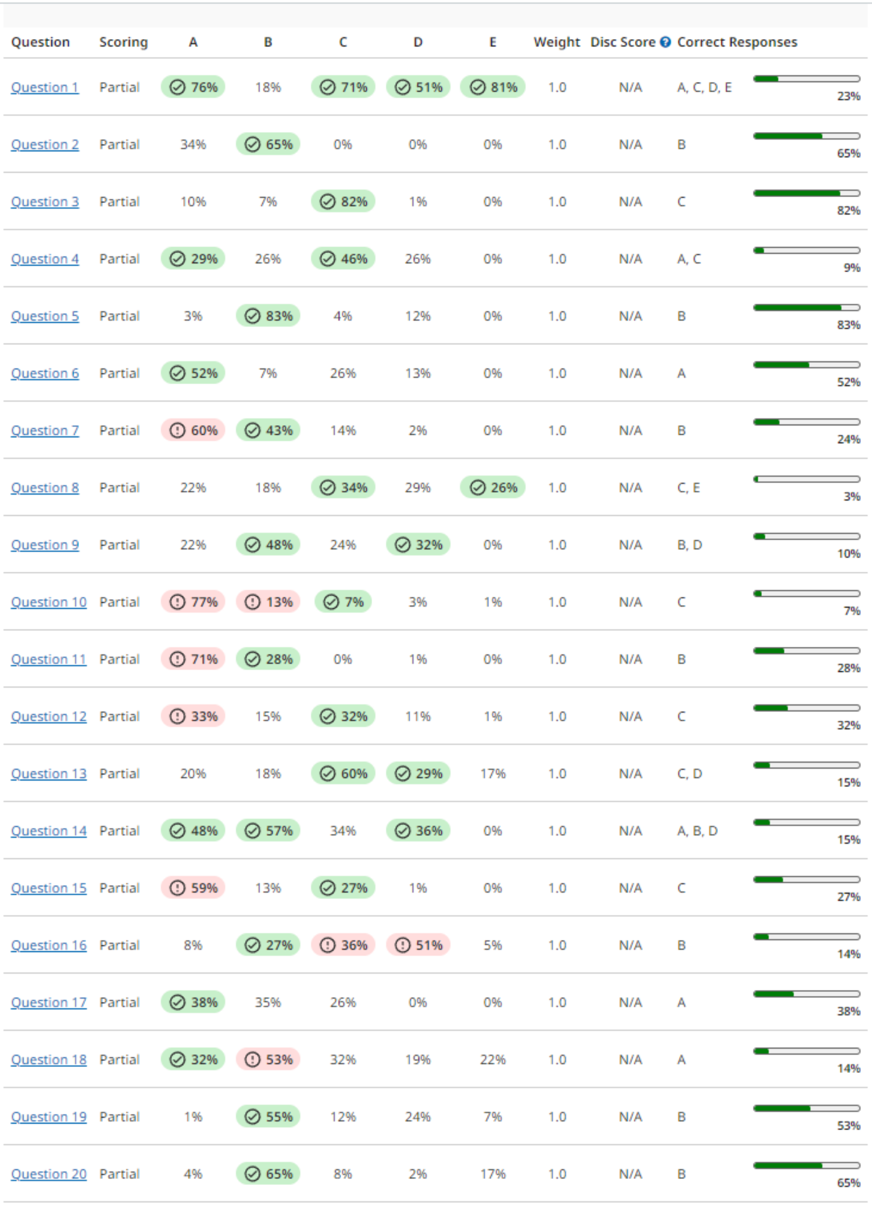
```
global power_table – array of size N
Algorithm power_memo (int X, int N)
  if (N == 0):
    return 1

  if power_table[N/2] is null
    power_table[N/2] = power_memo (X, N/2)

  if (N % 2 == 0) // N is even
    return power_table[N/2] * power_table[N/2]
  else
    return X* power_table[N/2] * power_table[N/2]
```

## Map or Set?

## Which ADT would you use for the following tasks?

1. Finding out whether array A contains duplicates
2. Producing count for each element of A
3. Finding the most popular name from a 2023 database of all newborn babies
4. Removing duplicates from the array
5. Given student id -- finding student’s phone number

# Sample exam: analysis



| Question | Scoring | A | B | C | D | E | Weight | Disc Score | Correct Responses | |
|---|---|---|---|---|---|---|---|---|---|---|
| Question 1 | Partial | ✓ 76% | 18% | ✓ 71% | ✓ 51% | ✓ 81% | 1.0 | N/A | A, C, D, E | 23% |
| Question 2 | Partial | 34% | ✓ 65% | 0% | 0% | 0% | 1.0 | N/A | B | 65% |
| Question 3 | Partial | 10% | 7% | ✓ 82% | 1% | 0% | 1.0 | N/A | C | 82% |
| Question 4 | Partial | ✓ 29% | 26% | ✓ 46% | 26% | 0% | 1.0 | N/A | A, C | 9% |
| Question 5 | Partial | 3% | ✓ 83% | 4% | 12% | 0% | 1.0 | N/A | B | 83% |
| Question 6 | Partial | ✓ 52% | 7% | 26% | 13% | 0% | 1.0 | N/A | A | 52% |
| Question 7 | Partial | ! 60% | ✓ 43% | 14% | 2% | 0% | 1.0 | N/A | B | 24% |
| Question 8 | Partial | 22% | 18% | ✓ 34% | 29% | ✓ 26% | 1.0 | N/A | C, E | 3% |
| Question 9 | Partial | 22% | ✓ 48% | 24% | ✓ 32% | 0% | 1.0 | N/A | B, D | 10% |
| Question 10 | Partial | ! 77% | ! 13% | ✓ 7% | 3% | 1% | 1.0 | N/A | C | 7% |
| Question 11 | Partial | ! 71% | ✓ 28% | 0% | 1% | 0% | 1.0 | N/A | B | 28% |
| Question 12 | Partial | ! 33% | 15% | ✓ 32% | 11% | 1% | 1.0 | N/A | C | 32% |
| Question 13 | Partial | 20% | 18% | ✓ 60% | ✓ 29% | 17% | 1.0 | N/A | C, D | 15% |
| Question 14 | Partial | ✓ 48% | ✓ 57% | 34% | ✓ 36% | 0% | 1.0 | N/A | A, B, D | 15% |
| Question 15 | Partial | ! 59% | 13% | ✓ 27% | 1% | 0% | 1.0 | N/A | C | 27% |
| Question 16 | Partial | 8% | ✓ 27% | ! 36% | ! 51% | 5% | 1.0 | N/A | B | 14% |
| Question 17 | Partial | ✓ 38% | 35% | 26% | 0% | 0% | 1.0 | N/A | A | 38% |
| Question 18 | Partial | ✓ 32% | ! 53% | 32% | 19% | 22% | 1.0 | N/A | A | 14% |
| Question 19 | Partial | 1% | ✓ 55% | 12% | 24% | 7% | 1.0 | N/A | B | 53% |
| Question 20 | Partial | 4% | ✓ 65% | 8% | 2% | 17% | 1.0 | N/A | B | 65% |

- Question 10
- Question 11
- Question 7
- Question 15
- Question 18
- Question 16
- Question 12

# Question 10

First: sort

n log n

```
int n=A.length;
int maxCount = 0;
int currentCount = 0;
int bestNumber = A[0];
int currentNumber = A[0];

for (int i=1; i<n; i++) {
    if (A[i] == currentNumber)
         currentCount ++;
    else {
         currentNumber = A[i];
         currentCount = 1;
     }

    if (currentCount > maxCount) {
         maxCount = currentCount;
         bestNumber = currentNumber;
    }
}
return bestNumber;
```

n

What is the **total** running time of this **entire** solution?

- O (n)
- O ($n^2$)
- O (n log n) for sorting

O(n) + O(n log n) = O(n log n)

## Question 11

In each iteration of insertion sort we insert the first element of an unsorted partition into its proper place in the sorted partition.

Because this partition is sorted, we could find the place for a new element using binary search in time O(log n).

If we do O(n) iterations and in each iteration we do the search for the correct position in time O(log n), this will make the improved insertion sort run in time O(n log n).

Ask yourself:

- Have I ever heard of Insertion sort in time O(n log n)?
- Even after I found the place I need to shift to make space inside the sorted part
- And if not shift – Linked List – I cannot search in time log n

# Question 7

We are devising a recursive algorithm to compute a sum of integer values contained in the nodes of a Linked List.
Which of the following implementations are correct? Select all that apply.

A.

```
int sum (Node node) {
        if (node.next == null)
                return node.data;
        return node.data + sum(node.next);
}
```

Can node be null?

B.

```
int sum (Node node) {
        if (node == null)
                return 0;
        return node.data + sum(node.next);
}
```

C.

```
int sum (Node node) {
        if (node == null)
                return node.data;
        return node.data + sum(node.next);
}
```

Can node be null?

## Question 15

We have an array which is sorted in a reverse order (from the largest to the smallest).
For the Selection sort algorithm this is:

A. The worst-case input
B. The best-case input
C. There are no worst case / best case inputs for Selection sort

Selection sort does not stop early

# Question 18

- The Rabin-Karp pattern matching algorithm matches the hash value of a pattern to a hash value of each substring in the text. Consider using the Rabin Karp pattern matching algorithm for finding ALL occurrences of pattern P in text T (not the first occurrence of a pattern). Now consider that we use the **Monte Carlo** version of the algorithm for solving this task (if the hashes match, we DO NOT check the substring).

Select all the statements about this algorithm that are true.

1. The algorithm runs in time O(N).
2. The algorithm runs in time O(NM).
3. The worst-case input is when every substring of T hashes into the same value as the pattern.
4. The worst-case input is when every substring of T matches the pattern.
5. The worst-case input is when none of substrings of T match the pattern.
6. There is no worst-case input for this algorithm: it performs the same number of operations for any input.

## Question 16

```
1.Algorithm partition (A, l, r)
2.    x ← A[ℓ] # pivot
3.    j ← ℓ
4.    for i from ℓ + 1 to r :
5.        if A[i] > x :
6.            j ← j + 1
7.            swap A[j] and A[i ]
8.    swap A[ℓ] and A[j]
9.    return j
```

J is a divider between <= and >

It will increment only if we found <x

Where is a bug?

## Question 12

Given an array of 3-letter strings A = {“bar”, “dad”, “are”}.

What would this array look like after the first iteration of the Radix Sort?

1. {“are”, “bar”, “dad”}
2. {“bar”, “dad”, “are”}
3. {“dad”, “are”, “bar”}
4. None of the above

# Question 12

Given an array of 3-letter strings A = {“bar”, “dad”, “are”}.

What would this array look like after the first iteration of the Radix Sort?

1. {“are”, “bar”, “dad”}
2. {“bar”, “dad”, “are”}
3. {“dad”, “are”, “bar”}
4. None of the above

The radix sort was distributing to the bins starting with the last character, and then putting back into A according to the order of these bins

## Some useful things to remember

- When placing things into a hash table array with open addressing: this is a circular array so the positions wrap around
- Some sorting algorithms are not based on the comparison of pairs of elements.
- What is it easier to sort: Array? Linked List? Hash Table?