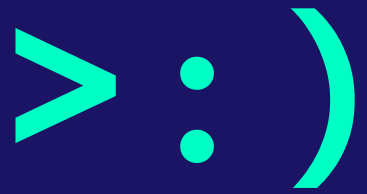
CS2040

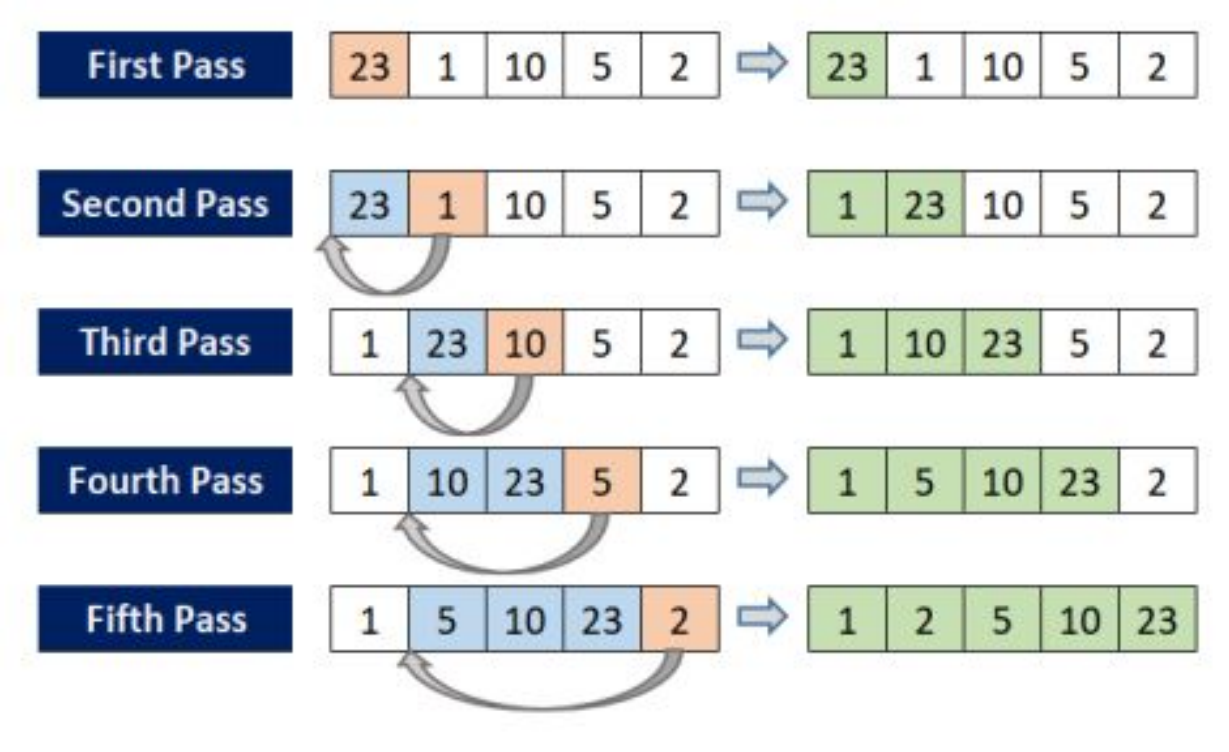
Tutorial 2: Sorting

Nicholas **Russell** Saerang (russellsaerang@u.nus.edu)



ADMIN

Let's start with...



:D

SORTING ALGORITHMS

Selection Sort

Radix Sort

Merge Sort

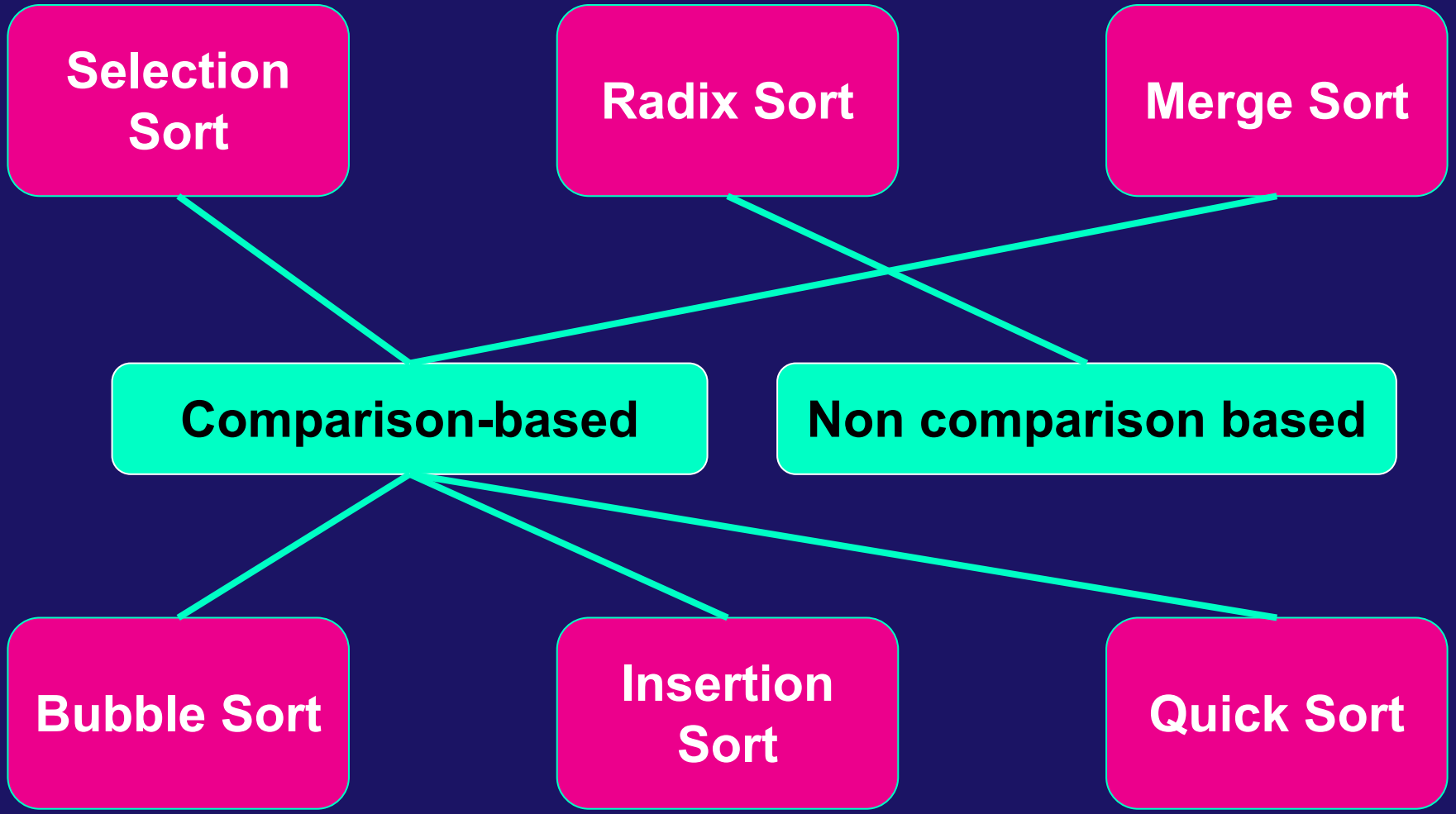
Comparison-based

Non comparison based

Bubble Sort

Insertion Sort

Quick Sort


Selection Sort
Radix Sort
Merge Sort
Comparison-based
Non comparison based
Bubble Sort
Insertion Sort
Quick Sort

Selection Sort

Merge Sort

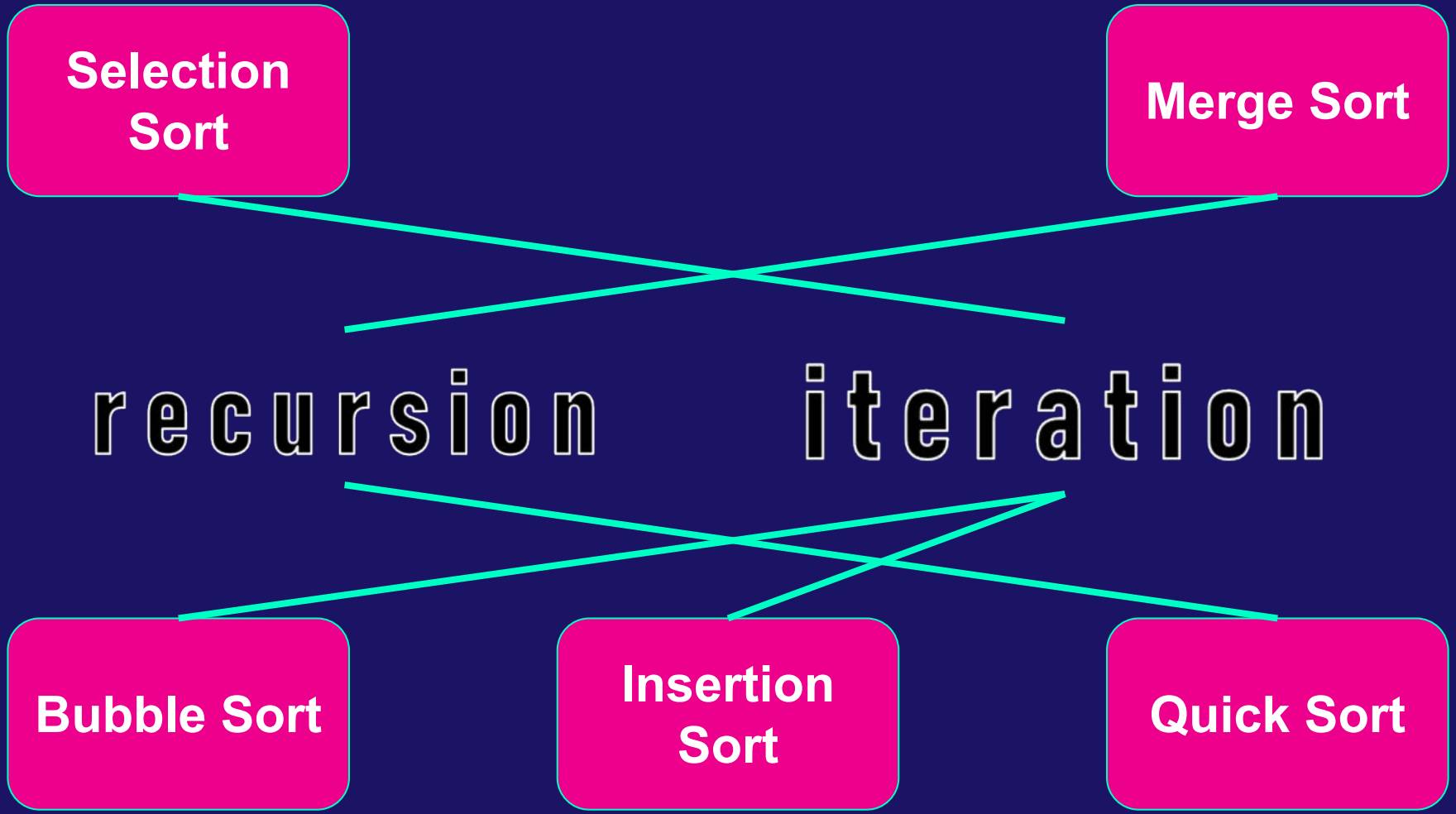
recursion

iteration

Bubble Sort

Insertion Sort

Quick Sort


Selection Sort
Merge Sort
recursion
iteration
Bubble Sort
Insertion Sort
Quick Sort

Selection Sort

Radix Sort

Merge Sort

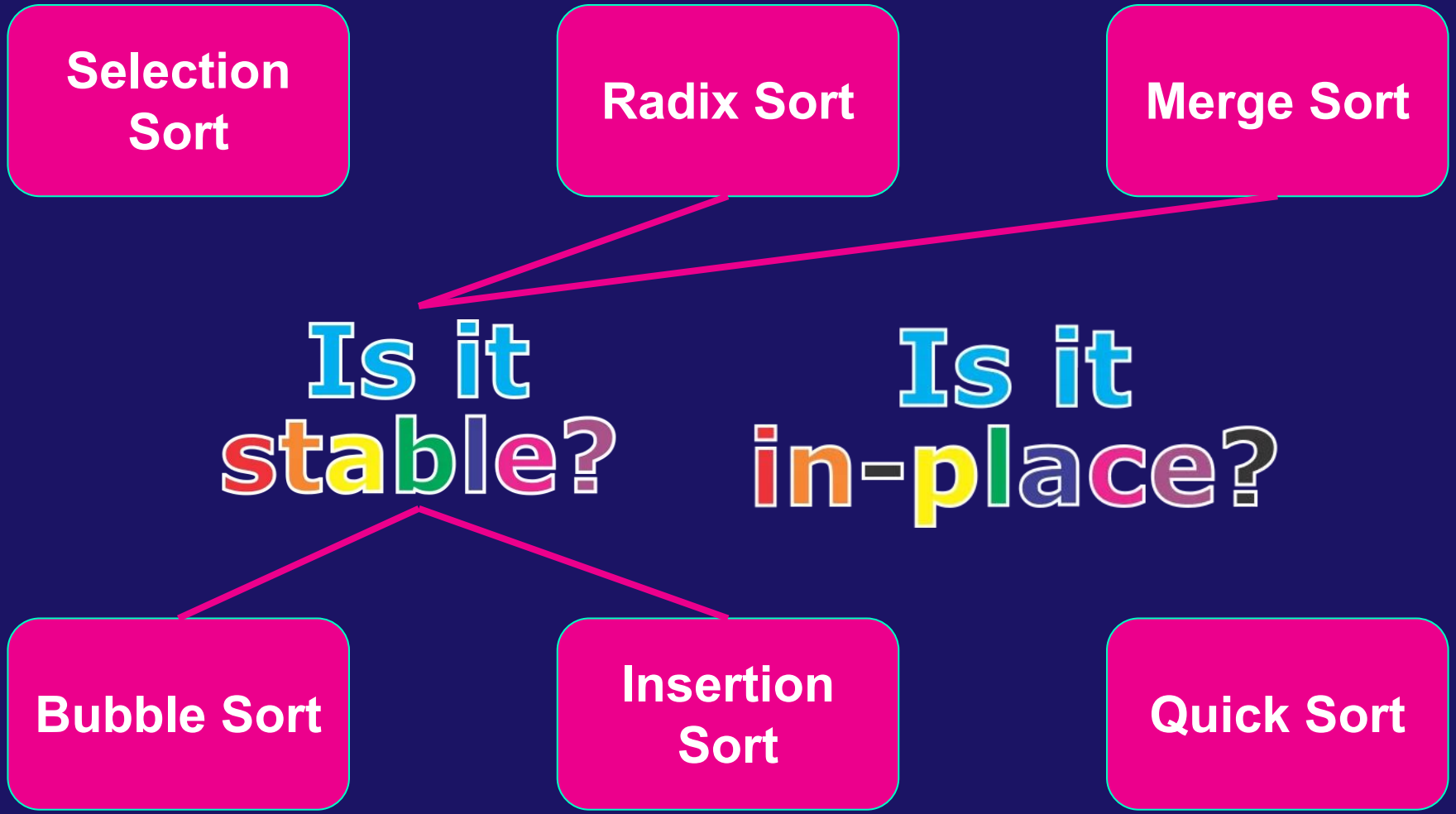
Is it stable?

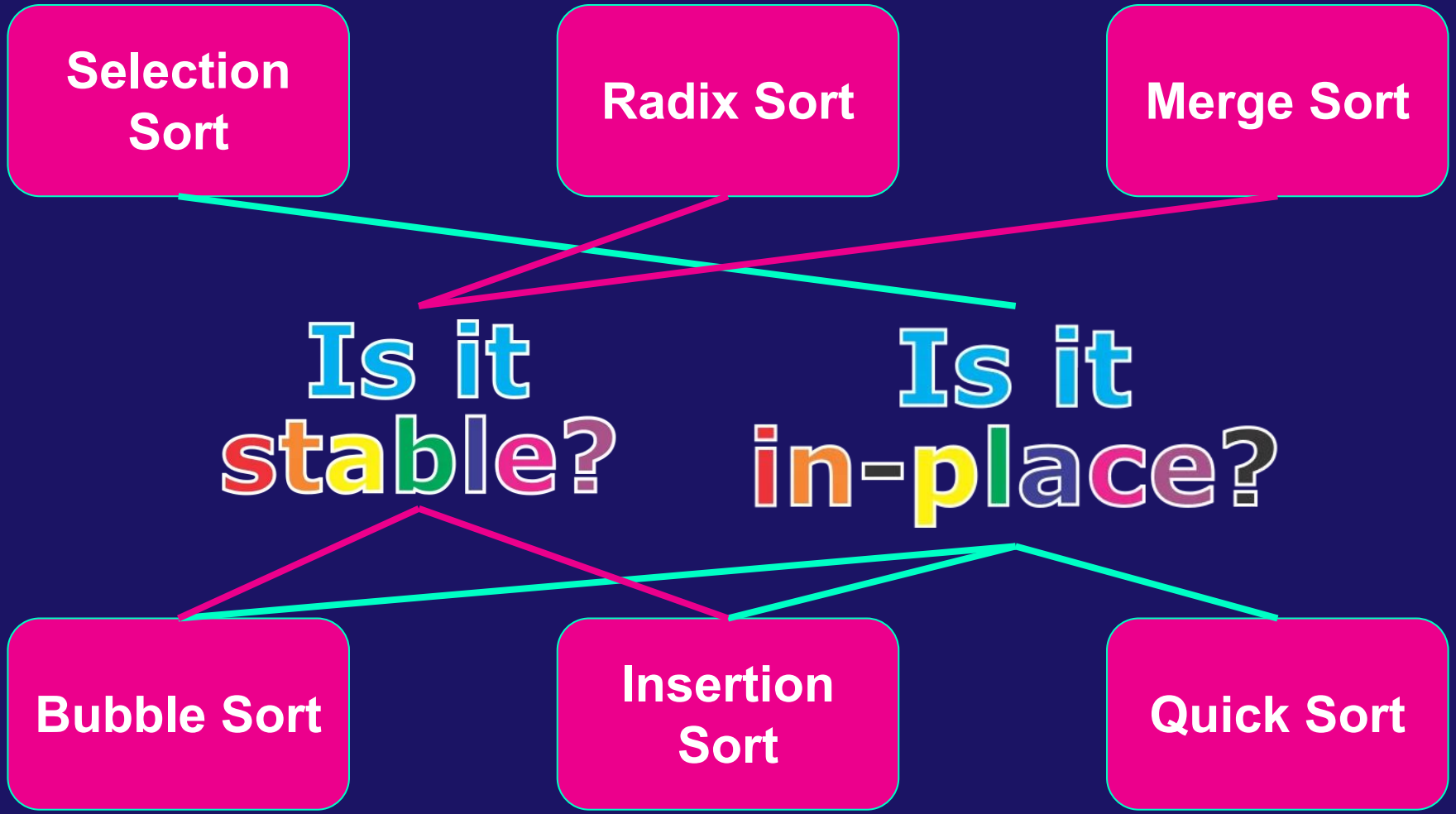
Is it in-place?

Bubble Sort

Insertion Sort

Quick Sort


Selection Sort
Radix Sort
Merge Sort
Is it stable?
Is it in-place?
Bubble Sort
Insertion Sort
Quick Sort


Selection Sort
Radix Sort
Merge Sort
Is it stable?
Is it in-place?
Bubble Sort
Insertion Sort
Quick Sort

Selection Sort

Method:

1. Find largest item in the array
2. Swap that item with the last element in the array
3. Go back to step 1 and exclude the largest item from the array

Example

{ 6, 2, 2, 7, 5 } → { 6, 2, 2, 5, 7 } // first iteration
{ 6, 2, 2, 5, 7 } → { 5, 2, 2, 6, 7 } // second iteration
{ 5, 2, 2, 6, 7 } → { 2, 2, 5, 6, 7 } // third iteration
{ 2, 2, 5, 6, 7 } → { 2, 2, 5, 6, 7 } // fourth iteration
Sorted!

Selection Sort

Time complexity
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$
- Best Case: $O(n^2)$

Stable means if 2 values are the same, the relative position between the two is maintained

{ 6, 2a, 2b, 7, 5 } → { 6, 2a, 2b, 5, 7 } // first iteration
{ 6, 2a, 2b, 5, 7 } → { 5, 2a, 2b, 6, 7 } // second iteration
{ 5, 2a, 2b, 6, 7 } → { 2b, 2a, 5, 6, 7 } // third iteration
{ 2b, 2a, 5, 6, 7 } → { 2b, 2a, 5, 6, 7 } // fourth iteration
Sorted! SELECTION SORT NOT STABLE

Bubble Sort

Method:

1. “Bubble” down the largest item to the end of the array in each iteration by examining the i-th and (i+1)-th item
2. Swap a[i] with a[i+1] if a[i] > a[i+1]

Example

{ 6, 2, 2, 7, 5 } → { 2, 2, 6, 5, 7 } // first iteration
{ 2, 2, 6, 5, 7 } → { 2, 2, 5, 6, 7 } // second iteration
{ 2, 2, 5, 6, 7 } → { 2, 2, 5, 6, 7 } // third iteration
Sorted!
(early termination when there is no swap in one iteration)

Bubble Sort

Time complexity
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$
- Best Case: $O(n)$ when sorted

Is it stable?
Yes, because it only swaps with neighbouring elements if left item is bigger than right item.

Insertion Sort

Method:

1. Start with first element in the array
2. Pick the next element and insert into its proper sorted order
3. Repeat previous step for the rest of the elements in the array

Example

{ 6, 2, 2, 7, 5 }
{ 6, 2, 2, 7, 5 } → { 2, 6, 2, 5, 7 } // first iteration
{ 2, 6, 2, 5, 7 } → { 2, 2, 6, 5, 7 } // second iteration
{ 2, 2, 6, 5, 7 } → { 2, 2, 5, 6, 7 } // third iteration
{ 2, 2, 5, 6, 7 } → { 2, 2, 5, 6, 7 } // fourth iteration
Sorted!

Insertion Sort

Time complexity
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$, faster than bubble sort, see https://youtu.be/TZRWRjq2CAg
- Best Case: $O(n)$ when sorted

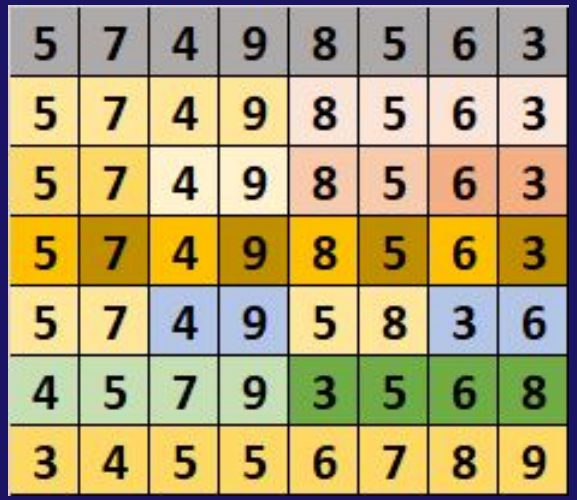
Is it stable?

Yes, because it processes/inserts from left to right, and if we have 2 items with the same value (say a[i] = a[k] where i < k), we can always choose to put the a[k] after a[i].

Merge Sort

Method:

1. Divide Step: divide the larger problem into smaller problems.
2. (Recursively) solve the smaller problems.
3. Conquer Step: combine the results of the smaller problems to produce the result of the larger problem.



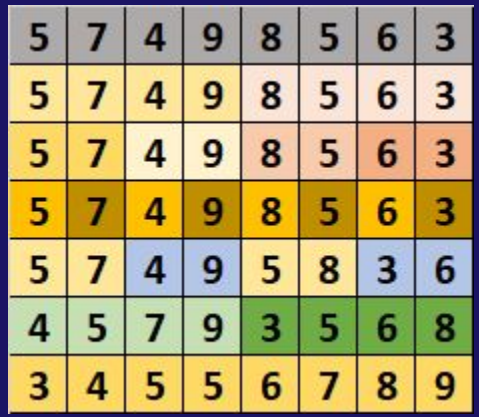

```
// MergeSort(0,7)
// MergeSort(0,3) and MergeSort(4,7)
// MergeSort(0,1), MergeSort(2,3),
   MergeSort(4,5) and MergeSort(6,7)
// MergeSort(0,0), …, MergeSort(7,7)

// Conquer step
// Another conquer step
// Last conquer step
```

Merge Sort

In every conquer step, for example when combining {5,7} and {4,9} into {4,5,7,9}, it requires a temporary array of size n (size of array). So, merge sort has additional space complexity O(n).

There's also a memory for call stack since it is a recursive call (approximately O(log n)) but it is way less than O(n) so we say the space complexity is O(n).



Merge Sort

Time complexity
- Worst Case: O(n log n)
- Average Case: O(n log n)
- Best Case: O(n log n)

There are log n levels, each level merge (conquer step) costs O(n).

Is it stable?

Yes, because we can maintain stability when merging. For example, when we want to merge {2a, 5} with {2b, 7}, if the first number in both array are the same, pick the number from left array first, so it becomes {2a, 2b, 5, 7}.

Quick Sort

Method:

1. Divide Step: Choose pivot, divide array into 2 parts [<p][>=p]
2. (Recursively) sort the two parts
3. Conquer Step: do nothing! No merging needed.

Example

```
{(6, 2, 2, 7, 5, 3)}     // QuickSort(0,5)
{(3, 2, 2, 5), 6, (7)}   // QuickSort(0,3) and QuickSort(5,5)
{(2, 2), 3, (5), 6, 7}   // QuickSort(0,1) and QuickSort(3,3)
{2, (2), 3, 5, 6, 7}     // QuickSort(1,1)
```

Quick sort partition can be done in place (no need temporary array like conquer step in merge sort). However there's still memory for call stack since it is a recursive call (approximately O(log n)).

Quick Sort

Time complexity
- Worst Case: $O(n^2)$ when partition is always the smallest/largest element (only reduce 1 element per level)
- Average Case: $O(n \log n)$
- Best Case: $O(n \log n)$ when partition divides array into halves

There are on average log n levels (max n levels), each level partition costs O(n).

Is it stable?

No, because swapping when partitioning does not maintain stability. Example,

```
{(6, 2a, 2b, 7, 5, 3)}    // QuickSort(0,5)
{(3, 2a, 2b, 5), 6, (7)} // QuickSort(0,3) and QuickSort(5,5)
{(2b, 2a), 3, (5), 6, 7} // QuickSort(0,1) and QuickSort(3,3)
{2b, (2a), 3, 5, 6, 7}    // QuickSort(1,1)
```

Radix Sort

Method:

1. Treat each data to be sorted as a character string.
2. In each iteration, organize the data into groups according to the next character in each data.

Radix Sort

Time complexity

- $O(d \times n)$ where d is the maximum number of digits of the n numeric strings in the array.
  Since d is fixed or bounded, the complexity is $O(n)$.

Is it stable?

Yes, because array is iterated from left to right before pushing elements to queue of their groups.

Summary of Sorting Algos

	Worst Case	Best Case	In-place?	Stable?
Selection Sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	$O(n^2)$	$O(n)$	Yes	Yes
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble Sort 2 (improved with flag)	$O(n^2)$	$O(n)$	Yes	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	No	Yes
Radix Sort (non-comparison based)	$O(n)$ (see notes 1)	$O(n)$	No	Yes
Quick Sort	$O(n^2)$	$O(n \log n)$	Yes	No

01

CHOICE OF SORTING ALGORITHM

Problem 1a

Problem 1.a. You are compiling a list of students (ID, weight) in Singapore, for your CCA. However, due to budget constraints, you are facing a problem in the amount of memory available for your computer. After loading all students in memory, the extra memory available can only hold up to 20% of the total students you have! **Which sorting algorithm should be used to sort all students based on weight (no fixed precision)? Why?**

Answer: Quick Sort.
Due to memory constraint, you will need an in-place sorting algorithm. Hence, a sorting algorithm that is both in-place and works for floating point is Quick Sort.

Do note that: The system requires some extra space on the call stack, due to the recursive implementation of Quick Sort.

Problem 1b

Problem 1.b. After your success in creating the list for your CCA, you are hired as an intern in NUS to manage a student database. There are student records, already sorted by name. However, we want a list of students first ordered by age. For all students with the same age, we want them to be ordered by name. In other words, we need to preserve the ordering by name as we sort the data by age. **Which sorting algorithm should be used to sort the data first by name, then by age, given that the data is already sorted by name? Why?**

Answer: Radix Sort.

The requirements call for a stable sorting algorithm, so that the ordering by name is not lost. Since memory is not an issue, Radix Sort can be used.

Radix Sort has a lower time complexity than comparison based sorts here, O(dn) where d = 2, vs O(n log n) for Merge Sort.

Problem 1c

Problem 1.c. After finishing internship in NUS, you are invited to be an instructor for CS1010E. You have just finished marking the final exam papers randomly. You want to determine your students grades, so you need to sort the students in order of marks. As there are many CA components, the marks have no fixed precision. **Which sorting algorithm should you use to sort the student by marks? Why?**

Answer: Quick Sort.

Being a comparison-based sort, Quick Sort can sort floating point numbers, unlike Radix Sort. Quick Sort is also a good choice because the grades are randomly distributed, resulting in O(n log n) average-case time.

Comparing Quick Sort with Merge Sort here, Quick Sort is in-place, and may run faster.

Problem 1d

Problem 1.d. Before you used the sorting method in Problem 1c, you realize the marks are already in sorted order. However, just to be very sure that you did not cut and paste a student record in the wrong order, you still want to sort the result. **Which sorting algorithm should you use? Why?**

Answer: Insertion Sort.
Insertion sort has an O(n) best-case time, which occurs when elements are already in almost sorted order. Why not bubble?

Hint: [2,3,4,5,6,7,1]

02

K-TH SMALLEST ELEMENT

Problem 2a

Given an unsorted array of n non-repeating (i.e. unique) integers A[1...n], we wish to find the k-th smallest element in the array.

Design an algorithm that solves the above problem in O(n log n) time.

Quick Sort/Merge Sort then output A[k]

Problem 2b

Given an unsorted array of n non-repeating (i.e. unique) integers A[1...n], we wish to find the k-th smallest element in the array.

Design an algorithm that solves the above problem in O(n) time.

Answer: QuickSelect Algorithm
Select random pivot, partition the array, recurse to the part that contain our target search.

QuickSelect Algorithm

Find the 6th smallest from {5, 2, 8, 1, 4, 7, 9}
{4, 2, 1, 5, 8, 7, 9}
// partition with 5 as pivot, we know 5 is in the correct position, i.e. 5 is the 4th smallest. Since 6 > 4, we recurse to the right.

{8, 7, 9} and we want to find the 6 – 4 = 2nd smallest in that set.

{7, 8, 9}
// after partition the right part with 8 as pivot, 8 is now the 2nd smallest from {7,8,9}, so 8 is the 6th smallest from the original array and is the element that we are looking for.

Since we only recurse to one side, and on average the pivot divides the array into halves.
Therefore, the time complexity is n + n/2 + n/4 + … + 1 = O(n).

QuickSelect Algorithm

Algorithm 1 Quickselect Algorithm

```
1: function QuickSelect(A, k, start, end)
2:     j ← Partition(A, start, end)
3:     if k = j then
4:         return A[j]
5:     else if k < j then
6:         return QuickSelect(A, k, start, j − 1)
7:     else
8:         return QuickSelect(A, k, j + 1, end)
9:     end if
10: end function
```

03

WAITING FOR THE DOCTOR

Problem 3

Abridged problem description:

- There are n patients waiting to see the doctor.
- The i-th patient requires consultation time of t_i minutes.
- Doctor serves 1 patient at a time, others wait.
- Any time in between serving two patients is negligible.
- All n patients must be served.

Describe the most efficient algorithm to find the minimum total waiting time required to serve all patients. What is the running time of your algorithm?

Problem 3

Answer:
The doctor should serve the patients with shorter consultation times first, so that patients with shorter consultation time would not need to unnecessarily spend more time at the clinic by waiting for patients with longer consultation time. This suggests that we should process patients in increasing consultation time.

Proof by contradiction:
Suppose not. Then in the optimal ordering of patients, there is at least one pair of patients a and b such that $t_a < t_b$ but a visits the doctor after b. If the positions of a and b are exchanged, then there will be a reduction in the total waiting time of at least $t_b - t_a$, implying that this is not the optimal ordering of patients, a contradiction.

Problem 3

Example:

i	1	2	3	4
t_i	3	5	8	11
Waiting time	0	3	8	16

i	1	2	3	4
t_i	3	**8**	**5**	11
Waiting time	0	3	**11**	16

Problem 3

Algorithm 2 Solution to Problem 3

1: $T[1 \ldots n] \leftarrow$ consultation time t_i of patients
2: sort T in ascending order
3: $total \leftarrow 0$ ▷ total waiting time
4: **for** $i \leftarrow 1$ to $n-1$ **do**
5: $\quad total \leftarrow total + (n-i) \times T[i]$ ▷ $n-i$ patients need to wait for ith patient
6: **end for**
7: **output** *total*

Algorithm 2 runs in $O(n \log n)$ time.

04

MISSING FAMILY MEMBERS

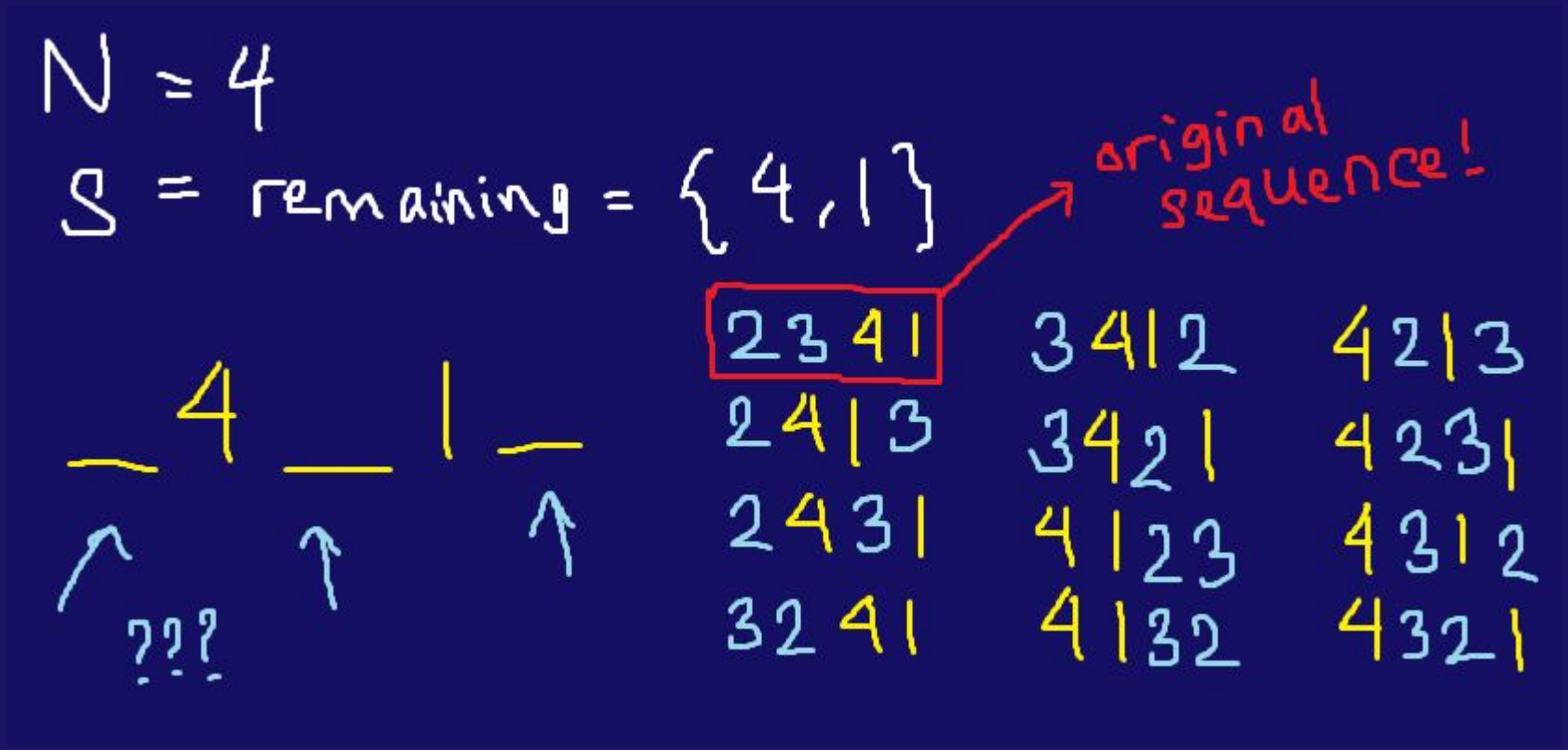
Problem 4

Abridged problem description:

- N family members line up to take photos, each will wear a shirt having a number x where $1 \leq x \leq N$.
- Some numbers are removed, the order of the remaining number does not change. For example, 1,2,4,5,6,3 becomes 1,4,6,3 after removing 2 and 5 from the photo.
- The original sequence in the photo will be the first such permutation that contains the remaining subsequence.

Describe the most efficient algorithm to restore the original photo sequence. What is the running time of your algorithm?

Problem 4


N = 4
S = remaining = {4, 1}
original sequence!
_ 4 _ _ 1 _
???
2341
2413
2431
3241
3412
3421
4123
4132
4213
4231
4312
4321

Problem 4

Answer:
Since the original sequence is the 1st permutation in ascending order to include the remaining subsequence S, it also means that the missing subsequence S’ in that permutation must be in ascending order (otherwise it cannot be the 1st permutation in ascending order to include S)!

To piece back the original sequence, simply merge S and S’ using the merge method of merge sort!

Problem 4

Algorithm 3 Solution to Problem 4

```
1: A[1...N] ← array containing S
2: B[1...N] ← boolean array initialized to false
3: C[1...N] ← array to contain S'
4: for each element x in A do
5:     B[x] ← true
6: end for
7: for i ← 1 to N do
8:     if B[i] = false then
9:         append i to the back of C
10:    end if
11: end for
12: output MERGE(A, C)
```

THE END!