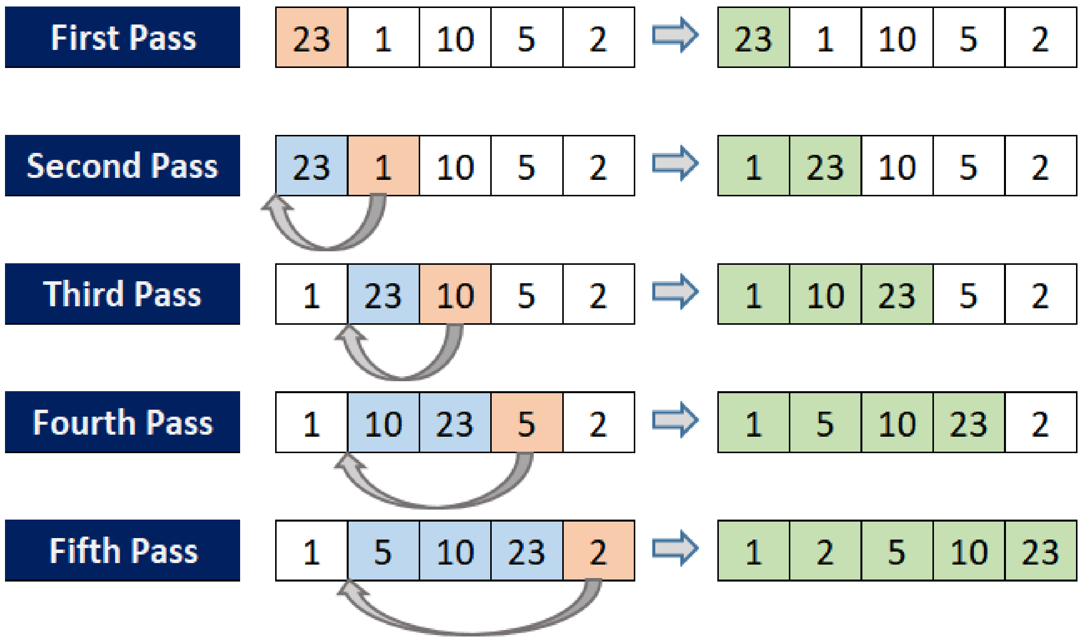
CS2040S TUTORIAL 2

Lucia Tirta Gunawan
luciatirtag@u.nus.edu

WELCOME TO THE SECOND TUTORIAL

THIS WEEK'S TOPIC IS…

SORTING



Selection Sort

Radix Sort

Merge Sort

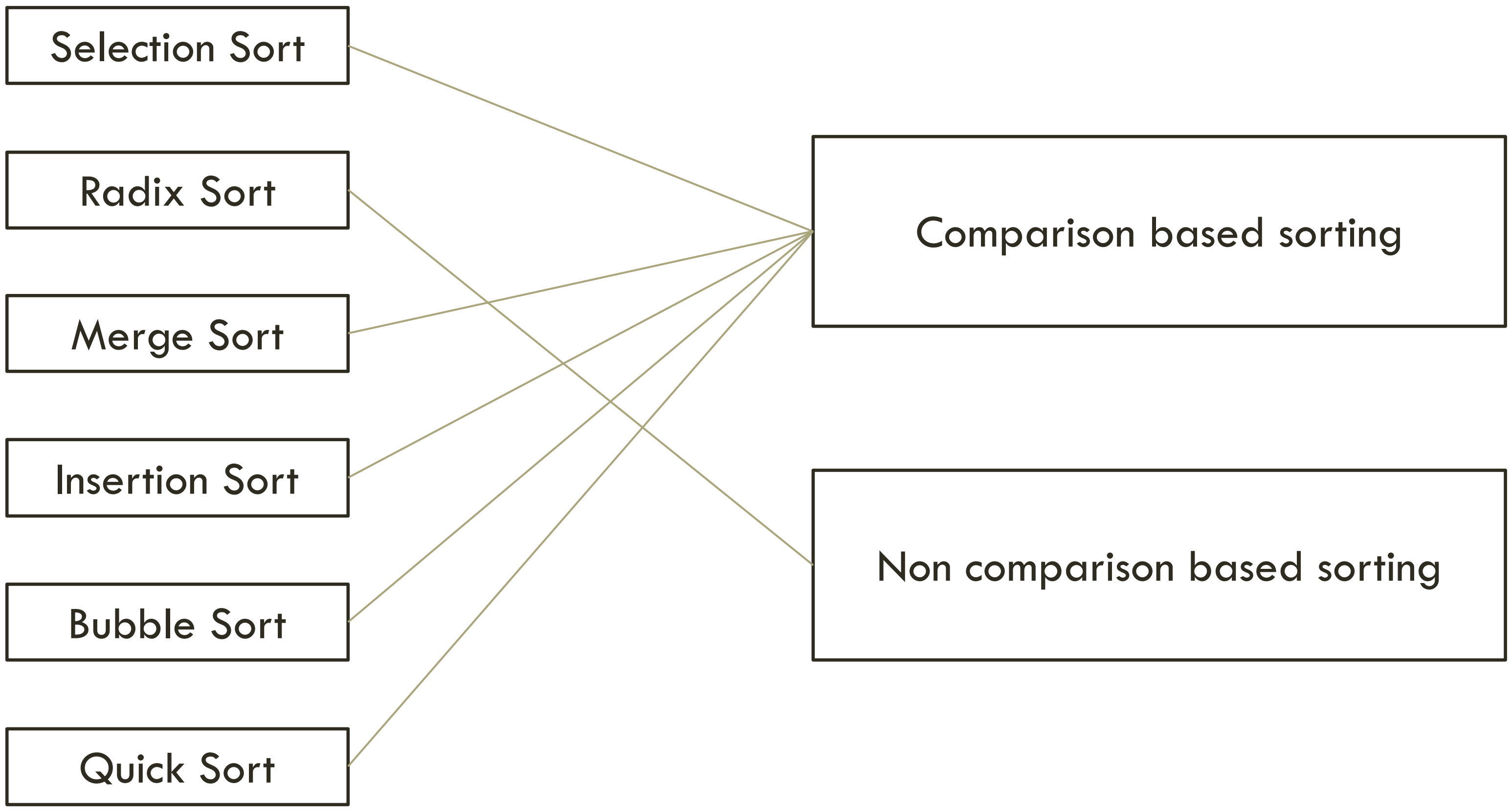
Insertion Sort

Bubble Sort

Quick Sort

Comparison based sorting

Non comparison based sorting


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

Selection Sort

Radix Sort

Merge Sort

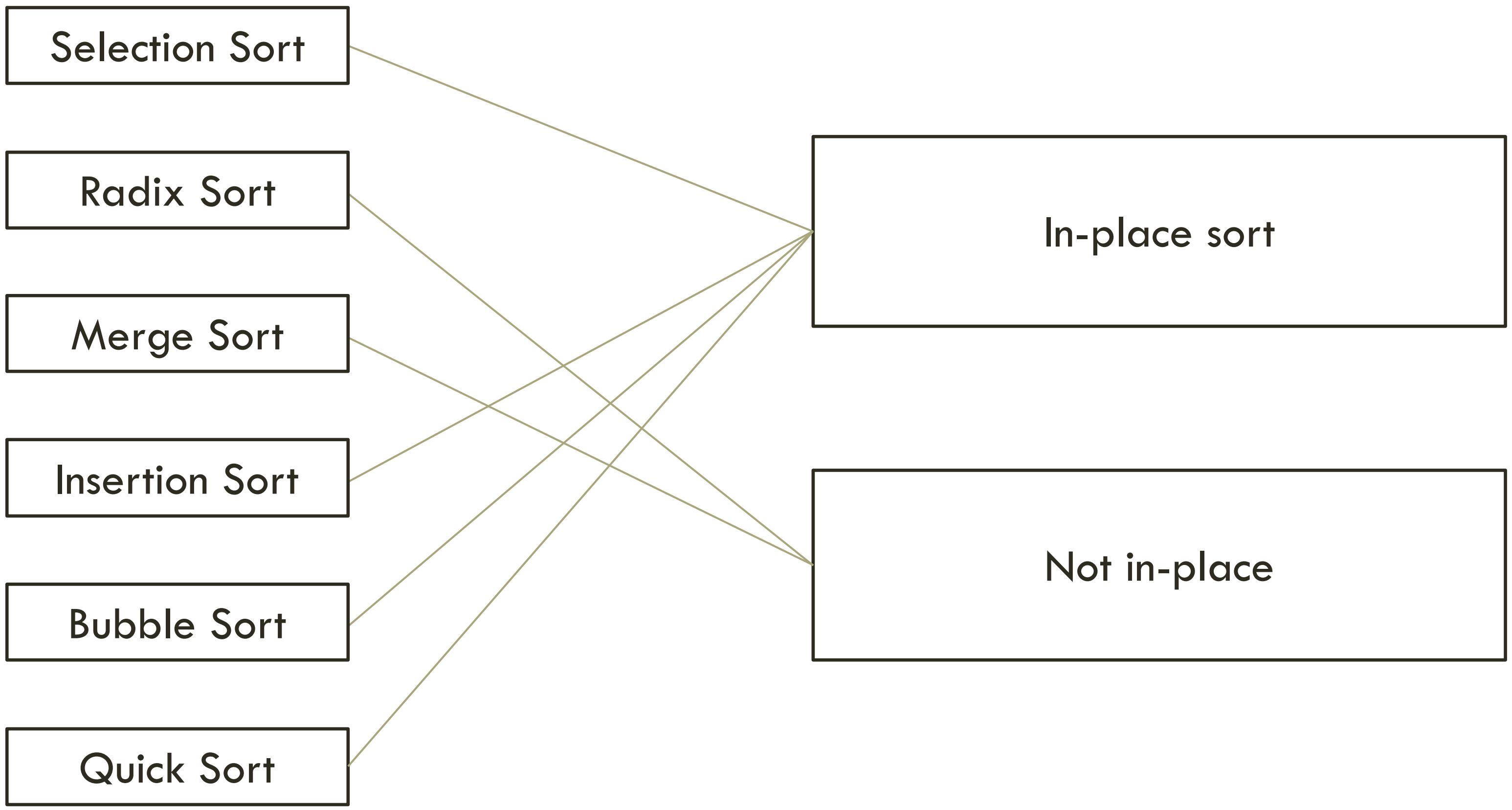
Insertion Sort

Bubble Sort

Quick Sort

In-place sort

Not in-place


Selection Sort
Radix Sort
Merge Sort
Insertion Sort
Bubble Sort
Quick Sort
In-place sort
Not in-place

Selection Sort

Radix Sort

Merge Sort

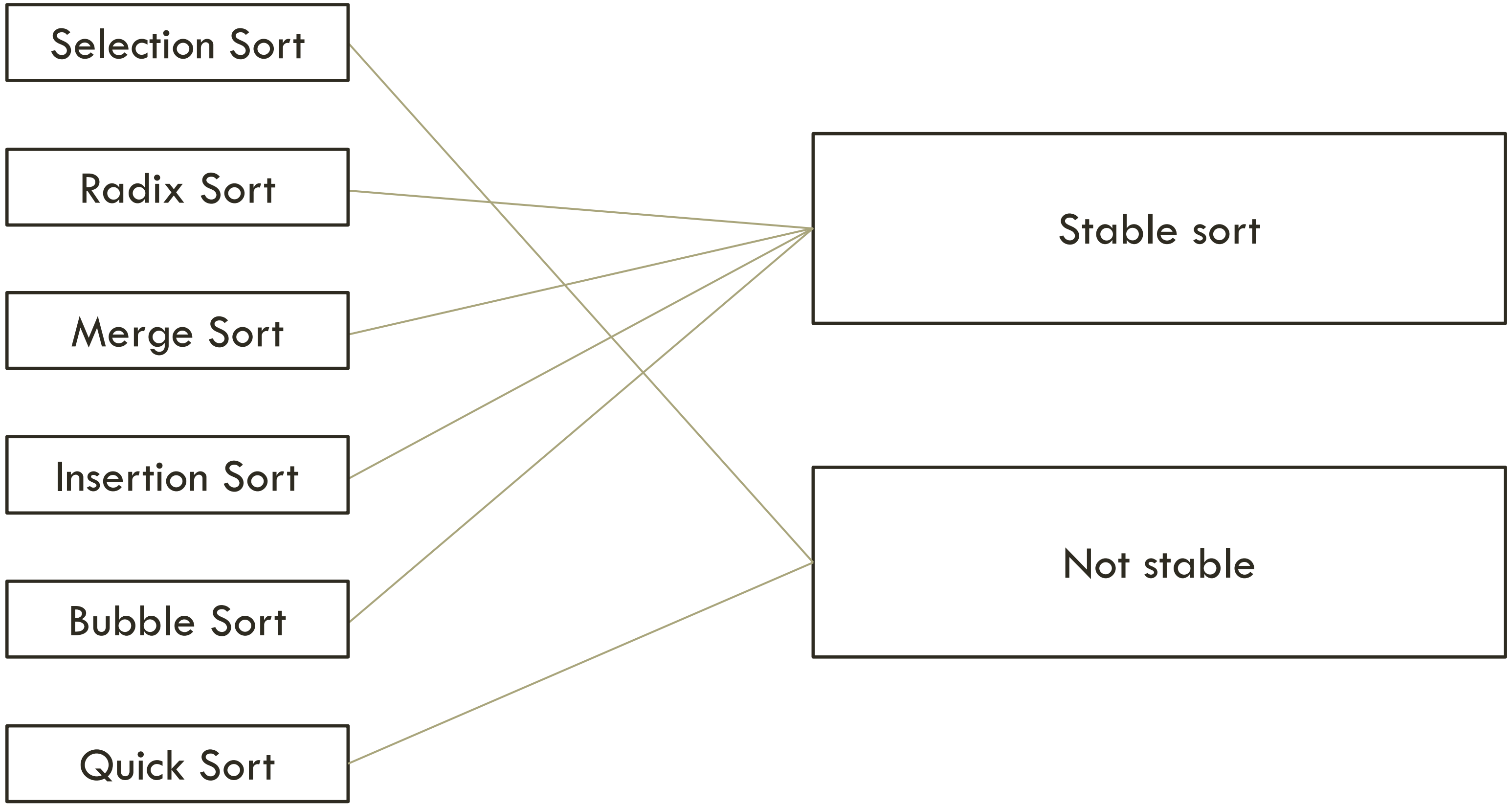
Insertion Sort

Bubble Sort

Quick Sort

Stable sort

Not stable


Selection Sort
Radix Sort
Merge Sort
Insertion Sort
Bubble Sort
Quick Sort
Stable sort
Not stable

1. 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!

1. SELECTION SORT

Time complexity

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

Is it stable?

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

2. 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 items
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)

2. 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 swap with neighbouring elements if left item is bigger than right item

3. 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!

3. INSERTION SORT

Time complexity

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

Is it stable?

YES because it process/insert from left to right, and if we have 2 items with the same value (lets say $a[i] = a[k]$ where $i < k$) , we can always choose to put the $a[k]$ after $a[i]$

4. 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.

Example

{ 6, 2, 2, 7, 5, 3 }	// MergeSort(0,5)
{ 6, 2, 2 } { 7, 5, 3 }	// MergeSort(0,2) and MergeSort(3,5)
{ 6, 2 } { 2 } { 7, 5 } { 3 }	// MergeSort(0,1), MergeSort(2,2), MergeSort(3,4), MergeSort(5,5)
{ 6 } { 2 } { 2 } { 7 } { 5 } { 3 }	// MergeSort(0,0), MergeSort(1,1), MergeSort(3,3), MergeSort(4,4)
{ 2, 6 } { 2 } { 5, 7 } { 3 }	// Conquer step
{ 2, 2, 6 } { 3, 5, 7 }	// Conquer step
{ 2, 2, 3, 5, 6, 7 }	// Conquer step

In every conquer step, for example when combining { 2, 2, 6 } and { 3, 5, 7 } into { 2, 2, 3, 5, 6, 7 }, it requires 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)

4. 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 }

5. 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), 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))

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

6. RADIX SORT (NON COMPARISON BASED SORT)

Method:

1. Treats 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.

Example

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

6. RADIX SORT (NON COMPARISON BASED SORT)

Time complexity

- O(dxn) 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 ALGORITHM

	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

Q1A. CHOICE OF SORTING ALGORITHM

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

Q1B. CHOICE OF SORTING ALGORITHM

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.

Q1C. CHOICE OF SORTING ALGORITHM

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.

Q1D. CHOICE OF SORTING ALGORITHM

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.

Q2A. K-TH SMALLEST ELEMENT

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

Problem 2.a. Design an algorithm that solves the above problem in $O(n \log n)$ time.

Answer:

Quick Sort/MergeSort then output A[k]

Q2B. K-TH SMALLEST ELEMENT

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

Problem 2.b. Design an algorithm that solves the above problem in expected $O(n)$ time. Briefly explain why your algorithm is correct. *Hint: Modify the quicksort algorithm.*

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

Find the 6th smallest
{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 the set {8, 7, 9}.
{4, 2, 1, 5, 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)

Q2B. K-TH SMALLEST ELEMENT

Pseudocode for QuickSelect Algorithm

Algorithm 1 Quickselect Algorithm

```
function QuickSelect(A, k, start, end)
    j ← Partition(A, start, end)
    if k = j then
        return A[j]
    else if k < j then
        return QuickSelect(A, k, start, j − 1)
    else
        return QuickSelect(A, k, j + 1, end)
    end if
end function
```

Q3. WAITING FOR THE DOCTOR

Abridged problem description:

- There are n patients waiting to see the doctor.
- The ith patient requires consultation time of ti minutes.
- Doctor serve 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?

Q3. WAITING FOR THE DOCTOR

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 $ta < tb$ 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 $tb - ta$, implying that this is not the optimal ordering of patients, a contradiction.

Q3. WAITING FOR THE DOCTOR

Algorithm 2 Solution to Problem 3

```
T[1...n] ← consultation time t_i of patients
sort T in ascending order
total ← 0                                          ▷ total waiting time
for i ← 1 to n − 1 do
    total ← total + (n − i) × T[i]                 ▷ n − i patients need to wait for ith patient
end for
output total
```

This algorithm runs in O(n log n)

Q4. MISSING FAMILY MEMBERS

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 example1,2,4,5,6,3 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. For example, if N = 4 and the remaining subsequence is 4,1. Possible permutations are (2,3,4,1), (2,4,1,3), (2,4,3,1), (3,2,4,1), (3,4,1,2), (3,4,2,1), (4,1,2,3), (4,1,3,2), (4,2,1,3), (4,2,3,1), (4,3,1,2), (4,3,2,1). Among all possibilities, the smallest is 2,3,4,1. Therefore the original photo is 2,3,4,1

Describe the most efficient algorithm to restore the original photo sequence

What is the running time of your algorithm?

Q4. MISSING FAMILY MEMBERS

Answer:

Since the original sequence is the 1st permutation in ascending order to include the remaining subsequence S, it means that the removed subsequence S’ in the original sequence 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!

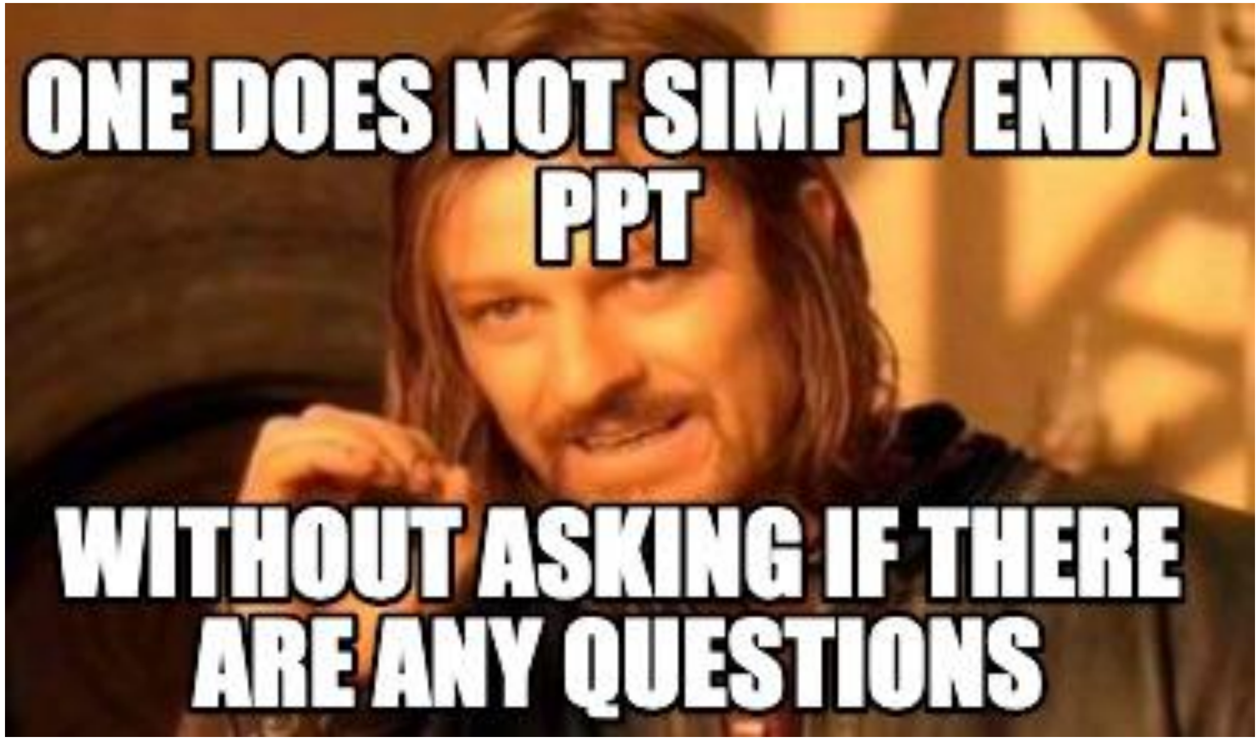
Q4. MISSING FAMILY MEMBERS

Algorithm 3 Solution to Problem 4

$A[1 \ldots N] \leftarrow$ array containing S
$B[1 \ldots N] \leftarrow$ boolean array initialized to $false$
$C[1 \ldots N] \leftarrow$ array to contain S'
for each element x in A **do**
    $B[x] \leftarrow true$
end for
for $i \leftarrow 1$ to N **do**
    if $B[i] = false$ **then**
        append i to the back of C
    end if
end for
output $\textsc{Merge}(A, C)$

This algorithm runs in O(n)

ANY QUESTION?



SEE YOU IN THE NEXT TUTORIAL!