Tutorial 2: Sorting

August 29, 2022

Gu Zhenhao

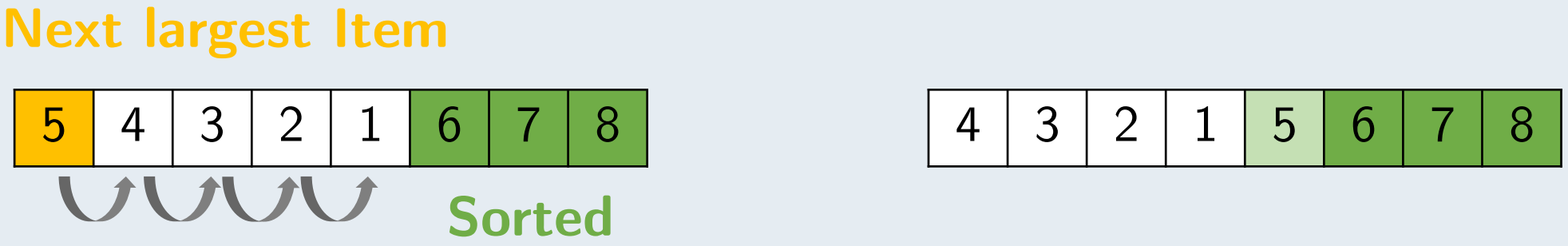
** Partly adopted from tutorial slides by Wang Zhi Jian.*

Sorting Algorithms

Which sorting algorithm to choose?

Bubble Sort

- **Intuition**: compare neighbouring numbers and fix all wrong orders.

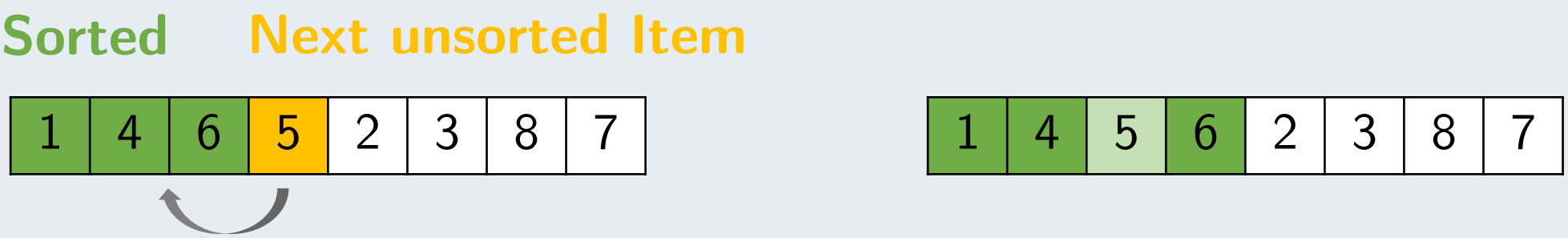


If no swap is done in an iteration, can terminate early.

- **Pros**: Easiest to implement; in-place; fast on (almost) sorted array.
- **Cons**: Generally takes $O(n^2)$ time.

Insertion Sort

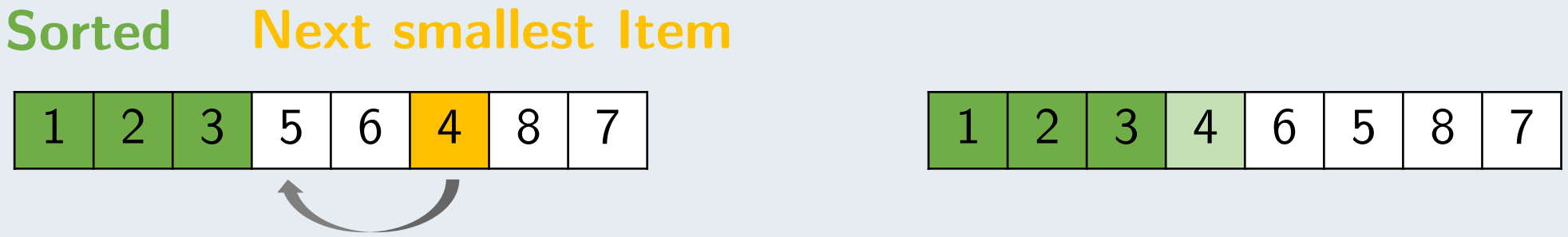
- **Intuition**: Insert each unsorted item into the sorted array.



- **Pros**: simple; in-place; fast on (almost) sorted or small array.
- **Cons**: Generally takes $O(n^2)$ time.

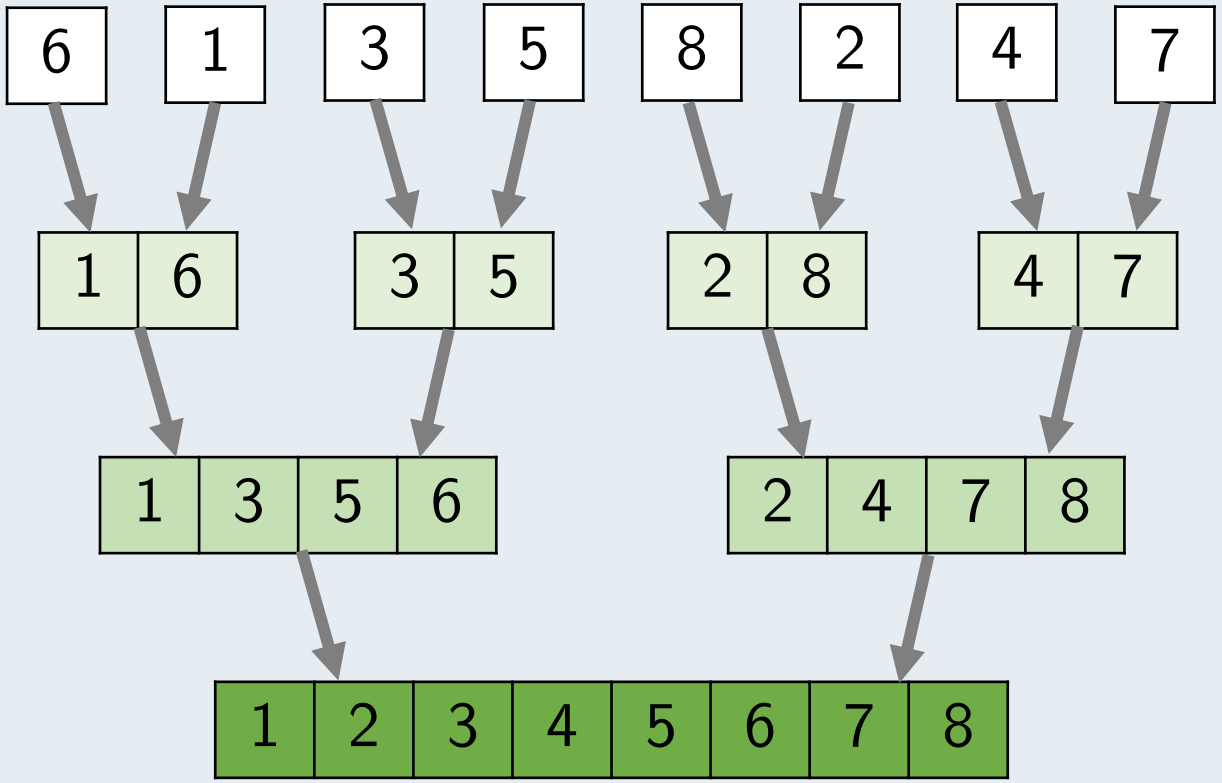
Selection Sort

- **Intuition**: Repeatedly find smallest item among the unsorted ones.



- **Pros**: in-place; fast on small array; minimizes number of swaps.
- **Cons**: Always takes $O(n^2)$ time; not stable.

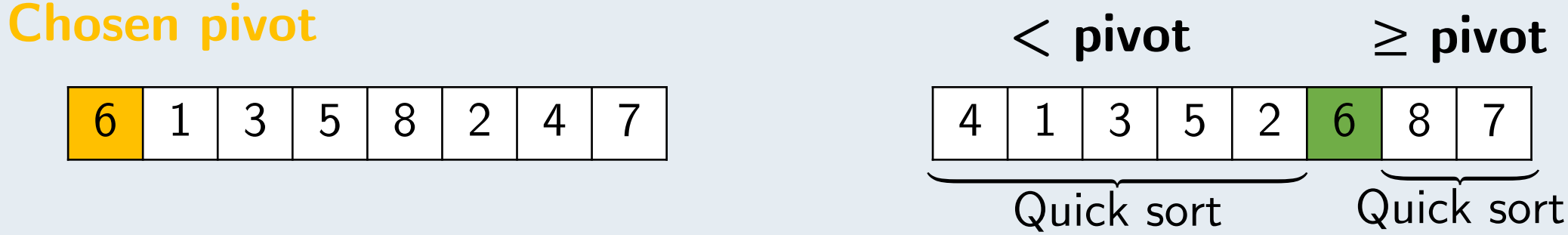
Merge Sort



- **Intuition**: Divide the array into smaller ones, sort each them merge them.
- **Pros**: Always $O(n \log n)$ time.
- **Cons**: Need $O(n)$ extra space for merging and $O(\log n)$ stack space for recursion.

Quick Sort

- **Intuition**: Use *partition* method to find the correct position for each item.



- **Pros**: generally fast $O(n \log n)$; "weakly" in-place (still need $O(\log n)$ stack space for recursion).
- **Cons**: not stable.

Radix Sort

62, 33, 82, 47, 35

(6**2**, 8**2**), (3**3**), (3**5**), (4**7**)

62, 82, 33, 35, 47

(**3**3, **3**5), (**4**7), (**6**2), (**8**2)

- **Intuition**: Sort numbers from the least significant digit to most significant ones.
- **Pros**: Only $O(n)$ time; no comparison needed.
- **Cons**: Assumes that data is in certain range; not in-place.

Pause and Ponder 1

It seems that radix sort is generally faster than comparison-based sorting.

Question: If the largest integer in the array is of $O(n)$, what should be the complexity of radix sort?

** Find the answer in the appendix!*

Summary

	Worst Case	Average Case	Best Case	In-place	Stable
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$	Yes	Yes
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble Sort 2	$O(n^2)$	$O(n^2)$	$O(n)$	Yes	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	Yes
Radix Sort	$O(n)$	$O(n)$	$O(n)$	No	Yes
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	Yes (weakly)	No

Choosing the correct algorithm

Recipe: you should know

1. What are the **requirements**? Do we need ...
   - **In-place** (use no extra memory)?
   - **Stable** (keep relative position of equal elements)?
2. What do the given **input** look like? Does it favour some of the algorithms?
3. Among the remaining algorithms, which is the **best** (time/space)?

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!
Sort all students based on weight (no fixed precision).

Requirements: in-place. (Merge Sort X)

Input: weight, with no fixed precision. (Radix Sort X)

Best: Insertion Sort / Quick Sort

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.

Requirements: stable. (Quick Sort X)

Input: age. (Radix Sort $\surd$)

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.

Requirements: None.

Input: marks, with no fixed precision. (Radix Sort X)

Best: Insertion Sort / Merge Sort / Quick Sort

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.

Requirements: None.

Input: almost sorted. (Insertion Sort $\surd$)

Pause and Ponder 2

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.

Question. Is it as good to also use Bubble Sort 2?

** Find the answer in the appendix!*

Applications of Sorting

How can we utilize those algorithms (and their variants)?

Problem 2

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.

Strategy:

1. Think of a trivial answer.
2. Try with some simple example
3. See where we can improve!

Problem 2

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.

Trivial answer:

- Use quick sort to sort the array. ($O(n \log n)$ time)
- Output the k-th element.

Problem 2

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.

Simple Example: suppose we want find the **6-th** smallest element:

Chosen pivot

6	1	3	5	8	2	4	7

$<$ pivot **$\geq$ pivot**

4	1	3	5	2	6	8	7

No need to continue sorting!

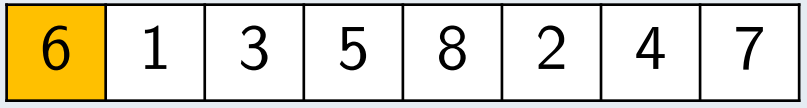
Problem 2

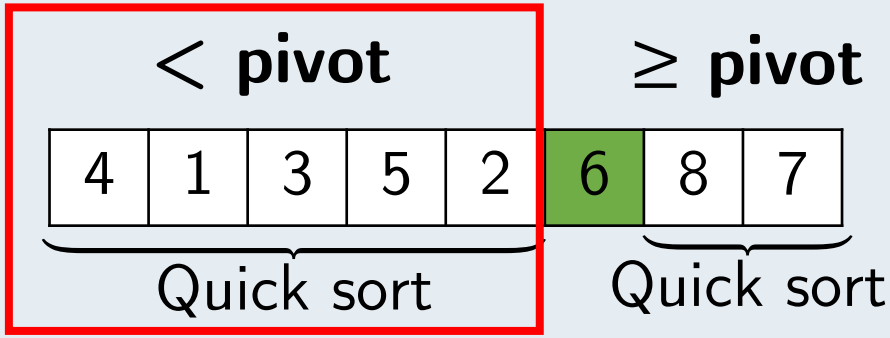
Given an unsorted array of n non-repeating (i.e. unique) integers $A[1 \ldots n]$, we wish to find the k-th smallest element in the array.

Simple Example: suppose we want find the **7-th** smallest element:

No need to worry about this part!

Chosen pivot





Problem 2

Idea:

- Use *partition* method,
- Do Quick Sort only on the relevant part.
- Stop when the pivot is in the correct position!

Simple Example: suppose we want find the **7-th** smallest element:

Quick Select 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
```

Question: What is the time complexity?

Quick Select Algorithm

Algorithm 1 Quickselect Algorithm

```
1:  function QuickSelect(A, k, start, end)
2:      j ← Partition(A, start, end)                    n
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
```

$T(n)$

$$T(n) = T(n/2) + n$$
$$= T(n/4) + (1 + 1/2)n$$
$$= \cdots$$
$$= T(1) + \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right)n$$
$$= O(n) \quad \text{(best case)}$$

$T(n-1)$ in worst case

$T(n/2)$ in best case

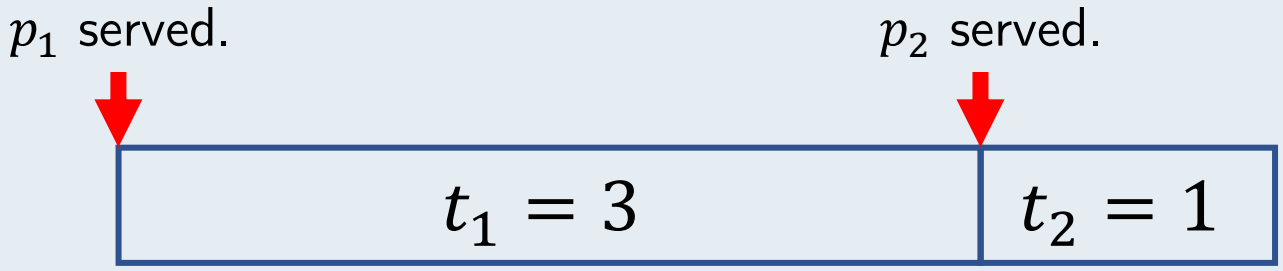
$T(3n/4)$ in average case

Problem 3

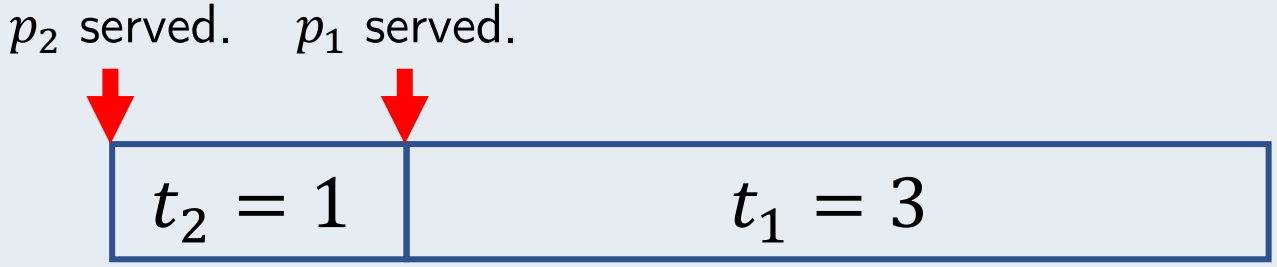
- One doctor, n patients,
- requires consultation time $t_1, \ldots, t_n$ respectively,
- Order the patients so that total waiting time is minimized.

$t_1 = 3$ $t_2 = 1$

Problem 3



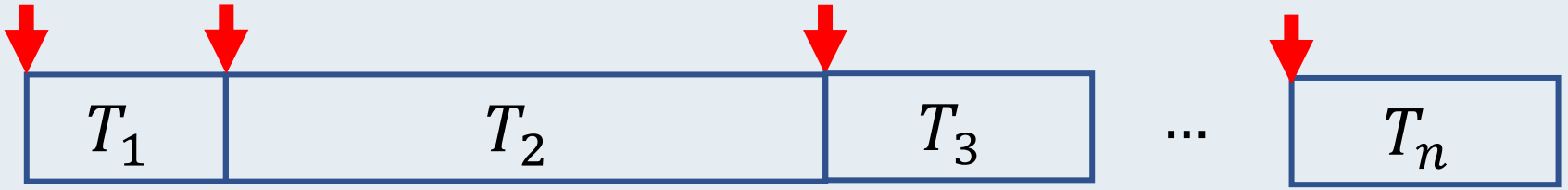
Total waiting time $0 + 3 = 3$ minutes.



Total waiting time $0 + 1 = 1$ minute.

Observation: put shorter time in front!

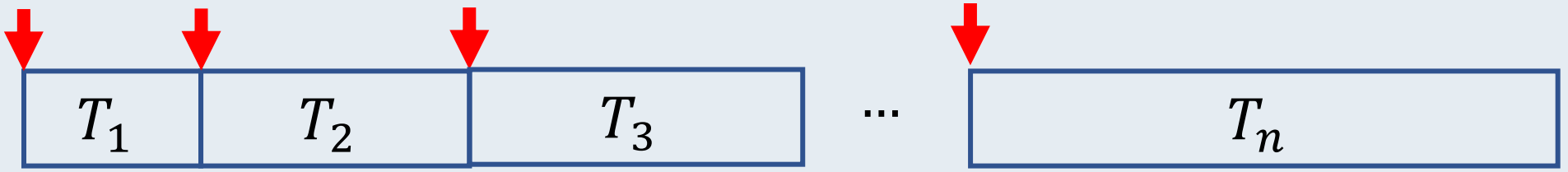
Problem 3



Total waiting time:

$$0 + T_1 + (T_1 + T_2) + \cdots + (T_1 + T_2 + \cdots + T_{n-1})$$
$$= (n-1)T_1 + (n-2)T_2 + \cdots + 2T_{n-2} + T_{n-1}$$

Problem 3



Total waiting time:

$$0 + T_1 + (T_1 + T_2) + \cdots + (T_1 + T_2 + \cdots + T_{n-1})$$
$$= (n-1)T_1 + (n-2)T_2 + \cdots + 2T_{n-2} + T_{n-1}$$

Idea: Rank the waiting time in ascending order!

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: $total \leftarrow 0$
4: **for** $i \leftarrow 1$ to $n - 1$ **do**
5: $\quad total \leftarrow total + (n - i) \times T[i]$
6: **end for**
7: **output** $total$

- Sorting: $O(n \log n)$
- Calculating total time: $O(n)$

Pause and Ponder 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: $total \leftarrow 0$
4: **for** $i \leftarrow 1$ to $n - 1$ **do**
5: $\quad total \leftarrow total + (n - i) \times T[i]$
6: **end for**
7: **output** $total$

Question: How do we prove that this ordering is the best?

** Find the answer in the appendix!*

Problem 4

- n family members, standing in random ordering.

Problem 4

- n family members, standing in random ordering.
- Some members are removed.

Problem 4

- Left with a **subsequence** S, where we only know the relative order of the remaining numbers.

Note. Contrary to **substring** where we require the remaining numbers to be consecutive.

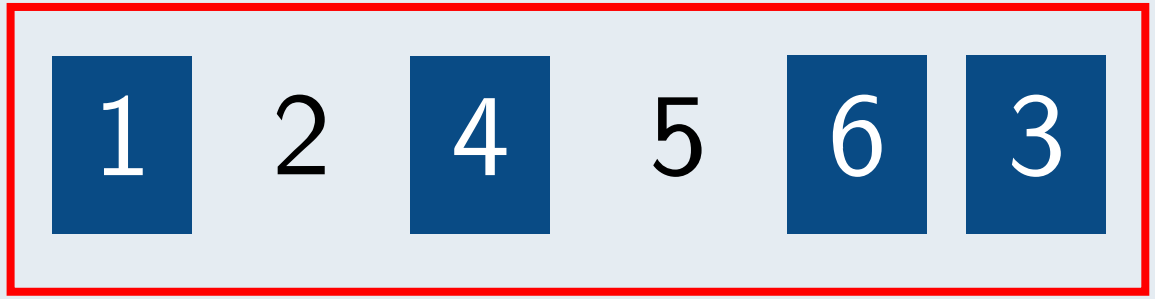
S: 1 4 6 3

Problem 4

- Find the first permutation that contains S.

$O(n!)$

Permuta-
tions!

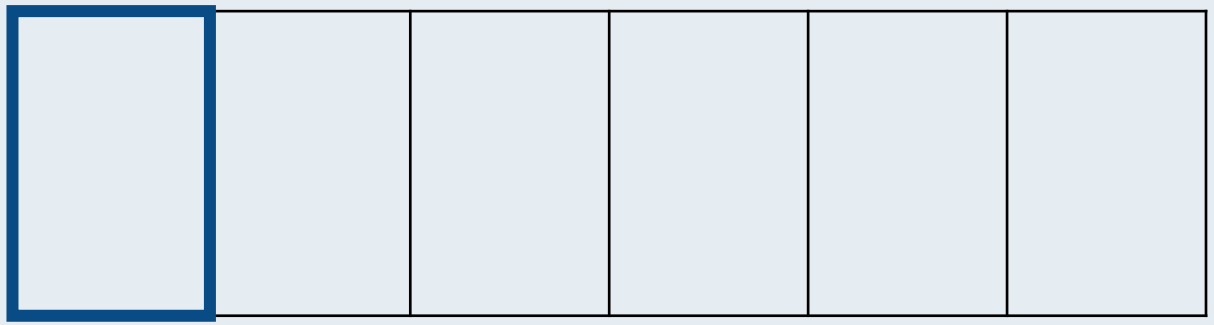


Problem 4

Goal:

1. Let number in the front be as small as possible.
2. Preserve the relative ordering in the subsequence.

Problem 4



Subsequence:

1 4 6 3

Put anywhere we want:

2 5

Problem 4

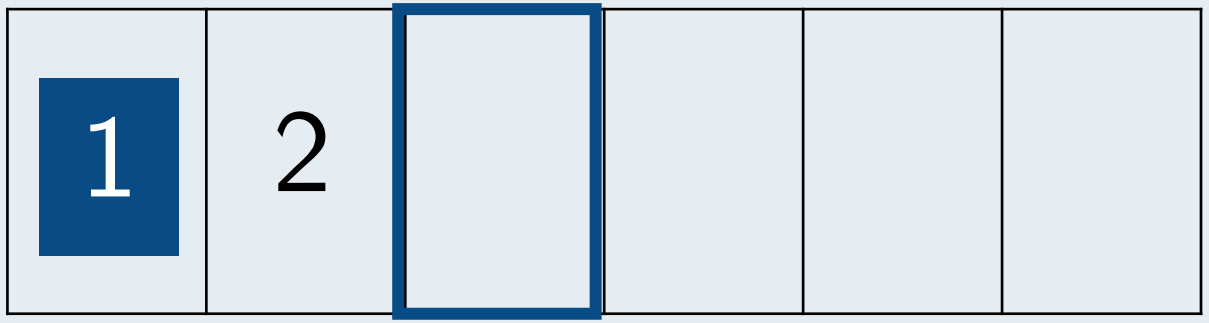
Subsequence:

4 6 3

Put anywhere we want:

2 5

Problem 4

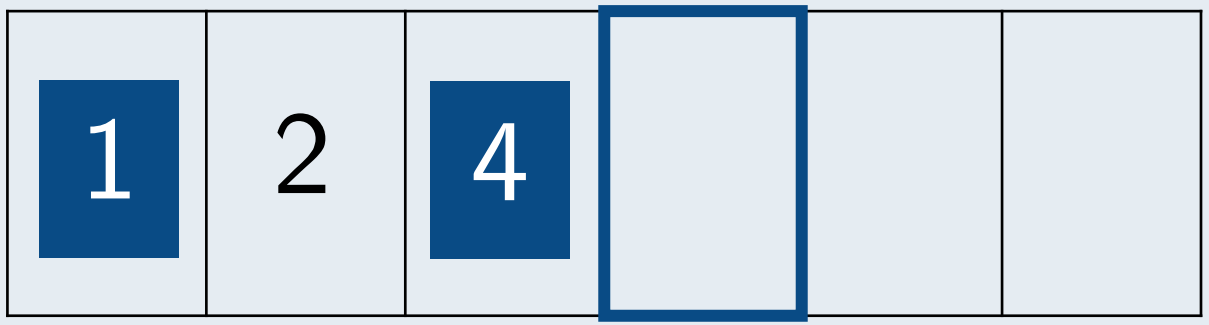


Subsequence:

Put anywhere we want:

5

Problem 4



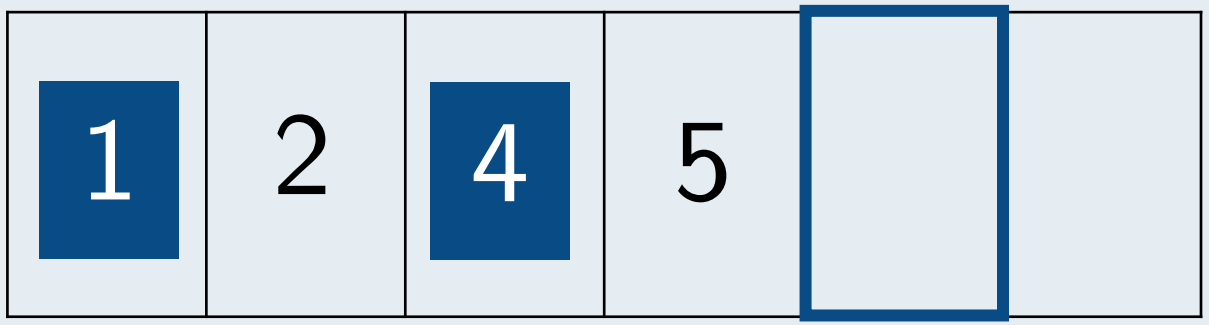
Subsequence:

6 3

Put anywhere we want:

5

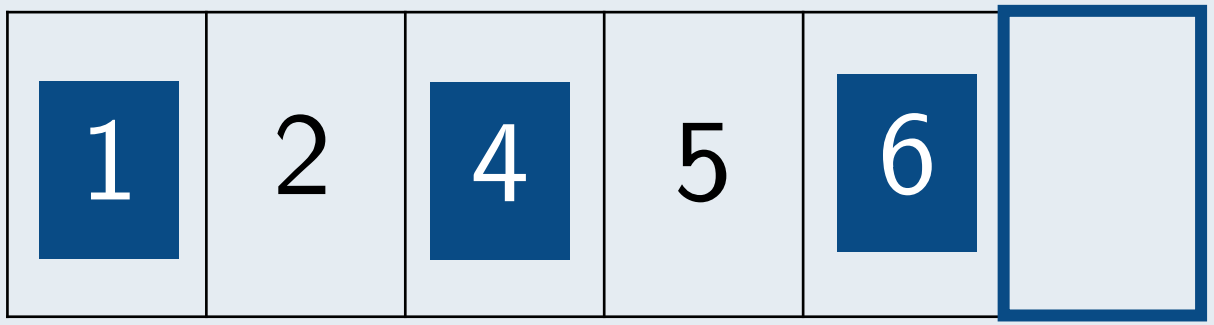
Problem 4



Subsequence:

Put anywhere we want:

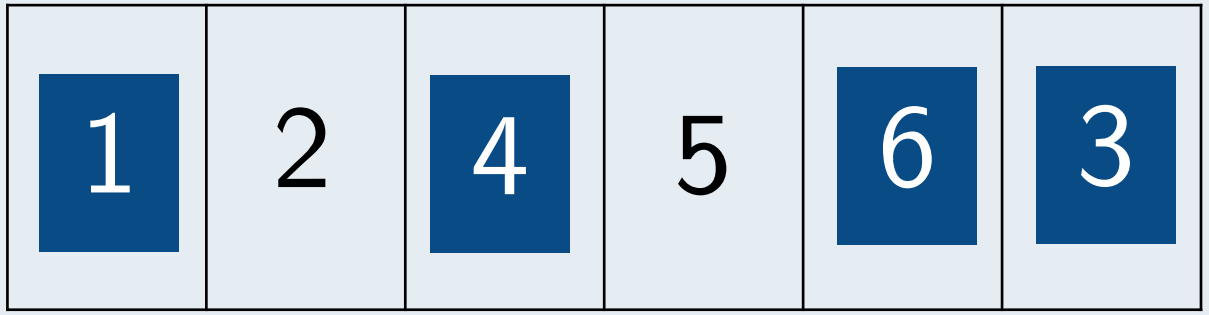
Problem 4



Subsequence:

Put anywhere we want:

Problem 4



This is basically the *merge* method from 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'          // S' is the missing subsequence
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)
```

In total we need $O(n)$ time.

Appendix

Pause and Ponder 1

Question. What is the time complexity of radix sort if largest item is $O(n)$?

Going to be $O(n \log n)$! The original time complexity is $O(dn)$ where d is the number of digits. Now $d = \log_{10} n$.

Pause and Ponder 2

Question. Is it as good to also use Bubble Sort 2 on almost sorted array?

Sometimes it can fail miserably! For example, the almost sorted array

2, 3, 4, 5, 6, 7, 8, 1

will take $O(n^2)$ time!

Pause and Ponder 3 *

Question. How do we prove that ascending ordering is best for Problem 3?

(For CS3230) Use contradiction!

- Suppose ascending order isn't the best.
- The best one is not ascending: so at least a pair of patients a, b with $t_a < t_b$ but a visits doctor after b.
- But exchanging a and b will reduce the total waiting time by at least $t_b - t_a$.
- So this one is still not the best! (contradiction)

End of File

Thank you very much for your attention :-)