Tutorial 02 - Sorting (part 2) ADT

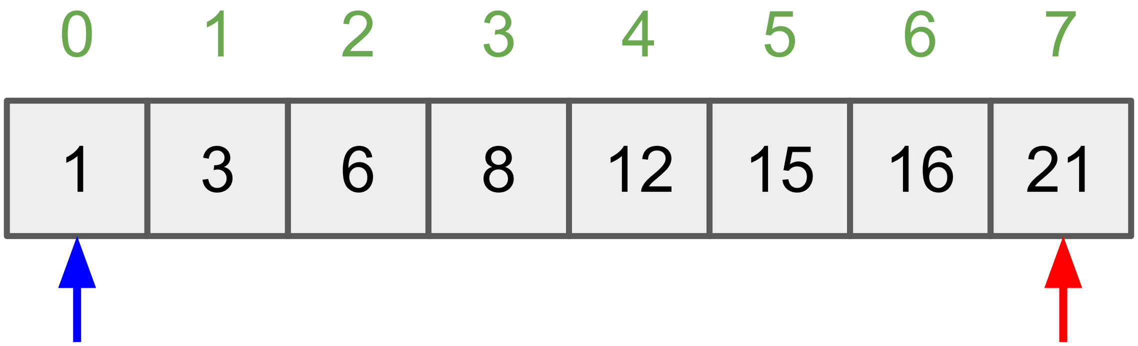
CS2040C Semester 2 2018/2019

Question 1: Applications

Application 1: Binary search

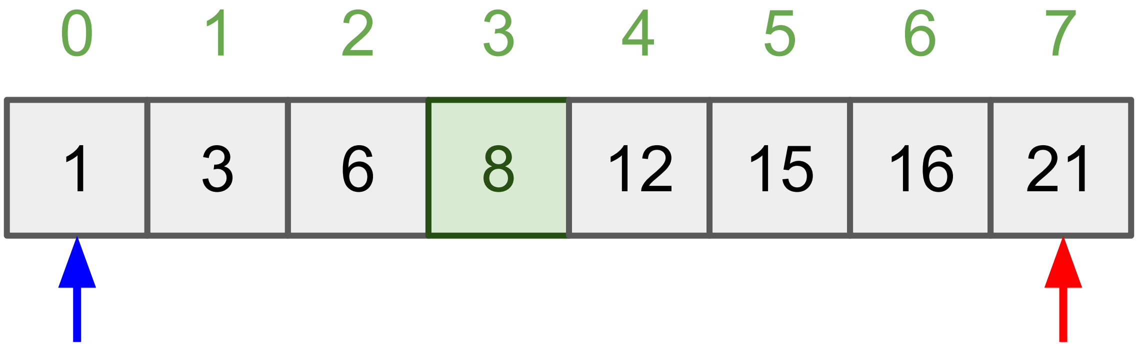
Searching for a specific value v in array A

Say for instance, we want to find if this array contain the target value 6



We start with two pointers, low and high

Application 1: Binary search



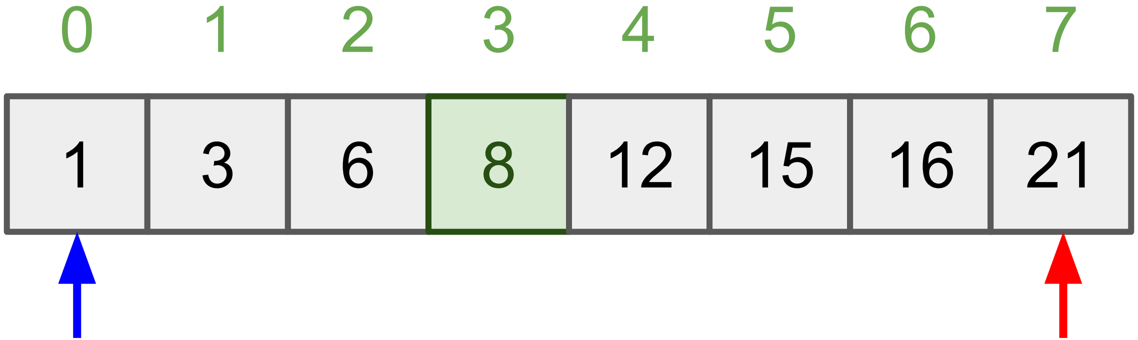
We check for the middle item bounded by the two pointers

If middle is the value we are looking for, we are done

Else if middle is greater, than the value can only be on its RHS

Else the value can only be on its LHS

Application 1: Binary search

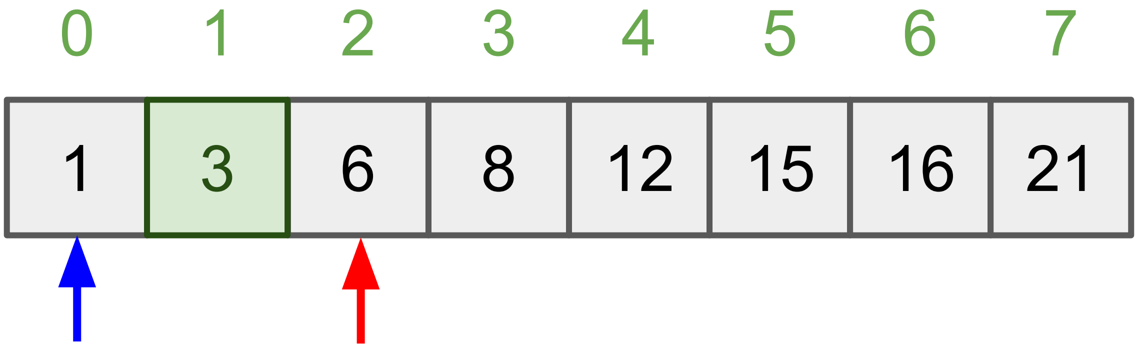


Here we have middle item at $index = \lfloor (0+7) / 2 \rfloor = 3$

Middle has value 8 so target value 6 can only be on its LHS in the array!

Hence we will update high to index 2 (immediately left of middle)

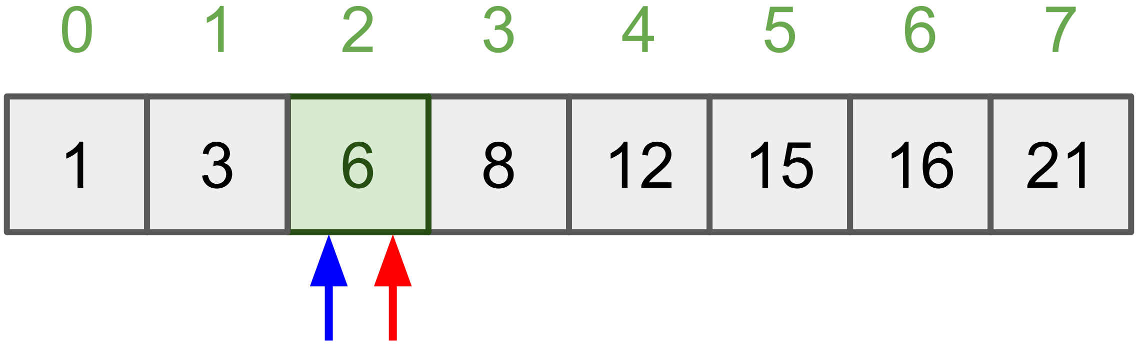
Application 1: Binary search



Middle now has value 3 so target value 6 can only be on its RHS in the array!

We will update low to index 2

Application 1: Binary search



Middle now has value 6, we are done!

Test your understanding!

- If the two pointers have collapsed to the same index (like above) but we still haven't found the target value, what does this mean?
- What is the worst case?
- What is the time complexity?

Application 2: Order statistic

Find min, max, k^{th} smallest/largest item

In a sorted array of size N,

- Min is at index `0`
- Max is at index `N - 1`
- k^{th} (1-based) smallest is at index `k - 1`
- k^{th} (1-based) largest is at index `N - k`

Application 3: Duplicates

Testing for uniqueness and deleting duplicates in array

Approach: Iterate down the array with adjacent pair pointers, if their respective items has the same value then they are duplicates

Application 4: Counting repetitions

Count how many times a specific value `v` appear in array **A**

Approach 1 (binary search)

1. Find the position `i` of `v` using binary search (application 1)
2. Search left from `i` to find lower bound `l` (i.e. first occurrence is `A[l] == v`)
3. Search right from `i` to find upper bound `u` (i.e. last occurrence is `A[u] == v`)
4. Return answer `u - l + 1`

Complexity: $O(\log N + f)$ where f is the number of occurrences

Approach 2 (modified binary search)

We can also modify binary search to directly find lower bound `l` and upper bound `u` respectively. *How?*

So,

1. Find lower bound `l` using modified binary search
2. Find upper bound `u` using modified binary search
3. Return `u - l + 1`

Complexity: $O(2 \log N) = O(\log N)$.

Approach 3 (Counting sort)

It is also possible to populate the count table using the counting sort subroutine

However that requires $O(N)$ scan and potentially a big $O(k)$ memory (not feasible if the range of the values is big)

Application 5: Set operations

Set intersection/union between two sorted arrays **A** (size m) and **B** (size n). We define **A** to be the smaller set (i.e. $m < n$)

For now let's deal with the simpler case that there are only distinct numbers in set. That is to say, they are not *multisets* where duplicated values are permitted

Brute force approach (linear search)

For intersection

- For each number x in array **A**, loop through array **B**
  - if x exists in array **B**, append to output array **C**

For union

- Copy **A** to **C**
- For each number x in array **B**, loop through array **A**
  - If x does not exists in **A**, append to output array **C**

Complexity: $O(mn)$

A better approach (binary search)

Property**:** Arrays **A** and **B** are sorted!

Instead of using linear search for x, we can use binary search

Complexity

- Set intersection: $O(m \log n)$
- Set union: $O(n + m \log n)$
- Note: Choosing which array to binary search over affects performance! The complexities above describe the most optimal choice since we have stated that $\boldsymbol{m < n}$

An even better approach (2 pointers method)!

Properties:

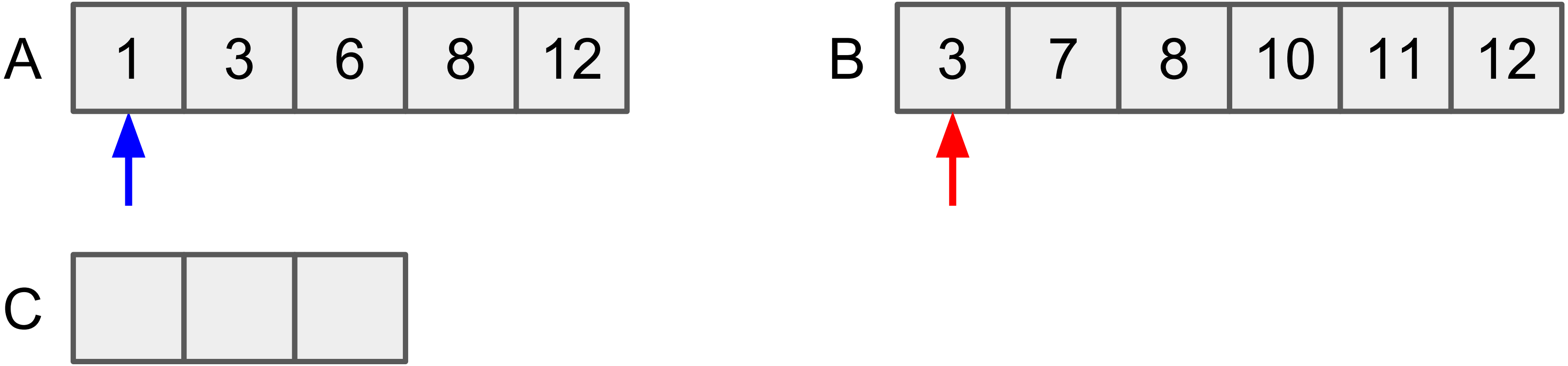
- x_A in **A** is *non-decreasing* because **A** is sorted
- x_B in **B** is *non-decreasing* because **B** is sorted

So,

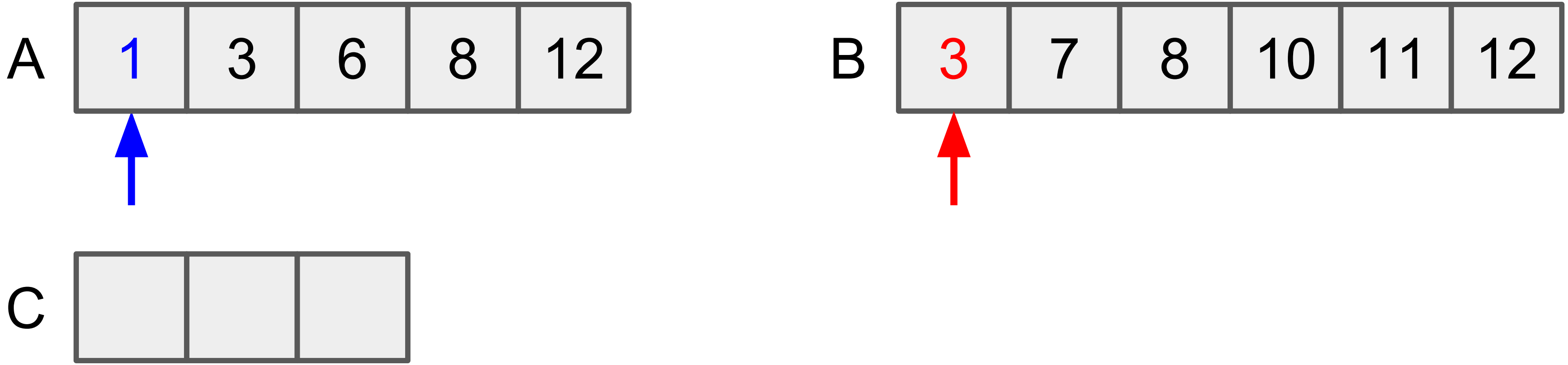
- When searching for x_A in **B**, we can halt once $x_B > x_A$
- For the next value of x_A we can continue from where we left off previously in **B** at x_B

An even better approach (merge subroutine)!

Let's illustrate using the example of finding the intersection between the following arrays **A** and **B**. **C** is the output array

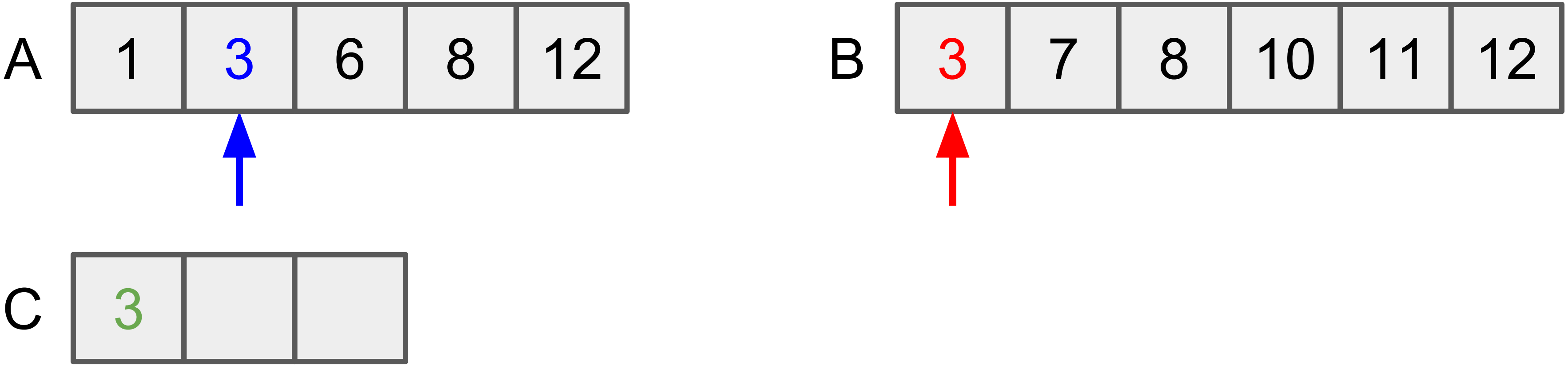


An even better approach (merge subroutine)!



3 is already greater than 1 so **B** cannot contain 1
So we will look at the next x_A

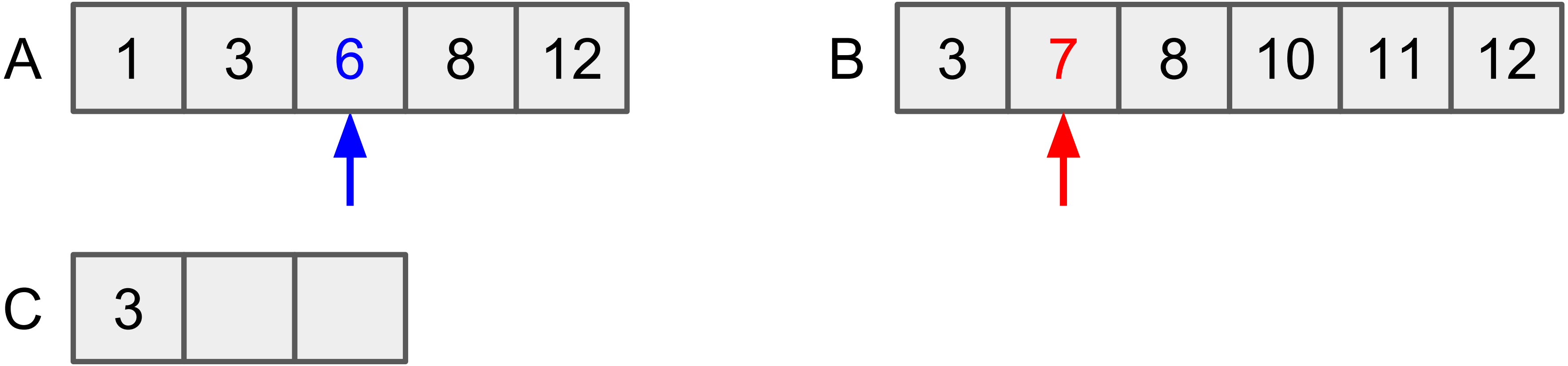
An even better approach (merge subroutine)!



3 is equal to 3 so we append to **C**

We will look at the next x_A and x_B

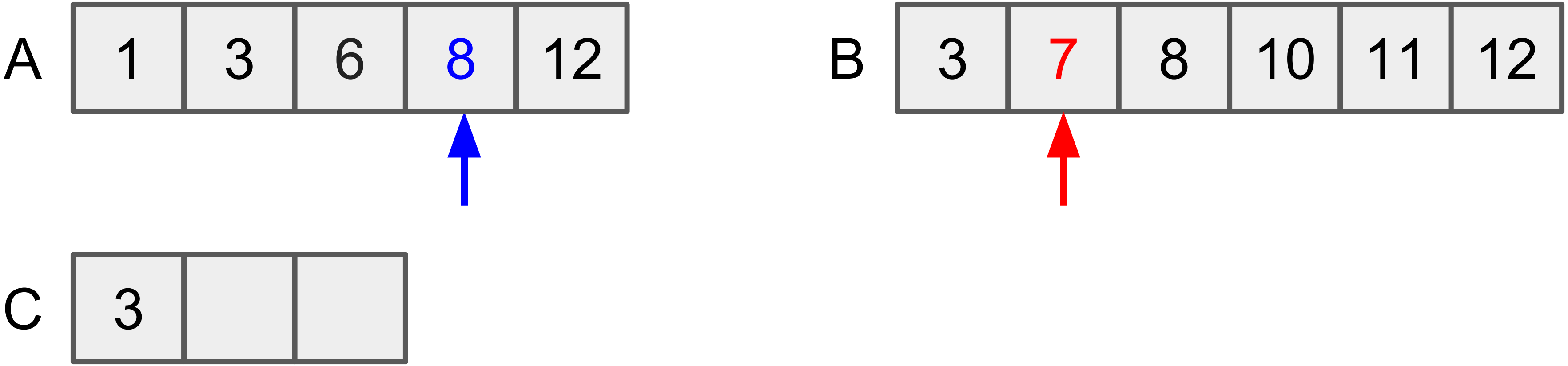
An even better approach (merge subroutine)!



7 is already greater than 6 is so 6 cannot be in **B**

So we will look at the next x_A

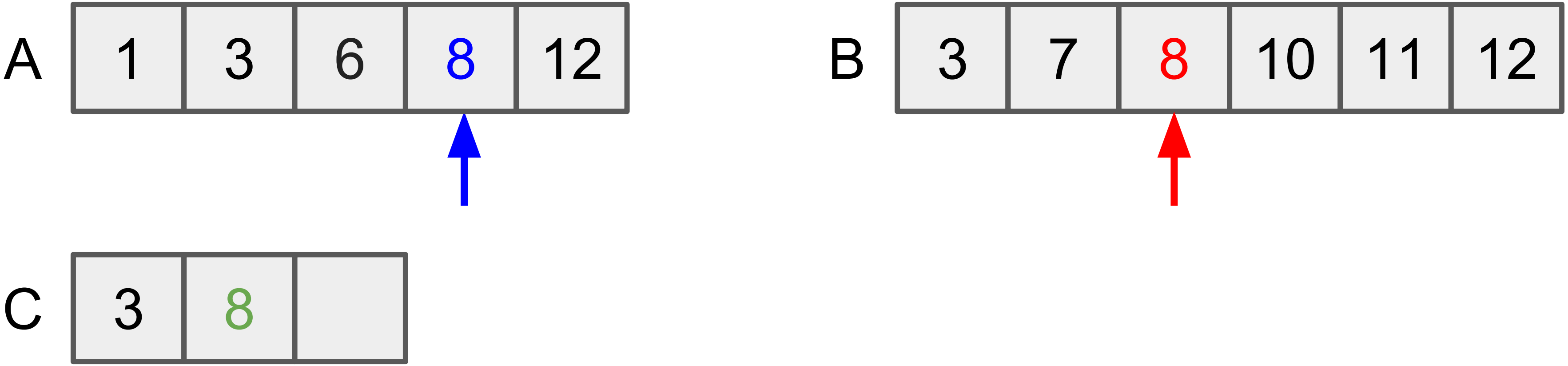
An even better approach (merge subroutine)!



7 is lesser than 8

So we will look at the next x_B

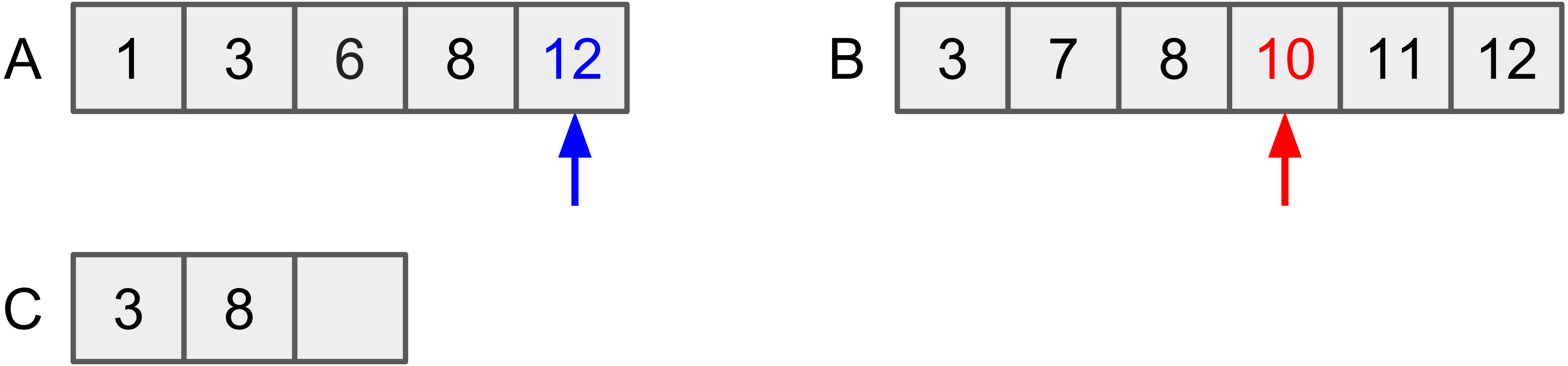
An even better approach (merge subroutine)!



8 is equal to 8 so we append to **C**

We will look at the next x_A and x_B

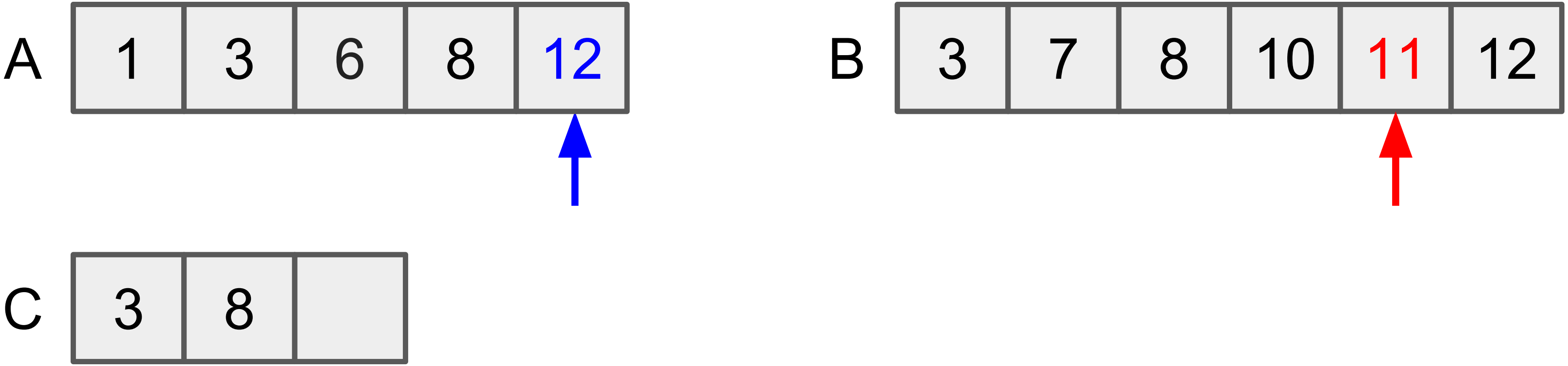
An even better approach (merge subroutine)!



10 is lesser than 12

So we will look at the next x_B

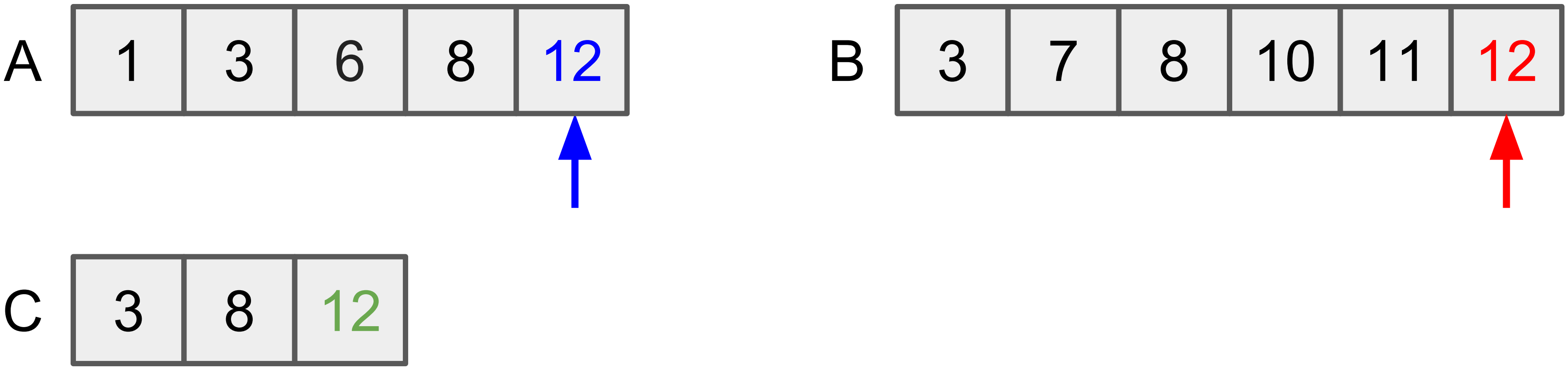
An even better approach (merge subroutine)!



11 is still lesser than 12

So we will look at the next x_B

An even better approach (merge subroutine)!



12 is equal to 12 so we append to **C**

We have covered every item in **A** so we are done!

An even better approach (merge subroutine)!

This is essentially the same principle as the merge subroutine of merge sort!

Complexity: $O(m + n)$

Will be demonstrated in lab 2.

Application 6: Target pair (AKA Two sum)

- Find a target pair x and y such that $x + y$ equals to a target z, etc. (in the same array)
- Popular programming interview question!

Brute force approach (linear search)

For each number x in array, loop through every other item y in the array to check if $x + y = z$.

Complexity: $O(\boldsymbol{n}^2)$

A better approach (binary search)

Property: Array is sorted!

Instead of using linear search, we could use binary search.
i.e. For every x, search for $z - x$ on RHS using binary search

Complexity: $O(n \log n)$

An even better approach!

We use two pointers: one *low-pointer* and one *high-pointer*. Respectively, they are initialized to point to the first and last item of the array.

Let's see an actual example with $z = 18$ for the following array

An even better approach!

1	3	6	8	12	15	16	21

$$1 + 21 = 22 > 18$$

It’s too large!

So we have to retreat high-pointer

An even better approach!

1	3	6	8	12	15	16	21

$$1 + 16 = 17 < 18$$

It’s too small!

So we have to advance low-pointer

An even better approach!

$$3 + 16 = 19 > 18$$

It’s too large!

So we have to retreat high-pointer

An even better approach!

$$3 + 15 = 18$$

Found it!

$$x = 3,\ y = 15$$

Notice there exists another pair which also add up to 18!

Challenge yourself!

- How shall we continue the process if we want to output all the possible pairs that sum up to 18?
- Can we generalize this approach for finding n-tuples which add up to the target sum? *Hint: can you come up with a recurrence relation?*

Question 2: Mini Experiment

Input order →	Random	Sorted		Nearly Sorted		Homogeneous
Algorithm ↓		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best				
(Min) Selection Sort					$O(N^2)$	
Insertion Sort			$O(N^2)$			
Merge Sort				$O(N \log N)$		
Quick Sort		$O(N^2)$				
(Rand) Quick Sort	$O(N \log N)$					
Counting Sort					$O(N)$	
Radix Sort		$O(N)$				

Input order →	Random	Sorted		Nearly Sorted		Homogeneous
Algorithm ↓		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best				
(Min) Selection Sort					$O(N^2)$	
Insertion Sort			$O(N^2)$			
Merge Sort				$O(N \log N)$		
Quick Sort		$O(N^2)$				
(Rand) Quick Sort	$O(N \log N)$					
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Counting sort and radix sort are *non-comparative* sorts
Therefore they are not affected by the input order!

Input order →	Random	Sorted		Nearly Sorted		Homogeneous
Algorithm ↓		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best				
(Min) Selection Sort	$O(N^2)$				$O(N^2)$	
Insertion Sort	$O(N^2)$		$O(N^2)$			
Merge Sort	$O(N \log N)$			$O(N \log N)$		
Quick Sort	$O(N \log N)$	$O(N^2)$				
(Rand) Quick Sort	$O(N \log N)$					
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

You have learnt that these are the time complexities of comparative algorithms when they are applied on random input

Input order → Algorithm ↓	Random	Sorted		Nearly Sorted		Homogeneous
		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	
(Min) Selection Sort	$O(N^2)$				$O(N^2)$	
Insertion Sort	$O(N^2)$		$O(N^2)$			
Merge Sort	$O(N \log N)$			$O(N \log N)$		
Quick Sort	$O(N \log N)$	$O(N^2)$				
(Rand) Quick Sort	$O(N \log N)$					
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Optimized bubble sort terminates the moment a full pass of the array succeeds without any swaps. You should convince yourself that sorted descending and nearly sorted descending are indeed the worst case and must therefore be $O(N^2)$. What about nearly sorted ascending? Consider `{1,2,3,...,100000,0}`. It’s a also worst case!

Input order → Algorithm ↓	Random	Sorted		Nearly Sorted		Homogeneous
		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	
(Min) Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N^2)$		$O(N^2)$			
Merge Sort	$O(N \log N)$			$O(N \log N)$		
Quick Sort	$O(N \log N)$	$O(N^2)$				
(Rand) Quick Sort	$O(N \log N)$					
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Recall that selection sort iteratively traverse the entire unsorted region to *select* (therefore the name!) the minimum item to append to the sorted region. The first pass selects the 1st smallest item, the second pass selects the 2nd smallest item and so on. Therefore **regardless of input order**, it will take $O(N-1 + N-2 + \dots\ 1) = O(N^2)$ time

Input order → Algorithm ↓	Random	Sorted		Nearly Sorted		Homogeneous
		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	
(Min) Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N)$	$O(N^2)$	
Merge Sort	$O(N \log N)$			$O(N \log N)$		
Quick Sort	$O(N \log N)$	$O(N^2)$				
(Rand) Quick Sort	$O(N \log N)$					
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Recall insertion sort: for every item in the unsorted region, their correct placement in the sorted region is determined and inserted to that spot. Sorted ascending is the best case since all items in unsorted region is already in their correct placements. Sorted and nearly sorted descending will clearly incur the worst case. For nearly sorted ascending, consider `{1,2,3, ...,100000,0}`. $N-1$ comparisons to update sorted region until index $N-2$, N time to insert 0 at first index. So it is $O(N)$

Input order →	Random	Sorted		Nearly Sorted		Homogeneous
Algorithm ↓		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N)$
(Min) Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N)$
Merge Sort	$O(N \log N)$			$O(N \log N)$		
Quick Sort	$O(N \log N)$	$O(N^2)$				
(Rand) Quick Sort	$O(N \log N)$					
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Realise that a homogeneous array is both sorted ascending and descending at the same time! In VisualAlgo's implementation of optimised bubble sort, we will only swap if left item is **strictly greater** than right item and so a homogeneous array will experience a single pass without any swaps. In VisualAlgo's implementation of insertion sort, again we will only shift sorted region items if they are **strictly greater** than the one to be inserted. Thus it is also the best case for insertion sort.

Input order → Algorithm ↓	Random	Sorted		Nearly Sorted		Homogeneous
		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N)$
(Min) Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N)$
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Quick Sort	$O(N \log N)$	$O(N^2)$				
(Rand) Quick Sort	$O(N \log N)$					
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Merge sort's divide and conquer strategy is agnostic to the input order and therefore will always be $O(N \log N)$

Input order → Algorithm ↓	Random	Sorted Ascending	Sorted Descending	Nearly Sorted Ascending	Nearly Sorted Descending	Homogeneous
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N)$
(Min) Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N)$
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Quick Sort	$O(N \log N)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
(Rand) Quick Sort	$O(N \log N)$					
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Naive quick sort will fall into worst case behavior when pivot subroutine reduces the problem size by 1 most of the time. This will happen so long as the input is mostly ordered.

Input order → Algorithm ↓	Random	Sorted		Nearly Sorted		Homogeneous
		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N)$
(Min) Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N)$
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Quick Sort	$O(N \log N)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
(Rand) Quick Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N^2)$*, $O(N)$
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Randomized quick sort overcomes all the limitations of naive quick sort when faced with ordered and semi-ordered input. Note that VisualAlgo's implementation of randomized quick sort as per 29 Jan 2019 is flawed for homogenous input. This can be easily fixed to achieve $O(N)$ best case.

Input order → Algorithm ↓	Random	Sorted		Nearly Sorted		Homogeneous
		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N)$
(Min) Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N)$
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Quick Sort	$O(N \log N)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
(Rand) Quick Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N^2)$*, $O(N)$
Counting Sort†	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort‡	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

* Flawed implementation of randomized quick sort on visualgo as per 29th Jan 2019
† More precisely $O(w(N + k))$ where w is the number of digits position and k is the radix/base (i.e. 10 for decimal)
‡ More precisely $O(N + k)$ where k is the maximum value in input

Non-comparative sorting

If counting/radix sort has a “better” time complexity, why don’t we always just use these sorts?

What are the special constraints of these non-comparison based sorts?

Are their time complexities purely a function of input size N?

HINT HINT

Non-comparative sorting

- Realize that we cannot generalize these algorithms to work on non-integers. You might be able to come up with ways to deal with negative numbers and floats but what if we want to sort datetime strings or ADT objects?
- Realize also that it is not possible to mount a custom comparator because these algorithms are not comparison based in the first place!

Non-comparative sorting

Precise complexities

- Radix sort runs in $O(w(N + k))$ time and requires $O(N + k)$ space, where w is the number of digit positions and k is the base/radix for each digit (i.e. 10 for decimal numbers)
- Counting sort runs in $O(N + k)$ time and requires $O(k)$ space, where k is the size of value range in the array

Constant time differences

Notice the different constant time terms. i.e. non-N terms in the precise time complexity. Depending on the input, these constant terms *may* bear significant influence on the time complexity

In the real world, *benchmarking* and understanding your data is important.

May the sort be with you

Among Merge Sort, Counting Sort and Radix Sort, choose the most appropriate sorting algorithm for the following arrays which contains N numbers, with each number between 0 and K inclusive.

1. $N = 10^7, K = 31$ (Days of the month)
2. $N = 10^{15}, K = 10^{12}$ (Big data, *memory issue*?)
3. $N = 10^6, K = 10^{21}$ (Generic)

May the sort be with you

	Merge sort	Counting sort	Radix sort
$N = 10^7, K = 31$	$O(10^7 \log_2 10^7)$	$O(10^7 + 31)$	$O(2(10^7 + 10))$
$N = 10^{15}, K = 10^{12}$	$O(10^{15} \log_2 10^{15})$	$O(10^{15} + 10^{12})$	$O(12(10^{15} + 10))$
$N = 10^6, K = 10^{21}$	$O(10^6 \log_2 10^6)$	$O(10^6 + 10^{21})$	$O(21(10^6 + 10))$

Learning outcome: The choice of sorting algorithm to use is problem specific! There’s no one-size-fits-all!

Challenge

Find out what sorting algorithm C++ STL `sort` uses now.

What sorting algorithm does C++ STL `stable_sort` use?

What about other languages?

Java, Python?

Why do you think they implemented it this way?

Question 3: Quick Select

Popular programming interview question!

Quick select

Find the k^{th} smallest element in an **unsorted** array (Selection Algorithm)

C++ std library API:
http://en.cppreference.com/w/cpp/algorithm/nth_element

Algorithm outline

1. Randomly pick a number as **pivot**
2. Compute its `rank`
    a. If `k == rank`, we are done
    b. If `k < rank`, our target is to the left
    c. If `k > rank`, our target is to the right
3. Repeat steps 1-2, limiting pivot within the new subrange we are searching in

Expected time complexity: ***O(N)*** [Explanation in CS3230]

Test yourself!

What is the best case for this algorithm for the following choices of pivot? i.e. If we ‘luckily’ selected the answer on the first try.

- Non-randomized
- Randomized

Test yourself!

What is the best case for this algorithm for the following choices of pivot? i.e. If we ‘luckily’ selected the answer on the first try.

- Non-randomized
- Randomized

Both $O(N)$

Test yourself!

What happens if we do not randomize the pivot of partition?

Hint: What is the pitfall of quick sort that doesn’t use randomized pivots?

Test yourself!

What happens if we do not randomize the pivot of partition?

Easy to hit near-worse case behaviour!

Unless the array itself is randomized, in which case a non-randomized pivot choice will behave like a random pivot

Question 4: ADT

Introduction to ADT
List ADT

Abstract Data Type (ADT)

An *abstract* data type that is defined by the **operations** you can perform on it.

Implementations of the same ADT can vary!

Each have their own pros and cons and so its very problem/task dependent as to which implementation is 'better'

Abstract Data Types (ADT)

Since ADTs are defined by operations:

- Implementation can be changed without affecting **functionality** of existing code
  - STL Libraries
- Usually implemented in OOP fashion
  - **Encapsulation**

Common List Array ADT operations

`get(i)`	Gets the `i`-th element from the front. (0-indexed)
`search(v)`	Return the first index which contains `v`, or returns -1/`NULL` (to indicate failure)
`insert(i, v)`	Insert element `v` at index `i`.
`remove(i)`	Remove the element at index `i`.

PS1 Discussion

Questions?