

Tutorial 02 - Sorting, ADT

CS2040C Semester 1 2021/2022

Set real display name



<https://pollev.com/rezwanarefin430>

Agenda

1. Applications on Sorted Array
2. Sorting Mini-Experiment
3. Quick Select
4. Abstract Data Types
5. PS2 Hints
6. Hands on Exercise

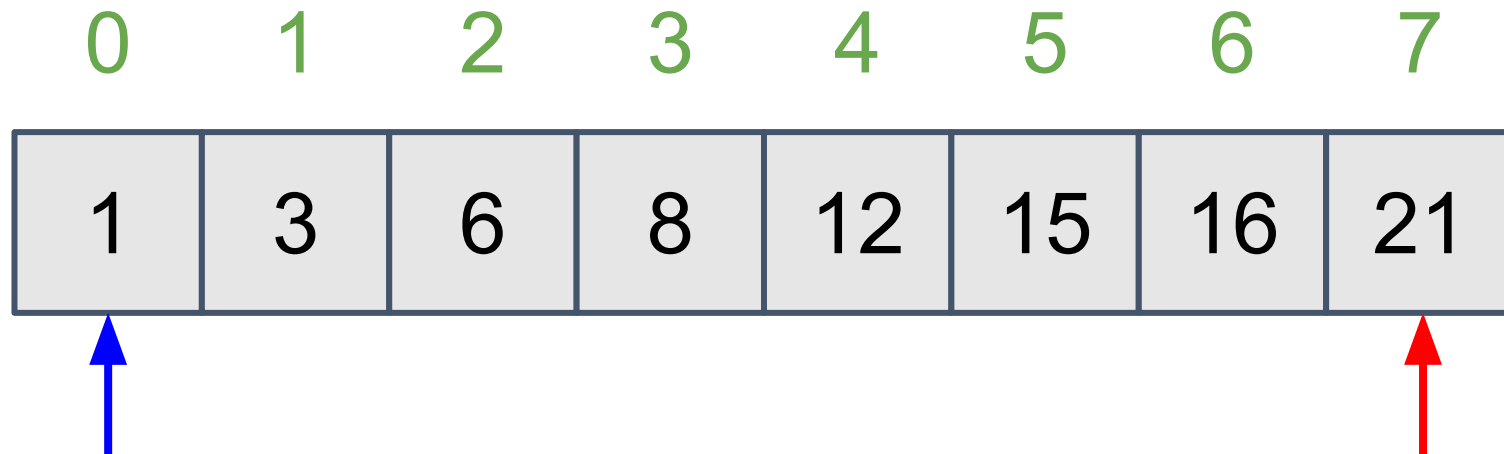
Question 1: Applications on Sorted Array

At <https://visualgo.net/en/sorting?slide=1-2>

Application 1: Binary search

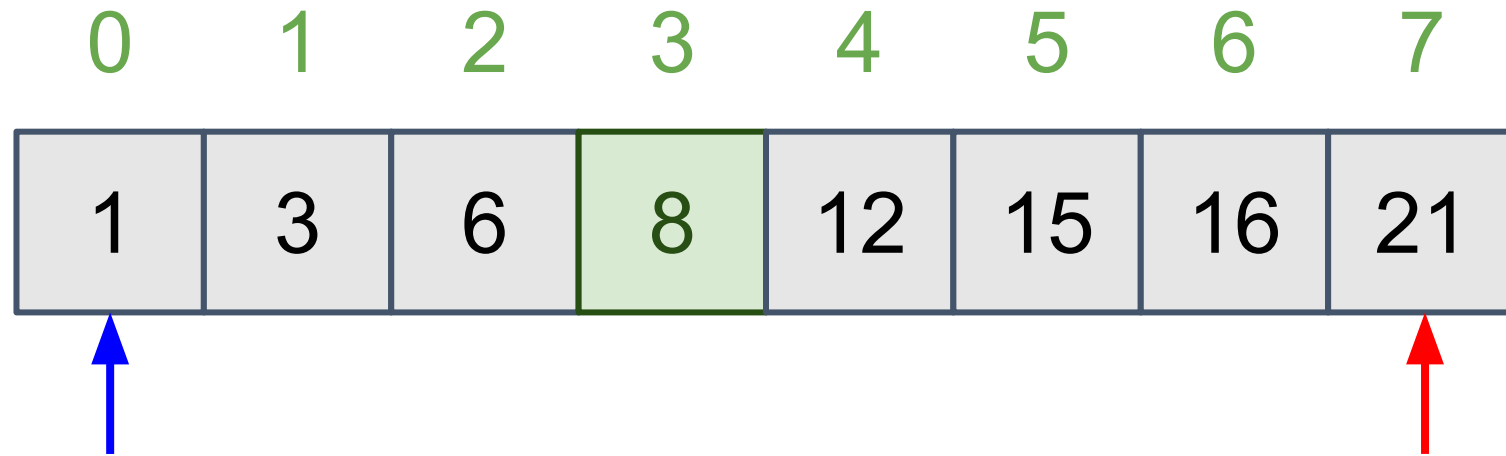
Searching for a specific value v in array A

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



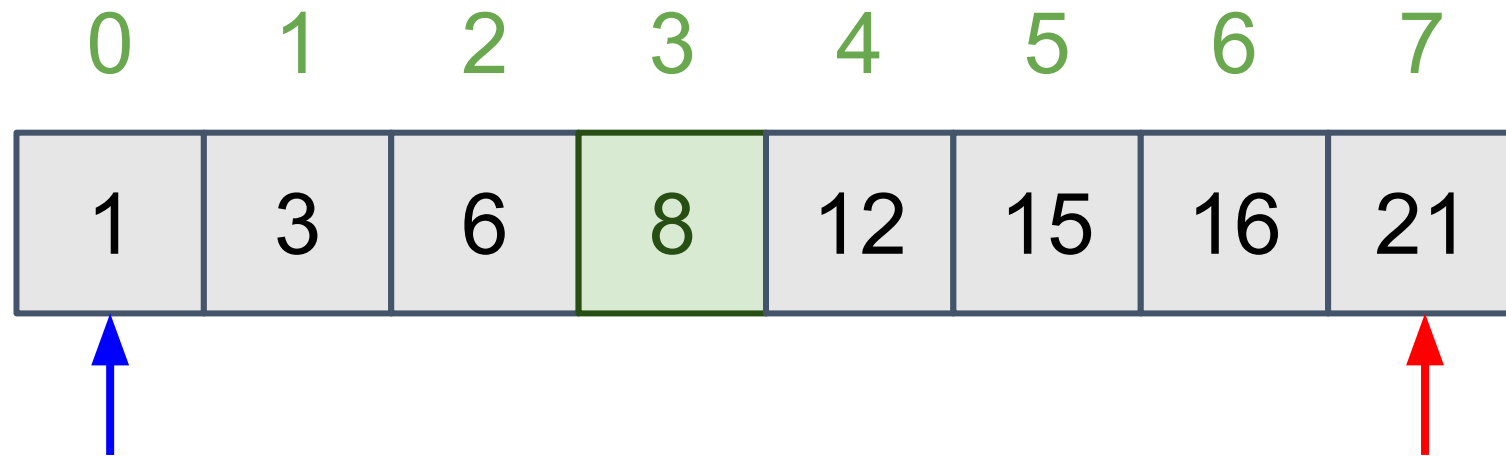
We check for the **middle** item bounded by the two pointers

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

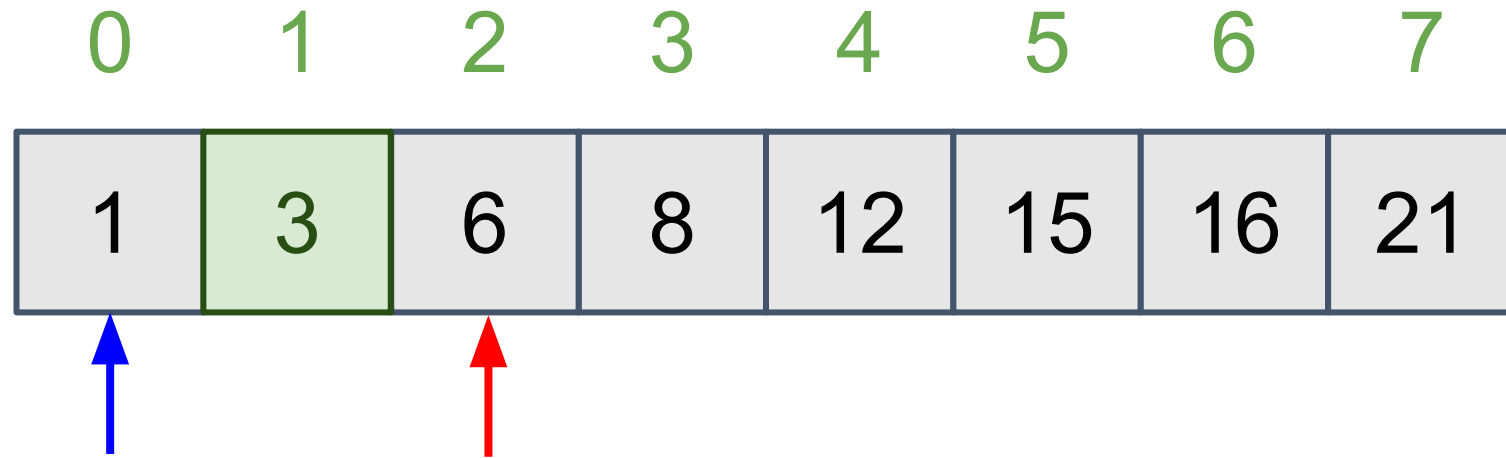


Here we have middle item at index = $(0 + 7) / 2 = 3$.

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

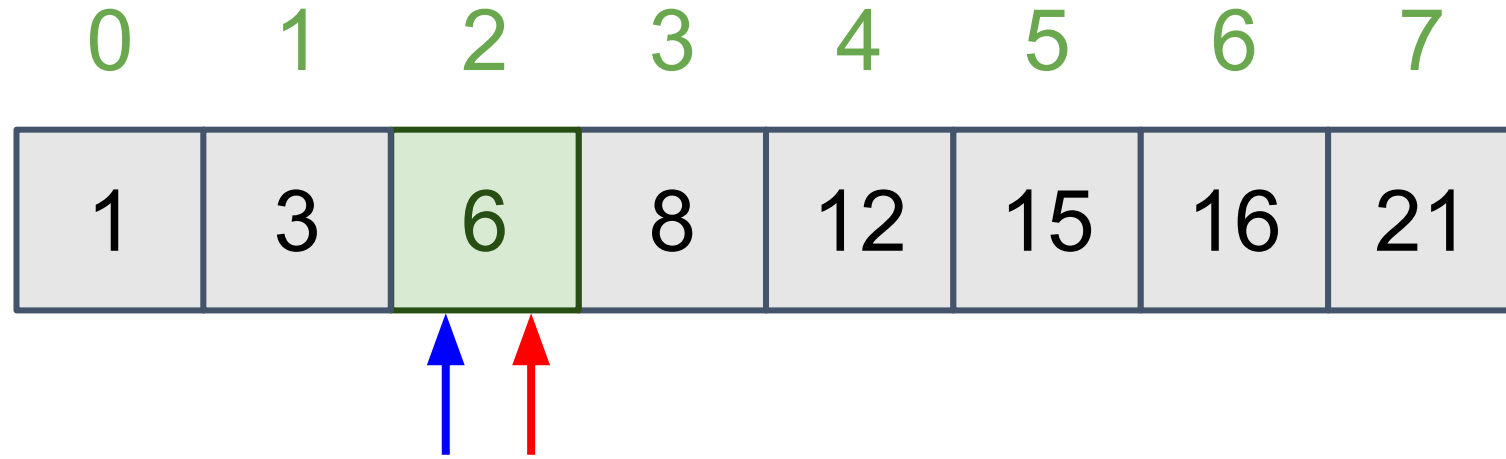


Middle now has value **3** so target value **6** can only be on its RHS in the array!

We will update **low** to index **2** (immediately right of **middle**)

Application 1: Binary search

0	1	2	3	4	5	6	7
1	3	6	8	12	15	16	21



Middle now has value 6, we are done!

Application 1: Binary search

```
public static <T extends Comparable<T>> boolean search(T[] a, T v) {  
    int lo = 0, hi = a.length - 1;  
    while (lo <= hi) {  
        int mid = (lo + hi) / 2;  
        int cmp = a[mid].compareTo(v);  
        if (cmp < 0) lo = mid + 1; // a[mid] < v  
        else if (cmp == 0) return true; // a[mid] == v  
        else hi = mid - 1; // a[mid] > v  
    }  
    return false;  
}
```

Test your understanding!

- If the two pointers have collapsed to the same index (like what we just saw) 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

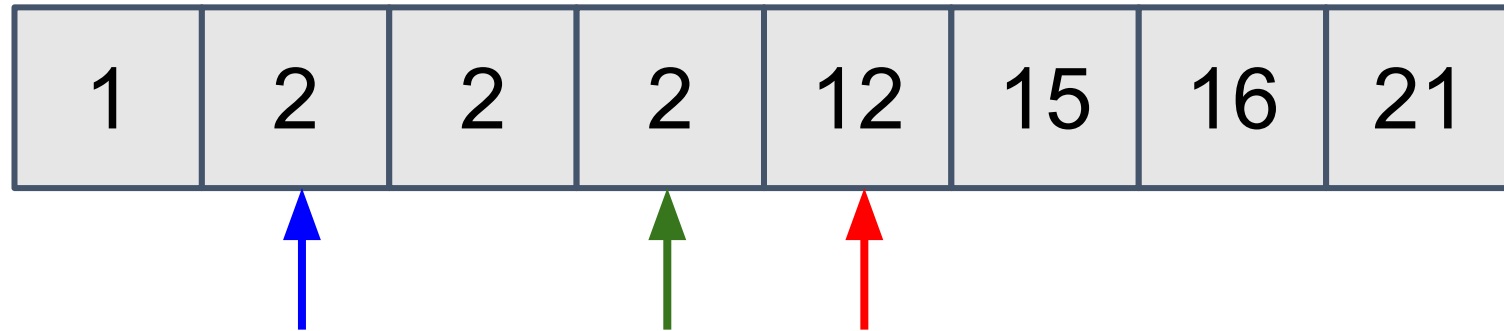
Approach: Iterate down the array with adjacent pair pointers, if their respective items has the same value then they are duplicates

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 $A[l] \geq v$).
3. Search right from i to find upper bound r (i.e. first occurrence is $A[r] > v$).
4. Return answer $r - l$.



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?

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

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



Approach 2: lower_bound

```
// lower_bound: Find lowest i, such that a[i] >= v
public static <T extends Comparable<T>> int lower_bound(T[] a, T v) {
    int lo = 0, hi = a.length - 1, ans = a.length;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        int cmp = a[mid].compareTo(v);
        if (cmp >= 0) ans = mid, hi = mid - 1;
        else lo = mid + 1;
    }
    return ans;
}
```

Why this?

Read this line like:

- We found a match for $a[i] \geq v$.
- However we would like to search left side for a better match.



Approach 2: upper_bound

How to update this code to find
upper_bound instead?

```
// upper_bound: Find lowest i, such that a[i] > v
public static <T extends Comparable<T>> int upper_bound(T[] a, T v) {
    int lo = 0, hi = a.length - 1, ans = a.length; // 1
    while (lo <= hi) {                               // 2
        int mid = (lo + hi) / 2;                     // 3
        int cmp = a[mid].compareTo(v);              // 4
        if (cmp >= 0) ans = mid, hi = mid - 1;       // 5
        else lo = mid + 1;                           // 6
    }                                                 // 7
    return ans;                                       // 8
}
```

Approach 2: upper_bound

```
// upper_bound: Find lowest i, such that a[i] > v
public static <T extends Comparable<T>> int upper_bound(T[] a, T v) {
    int lo = 0, hi = a.length - 1, ans = a.length; // 1
    while (lo <= hi) {                               // 2
        int mid = (lo + hi) / 2;                     // 3
        int cmp = a[mid].compareTo(v);               // 4
        if (cmp > 0) ans = mid, hi = mid - 1;         // 5
        else lo = mid + 1;                           // 6
    }                                                  // 7
    return ans;                                       // 8
}
```



Approach 2: upper_bound2

```
// upper_bound2: Find largest i, such that a[i] <= v
public static <T extends Comparable<T>> int upper_bound2(T[] a, T v) {
    int lo = 0, hi = a.length - 1, ans = a.length; // 1
    while (lo <= hi) {                               // 2
        int mid = (lo + hi) / 2;                     // 3
        int cmp = a[mid].compareTo(v);               // 4
        if (cmp > 0) ans = mid, hi = mid - 1;         // 5
        else lo = mid + 1;                           // 6
    }                                                  // 7
    return ans;                                       // 8
}
```

Approach 2: upper_bound2

```
// upper_bound2: Find largest i, such that a[i] <= v
public static <T extends Comparable<T>> int upper_bound2(T[] a, T v) {
    int lo = 0, hi = a.length - 1, ans = -1;           // 1
    while (lo <= hi) {                                  // 2
        int mid = (lo + hi) / 2;                        // 3
        int cmp = a[mid].compareTo(v);                  // 4
        if (cmp <= 0) ans = mid, lo = mid + 1;          // 5
        else hi = mid - 1;                              // 6
    }                                                    // 7
    return ans;                                         // 8
}
```

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 exist 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 $m < n$

An even better approach (two pointers)!

Properties:

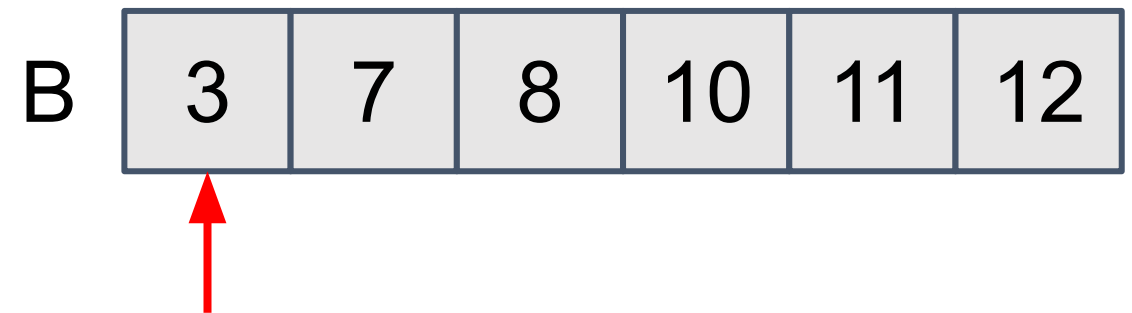
- x_A in **A** is non-decreasing because **A** is sorted
- x_B in **B** is non-decreasing because **B** is sorted

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

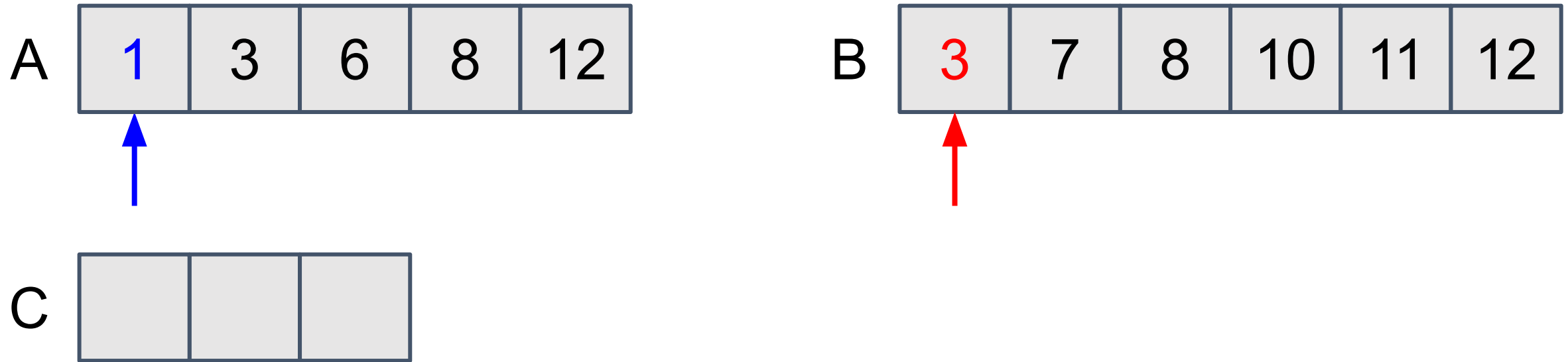
An even better approach (two pointers)!

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



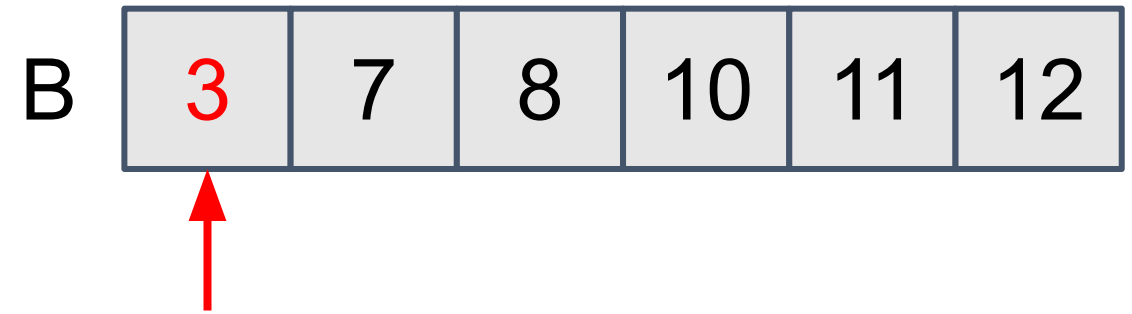
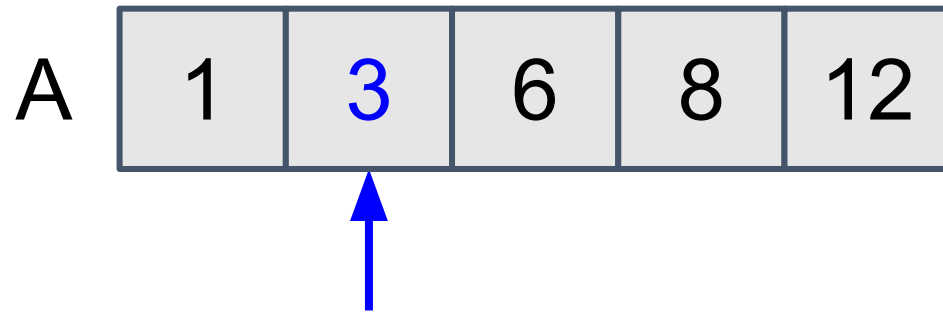
To reason about the correctness, think like the two pointers point to minimum element of the arrays.

An even better approach (two pointers)!



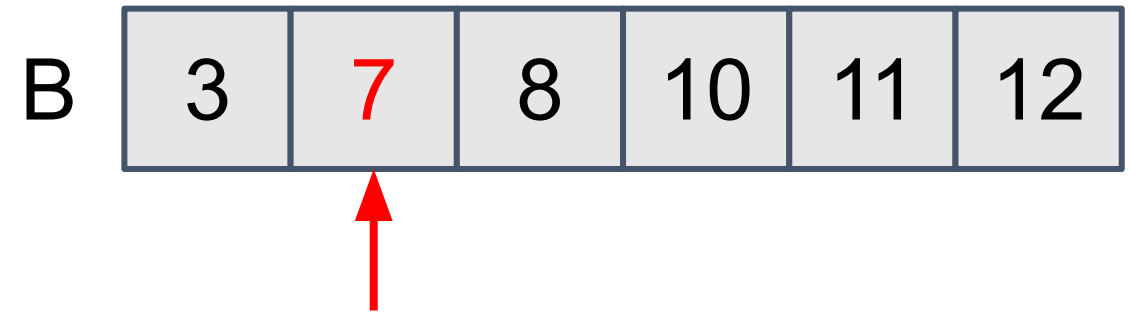
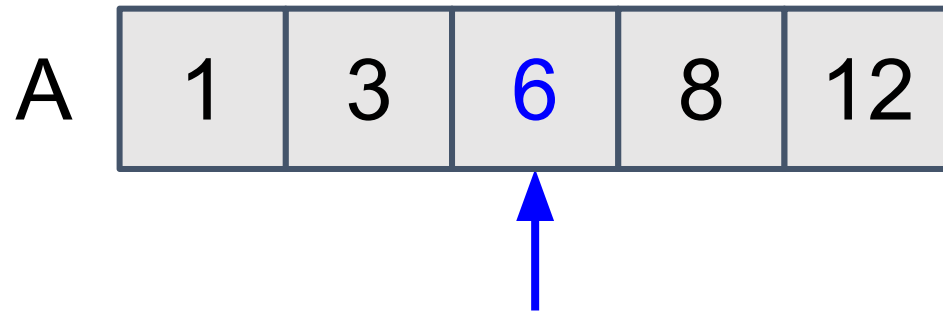
3 is already greater than **1** so **B** cannot contain **1**
So we will look at the next x_A

An even better approach (two pointers)!



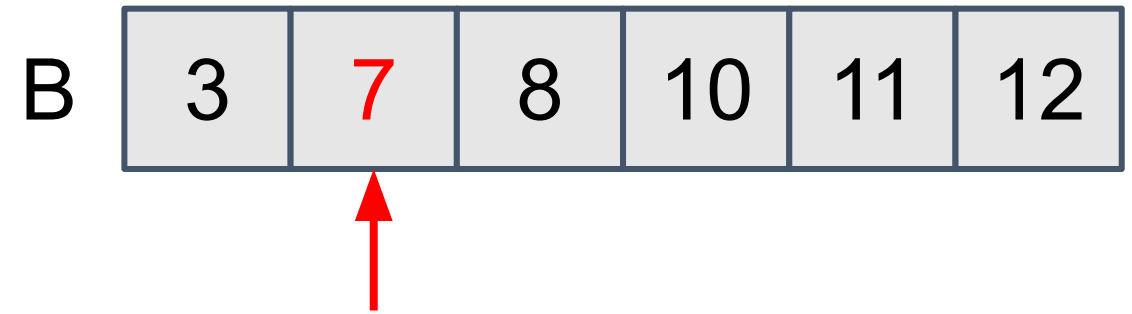
3 is equal to 3 so we append to C
We will look at the next x_A and x_B

An even better approach (two pointers)!



7 is already greater than **6** is so **6** cannot be in **B**
So we will look at the next x_A

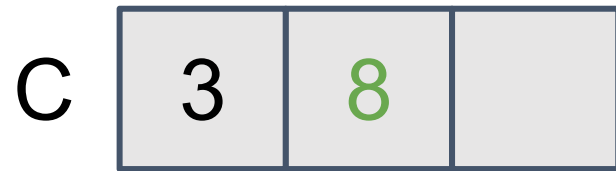
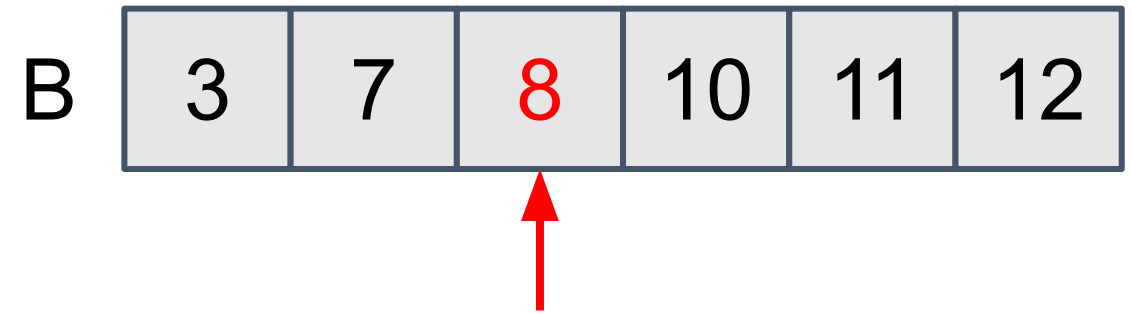
An even better approach (two pointers)!



7 is lesser than 8

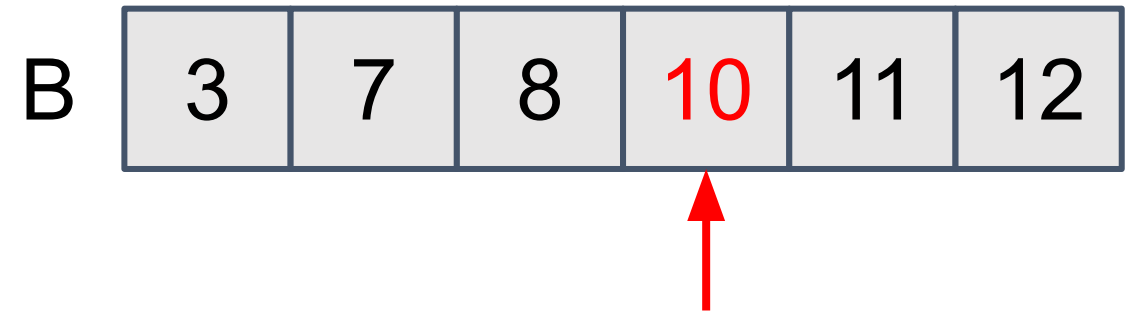
So we will look at the next x_B

An even better approach (two pointers)!



8 is equal to 8 so we append to **C**
We will look at the next x_A and x_B

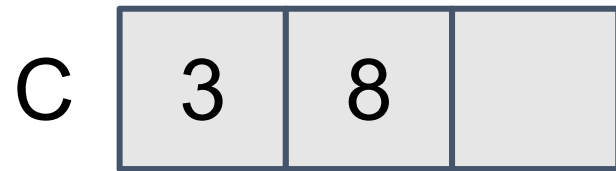
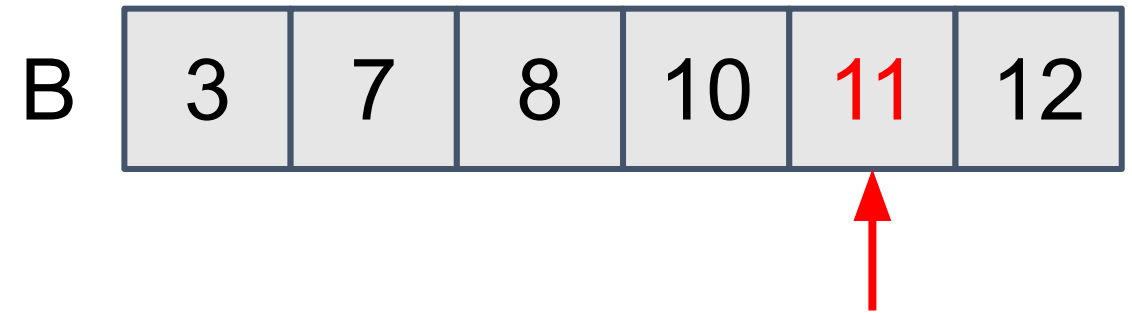
An even better approach (two pointers)!



10 is lesser than 12

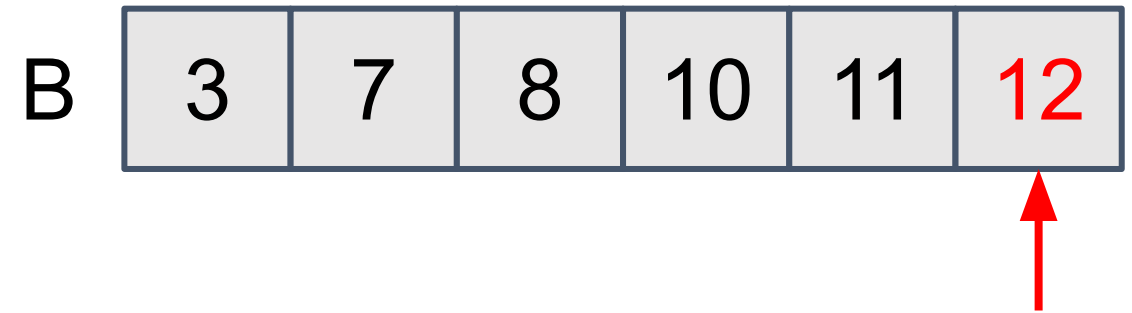
So we will look at the next x_B

An even better approach (two pointers)!



11 is still lesser than 12
So we will look at the next x_B

An even better approach (two pointers)!



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 (two pointers)!

This is almost the same as merge subroutine in merge sort.

Complexity: **$O(m + n)$**

Application 6: Target pair (AKA Two sum)

- Given an array, and a number **z**.
- Find a numbers **x**, **y** in the array such that **$x + y = z$** .
- 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: $O(n^2)$

A better approach (binary search)

Property: Array is sorted!

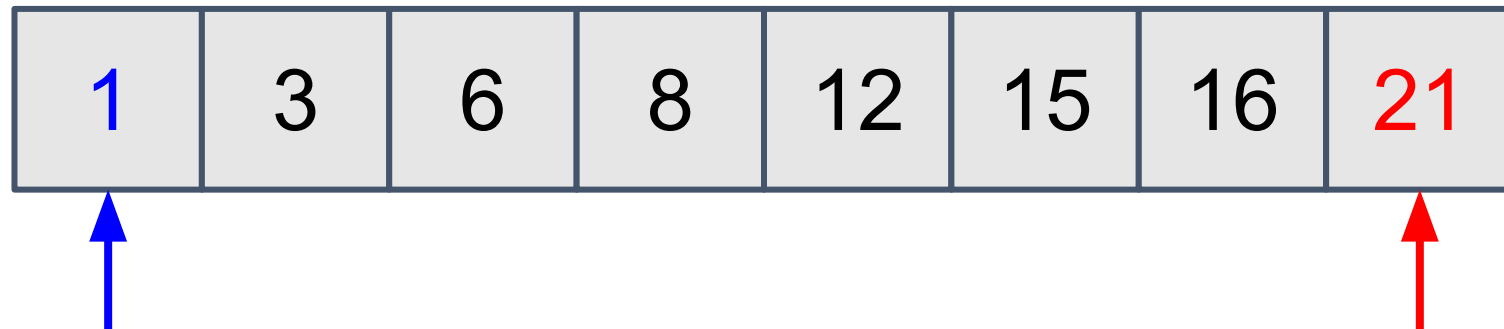
Instead of using linear search, we could use binary search. i.e. For every \mathbf{x} , search for $\mathbf{z} - \mathbf{x}$ on RHS using binary search

Complexity: **$O(n \log n)$**

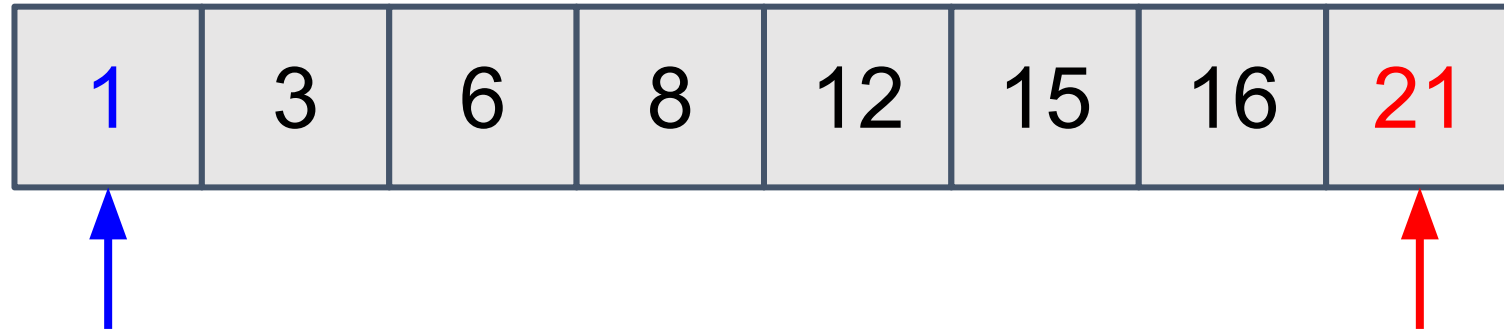
An even better approach!

We use two pointers: one **low-pointer** and one **high-pointer**. Our algorithm assumes that **[low, high]** has chance of finding an answer. 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!

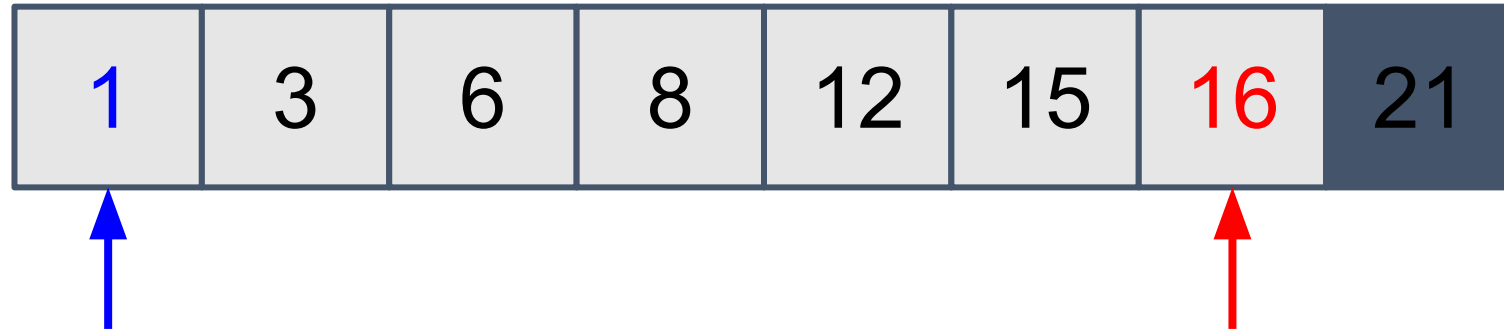


$$1 + 21 = 22 > 18$$

It's too large! **21** cannot be summed up with any number to target z , therefore we will eliminate it.

We shall retreat **high-pointer**.

An even better approach!

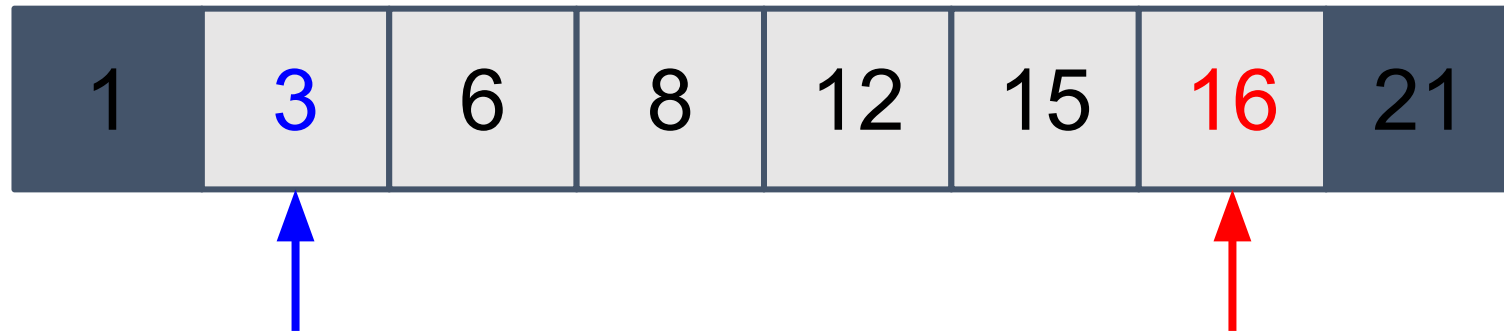


$$1 + 16 = 17 < 18$$

It's too small! **1** cannot be summed up with any number to target z , therefore we will eliminate it.

We shall advance **low-pointer**.

An even better approach!

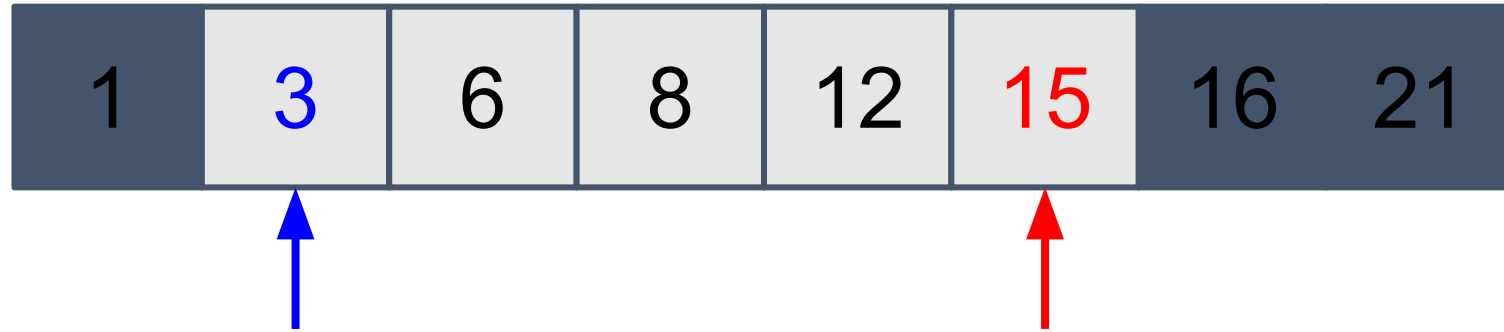


$$3 + 16 = 19 > 18$$

It's too large! **16** cannot be summed up with any number to target z , therefore we will eliminate it.

We shall 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		$O(N)$				

- Ascending/Descending vs non-Decreasing/non-Increasing.
- Nearly Sorted how? Many possibilities:
 - Low inversion count
 - Number of mismatch low.
 - Deviation from intended position low.

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	$O(N \log N)$	$O(N^2)$		$O(N \log N)$		
Quick Sort						
(Rand) Quick Sort						
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 →	Random	Sorted		Nearly Sorted		Homogeneous
Algorithm ↓		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	
(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 $\{1, 2, 3, \dots, 100000, 0\}$. It's a also worst case!

Input order →	Random	Sorted		Nearly Sorted		Homogeneous
Algorithm ↓		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	
(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 →	Random	Sorted		Nearly Sorted		Homogeneous
Algorithm ↓		Ascending	Descending	Ascending	Descending	
(Opt) Bubble sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N^2)$	$O(N^2)$	
(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 $\{1, 2, 3, \dots, 100000, 0\}$. $N-1$ comparisons to update sorted region until index $N-2$, N time to insert 0 at first index. So it is $O(N)$ ⁵²

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 →	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	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
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 →	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	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
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 →	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	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
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 2 Sep 2019 is flawed for homogenous input. This can be easily fixed to achieve $O(N)$ best case.

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	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
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 [‡]	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

[†] More precisely $O(N + k)$ where k is the maximum value in input

[‡] 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)

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 ?

Non-comparative sorting

- Realize that we cannot easily 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

- Counting sort runs in **$O(N + k)$** time and requires **$O(k)$** space, where **k** is the size of value range in the array
- 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).

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.

Pick the sorting algorithm

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)

Pick the sorting algorithm

	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!

Question 3: Quick Select

Popular programming interview question!

Quick select

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

Algorithm outline

1. Randomly pick a number as **pivot**
2. Compute its rank
 - a. If $k == \text{rank}$, we are done
 - b. If $k < \text{rank}$, our target is to the left
 - c. If $k > \text{rank}$, 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 Array 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 Type (ADT)

Since ADTs are defined by operations:

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

Common List ADT operations

- We have seen List Array ADT in tutorial 1
- List ADT is a more generalised and common type

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

Questions?

PS2 Discussion

PS2: /universityzoning

- First let's find how much a student needs to move:

PS2: /universityzoning

- First let's find how much a student needs to move.
- For each faculty do:
 - Sort the cells row wise, if tie then column wise.
 - Sort the student ids of students of the faculty.
 - First student needs to go to first cell, second to second etc.

PS2: /universityzoning

- First let's find how much a student needs to move.
- For each faculty do:
 - Sort the cells row wise, if tie then column wise.
 - Sort the student ids of students of the faculty.
 - First student needs to go to first cell, second to second etc.
- Note that multiple students can be in the same cell.

PS2: /universityzoning

- First let's find how much a student needs to move.
- For each faculty do:
 - Sort the cells row wise, if tie then column wise.
 - Sort the student ids of students of the faculty.
 - First student needs to go to first cell, second to second etc.
- Note that multiple students can be in the same cell.
- Distance from **(a, b)** to **(c, d)** is just **$|a - c| + |b - d|$** .

PS2: /universityzoning

- We need to make minimum student movement, such that at least **G** faculties have their threshold met.

PS2: /universityzoning

- We need to make minimum student movement, such that at least **G** faculties have their threshold met.
- It will be optimal to prioritize the faculties that need less student movement to fulfil its threshold!

PS2: /universityzoning

- We need to make minimum student movement, such that at least **G** faculties have their threshold met.
- It will be optimal to prioritize the faculties that need less student movement to fulfil its threshold!
- So, for each faculty **f**, we need to find how minimal movement it needs to reach the threshold **T_f**.

PS2: /universityzoning

- We need to make minimum student movement, such that at least **G** faculties have their threshold met.
- It will be optimal to prioritize the faculties that need less student movement to fulfil its threshold!
- So, for each faculty **f**, we need to find how minimal movement it needs to reach the threshold **T_f**. Call this value **movement_f**.
- To minimize movement to reach threshold, it will be optimal to move in the closest **T_f** students!

PS2: /universityzoning

- To calculate **movement_f**:
 - Sort the students of faculty **f** by their distance to assigned cell.
 - Sum up the lowest **T_f** distances.

PS2: /universityzoning

- To calculate **movement_f**:
 - Sort the students of faculty **f** by their distance to assigned cell.
 - Sum up the lowest **T_f** distances.
- Sort the **movement** values.
- The answer to the problem will be sum of lowest **G** movement values.

PS2: /jobbyte

- Assume that the array is 1-indexed.

PS2: /jobbyte

- Assume that the array is 1-indexed.
- For an index i , where should the element $a[i]$ go?

PS2: /jobbyte

- Assume that the array is 1-indexed.
- For an index i , where should the element $a[i]$ go?
 - Element at index i should go to index $a[i]$.
- Where should the element at index $a[i]$ go?

PS2: /jobbyte

- Assume that the array is 1-indexed.
- For an index i , where should the element $a[i]$ go?
 - Element at index i should go to index $a[i]$.
- Where should the element at index $a[i]$ go?
 - Element at index $a[i]$ should go to index $a[a[i]]$.
-
-

PS2: /jobbyte

- Can this go on forever?

$i \rightarrow j \rightarrow k \rightarrow l \rightarrow \dots$

PS2: /jobbyte

- Can this go on forever?

$i \rightarrow j \rightarrow k \rightarrow l \rightarrow \dots$

- No. At some point, this chain must intersect itself!
 - Since we have finite number of elements.

PS2: /jobbyte

- Can this go on forever?

$i \rightarrow j \rightarrow k \rightarrow l \rightarrow \dots$

- No. At some point, this chain must intersect itself!
 - Since we have finite number of elements.
- Can this happen?

$i \rightarrow j \rightarrow k \rightarrow l \rightarrow \dots \rightarrow x \rightarrow k \rightarrow l \rightarrow \dots$

PS2: /jobbyte

- Can this go on forever?

$i \rightarrow j \rightarrow k \rightarrow l \rightarrow \dots$

- No. At some point, this chain must intersect itself!

- Since we have finite number of elements.

- Can this happen?

$i \rightarrow j \rightarrow k \rightarrow l \rightarrow \dots \rightarrow x \rightarrow k \rightarrow l \rightarrow \dots$

- No. Because this means $a[j] = k$, as well as $a[x] = k$. But the array has no duplicates.

- So only way the chain can intersect itself is by making a cycle!

$i \rightarrow j \rightarrow k \rightarrow l \rightarrow \dots \rightarrow i \rightarrow \dots$

PS2: [/jobbyte](#)

- Suppose the entire array contains only one cycle. What is the answer?

PS2: /jobbyte

- Suppose the entire array contains only one cycle. What is the answer?
 - **Answer: $N - 1$.**
- What if it has two cycles?

PS2: /jobbyte

- Suppose the entire array contains only one cycle. What is the answer?
 - **Answer: $N - 1$.**
- What if it has two cycles?
 - It is never optimal to swap two indices that are part of two different cycles.
 - Just sort the two cycles separately!

PS2: /jobbyte

- Suppose the entire array contains only one cycle. What is the answer?
 - **Answer: $N - 1$.**
- What if it has two cycles?
 - It is never optimal to swap two indices that are part of two different cycles.
 - Just sort the two cycles separately!
 - **Answer: $(\text{len of cycle 1} - 1) + (\text{len of cycle 2} - 1) = N - 2$.**
-
- What if it has k cycles?

PS2: /jobbyte

- Suppose the entire array contains only one cycle. What is the answer?
 - **Answer: $N - 1$.**
- What if it has two cycles?
 - It is never optimal to swap two indices that are part of two different cycles.
 - Just sort the two cycles separately!
 - **Answer: $(\text{len of cycle}_1 - 1) + (\text{len of cycle}_2 - 1) = N - 2$.**
-
- What if it has k cycles?
 - **Answer: $N - k$.**

Hands-on session

<https://visualgo.net/training?diff=Medium&n=5&tl=5&module=sorting>

Hands-on session: [/basicprogrmaming2](#)

Thank You!

Anonymous Feedback:

<https://forms.gle/MkETeXdUT53Vhh896>