PSO 5

Sorting

Announcements

Project deadline extended to this Saturday

GL on 250 exam

(Heap sort) In the following questions, we consider Heap sort using **Heapify**.

(1) Show the array $\{3, 4, 1, 0, 9, 2\}$ as it goes through Heap sort (in the ascending order).

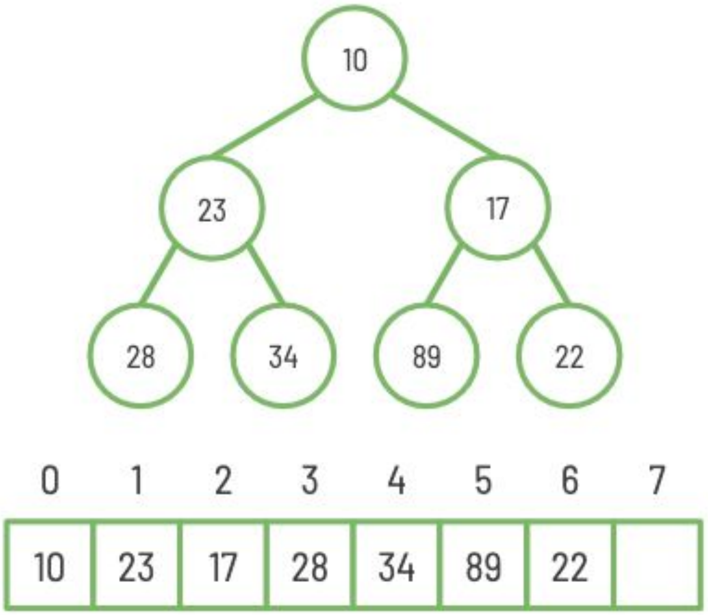
(2) Given K number of sorted (ascending ordered) arrays each having N/K elements in it, your task is to merge all these arrays to form a N-element final sorted array (also in the ascending order).

(2.1) Propose a simple solution to the problem which may run in $O(N \log(N))$ time.

(2.2) Can you propose a better algorithm to solve the problem? What is the time complexity of your proposed solution?

Heap Insertion

1. Insert at next leaf
2. Sift up



Demonstration: insert(9)

(Max) Heapify: Turning your arrays into Heaps

For each **non-leaf node** from the last to the first:

while it is less than its largest child:

swap it downward

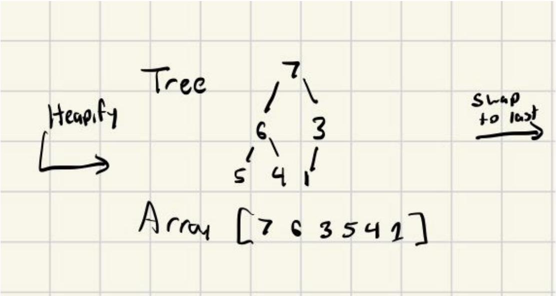
Demonstr. : Heapify [4 6 3 5 7 1]

Heap Sort

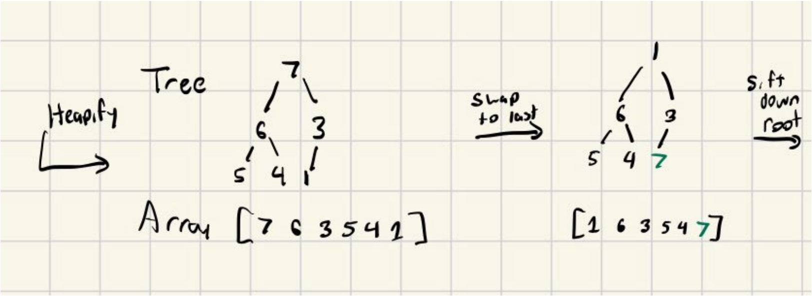
Idea: In a max heap, the max element is always at the root, sort backwards, from largest to smallest

1. Heapify your array
2. Swap root with last leaf, excluding the elements you’ve already swapped
3. Fix heap by sifting down, excluding the elements you’ve already swapped
4. Repeat steps 2-4

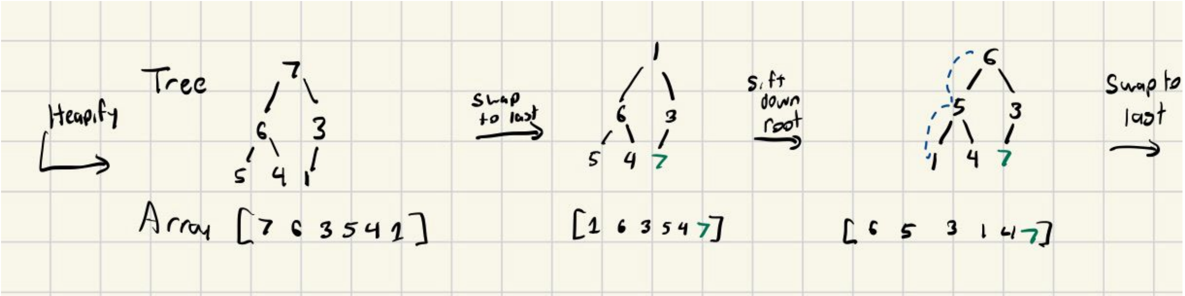
[4 6 3 5 7 1]



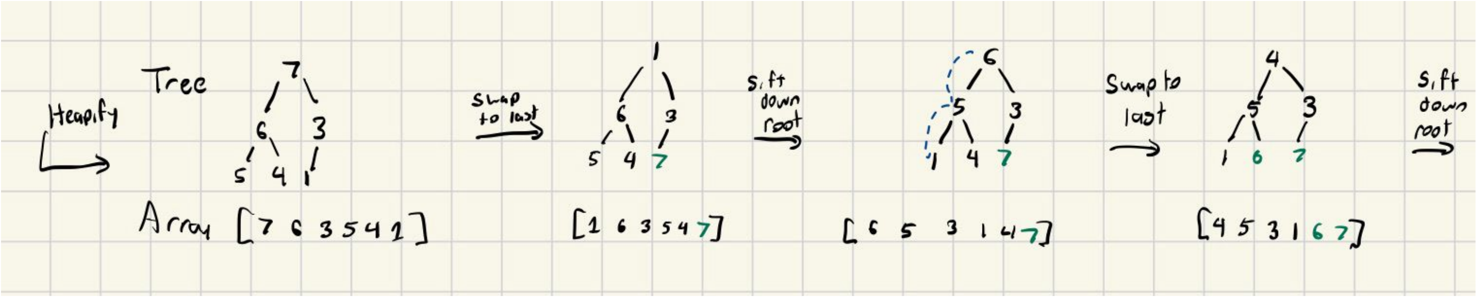
1. Heapify your array
2. Swap root with last leaf, excluding the elements you've already swapped
3. Fix heap by sifting down, excluding the elements you've already swapped
4. Repeat steps 2-4



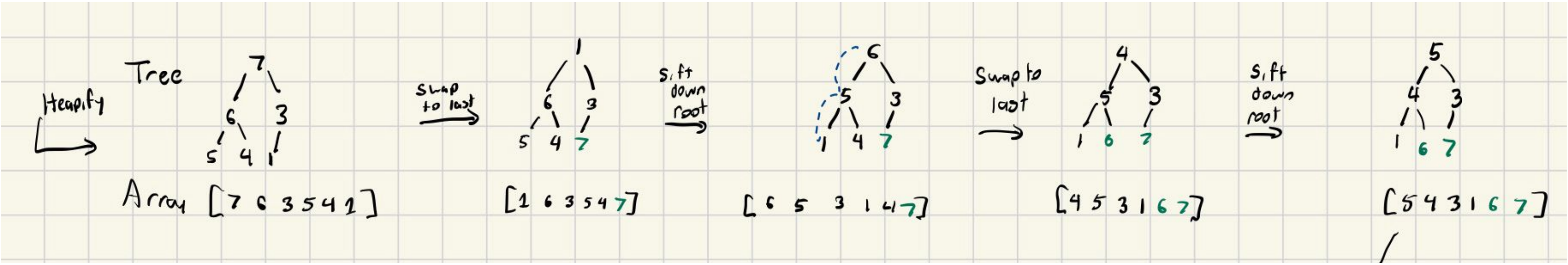
1. Heapify your array
2. Swap root with last leaf, excluding the elements you've already swapped
3. Fix heap by sifting down, excluding the elements you've already swapped
4. Repeat steps 2-4



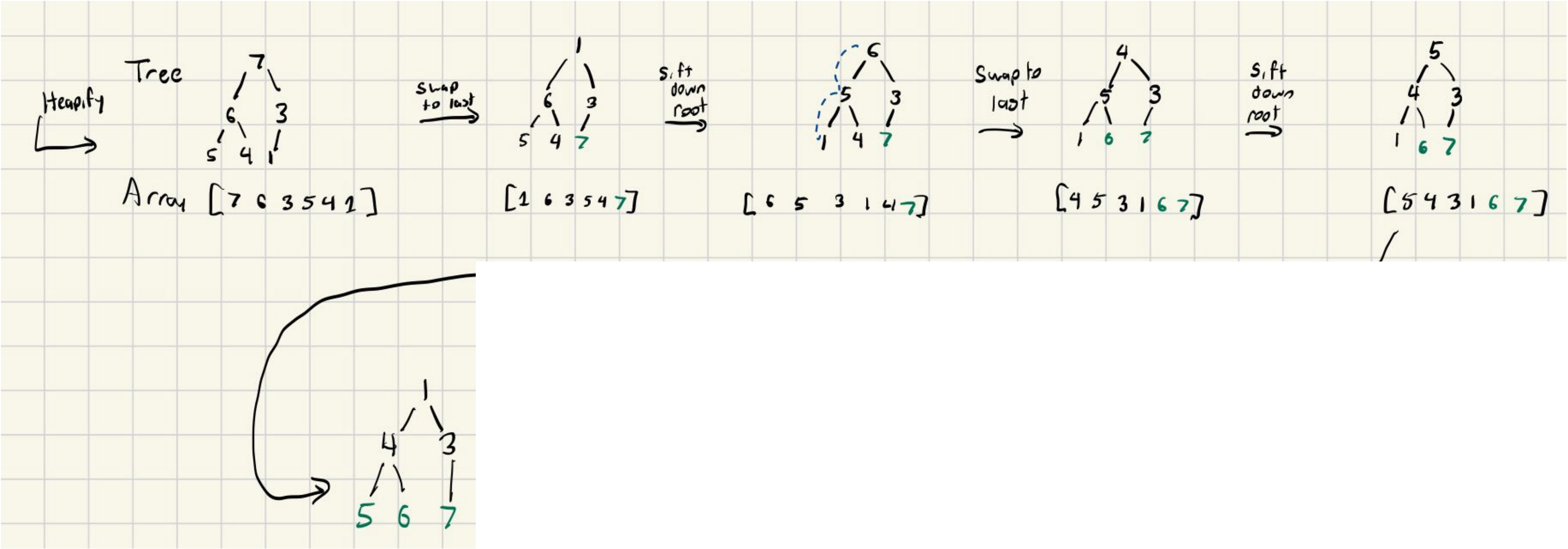
1. Heapify your array
2. Swap root with last leaf, excluding the elements you've already swapped
3. Fix heap by sifting down, excluding the elements you've already swapped
4. Repeat steps 2-4



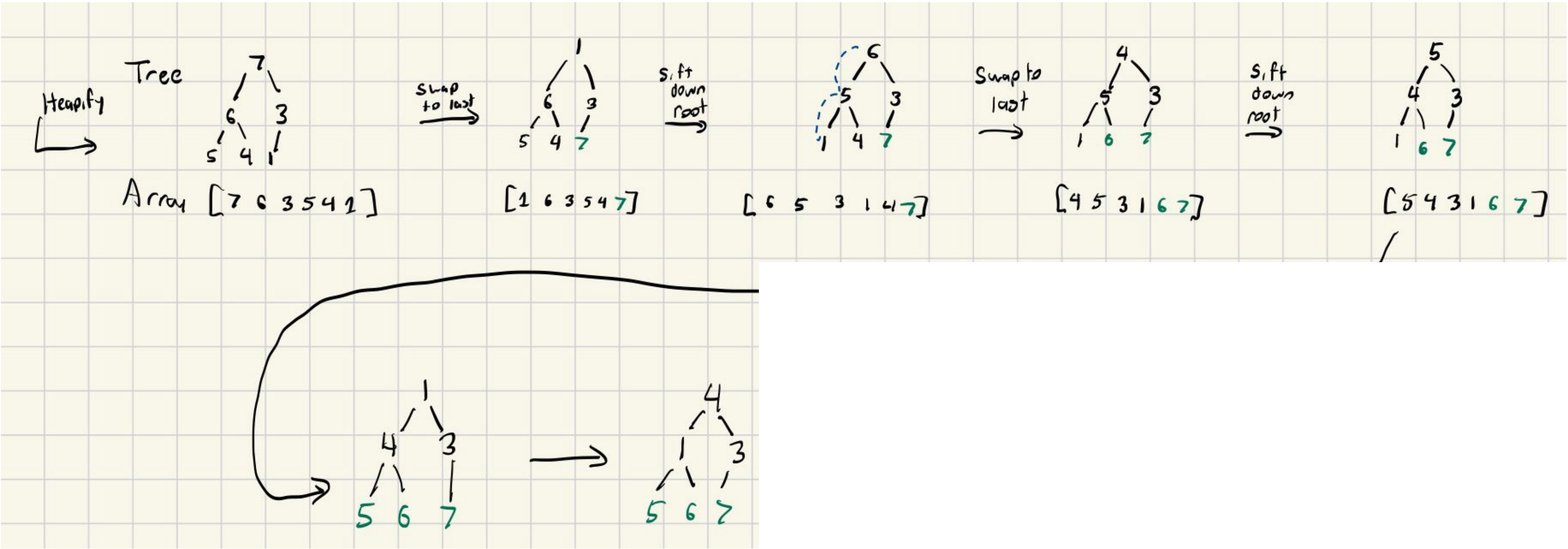
1. Heapify your array
2. Swap root with last leaf, excluding the elements you've already swapped
3. Fix heap by sifting down, excluding the elements you've already swapped
4. Repeat steps 2-4



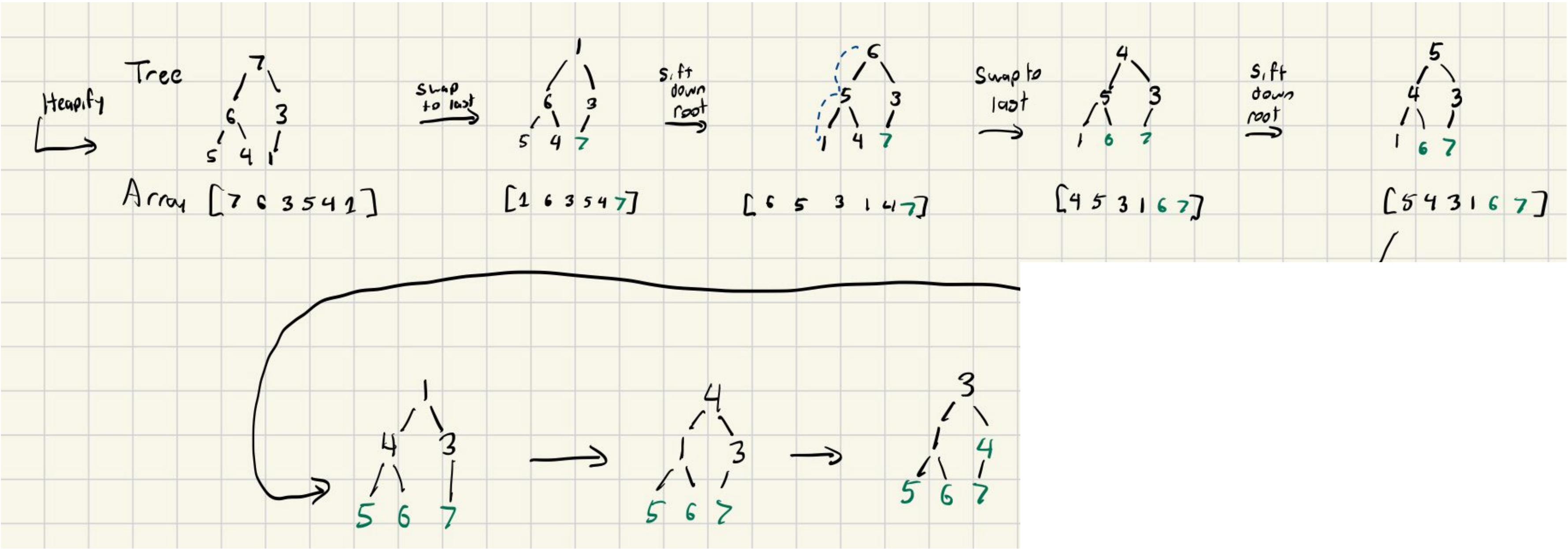
1. Heapify your array
2. Swap root with last leaf, excluding the elements you've already swapped
3. Fix heap by sifting down, excluding the elements you've already swapped
4. Repeat steps 2-4



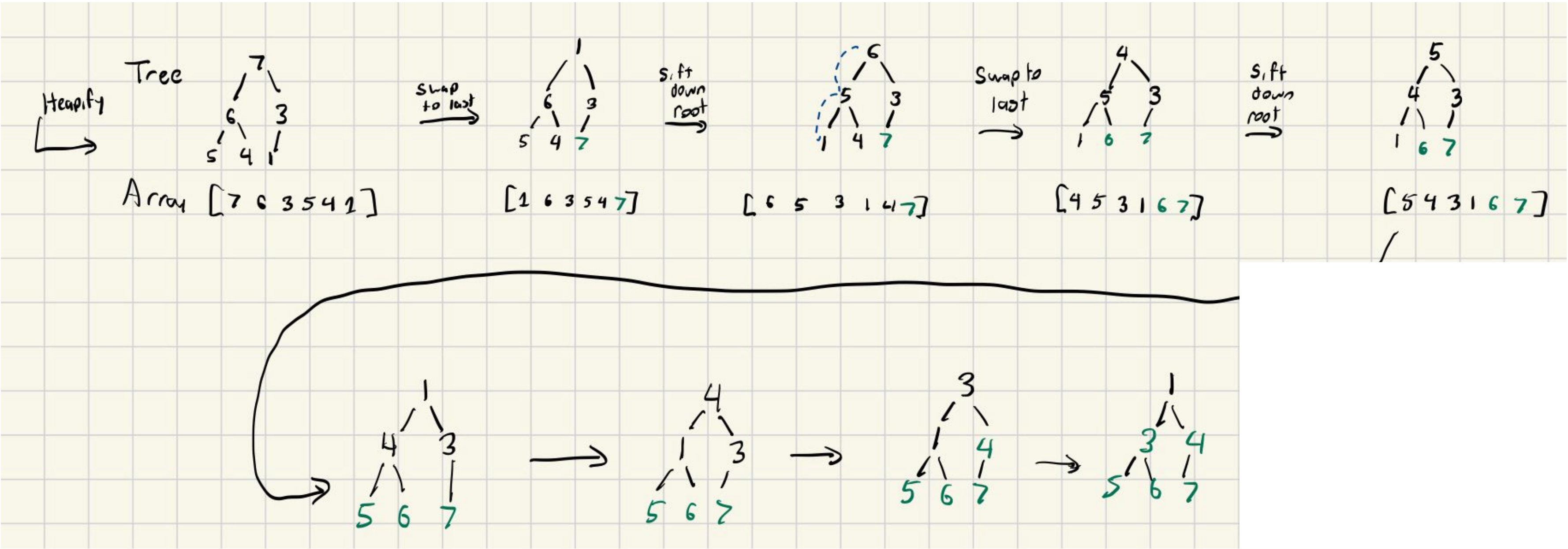
1. Heapify your array
2. Swap root with last leaf, excluding the elements you've already swapped
3. Fix heap by sifting down, excluding the elements you've already swapped
4. Repeat steps 2-4



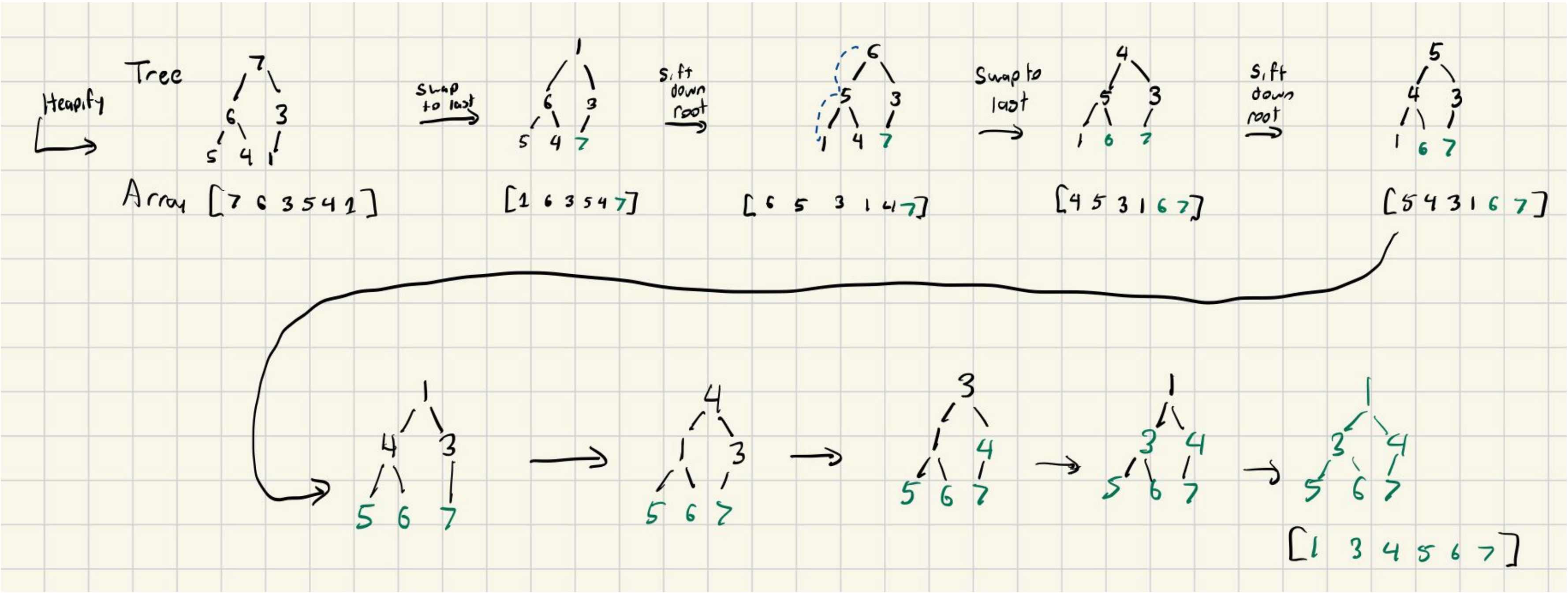
1. Heapify your array
2. Swap root with last leaf, excluding the elements you've already swapped
3. Fix heap by sifting down, excluding the elements you've already swapped
4. Repeat steps 2-4



1. Heapify your array
2. Swap root with last leaf, excluding the elements you've already swapped
3. Fix heap by sifting down, excluding the elements you've already swapped
4. Repeat steps 2-4



1. Heapify your array
2. Swap root with last leaf, excluding the elements you've already swapped
3. Fix heap by sifting down, excluding the elements you've already swapped
4. Repeat steps 2-4



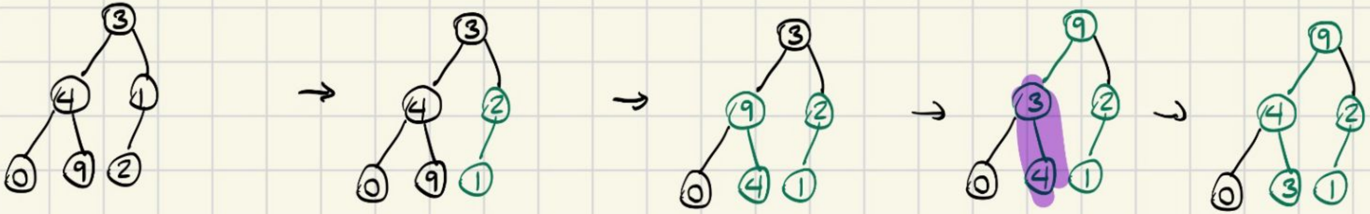
1. Heapify your array
2. Swap root with last leaf, excluding the elements you've already swapped
3. Fix heap by sifting down, excluding the elements you've already swapped
4. Repeat steps 2-4

(Heap sort) In the following questions, we consider Heap sort using **Heapify**.

(1) Show the array $\{3, 4, 1, 0, 9, 2\}$ as it goes through Heap sort (in the ascending order).

max

1) Heapify [3, 4, 1, 0, 9, 2]



= [9, 4, 2, 0, 3, 1]

(Heap sort) In the following questions, we consider Heap sort using **Heapify**.

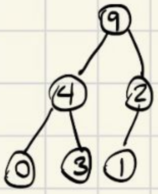
(1) Show the array $\{3, 4, 1, 0, 9, 2\}$ as it goes through Heap sort (in the ascending order).

2) Sorting

a) swap root with last leaf (not seen)

b) sift down (not seen)

c) repeat



(Heap sort) In the following questions, we consider Heap sort using **Heapify**.

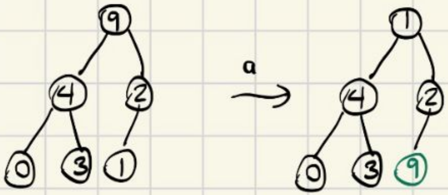
(1) Show the array $\{3, 4, 1, 0, 9, 2\}$ as it goes through Heap sort (in the ascending order).

2) Sorting

a) swap root with last leaf (not seen)

b) sift down (not seen)

c) repeat



(Heap sort) In the following questions, we consider Heap sort using **Heapify**.

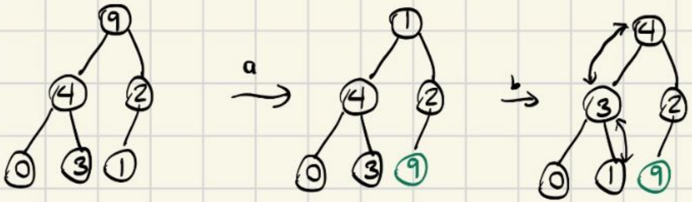
(1) Show the array $\{3, 4, 1, 0, 9, 2\}$ as it goes through Heap sort (in the ascending order).

2) Sorting

a) swap root with last leaf (not seen)

b) Sift down (not seen)

c) repeat



(Heap sort) In the following questions, we consider Heap sort using **Heapify**.

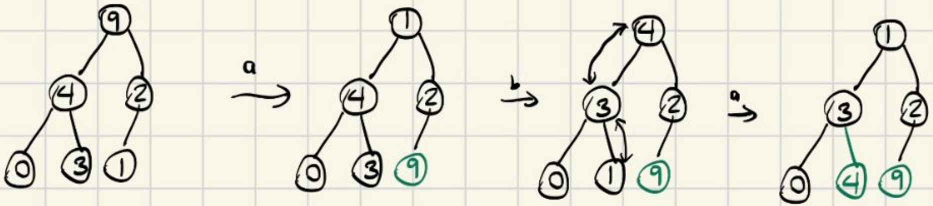
(1) Show the array $\{3, 4, 1, 0, 9, 2\}$ as it goes through Heap sort (in the ascending order).

2) Sorting

a) swap root with last leaf (not seen)

b) sift down (not seen)

c) repeat



(Heap sort) In the following questions, we consider Heap sort using **Heapify**.

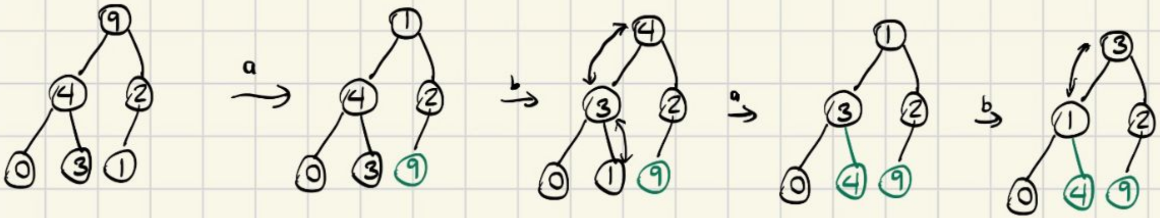
(1) Show the array $\{3, 4, 1, 0, 9, 2\}$ as it goes through Heap sort (in the ascending order).

2) Sorting

a) swap root with last leaf (not seen)

b) sift down (not seen)

c) repeat



(Heap sort) In the following questions, we consider Heap sort using **Heapify**.

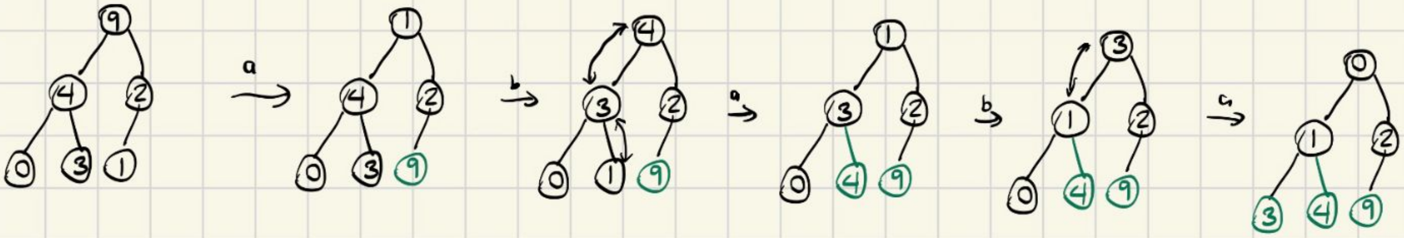
(1) Show the array $\{3, 4, 1, 0, 9, 2\}$ as it goes through Heap sort (in the ascending order).

2) Sorting

a) swap root with last leaf (not seen)

b) sift down (not seen)

c) repeat



(Heap sort) In the following questions, we consider Heap sort using **Heapify**.

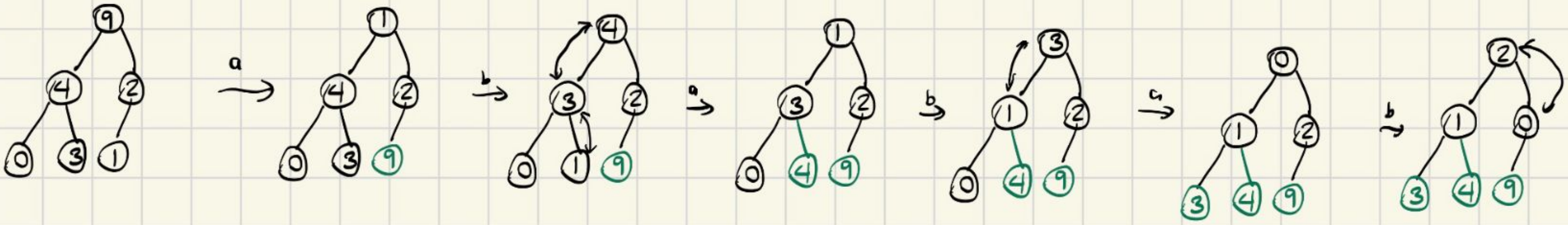
(1) Show the array $\{3, 4, 1, 0, 9, 2\}$ as it goes through Heap sort (in the ascending order).

2) Sorting

a) swap root with last leaf (not seen)

b) sift down (not seen)

c) repeat



And so forth

Heap Summary Costs

For a heap with 🎺 items,

Heapify: O(🎺)

Add/Pop: O(log 🎺)

Heap Sort: O(🎺log 🎺)

(2) Given K number of sorted (ascending ordered) arrays each having N/K elements in it, your task is to merge all these arrays to form a N-element final sorted array (also in the ascending order).

(2.1) Propose a simple solution to the problem which may run in $O(N \log(N))$ time.

(2) Given K number of sorted (ascending ordered) arrays each having N/K elements in it, your task is to merge all these arrays to form a N-element final sorted array (also in the ascending order).

(2.1) Propose a simple solution to the problem which may run in $O(N \log(N))$ time.

Just run merge sort on the combined array

(2) Given K number of sorted (ascending ordered) arrays each having N/K elements in it, your task is to merge all these arrays to form a N-element final sorted array (also in the ascending order).

(2.1) Propose a simple solution to the problem which may run in $O(N \log(N))$ time.

(2.2) Can you propose a better algorithm to solve the problem? What is the time complexity of your proposed solution?

Example (N = 12, K = 3):

1	3	5	7

2	4	5	5

9	10	11	12

Can I use a heap somehow?

Idea: index-wise heap sorting

Example ($N = 12$, $K = 3$):

1	3	5	7

2	4	5	5

9	10	11	12

Iteratively sort and add items to the heap

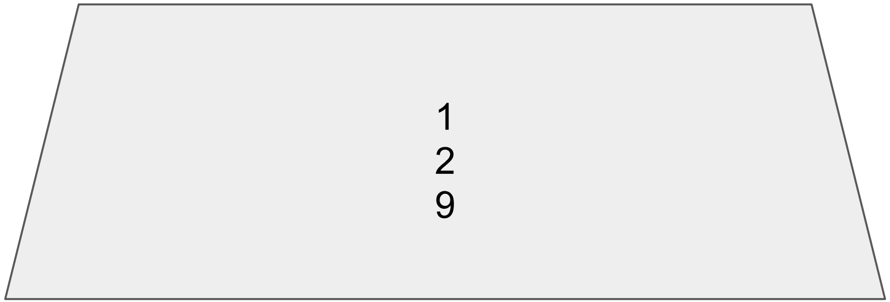
This is a heap (as seen in the wild)

Idea: index-wise heap sorting

Example (N = 12, K = 3):

1	3	5	7
2	4	5	5
9	10	11	12

Iteratively sort and add items to the heap



1. Add first index elements to heap

Idea: index-wise heap sorting

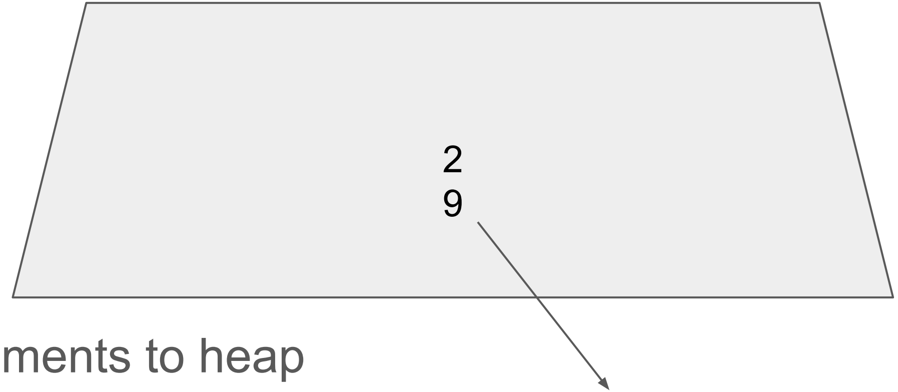
Example (N = 12, K = 3):z

1	3	5	7

2	4	5	5

9	10	11	12

Iteratively sort and add items to the heap



1. Add first index elements to heap
2. Pop heap and append to res array

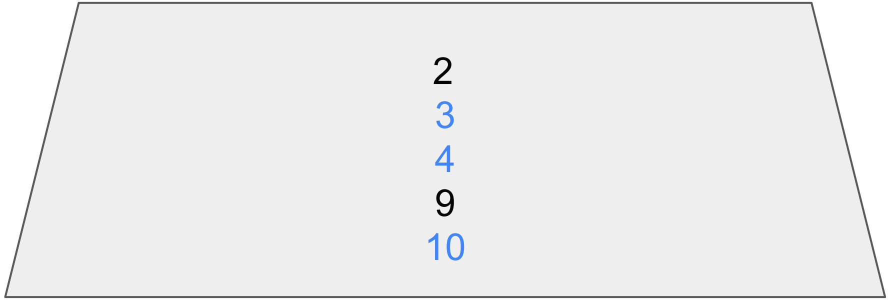
res = 1

Idea: index-wise heap sorting

Example (N = 12, K = 3):z

1	3	5	7
2	4	5	5
9	10	11	12

Iteratively sort and add items to the heap



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

res = 1

Idea: index-wise heap sorting

Problem
Heap size at most N
Overall will be O(NlogN)

Example (N = 12, K = 3):z

It ... items to the heap

1	3	5	7

2	4	5	5

9	10	11	12

2
3
4
9
10

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

res = 1

Why will the heap has size O(n)?

Problem
Heap size at most
N
Overall will be
O(NlogN)

Example (N = 12, K = 3):z

1	3	5	7
2	4	5	5
9	10	11	12

Iterativ d ms to the heap

3
4
5
5
9
10
11

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

res = 1

Why will the heap has size O(n)?

Problem
Heap size at most
N
Overall will be
O(NlogN)

Example (N = 12, K = 3):z

1	3	5	7

2	4	5	5

9	10	11	12

Iterativ... ms to the heap

4
5
5
5
7
9
10
11
12

(12 - 3) = 9 elts

In general:
Will have <= (n-k) elts in heap

Can we limit to just k?

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

res = 1

Keeping our heap to size K

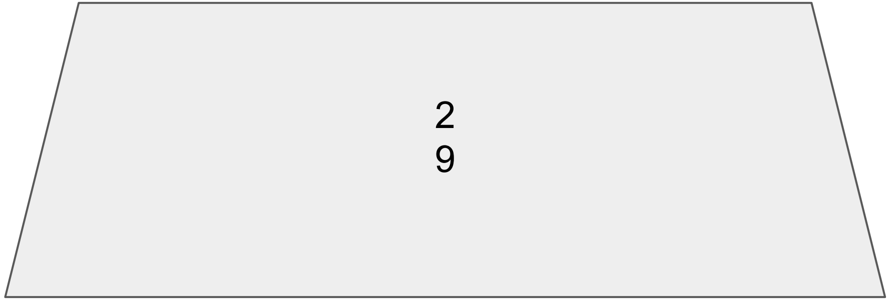
Example (N = 12, K = 3):z

1	3	5	7

2	4	5	5

9	10	11	12

Iteratively sort and add items to the heap



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

Only add next index element from popped array

res = **1**

Keeping our heap to size K

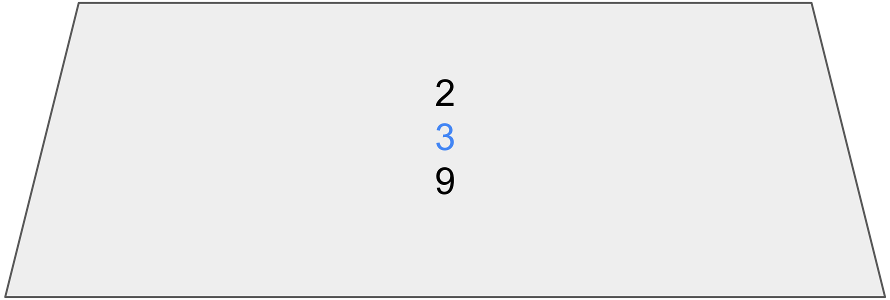
Example (N = 12, K = 3):z

1	3	5	7

2	4	5	5

9	10	11	12

Iteratively sort and add items to the heap



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

   Only add next index element from popped array

res = 1

Keeping our heap to size K

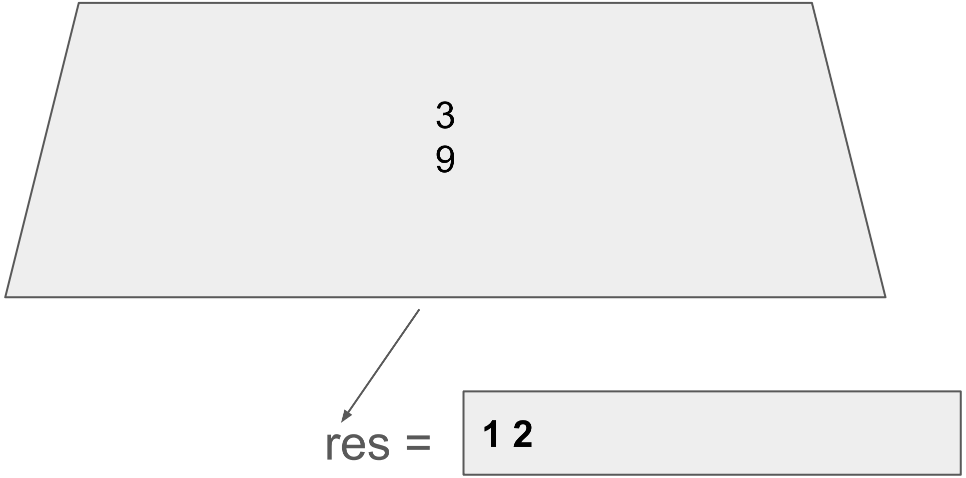
Example (N = 12, K = 3):z

1	3	5	7

2	4	5	5

9	10	11	12

Iteratively sort and add items to the heap



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

   Only add next index element from popped array

Keeping our heap to size K

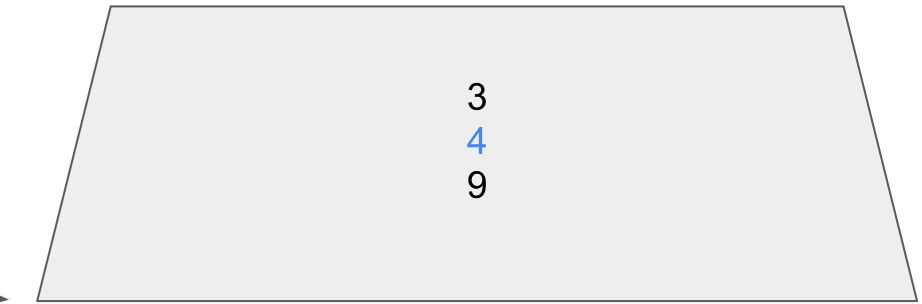
Example (N = 12, K = 3):z

1	3	5	7

2	4	5	5

9	10	11	12

Iteratively sort and add items to the heap



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?
   Only add next index element from popped array

res = **1 2**

Keeping our heap to

Okay.. but how do we know:

1. which array the popped element belongs to?
2. Its index in the array

Example (N = 12, K =

s to the heap

1	3	5	7

2	4	5	5

9	10	11	12

3
4
9

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?
   Only add next index element from popped array

res = **1 2**

Store the array number and the index!

Example (N = 12, K = 3):z

0

1	3	5	7

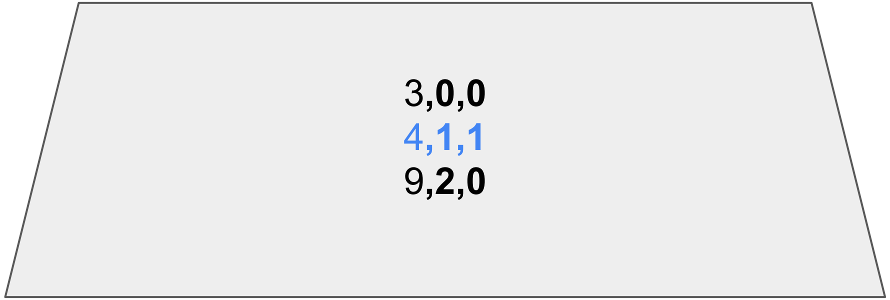
1

2	4	5	5

2

9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?
   Only add next index element from popped array

res = **1 2**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	5	7
1	**2**	4	5	5
2	9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**

4,1,1
9,2,0

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

Only add next index element from popped array

res = **1 2 3**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	5	7
1	**2**	4	5	5
2	9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**

4,1,1
5,0,2
9,2,0

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?
   Only add next index element from popped array

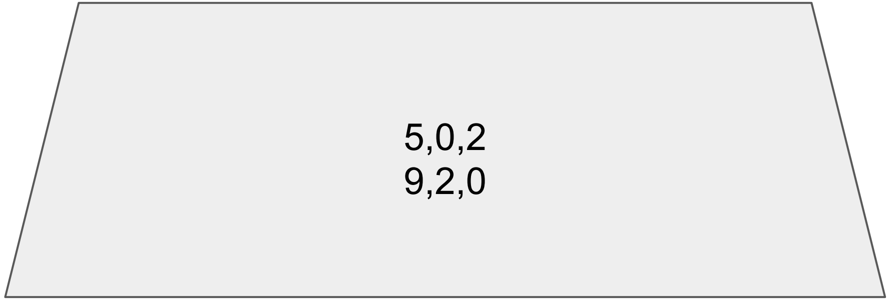
res = **1 2 3**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	5	7
1	**2**	**4**	5	5
2	9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

Only add next index element from popped array

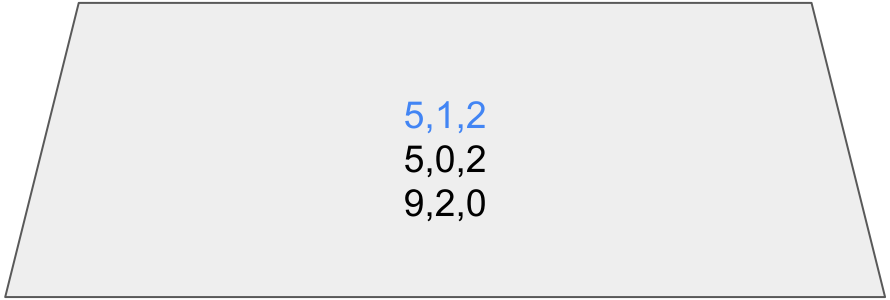
res = **1 2 3 4**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	5	7
1	**2**	**4**	5	5
2	9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**



res = **1 2 3 4**

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

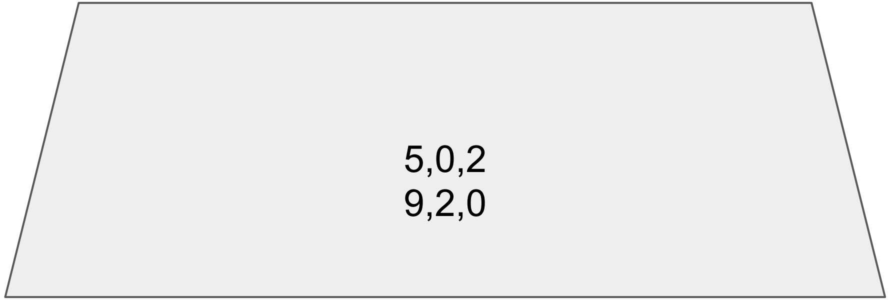
Only add next index element from popped array

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	5	7
1	**2**	**4**	**5**	5
2	9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

   Only add next index element from popped array

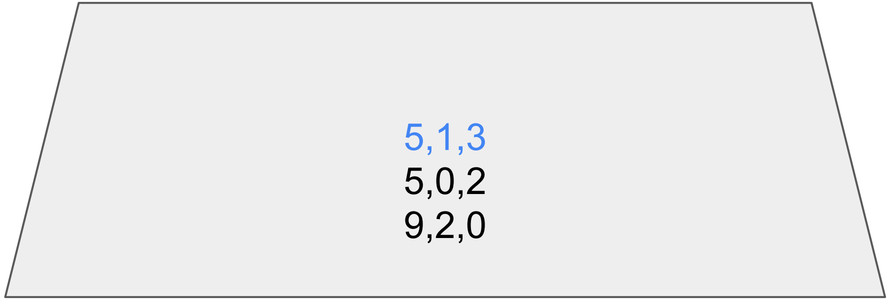
res = **1 2 3 4 5**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	5	7
1	**2**	**4**	**5**	5
2	9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

   Only add next index element from popped array

res = **1 2 3 4 5**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	5	7
1	**2**	**4**	**5**	**5**
2	9	10	11	12

Keep sorting the remaining arrays

5,0,2
9,2,0

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

   Only add next index element from popped array

res = **1 2 3 4 5 5**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	5	7
1	**2**	**4**	**5**	**5**
2	9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**

5,0,2
9,2,0

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

Only add next index element from popped array

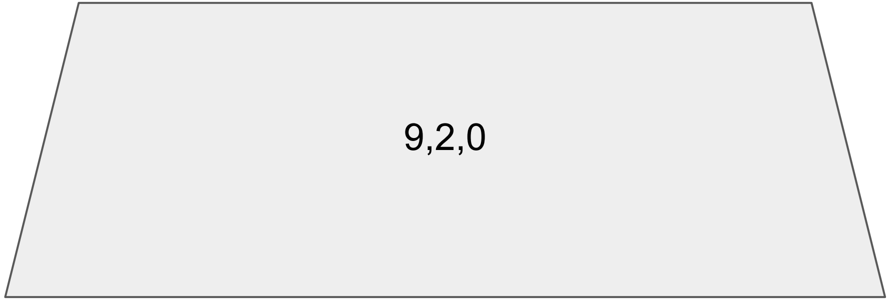
res = **1 2 3 4 5 5**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	**5**	7
1	**2**	**4**	**5**	**5**
2	9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?
   Only add next index element from popped array

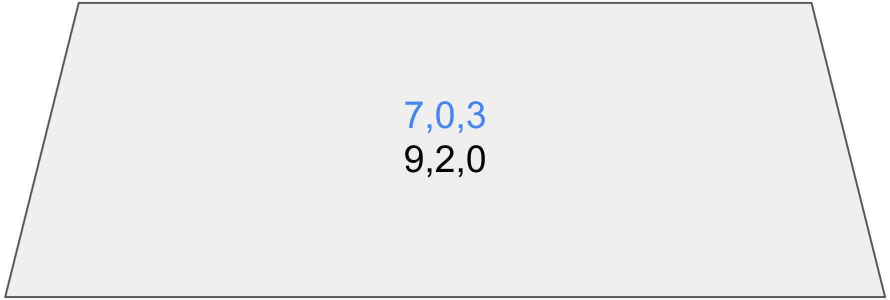
res = **1 2 3 4 5 5 5**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	**5**	7
1	**2**	**4**	**5**	**5**
2	9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?
   Only add next index element from popped array

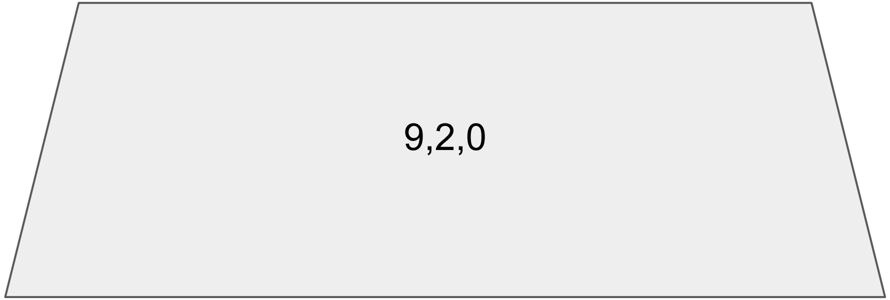
res = **1 2 3 4 5 5 5**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	**5**	7
1	**2**	**4**	**5**	**5**
2	9	10	11	12

Iteratively sort and add items to the heap **of the form x,array #,index**



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?
   Only add next index element from popped array

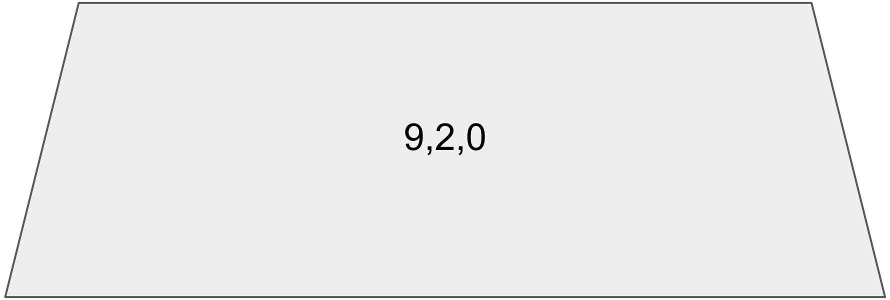
res = **1 2 3 4 5 5 5 7**

Store the index

Example (N = 12, K = 3):z

0	**1**	**3**	**5**	7
1	**2**	**4**	**5**	**5**
2	9	10	11	12

Add everything left from last array to res



res = **1 2 3 4 5 5 5 7**

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

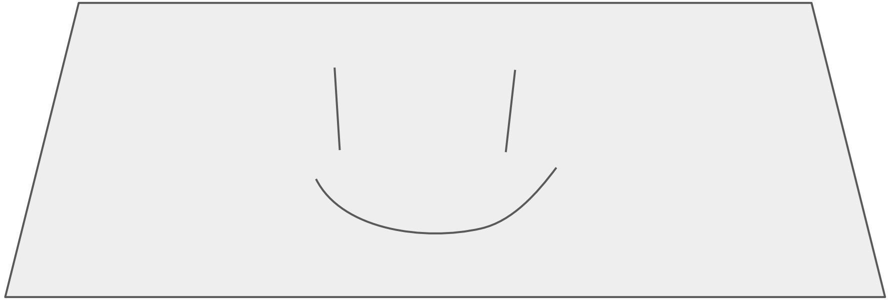
   Only add next index element from popped array

Time complexity?

Example (N = 12, K = 3):z

0	**1**	**3**	**5**	7
1	**2**	**4**	**5**	**5**
2	9	10	11	12

Add everything left from last array to res



1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?

   Only add next index element from popped array

res = **1 2 3 4 5 5 5 7** 9 10 11 12

Time complexity?

Example (N = 12, K = 3):z

0	**1**	**3**	**5**	7
1	**2**	**4**	**5**	**5**
2	9	10	11	12

N add/pop heap operations on a heap of size K

O(NlogK) cost

1. Add first index elements to heap
2. Pop heap and append to res array
3. Repeat for each index?
   Only add next index element from popped array

res = **1 2 3 4 5 5 5 7** 9 10 11 12

(Merge sort) Merge sort is in its nature, a Divide-and-Conquer algorithm.

(1) Suppose that when doing a Mergesort you recursively break lists into 4 equal-sized sub-arrays instead of 2. Will you get a better runtime performance asymptotically?

(2) You are given two sorted arrays that are identical except that one of them is missing a single element. In other words, one array has length n and the other has length $n-1$. The goal is to design an efficient algorithm with $O(\log n)$ runtime that finds the missing element.

(Merge sort) Merge sort is in its nature, a Divide-and-Conquer algorithm.

(1) Suppose that when doing a Mergesort you recursively break lists into 4 equal-sized sub-arrays instead of 2. Will you get a better runtime performance asymptotically?

(2) You are given two sorted arrays that are identical except that one of them is missing a single element. In other words, one array has length n and the other has length $n-1$. The goal is to design an efficient algorithm with $O(\log n)$ runtime that finds the missing element.

(Merge sort) Merge sort is in its nature, a Divide-and-Conquer algorithm.

(1) Suppose that when doing a Mergesort you recursively break lists into 4 equal-sized sub-arrays instead of 2. Will you get a better runtime performance asymptotically?

(2) You are given two sorted arrays that are identical except that one of them is missing a single element. In other words, one array has length n and the other has length $n-1$. The goal is to design an efficient algorithm with $O(\log n)$ runtime that finds the missing element.

Whenever you see

1. Array is sorted
2. O(log n) time required

99%[*] of the time, you can use a **modified binary search**

* Source: It was revealed to me in a dream

Example of Binary Search for x = 5

1	2	3	4	5	6	7	8

```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

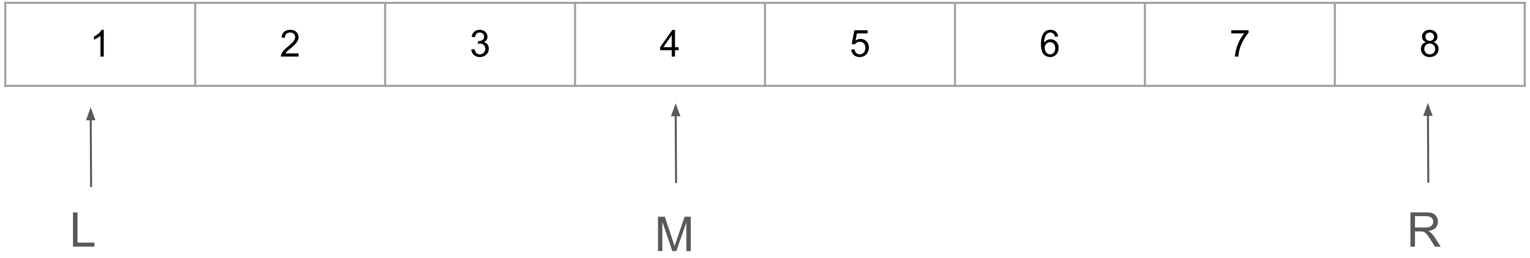
Example of Binary Search for x = 5

1	2	3	4	5	6	7	8

L R

```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

Example of Binary Search for x = 5




```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

Example of Binary Search for x = 5

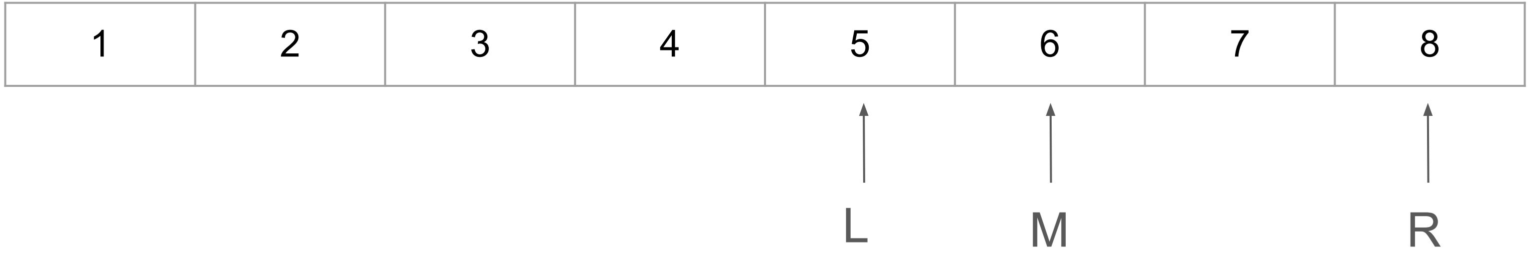
1	2	3	4	5	6	7	8

M L R

```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

First iteration done.

Example of Binary Search for x = 5




```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

Example of Binary Search for x = 5

1	2	3	4	5	6	7	8


```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

Since L = R, we found x!

Important parts of Binary Search

```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case

1	2	3	4	5	6	7	8

1	2	3	4	6	7	8

What should the search range be?

```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case

1	2	3	4	5	6	7	8

1	2	3	4	6	7	8

L

R

What should the search range be?

L = 1, R = n, the missing index could be any index

```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case

1	2	3	4	5	6	7	8

1	2	3	4	6	7	8

L

R

When should we stop?

This is often much trickier, run through the algorithm to figure this out.

```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case

1	2	3	4	5	6	7	8

1	2	3	4	6	7	8

L M R

When should we stop?

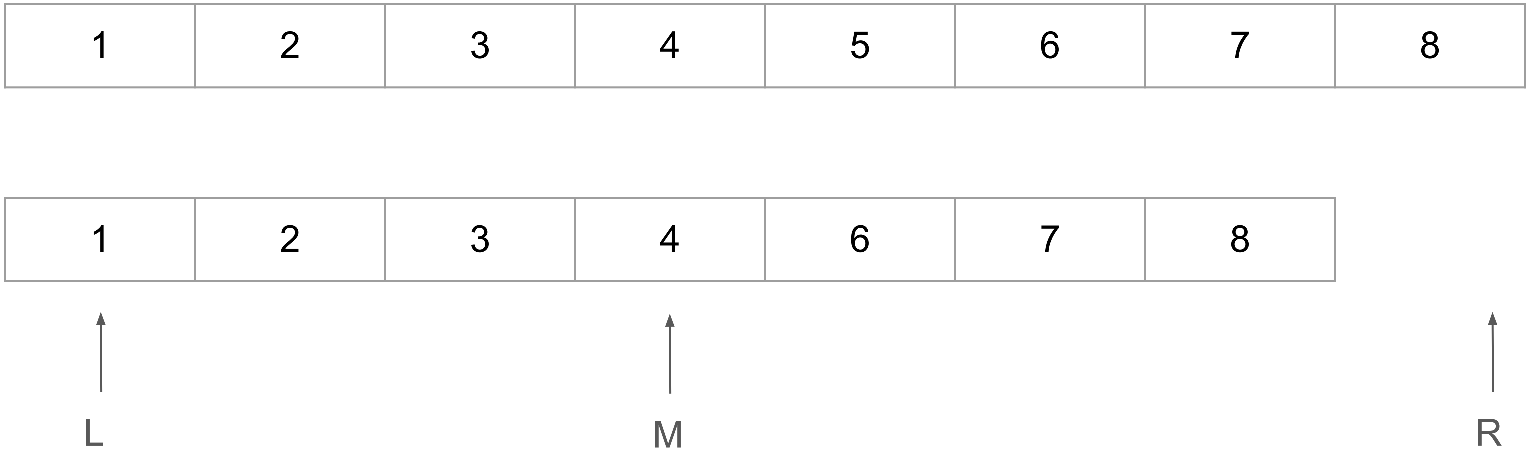
This is often much trickier to figure out next.

Instead, run through the binary search to figure out how to *interval cut*

```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case



How should we cut the interval? What does A[m] and B[m] tell us?

```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case

1	2	3	4	5	6	7	8

1	2	3	4	6	7	8

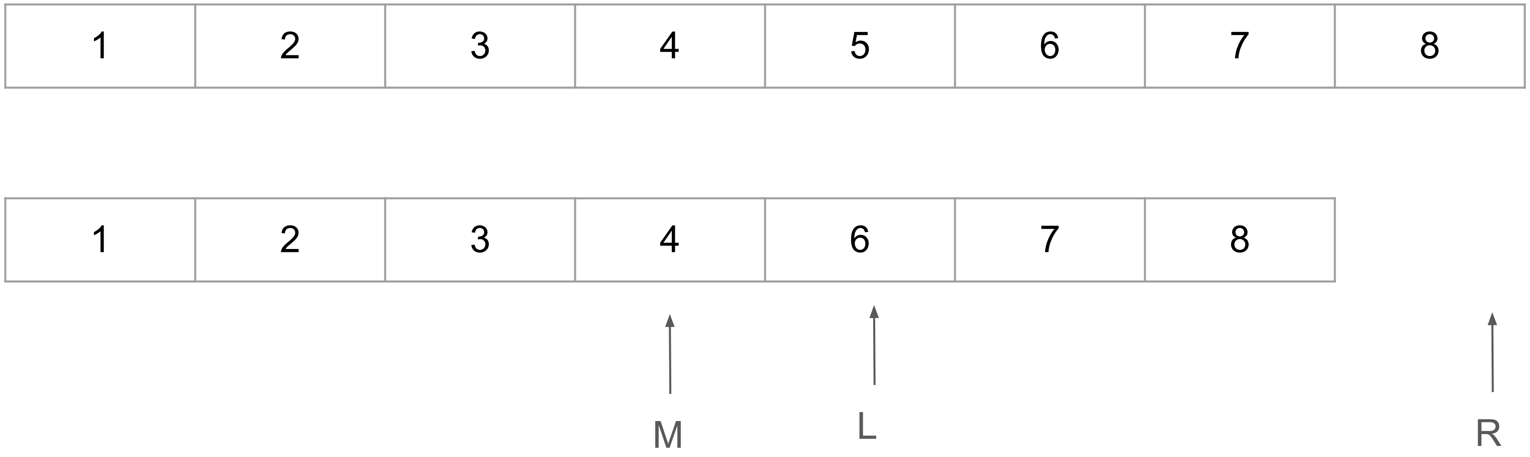
How should we cut the interval?

- **A[m] == B[m] → everything before is equal.**
  - **The missing element must be in the right half!**

```
def binarySearch(A[1:n],B[1:n-1]x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if ???
            return m
        if A[m] = B[m]:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case



How should we cut the interval?

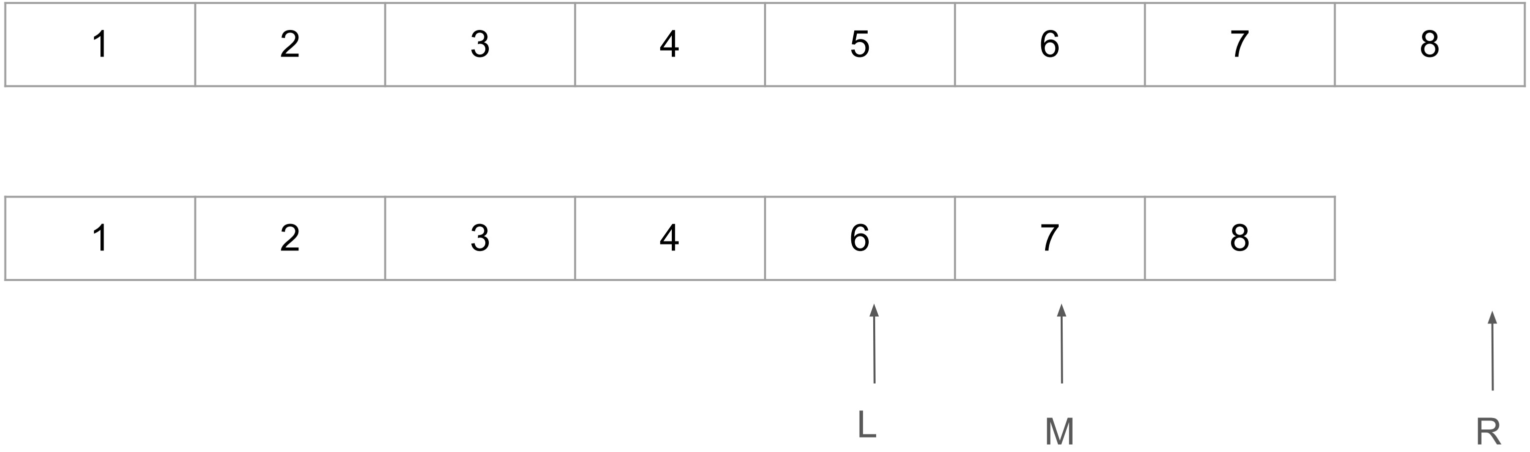
- A[m] == B[m] implies that everything before is equal. The missing element must be in the right half!

Continue running the algorithm to figure out when to stop

```
def binarySearch(A[1:n],B[1:n-1]x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if ???
            return m
        if A[m] == B[m]:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case



How should we cut the interval?

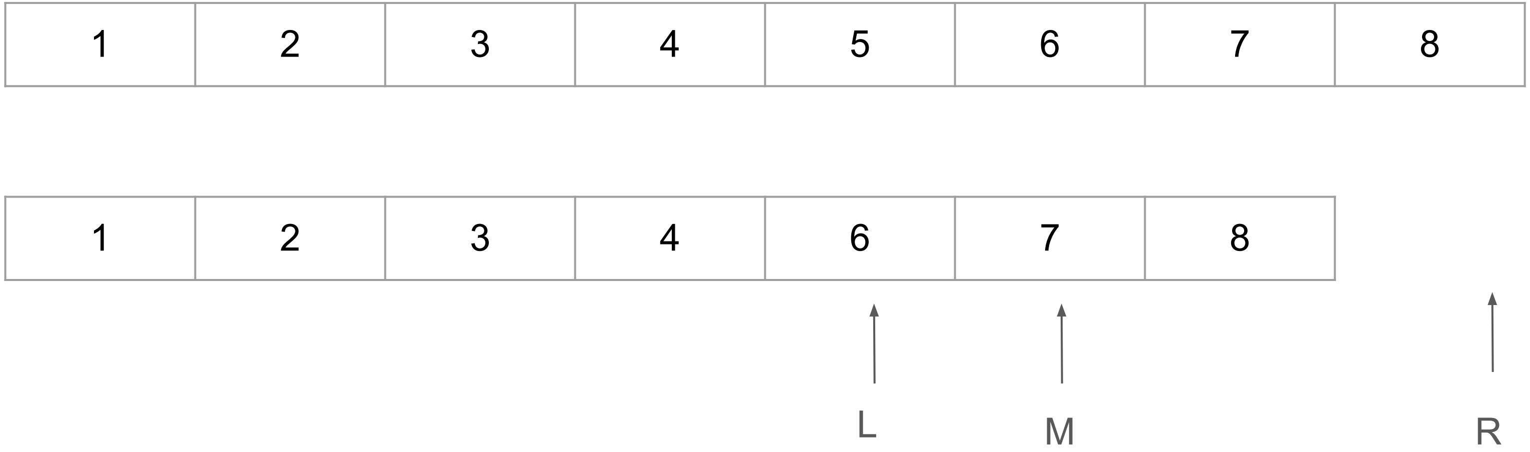
- A[m] == B[m] implies that everything before is equal. The missing element must be in the right half!

Continue running the algorithm to figure out when to stop

```
def binarySearch(A[1:n],B[1:n-1]x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if ???
            return m
        if A[m] = B[m]:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case



How should we cut the interval?

- A[m] == B[m] implies that everything before is equal. The missing element must be in the right half!
- **A[m] != B[m] → something in range [l,m] must be missing.**
  - **The missing element must be in the left half!**

Continue running the algorithm to figure out when to stop

```
def binarySearch(A[1:n],B[1:n-1]x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if ???
            return m
        if A[m] = B[m]:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case

1	2	3	4	5	6	7	8

1	2	3	4	6	7	8

This seems like a good place to stop our algorithm. What's the stop condition?

```
def binarySearch(A[1:n],B[1:n-1]x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if ???
            return m
        if A[m] = B[m]:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Our case

1	2	3	4	5	6	7	8

1	2	3	4	6	7	8

This seems like a good place to stop our algorithm. What's the stop condition?

When l == r

```
def binarySearch(A[1:n],B[1:n-1]x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if l == r:
            return m
        if A[m] = B[m]:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

Last book keeping

Written slightly cleaner since we are guaranteed to have a missing element

```
def binarySearch(A[1:n],B[1:n-1]x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if l == r:
            return m
        if A[m] = B[m]:
            l = m + 1
        else:
            r = m - 1
    return -1
```

1. Search Range
2. Stop condition
3. Interval cutting

```
def binarySearch(A[1:n],B[1:n-1]x):
    l = 1, r = n
    while l < r:
        m = ⌊(l + r) / 2⌋
        if A[m] = B[m]:
            l = m + 1
        else:
            r = m - 1
    return l
```

Question 2

(Quick sort)

(1) Illustrate the operation of the **Partition** step in Quick sort on $A = [2, 8, 7, 1, 3, 5, 6, 4]$.

(2) Can we understand the average-case runtime of Quick sort? What is the best policy for selecting the pivot value in the quick sort?

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

Question 2

(Quick sort)

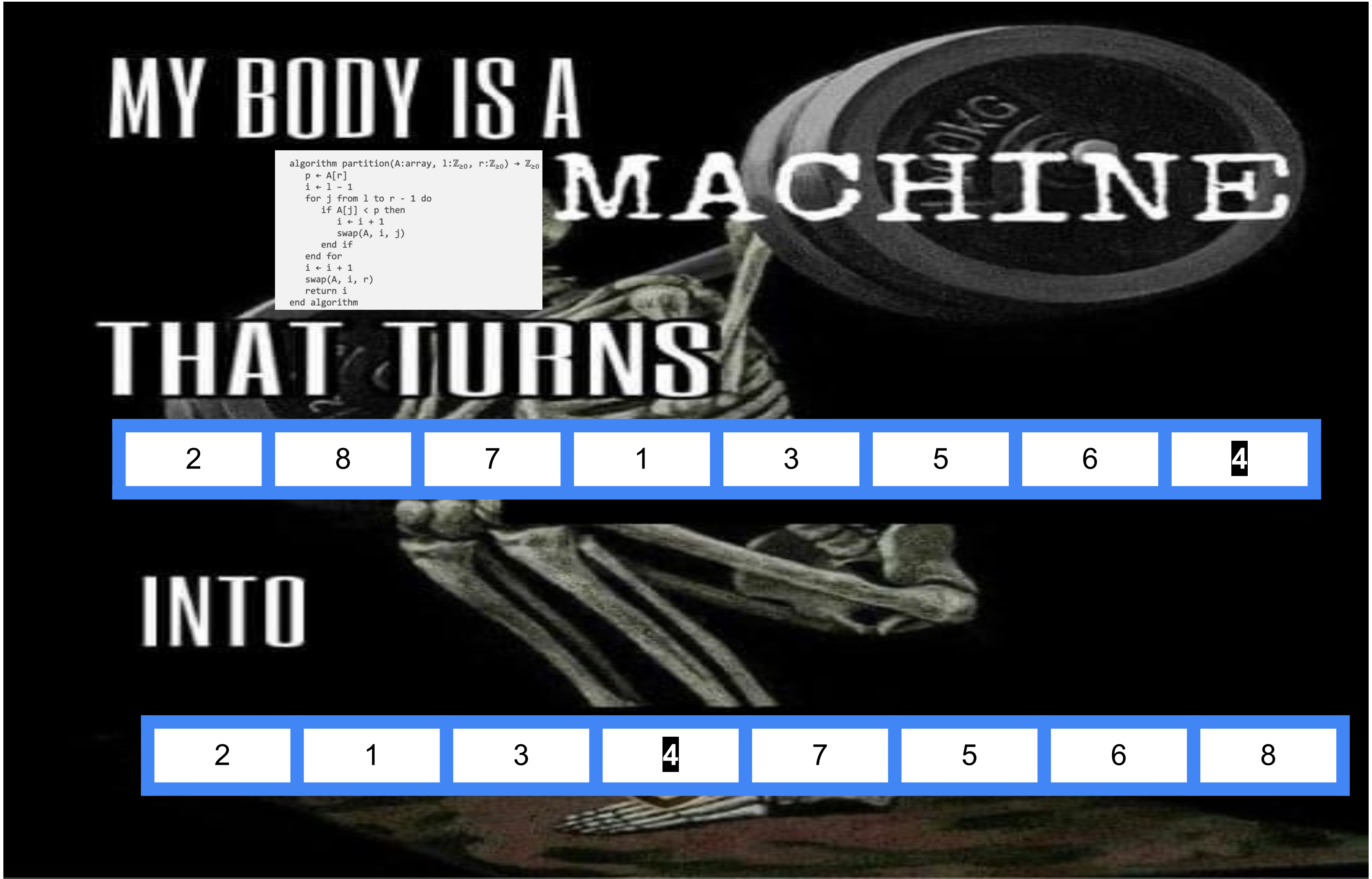
(1) Illustrate the operation of the **Partition** step in Quick sort on $A = [2, 8, 7, 1, 3, 5, 6, 4]$.

(2) Can we understand the average-case runtime of Quick sort? What is the best policy for selecting the pivot value in the quick sort?

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

pivot p is set as last element

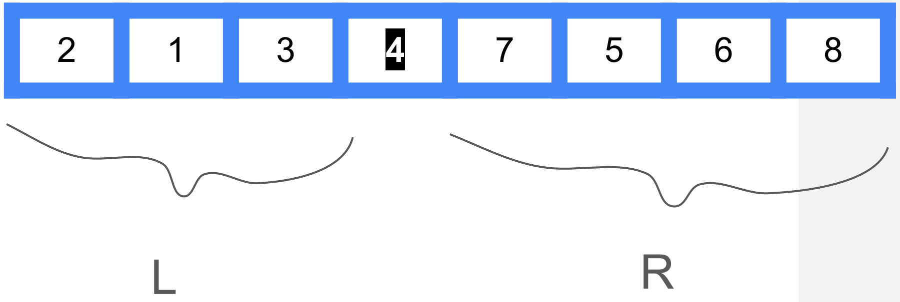
Ok… what does this do tho?


MY BODY IS A
MACHINE
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
p ← A[r]
i ← l - 1
for j from l to r - 1 do
if A[j] < p then
i ← i + 1
swap(A, i, j)
end if
end for
i ← i + 1
swap(A, i, r)
return i
end algorithm
THAT TURNS
2 8 7 1 3 5 6 4
INTO
2 1 3 4 7 5 6 8

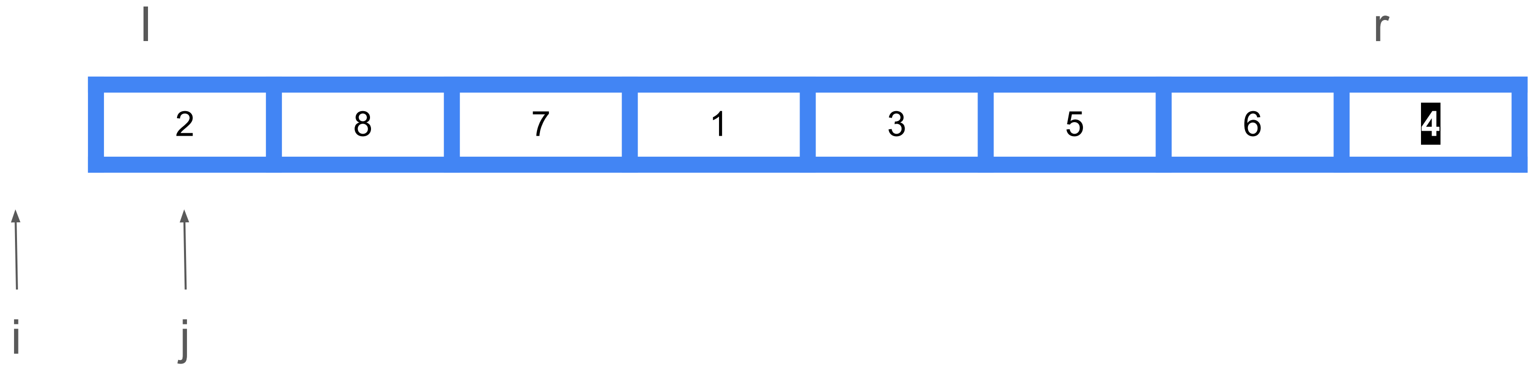
p: pivot

j: goes through entire array

i : growing index of L



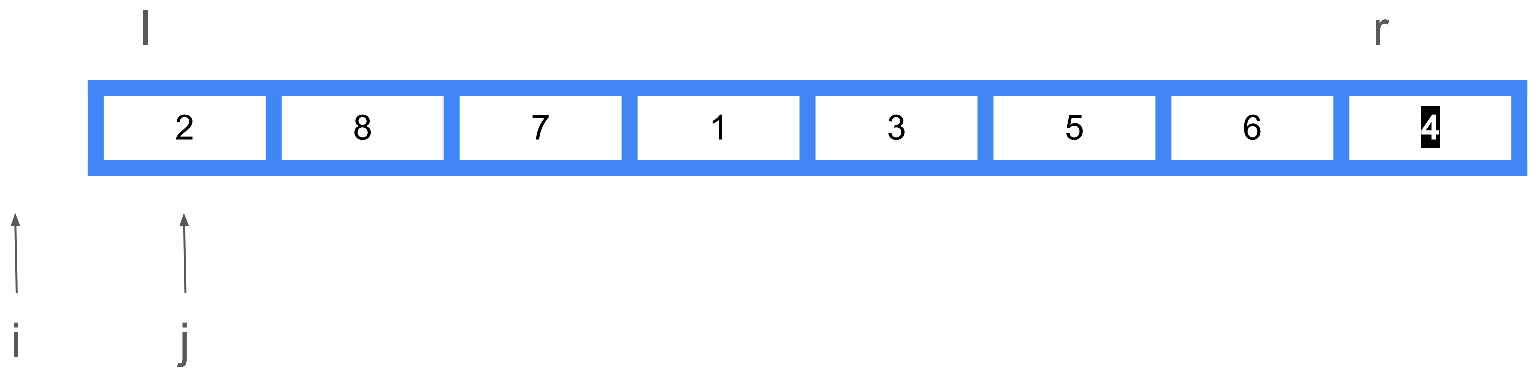

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```



Pivot = 4

Start of the algorithm

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```



Pivot = 4

$A[j] = 2 < 4$

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

l r

2	8	7	1	3	5	6	4

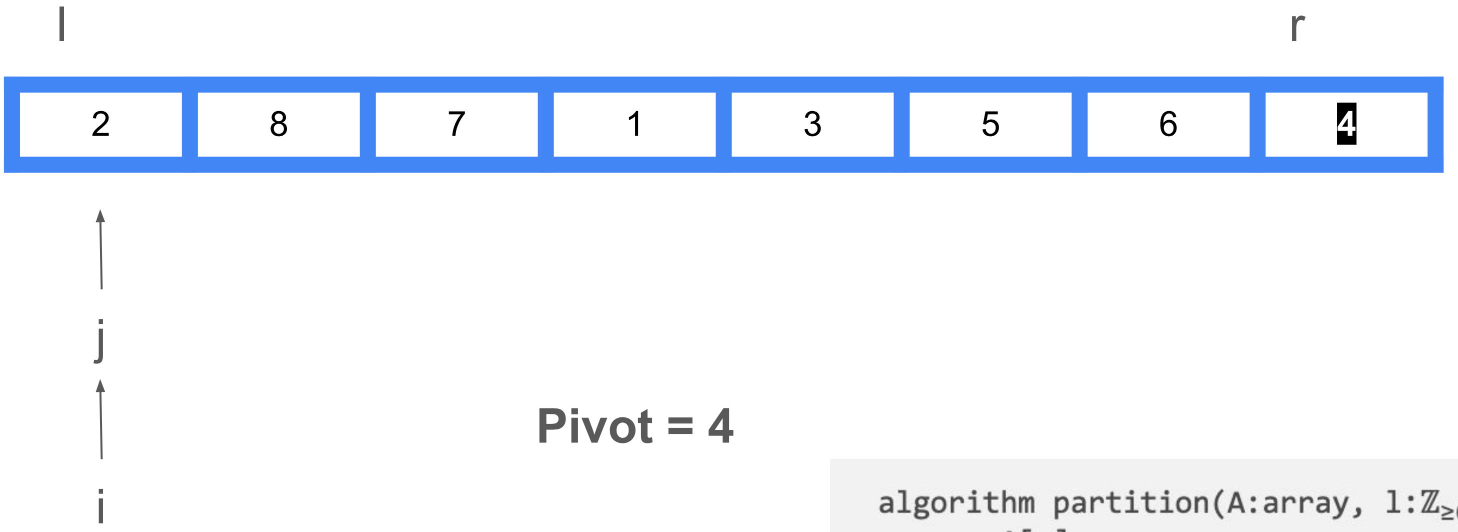
j

i

Pivot = 4

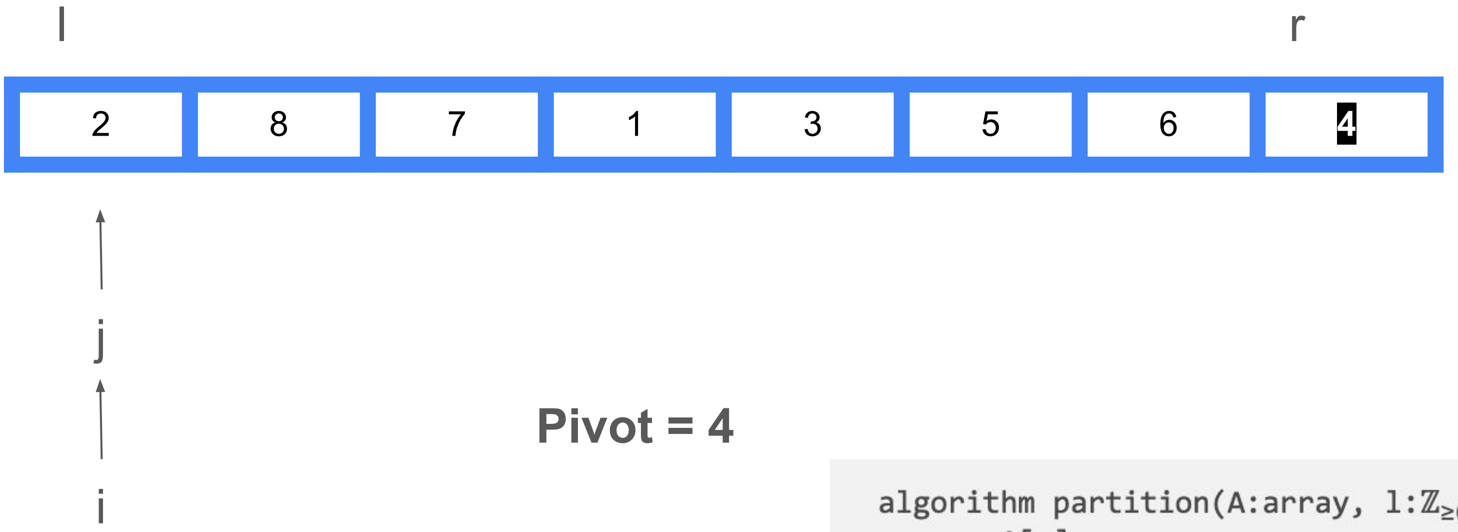
A[j] = 2 < 4

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



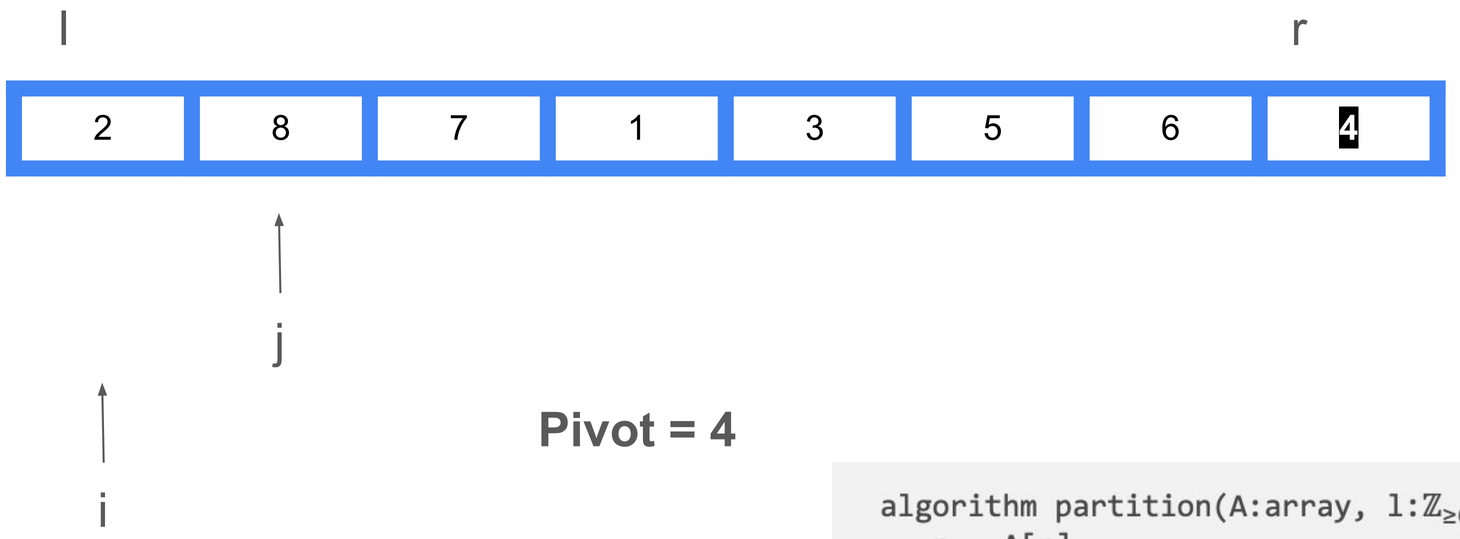
Pivot = 4

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```



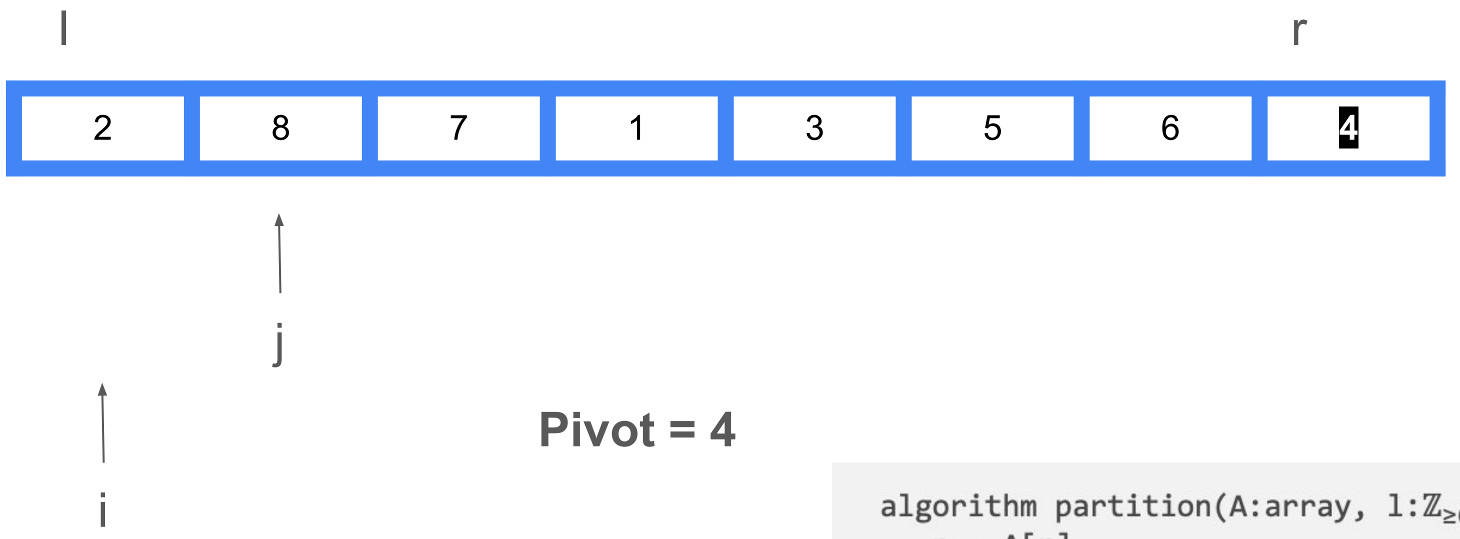
Pivot = 4

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```



Pivot = 4

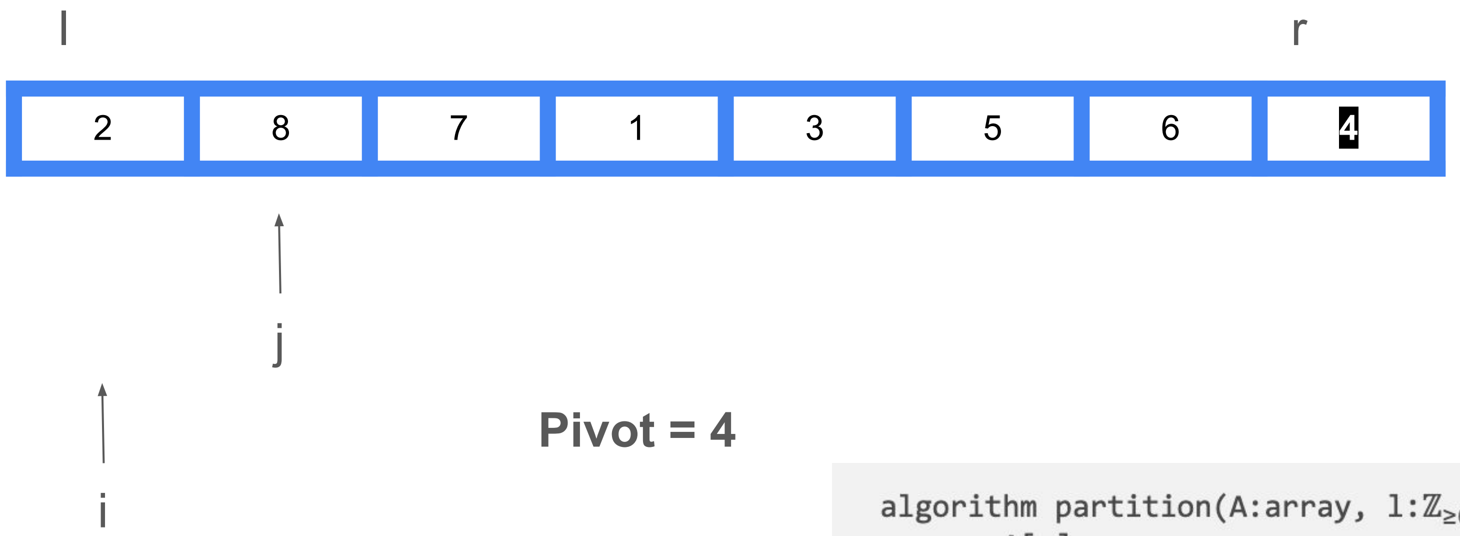
```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



Pivot = 4

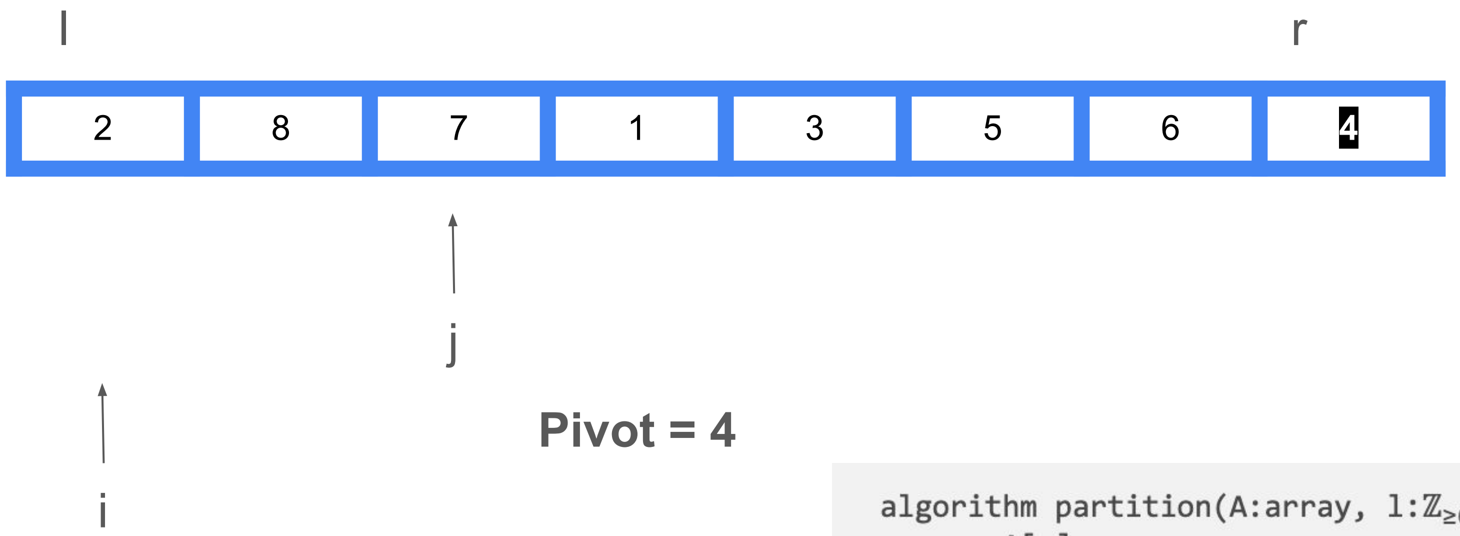
A[j] = 8 < 4? No

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

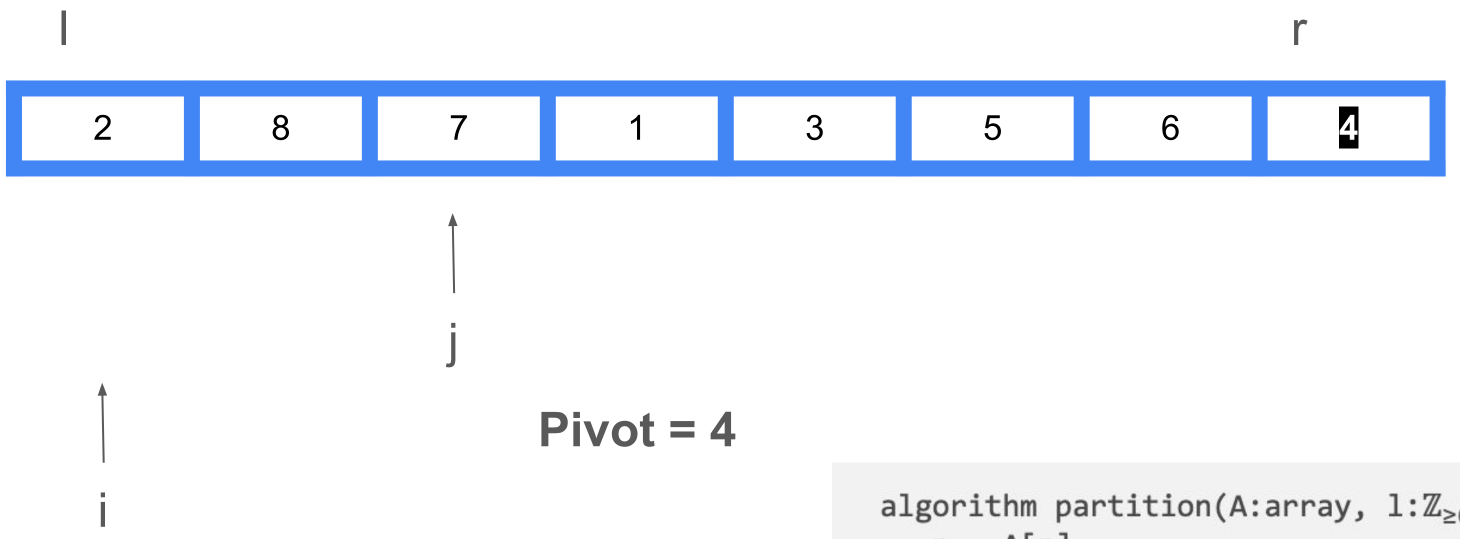


Pivot = 4

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



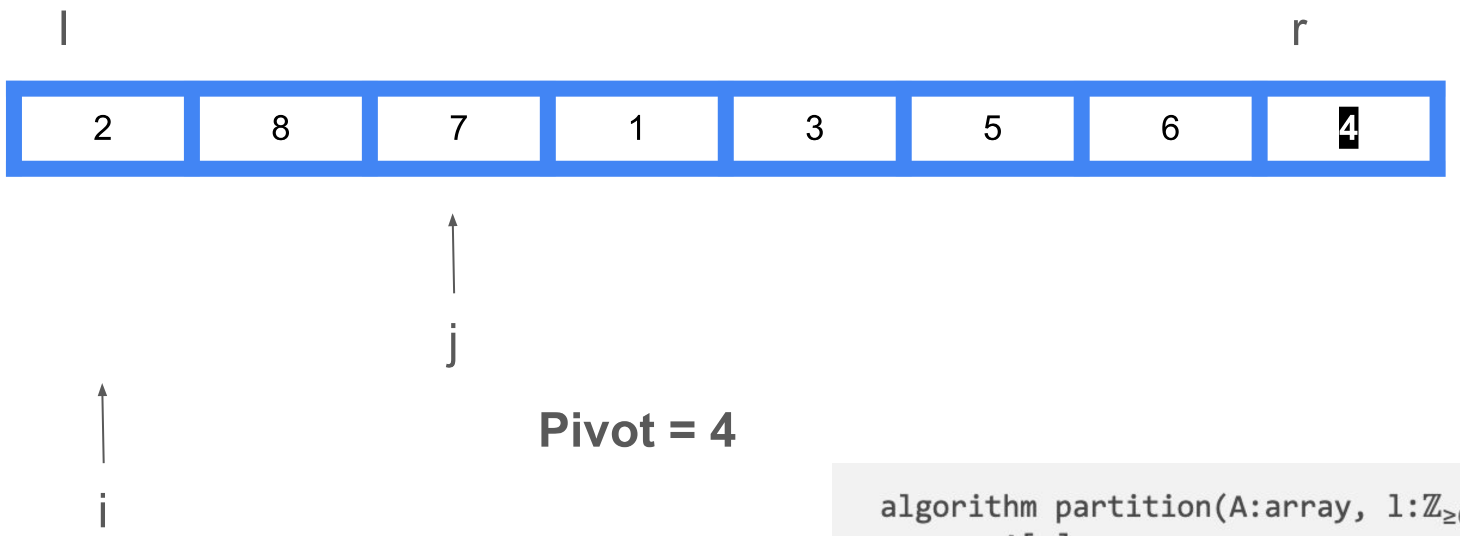

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



Pivot = 4

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

A[j] < 4? Nope



Pivot = 4

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

l r

2	8	7	1	3	5	6	4

j

i

Pivot = 4

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
  p ← A[r]
  i ← l - 1
  for j from l to r - 1 do
    if A[j] < p then
      i ← i + 1
      swap(A, i, j)
    end if
  end for
  i ← i + 1
  swap(A, i, r)
  return i
end algorithm
```

l r

2	8	7	1	3	5	6	4

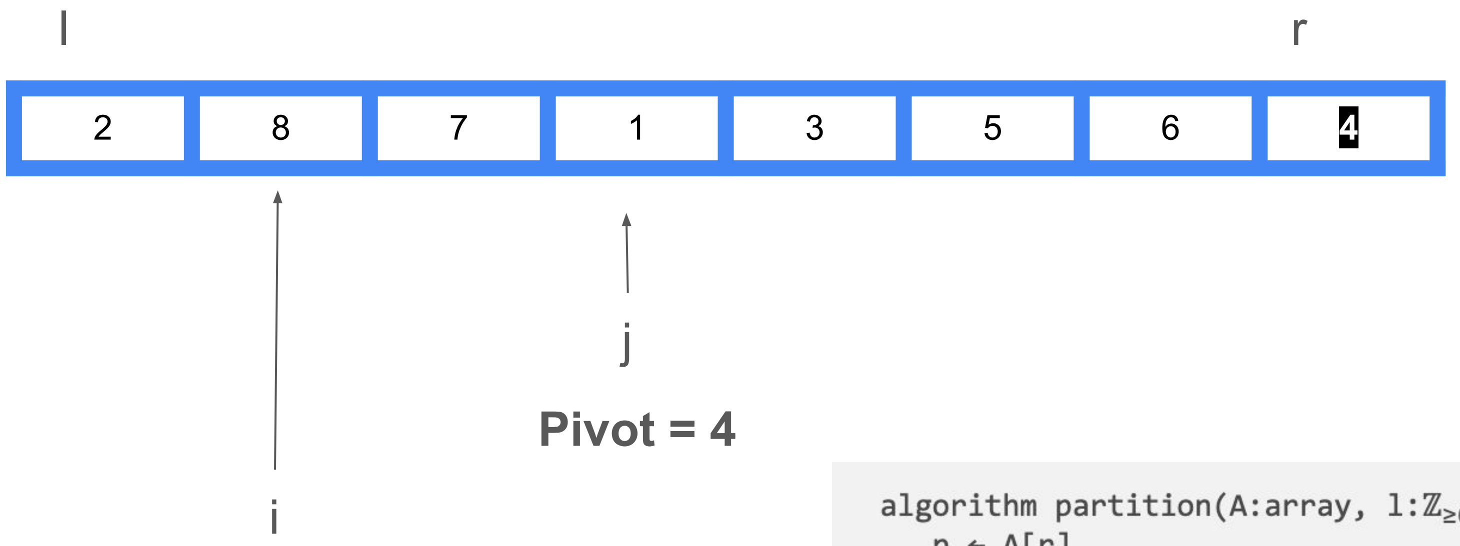
j

Pivot = 4

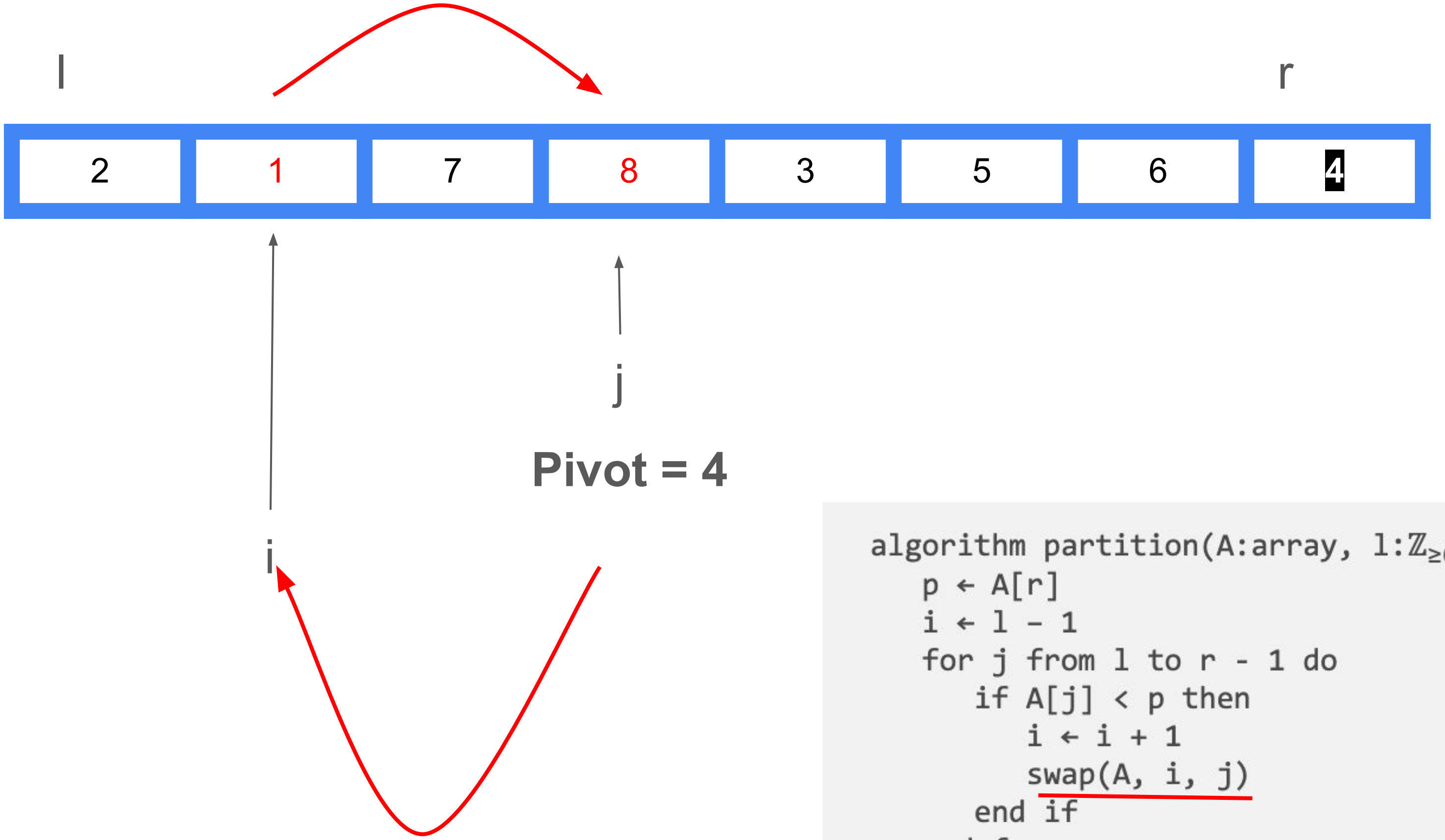
i

A[j] = 1 < 4

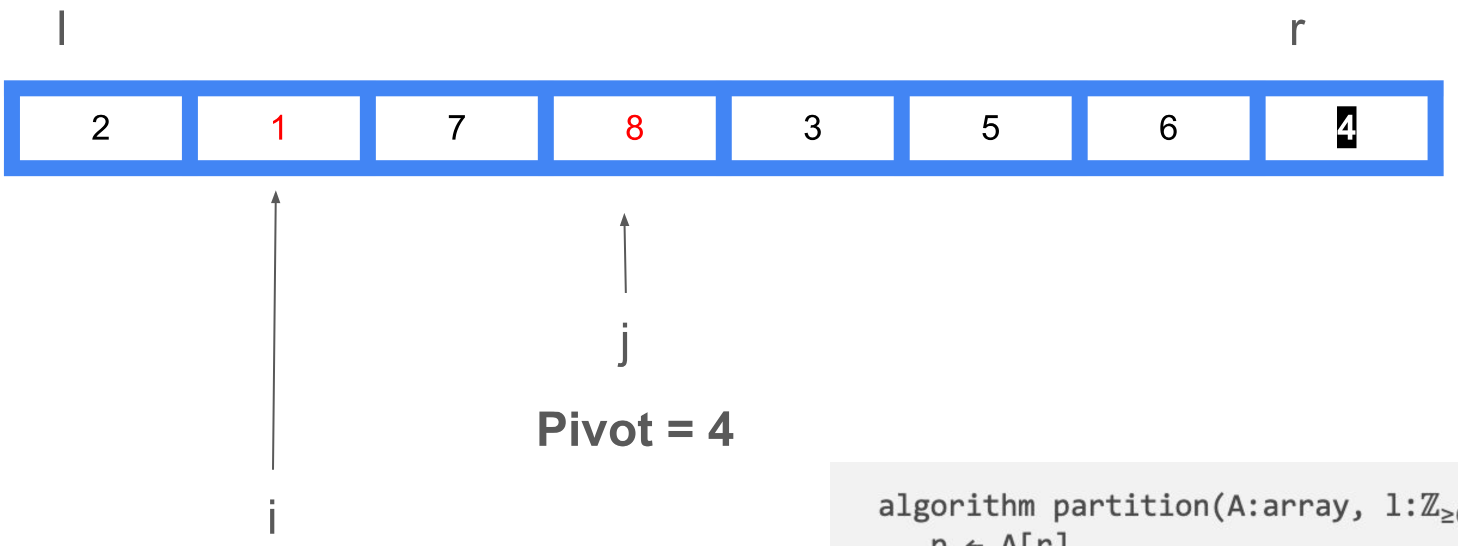
```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```




```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



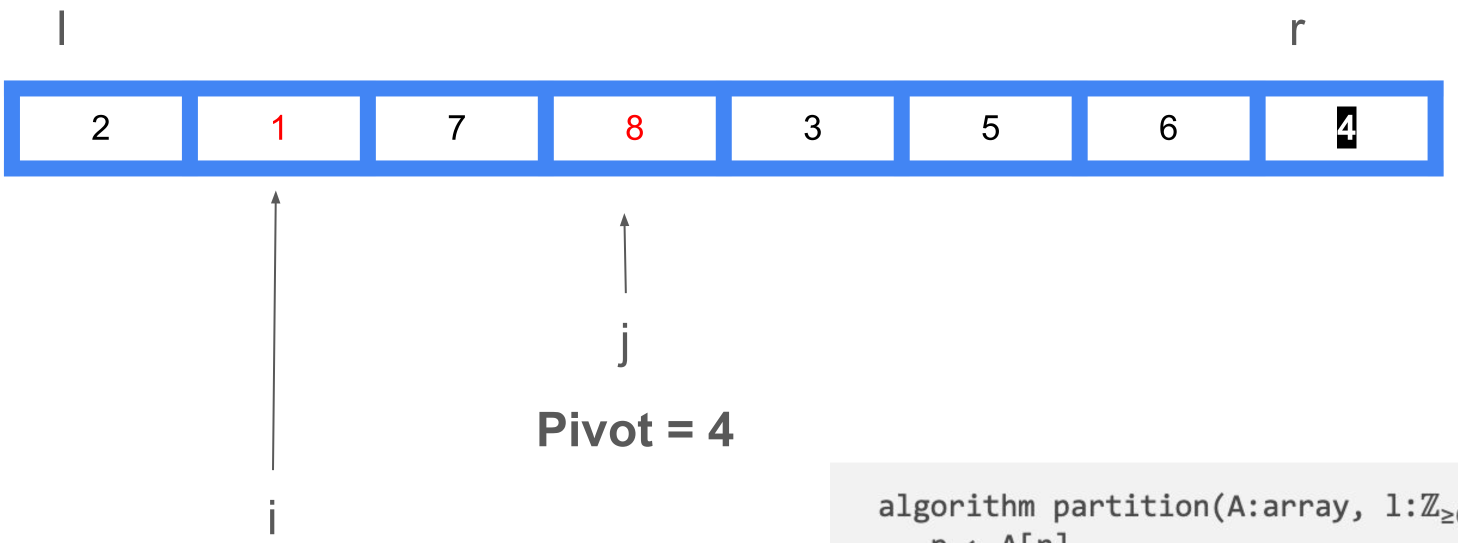

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



Pivot = 4

We’ve done two swaps now. Insight?

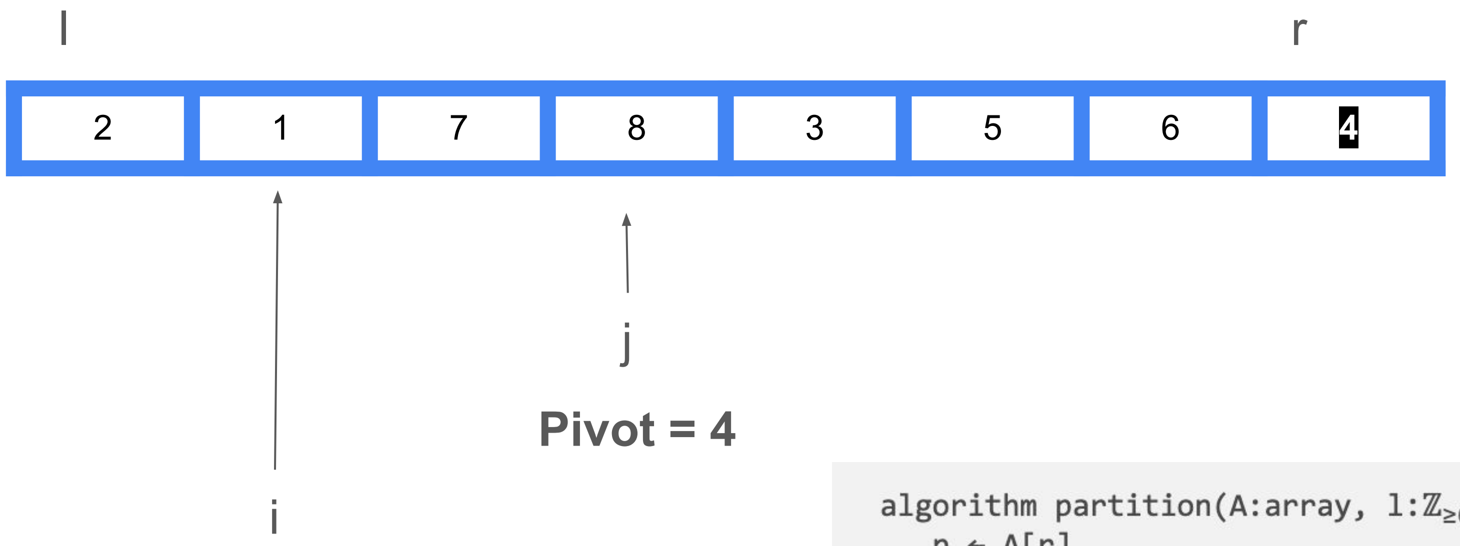
```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```



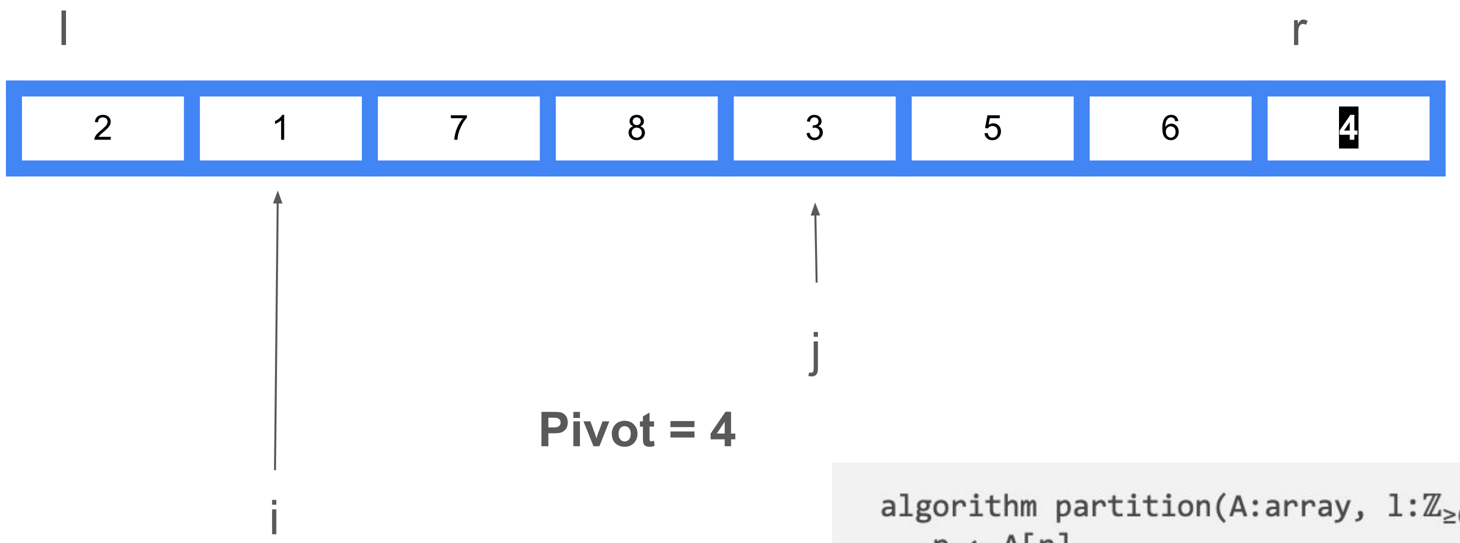

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

We’ve done two swaps now. Insight?

Everything up to i is less than the pivot

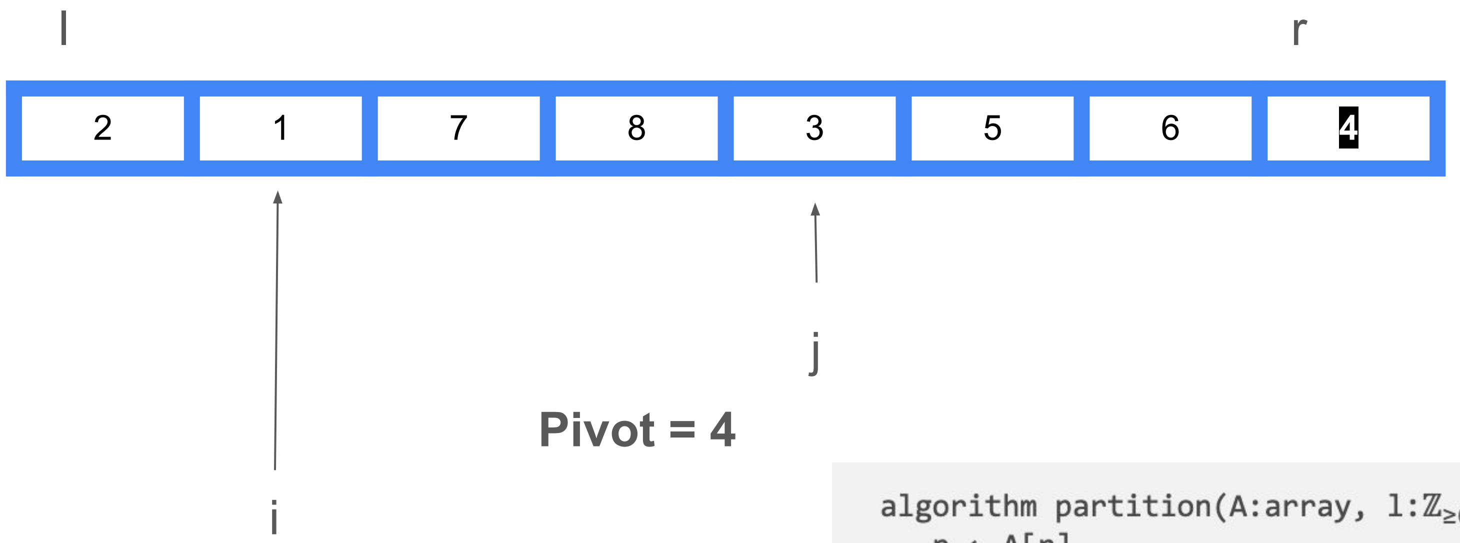



```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```



Pivot = 4

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



Pivot = 4

A[j] = 3 < 4, swap!

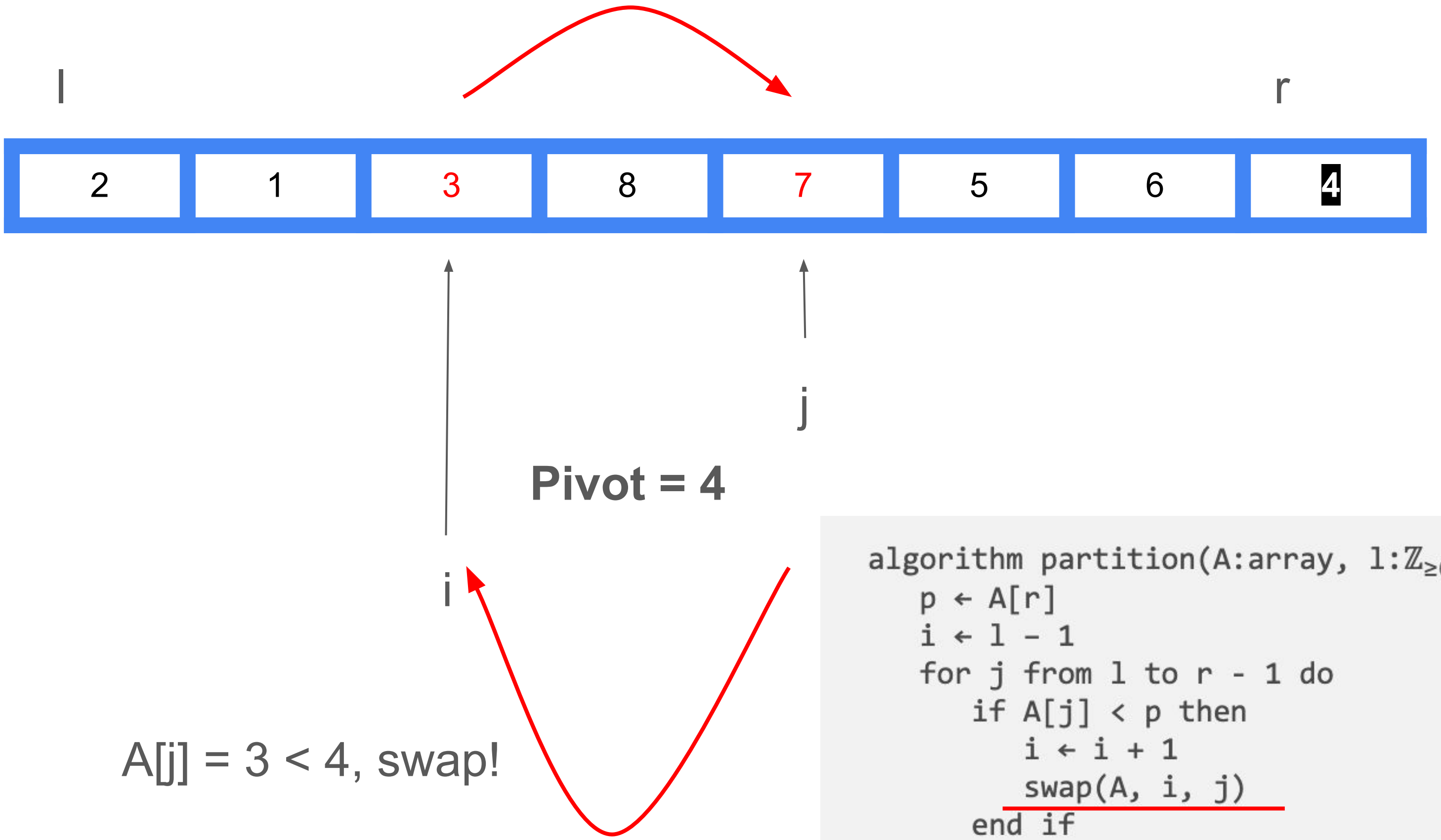
```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

l							r
2	1	7	8	3	5	6	4
		i		j			

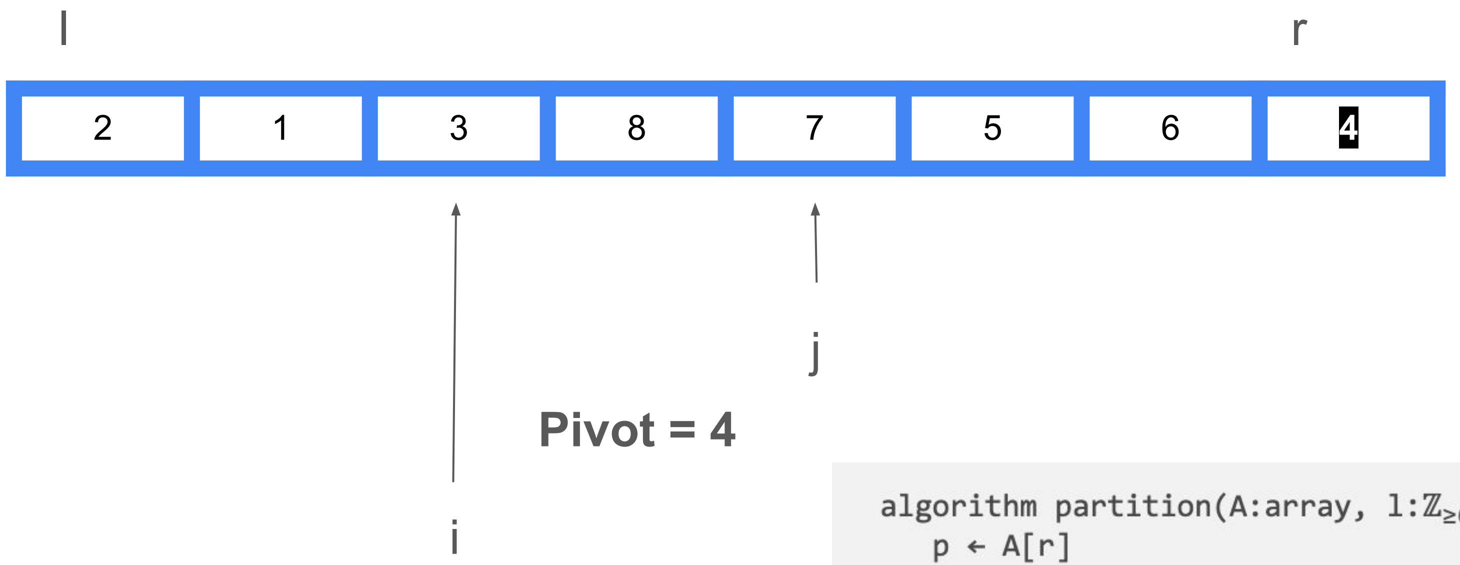
Pivot = 4

A[j] = 3 < 4, swap!

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```



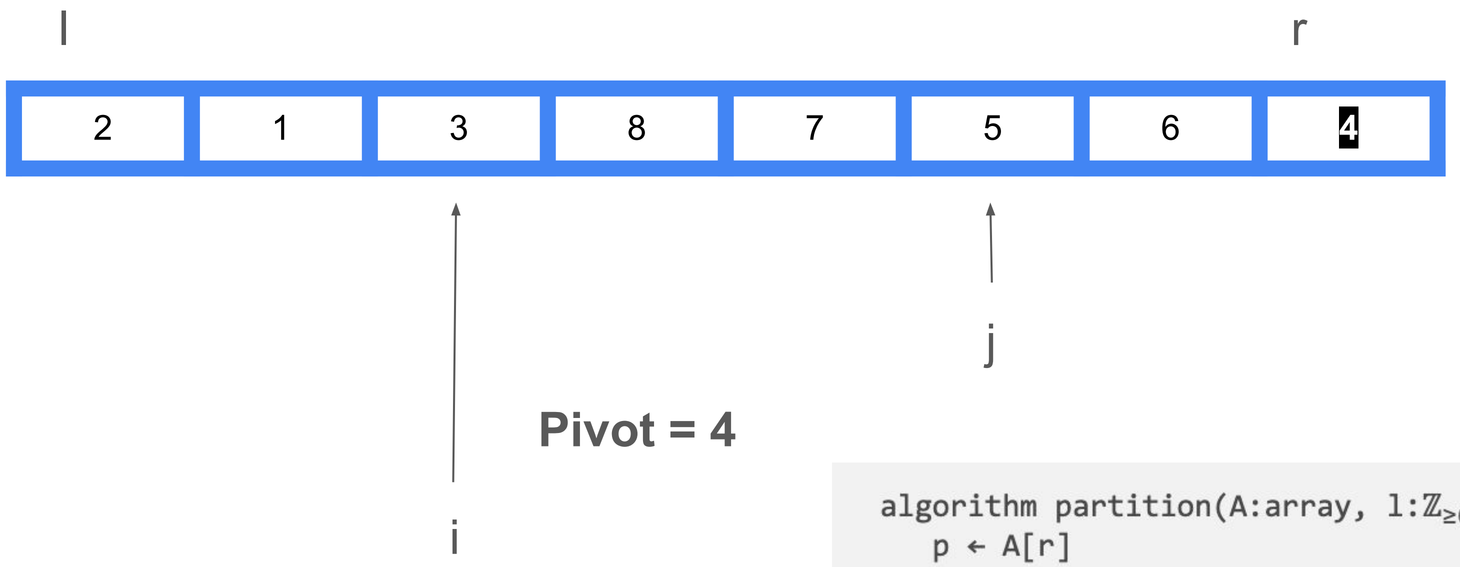

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



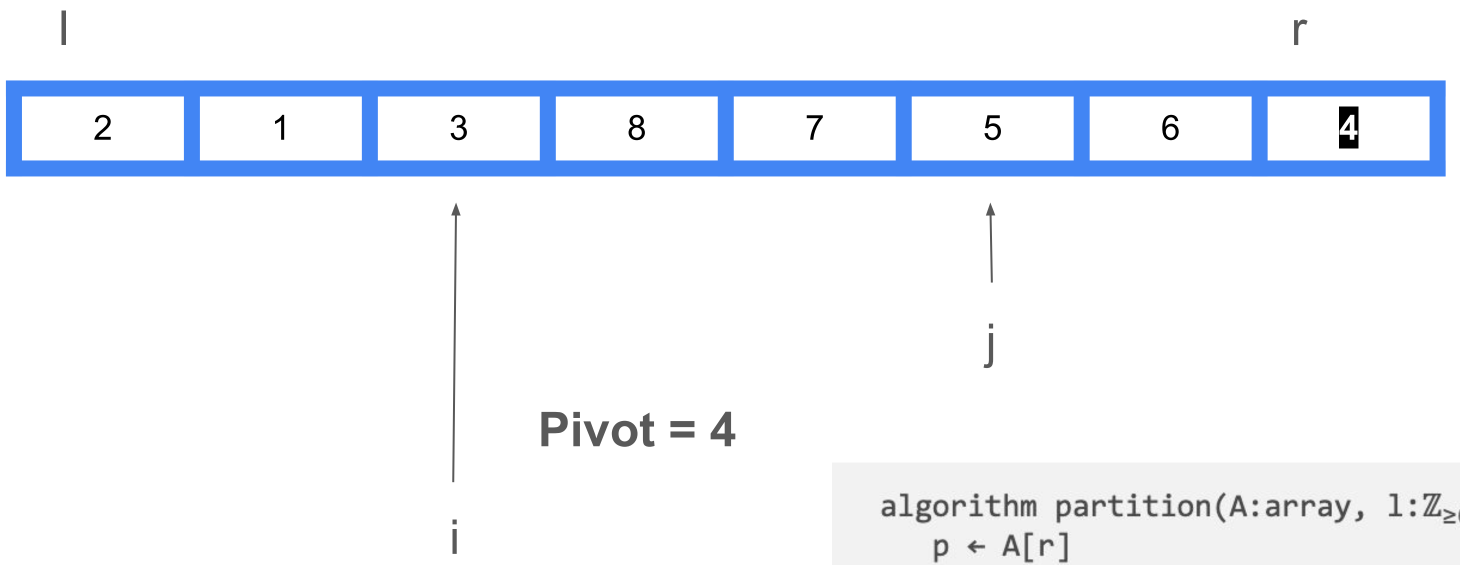
Pivot = 4

Everything up to i is less than the pivot

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

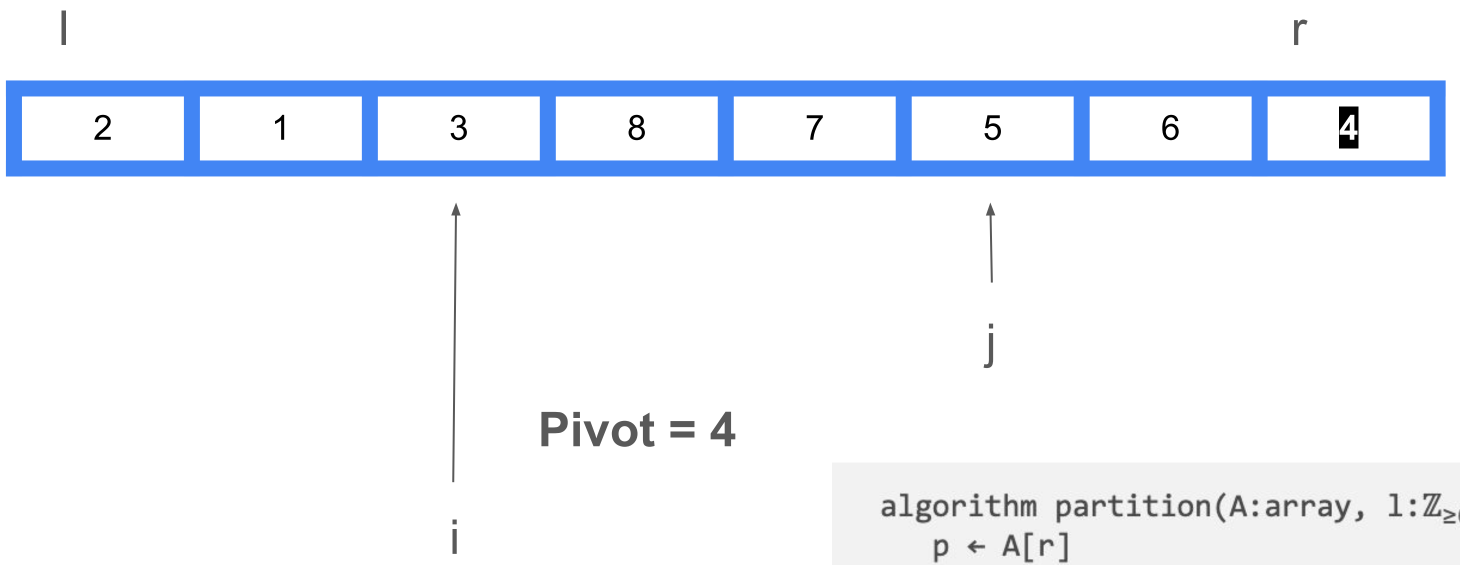



```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



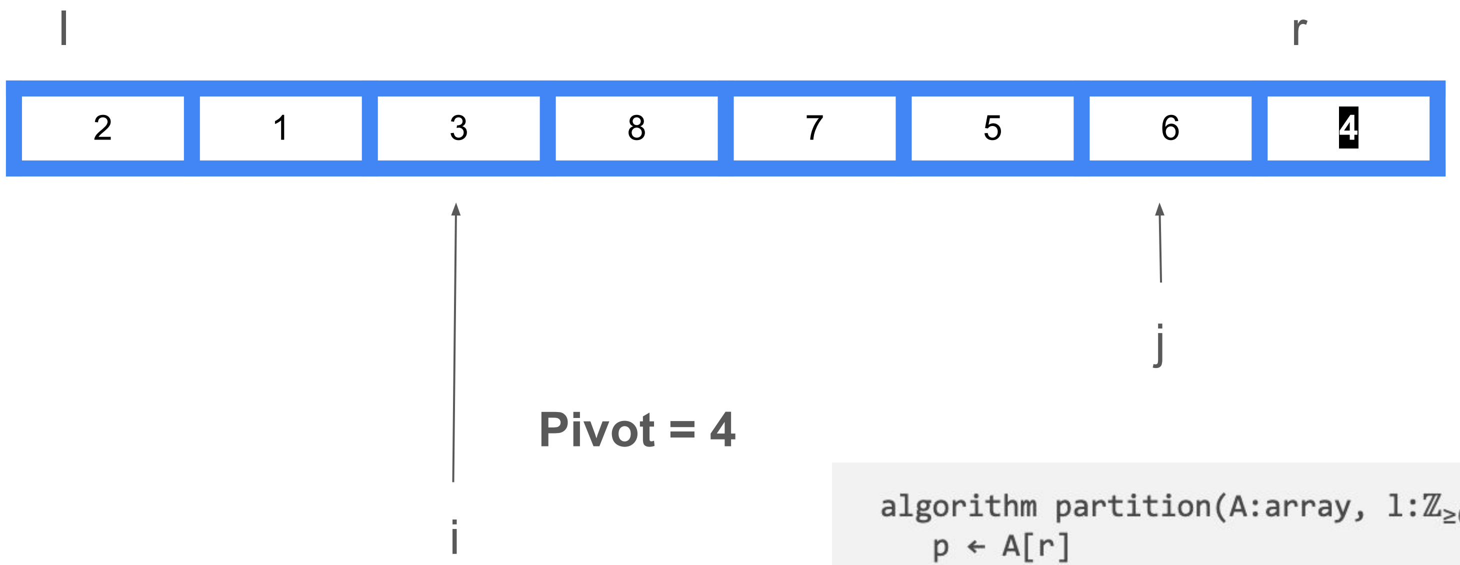
A[j] = 5 < 4? Nah

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```

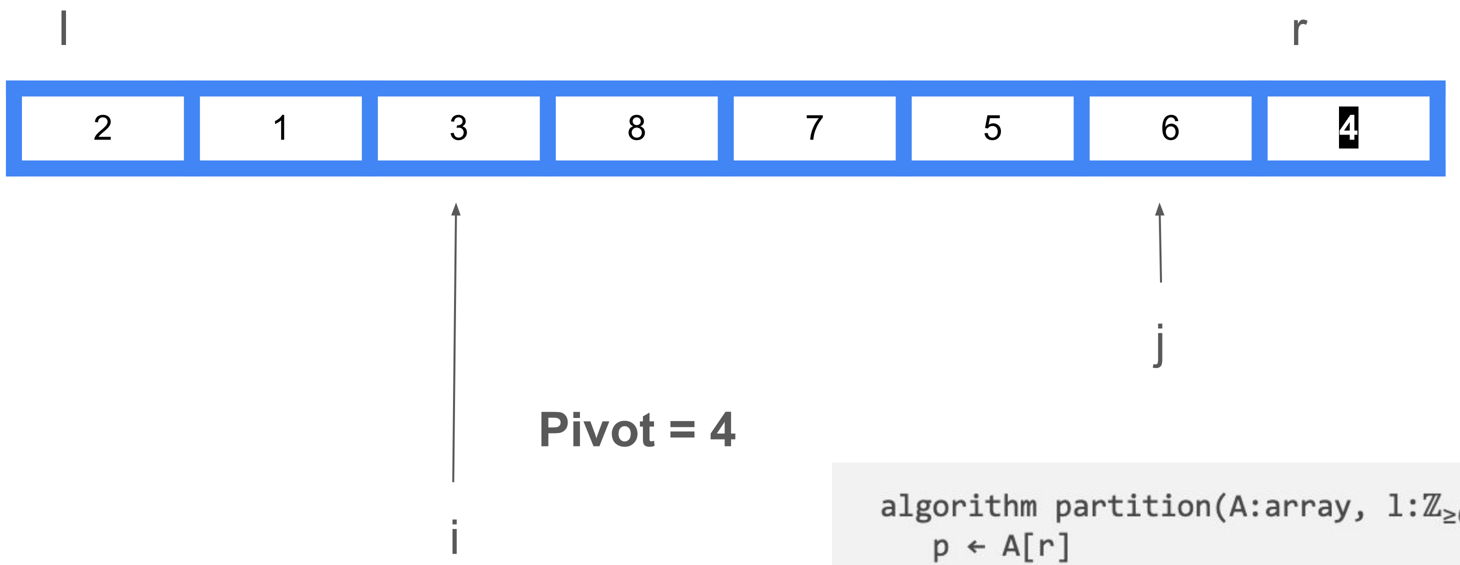


Pivot = 4

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

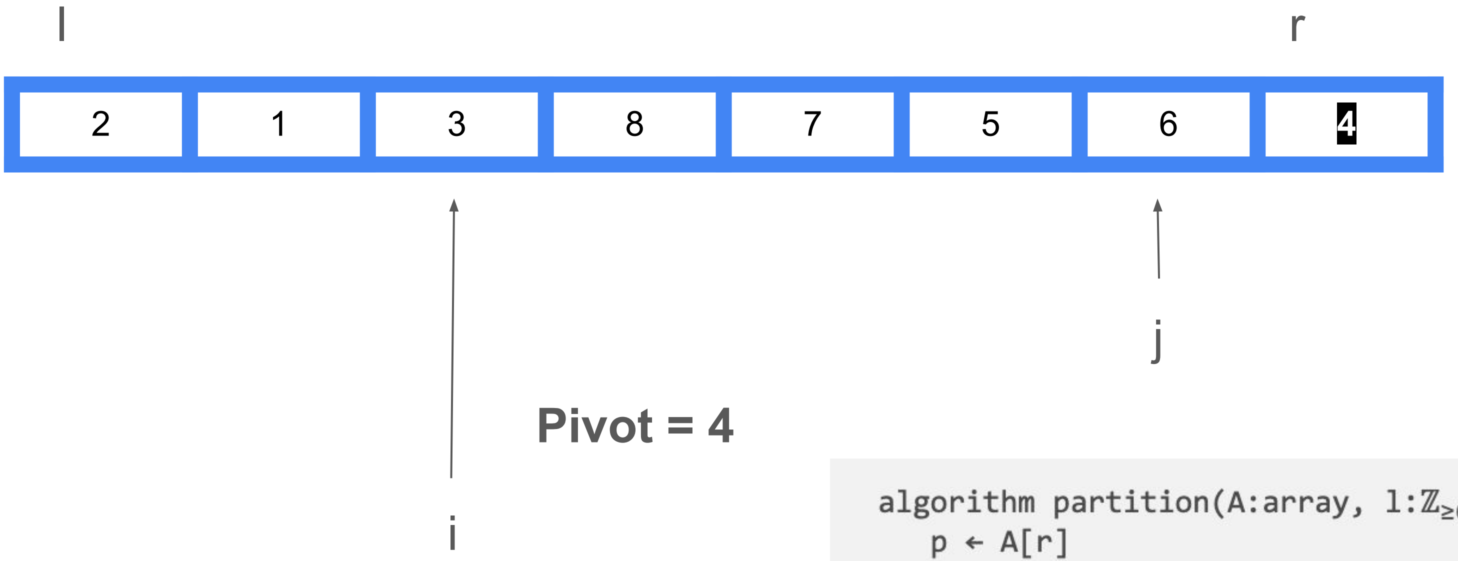



```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



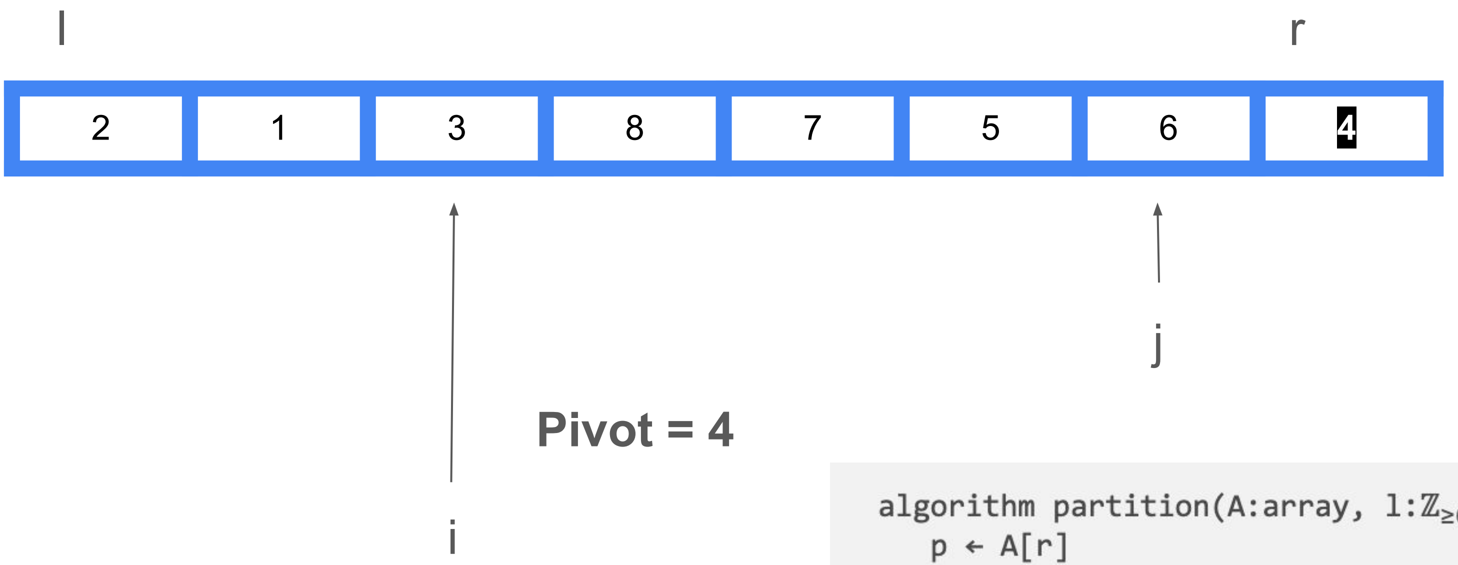
Pivot = 4

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```



Pivot = 4

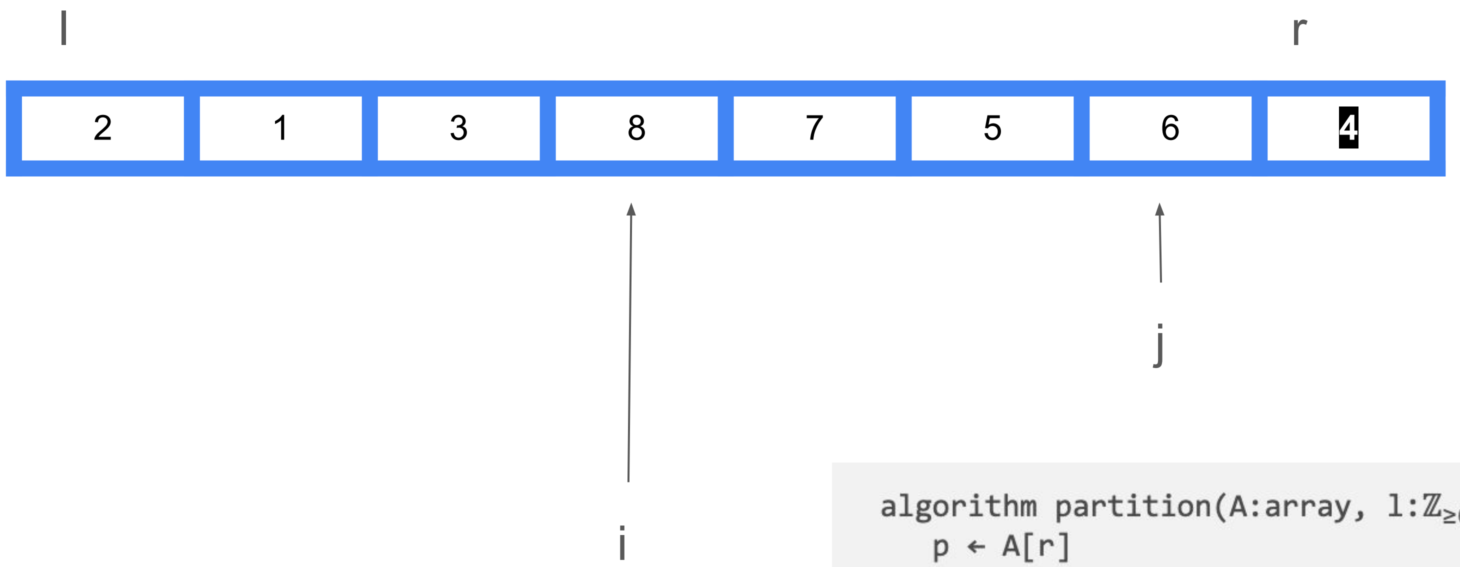
```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```



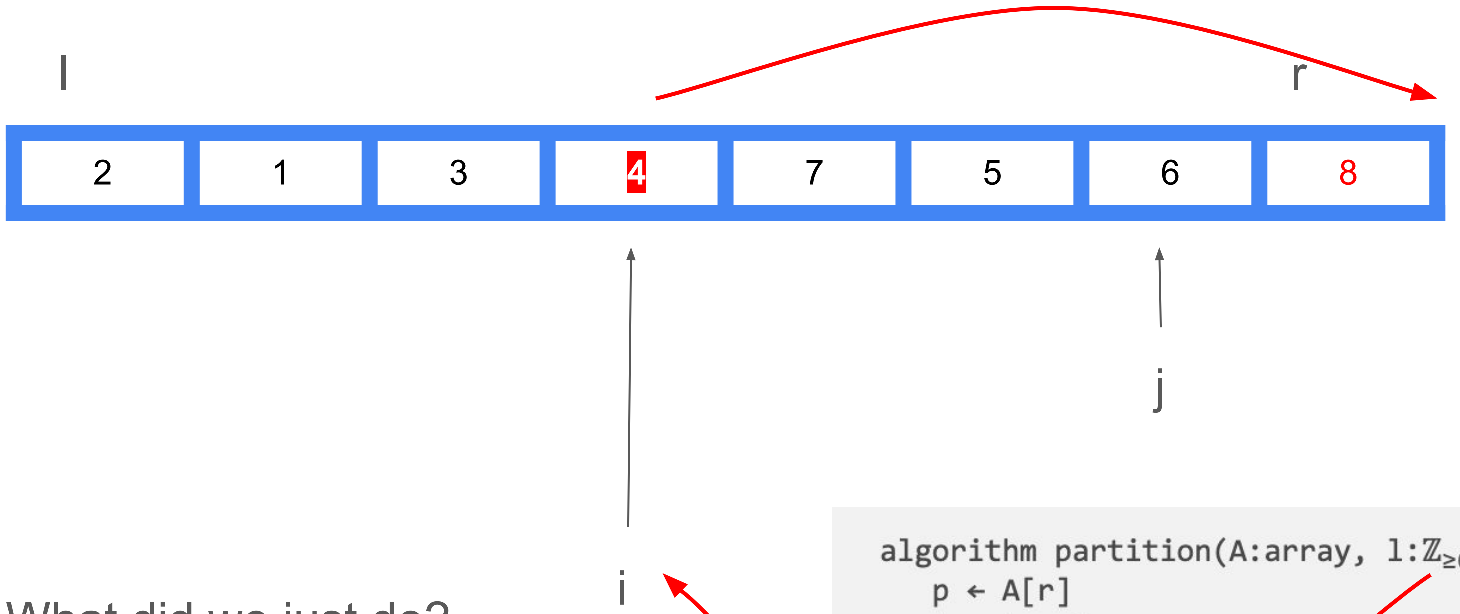
Pivot = 4

That was the last for loop iteration

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```

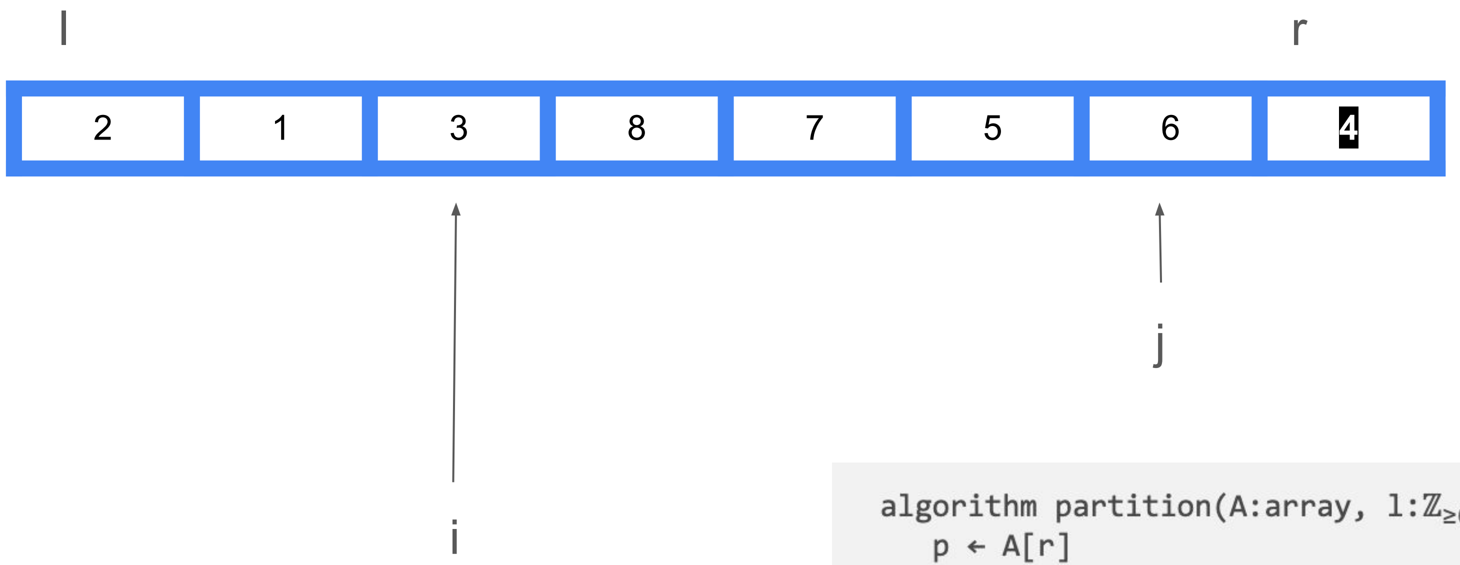



```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
  p ← A[r]
  i ← l - 1
  for j from l to r - 1 do
    if A[j] < p then
      i ← i + 1
      swap(A, i, j)
    end if
  end for
  i ← i + 1
  swap(A, i, r)
  return i
end algorithm
```



What did we just do?
Lets rewind a little

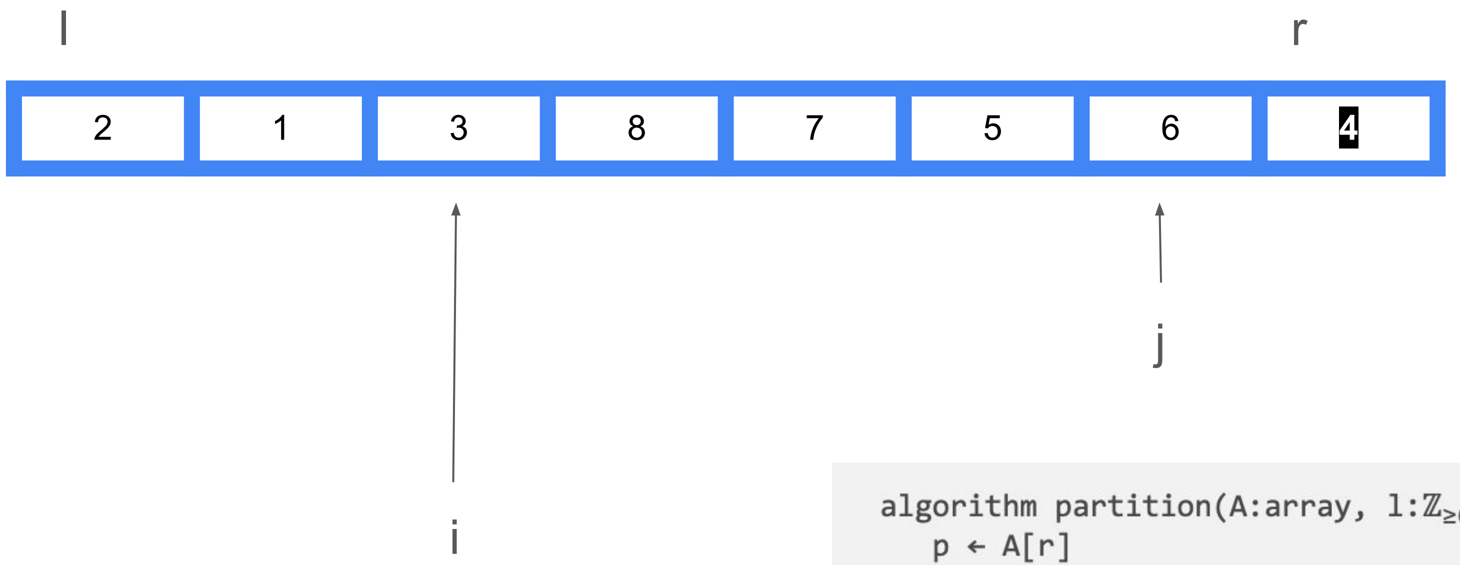
```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```



That was the last for loop iteration

Everything up to i is less than the pivot

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return i
end algorithm
```

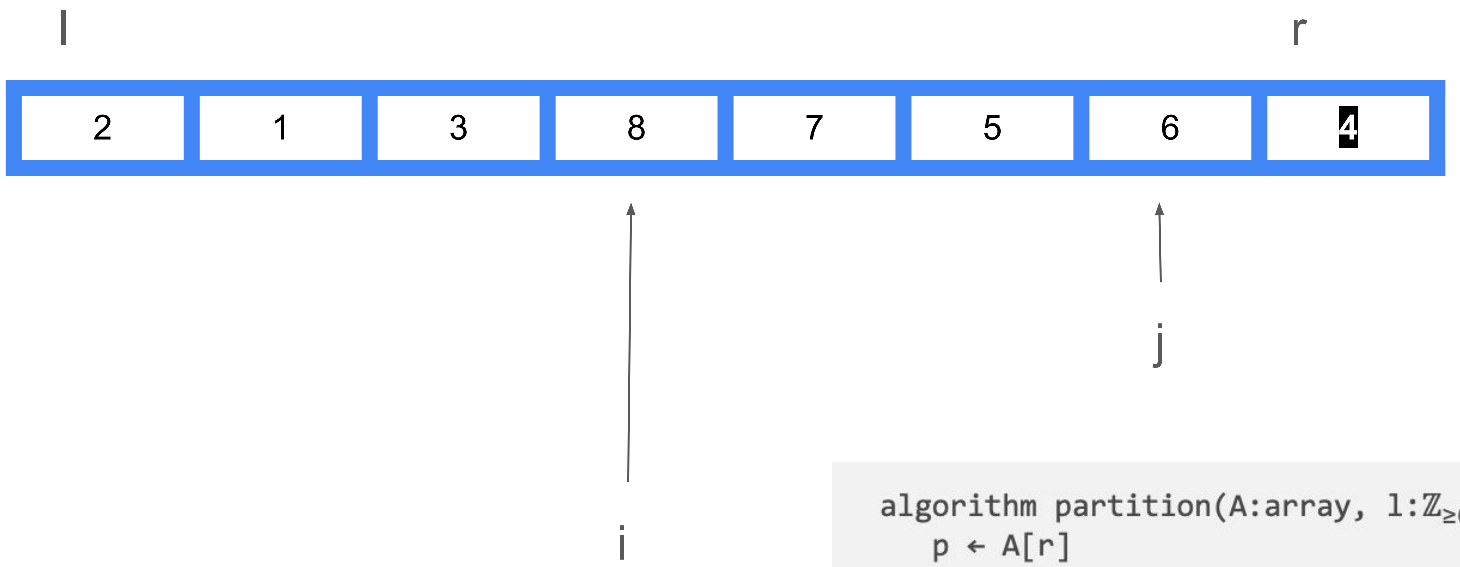


That was the last for loop iteration

Everything up to i is less than the pivot

pivot index = i + 1

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

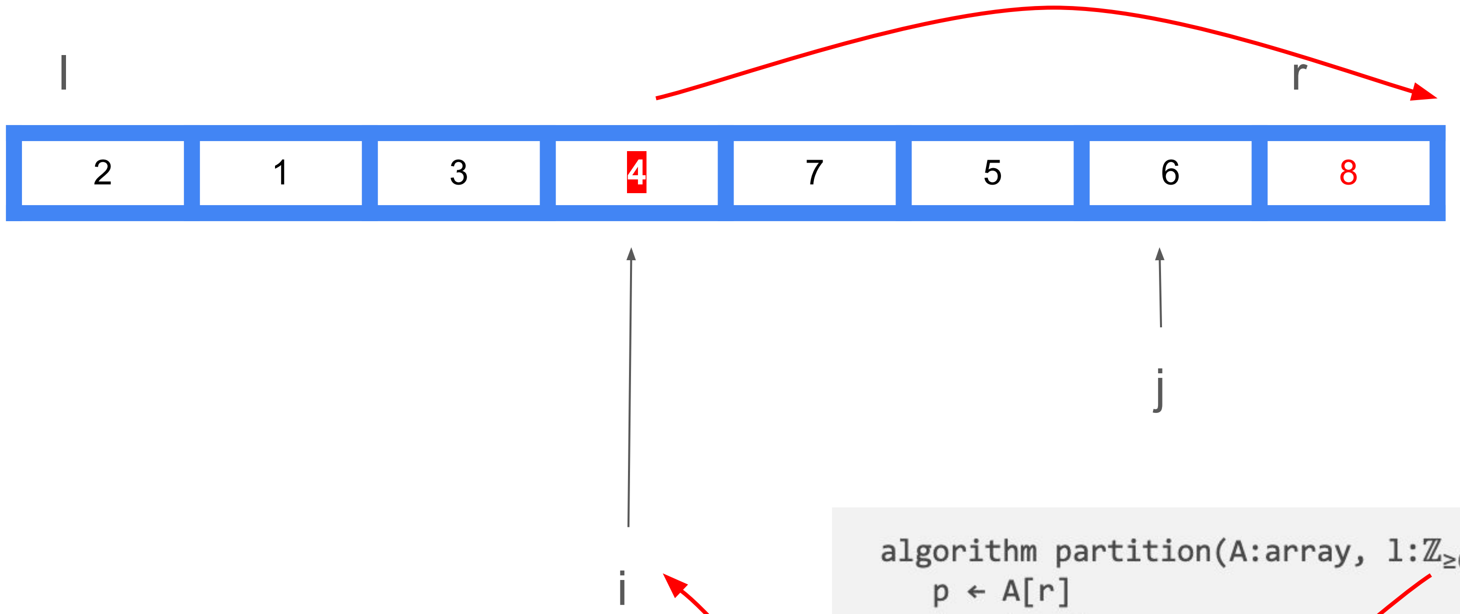



```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

That was the last for loop iteration

Everything up to i is less than the pivot

pivot index = i + 1




```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

That was the last for loop iteration

Everything up to i is less than the pivot

pivot index = i + 1

Question 2

(Quick sort)

(1) Illustrate the operation of the **Partition** step in Quick sort on $A = [2, 8, 7, 1, 3, 5, 6, 4]$.

(2) Can we understand the average-case runtime of Quick sort? What is the best policy for selecting the pivot value in the quick sort?

Question 2

(Quick sort)

(1) Illustrate the operation of the **Partition** step in Quick sort on $A = [2, 8, 7, 1, 3, 5, 6, 4]$.

(2) Can we understand the average-case runtime of Quick sort? What is the best policy for selecting the pivot value in the quick sort?

(2) We learned in lecture that the best-case runtime is $O(n \log n)$ and the worst-case runtime is $O(n^2)$. There is no optimal solutions for selecting a pivot. Ideally we want to select the median one, but we can't guarantee this. However, the average-case running time of Quick sort is much closer to the best case than to the worst case. Hence, Quick sort is usually good and randomized Quick sort is good with high probability. The following example provides an analyzable situation.

Question 2

(Quick sort)

(1) Illustrate the operation of the **Partition** step in Quick sort on $A = [2, 8, 7, 1, 3, 5, 6, 4]$.

(2) Can we understand the average-case runtime of Quick sort? What is the best policy for selecting the pivot value in the quick sort?

(2) We learned in lecture that the best-case runtime is $O(n \log n)$ and the worst-case runtime is $O(n^2)$. There is no optimal solutions for selecting a pivot. Ideally we want to select the median one, but we can't guarantee this. However, the average-case running time of Quick sort is much closer to the best case than to the worst case. Hence, Quick sort is usually good and randomized Quick sort is good with high probability. The following example provides an analyzable situation.

Average case of quick sort is close to O(nlogn)

Suppose at each partition, we can guarantee a 999-to-1 split

How does our recurrence cost look like?

T(n) =

Average case of quick sort is close to O(nlogn)

Suppose at each partition, we can guarantee a 999-to-1 split

How does our recurrence cost look like?

$$T(n) = T(n / 1000) + T(999n / 1000) + cn$$

How about the tree?

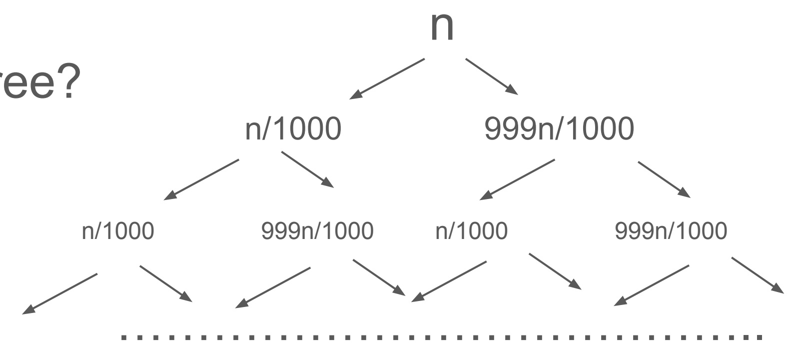
Average case of quick sort is close to O(nlogn)

Suppose at each partition, we can guarantee a 999-to-1 split

How does our recurrence cost look like?

$$T(n) = T(n / 1000) + T(999n / 1000) + cn$$

How about the tree?



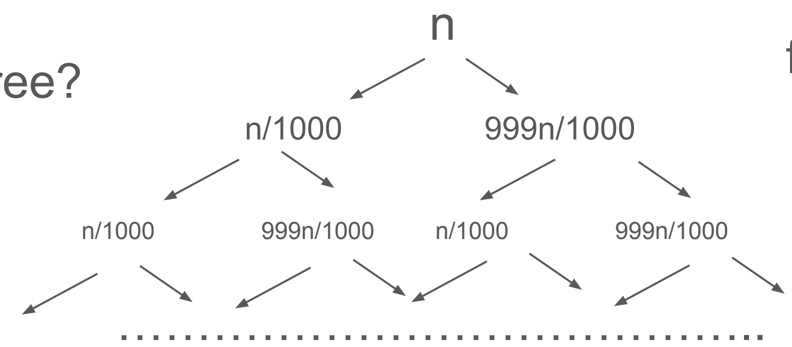
Average case of quick sort is close to O(nlogn)

Suppose at each partition, we can guarantee a 999-to-1 split

How does our recurrence cost look like?

$$T(n) = T(n / 1000) + T(999n / 1000) + cn$$

How about the tree?



Takeaway: any constant fraction split is O(n log n)