PSO 5

Sort

Slides now posted right before class :)

justin-zhang.com/teaching/cs251

Announcements

- Hw 4 out! Due today.
- CS Graduate Symposium – I’m presenting!
  - https://www.cs.purdue.edu/gsa/symposium.html

Question 1

(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.

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 3

(Counting sort)

(1) Illustrate the operations of Counting sort on $A = [6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2]$.

(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

Question 4

The closed-form runtime expression $T(n)$ for the number of compares between array items executed by ExchangeSort is:

- A. $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$
- B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$
- C. $T(n) = n^2 - 1$
- D. $T(n) = n^2 + 1$
- E. $T(n) = n^2$

Question 5

The closed-form runtime expression $T(n)$ for the maximum number of Swap calls made by ExchangeSort is:

- A. $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$
- B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$
- C. $T(n) = n^2 - 1$
- D. $T(n) = n^2 + 1$
- E. $T(n) = n^2$

Question 6

The closed-form runtime expression $T(n)$ for the maximum number of Swap calls made by BubbleSort is:

- A. $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$
- B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$
- C. $T(n) = n^2 - 1$
- D. $T(n) = n^2 + 1$
- E. $T(n) = n^2$

Question 1

(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?

Question 1

(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.

Question 1

(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.

Question 1

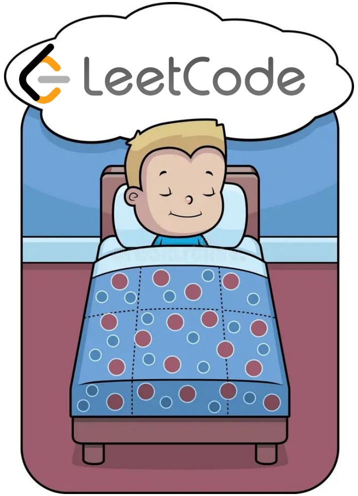
(Merge sort) Merge sort is in its nature, a Divide-and-Conquer algorithm.

(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

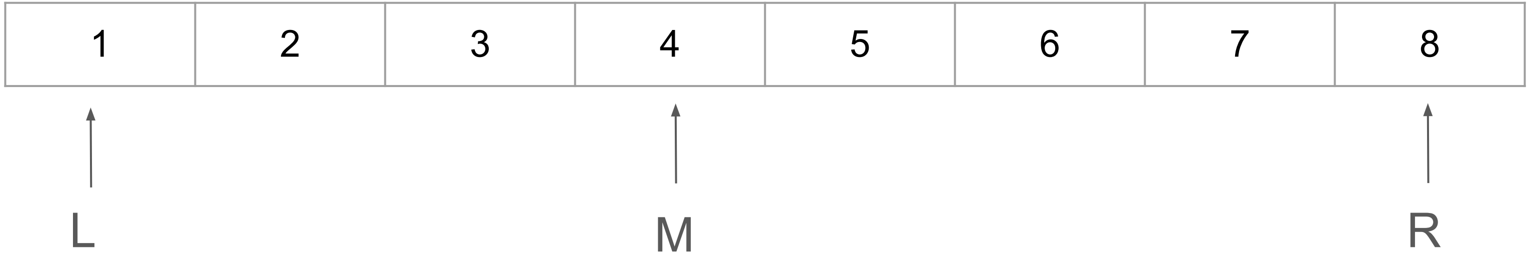
```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

Example of Binary Search for x = 5

1	2	3	4	5	6	7	8

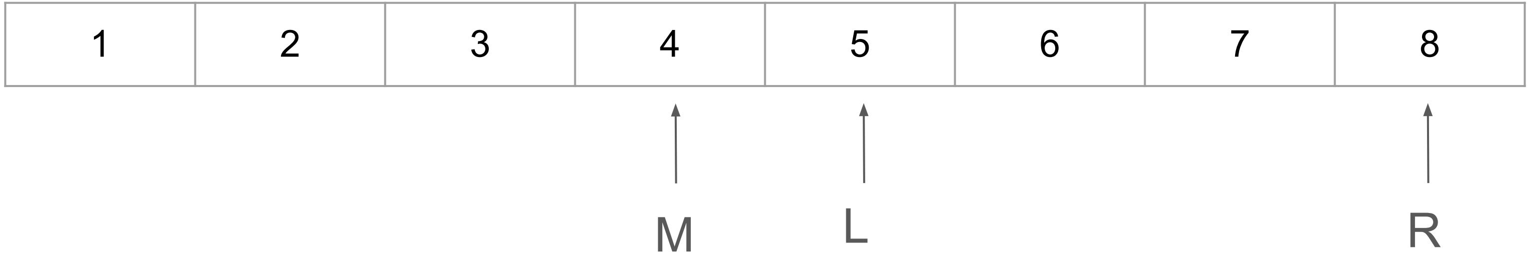
```
def binarySearch(A[1:n],x):
    l = 1, r = n
    while l <= r:
        m = ⌊(l + r) / 2⌋
        if A[m] == x:
            return m
        elif A[m] < x:
            l = m + 1
        else:
            r = m - 1
    return -1
```

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

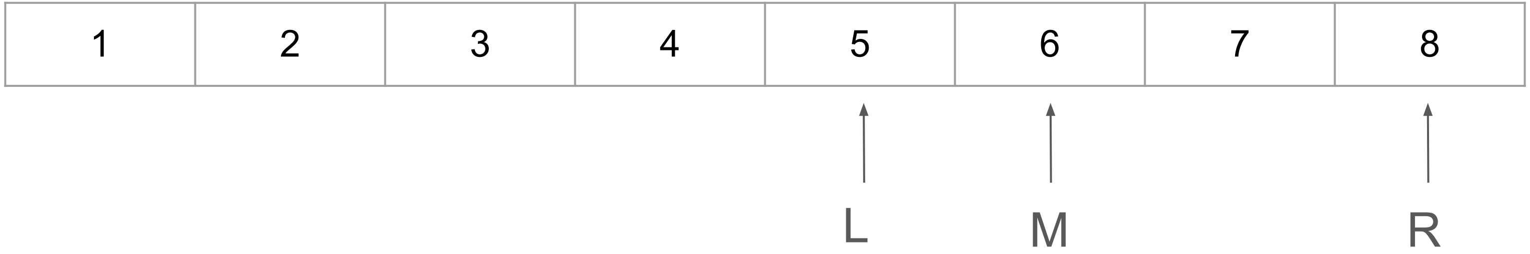


1	2	3	4	5	6	7	8

```
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.

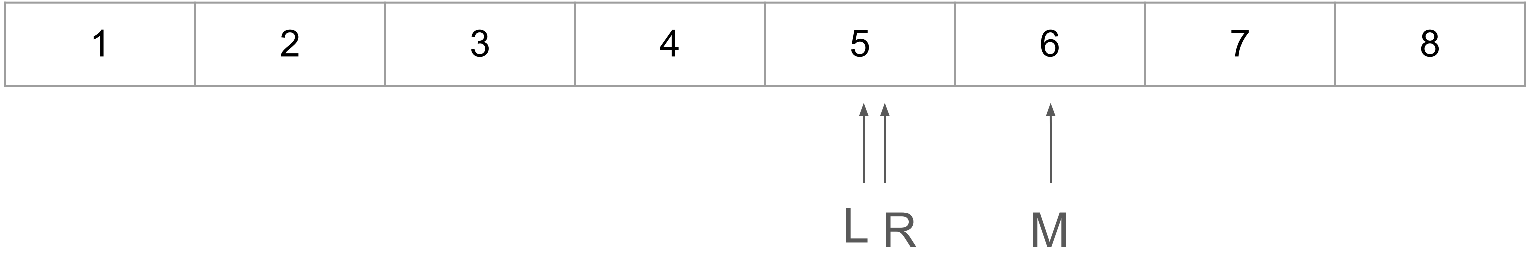
Example of Binary Search for x = 5



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

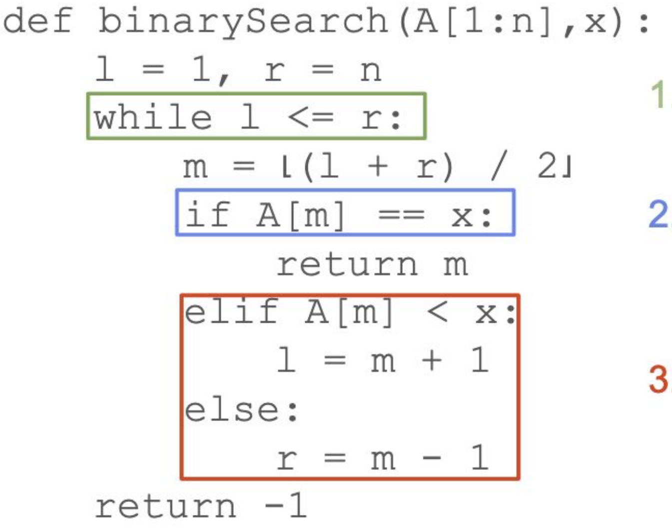


1	2	3	4	5	6	7	8

```
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

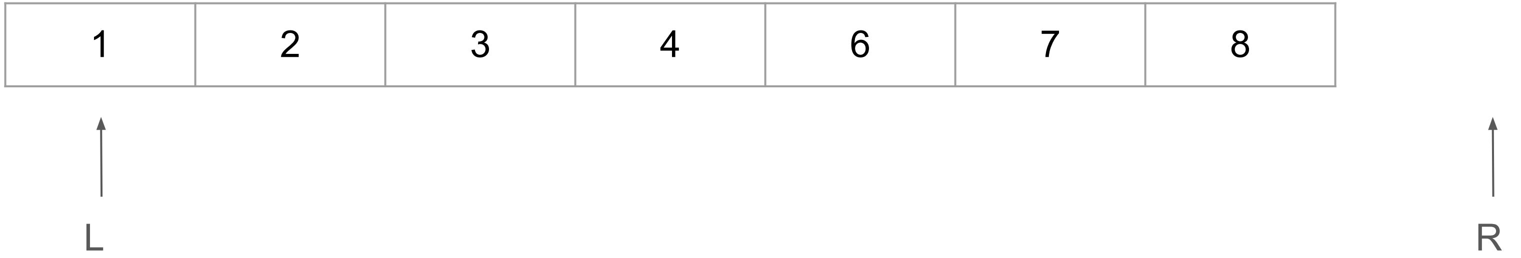
What should the search range be?

```
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

What should the search range be?

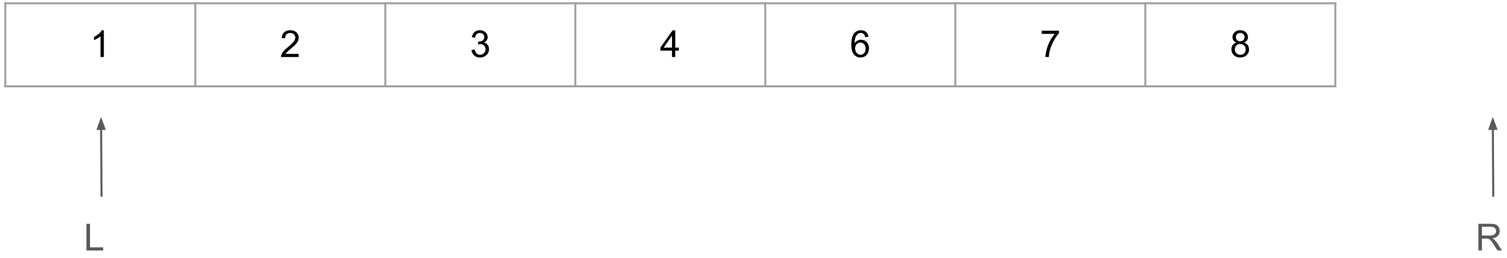
L = 1, R = n, the missing index could be any index

```
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

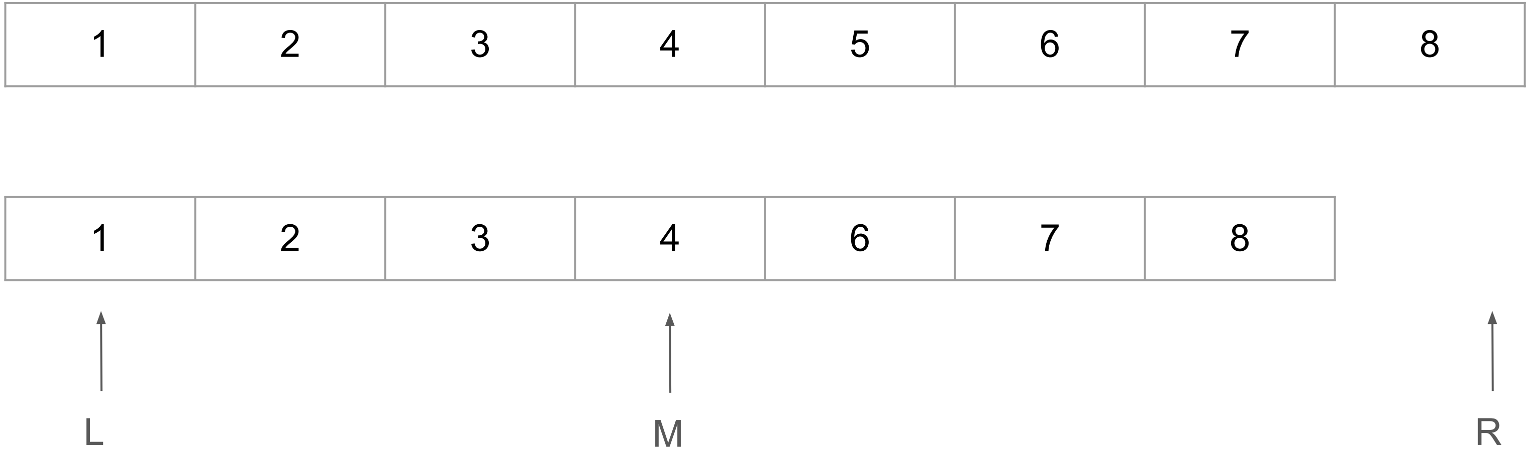
When should we stop?

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

When should we stop?

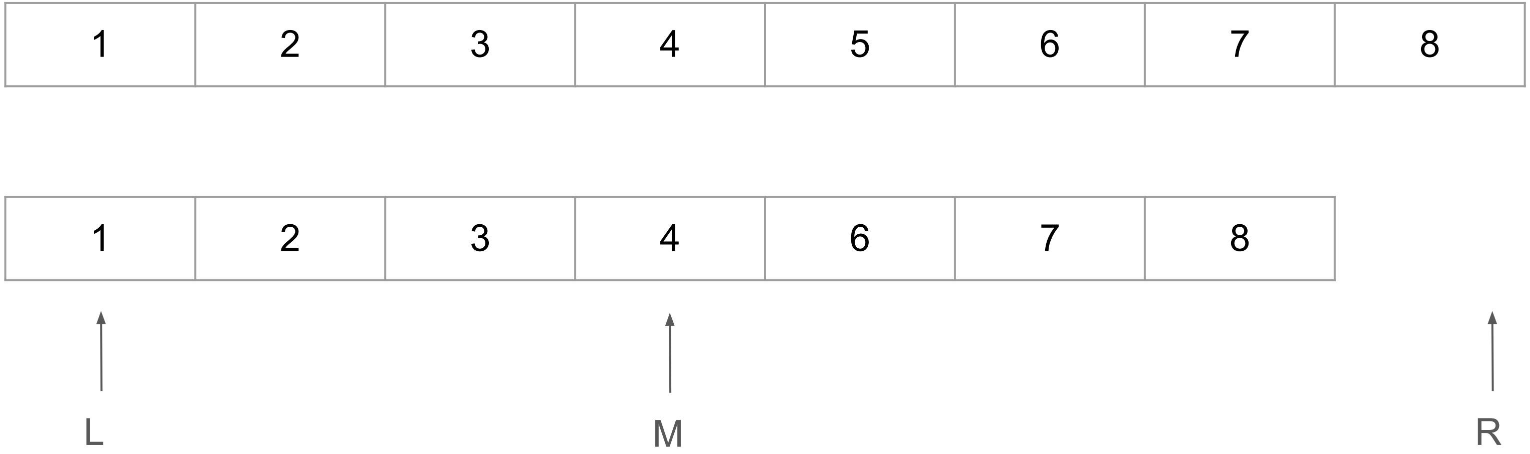
This is often much trickier to figure out next.

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



1	2	3	4	5	6	7	8

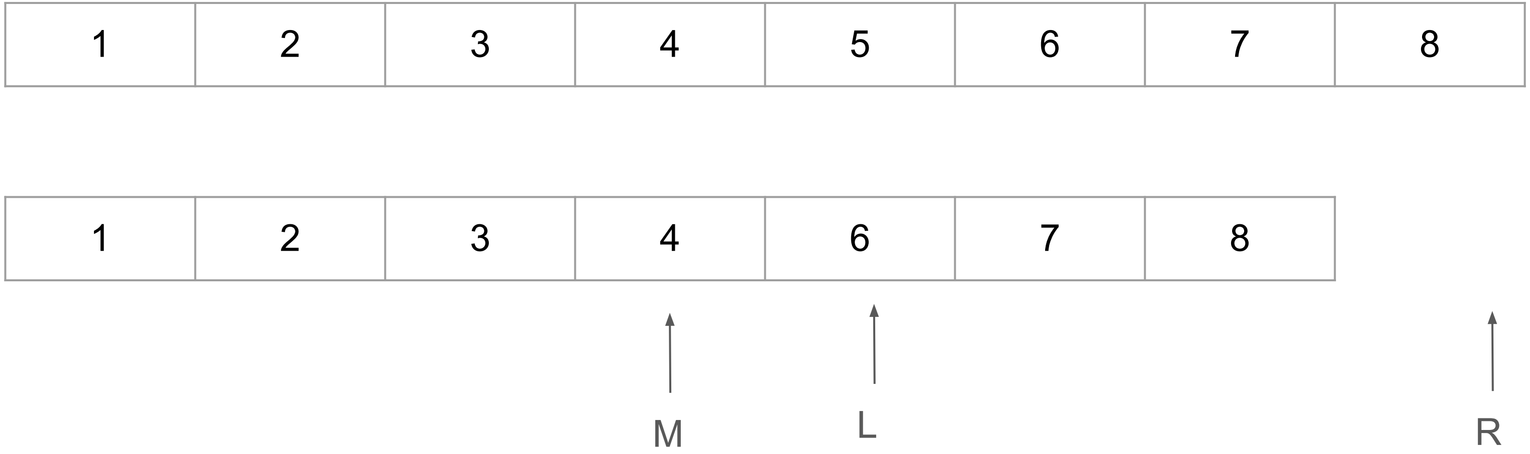
1	2	3	4	6	7	8

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

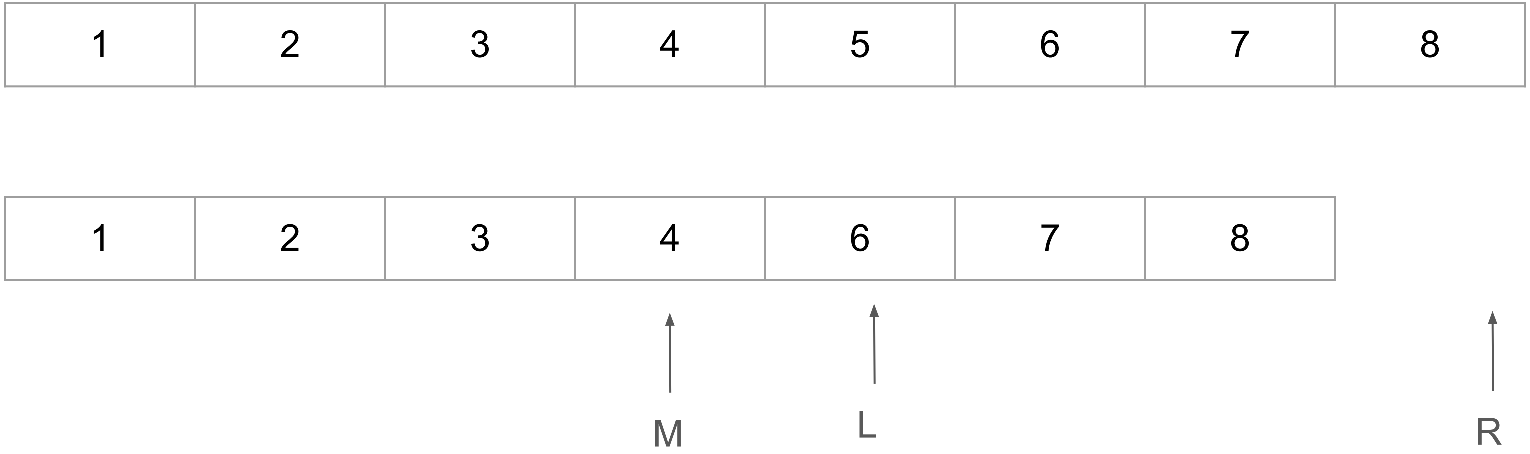
How should we cut the interval?

- **A[m] == B[m] implies that 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



1	2	3	4	5	6	7	8

1	2	3	4	6	7	8

How should we cut the interval?

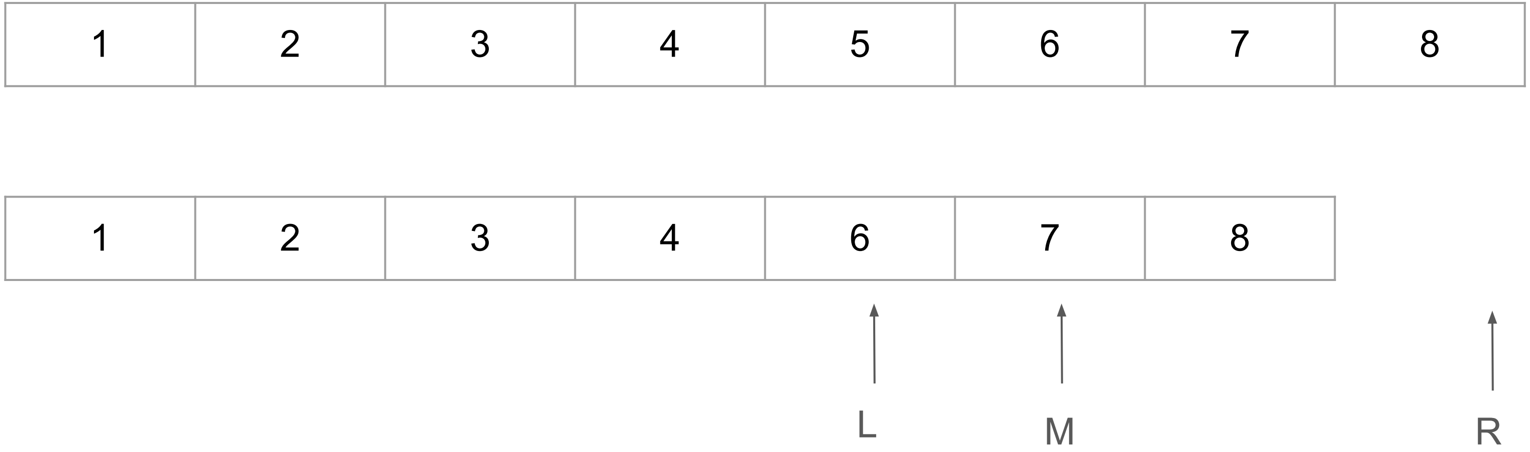
- A[m] == B[m] implies that everything before is equal. The missing element must be in the right half!

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



1	2	3	4	5	6	7	8

1	2	3	4	6	7	8

How should we cut the interval?

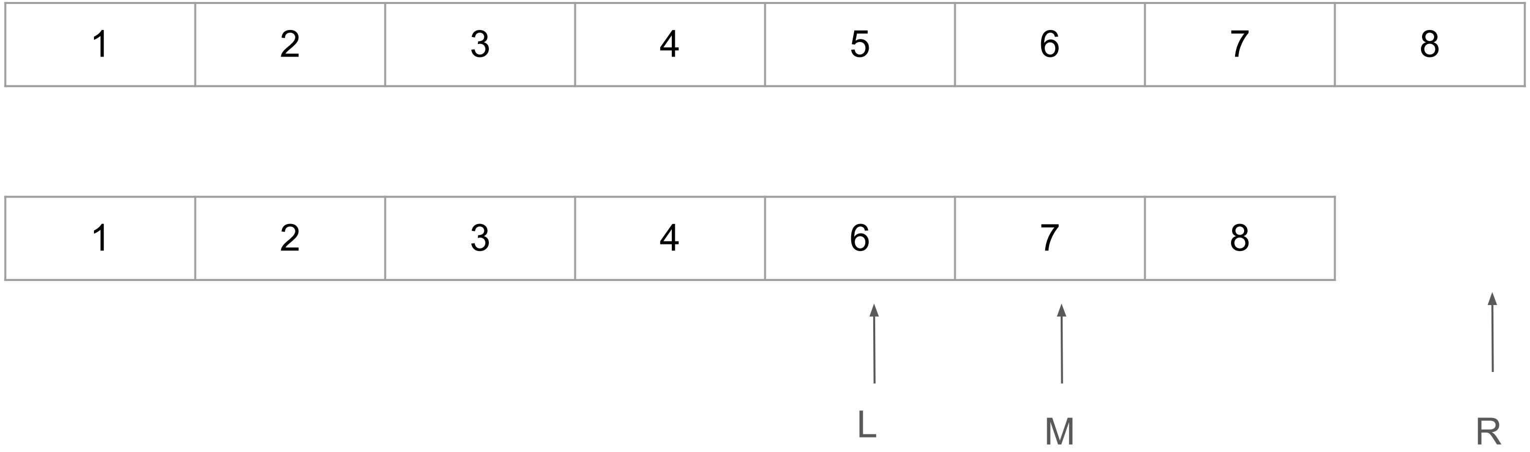
- A[m] == B[m] implies that everything before is equal. The missing element must be in the right half!

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



1	2	3	4	5	6	7	8

1	2	3	4	6	7	8

How should we cut the interval?

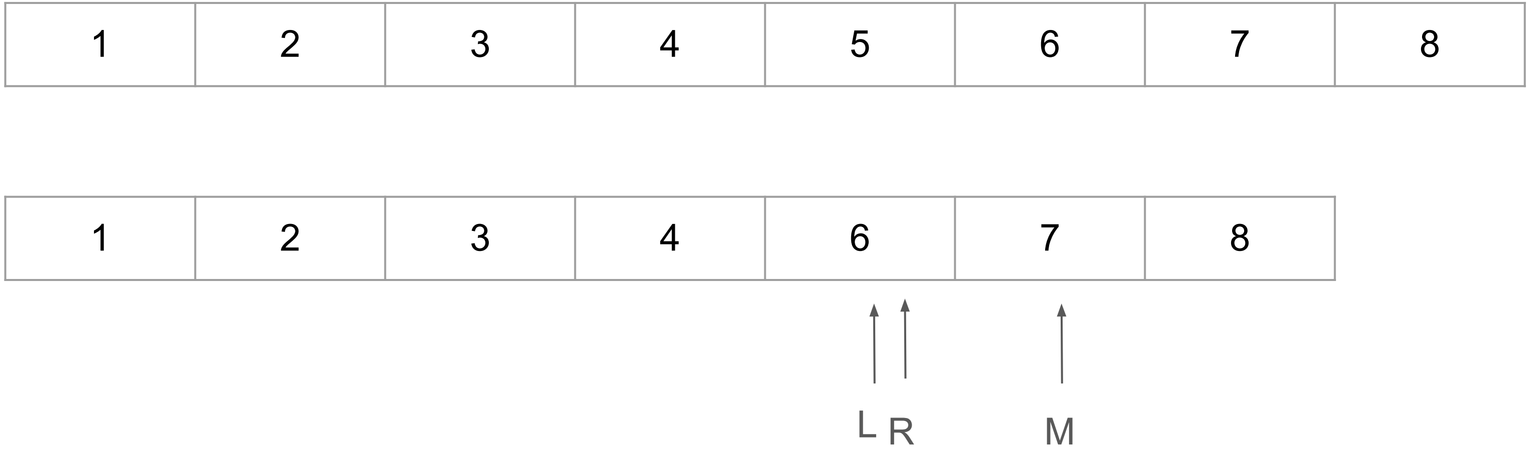
- A[m] == B[m] implies that everything before is equal. The missing element must be in the right half!
- **A[m] != B[m] implies that something in range [l,m] must be missing. The missing element must be in the left half!**

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

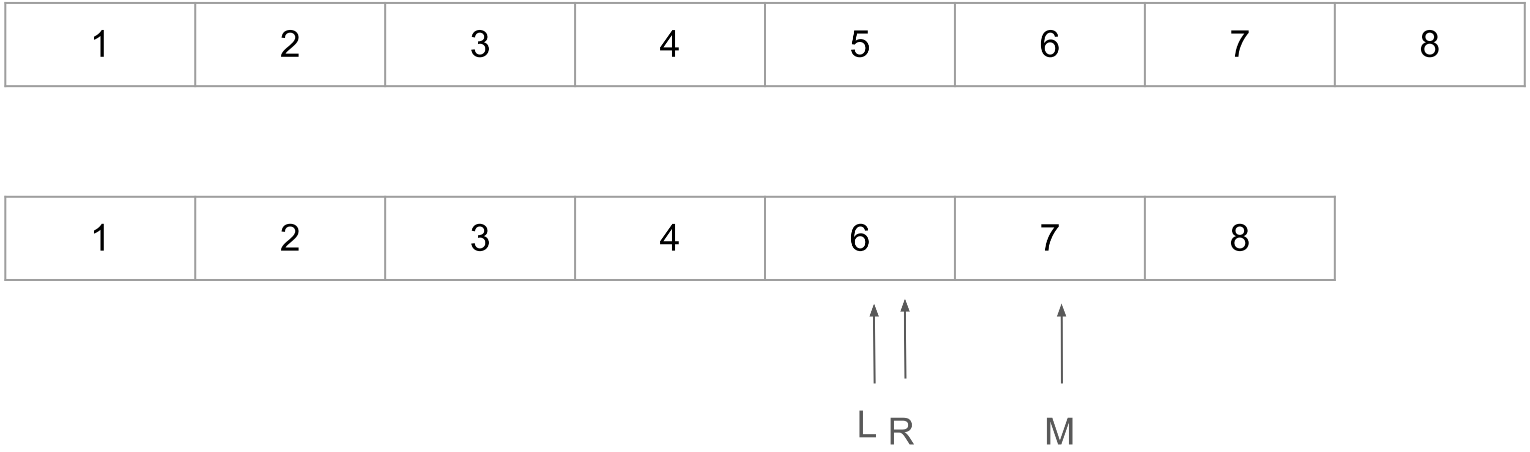


This seems like a good place to stop our algorithm. What's the stop condition?

```
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?

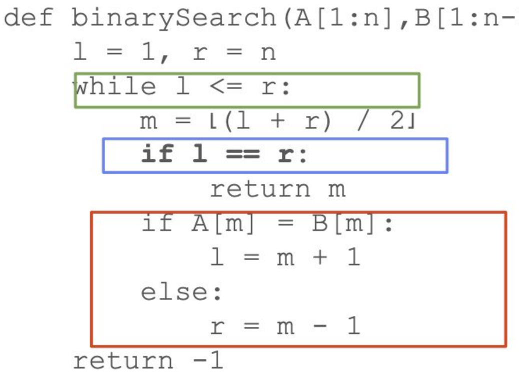
When l == r

```
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

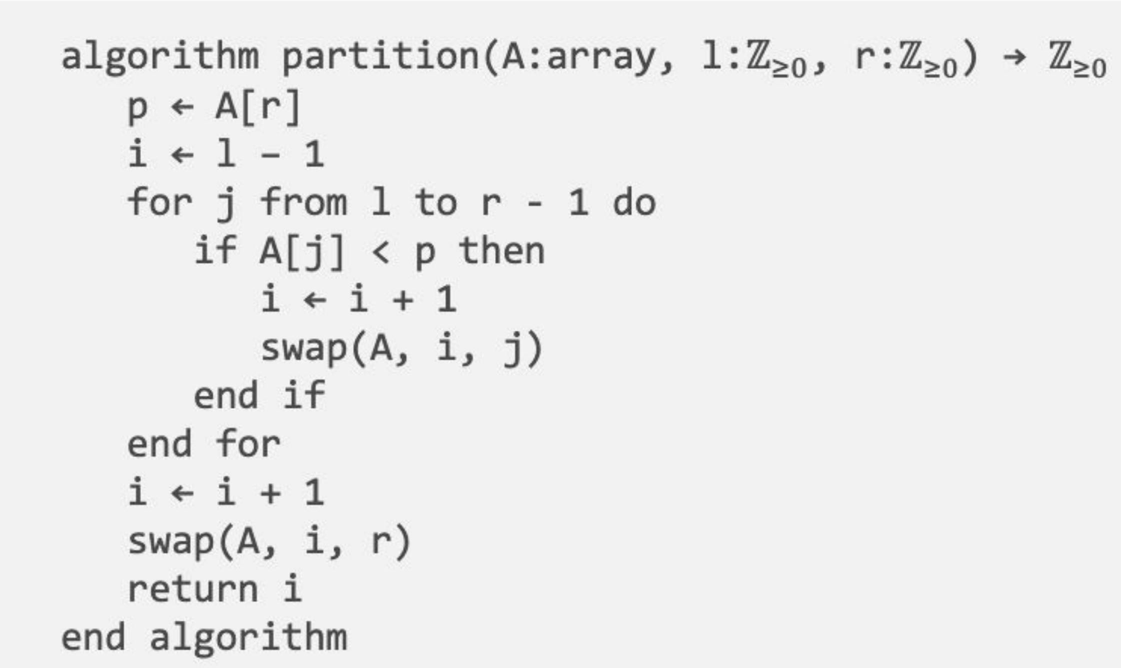
```
def binarySearch(A[1:n],B[1:n-1]x):
    l = 1, r = n
    while l < r:
        m = ⌊(l + r) / 2⌋
        if A[m] = B[m]:
            l = m + 1
        else:
            r = m - 1
    return l
```

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?



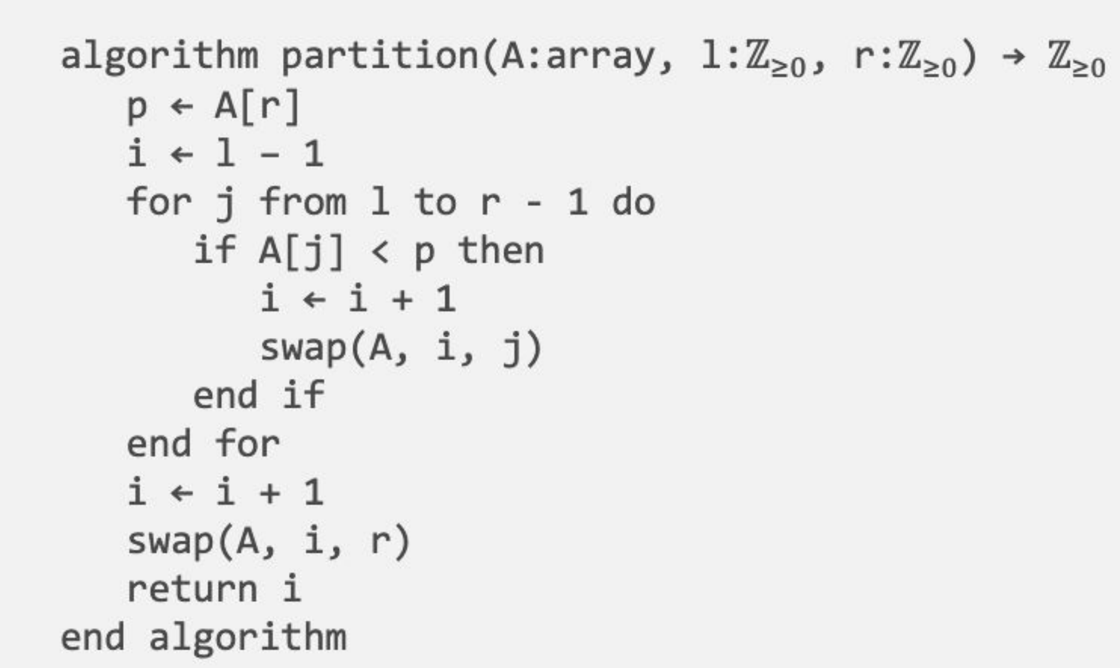
```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return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

Question 2

(Quick sort)

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

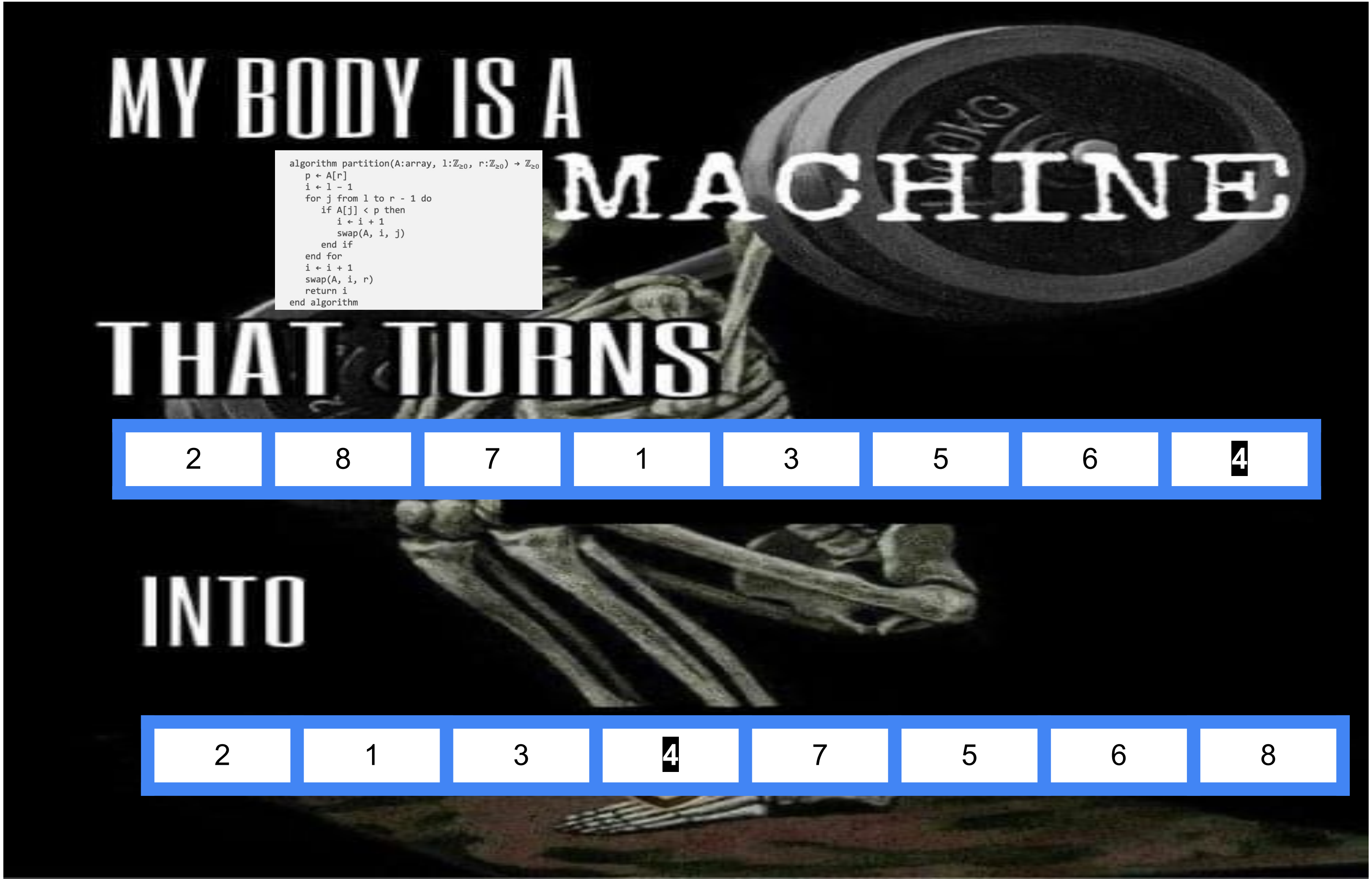
(2) Can we understand the average-case runtime of Quick sort? What is the best policy for selecting the pivot value in the quick sort?



```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, 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
```

pivot p is set as last element

Ok… what does this do tho?



```
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
```

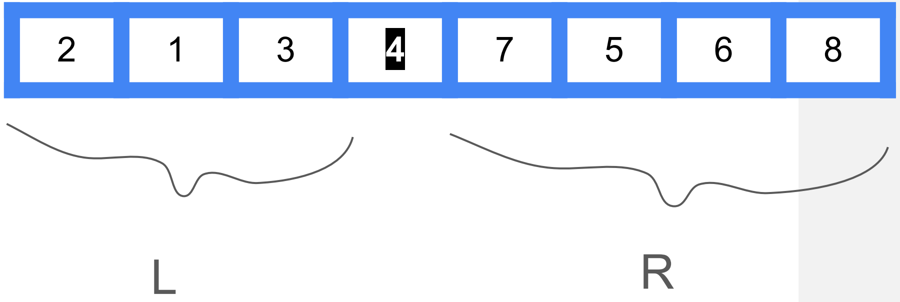
2	8	7	1	3	5	6	4

2	1	3	4	7	5	6	8

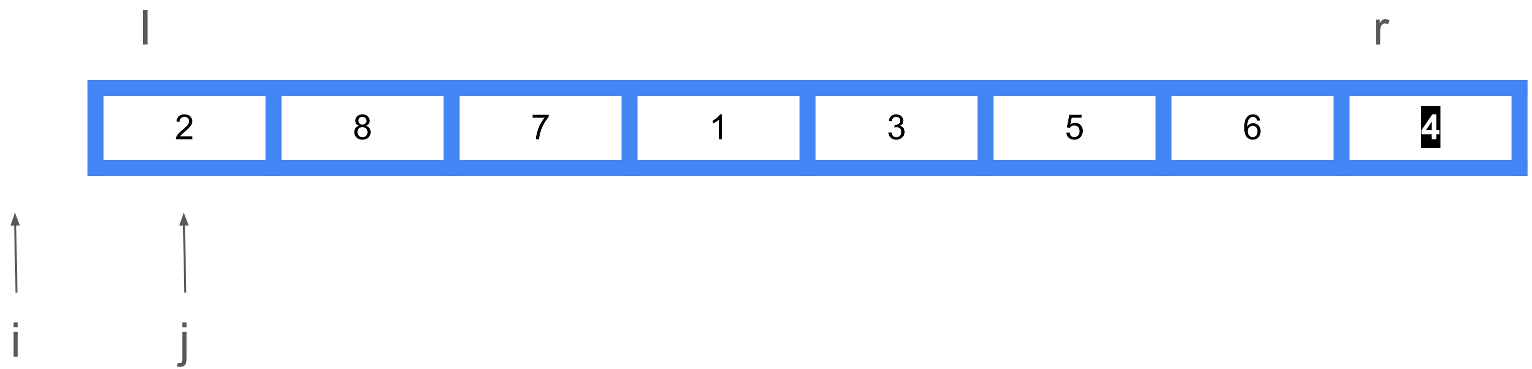
p: pivot

j: goes through entire array

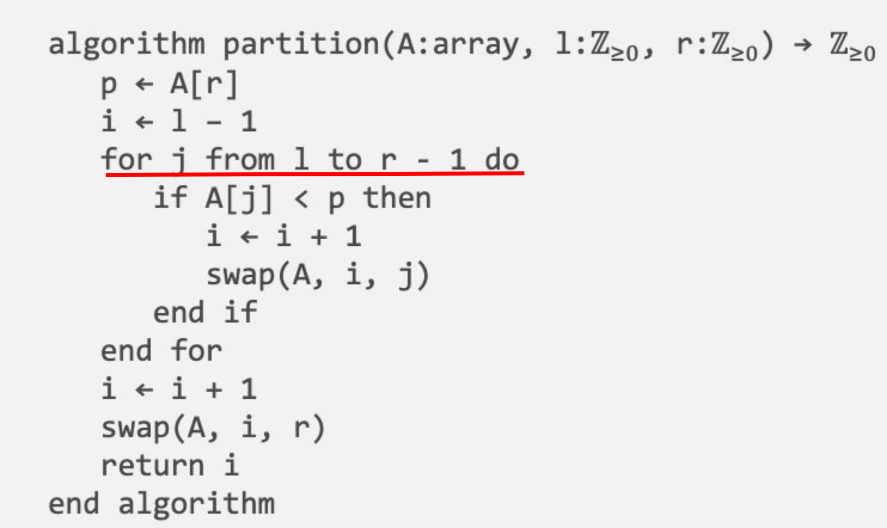
i : growing index of L



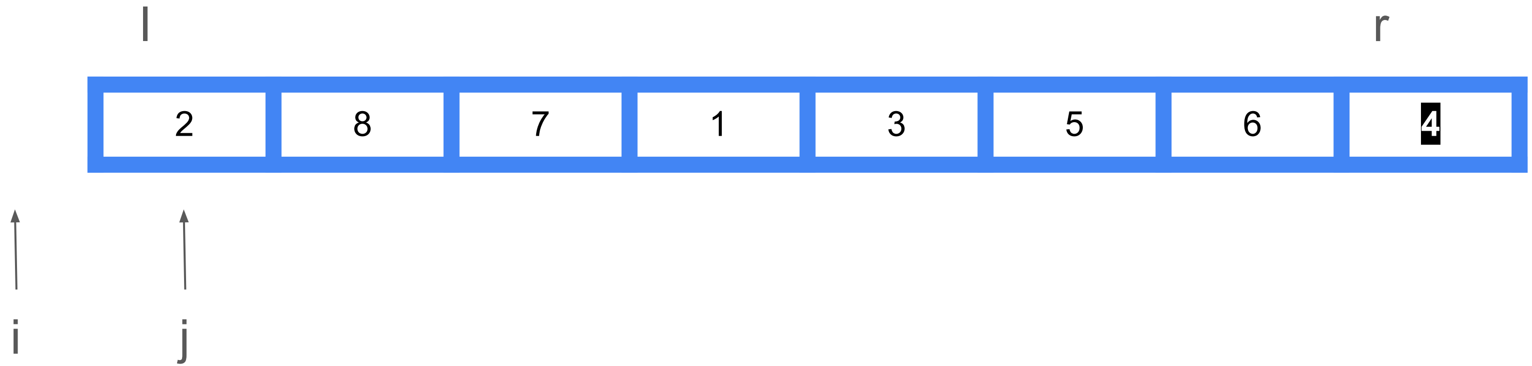
```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return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



Start of the algorithm

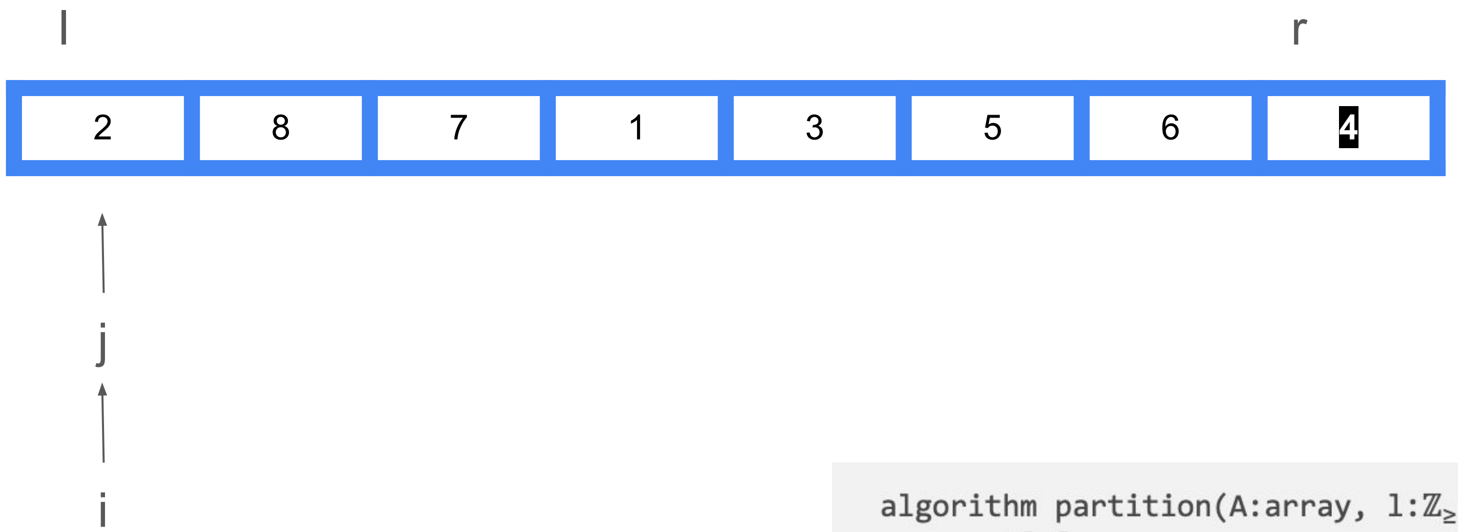


```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return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



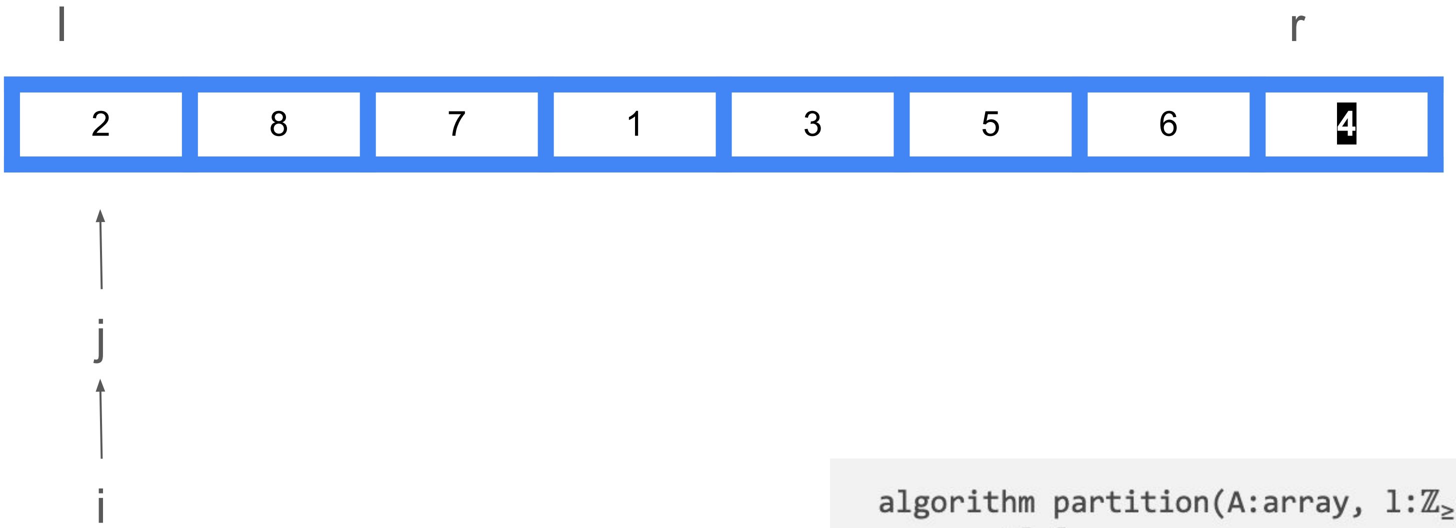
$A[j] = 2 < 4$

```
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
```

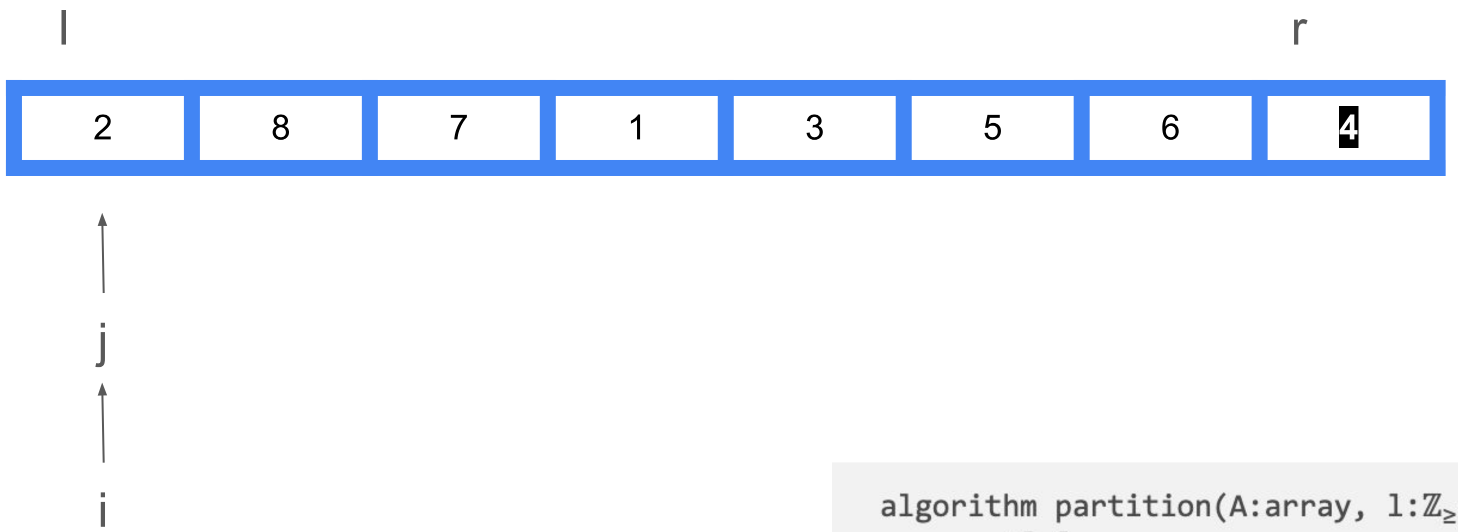


$A[j] = 2 < 4$

```
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
```

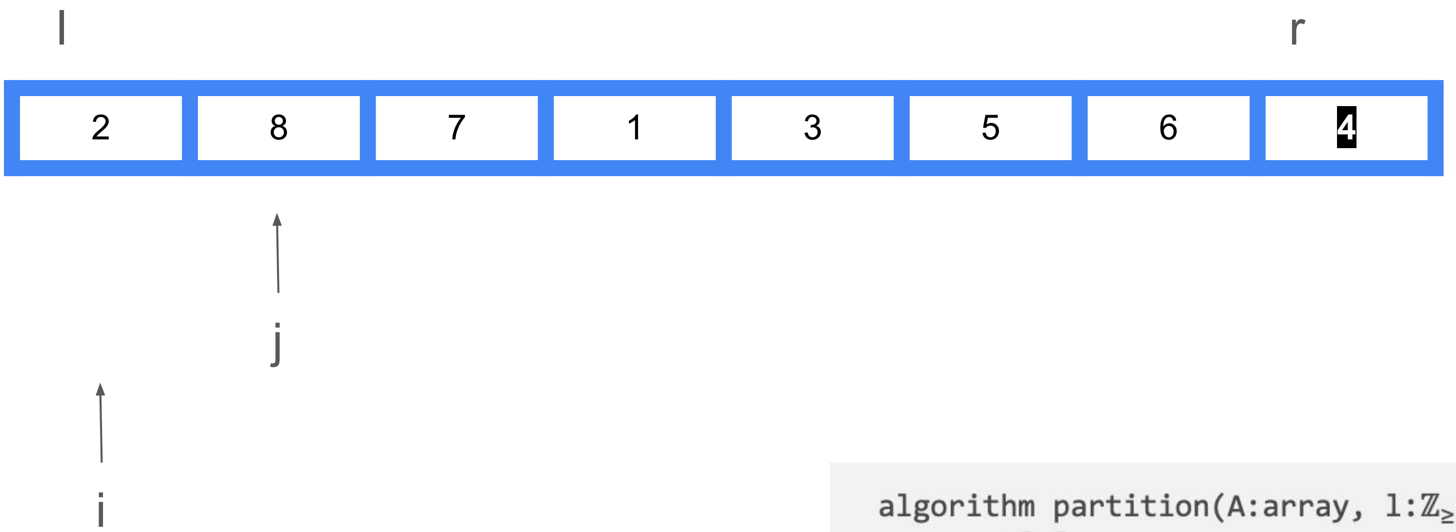


```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm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

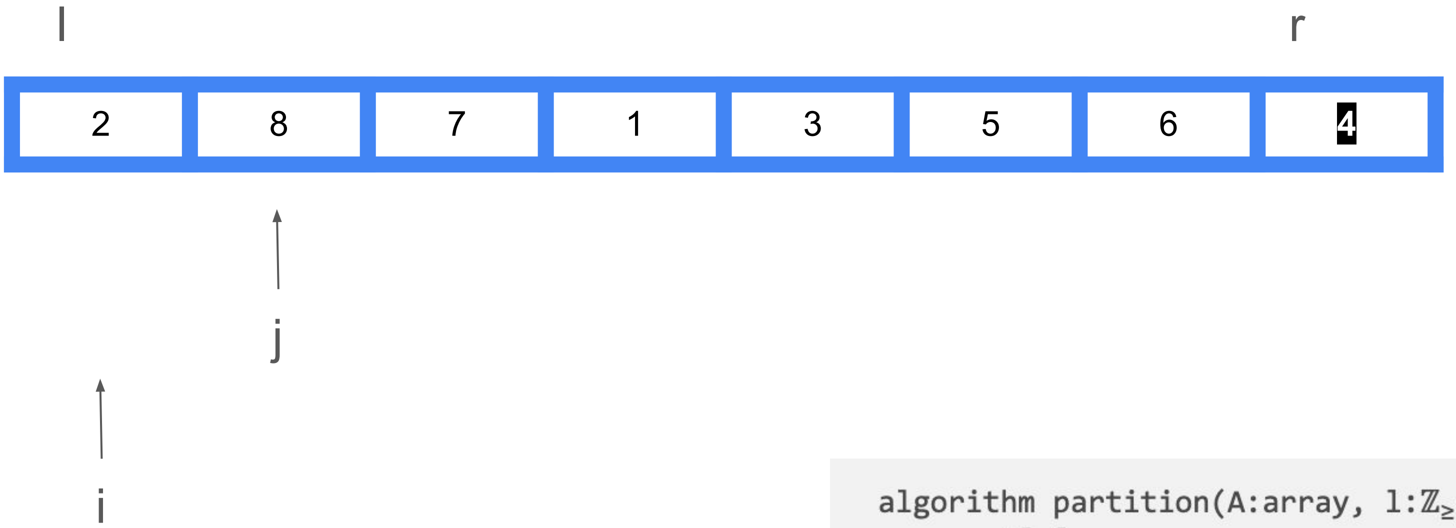


2	8	7	1	3	5	6	4

```
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
```

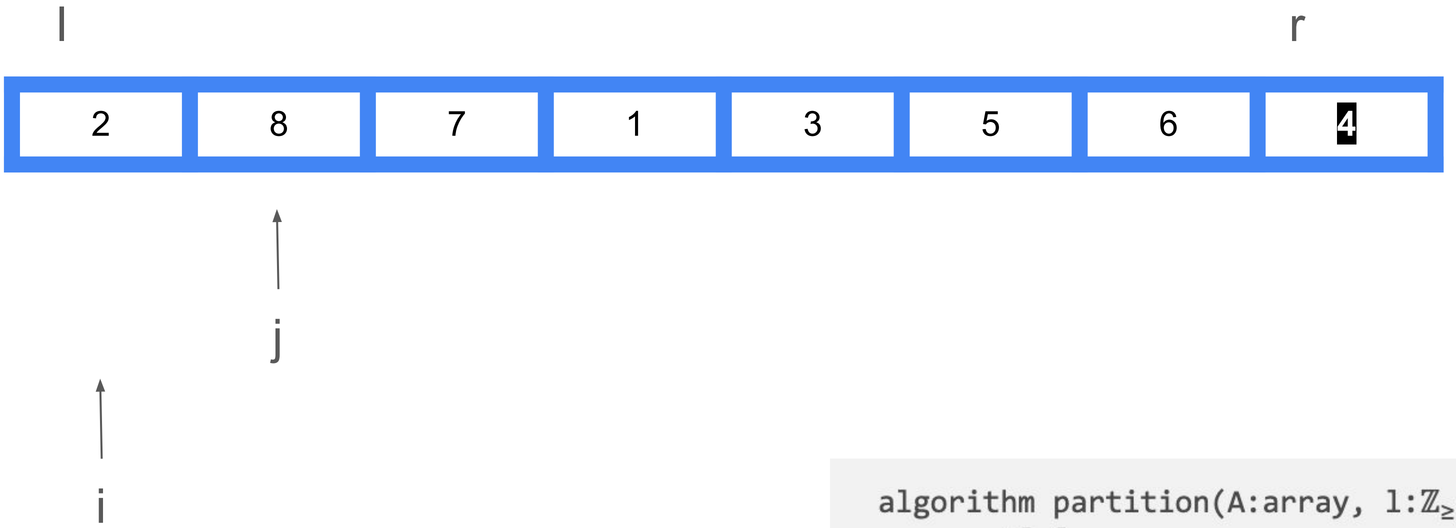


```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return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

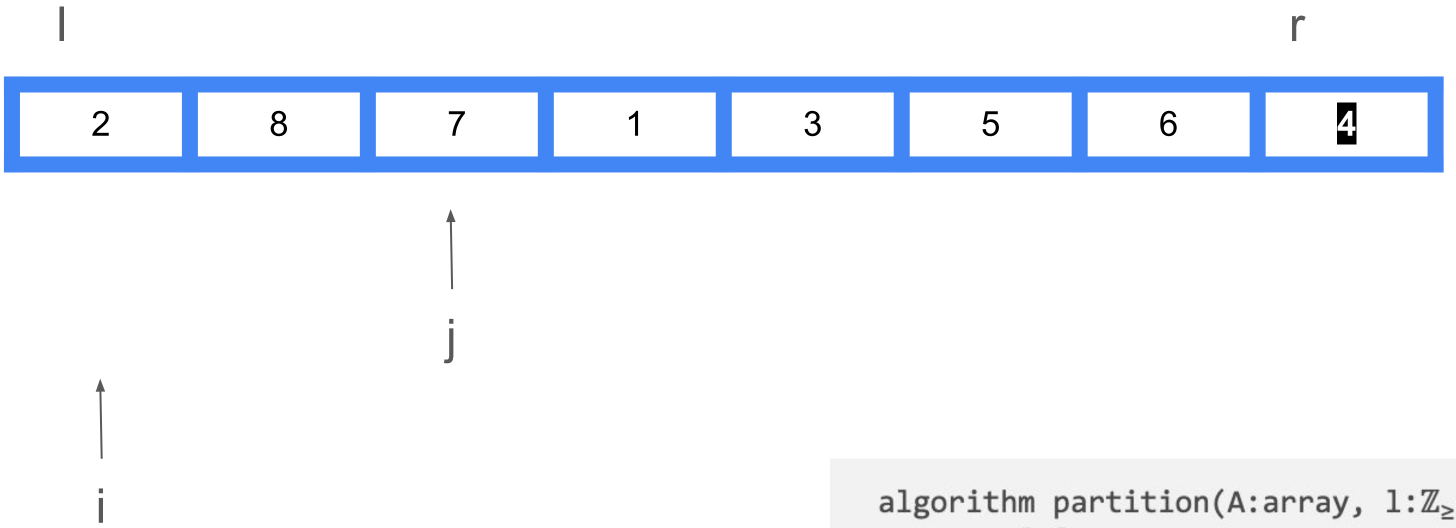


A[j] = 8 < 4? No

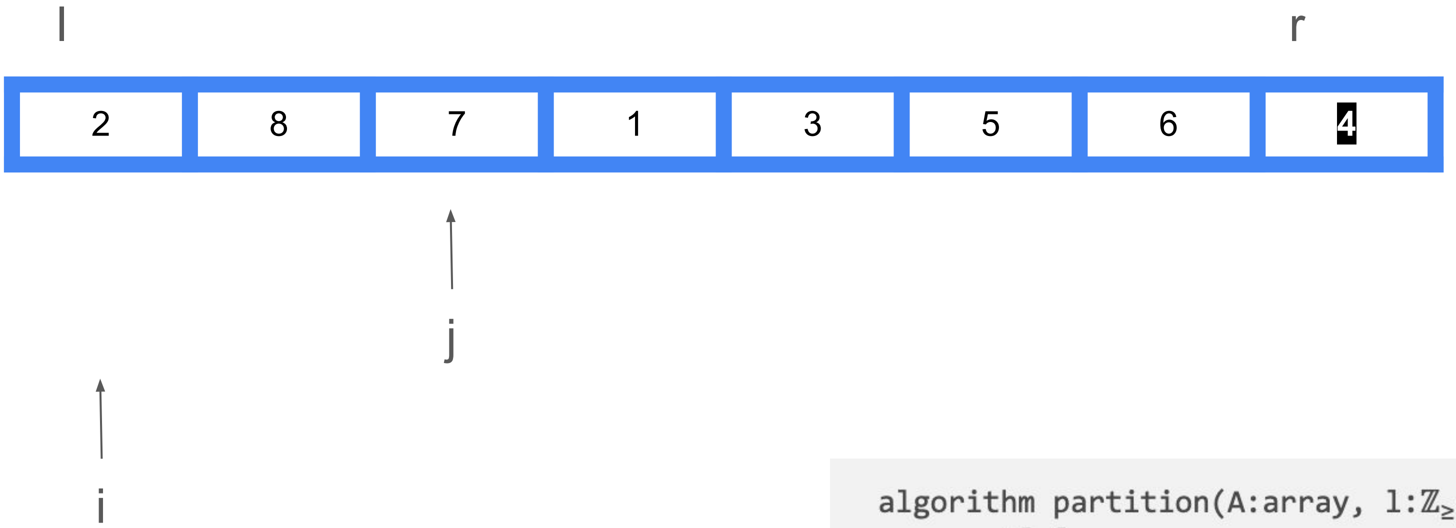
```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return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



```
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
```



```
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
```



l r

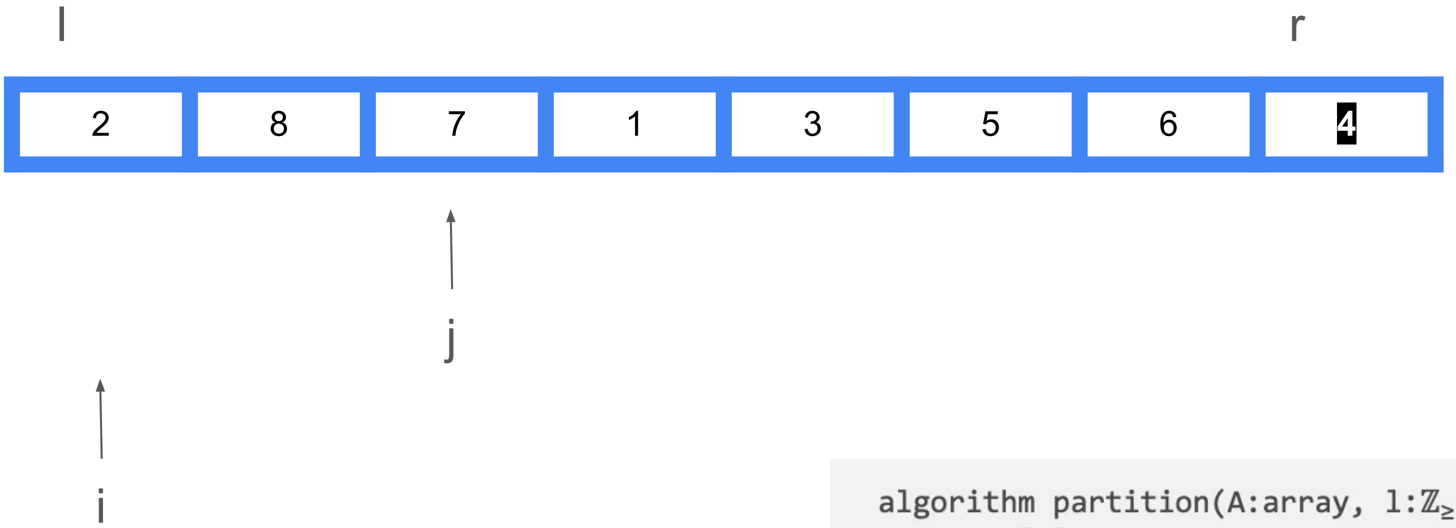
2	8	7	1	3	5	6	4

j

i

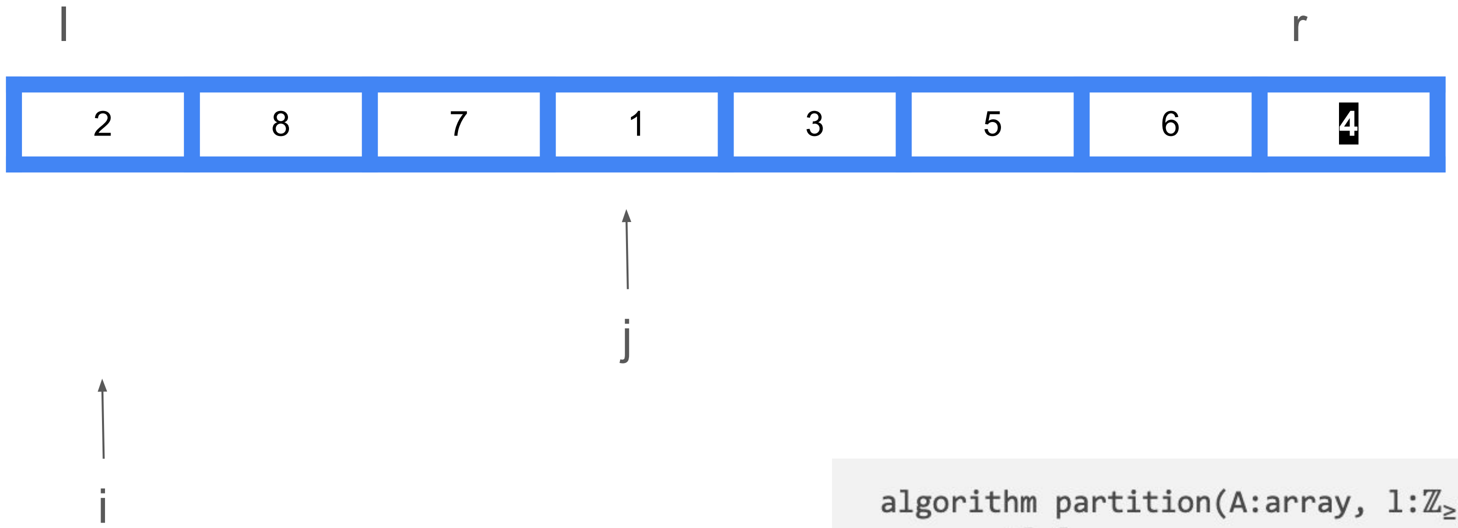
A[j] < 4? Nope

```
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
```



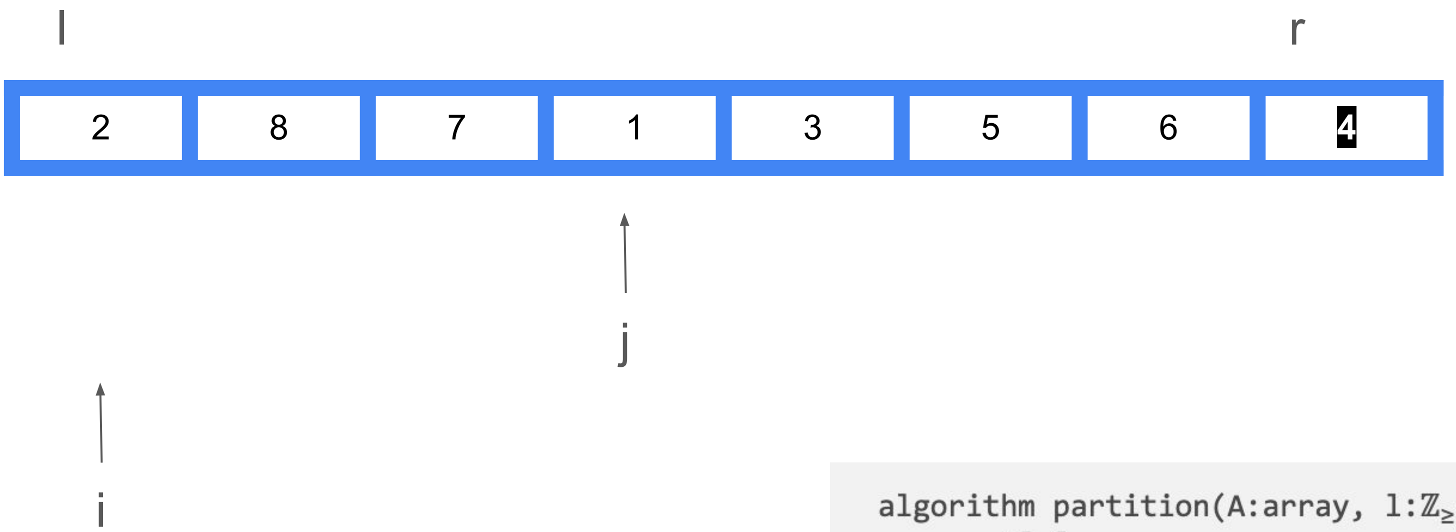
2	8	7	1	3	5	6	4

```
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
```



2	8	7	1	3	5	6	4

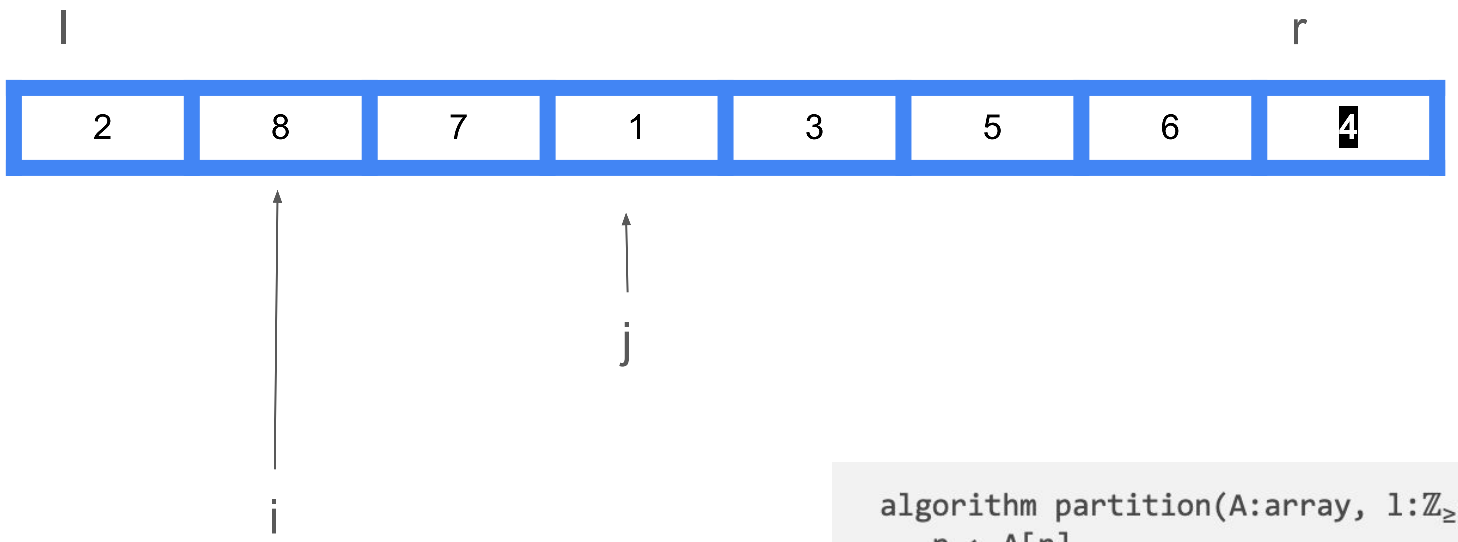
```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return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



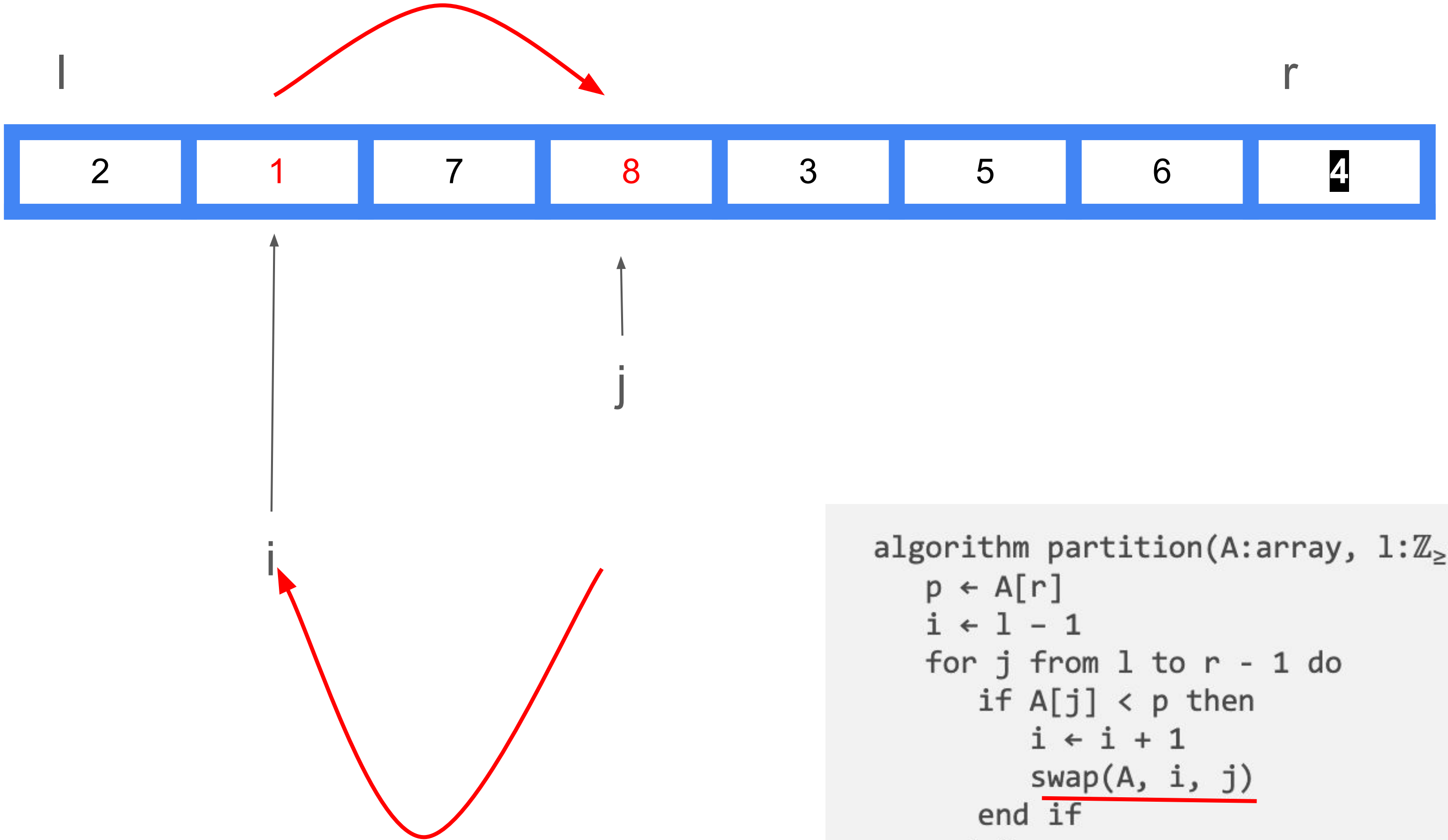
2	8	7	1	3	5	6	4

$A[j] = 1 < 4$

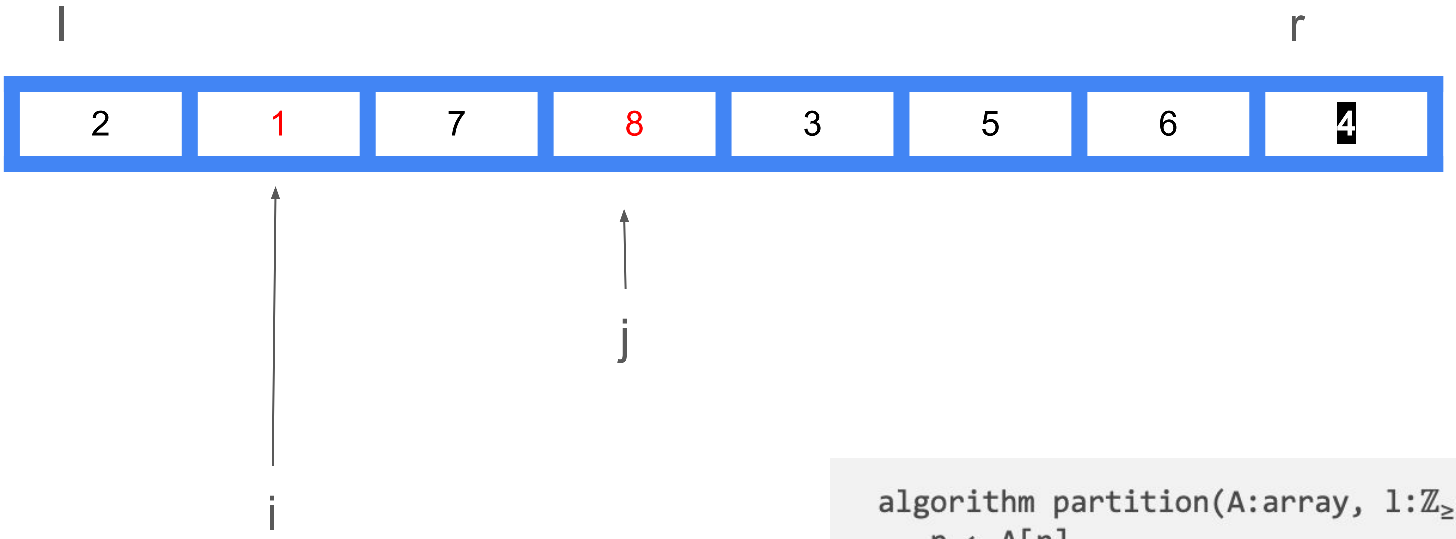
```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm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



```
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
```

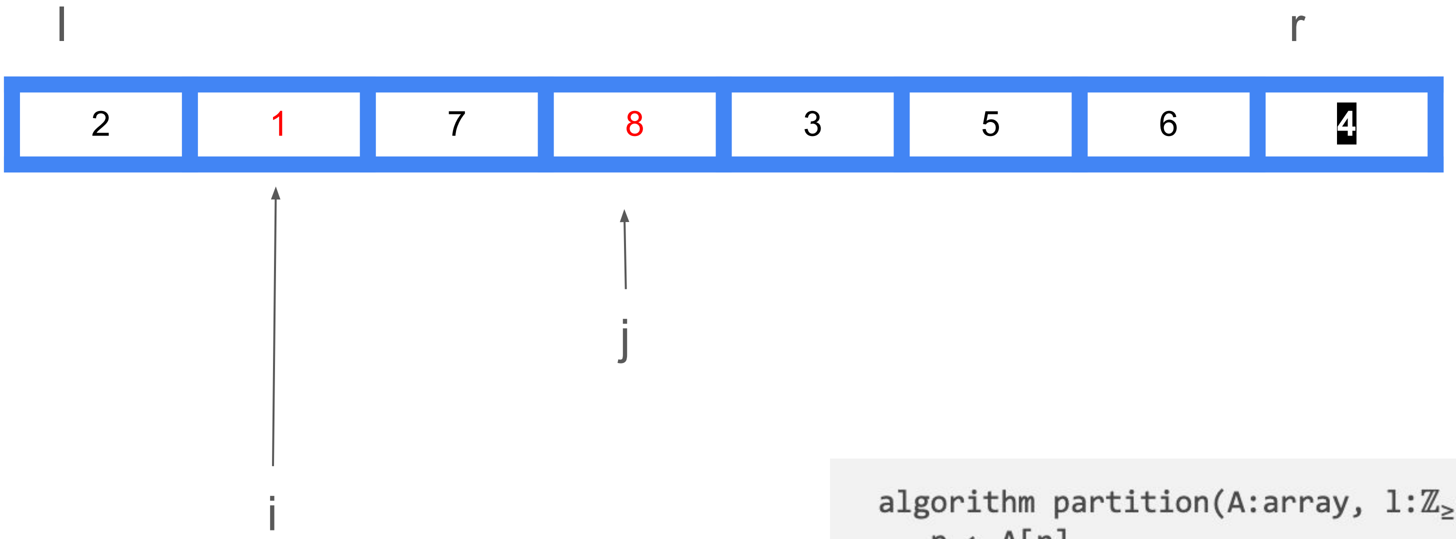


```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return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



We've done two swaps now. Insight?

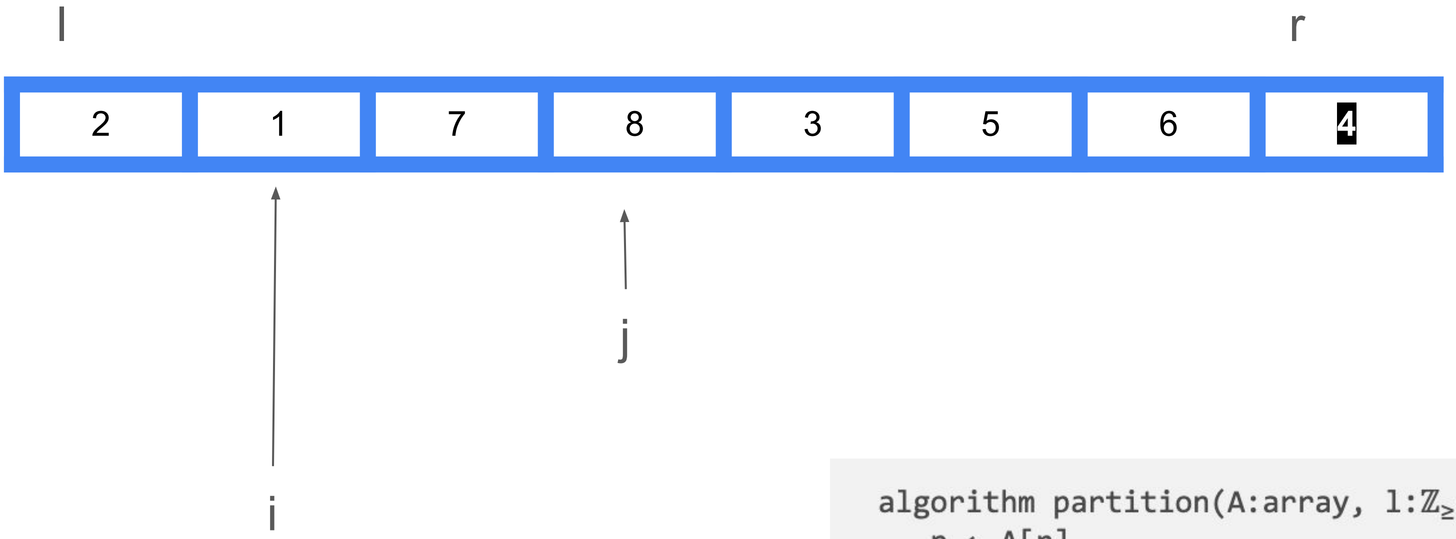
```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm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



We’ve done two swaps now. Insight?

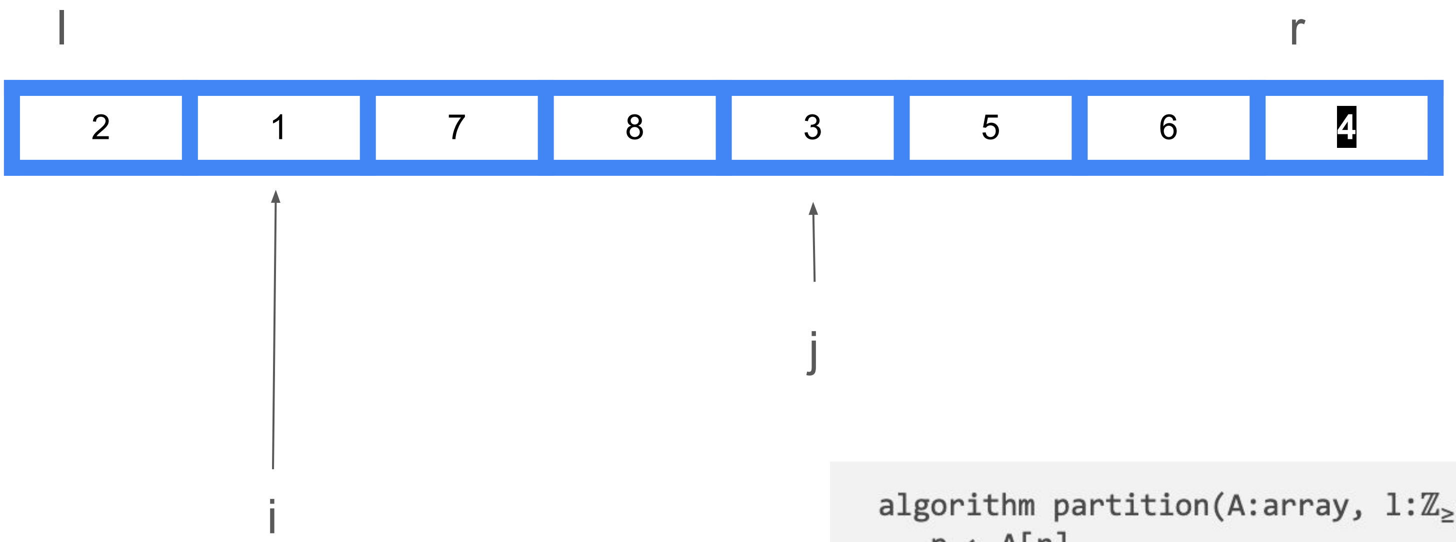
Everything up to i is less than the pivot

```
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
```

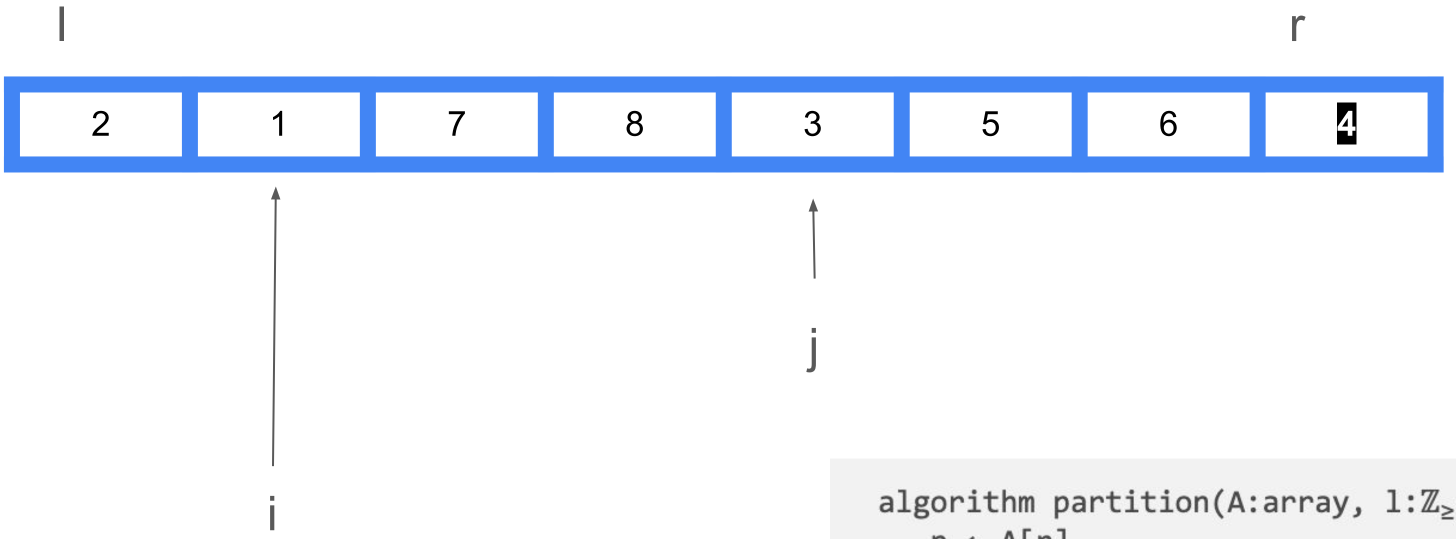


2	1	7	8	3	5	6	4

```
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
```

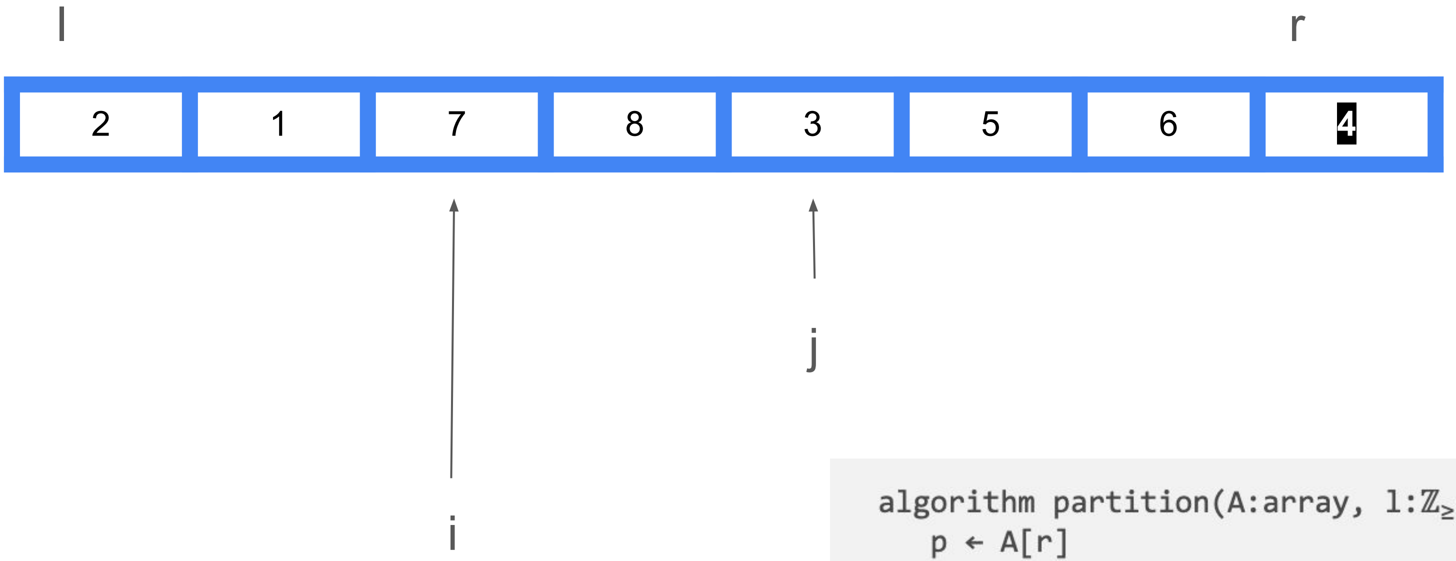


```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm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



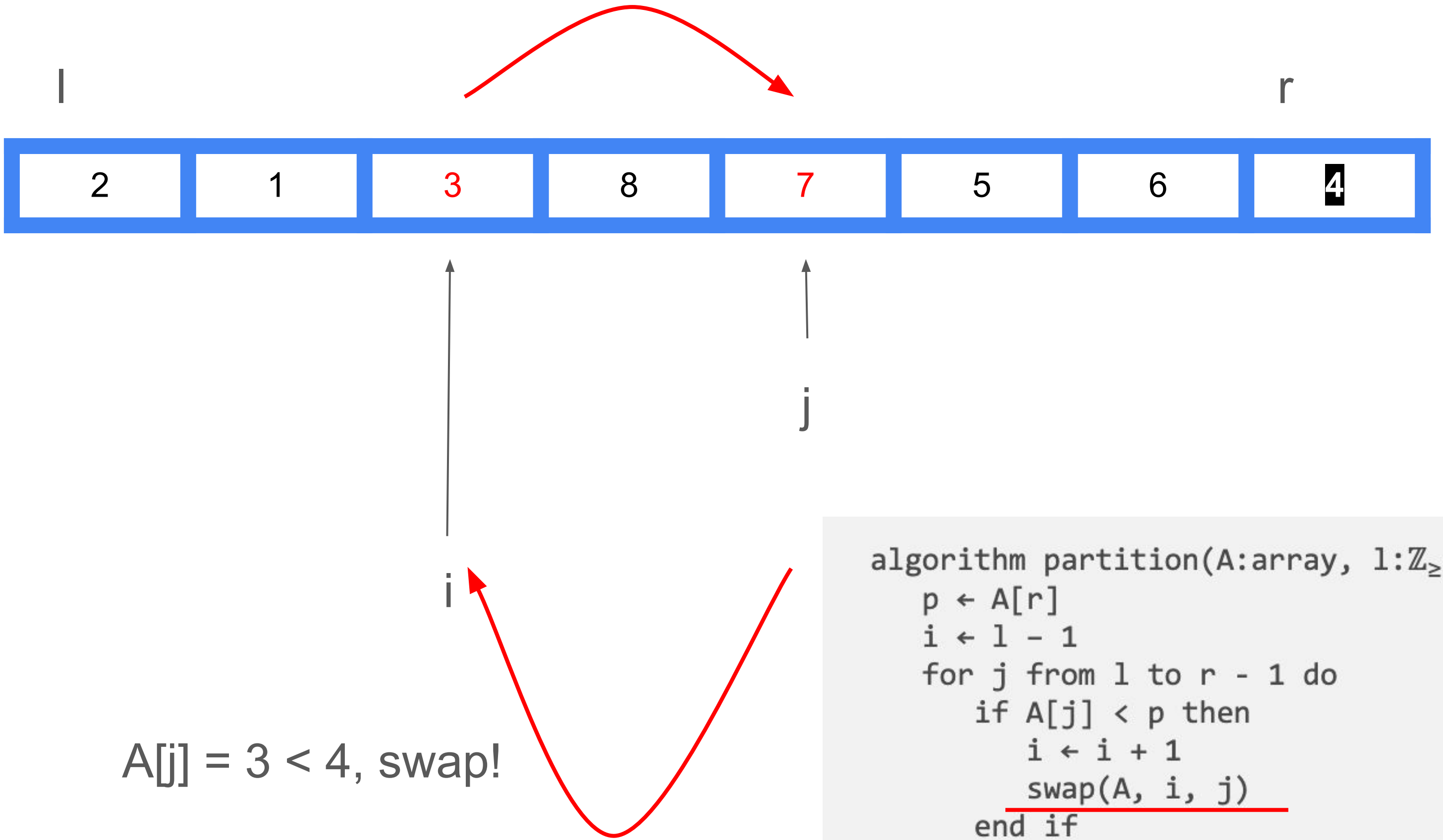
A[j] = 3 < 4, swap!

```
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
```



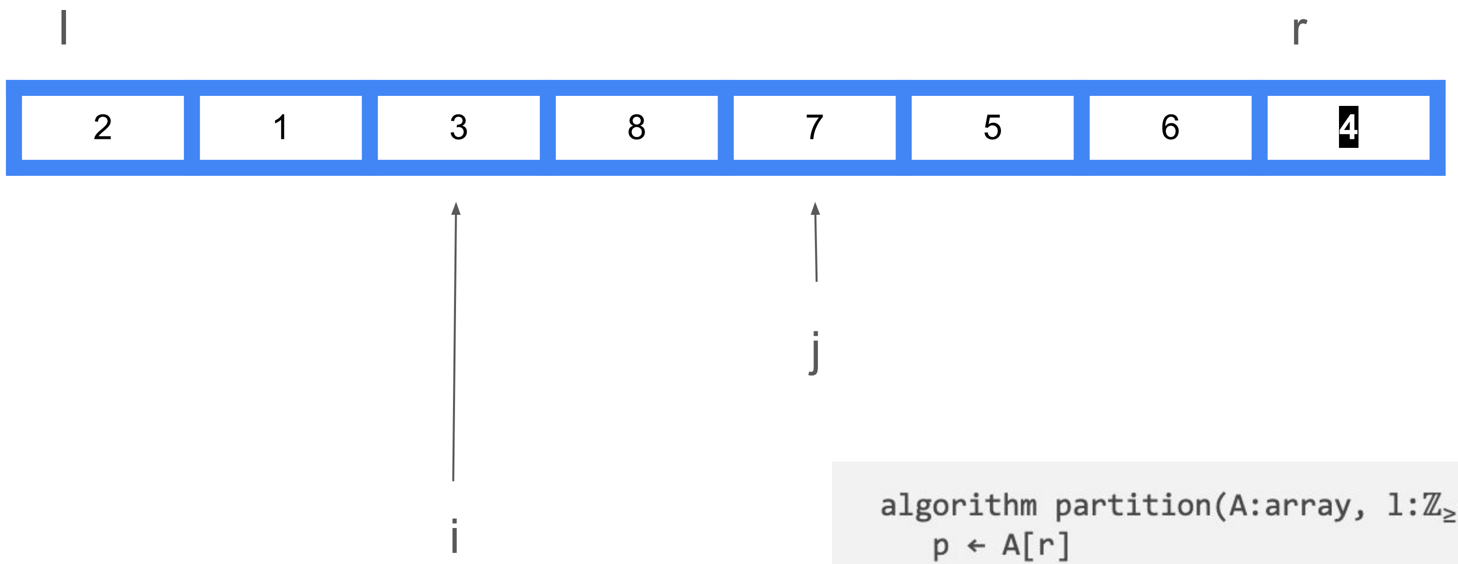
A[j] = 3 < 4, swap!

```
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
```



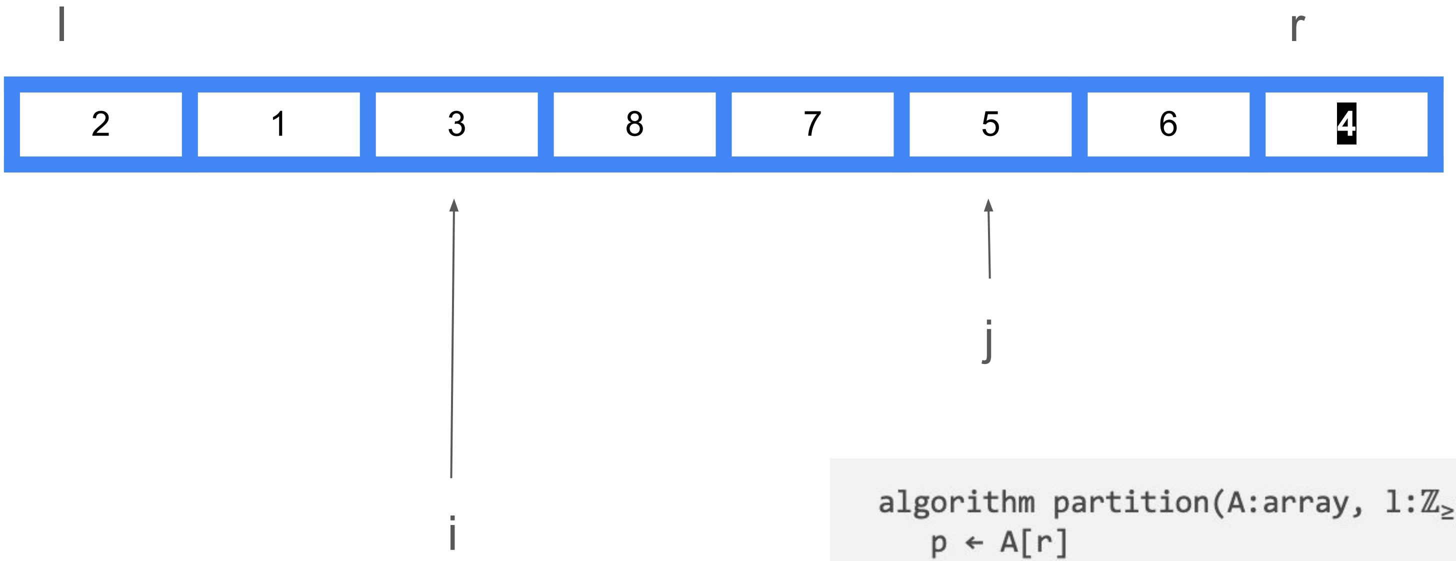
A[j] = 3 < 4, swap!

```
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
```

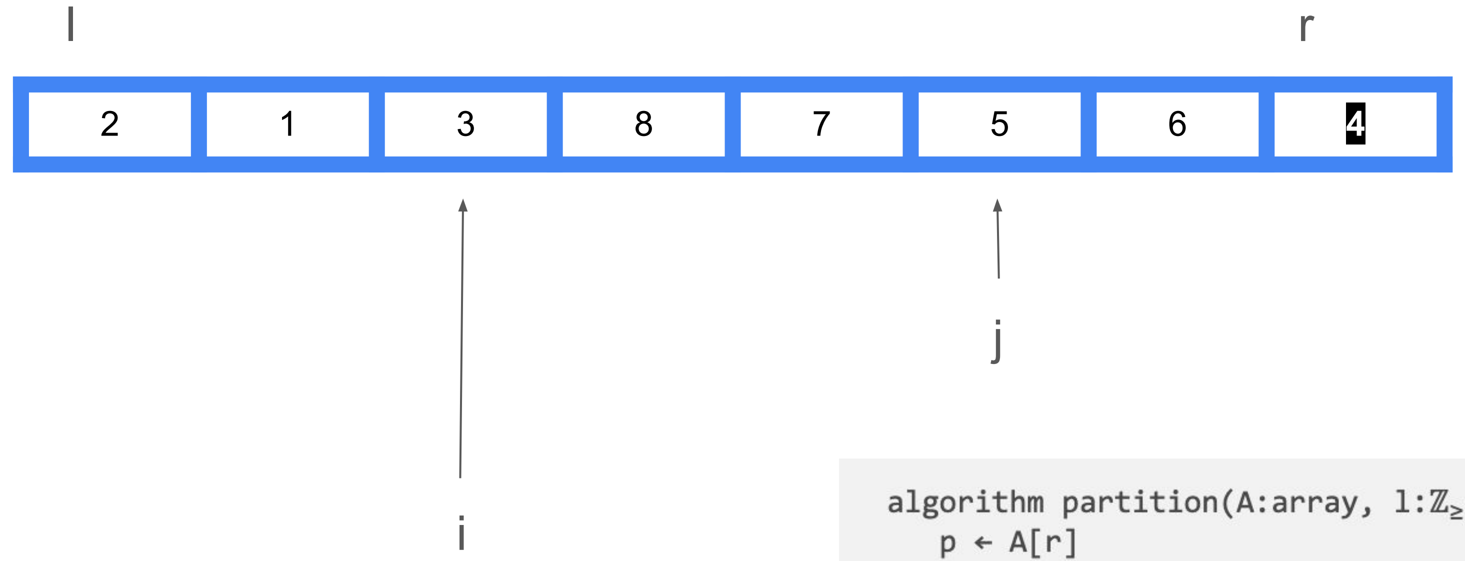


```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return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

Everything up to i is less than the pivot



```
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
```



l

r

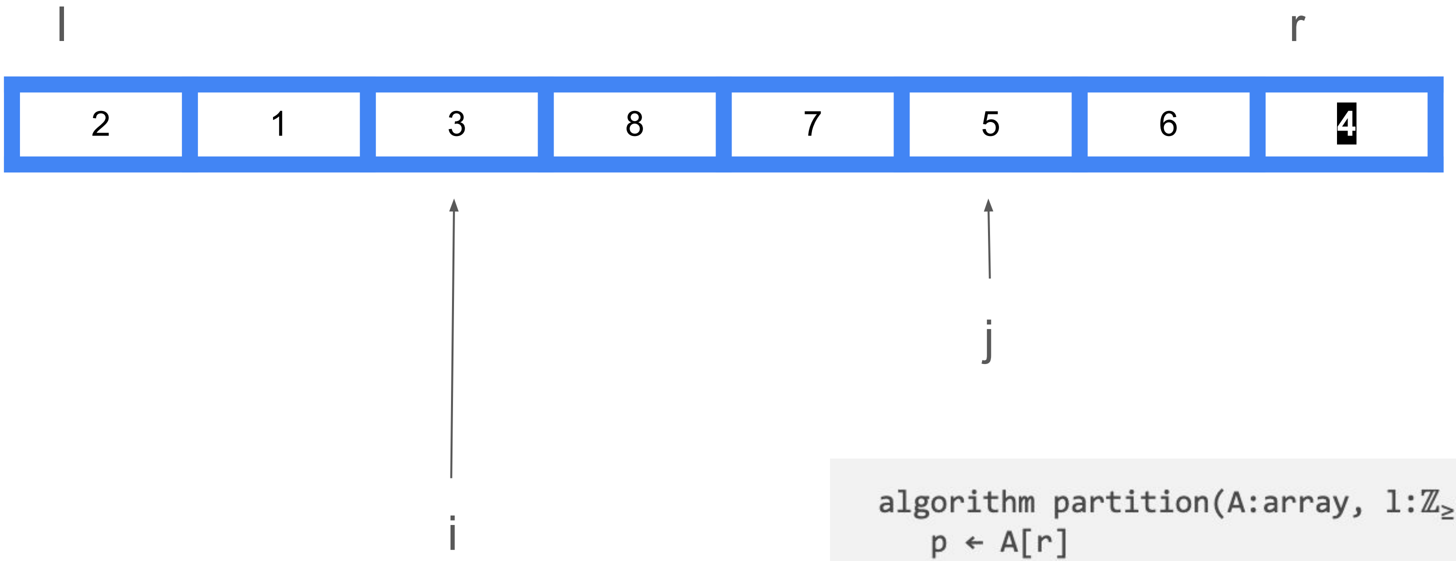
2	1	3	8	7	5	6	4

j

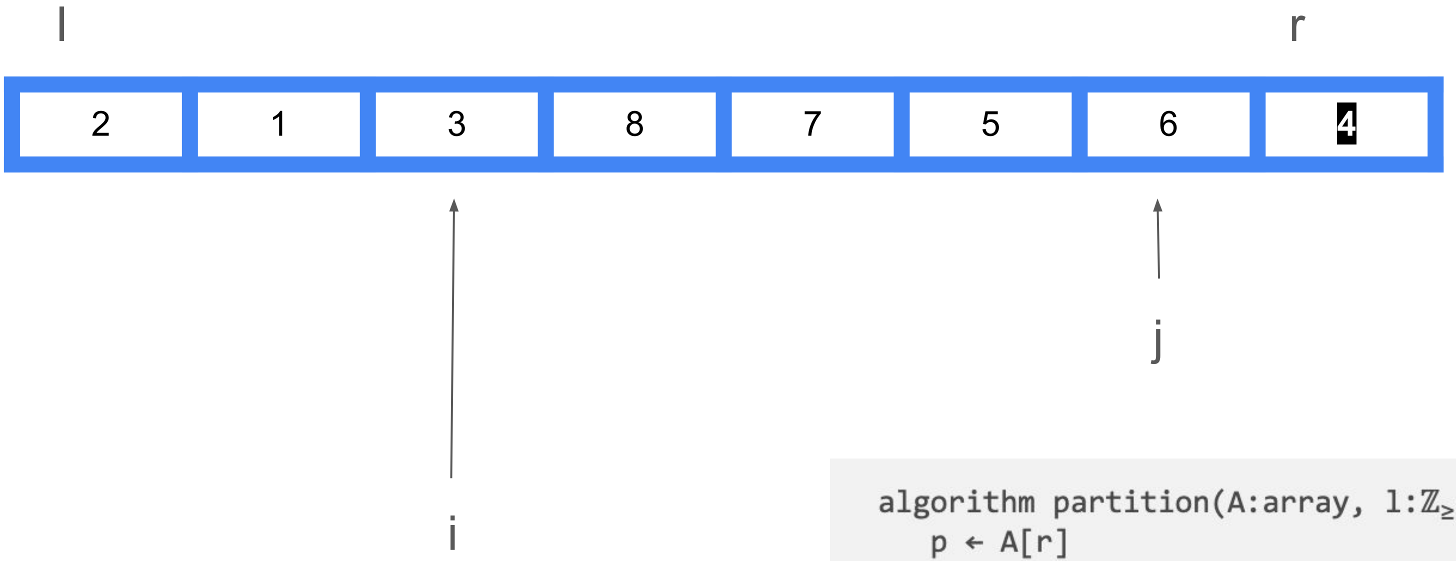
i

A[j] = 5 < 4? Nah

```
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
```

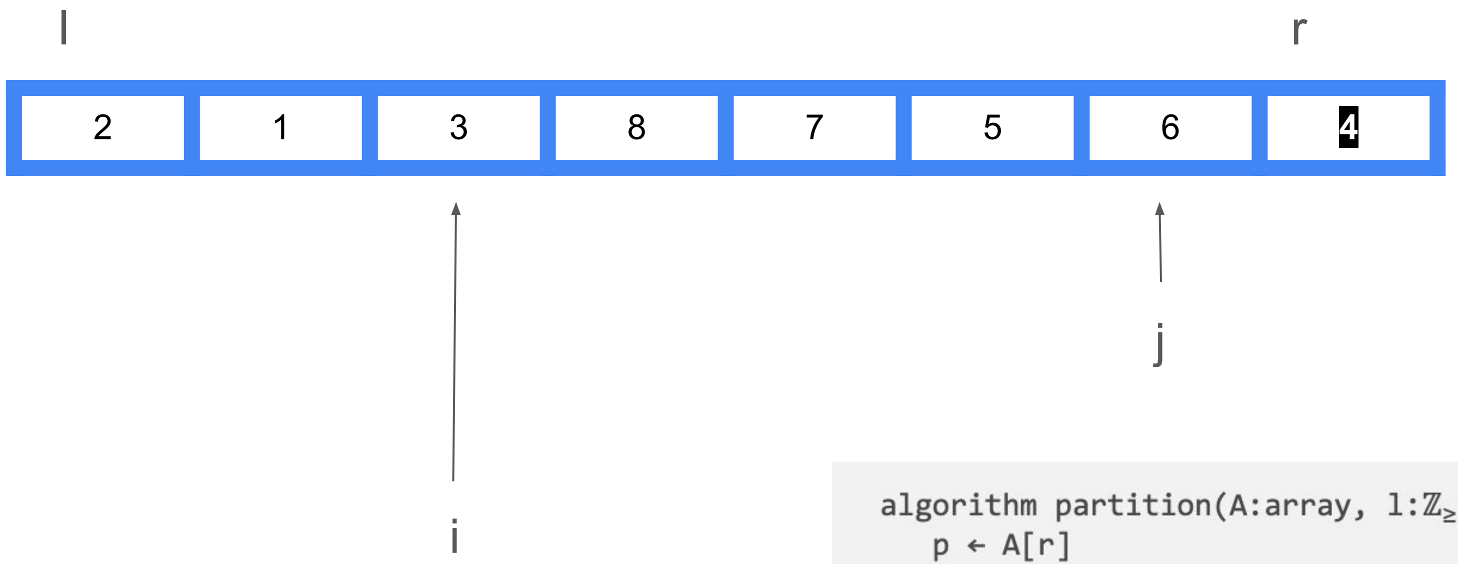


```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm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

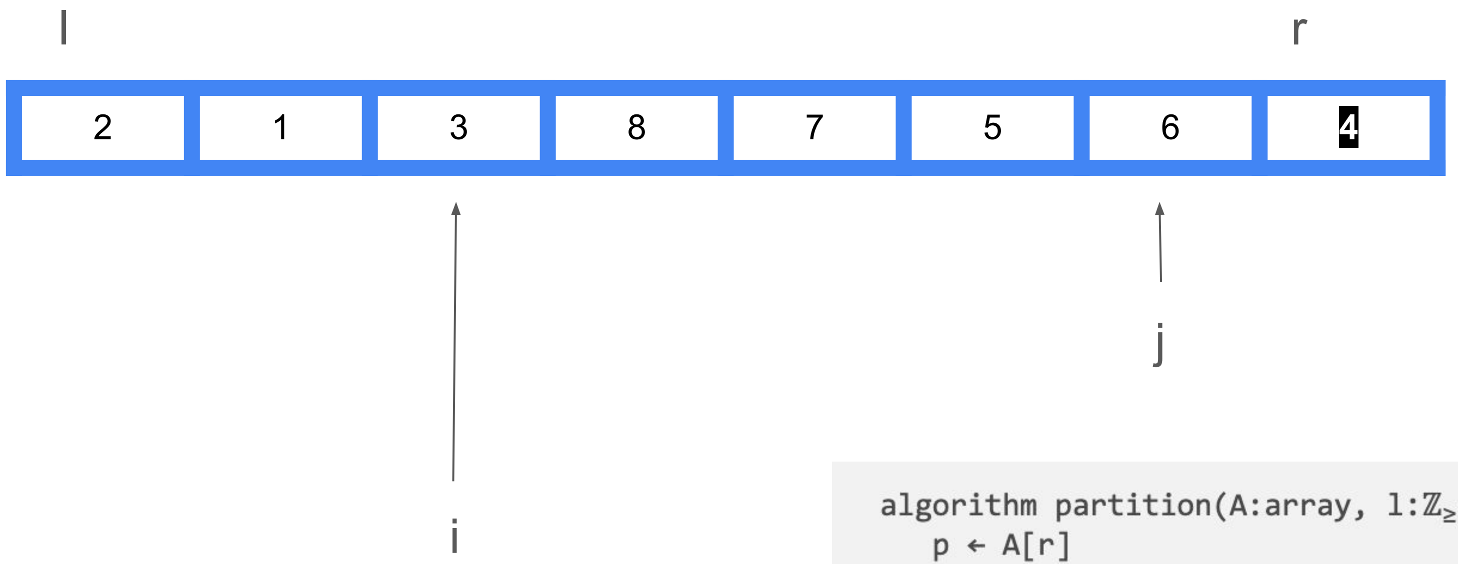


2	1	3	8	7	5	6	4

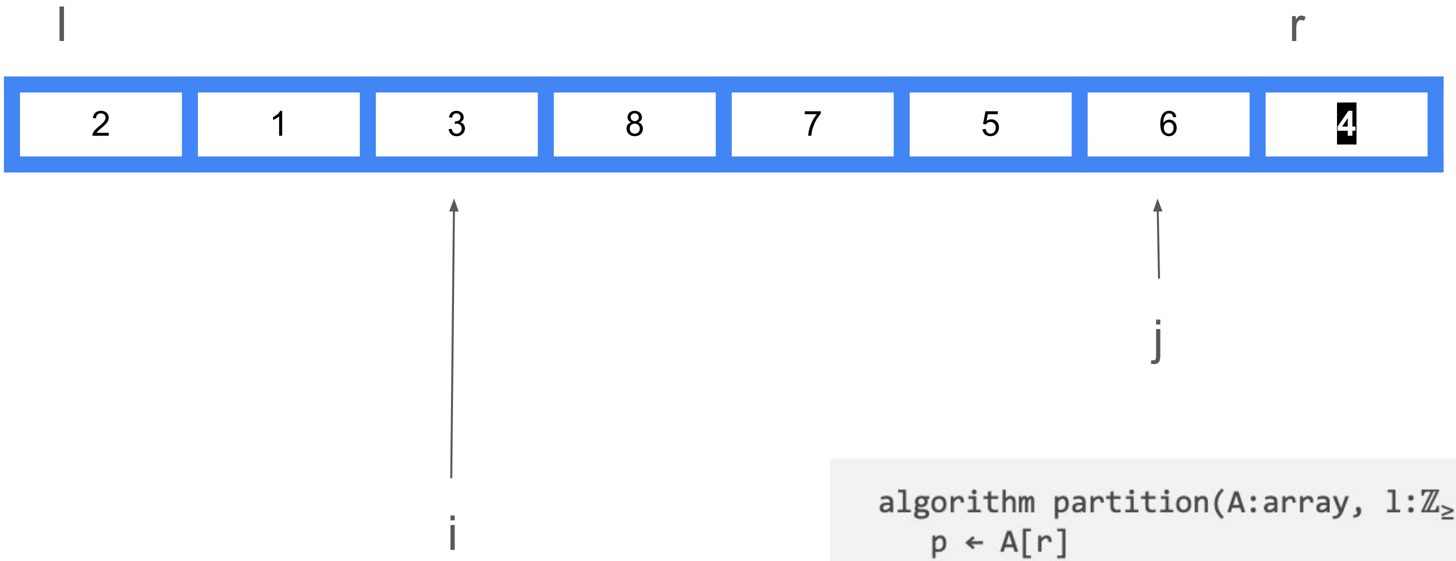
```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
    p ← A[r]
    i ← l - 1
    for j from l to r - 1 do
        if A[j] < p then
            i ← i + 1
            swap(A, i, j)
        end if
    end for
    i ← i + 1
    swap(A, i, r)
    return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



```
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
```



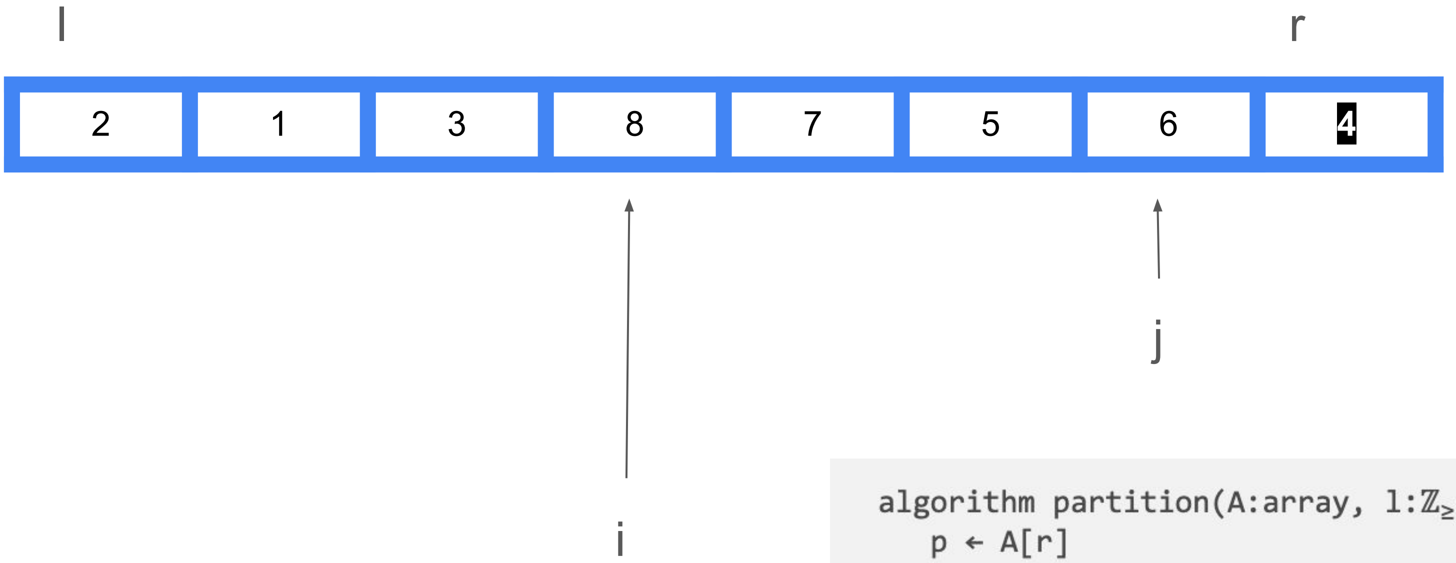
```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm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



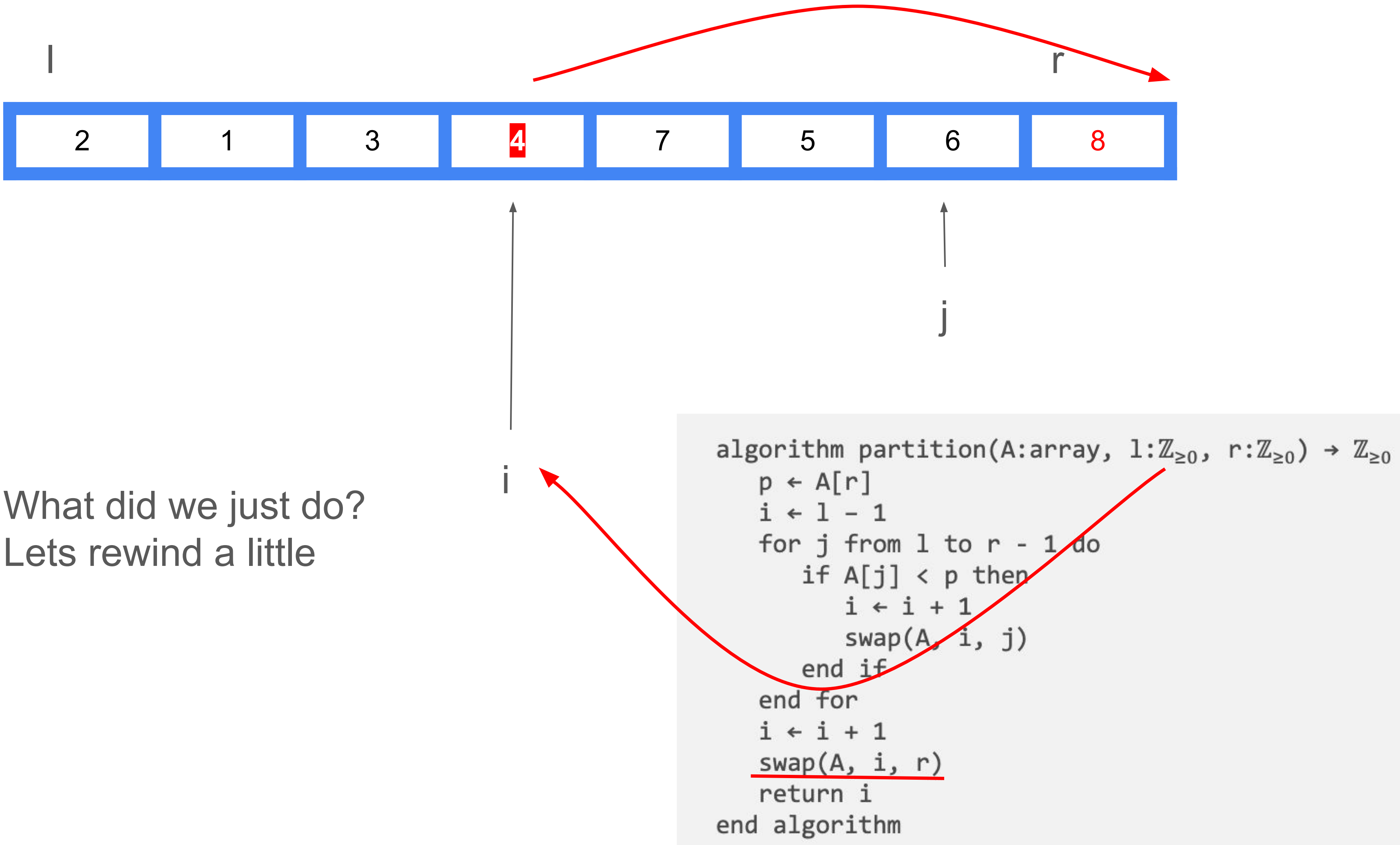
2	1	3	8	7	5	6	4

That was the last for loop iteration

```
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
```

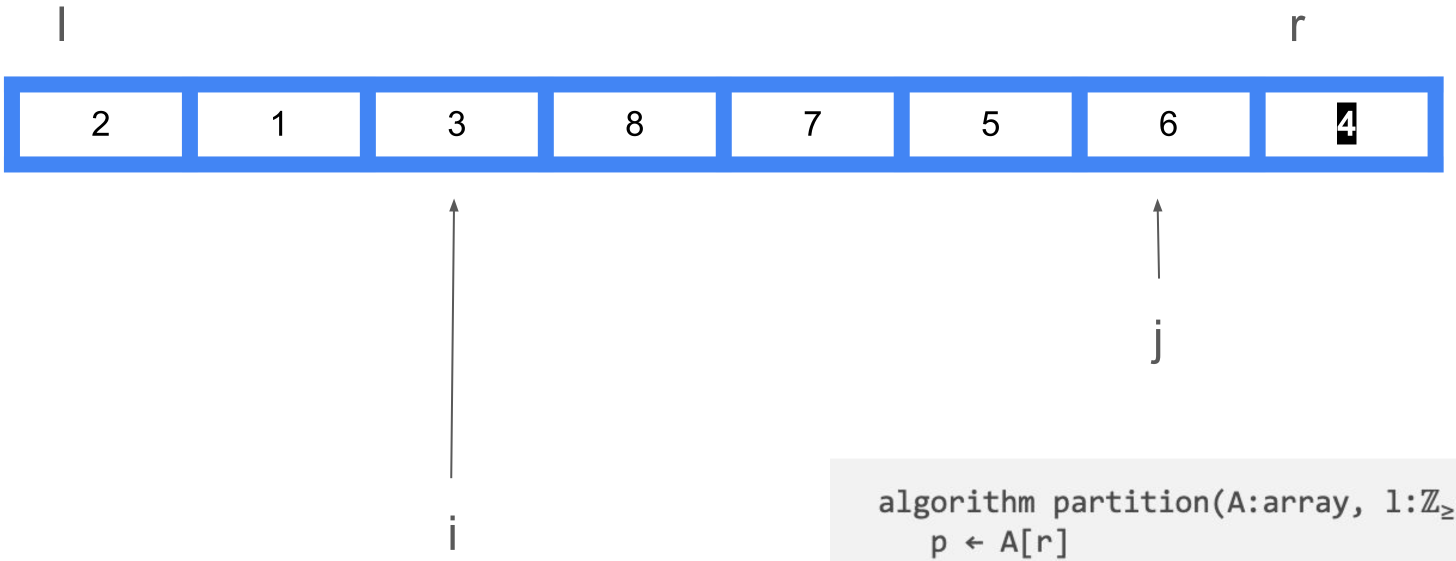


```
algorithm partition(A:array, l:$\mathbb{Z}_{\geq 0}$, r:$\mathbb{Z}_{\geq 0}$) → $\mathbb{Z}_{\geq 0}$
   p ← A[r]
   i ← l - 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm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



```
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
```

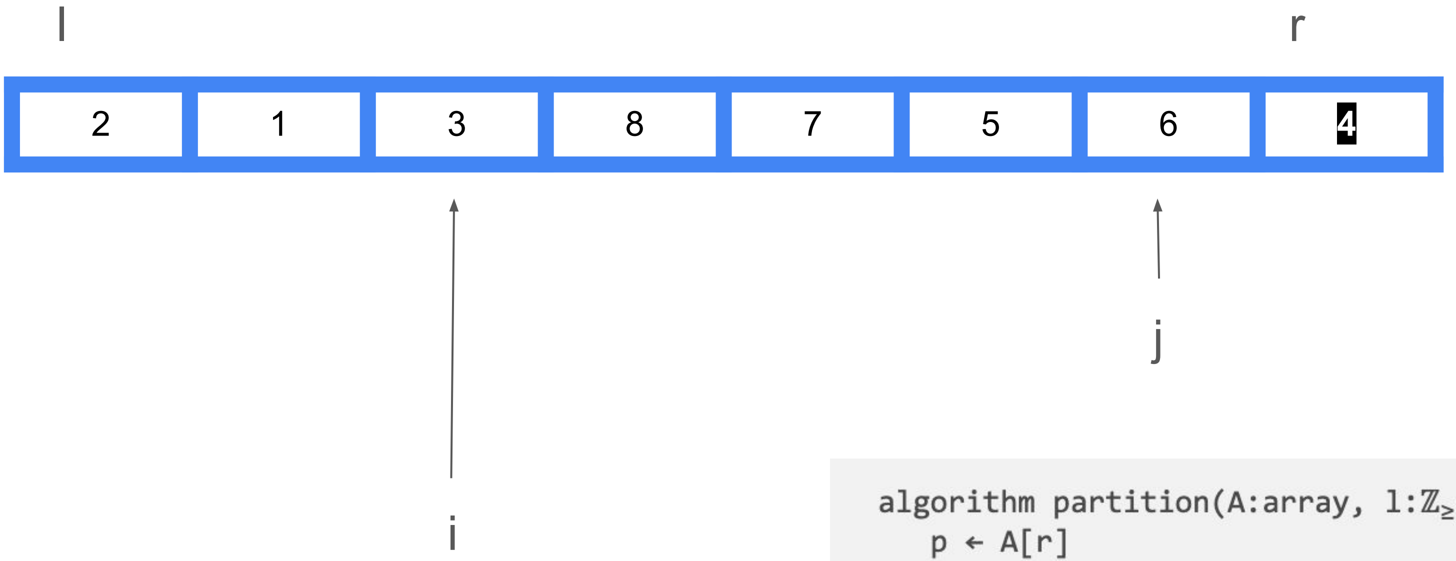
What did we just do?
Lets rewind a little



That was the last for loop iteration

Everything up to i is less than the pivot

```
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
```

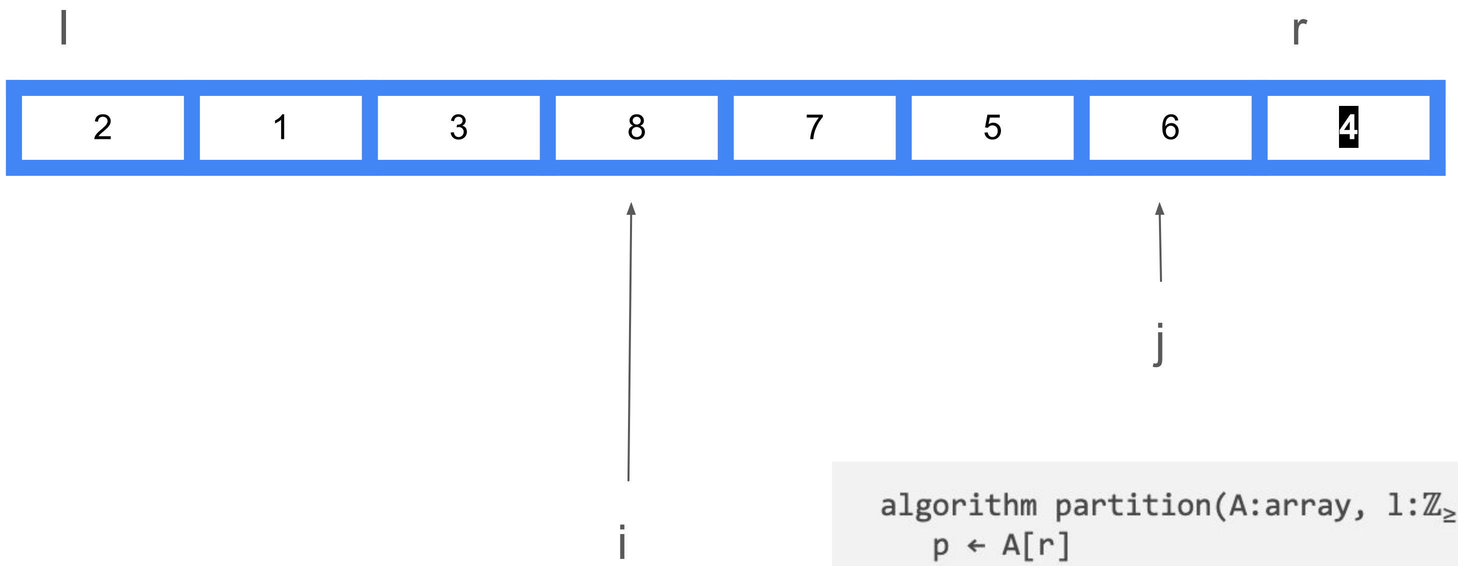


That was the last for loop iteration

Everything up to i is less than the pivot

pivot index = i + 1

```
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
```



l

r

2	1	3	8	7	5	6	4

j

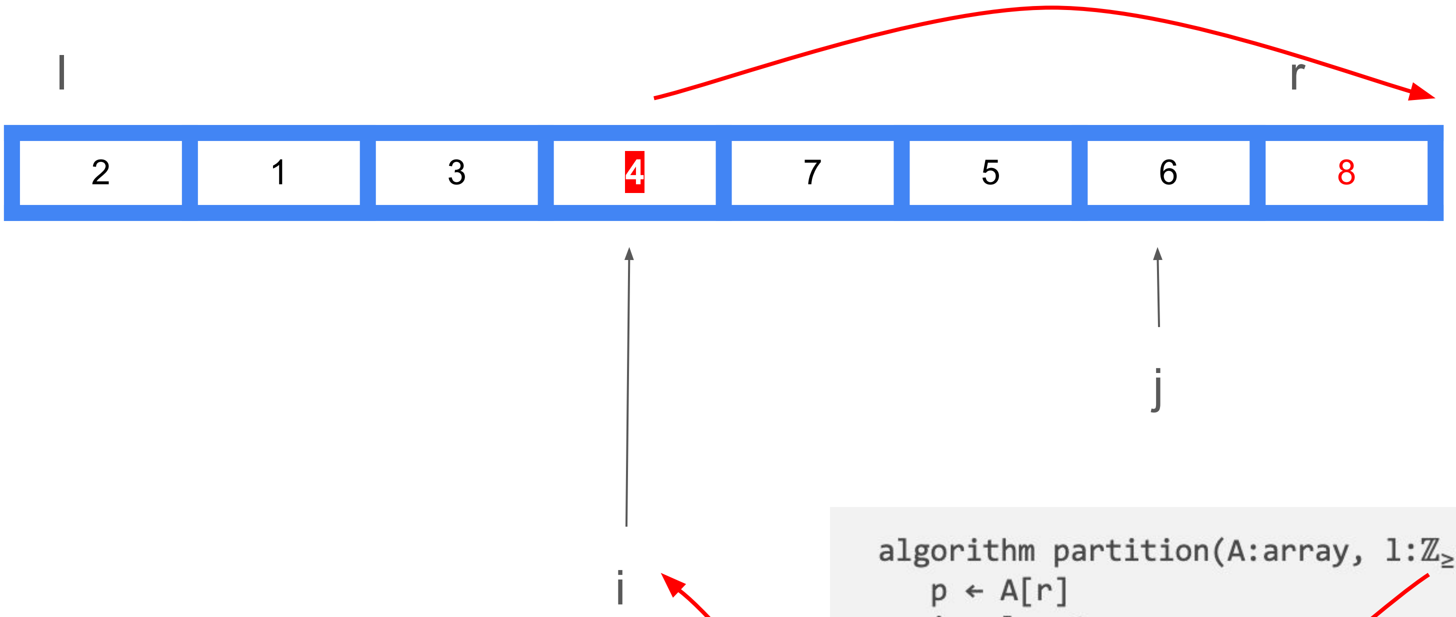
i

```
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 was the last for loop iteration

Everything up to i is less than the pivot

pivot index = i + 1



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
```

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?

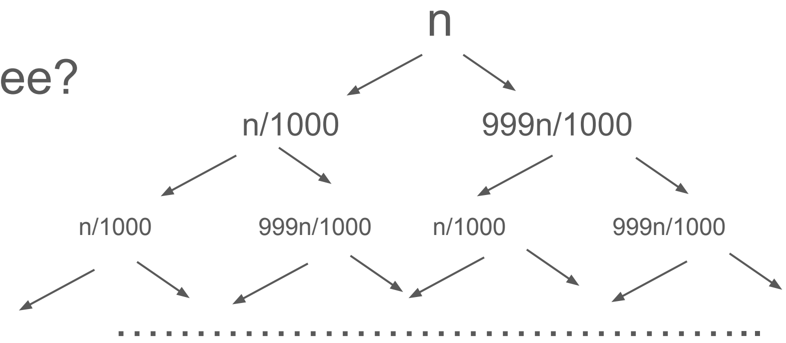
Average case of quick sort is close to O(nlogn)

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?



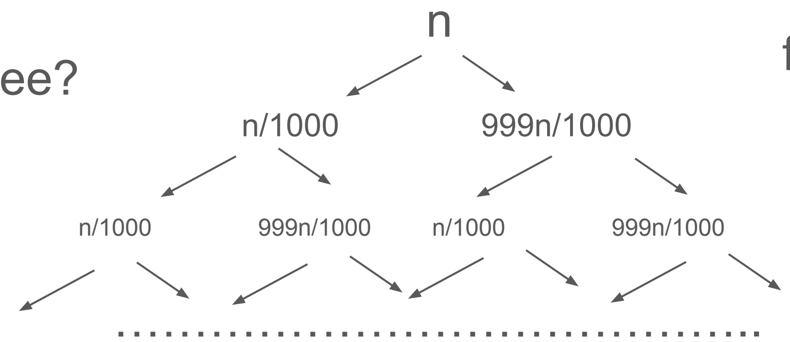
Average case of quick sort is close to O(nlogn)

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?



n

n/1000 999n/1000

n/1000 999n/1000 n/1000 999n/1000

……………………………………

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

Question 3

(Counting sort)

(1) Illustrate the operations of Counting sort on $A = [6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2]$.

(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

Step 1: Array C keeps the number of occurrences for each element in A.

Step 2: Count the occurrences of each item in A. Use A[i] as the indices of C.

Step 3: Accumulate the count values in C from left to right.

Step 4: Use values in C to determine the final index for each element in A.

Step 5 (optional): Copy the elements from B to A if they must be in the original array.

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	0	0	0	0	0	0	0

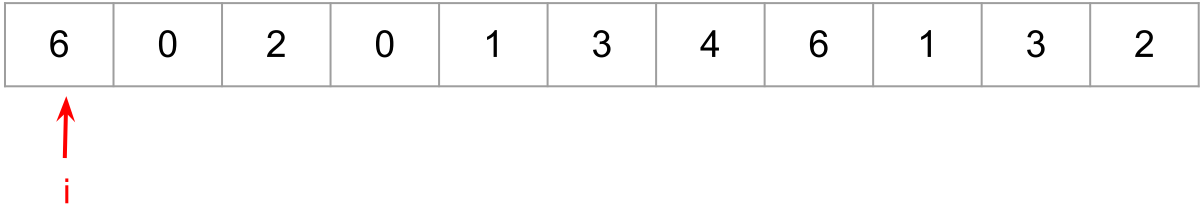
```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	0	0	0	0	0	0	0

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

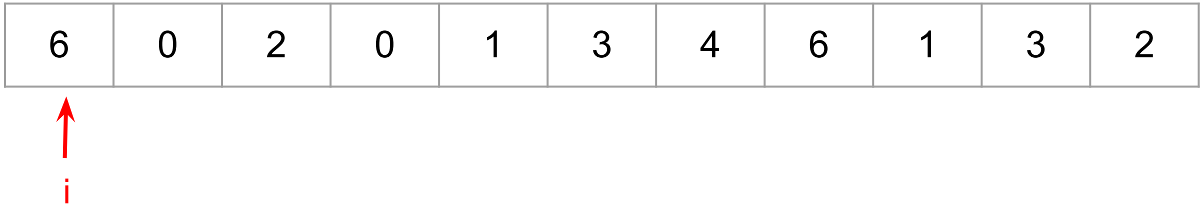


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	0	0	0	0	0	0	0

```
algorithm countingsort(A:array, k:ℤ⁺)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```



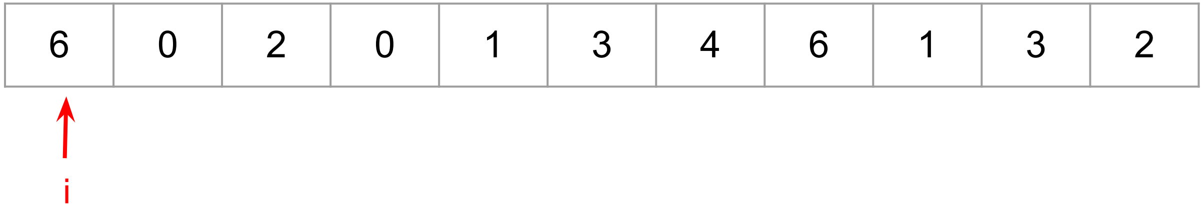
6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	0	0	0	0	0	0	0

A[i] = 6

```
algorithm countingsort(A:array, k:ℤ⁺)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

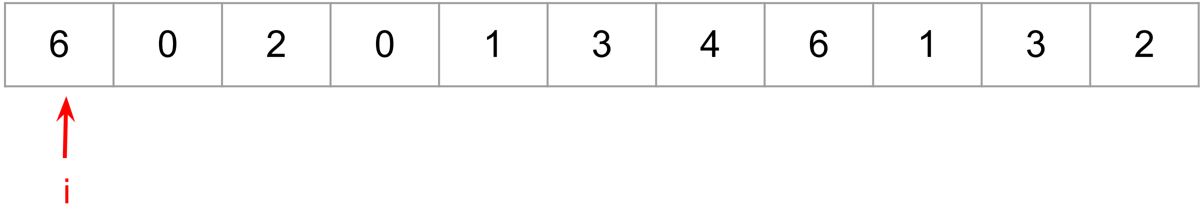
Initialize our C array

k	0	1	2	3	4	5	6
freq	0	0	0	0	0	0	0

A[i] = 6

C[A[i]] = C[6]

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

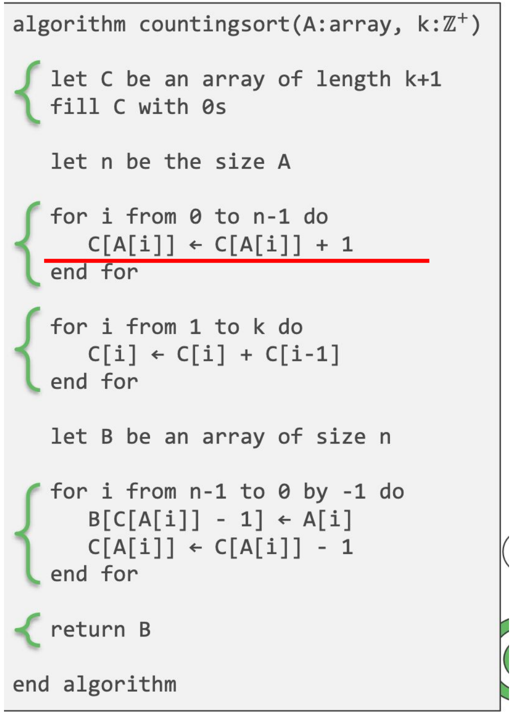
Initialize our C array

k	0	1	2	3	4	5	6
freq	0	0	0	0	0	0	1

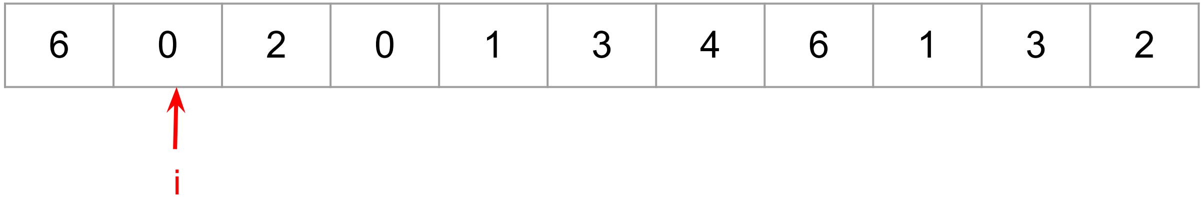
A[i] = 6

C[A[i]] = C[6]

C[6] += 1



```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

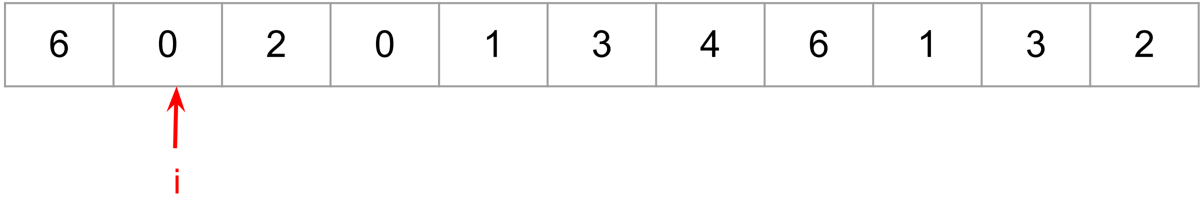


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	0	0	0	0	0	0	1

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
       C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
       C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
       B[C[A[i]] - 1] ← A[i]
       C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



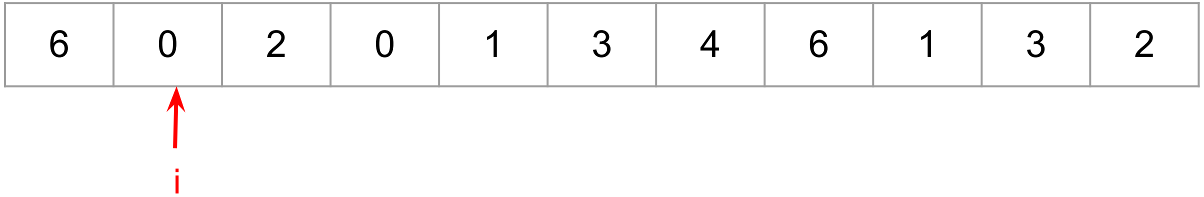
6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	0	0	0	0	0	0	1

$A[i] = 0$

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

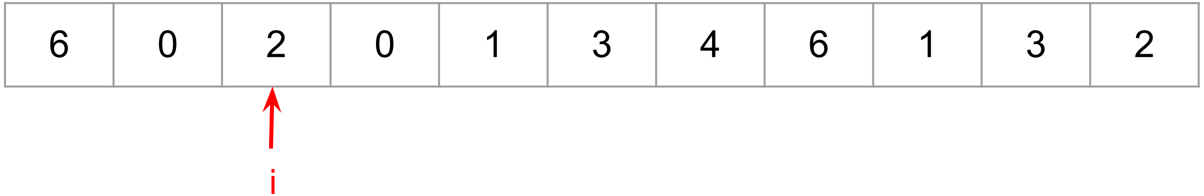
Initialize our C array

k	0	1	2	3	4	5	6
freq	1	0	0	0	0	0	1

A[i] = 0

C[0] += 1

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

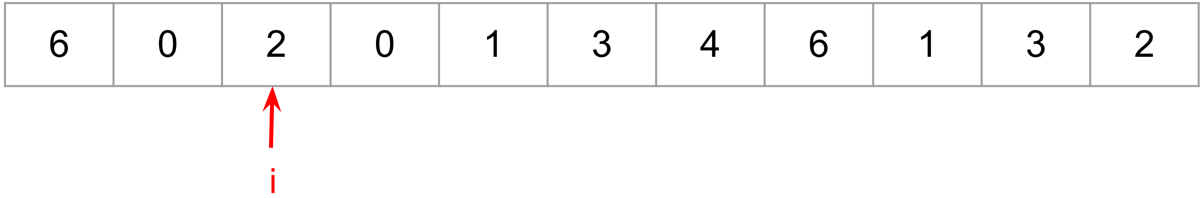


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	1	0	0	0	0	0	1

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

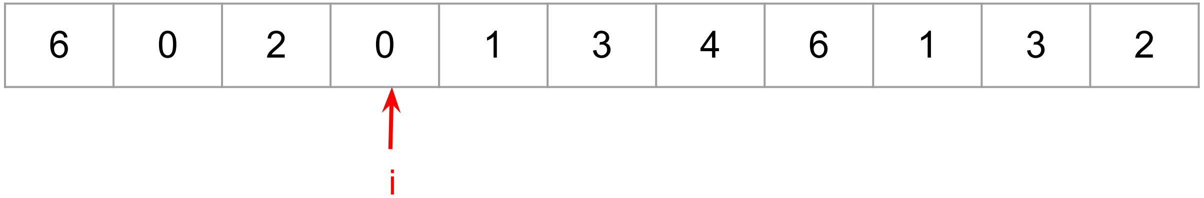


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	1	0	1	0	0	0	1

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
       C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
       C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
       B[C[A[i]] - 1] ← A[i]
       C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

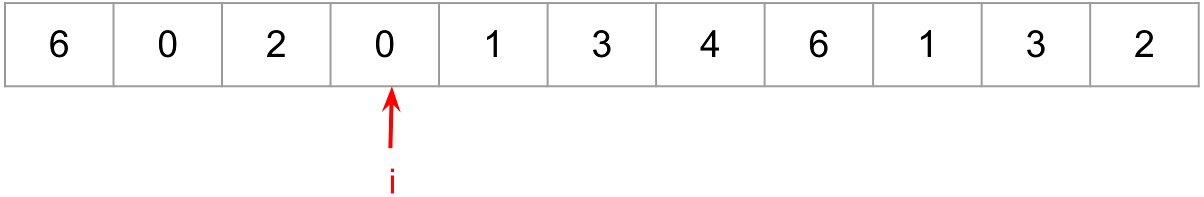


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	1	0	1	0	0	0	1

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



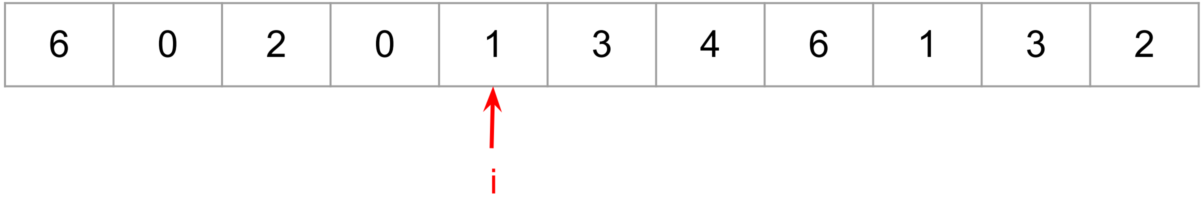
6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	2	0	1	0	0	0	1

Going faster..

```
algorithm countingsort(A:array, k:ℤ⁺)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```

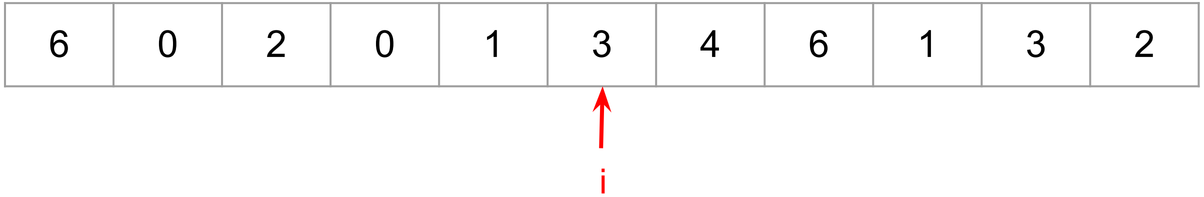


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	2	1	1	0	0	0	1

```
algorithm countingsort(A:array, k:ℤ⁺)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```

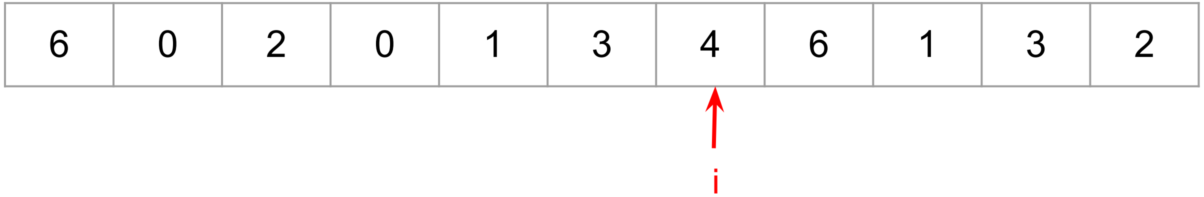


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	2	1	1	1	0	0	1

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

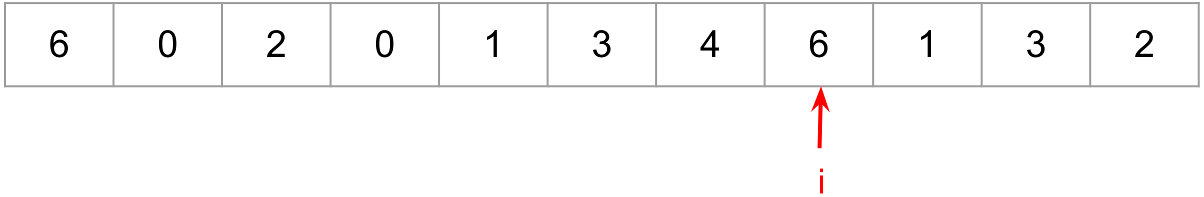


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	2	1	1	1	1	0	1

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

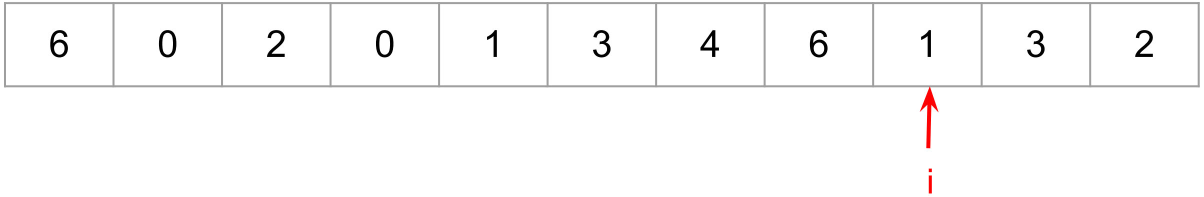


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	2	1	1	1	1	0	2

```
algorithm countingsort(A:array, k:ℤ⁺)

let C be an array of length k+1
fill C with 0s

let n be the size A

for i from 0 to n-1 do
   C[A[i]] ← C[A[i]] + 1
end for

for i from 1 to k do
   C[i] ← C[i] + C[i-1]
end for

let B be an array of size n

for i from n-1 to 0 by -1 do
   B[C[A[i]] - 1] ← A[i]
   C[A[i]] ← C[A[i]] - 1
end for

return B

end algorithm
```

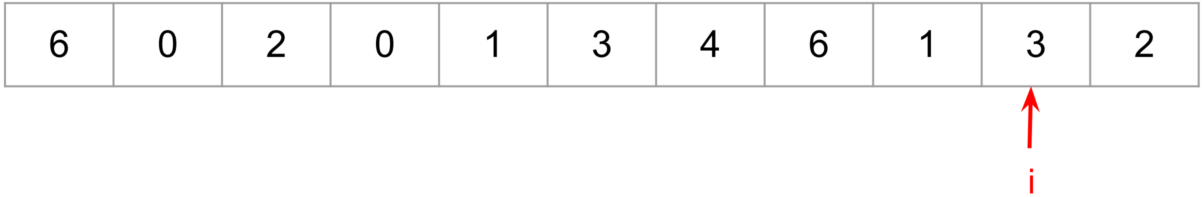


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	2	2	1	1	1	0	2

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

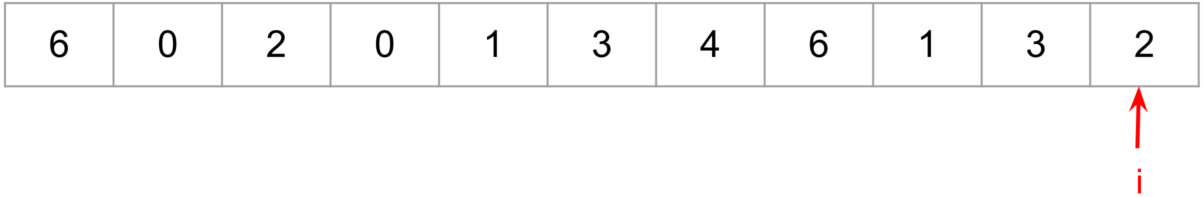


6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	2	2	1	2	1	0	2

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	2	2	2	2	1	0	2

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

Initialize our C array

k	0	1	2	3	4	5	6
freq	2	2	2	2	1	0	2

Next up

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	2	2	2	1	0	2

↑

i

```
algorithm countingsort(A:array, k:ℤ⁺)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	2	2	2	1	0	2

↑

i

$$C[1] = C[1] + C[0]$$

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	2	2	1	0	2

$$C[1] = C[1] + C[0] = 2 + 2$$

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	2	2	1	0	2

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	2	2	1	0	2

C[2] += C[1]

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	2	1	0	2

C[2] += C[1] = 6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

let C be an array of length k+1
fill C with 0s

let n be the size A

for i from 0 to n-1 do
    C[A[i]] ← C[A[i]] + 1
end for

for i from 1 to k do
    C[i] ← C[i] + C[i-1]
end for

let B be an array of size n

for i from n-1 to 0 by -1 do
    B[C[A[i]] - 1] ← A[i]
    C[A[i]] ← C[A[i]] - 1
end for

return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	2	1	0	2

C[2] += C[1] = 6

C[i - 1] = # of elements before i in the sorted array

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	1	0	2

↑

i

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	0	2

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	2

i

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

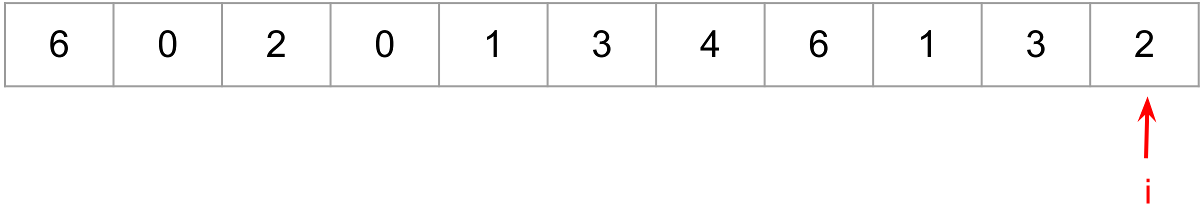

```
algorithm countingsort(A:array, k:ℤ⁺)

let C be an array of length k+1
fill C with 0s

let n be the size A

for i from 0 to n-1 do
   C[A[i]] ← C[A[i]] + 1
end for

for i from 1 to k do
   C[i] ← C[i] + C[i-1]
end for

let B be an array of size n

for i from n-1 to 0 by -1 do
   B[C[A[i]] - 1] ← A[i]
   C[A[i]] ← C[A[i]] - 1
end for

return B

end algorithm
```

6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```



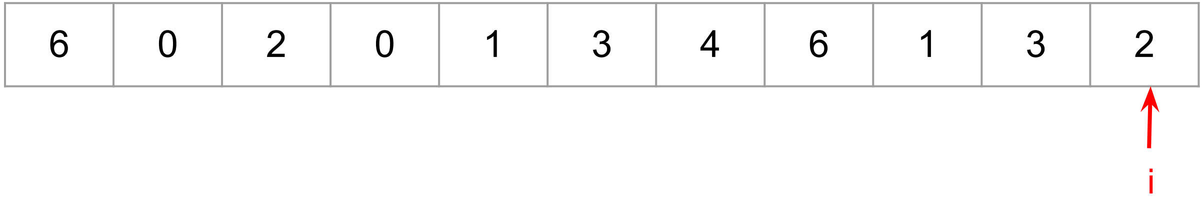
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

A[i] = 2

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



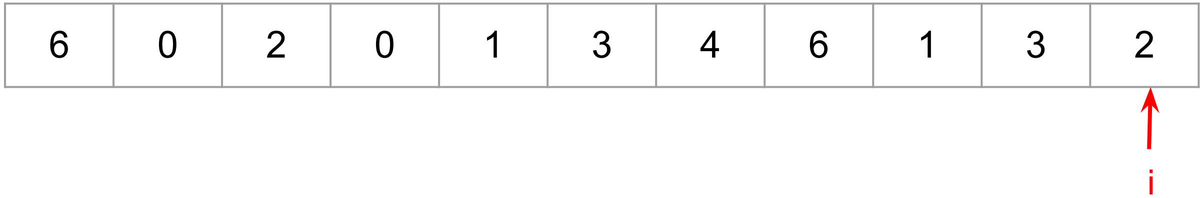
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

A[i] = 2
C[A[i]] = 6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
       C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
       C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
       B[C[A[i]] - 1] ← A[i]
       C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



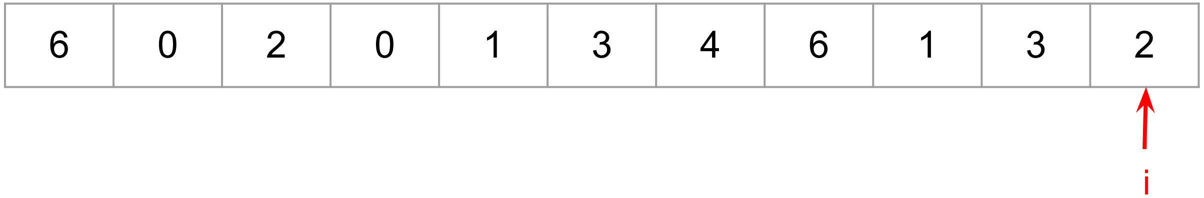
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

A[i] = 2
C[A[i]] = 6

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

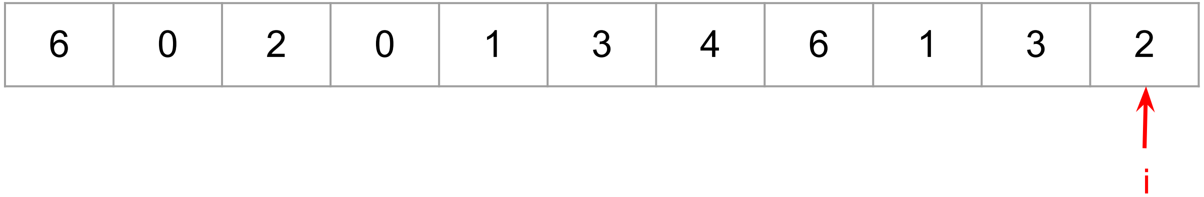
B

A[i] = 2

C[A[i]] = 6

B[C[A[i]] - 1] = B[5]

```
algorithm countingsort(A:array, k:ℤ⁺)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

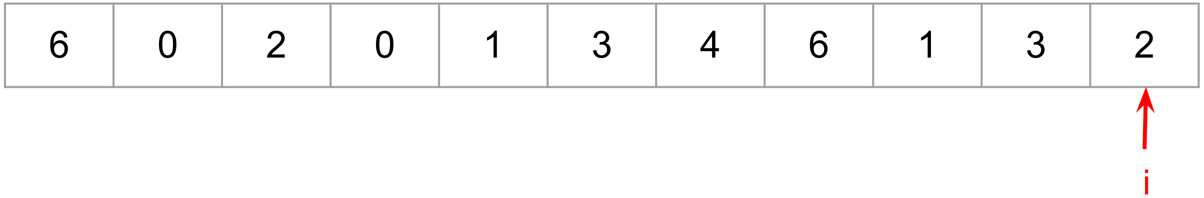
					2					

A[i] = 2

C[A[i]] = 6

B[C[A[i]] - 1] = B[5]   Set B[5] = A[i] = 2

```
algorithm countingsort(A:array, k:ℤ⁺)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

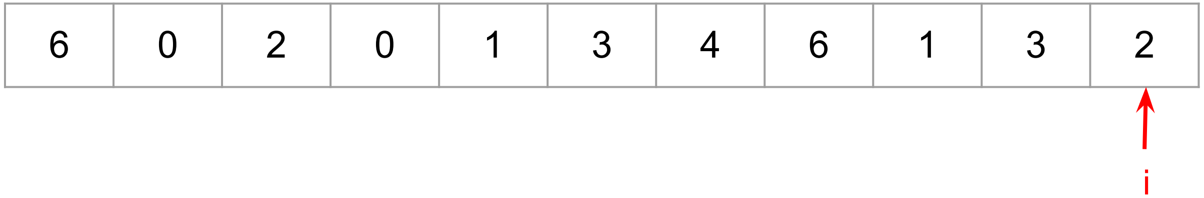
C[i] = # elements less than or equal to i

B

					2					

A[i] = 2
C[A[i]] = 6
B[C[A[i]] - 1] = B[5] Set B[5] = A[i] = 2 **why?**

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

C[i] = # elements less than or equal to i

When sorted, elements before B[5] look like..

B

0	0	1	1	2	2					

A[i] = 2

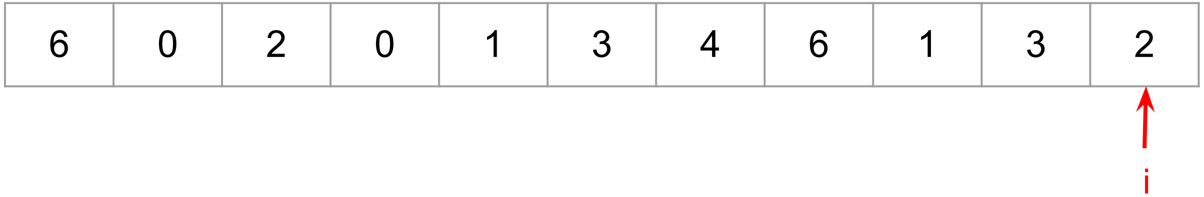
C[A[i]] = 6

B[C[A[i]] - 1] = B[5]

Set B[5] = A[i] = 2

why?

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

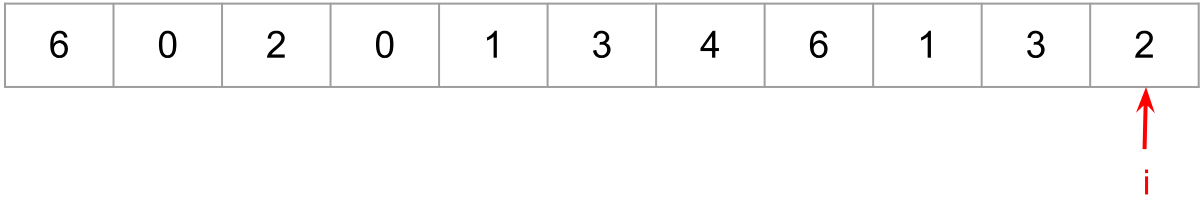
k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

0	0	1	1	2	2					

A[i] = 2

```
algorithm countingsort(A:array, k:ℤ+)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

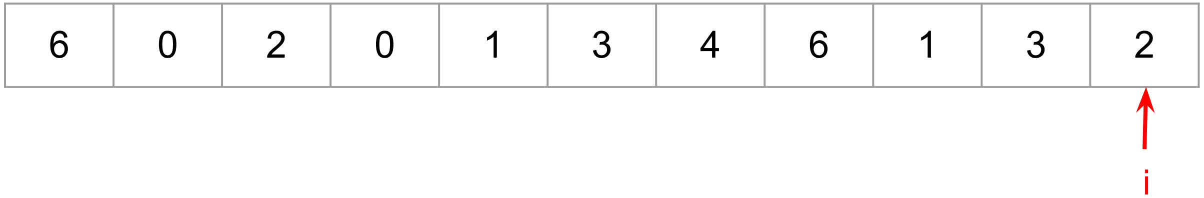
k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

0	0	1	1	2	2					

A[i] = 2
C[A[i]] = C[2]

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	5	8	9	9	11

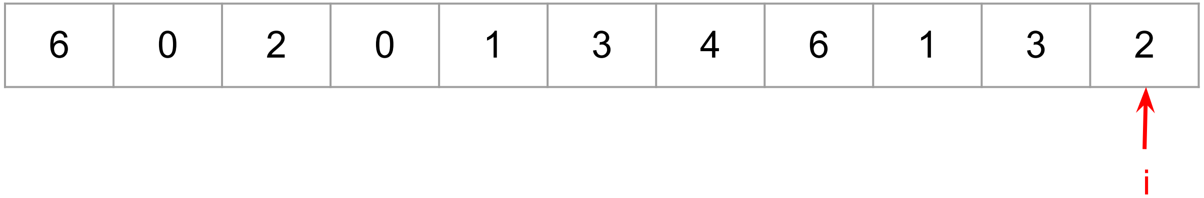
B

0	0	1	1	2	2					

A[i] = 2

C[A[i]] = C[2] **C[2] -= 1**

```
algorithm countingsort(A:array, k:ℤ⁺)

let C be an array of length k+1
fill C with 0s

let n be the size A

for i from 0 to n-1 do
   C[A[i]] ← C[A[i]] + 1
end for

for i from 1 to k do
   C[i] ← C[i] + C[i-1]
end for

let B be an array of size n

for i from n-1 to 0 by -1 do
   B[C[A[i]] - 1] ← A[i]
   C[A[i]] ← C[A[i]] - 1
end for

return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

Intuition: We placed the first 2 down, only one 2 left

k	0	1	2	3	4	5	6
freq	2	4	5	8	9	9	11

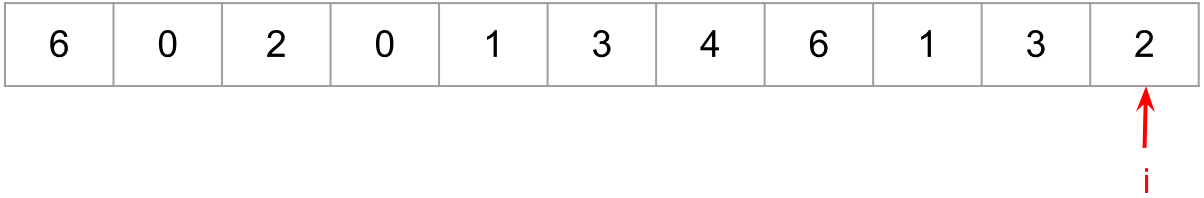
B

0	0	1	1	2	2					

A[i] = 2

C[A[i]] = C[2]   **C[2] -= 1**

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

Intuition: We placed the first 2 down, only **C[2] - C[1]** 2 left

k	0	1	2	3	4	5	6
freq	2	4	5	8	9	9	11

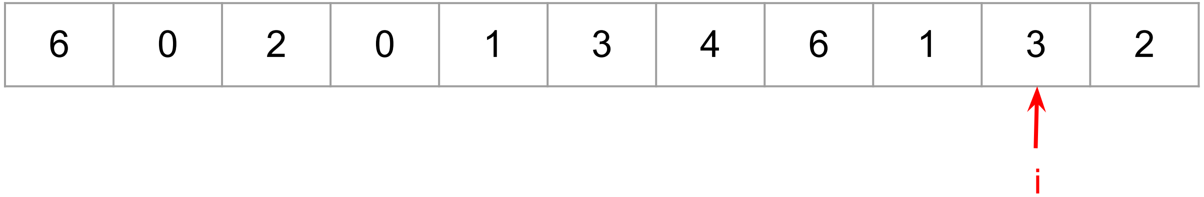
B

0	0	1	1	2	2					

A[i] = 2

C[A[i]] = C[2] **C[2] -= 1**

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



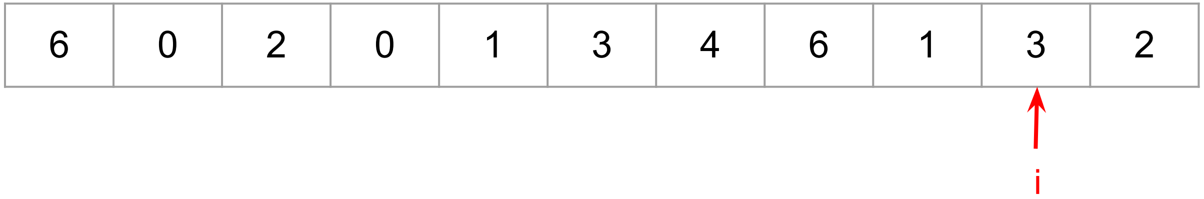
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	5	8	9	9	11

B

0	0	1	1	2	2					

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

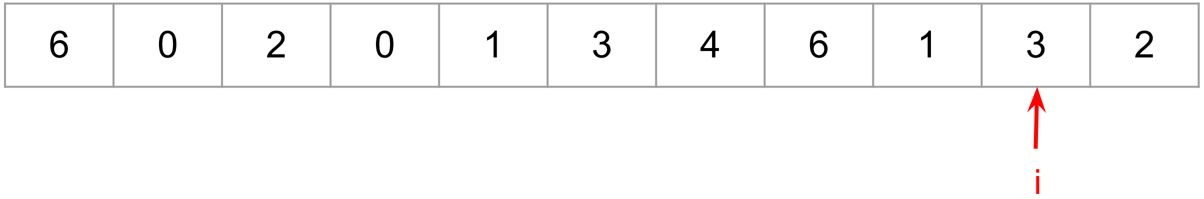
k	0	1	2	3	4	5	6
freq	2	4	5	8	9	9	11

B

0	0	1	1	2	2					

A[i] = 3

```
algorithm countingsort(A:array, k:ℤ⁺)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	5	8	9	9	11

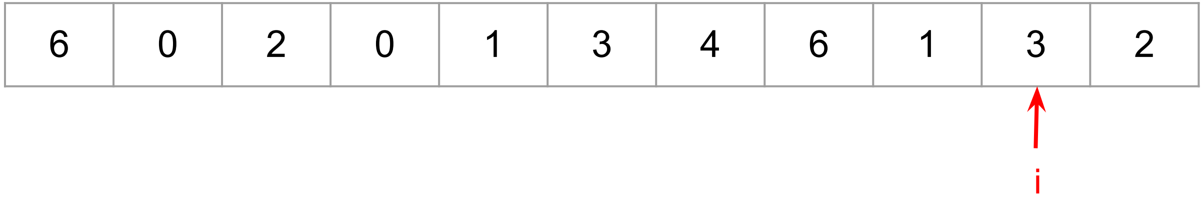
B

0	0	1	1	2	2					

A[i] = 3

C[A[i]] = C[3] = 8

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

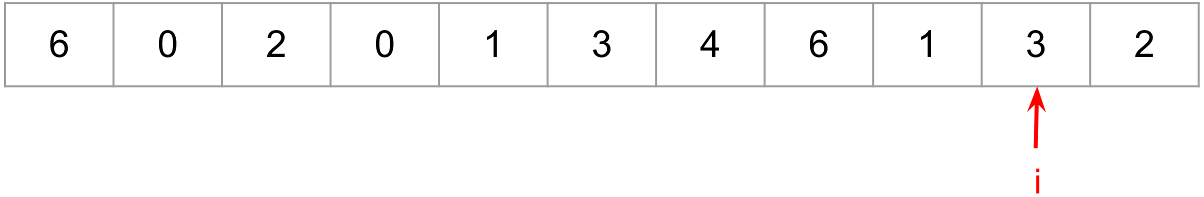
k	0	1	2	3	4	5	6
freq	2	4	5	8	9	9	11

B

0	0	1	1	2	2					

A[i] = 3
C[A[i]] = C[3] = 8
Set B[8 - 1] = 3

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	5	8	9	9	11

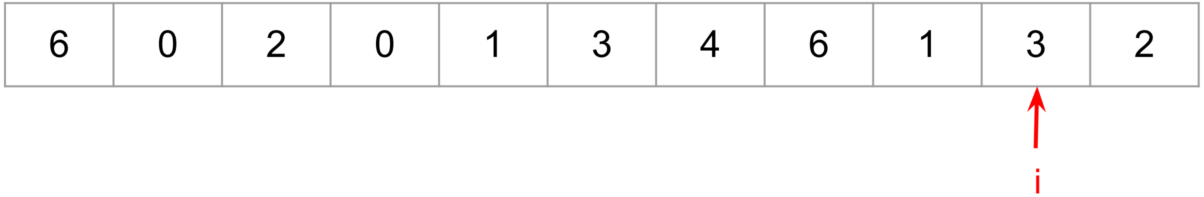
B

0	0	1	1	2	2		3			

A[i] = 3
C[A[i]] = C[3] = 8
Set B[8 - 1] = 3

why?

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	4	5	8	9	9	11

B

0	0	1	1	2	2	3	3			

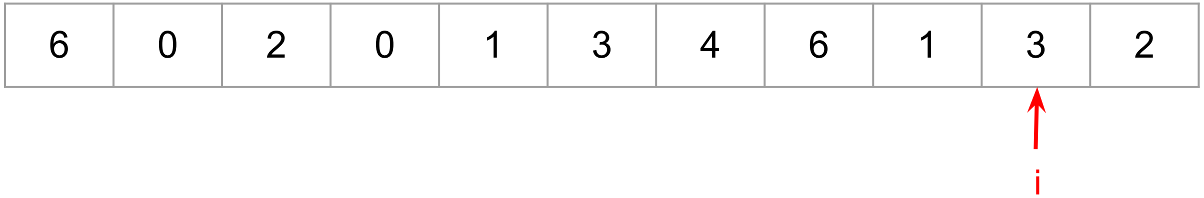
A[i] = 3

C[A[i]] = C[3] = 8

Set B[8 - 1] = 3

why?

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

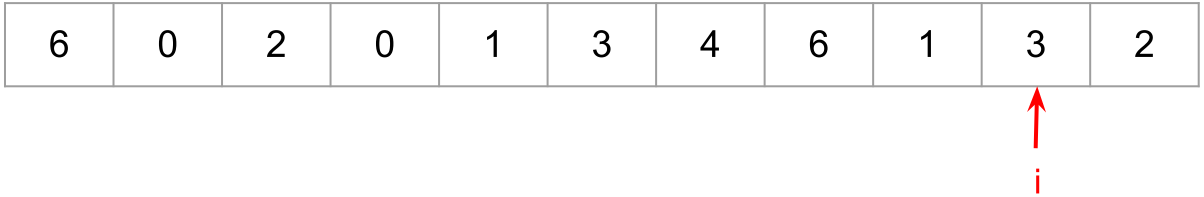
k	0	1	2	3	4	5	6
freq	2	4	5	8	9	9	11

B

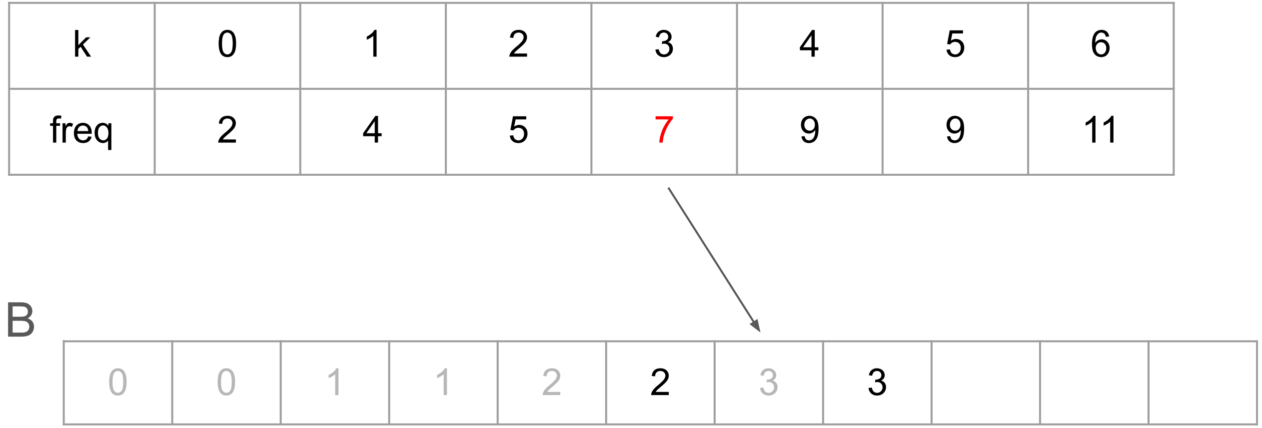
0	0	1	1	2	2	3	3			

A[i] = 3
C[A[i]] = C[3]

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2



k	0	1	2	3	4	5	6
freq	2	4	5	7	9	9	11

B

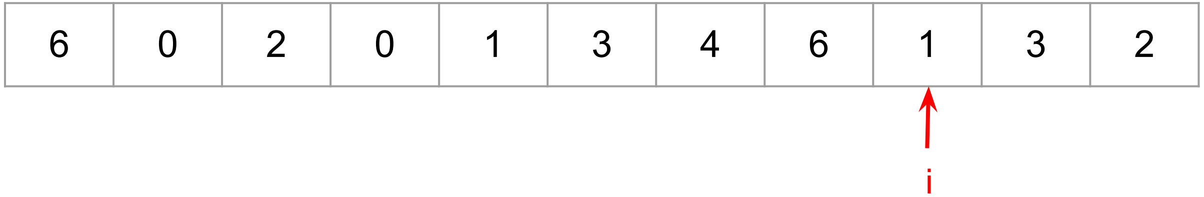
0	0	1	1	2	2	3	3			

A[i] = 3

C[A[i]] = C[3]

C[3] -= 1

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

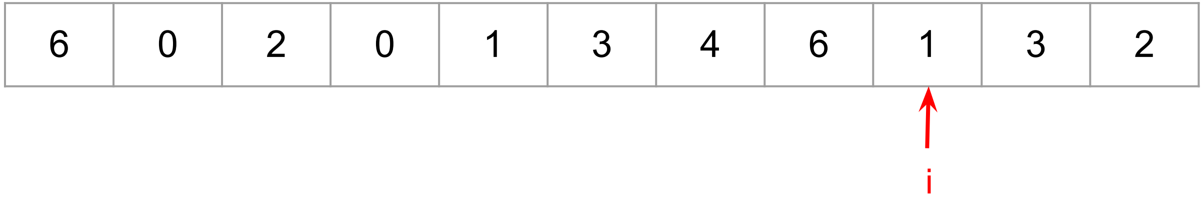


k	0	1	2	3	4	5	6
freq	2	4	5	7	9	9	11

B

0	0	1	1	2	2	3	3			

```
algorithm countingsort(A:array, k:ℤ⁺)

let C be an array of length k+1
fill C with 0s

let n be the size A

for i from 0 to n-1 do
    C[A[i]] ← C[A[i]] + 1
end for

for i from 1 to k do
    C[i] ← C[i] + C[i-1]
end for

let B be an array of size n

for i from n-1 to 0 by -1 do
    B[C[A[i]] - 1] ← A[i]
    C[A[i]] ← C[A[i]] - 1
end for

return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

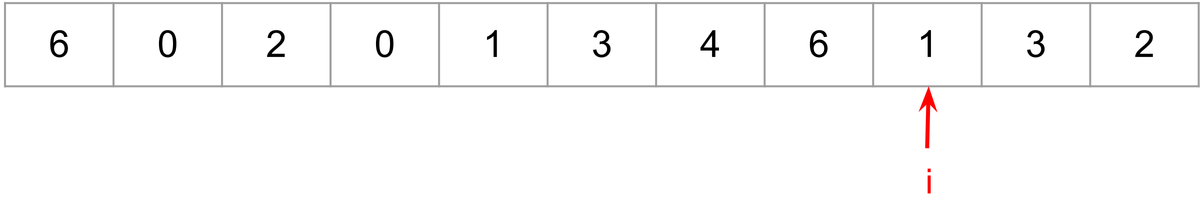
k	0	1	2	3	4	5	6
freq	2	4	5	7	9	9	11

B

0	0	1	1	2	2	3	3			

This should be B[3] from our picture

```
algorithm countingsort(A:array, k:ℤ⁺)

let C be an array of length k+1
fill C with 0s

let n be the size A

for i from 0 to n-1 do
   C[A[i]] ← C[A[i]] + 1
end for

for i from 1 to k do
   C[i] ← C[i] + C[i-1]
end for

let B be an array of size n

for i from n-1 to 0 by -1 do
   B[C[A[i]] - 1] ← A[i]
   C[A[i]] ← C[A[i]] - 1
end for

return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

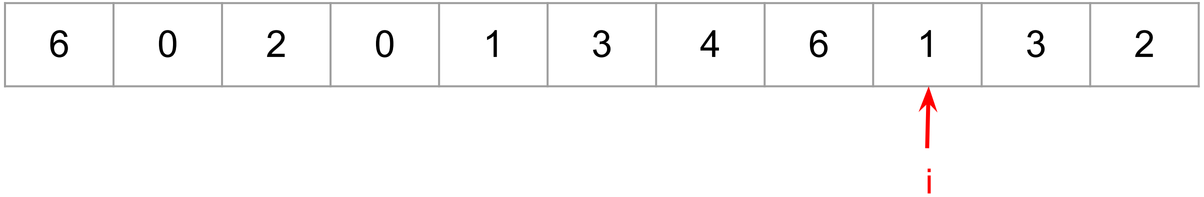
k	0	1	2	3	4	5	6
freq	2	4	5	7	9	9	11

B

0	0	1	1	2	2	3	3			

This should be B[3] from our picture
A[i] = 1, C[A[i]] = 4, so true

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

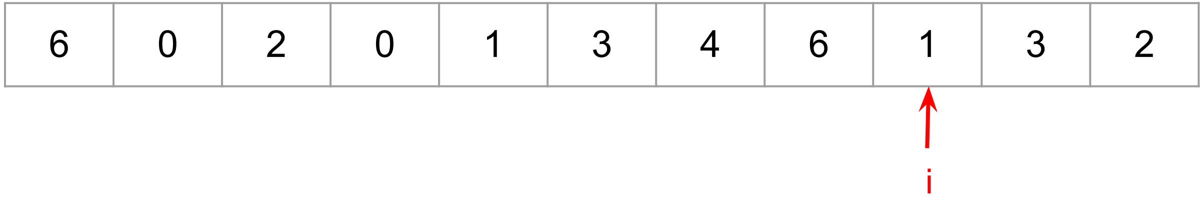
k	0	1	2	3	4	5	6
freq	2	4	5	7	9	9	11

B

0	0	1	1	2	2	3	3			

This should be B[3] from our picture
A[i] = 1, C[A[i]] = 4, so true

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	7	9	9	11

B

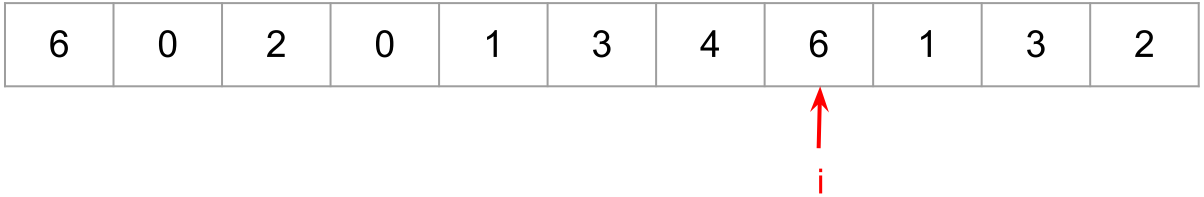
0	0	1	1	2	2	3	3			

This should be B[3] from our picture

A[i] = 1, C[A[i]] = 4, so true

Then decrement the count

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



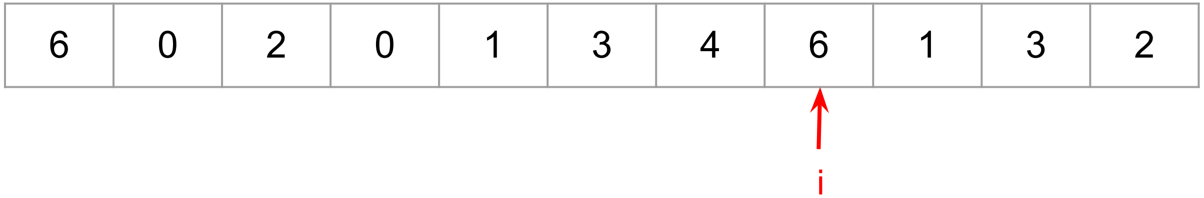
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	7	9	9	11

B

0	0	1	1	2	2	3	3			

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

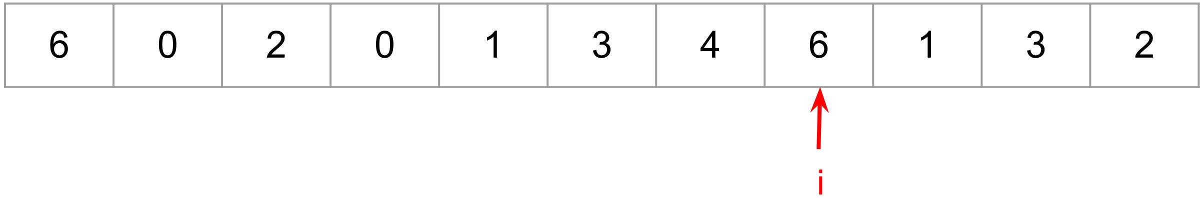
k	0	1	2	3	4	5	6
freq	2	3	5	7	9	9	11

B

0	0	1	1	2	2	3	3			

C[6] = 11

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

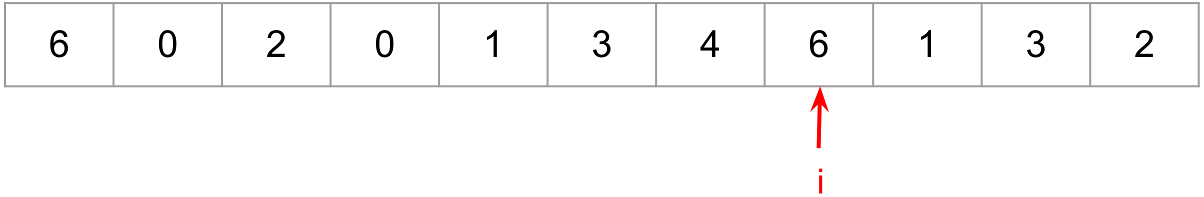
k	0	1	2	3	4	5	6
freq	2	3	5	7	9	9	11

B

0	0	1	1	2	2	3	3			6

C[6] = 11

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

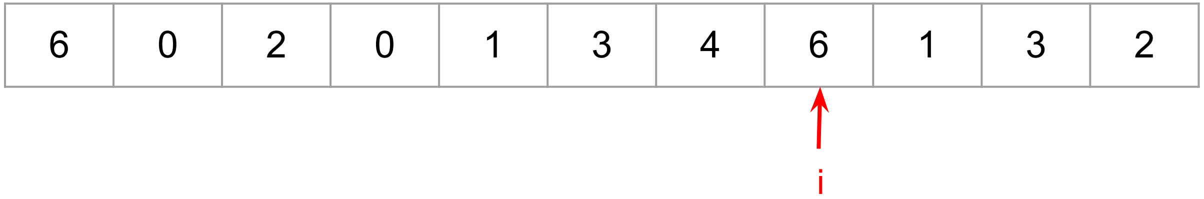
k	0	1	2	3	4	5	6
freq	2	3	5	7	9	9	11

B

0	0	1	1	2	2	3	3	4	6	6

C[6] = 11

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



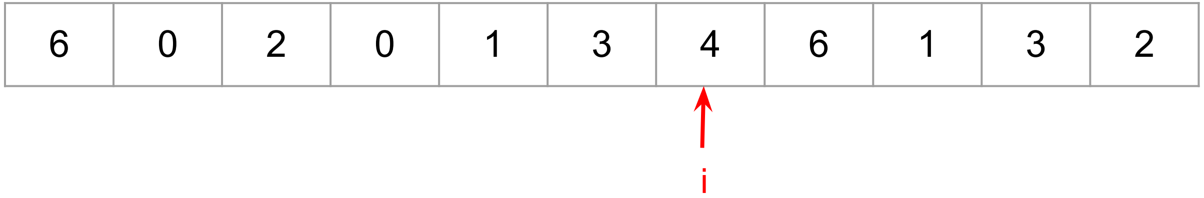
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	7	9	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



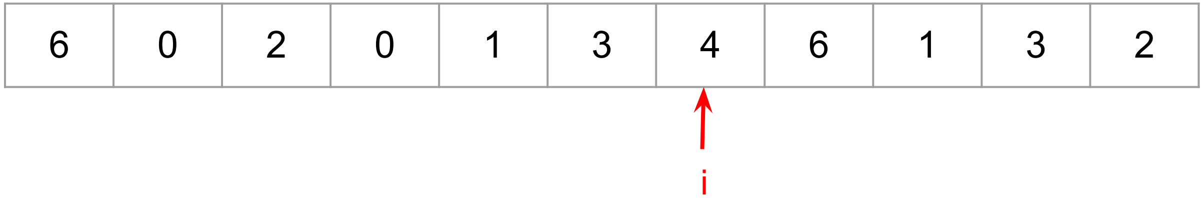
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	7	9	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```



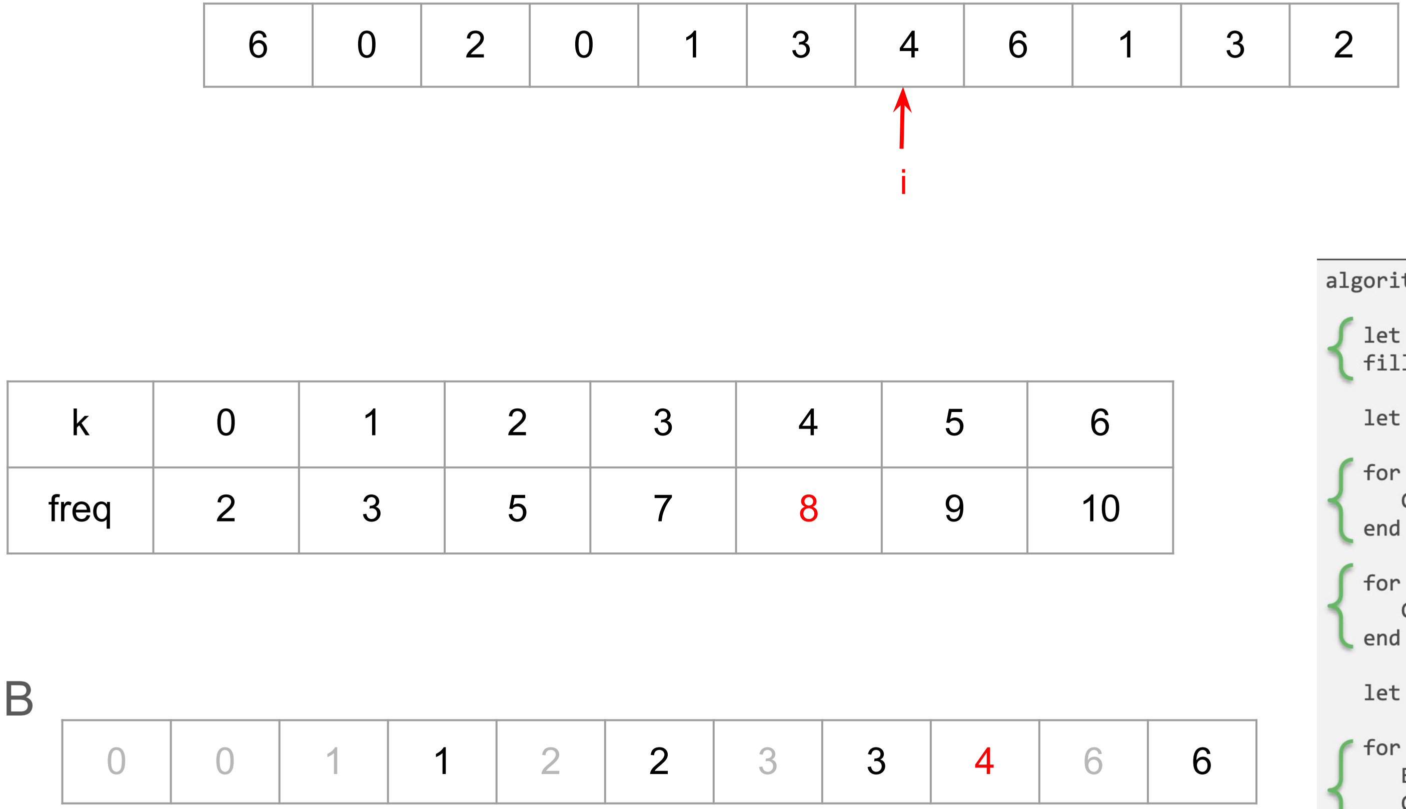
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	7	9	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



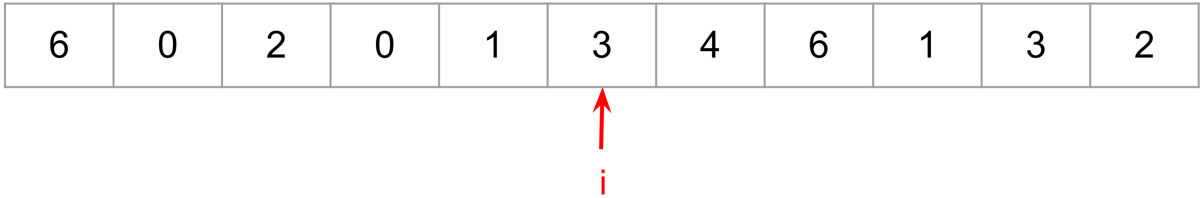
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	7	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



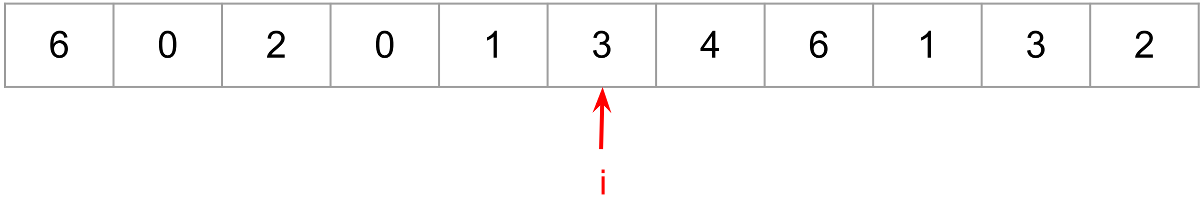
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	7	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	7	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:ℤ⁺)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

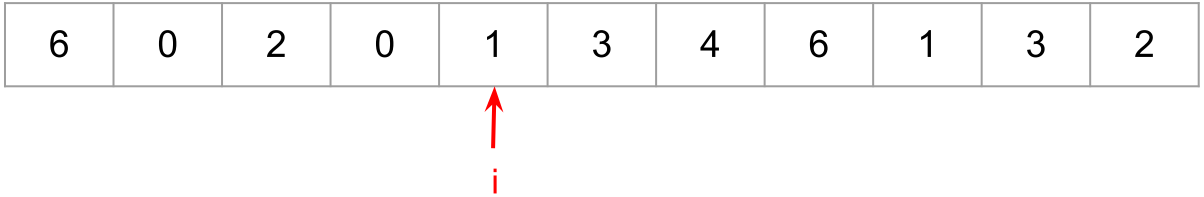
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	6	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



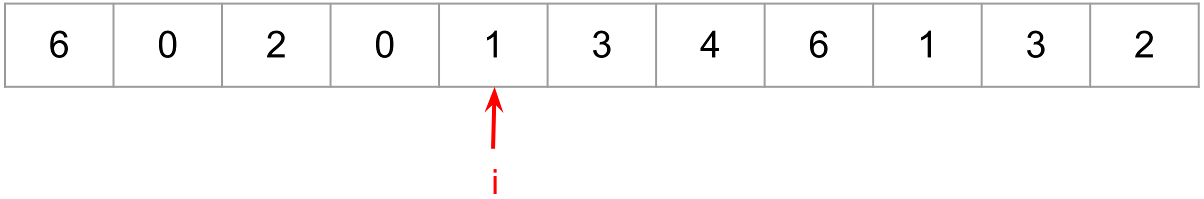
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	6	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



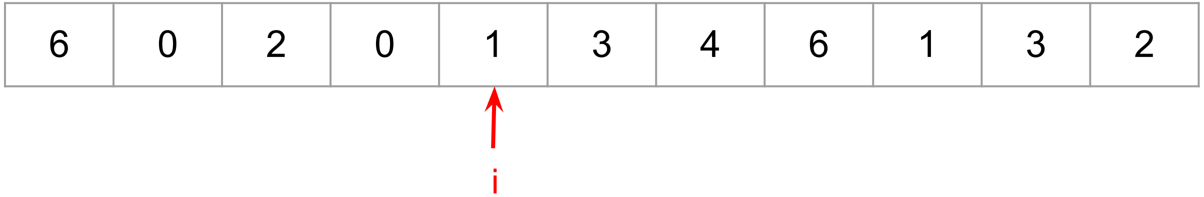
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	3	5	6	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



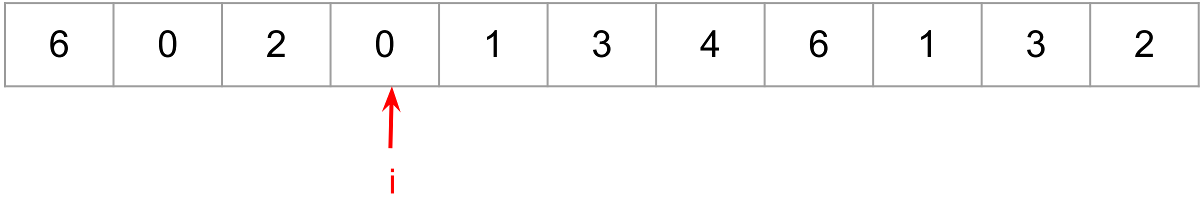
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	2	5	6	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:ℤ⁺)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



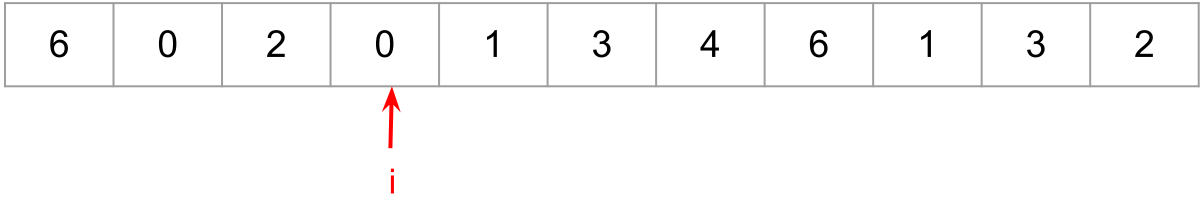
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	2	5	6	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:ℤ⁺)

   let C be an array of length k+1
   fill C with 0s

   let n be the size A

   for i from 0 to n-1 do
      C[A[i]] ← C[A[i]] + 1
   end for

   for i from 1 to k do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      B[C[A[i]] - 1] ← A[i]
      C[A[i]] ← C[A[i]] - 1
   end for

   return B

end algorithm
```



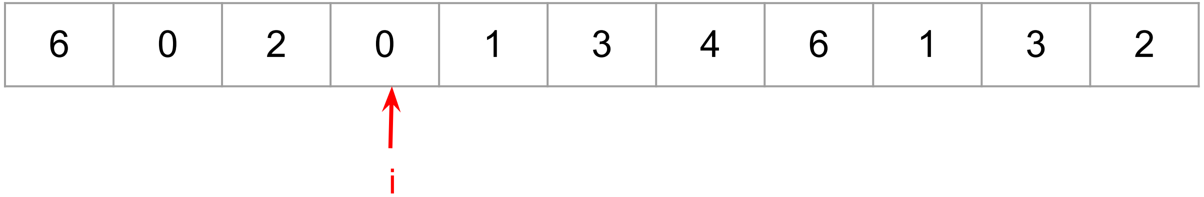
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	2	2	5	6	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



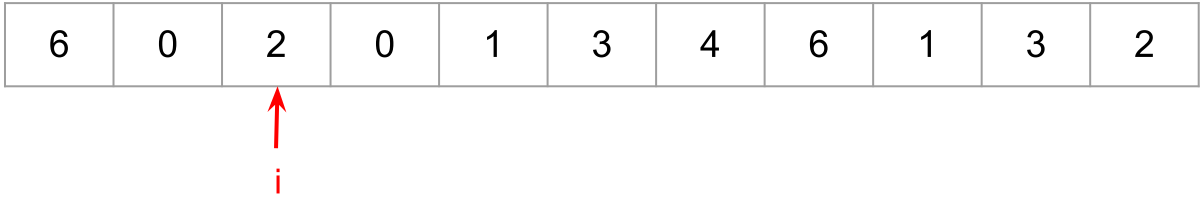
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	1	2	5	6	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



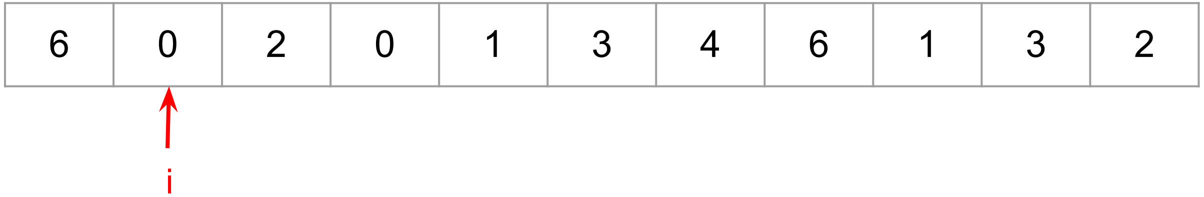
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	1	2	4	6	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```



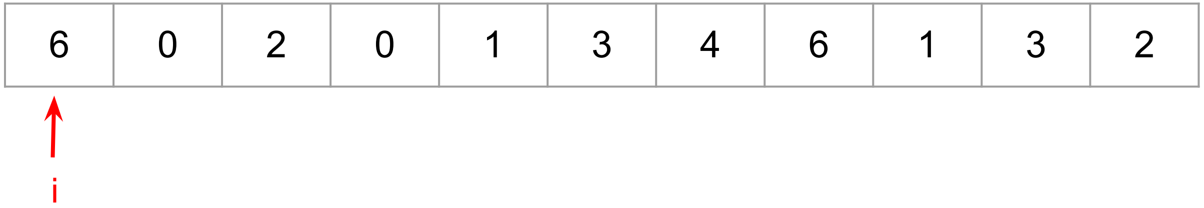
6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	0	2	4	6	8	9	10

B

0	0	1	1	2	2	3	3	4	6	6

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
       C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
       C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
       B[C[A[i]] - 1] ← A[i]
       C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```



6	0	2	0	1	3	4	6	1	3	2

k	0	1	2	3	4	5	6
freq	0	2	4	6	8	9	9

B

0	0	1	1	2	2	3	3	4	6	6

Done!

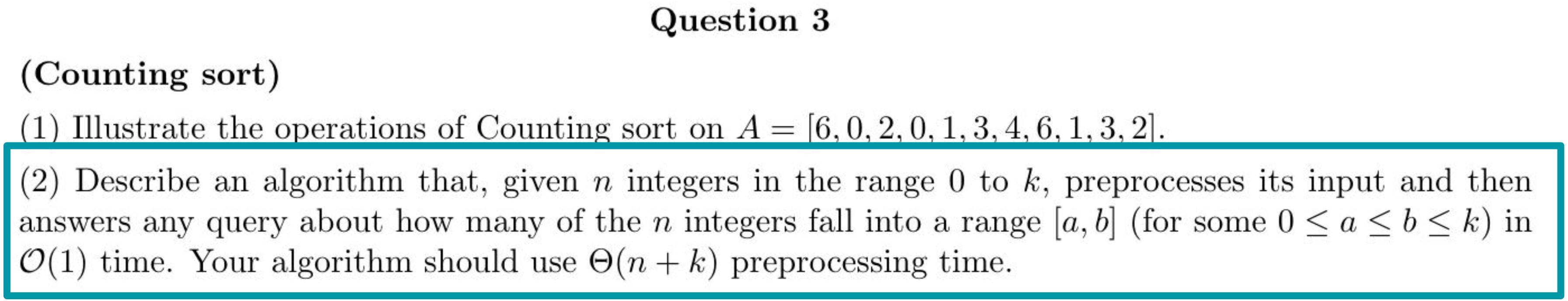
```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

    let C be an array of length k+1
    fill C with 0s

    let n be the size A

    for i from 0 to n-1 do
        C[A[i]] ← C[A[i]] + 1
    end for

    for i from 1 to k do
        C[i] ← C[i] + C[i-1]
    end for

    let B be an array of size n

    for i from n-1 to 0 by -1 do
        B[C[A[i]] - 1] ← A[i]
        C[A[i]] ← C[A[i]] - 1
    end for

    return B

end algorithm
```

Question 3

(Counting sort)

(1) Illustrate the operations of Counting sort on $A = [6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2]$.

(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.



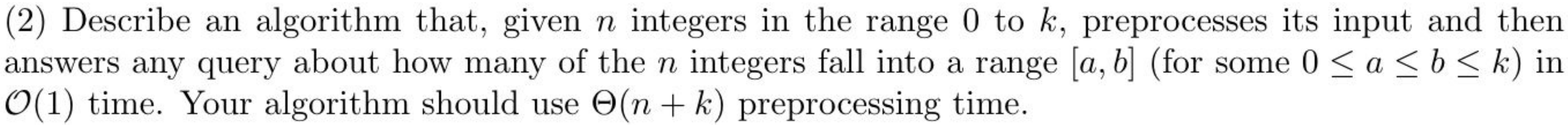
Question 3

(Counting sort)

(1) Illustrate the operations of Counting sort on $A = [6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2]$.

(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

Wait! Sounds familiar..



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

A

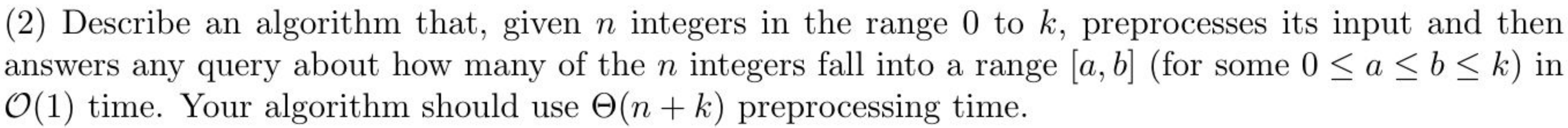
6	0	2	0	1	3	4	6	1	3	2

The counting array kept track of **C[i] = # elements less than or equal to i**

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

0	0	1	1	2	2	3	3	4	6	6



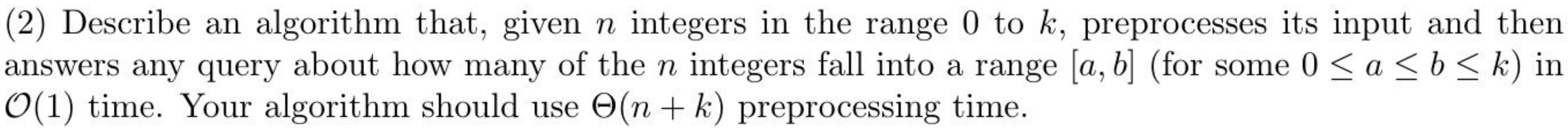
(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**

k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

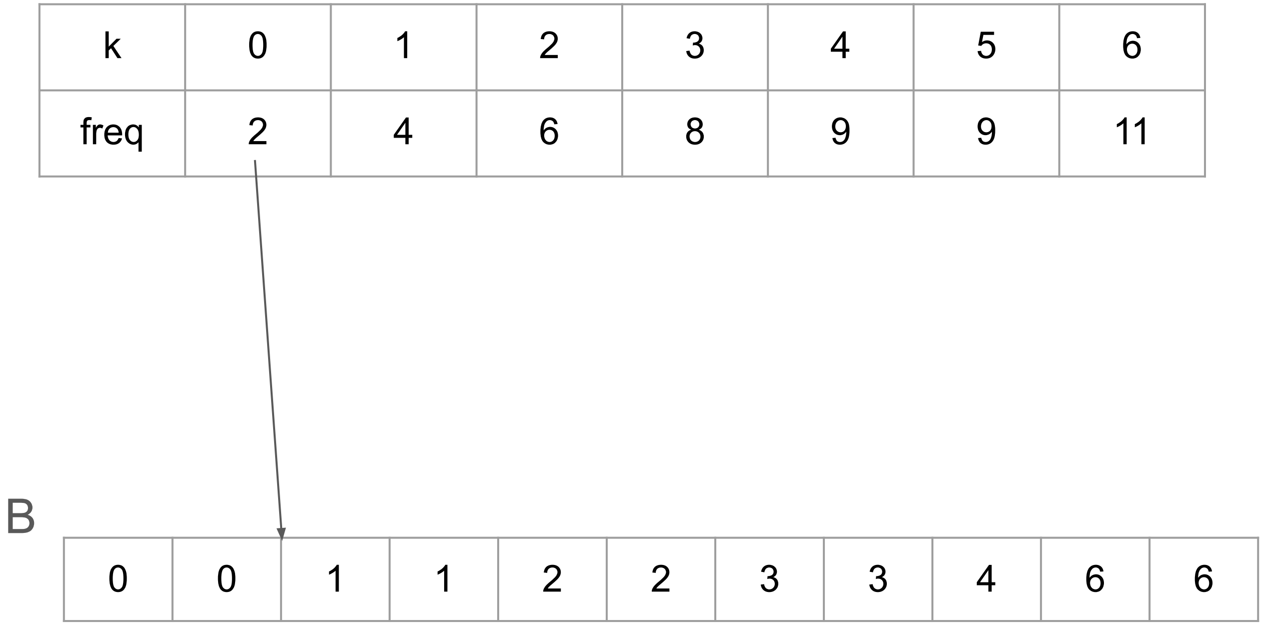
B

0	0	1	1	2	2	3	3	4	6	6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**



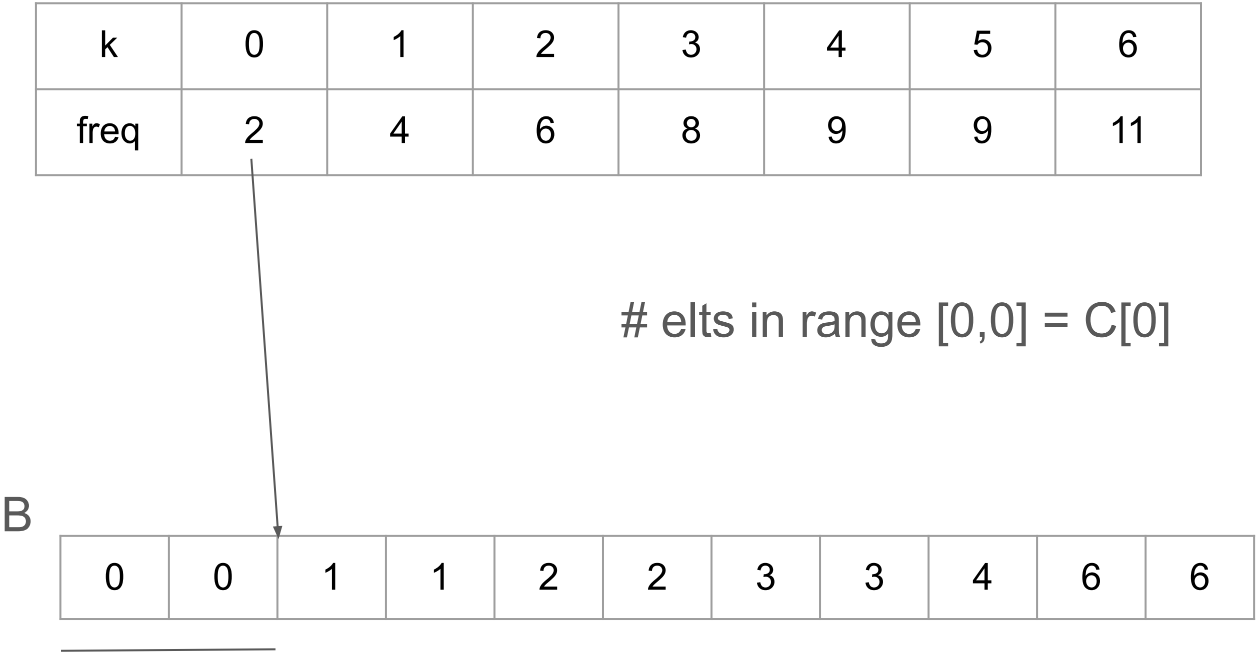
k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

0	0	1	1	2	2	3	3	4	6	6

(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**



k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

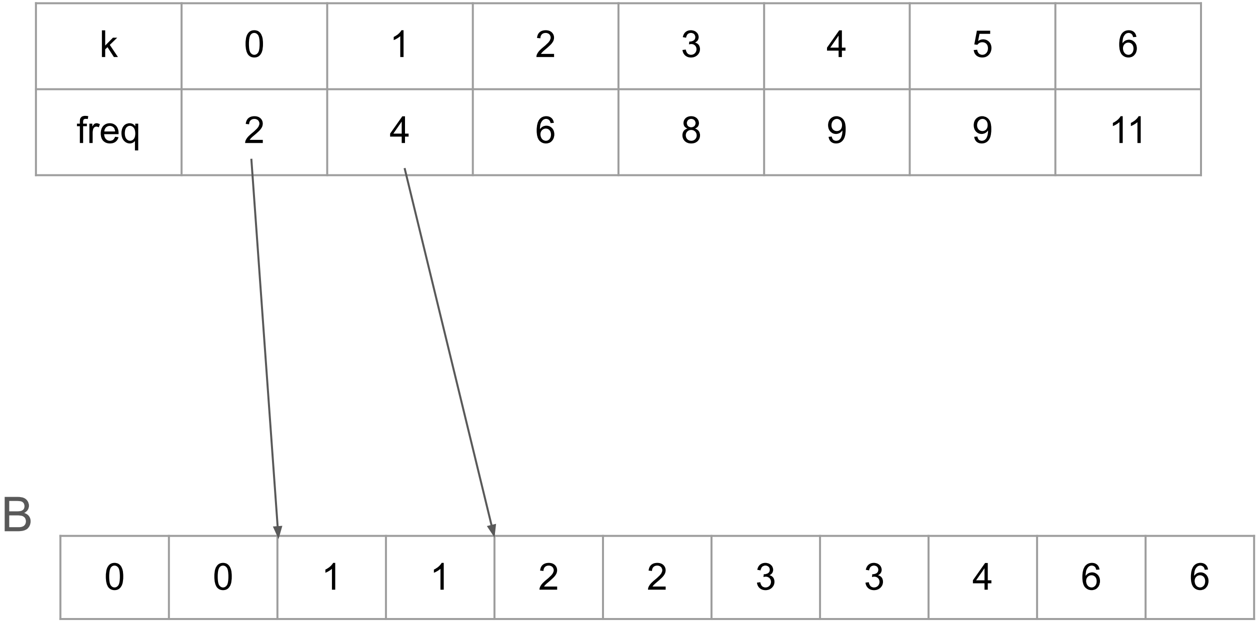
elts in range [0,0] = C[0]

B

0	0	1	1	2	2	3	3	4	6	6

(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

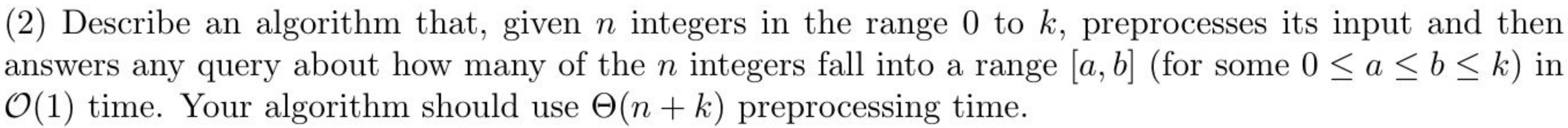
The counting array kept track of **C[i] = # elements less than or equal to i**



k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

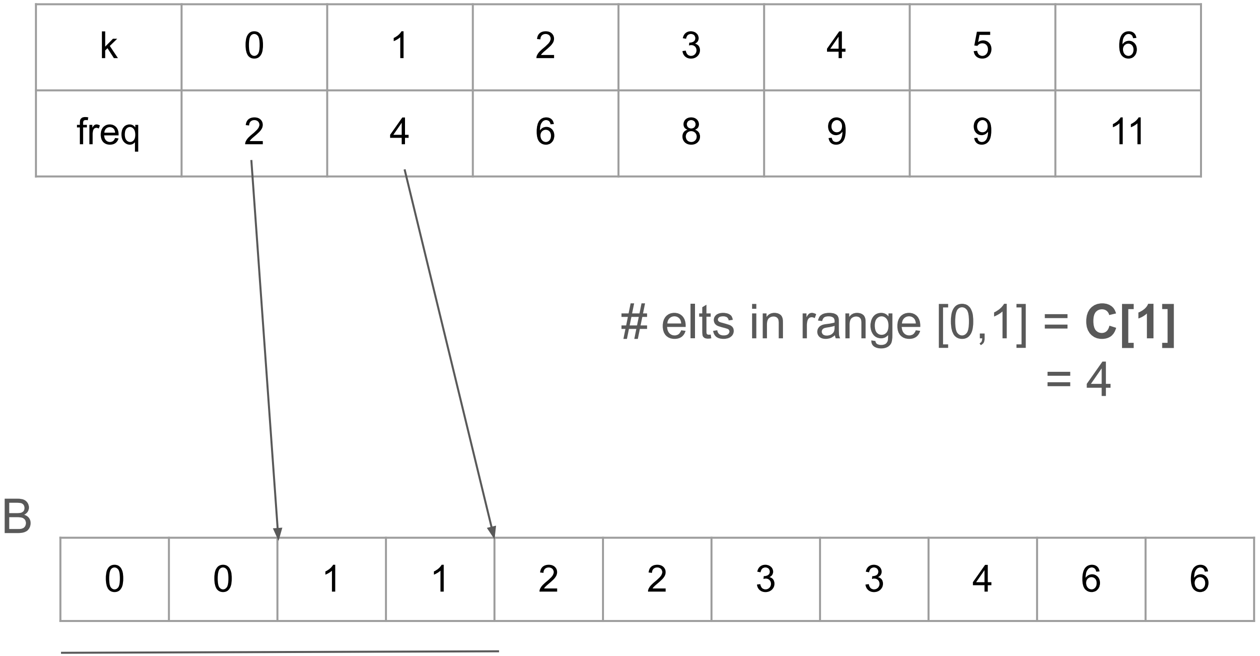
B

0	0	1	1	2	2	3	3	4	6	6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \le a \le b \le k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**

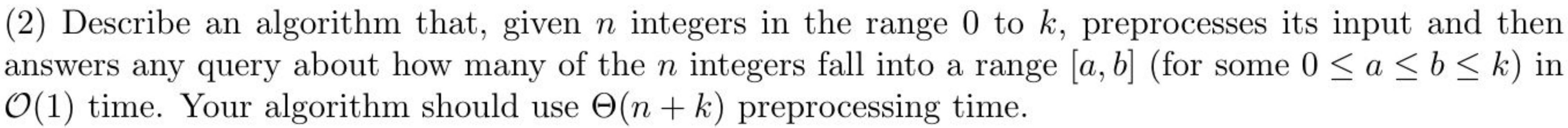


k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

\# elts in range [0,1] = **C[1]**
= 4

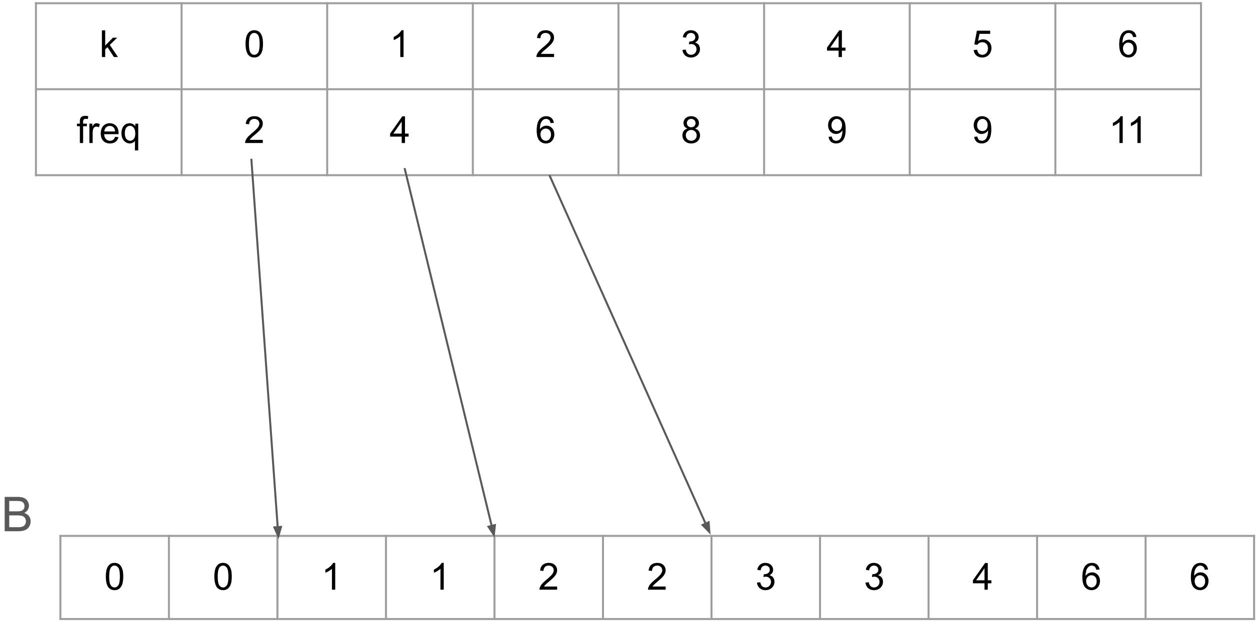
B

0	0	1	1	2	2	3	3	4	6	6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

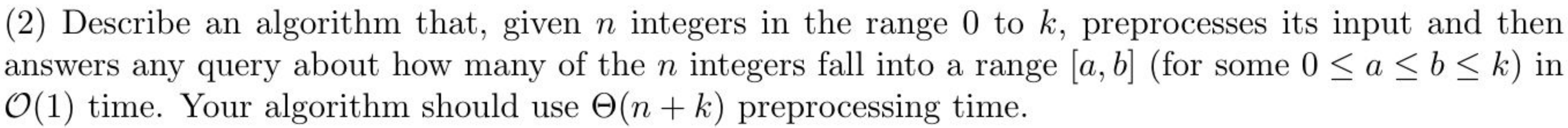
The counting array kept track of **C[i] = # elements less than or equal to i**



k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

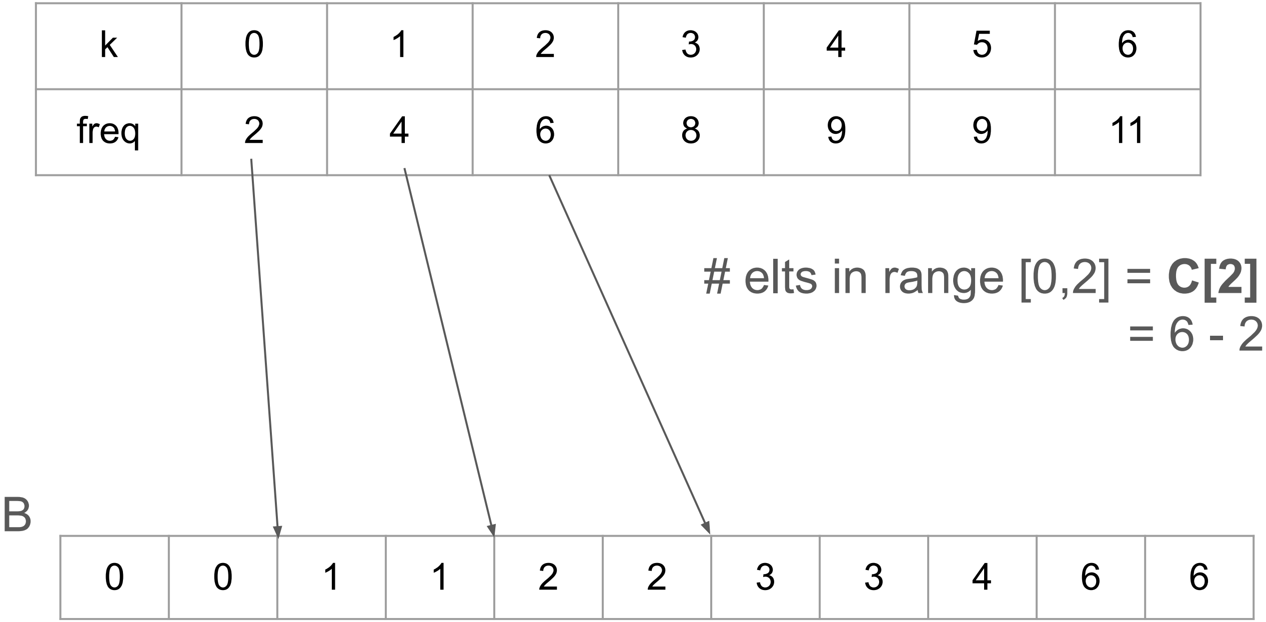
B

0	0	1	1	2	2	3	3	4	6	6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \le a \le b \le k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**

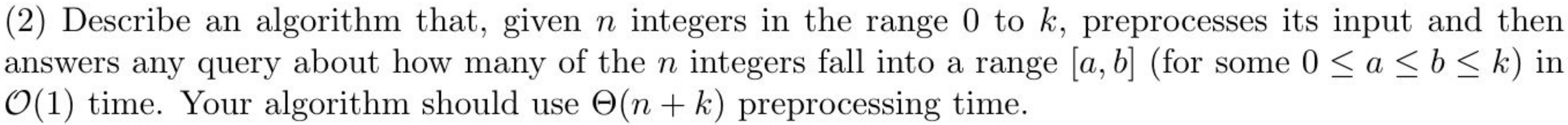


k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

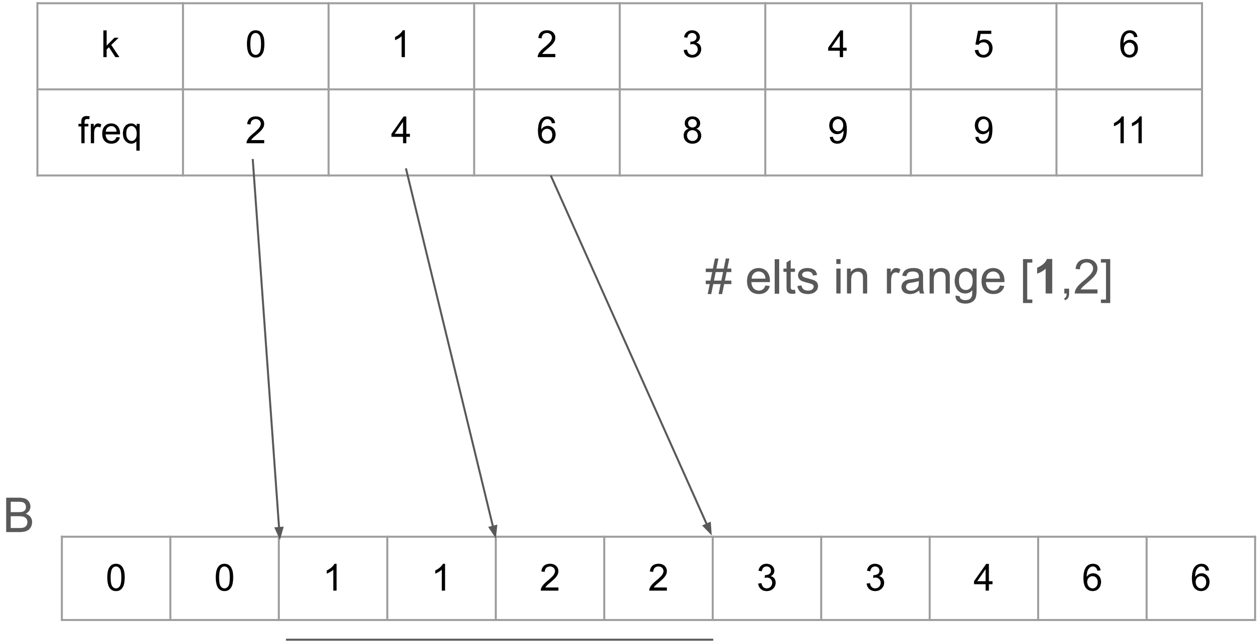
0	0	1	1	2	2	3	3	4	6	6

elts in range [0,2] = **C[2]**
= 6 - 2 + 2 = 6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**

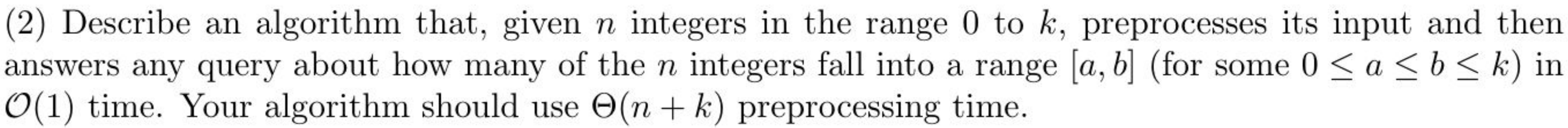


k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

elts in range [**1**,2]

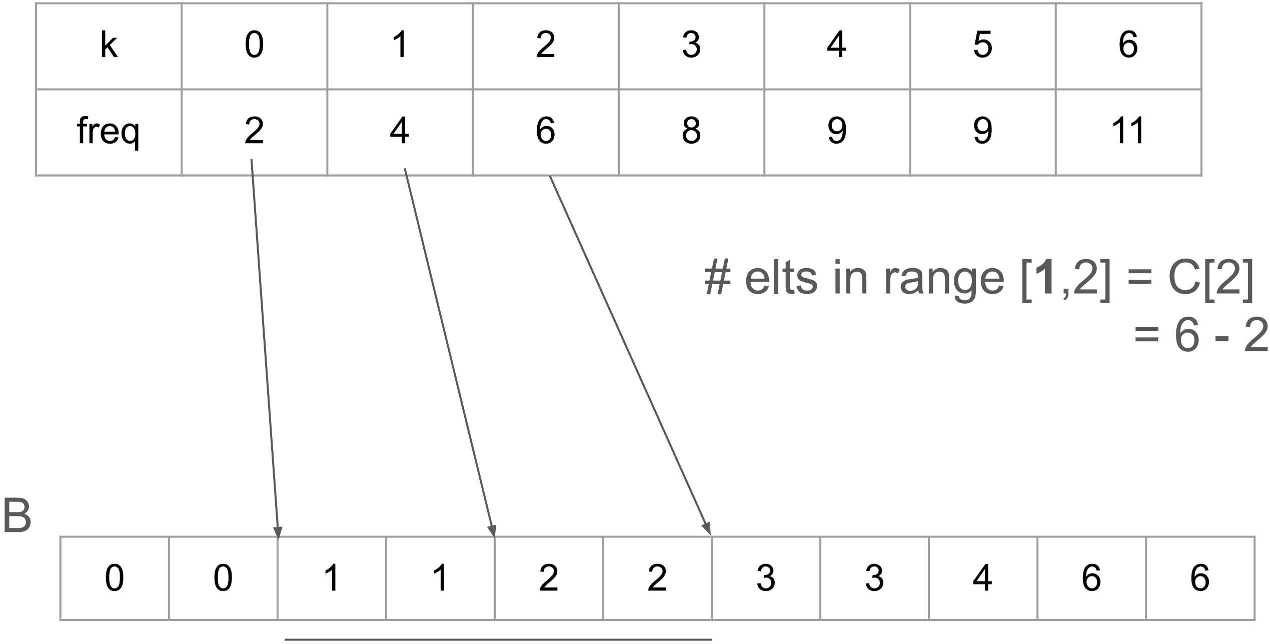
B

0	0	1	1	2	2	3	3	4	6	6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**

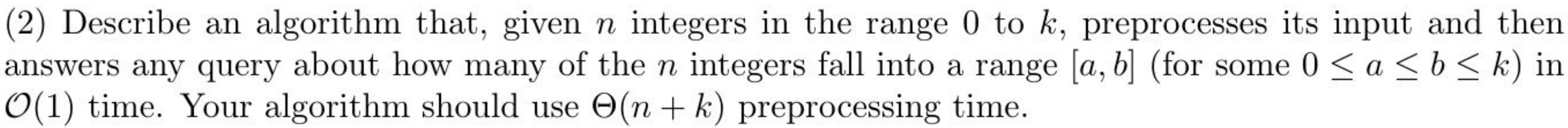


k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

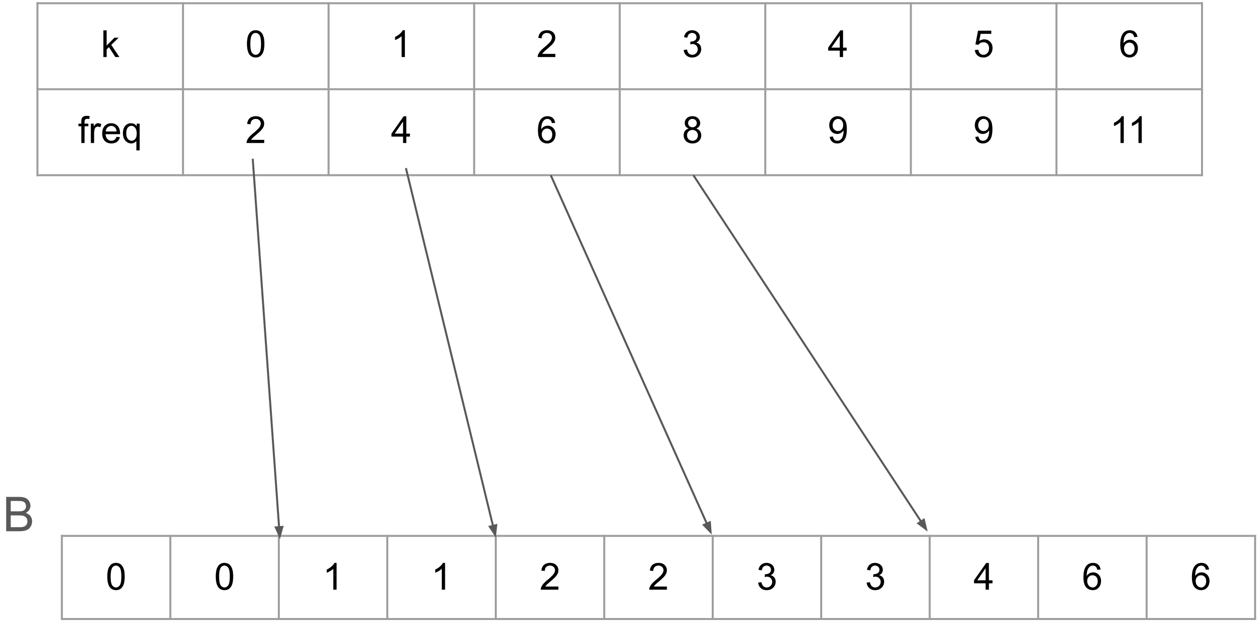
0	0	1	1	2	2	3	3	4	6	6

elts in range [**1**,2] = C[2] - C[0]
= 6 - 2 = 4



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**



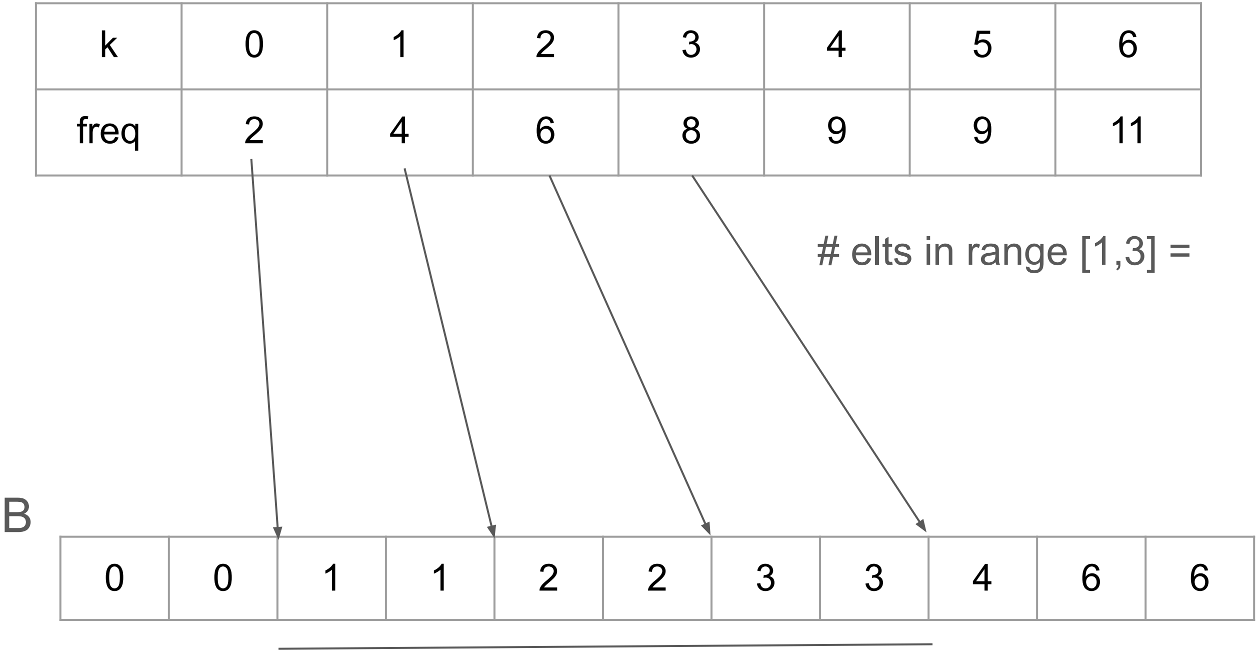
k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

0	0	1	1	2	2	3	3	4	6	6

(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**



k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

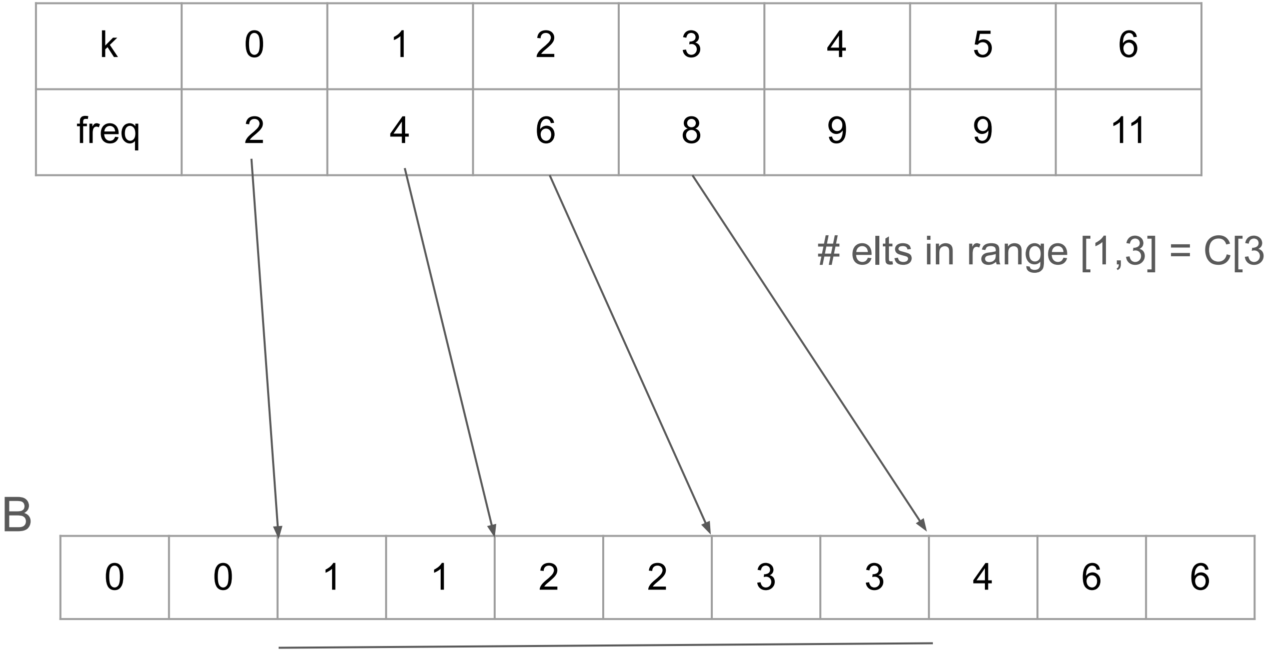
elts in range [1,3] =

B

0	0	1	1	2	2	3	3	4	6	6

(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**

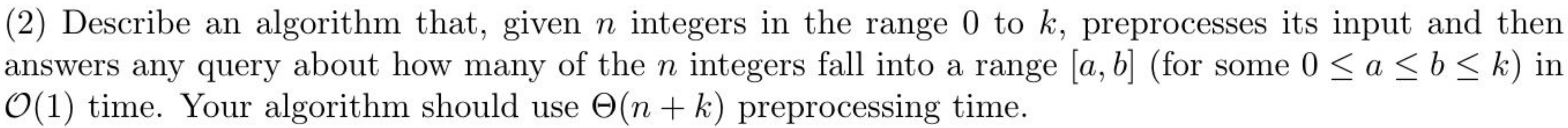


k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

elts in range [1,3] = C[3] - C[0]

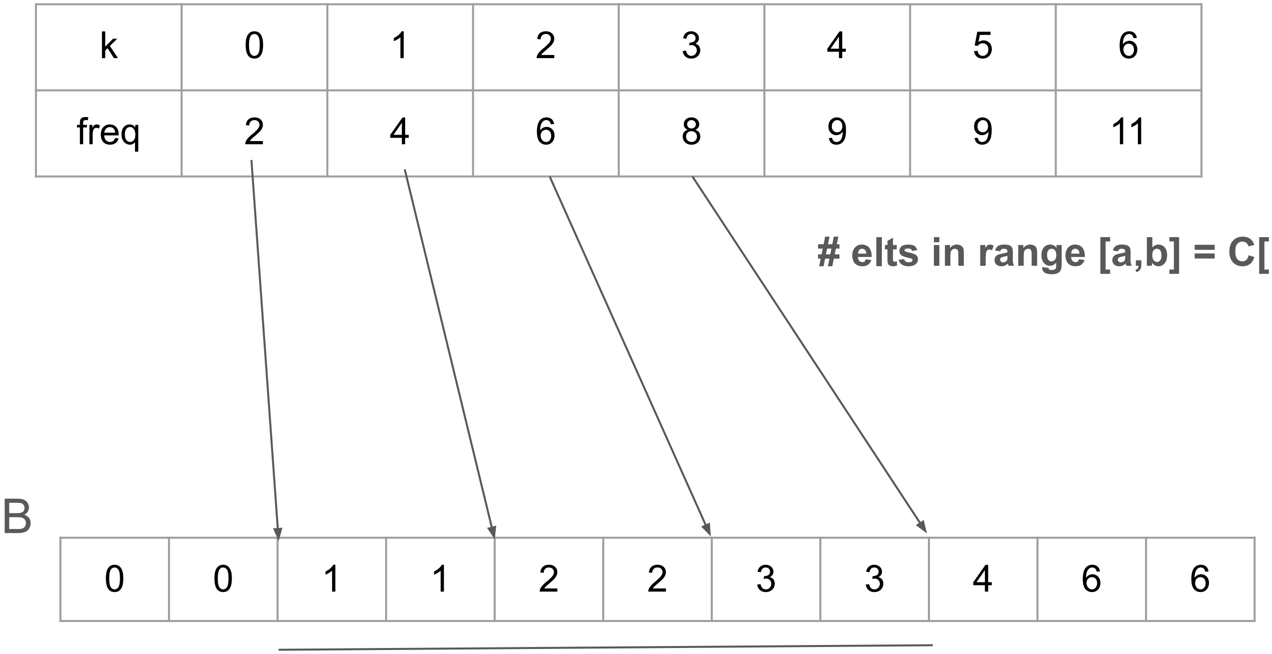
B

0	0	1	1	2	2	3	3	4	6	6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**

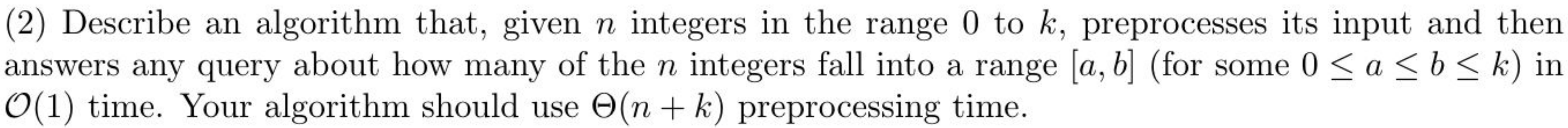


k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

 # elts in range [a,b] = C[b] - C[a - 1]

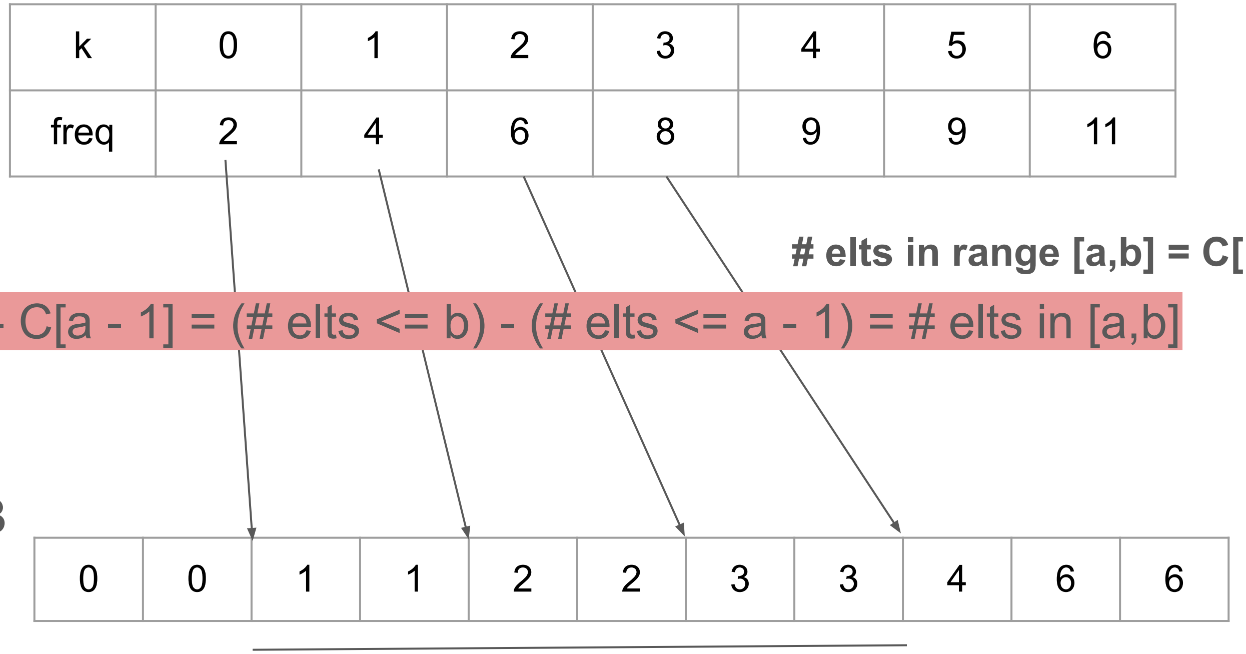
B

0	0	1	1	2	2	3	3	4	6	6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

The counting array kept track of **C[i] = # elements less than or equal to i**



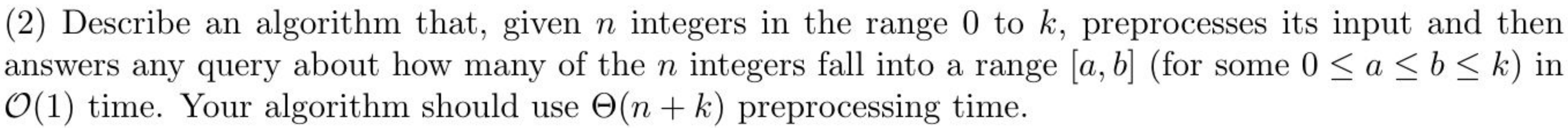
k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

 # elts in range [a,b] = C[b] - C[a - 1]

C[b] - C[a - 1] = (# elts <= b) - (# elts <= a - 1) = # elts in [a,b]

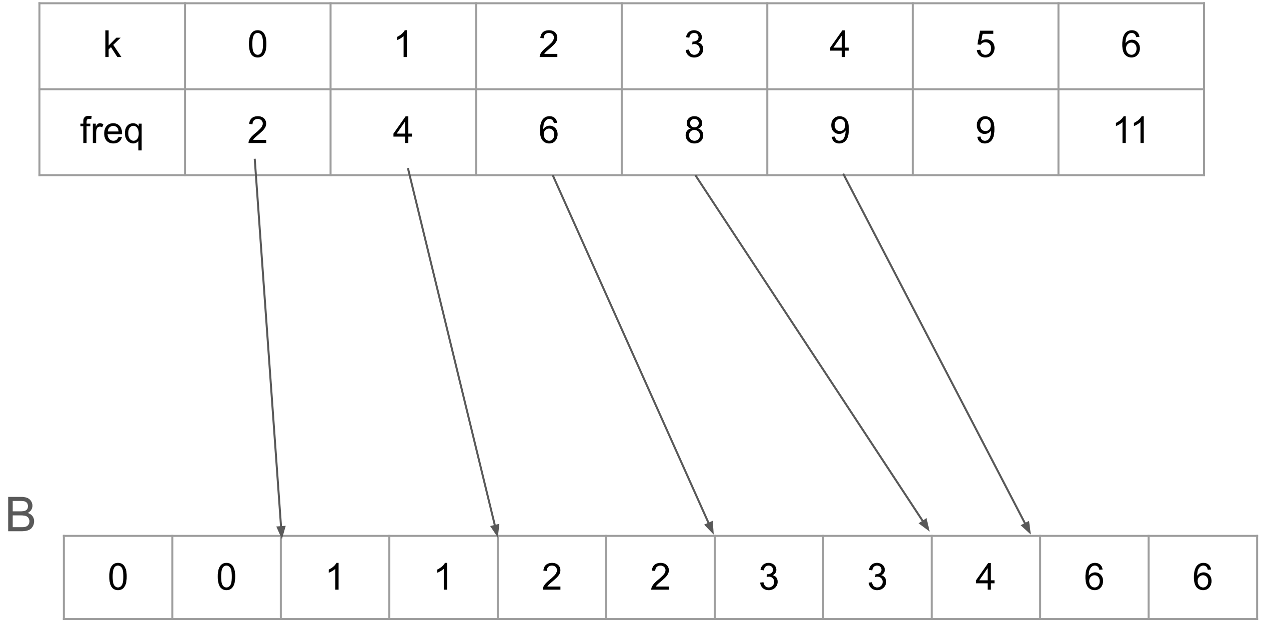
B

0	0	1	1	2	2	3	3	4	6	6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

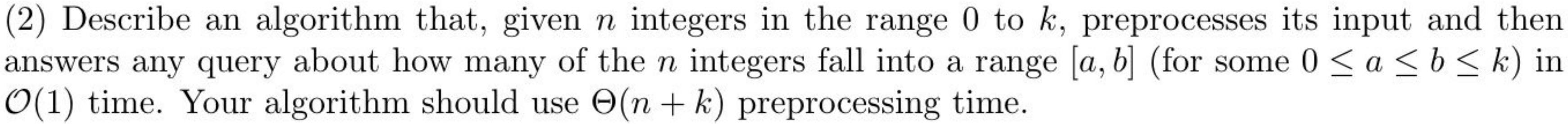
The counting array kept track of **C[i] = # elements less than or equal to i**



k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

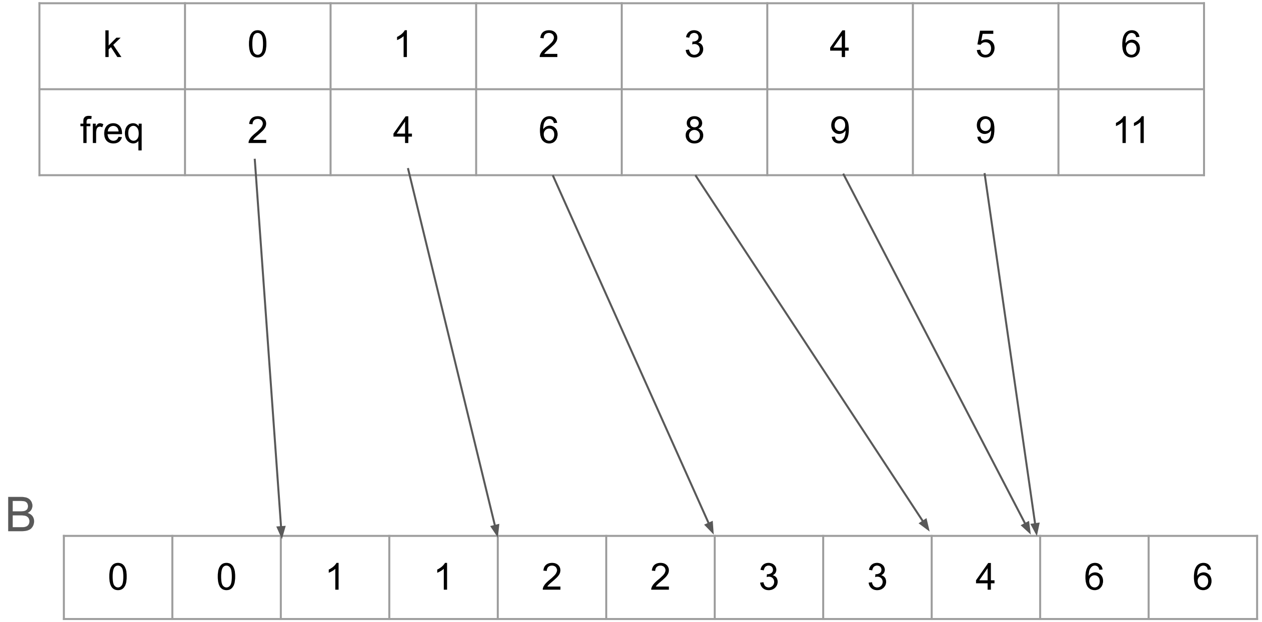
B

0	0	1	1	2	2	3	3	4	6	6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

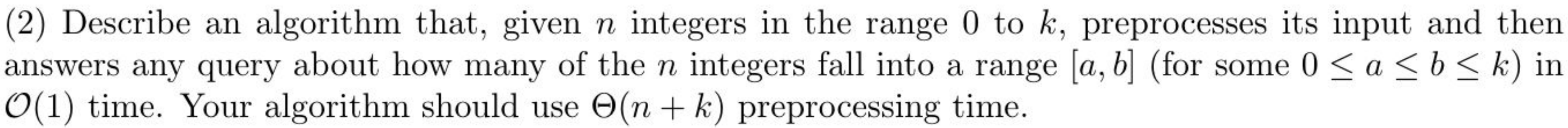
The counting array kept track of **C[i] = # elements less than or equal to i**



k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

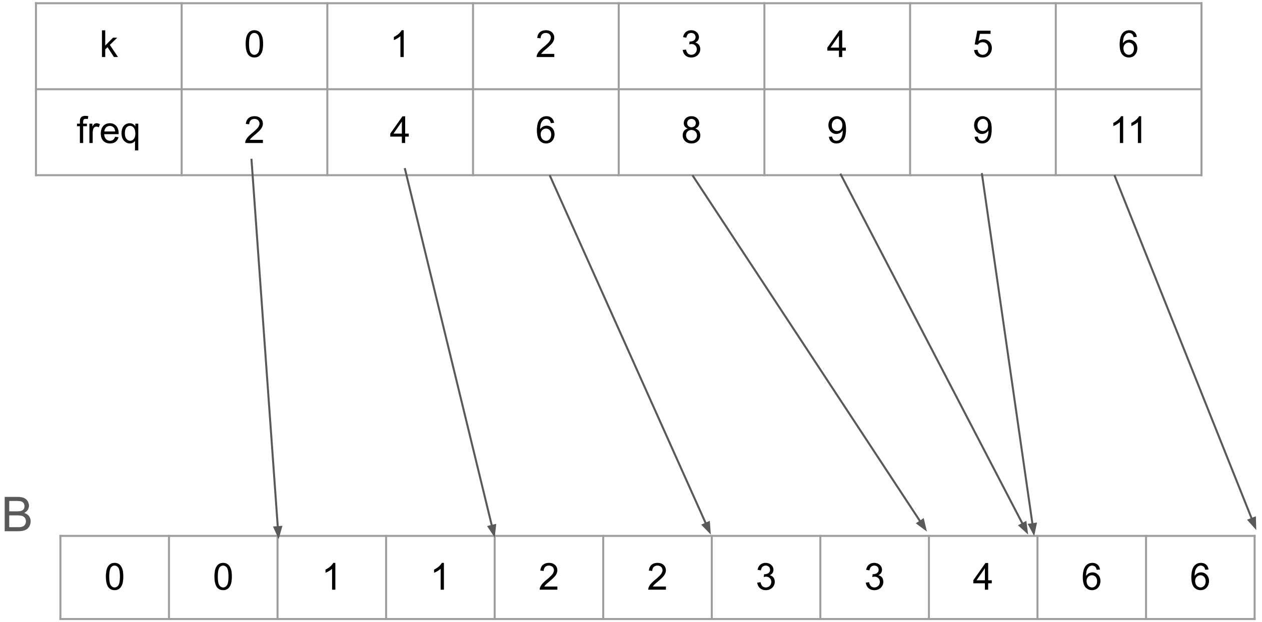
B

0	0	1	1	2	2	3	3	4	6	6



(2) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ (for some $0 \leq a \leq b \leq k$) in $\mathcal{O}(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

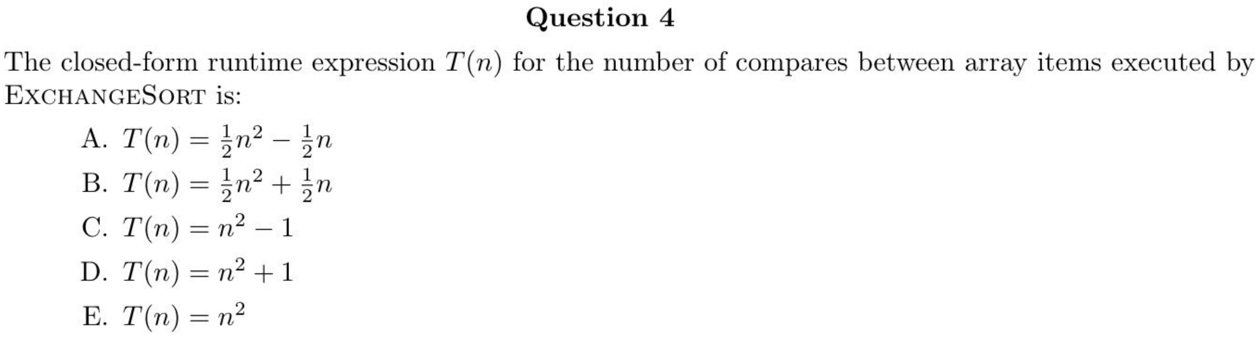
The counting array kept track of **C[i] = # elements less than or equal to i**



k	0	1	2	3	4	5	6
freq	2	4	6	8	9	9	11

B

0	0	1	1	2	2	3	3	4	6	6



Question 4

The closed-form runtime expression $T(n)$ for the number of compares between array items executed by ExchangeSort is:

A. $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$

B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$

C. $T(n) = n^2 - 1$

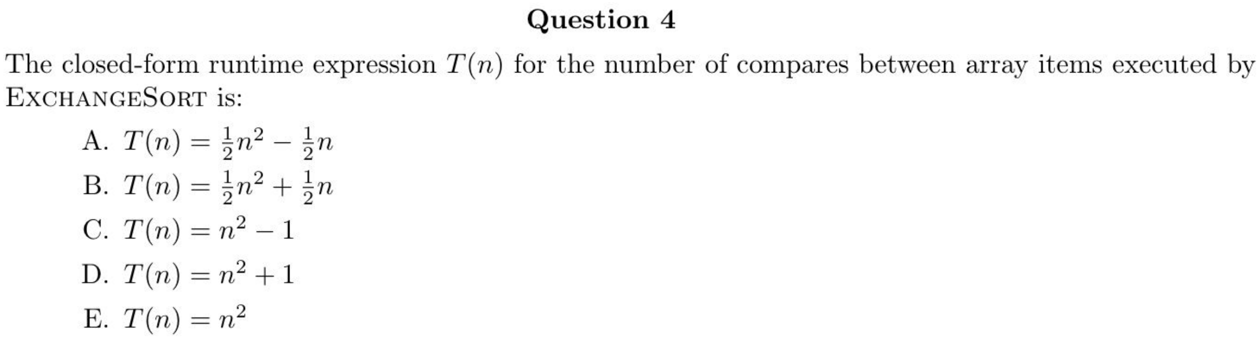
D. $T(n) = n^2 + 1$

E. $T(n) = n^2$

Exchange Sort: for i = 0,..., n - 2:

Compare i’th elt with j = i +1, …, n - 1 elt

Swap elements if i’th > j’th



Question 4

The closed-form runtime expression $T(n)$ for the number of compares between array items executed by ExchangeSort is:

A. $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$

B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$

C. $T(n) = n^2 - 1$

D. $T(n) = n^2 + 1$

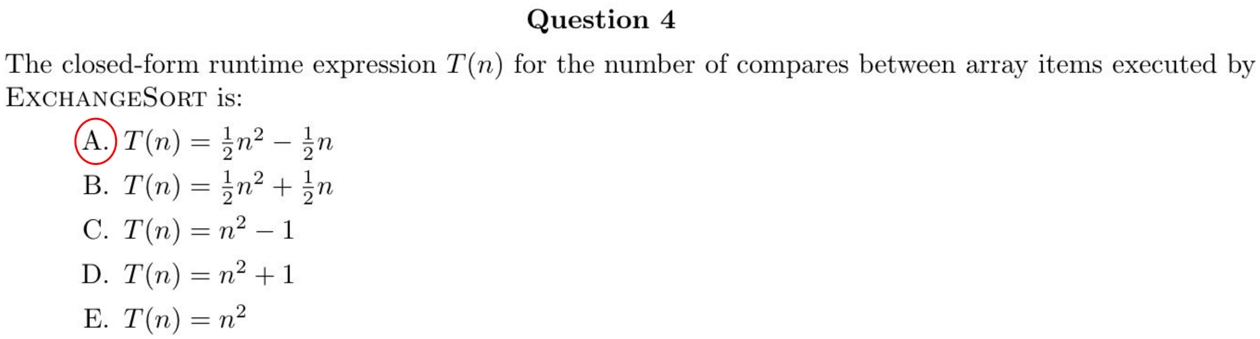
E. $T(n) = n^2$

Exchange Sort: for i = 0,..., n - 2:

Compare i’th elt with j = i +1, …, n - 1 elt

Swap elements if i’th > j’th

# of compares = $\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$



Question 4

The closed-form runtime expression $T(n)$ for the number of compares between array items executed by EXCHANGESORT is:

[✓ A.] $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$

B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$

C. $T(n) = n^2 - 1$

D. $T(n) = n^2 + 1$

E. $T(n) = n^2$

Exchange Sort: for i = 0,..., n - 2:

Compare i’th elt with j = i +1, …, n - 1 elt

Swap elements if i’th > j’th

Exercise: do this math at home

# of compares = $\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

Question 5

The closed-form runtime expression $T(n)$ for the maximum number of Swap calls made by Exchange-Sort is:

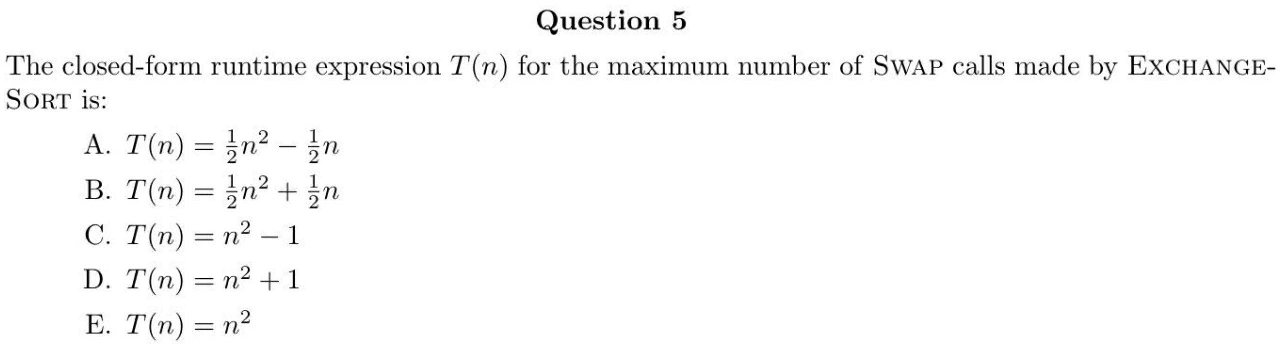
A. $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$

B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$

C. $T(n) = n^2 - 1$

D. $T(n) = n^2 + 1$

E. $T(n) = n^2$



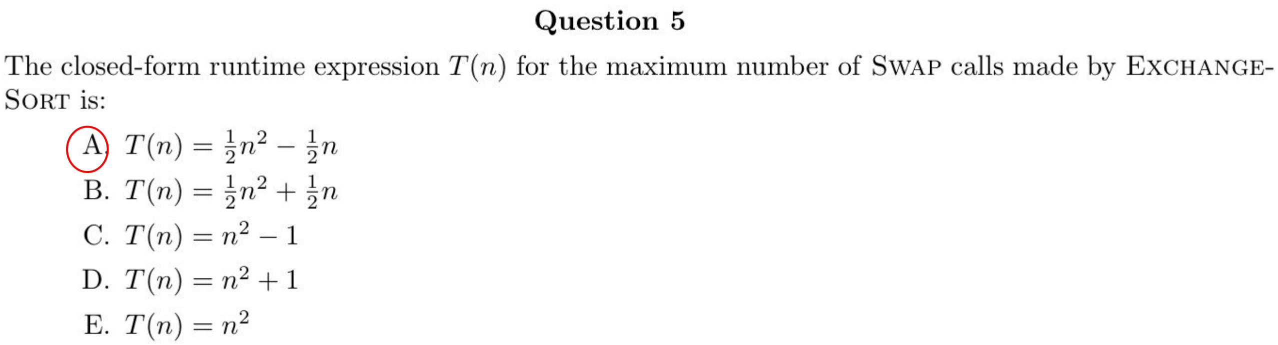
Question 5

The closed-form runtime expression $T(n)$ for the maximum number of SWAP calls made by EXCHANGE-SORT is:

- A. $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$
- B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$
- C. $T(n) = n^2 - 1$
- D. $T(n) = n^2 + 1$
- E. $T(n) = n^2$

If we swap for every comparison, by the prev question, answer would be A

Are there arrays where we do this?



Question 5

The closed-form runtime expression $T(n)$ for the maximum number of SWAP calls made by EXCHANGE-SORT is:

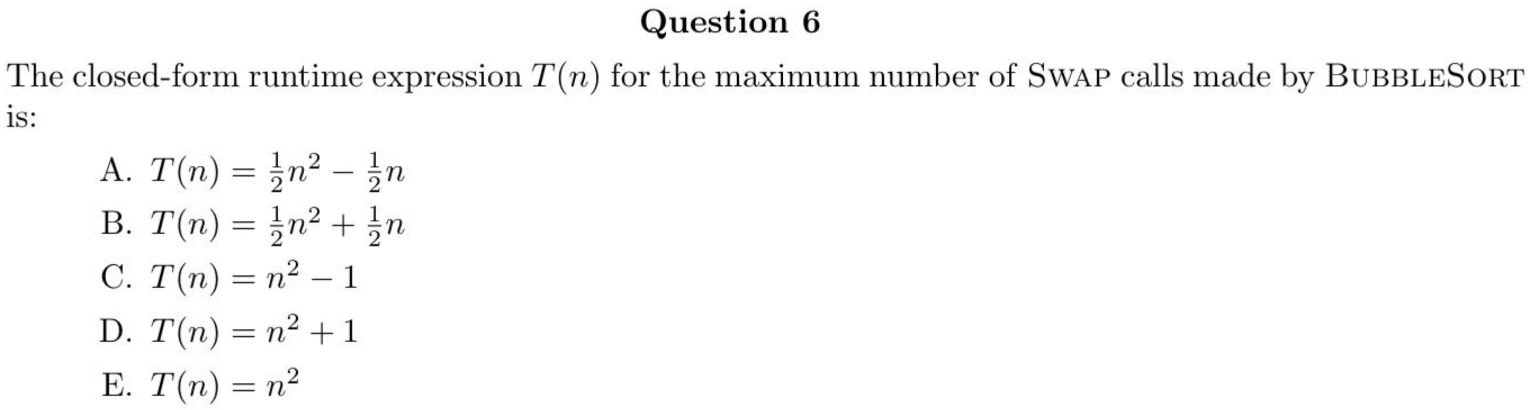
- [✓ A.] $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$
- B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$
- C. $T(n) = n^2 - 1$
- D. $T(n) = n^2 + 1$
- E. $T(n) = n^2$

If we swap for every comparison, by the prev question, answer would be A

Are there arrays where we do this?

5	4	3	2	1

Exercise: confirm this works at home

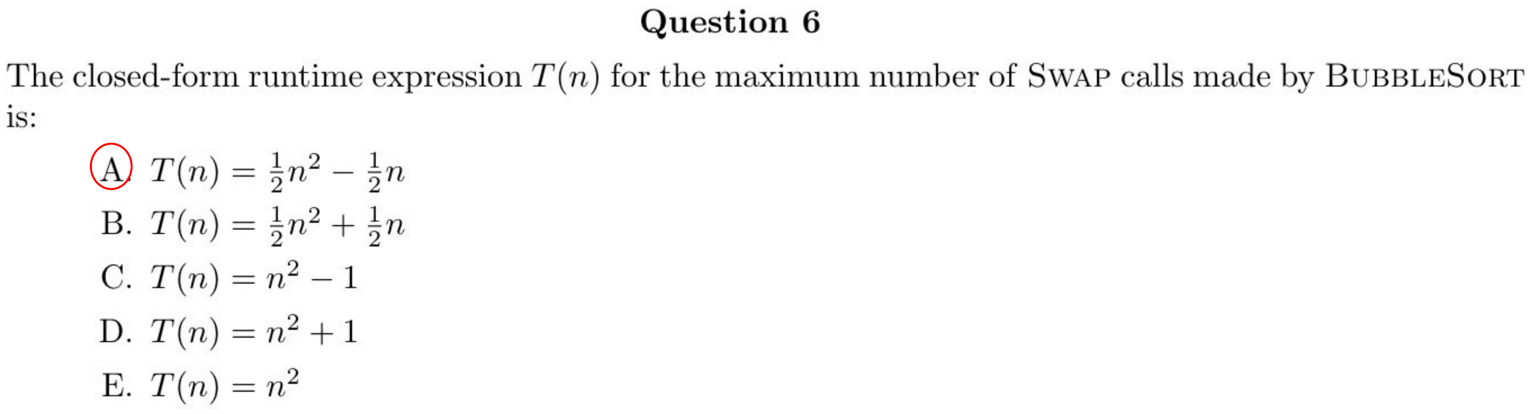


Question 6

The closed-form runtime expression $T(n)$ for the maximum number of SWAP calls made by BUBBLESORT is:

A. $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$
B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$
C. $T(n) = n^2 - 1$
D. $T(n) = n^2 + 1$
E. $T(n) = n^2$

“Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.”



Question 6

The closed-form runtime expression $T(n)$ for the maximum number of SWAP calls made by BUBBLESORT is:

[✓ A.] $T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$
B. $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$
C. $T(n) = n^2 - 1$
D. $T(n) = n^2 + 1$
E. $T(n) = n^2$

“Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.”

5	4	3	2	1

Exercise: confirm this works at home