# Analysis of Algorithms

# Contains Duplicate

Leetcode 217 (easy, Blind75): Contains duplicate

Given an array of $n$ numbers, determine if any value appears at least twice.

Last time we saw an algorithm for this problem using two for loops.

We argued that this was not a very efficient solution.

Let's see now how we can claim this more precisely.

# Double For Loop

```
bool containsDuplicate(const std::vector<int>& arr) {
    for(int i = 0; i < arr.size(); ++i) {
        for(int j = 0; j < i; ++j) {
            if(arr[i] == arr[j]) {
                return true;
            }
        }
    }
    return false;
}
```

https://godbolt.org/z/fvo5GjE9e

Say that the size of `arr` is $n$. How many operations does this algorithm take?

```cpp
bool containsDuplicate(const std::vector<int>& arr) {
    for(int i = 0; i < arr.size(); ++i) {
        for(int j = 0; j < i; ++j) {
            if(arr[i] == arr[j]) {
                return true;
            }
        }
    }
    return false;
}
```

Let's assume there is no duplicate, so we have to finish the outer for loop.

| value of $i$ | # comparisons in inner loop |
|---|---|
| $0$ | $0$ |
| $1$ | $1$ |
| $2$ | $2$ |

```cpp
bool containsDuplicate(const std::vector<int>& arr) {
    for(int i = 0; i < arr.size(); ++i) {
        for(int j = 0; j < i; ++j) {
            if(arr[i] == arr[j]) {
                return true;
            }
        }
    }
    return false;
}
```

| value of $i$ | # comparisons in inner loop |
|---|---|
| $0$ | $0$ |
| $1$ | $1$ |
| $2$ | $2$ |
| $\vdots$ | $\vdots$ |
| $n-1$ | $n-1$ |

Total number of comparisons is

$$1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

$$=$$

$$\begin{array}{c} 1 \\ + \\ n-1 \end{array}$$

Total number of comparisons is

$$1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

$$=$$

$$\begin{matrix} 1 \\ + \\ n-1 \end{matrix} \quad + \quad \begin{matrix} 2 \\ + \\ n-2 \end{matrix} \quad +$$

Total number of comparisons is

$$1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

$$=$$

$$\begin{matrix} 1 \\ + \\ n-1 \end{matrix} \quad + \quad \begin{matrix} 2 \\ + \\ n-2 \end{matrix} \quad + \quad \begin{matrix} 3 \\ + \\ n-3 \end{matrix} \quad +$$

Total number of comparisons is

$$1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

$$=$$

If $n-1$ is even

$$\begin{matrix} 1 & & 2 & & 3 & & & & & (n-1)/2 \\ + & + & + & + & + & + & \cdots & + & & + \\ n-1 & & n-2 & & n-3 & & & & & (n+1)/2 \end{matrix}$$

Then we have $(n-1)/2$ pairs that sum to $n$. The sum is

$$\frac{(n-1)n}{2}$$

Total number of comparisons is

$$1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

$$=$$

If $n-1$ is odd

$$\begin{matrix} 1 \\ + \\ n-1 \end{matrix} + \begin{matrix} 2 \\ + \\ n-2 \end{matrix} + \begin{matrix} 3 \\ + \\ n-3 \end{matrix} + \cdots + \begin{matrix} (n-2)/2 \\ + \\ (n+2)/2 \end{matrix} + n/2$$

The sum is

$$\frac{(n-2)n}{2} + \frac{n}{2} = \frac{(n-1)n}{2}$$

# Conclusion

$$1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1) = \frac{(n-1)n}{2}$$

Our double for loop makes roughly $\dfrac{n^2}{2}$ comparisons.

Now we have an analytical benchmark to compare against when looking at other algorithms for Contains Duplicate.

# Big Oh Notation

# Simplifying Running Times

We saw that our double for loop algorithm for Contains Duplicate on an array of size $n$ made $(n-1)n/2$ comparisons when there was no duplicate.

But can we really use this level of precision?

- How much time does a comparison take?
- To truly predict running time we would need details of the processor, memory layout, caching strategy, etc.
- And this analysis would have to be done for each computing platform.

# Big Oh Motivation

Can we run the double for loop algorithm for contains duplicate on an array with a million elements?

Big Oh notation gives a "back of the envelope" way to answer these questions.

It talks about how complexity of an algorithm grows as a function of the input size.

We can use it to broadly classify algorithms by running time or memory usage.

# Simplifying Running Times

The level of slack we typically use in analysis of algorithms is "up to constant factors".

Example: $n^2/2$ and $n^2$ are the same up to a multiplicative constant factor.

This approach is more robust to particular details of the implementation and hardware being used.

This lets us classify algorithms in broad categories, e.g. about $n$ steps versus about $n^2$ steps.

# Simplifying Our Jobs

Ignoring constant factors makes analyzing the running times of algorithms easier.

```
bool containsDuplicate(const std::vector<int>& arr) {
    for(int i = 0; i < arr.size(); ++i) {
        for(int j = 0; j < i; ++j) {
            if(arr[i] == arr[j]) {
                return true;
            }
        }
    }
    return false;
}
```

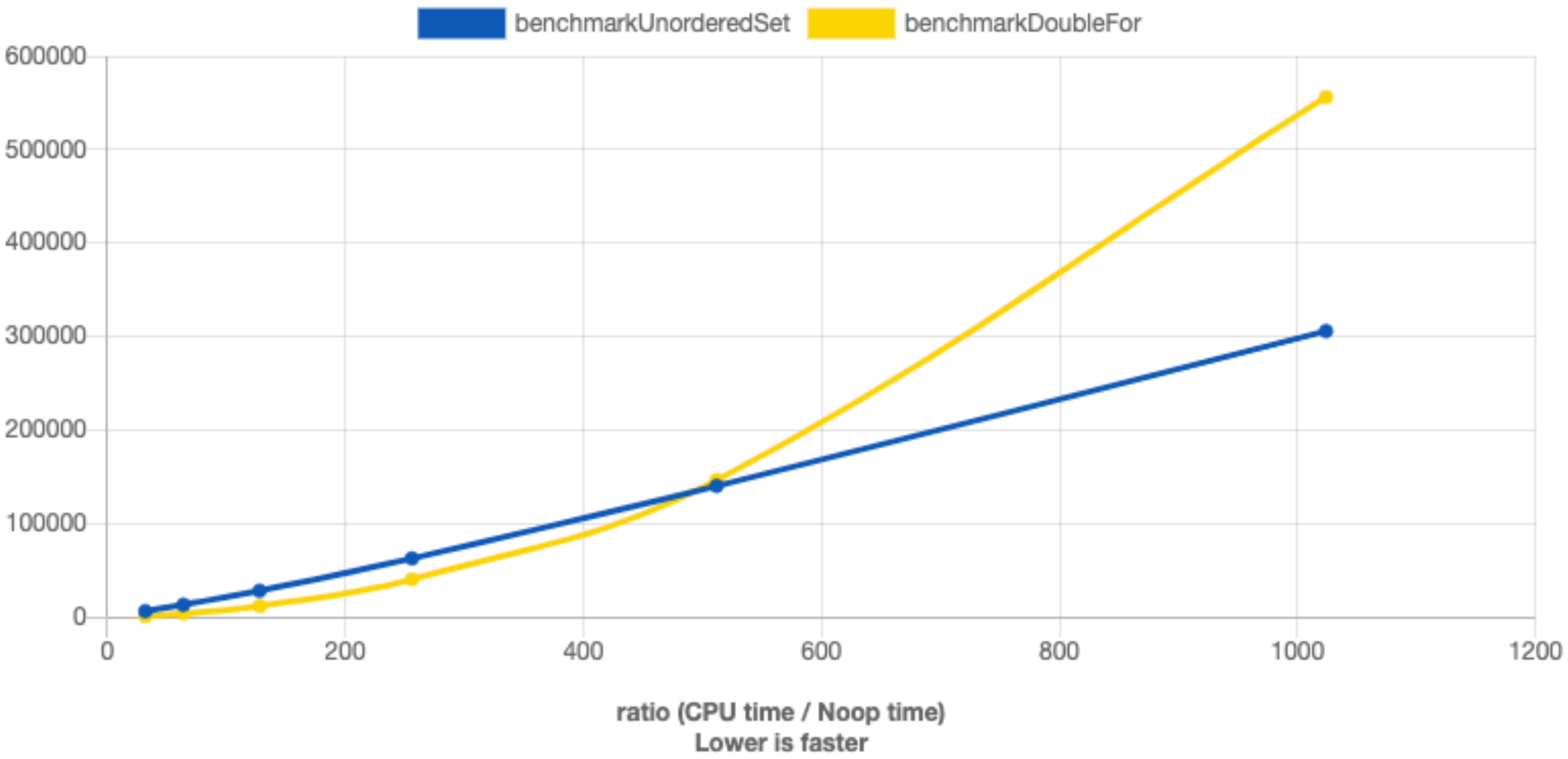
It is easier to see this algorithm makes at most $n^2$ comparisons.

# Factor of 2

Of course if one algorithm is twice as fast as another it can make a huge difference in practice.

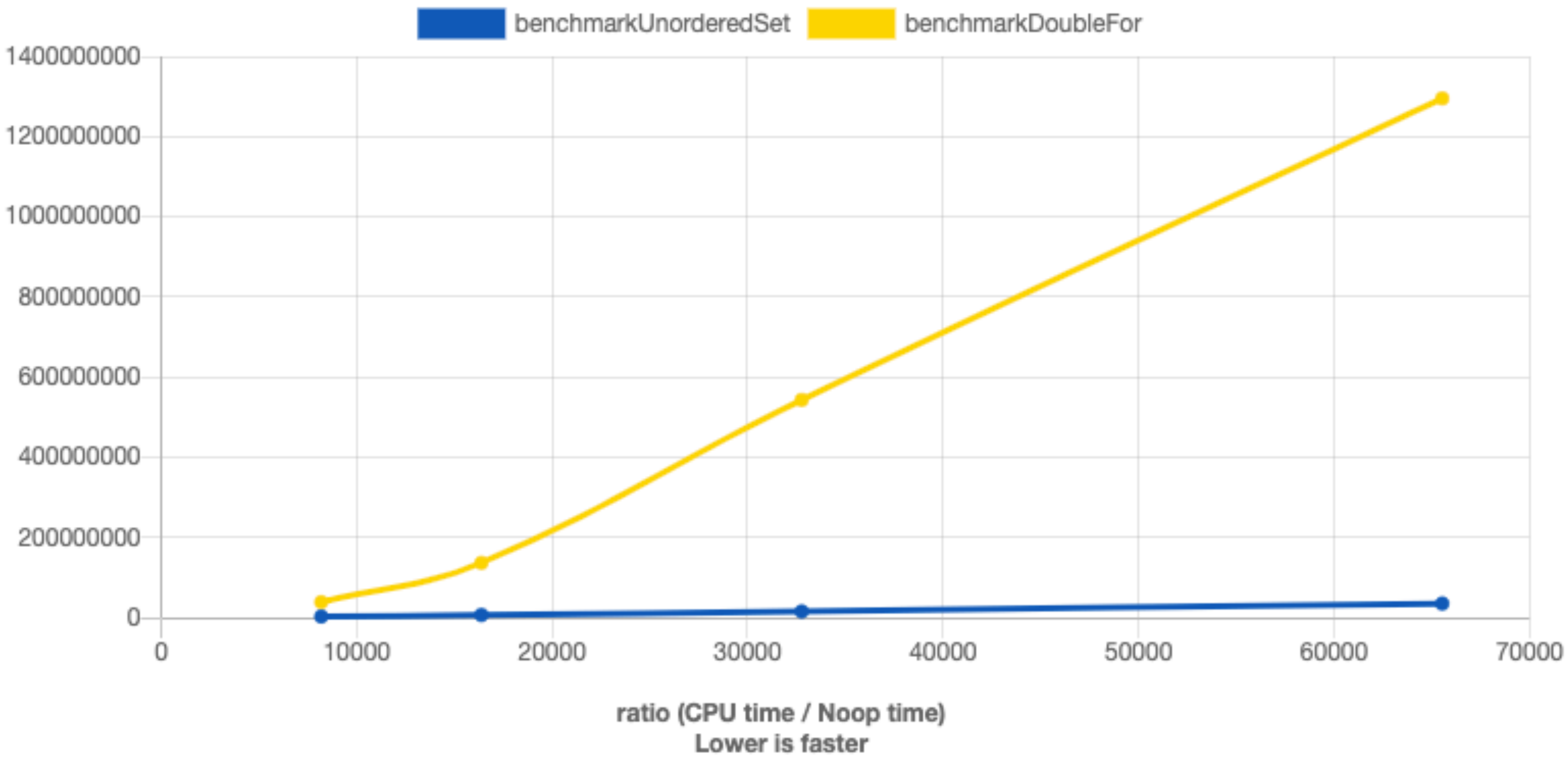
But for this level of optimization one is better off benchmarking rather than computing detailed constants of the number of steps in pseudocode.

# Small Size Effects




Which algorithm is better?

# Small Size Effects



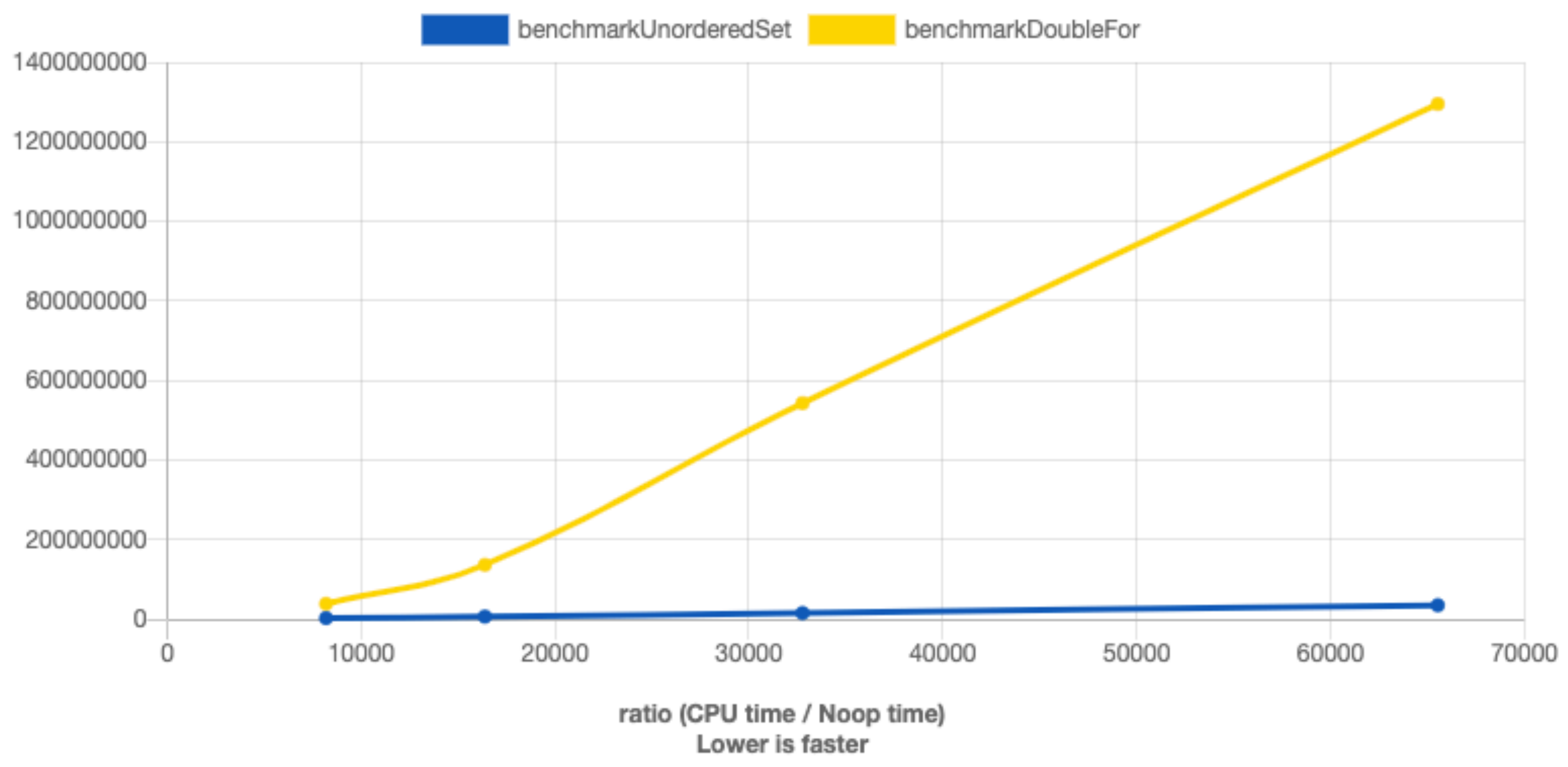

How about now?

# Large Problem Size

The first simplification of big Oh notation is that it ignores constant factors.

The second is that it is only concerned with how a function grows as the input becomes large.

We want to say the blue line grows more slowly than the yellow one.



# Definition

Let $f$ be a function which maps a natural number to a non-negative real number.

Think about the input as a problem size and the output as a complexity measure, like running time or memory usage.

Formally, we say $f : \{0, 1, 2, 3, \ldots\} \to \mathbb{R}_{\geq 0}$ .

# Definition

For $f, g : \{0, 1, 2, 3, \ldots\} \to \mathbb{R}_{\geq 0}$ we say that $f(n) = O(g(n))$
if and only if there are positive constants $c$ and $n_0$ such that

$$f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$.

# Example

For $f, g : \{0, 1, 2, 3, \ldots\} \to \mathbb{R}_{\geq 0}$ we say that $f(n) = O(g(n))$ if and only if there are positive constants $c$ and $n_0$ such that

$$f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$.

- $2n = O(n)$

  Take the constant $c$ to be 2 and $n_0$ to be 0.

# Example

For $f, g : \{0, 1, 2, 3, \ldots\} \to \mathbb{R}_{\geq 0}$ we say that $f(n) = O(g(n))$ if and only if there are positive constants $c$ and $n_0$ such that

$$f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$.

Example: $5n = O(n^2)$

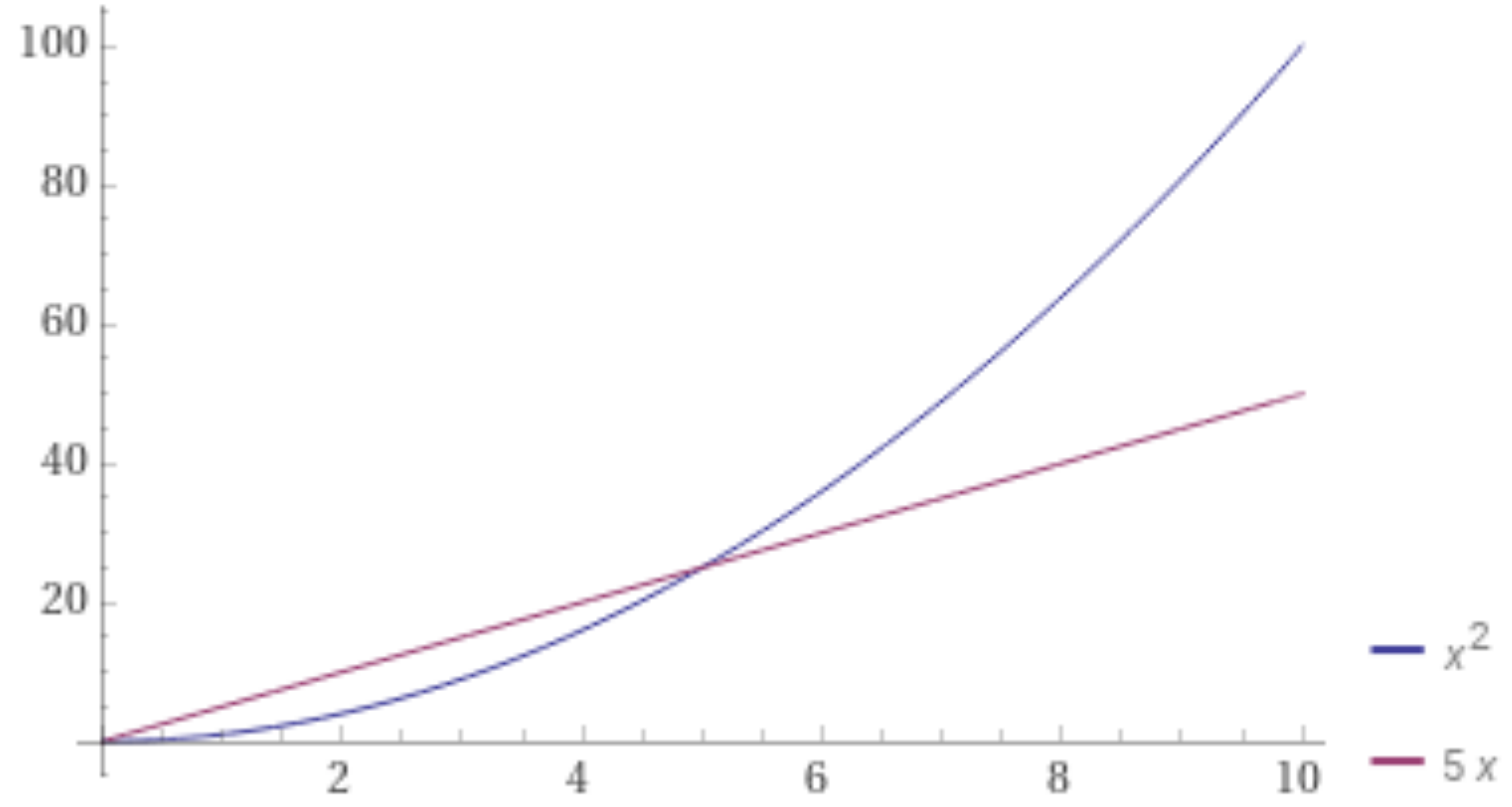
Take the constant $c$ to be 5 and $n_0$ to be 0.

For $f, g : \{0, 1, 2, 3, \ldots\} \to \mathbb{R}_{\geq 0}$ we say that $f(n) = O(g(n))$ if and only if there are positive constants $c$ and $n_0$ such that

$$f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$.

Example: $5n = O(n^2)$




Or take the constant $c$ to be 1 and $n_0$ to be 6.

# Non-Example

For $f, g : \{0, 1, 2, 3, \ldots\} \to \mathbb{R}_{\geq 0}$ we say that $f(n) = O(g(n))$ if and only if there are positive constants $c$ and $n_0$ such that

$$f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$.

Non-Example: $n^3 \neq O(n^2)$

$$n^3 > cn^2 \text{ for } n > c\,.$$

# Sufficient Condition

When trying to figure out if $f(n) = O(g(n))$ look at the ratio as becomes large. If there is a constant $n$ such that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c$$

then $f(n) = O(g(n))$.

# Example

We can use the sufficient condition to show that if $a < b$ then

$$n^a = O(n^b)$$

Look at the limit of the ratio

$$\lim_{n\to\infty} \frac{n^a}{n^b} = \lim_{n\to\infty} \frac{1}{n^{b-a}}$$

$$= 0$$

# Big Oh is not enough

Big Oh is just an upper bound. There is no implication that it is the best upper bound possible.

"The running time of the double for loop contains duplicate algorithm is $O(n^3)$."

This is a true statement.

To compare algorithms we also want to lower bound their running time.

Unfortunately, people have occasionally been using the O-notation for lower bounds, for example when they reject a particular sorting method "because its running time is $O(n^2)$ ." I have seen instances of this in print quite often, and finally it has prompted me to sit down and write a Letter to the Editor about the situation.

Donald E. Knuth, "Big Omicron and Big Omega and Big Theta", 1976.

In this paper Knuth introduced the Big Omega notation to computer science that we now use to talk about lower bounds on the running time of algorithms.

# Big Omega

For $f, g : \{0, 1, 2, 3, \ldots\} \to \mathbb{R}_{\geq 0}$ we say that $f(n) = \Omega(g(n))$ if and only if there are positive constants $c$ and $n_0$ such that

$$f(n) \geq c \cdot g(n)$$

for all $n \geq n_0$ .

Alternatively, $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$ .

# Example Usage

We saw that the double for loop contains duplicate algorithm makes $n(n-1)/2$ comparisons when the input does not have a duplicate.

"The worst-case number of comparisons is $n(n-1)/2 = \Omega(n^2)$ "

Worst-case means there exists an input which makes the algorithm have this number of comparisons (or running time).

$$\frac{n(n-1)}{2} \geq \frac{n^2}{4} \qquad \text{for } n \geq 2\,.$$

# Example Usage

We can also say the worst-case running time of the double for loop contains duplicate algorithm is $O(n^2)$ .

For every input the number of steps is $O(n^2)$ .

We have matching upper and lower bounds on the running time of this algorithm...this is a job for Big Theta.

# Big Theta

For $f, g : \{0, 1, 2, 3, \ldots\} \to \mathbb{R}_{\geq 0}$ we say that $f(n) = \Theta(g(n))$ if and only if there are positive constants $c_1, c_2$ and $n_0$ such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

for all $n \geq n_0$.

Alternatively, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

# Example Usage

The worst-case running time of double for loop contains duplicate is $\Theta(n^2)$.

For every input it runs in $O(n^2)$ steps.

There exists an input for which it takes $\Omega(n^2)$ steps.

# Caution

Examples of incorrect usage of big Oh abound.

Knuth's observation from 1976 still holds today: many people use big Oh when they mean big Omega or big Theta.

"In industry, people seem to have merged $O$ and $\Theta$ together. Industry's meaning of big $O$ is closer to what academics mean by $\Theta$, in that it would be seen as incorrect to describe printing an array as $O(n^2)$ ."

Gayle Laakmann McDowell, "Cracking the Coding Interview"

Be a force for good, and use the terms properly!

# Common Functions

# Common Functions

Here are the most common functions you will see in the analysis of algorithms.

$\Theta(1)$ — assigning a word in memory, arithmetic operation on words.

$\Theta(\log n)$ — finding an element in a sorted array of size $n$.

$\Theta(n)$ — iterate through an array of size $n$ .

$\Theta(n \log n)$ — sorting an array of size $n$ with mergesort.

$\Theta(n^2)$ — sorting an array of size $n$ with insertion sort.

$\Theta(n^3)$ — solving $n$ linear equations in $n$ vars with Gaussian elim.

$\Theta(2^n)$ — enumerating all subsets of an $n$ element set.

# Big Oh vs. Problem Size

| $n$ | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(n)$ | $\Theta(n \log n)$ | $\Theta(n^2)$ | $\Theta(n^3)$ | $\Theta(2^n)$ |
|---|---|---|---|---|---|---|---|
| **10** | 1 ns | 3 ns | 10 ns | 30 ns | 100 ns | 1 microsec | 1 microsec |
| **100** | 1 ns | 6 ns | 100 ns | 600 ns | 10 microsec | 1 ms | 40 trillion yrs |
| **1,000** | 1 ns | 10 ns | 1 microsec | 10 microsec | 1 ms | 1 sec | |
| **10,000** | 1 ns | 13 ns | 10 microsec | 130 microsec | 100 ms | 16 min | |
| **100,000** | 1 ns | 16 ns | 100 microsec | 1.6 ms | 10 sec | 277 hours | |
| **1,000,000** | 1 ns | 20 ns | 1 ms | 20 ms | 16 min | 32 yrs | |

one operation per nanosecond

# Time Estimates

| n | O(1) | O(log n) | O(n) | O(n log n) | $O(n^2)$ | $O(n^3)$ | $O(2^n)$ | O(n!) |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 μs | 1 μs | 1 μs | 1 μs | 1 μs | 1 μs | 2 μs | 1 μs |
| 10 | 1 μs | 3 μs | 10 μs | 34 μs | 100 μs | 1 ms | 1 ms | 3.6 seconds |
| 100 | 1 μs | 6 μs | 100 μs | 665 μs | 10 ms | 1 sec | >400 trillion centuries | >googol centuries |
| 1,000 | 1 μs | 9 μs | 1 ms | ~10 ms | 1 sec | 16.67 min | ... | ... |
| 10,000 | 1 μs | 13 μs | 10 ms | ~133 ms | 1.67 min | ~12 days | ... | ... |
| 100,000 | 1 μs | 16 μs | 100 ms | 1.67 sec | 2.78 hours | ~32 years | ... | ... |
| 1,000,000 | 1 μs | 19 μs | 1 sec | ~20 sec | ~12 days | ~32,000 years | ... | ... |

* let's assume our single operation takes 1 μs

Cppcon 2021: https://www.youtube.com/watch?v=AY2FqpDCBGs

# Intro to Sorting

# Sorting

A sequence $a_0, a_1, \ldots, a_{n-1}$ is sorted in ascending order if

$$a_0 \leq a_1 \leq \cdots \leq a_{n-1}$$

Example: $1, 2, 3, 5, 7, 7, 8$

A sequence $a_0, a_1, \ldots, a_{n-1}$ is sorted in descending order if

$$a_0 \geq a_1 \geq \cdots \geq a_{n-1}$$

Example: $8, 7, 7, 5, 3, 2, 1$

# Sorting Algorithm

A sorting algorithm takes an input array and puts it in sorted order (either ascending or descending).

We can sort any objects where a comparison function $<$ is defined.

The default comparison for strings is by alphabetical order.

For pairs of numbers, by default $(a, b) < (c, d)$ if and only if

$$a < c \text{ or } a = c \text{ and } b < d$$

# Sorting in C++

```
// Example 1: Sort vector of integers in ascending order
std::vector<int> intVec {3,1,7,2,5,8,7};
std::sort(intVec.begin(), intVec.end());
```

Now `intVec` is sorted in ascending order.

```
// Example 4: Sort in descending order
std::sort(intVec.begin(), intVec.end(), std::greater<>());
```

This sorts in descending order.

These and more examples at Godbolt: https://godbolt.org/z/aossW7jE9

# Sorting Application

Sorting is used as a subroutine in many algorithms.

You can also use sorting to solve Contains Duplicate!

First sort the array.

If there are duplicate elements, they will appear next to each other in the sorted array!

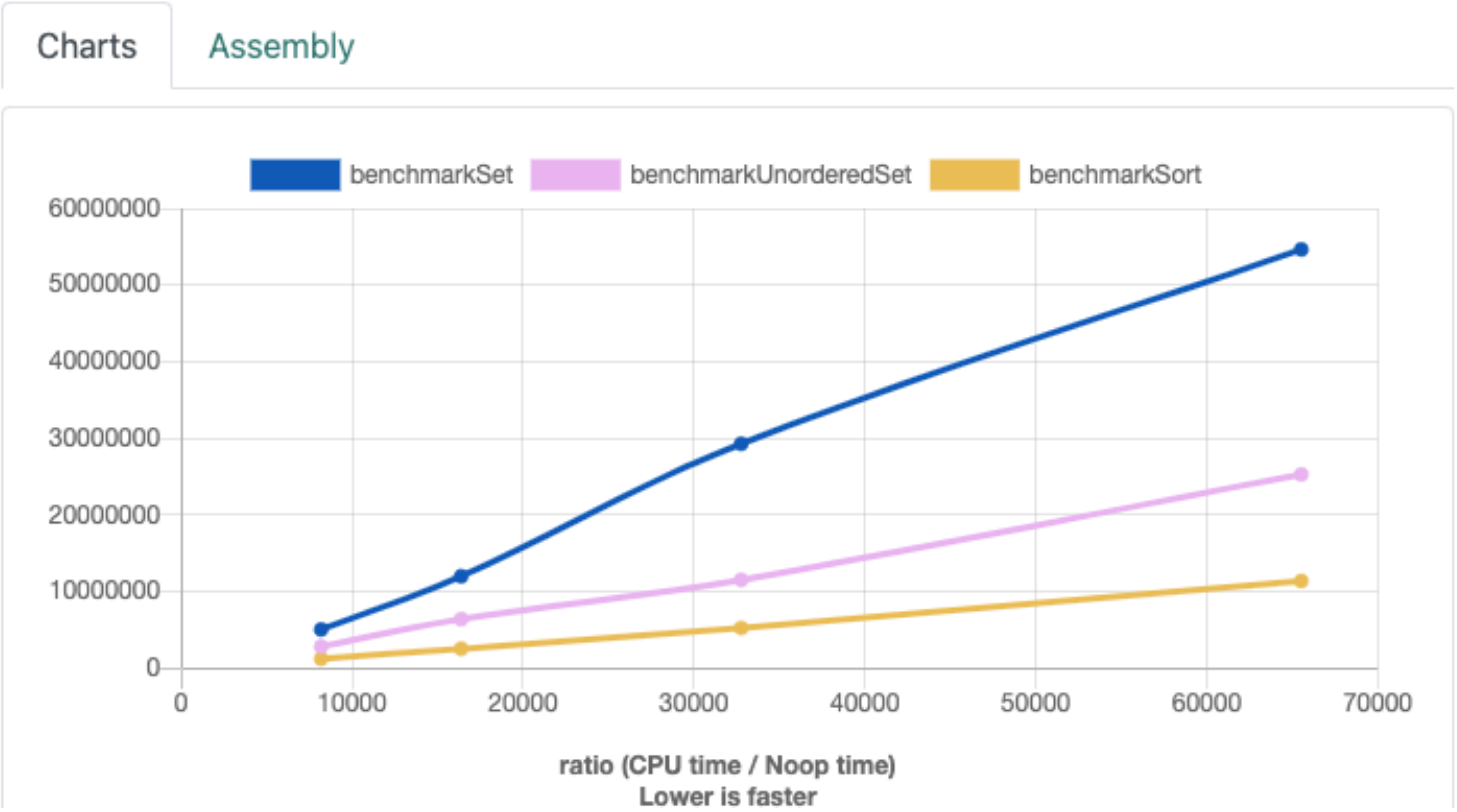
We can check this with one more pass through the array.

# Duplicates Via Sorting

```
bool containsDuplicateSort(std::vector<int>& vec) {
    std::sort(vec.begin(), vec.end());
    for(std::size_t i = 1; i < vec.size(); ++i) {
        if(vec[i] == vec[i-1]) {
            return true;
        }
    }
    return false;
}
```

https://godbolt.org/z/b6W9WKTbc

# Duplicates Via Sorting




https://quick-bench.com/q/ObREPxdViS_tS0iDF7jDkGR5MQg

# Example 2

Cracking the Coding Interview Problem 1.2:

Given two strings, determine if one is a permutation of the other.

Example: "cab" is a permutation of "abc".

How could you solve this problem?

# Sorting Algorithms

# Sorting Algorithms

There is a huge literature on sorting algorithms.

Knuth's The Art of Computer Programming, Volume 3, has nearly 400 pages on sorting algorithms

We will focus on 6 sorting algorithms:

- Insertion Sort
- Mergesort
- Quicksort
- Heapsort
- Counting Sort
- Radix Sort

# Insertion Sort

# Insertion Sort

Insertion sort is how we might sort cards in our hands.

We maintain the cards in our hand sorted, then pick up a new card and insert it in the right position.

# Inserting One Element

Imagine we have an array which is sorted except for the last element.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 7 | 8 | 2 |

We now want to insert the last element into its proper position.

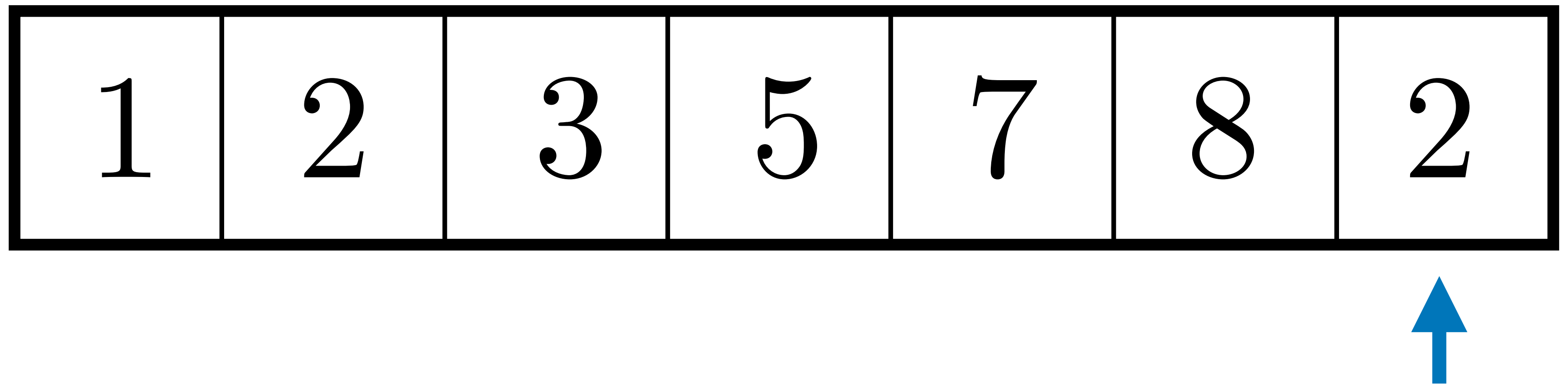
Idea: As long as the last 2 is smaller than the element before it, swap their positions.

# Inserting One Element

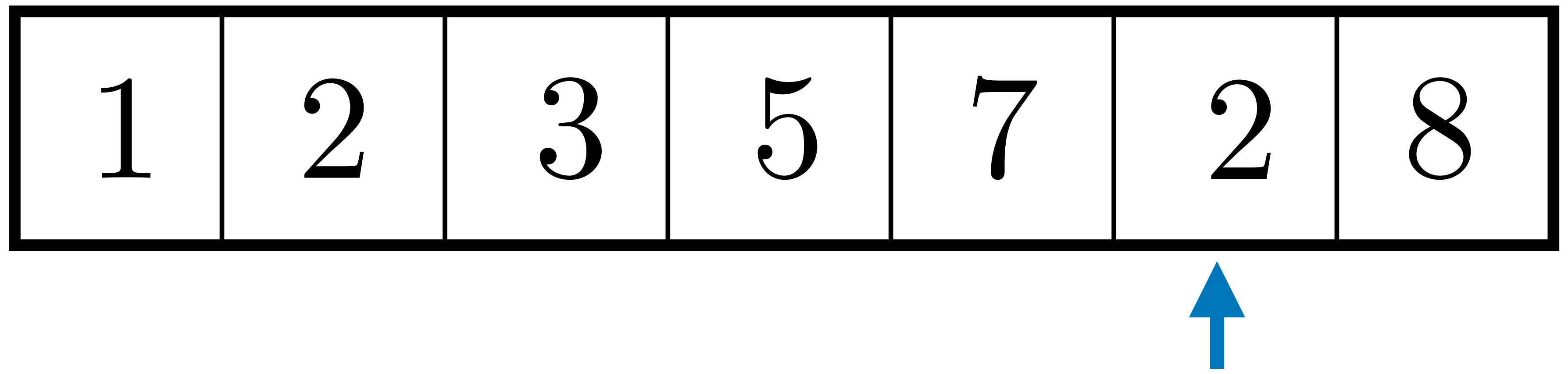
| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 7 | 8 | 2 |

$2 < 8$ : swap them.

# Inserting One Element




$2 < 8$ : swap them.




$2 < 7$ : swap them.

# Inserting One Element

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 7 | 8 | 2 |

$2 < 8$ : swap them.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 7 | 2 | 8 |

$2 < 7$ : swap them.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 2 | 7 | 8 |

# Inserting One Element

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 2 | 7 | 8 |

$2 < 5$ : swap them.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 5 | 7 | 8 |

# Inserting One Element

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 2 | 7 | 8 |

$2 < 5$ : swap them.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 2 | 5 | 7 | 8 |

$2 < 3$ : swap them.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 5 | 7 | 8 |

# Inserting One Element

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 5 | 7 | 8 |

$2 \not< 2$ : we are done.

The whole array is sorted now.

# Inserting One Element

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 5 | 7 | 8 |

$2 \not< 2$ : we are done.

The whole array is sorted now.

Note: When we only swap if the element is strictly smaller we preserve the original ordering of equal elements in the array.

The 2 that started in the last position stays to the right of the 2 that started in the second position.

# Complexity: Memory

```cpp
// Assumption: elements 0 through i-1 of vec are sorted
void insertOne(std::vector<int>& vec, std::size_t i) {
  while(i >= 1 && vec[i] < vec[i-1]) {
    std::swap(vec[i], vec[i-1]);
    --i;
  }
}
```

https://godbolt.org/z/EfPfb3a5P

Notice we just use the variable $i$ to keep track of our position.

Doing a swap also requires a temporary variable holding an `int`.

In Place: An algorithm that only uses a constant number of extra variables to hold indices and elements is called in place.

# Complexity: Time

How many operations do we have to do in the worst case?

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 7 | 8 | 0 |

In the worst case, the inserted element must travel all the way to the beginning.

When the sorted portion of the array has size $i$, we have to do $i$ comparisons and $i$ swaps.

A swap can be done in a constant number of operations, so the total number of operations in the worst case is $\Theta(i)$.

# Insertion Sort

```cpp
// Assumption: elements 0 through i-1 of vec are sorted
void insertOne(std::vector<int>& vec, std::size_t i) {
  while(i >= 1 && vec[i] < vec[i-1]) {
    std::swap(vec[i], vec[i-1]);
    --i;
  }
}

void insertionSort(std::vector<int>& vec) {
  for(std::size_t i = 1; i < vec.size(); ++i) {
    insertOne(vec, i);
  }
}
```

We iterate through the list starting from $i = 1$ and insert the $i^{th}$ element into the right position among elements $0, \ldots, i-1$.

# Insertion Sort: Invariant

An invariant is something that stays true throughout an algorithm.

They can help us argue that an algorithm is correct.

```
void insertionSort(std::vector<int>& vec) {
  for(std::size_t i = 1; i < vec.size(); ++i) {
    insertOne(vec, i);
  }
}
```

Invariant: At the start of the $i^{th}$ iteration of the for loop, $\texttt{vec}[0], \ldots, \texttt{vec}[\texttt{i} - 1]$ are in sorted order.

# Insertion Sort: Invariant

```
void insertionSort(std::vector<int>& vec) {
  for(std::size_t i = 1; i < vec.size(); ++i) {
    insertOne(vec, i);
  }
}
```

Invariant: At the start of the $i^{th}$ iteration of the for loop, $\texttt{vec}[0], \ldots, \texttt{vec}[\texttt{i} - 1]$ are in sorted order.

Initialization: When $i = 1$ this just says $\texttt{vec}[0]$ is sorted, which is true.

# Insertion Sort: Invariant

```
void insertionSort(std::vector<int>& vec) {
  for(std::size_t i = 1; i < vec.size(); ++i) {
    insertOne(vec, i);
  }
}
```

Invariant: At the start of the $i^{th}$ iteration of the for loop, $\texttt{vec}[0], \ldots, \texttt{vec}[\texttt{i} - 1]$ are in sorted order.

Maintenance: If the invariant holds in the $i^{th}$ iteration then $\texttt{vec}[0], \ldots, \texttt{vec}[\texttt{i} - 1]$ are sorted when we call $\texttt{insertOne}(\texttt{vec}, \texttt{i})$.

This will insert $\texttt{vec}[\texttt{i}]$ in the correct position so that $\texttt{vec}[0], \ldots, \texttt{vec}[\texttt{i}]$ are in sorted order.

The invariant holds at the start of iteration $i + 1$.

# Insertion Sort: Invariant

```
void insertionSort(std::vector<int>& vec) {
  for(std::size_t i = 1; i < vec.size(); ++i) {
    insertOne(vec, i);
  }
}
```

Invariant: At the start of the $i^{th}$ iteration of the for loop, $\texttt{vec}[0], \ldots, \texttt{vec}[\texttt{i} - 1]$ are in sorted order.

Termination: The for loop terminates when $i = \texttt{vec.size}()$.

The invariant tells us that $\texttt{vec}[0], \ldots, \texttt{vec}[\texttt{vec.size}() - 1]$ are sorted!

# Insertion Sort: Running Time

```
void insertionSort(std::vector<int>& vec) {
  for(std::size_t i = 1; i < vec.size(); ++i) {
    insertOne(vec, i);
  }
}
```

We have seen that $\texttt{insertOne}(\texttt{vec}, \texttt{i})$ takes $\Theta(i)$ steps in the worst case.

The total running time is at most a constant times

$$1 + 2 + \cdots + n - 1 = n(n-1)/2$$

The running time is $O(n^2)$ .

# Insertion Sort: Complexity

```
void insertionSort(std::vector<int>& vec) {
  for(std::size_t i = 1; i < vec.size(); ++i) {
    insertOne(vec, i);
  }
}
```

Is there an input that makes the algorithm take $\Omega(n^2)$ steps?

# Insertion Sort: Complexity

```
void insertionSort(std::vector<int>& vec) {
  for(std::size_t i = 1; i < vec.size(); ++i) {
    insertOne(vec, i);
  }
}
```

Is there an input that makes the algorithm take $\Omega(n^2)$ steps?

Remember the worst case for $\texttt{insertOne}(\texttt{vec}, \texttt{i})$ is when $\texttt{vec}[\texttt{i}]$ is smaller than all of $\texttt{vec}[0], \ldots, \texttt{vec}[\texttt{i} - 1]$ .

# Insertion Sort: Complexity

```
void insertionSort(std::vector<int>& vec) {
  for(std::size_t i = 1; i < vec.size(); ++i) {
    insertOne(vec, i);
  }
}
```

Is there an input that makes the algorithm take $\Omega(n^2)$ steps?

Remember the worst case for $\mathtt{insertOne}(\mathtt{vec}, \mathtt{i})$ is when $\mathtt{vec}[\mathtt{i}]$ is smaller than all of $\mathtt{vec}[0], \ldots, \mathtt{vec}[\mathtt{i}-1]$ .

Is there an input that always realizes the worst case of $\mathtt{insertOne}(\mathtt{vec}, \mathtt{i})$?

# Reverse Sorted

| | | | | | | |
|---|---|---|---|---|---|---|
| 8 | 7 | 5 | 3 | 2 | 1 | 0 |

If the original array is sorted in descending order then $\texttt{insertOne}(\texttt{vec}, \texttt{i})$ must always move $\texttt{vec}[\texttt{i}]$ to the front, which takes $\Omega(i)$ steps.

On an array of size $n$ sorted in descending order insertion sort will take $\Omega(n^2)$ steps.

The worst-case running time of insertion sort is $\Theta(n^2)$ .

# Best Case

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 5 | 7 | 8 |

If the original array is already sorted in ascending order then insertion sort takes $\Theta(n)$ steps.

The best-case running time of insertion sort is $\Theta(n)$.

# Properties of Insertion Sort

# Properties

This is a good time to introduce some general properties of sorting algorithms.

In Place: Only a constant number of auxiliary variables (to hold indices or elements) are used, in addition to the input array.

Stable: In the sorted array elements that compare equal are in the same relative order as in the input.

Comparison Based: The algorithm only makes use of a comparison function $<$ on the elements.

Insertion sort has all of these properties.

# Insertion Sort is Stable

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 7 | 8 | 2 |

start of `insertOne`

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 5 | 7 | 8 |

end of `insertOne`

$2 \not< 2$ : we are done.

Insertion sort is stable because we only swap when an element is strictly less than its predecessor.

An element cannot move past an equal element that begins to the left of it.

# Stable

Why would you want a sorting algorithm to be stable?

Say we wanted to sort this list of Last Name, First Name pairs.

Apple, John

Orange, Tim

Apple, Elsa

Orange, Anna

# Stable

Why would you want a sorting algorithm to be stable?

Say we wanted to sort this list of Last Name, First Name pairs.

| | | |
|---|---|---|
| Apple, John | | Orange, Anna |
| Orange, Tim | → | Apple, Elsa |
| Apple, Elsa | sort by first name | Apple, John |
| Orange, Anna | | Orange, Tim |

# Stable

Say we wanted to sort this list of Last Name, First Name pairs.

Orange, Anna

Apple, Elsa

Apple, John

Orange, Tim

For people with the same last name, a stable sort preserves the sorted order of first names.

# Stable

Say we wanted to sort this list of Last Name, First Name pairs.

| | | |
|---|---|---|
| Orange, Anna | | Apple, Elsa |
| Apple, Elsa | → | Apple, John |
| Apple, John | stable sort by last name | Orange, Anna |
| Orange, Tim | | Orange, Tim |

For people with the same last name, a stable sort preserves the sorted order of first names.

# Comparison Based

```
// Assumption: elements 0 through i-1 of vec are sorted
void insertOne(std::vector<int>& vec, std::size_t i) {
  while(i >= 1 && vec[i] < vec[i-1]) {
    std::swap(vec[i], vec[i-1]);
    --i;
  }
}
```

We do not use the values `vec[i]` themselves in the algorithm.

We only compare elements to determine if we should swap them.

The algorithm can work on any type where $<$ is defined.

# Binary Search

# Insert Into Sorted Array

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 7 | 8 | 2 |

Recall the basic subroutine in insertion sort:

$$\texttt{vec}[0] \leq \texttt{vec}[1] \leq \cdots \leq \texttt{vec}[\texttt{i} - 1]$$

and we want to insert $\texttt{vec}[\texttt{i}]$ into its correct position.

We gave an algorithm with running time $\Theta(i)$ to do this.

Is there a better way?

# Binary Search

$$\texttt{vec} = \begin{array}{|c|c|c|c|c|c|}\hline 1 & 2 & 3 & 5 & 7 & 8 \\ \hline\end{array} \qquad a = 2$$

Let's abstract out the problem. Say we have a sorted array

$$\texttt{vec}[0] \leq \cdots \leq \texttt{vec}[\texttt{n} - 1]$$

We also have a number $a$. We want to find an index $i$ such that

$$\texttt{vec}[\texttt{i} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$$

# Binary Search

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $-\infty$ | 1 | 2 | 3 | 5 | 7 | 8 | $\infty$ |
| -1 | 0 | | $\cdots$ | | | n-1 | n |

We want to find an index $i$ such that $\texttt{vec}[\texttt{i} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

Let $\texttt{vec}[-1] = -\infty$ and $\texttt{vec}[\texttt{n}] = \infty$ so that such an index $i$ always exists.

(We won't actually do this in the algorithm, but it is helpful to imagine these sentinel values for the analysis.)

# Examples

| $-\infty$ | $1$ | $2$ | $3$ | $5$ | $7$ | $8$ | $\infty$ |
|---|---|---|---|---|---|---|---|
| -1 | 0 | | … | | | n-1 | n |

We want to find an index $i$ such that $\texttt{vec}[\texttt{i} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$ .

# Examples

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $-\infty$ | $1$ | $2$ | $3$ | $5$ | $7$ | $8$ | $\infty$ |
| -1 | 0 | | | … | | n-1 | n |

We want to find an index $i$ such that $\texttt{vec}[\texttt{i}-1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$ .

$a = 2$ then the output should be 2.

# Examples

| $-\infty$ | $1$ | $2$ | $3$ | $5$ | $7$ | $8$ | $\infty$ |
|---|---|---|---|---|---|---|---|
| -1 | 0 | | | … | | n-1 | n |

We want to find an index $i$ such that $\texttt{vec}[\texttt{i}-1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$ .

$a = 2$ then the output should be 2.

$a = -3$ then the output should be 0.

# Examples

| $-\infty$ | $1$ | $2$ | $3$ | $5$ | $7$ | $8$ | $\infty$ |
|---|---|---|---|---|---|---|---|
| -1 | 0 | | | $\cdots$ | | n-1 | n |

We want to find an index $i$ such that $\texttt{vec}[\texttt{i}-1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$ .

$a = 2$ then the output should be 2.

$a = -3$ then the output should be 0.

$a = 6$ then the output should be 4.

# Examples

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $-\infty$ | $1$ | $2$ | $3$ | $5$ | $7$ | $8$ | $\infty$ |
| -1 | 0 | | ... | | | n-1 | n |

We want to find an index $i$ such that $\texttt{vec}[\texttt{i}-\texttt{1}] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$ .

$a = 2$ then the output should be 2.

$a = -3$ then the output should be 0.

$a = 6$ then the output should be 4.

$a = 8$ then the output should be 6.

# Binary Search: Invariant

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $-\infty$ | $1$ | $2$ | $3$ | $5$ | $7$ | $8$ | $\infty$ |
| -1 | 0 | | ... | | | n-1 | n |

We want to find an index $i$ such that $\texttt{vec}[\texttt{i} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$ .

Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

# Binary Search: Invariant

| $-\infty$ | $1$ | $2$ | $3$ | $5$ | $7$ | $8$ | $\infty$ |
|---|---|---|---|---|---|---|---|
| -1 | 0 | | … | | | n-1 | n |

We want to find an index $i$ such that $\texttt{vec}[\texttt{i} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Initialization: Let $\texttt{left} = 0, \texttt{right} = \texttt{n}$.

The invariant holds!

# Binary Search: Invariant

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $-\infty$ | 1 | 2 | 3 | 5 | 7 | 8 | $\infty$ |
| -1 | 0 | | … | | | n-1 | n |

We want to find an index $i$ such that $\texttt{vec}[\texttt{i}-1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left}-1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Termination: When $\texttt{left} = \texttt{right}$ we are done.

Return `left` as the answer.

# Binary Search: Invariant

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $-\infty$ | $1$ | $2$ | $3$ | $5$ | $7$ | $8$ | $\infty$ |
| -1 | 0 | | | ... | | n-1 | n |

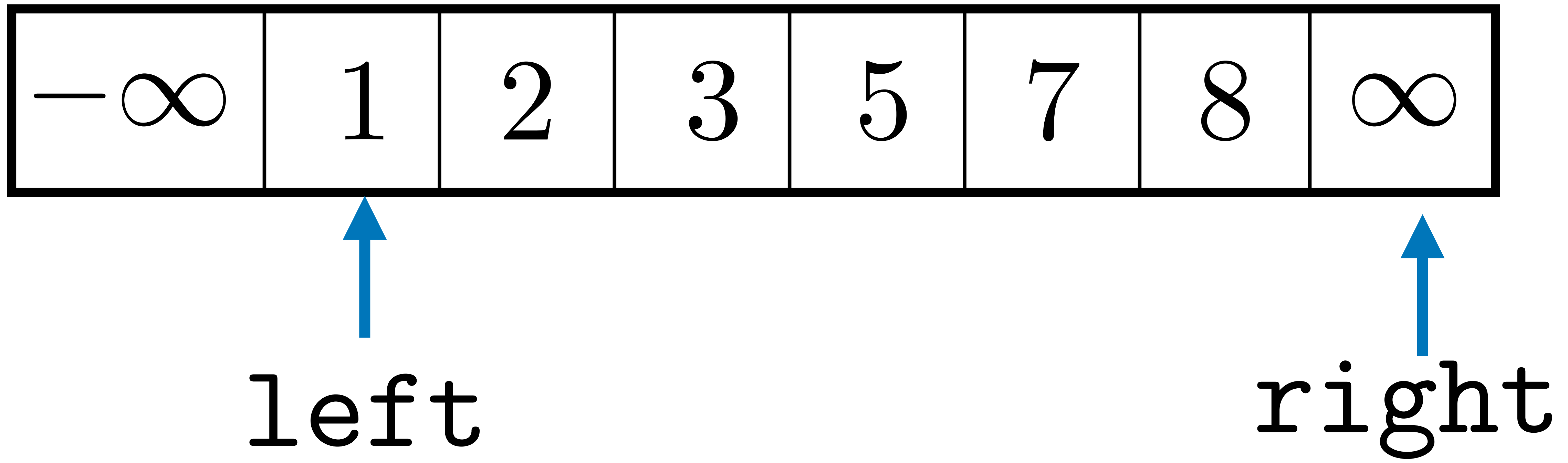
We want to find an index $i$ such that $\texttt{vec}[\texttt{i}-1] \le \texttt{a} < \texttt{vec}[\texttt{i}]$.

Invariant: Maintain two indices $\texttt{left} \le \texttt{right}$ such that

$$\texttt{vec}[\texttt{left}-1] \le \texttt{a} < \texttt{vec}[\texttt{right}]$$

Maintenance: We want to bring `left` and `right` closer together while maintaining the invariant.

# Update




$a = 2$

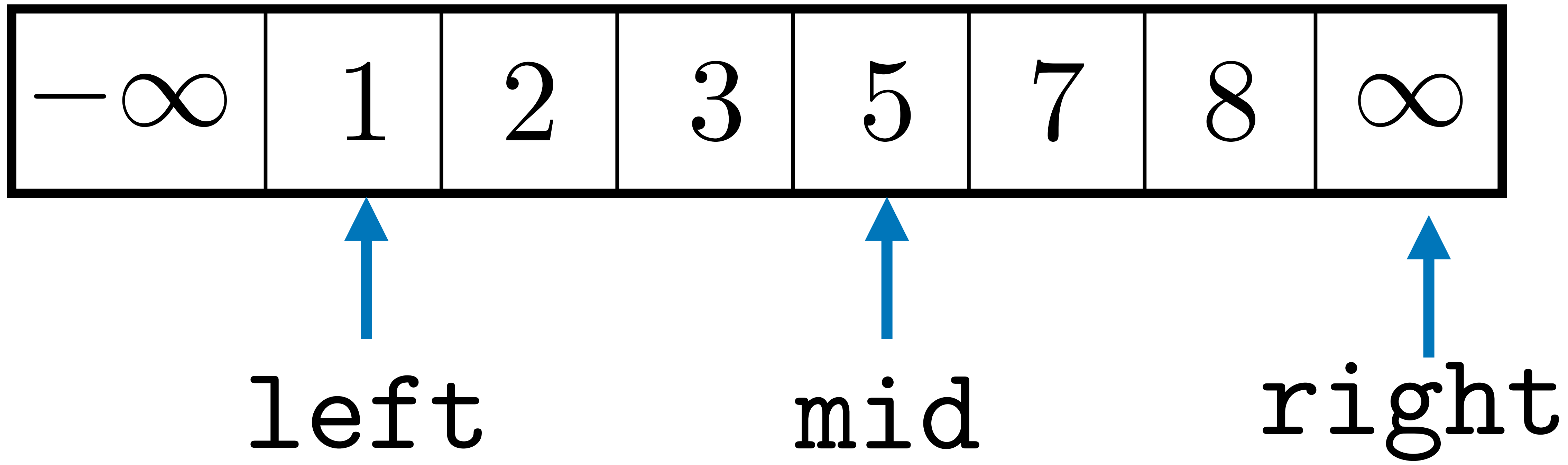
Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Update Idea: Probe the middle element between `left` and `right`.

If $a < \texttt{vec}[\texttt{mid}]$ we can update $\texttt{right} = \texttt{mid}$ and maintain the invariant.

# Update




$a = 2$

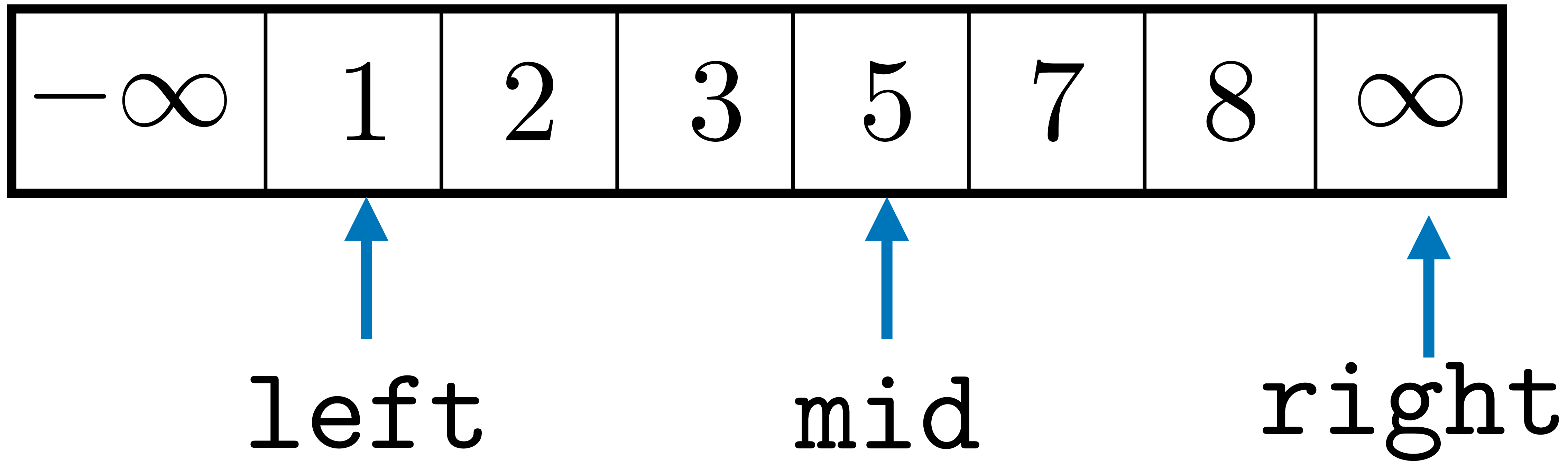
Invariant: Maintain two indices `left` $\leq$ `right` such that

$$\texttt{vec}[\texttt{left} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Update Idea: Probe the middle element between `left` and `right`.

If $a < \texttt{vec}[\texttt{mid}]$ we can update $\texttt{right} = \texttt{mid}$ and maintain the invariant.

# Update




$a = 2$

Invariant: Maintain two indices $\texttt{left} \le \texttt{right}$ such that

$$\texttt{vec}[\texttt{left} - 1] \le \texttt{a} < \texttt{vec}[\texttt{right}]$$

Update Idea: Probe the middle element between $\texttt{left}$ and $\texttt{right}$.

If $a \ge \texttt{vec}[\texttt{mid}]$ we can update $\texttt{left} = \texttt{mid} + 1$ and maintain the invariant.

# Algorithm

```
std::size_t insertionPoint(const std::vector<int>& vec, int a) {
  std::size_t left = 0;
  std::size_t right = vec.size();
  while(left < right) {
    std::size_t middle = left + (right - left)/2;
    if(a < vec[middle]) {
      right = middle;
    } else {
      left = middle + 1;
    }
  }
  return left;
}
```

https://godbolt.org/z/e7T7nTzzs

# Binary Search: Time

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 7 | 8 |

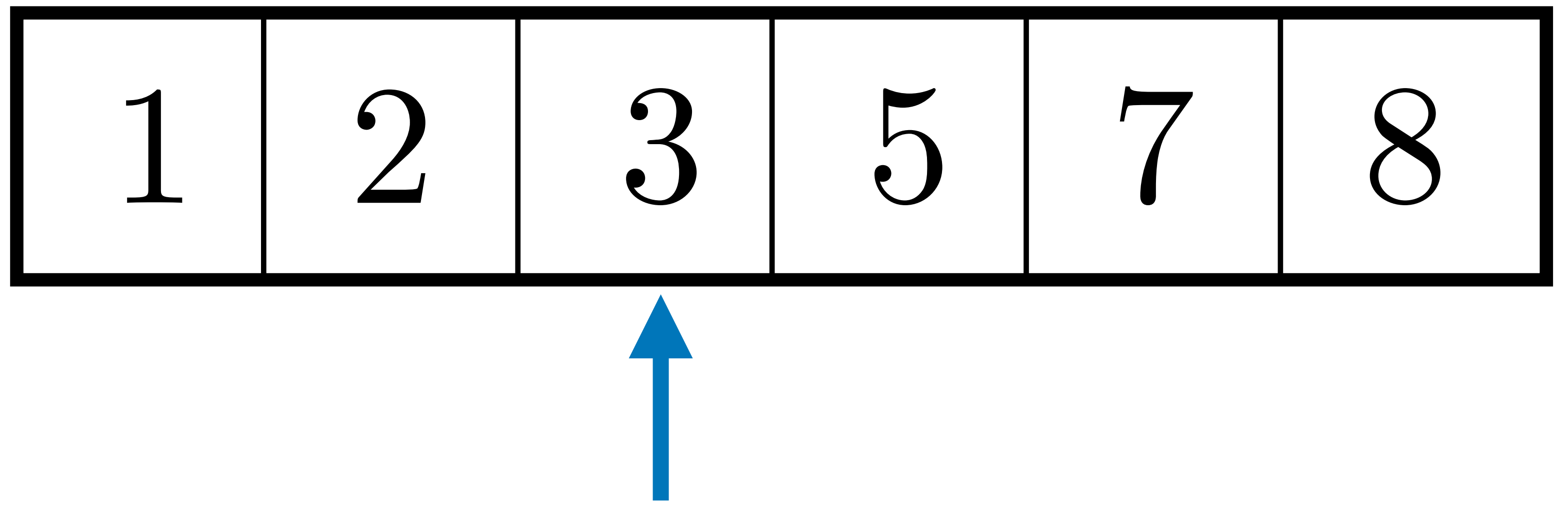
$a = 2$

The algorithm terminates when $\texttt{left} = \texttt{right}$.

The initial distance between them is $n$ , and the distance halves in each iteration.

The worst-case running time of the algorithm is $\Theta(\log n)$ .

# Inserting

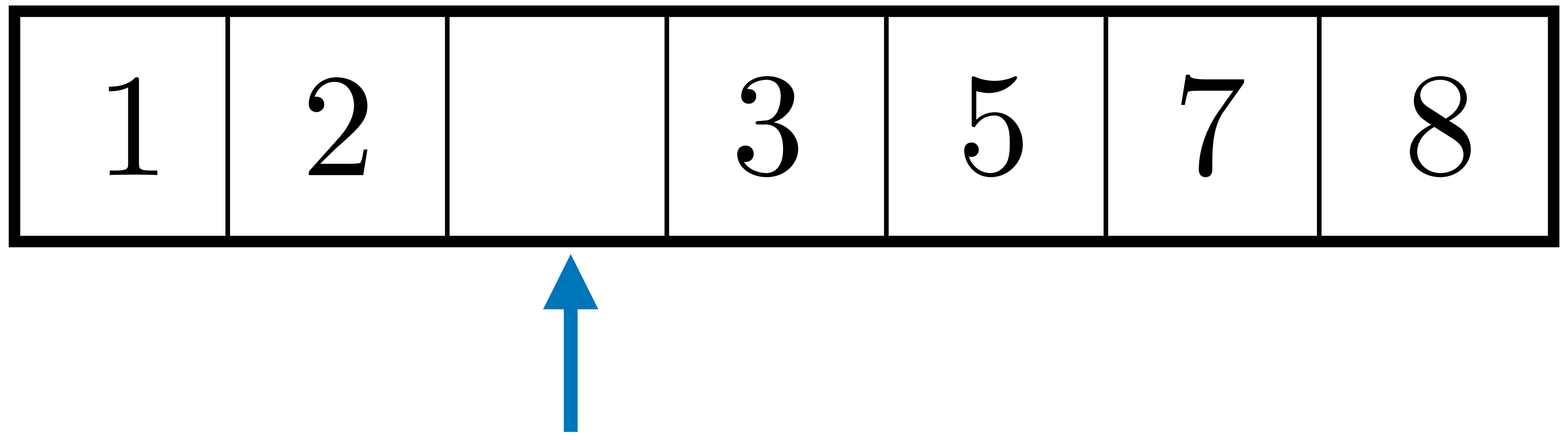



$a = 2$

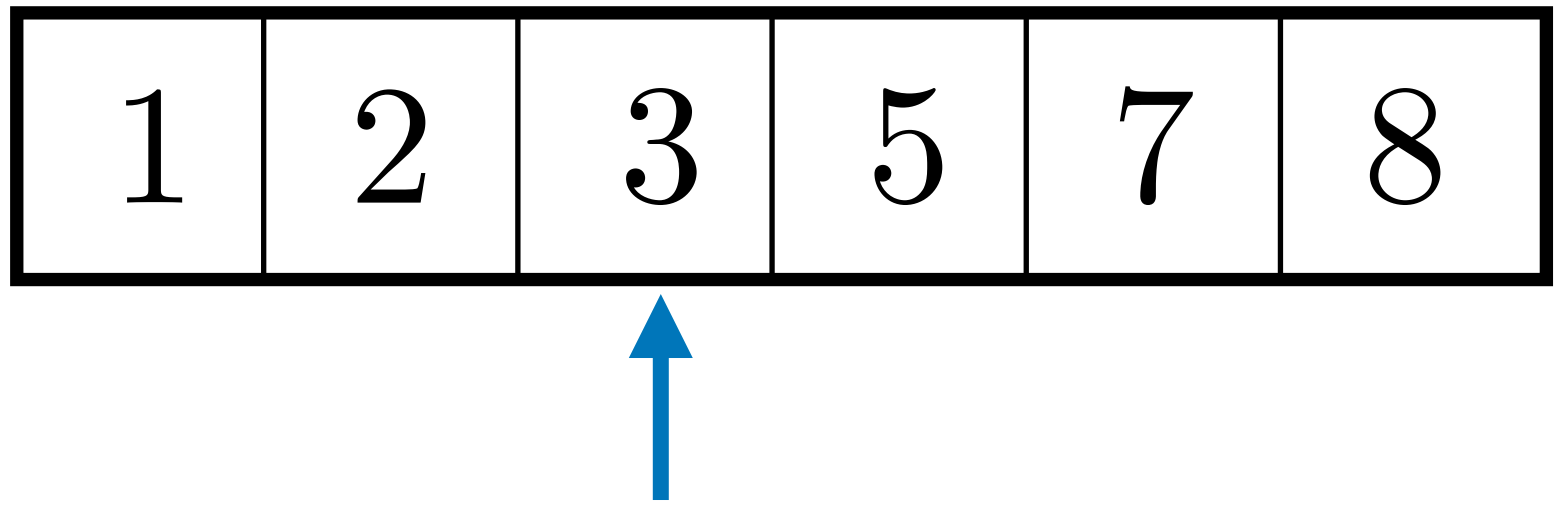
We have now found where $a$ should be inserted.

But what about the complexity of actually inserting $a$?

In a resizable array, we have to shift over all the elements to the right of the insertion point.



# Inserting

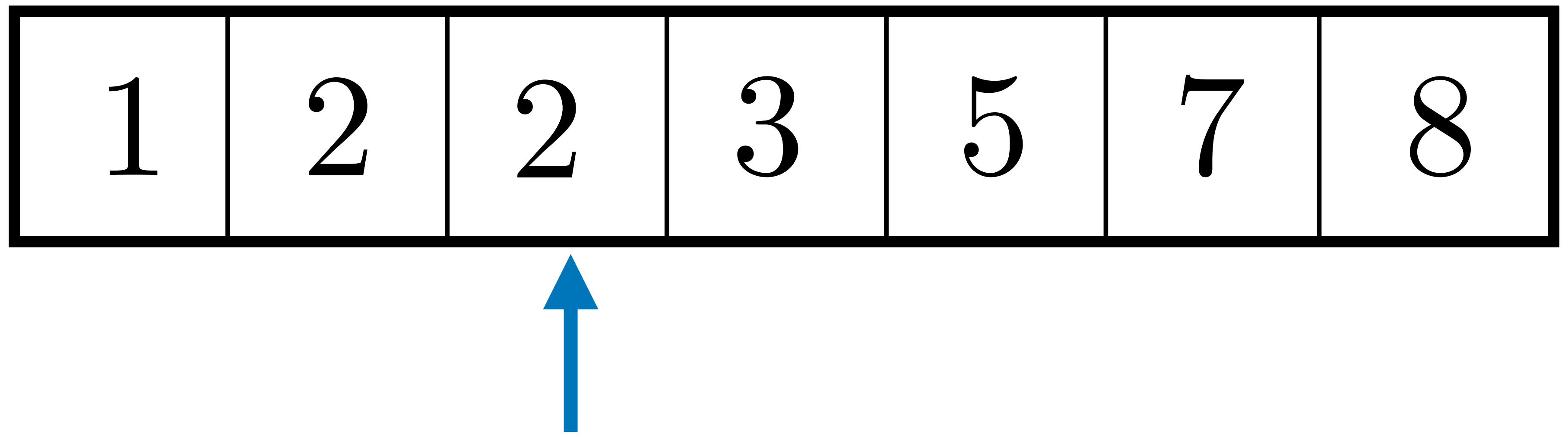


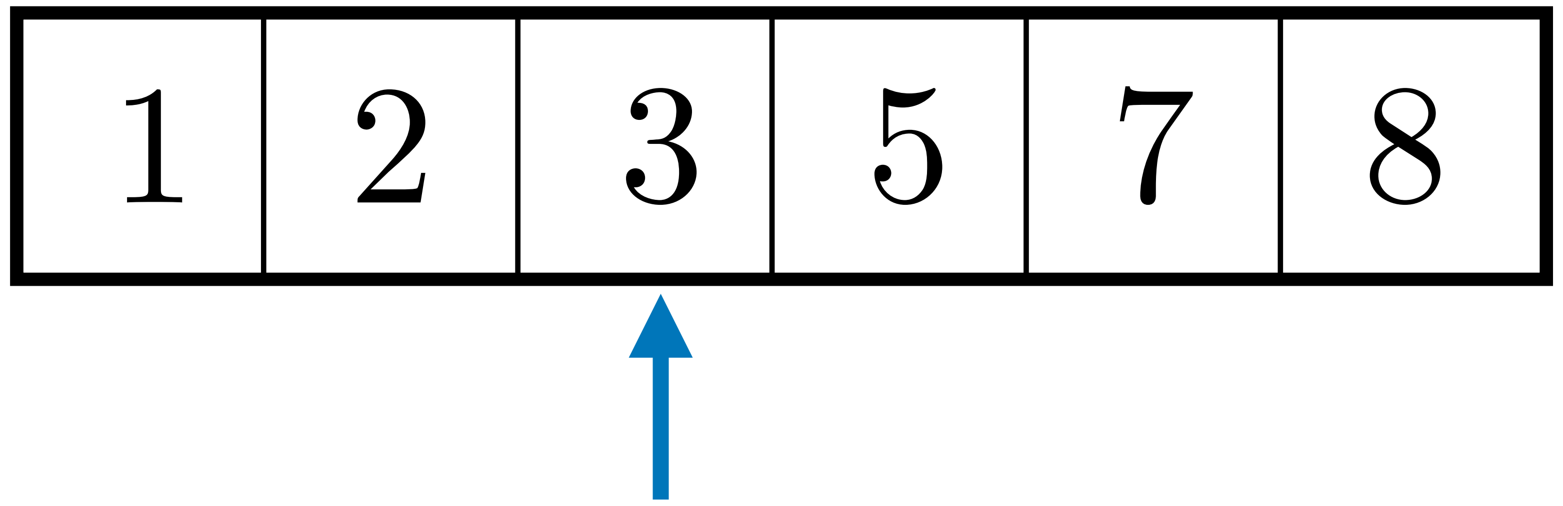

$$a = 2$$

We have now found where $a$ should be inserted.

But what about the complexity of actually inserting $a$?

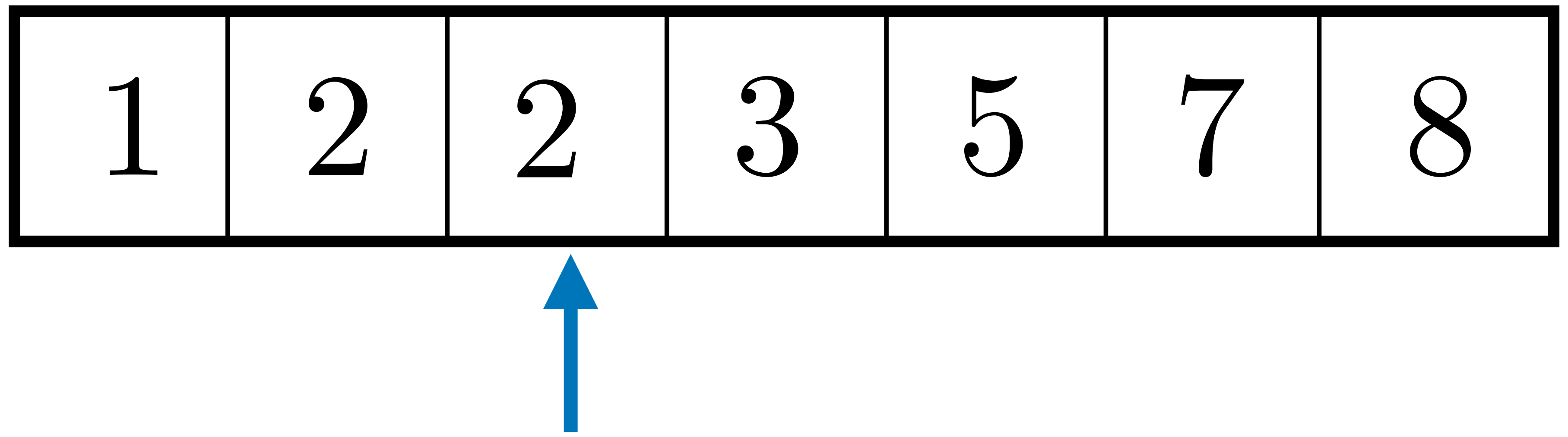
In a resizable array, we have to shift over all the elements to the right of the insertion point.






$a = 2$

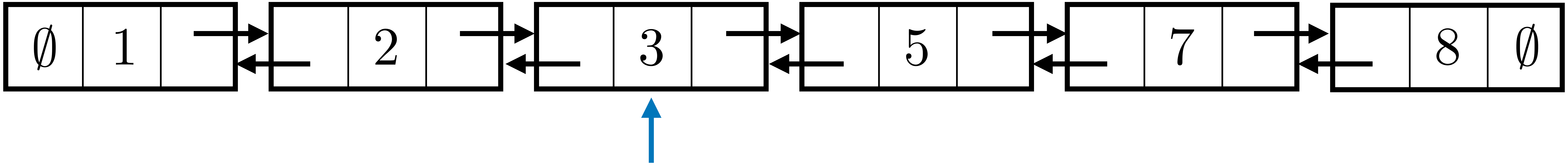
In a resizable array, we have to shift over all the elements to the right of the insertion point.




This still has a worst-case complexity of $\Theta(n)$ .

We do not realize an improvement for insertion sort.

# Doubly Linked List




Can this idea work if we use a linked list instead?

In a linked list we can insert a new node into the list in constant time.

However, we do not have random access to the elements so we cannot do binary search in $O(\log n)$ time.

Later we will look at balanced binary search trees which can maintain an ordered list with $O(\log n)$ insertion time.

# Counting Sort

# Counting Sort

Counting sort is a non-comparison based sorting algorithm.

Say that we want to sort an array of $n$ non-negative integers between $0$ and $k$ .

Counting sort can do this in time $O(n+k)$ .

For $k = O(n)$ , this is better than is possible with a comparison based sort.

Counting sort is a stable sort, but it is not in place.

# First Step

Let $A$ be the input array of size $n$ which holds non-negative integers between $0$ and $k$ .

We create an auxiliary array `counts` of size $k+2$ to hold the number of times each element of $A$ appears.

$$\texttt{counts}[\texttt{i}+\texttt{1}] = \text{number of } j \text{ where } A[j] = i$$

Example: $A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | 2 | 5 | 2 | 8 | 3 | 2 | 5 |

$\texttt{counts} =$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 3 | 1 | 0 | 2 | 0 | 0 | 1 | 1 |

# First Step

$$\texttt{counts}[\texttt{i}+\texttt{1}] = \text{number of } j \text{ where } A[j] = i$$

Example: $A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | 2 | 5 | 2 | 8 | 3 | 2 | 5 |

$\texttt{counts} =$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 3 | 1 | 0 | 2 | 0 | 0 | 1 | 1 |

This can be done in one pass through $A$ in time $O(n)$.

# Second Step

Convert the counts into indices so that $\texttt{counts}[i]$ is first location of $i$ in the sorted order (if $i$ appears in $A$).

We do this by computing the prefix sums of $\texttt{counts}$.

Example: $A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | 2 | 5 | 2 | 8 | 3 | 2 | 5 |

$\texttt{counts} =$ (after step 1)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 3 | 1 | 0 | 2 | 0 | 0 | 1 | 1 |

$\texttt{counts} =$ (after step 2)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 3 | 4 | 4 | 6 | 6 | 6 | 7 | 8 |

# Third Step

Example: $A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | 2 | 5 | 2 | 8 | 3 | 2 | 5 |

$\texttt{counts} =$ (after step 2)

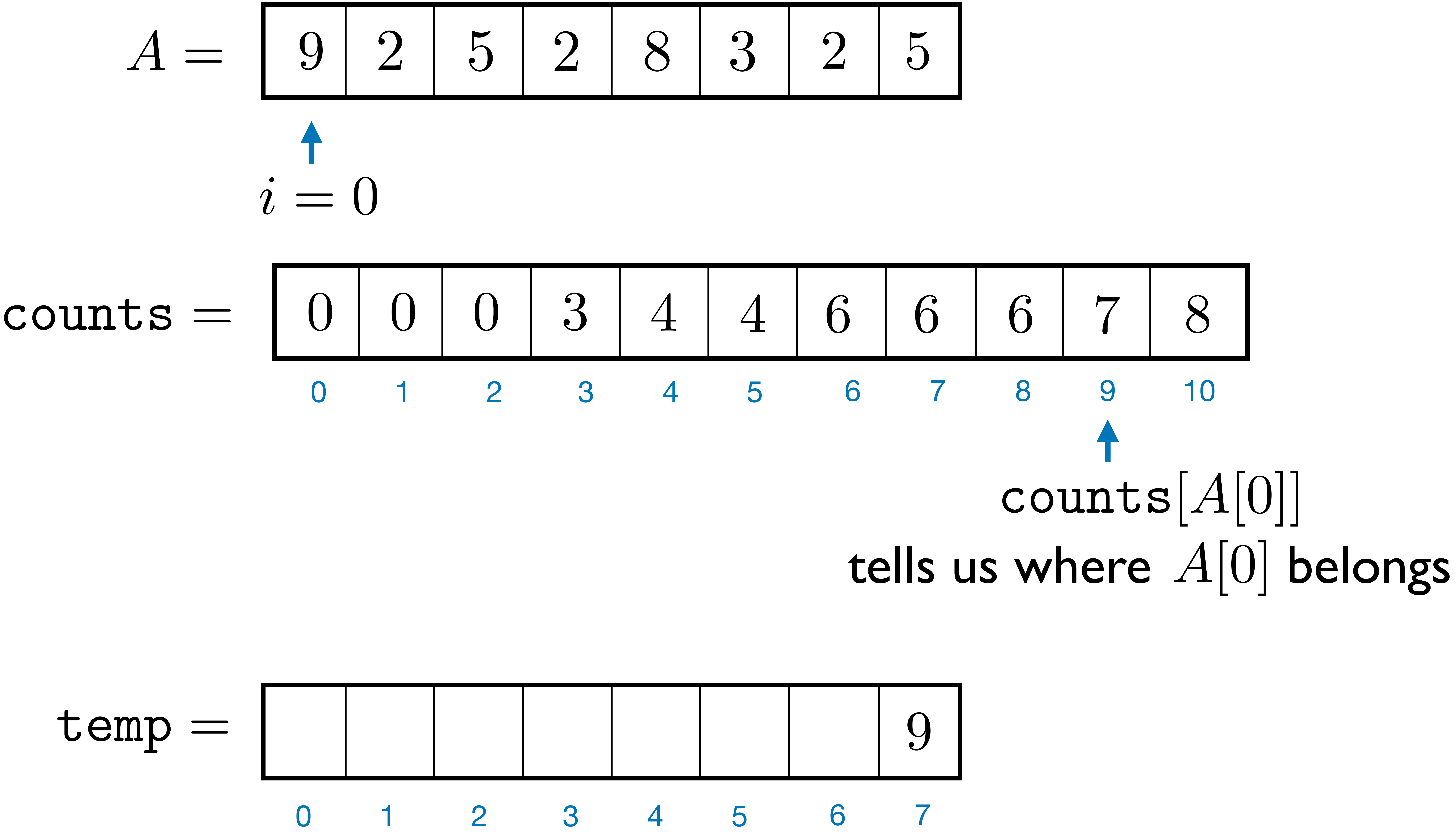
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 3 | 4 | 4 | 6 | 6 | 6 | 7 | 8 |

Write the sorted elements to an auxiliary array $\texttt{temp}$ the same size as $A$.

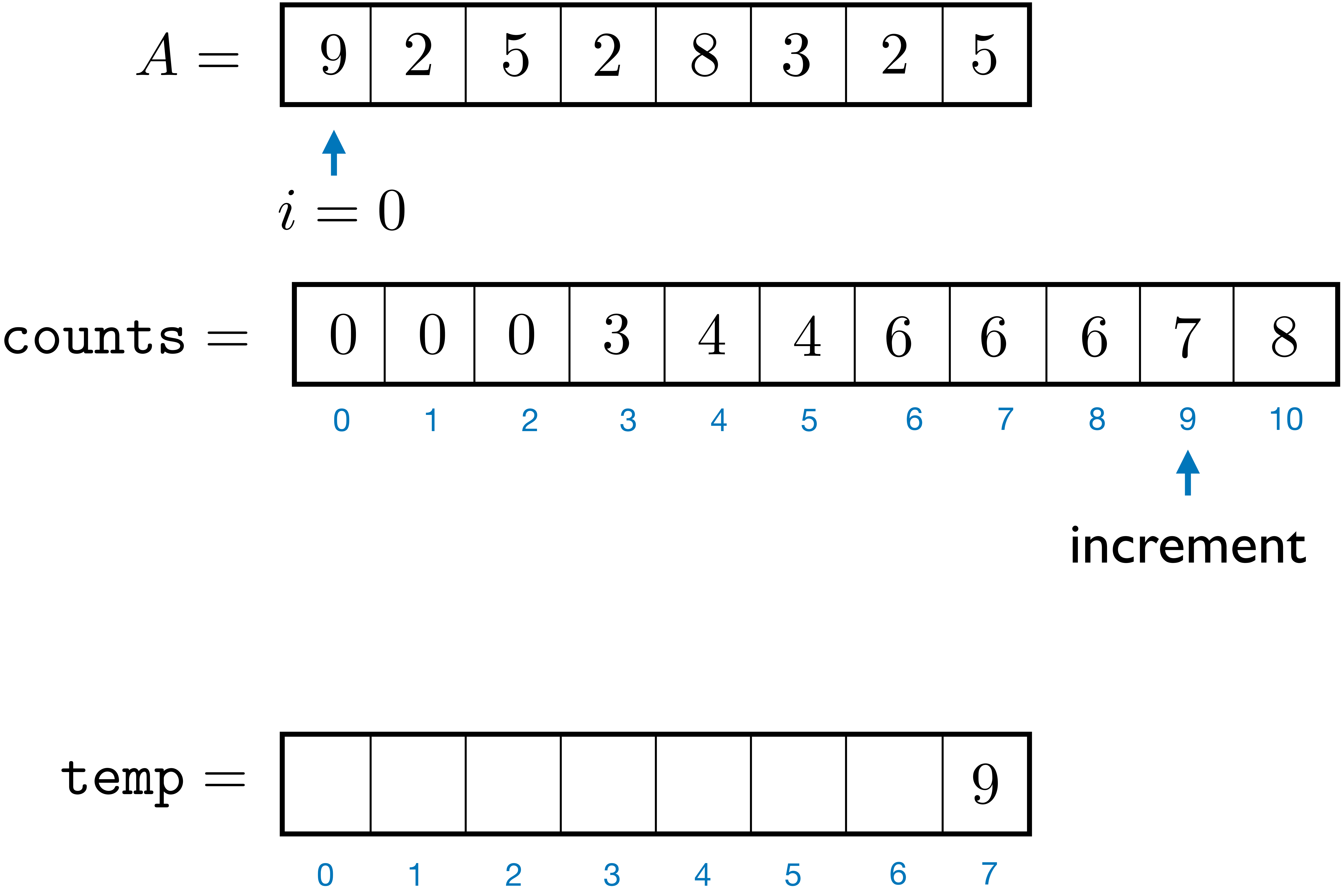
Make a pass through $A$, let $i$ be our loop variable.

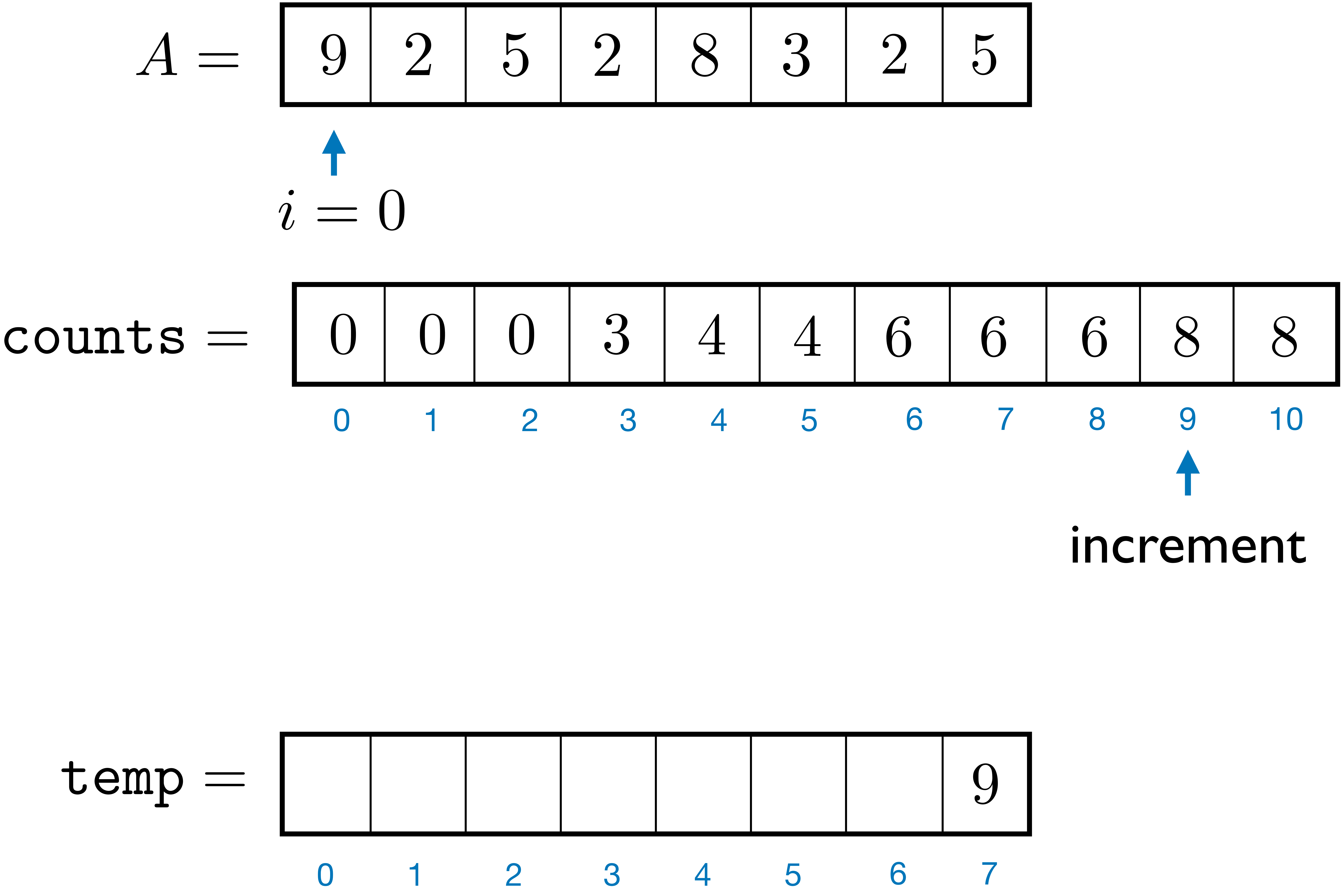
Set $\texttt{temp}[\texttt{counts}[\texttt{A}[\texttt{i}]]] = \texttt{A}[\texttt{i}]$.

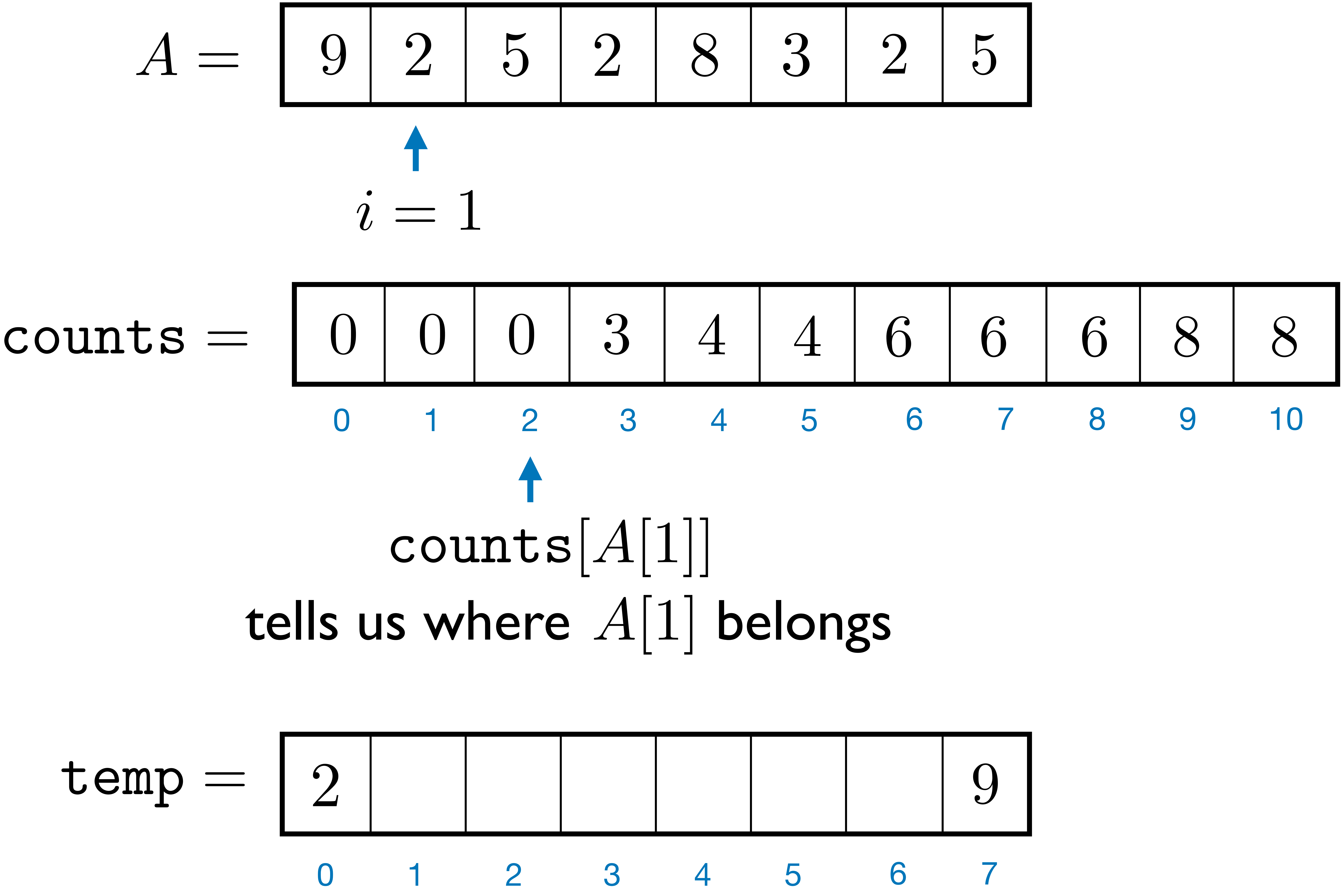
Increment $\texttt{counts}[\texttt{A}[\texttt{i}]]$.

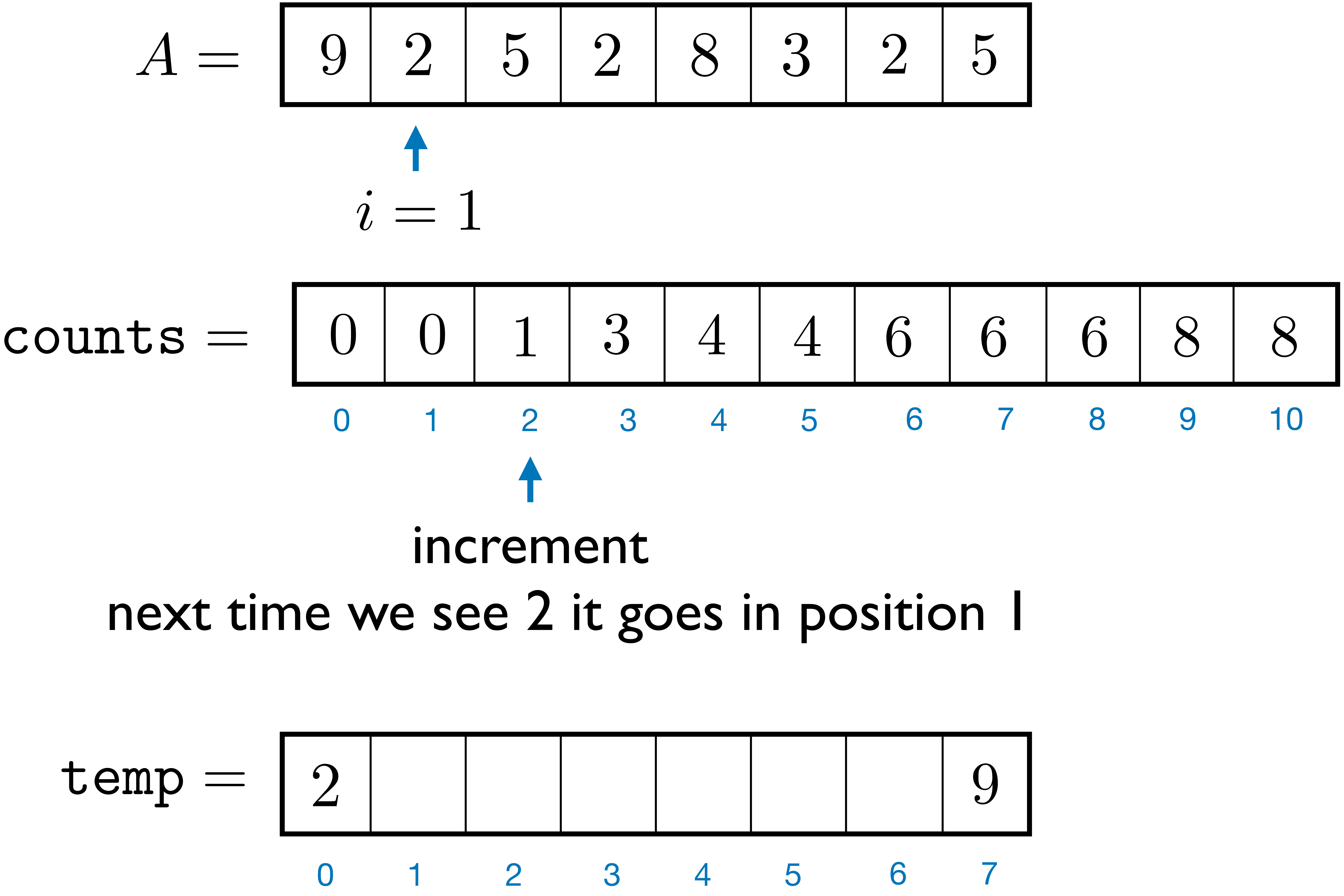


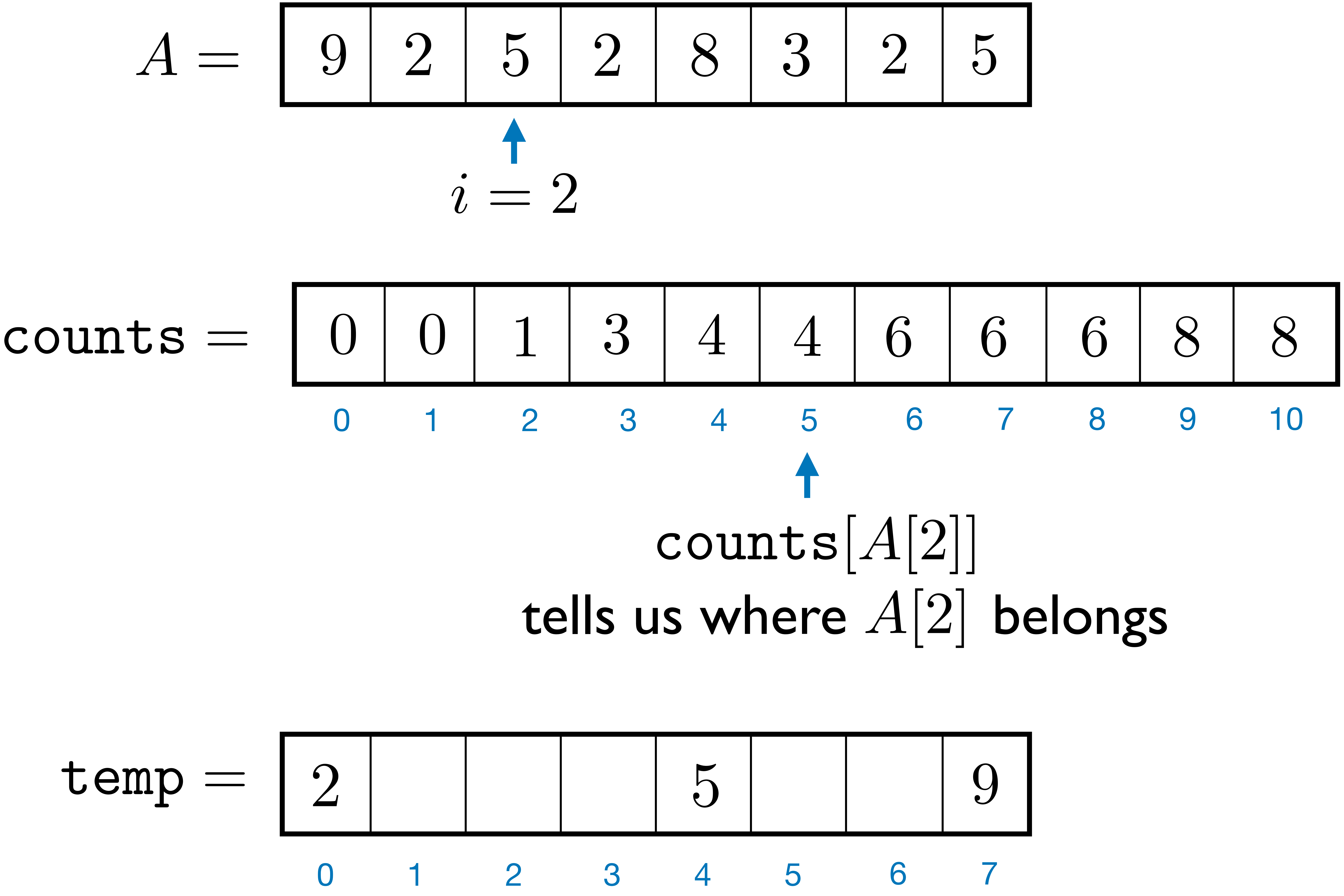
A =
9 2 5 2 8 3 2 5
i = 0
counts =
0 0 0 3 4 4 6 6 6 7 8
0 1 2 3 4 5 6 7 8 9 10
counts[A[0]]
tells us where A[0] belongs
temp =
9
0 1 2 3 4 5 6 7

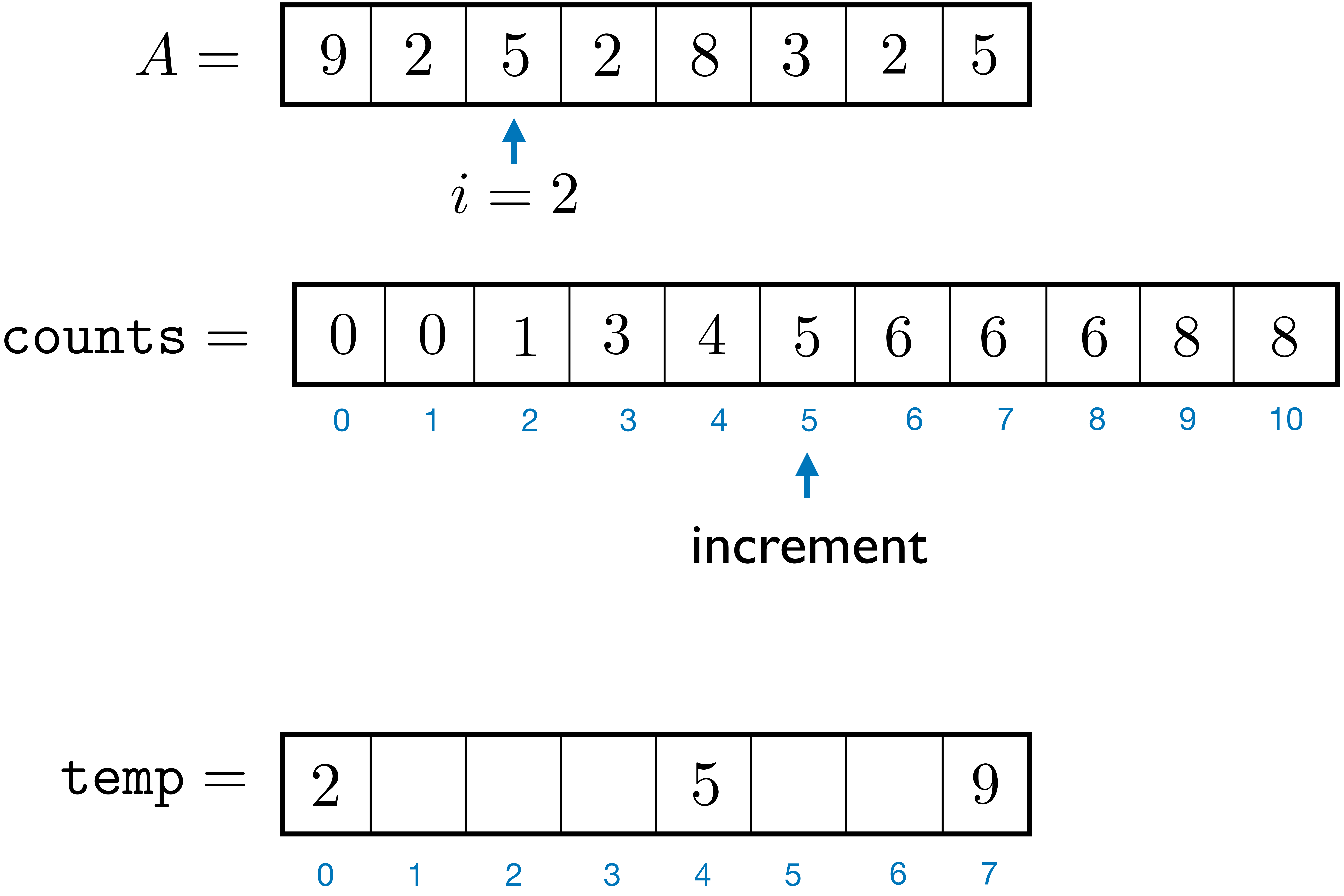

A = 9 2 5 2 8 3 2 5
i = 0
counts = 0 0 0 3 4 4 6 6 6 7 8
0 1 2 3 4 5 6 7 8 9 10
increment
temp = 9
0 1 2 3 4 5 6 7


A = 9 2 5 2 8 3 2 5
i = 0
counts = 0 0 0 3 4 4 6 6 6 8 8
0 1 2 3 4 5 6 7 8 9 10
increment
temp = 9
0 1 2 3 4 5 6 7


A =
9 2 5 2 8 3 2 5
i = 1
counts =
0 0 0 3 4 4 6 6 6 8 8
0 1 2 3 4 5 6 7 8 9 10
counts[A[1]]
tells us where A[1] belongs
temp =
2 9
0 1 2 3 4 5 6 7


A = 9 2 5 2 8 3 2 5
i = 1
counts = 0 0 1 3 4 4 6 6 6 8 8
0 1 2 3 4 5 6 7 8 9 10
increment
next time we see 2 it goes in position 1
temp = 2 9
0 1 2 3 4 5 6 7


$A =$
9 2 5 2 8 3 2 5
$i = 2$
counts $=$
0 0 1 3 4 4 6 6 6 8 8
0 1 2 3 4 5 6 7 8 9 10
counts$[A[2]]$
tells us where $A[2]$ belongs
temp $=$
2 5 9
0 1 2 3 4 5 6 7


A = 9 2 5 2 8 3 2 5
i = 2
counts = 0 0 1 3 4 5 6 6 6 8 8
0 1 2 3 4 5 6 7 8 9 10
increment
temp = 2 5 9
0 1 2 3 4 5 6 7

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | 2 | 5 | 2 | 8 | 3 | 2 | 5 |

↑ $i = 3$

$\texttt{counts} =$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 3 | 4 | 5 | 6 | 6 | 6 | 8 | 8 |

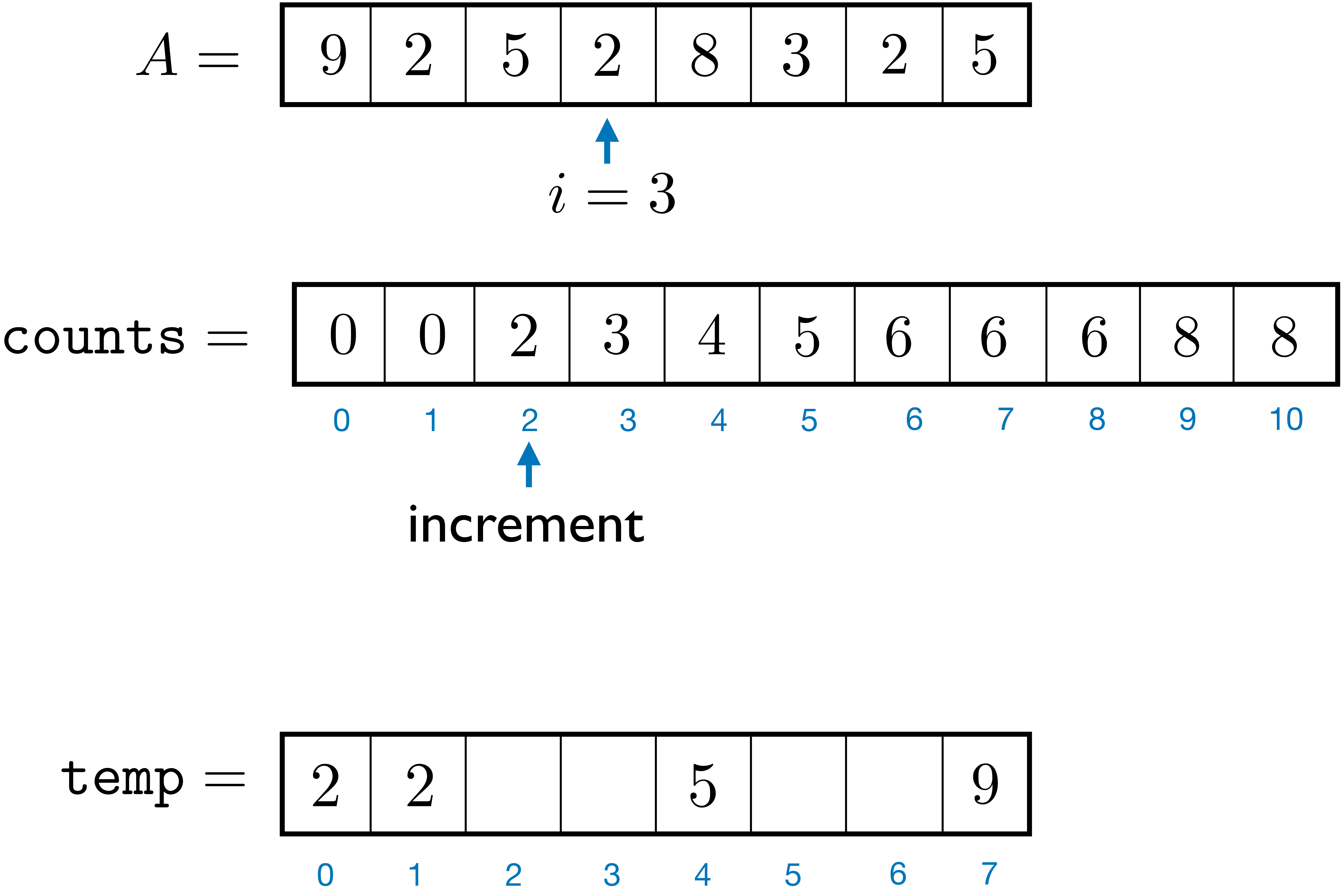
↑ $\texttt{counts}[A[3]]$

tells us where $A[3]$ belongs

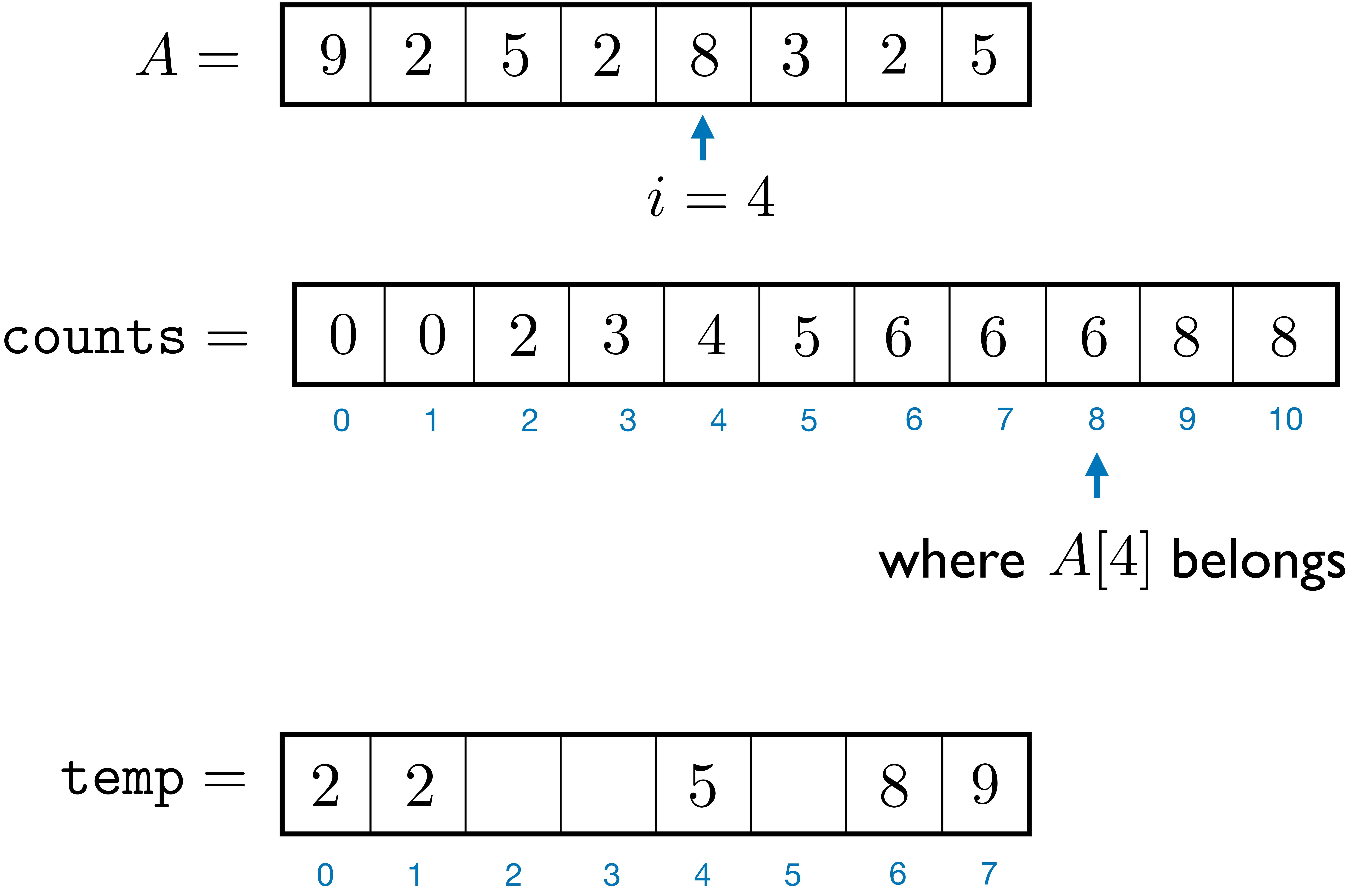
$\texttt{temp} =$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 2 | | | 5 | | | 9 |

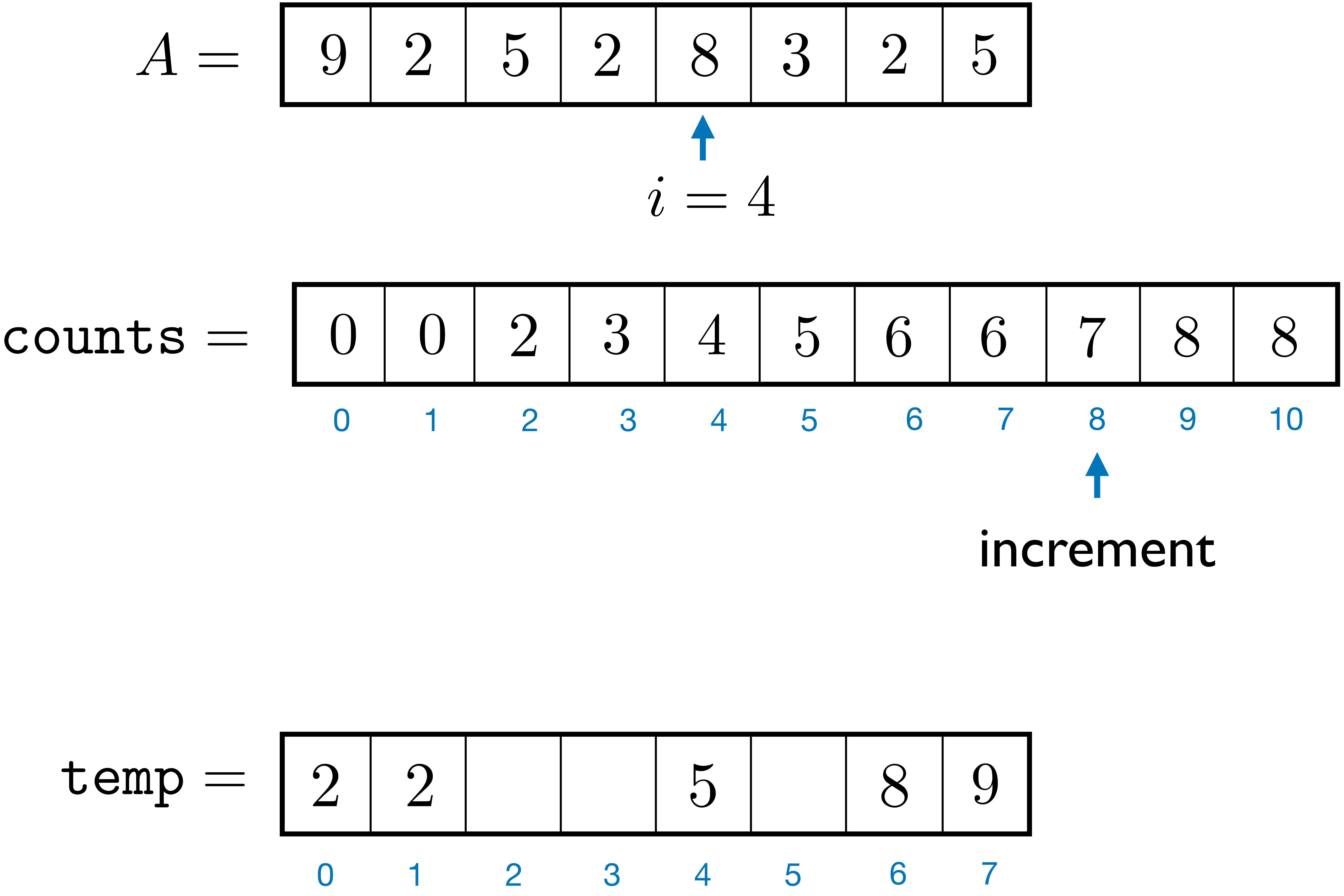
Notice this is a stable sort!

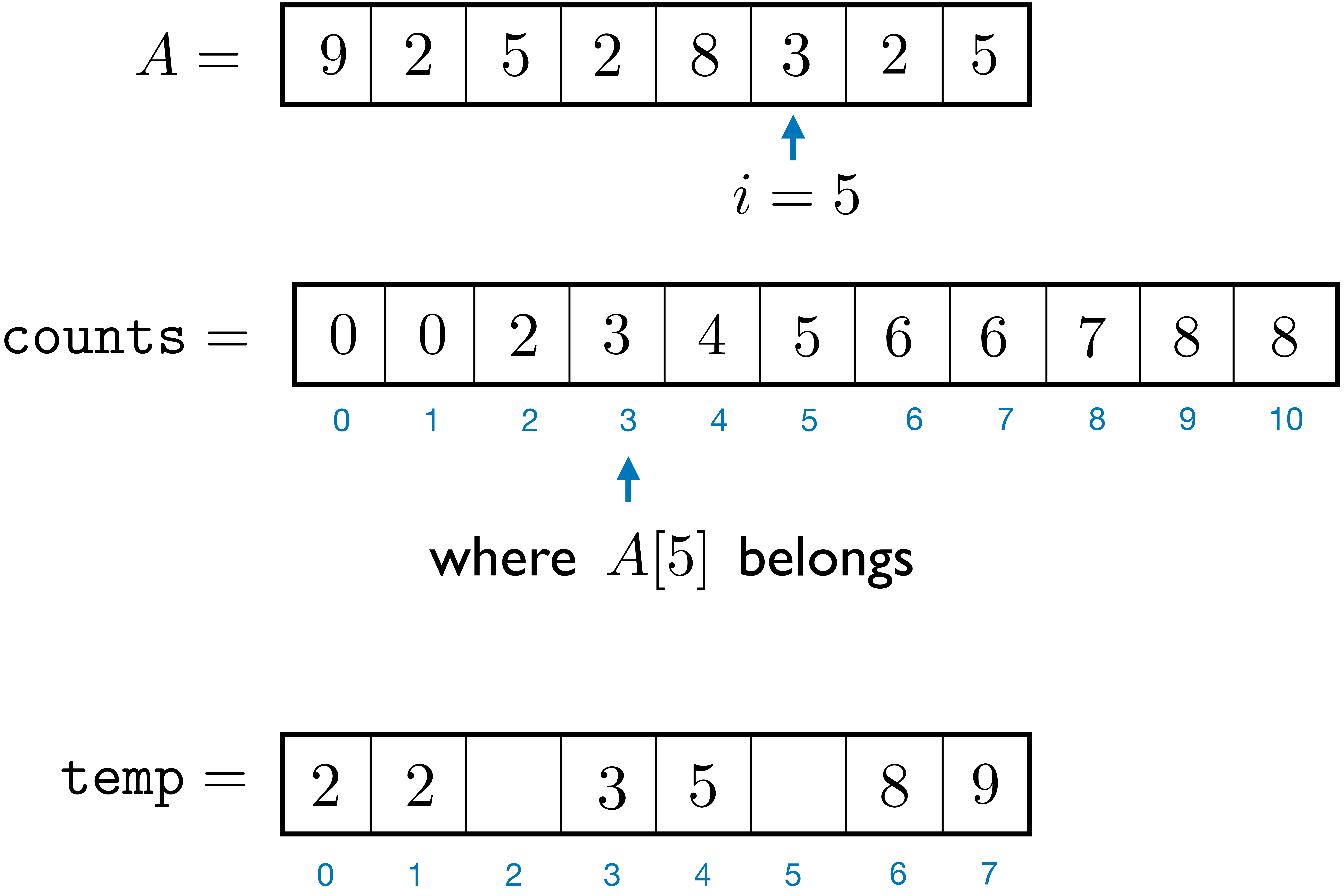


$A =$ 9 2 5 2 8 3 2 5
$i = 3$
counts = 0 0 2 3 4 5 6 6 6 8 8
0 1 2 3 4 5 6 7 8 9 10
increment
temp = 2 2 5 9
0 1 2 3 4 5 6 7

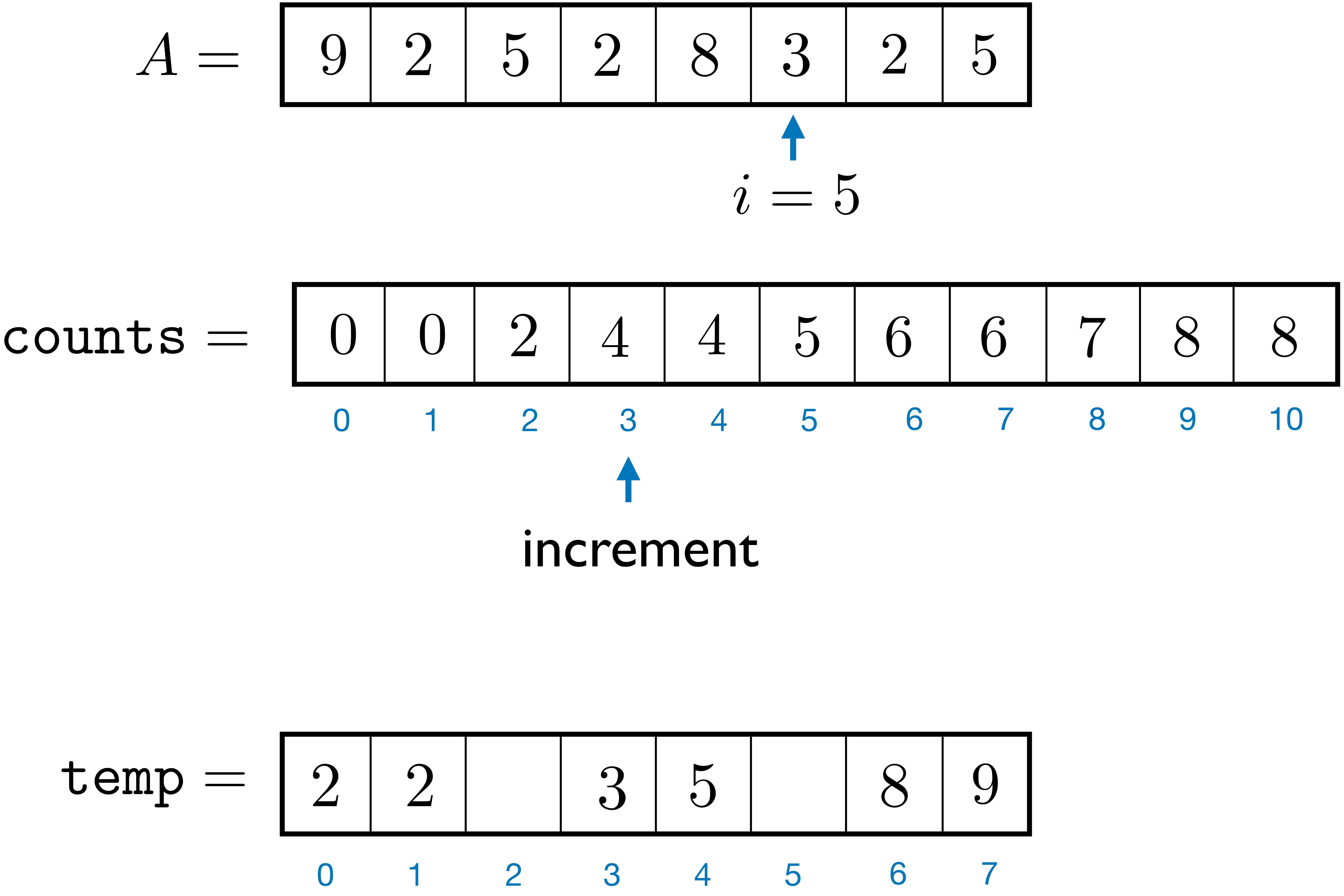


A = 9 2 5 2 8 3 2 5
i = 4
counts = 0 0 2 3 4 5 6 6 6 8 8
0 1 2 3 4 5 6 7 8 9 10
where A[4] belongs
temp = 2 2 5 8 9
0 1 2 3 4 5 6 7

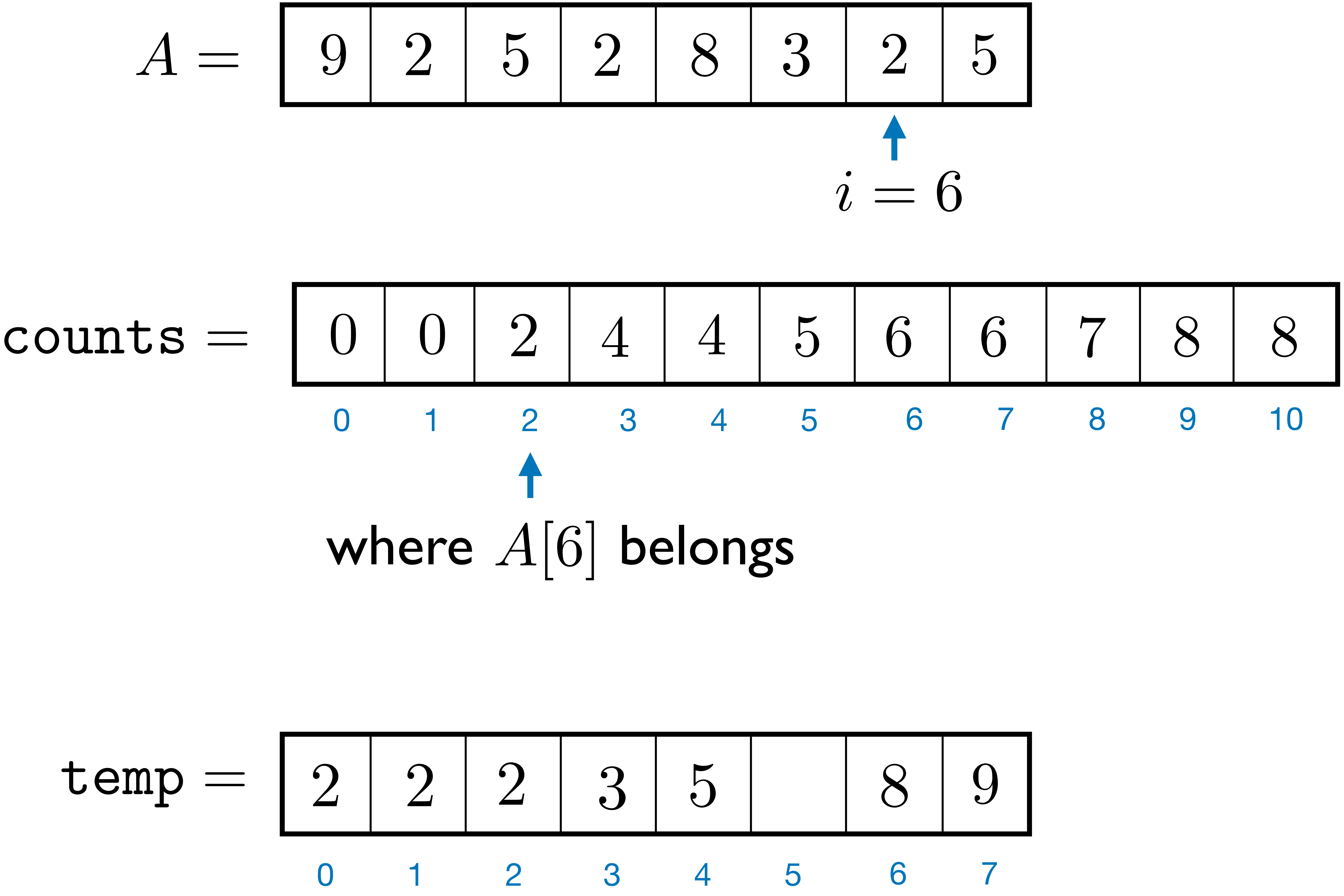

$A = [9, 2, 5, 2, 8, 3, 2, 5]$
$i = 4$
counts $= [0, 0, 2, 3, 4, 5, 6, 6, 7, 8, 8]$
0 1 2 3 4 5 6 7 8 9 10
increment
temp $= [2, 2, \ , \ , 5, \ , 8, 9]$
0 1 2 3 4 5 6 7



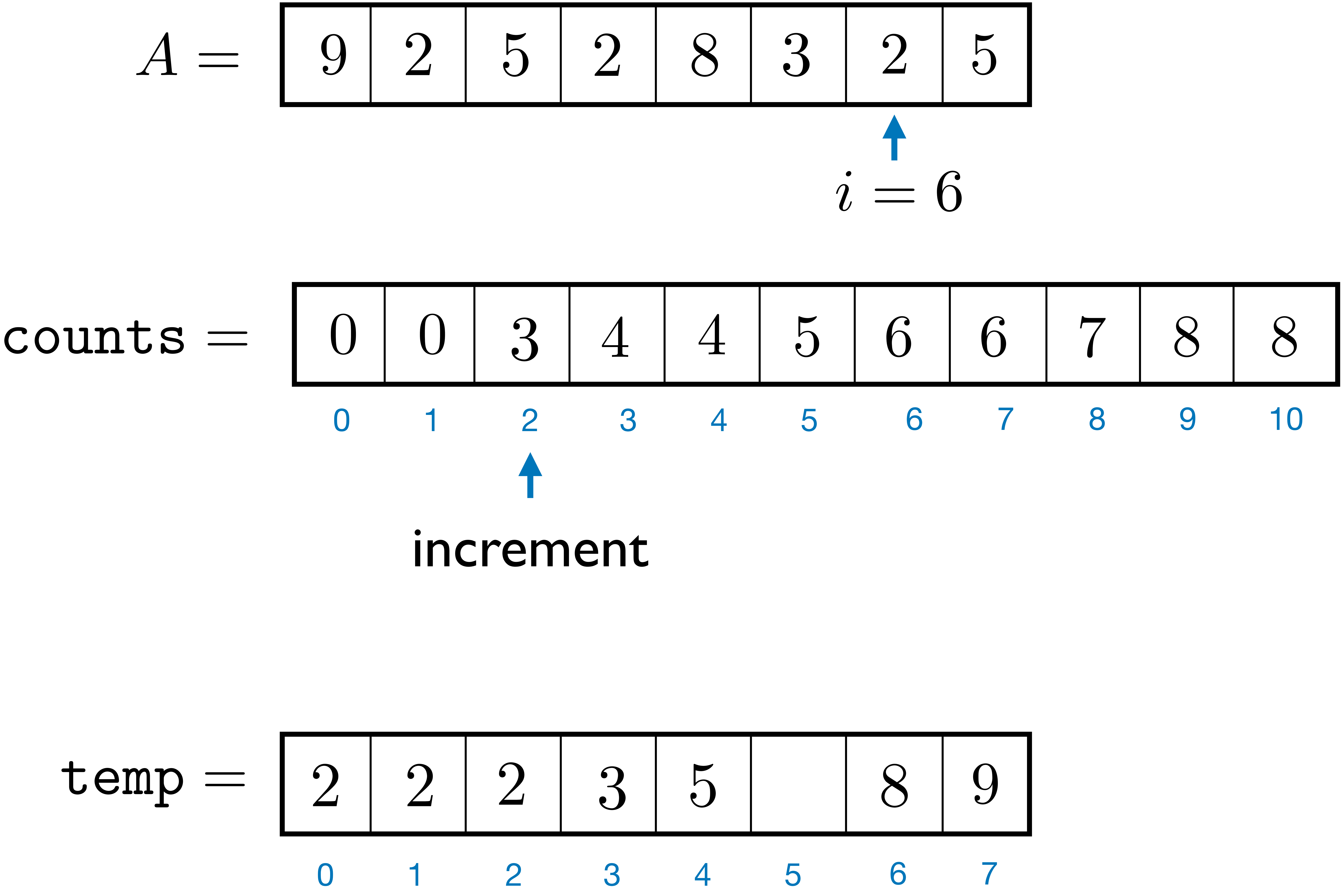
A = 9 2 5 2 8 3 2 5
i = 5
counts = 0 0 2 3 4 5 6 6 7 8 8
0 1 2 3 4 5 6 7 8 9 10
where A[5] belongs
temp = 2 2 3 5 8 9
0 1 2 3 4 5 6 7

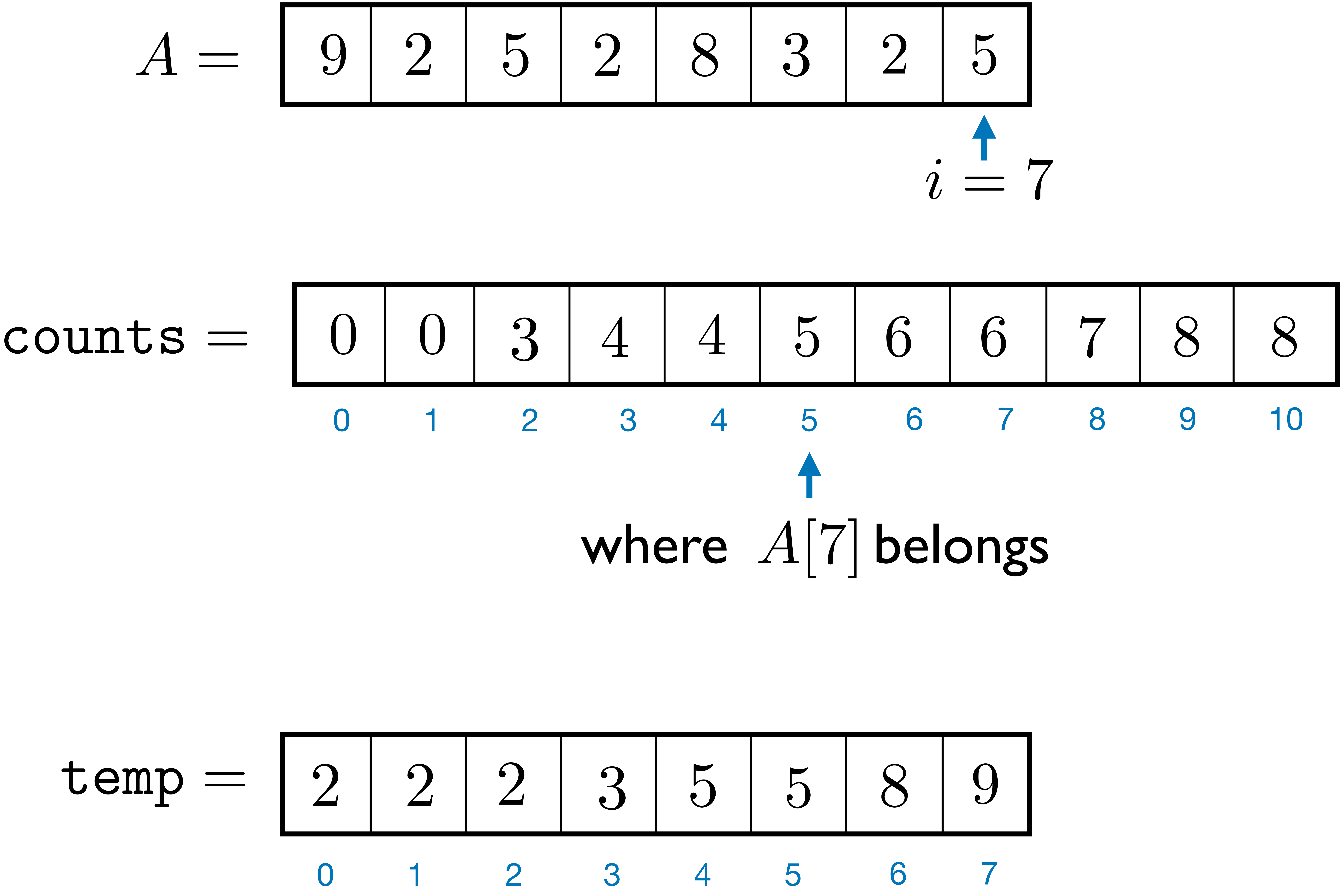


A = 9 2 5 2 8 3 2 5
i = 5
counts = 0 0 2 4 4 5 6 6 7 8 8
0 1 2 3 4 5 6 7 8 9 10
increment
temp = 2 2 3 5 8 9
0 1 2 3 4 5 6 7

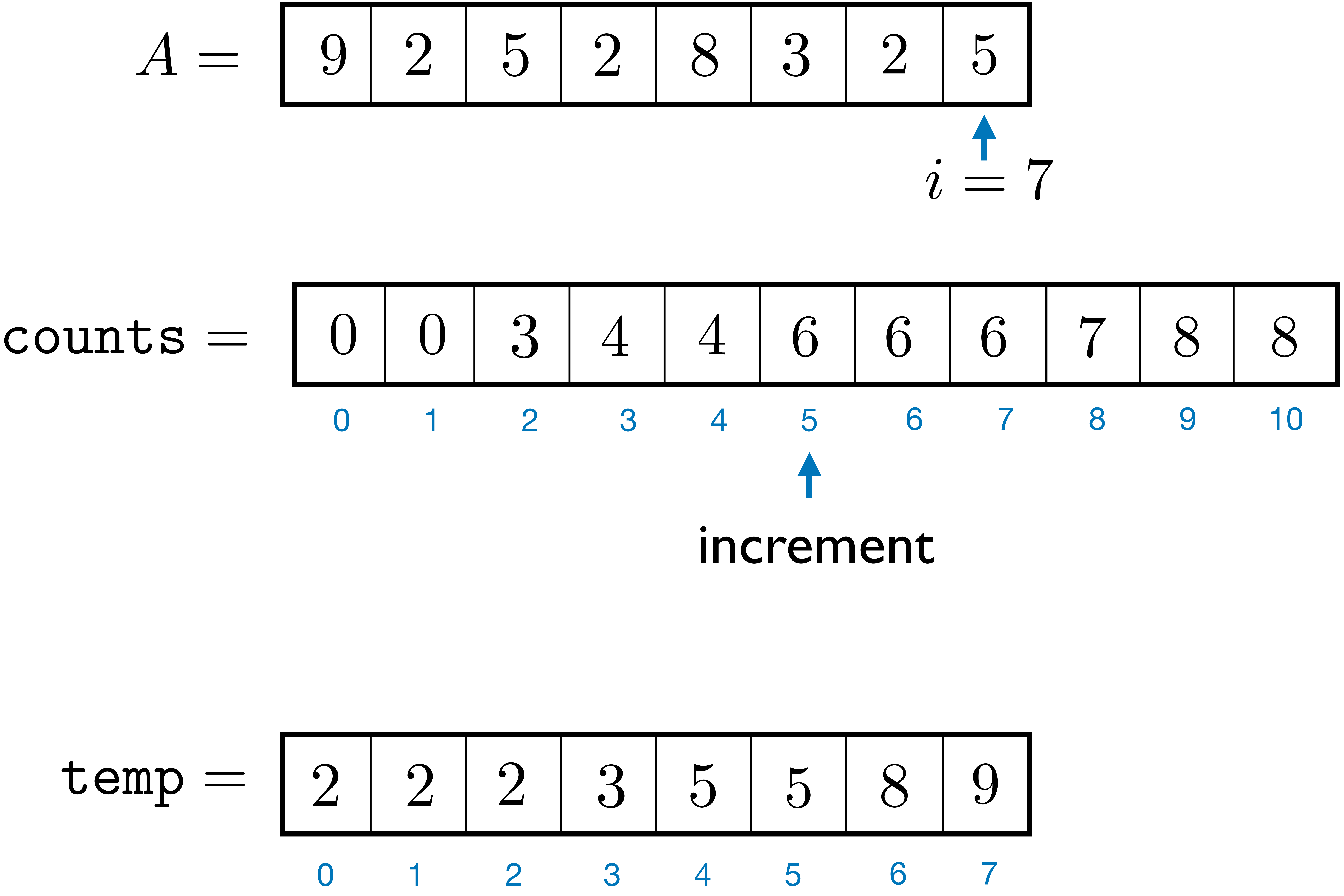


A = 9 2 5 2 8 3 2 5
i = 6
counts = 0 0 2 4 4 5 6 6 7 8 8
0 1 2 3 4 5 6 7 8 9 10
where A[6] belongs
temp = 2 2 2 3 5 8 9
0 1 2 3 4 5 6 7


A = 9 2 5 2 8 3 2 5
i = 6
counts = 0 0 3 4 4 5 6 6 7 8 8
0 1 2 3 4 5 6 7 8 9 10
increment
temp = 2 2 2 3 5 8 9
0 1 2 3 4 5 6 7



A = 9 2 5 2 8 3 2 5
i = 7
counts = 0 0 3 4 4 5 6 6 7 8 8
0 1 2 3 4 5 6 7 8 9 10
where A[7] belongs
temp = 2 2 2 3 5 5 8 9
0 1 2 3 4 5 6 7



A = 9 2 5 2 8 3 2 5
i = 7
counts = 0 0 3 4 4 6 6 6 7 8 8
0 1 2 3 4 5 6 7 8 9 10
increment
temp = 2 2 2 3 5 5 8 9
0 1 2 3 4 5 6 7

# Fourth Step

Copy the sorted array back into $A$.

$$A = \texttt{temp}$$

Find an example of the code here:

https://godbolt.org/z/38vKsMc59

# Complexity

Step 1: Count how many times each element of $A$ appears. Record results in `counts`.

Time: $\Theta(n)$

Step 2: Compute prefix sums of `counts`.

Time: $\Theta(k)$

Step 3: Pass through $A$ and write elements in their sorted place in `temp`.

Time: $\Theta(n)$

Step 4: $A = \texttt{temp}$

Time: $\Theta(n)$

Total: $\Theta(n + k)$

# Space Complexity

A drawback of counting sort is that it is not in place.

We have to use extra space of size $\Theta(n + k)$ for `counts` and `temp`.

# Stable Sort

Counting sort is a stable sort.

In step 3 we pass through $A$ in forward order.

The leftmost element of a given value is placed in `temp` first.

After this element is placed, `counts` is incremented. The next element of the same value will be placed to its right.

# Radix Sort

# Radix Sort

Say that we want to sort an array of 10 digit integers.

To do this with counting sort would require an auxiliary array of size $10^{10}$.

Radix sort uses the fact that the numbers are composed of digits to sort them in several passes (one for each digit) with much less extra memory.

Any stable sort can be used in each pass, but radix sort pairs well with counting sort.

# Radix Sort

**Example:** Let's say we want to sort these 6-digit numbers.

103588

122339

177633

172808

859720

130520

675063

668175

# Radix Sort

Example: Let's say we want to sort these 6-digit numbers.

103588
122339
177633
172808
859720
130520
675063
668175

First we sort them by the least significant digit.

# Radix Sort

Example: Let's say we want to sort these 6-digit numbers.

| | | |
|---|---|---|
| 103588 | | 859720 |
| 122339 | | 130520 |
| 177633 | | 177633 |
| 172808 | → | 675063 |
| 859720 | | 668175 |
| 130520 | | 172808 |
| 675063 | | 103588 |
| 668175 | | 122339 |

First we sort them by the least significant digit.

# Radix Sort

**Example:** Let's say we want to sort these 6-digit numbers.

859720

130520

177633

675063

668175

172808

103588

122339

# Radix Sort

Example: Let's say we want to sort these 6-digit numbers.

859720
130520
177633
675063
668175
172808
103588
122339

Stably sort by the second least significant digit.

# Radix Sort

Example: Let's say we want to sort these 6-digit numbers.

| | | |
|---|---|---|
| 859720 | | 172808 |
| 130520 | | 859720 |
| 177633 | | 130520 |
| 675063 | → | 177633 |
| 668175 | | 122339 |
| 172808 | | 675063 |
| 103588 | | 668175 |
| 122339 | | 103588 |

Stably sort by the second least significant digit.

# Radix Sort

Example: Let's say we want to sort these 6-digit numbers.

| | |
|---|---|
| 859720 | 172808 |
| 130520 | 859720 |
| 177633 | 130520 |
| 675063 | 177633 |
| 668175 | 122339 |
| 172808 | 675063 |
| 103588 | 668175 |
| 122339 | 103588 |

Stably sort by the second least significant digit.

177633 and 122339 agree on the second least significant digit.

177633 comes first because the sort is stable.

# Radix Sort

Example: Let's say we want to sort these 6-digit numbers.

| | | |
|---|---|---|
| 172808 | | 675063 |
| 859720 | | 668175 |
| 130520 | | 122339 |
| 177633 | → | 130520 |
| 122339 | | 103588 |
| 675063 | | 177633 |
| 668175 | | 859720 |
| 103588 | | 172808 |

Stably sort by the third least significant digit.

# Radix Sort

Example: Let's say we want to sort these 6-digit numbers.

| | |
|---|---|
| 172808 | 675063 |
| 859720 | 668175 |
| 130520 | 122339 |
| 177633 | 130520 |
| 122339 | 103588 |
| 675063 | 177633 |
| 668175 | 859720 |
| 103588 | 172808 |

Stably sort by the third least significant digit.

130520 and 103588 agree on the third least significant digit.

130520 comes first because the sort is stable.

# Radix Sort

Example: Let's say we want to sort these 6-digit numbers.

| | | |
|---|---|---|
| 675063 | | 130520 |
| 668175 | | 122339 |
| 122339 | | 172808 |
| 130520 | → | 103588 |
| 103588 | | 675063 |
| 177633 | | 177633 |
| 859720 | | 668175 |
| 172808 | | 859720 |

Stably sort by the fourth least significant digit.

For any numbers that agree on the fourth digit, the one whose first three digits are smaller comes first.

# Radix Sort

Example: Let's say we want to sort these 6-digit numbers.

| | | |
|---|---|---|
| 130520 | | 103588 |
| 122339 | | 122339 |
| 172808 | | 130520 |
| 103588 | | 859720 |
| 675063 | → | 668175 |
| 177633 | | 172808 |
| 668175 | | 675063 |
| 859720 | | 177633 |

Stably sort by the fifth least significant digit.

# Radix Sort

Example: Let's say we want to sort these 6-digit numbers.

| | | |
|---|---|---|
| 103588 | | 103588 |
| 122339 | | 122339 |
| 130520 | | 130520 |
| 859720 | | 172808 |
| 668175 | → | 177633 |
| 172808 | | 668175 |
| 675063 | | 675063 |
| 177633 | | 859720 |

Stably sort by the most significant digit.

# Why it works

Say we have two numbers $a < b$ . Let's see why $a$ comes before $b$ after radix sorting.

Say that $a = a_1a_2a_3a_4a_5$

$$b = a_1a_2b_3b_4b_5$$

They agree on the first two digits and first disagree on the third digit.

Since $a_3 < b_3$ after sorting on the third digit $a$ will be placed before $b$ .

# Why it works

Say that $a = a_1 a_2 a_3 a_4 a_5$

$b = a_1 a_2 b_3 b_4 b_5$

Since $a_3 < b_3$ after sorting on the third digit $a$ will be placed before $b$ .

When we sort on the second and first digit, $a$ and $b$ will compare equal.

But because we use a stable sort $a$ will be placed before $b$.

# Complexity

Say we have an array of size $n$ with $d$-digit numbers where each digit is between $0$ and $k-1$.

If we use counting sort for the stable sort then each sort of a digit takes time $\Theta(n+k)$.

The total time is $\Theta(d(n+k))$.

The space used by the algorithm is $\Theta(n+k)$.