Searching and Sorting

Charlie Li {cmli}

2021-02-27

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Content

Searching

- Linear Search
- Binary Search
- Ternary Search

Sorting

- Selection Sort, Insertion Sort, Bubble Sort
- Quick Sort, Merge Sort
- Counting Sort, Radix Sort

Searching

Searching

Three questions:

- What to search?
- Where to search?
- How to search?

What to search?

Usually, we will search for

- a root / solution / position / existence or
- an optimal point / value

Where to search?

Usually, we will search on

- a function or
- an array

What and where to search? Examples

On a function:

- Find a root of $x^2 + 2x + 1$ (root)
- Find the minimum point of $x^2 + 2x + 1$ (optimal point)

On an array:

- Find the position of 7 in the array (position)
  2 3 5 7 11
- Find an element in the following array which is closest to 7 (optimal value)
  1 3 6 10 15

How to Search? Linear Search

Linear Search

Aka

- Sequential Search
- Dictionary Search

Algorithm: Check the candidates one by one

This is one of the naive ways to locate an element in an array or to find the answer of a function having a finite domain

Linear Search

Example: Find any 3 in the following array

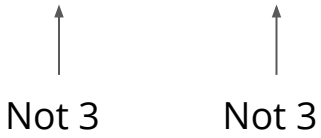
Index	0	1	2	3	4	5
Value	1	5	1	3	4	2

Not 3

Linear Search

Example: Find any 3 in the following array

Index	0	1	2	3	4	5
Value	1	5	1	3	4	2



Linear Search

Example: Find any 3 in the following array

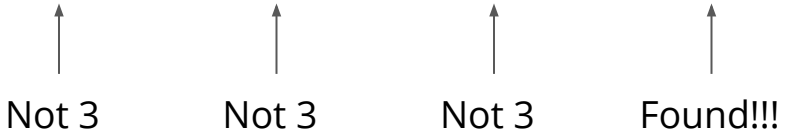
Index	0	1	2	3	4	5
Value	1	5	1	3	4	2

Not 3 Not 3 Not 3

Linear Search

Example: Find any 3 in the following array

Index	0	1	2	3	4	5
Value	1	5	1	3	4	2



Linear Search

Example: Find any 0 in the following array

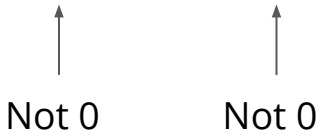
Index	0	1	2	3	4	5
Value	1	5	1	3	4	2

Not 0

Linear Search

Example: Find any 0 in the following array

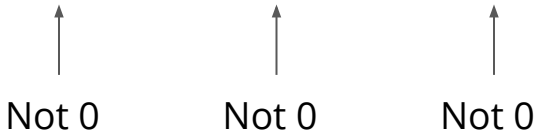
Index	0	1	2	3	4	5
Value	1	5	1	3	4	2



Linear Search

Example: Find any 0 in the following array

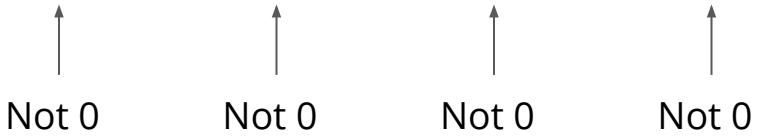
Index	0	1	2	3	4	5
Value	1	5	1	3	4	2



Linear Search

Example: Find any 0 in the following array

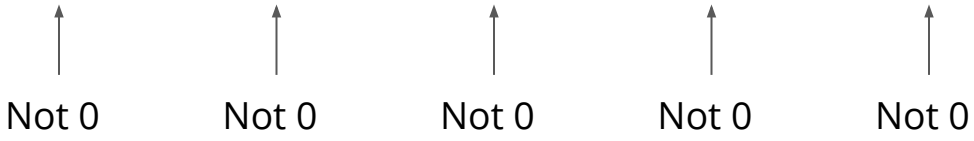
Index	0	1	2	3	4	5
Value	1	5	1	3	4	2



Linear Search

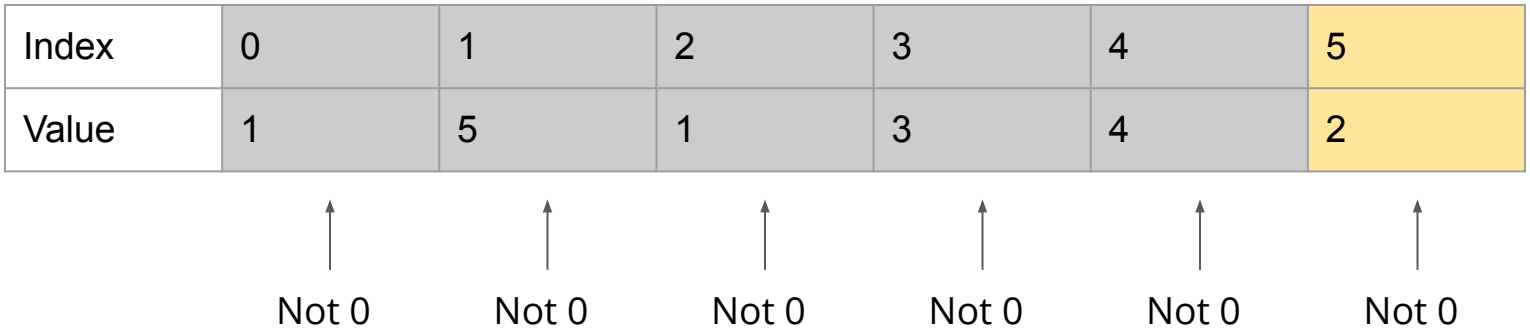
Example: Find any 0 in the following array

Index	0	1	2	3	4	5
Value	1	5	1	3	4	2



Linear Search

Example: Find any 0 in the following array



Linear Search

Example: Find any 0 in the following array

Index	0	1	2	3	4	5
Value	1	5	1	3	4	2

Not 0 Not 0 Not 0 Not 0 Not 0 Not 0

Error 404
Not Found

Linear Search

This linear search can be easily implemented using a for loop

Worst case: O(n) checking

Average: O(n) checking

Checking: checks whether the element satisfy the condition

```
int find(int a[], int n, int k) {
  for (int i = 0; i < n; i++) {
    if (a[i] == k) {
      return i;
    }
  }
  return -1; // Not found
}
```

Linear Search

A different one: Find the largest element that is less than 7 in the array.

Index	0	1	2	3	4	5
Value	3	7	4	9	5	11

Linear Search

A different one: Find the largest element that is less than 7 in the array.

Index	0	1	2	3	4	5
Value	3	7	4	9	5	11

Current best:
Not found

Linear Search

A different one: Find the largest element that is less than 7 in the array.

Index	0	1	2	3	4	5
Value	3	7	4	9	5	11

Current best:
3

Linear Search

A different one: Find the largest element that is less than 7 in the array.

Index	0	1	2	3	4	5
Value	3	7	4	9	5	11

Current best:
3

Linear Search

A different one: Find the largest element that is less than 7 in the array.

Index	0	1	2	3	4	5
Value	3	7	4	9	5	11

Current best:
4

Linear Search

A different one: Find the largest element that is less than 7 in the array.

Index	0	1	2	3	4	5
Value	3	7	4	9	5	11

Current best:
4

Linear Search

A different one: Find the largest element that is less than 7 in the array.

Index	0	1	2	3	4	5
Value	3	7	4	9	5	11

Current best:
5

Linear Search

A different one: Find the largest element that is less than 7 in the array.

Index	0	1	2	3	4	5
Value	3	7	4	9	5	11

Current best: 5

Linear Search

A different one: Find the largest element that is less than 7 in the array.

Index	0	1	2	3	4	5
Value	3	7	4	9	5	11

Answer: 5

Linear Search

This linear search can also be easily implemented using a for loop

Worst case: O(n) checking

Average: O(n) checking

Checking: checks whether the element is better that current best

```
int find_best(int a[], int n) {
  int best = -1; // will return -1 when not found
  for (int i = 0; i < n; i++) {
    if (a[i] < 7 && a[i] > best) {
      best = a[i];
    }
  }
  return best;
}
```

Linear Search

Pros:

- Linear search can always solve the problem if we know how to do the checking
- The array need not to be sorted

Cons:

- Linear search is slow, especially when you need to perform the search many times

Linear Search

Practice: 01023

How to Search? Binary Search

Binary Search

Aka

- Half-interval Search
- Logarithm Search
- Binary Chop

Algorithm: Check the candidate at the middle and than discard one half

One of the most important searching algorithms

Binary Search

Example: Find the largest number less than or equal to 7 in an array

Requirement: The array must be sorted

Binary Search

Requirement: The array must be sorted

Algorithm: Check the element at the middle, all the element on the left are first-half, all other elements are second-half

- If it is smaller than or equal to 7, discard the first half
- Otherwise (greater than 7), discard the second half

Until there is only one element left, check if it can be the answer.

If yes, then it is the answer.

If no, then there is no answer.

Binary Search

Example: Find the largest number less than or equal to 7

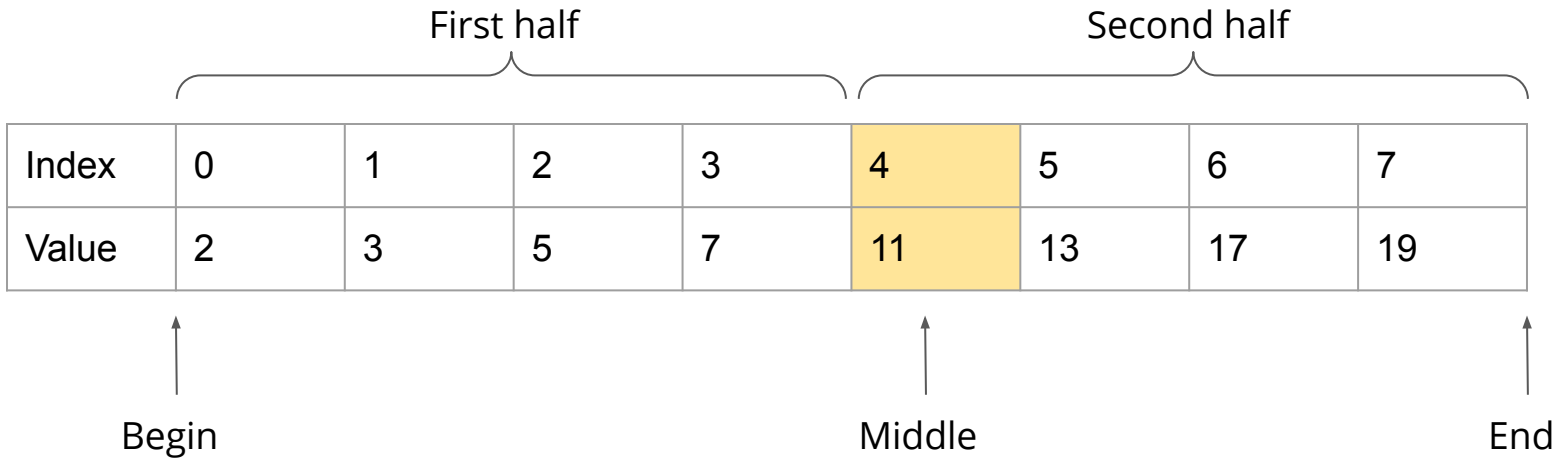
Index	0	1	2	3	4	5	6	7
Value	2	3	5	7	11	13	17	19

Begin

End

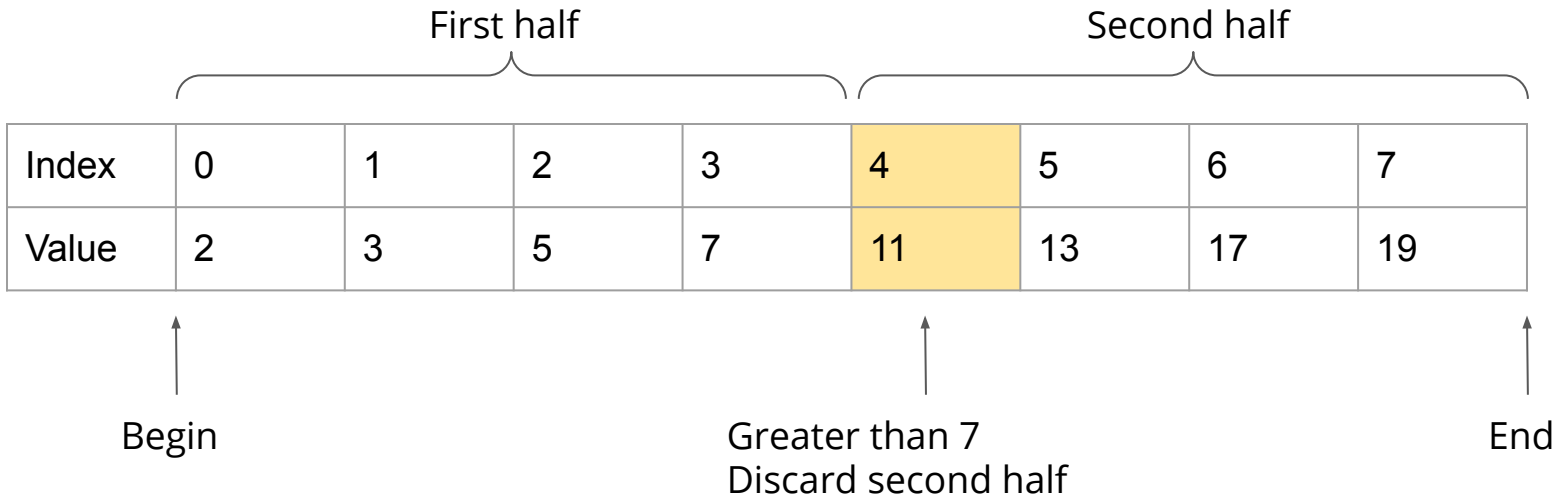
Binary Search

Example: Find the largest number less than or equal to 7



Binary Search

Example: Find the largest number less than or equal to 7



Binary Search

Example: Find the largest number less than or equal to 7

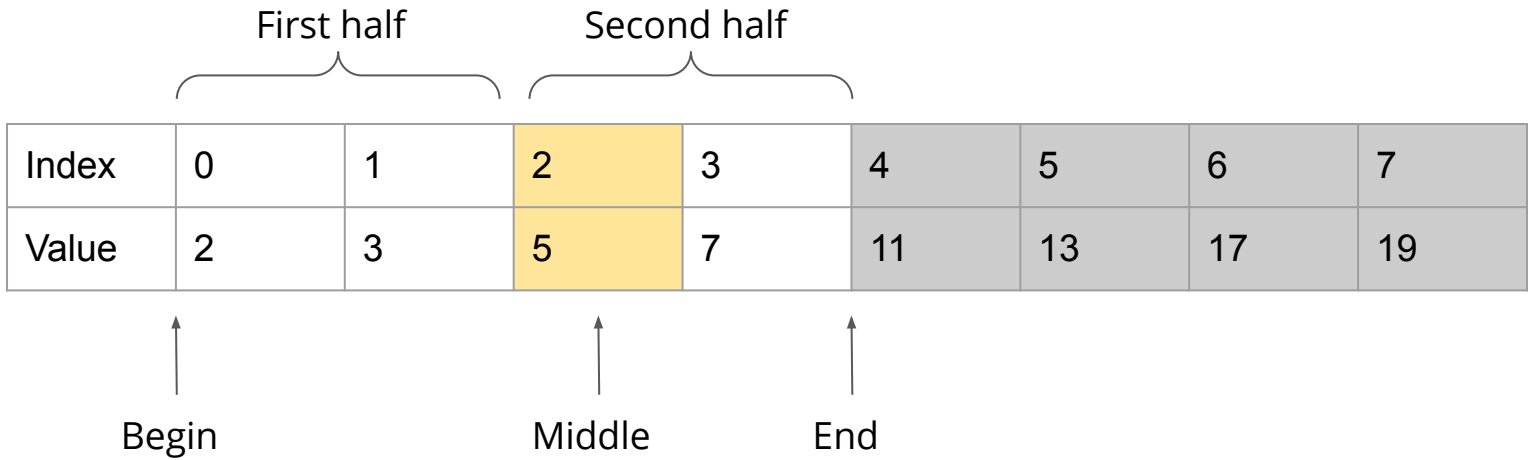
Index	0	1	2	3	4	5	6	7
Value	2	3	5	7	11	13	17	19

Begin

End

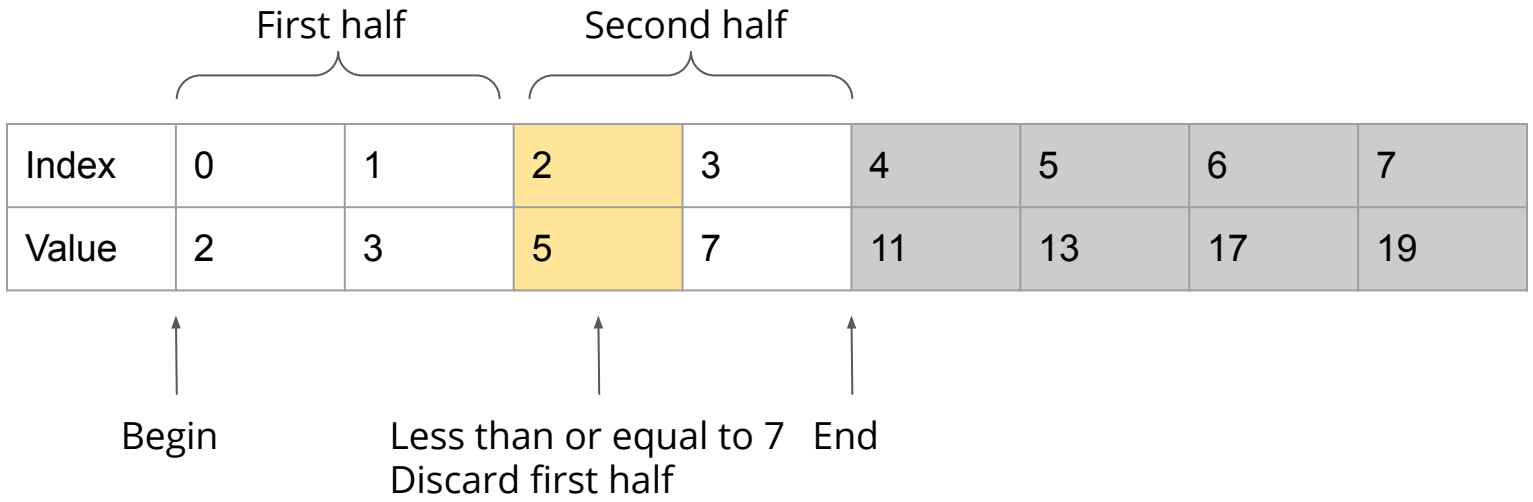
Binary Search

Example: Find the largest number less than or equal to 7



Binary Search

Example: Find the largest number less than or equal to 7



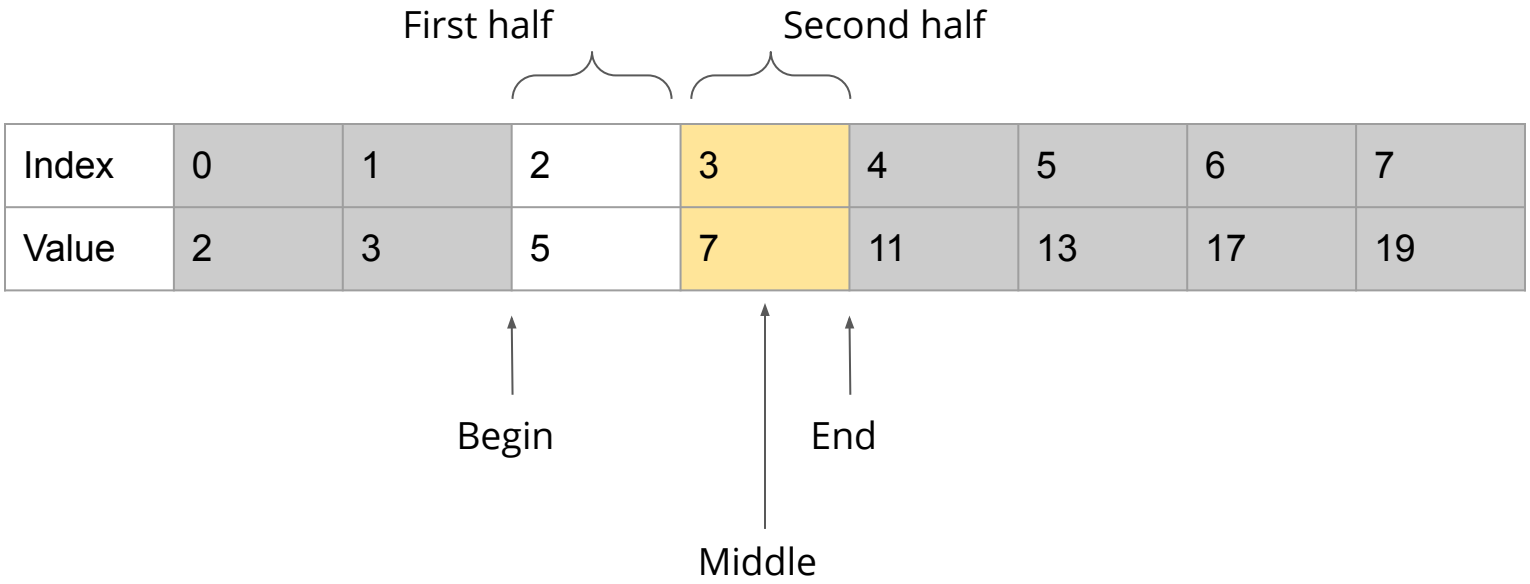
Binary Search

Example: Find the largest number less than or equal to 7

Index	0	1	2	3	4	5	6	7
Value	2	3	5	7	11	13	17	19

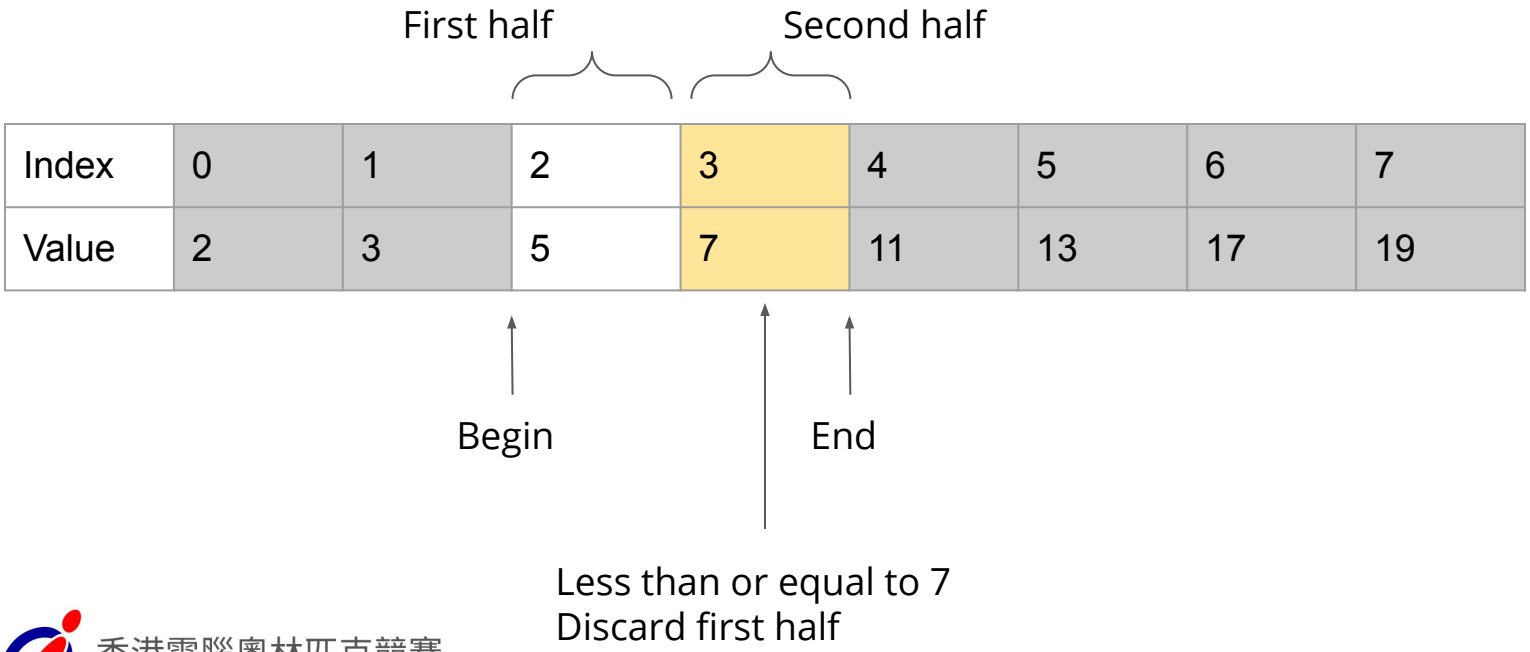
Binary Search

Example: Find the largest number less than or equal to 7



Binary Search

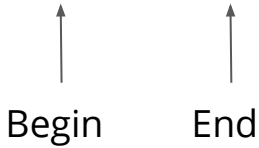
Example: Find the largest number less than or equal to 7



Binary Search

Example: Find the largest number less than or equal to 7

Index	0	1	2	3	4	5	6	7
Value	2	3	5	7	11	13	17	19



Binary Search

Example: Find the largest number less than or equal to 7

Index	0	1	2	3	4	5	6	7
Value	2	3	5	7	11	13	17	19

Less than or equal to 7
This must be the answer

Binary Search

Example: Find the largest number less than or equal to 1

Index	0	1	2	3	4	5	6	7
Value	2	3	5	7	11	13	17	19

Begin

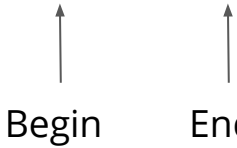
End

A few moments later...

Binary Search

Example: Find the largest number less than or equal to 1

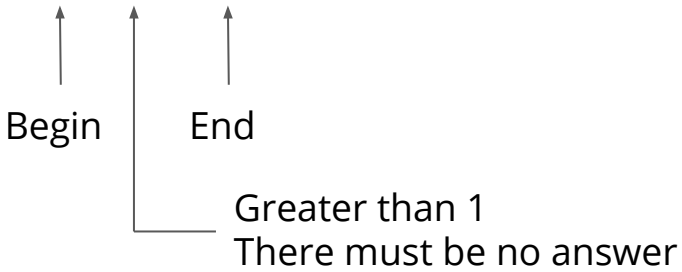
Index	0	1	2	3	4	5	6	7
Value	2	3	5	7	11	13	17	19



Binary Search

Example: Find the largest number less than or equal to 1

Index	0	1	2	3	4	5	6	7
Value	2	3	5	7	11	13	17	19



Binary Search

Example: Find the largest number less than or equal to 1

Index	0	1	2	3	4	5	6	7
Value	2	3	5	7	11	13	17	19

Begin

End

Greater than 7
There must be no answer

Many students' algorithm are wrong just because that they have missed this corner case!

After k checkings, only 1/(2^k) of the original array left. n/(2^k) = 1

Binary Search

T(n) = T(n/2) + 1 => T(n) = O(lg n)

The binary search are usually being implemented using a while loop.

Worse case: O(lg n) checking

Average: O(lg n) checking

Checking: checks whether the middle element may be the answer

```
int find(int a[], int n, int k) {
  // largest number less than or equal to k
  int lo = 0, hi = n; // include lo, exclude hi
  while (lo + 1 < hi) {
    int mi = (lo + hi) / 2;
    if (a[mi] <= k) {
      lo = mi;
    } else {
      hi = mi;
    }
  }
  return a[lo] <= k ? lo : -1; // -1 for no answer
}
```

Binary Search

Checklist for testing the algorithm:

- sample test cases
- will there be infinite loop
- corner cases (no answer, answer is the first, answer is the last...)
- long long

...

Binary Search

Actually, there are in total 4 variations of binary search

- Find the largest element less than or equal to k ← The examples
- Find the largest element less than k
- Find the smallest element greater than or equal to k
- Find the smallest element greater than k

Binary Search

For C++ users, we have a good news for you.

The STL has already implemented two of those binary search for you.

In the algorithm library, there are

- lower_bound: find the first (smallest) element greater than or equal to k
- upper_bound: find the first (smallest) element greater than k

Requirement: The array must be sorted

Binary Search - Binary Search on Answer

Recall that only the requirement for performing binary search is that **the array must be sorted**

From the previous examples, there are only some elements in the array that we need to access. Actually, we don't even need to know what's every element in the array, it is only required that **the array must be sorted**

So, we may convert some functions into a virtual array and then perform binary search on it.

Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$

If you are good in arithmetic, you may instant find that k = floor(sqrt(50)) = 7

But we may assume that there is an array a where $a[i] = i^2$ and perform binary search on this array.

Binary Search - Binary Search on Answer

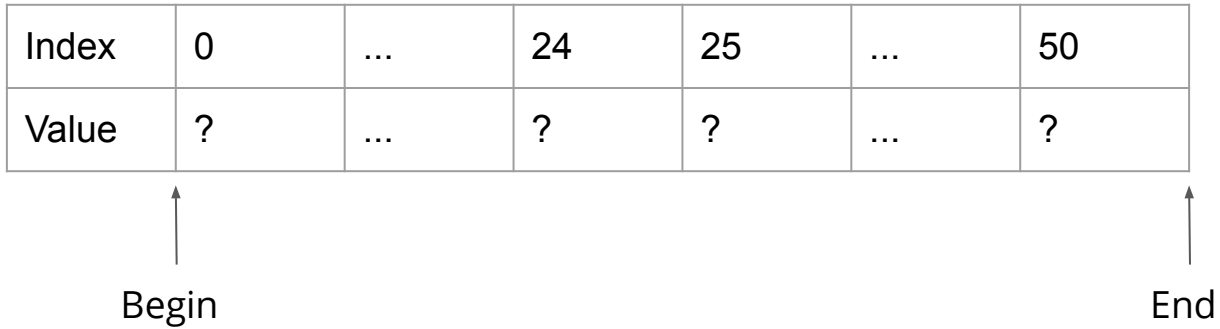
However, we do not have the size of the array, we need to first determine the boundary of the array.

Make sure that the answer lies in our boundary.

For this problem, we know that the answer must be smaller than 50 and larger than 0. So we may set our range to be 0 to 50.

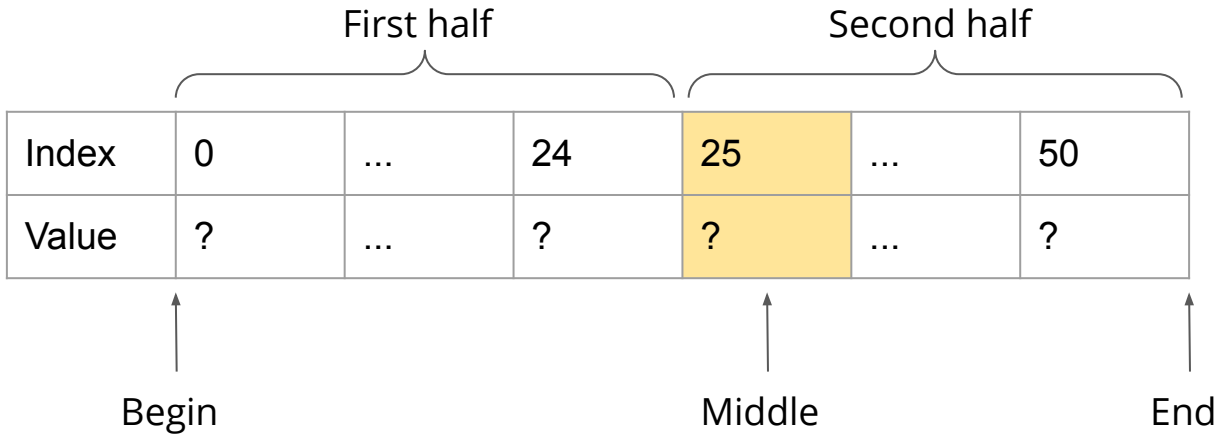
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



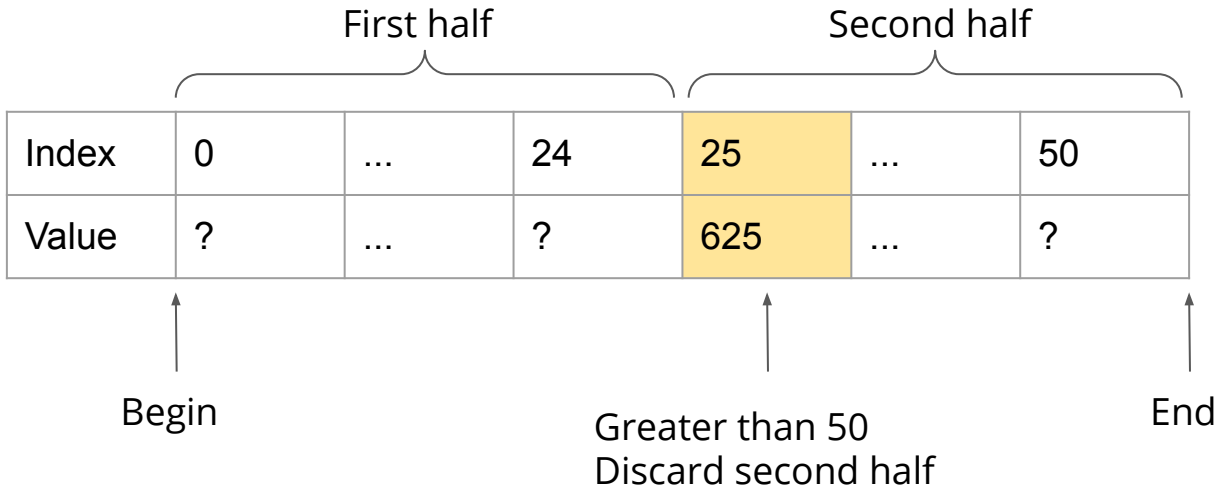
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



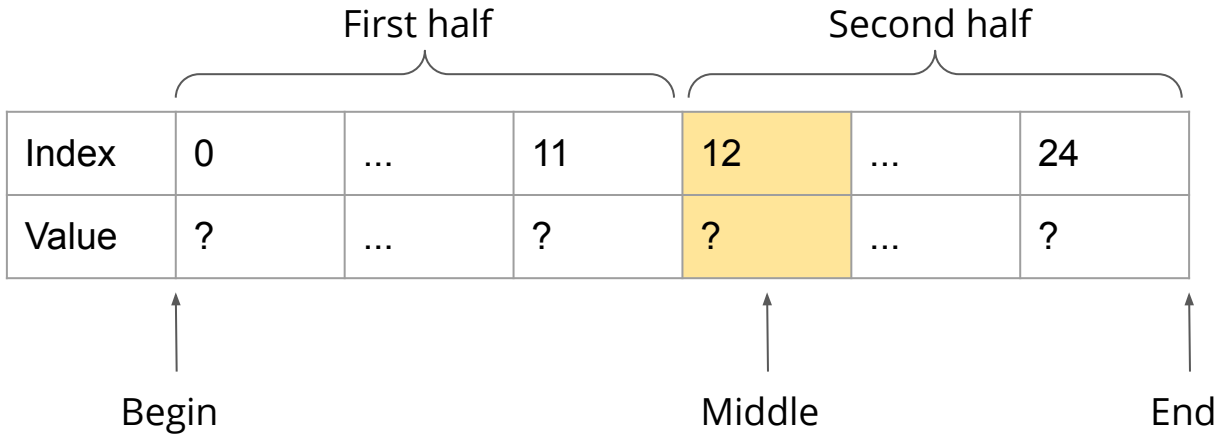
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



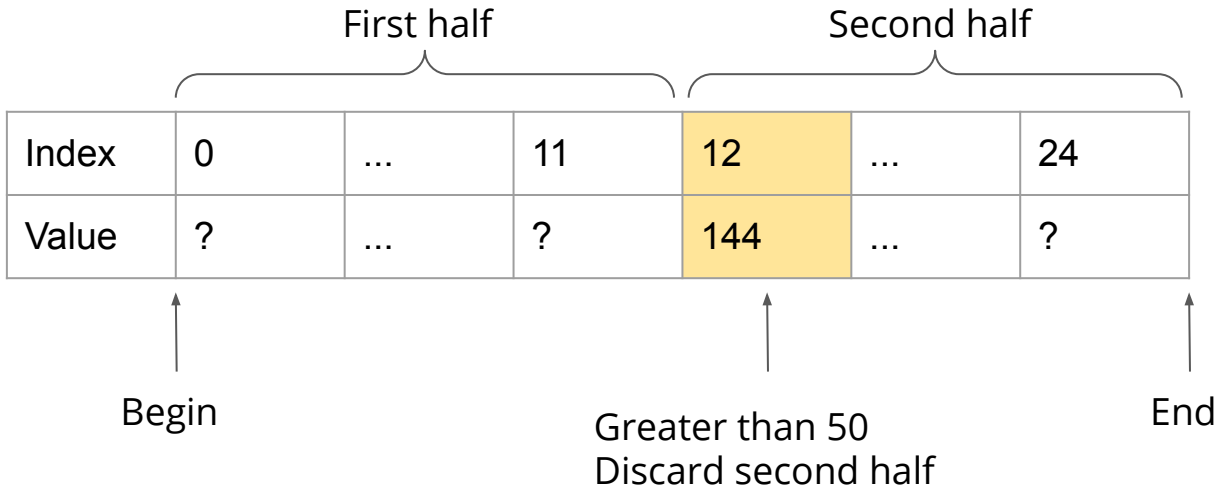
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



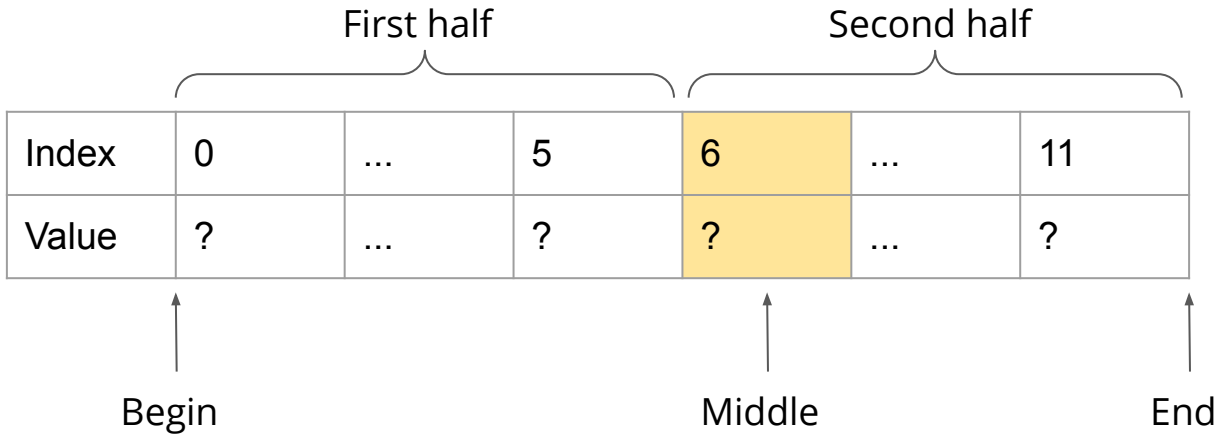
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



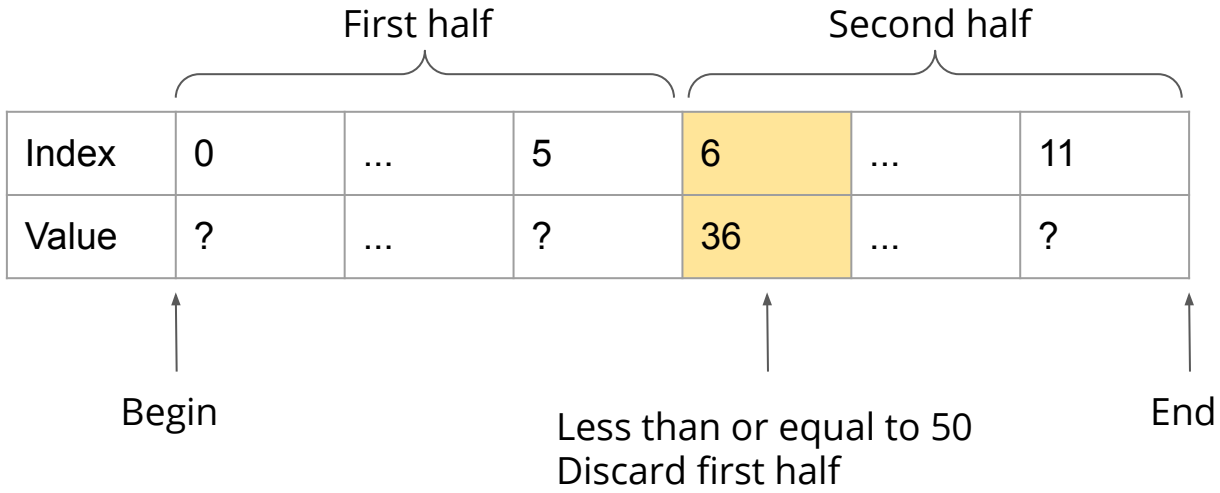
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



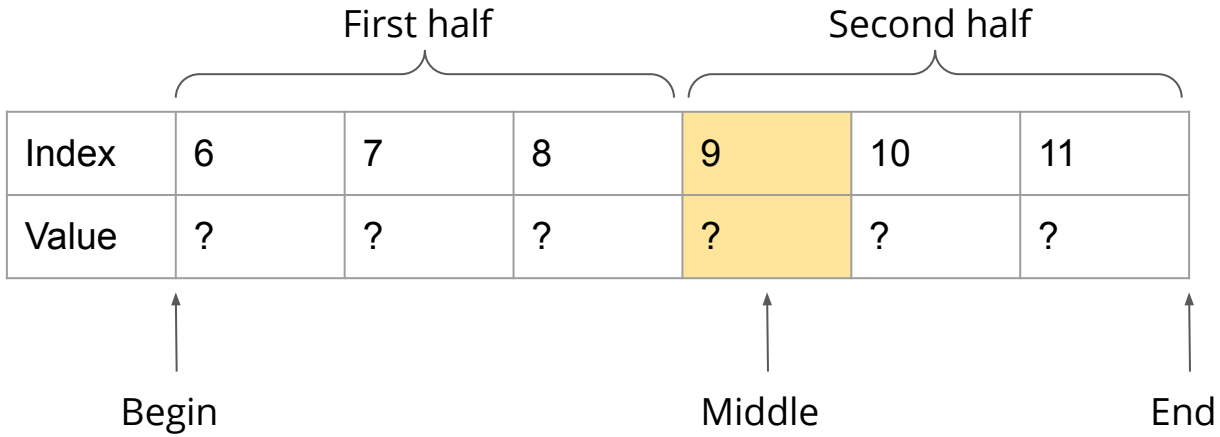
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



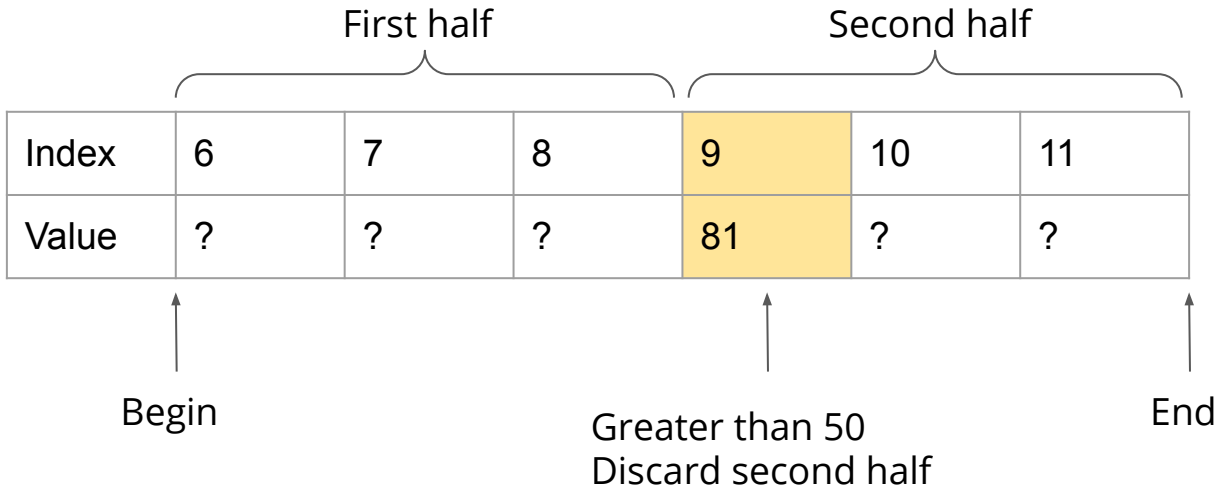
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



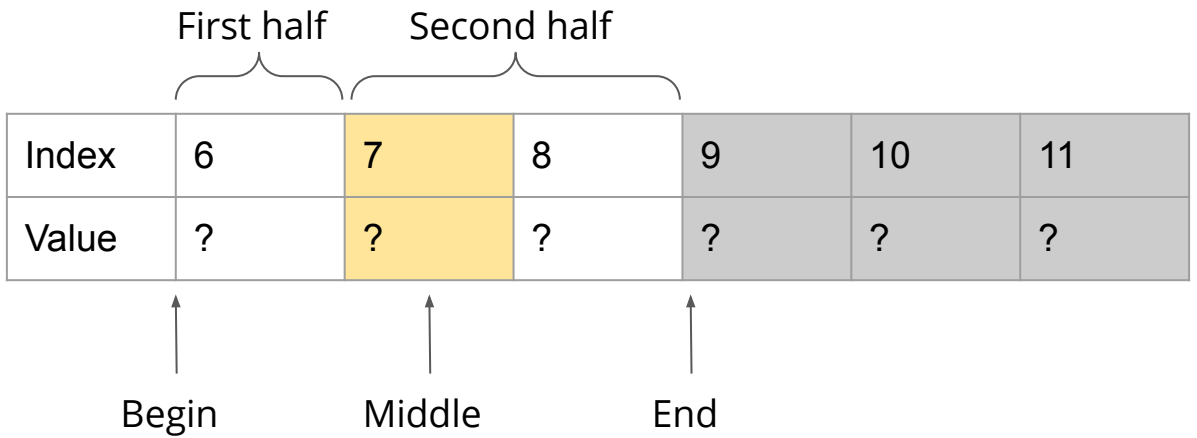
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



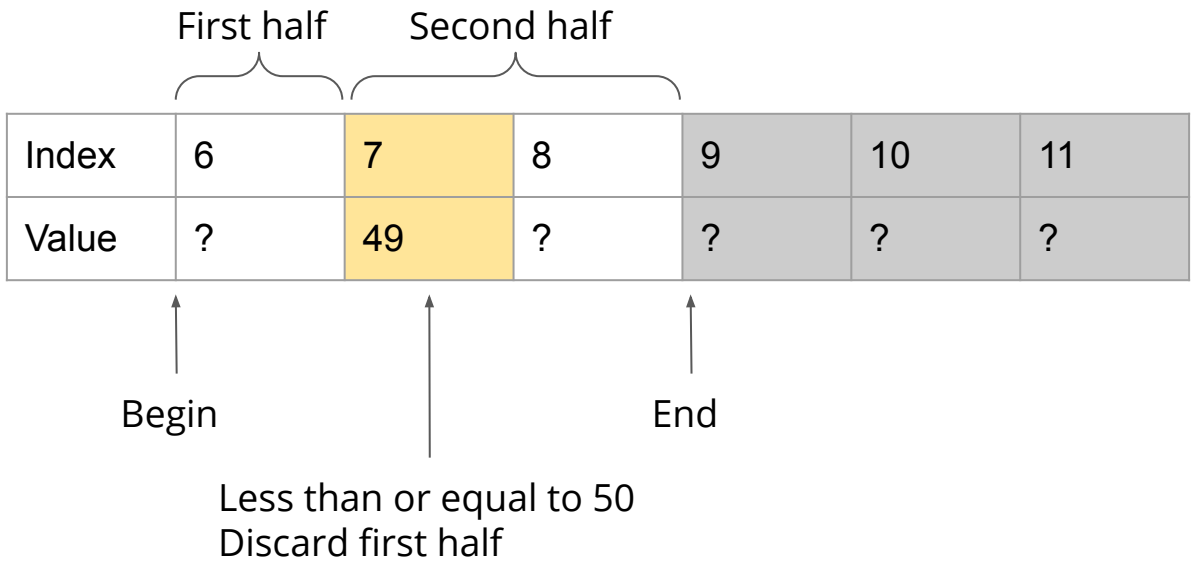
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



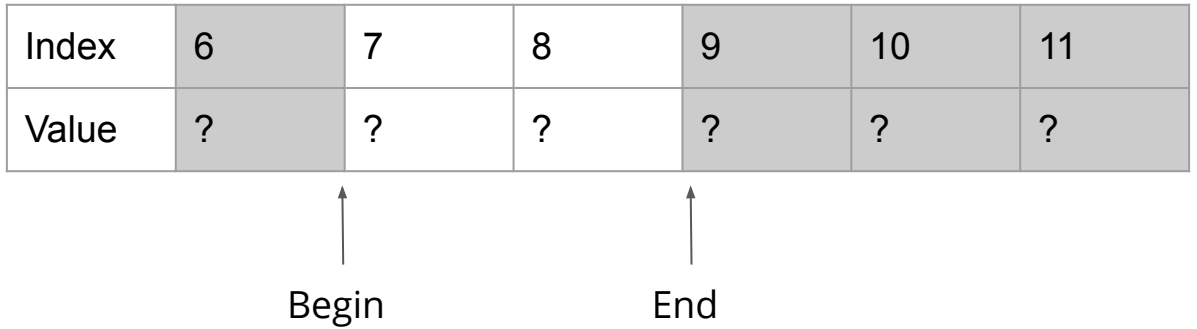
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



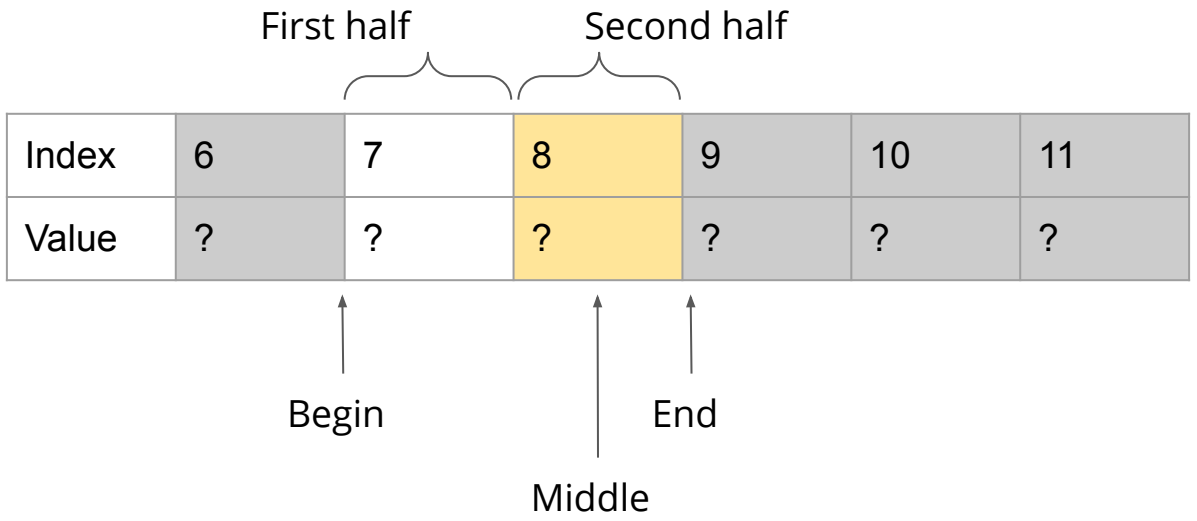
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



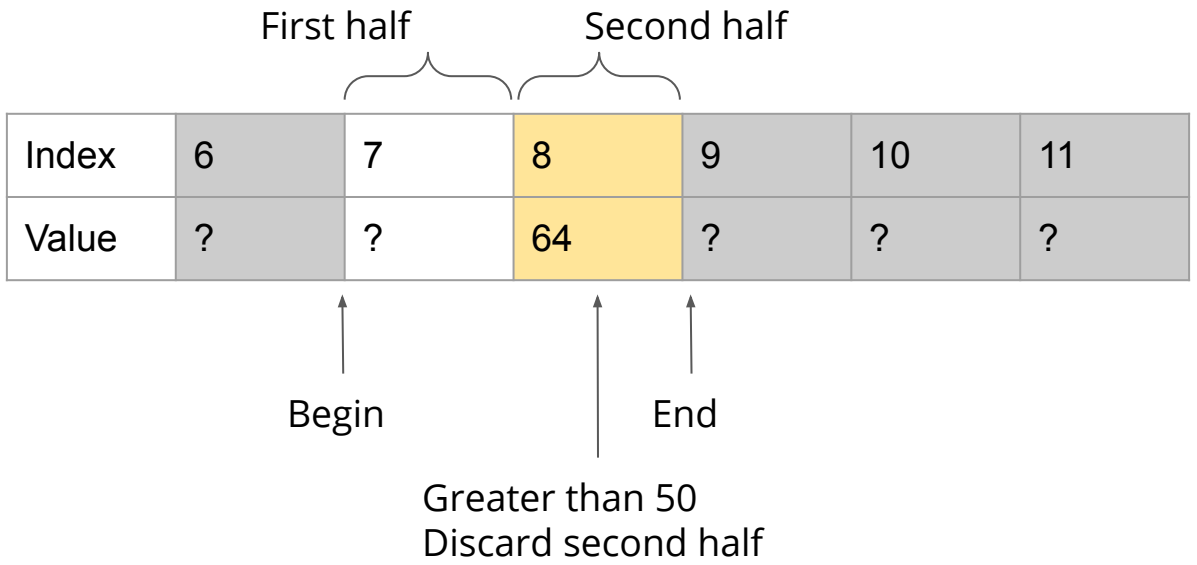
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



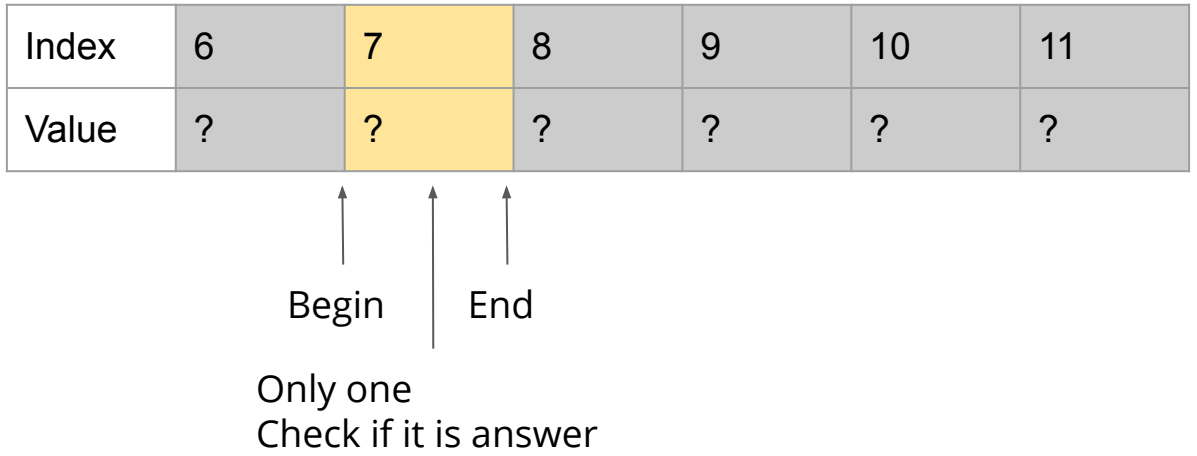
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



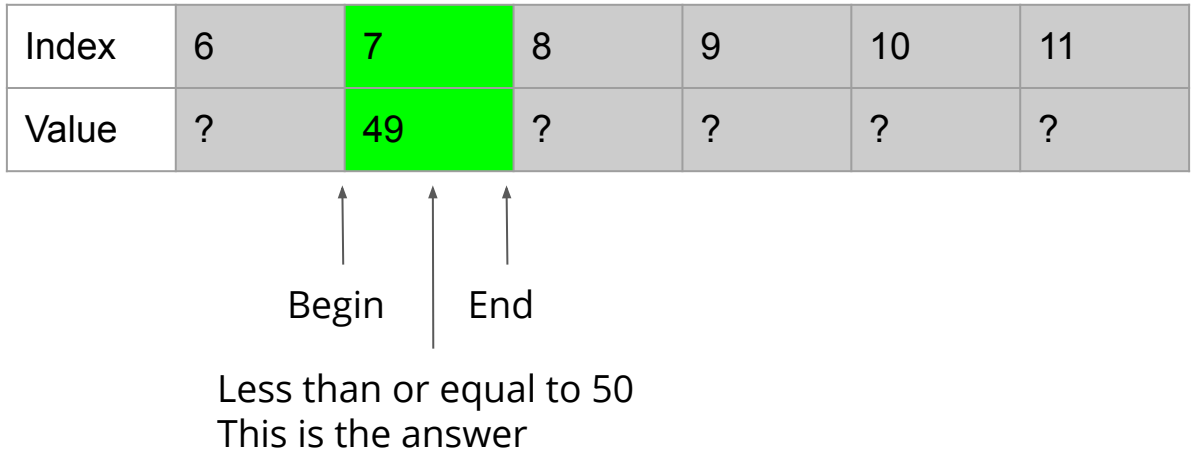
Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



Binary Search - Binary Search on Answer

Example: Find the largest integer k such that $k^2 <= 50$



Binary Search - Binary Search on Answer

```
int find(int x) {
  int lo = 0, hi = x; // include lo, exclude hi
  while (lo + 1 < hi) {
    int mi = (lo + hi) / 2;
    if (mi * mi <= x) {
      lo = mi;
    } else {
      hi = mi;
    }
  }
  return mi * mi <= x ? lo : -1; // -1 for no answer
}
```

Binary Search - Binary Search on Answer

A more generic way is to make the virtual array having only 0 or 1 as value.

For the previous example, we may define the array as

a[i] =

- 0, if $i^2 <= 50$
- 1, if $i^2 > 50$

Then we are also finding the largest index such that the value < 1.

Binary Search - Binary Search on Answer

```
inf f(int k, int x) {
  if (k * k <= x) {
    return 0;
  } else {
    return 1;
  }
}

int find(int x) {
  int lo = 0, hi = x; // include lo, exclude hi
  while (lo + 1 < hi) {
    int mi = (lo + hi) / 2;
    if (f(mi, x) == 0) { // f returns 0 if mi * mi <= x, returns 1 otherwise
      lo = mi;
    } else {
      hi = mi;
    }
  }
  return mi * mi <= x ? lo : -1; // -1 for no answer
}
```

Binary Search - Binary Search on Answer

We may also extend the algorithm so that the virtual index need not to be integer, but we can only approximate the answer in this case.

Binary Search - Binary Search on Answer

When to consider using binary search on answer?

- When you see the word similar to “largest OOO smaller than XXX” or “smallest OOO larger than XXX”
- The reverse problem is easier to solve (easy to check whether there is a solution for a given k)
- Some properties like “if x is not the answer then x-1 must not be the answer”

Binary Search

Pros:

- Fast

Cons:

- **The array must be sorted**
- Easy to have bug

Binary Search

Practice: S144, M1714, M1222

On codeforces: CF1366A, CF371C, CF448D, CF's tag

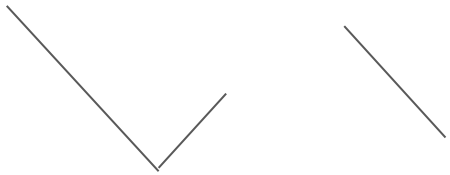
How to Search? Ternary Search

Ternary Search

This algorithm is used to find the minimum or maximum point of a **convex function**.

This may not be very useful for OI contest since the discrete case can be easily done using binary search. (binary search on the turning point)

Ternary Search



Suppose we want to find the minimum point

Algorithm:

1. Divide the range into three part, let m1 be the one-third point, m2 be the two-third point where m1 = lo + (hi-lo)/3, m2 = hi - (hi-lo)/3
2. Compare f(m1) and f(m2)

- If f(m1) > f(m2) then discard the smallest one-third
- If f(m1) < f(m2) then discard the largest one-third
- If f(m1) = f(m2) then keep only the middle one-third

3. Repeat 2 until we find the answer

* This algorithm only works if both the increasing and decreasing part are strict.

Ternary Search

$T(n) = T(2n/3) + 1 \Rightarrow T(n) = O(\lg n)$

Time complexity: O(lg n), the same as binary search

It is not recommended to use this algorithm during contest, try to rewrite it using binary search, there will be less bugs.

Let’s have a break!

Sorting

What is Sorting?

Sorting is the process of rearranging elements in an array according to some order (eg. ascending order for numbers, lexicographical order for strings)

Sorting is required so as to facilitate other algorithms, such as binary search.

There are mainly 2 types of sorting:

- comparison sorts
- non-comparison sorts

Sorting

comparison sorting algorithms:

- Selection sort
- Insertion sort
- Bubble sort
- Quick sort
- Merge sort
- Heapsort
- Introsort
- Shellsort

More...

non-comparison sorting algorithms:

- Counting sort
- LSD Radix sort
- MSD Radix sort

More...

Selection Sort

Algorithm:

We will try to maintain a sorted prefix.

At each round, we will select the smallest element in the unsorted suffix and swap it to the back of the sorted prefix.

Visualization: https://visualgo.net/bn/sorting

Selection Sort

	Worst	Average
Swaps	O(n)	O(n)
Comparisons	$O(n^2)$	$O(n^2)$
Overall	$O(n^2)$	$O(n^2)$

```
for (int i = 0; i < n; i++) {
  int m = i; // index of smallest element
  for (int j = i + 1; j < n; j++) {
    if (a[j] < a[m]) {
      m = j;
    }
  }
  swap(a[i], a[m]);
}
```

Selection Sort

Pros:

- It do the least number of swaps possible (O(n)), even less than O(n lg n) sorting algorithms
- O(1) extra space (in-place)

Cons:

- Slow ($O(n^2)$)

Insertion Sort

Algorithm:

We will try to maintain a sorted prefix.

At each round, we will insert a new element into of the sorted prefix by swapping

Visualization: https://visualgo.net/bn/sorting

Insertion Sort

	Worst	Average
Swaps	$O(n^2)$	$O(n^2)$
Comparisons	$O(n^2)$	$O(n^2)$
Overall	$O(n^2)$	$O(n^2)$

```
for (int i = 0; i < n; i++) {
  for (int j = i; j > 0; j--) {
    if (a[j] < a[j - 1]) {
      swap(a[j], a[j - 1]);
    } else {
      break;
    }
  }
}
```

Insertion Sort

Pros:

- Fast (O(n)) if the initial array is almost sorted
- O(1) extra space (in-place)

Cons:

- Slow ($O(n^2)$)

Bubble Sort

Algorithm:

We will try to maintain a sorted suffix.

At each round, we will try from left to right swapping the neighbouring elements if their order is wrong

Visualization: https://visualgo.net/bn/sorting

Bubble Sort

	Worst	Average
Swaps	$O(n^2)$	$O(n^2)$
Comparisons	$O(n^2)$	$O(n^2)$
Overall	$O(n^2)$	$O(n^2)$

```
for (int i = 0; i < n; i++) {
  for (int j = 1; j + i < n; j++) {
    if (a[j] < a[j - 1]) {
      swap(a[j], a[j - 1]);
    }
  }
}
```

Bubble Sort

Pros:

- Easy to code
- O(1) extra space (in-place)

Cons:

- Slow ($O(n^2)$)

Bubble Sort

A easy to code version of bubble sort

```
for (int i = 0; i < n; i++) {
  for (int j = 1; j < n; j++) {
    if (a[j] < a[j - 1]) {
      swap(a[j], a[j - 1]);
    }
  }
}
```

Quick Sort

A divide-and-conquer algorithm

Algorithm:

Pick a pivot element (maybe random)

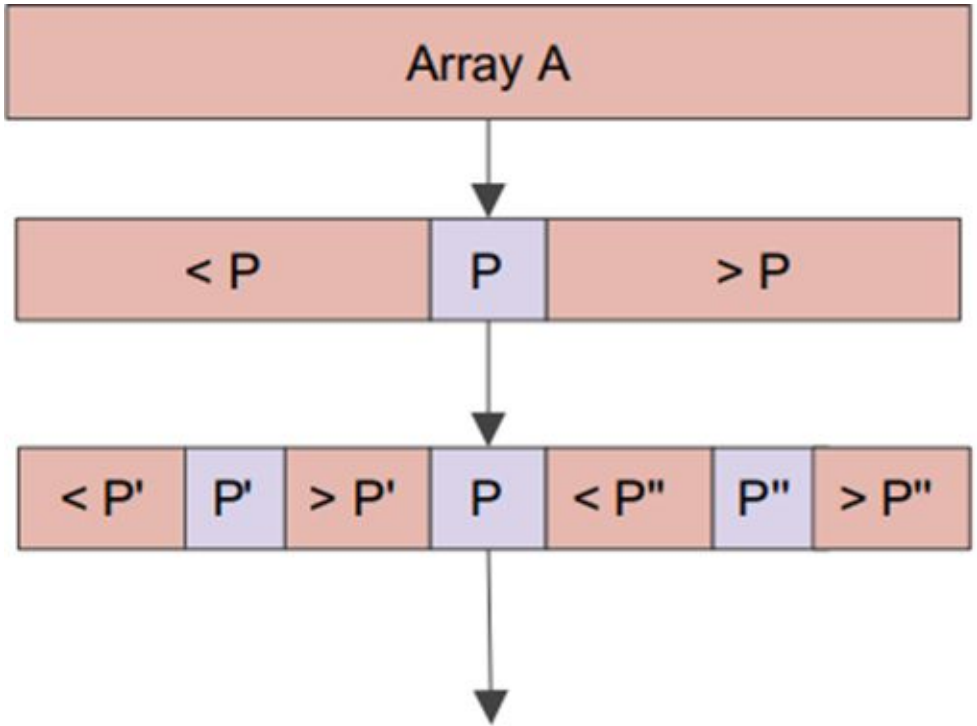
Split the array into two parts, the first part is those less than or equal to the pivot, the second part is those greater than the pivot.

Sort both parts using quick sort (base case: size = 1)

Put those parts together in order.

Quick Sort

Visualization: https://visualgo.net/bn/sorting



Quick Sort

	Worst	Average
Swaps	$O(n^2)$	O(n lg n)
Comparisons	$O(n^2)$	O(n lg n)
Overall	$O(n^2)$	O(n lg n)

```
void quick_sort(int a[], int begin, int end) {
  if (begin + 1 <= end) {
    return;
  }
  int l = 0, p = a[begin];
  for (int i = begin + 1; i < end; i++) {
    if (a[i] <= p) {
      swap(a[i], a[begin + l + 1]);
      l++;
    }
  }
  swap(a[begin], a[begin + l]);
  quick_sort(a, begin, begin + l);
  quick_sort(a, begin + l + 1, end);
}
```

Quick Sort

Pros:

- Fast on average

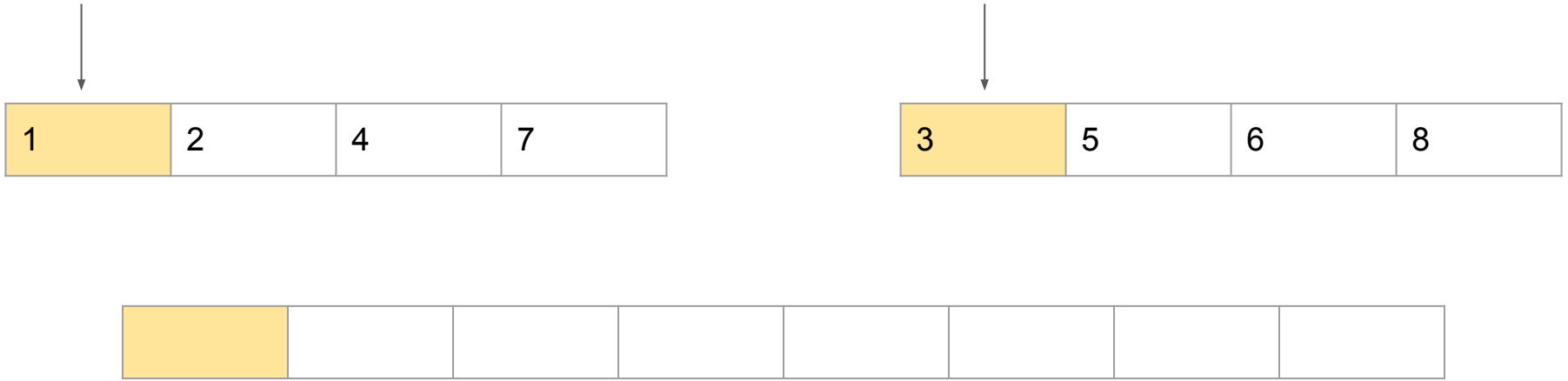
Cons:

- Slow worst case complexity ($O(n^2)$)

Merge Sort

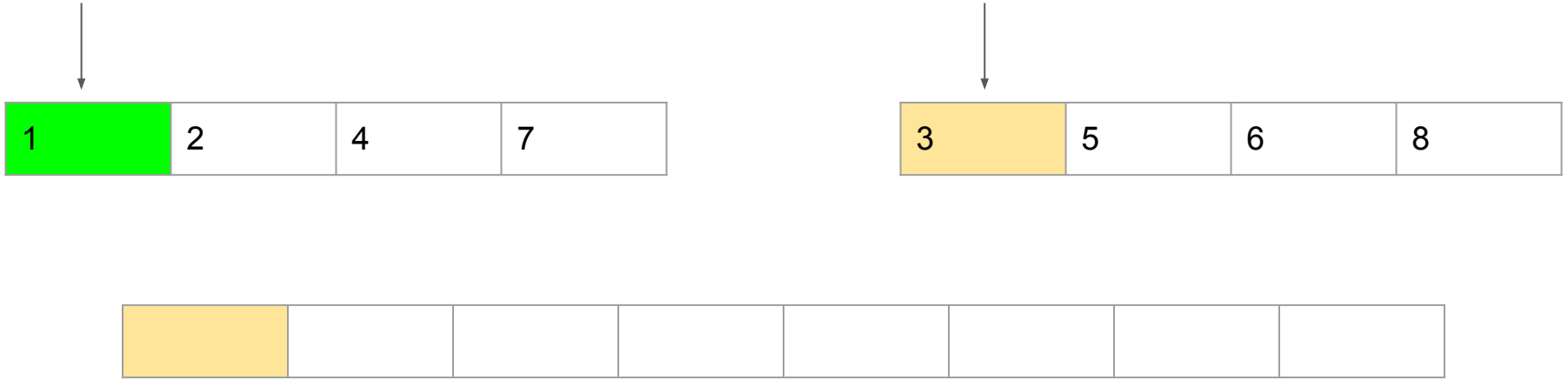
Before talking about merge sort, we need to know how to merge two sorted array efficiently. (Problem: D804)

Merge Sort

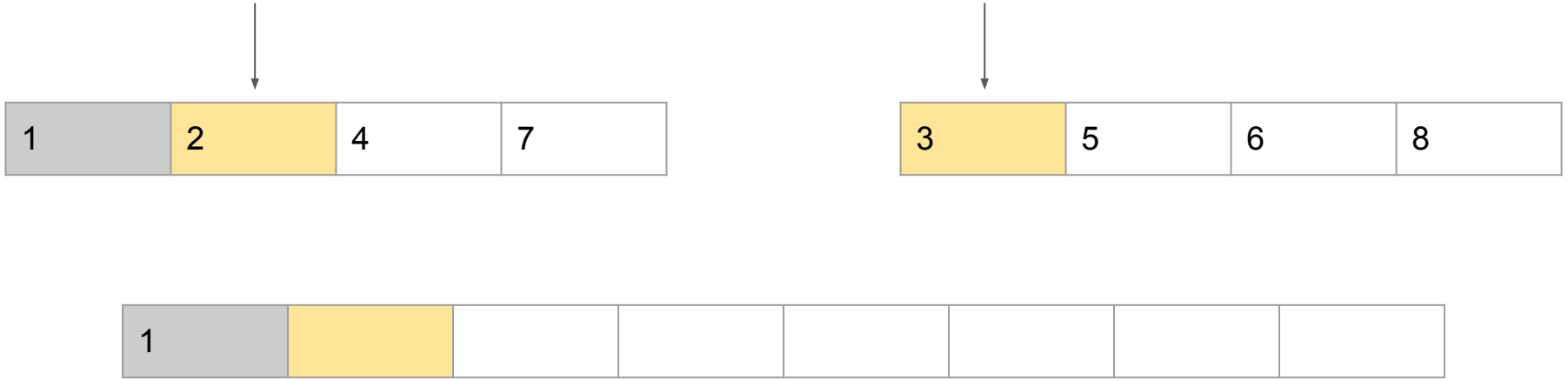


Merge Sort

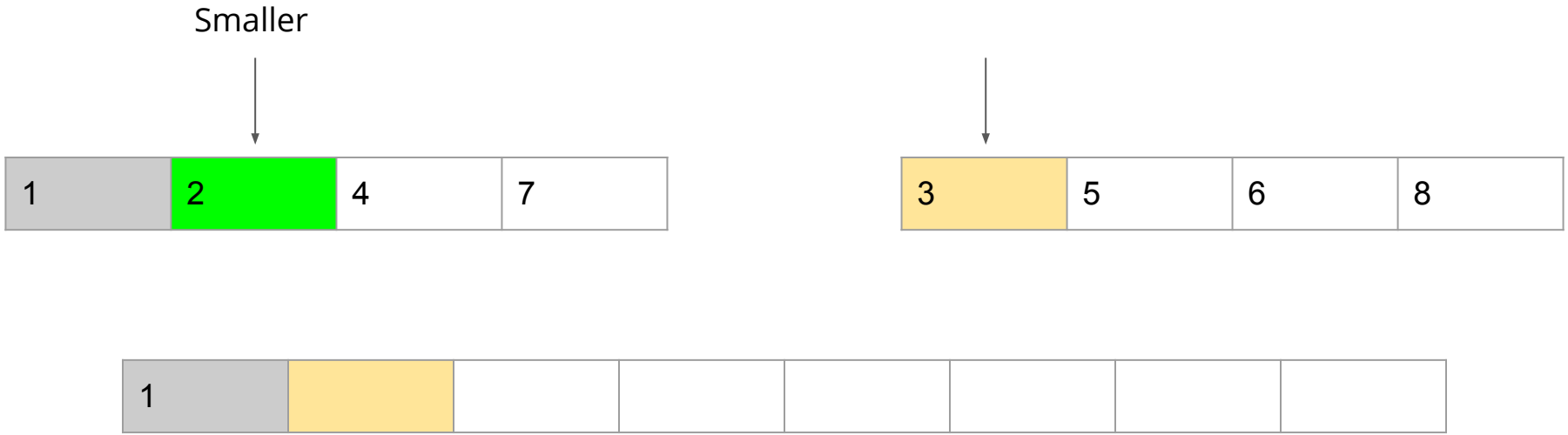
Smaller



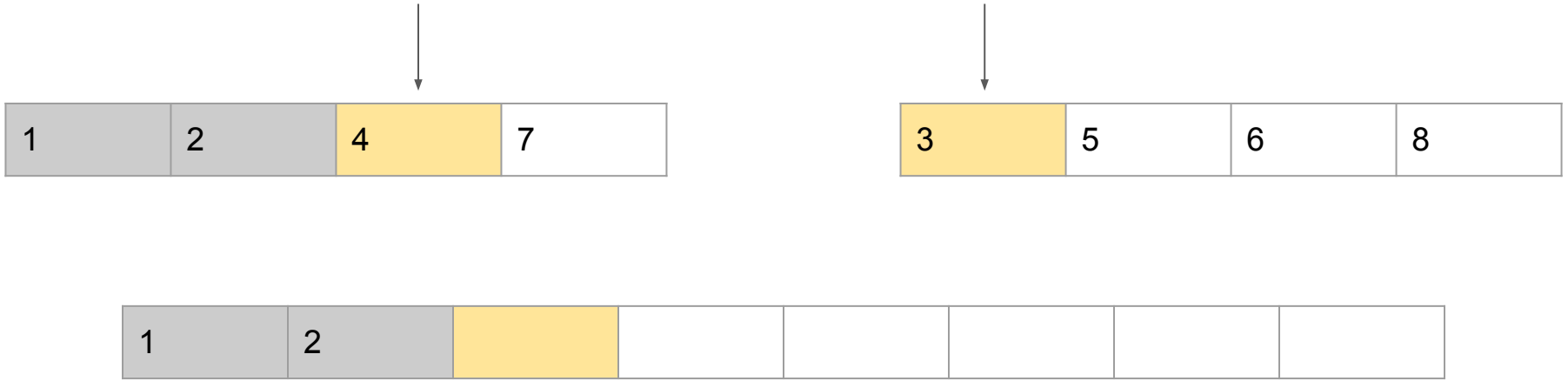
Merge Sort



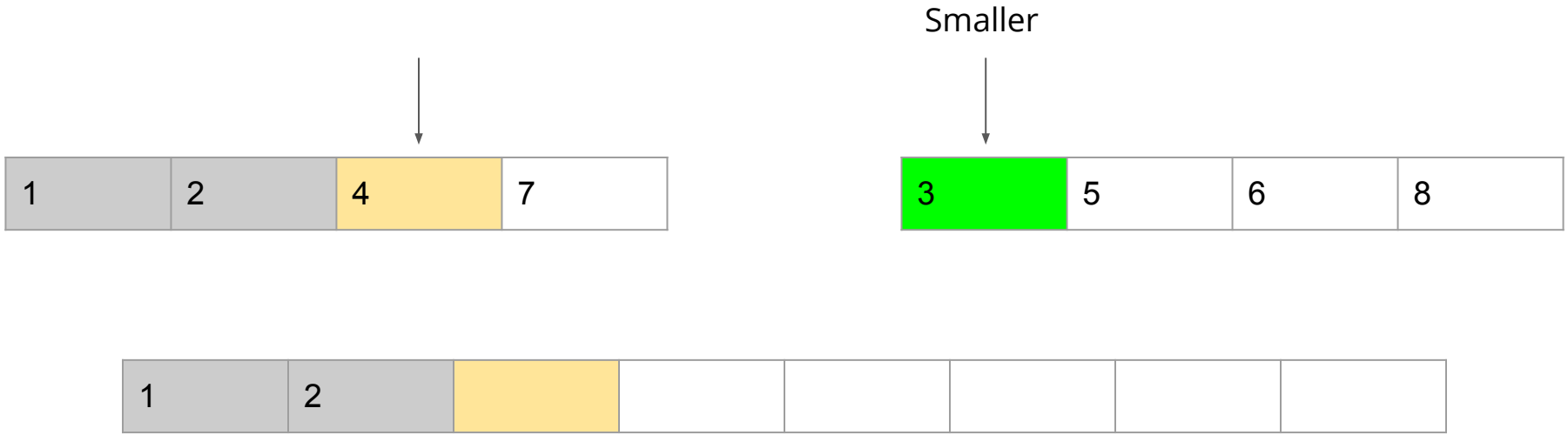
Merge Sort



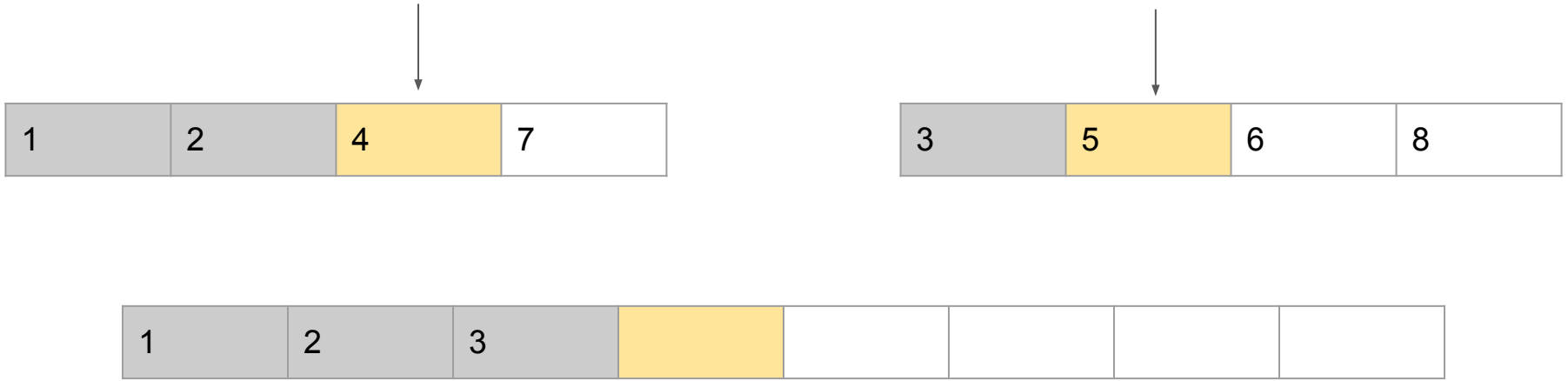
Merge Sort



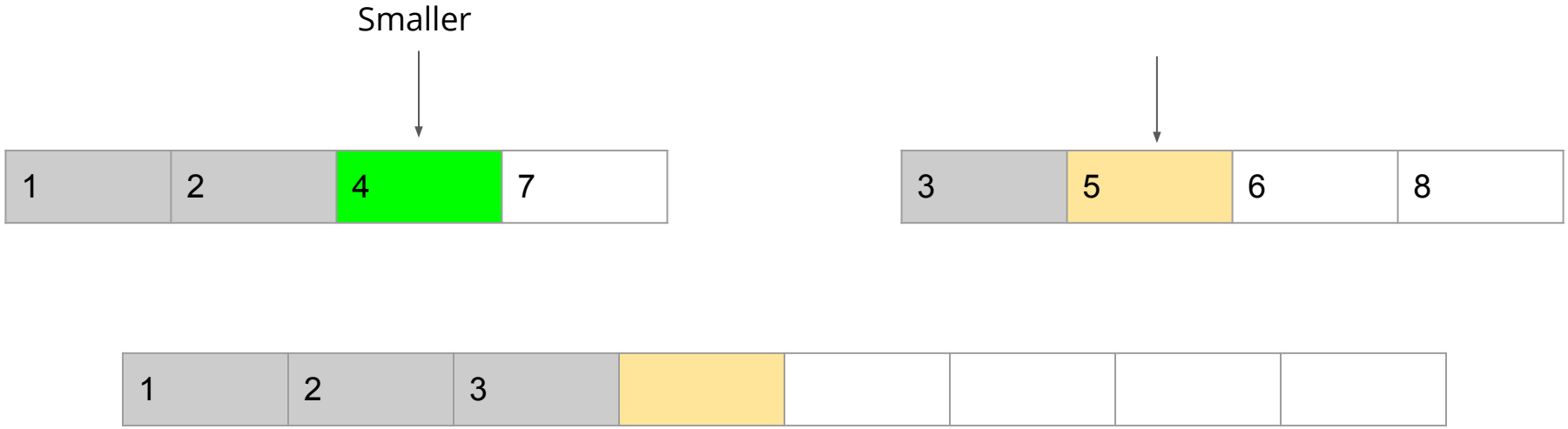
Merge Sort



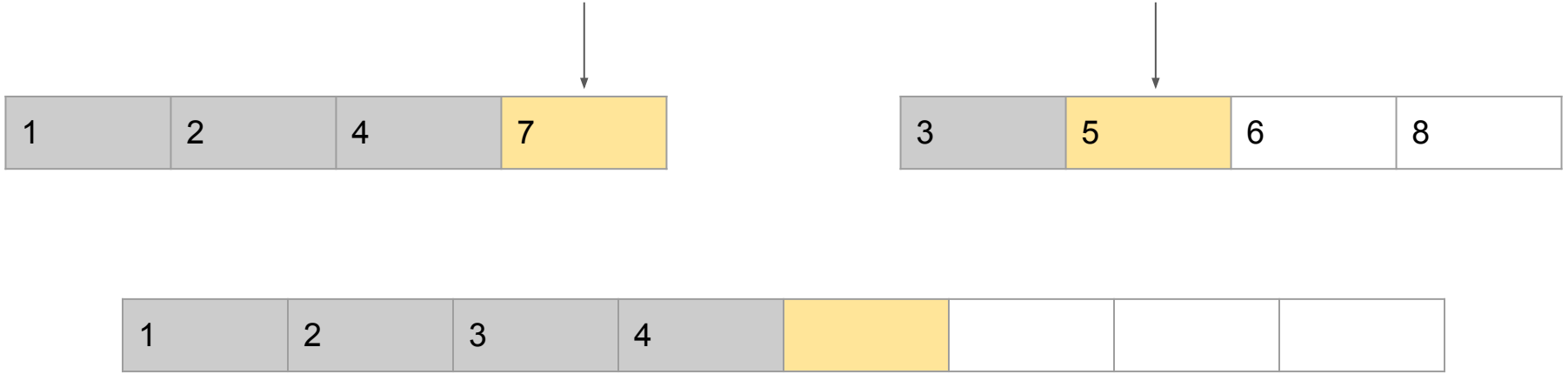
Merge Sort



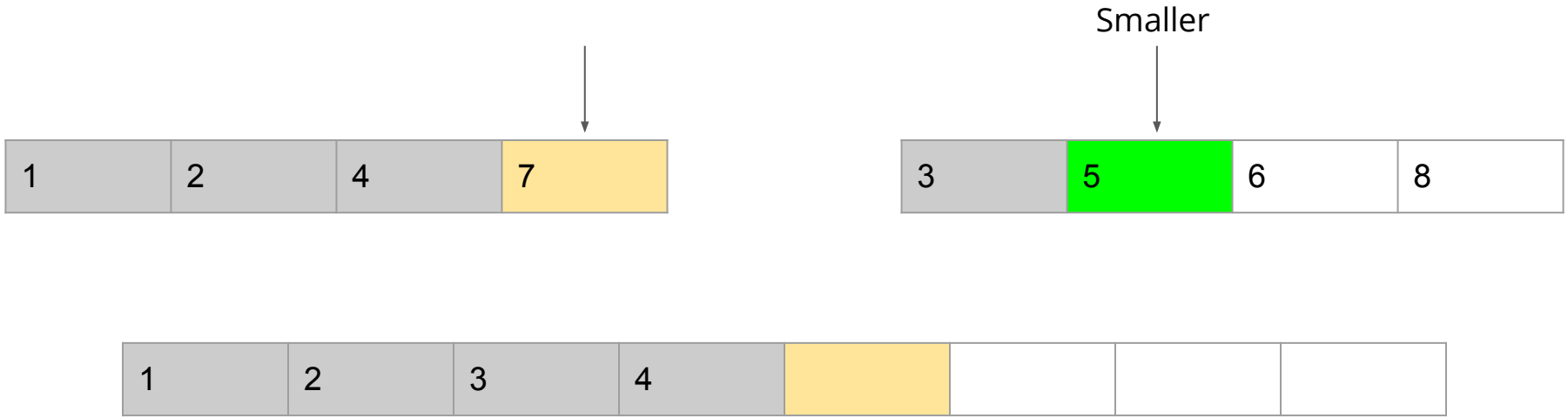
Merge Sort



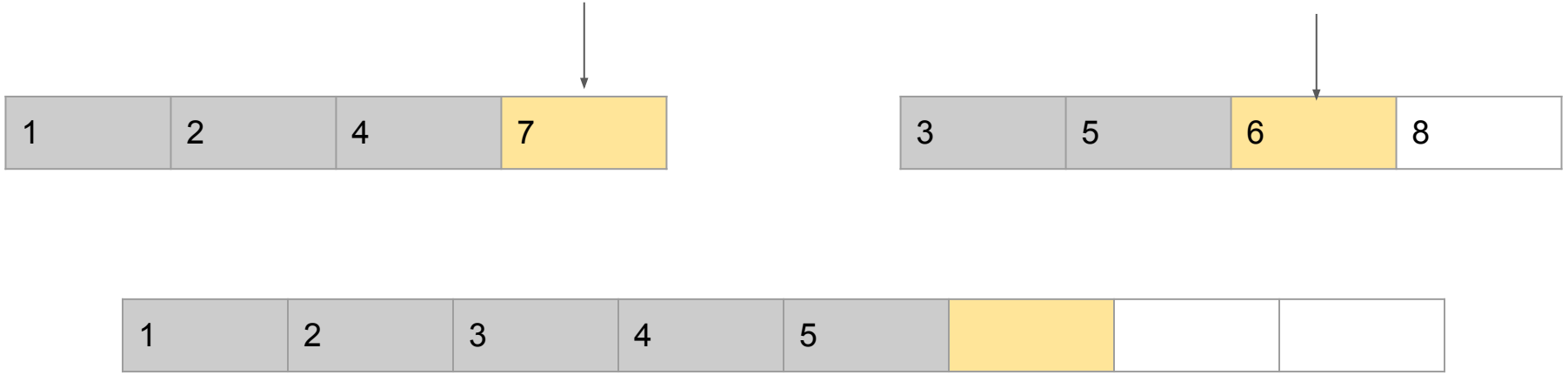
Merge Sort



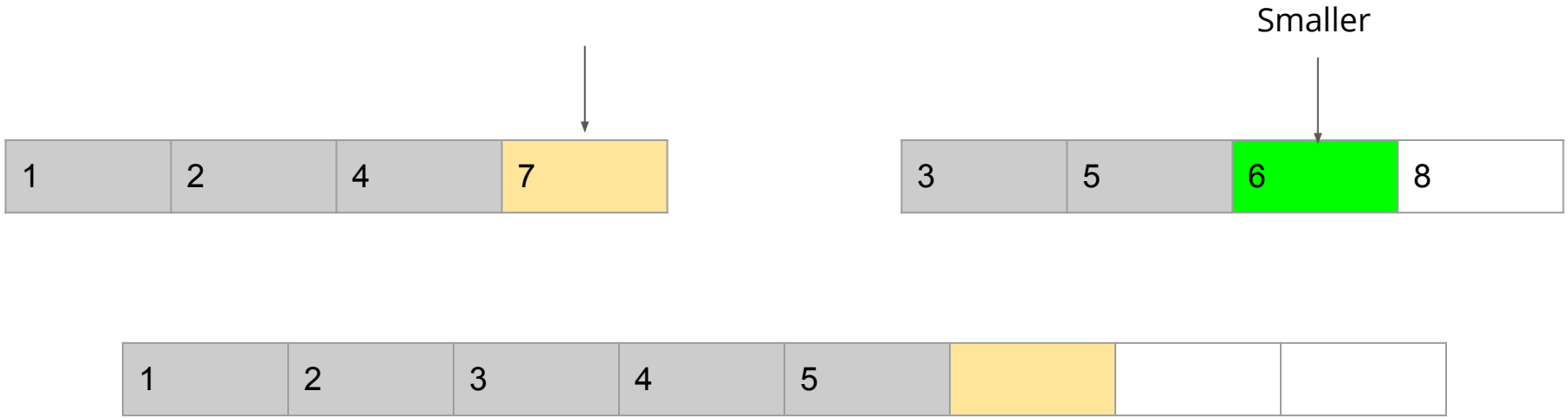
Merge Sort



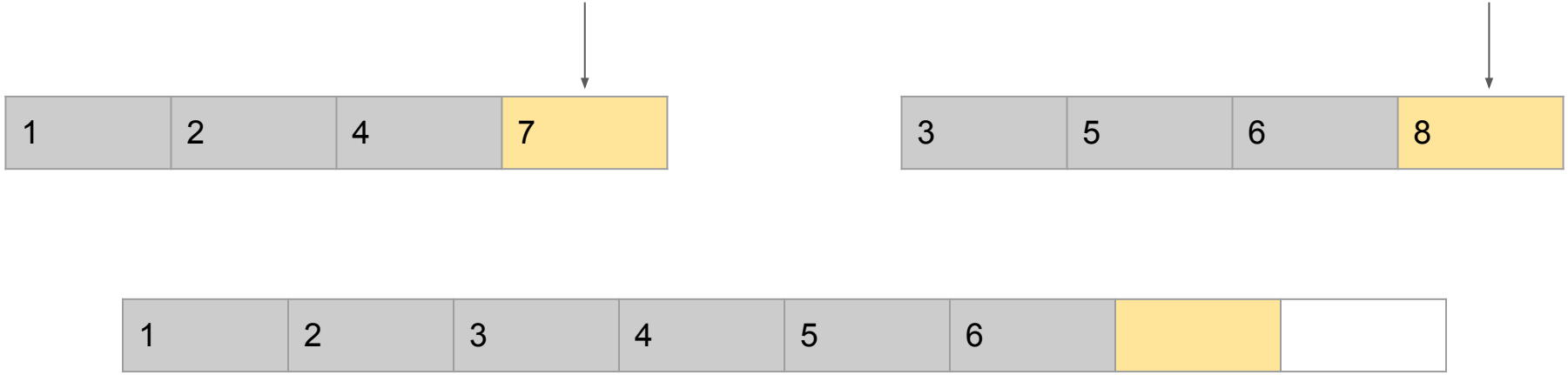
Merge Sort



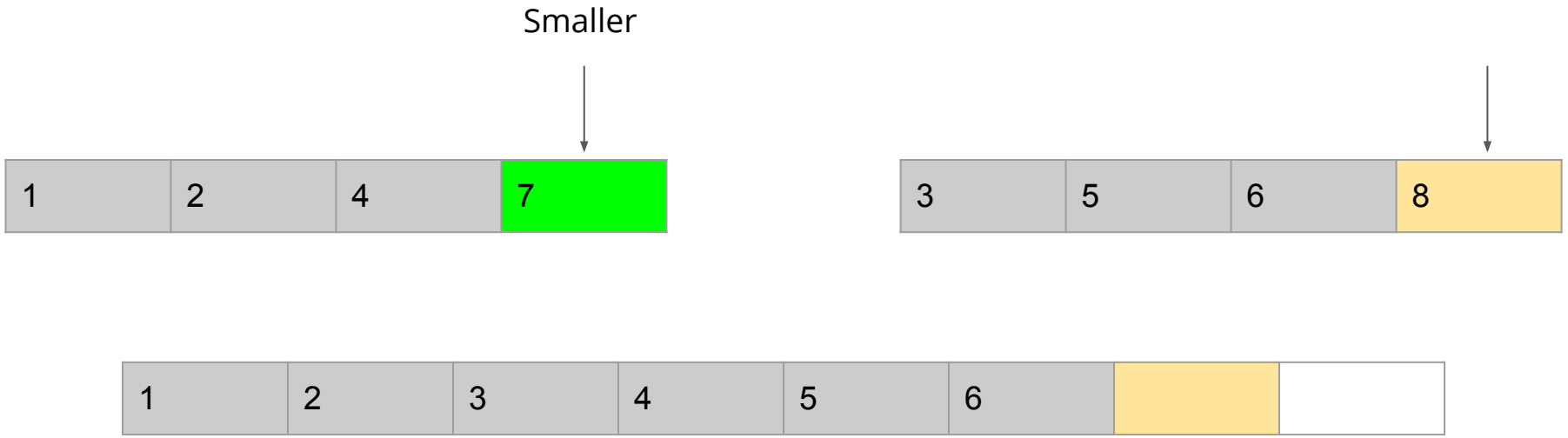
Merge Sort



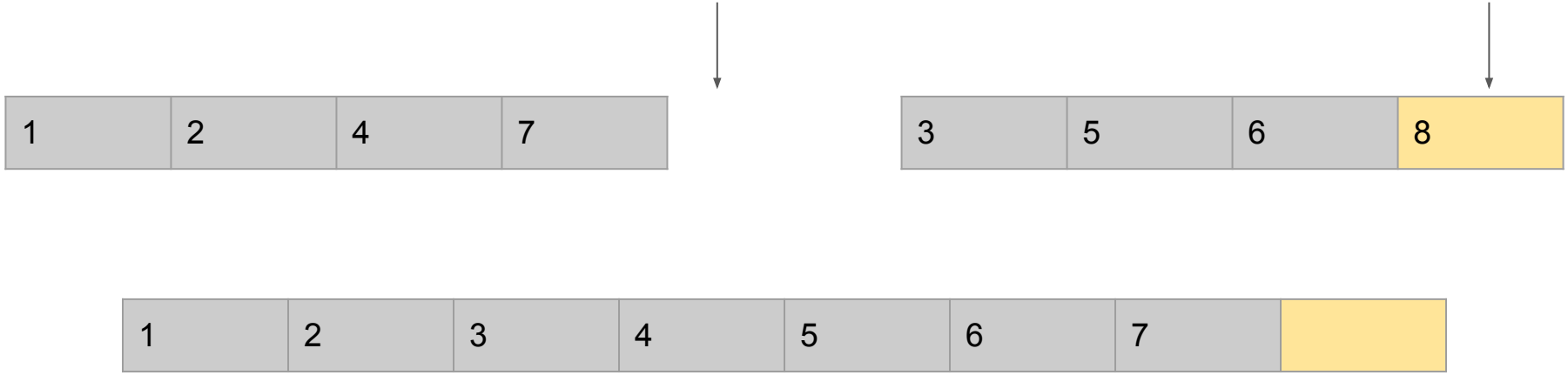
Merge Sort



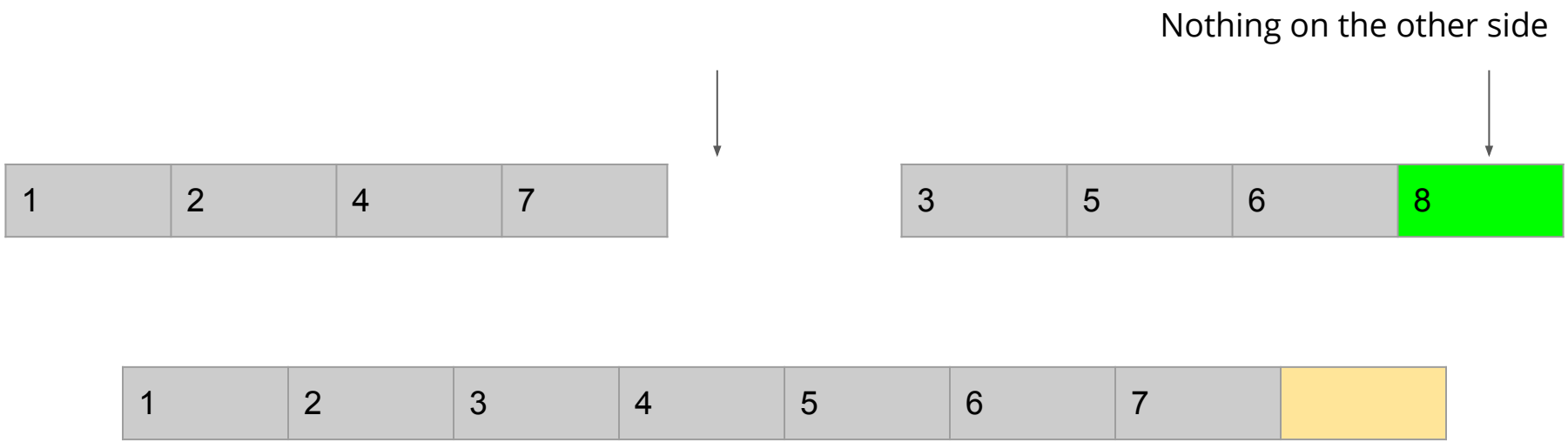
Merge Sort



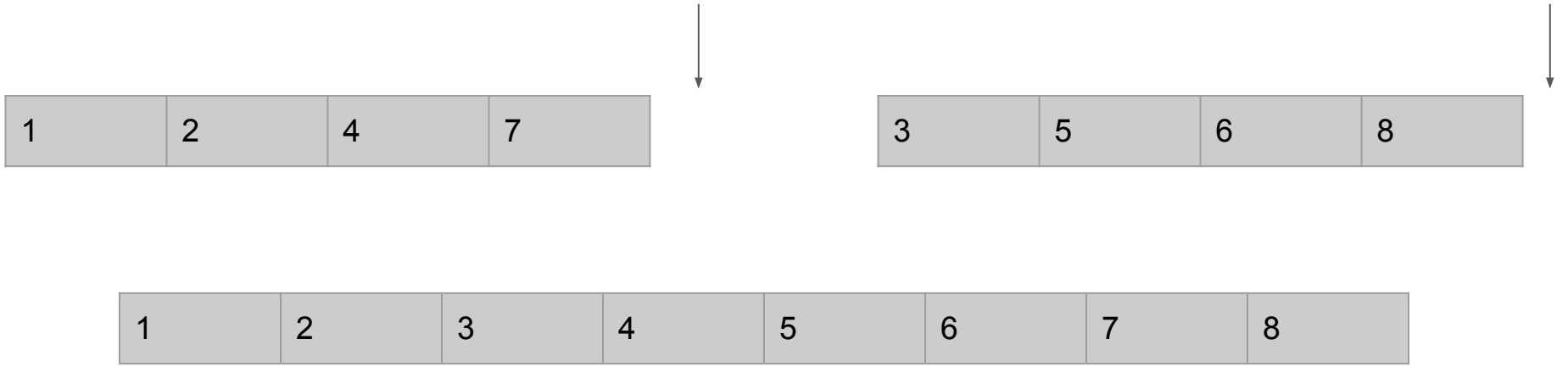
Merge Sort



Merge Sort



Merge Sort



Merge Sort

The merging algorithm works in O(n).

Merge Sort

A divide-and-conquer algorithm

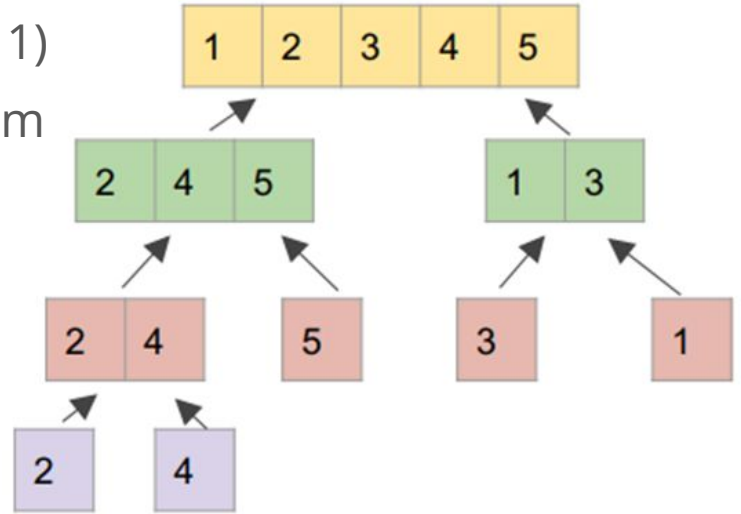
Algorithm:

Divide the array into two half.

Sort both half using merge sort (base case: size = 1)

Merge both parts using the O(n) merging algorithm

Visualization: https://visualgo.net/bn/sorting



Merge Sort

$T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$

	Worst	Average
Swaps	O(n lg n)	O(n lg n)
Comparisons	O(n lg n)	O(n lg n)
Overall	O(n lg n)	O(n lg n)

```
void merge_sort(int a[], int begin, int end) {
  if (begin + 1 == end) {
    return;
  }
  int mid = (begin + end) / 2;
  merge_sort(a, begin, mid);
  merge_sort(a, mid, end);
  merge(begin, mid, end); // merge the two sorted parts
}
```

Inversions

Given any permutation, the number of inversions of the permutation is the number of ordered tuple (i, j) such that i < j but a[i] > a[j]

Practice: D802

Easy: Find the number of inversions in $O(n^2)$

Hard: Find the number of inversions in O(n lg n)

Comparison Sorts

It can been shown that O(n lg n) is the lower bound for a comparison sorting algorithm, so we cannot find any comparison based sorting algorithm faster than O(n lg n).

See wiki.

All the five previous algorithms are comparison sorting algorithms, so their limit is O(n lg n), no more faster algorithms unless you are not doing comparisons.

C++ sort

Again in C++'s algorithm library, there is a very handy sorting function which support custom order sorting.

We may pass a lambda function as the last argument, it will be considered as a custom "less than" function. (we can also do this for lower_bound and upper_bound)

```
sort(a.begin(), a.end()); // sort in ascending order
sort(a.begin(), a.end(), [] (int x, int y) {
  return x > y;
}); // sort in descending order
```

Counting Sort

This algorithm is useful when we have a lot of elements in an array but the possible types of elements are not many (e.g. small range for integers or characters)

Algorithm:

Build a frequency table and then go through the frequency table in ascending order.

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Build the frequency table with 4 buckets.

Array:	2	4	1	3	1

Element:	1	2	3	4
Count:	0	1	0	0

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Build the frequency table with 4 buckets.

Array:	2	4	1	3	1

Element:	1	2	3	4
Count:	0	1	0	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Build the frequency table with 4 buckets.

Array:	2	4	1	3	1

Element:	1	2	3	4
Count:	1	1	0	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Build the frequency table with 4 buckets.

Array:	2	4	1	3	1

Element:	1	2	3	4
Count:	1	1	1	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Build the frequency table with 4 buckets.

Array:	2	4	1	3	1

Element:	1	2	3	4
Count:	2	1	1	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Build the frequency table with 4 buckets.

Array:	2	4	1	3	1

Element:	1	2	3	4
Count:	2	1	1	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Construct the sorted array.

Sorted Array:					

Element:	1	2	3	4
Count:	2	1	1	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Construct the sorted array.

Sorted Array:	1	1			

Element:	1	2	3	4
Count:	2	1	1	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Construct the sorted array.

Sorted Array:	1	1			

Element:	1	2	3	4
Count:	2	1	1	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Construct the sorted array.

Sorted Array:	1	1	2		

Element:	1	2	3	4
Count:	2	1	1	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Construct the sorted array.

Sorted Array:	1	1	2		

Element:	1	2	3	4
Count:	2	1	1	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Construct the sorted array.

Sorted Array:	1	1	2	3	

Element:	1	2	3	4
Count:	2	1	1	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Construct the sorted array.

Sorted Array:	1	1	2	3	

Element:	1	2	3	4
Count:	2	1	1	1

Counting Sort

Suppose the array contains only 1, 2, 3 or 4.

Construct the sorted array.

Sorted Array:	1	1	2	3	4

Element:	1	2	3	4
Count:	2	1	1	1

Counting Sort

Visualization: https://visualgo.net/bn/sorting

Counting Sort

Let n be the size of the array

Let m be the number of buckets

Time complexity: O(n + m)

Space complexity: O(n + m)

Use counting sort only when m is small

```
for (int i = 0; i < n; i++) {
  cnt[a[i]]++;
}
int l = 0; // size of constructed array
for (int i = lo; i < hi; i++) {
  for (int j = 0; j < cnt[i]; j++) {
    a[l++] = i;
  }
}
```

Counting Sort

Another implementation

```
for (int i = 0; i < n; i++) {
  cnt[a[i]]++;
}
for (int i = lo; i < hi; i++) {
  cnt[i] += cnt[i - 1];
}
for (int i = 0; i < n; i++) {
  tmp[--cnt[a[i]]] = a[i];
}
for (int i = 0; i < n; i++) {
  a[i] = tmp[i];
}
```

LSD Radix Sort

Least **S**ignificant **D**igit Radix Sort

It is like multiple rounds of counting sort, one round for every digit, starting from the least significant digit.

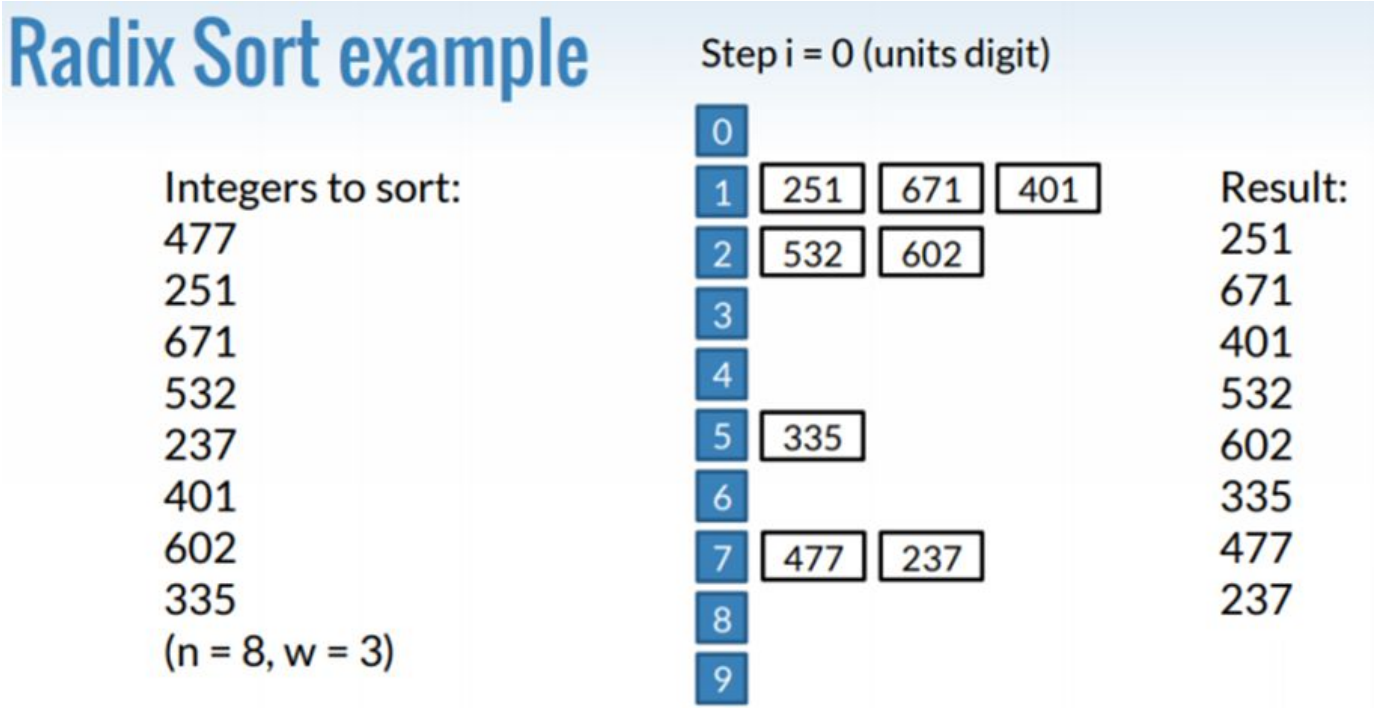
Algorithm:

At the i-th round, we do a counting sort for the i-th digit. Instead of recording the frequency, we record the element in the bucket in the form of queue.

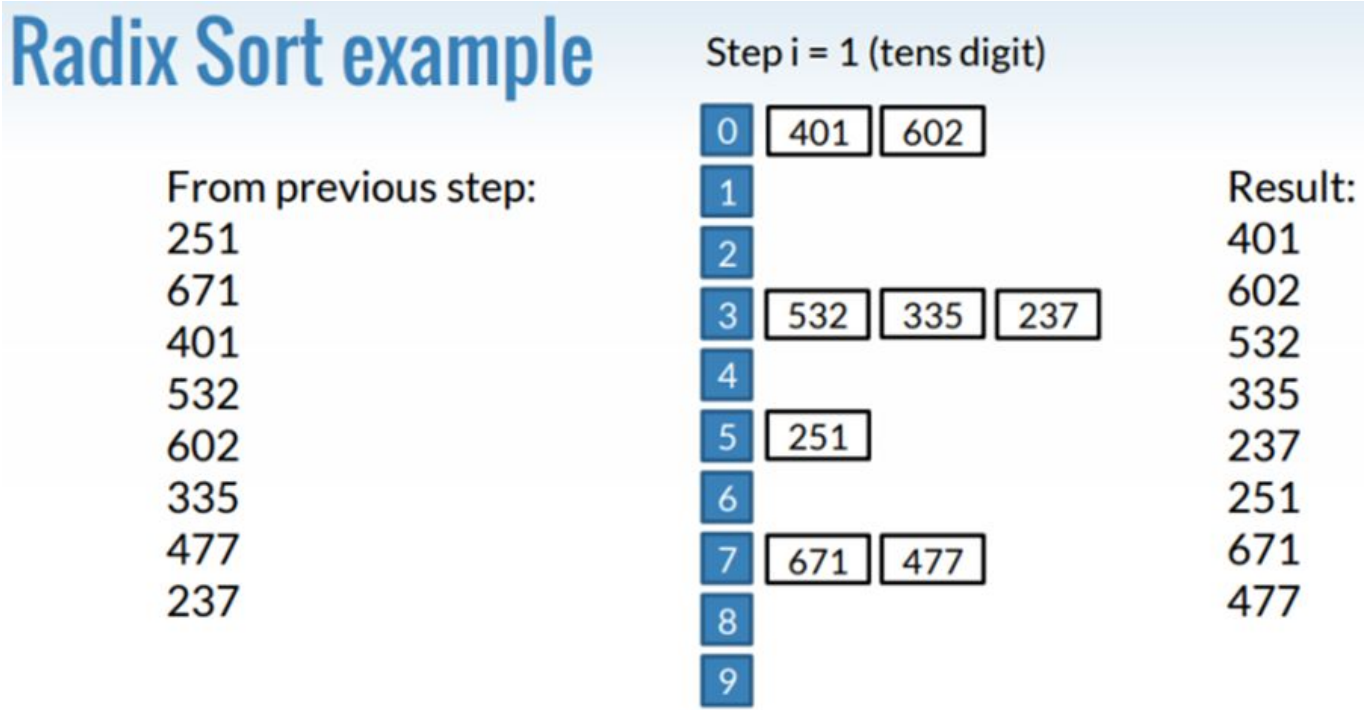
After that, we reconstruct the array from the buckets in ascending order.

Repeat until the most significant digit.

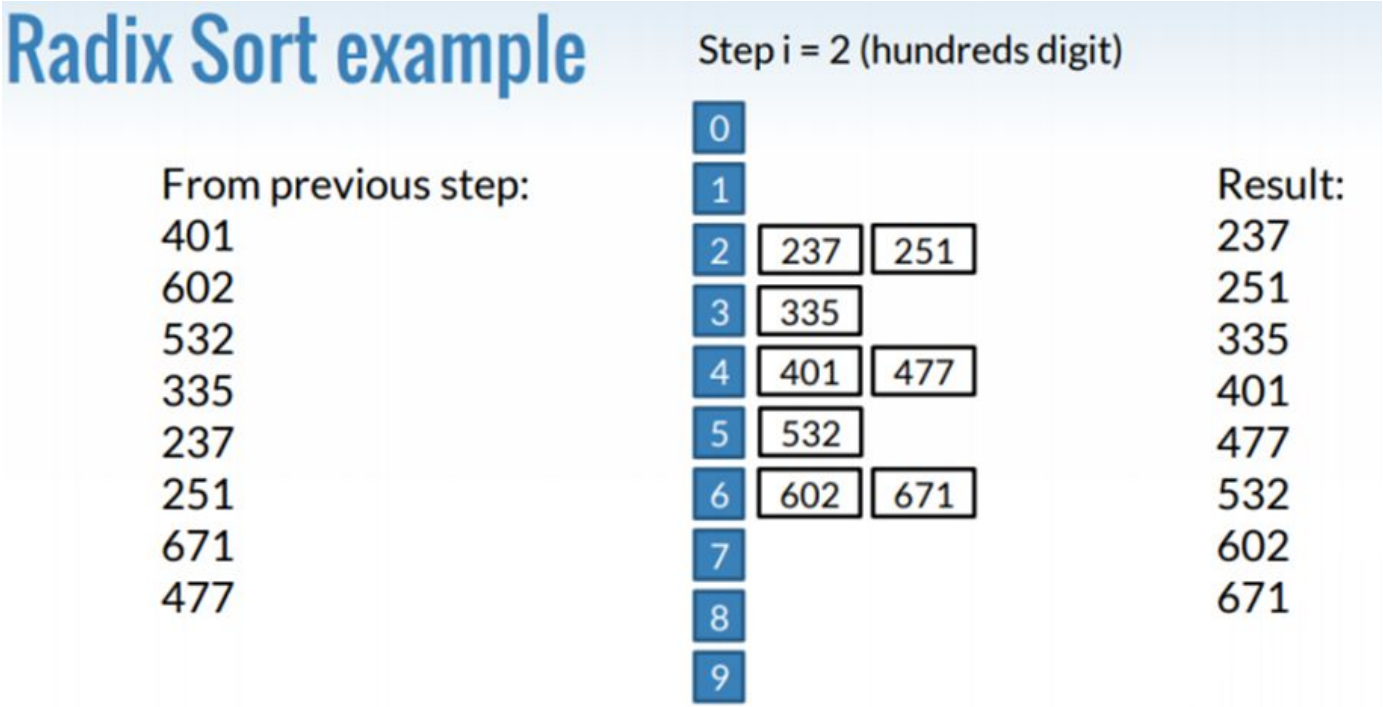
LSD Radix Sort



LSD Radix Sort



LSD Radix Sort



LSD Radix Sort

Visualization: https://visualgo.net/bn/sorting

LSD Radix Sort

Let n be the size of the array

Let m be the number of buckets in each round

Let w be the number of rounds

Time complexity: O(w(n + m))

Space complexity: O(n + m)

```
// buckets are array of vector
for (int i = 0; i < w; i++) {
  for (int j = 0; j < m; j++) {
    buckets[j].clear();
  }
  for (int j = 0; j < n; j++) {
    buckets[digit(a[j], i)].push_back(a[j]);
  }
  int l = 0; // size of reconstructed array
  for (int j = 0; j < m; j++) {
    for (int x: buckets[j]) {
      a[l++] = x;
    }
  }
}
```

LSD Radix Sort

Another implementation

```
for (int i = 0; i < w; i++) {
  for (int j = 0; j < m; j++) {
    cnt[m] = 0;
  }
  for (int j = 0; j < n; j++) {
    cnt[digit(a[j], i)]++;
  }
  for (int j = 1; j < m; j++) {
    cnt[j] += cnt[j - 1];
  }
  for (int j = n - 1; j > 0; j--) {
    tmp[--cnt[digit(a[j], i)]] = a[j];
  }
  for (int j = 0; j < n; j++) {
    a[j] = tmp[j];
  }
}
```

Q&A

Eye Examination (only if we have time)

From HKOI 2016/17 Junior Q11 & Senior Q11

Which one is bubble sort?

Which one is insertion sort?

```
for (i = 0; i <= n - 1; i++)
  for (j = i; j >= 1; j--)
    if (a[j] < a[j - 1]) {
      temp = a[j];
      a[j] = a[j - 1];
      a[j - 1] = temp;
    }
```

```
for (i = 0; i <= n - 1; i++)
  for (j = 0; j <= i - 1; j++)
    if (a[j] > a[j + 1]) {
      temp = a[j];
      a[j] = a[j + 1];
      a[j + 1] = temp;
    }
```

```
for (i = 0; i <= n - 1; i++)
  for (j = i; j <= n - 2; j++)
    if (a[j] > a[j + 1]) {
      temp = a[j];
      a[j] = a[j + 1];
      a[j + 1] = temp;
    }
```

```
for (i = 0; i <= n - 1; i++)
  for (j = n - 1; j >= i + 1; j--)
    if (a[j] < a[j - 1]) {
      temp = a[j];
      a[j] = a[j - 1];
      a[j - 1] = temp;
    }
```