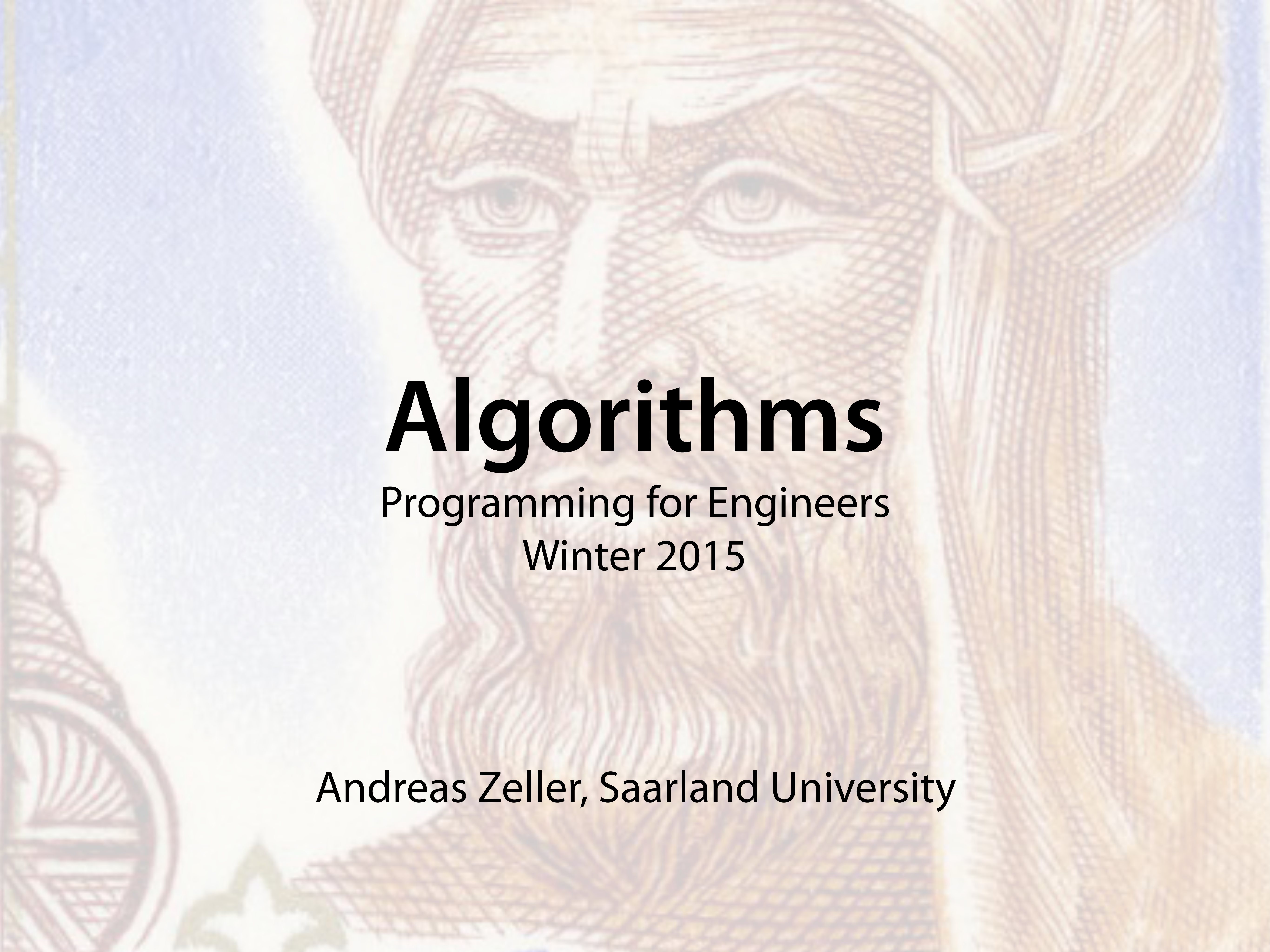


Algorithms

Programming for Engineers
Winter 2015

Andreas Zeller, Saarland University

Setting up the LCD

- This code sets up an LCD object, whose function we can then use

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);
```

- 0x27 is the I2C Address of the LCD Module
- The two other parameters represent the number of characters of the LCD (16x2)

Drink Menu

Characters in C

- A single character in C is written enclosed between two single quotes:

```
char c = 'a';
Serial.println(c);
```

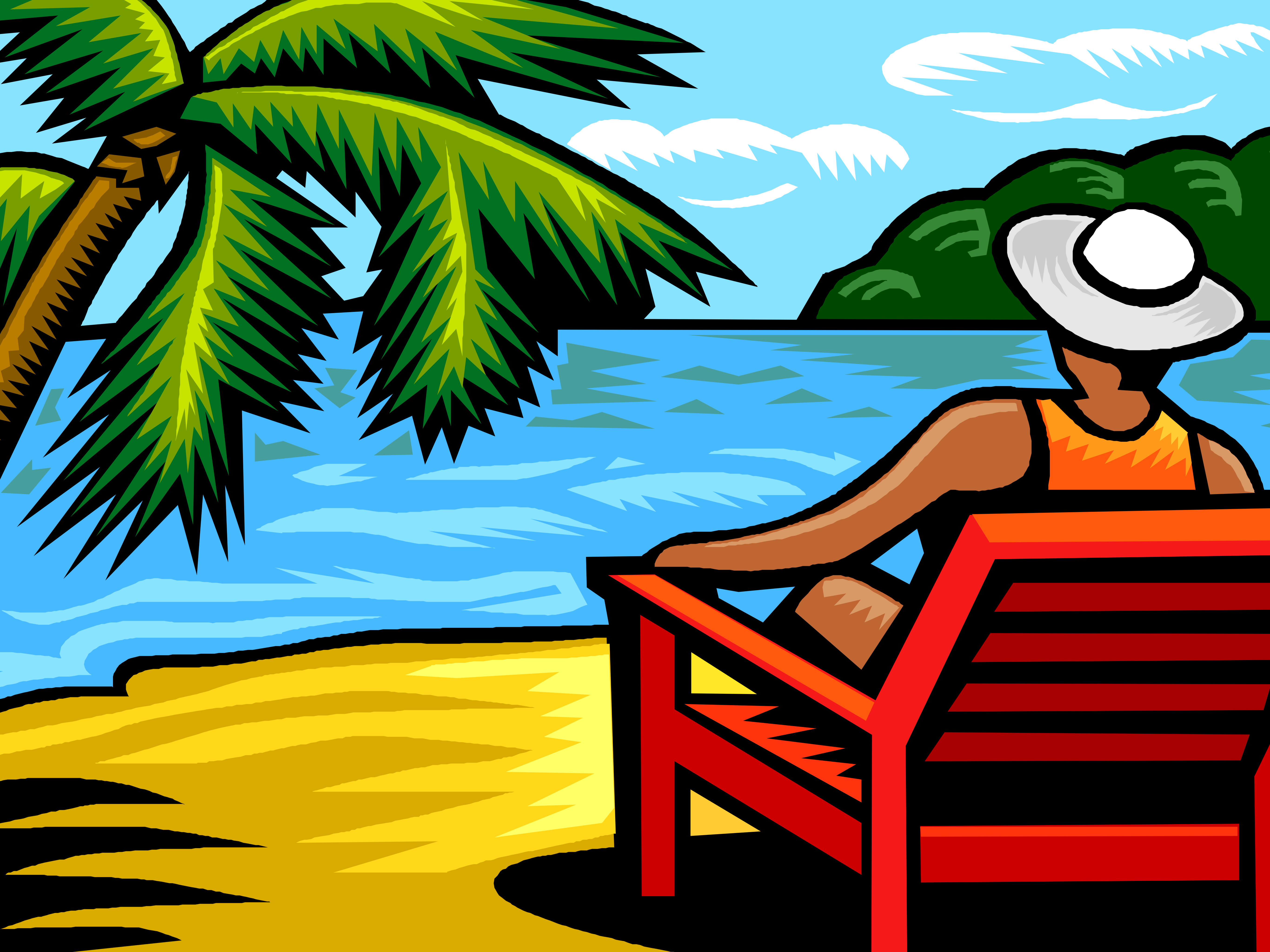
- The most important use is as an array of characters (a string)
- Strings end with a special "null character", written as '\0'

Menu with Price

```
int DRINKS = 3;
char *drink_name[] = { "Water", "Soda", "Beer" };
int drink_price[] = { 100, 150, 250 };

void print_prices() {
    int x = 0;
    for (int i = 0; i < DRINKS; i++) {
        char buffer[100];

        lcd.setCursor(x, 1);
        sprintf(buffer, "%d.%02d",
                drink_price[i] / 100,
                drink_price[i] % 100);
        lcd.print(buffer);
        x += strlen(drink_name[i]) + 1;
    }
}
```



Today's Topics

- Algorithms
- Search and Sort
- Complexity
- Sounds!

An Algorithm

- unambiguous instruction
  to solve a problem
- consists of finitely many, well defined steps
- typically implemented as computer programs

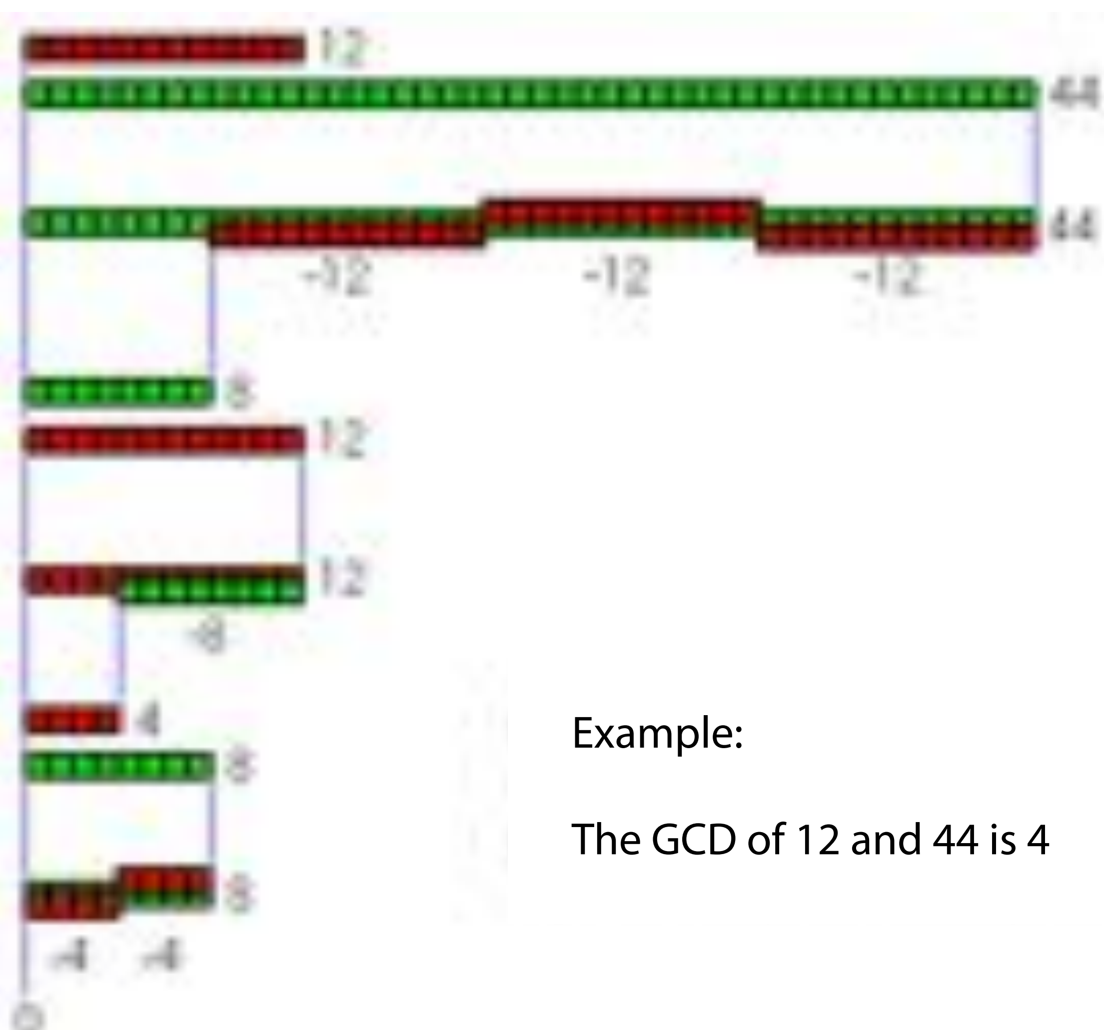
The First Algorithm

"However, if CD does not measure AB, and if one repeatedly and alternately deducts the smaller from the larger (of AB, CD), then finally a number must remain which measures the previous one."
(Euclid, Elements)

- computes the GCD
- Biggest integer by which AB and CD can be divided

Euclidean Algorithm

"However, if CD does not measure AB, and if one repeatedly and alternately deducts the smaller from the larger (of AB, CD), then finally a number must remain which measures the previous one."
(Euclid, Elements)



Example:

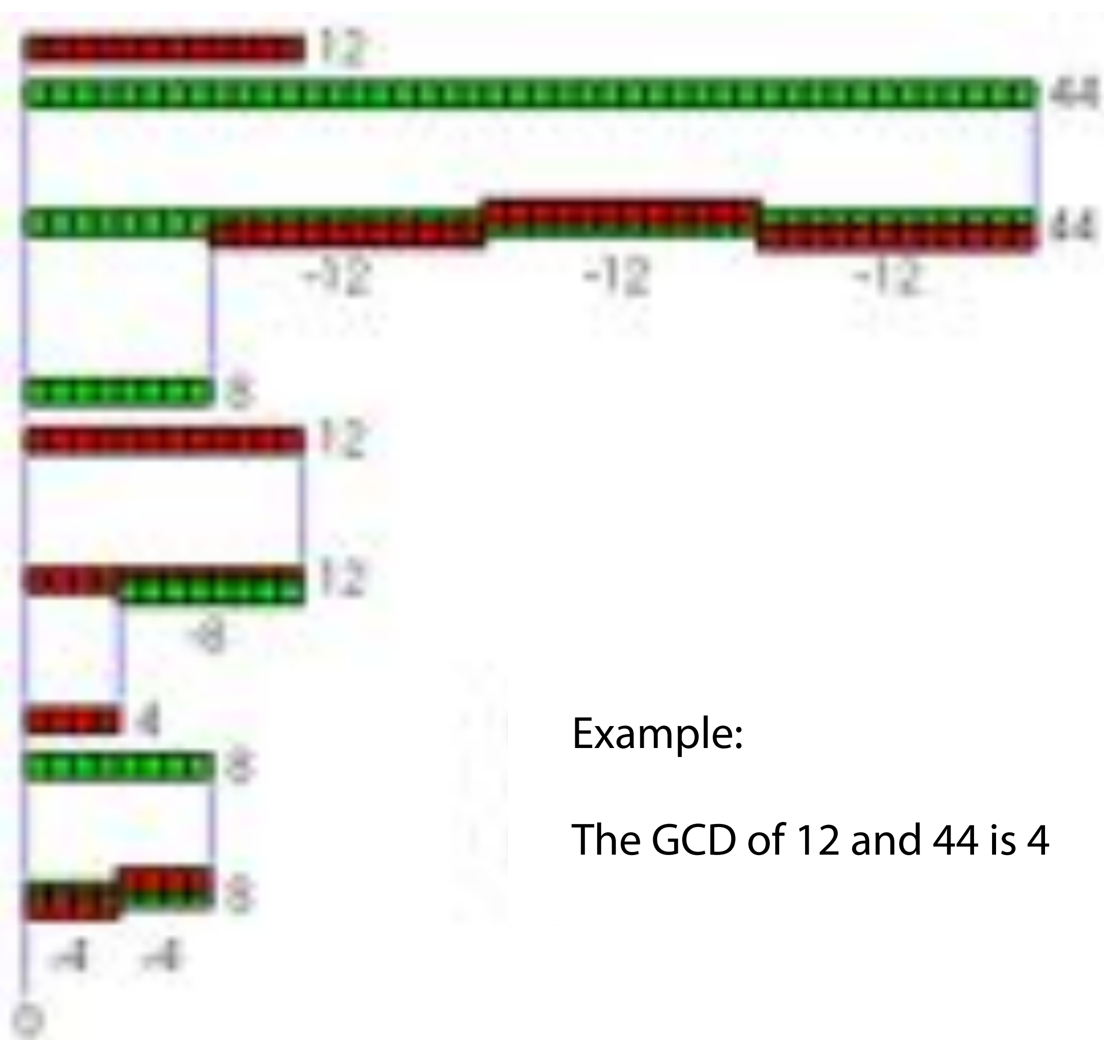
The GCD of 12 and 44 is 4

Euclidean Algorithm

```
int gcd(int a, int b) {
  if (a == 0)
    return b;

  while (b != 0) {
    if (a > b)
      a = a − b;
    else
      b = b − a;
  }

  return a;
}
```



Example:

The GCD of 12 and 44 is 4

أبو جعفر محمد بن موسى الخوارزمي

Muhammad ibn Musa al-Chwarizmi, * ~780 • †835–850

الكتاب المختصر في حساب الجبر والمقابلة

The concise Book about calculus

The First Algorithm Developed for Computers

- Computes *Bernoulli numbers*
  $B_0 = 1, B_1 = \pm 1/2, B_2 = 1/6, B_3 = 0, B_4 = -1/30, B_5 = 0, B_6 = 1/42, B_7 = 0, B_8 = -1/30.$

- Sequence of rational numbers that occur in mathematics in many contexts

- Example: Sum of powers

$$S_m(n) = \sum_{k=1}^{n} k^m = 1^m + 2^m + \cdots + n^m.$$

$$S_m(n) = \frac{1}{m+1} \sum_{k=0}^{m} \binom{m+1}{k} B_k \, n^{m+1-k},$$


Ada Lovelace
1815–1852

Analytical Engine

Charles Babbage, 1837

Alan Turing

Turing Machine

Halting Problem

- Not all problems can be solved by programs
- E.g. the halting problem states that there is no program which can decide for an arbitrary program *P*, whether it will (eventually) return a result or not.

Collatz Conjecture

(Wolfgang Collatz, 1937)

- Start with an integer *n*
- If *n* is even, take *n*/2 next
- If *n* is odd, take 3*n*+1next
- repeat

19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26,

13, 40, 20, 10, 5, 16, 8, 4, 2, 1, ...

Collatz Conjecture

(Wolfgang Collatz, 1937)

- Apparently every sequence defined in this manner ends in 4, 2, 1, …
- This property remains unproven

19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26,

13, 40, 20, 10, 5, 16, 8, 4, 2, 1, ...

Halting Problem

```
void collatz(int n) {
  while (n != 1) {
    if (n % 2 == 0)
      n = n / 2;
    else
      n = 3 * n + 1;
  }
}
```

- Will collatz() return for every n?
- Solution only by trial (in infinite time)

It is impossible to show correctness automatically for all programs

Halting Problem

To show that a real program fulfils its requirements, we must either

- use mathematical knowledge and assumptions to prove it by hand (which is very hard), or
- we must test it and hope that our tests suffice.

Algorithms

Calculate

- Fibonacci
- GCD
- Collatz

Search

- Linear
- Binary

Sort

- Insertion
- Merge

Algorithms

Calculate

- Fibonacci
- GCD
- Collatz

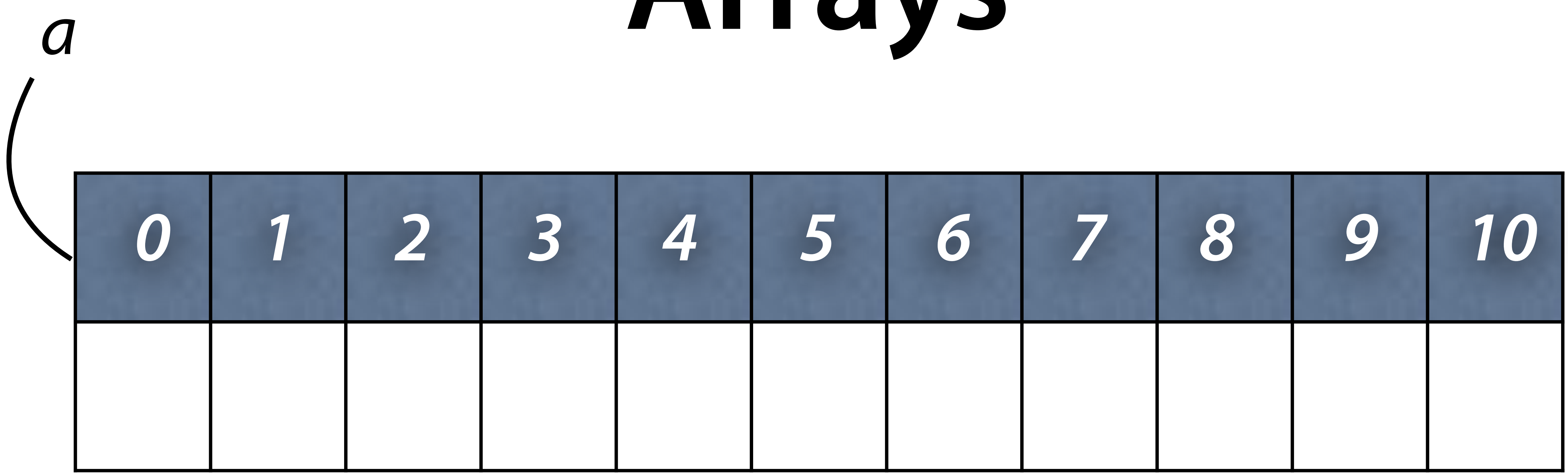
Search

- Linear
- Binary

Sort

- Insertion
- Merge

Arrays




```
int a[11];    // 11 Elements
```

Type of the elements

Name of the array

Size of the array

Search

a

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
10	−10	7	2	2	−4	−7	−10	1	4	5

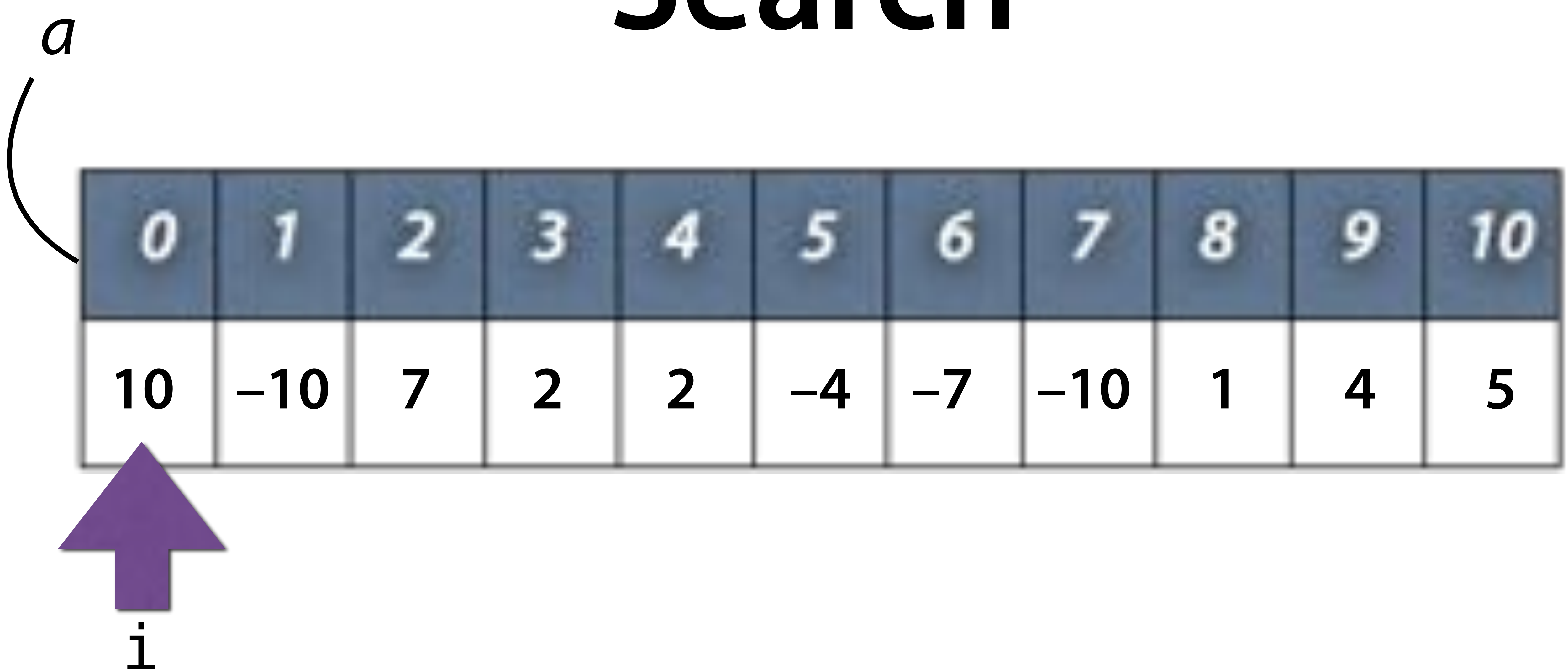
```
// If x is in a[0..size], return index,
// such that x == a[index];
// otherwise < 0
int find(int a[], int size, int x) {
    // what happens here?
}
```

Demo

Search

```
// If x is in a[0..size], return index,
// such that x == a[index];
// otherwise < 0
int find(int a[], int size, int x) {
    for (int i = 0; i < size; i++) {
        if (x == a[i])
            return i;
    }
    return -1;
}
```

Search

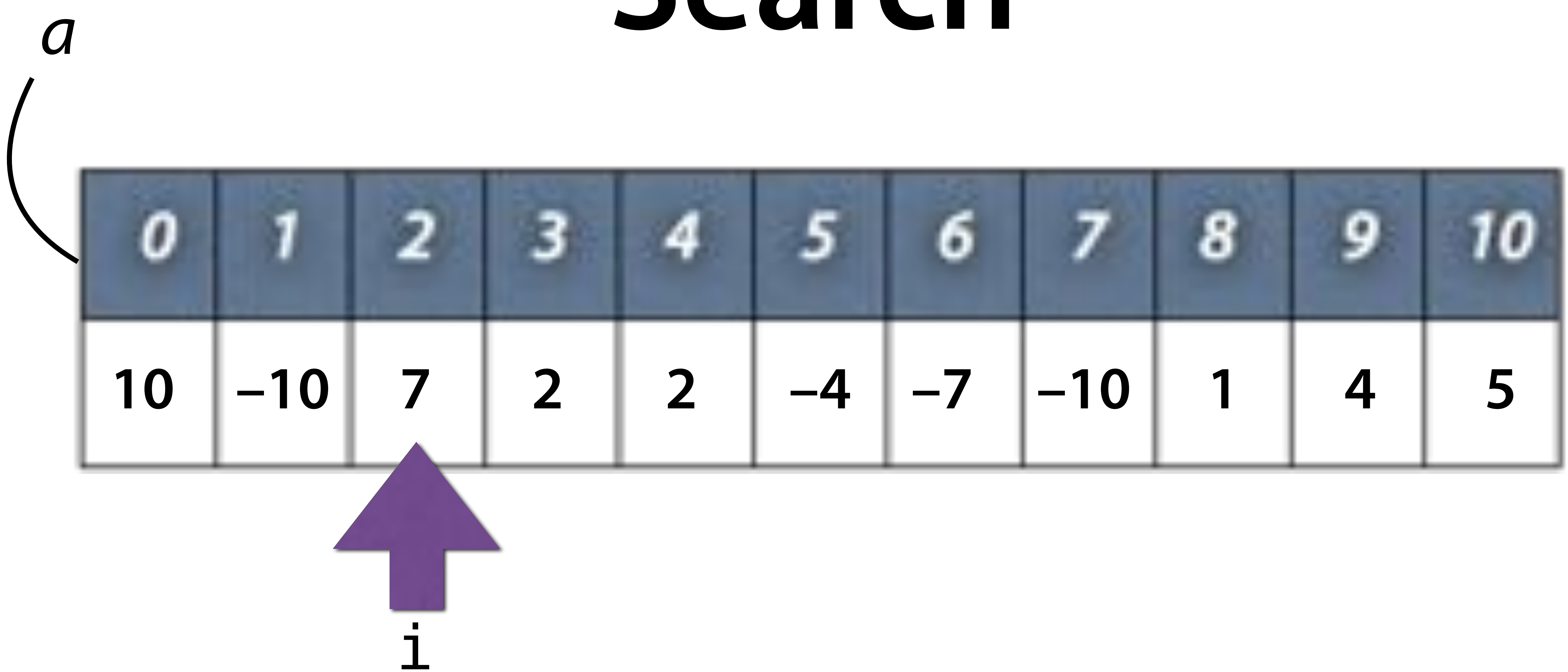



```
int index = find(a, 11, -4);
```

Search

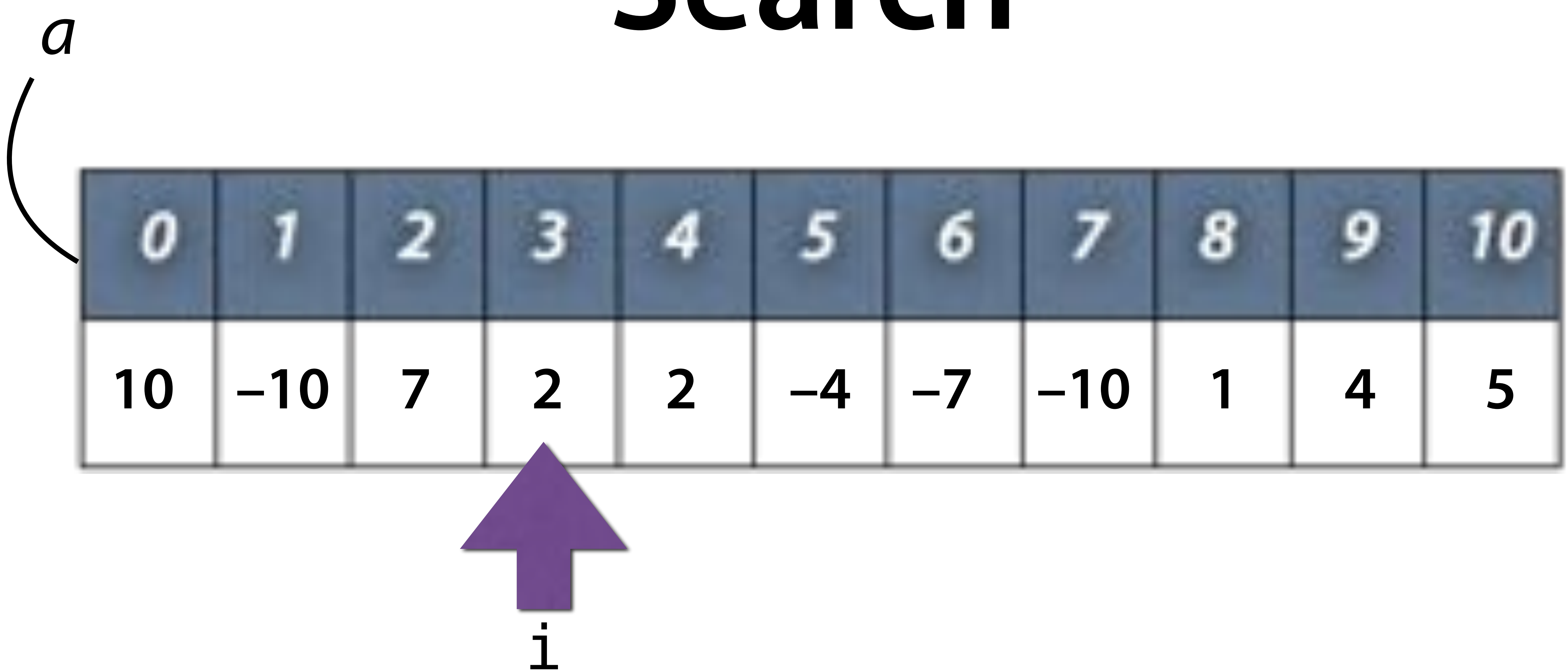

```
int index = find(a, 11, −4);
```

Search



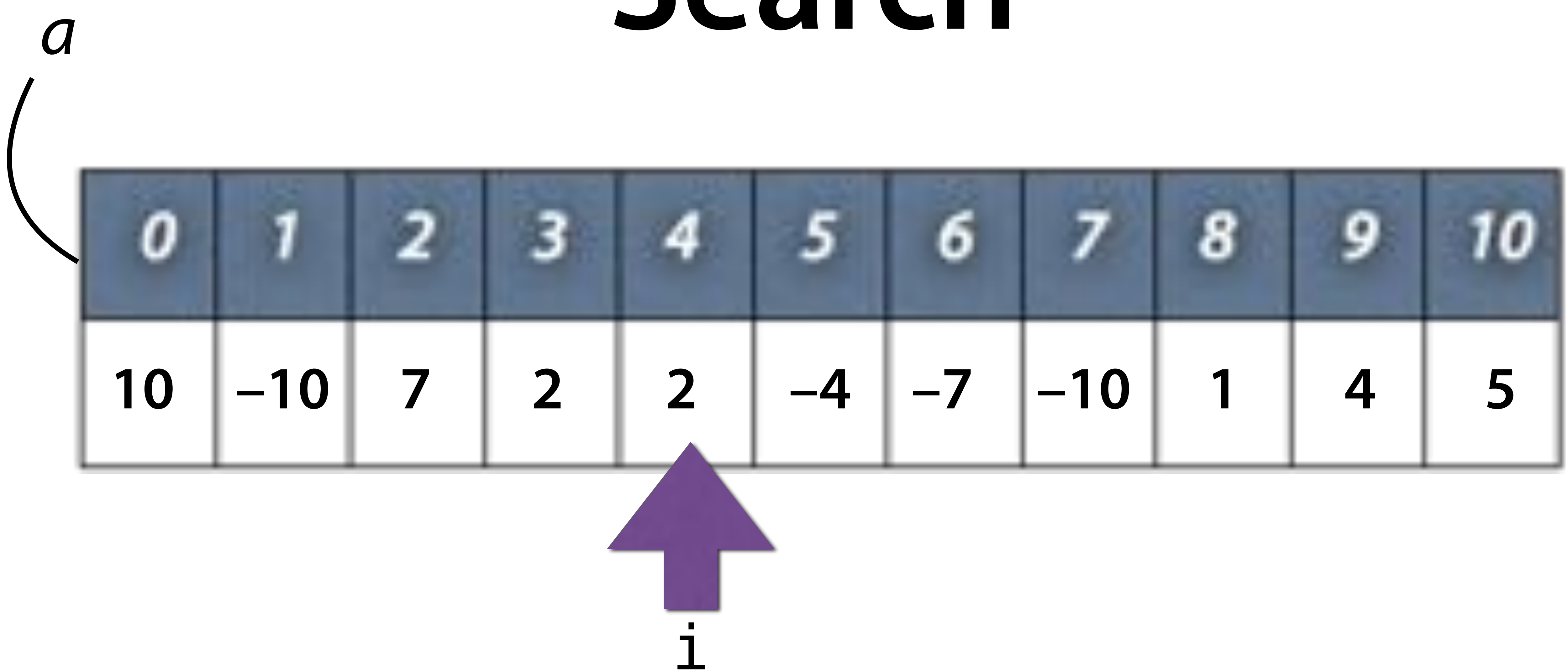

```
int index = find(a, 11, −4);
```

Search



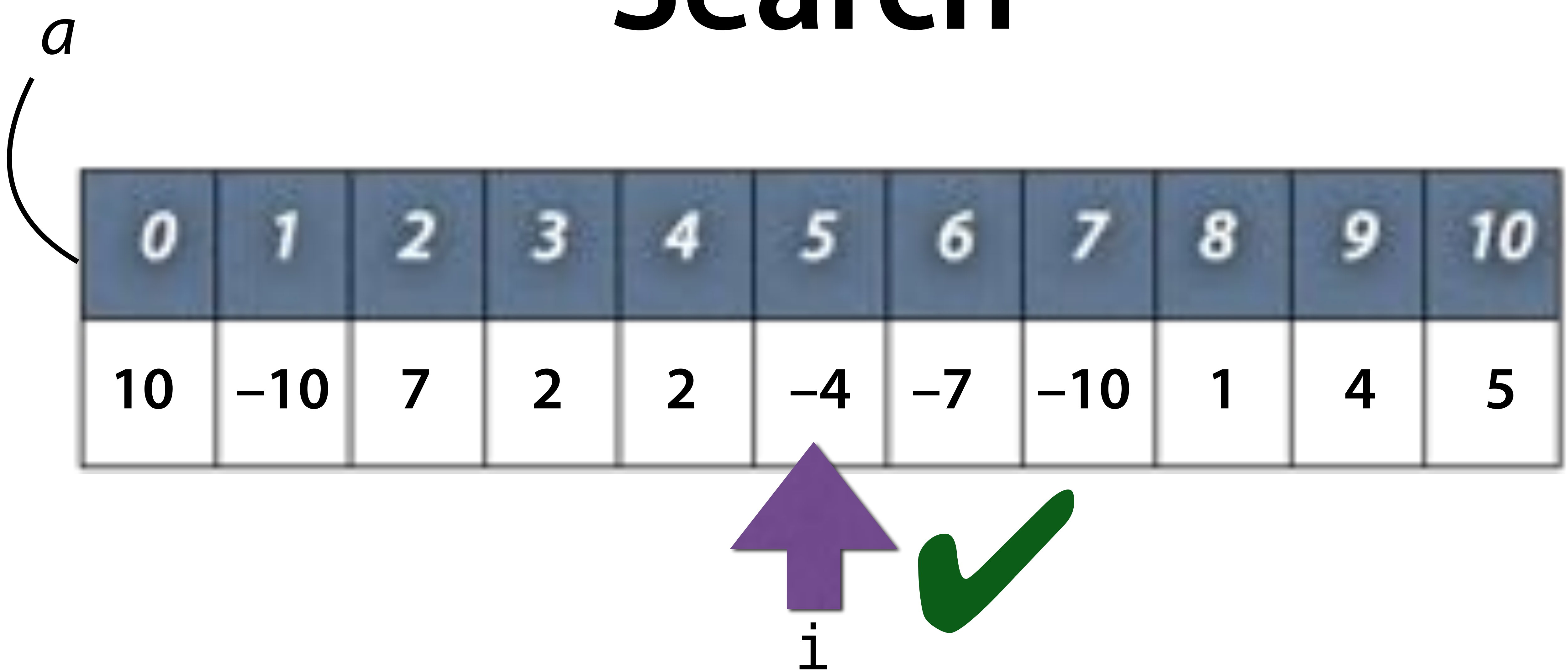

```
int index = find(a, 11, -4);
```

Search




```
int index = find(a, 11, -4);
```

Search




```
int index = find(a, 11, −4);
```

Complexity

```
// If x is in a[0..size], return index,
// such that x == a[index];
// otherwise < 0
int find(int a[], int size, int x) {
    for (int i = 0; i < size; i++) {
        if (x == a[i])
            return i;
    }
    return -1;
}
```

- How many comparisons does find() need?

Complexity

Having *n* elements:

- Linear search: *n/2* comparisons
- What to do when we have millions of data points?

Search

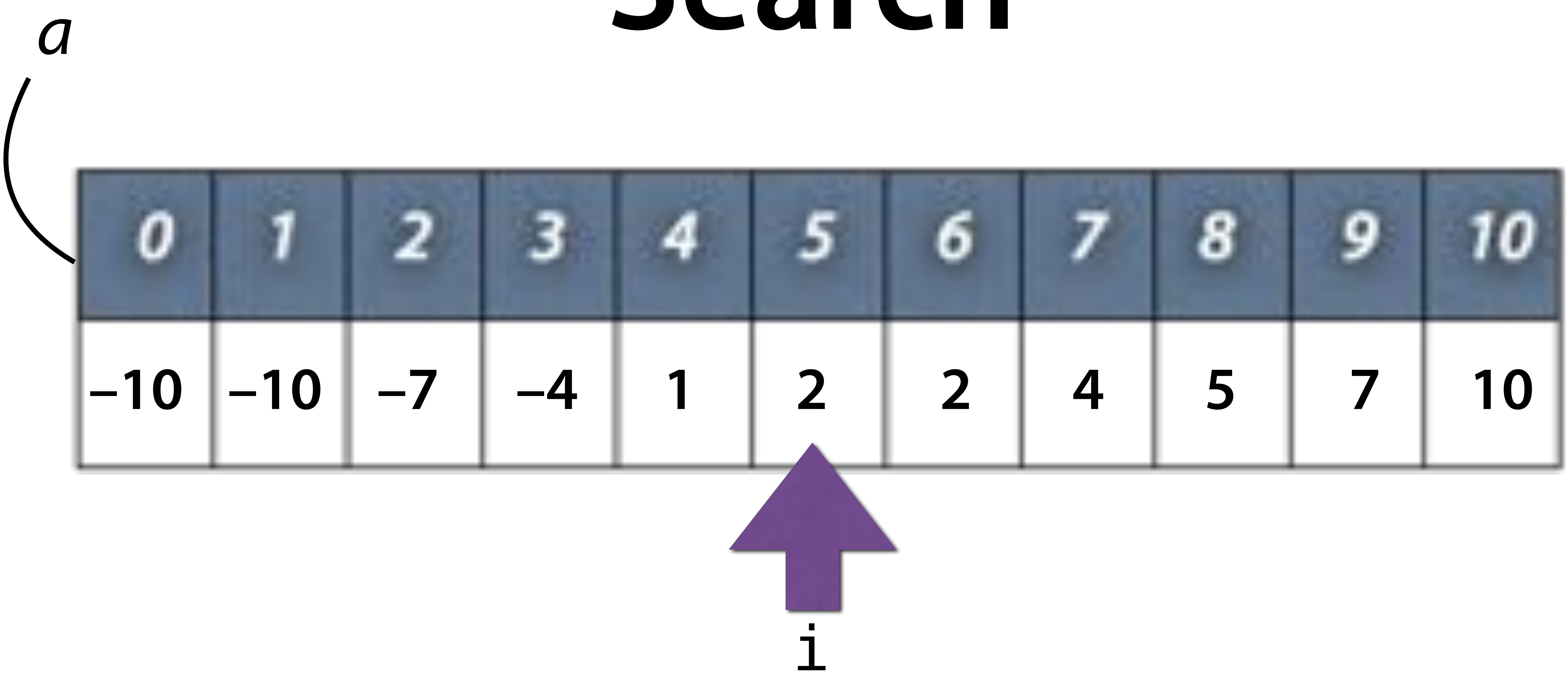
a

0	1	2	3	4	5	6	7	8	9	10
10	–10	7	2	2	–4	–7	–10	1	4	5

Search

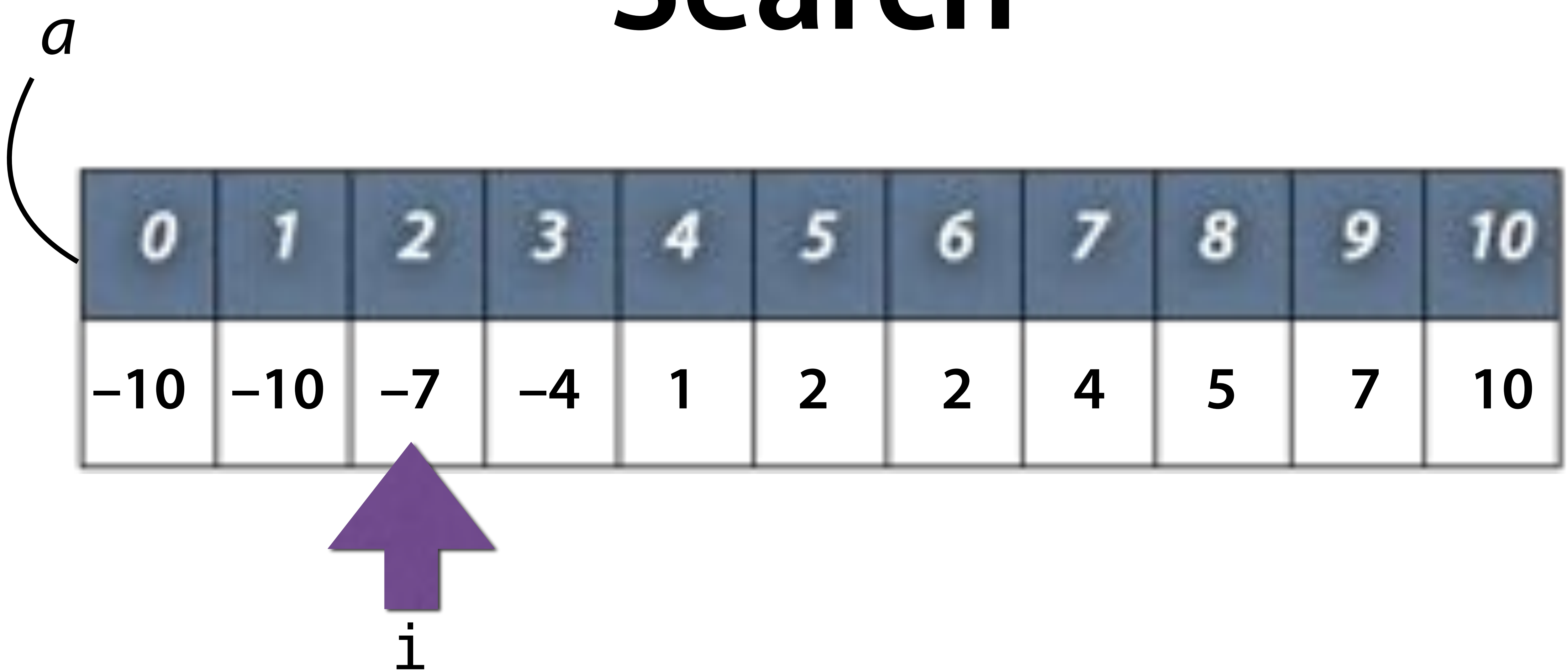

```
// If x is in a[0..size], return index,
// such that x == a[index];
// otherwise < 0
int sorted_find(int a[], int size, int x) {
   // Would this be faster?
}
```

Search



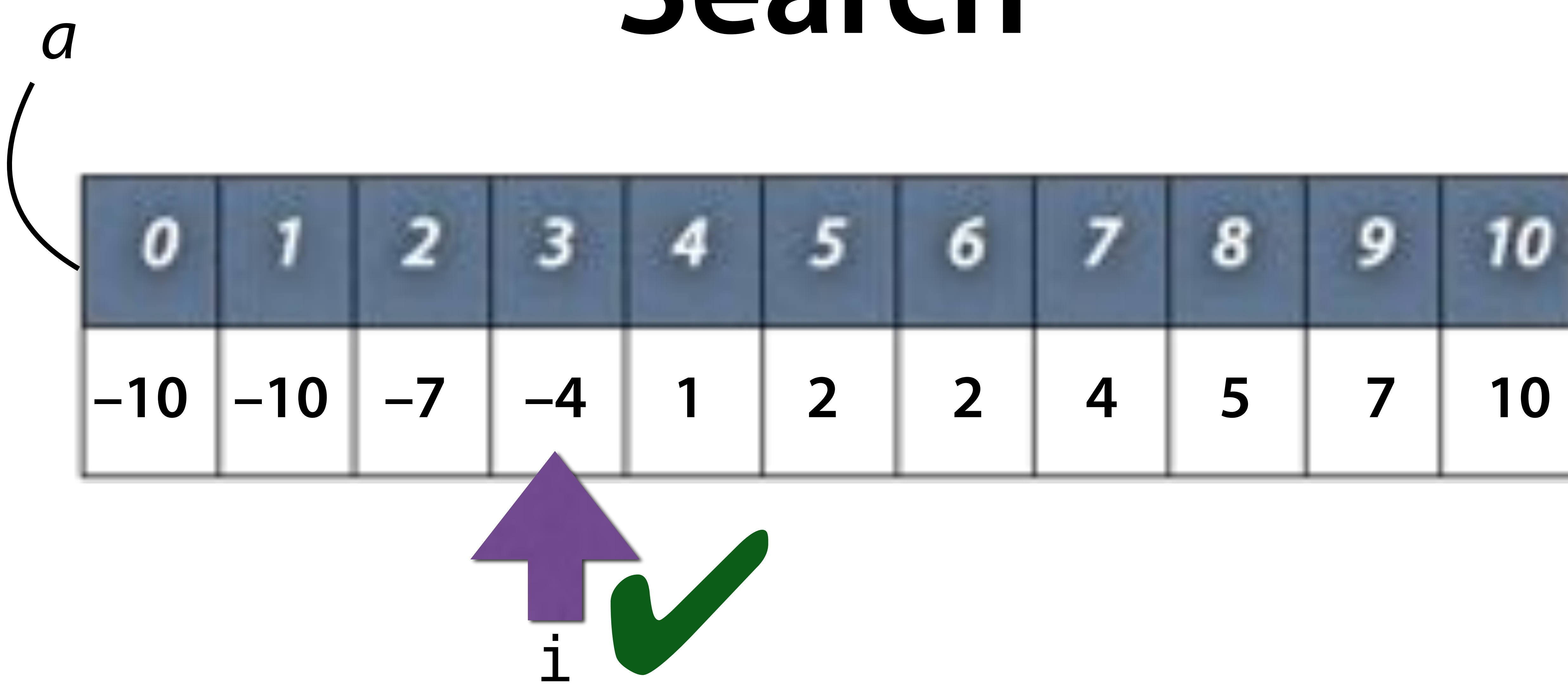

```
int index = sorted_find(a, 11, −4);
```

Search




```
int index = sorted_find(a, 11, −4);
```

Search

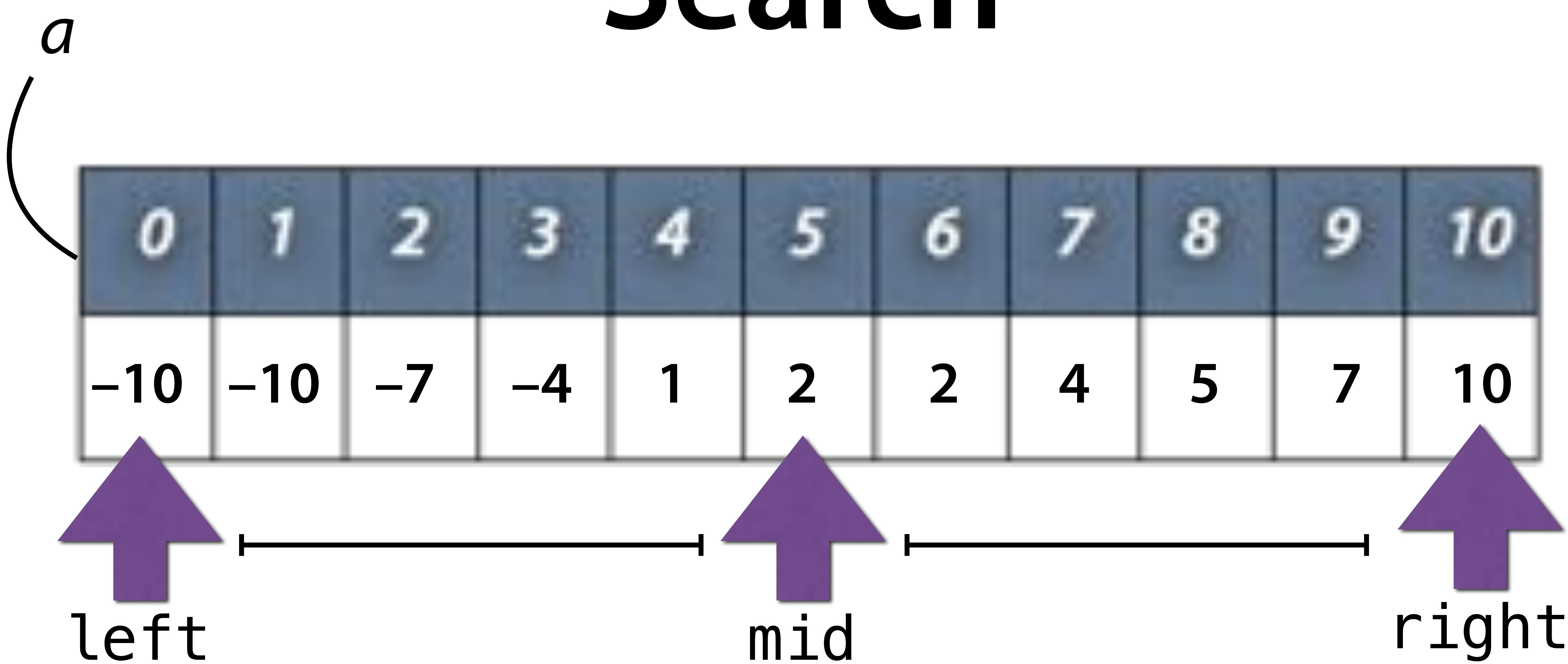



```
int index = sorted_find(a, 11, -4);
```

The Plan

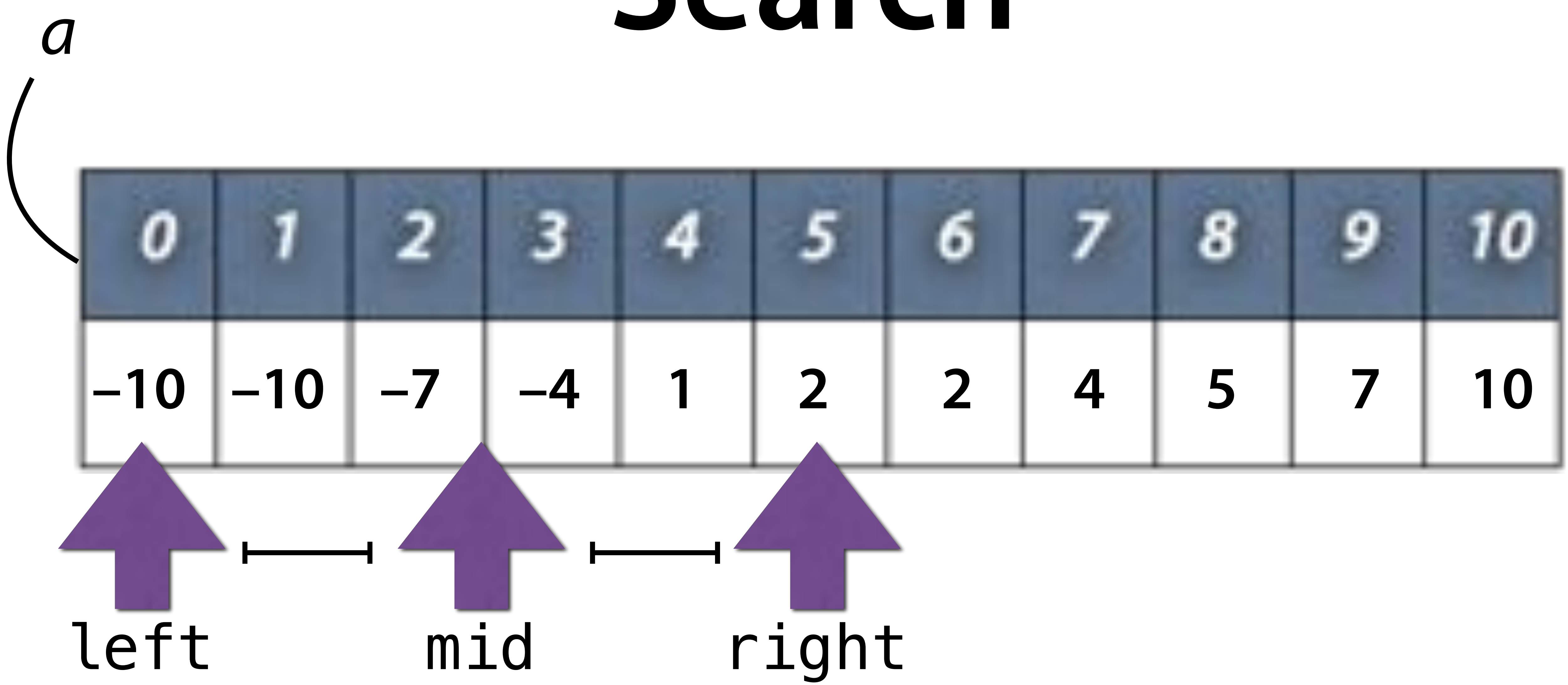
- Remember the i*nterval* in which to search
- Look at the element at the center of the interval
- If it is larger than the sought after element, continue the search in the left half
- Otherwise continue the search in the right half

Search



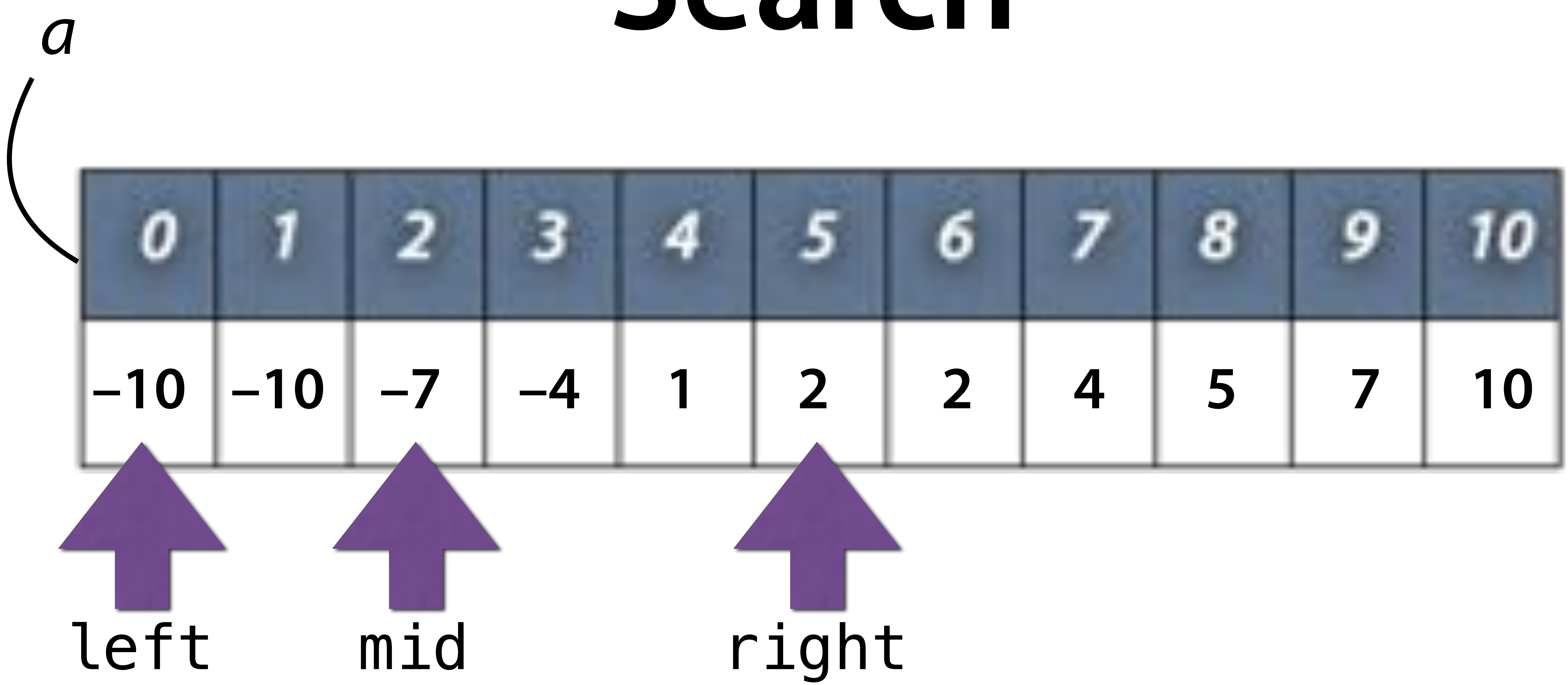

```
int index = sorted_find(a, 11, −4);
```

Search



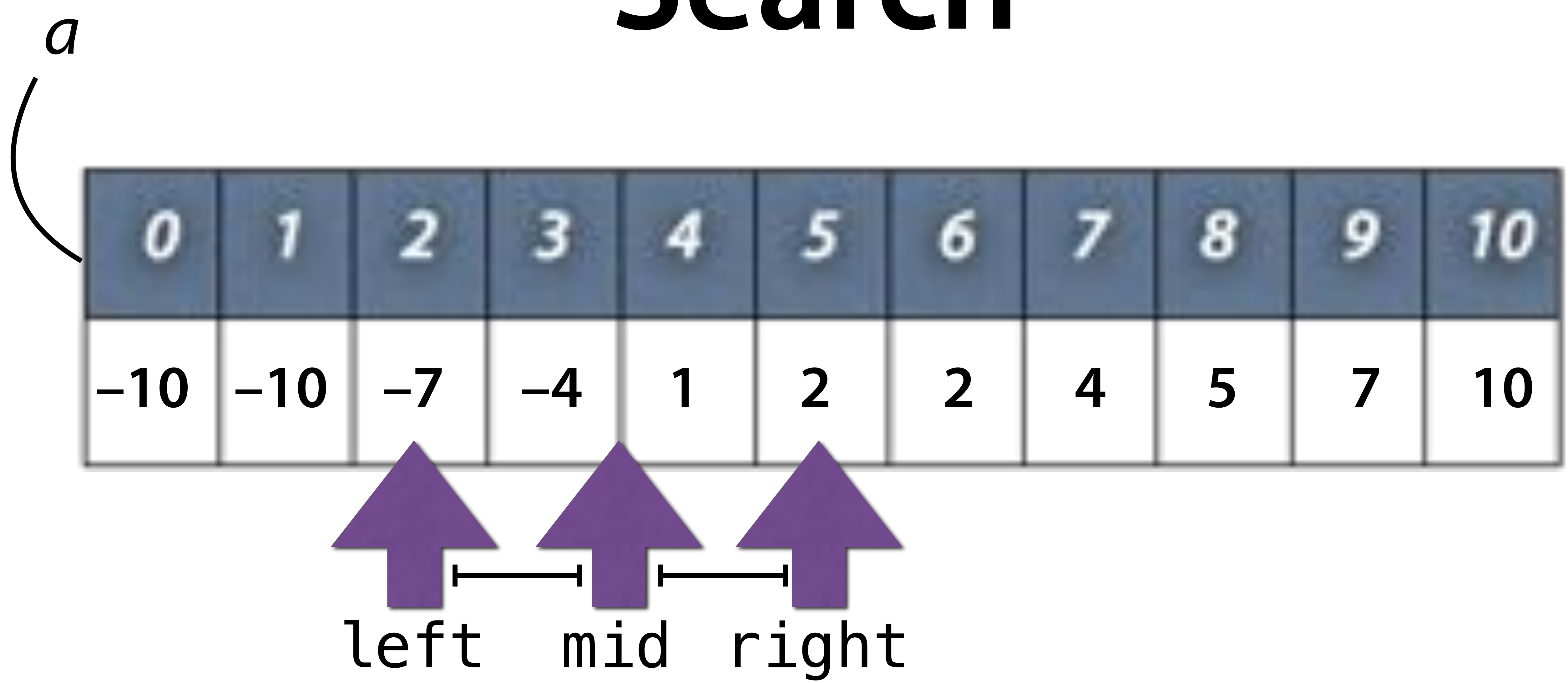

```
int index = sorted_find(a, 11, -4);
```

Search



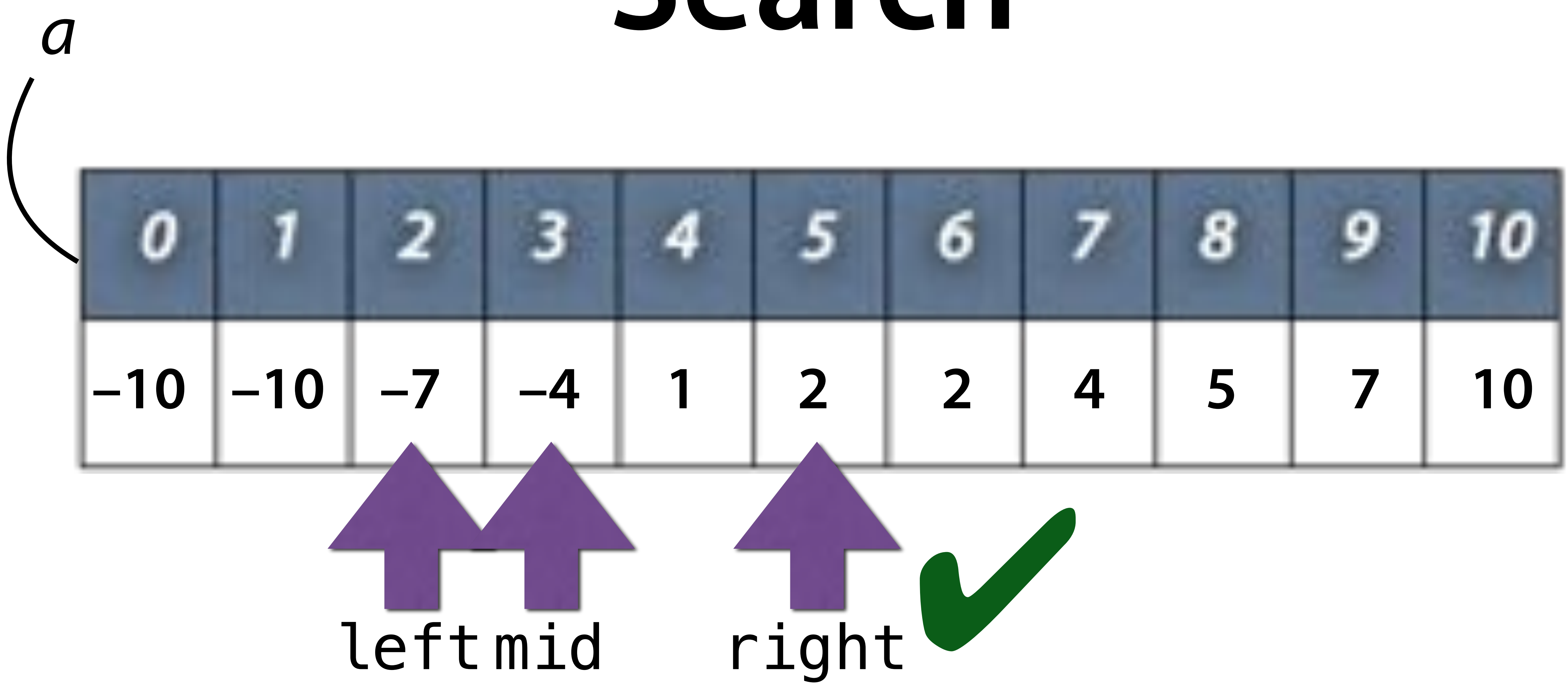

```
int index = sorted_find(a, 11, -4);
```

Search




```
int index = sorted_find(a, 11, −4);
```

Search




```
int index = sorted_find(a, 11, −4);
```

- Wie viele Vergleiche braucht sorted_find()?

Demo

Binary Search

```
int sorted_find(int a[], int size, int x) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + ((right - left) / 2);
        if (x == a[mid])
            return mid;
        else if (x < a[mid])
            right = mid - 1;
        else // (x > a[mid])
            left  = mid + 1;
    }
    return -1;
}
```

Complexity Compared

Having *n* elements:

- Linear search: *n/2* comparisons
- Binary Search: $\log_2 n$ comparisons

Server Farm

Algorithms

Calculate

- Fibonacci
- GCD
- Collatz

Search

- Linear
- Binary

Sort

- Insertion
- Merge

Algorithms

Calculate

- Fibonacci
- GCD
- Collatz

Search

- Linear
- Binary

Sort

- Insertion
- Merge

Sort

a

0	1	2	3	4	5	6	7	8	9	10
10	–10	7	2	2	–4	–7	–10	1	4	5

Sort

a

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	–4	1	2	2	4	5	7	10

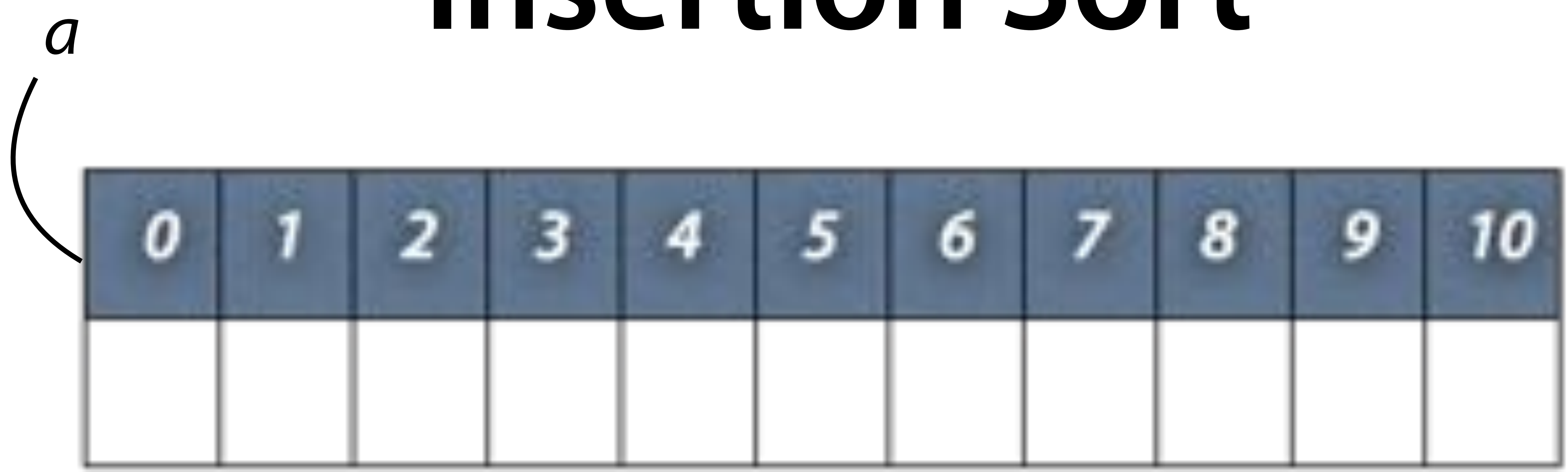
How do I sort an array?

Insertion Sort

a

0	1	2	3	4	5	6	7	8	9	10
10	–10	7	2	2	–4	–7	–10	1	4	5

Insertion Sort



4

–10

–7

10

–4

–10

7

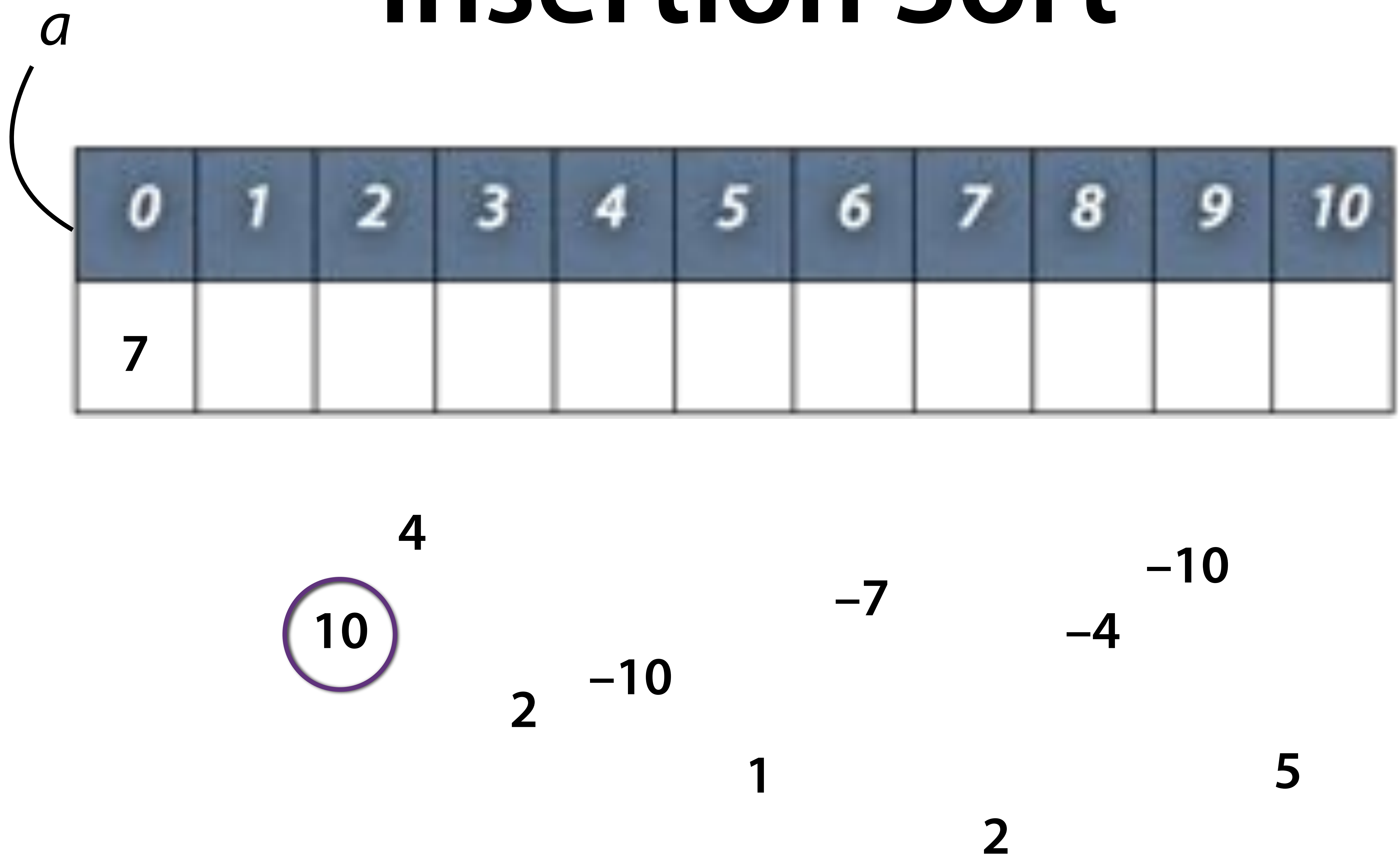
2

1

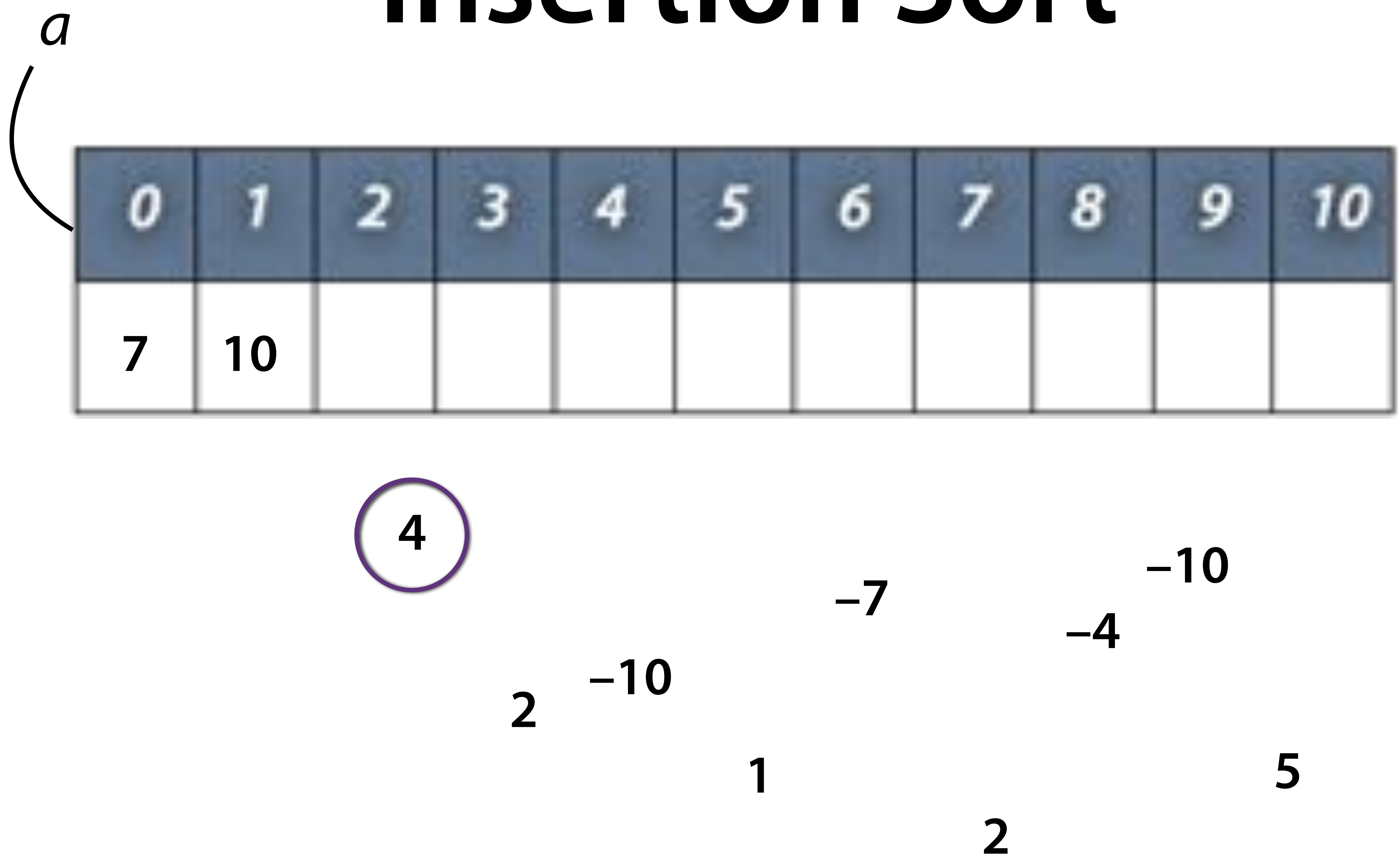
5

2

Insertion Sort



Insertion Sort



Insertion Sort

a

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

4

–10

–7

–4

–10

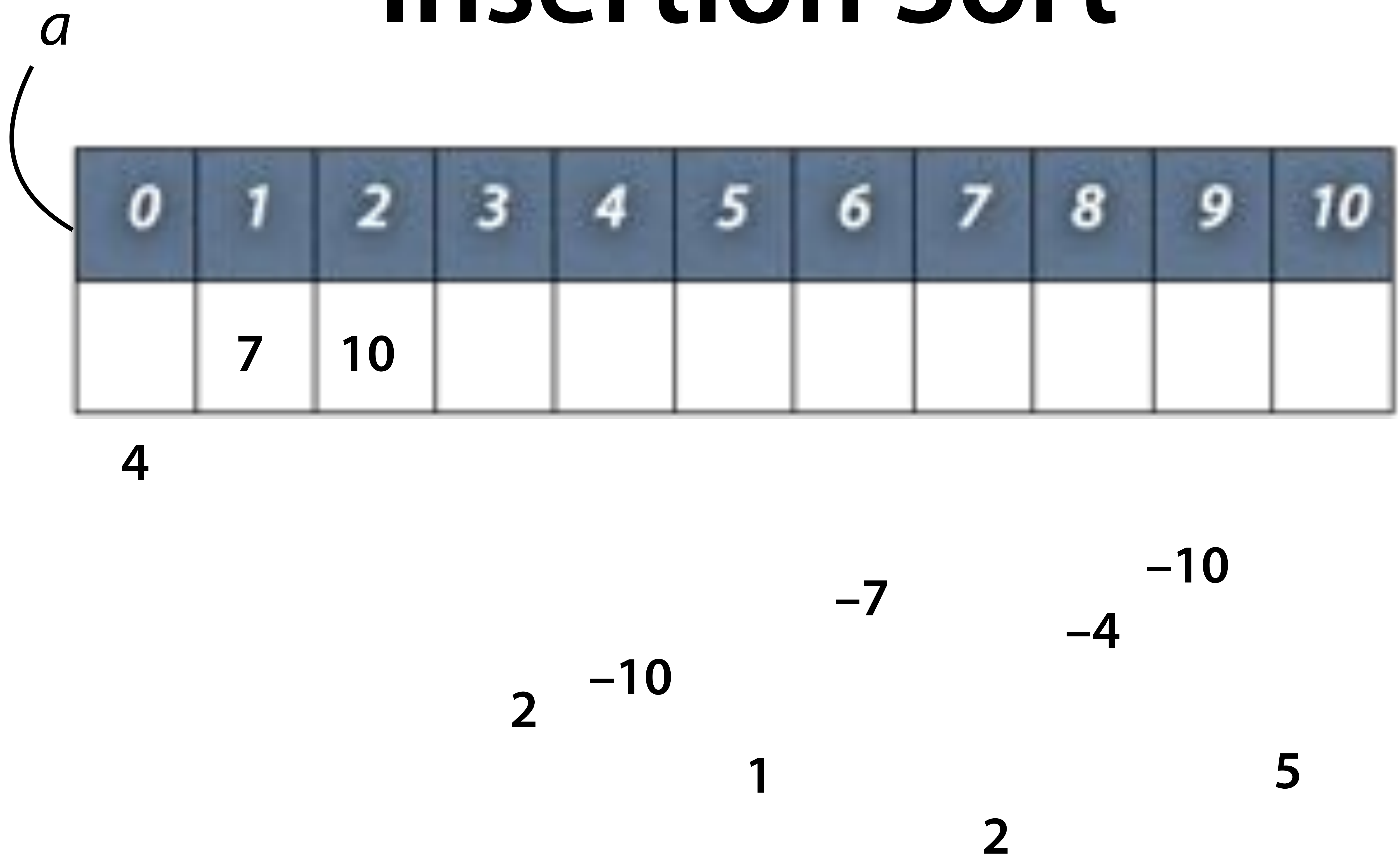
2

5

1

2

Insertion Sort



Insertion Sort

a

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

–10

–7

–4

2 –10

1

5

2

Insertion Sort

a

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

–10

–7

–4

–10

1

5

2

Insertion Sort

a

0	1	2	3	4	5	6	7	8	9	10
–10	2	4	7	10						

–10

–7

–4

1

5

2

Insertion Sort

a

0	1	2	3	4	5	6	7	8	9	10
–10	–7	2	4	7	10					

–10

–4

1

5

2

Insertion Sort

a

0	1	2	3	4	5	6	7	8	9	10
–10	–7	1	2	4	7	10				

–10

–4

5

2

Insertion Sort

a

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	1	2	4	7	10			

–4

5

2

Insertion Sort

a

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	–4	1	2	4	7	10		

5

2

Insertion Sort

a

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	–4	1	2	4	5	7	10	

2

Insertion Sort

a

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	–4	1	2	2	4	5	7	10

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
10	–10	7	2	2	–4	–7	–10	1	4	5

- We want to sort inside of an array
- Assume: array a[0…i–1] is already sorted
- We inspect the element a[i]…
- …and insert it into the sorted array

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
10	–10	7	2	2	–4	–7	–10	1	4	5

already
sorted

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
–10	10	7	2	2	–4	–7	–10	1	4	5

already
sorted

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
–10	7	10	2	2	–4	–7	–10	1	4	5

already
sorted

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
–10	7	2	10	2	–4	–7	–10	1	4	5

already
sorted

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
–10	2	7	10	2	–4	–7	–10	1	4	5

already sorted

- In order to insert, we swap until the element reaches the correct position

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
–10	2	7	2	10	–4	–7	–10	1	4	5

already sorted

- In order to insert, we swap until the element reaches the correct position

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
–10	2	2	7	10	–4	–7	–10	1	4	5

already sorted

- In order to insert, we swap until the element reaches the correct position

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
–10	2	2	7	–4	10	–7	–10	1	4	5

already sorted

- In order to insert, we swap until the element reaches the correct position

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
–10	2	2	–4	7	10	–7	–10	1	4	5

already sorted

- In order to insert, we swap until the element reaches the correct position

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
–10	2	–4	2	7	10	–7	–10	1	4	5

already sorted

- In order to insert, we swap until the element reaches the correct position

Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
–10	–4	2	2	7	10	–7	–10	1	4	5

already sorted

- In order to insert, we swap until the element reaches the correct position

Demo

Insertion Sort

```
void insertion_sort(int a[], int size) {
    for (int i = 1; i < size; i++) {
        int j = i;
        while (j > 0 && a[j - 1] > a[j]) {
            // Swap a[j] and a[j - 1]
            int tmp = a[j];
            a[j] = a[j - 1];
            a[j - 1] = tmp;
            j--;
        }
    }
}
```

- How many comparisons does insertion_sort() need?

Complexity

Having n elements:

- Insertion sort: n^2 comparisons

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
10	–10	7	2	2	–4	–7	–10	1	4	5

Merge Sort

0	1	2	3	4	5
10	–10	7	2	2	–4

6	7	8	9	10
–7	–10	1	4	5

Merge Sort

0	1	2	3	4	5
10	–10	7	2	2	–4

6	7	8	9	10
–7	–10	1	4	5

Merge Sort

0	1	2	3	4	5
−10	−4	2	2	7	10

6	7	8	9	10
−10	−7	1	4	5

Merge Sort

0	1	2	3	4	5	6	7	8	9	10

0	1	2	3	4	5
–10	–4	2	2	7	10

6	7	8	9	10
–10	–7	1	4	5

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
–10										

0	1	2	3	4	5
	–4	2	2	7	10

6	7	8	9	10
–10	–7	1	4	5

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
–10	–10									

0	1	2	3	4	5
	–4	2	2	7	10

6	7	8	9	10
	–7	1	4	5

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7								

0	1	2	3	4	5
	–4	2	2	7	10

6	7	8	9	10
		1	4	5

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	–4							

0	1	2	3	4	5
		2	2	7	10

6	7	8	9	10
		1	4	5

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	–4	1						

0	1	2	3	4	5
		2	2	7	10

6	7	8	9	10
			4	5

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	–4	1	2					

0	1	2	3	4	5
			2	7	10

6	7	8	9	10
			4	5

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	–4	1	2	2				

0	1	2	3	4	5
				7	10

6	7	8	9	10
			4	5

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	–4	1	2	2	4			

0	1	2	3	4	5
				7	10

6	7	8	9	10
				5

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
–10	–10	–7	–4	1	2	2	4	5		

0	1	2	3	4	5
				7	10

6	7	8	9	10

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
−10	−10	−7	−4	1	2	2	4	5	7	

0	1	2	3	4	5
					10

6	7	8	9	10

Merge Sort

0	1	2	3	4	5	6	7	8	9	10
−10	−10	−7	−4	1	2	2	4	5	7	10

0	1	2	3	4	5

6	7	8	9	10

Merge Sort

0	1	2	3	4	5
10	–10	7	2	2	–4

6	7	8	9	10
–7	–10	1	4	5

Merge Sort

How do I sort partial arrays?

0	1	2	3	4	5
–10	–4	2	2	7	10

6	7	8	9	10
–10	–7	1	4	5

Sorting Recursively

6 5 3 1 8 7 2 4

- Basic idea: apply merge_sort() recursively to the partial arrays

John von Neumann

Demo

Call

```
// Sort a[0..size], using b[0..size] as a buffer
void merge_sort(int a[], int b[], int size)
{
    partial_merge_sort(a, b, 0, size);
}
```

- We use *b[]* as an auxiliary array
- Must have the same size as *a*[]

Sort

```
// Sort a[begin..end], using b[begin..end] as buffer
void partial_merge_sort(int a[], int b[],
                        int begin, int end)
{
    if (end - begin < 2)
        return;

    // Split and sort
    int mid = begin + (end - begin) / 2;
    partial_merge_sort(a, b, begin, mid);
    partial_merge_sort(a, b, mid, end);

    // Merge and copy
    merge(a, b, begin, mid, end);
    copy(a, b, begin, end);
}
```

Merge

```
// Merge a[begin..mid - 1] and a[mid..end] into b[begin..end]
void merge(int a[], int b[], int begin, int mid, int end) {
    int i_begin = begin;
    int i_mid   = mid;
    for (int j = begin; j < end; j++)
        if (i_begin < mid &&
            (i_mid >= end || a[i_begin] <= a[i_mid]))
            b[j] = a[i_begin++];
        else
            b[j] = a[i_mid++];
}
```

- P && Q only evaluates Q if P is true
- P || Q only evaluates Q if P is false
- Protects the array borders from access

Copy

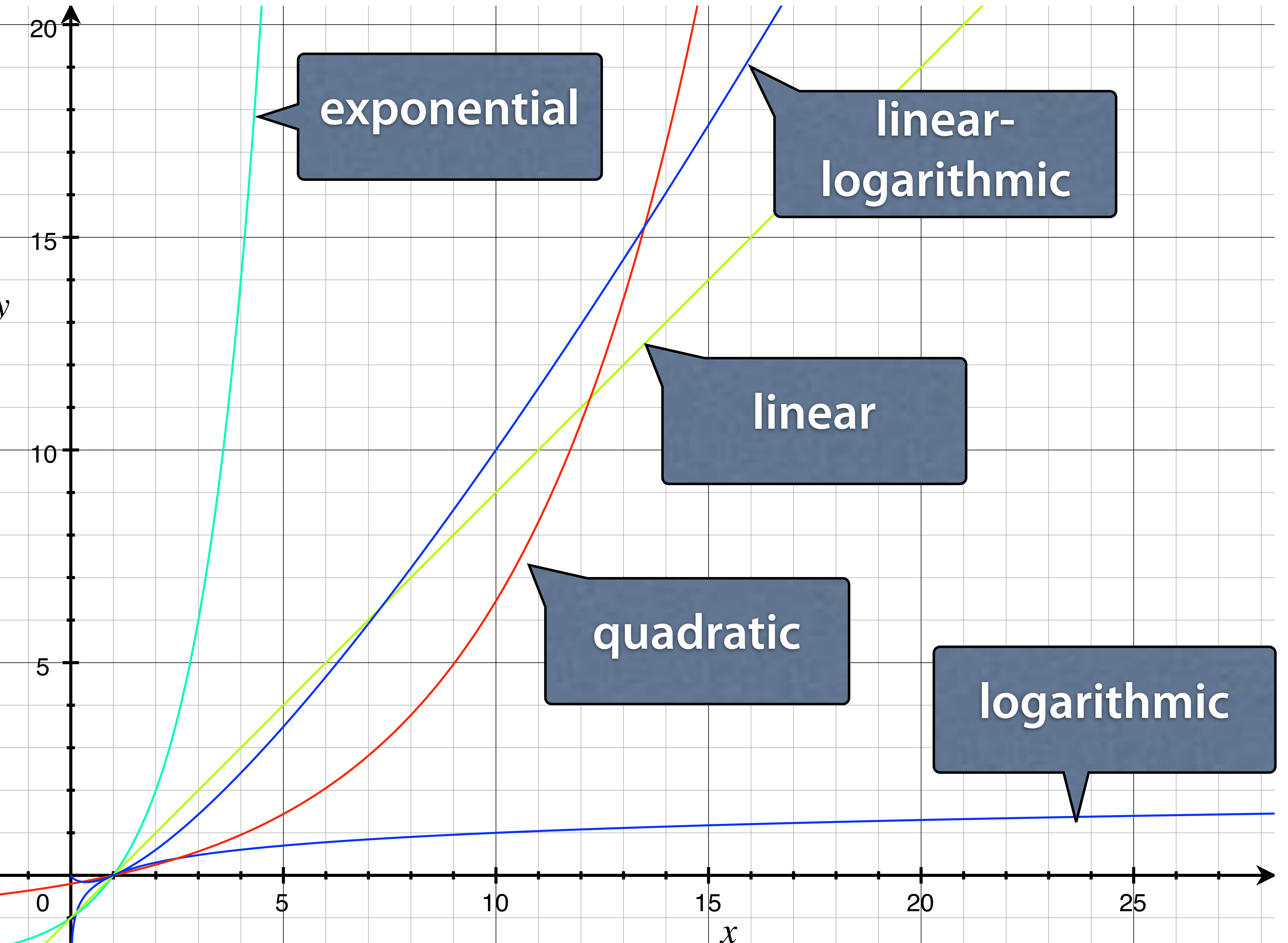
```
// Copy b[begin..end] into a[begin..end]
void copy(int a[], int b[], int begin, int end) {
    for (int k = begin; k < end; k++)
        a[k] = b[k];
}
```

- How many comparisons does merge_sort() need?

Complexity Compared

Having n elements:

- Insertion sort: n^2 comparisons
- Merge Sort: $n \log_2 n$ comparisons


exponential
linear-logarithmic
linear
quadratic
logarithmic
20
15
10
5
0
5
10
15
20
25
x
y

Algorithms

Calculate

- Fibonacci
- GCD
- Collatz

Search

- Linear
- Binary

Sort

- Insertion
- Merge

Algorithms

Calculate

- Fibonacci
- GCD
- Collatz

Search

- Linear
- Binary

Sort

- Insertion
- Merge

Ultrasound!

- HC-SR04 combines an ultrasound-speaker and a microphone
- Can measure signal delays

Buzzer

- Simple
  Piezo Buzzer
- Can output
  sounds in arbitrary
  frequencies

Theremin

+

Theremin

An Algorithm

- unambiguous instruction to solve a problem
- consists of finitely many, well defined steps
- typically implemented as computer programs

Algorithms

Calculate	Search	Sort
• Fibonacci • GCD • Collatz	• Linear • Binary	• Insertion • Merge

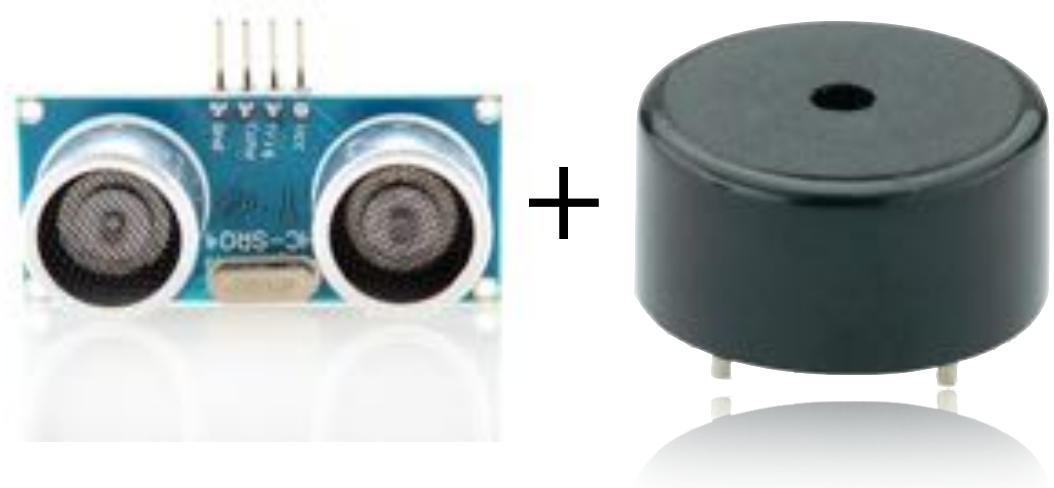
Sort In-Place

a

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

- We want to sort inside of an array
- Assume: array a[0....i-1] is already sorted
- We inspect the element a[i]...
- ...and insert it into the sorted array

Theremin =



Handouts

Halting Problem

```
void collatz(int n) {
  while (n != 1) {
    if (n % 2 == 0)
      n = n / 2;
    else
      n = 3 * n + 1;
  }
}
```

- Will collatz() return for every n?
- Solution only by trial (in infinite time)

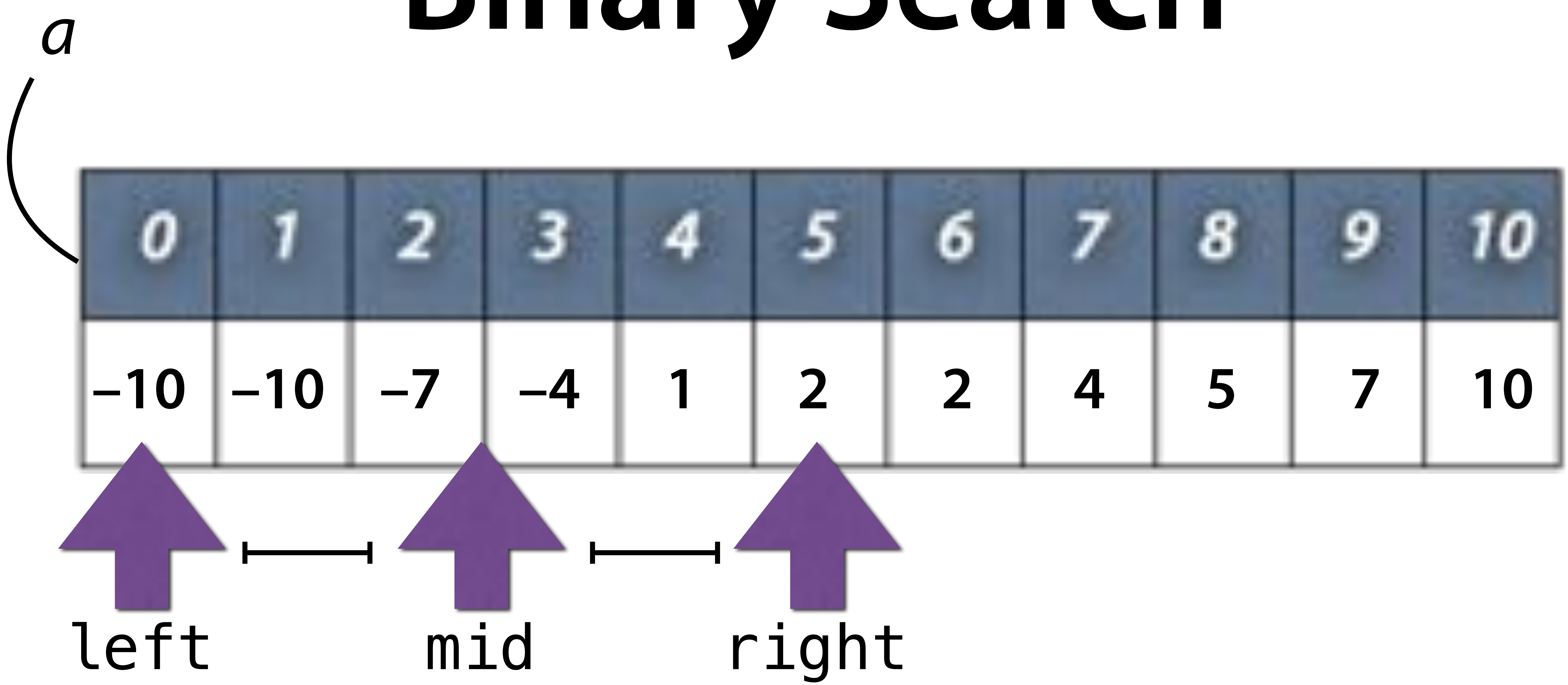
It is impossible to show correctness automatically for all programs

Binary Search

```
int sorted_find(int a[], int size, int x) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + ((right - left) / 2);
        if (x == a[mid])
            return mid;
        else if (x < a[mid])
            right = mid - 1;
        else // (x > a[mid])
            left  = mid + 1;
    }
    return -1;
}
```

Binary Search




```
int index = sorted_find(a, 11, −4);
```

Insertion Sort

```
void insertion_sort(int a[], int size) {
    for (int i = 1; i < size; i++) {
        int j = i;
        while (j > 0 && a[j - 1] > a[j]) {
            // Swap a[j] and a[j - 1]
            int tmp = a[j];
            a[j] = a[j - 1];
            a[j - 1] = tmp;
            j--;
        }
    }
}
```

Merge Sort

```
// Sort a[0..size], using b[0..size] as a buffer
void merge_sort(int a[], int b[], int size)
{
    partial_merge_sort(a, b, 0, size);
}
```

- We use *b[]* as an auxiliary array
- Must have the same size as *a*[]

Sort

```
// Sort a[begin..end], using b[begin..end] as buffer
void partial_merge_sort(int a[], int b[],
                        int begin, int end)
{
    if (end - begin < 2)
        return;

    // Split and sort
    int mid = begin + (end - begin) / 2;
    partial_merge_sort(a, b, begin, mid);
    partial_merge_sort(a, b, mid, end);

    // Merge and copy
    merge(a, b, begin, mid, end);
    copy(a, b, begin, end);
}
```

Merge

```
// Merge a[begin..mid - 1] and a[mid..end] into b[begin..end]
void merge(int a[], int b[], int begin, int mid, int end) {
    int i_begin = begin;
    int i_mid   = mid;
    for (int j = begin; j < end; j++)
        if (i_begin < mid &&
            (i_mid >= end || a[i_begin] <= a[i_mid]))
            b[j] = a[i_begin++];
        else
            b[j] = a[i_mid++];
}
```

- P && Q only evaluates Q if P is true
- P || Q only evaluates Q if P is false
- Protects the array borders from access

Copy

```
// Copy b[begin..end] into a[begin..end]
void copy(int a[], int b[], int begin, int end) {
    for (int k = begin; k < end; k++)
        a[k] = b[k];
}
```