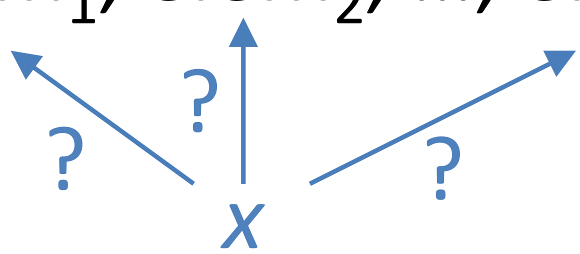
CS 2210 Data Structures and Algorithms

The Search Problem

A Fundamental Problem

Given a set S of n elements and a particular element x the search problem is to decide whether x is in S.

$$S = \{\text{elem}_1, \text{elem}_2, \ldots, \text{elem}_n\}$$



A Fundamental Problem

Given a set S of n elements and a particular element x the search problem is to decide whether x is in S.

- If x is in S we want to know where in S it appears

$$S = \{\text{elem}_1, \text{elem}_2, \ldots, x, \ldots, \text{elem}_n\}$$

- If x is not in S we want and indication that this is the case

$$x \text{ is not in } S = \{\text{elem}_1, \text{elem}_2, \ldots, \text{elem}_n\}$$

The Search Problem

This problem has a large number of applications:

- S = Names in a phone book

  x = name of a person

Application: Find phone number of person

The Search Problem

This problem has a large number of applications:

- *S* = Student records

  x = student ID

Application:

Print a

transcript

The Search Problem

This problem has a large number of applications:

- *S* = Variables in a program

  x = name of a variable

Application:
Find compilation errors

```
/* When the user has selected a play, this
   process the selected play */
public void actionPerformed(ActionEvent ev
    if(event.getSource() instanceof JButton) { /* Some position of the
                                                  board was selected */
        int row = -1, col = -1;
        PosPlay pos;

        if (game_ended) System.exit(0);
        /* Find out which position was selected by th eplayer */
        for (int i = 0; i < board_size; i++) {
            for (int j = 0; j < board_size; j++)
                if(event.getSource() == board[i][j]) {
                    row = i;
                    col = j;
                    break;
                }
            if (row != -1) break;
```

The Search Problem

This problem has a large number of applications:

- *S* = Variables in a program

  x = name of a variable

S = Symbol Table

?

```
/* When the user has selected a play, this method is invoked to
   process the selected play */
public void actionPerformed(ActionEvent event) {
    if(event.getSource() instanceof JButton) { /* Some position of the
                                                   board was selected */
        int row = -1, col = -1;
        PosPlay pos;

        if (game_ended) System.exit(0);
        /* Find out which position was selected by th eplayer */
        for (int i = 0; i < board_size; i++) {
            for (int j = 0; j < board_size; j++)
                if(event.getSource() == board[i][j]) {
                    row = i;
                    col = j;
                    break;
                }
            if (row != -1) break;
        }
```

The Search Problem

This problem has a large number of applications:

- *S* = Web host names

  x = URL

Application: Display a Web page

Solving a Problem

The solution of a problem has 2 parts:

- How to organize data

  Data structure:

  - a systematic way of organizing and accessing data

- How to solve the problem

  Algorithm:

  - a step-by-step procedure for performing some task in finite time

Data Structure for the Search Problem

For simplicity, let us assume that *S* is a set of n different integers stored in non-decreasing order in an array *L*.

L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

x

Algorithms

- How to solve the problem

  An algorithm must have two properties:

  1. It must be correct: Always produces the correct answer/outcome
  2. It must be efficient.

Algorithms

Given a problem that can be solved using a computer there is an infinite number of different algorithms to solve it.

The job of a programmer is to select/design the most appropriate algorithm for a particular situation.

Software Development Life Cycle

- Specification
- Design
- Implementation
- Testing and Debugging
- Deployment

Linear Search

$i = 0$

L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

x

Compare x with L[0]

Linear Search

$i = 1$

L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

x

Compare *x* with L[1]

Linear Search

$i = 2$

L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

x

Compare x with L[2] …

Linear Search

i = 5

L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

x = 26

Compare *x* with L[5]. Value *x* found!

Linear Search

$i = n = 10$

L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

$x = 40$

If x is not in L then eventually i will have value n and the algorithm then terminates.

Writing an Algorithm

It is not a good practice to write algorithms directly on the computer in some programming language.

Instead, we should write the algorithms in pseudocode.

Pseudocode

Pseudocode is a combination of English statements and programming-like control statements that

- allows us to express in detail an algorithm
- without having to deal with syntactic rules of a programming language

  (;, variable declarations, public, private, protected, static, casting, generics, …)

Pseudocode

Writing an algorithm in pseudocode makes it easier to design an algorithm because

- we only need to think about how to solve a problem and
- we do not have to think about how to express that algorithm in a particular programming language.

Algorithm LinearSearch (L,n,x)

Input: Array L of size n and value x

Output: Position i, $0 \leq i < n$, such that L[i] = x if x in L, or -1 if x not in L

Specification

Algorithm LinearSearch (L,n,x)

Input: Array L of size n and value x

Output: Position i, $0 \le i < n$, such that $L[i] = x$, if x in L, or -1, if x not in L

$i \leftarrow 0$

while $(i < n)$ **and** $(L[i] \neq x)$ **do**

$\quad i \leftarrow i+1$

if $i=n$ **then return** -1

else return i

Proving the Correctness of an Algorithm

To prove that an algorithm is correct we need to show 2 things:

- The algorithm terminates
- The algorithm produces the correct output

Correctness of Linear Search

Termination

- i takes values 0, 1, 2, 3, …
- The while loop cannot perform more than n iterations because of the condition ($i < n$)

Correctness of Linear Search

Correct Output

- The algorithm compares x with L[0], L[1], L[2], …
- Hence, if x is in L then x = L[i] in some iteration of the **while** loop; this ends the loop and then the algorithm correctly returns the value i
- If x is not in L then in some iteration $i = n$; this ends the loop and the algorithm returns -1.

Another Algorithm for the Search Problem

mid = 4

L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

x

Compare *x* with the value stored in the middle of array L

Another Algorithm for the Search Problem

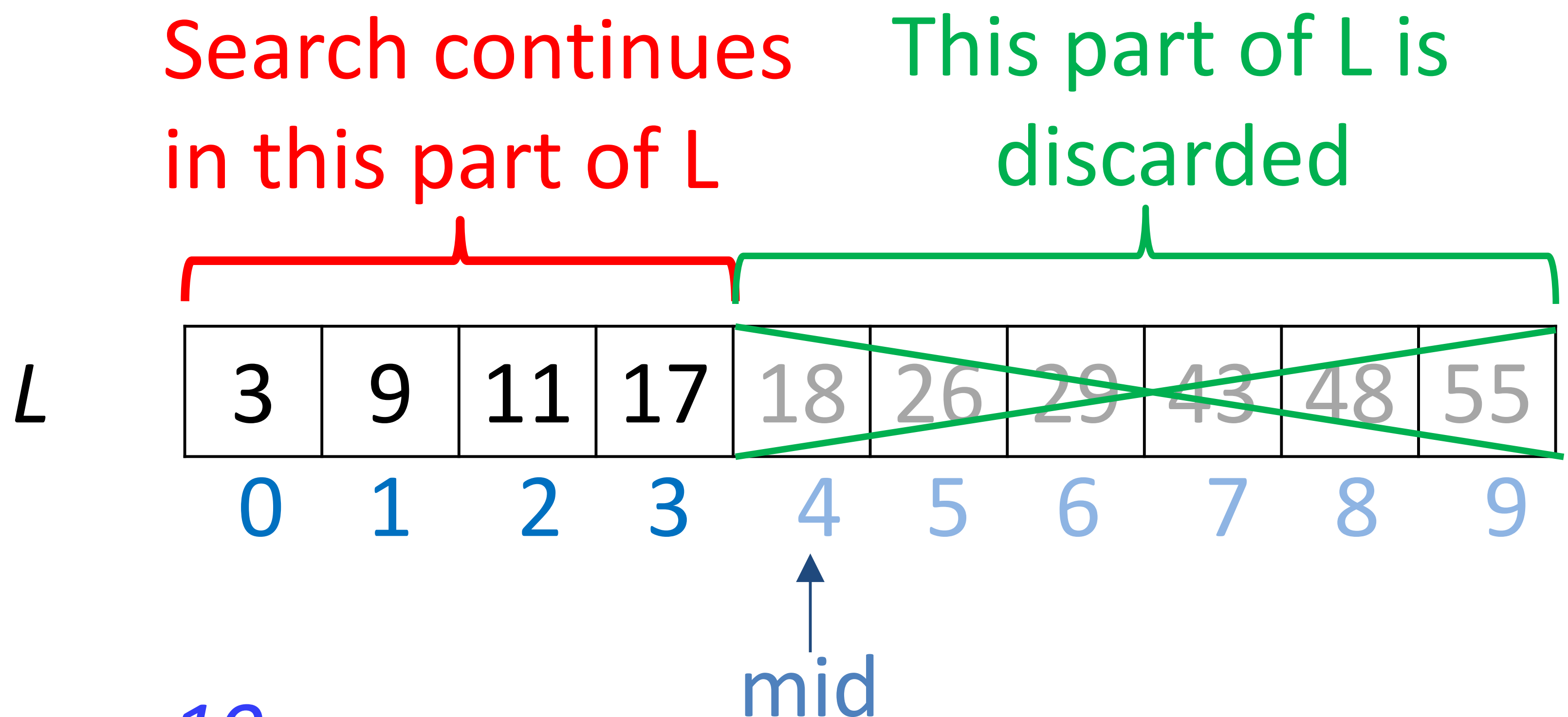
mid = 4

L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

x = 18

If *x* = L[mid] the algorithm ends

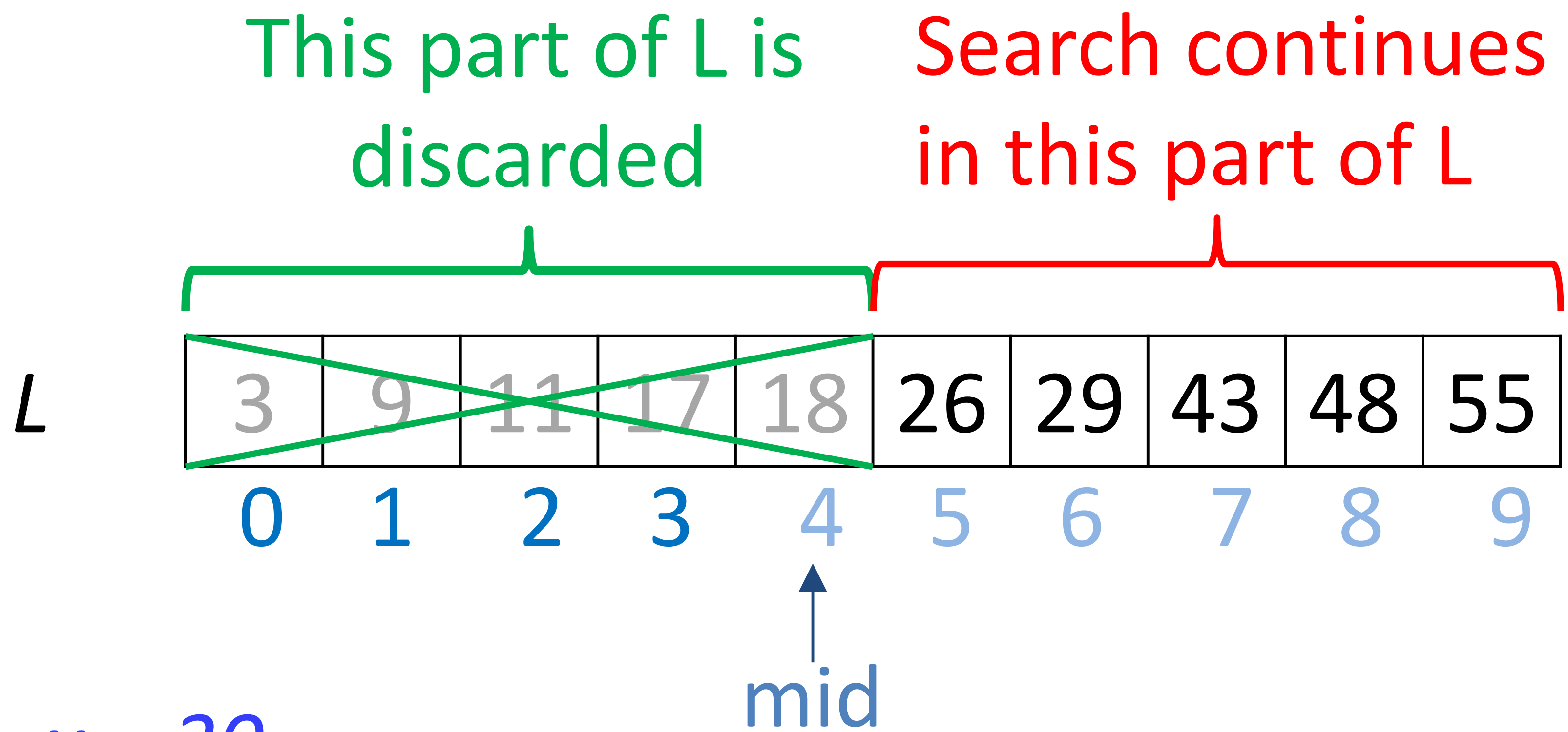
Another Algorithm for the Search Problem



x = *10*

If *x* < L[mid] the search continues on the first half of the array

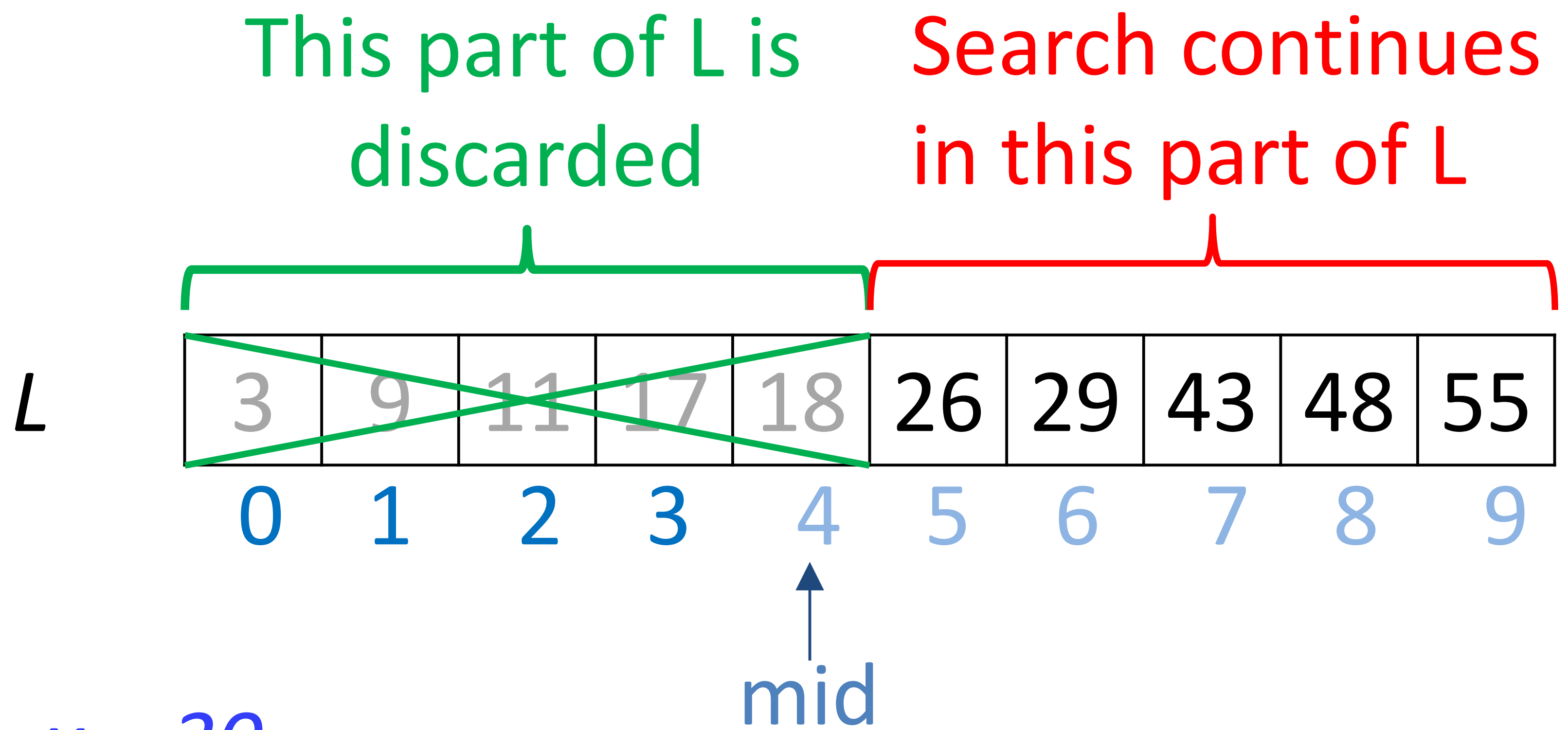
Another Algorithm for the Search Problem



x = 30

If *x* > L[mid] the search continues on the second half of the array

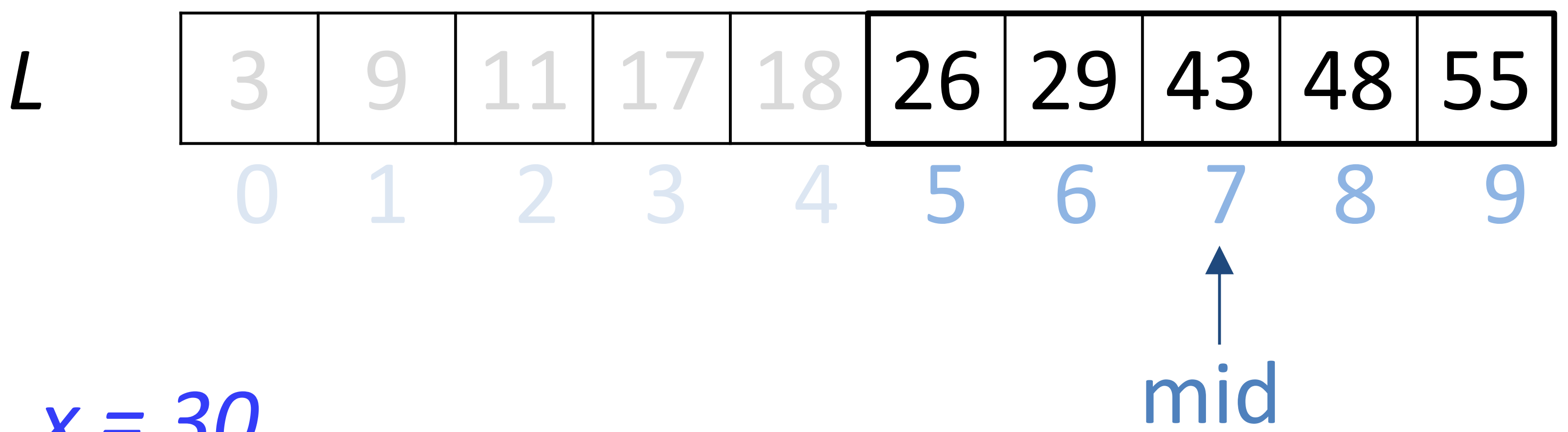
Another Algorithm for the Search Problem



x = 30

The search proceeds in the same manner in the remaining part of the array

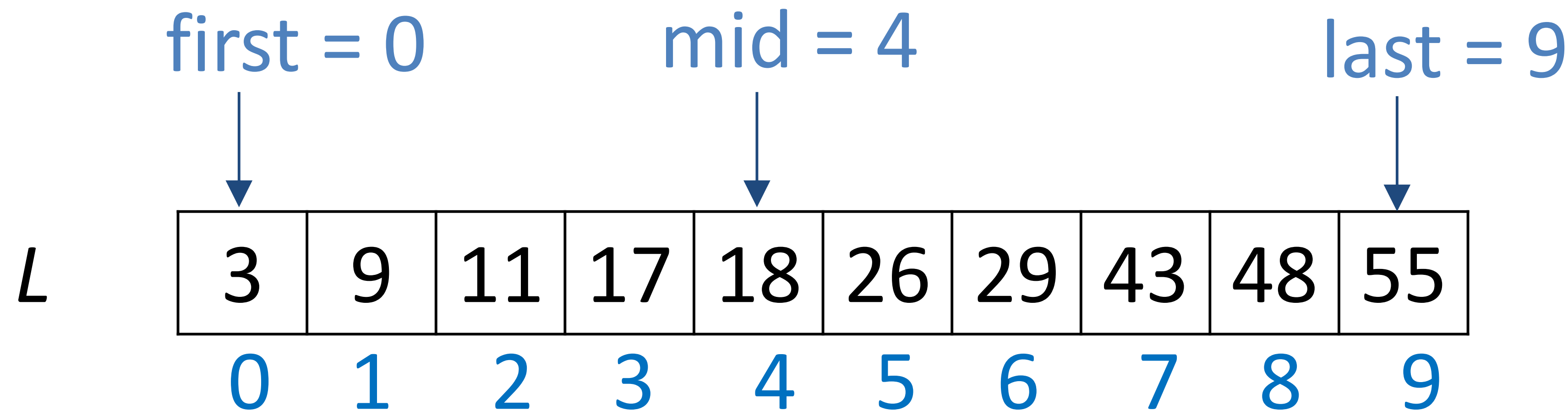
Another Algorithm for the Search Problem



x = 30

The search proceeds in the same manner in the remaining part of the array. This algorithm is called binary search.

Binary Search



x

We use two indices: first and last to bound the part of the array where the search is performed

Binary Search

first = 0 mid = 4 last = 9

L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

x = 18

If *x* = L[mid] the algorithm ends

Binary Search

first = 0 last = 3

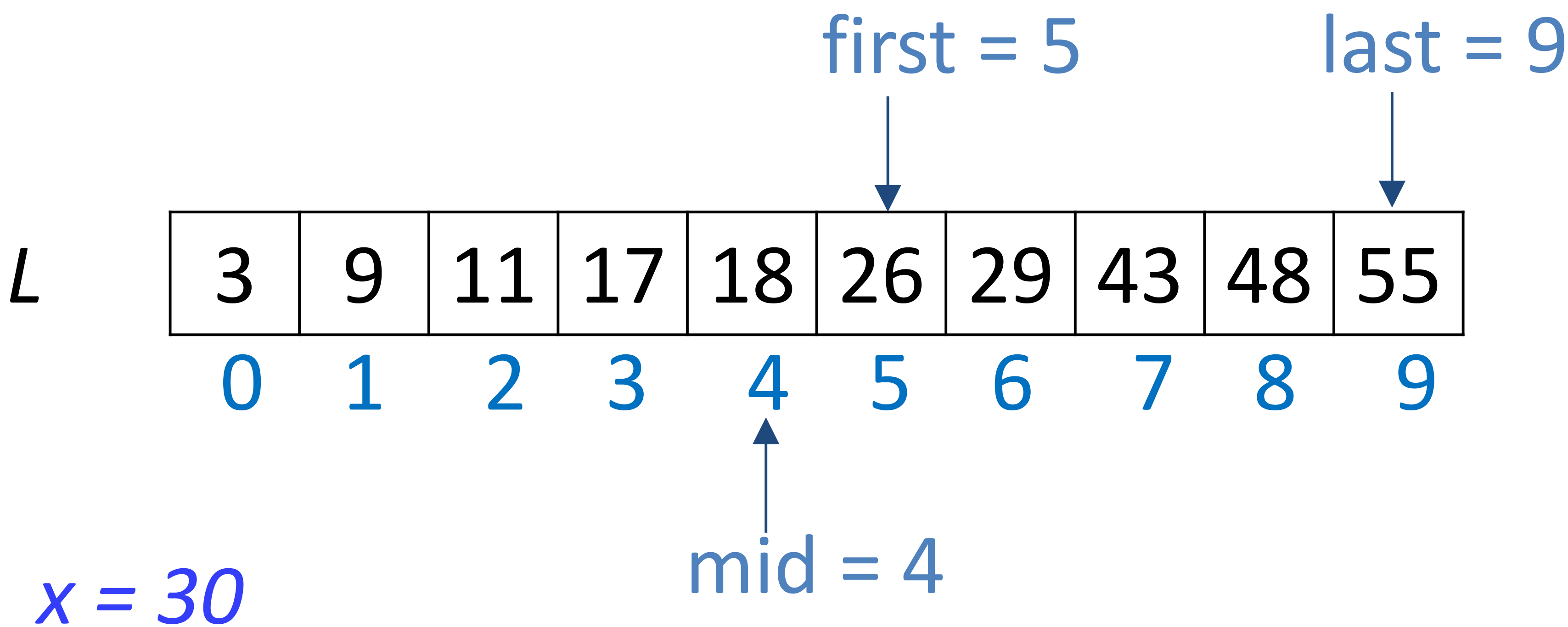
L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

mid = 4

x = 10

If x < L[mid] the value of last changes to mid – 1, so the search continues in the first half of L

Binary Search



If *x* > L[mid] the value of first changes to mid + 1, so the search continues in the second half of L

Binary Search

last = 6 first = 7

L	3	9	11	17	18	26	29	43	48	55
	0	1	2	3	4	5	6	7	8	9

mid = 6

x = 30

If first > last the algorithm terminates as *x* is not in L

Algorithm BinarySearch (L,x, first, last)

Input: Array L of size n and value x

Output: Position i, $0 \le i < n$, such that L[i] = x, if x in L, or -1, if x not in L

if first > last **then return** -1

else mid $\leftarrow \lfloor (first + last)/2 \rfloor$

This symbol means "round down"

if x = L[mid] **then return** mid

else if x < L[mid] **then**

  return BinarySearch (L,x,first,mid -1)

 else return BinarySearch (L,x,mid +1,last)

Correctness of Binary Search

Termination

- If $x = L[mid]$ the algorithm terminates
- If $x < L[mid]$ or $x > L[mid]$, the value $L[mid]$ is discarded from the next recursive call. Hence, in each recursive call the size of L decreases by at least 1.

L										
	3	9	11	17	18	26	29	43	48	55
					mid					

Correctness of Binary Search

Termination

- If x = L[*mid*] the algorithm terminates
- If x < L[*mid*] or x > L[*mid*], the value L[*mid*] is discarded from the next recursive call. Hence, in each recursive call the size of L decreases by at least 1.
- After a finite number of recursive calls the size of L must be zero and the algorithm ends

Correctness of Binary Search

Correct Output. Case 1: *x* is not in *L*

In each recursive call the algorithm only discards values different from x

Correctness of Binary Search

Correct Output. Case 1: *x* is not in *L*

In each recursive call the algorithm only discards values different from *x*

if x < L[mid] **then**

return BinarySearch (L,x,first,mid-1)

L											
	3	9	11	17	~~18~~	~~26~~	~~29~~	~~43~~	~~48~~	~~55~~	*x* < L[mid]
					mid						

else return BinarySearch (L,x,mid+1,last)

L											
	~~3~~	~~9~~	~~11~~	~~17~~	~~18~~	26	29	43	48	55	*x* > L[mid]
					mid						

Correctness of Binary Search

Correct Output. Case 1: *x* is not in *L*

Since the algorithm only discards values different from *x*, all values of *L* will be discarded (so *L* is empty) and the algorithm correctly returns -1:

<u>if</u> first > last **<u>then return</u>** -1

Correctness of Binary Search

Correct Output. Case 2: *x* is in *L*

Since the algorithm terminates then either

- **return** -1, or
- **return** mid

must be executed.

Since **return** -1 is executed only if *x* is not in *L* then

if $x = L[mid]$ **then return** *mid*

is executed when *x* is in *L* and so the algorithm correctly returns the position of *x* in *L*.