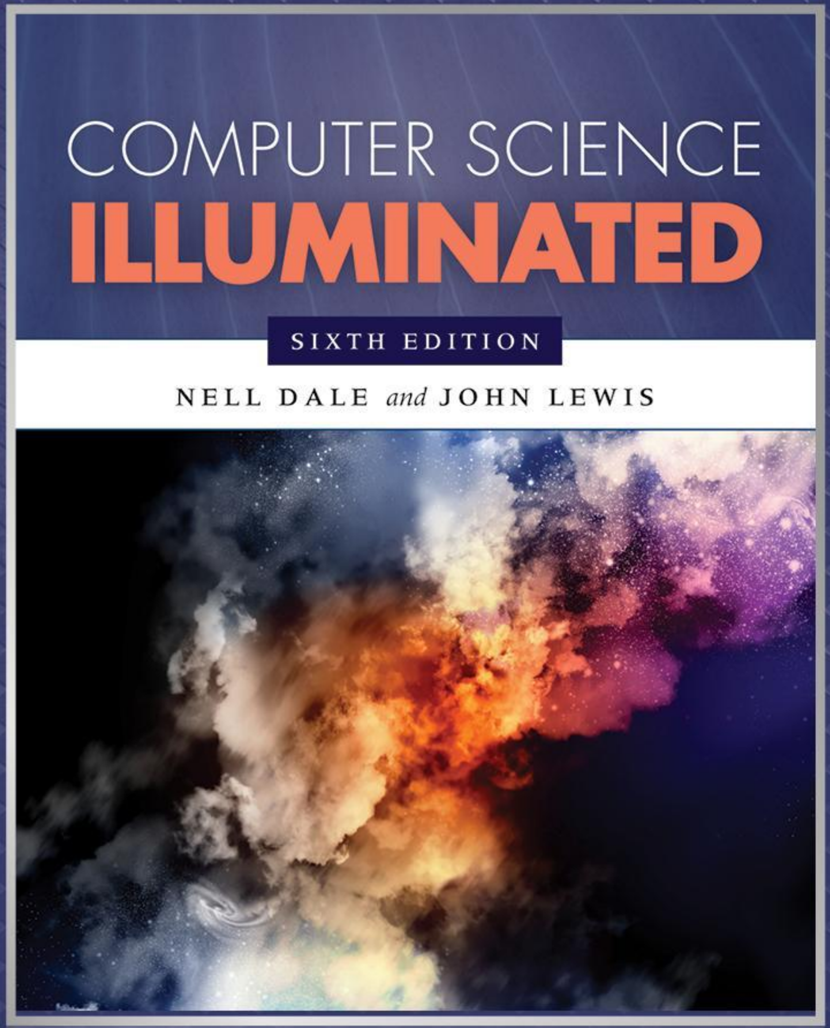
Problem Solving and Algorithms



Edits by Michael Edwards and Chris Whyley

Chapter Goals

- *Describe the computer problem-solving process and relate it to* Polya's How to Solve It *list*
- *Distinguish between a simple type and a composite type*
- *Describe two composite data-structuring mechanisms*
- *Recognize a recursive problem and write a recursive algorithm to solve it*
- *Distinguish between an unsorted array and a sorted array*
- *Distinguish between a selection sort and an insertion sort*

Chapter Goals

- *Describe the Quicksort algorithm*
- *Apply the selection sort, the bubble sort, insertion sort, and Quicksort to an array of items by hand*
- *Apply the binary search algorithm*
- *Demonstrate an understanding of the algorithms in this chapter by hand-simulating them with a sequence of items*

Problem Solving

Problem solving

The act of finding a solution to a perplexing, distressing, vexing, or unsettled question

How do **you** define problem solving?

Problem Solving

How to Solve It: A New Aspect of Mathematical Method *by George Polya*

"How to solve it list" written within the context of mathematical problems

But list is quite general

→ *We can use it to solve computer related problems!*

Problem Solving

How do you solve problems?

Understand the problem

Devise a plan

Carry out the plan

Look back

Strategies

Ask questions!

- What do I know about the problem?
- What is the information that I have to process in order the find the solution?
- What does the solution look like?
- What sort of special cases exist?
- How will I recognize that I have found the solution?

Strategies

Ask questions! Never reinvent the wheel!

Similar problems come up again and again in different guises

A good programmer recognizes a task or sub-task that has been solved before and plugs in the solution

Can you think of two similar problems?

Strategies

Divide and Conquer!

Break up a large problem into smaller units and solve each smaller problem

- *Applies the concept of abstraction*
- *The divide-and-conquer approach can be applied over and over again until each sub-task is manageable*

Computer Problem-Solving

Analysis and Specification Phase

- *Analyze*
- *Specification*

Algorithm Development Phase

- *Develop algorithm*
- *Test algorithm*

Implementation Phase

- *Code algorithm*
- *Test algorithm*

Maintenance Phase

- *Use*
- *Maintain*

Phase Interactions

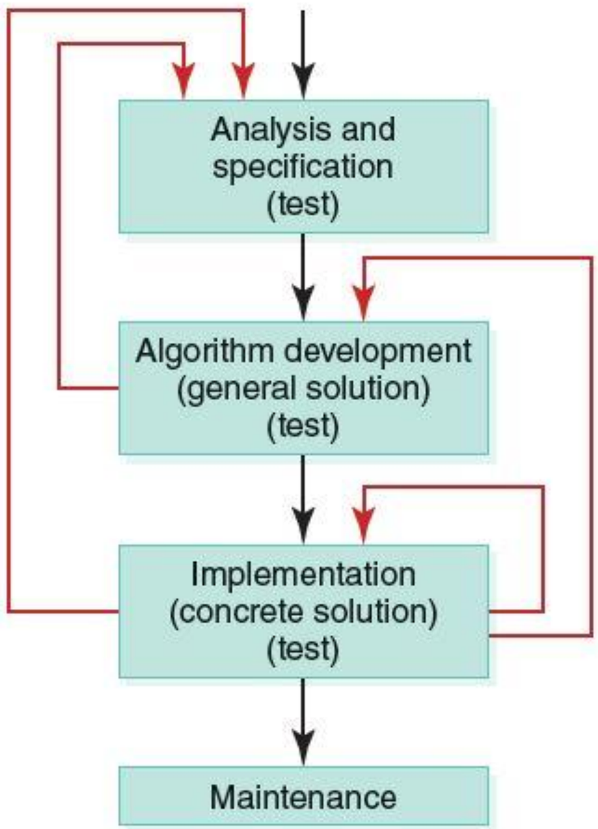


FIGURE 7.3 The interactions among the four problem-solving phases

Should we add another arrow?

(What happens if the problem is revised?)

Algorithms

Algorithm

A set of unambiguous instructions for solving a problem or sub-problem in a finite amount of time using a finite amount of data

Abstract Step

An algorithmic step containing unspecified details

Concrete Step

An algorithm step in which all details are specified

Developing an Algorithm

Two methodologies used to develop computer solutions to a problem

- *Top-down design focuses on the **tasks** to be done*
- *Object-oriented design focuses on the **data** involved in the solution*

Summary of Methodology

Analyze the Problem

Understand the problem!!

Develop a plan of attack

List the Main Tasks (becomes Main Module)

Restate problem as a list of tasks (modules)

Give each task a name

Write the Remaining Modules

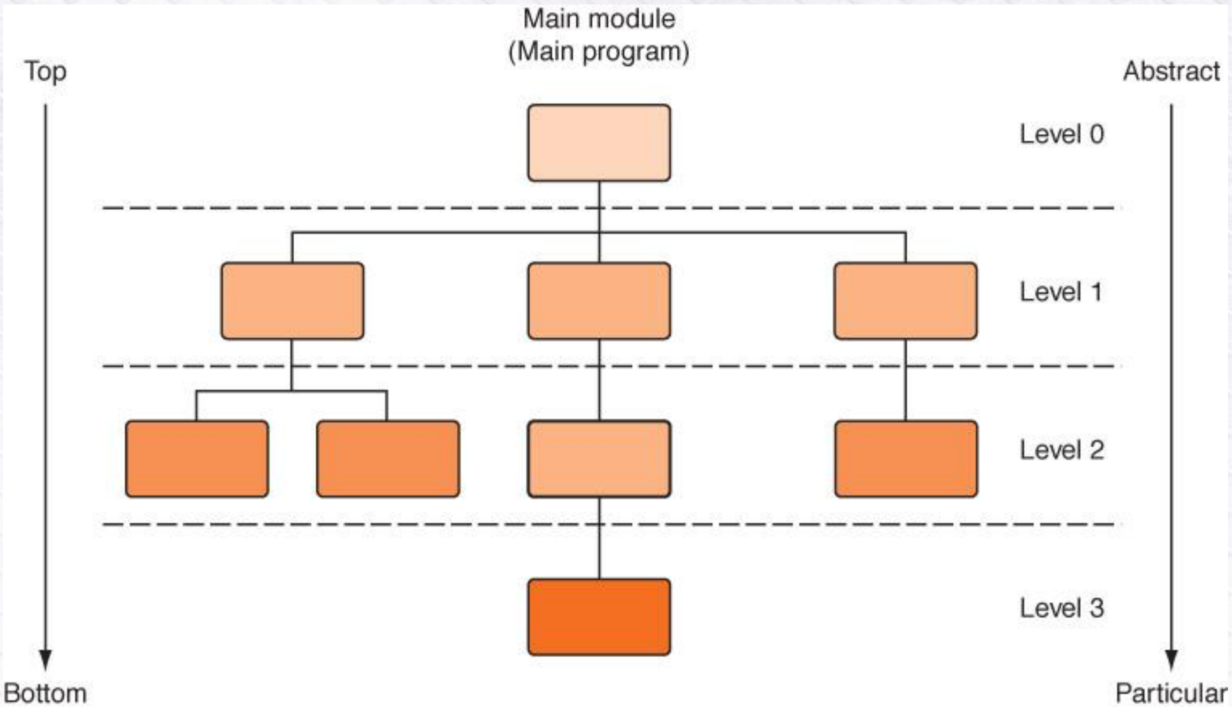
Restate each abstract module as a list of tasks

Give each task a name

Re-sequence and Revise as Necessary

Process ends when all steps (modules) are concrete

Top-Down Design



Process continues for as many levels as it takes to make every step concrete

Name of (sub)problem at one level becomes a module at next lower level

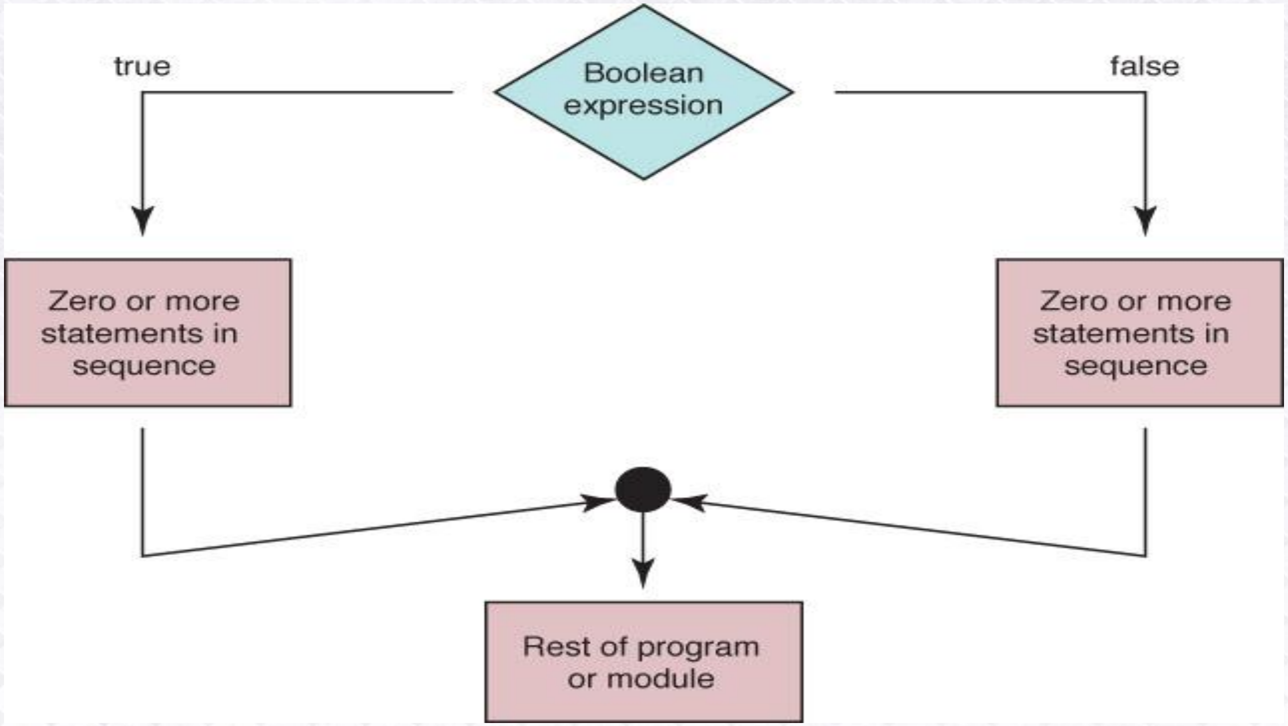
Control Structures

Control structure

An instruction that determines the order in which other instructions in a program are executed

Can you name the ones we defined in the functionality of pseudocode (CS-150)?

Selection Statements



Flow of control of if statement

Algorithm with Selection

Problem: Write the appropriate dress for a given temperature.

Write "Enter temperature"
Read temperature
Determine Dress

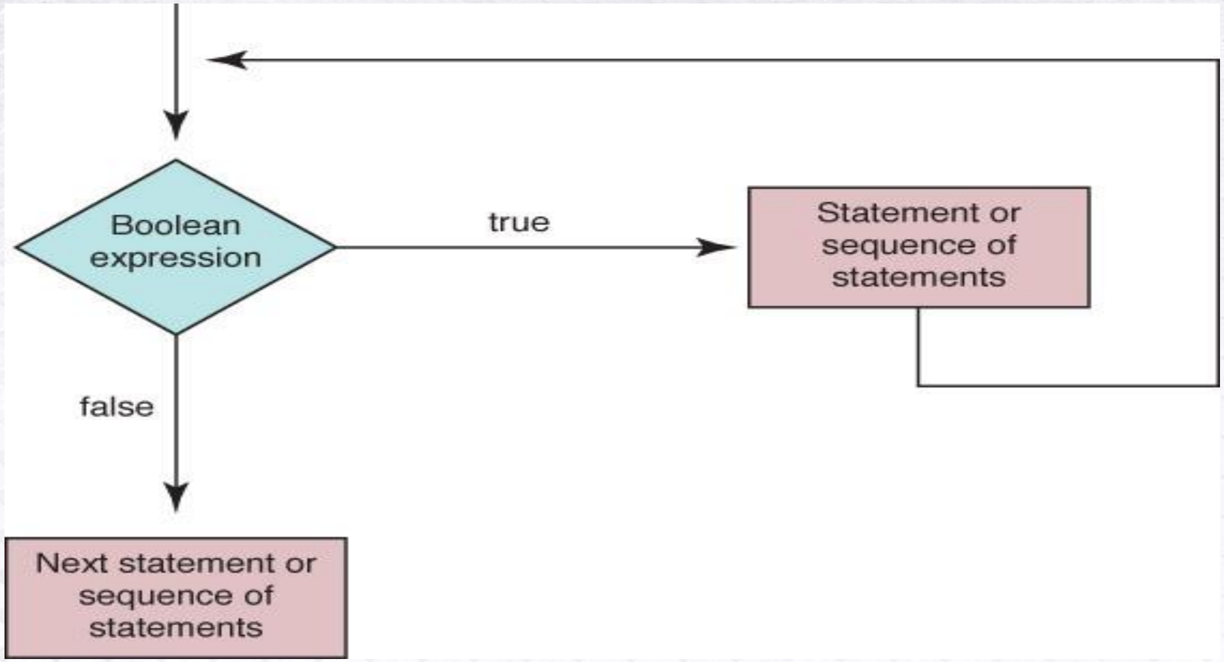
Which statements are concrete?
Which statements are abstract?

Algorithm with Selection

Determine Dress

```
IF (temperature > 90)
    Write "Texas weather: wear shorts"
ELSE IF (temperature > 70)
    Write "Ideal weather: short sleeves are fine"
ELSE IF (temperature > 50)
    Write "A little chilly: wear a light jacket"
ELSE IF (temperature > 32)
    Write "Philadelphia weather: wear a heavy coat"
ELSE
    Write "Stay inside"
```

Looping Statements



Flow of control of while statement

Looping Statements

A count-controlled loop

```
Set sum to 0
Set count to 1
While (count <= limit)
    Read number
    Set sum to sum + number
    Increment count
Write "Sum is " + sum
```

Why is it called a count-controlled loop?

Looping Statements

An event-controlled loop

```
Set sum to 0
Set allPositive to TRUE
WHILE (allPositive)
    Read number
    IF (number > 0)
        Set sum to sum + number
    ELSE
        Set allPositive to FALSE
Write "Sum is " + sum
```

Why is it called an event-controlled loop?
What is the event?

Looping Statements

Calculate Square Root

Read in square
Calculate the square root
Write out square and the square root

Are there any abstract steps?

Looping Statements

Calculate Square Root

```
Set epsilon to 1
WHILE  (epsilon > 0.001)
    Calculate new guess
    Set epsilon to abs(square - guess * guess)
```

Are there any abstract steps?

Looping Statements

Calculate New Guess

```
Set newGuess to
    (guess + (square/guess)) / 2.0
```

Are there any abstract steps?

Looping Statements

```
Read in square
Set guess to square/4
Set epsilon to 1
WHILE  (epsilon > 0.001)
    Calculate new guess
    Set epsilon to abs(square - guess * guess)
Write out square and the guess
```

Composite Data Types

Records

A named heterogeneous collection of items in which individual items are accessed by name. For example, we could bundle name, age and hourly wage items into a record named Employee

Arrays

A named homogeneous collection of items in which an individual item is accessed by its position (index) within the collection

Composite Data Types

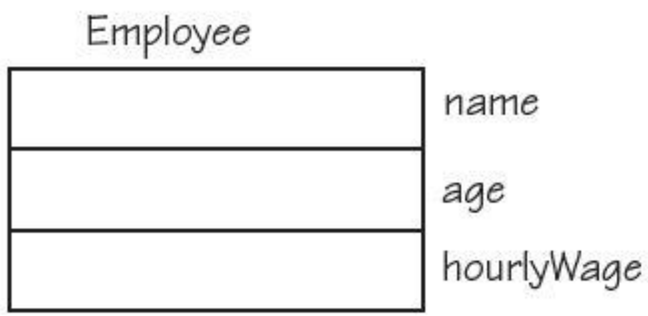


FIGURE 7.6 Record Employee

Following algorithm, stores values into the fields of record:

Employee employee // Declare and Employee variable
Set employee.name to "Frank Jones"
Set employee.age to 32
Set employee.hourlyWage to 27.50

Composite Data Types

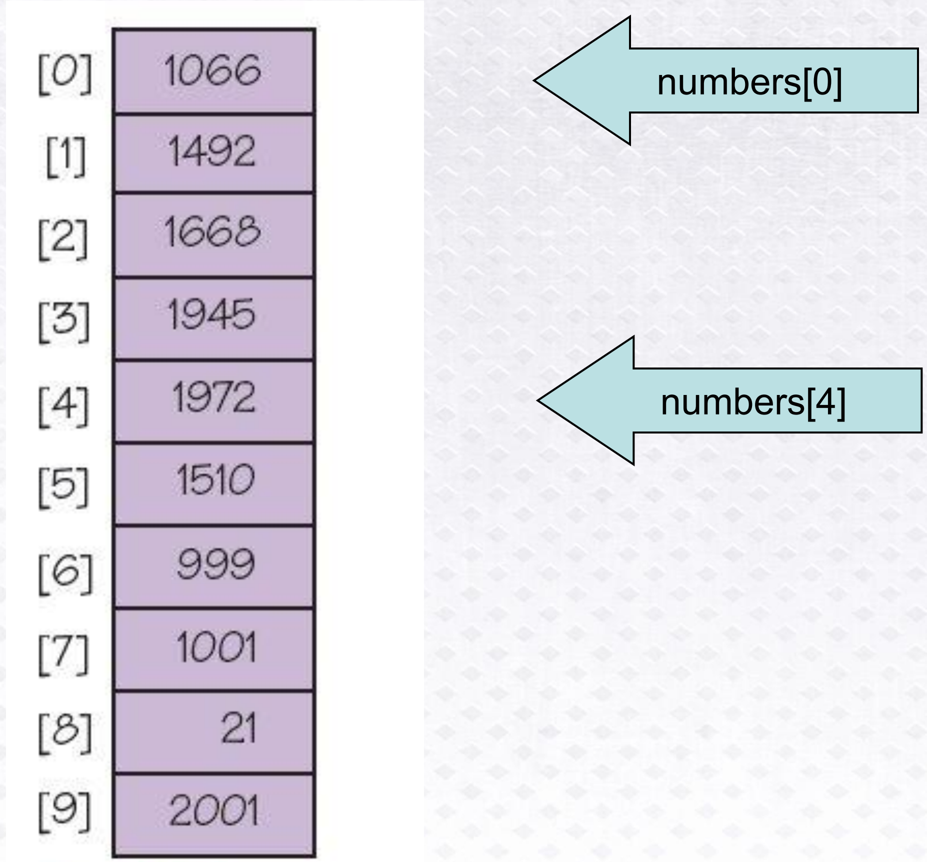


FIGURE 7.5 An array of ten numbers

Arrays

As data is being read into an array, a counter is updated so that we always know how many data items were stored

If the array is called **list**, *we are working with*

list[0] to list[length-1]   or

list[0]..list[length-1]

An Unsorted Array

FIGURE 7.7 An unsorted array

list[0]...list[length-1]
is of interest

Composite Data Types

Fill array numbers with limit *values*

```
integer data[20]
Write "How many values?"
Read length
Set index to 0
WHILE (index < length)
    Read data[index]
    Set index to index + 1
```

Sequential Search of an Unsorted Array

A sequential search examines each item in turn and compares it to the one we are searching.

If it matches, we have found the item. If not, we look at the next item in the array.

We stop either when we have found the item or when we have looked at all the items and not found a match

Thus, a loop with two ending conditions

Sequential Search Algorithm

```
Set position to 0
Set found to FALSE
WHILE (position < length AND NOT found )
    IF (numbers[position] equals searchitem)
        Set found to TRUE
    ELSE
        Set position to position + 1
RETURN position
```

Sequential Search Algorithm w/ Sentinel

The previous algorithm had two conditions to check in the WHILE

We can reduce the number of tests per iteration by posting a sentinel

We insert a copy of the value searched for after the last item so a copy of the item is always found

Now the WHILE only has one condition to check!

Sequential Search Algorithm w/ Sentinel

```
Set position to 0
Set data[length] = searchitem

WHILE (position < length)
        IF (data [position] equals searchitem)
            RETURN position
        ELSE
            Set position to position + 1

Clear data[length]

RETURN position as zero
```

Posting the Sentinel

Removing the Sentinel

Sequential Search Complexity

In both algorithms the number of tests is proportional to the length of the list – O(n)

The order of the testing can influence the number of tests required to be carried out.

Booleans Expression

AND *returns* TRUE *if both operands are true and* FALSE *otherwise*

OR *returns* TRUE *if either operand is true and* FALSE *otherwise*

NOT *returns* TRUE *if its operand is false and* FALSE *if its operand is true*

The order matters (sometimes). If *A* is FALSE in the expression *A AND B*, do we need to check *B*?

What about A OR B?

Sorted Arrays

The values stored in an array have unique keys of a type for which the relational operators are defined

Sorting rearranges the elements into either ascending or descending order within the array

A sorted array is one in which the elements are in order

Sequential Search in a Sorted Array

If items in an array are sorted, we can stop looking when we pass the place where the item would be it were present in the array

Is this better?

A Sorted Array

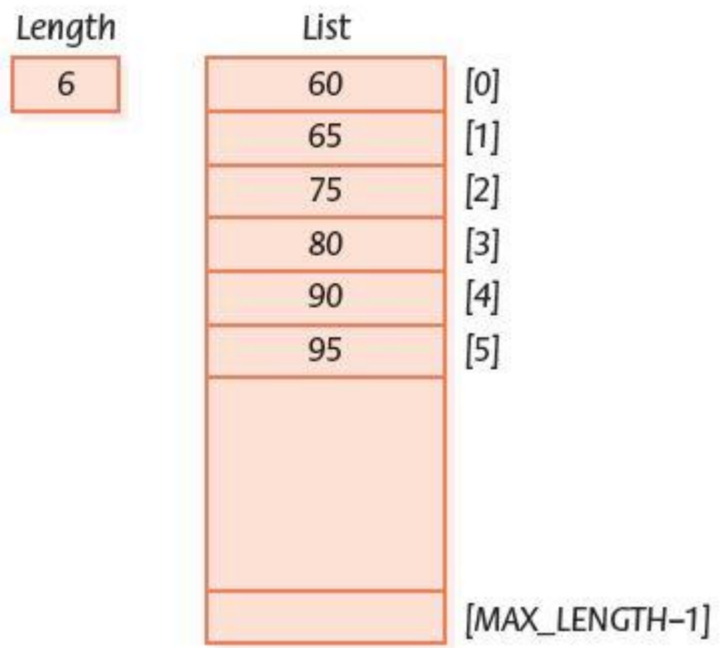


FIGURE 7.8 A sorted array

A sorted array of integers

A Sorted Array

Read in array of values

Write “Enter value for which to search”

Read searchItem

Set found to TRUE if searchItem is there

IF (found)

    Write “Item is found”

ELSE

    Write “Item is not found”

A Sorted Array

<u>Set found to TRUE if searchItem is there</u>

```
Set index to 0
Set found to FALSE
WHILE (index < length AND NOT found)
    IF (data[index] equals searchItem)
        Set found to TRUE
    ELSE IF (data[index] > searchItem)
        Set index to length
    ELSE
        Set index to index + 1
```

Binary Search

Sequential search

Search begins at the beginning of the list and continues until the item is found or the entire list has been searched

Binary search *(list must be sorted)*

Search begins at the middle and finds the item or eliminates half of the unexamined items; process is repeated on the half where the item might be

Say that again…

Binary Search

```
Set lower to 0
Set upper to length - 1
Set found to FALSE

WHILE (upper >= lower AND NOT found)
    Set middle to (upper + lower) // 2
    IF (data[middle] equals target))
        Set found to TRUE
    ELSE IF (data[middle] < target)
        Set last to middle + 1
    ELSE
        Set first to middle – 1
RETURN found
```

Binary Search

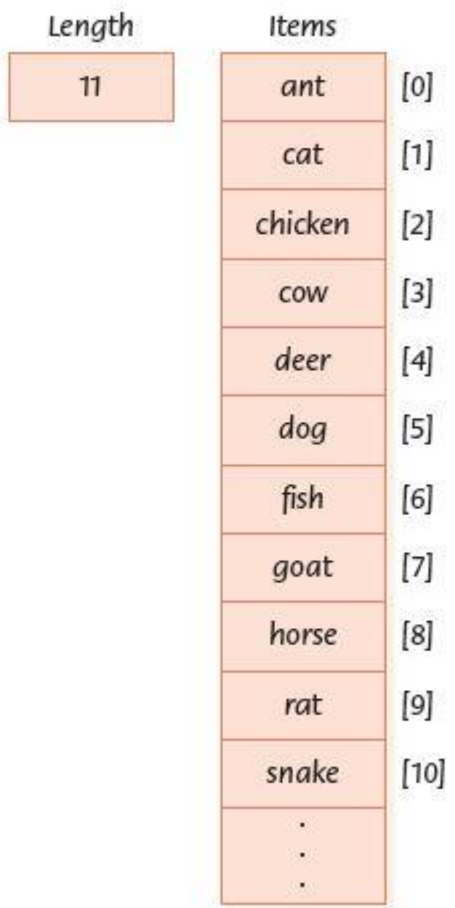


FIGURE 7.9 Binary search example

Searching for *cat*

First	Last	Middle	Comparison	
0	10	5	*cat < dog*	
0	4	2	*cat < chicken*	
0	1	0	*cat > ant*	
1	1	1	*cat = cat*	**Return: true**

Searching for *fish*

First	Last	Middle	Comparison	
0	10	5	*fish > dog*	
6	10	8	*fish < horse*	
6	7	6	*fish = fish*	**Return: true**

Searching for *zebra*

First	Last	Middle	Comparison	
0	10	5	*zebra > dog*	
6	10	8	*zebra > horse*	
9	10	9	*zebra > rat*	
10	10	10	*zebra > snake*	
11	10		*first > last*	**Return: false**

FIGURE 7.10 Trace of the binary search

Binary Search Worked Example 1

Target: 13

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

Binary Search Worked Example 1

Target: 13

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower

Binary Search
Worked Example 1

Target: 13

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower *upper*

Binary Search Worked Example 1

Target: 13

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower

mid_pt

upper

Binary Search Worked Example 1

Target: 13

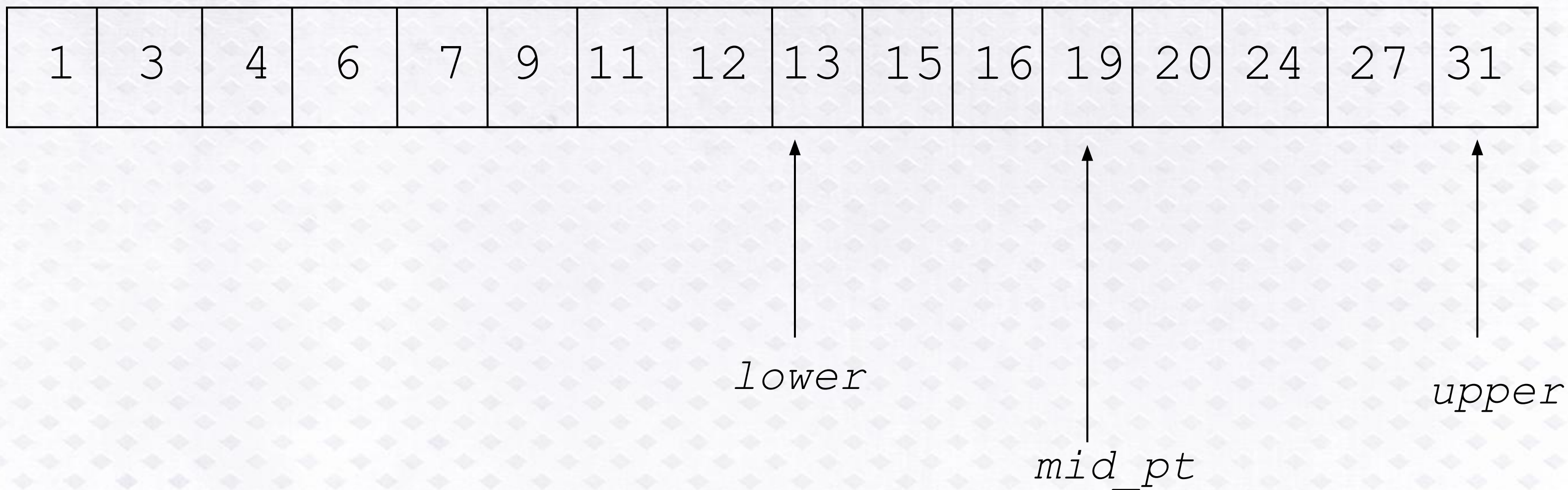
1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower

upper

Binary Search Worked Example 1

Target: 13



Binary Search Worked Example 1

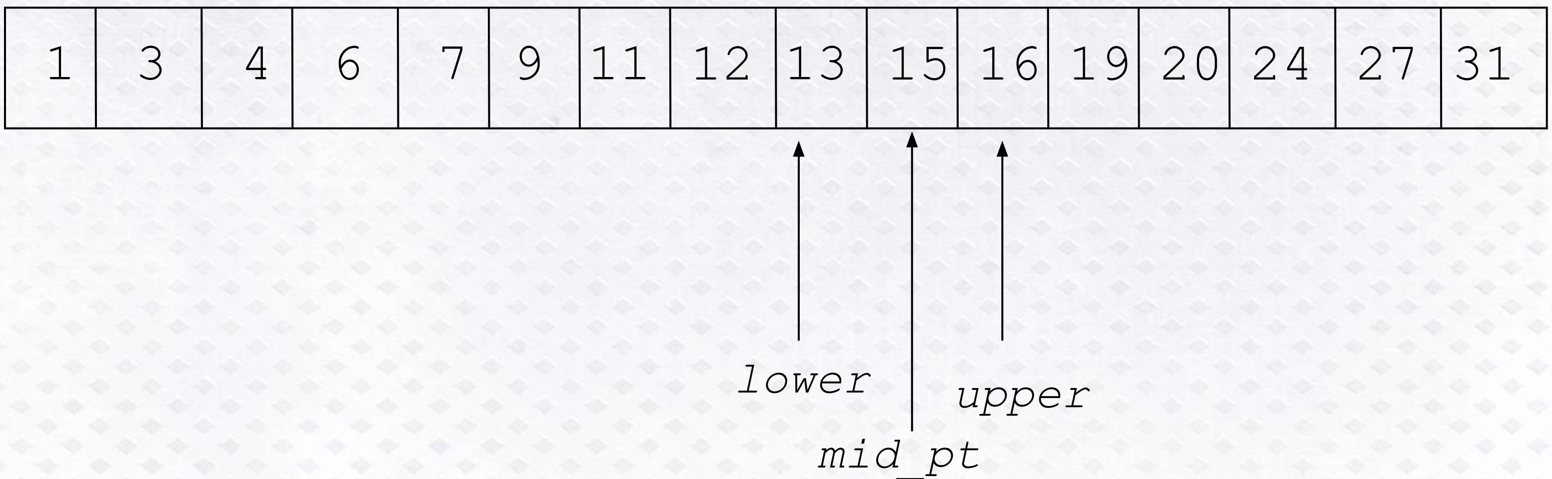
Target: 13

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower *upper*

Binary Search Worked Example 1

Target: 13



Binary Search Worked Example 1

Target: 13

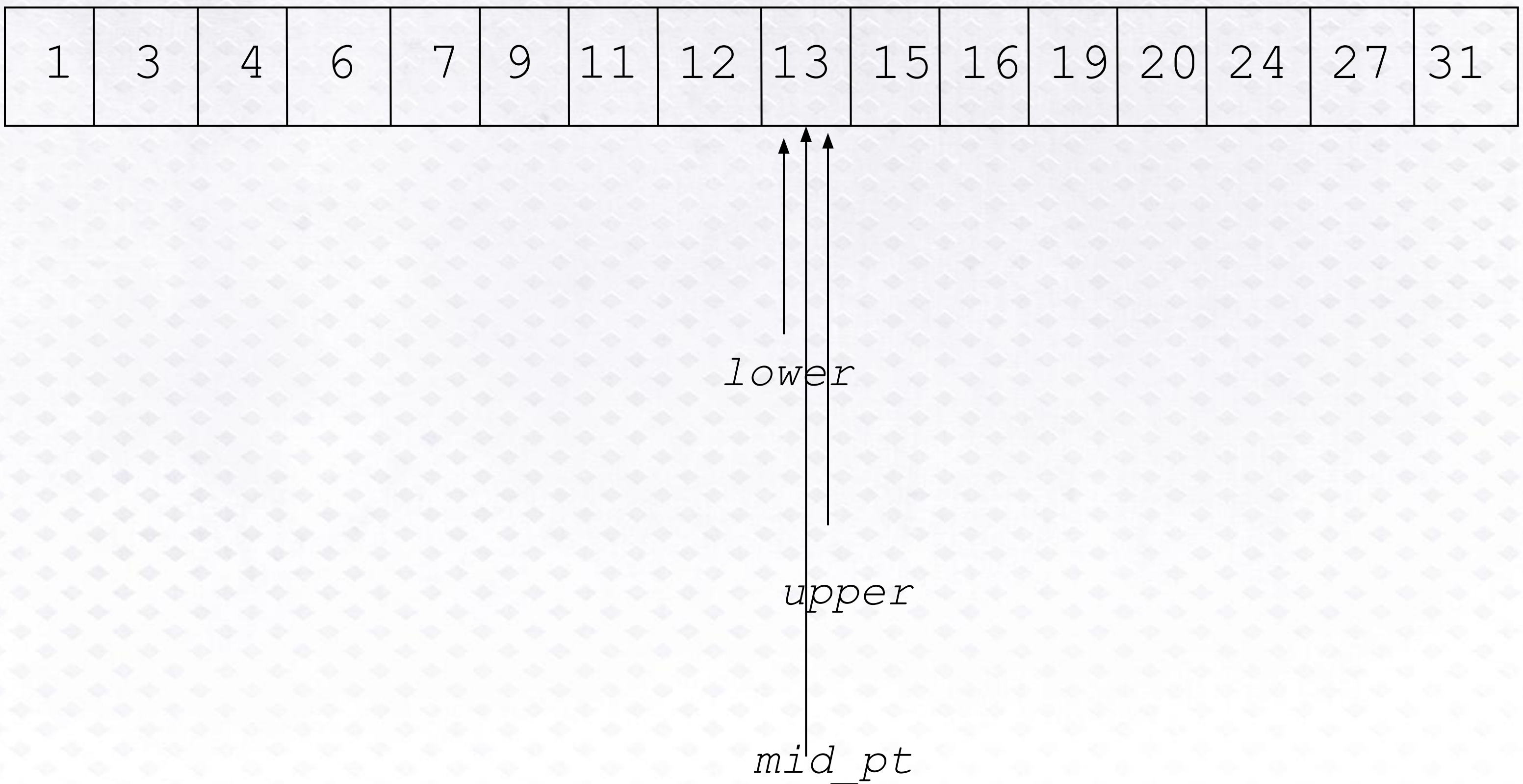
1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower

upper

Binary Search Worked Example 1

Target: 13,
FOUND



Binary Search Worked Example 2

Target: 6

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

Binary Search Worked Example 2

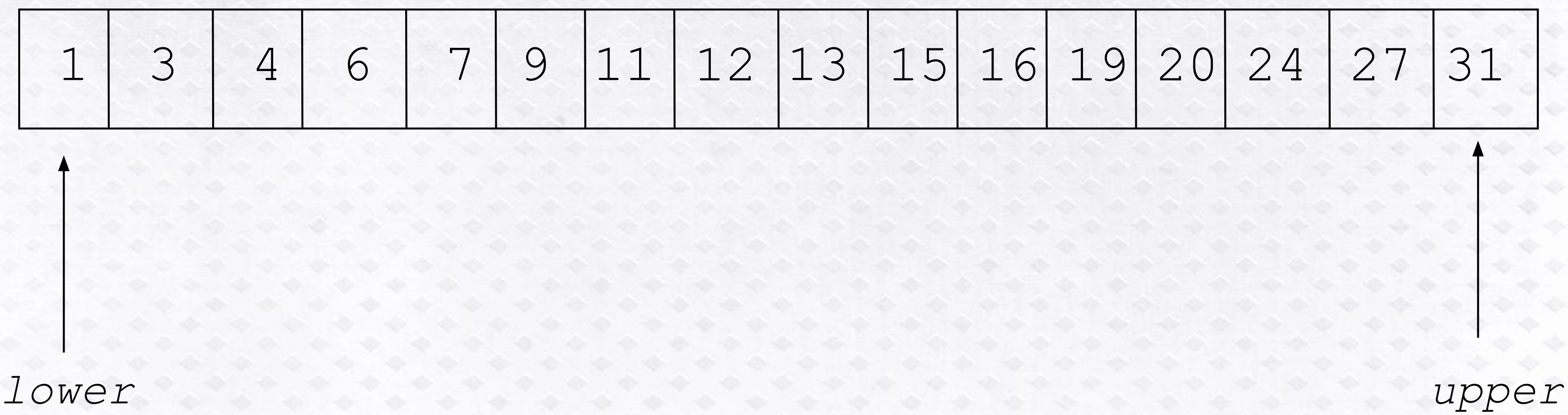
Target: 6

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower

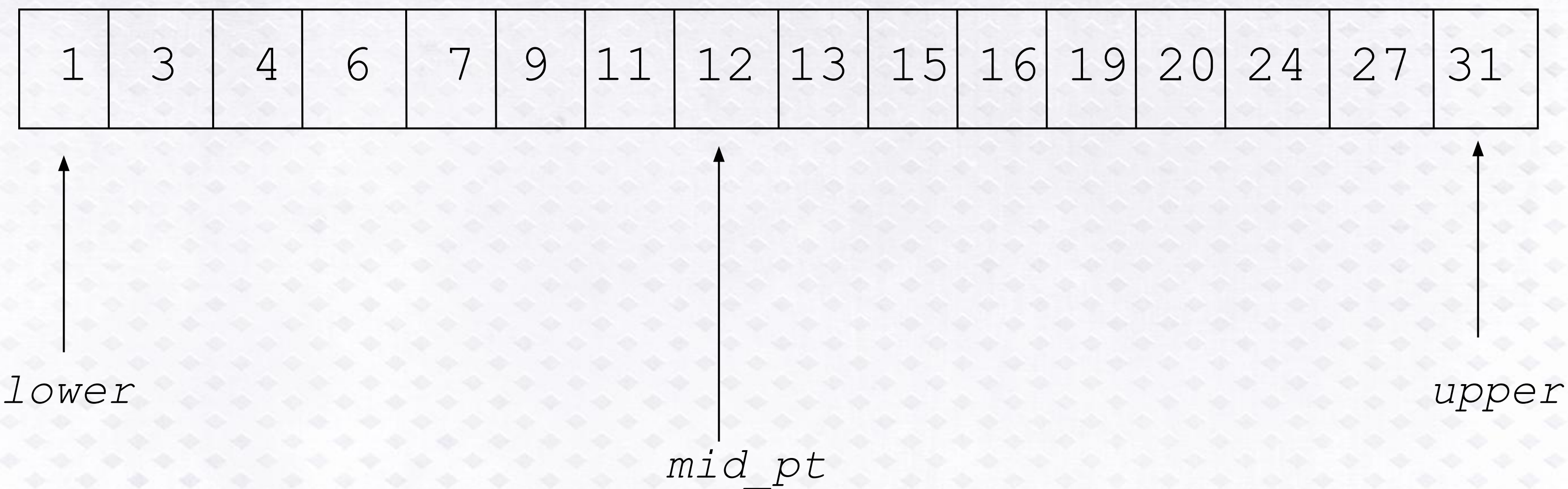
Binary Search Worked Example 2

Target: 6



Binary Search Worked Example 2

Target: 6



Binary Search
Worked Example 2

Target: 6

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower *upper*

Binary Search
Worked Example 2

Target: 6
FOUND

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower

mid_pt

upper

Binary Search Worked Example 3

Target: 14 (Doesn't exist)

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

Binary Search Worked Example 3

Target: 14 (Doesn't exist)

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower

Binary Search Worked Example 3

Target: 14 (Doesn't exist)

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower *upper*

Binary Search
Worked Example 3

Target: 14 (Doesn't exist)

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower *mid_pt* *upper*

Binary Search
Worked Example 3

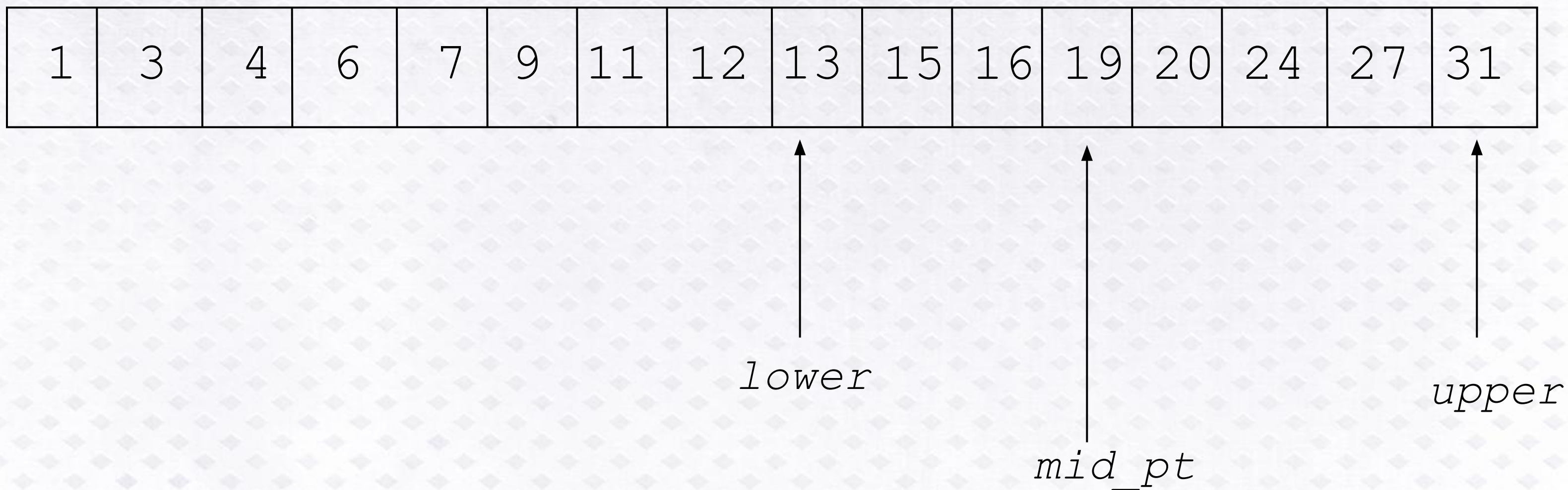
Target: 14 (Doesn’t exist)

1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

`lower` `upper`

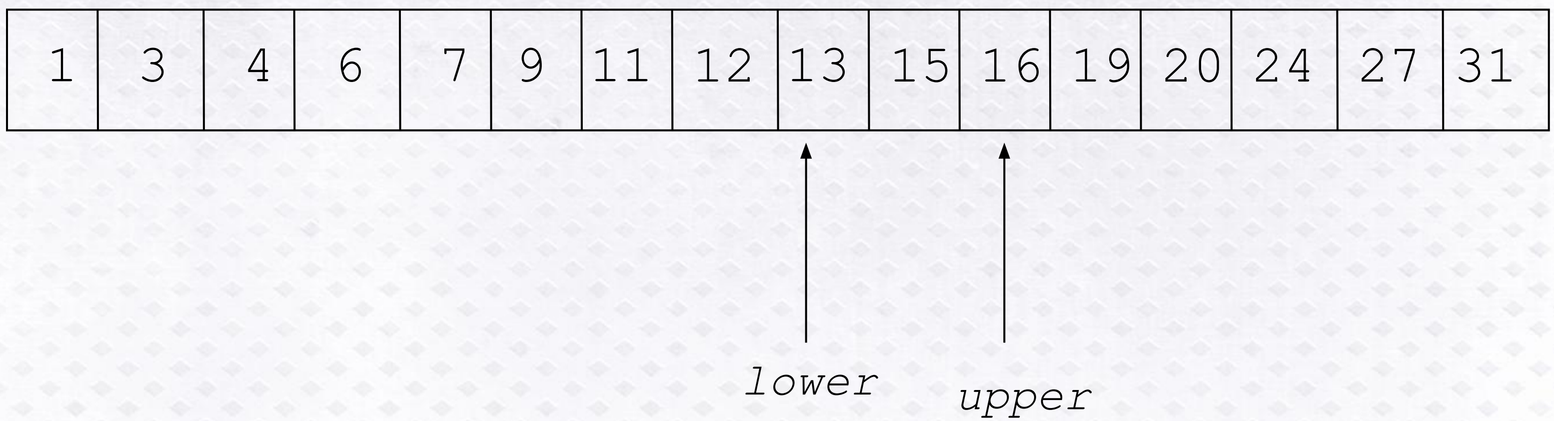
Binary Search Worked Example 3

Target: 14 (Doesn't exist)



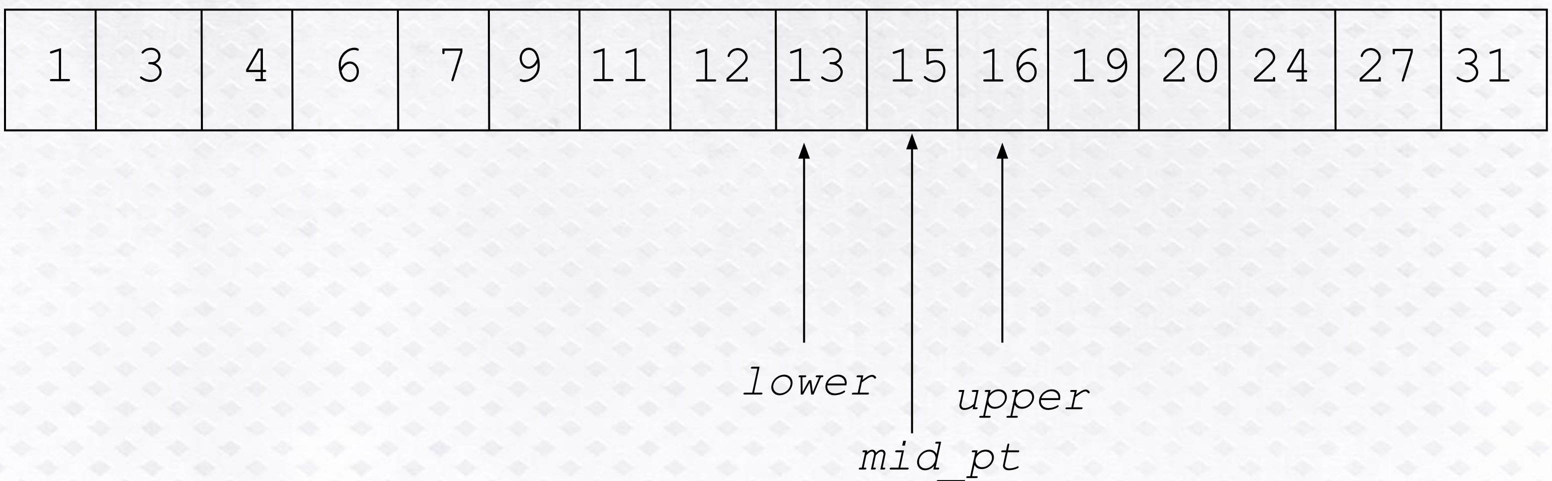
Binary Search
Worked Example 3

Target: 14 (Doesn't exist)



Binary Search Worked Example 3

Target: 14 (Doesn't exist)



Binary Search Worked Example 3

Target: 14 (Doesn't exist)

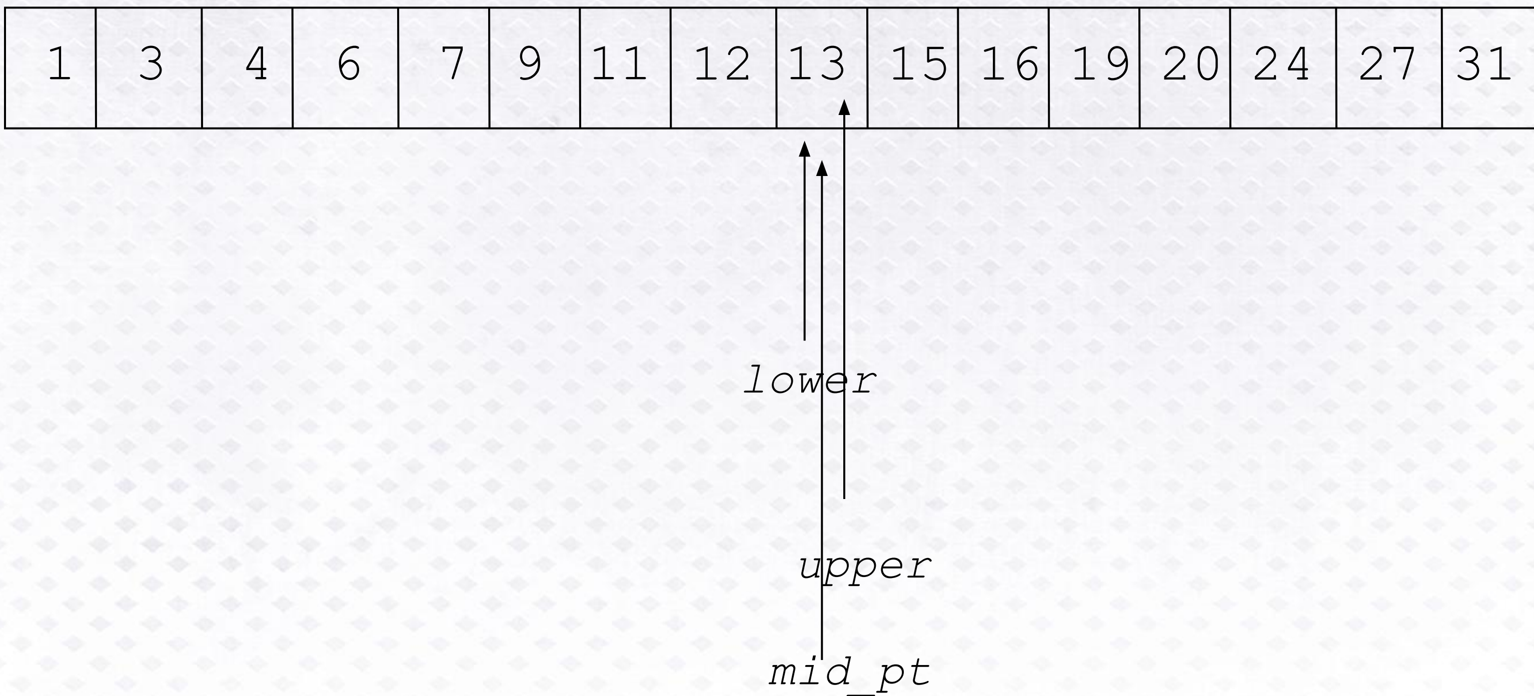
1	3	4	6	7	9	11	12	13	15	16	19	20	24	27	31

lower

upper

Binary Search
Worked Example 3

Target: 14 (Doesn't exist)
NOT FOUND



Recursive Binary Search

```
binarysearch(target, lower, upper)
        found = FALSE
    middle = (upper + lower) / 2
    IF (data[middle] < target)
        binarysearch(target, middle + 1, upper)
    ELSE IF (data[middle] equals target)
        Set found to TRUE
            ELSE
                binarysearch(target, lower, middle – 1)
RETURN found
```

Binary Search

TABLE 7.1 Average Number of Comparisons

Onion: © matka_Wariatka/ShutterStock, Inc.

Length	Sequential Search	Binary Search
10	5.5	2.9
100	50.5	5.8
1000	500.5	9.0
10000	5000.5	12.0

Is a binary search always better?

Binary Search Complexity

What is the complexity of Binary Search?

How many times do we need to check for the entry?

In the example searches with 16 values we calculated the mid point 4 times

Binary Search Complexity

What is the relationship between the number of times we calculate the mid-point and the number of values?

We are repeatedly dividing by 2 so this question is the same as asking how many times can we divide n by 2, or if $n = 2^k$ what is the value of k?

Any number, n, can be written as b^x and x is the logarithm to base b of n.

$$b^x \Rightarrow x = \log_b(n)$$

Note that $\log_b(1) = 0$ for all bases b

Binary Search Complexity

To calculate logarithm base b *of a number* n *precisely requires the use of calculus, but in many cases it is good enough to determine how many times we can divide* n *by* b *until we get a number ≤ 1. This integer is* $\geq \log_b(n)$.

In general for our search algorithm, we will calculate the mid point $\log_2(n)$ *times for an unsuccessful search and between 1 and* $\log_2(n)$ *times for a successful search.*

So the algorithm is: $O(\log_2(n))$ – $O(\log n)$

Some Example Calculations

- $\log_3 27 = 3$ since $27/3/3/3 = 1$
- $\log_4 64 = 3$ since $64/4/4/4 = 1$
- $\log_4 27 \approx 3$ since $27/4/4/4/4 = 0.42$
- $\log_3 64 \approx 4$ since $64/3/3/3/3 = 0.79$

Some useful bases:

10 – this is what the log key on a calculator usually produces

2 – frequently occurs in computer applications as numbers are stored in binary and also problems are often solved by dividing into two sub-problems

e = 2.71828 – the so-called natural logarithm (ln)

What difference does the base make?

n	$\log_2 n$	$\log_{10} n$	$\ln(n)$
1	0	0	0
10	3.32	1	2.30
100	6.64	2	4.61
1000	9.97	3	6.91
10000	13.29	4	9.21

Sorting

Sorting

Arranging items in a collection so that there is an ordering on one (or more) of the fields in the items

Sort Key

The field (or fields) on which the ordering is based

Sorting

Sorting algorithms

Algorithms that order the items in the collection based on the sort key

Stable Sorting Algorithms

Algorithms that don't re-order sorted data

Selection Sort

Given a list of values, put them in ascending order

- *Find the value that comes first,*
  and write it on a second sheet of paper
- *Cross out the value off the original list*
- *Continue this cycle until all the values on the original list have been crossed out and written onto the second list, at which point the second list contains the same items but in sorted order*

Selection Sort

A slight adjustment to this manual approach does away with the need to duplicate space

- As you cross a value off the original list, a free space opens up
- Instead of writing the value found on a second list, exchange it with the value currently in the position where the crossed-off item should go

Selection Sort

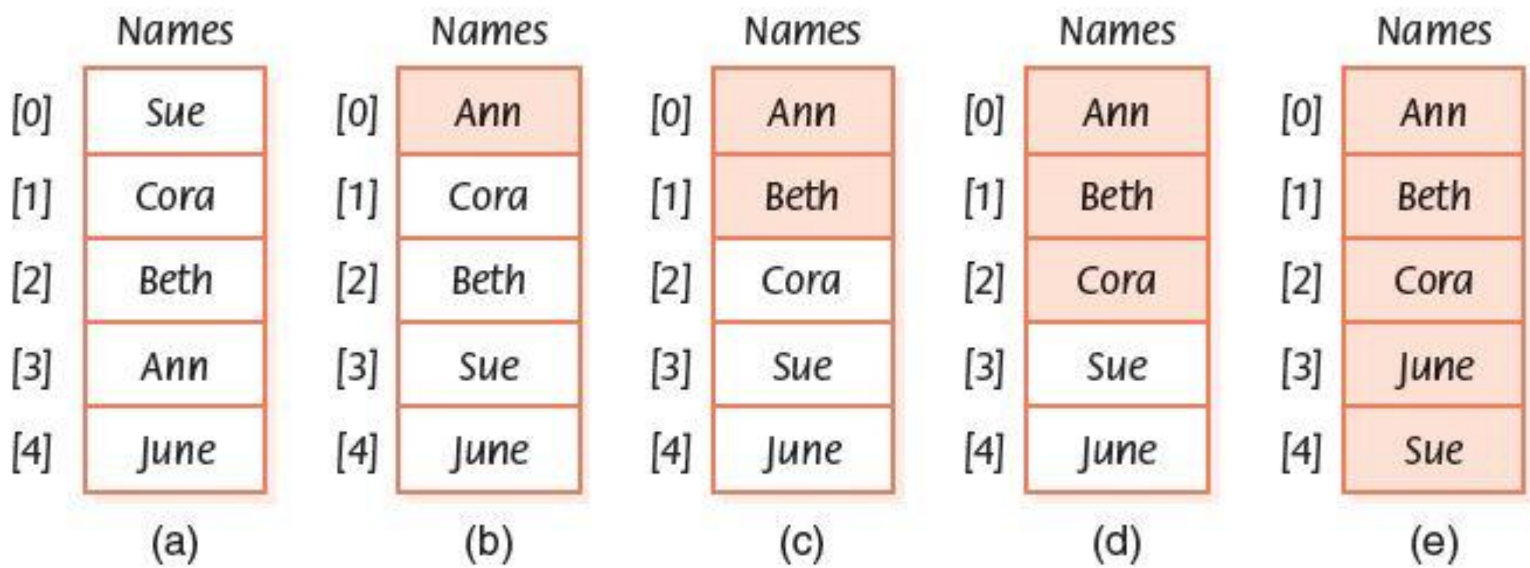


FIGURE 7.11 Examples of selection sort (sorted elements are shaded)

Selection Sort Worked Example

67	*42*	*23*	*97*	*18*	*35*	*48*	*56*	*51*	*85*	*79*	*25*

Selection Sort Worked Example

67	*42*	*23*	*97*	*18*	*35*	*48*	*56*	*51*	*85*	*79*	*25*

Current Index

i^{th} smallest value

Selection Sort Worked Example

18	42	23	97	67	35	48	56	51	85	79	25

Current Index

i^{th} smallest value

Selection Sort Worked Example

18	*23*	*42*	*97*	*67*	*35*	*48*	*56*	*51*	*85*	*79*	*25*

Current Index

i^{th} smallest value

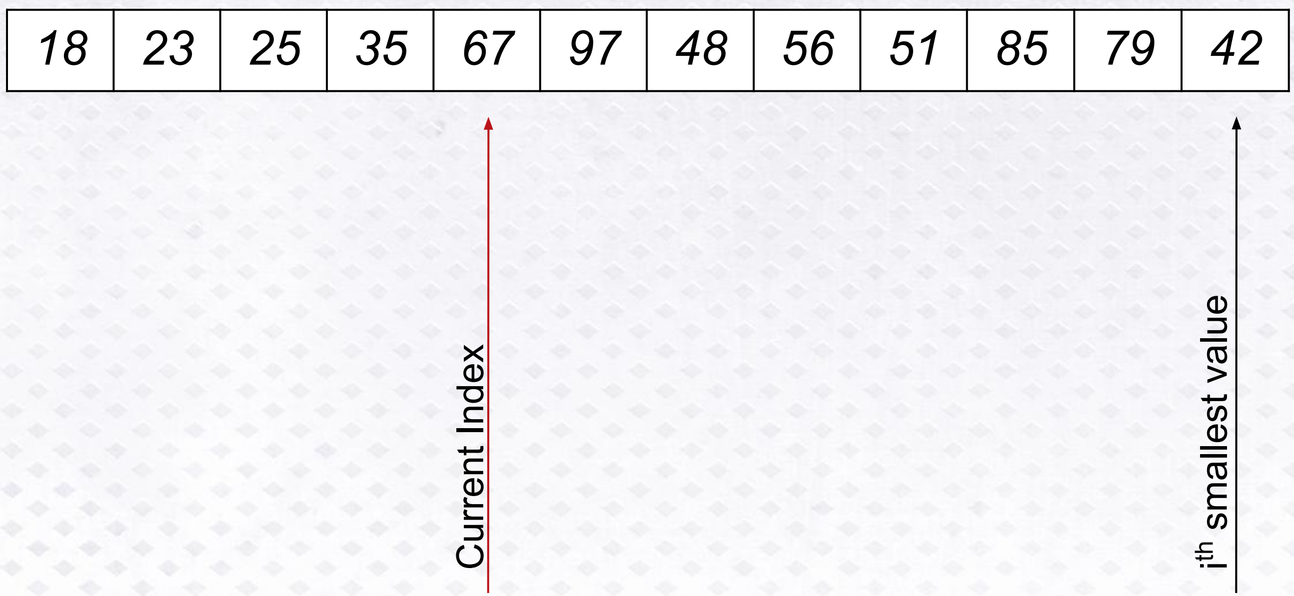
Selection Sort Worked Example

18	23	25	97	67	35	48	56	51	85	79	42

Current Index

i^{th} smallest value

Selection Sort Worked Example



Selection Sort Worked Example

18	23	25	35	42	97	48	56	51	85	79	67

Current Index

i^{th} smallest value

Selection Sort Worked Example

18	23	25	35	42	48	97	56	51	85	79	67

Current Index

i^{th} smallest value

Selection Sort
Worked Example

18	23	25	35	42	48	51	56	97	85	79	67

Current Index

i^{th} smallest value

Selection Sort
Worked Example

18	23	25	35	42	48	51	56	97	85	79	67

Current Index

i^{th} smallest value

Selection Sort Worked Example

18	*23*	*25*	*35*	*42*	*48*	*51*	*56*	*67*	*85*	*79*	*97*

Current Index

i^{th} smallest value

Selection Sort Worked Example

18	*23*	*25*	*35*	*42*	*48*	*51*	*56*	*67*	*79*	*85*	*97*

Current Index

i^{th} smallest value

Selection Sort Worked Example

18	*23*	*25*	*35*	*42*	*48*	*51*	*56*	*67*	*79*	*85*	*97*

Current Index

i^{th} smallest value

Selection Sort

Selection Sort

Set firstUnsorted to 0

WHILE (not sorted yet)

    Find smallest unsorted item

    Swap firstUnsorted item with the smallest

    Set firstUnsorted to firstUnsorted + 1

Not sorted yet

current < length – i

Selection Sort

Find smallest unsorted item

```
Set indexOfSmallest to firstUnsorted
Set index to firstUnsorted + 1
WHILE (index <= length – 1)
    IF (data[index] < data[indexOfSmallest])
            Set indexOfSmallest to index
    Set index to index + 1
Set index to indexOfSmallest
```

Selection Sort

Swap firstUnsorted with smallest

Set tempItem to data[firstUnsorted]

Set data[firstUnsorted] to data[indexOfSmallest]

Set data[indexOfSmallest] to tempItem

Selection Sort

```
FOR index1 ← 0 to length - 2
    min_pos = index1
    FOR index2 ← index1 + 1 to length - 1
    IF data[index2] < data[min_pos]
        min_pos = index2

    temp = data[index1]
    data[index1] = data[min_pos]
    data[min_pos] = temp
```

Exchange

Selection Sort: Complexity

What is the complexity of this algorithm?

Outer loop executes: n-1 times

Inner loop executes: 1 + 2 + … + n-1 times

Complexity is $O(n^2)$

Data movements: Maximum of n exchanges

Bubble Sort

Bubble Sort uses the same strategy:

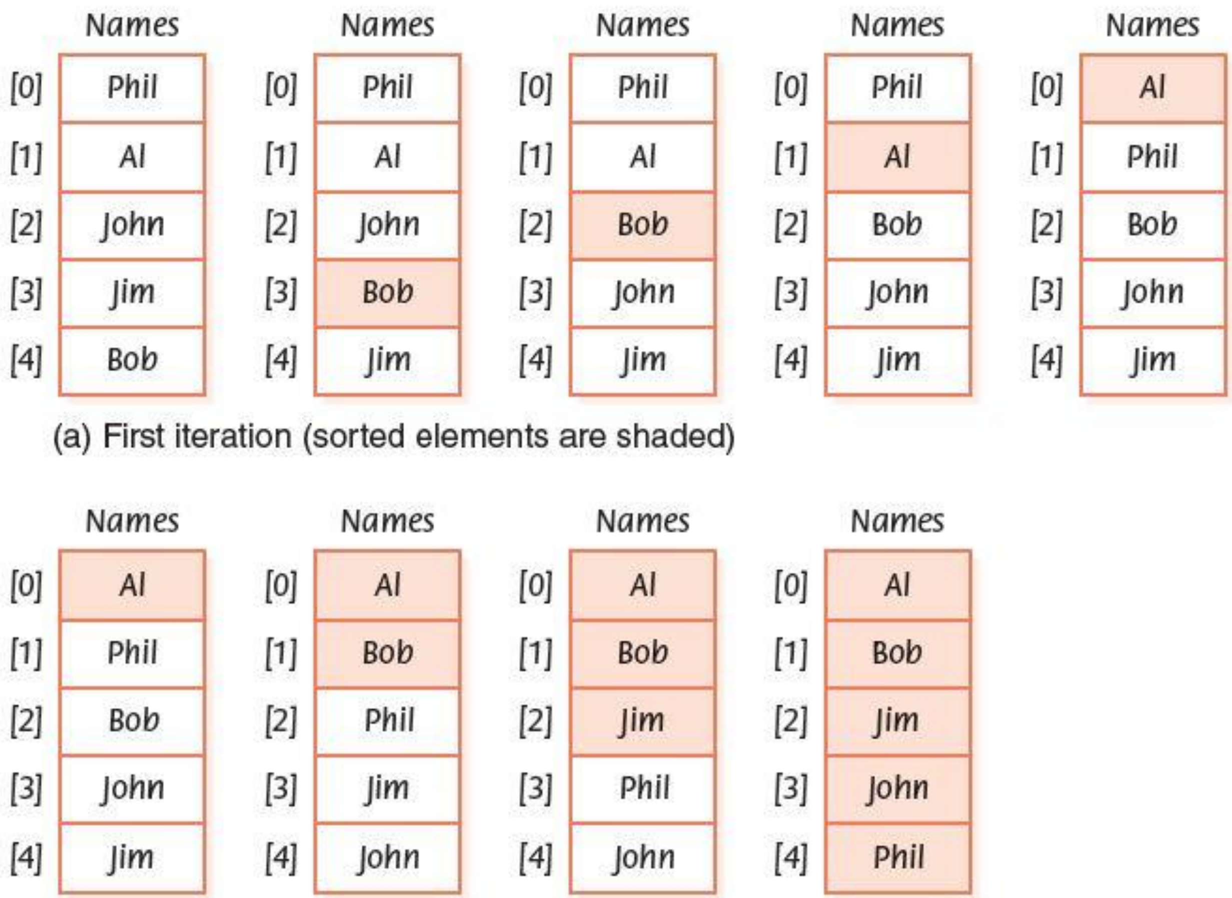
Find the next item

Put it into its proper place

But uses a different scheme for finding the next item

Starting with the last list element, compare successive pairs of elements, swapping whenever the elements of the pair are not sorted

Bubble Sort



(a) First iteration (sorted elements are shaded)

(b) Remaining iterations (sorted elements are shaded)

FIGURE 7.12 Examples of a bubble sort

Bubble Sort

Bubble sort is very slow!

Can you see a way to make it faster?

Under what circumstances is bubble sort fast?

Bubble Sort

Bubble Sort

Set firstUnsorted to 0

Set index to firstUnsorted + 1

Set swap to TRUE

WHILE (index < length AND swap)

    Set swap to FALSE

    "Bubble up" the smallest item in unsorted part

    Set firstUnsorted to firstUnsorted + 1

Bubble Sort

Bubble up

```
Set index to length – 1
WHILE (index > firstUnsorted + 1)
    IF (data[index] < data[index – 1])
        Swap data[index] and data[index – 1]
        Set swap to TRUE
    Set index to index - 1
```

Bubble Sort
Worked Example

Consider the following list of integers to sort:

48 35 67 23 51 56 25 18 42

Swap occurred this pass:

Bubble Sort Worked Example

48	35	67	23	51	56	25	18	42

Swap occurred this pass: FALSE

Bubble Sort Worked Example

48 35 67 23 51 56 25 18 42

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

48 35 67 23 51 56/18 18/56 25 42

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

48 35 67 23 18 51 56 25 42

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

48 35 67 18 28 51 56 25 42

Swap

Swap occurred this pass: TRUE

Bubble Sort
Worked Example

48 35 67/17 67/17 23 51 56 25 42

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

48 35 16 67 23 51 56 25 42

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

48 48 35 67 23 51 56 25 42

Swap

Swap occurred this pass: TRUE

Bubble Sort
Worked Example

18	48	35	67	23	51	56	25	42

Swap occurred this pass: FALSE

Bubble Sort Worked Example

18 | 48 35 67 23 51 56 25 42

Swap occurred this pass: FALSE

Bubble Sort Worked Example

18 | 48 35 67 23 51 56 26 42

Swap

Swap occurred this pass: FALSE TRUE

Bubble Sort Worked Example

18 | 48 35 67 23 25 21 56 42

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18	48	35	67	23	25	51	56	42

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 | 48 35 67 23 25 51 56 42

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 | 48 25 35 67 25 51 56 42

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 | 48 28 35 67 25 51 56 42

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 23 | 48 35 67 25 51 56 42

Swap occurred this pass: FALSE

Bubble Sort
Worked Example

18 23 | 48 35 67 25 51 50 42

Swap

Swap occurred this pass: FALSE

Bubble Sort Worked Example

18 23 48 35 67 25 51 42 56

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18	23	48	35	67	25	42	51	56

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 23 | 48 35 65 25 42 51 56

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18   23 | 48   25   35   67   42   51   56

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 23 46 25 35 67 42 51 56

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 23 25 | 48 35 67 42 51 56

Swap occurred this pass: FALSE

Bubble Sort
Worked Example

18 23 25 | 48 35 67 42 51 56

Swap occurred this pass: FALSE

Bubble Sort
Worked Example

18 23 25 | 48 35 67 42 51 56

Swap occurred this pass: FALSE

Bubble Sort
Worked Example

18 23 25 | 48 35 42 67 51 56

Swap

Swap occurred this pass: TRUE

Bubble Sort
Worked Example

18 23 25 | 48 35 42 67 51 56

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 23 25 | 35 46 42 67 51 56

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18	23	25	35	|	48	42	67	51	56

Swap occurred this pass: FALSE

Bubble Sort Worked Example

18 23 25 35 | 48 42 67 51 56

Swap occurred this pass: FALSE

Bubble Sort Worked Example

18 23 25 35 | 48 42 67 67 56

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 23 25 35 | 48 42 51 67 56

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 23 25 35 42 48 51 67 56

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 23 25 35 42 | 48 51 67 56

Swap occurred this pass: FALSE

Bubble Sort Worked Example

18 23 25 35 42 | 48 51 67 66

Swap

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 23 25 35 42 | 48 51 56 67

Swap occurred this pass: TRUE

Bubble Sort Worked Example

18 23 25 35 42 | 48 51 56 67

Swap occurred this pass: TRUE

Bubble Sort
Worked Example

18	23	25	35	42	48	51	56	67

Swap occurred this pass: FALSE

Bubble Sort Worked Example

18 23 25 35 42 48 | 51 56 67

Swap occurred this pass: FALSE

Bubble Sort Worked Example

18 23 25 35 42 48 | 51 56 67

Swap occurred this pass: FALSE

Bubble Sort Worked Example

18	23	25	35	42	48	51	56	67

Done!

Bubble Sort: Complexity

What is the complexity of this algorithm?

Outer loop executes: n-1 times

Inner loop executes: 1 + 2 + … + n-1 times

Complexity is O(n^2)

Data movements: Maximum of $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ *exchanges*

Selection vs Bubble Sort

Compared to Selection Sort, Bubble Sort has the same number of comparisons (on average), but many more data exchanges.

Can we improve the algorithm?

When might Bubble Sort be fast?

What about when the data is (almost) sorted?

Insertion Sort

Expressing sorting as a Divide and Conquer strategy, we could say

To sort a list:

- *Create a sorted list consisting of smallest item*
- *Repeatedly add the next smallest item to create a sorted list of all items*

Insertion sort works by adding items successively to a sorted list, but removes the constraint that the list so far contains all the smallest items.

Insertion Sort

- *A list of length 1 is sorted*
- *A list of length 2 is sorted by comparing (and exchanging if necessary the two entries*
- *A sorted list of length 3 can be created by adding a 3rd item into the correct place in a sorted list of length 2*
- *A sorted list of length n can be created by adding a 3rd item into the correct place in a sorted list of length n-1*

Insertion Sort

The item being added to the sorted portion can be bubbled up as in the bubble sort

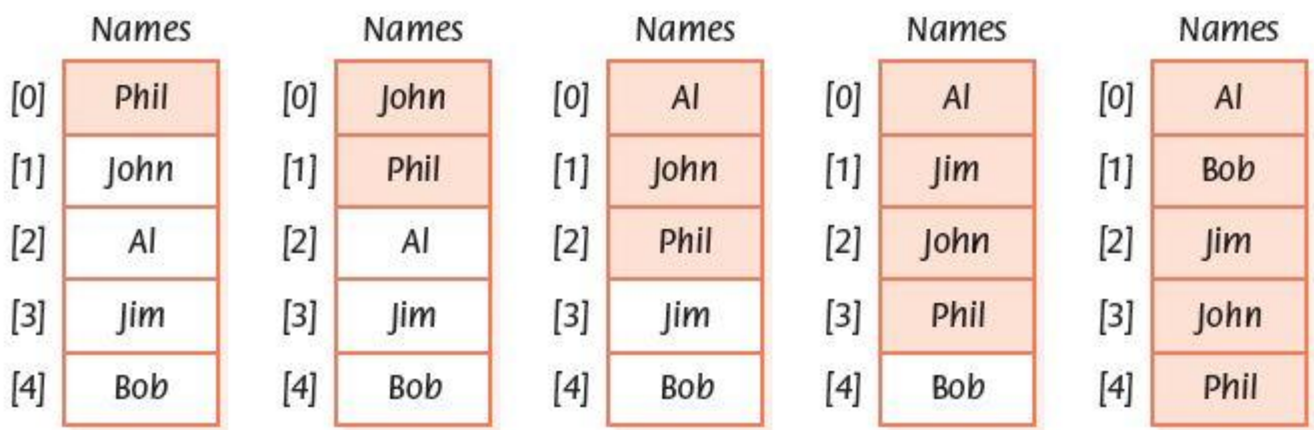


FIGURE 7.13 Insertion sort

Insertion Sort

```
InsertionSort
Set current to 1
WHILE (current < length)
    Set index to current
    Set placeFound to FALSE
    WHILE (index > 0 AND NOT placeFound)
        IF (data[index] < data[index – 1])
            Swap data[index] and data[index – 1]
            Set index to index – 1
        ELSE
            Set placeFound to TRUE
    Set current to current + 1
```

Insertion Sort Worked Example

Consider the following list of integers to sort:

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**

Insertion Sort Worked Example

Consider the following list of integers to sort:

48 35 67 23 51 56 25 18 42

48

Insertion Sort Worked Example

Consider the following list of integers to sort:

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**
48								
$\boxed{35}$	48							

Insertion Sort
Worked Example

Consider the following list of integers to sort:

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**
48								
35	48							
35	48	$\boxed{67}$						

Insertion Sort Worked Example

Consider the following list of integers to sort:

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**
48								
35	48							
35	48	67						
23	35	48	67					

Insertion Sort Worked Example

Consider the following list of integers to sort:

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**
48								
35	48							
35	48	67						
23	35	48	67					
23	35	48	51	67				

Insertion Sort Worked Example

Consider the following list of integers to sort:

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**
48								
35	48							
35	48	67						
23	35	48	67					
23	35	48	51	67				
23	35	48	51	56	67			

Insertion Sort Worked Example

Consider the following list of integers to sort:

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**
48								
35	48							
35	48	67						
23	35	48	67					
23	35	48	51	67				
23	35	48	51	56	67			
23	25	35	48	51	56	67		

Insertion Sort Worked Example

Consider the following list of integers to sort:

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**
48								
35	48							
35	48	67						
23	35	48	67					
23	35	48	51	67				
23	35	48	51	56	67			
23	25	35	48	51	56	67		
18	23	25	35	48	51	56	67	

Insertion Sort Worked Example

Consider the following list of integers to sort:

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**
48								
35	48							
35	48	67						
23	35	48	67					
23	35	48	51	67				
23	35	48	51	56	67			
23	25	35	48	51	56	67		
18	23	25	35	48	51	56	67	
18	23	25	35	$\boxed{42}$	48	51	56	67

Subprogram Statements

We can give a section of code a name and use that name as a statement in another part of the program

When the name is encountered, the processing in the other part of the program halts while the named code is executed

Remember?

Subprogram Statements

What if the subprogram needs data from the calling unit?

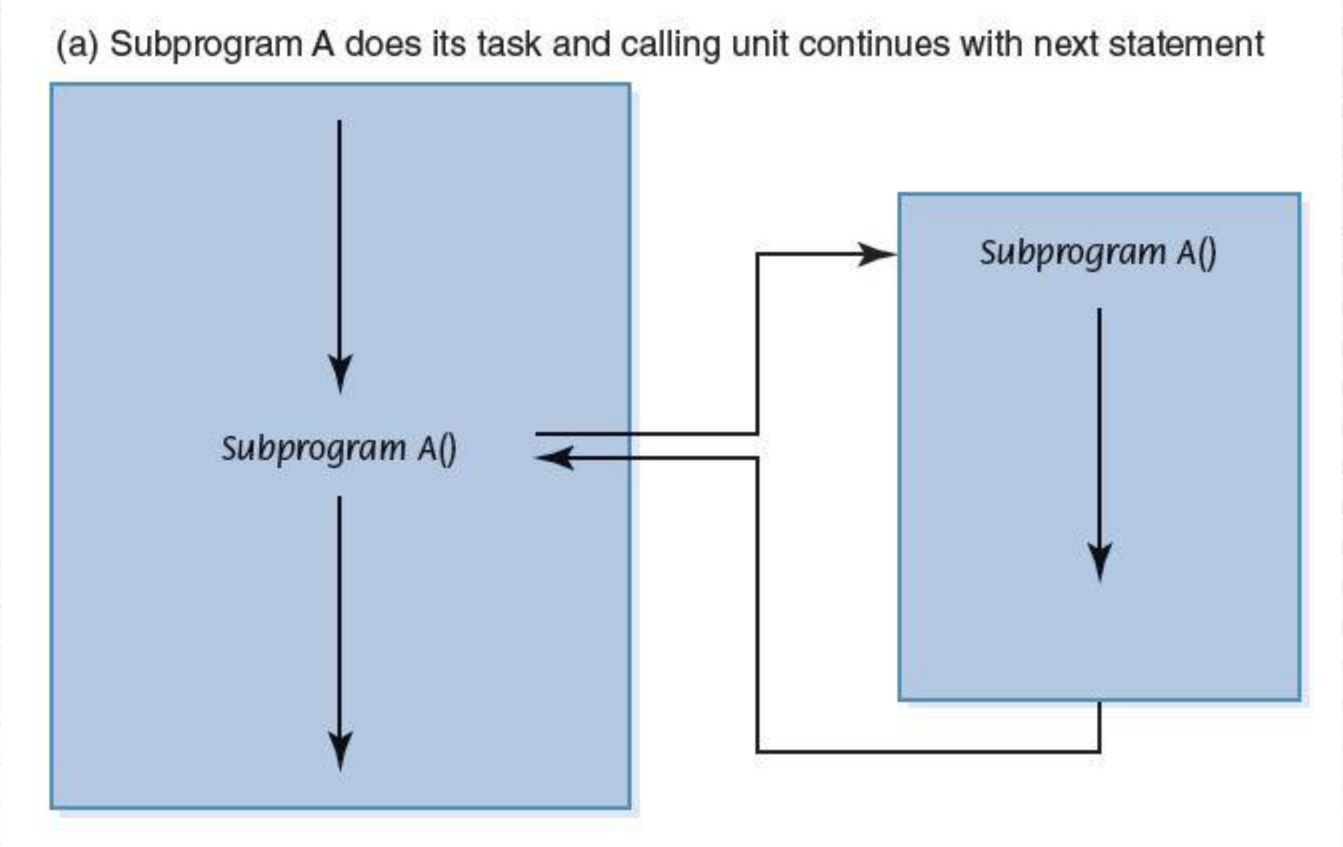
Parameters

*Identifiers listed in parentheses beside the subprogram declaration; sometimes called **formal parameters***

Arguments

*Identifiers listed in parentheses on the subprogram call; sometimes called **actual parameters***

Subprogram Statements



Recursion

Recursion

The ability of a subprogram to call itself

Base case

The case to which we have an answer

General case

The case that expresses the solution in terms of a call to itself with a smaller version of the problem

Recursion

For example, the factorial of a number is defined as the number times the product of all the numbers between itself and 0:

$$N! = N * (N - 1)!$$

Base case

Factorial(0) = 1 (0! is 1)

General Case

*Factorial(N) = N * Factorial(N-1)*

Recursion

```
Write “Enter n”
Read n
Set result to Factorial(n)
Write result + “ is the factorial of “ + n

Factorial(n)
IF (n equals 0)
    RETURN 1
ELSE
    RETURN n * Factorial(n-1)
```

Merge Sort

Recall that we expressed Binary Search recursively

We could express all our sorting algorithms recursively

But: All our current algorithms build up a sorted list one entry at a time

What if we split the list to be sorted into half (like we did with Binary Search)?

Merge Sort

Break the problem down into:

- *split the list into two*
- *sort each half*
- *merge the two sorted halves together*

How do we split our list?

How do we merge?

Merge Sort

```
MergeSort(indata, outdata, lower, upper)
    IF lower equals right
        outdata[lower] = indata[lower]
    ELSE
        mid_pt = (lower + upper) // 2
        MergeSort(indata, outdata, lower, mid_pt)
        MergeSort(indata, outdata, mid_pt + 1, upper)
        Merge(outdata, indata, lower, mid_pt, upper)
```

Merge Sort

Merge process:

Merge(indata, outdata, lower, mid_pt, upper)

Set pointer variables

WHILE Loop 1 - Merge two halves of indata

WHILE Loop 2 - Append remainder of first half

WHILE Loop 3 - Append remainder of second half

Merge Sort

Merge process:

Merge(indata, outdata, lower, mid_pt, upper)

```
    inptr1 = lower; inptr2 = mid_pt + 1; outptr = lower
Loop 1:
    WHILE (inptr1 <= mid_pt AND inptr2 <= upper)
        IF (indata[inptr1] <= indata[inptr2])
            outdata[outptr] = indata[inptr1]
            inptr1 = inptr1 + 1
        ELSE
            outdata[outptr] = indata[inptr2]
            inptr2 = inptr2 + 1
        outptr = outptr + 1
```

Merge Sort

Merge process continued:

```
Loop 2:     WHILE (inptr1 <= mid_pt)
                outdata[outptr] = indata[inptr1]
                inptr1 = inptr1 + 1
                outptr = outptr + 1
Loop 3:     WHILE (inptr2 <= upper)
                outdata[outptr] = indata[inptr2]
                inptr2 = inptr2 + 1
                outptr = outptr + 1
```

Merge Sort Worked Example

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42

48	35	67	23	51	56	25	18	42

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42

48	35	67	23	51	56	25	18	42

48	35	67	23

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42

48	35	67	23	51	56	25	18	42

48	35	67	23

48	35

Merge Sort Worked Example

48 35 67 23 51 56 25 18 42

48 35 67 23	51 56 25 18 42

48 35	67 23

48	35

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42

48 35 67 23	51 56 25 18 42

35 48	67 23

Merge Sort Worked Example

48 35 67 23 51 56 25 18 42

48 35 67 23	51 56 25 18 42

35 48	67 23

67	23

Merge Sort Worked Example

48 35 67 23 51 56 25 18 42

48 35 67 23 | 51 56 25 18 42

35 48 | 67 23

67 | 23

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
48	35	67	23	51	56	25	18	42
35	48	23	67					

Merge Sort Worked Example

48 35 67 23 51 56 25 18 42

| 48 35 67 23 | 51 56 25 18 42 |

| 35 48 | 23 67 |

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
23	35	48	67	51	56	25	18	42

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
23	35	48	67	51	56	25	18	42
				51	56	25	18	42

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
23	35	48	67	51	56	25	18	42
				51	56	25	18	42
				51	56			

Merge Sort Worked Example

48 35 67 23 51 56 25 18 42

23 35 48 67 | 51 56 25 18 42

51 56 | 25 18 42

51 | 56

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
23	35	48	67	51	56	25	18	42
				51	56	25	18	42

Merge Sort Worked Example

48 35 67 23 51 56 25 18 42

| 23 35 48 67 | | 51 56 25 18 42 |

| 51 56 | | 25 18 42 |

| 25 | | 18 42 |

Merge Sort Worked Example

48 35 67 23 51 56 25 18 42

23 35 48 67	51 56 25 18 42

51 56	25 18 42

$\boxed{25}$	18 42

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
23	35	48	67	51	56	25	18	42
				51	56	25	18	42
						25	18	42

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
23	35	48	67	51	56	25	18	42
				51	56	25	18	42
						25	18	42
							18	42

Merge Sort Worked Example

48 35 67 23 51 56 25 18 42

23 35 48 67 | 51 56 25 18 42

51 56 | 25 18 42

25 | 18 42

18 | 42

Merge Sort Worked Example

48 35 67 23 51 56 25 18 42

[23 35 48 67] [51 56 25 18 42]

[51 56] [25 18 42]

[25] [18 42]

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
23	35	48	67	51	56	25	18	42
				51	56	25	18	42
						25	18	42

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
23	35	48	67	51	56	25	18	42
				51	56	18	25	42

Merge Sort Worked Example

48 35 67 23 51 56 25 18 42

23	35	48	67	51	56	25	18	42
				51	56	18	25	42

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
23	35	48	67	18	25	42	51	56

Merge Sort Worked Example

48	35	67	23	51	56	25	18	42
23	35	48	67	18	25	42	51	56

Merge Sort Worked Example

18 23 25 35 42 48 51 56 67

Sorted!

Merge Sort

What is the complexity of this algorithm?

- *At each stage the size of the list is halved*
- *For a list of length n we can halve it log(n) times*
- *At each merge stage the time is proportional to the length of the lists being merged – O(n)*
- *Since we have log(n) layers of merging the overall complexity is*
    - *O(n log(n))*

Merge Sort: Doing better

Is this the best we can do?

Consider sorting 3 elements a, b, and c.

Possible orderings are

$a < b < c$	$a < c < b$
$b < a < c$	$b < c < a$
$c < a < b$	$c < b < a$

In general for n entries there are n! possible orderings

Merge Sort: Doing better

In a sort based on comparing elements at each stage two elements are compared and they are either:

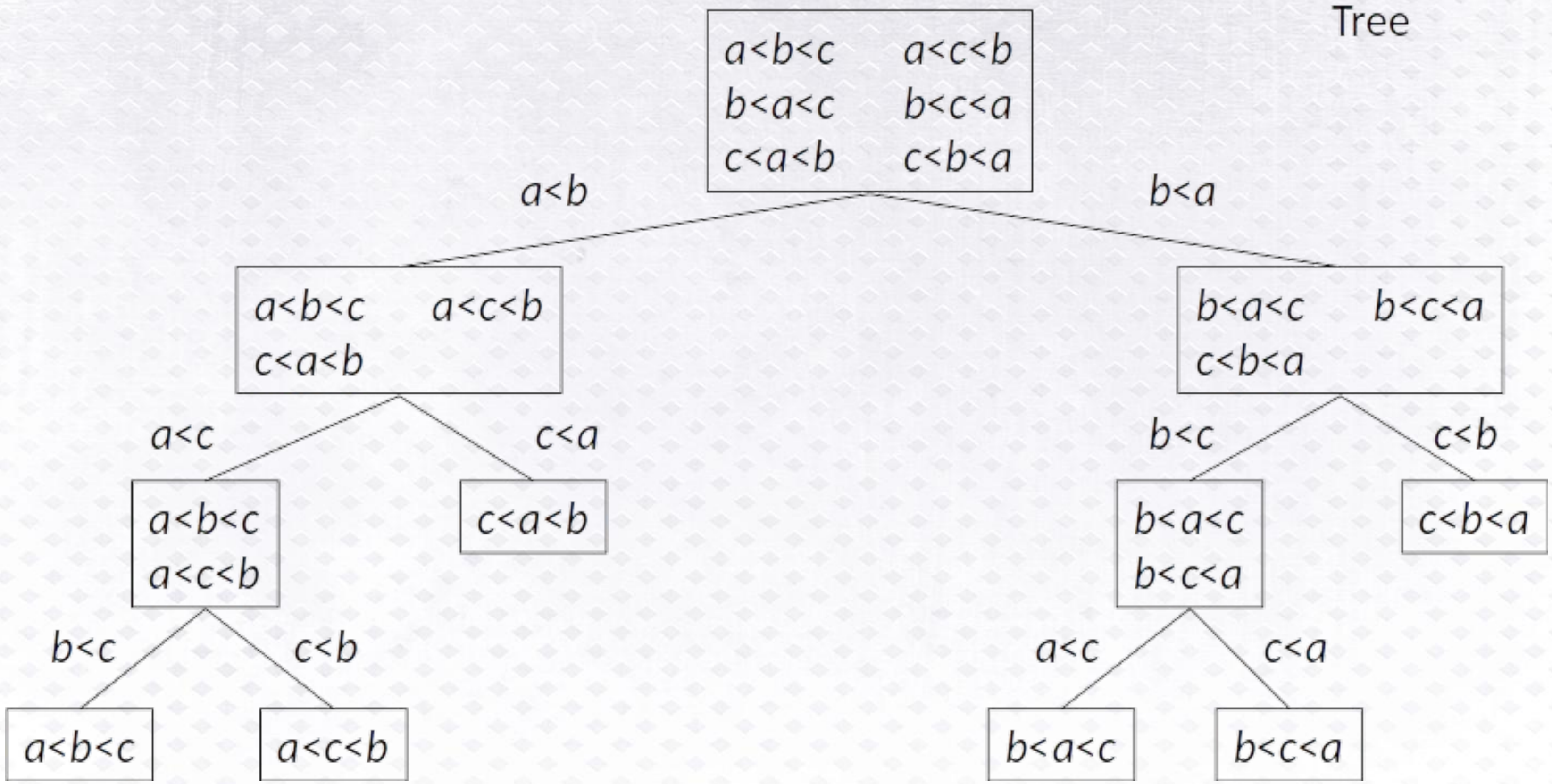
- *in the correct order*
- *in the wrong order and must be exchanged*

Another way of looking at this is that each comparison eliminates half the possible orderings

Merge Sort: Doing better

Consider our example of three values

Decision Tree



Merge Sort: Doing better

How big is this Decision Tree?

- *Top layer – one set of possible orderings*
- *Next layer – two sets of possible orderings*
- *n'th layer – 2n sets of possible orderings*

For n elements to be sorted we had n! permutations

To find sorted arrangement (ie ordering sets containing one entry) we need n! possible orderings at one level

We need log(n!) layers

Merge Sort: Doing better

This means we need at least n log(n) comparisons:

$$\begin{aligned} \log(n!) &= \log(n(n-1)(n-2)\ldots(2)(1)) \\ &= \log(n) + \log(n-1) + \ldots + \log(1) \\ &\geq (n \log(n)) \end{aligned}$$

Lower bound for sort based on comparisons is (n log(n))

But with merge sort we need an extra copy of the values (insertion, selection and bubble sort only required space for one value when swapping entries)

Merge Sort: Doing better

Can we find another divide and conquer algorithm similar to merge sort, ie based on splitting the data in half, which doesn't require so much space?

Quicksort

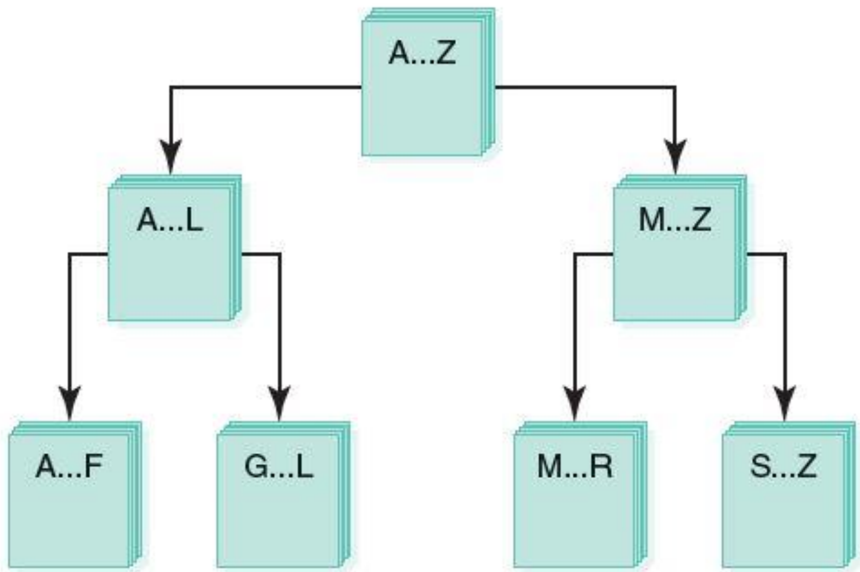


FIGURE 7.15 Ordering a list using the Quicksort algorithm

Quicksort algorithm

With each attempt to sort the stack of data elements, the stack is divided at a splitting value, splitVal, *and the same approach is used to sort each of the smaller stacks (a smaller case)*

Process continues until the small stacks do not need to be divided further (the base case)

The variables first *and* last *in Quicksort algorithm reflect the part of the array* data *that is currently being processed*

Quicksort

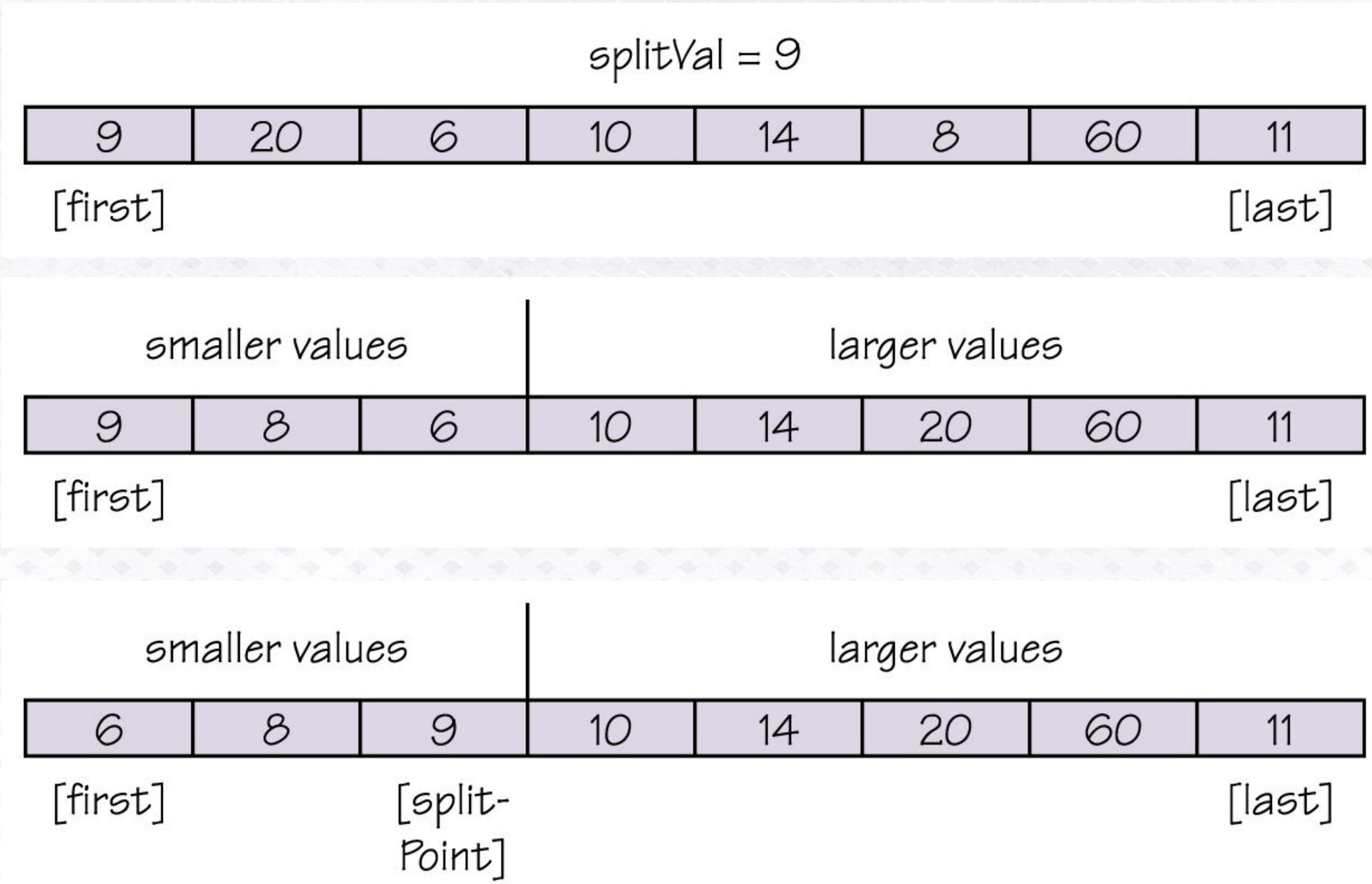
```
Quicksort(first, last)
IF (first < last)          // There is more than one item
      Select splitVal
      Split (splitVal)     // Array between first and
                           //   splitPoint–1 <= splitVal
                           //  data[splitPoint] = splitVal
                           // Array between splitPoint + 1
                                //   and last > splitVal
      Quicksort (first, splitPoint - 1)
      Quicksort (splitPoint + 1, last)
```

Quicksort

Split(splitVal)

```
Set left to first + 1
Set right to last
WHILE (left <= right)
    Increment left until data[left] > splitVal OR left > right
    Decrement right until data[right] < splitVal
        OR left > right
    IF(left < right)
        Swap data[left] and data[right]
Set splitPoint to right
Swap data[first] and data[splitPoint]
Return splitPoint
```

Quicksort



splitVal = 9
9
20
6
10
14
8
60
11
[first]
[last]
smaller values
larger values
9
8
6
10
14
20
60
11
[first]
[last]
smaller values
larger values
6
8
9
10
14
20
60
11
[first]
[split-Point]
[last]

Quicksort

(a) Initialization

9	20	6	10	14	8	60	11
[first]	[left]						[right]

(b) Increment *left* until list[*left*] > *splitVal* or *left* > *right*

9	20	6	10	14	8	60	11
[first]	[left]						[right]

(c) Decrement *right* until list[*right* > *splitVal* or *left* > *right*

9	20	6	10	14	8	60	11
[first]	[left]				[right]		

(d) Swap list[*left*] and list[*right*]; move *left* and *right* toward each other

9	8	6	10	14	20	60	11
[first]		[left]		[right]			

(e) Increment *left* until list[*left*] > *splitVal* or *left* > *right*
Decrement *right* until list[*right*] <= *splitVal* or *left* > *right*

9	8	6	10	14	20	60	11
[first]		[right]	[left]				

(f) *Left* > *right* so no swap occurs within the loop
Swap list[*first*] and list[*right*]

6	8	9	10	14	20	60	11
[first]		[right] (splitPoint)					

FIGURE 7.16 Splitting algorithm

Quick Sort Worked Example

48	35	67	23	51	56	25	18	42

Quick Sort Worked Example

First = 0

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0 Left = 1

48 35 67 23 51 56 25 18 42

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0 Left = 1 Right = 8

48 35 67 23 51 56 25 18 42

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0　　Left = 2　　Right = 8

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**

Split Point = 0
Split Value = 48

Quick Sort Worked Example

SWAP

First = 0 Left = 2 Right = 8

48	**35**	**67**	**23**	**51**	**56**	**25**	**18**	**42**

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0   Left = 2   Right = 8

48	**35**	**42**	**23**	**51**	**56**	**25**	**18**	**67**

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0 Left = 3 Right = 8

48 35 42 23 51 56 25 18 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0 Left = 4 Right = 8

48 35 42 23 51 56 25 18 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0 Left = 4 Right = 7

48 35 42 23 51 56 25 18 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0 Left = 4 SWAP Right = 7

48 35 42 23 51 56 25 18 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0

Left = 4

Right = 7

48 35 42 23 18 56 25 51 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0 Left = 5 Right = 7

48 35 42 23 18 56 25 51 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0 Left = 5 Right = 6

48 35 42 23 18 56 25 51 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

SWAP

First = 0

Left = 5 Right = 6

48 35 42 23 18 56 25 51 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

First = 0

Left = 5 Right = 6

48 35 42 23 18 25 56 51 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

Left = 6

First = 0

Right = 6

48 35 42 23 18 25 56 51 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

Left = 6

First = 0

Right = 5

48 35 42 23 18 25 56 51 67

Split Point = 0
Split Value = 48

Quick Sort Worked Example

Left = 6

First = 0

Right = 5

48 35 42 23 18 25 56 51 67

Split Point = 5
Split Value = 25

Quick Sort Worked Example

Left = 6

First = 0

Right = 5

48 35 42 23 18 25 56 51 67

SWAP

Split Point = 5
Split Value = 25

Quick Sort Worked Example

Left = 6

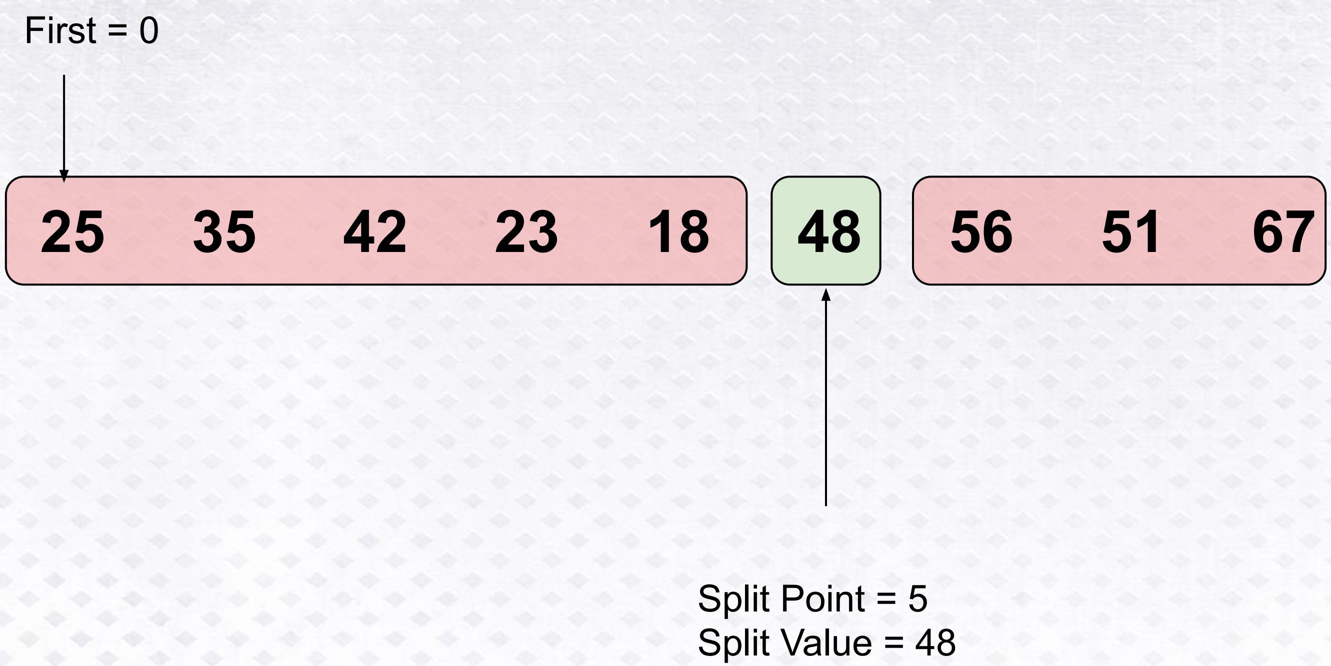
First = 0

Right = 5

25 35 42 23 18 48 56 51 67

Split Point = 5
Split Value = 48

Quick Sort Worked Example



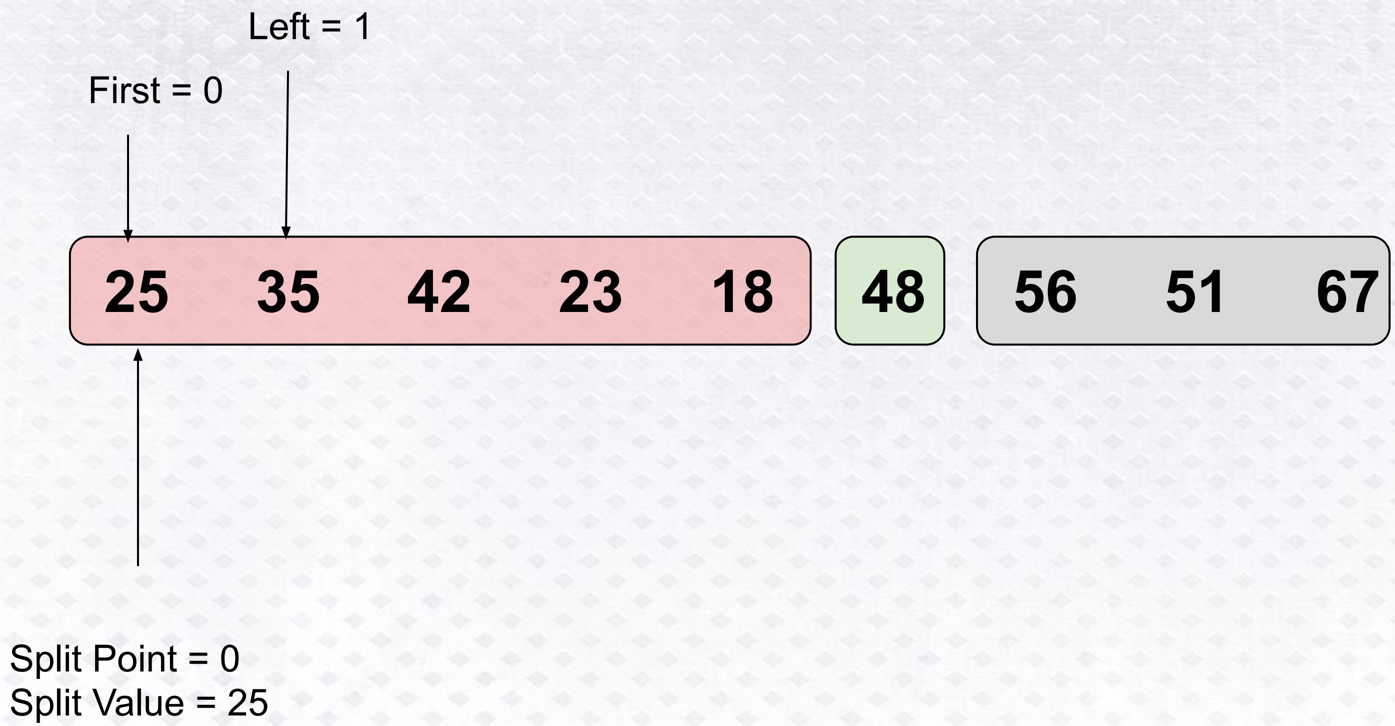
Quick Sort Worked Example

First = 0

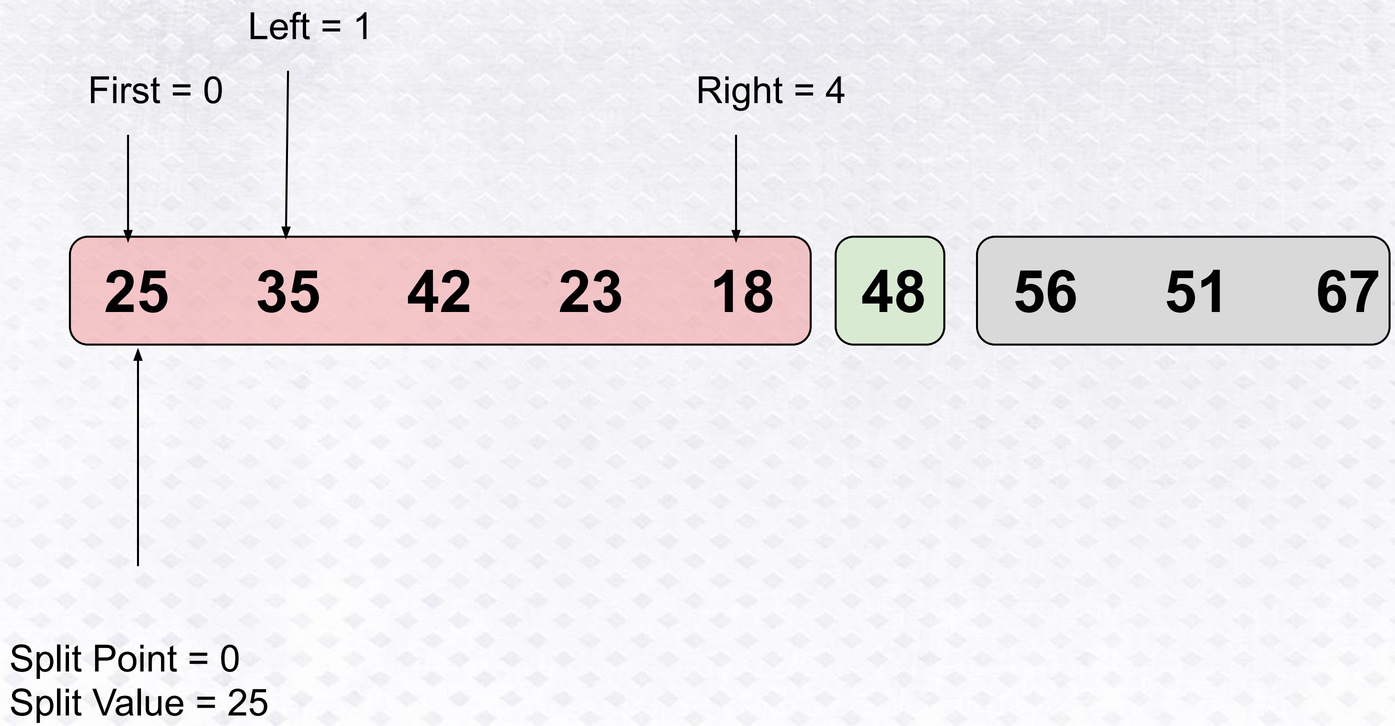
25	35	42	23	18	48	56	51	67

Split Point = 0
Split Value = 25

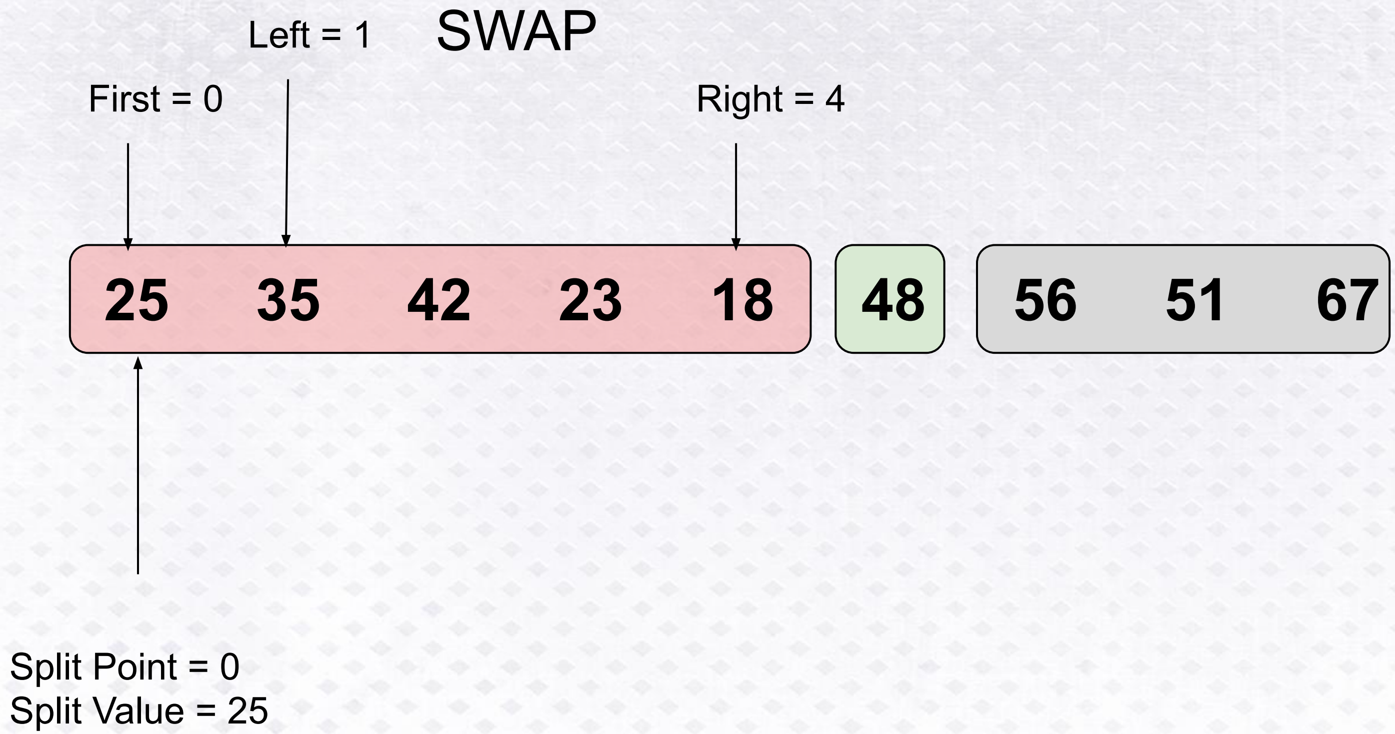
Quick Sort Worked Example



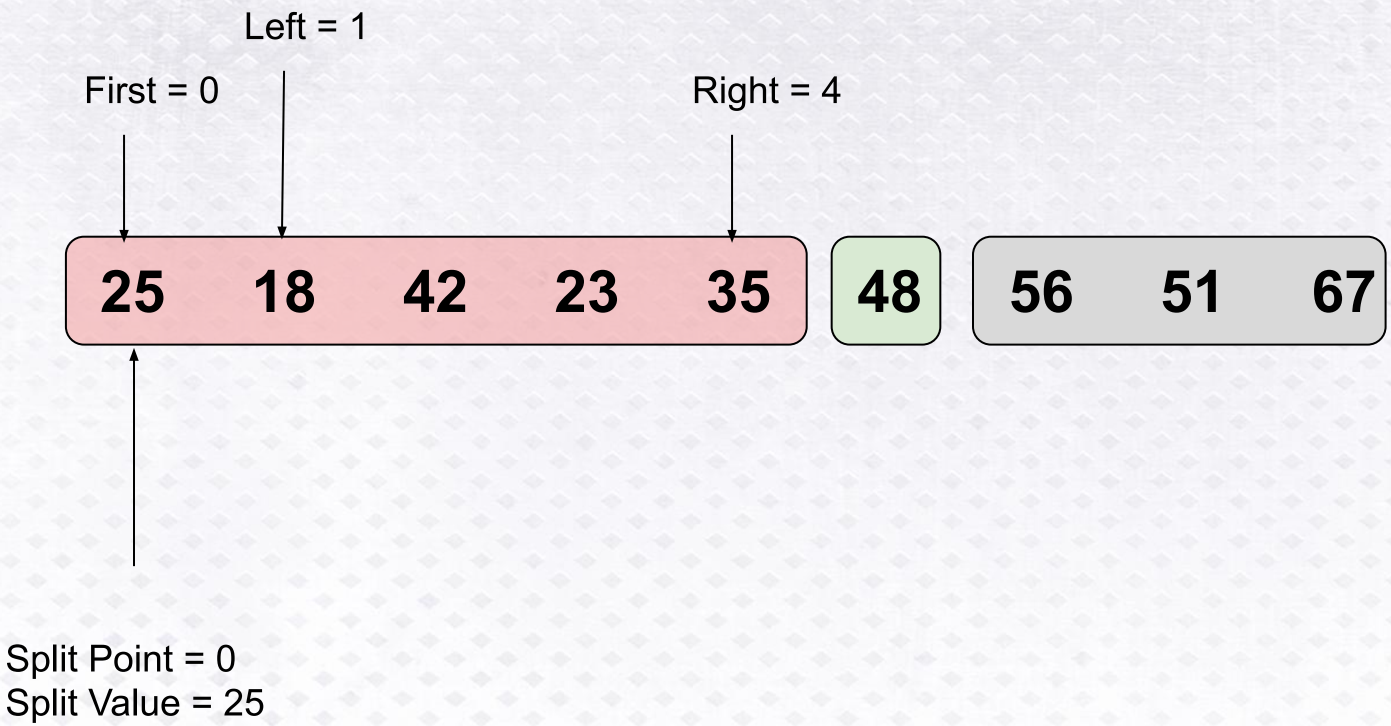
Quick Sort Worked Example



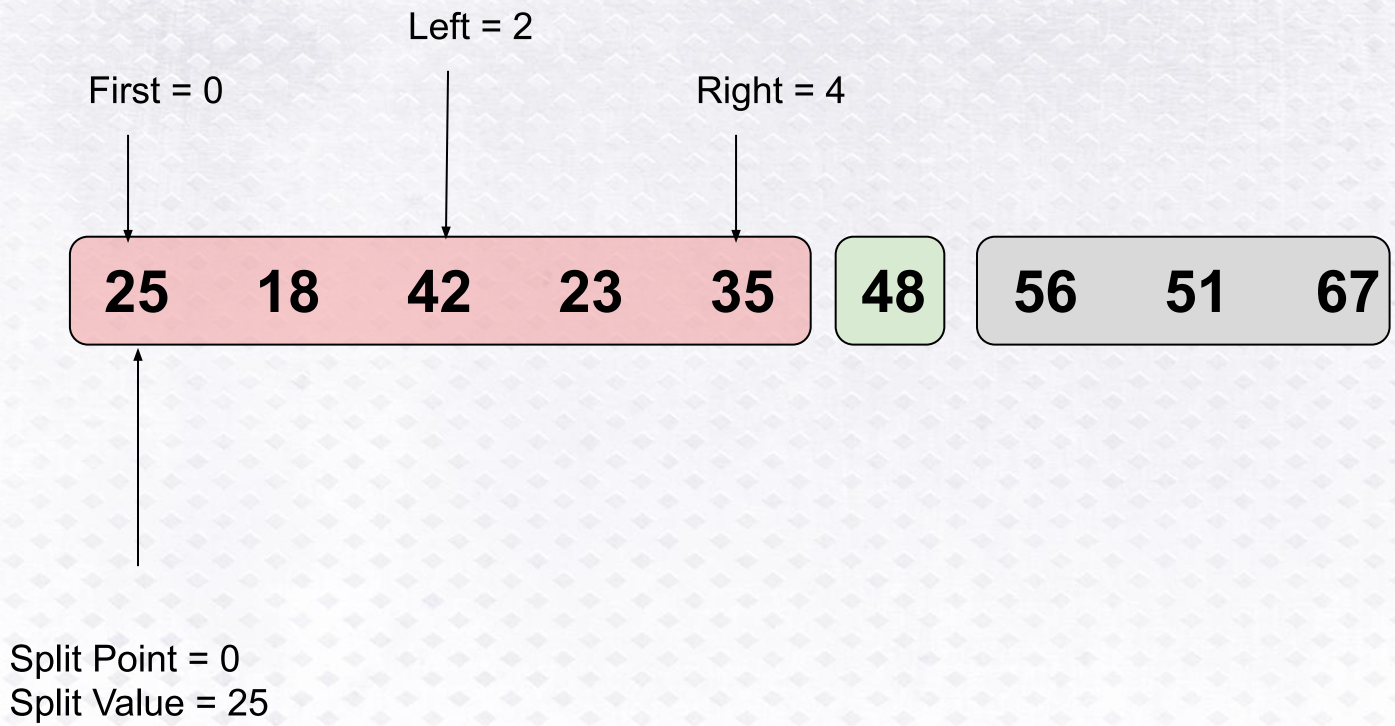
Quick Sort Worked Example



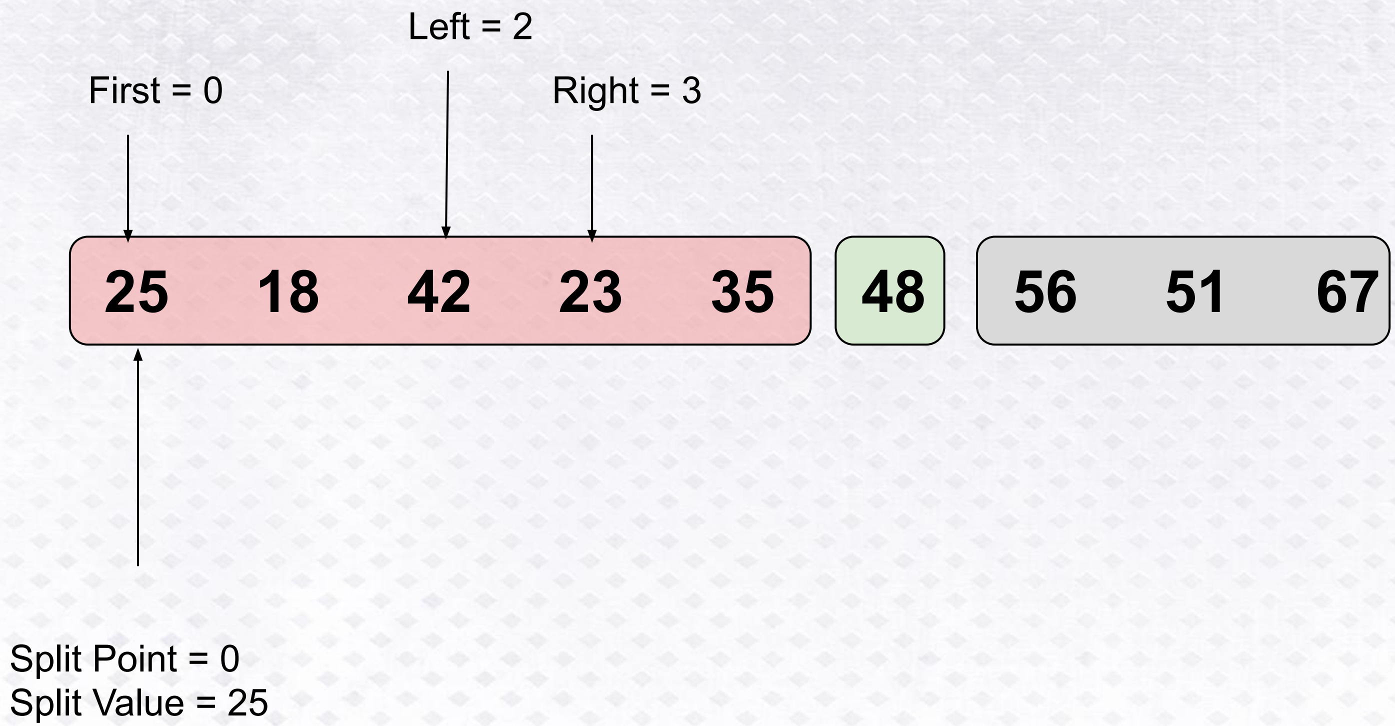
Quick Sort Worked Example



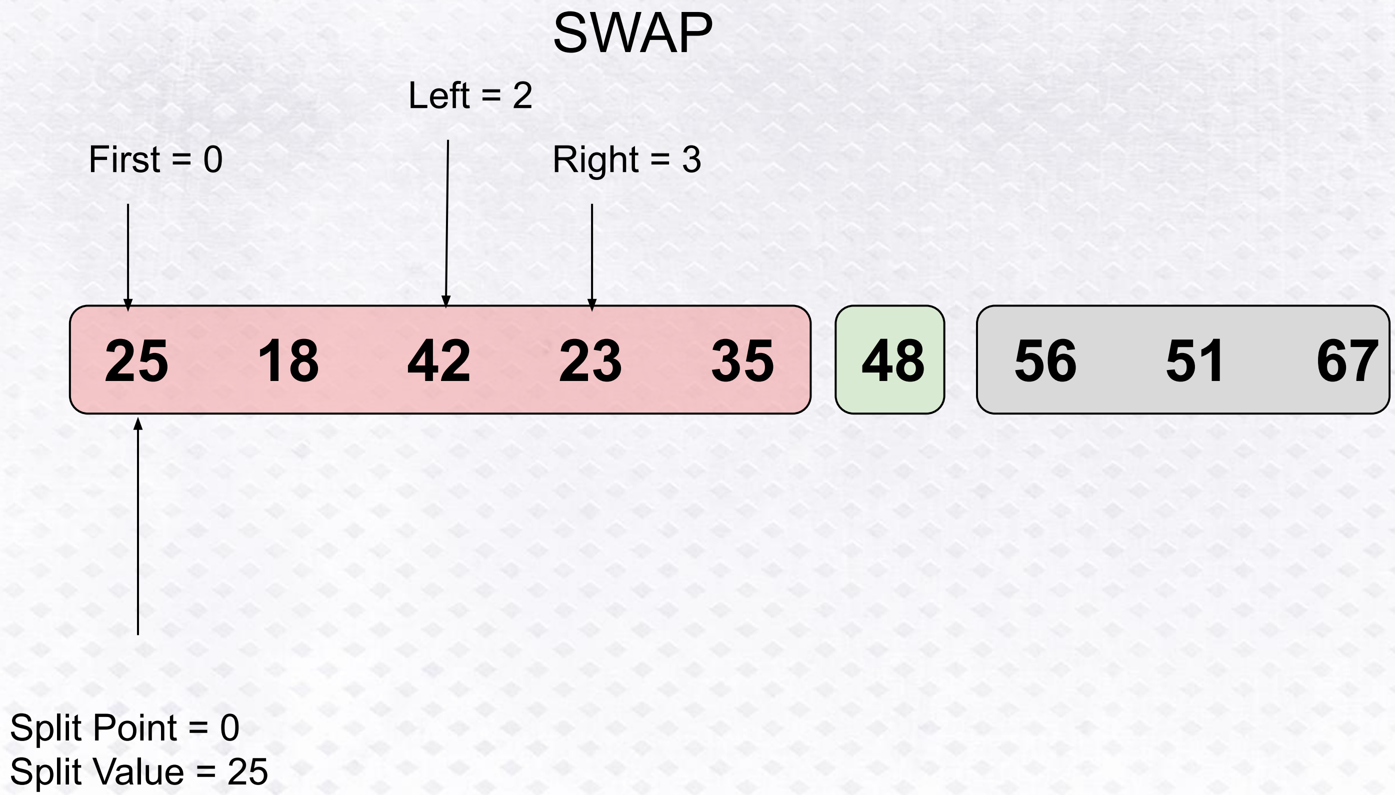
Quick Sort Worked Example



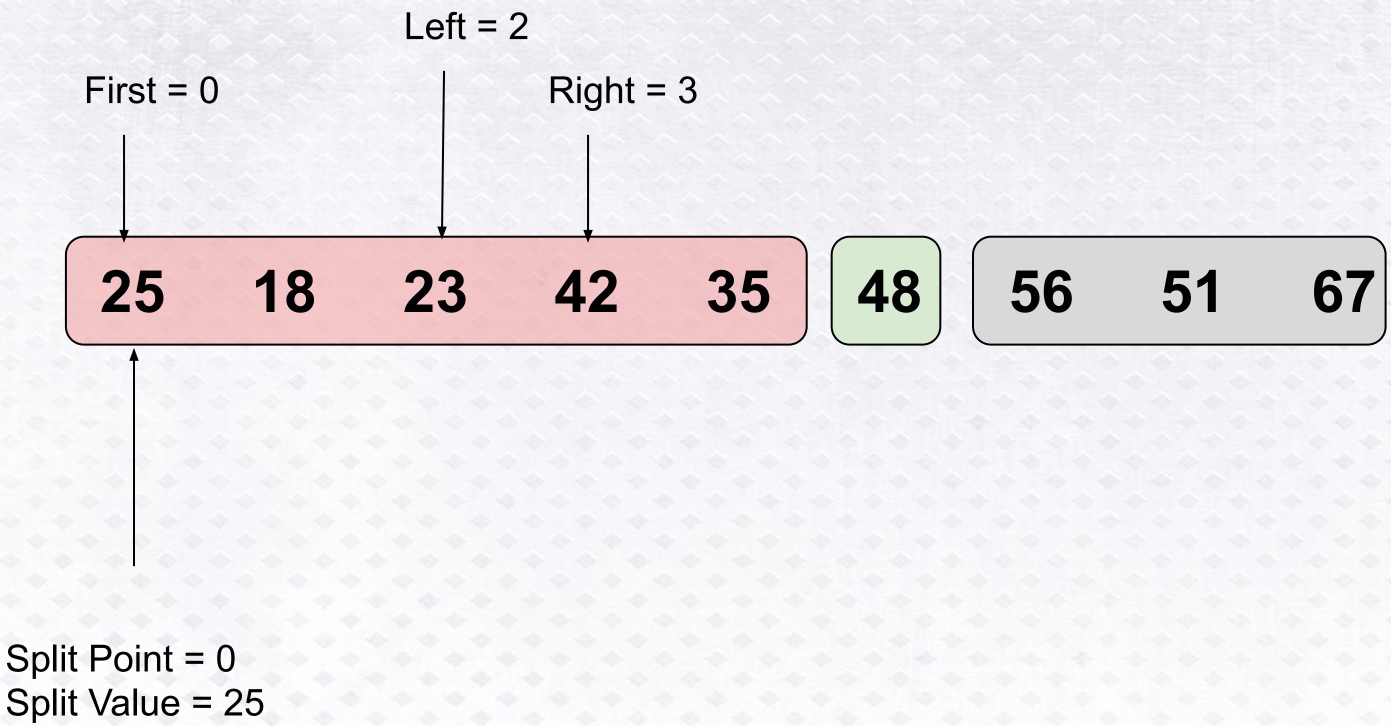
Quick Sort Worked Example



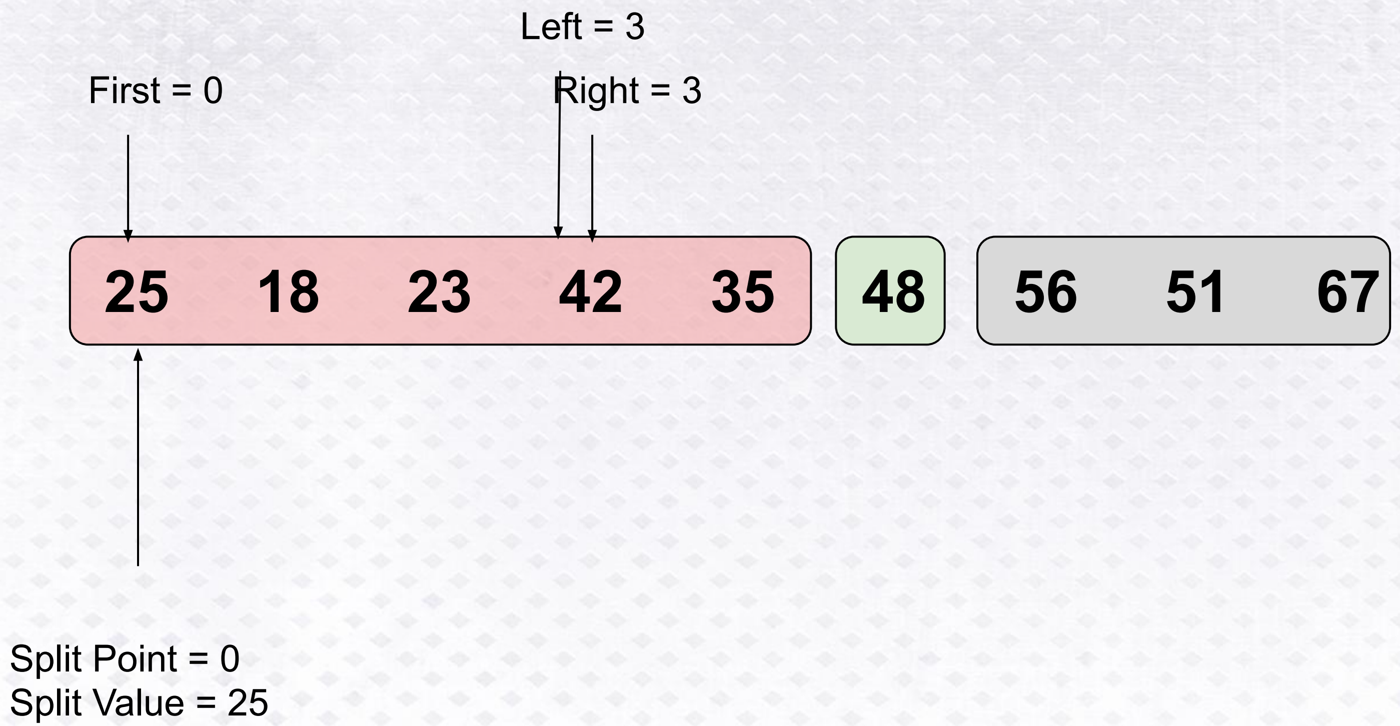
Quick Sort Worked Example



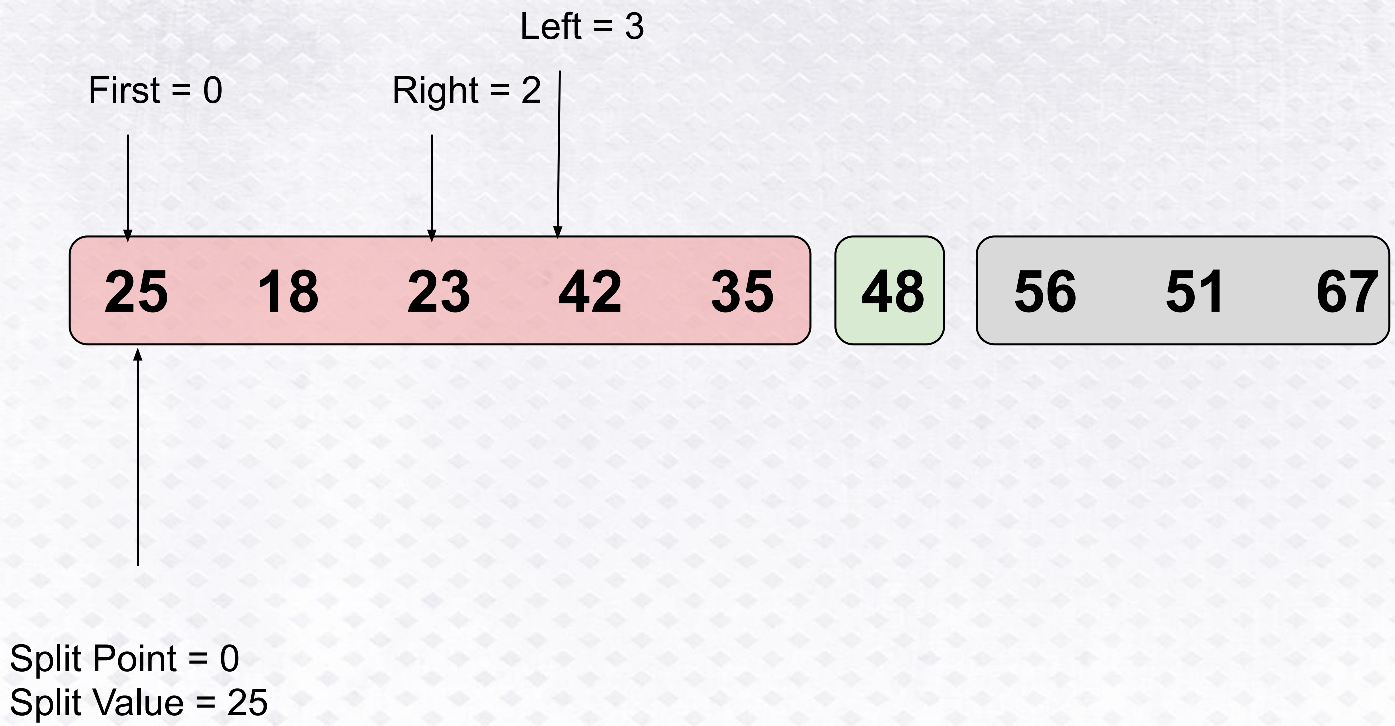
Quick Sort Worked Example



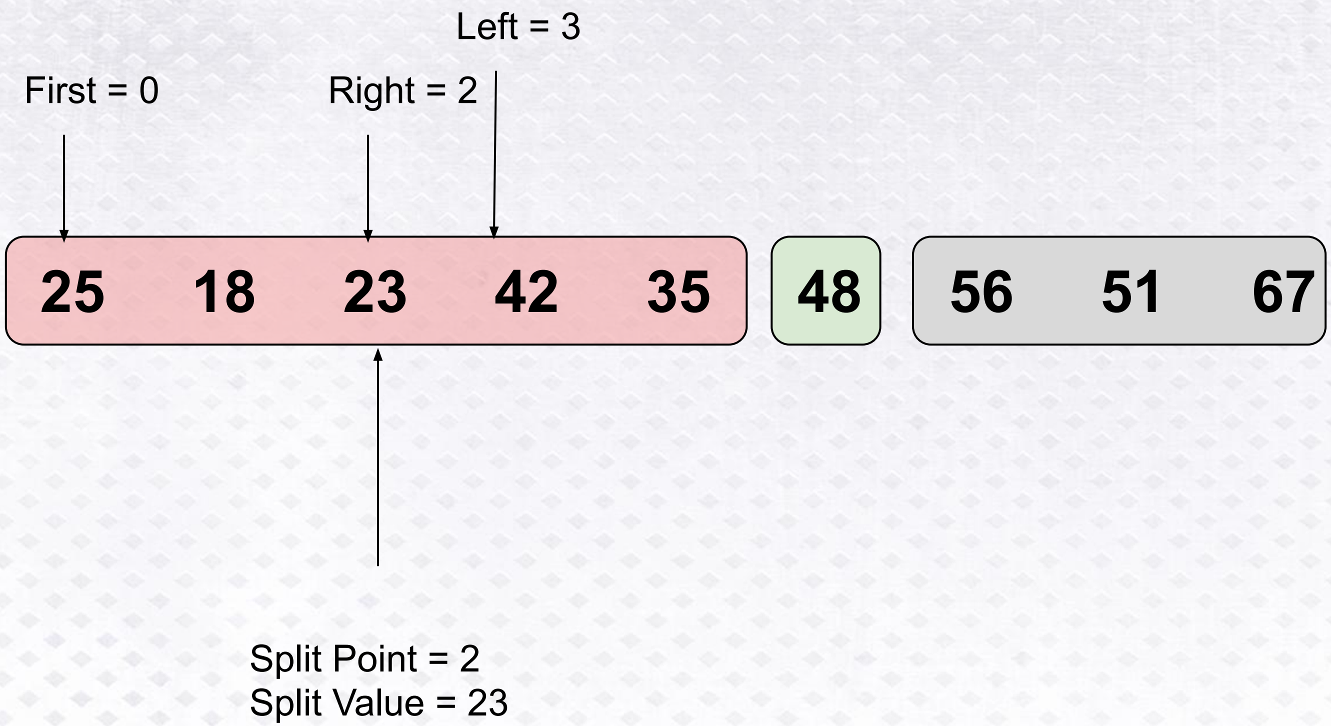
Quick Sort Worked Example



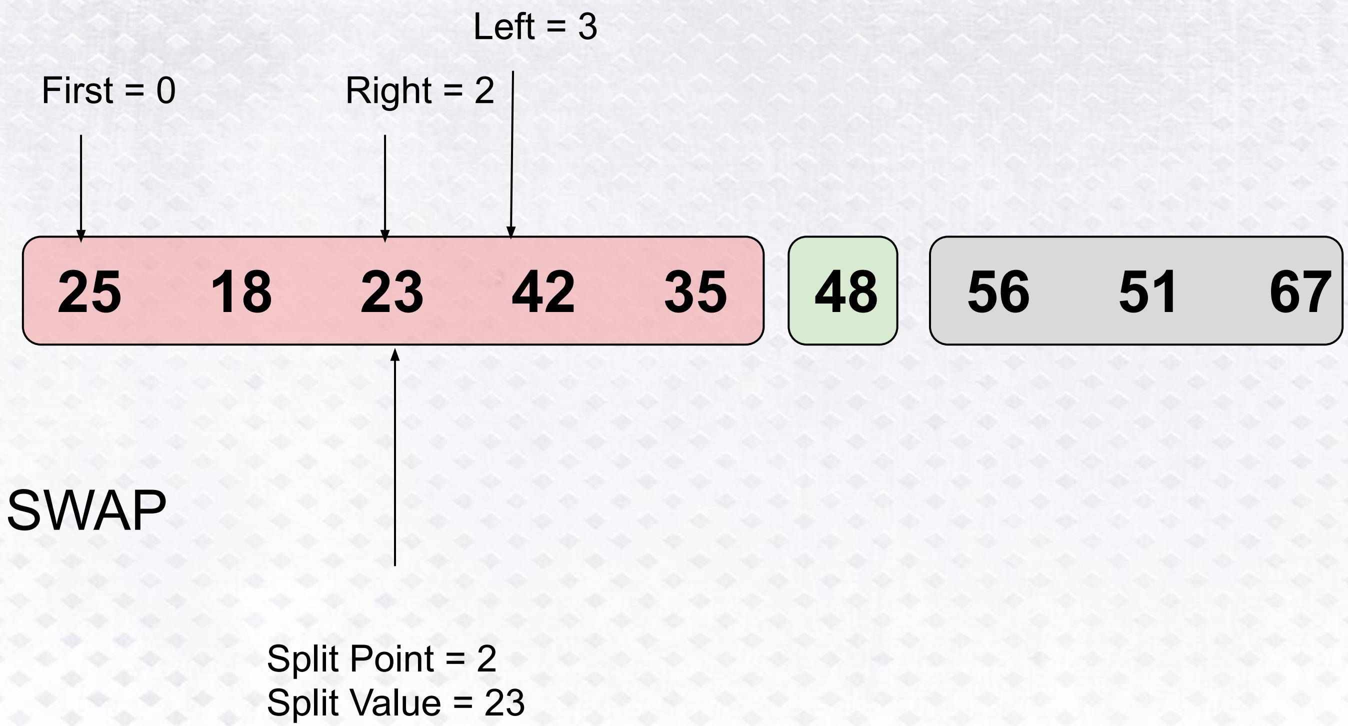
Quick Sort Worked Example



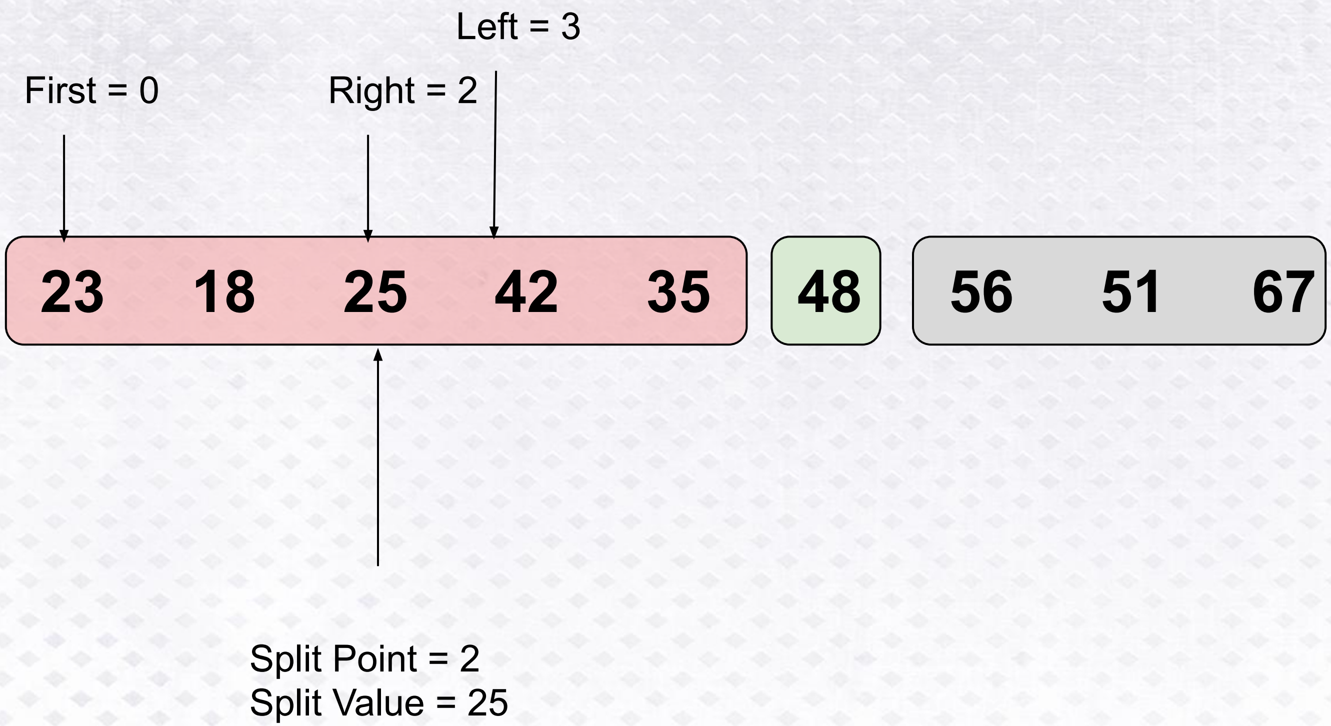
Quick Sort Worked Example



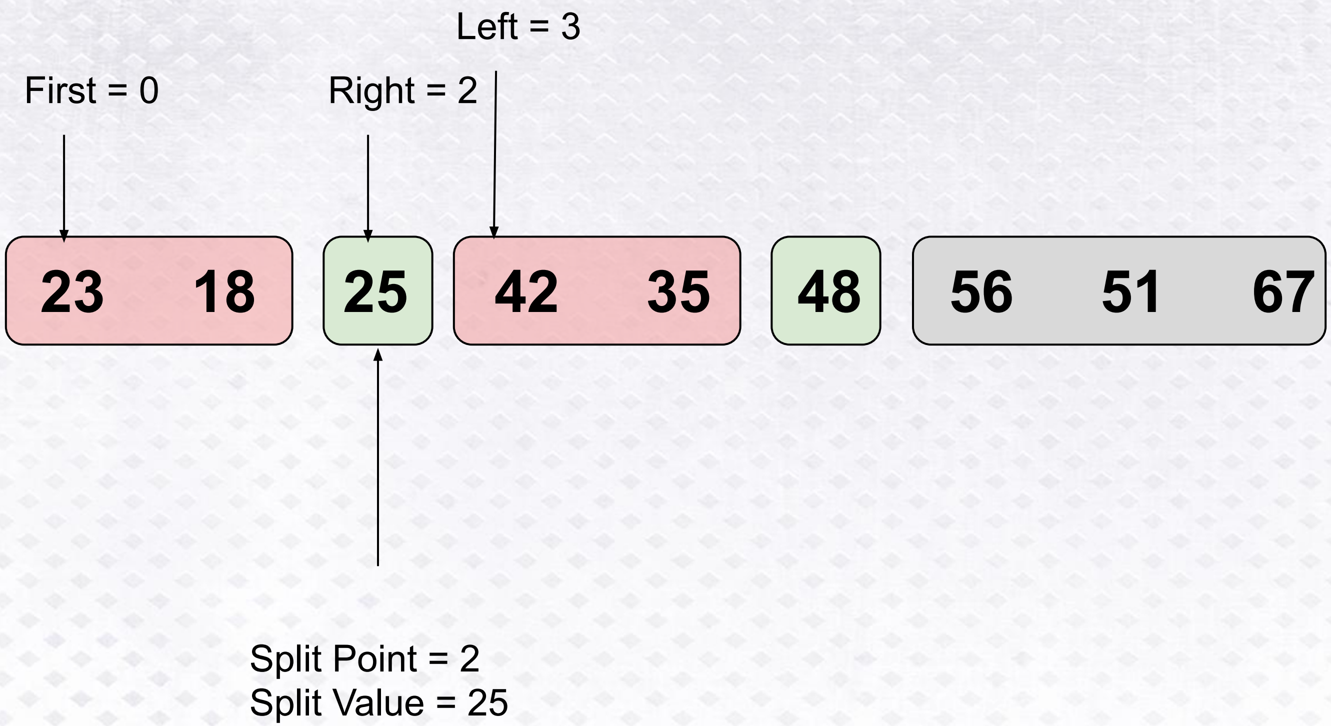
Quick Sort Worked Example



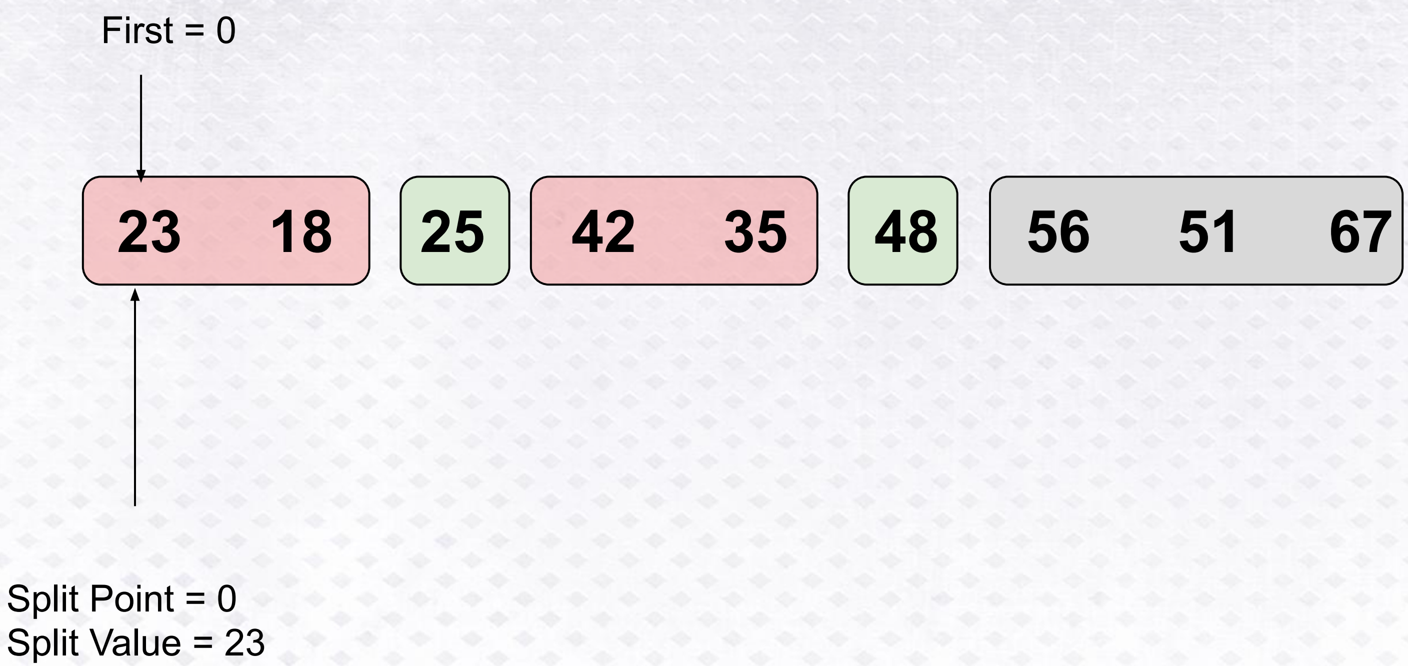
Quick Sort Worked Example



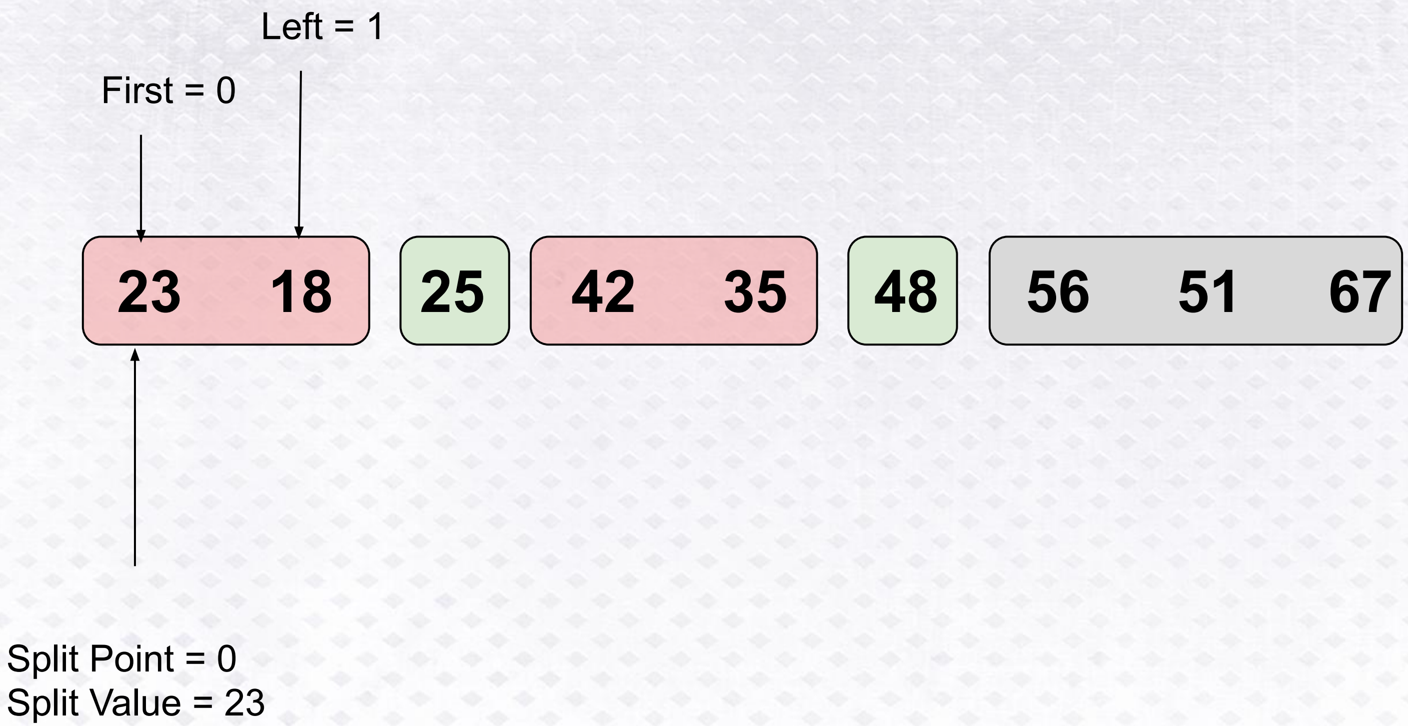
Quick Sort Worked Example



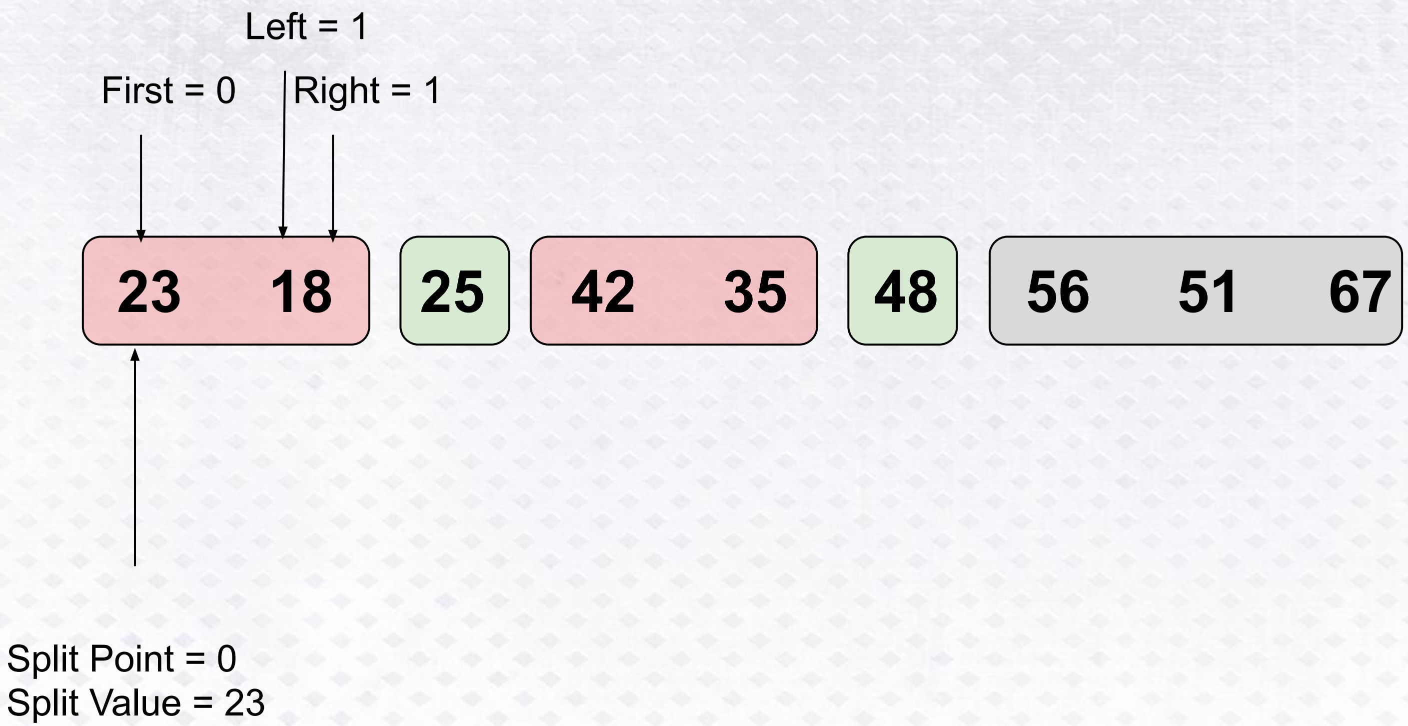
Quick Sort Worked Example



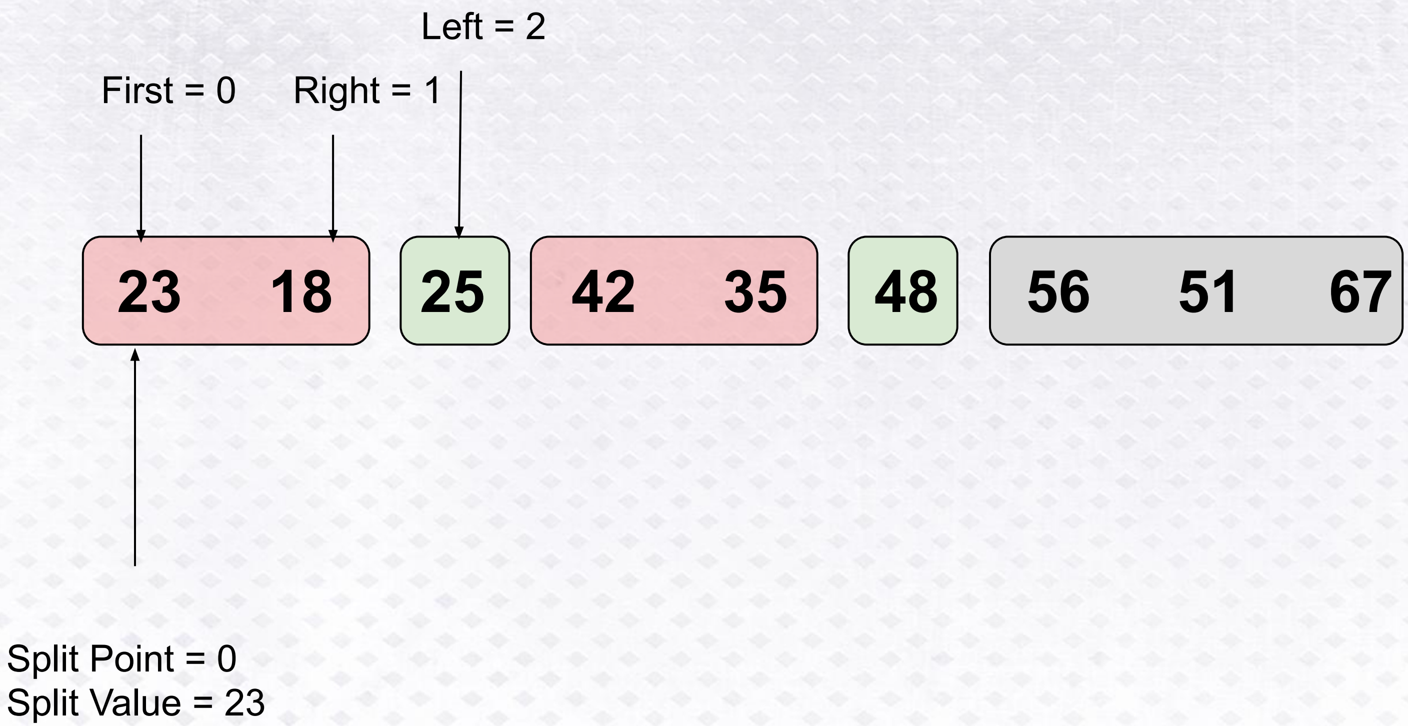
Quick Sort Worked Example



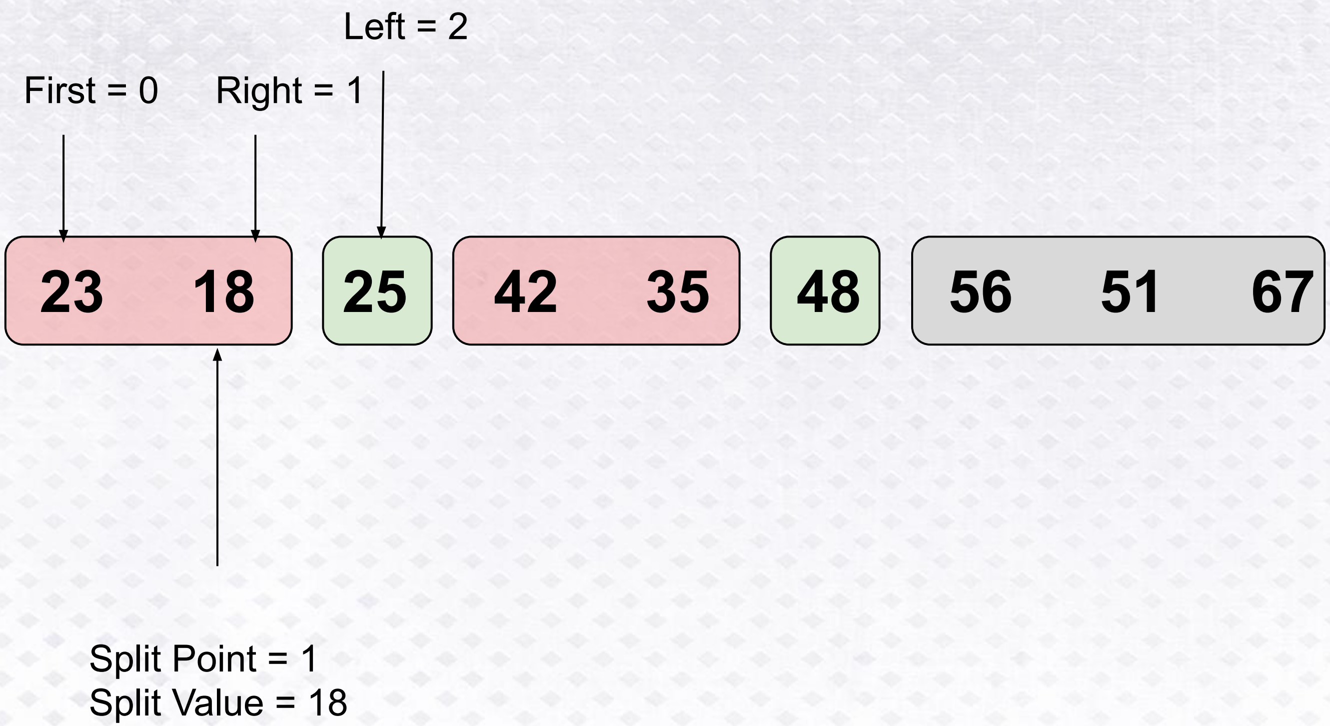
Quick Sort Worked Example



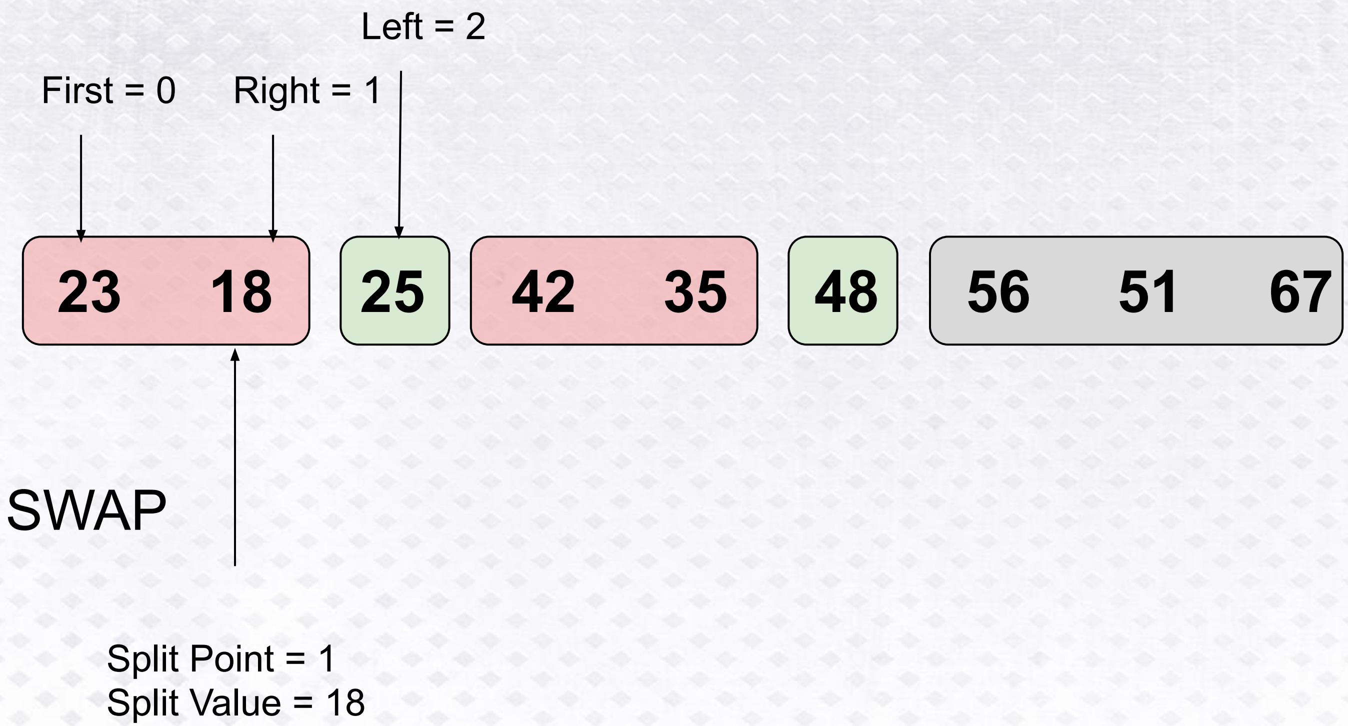
Quick Sort Worked Example



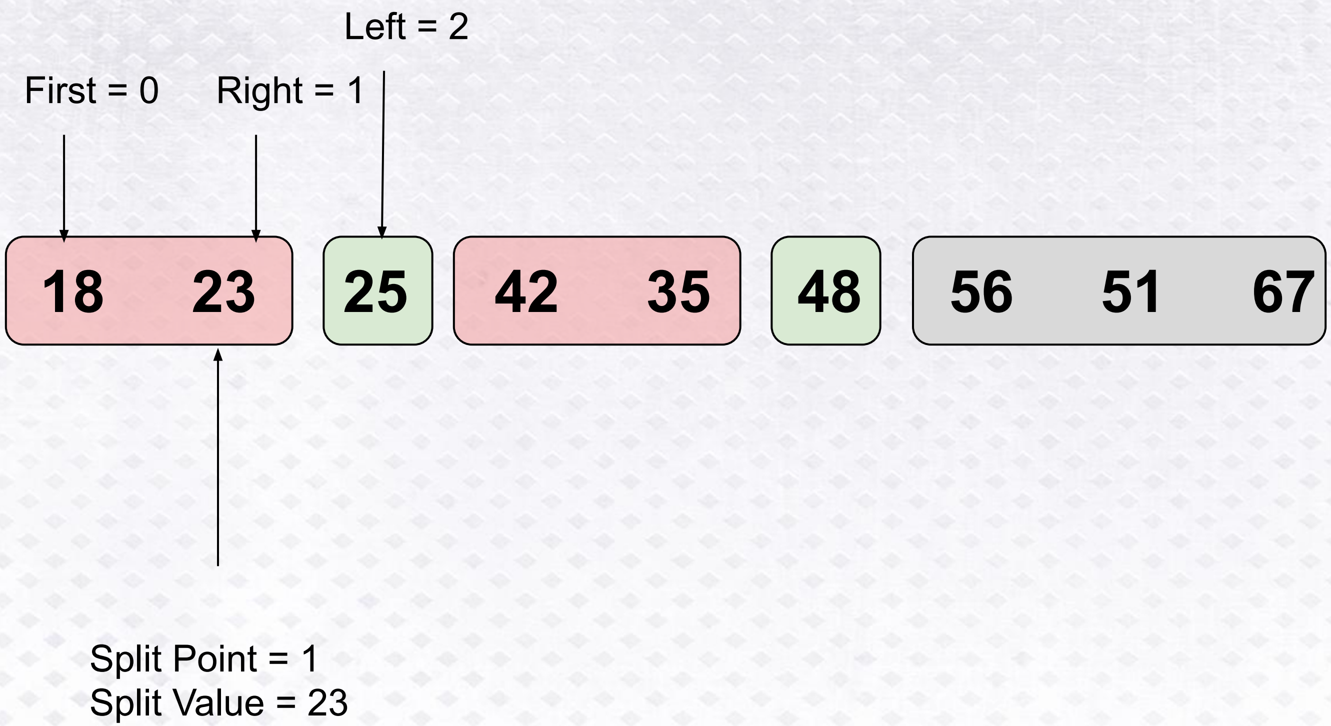
Quick Sort Worked Example



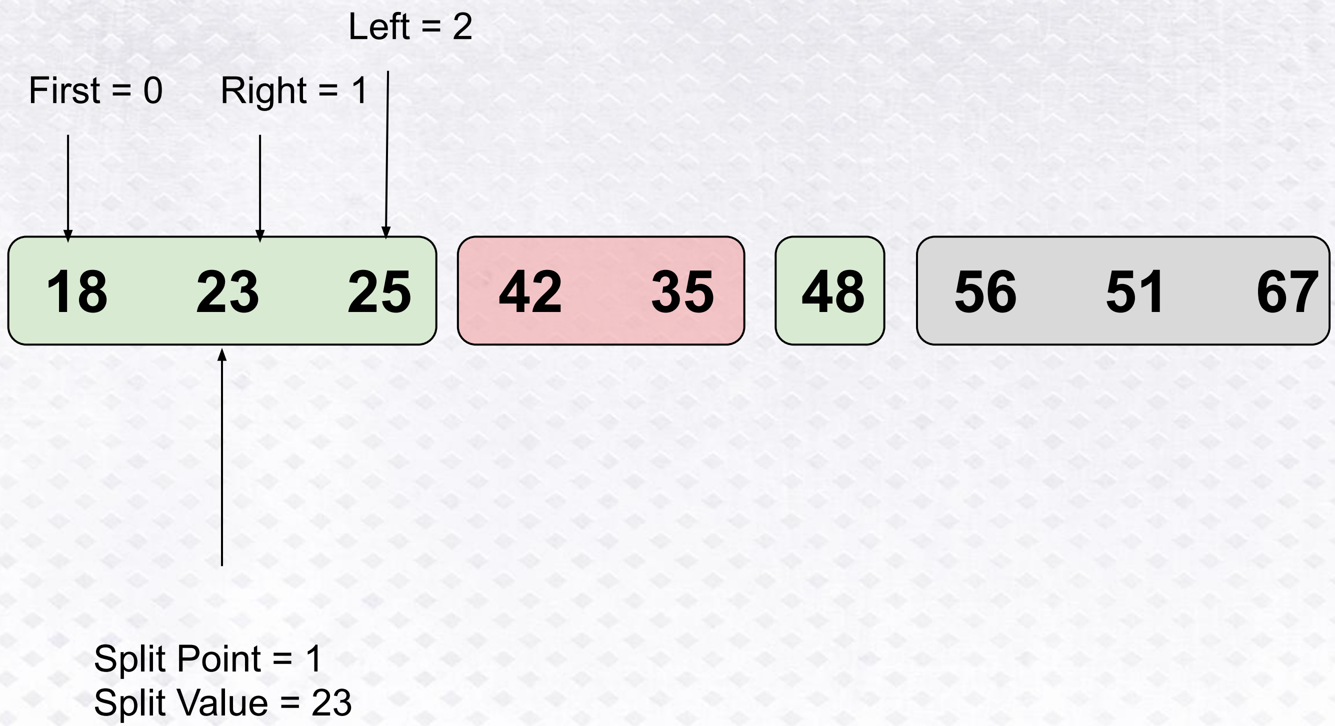
Quick Sort Worked Example



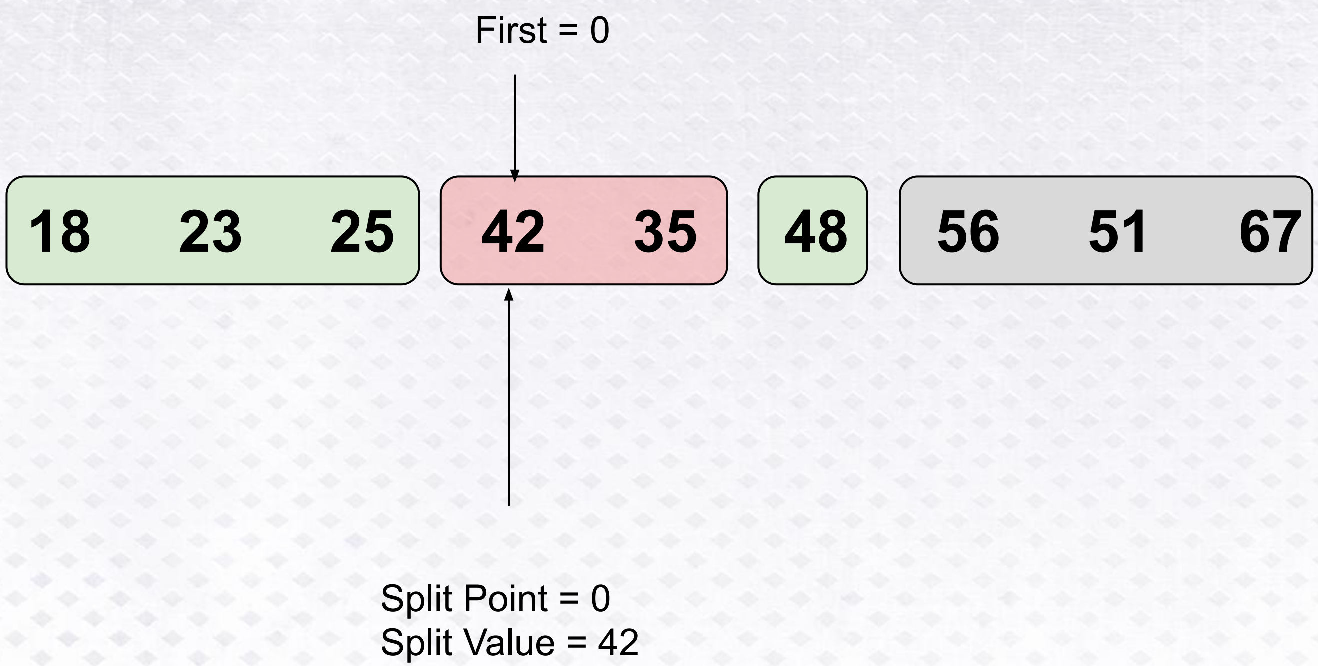
Quick Sort Worked Example



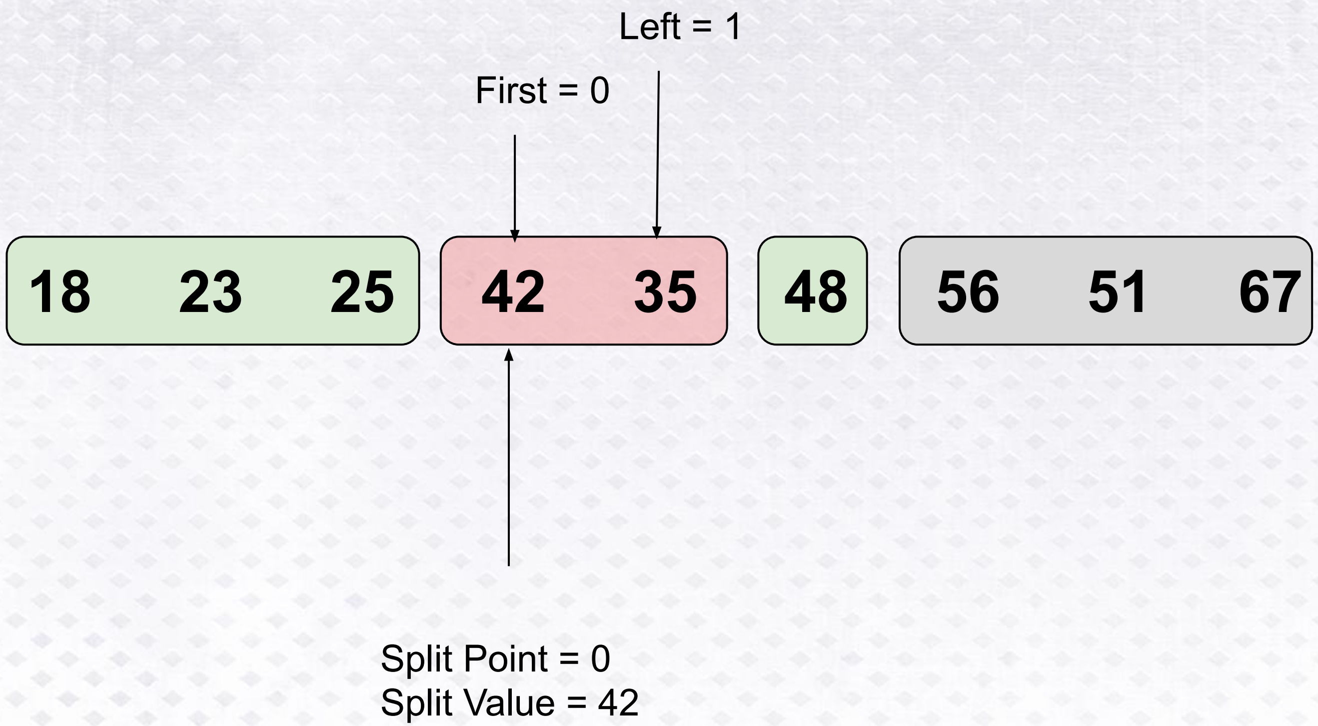
Quick Sort Worked Example



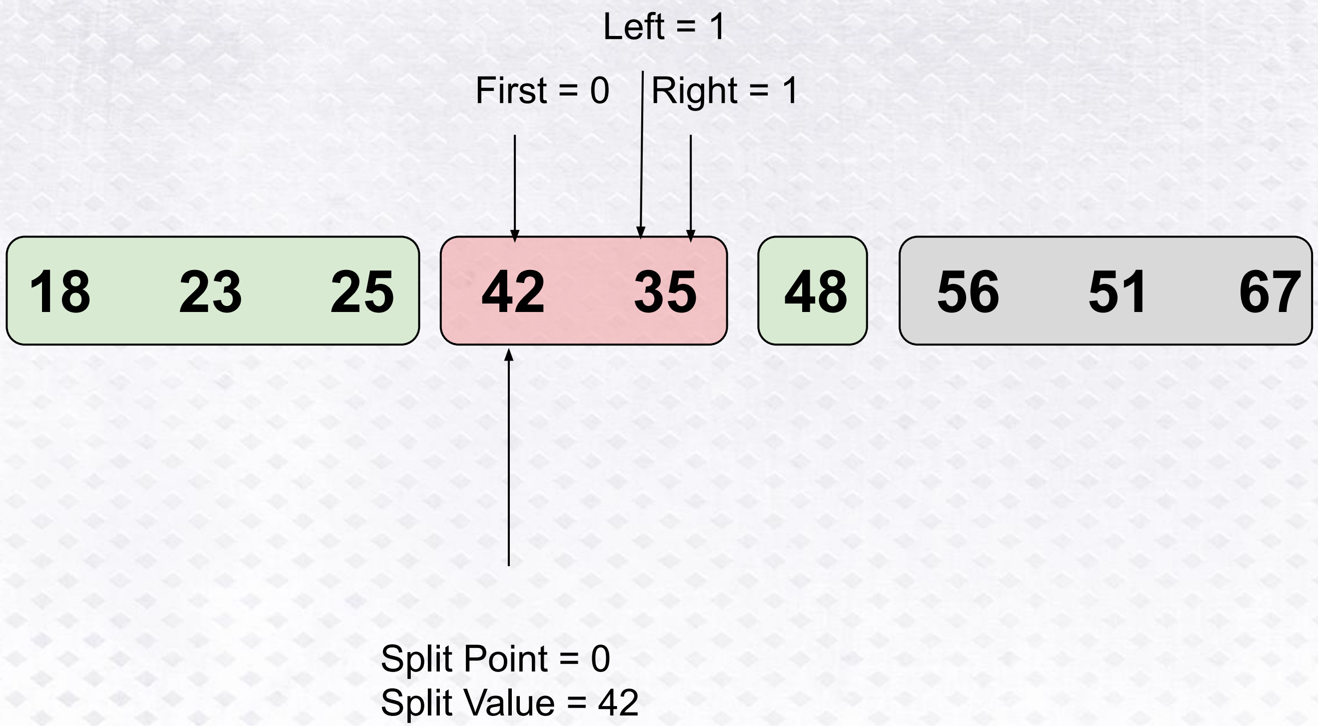
Quick Sort Worked Example



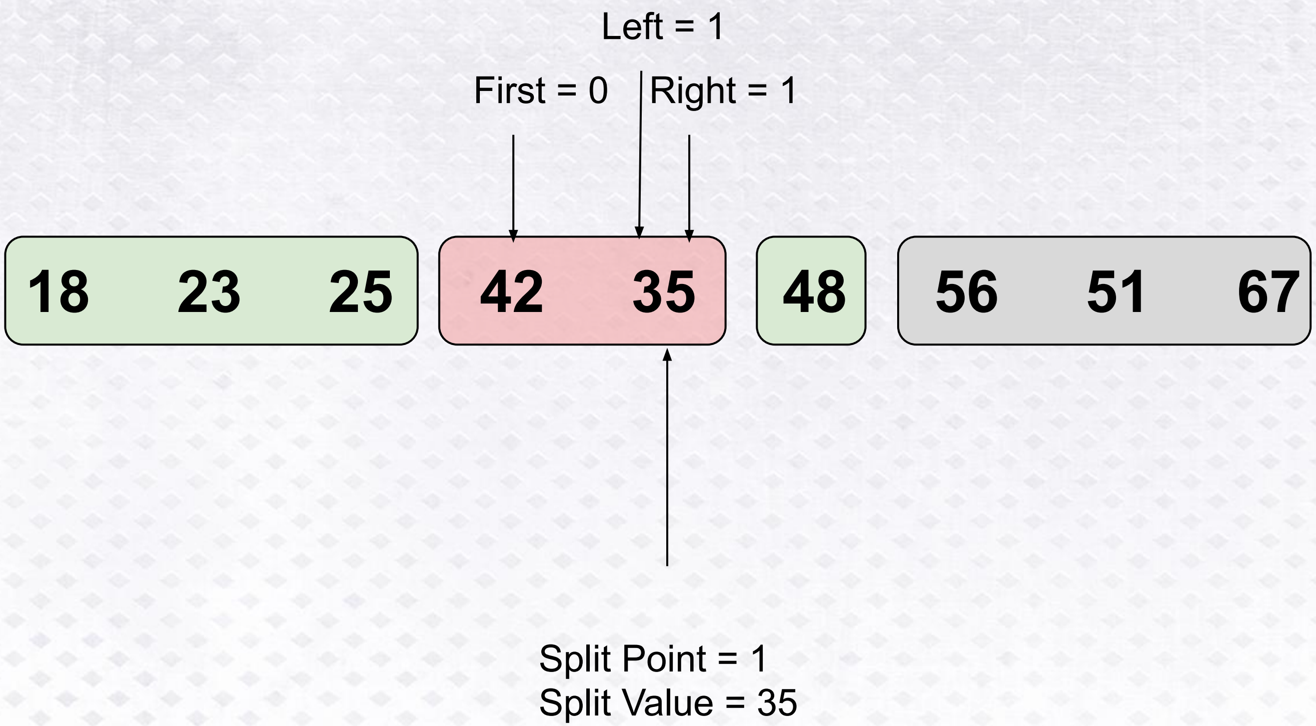
Quick Sort Worked Example



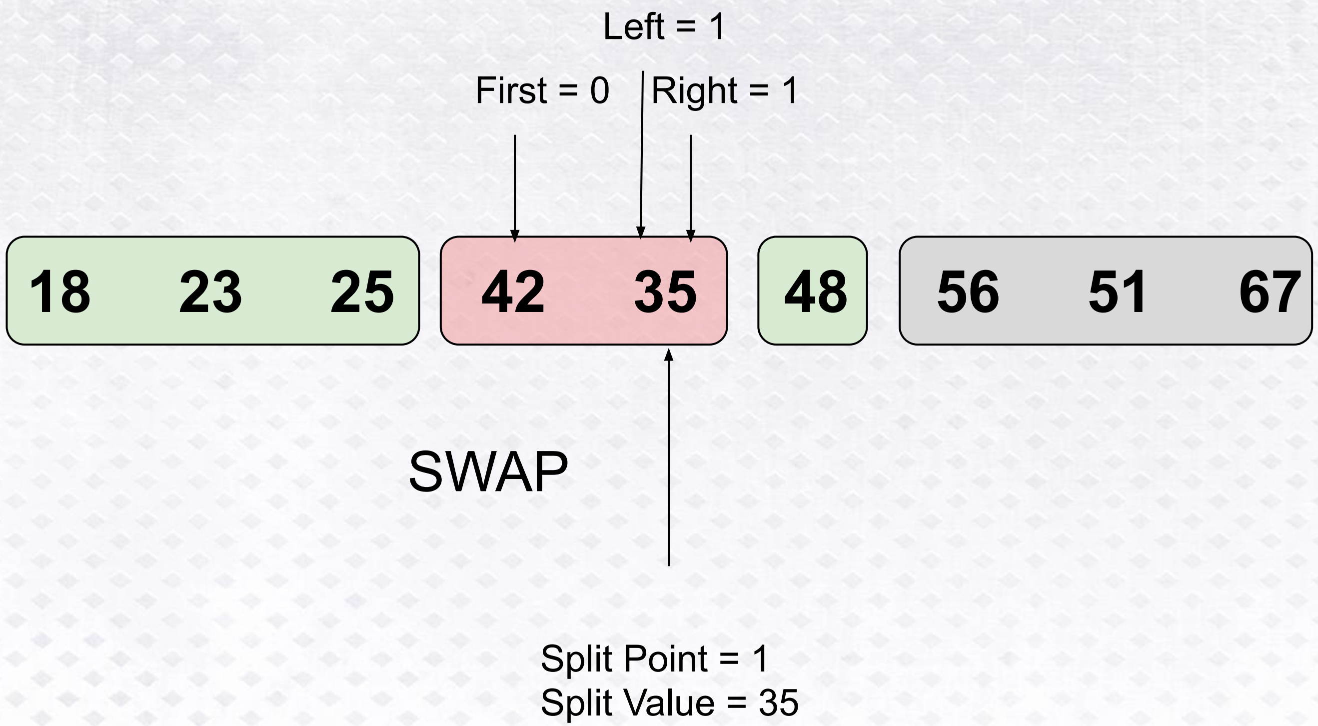
Quick Sort Worked Example



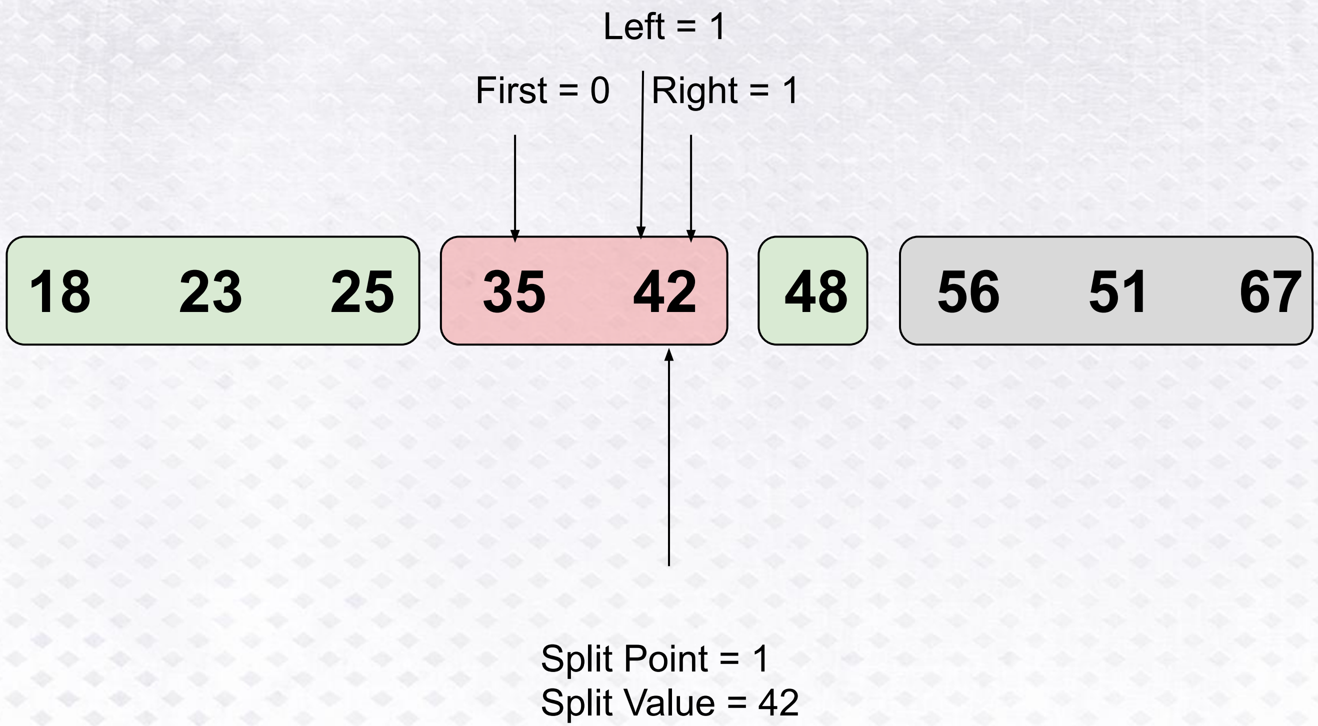
Quick Sort Worked Example



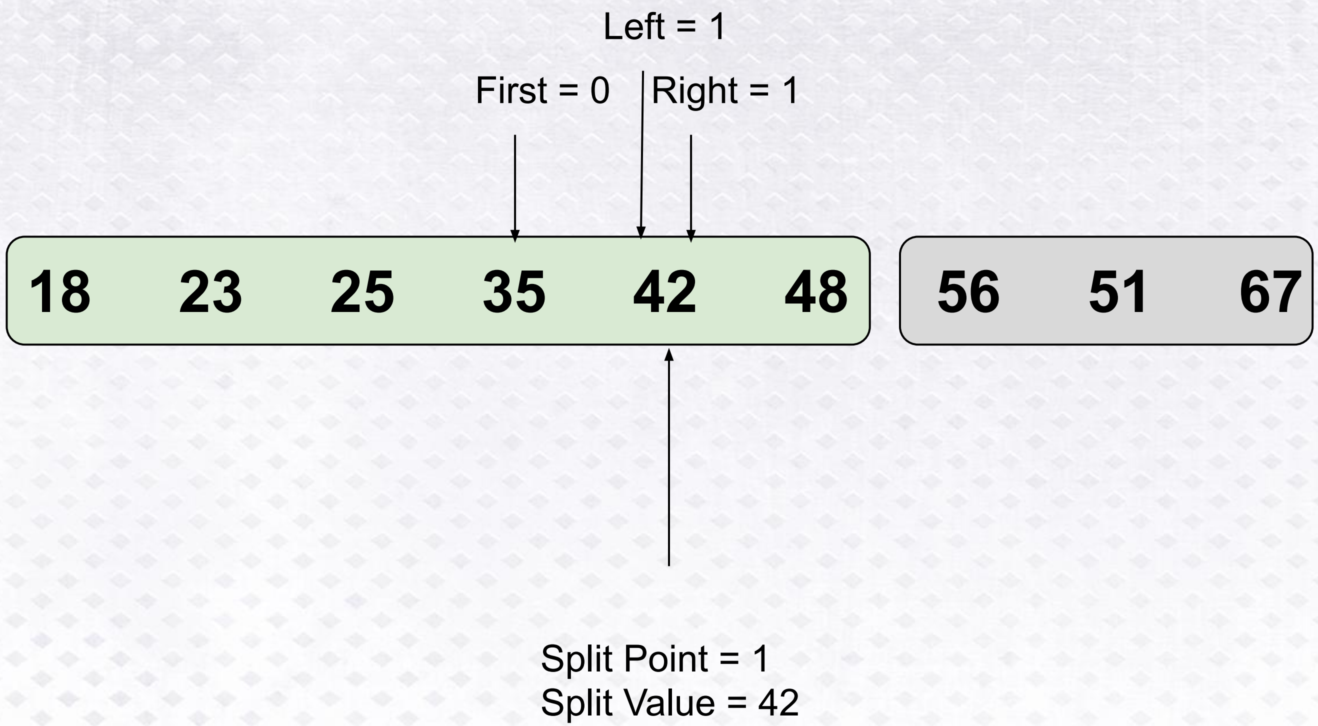
Quick Sort Worked Example



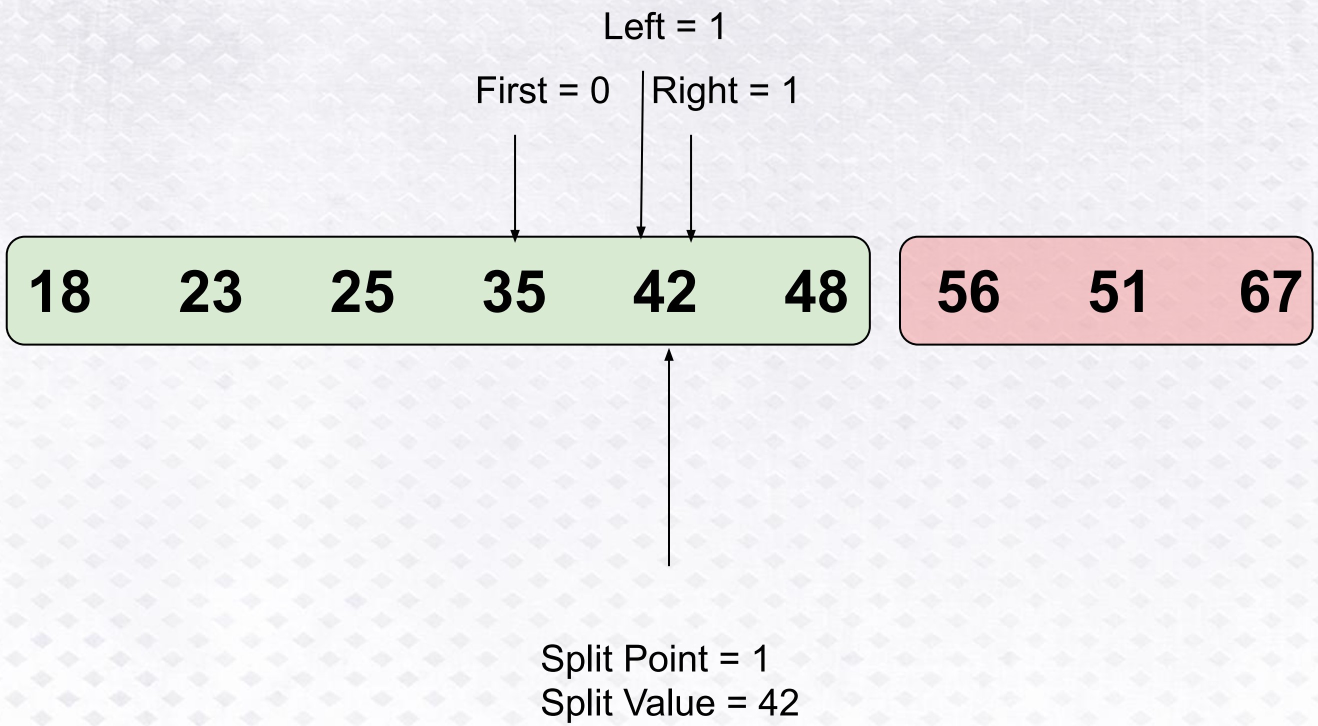
Quick Sort Worked Example



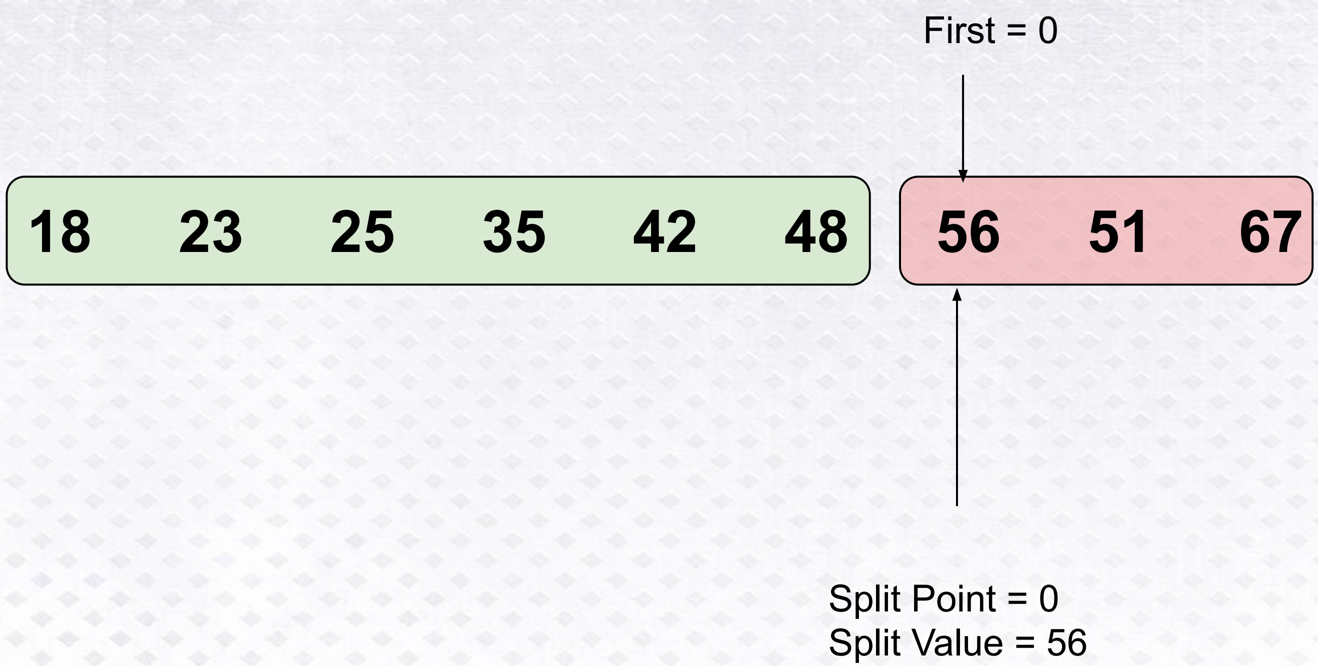
Quick Sort Worked Example



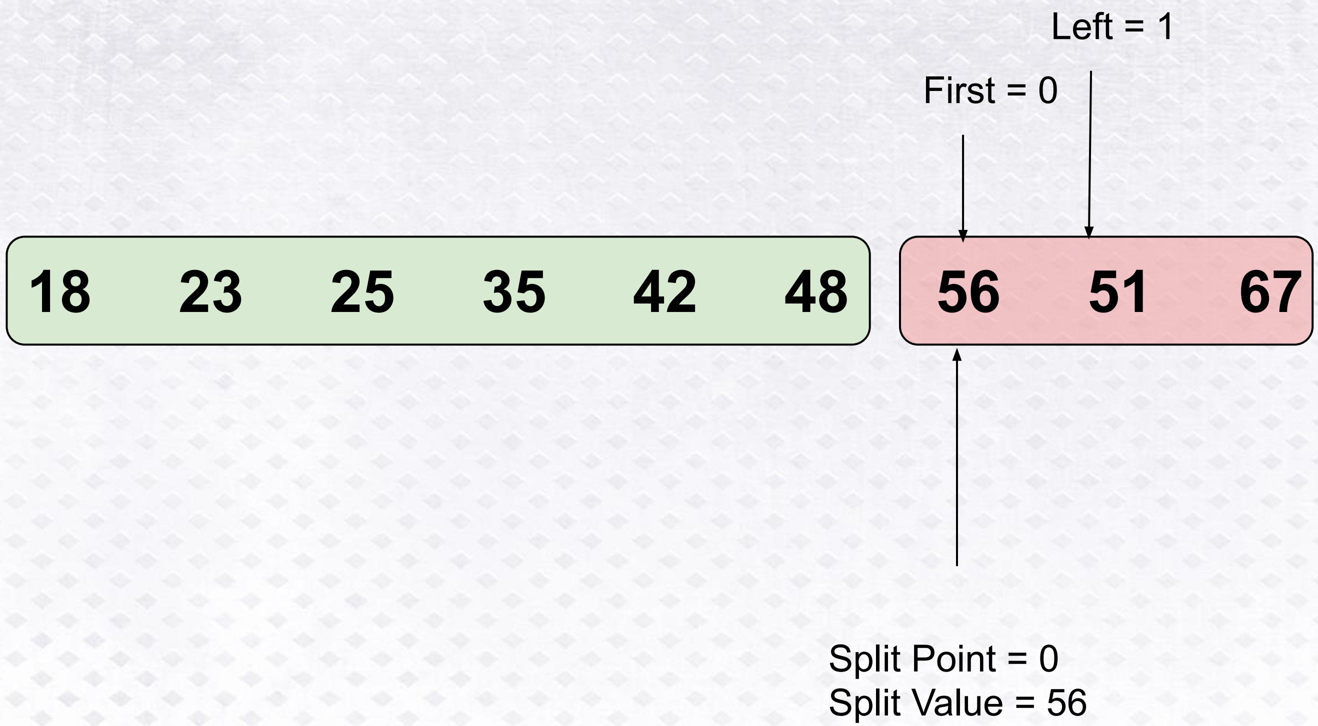
Quick Sort Worked Example



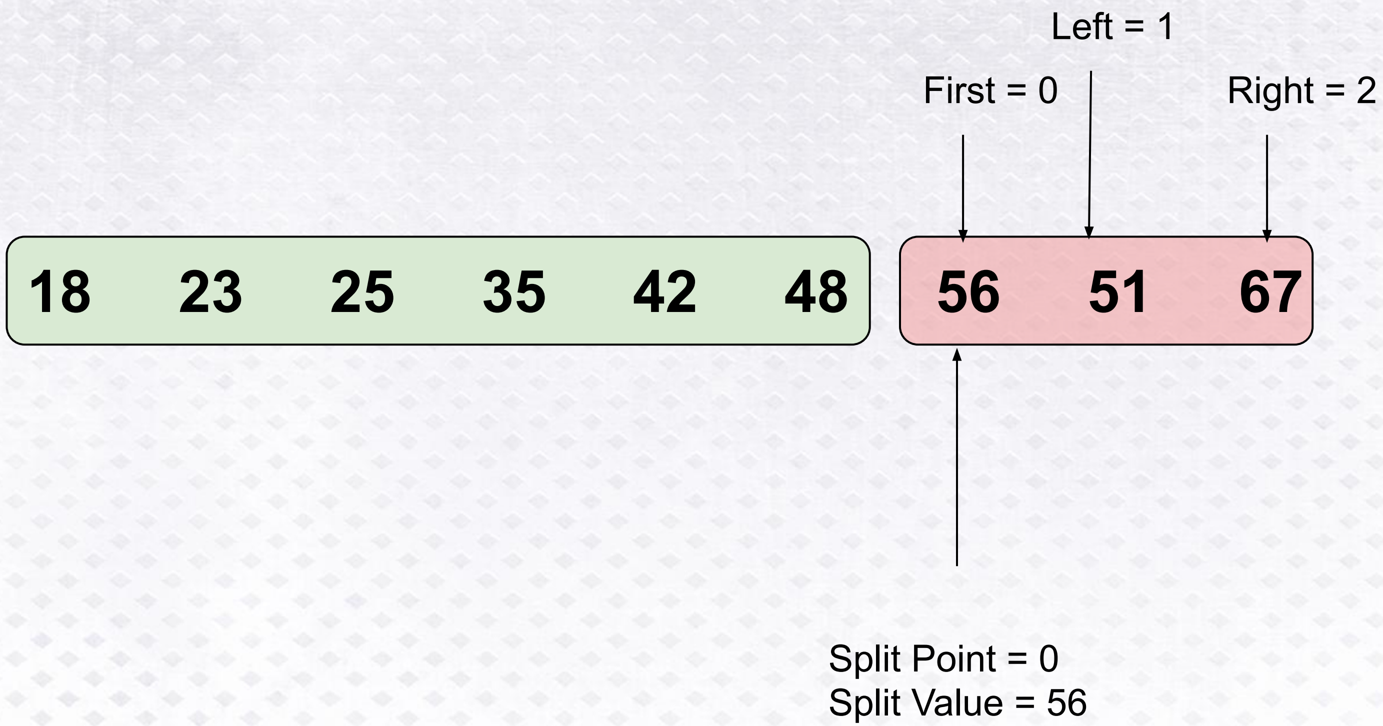
Quick Sort Worked Example



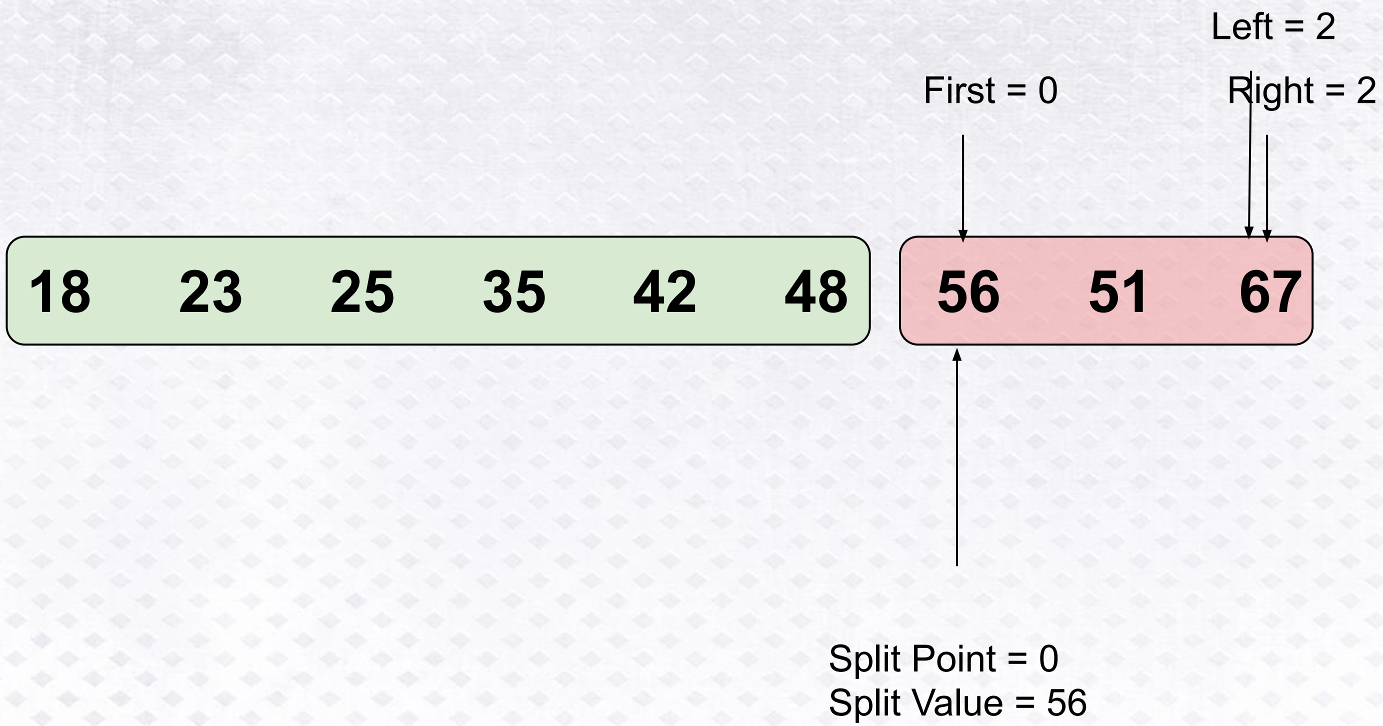
Quick Sort Worked Example



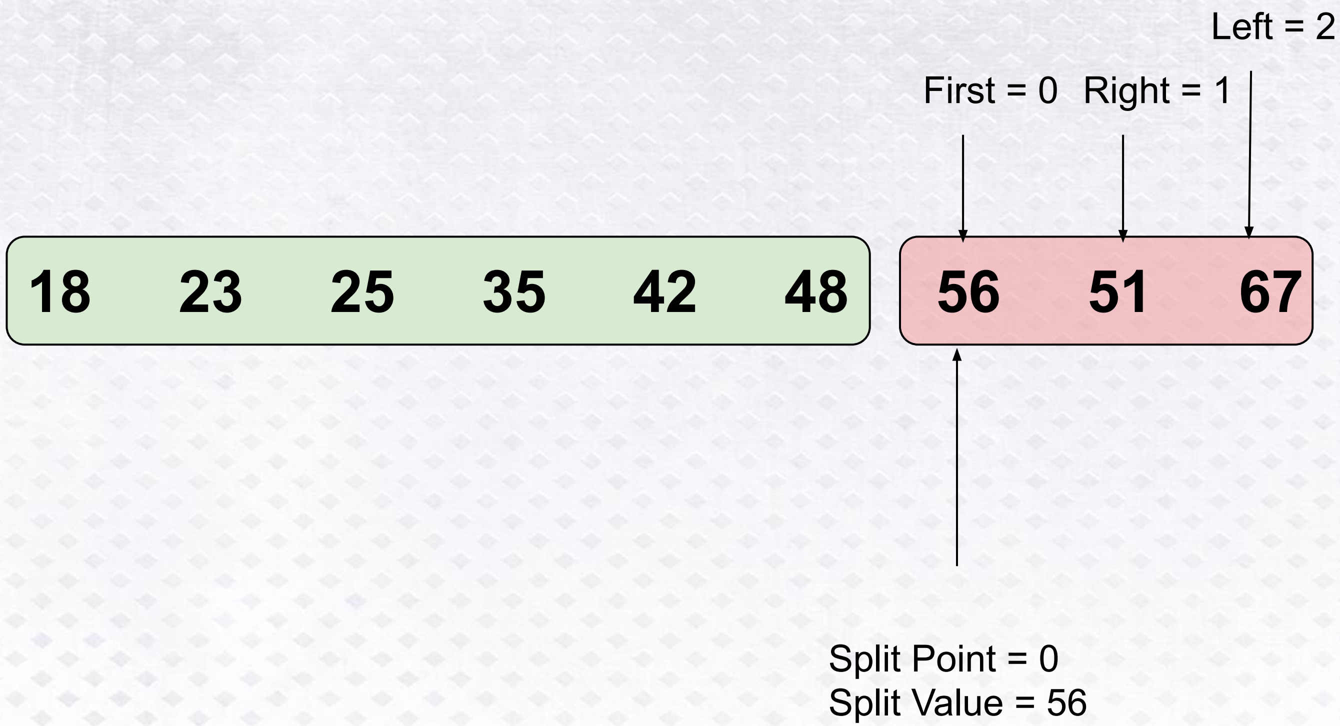
Quick Sort Worked Example



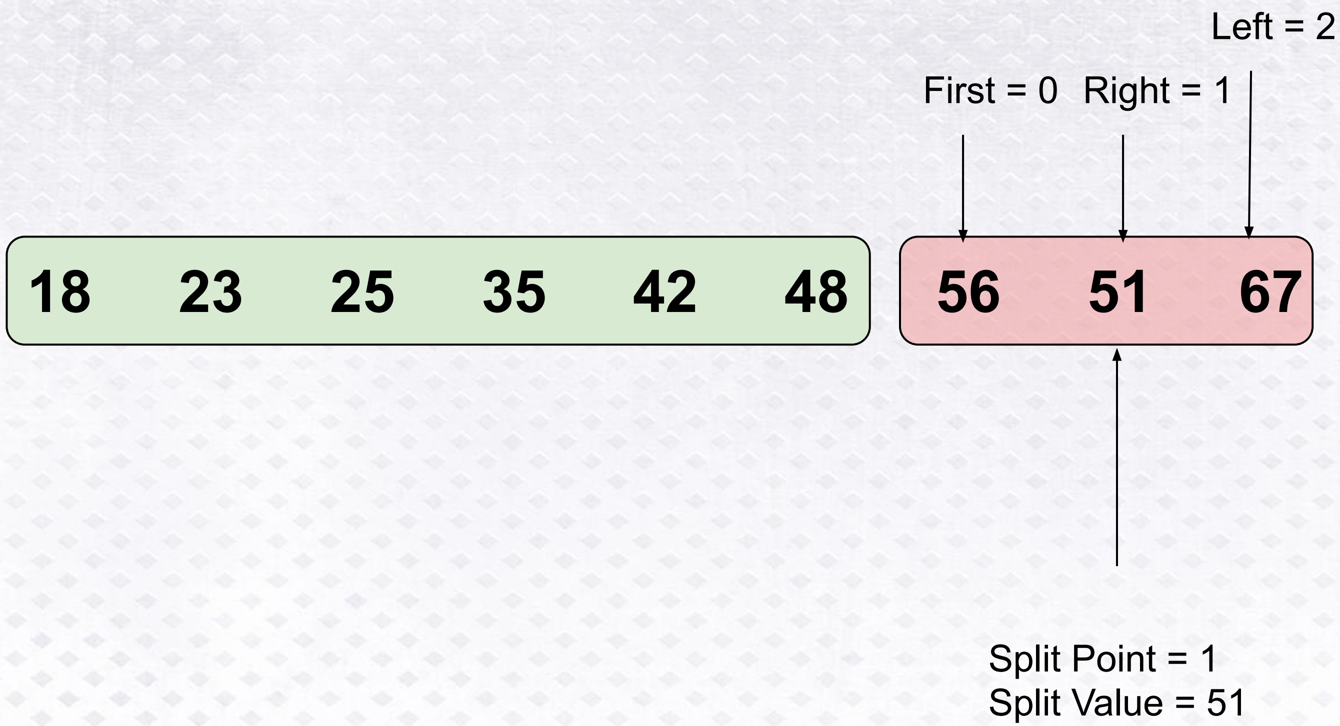
Quick Sort Worked Example



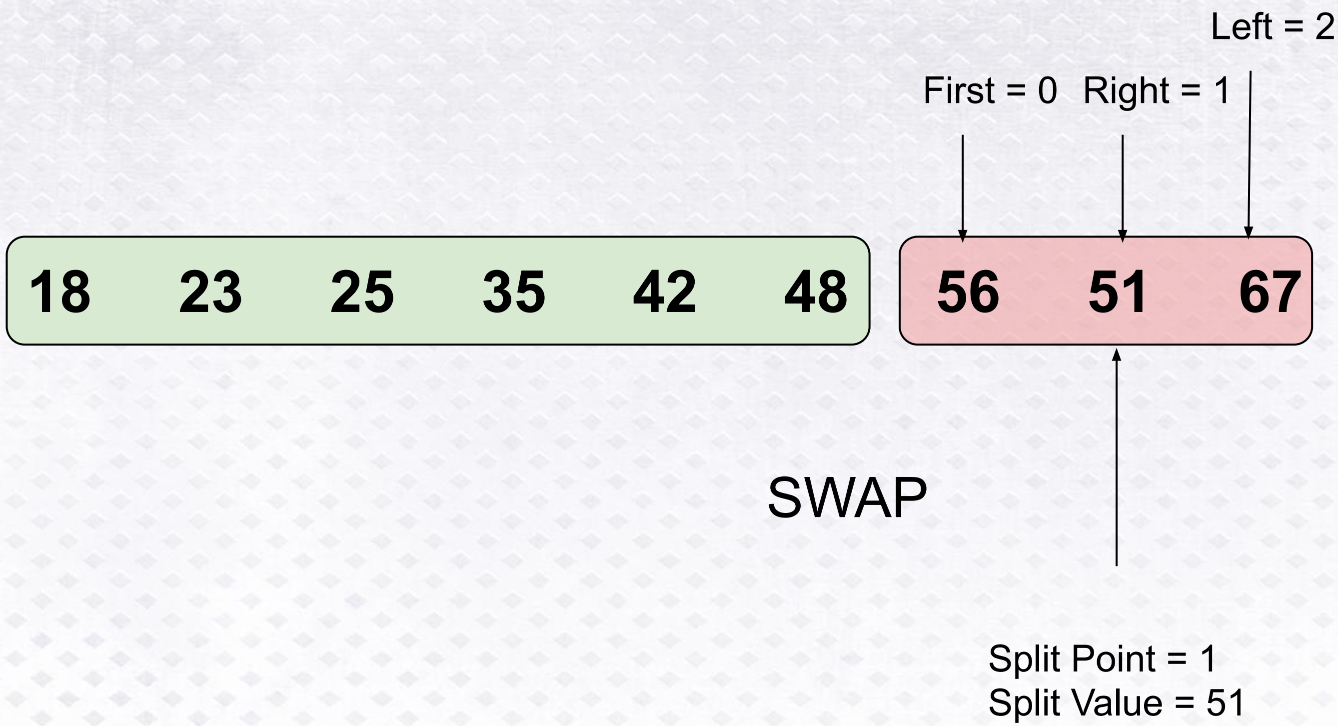
Quick Sort Worked Example



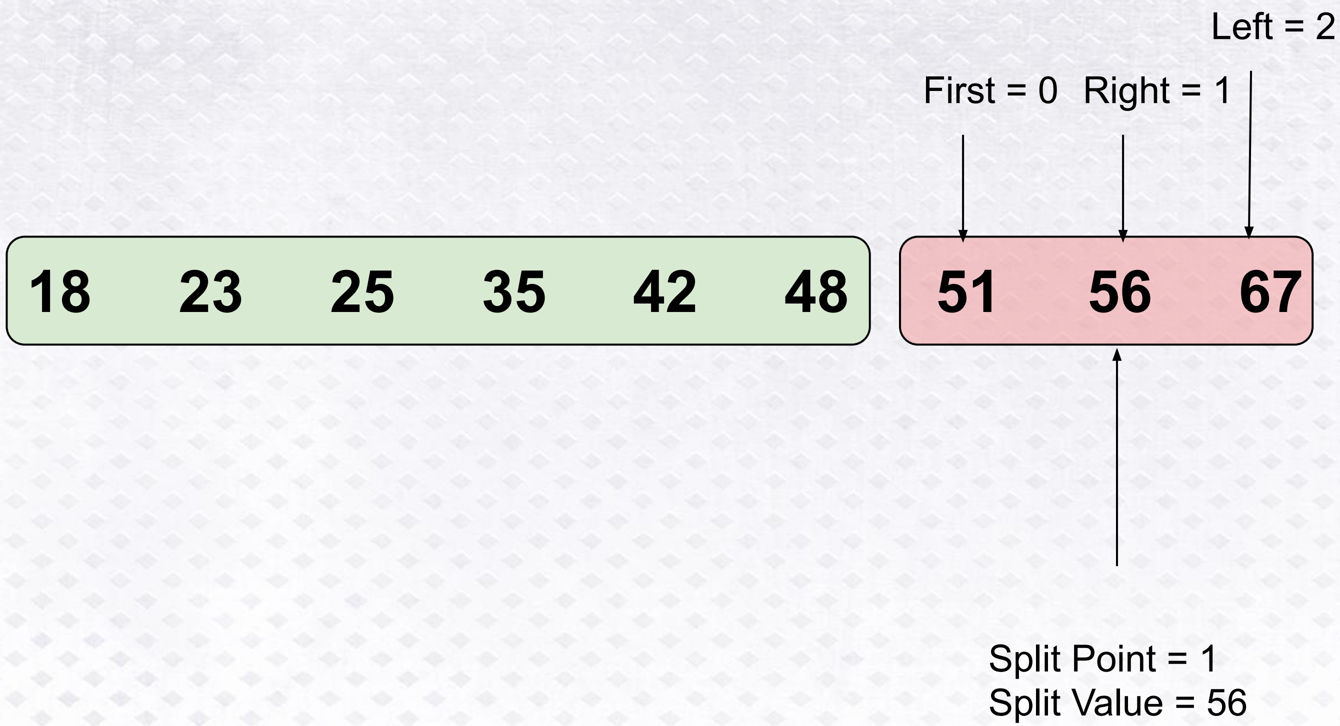
Quick Sort Worked Example



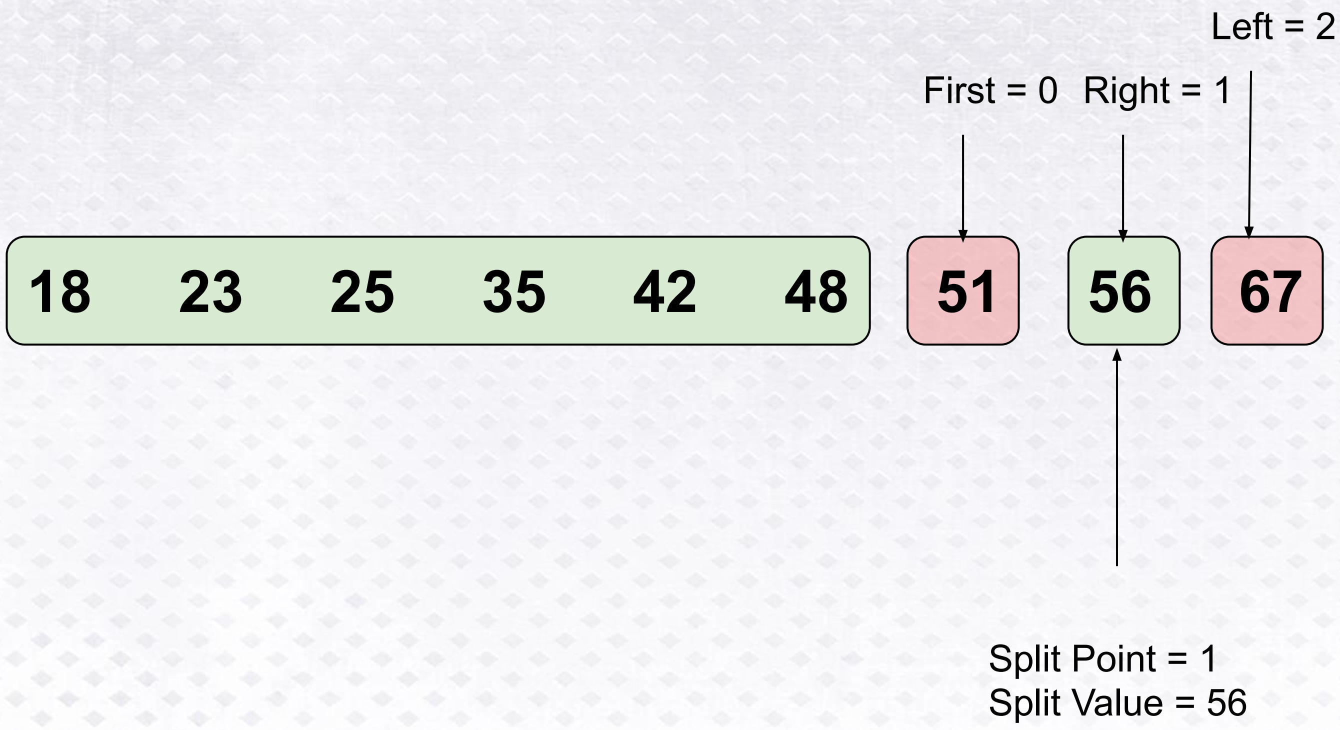
Quick Sort Worked Example



Quick Sort Worked Example



Quick Sort Worked Example



Quick Sort Worked Example

18	23	25	35	42	48	51	56	67

Complexity of Quicksort

In an ideal situation quicksort will split the list in half every time so its complexity will be the same as merge sort $O(n \log(n))$

But it is better than merge sort as it sorts in place (no extra array)

In the worst case it may only sort one value at a time and degenerate into an $O(n^2)$ algorithm

Comparison of Sorting Methods

- *Selection Sort*
  - *Time $O(n^2)$; exchanges $O(n)$*

- *Bubble Sort*
  - *Time $O(n^2)$; exchanges max $O(n^2)$*
  - *But can be modified to be much better with almost sorted data*

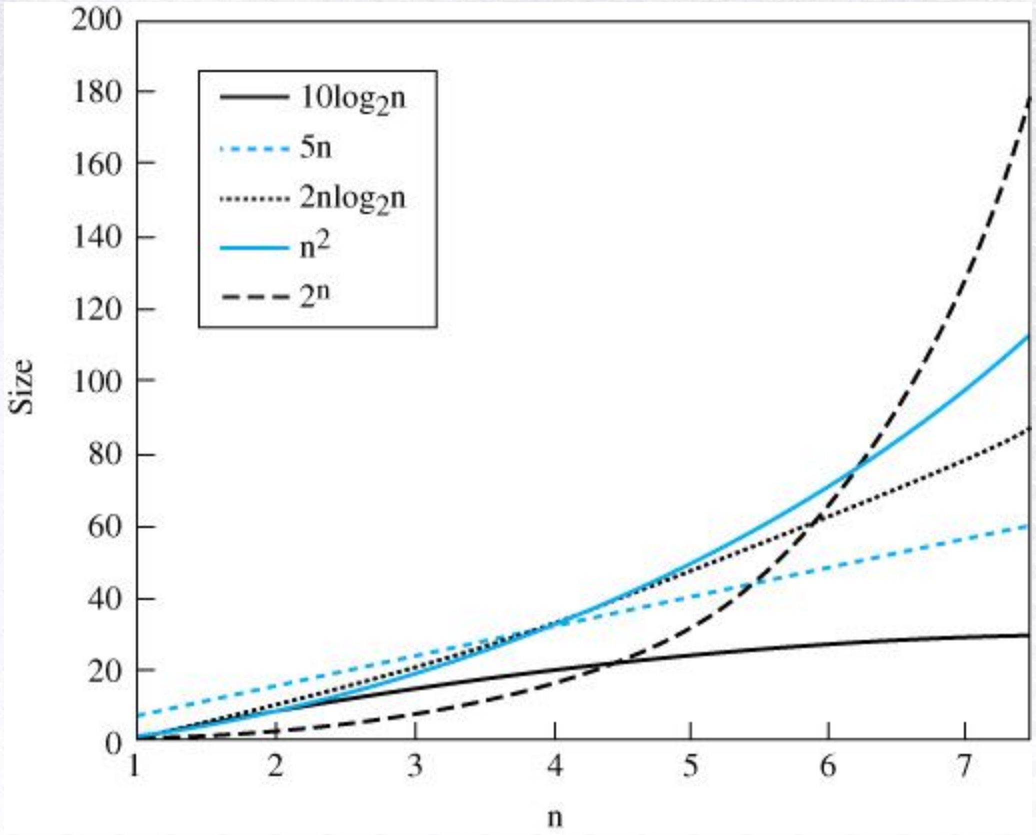
- *Insertion Sort*
  - *Time $O(n^2)$; exchanges max $O(n^2)$*
  - *Can be good choice to sort data as it is generated*

Comparison of Sorting Methods

- *Merge Sort*
  - *Time O(n log(n)); exchanges O(n), but requires 2n space*

- *Quicksort*
  - *Time (best) O(n log(n)); exchanges ??*
  - *Time can be $O(n^2)$ in worst case*

Comparison of Growth Curves



Important Threads

Information Hiding

The practice of hiding the details of a module with the goal of controlling access to it

Abstraction

A model of a complex system that includes only the details essential to the viewer

Information Hiding *and* ***Abstraction*** *are two sides of the same coin*

Important Threads

Data abstraction

Separation of the logical view of data from their implementation

Procedural abstraction

Separation of the logical view of actions from their implementation

Control abstraction

Separation of the logical view of a control structure from its implementation

Important Threads

Identifiers

Names given to data and actions, by which

- *we access the data and*

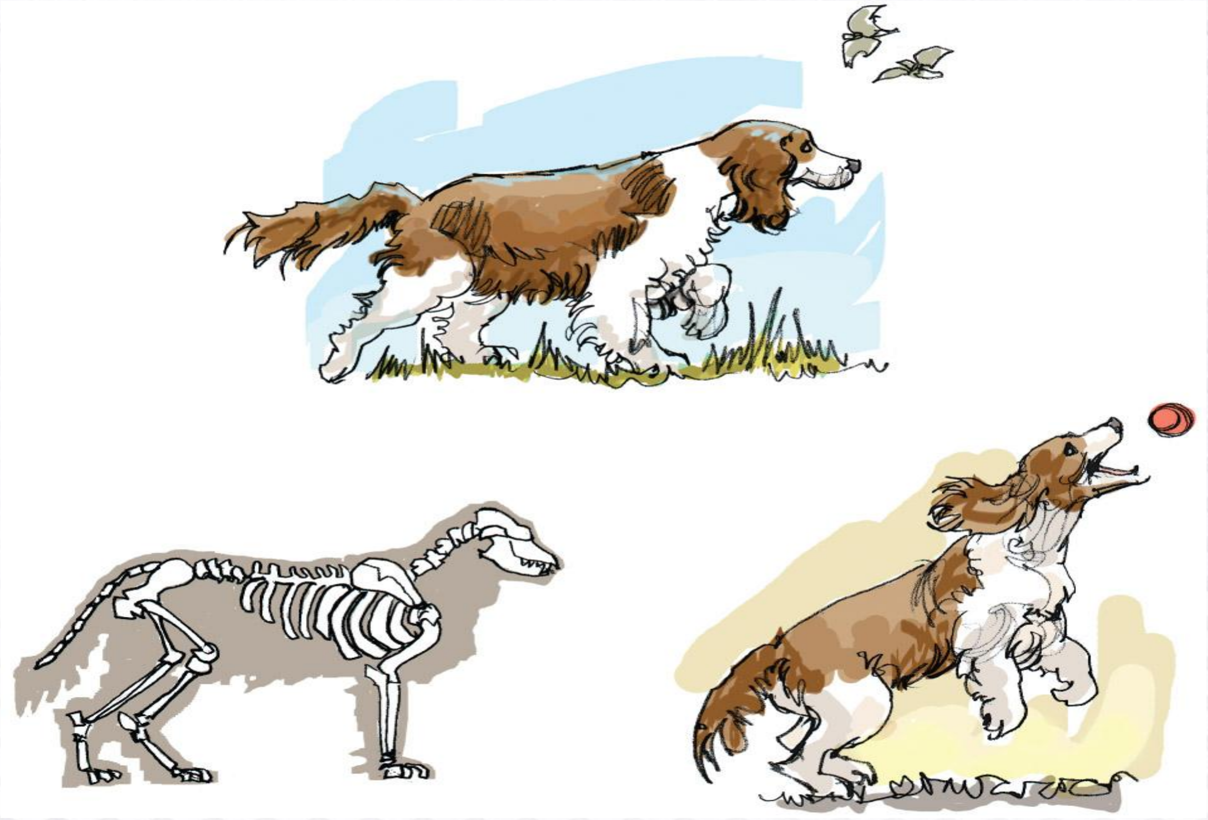
  Read firstName, Set count to count + 1

- *execute the actions*

  Split(splitVal)

Giving names to data and actions is a form of abstraction

Important Threads



Abstraction is the most powerful tool people have for managing complexity!

Ethical Issues

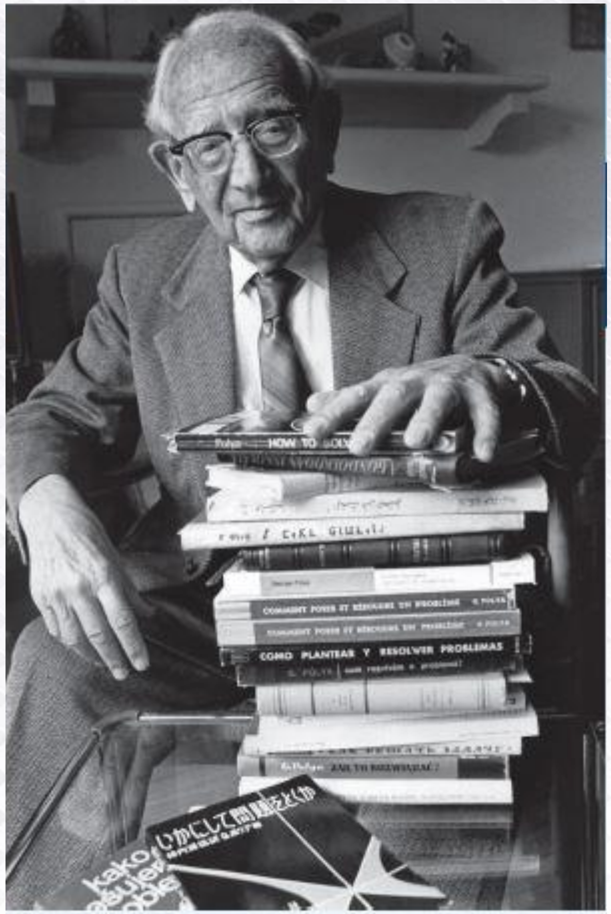
Open-Source Software Development

What are the advantages and disadvantages of open-source software?

What does the success of Linux suggest about the future of open-source software?

Should open-source software be licensed and subject to standard copyright laws?

Who am I?



© AP Photos

I am a mathematician.

Why is my picture in a book about computer science?

Do you know?

What writing system did the Rosetta stone serve as a key to translating?

What did the National Intellectual Property Rights Coordination Center warn the American people about in 2013?

What is piggybacking? Is it ethical?

What parallels are there between philosophy and object oriented software engineering?