Chapter 9

Algorithms | Searching and Sorting

Types of Algorithms

An ***algorithm*** is a sequence of steps or procedures for solving a specific problem

- ***Recursion vs Iteration***
- ***Search Algorithms***
- ***Sorting Algorithms***

Search Algorithms

Search Algorithms are designed to check for an element or retrieve an element from a data structure.

Search algorithms are usually categorized into one of the following:

- ***Sequential Search (Linear Search) -*** searching a list or vector by traversing sequentially and checking every element
- ***Interval Search (ex: Binary Search) -*** specified for searching sorted data-structures (lists). More efficient than linear search.

Linear Search

Linear Search is a search algorithm that starts from the beginning of an array / vector and checks each element until a search key is found or until the end of the list is reached.

myArray	10	23	4	1	32	47	7	84	3	34

Search Key = 47

Result: index **5**

example: linear_search.cpp

Binary Search

Binary Search is a search algorithm used to sort a sorted list by repeatedly diving the search interval in half.

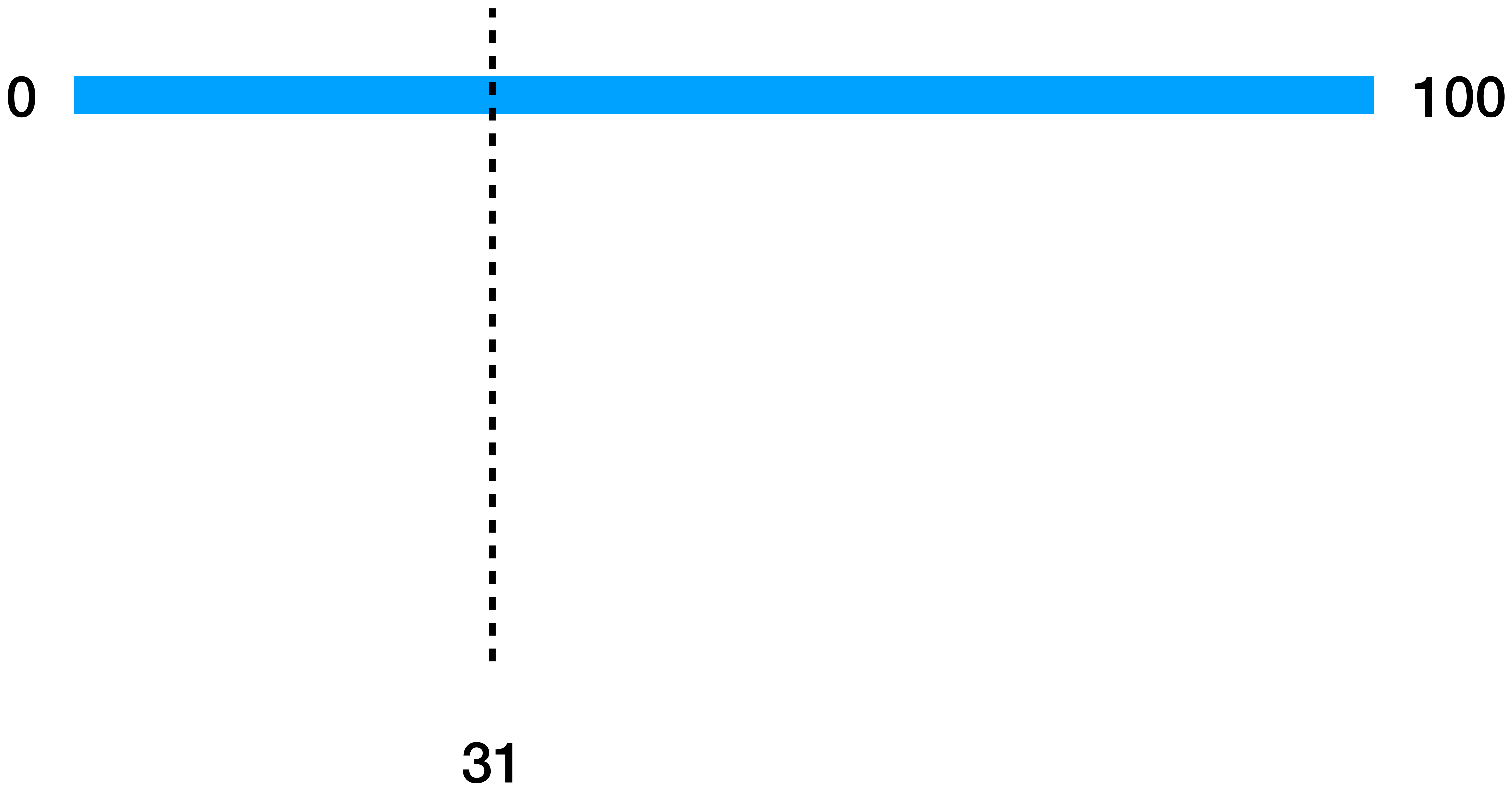
`myArray`	1	4	7	11	25	47	54	84	98	101

Search Key = 47

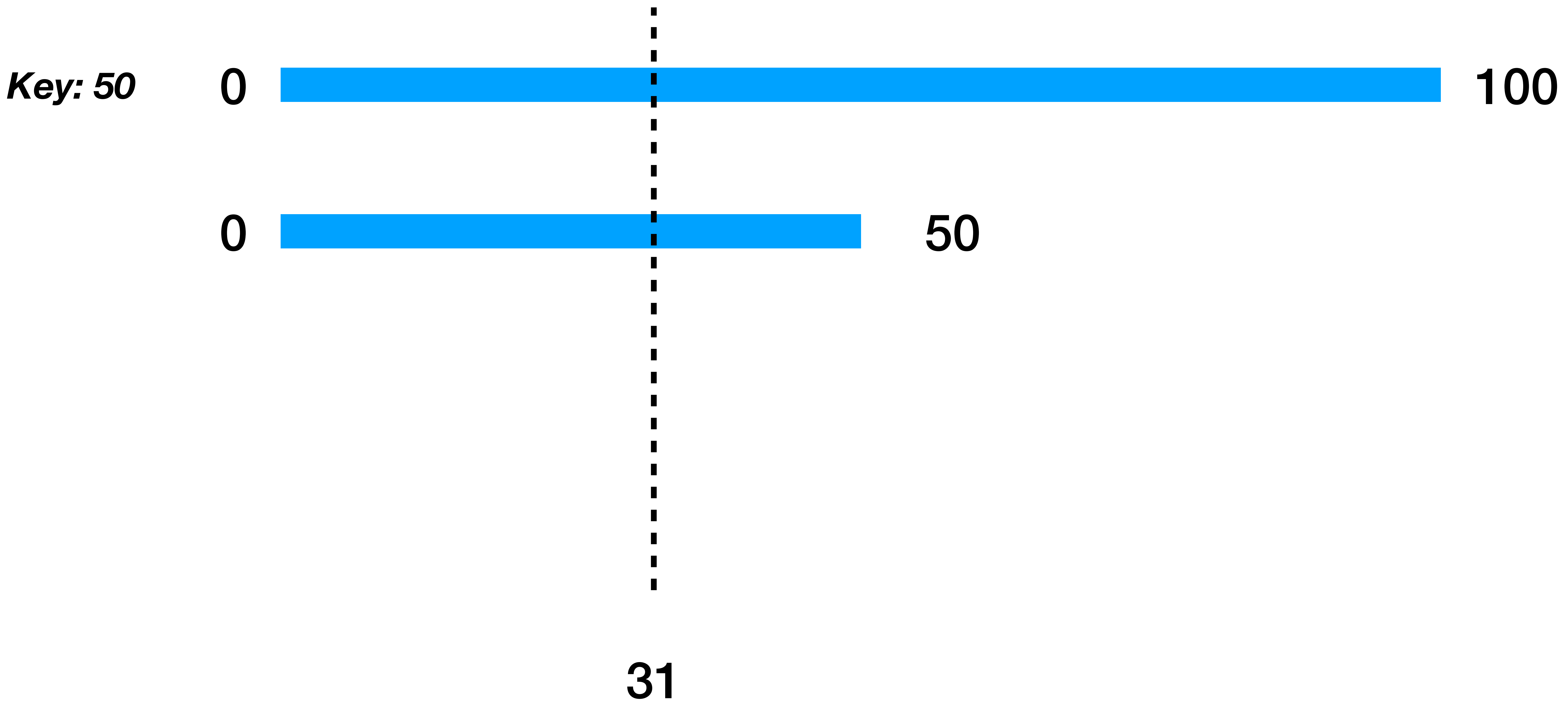
Result: index **5**

example: linear_search.cpp

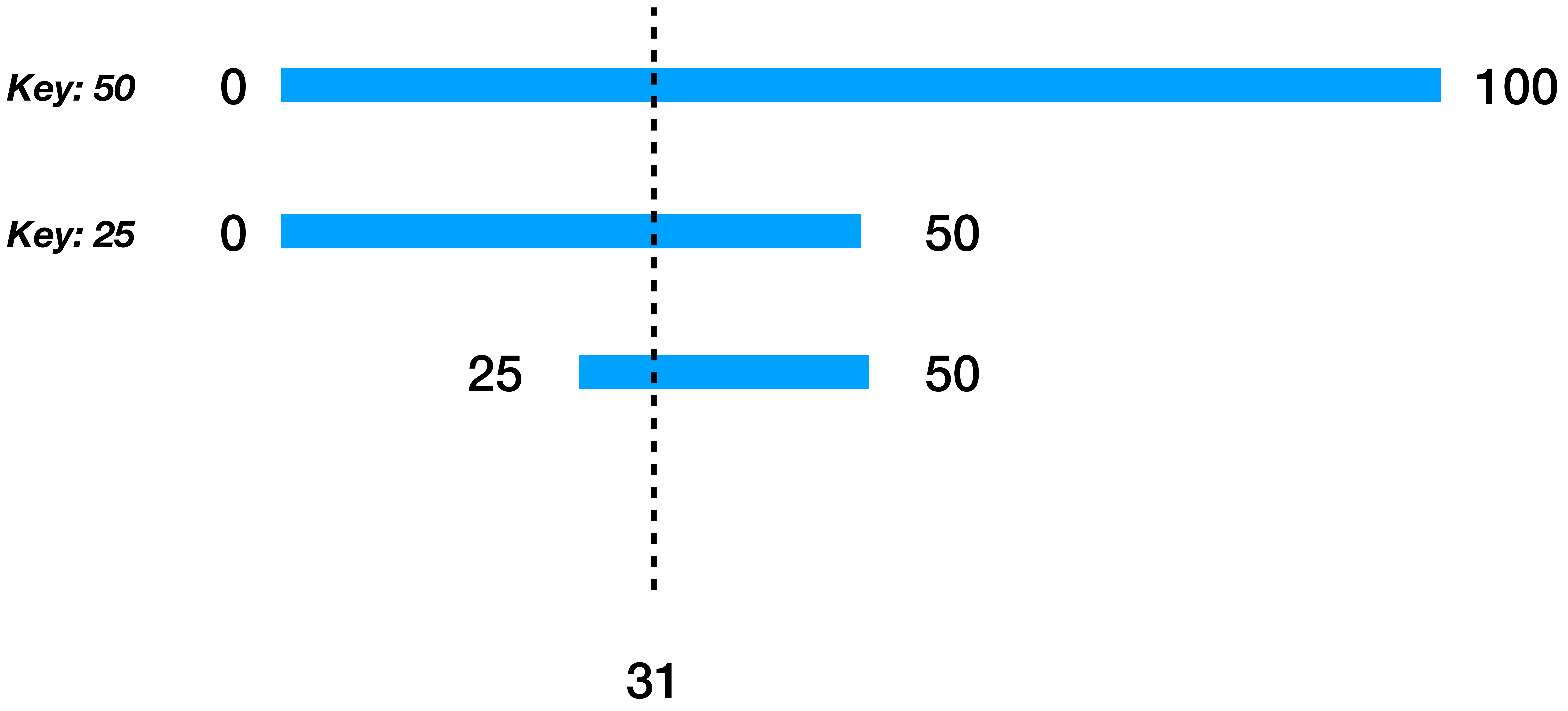
Binary Search



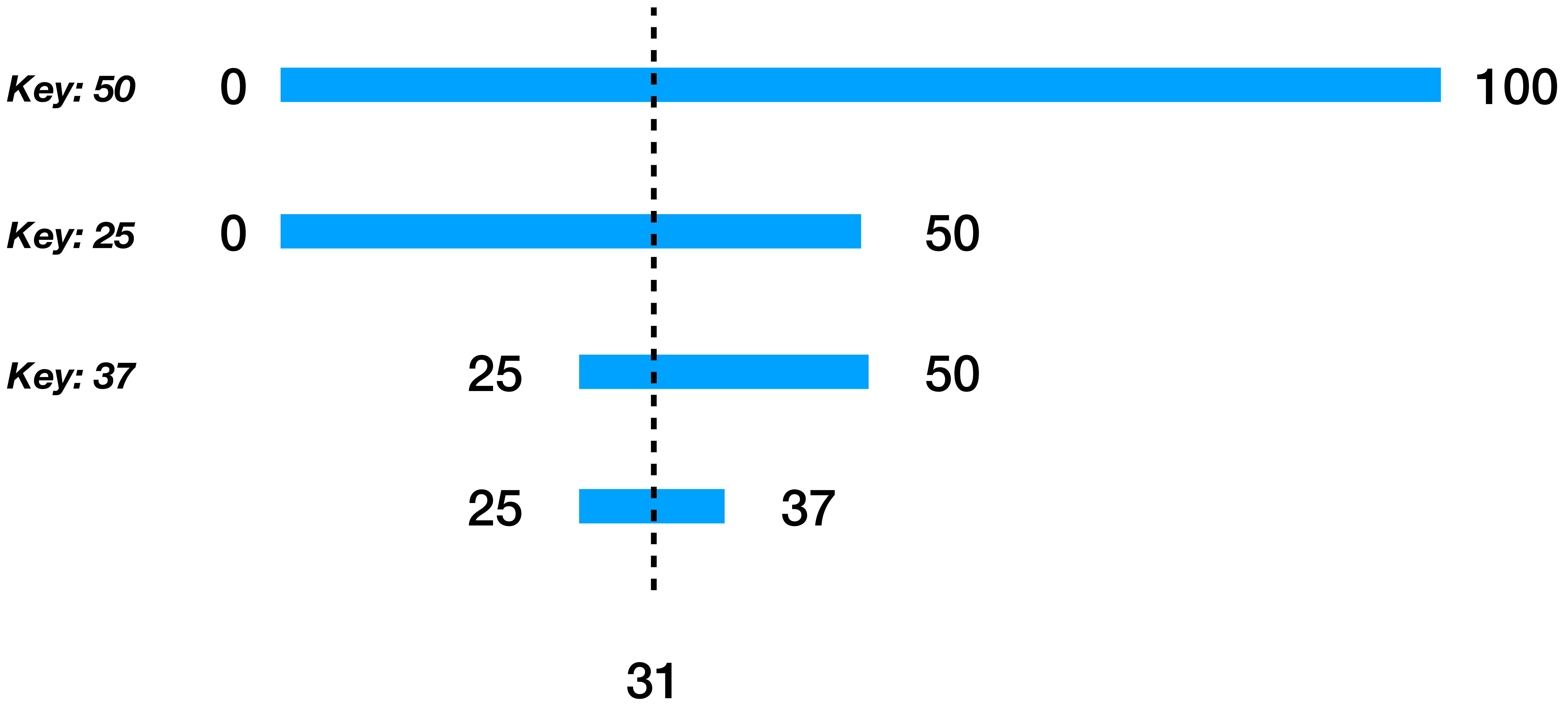
Binary Search



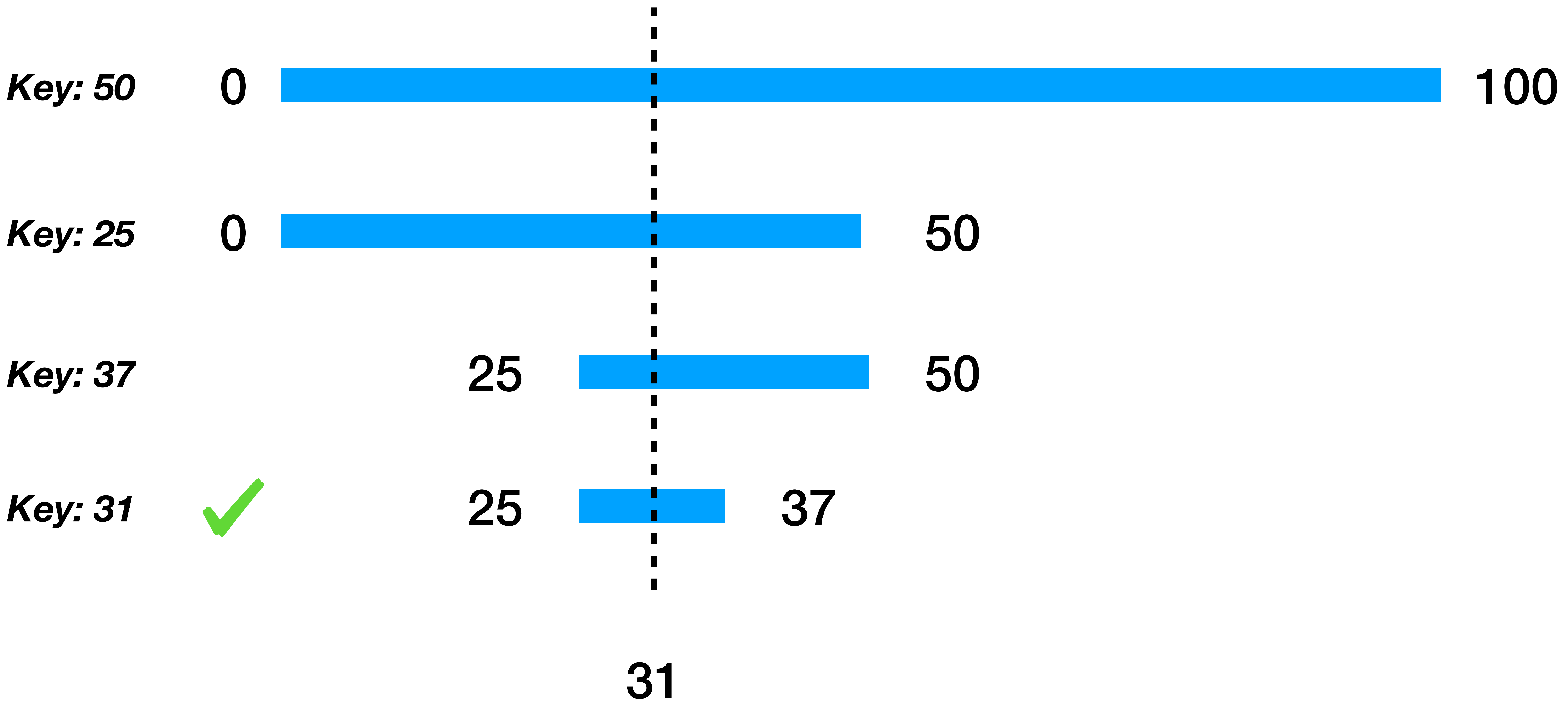
Binary Search



Binary Search



Binary Search



Binary Search

The ***Binary Search*** algorithm is a category of search algorithms that is used to find a value amongst an ordered list.

- binary search works exactly like the Guessing Game!
- This will find a number in an ordered list in at most ***log(n)*** iterations, where ***n*** is the number of items in the list
- This is more preferable to a linear search where we just check each item in the array one by one, performing at most ***n*** iterations.

Algorithm Performance

Runtime - the time an algorithm (function or program) takes to execute.

Example 1: Given a list of 10,000 elements, and if each comparison takes 2***μs,*** what is the fasted possible runtime for linear search?

Example 2: Given a list of 10,000 elements, and if each comparison takes 2***μs,*** what is the longest possible runtime for linear search?

Algorithm Performance

***Runtime* -** the time an algorithm (function or program) takes to execute.

Example 1: Given a list of 10,000 elements, and if each comparison takes 2 ***μs,*** what is the fasted possible runtime for linear search?

2 μs

Example 2: Given a list of 10,000 elements, and if each comparison takes 2 ***μs,*** what is the longest possible runtime for linear search?

20000 μs

Big-O Notation

Big-O Notation is a way of describing how a function generally behaves in relation to the input size.

We use Big-O to classify different algorithms in terms of their performance.

O = Ordnung (German) - means **order of approximation**

1. If ***f(x)*** is a sum of several terms, the highest order term is kept and others are discarded
2. If ***f(x)*** has a term of several factors, all constants are omitted

Big-O Notation Continued

We can also determine the ***Big-O*** for algorithms of composite functions

i.e. we can represent the overall Big-O of 2 or more algorithms or functions that are used in tandem.

Figure 9.3.1: Rules for determining Big O notation of composite functions.

Composite function	Big O notation
$c \cdot O(f(N))$	$O(f(N))$
$c + O(f(N))$	$O(f(N))$
$g(N) \cdot O(f(N))$	$O(g(N) \cdot f(N))$
$g(N) + O(f(N))$	$O(g(N) + f(N))$

Feedback?

Big-O Notation Continued

Let's look at some run times of different functions with different ***input*** sizes.

Table 9.3.1: Growth rates for different input sizes.

Function	N = 10	N = 50	N = 100	N = 1000	N = 10000	N = 100000
log N	3.3 μs	5.65 μs	6.6 μs	9.9 μs	13.3 μs	16.6 μs
N	10 μs	50 μs	100 μs	1000 μs	10 ms	1 s
N log N	.03 ms	.28 ms	.66 ms	.099 s	.132 s	1.66 s
N^2	.1 ms	2.5 ms	10 ms	1 s	100 s	2.7 hours
N^3	1 ms	.125 s	1 s	16.7 min	11.57 days	31.7 years
2^N	.001 s	35.7 years	*	*	*	*

Tip

As ***N*** grows, an algorithms performance has a greater impact on the runtime

Big-O Notation

O(1) - Constant Time: no matter the size of the input, the algorithm still completes in the same amount of time.

```
int getFirstItem(vector<int> &numbers) {
  cout << numbers.at(0) << endl;
  return numbers.at(0);
}
```

The efficiency of any ***constant time*** algorithm or operation is not affected by the input size.

Big-O Notation

O(n) - Linear Time: The amount of steps / time needed to complete this type of algorithm increases linearly as ***n*** grows.

```
int printVector(vector<int> myVector) {
  for (int i = 0; i < myVector.size(); i++) {
    cout << myVector.at(i) << endl;
  }
}
```

Big-O Notation

O(n²) - Quadratic Time: here we have to run ***n * n*** iterations to determine all possible ordered pairs of a vector

```
void printAllPossibleOrderedPairs(const vector<int>& items) {
  for (int firstItem : items) {
    for (int secondItem : items) {
      cout << firstItem << ", " << secondItem << endl;
    }
  }
}
```

Tip

A good rule of thumb is any time we see a nested loop, this typically is a good indicator of ***$O(n^2)$***

Sorting Algorithms

Sorting a list of elements

Sorting Algorithms

Sorting Algorithms convert lists of elements into ascending or descending order.

```
{ 14, 24, 4, 67, 2, 68, 12, 6 }
              ↓
{ 2, 4, 6, 12, 14, 24, 67, 68 }
```

The ***Sorting Algorithms*** that we will discuss are:

- Selection Sort
- Insertion Sort
- Quicksort
- Merge Sort

Selection Sort

Selection Sort - sorting algorithm that that treats the input as 2 different parts

- A Sorted list
- An Unsorted List

Selection sort repeatedly selects the appropriate value from the unsorted part and appends it to the end of the sorted list by utilizing a series of ***swaps***

example: selection_sort.cpp

Insertion Sort

Insertion Sort - sorting algorithm that that treats the input as 2 different parts

- A Sorted list
- An Unsorted List

Insertion Sort repeatedly inserts the next value from the unsorted list into the correct position of the sorted list using ***swaps***

example: insertion_sort.cpp

Quicksort

Quicksort partitions the input into low and high parts then recursively sorts each partition.

The bulk of the work is done once we identify the ***pivot***

- ***Pivot -*** quick sort uses a pivot (a selected value) as the division point in the list. Usually we arbitrarily choose one.

example: quicksort.cpp

Merge sort

Merge sort divides a list into 2 halves, recursively sorts each half, then merges the sorted halves to produce a sorted list

example: merge_sort.cpp

Watching sorting in action!

https://www.youtube.com/watch?v=kPRA0W1kECg