Lecture 6:
Sorting and Searching

Stephen Huang

March 21, 2023

Contents

1. Python Sorting Functions

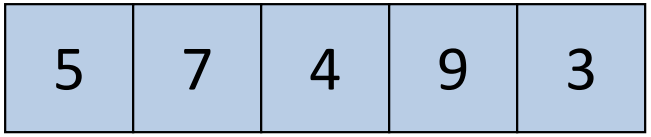
- A typical task that requires a list/array so that all the data values can be stored in memory simultaneously is sorting values into order.
- For numbers, the order desired might be ascending order, i.e.,

$$a[0] \leq a[1] \leq \ldots \leq a[n-1].$$

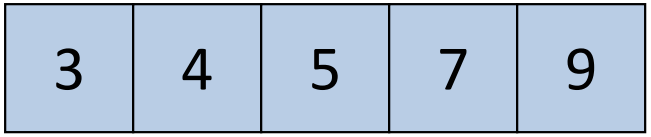
- There are many sorting algorithms known with different efficiency.

Example

Before:



After:



Sorting

- There are two functions related to sorting.
  - `sort(key, reverse)`
  - `sorted(iterable, key, reverse)`

- Note that `sort()` is a method of the list object, but `sorted()` is a built-in function of Python.

Sort()

- This method sorts the list in place, using only less-than-comparisons (<) between items.
- Exceptions are not suppressed - the entire sort operation will fail if any comparison operations fail.
- sort() accepts two arguments that can only be passed by keyword (keyword-only arguments):
  - The key specifies a function (of one argument) that is used to extract a comparison key from each element in *iterable*
  - The reverse is a Boolean value.

Sort()

- The default comparison is the comparison of the values in the list (<). With the function parameter, we can change that.
- The function, call it f(), is applied to all elements, and the sorting is based on the function values f(e) instead of e.
  - Compare by the length of strings
  - Compare by the absolute value of numbers
- The sort() method is guaranteed to be stable. A sorting is stable if it does not change the relative order of elements that compare equally.

Why sorted()?

Comparison	sort()	sorted()
Method	A method of list object	A built-in function
Calling Exam	list.sort (key, reverse)	list = sorted (iterable, key, reverse)
What's being sorted	The list itself	The iterable (list, tuple, etc.)
What's Returned	The list is being sorted in-site	Returns a list, iterable unchanged

Sorted()

- Default parameters:
  sorted(iterable, key=None, reverse=False)
  - Returns a new sorted list from the items in iterable.
  - Built-in, no need to import anything.
- The purposes of the two parameters are similar to those of the sort().

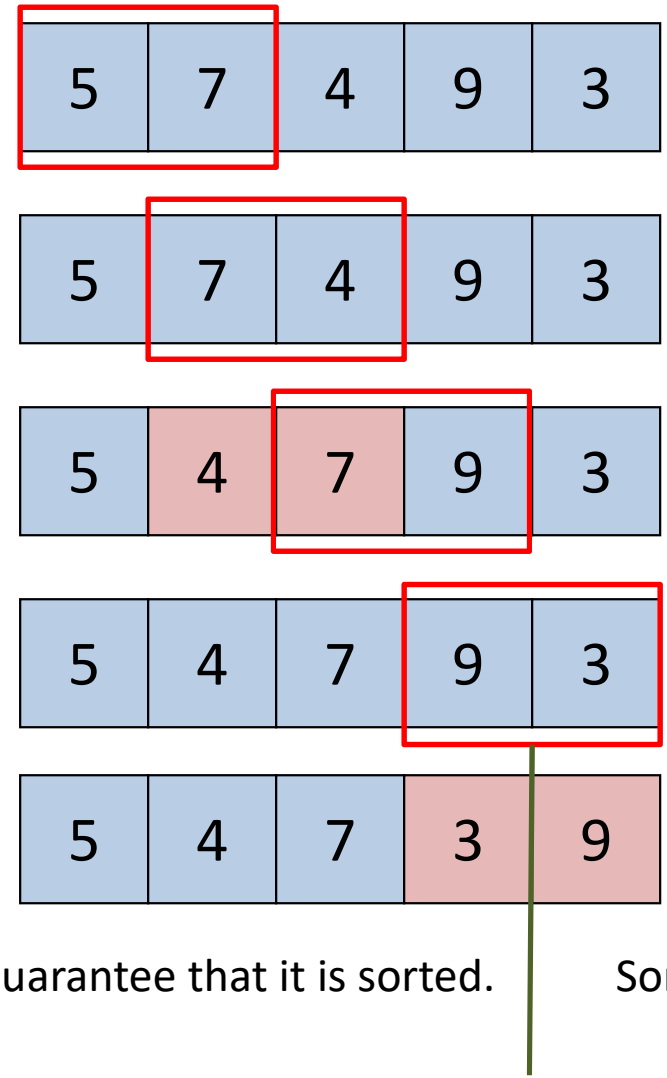
2. Sorting Algorithms

- We will introduce two "simple" sorting algorithms:
  - Bubble sort
  - Selection sort
- These algorithms are "simple" because it is relatively easy to explain. It does not mean the algorithms are efficient to execute.
- In fact, they are not efficient. They take $O(n^2)$ units of time for a list of length n.
- The more efficient algorithms take O(n log n) units of time.

Sorting Algorithms

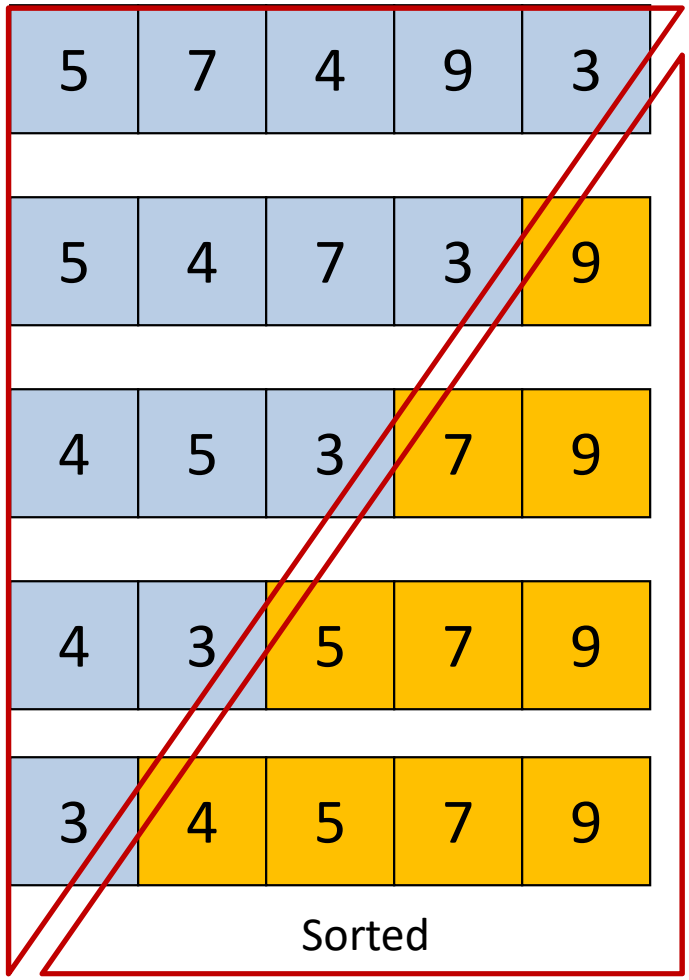
- Most sorting algorithms compare list elements (<, =, >) and move elements around to put them in the proper order.
- Typically, they "swap" elements in the list to move the elements.
- Sometimes, we measure the efficiency of these algorithms by the number of comparisons or swaps.
- Ideally, they should also use a small number of fixed memory spaces to achieve the goal.

Bubble sort


5 7 4 9 3
5 7 4 9 3
5 4 7 9 3
5 4 7 9 3
5 4 7 3 9
No guarantee that it is sorted.
Sorted

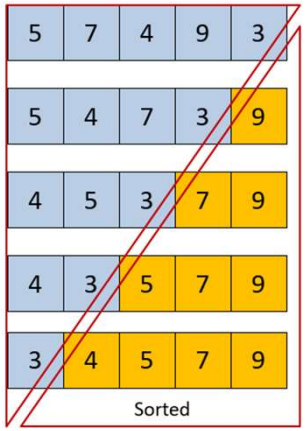

Bubble Sort

Bubble sort



Bubble sort

```
def bubble(a):
    for ub in range(len(a)-1, 0, -1):
        for i in range(ub):
            # for the unsorted part
            if a[i]>a[i+1]: # not in order
                # swap them
                a[i],a[i+1] = a[i+1],a[i]
    return(a)
```



Tracing the code

[19, 3, **41**, **45**, 22, **111**, 2, 29]

[3, 19, **41**, 22, **45**, 2, 29, **111**]

[3, 19, 22, **41**, 2, 29, **45**, **111**]

[3, 19, 22, 2, 29, **41**, **45**, **111**]

[3, 19, 2, 22, 29, **41**, **45**, **111**]

[3, 2, 19, 22, 29, **41**, **45**, **111**]

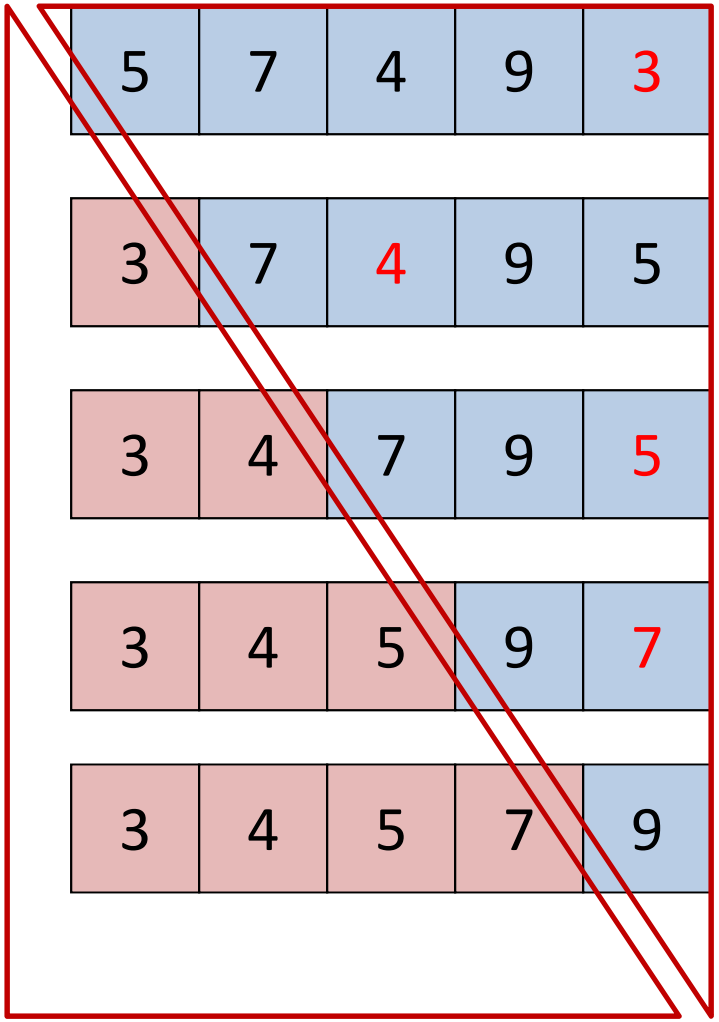
[2, 3, 19, 22, 29, **41**, **45**, **111**]

[2, 3, 19, 22, 29, **41**, **45**, **111**]

Selection Sort

- Strategy:
  - There are two parts to the list: one sorted and unsorted. The sorted part is empty initially. All the numbers are in the unsorted part.
  - Select the smallest element from the unsorted part and swap it with the first unsorted element.
  - We have decreased the unsorted part's size and increased the sorted part. Keep doing this, and we will have the whole list sorted.

Selection sort



Selection Sort

```
def swap(b, i, j):
    b[i], b[j] = b[j], b[i]

def selection_sort(a):
    for i in range(len(a)-1):
        min_i = i
        for j in range(i+1,len(a)):
            if a[min_i]>a[j]:
                min_i = j
        swap(a, i, min_i)
```

More

```
[5, 6, 8, 7, 4, 3, 2, 1]
[1, 6, 8, 7, 4, 3, 2, 5]
[1, 2, 8, 7, 4, 3, 6, 5]
[1, 2, 3, 7, 4, 8, 6, 5]
[1, 2, 3, 4, 7, 8, 6, 5]
[1, 2, 3, 4, 5, 8, 6, 7]
[1, 2, 3, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 7, 8]
```

3. Linear Search

- Search: Given a list L and a value x, find the index of x in L, or return None.
- Linear Search: Compare with one element of the list at a time from the beginning of the list.
- Binary search:
  - Assume the list is sorted in increasing order
  - Compare with the middle element of the list and search ½ of the list for x depending on the comparison result.

Linear Search 1

Find the (last) one with the highest index.

```
def search(x,a):
    idx = None
    for i in range(len(a)):
        if a[i]==x:
            idx = i
    return idx
```

Linear Search 2

Find the (first) one with the lowest index

```
def search(x,a):
    for i in range(len(a)):
        if a[i]==x:
            return i
    return None
```

You may return at any point inside a function

Make sure there is a return along every path

Linear Search 3

Find the first one with the lowest index.

```
def search(x, a):
    for i, v in enumerate(a):
        if v == x:
            return i
    return None
```

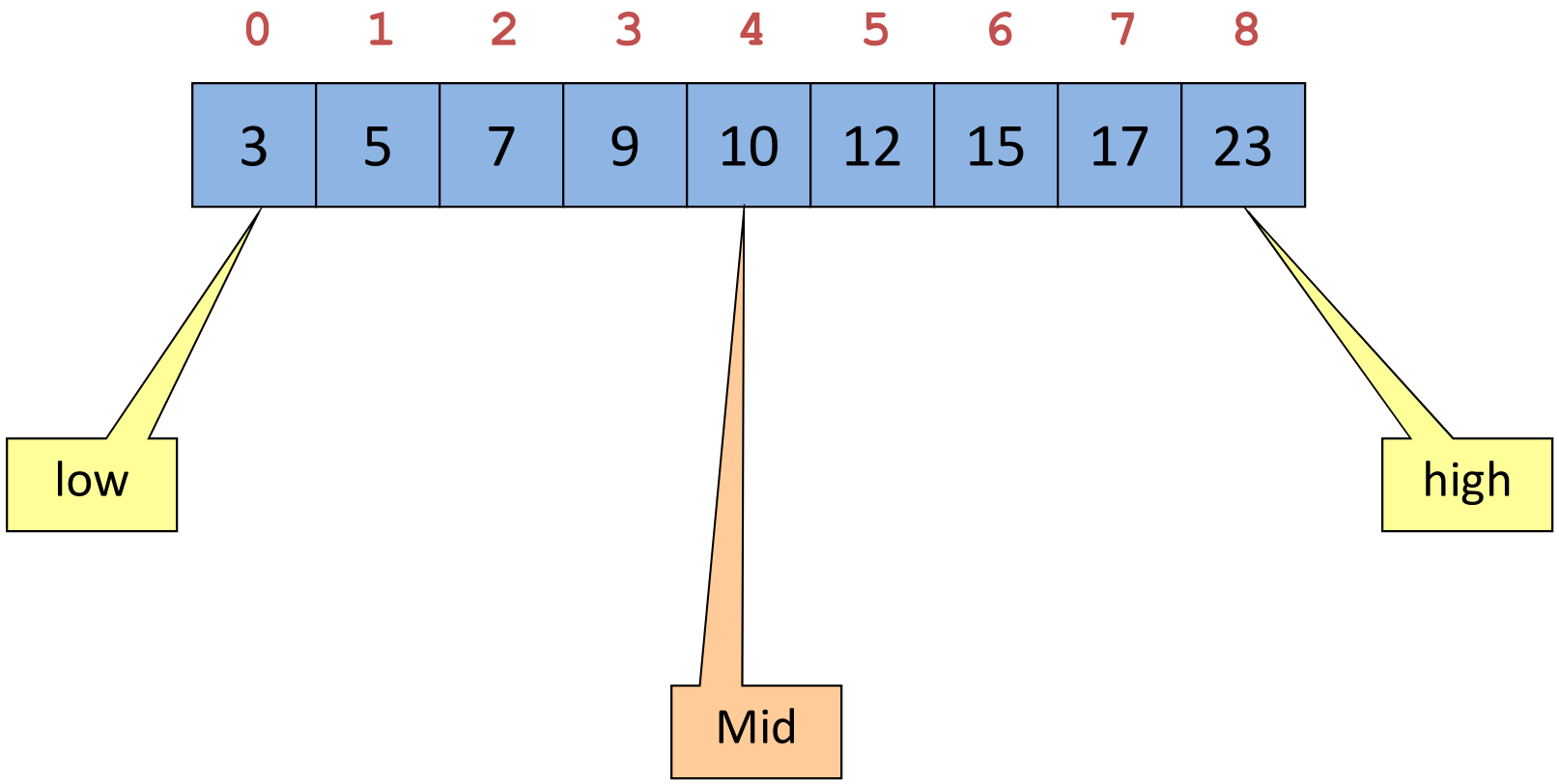
Linear Search 4

```
def search(x,a,lb,ub):
    if lb>=ub:
        return None
    elif a[lb]==x:
        return lb
    else:
        return(search(x,a,lb+1,ub))
```

4. Binary search

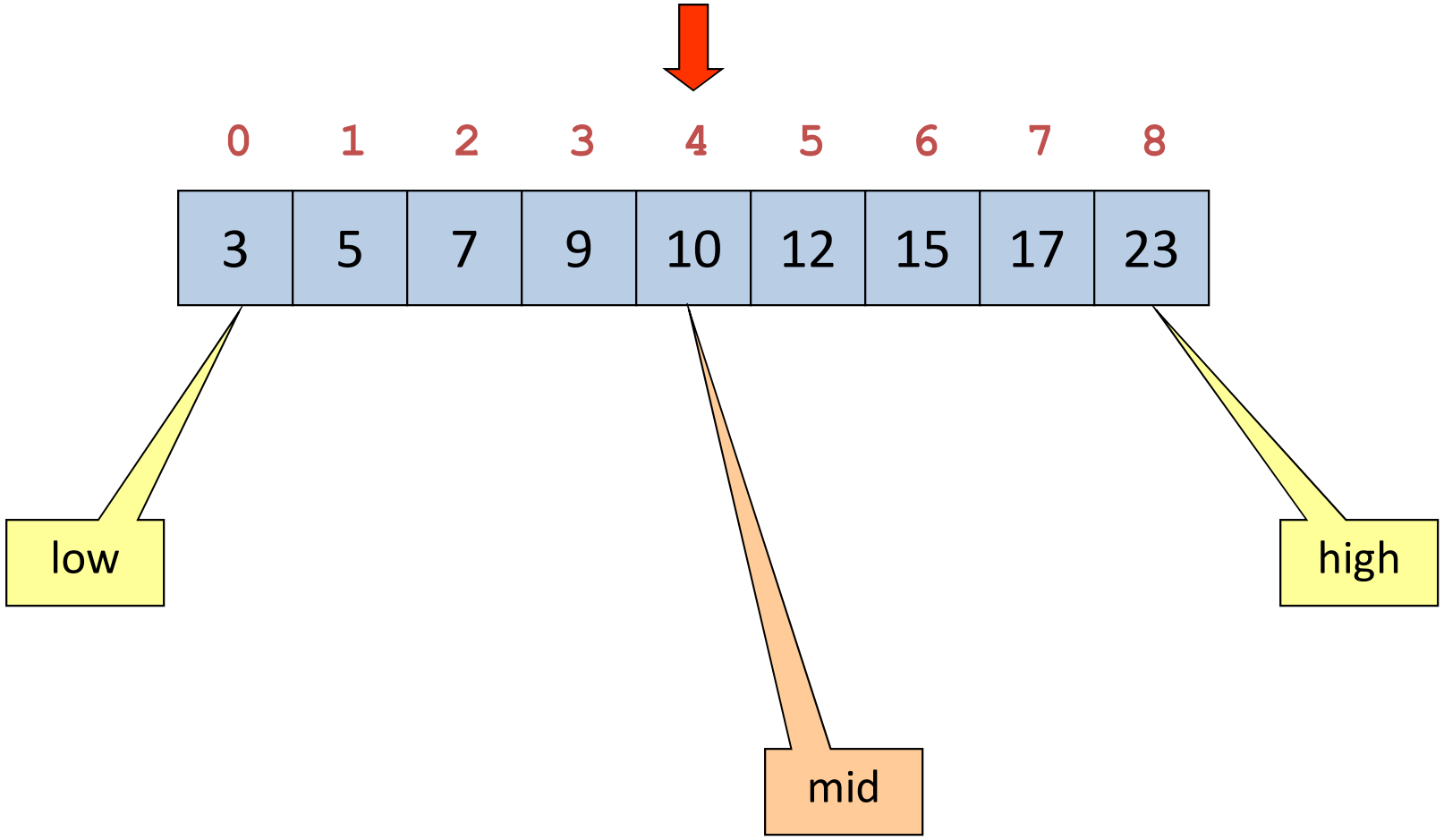
- The binary search assumes the list is already sorted.
- A comparison with the search value x with an element e in the list results in three possibilities:
  - e < x
  - e == x
  - e > x (e>= x if duplicates are allowed in the list)
- So, what will be the best selection of element e?
- It takes only O(log n) units of time to find the element in a list of length n.

Binary Search

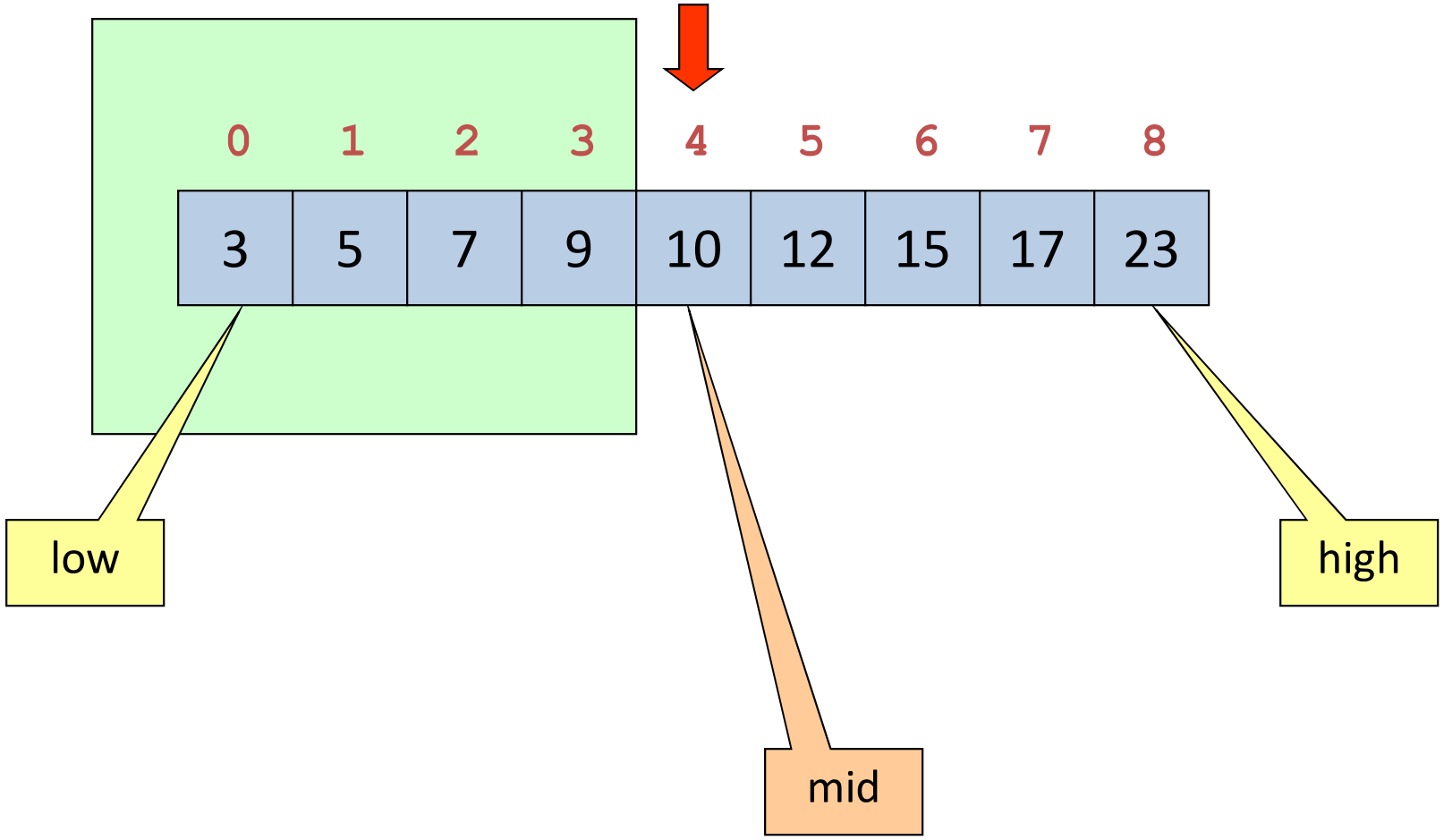


Mid = (0+8)/2 = 4

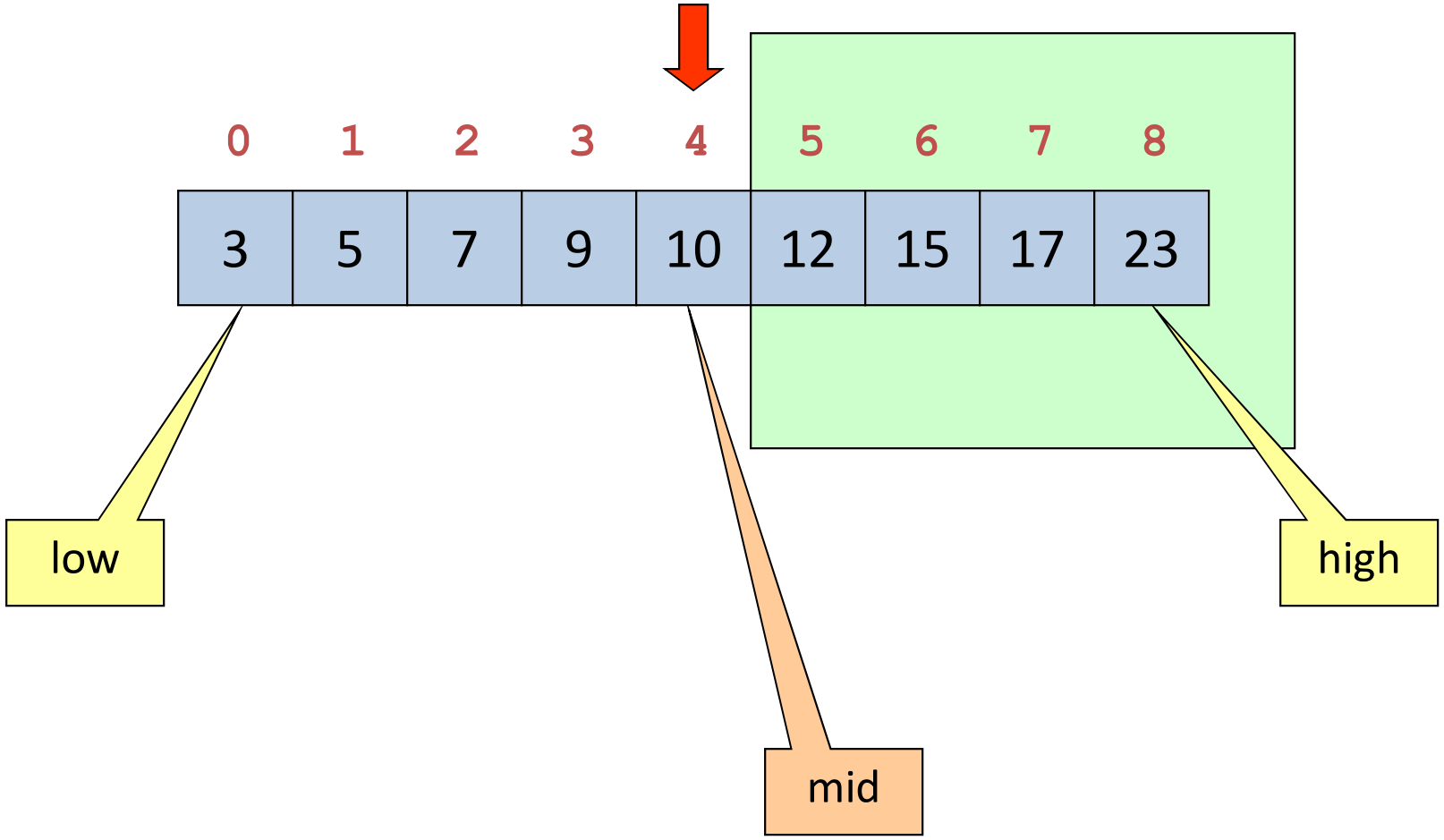
Case 1 (x=10)



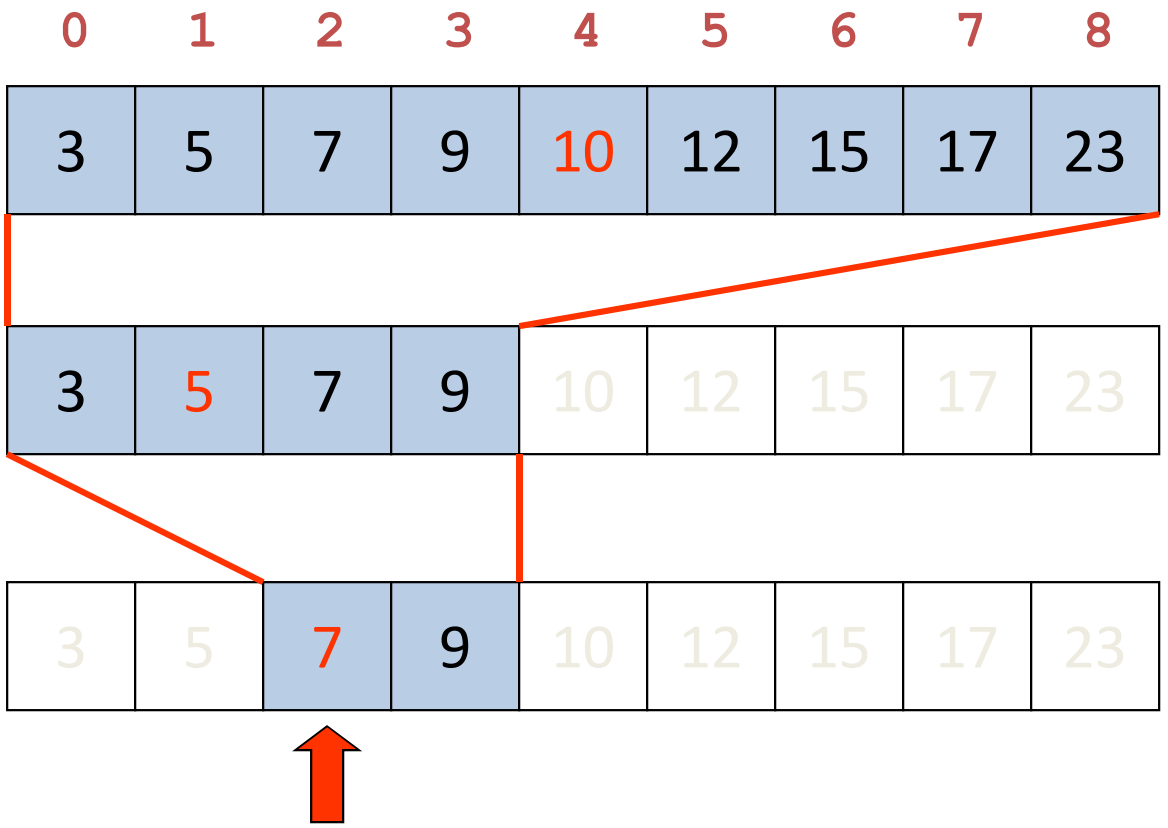
Case 2 (x=5)



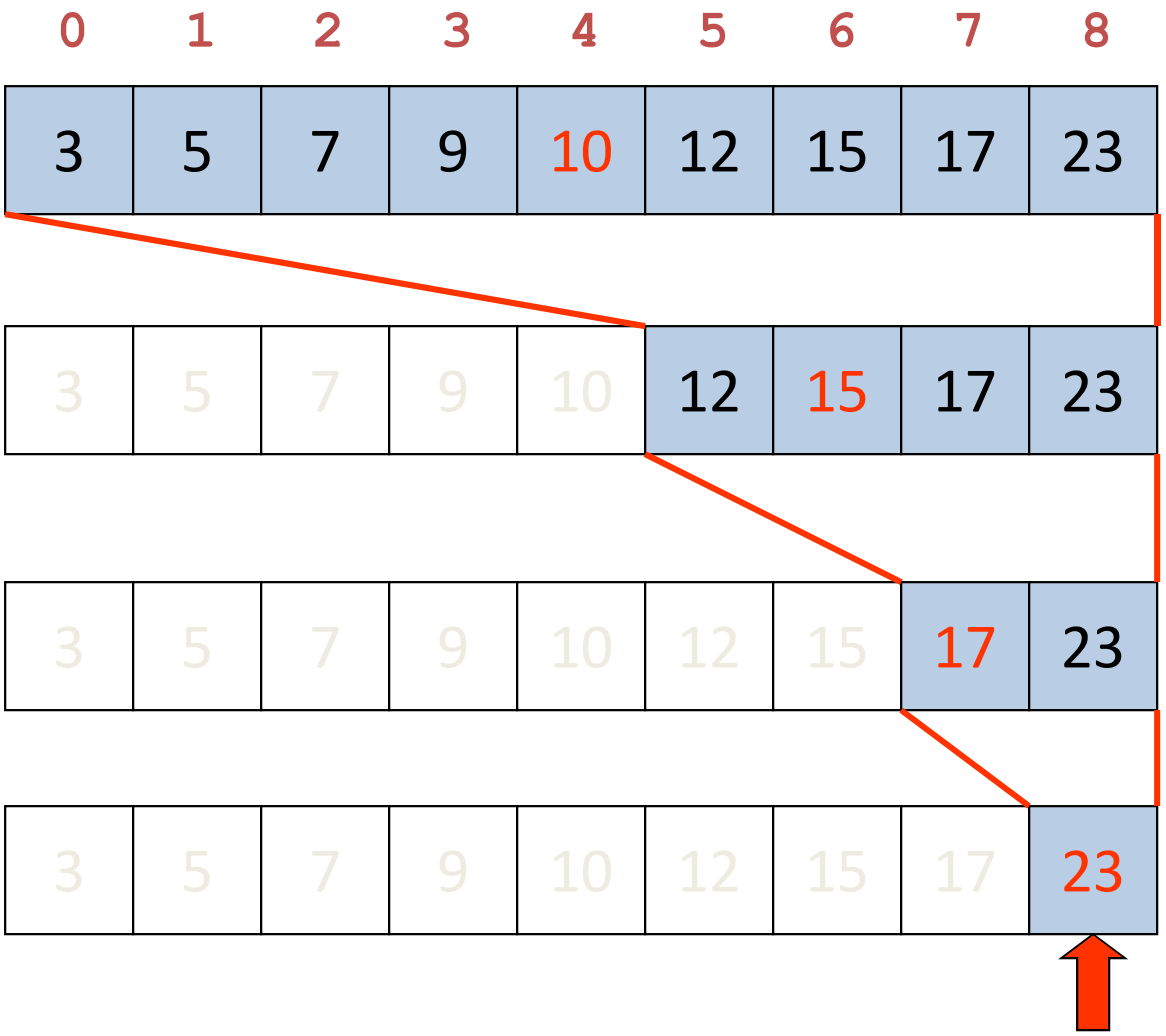
Case 3 (x=17)



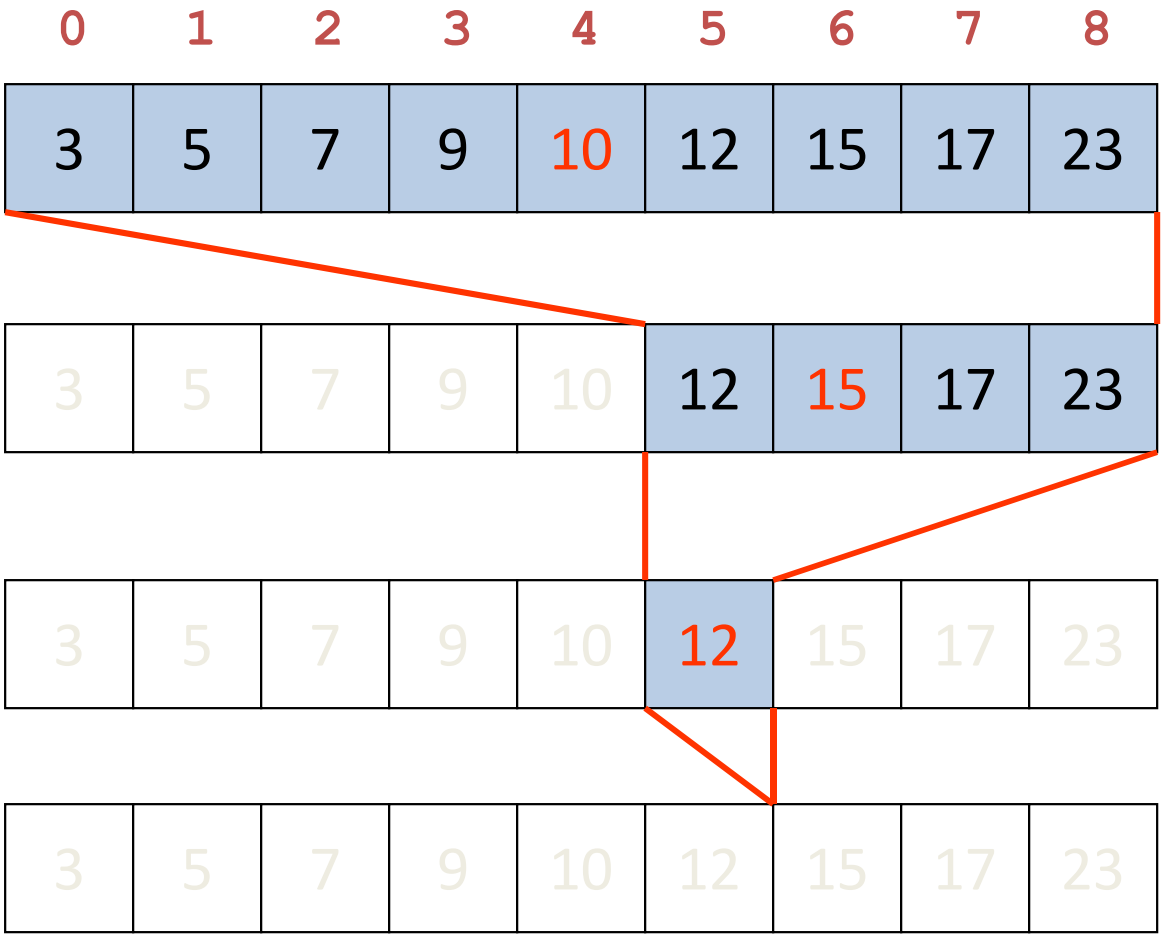
Find 7


0 1 2 3 4 5 6 7 8
3 5 7 9 10 12 15 17 23
3 5 7 9 10 12 15 17 23
3 5 7 9 10 12 15 17 23

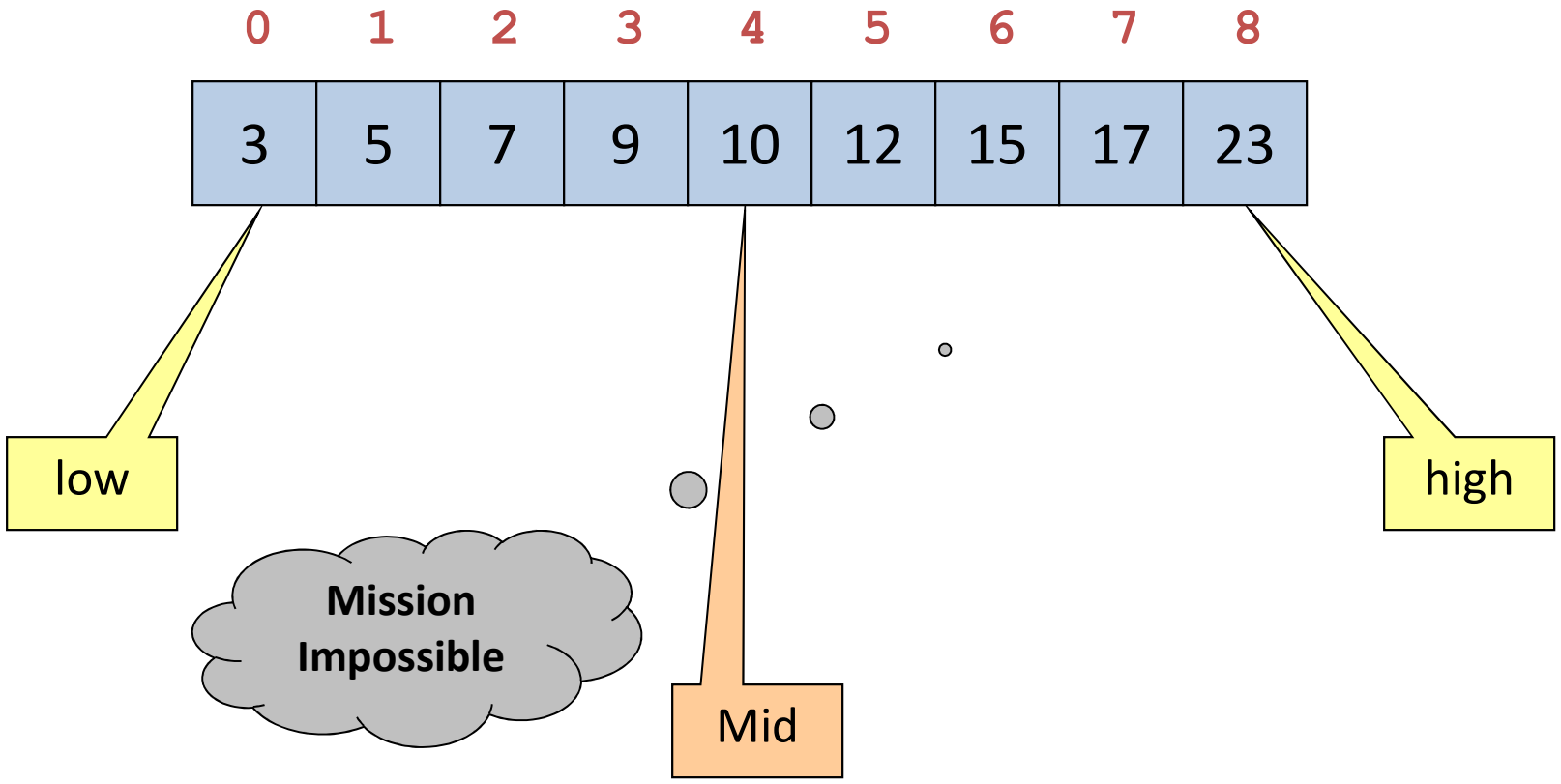
Find 23


0 1 2 3 4 5 6 7 8
3 5 7 9 10 12 15 17 23
3 5 7 9 10 12 15 17 23
3 5 7 9 10 12 15 17 23
3 5 7 9 10 12 15 17 23

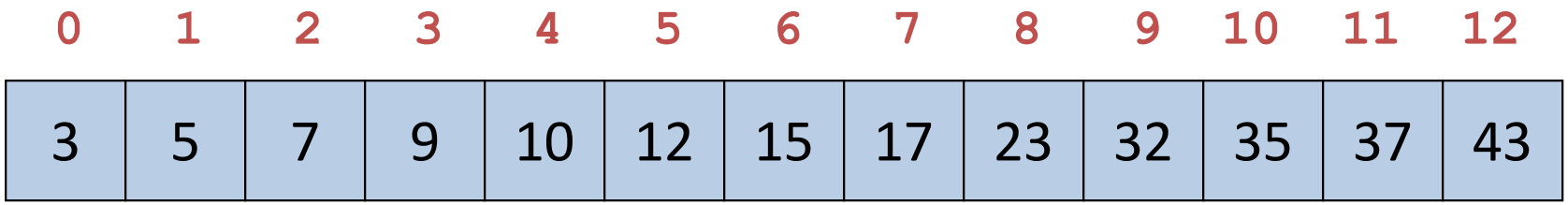
Find 13


0 1 2 3 4 5 6 7 8
3 5 7 9 10 12 15 17 23
3 5 7 9 10 12 15 17 23
3 5 7 9 10 12 15 17 23
3 5 7 9 10 12 15 17 23

Find 13


0 1 2 3 4 5 6 7 8
3 5 7 9 10 12 15 17 23
low
Mid
high
Mission Impossible

Working Example



Non-recursive Binary Search

```
def b_search(x,list):
    low=0
    high=len(list)
    while low<high:
        mid=(low+high)//2
        if x==list[mid]:  return mid
        elif x<list[mid]: high = mid
        else:             low  = mid+1
    return None
```

Recursive Binary Search

```
def b_search(x,list, low, high):
    while low<high:
        mid=(low+high)//2
        if x==list[mid]:
            return mid
        elif x<list[mid]:
            return(b_search(x,list,low,mid))
        else:
            return(b_search(x,list,mid+1,high))
    return None
```