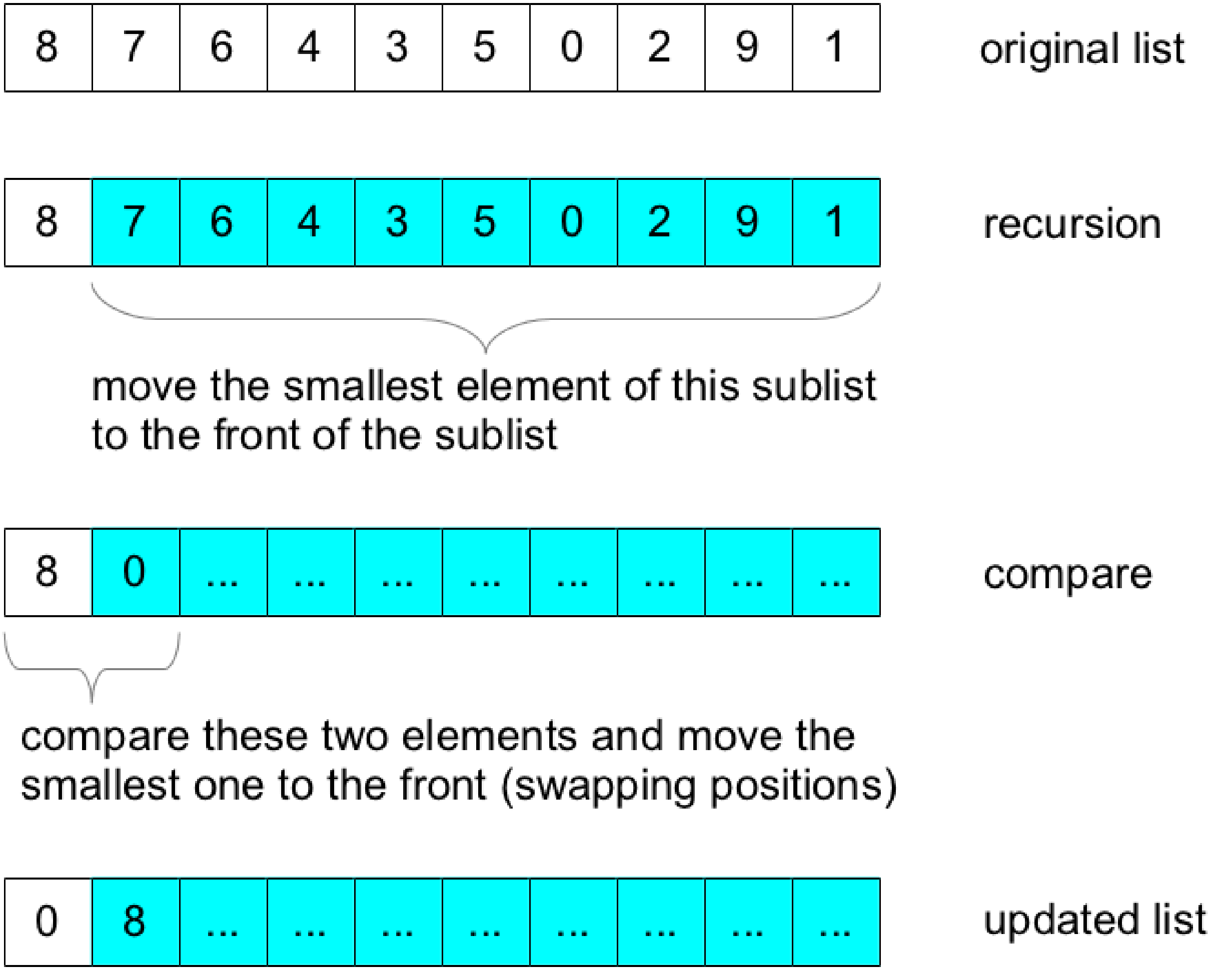
Recursion

notes Chapter 8

Recursively Move Smallest to Front

- recall that we developed a method that moves the smallest element in a list to the front of the list

Recursively Move Smallest to Front



8	7	6	4	3	5	0	2	9	1	original list
8	7	6	4	3	5	0	2	9	1	recursion

move the smallest element of this sublist to the front of the sublist

8	0	...	...	...	...	...	...	...	...	compare

compare these two elements and move the smallest one to the front (swapping positions)

0	8	...	...	...	...	...	...	...	...	updated list

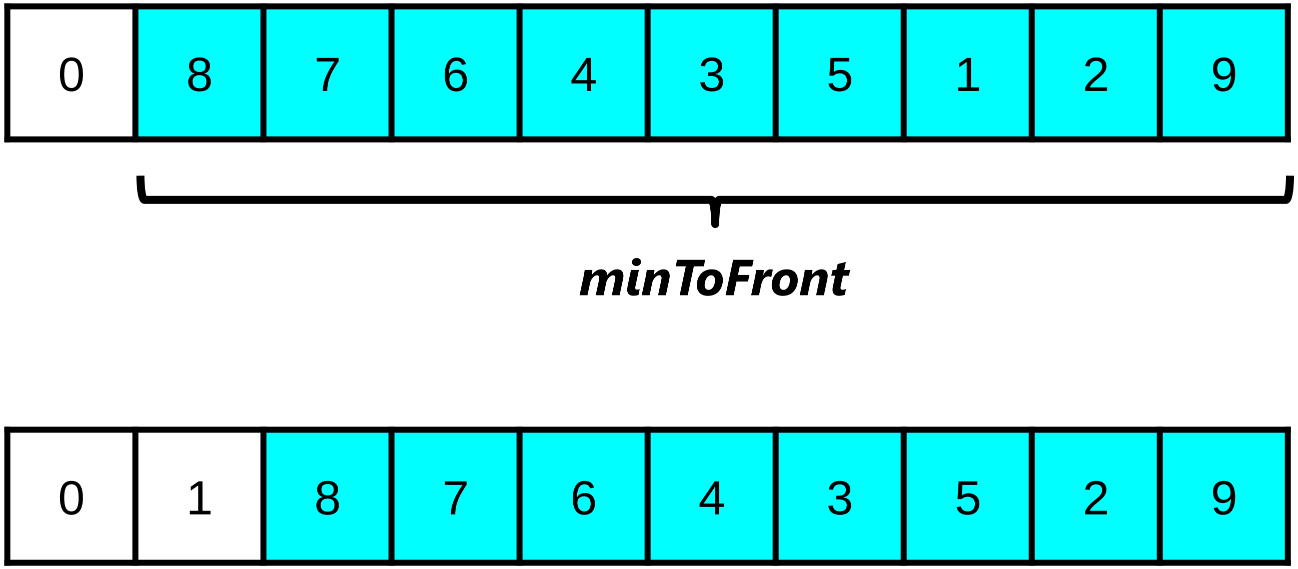
Recursively Move Smallest to Front

```
public class Sort {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Sort.minToFront(t.subList(1, t.size()));
    int first = t.get(0);
    int second = t.get(1);
    if (second < first) {
      t.set(0, second);
      t.set(1, first);
    }
  }
}
```

Sorting the List

- observe what happens if you repeat the process with the sublist made up of the second through last elements:



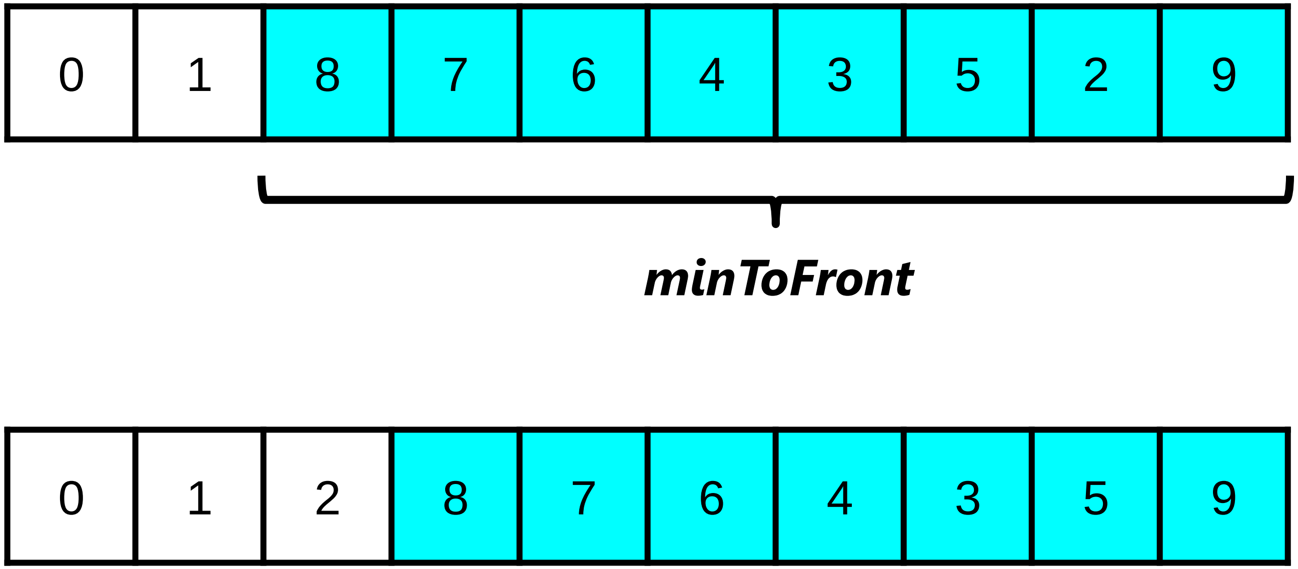
0	8	7	6	4	3	5	1	2	9

minToFront

0	1	8	7	6	4	3	5	2	9

Sorting the List

- observe what happens if you repeat the process with the sublist made up of the third through last elements:



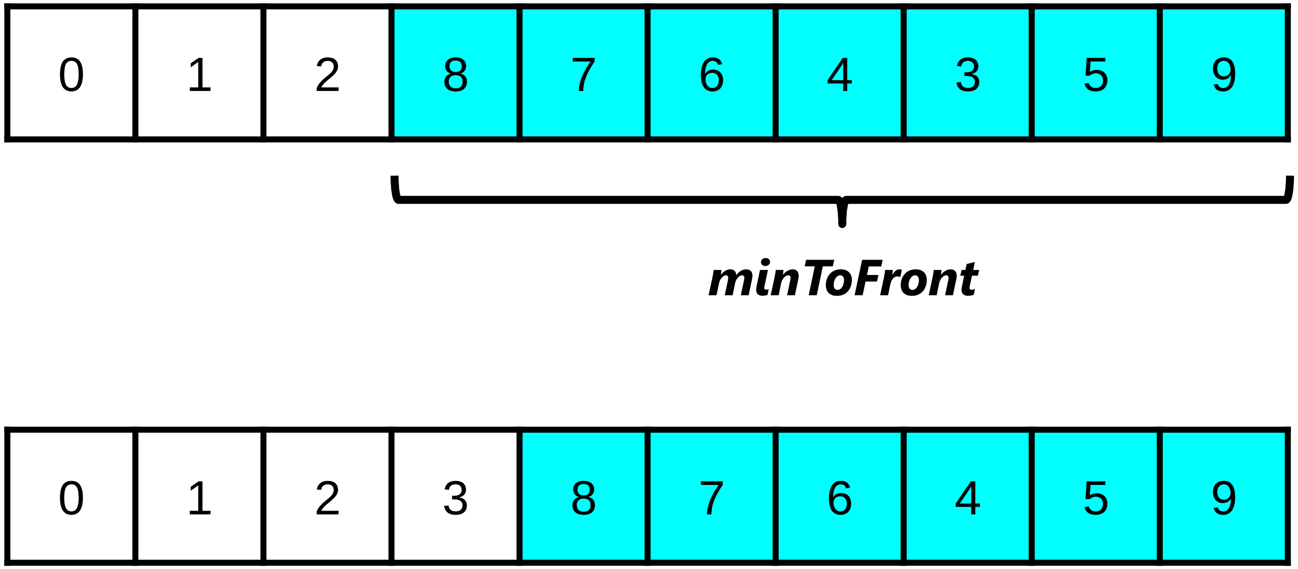
0	1	8	7	6	4	3	5	2	9

minToFront

0	1	2	8	7	6	4	3	5	9

Sorting the List

- observe what happens if you repeat the process with the sublist made up of the fourth through last elements:

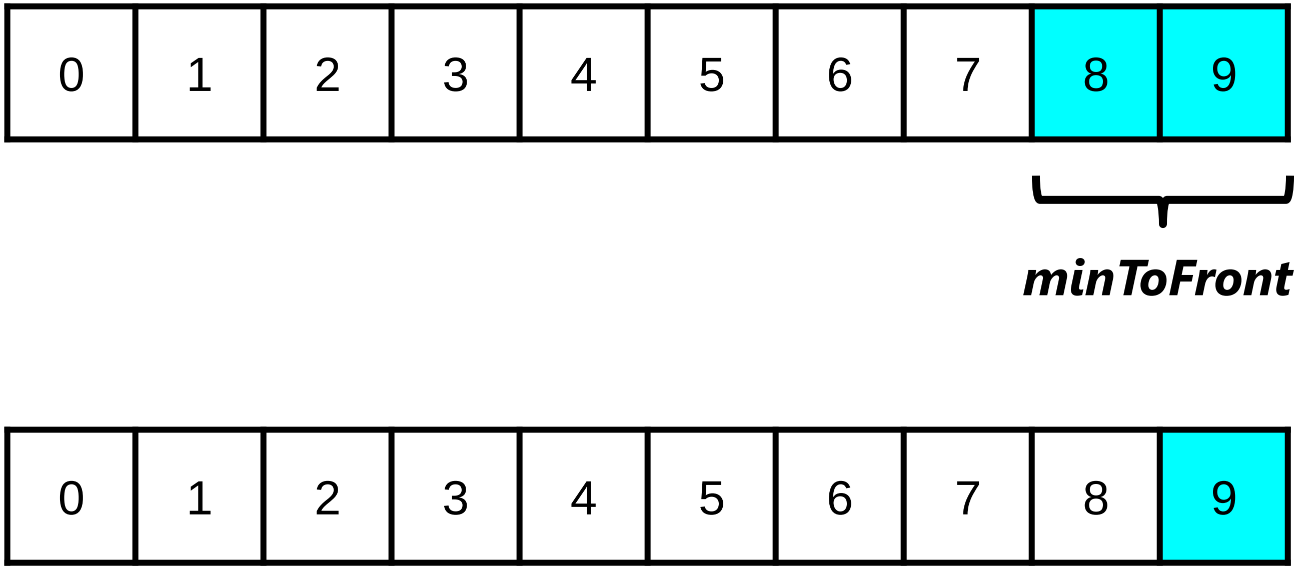


0	1	2	8	7	6	4	3	5	9

0	1	2	3	8	7	6	4	5	9

Sorting the List

- if you keep calling minToFront until you reach a sublist of size two, you will sort the original list:



- this is the *selection sort* algorithm

Selection Sort

```
public class Sort {

  // minToFront not shown

  public static void selectionSort(List<Integer> t) {
    if (t.size() > 1) {
      Sort.minToFront(t);
      Sort.selectionSort(t.subList(1, t.size()));
    }
  }


}
```

Selection Sort

- there are only two steps in the selection sort algorithm
  1. move the smallest element in the list to the front
     - this has complexity $O(n)$
  2. recursively selection sort the sublist of size $(n-1)$
- let $T(n)$ be the number of operations needed to selection sort a list of size n
  - then the recurrence relation is:

$$T(n) = T(n-1) + O(n)$$

  - solving the recurrence results in

$$T(n) = O(n^2)$$

Quicksort

- quicksort, like mergesort, is a divide and conquer algorithm for sorting a list or array
- it can be described recursively as follows:
  1. choose an element, called the *pivot*, from the list
  2. reorder the list so that:
     - values less than the pivot are located before the pivot
     - values greater than the pivot are located after the pivot
  3. quicksort the sublist of elements before the pivot
  4. quicksort the sublist of elements after the pivot

Quicksort

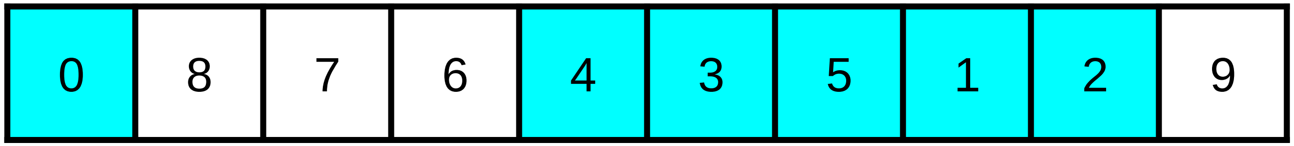
- step 2 is called the *partition* step
- consider the following list of unique elements

0	8	7	6	4	3	5	1	2	9

- assume that the pivot is 6

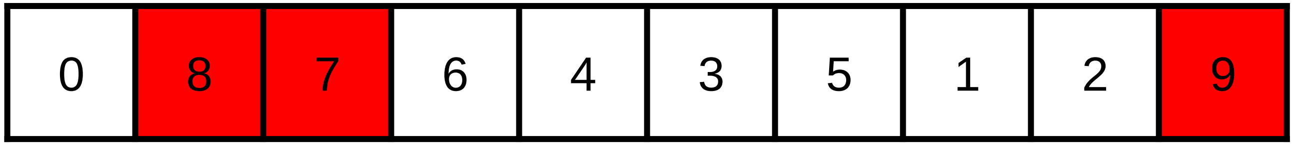
Quicksort

- the partition step reorders the list so that:
  - values less than the pivot are located before the pivot
    - we need to move the cyan elements before the pivot



0	8	7	6	4	3	5	1	2	9

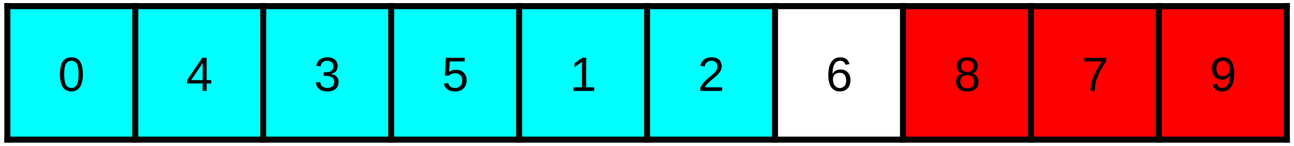
  - values greater than the pivot are located after the pivot
    - we need to move the red elements after the pivot



0	8	7	6	4	3	5	1	2	9

Quicksort

- after partitioning the list looks like:

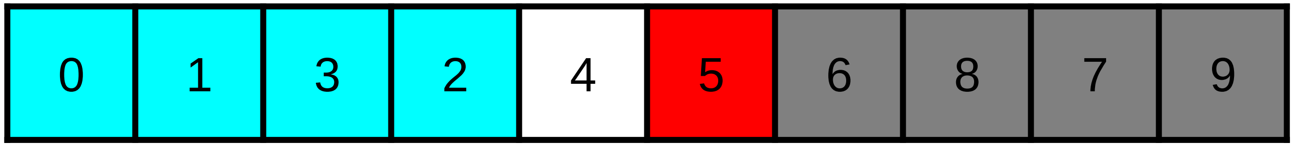


0	4	3	5	1	2	6	8	7	9

- partioning has 3 results:
  - the pivot is in its correct final sorted location
  - the left sublist contains only elements less than the pivot
  - the right sublist contains only elements greater than the pivot

Quicksort

- after partitioning we recursively quicksort the left sublist
- for the left sublist, let's assume that we choose 4 as the pivot
  - after partitioning the left sublist we get:

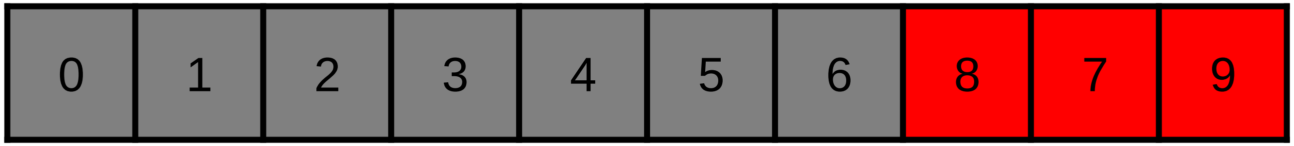


0	1	3	2	4	5	6	8	7	9

    - we then recursively quicksort the left and right sublists
      - and so on…

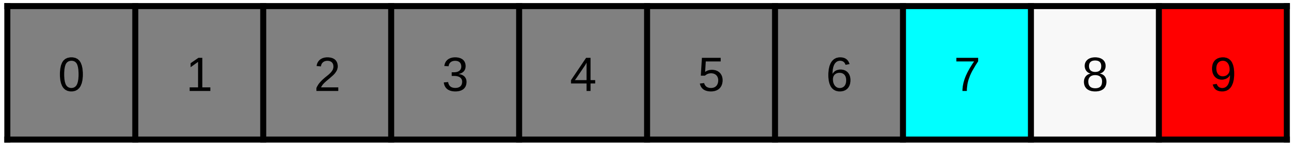
Quicksort

- eventually, the left sublist from the first pivoting operation will be sorted; we then recursively quicksort the right sublist:



0	1	2	3	4	5	6	8	7	9

- if we choose 8 as the pivot and partition we get:



0	1	2	3	4	5	6	7	8	9

- the left and right sublists have size 1 so there is nothing left to do

Quicksort

- the computational complexity of quicksort depends on:
  - the computational complexity of the partition operation
    - without proof I claim that this is $O(n)$ for a list of size n
  - how the pivot is chosen

Quicksort

- let's assume that when we choose a pivot we always choose the smallest (or largest) value in the sublist
  - yields a sublist of size $(n - 1)$ which we recursively quicksort
- let $T(n)$ be the number of operations needed to quicksort a list of size n when choosing a pivot as described above
  - then the recurrence relation is:

$$T(n) = T(n - 1) + O(n)$$

same as selection sort

  - solving the recurrence results in

$$T(n) = O(n^2)$$

Quicksort

- let's assume that when we choose a pivot we always choose the median value in the sublist
  - yields 2 sublists of size $\left(\frac{n}{2}\right)$ which we recursively quicksort
- let $T(n)$ be the number of operations needed to quicksort a list of size n when choosing a pivot as described above
  - then the recurrence relation is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

same as merge sort

  - solving the recurrence results in

$$T(n) = O(n \log_2 n)$$

Binary Search

- one reason that we care about sorting is that it is much faster to search a sorted list compared to sorting an unsorted list
- the classic algorithm for searching a sorted list is called *binary search*
- to search a list of size n for a value v:
  - look at the element e at index $\left(\frac{n}{2}\right)$
  - if $e > v$ recursively search the sublist to the left
  - if $e < v$ recursively search the sublist to the right
  - if $e == v$ then done

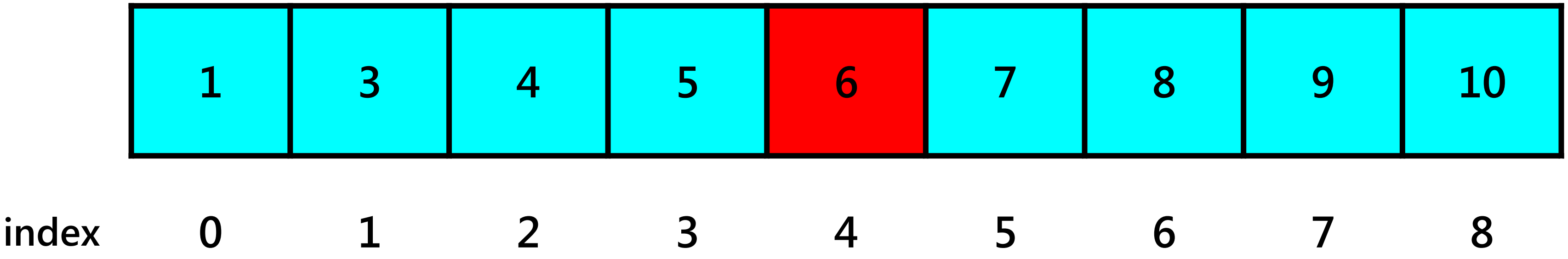
Binary Search

- consider the sorted list of size $n = 9$

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

Binary Search

- search for $v = 3$



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

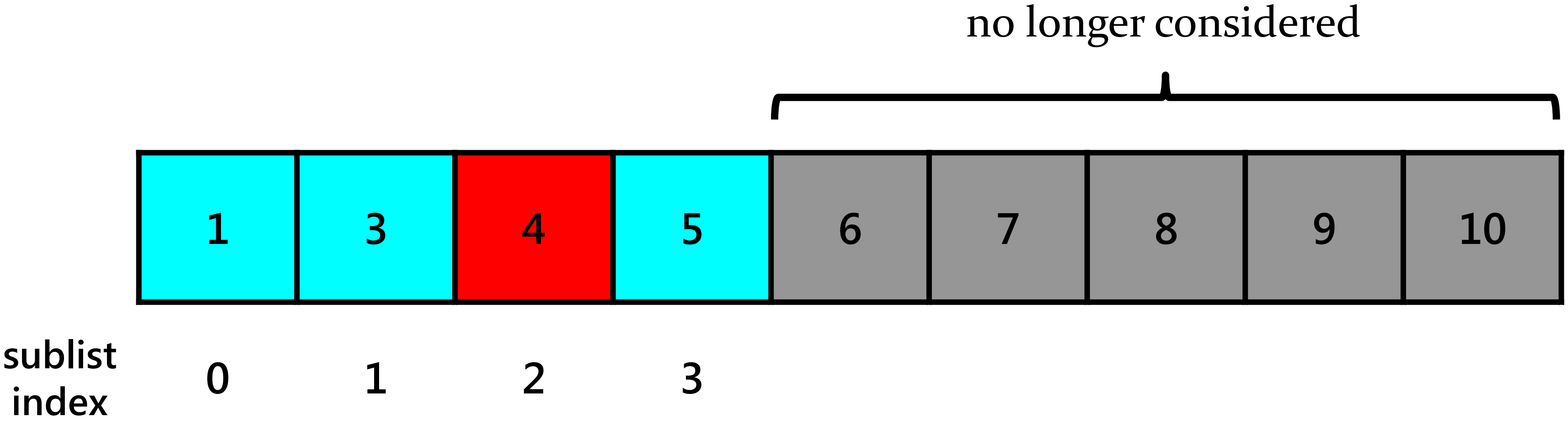
$$mid = \frac{9}{2} = 4$$

$$e = 6$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 3$



no longer considered

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

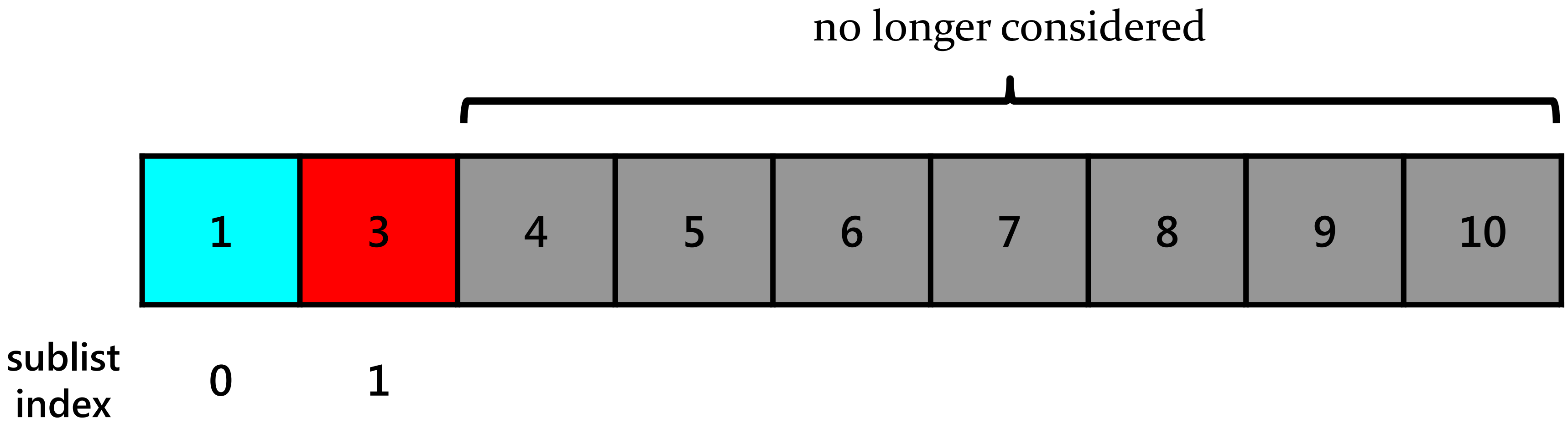
$$mid = \frac{4}{2} = 2$$

$$e = 4$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 3$



no longer considered

	1	3	4	5	6	7	8	9	10
sublist index	0	1							

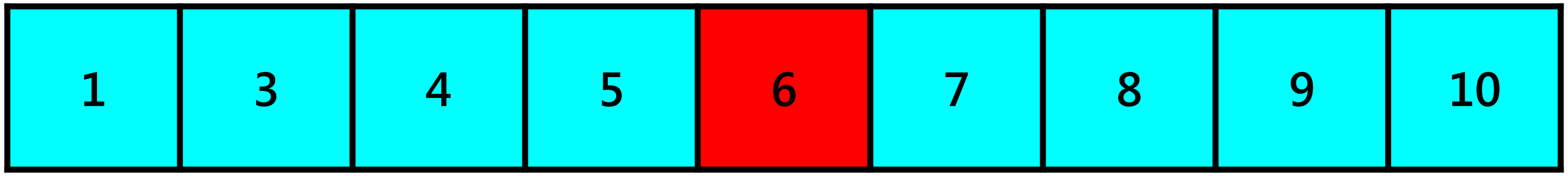
$$mid = \frac{2}{2} = 1$$

$$e = 3$$

$$v == e, \text{done}$$

Binary Search

- search for $v = 2$



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

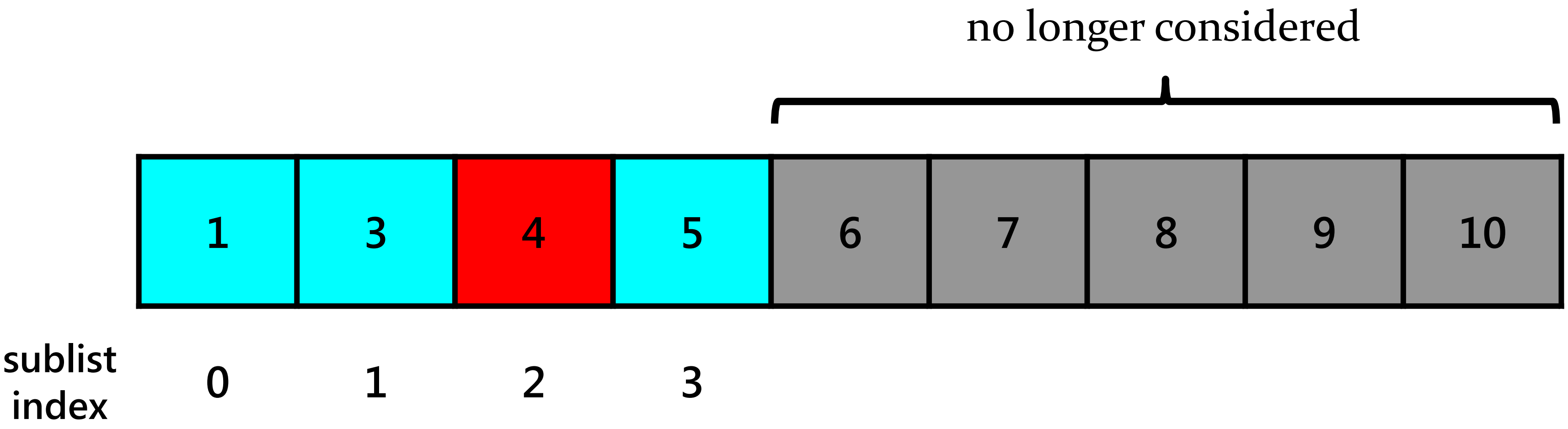
$$mid = \frac{9}{2} = 4$$

$$e = 6$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 2$



no longer considered

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

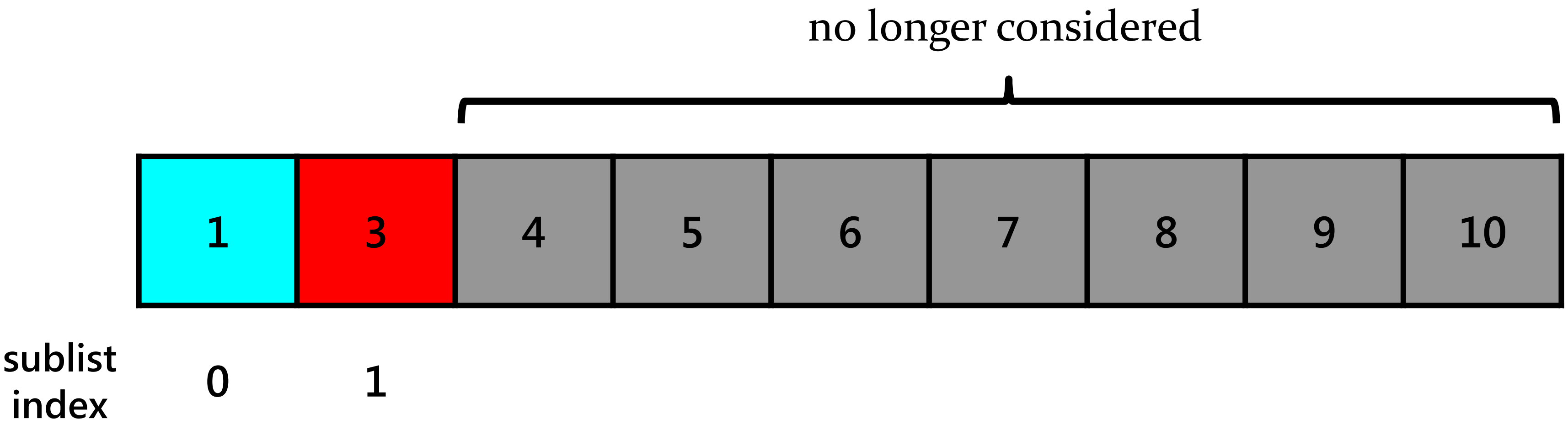
$$mid = \frac{4}{2} = 2$$

$$e = 4$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 2$



no longer considered

	1	3	4	5	6	7	8	9	10
sublist index	0	1							

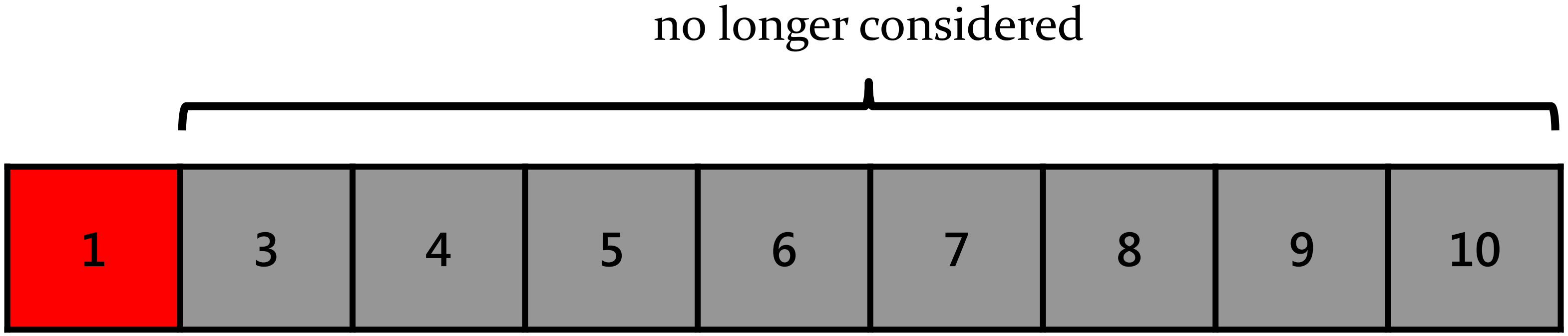
$$mid = \frac{2}{2} = 1$$

$$e = 3$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 2$



no longer considered

	1	3	4	5	6	7	8	9	10
sublist index	**0**								

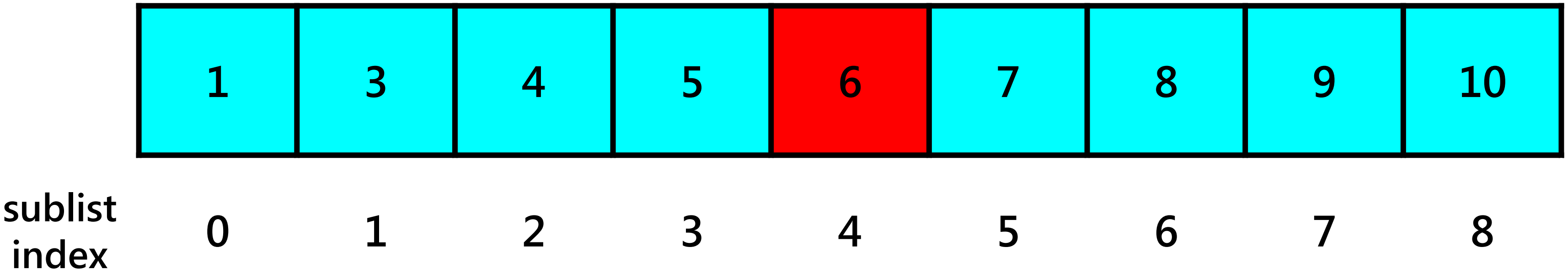
$$mid = \frac{1}{2} = 0$$

$$e = 1$$

$v > e$, recursively search the right sublist; right sublist is empty, done

Binary Search

- search for v =9



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

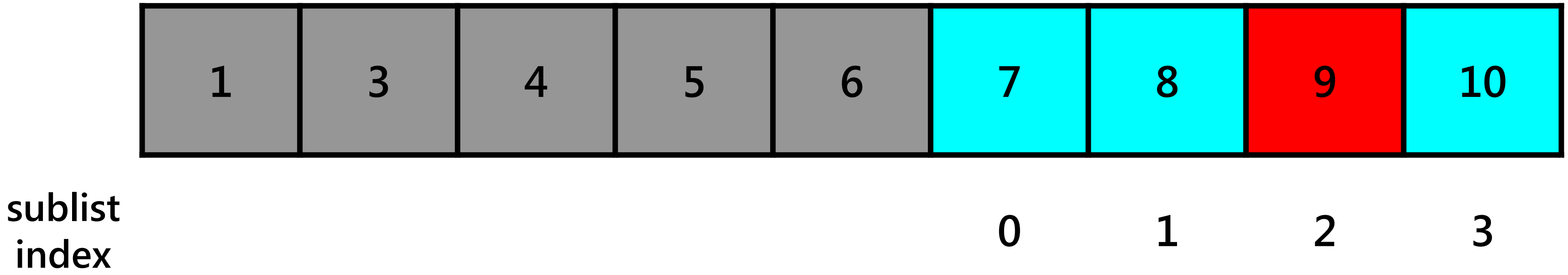
$$mid = \frac{9}{2} = 4$$

$$e = 6$$

$v > e$, recursively search the right sublist

Binary Search

- search for v =9



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

$$mid = \frac{4}{2} = 2$$

$$e = 9$$

$$v == e, \text{done}$$

```
/**
 * Searches a sorted list of integers for a given value using binary search.
 *
 * @param v the value to search for
 * @param t the list to search
 * @return true if v is in t, false otherwise
 */
public static boolean contains(int v, List<Integer> t) {
  if (t.isEmpty()) {
    return false;
  }
  int mid = t.size() / 2;
  int e = t.get(mid);
  if (e == v) {
    return true;
  }
  else if (v < e) {
    return Sort.contains(v, t.subList(0, mid));
  }
  else {
    return Sort.contains(v, t.subList(mid + 1, t.size()));
  }
}
```

Binary Search

- what is the recurrence relation?
- what is the big-O complexity?