Searching

Linear/sequential search, binary search, search analysis, java.util.Arrays

Searching for an element

Searching for an element in a collection can yield:

- the element itself
- the element's location (index)
- true or false to indicate whether the element belongs to the collection.

- If you are searching for an element in a collection with *n* elements, the worst that can happen is having to look at every element. Search algorithm complexity should be O(n) or less.
- Some search algorithms work on sorted collections only. They are usually more efficient than ones that work on both sorted and unsorted collection.

This lecture will cover two searches, one for sorted lists only and one for any list.

Simple Bicycle class

We'll look at lists with a simple Bicycle class. Bicycle, which implements Comparable<Bicycle>, compares two objects via their ids.

```
public class Bicycle implements Comparable<Bicycle> {
   private String model;
   private int id;
   private static int idGenerator = 1;

   public Bicycle(String model) {
      this.model = model;
      id = idGenerator++;
   }
   public int compareTo(Bicycle b) {
      return this.id - b.id;
   }
   // Other operations here
}
```

Linear search discussion

- A **linear search** visits all elements of a collection until it finds a match or until it has visited every element without finding a match.
- Assume a collection has *n* elements.
  - The best a linear search can do is find a match with the first element it visits.
  - The worst a linear search can do is to visit every element and still not find it, In that case, you need *n* visits.
  - If the element is in the collection, the expected number of elements to visit during the search is *n*/2.
- Nobody is ever interested in best case. The average case (sometimes) or worst case (most often) are all that matter.

Linear search code

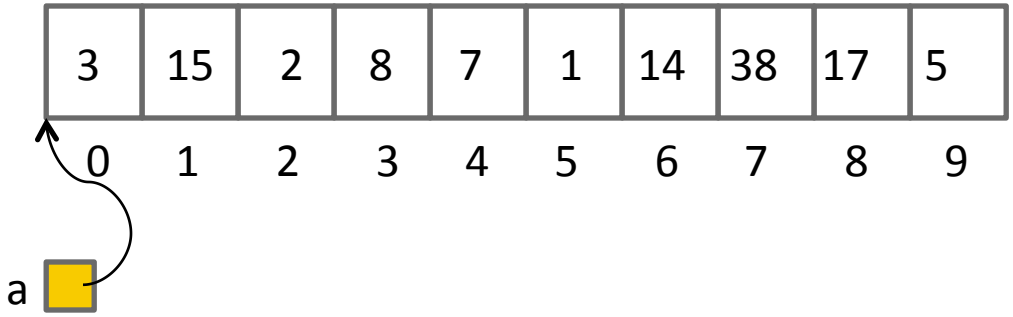
Search to find a Bicycle with a particular target id in an array of Bicycles. (The technique is identical with linked lists of Bicycles.)

ASSUME for an array: Each element of the array is an instance of Bicycle (no null values).

```
public Bicycle linearSearch(Bicycle[] a, int target) {
   for (int k = 0; k < a.length; k++) {
      if (a[k].getId() == target)
         return a[k];
   }
   return null;
}
```

Linear search analysis

- With a linear search, you start at the front and examine each element until you find the one you want.
- Assume the Bicycle ids are 3, 15, 2, 8, 7, 1, 14, 38, 17, 5 (shown below).
  Call: linearSearch(a, 17);

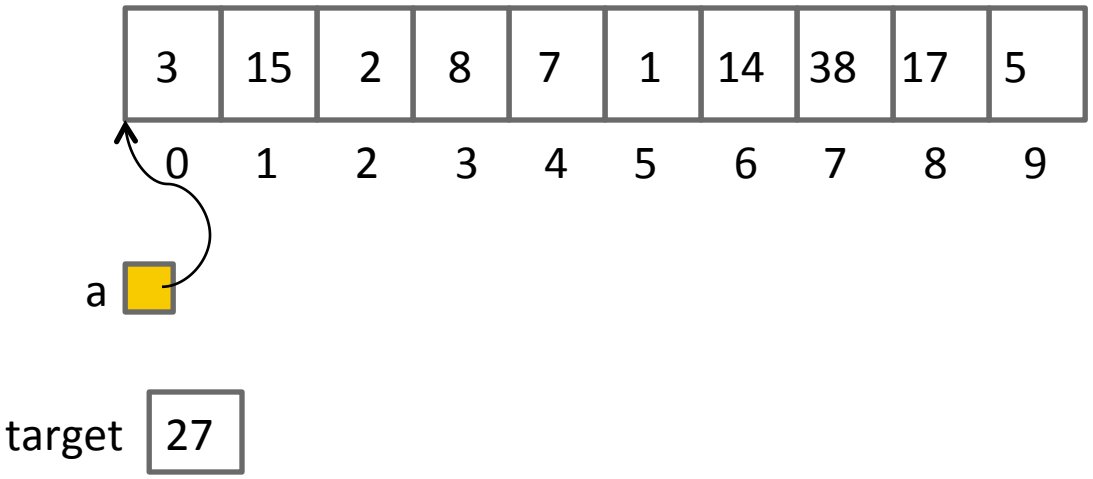


target 17

The search starts at index 0 and proceeds through index 8.

Linear search analysis (cont)

- Call: linearSearch(a, 27);
- The search starts at 0 and proceeds through every element.
- This is the worst case scenario, and it is how the efficiency of all searches are measured.



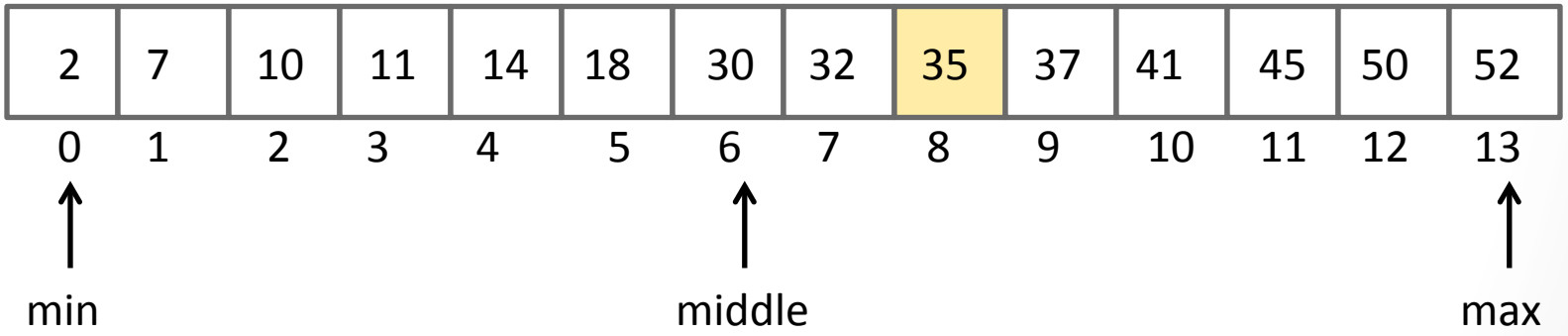
Binary search discussion

- Searches can be more efficient if you know something about the search space.
- A binary search on a collection has two restrictions:
  - The elements must be comparable (can compare one to another to determine which is larger).
  - The collection is sorted according to the order.
  - The collection is indexed (think array).
- Binary search starts with the entire collection as its search space.
- Every “visit” to find a match in a binary search eliminates half of the remaining elements in the search space.

Binary search trace

Search for a Bicycle with target id 35 in an array with ids shown below:

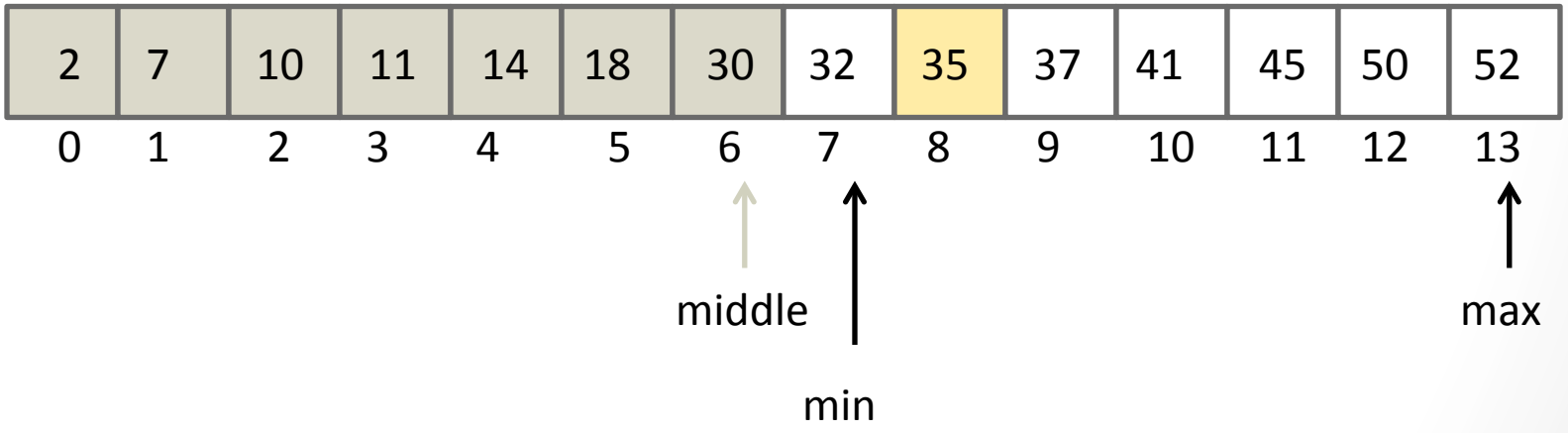
- 0 (min) is the smallest index of the search space.
- 13 (max) is the largest index of the search space.
- First guess is at 6, which is middle = (min + max) / 2



Binary search trace (cont)

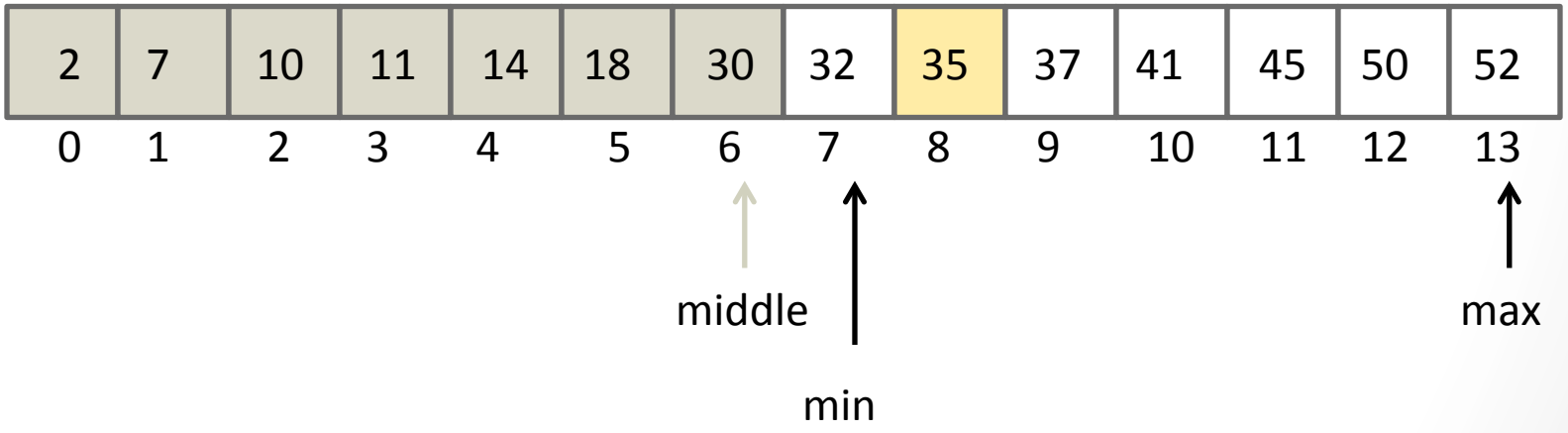
a[middle].getId() < 35, so:

- Assign min = middle + 1. This cuts the search space in half.
- Repeat



Binary search trace (cont 1)

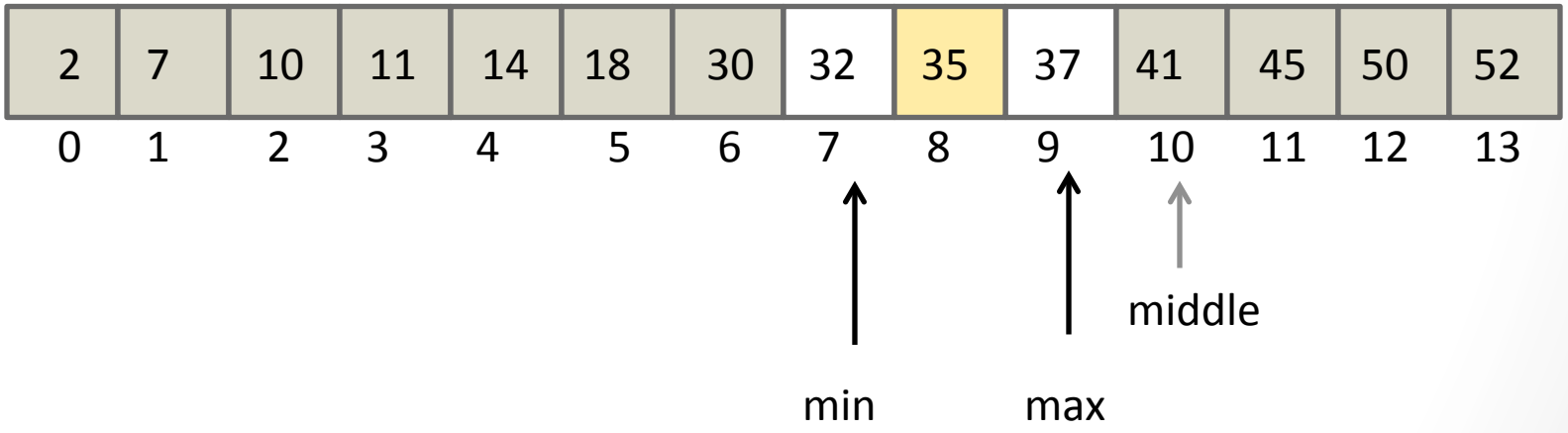
- Assign middle = (max + min)/2
- Compare a[middle].getId() to 35



Binary search trace (cont 2)

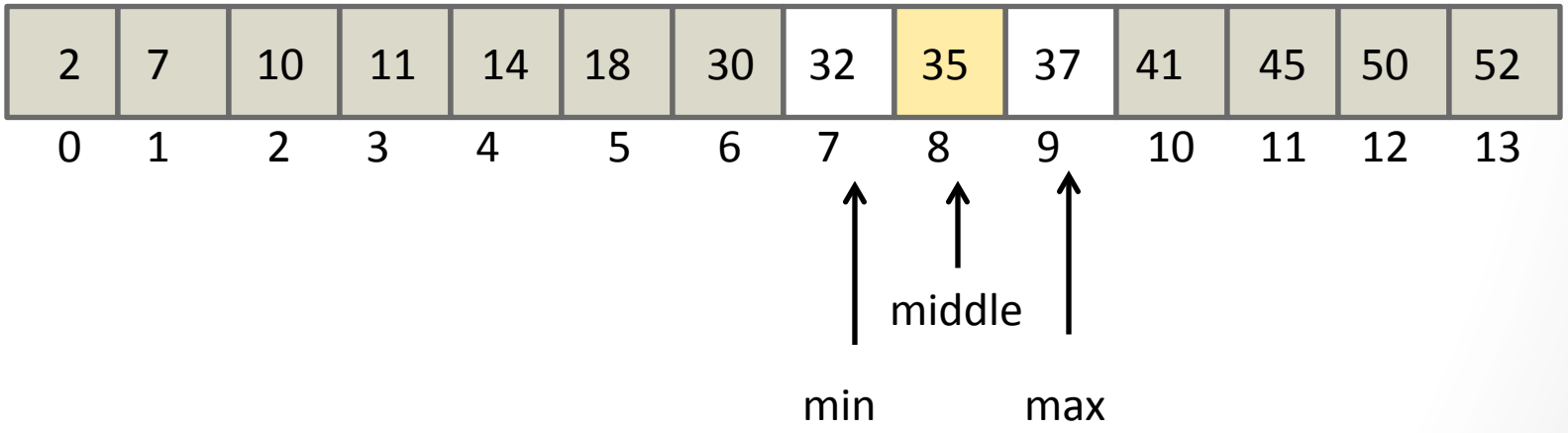
a[middle].getId() > 35, so:

- Assign max = middle - 1. This cuts the search space in half.
- Repeat



Binary search trace (cont 3)

- Assign middle = (max + min)/2
- Compare a[middle].getId() and 35
- Match! – the entire process took 3 guesses.



Binary search algorithm

Examine the middle element of the search space (initially every element).

- If the middle is too big, eliminate the right half of the search space and repeat.
- If the middle is too small, eliminate the left half of the search space and repeat.
- If the middle is a match, quit. You've found the element.

A binary search can be written recursively or iteratively.

Binary search (recursive code)

```
public static Bicycle binarySearch(Bicycle[] a,
                        int min, int max, int target) {
  if (min > max)
     return null;
  int middle = (min + max)/2;
  if (a[middle].getId() == target)
     return a[middle];
  if (a[middle].getId() < target)
     return binarySearch(a, middle + 1, max, target);
  else
     return binarySearch(a, min, middle - 1, target);
}
```

Binary search (iterative code)

```
public static Bicycle binarySearch1(Bicycle[] a,
                                    int target) {
   int min = 0;
   int max = a.length - 1;  // or size - 1
   while (min <= max) {
      int middle = (min + max) / 2;
      if (a[middle].getId() == target)
         return a[middle];
      if (a[middle].getId() < target)
         min = middle + 1;
      if (a[middle].getId() > target)
         max = middle - 1;
   }
   return null;
}
```

Binary search complexity

How many “guesses” are made? (a[middle].getId() == target)?

Size of array	Number of guesses
$1 = 2^0$	1
$2 = 2^1$	$2 = 1 + 1$
$4 = 2^2$	$3 = 2 + 1$
$8 = 2^3$	$4 = 3 + 1$
$16 = 2^4$	$5 = 4 + 1$
$32 = 2^5$	$6 = 5 + 1$
$64 = 2^6$	$7 = 6 + 1$

$x + 1$: number of guesses
n : size of array

$n = 2^x$
$x = \log_2 n$

Binary search is O(log n) or **logarithmic complexity**.

Linear search complexity

How many “guesses” are made? (a[k].getId() == target)?

Size of array	Number of guesses
1	1
2	2
4	4
8	8
16	16
32	32
64	64

x : number of guesses
n : size of array

x = n

Linear search is O(n) or **linear complexity**.

Using the Arrays class

java.util.Arrays has useful static array utilities, including searches. This table lists some:

Operation	Description
copyOf(array, length)	Returns a resized copy of the array.
equals(array1,array2)	Returns true if the two arrays contain the same elements in the same order
toString(array)	Returns a string representation of the array, such as “[12, 34, 56]”
sort(array)	Sorts the array elements according to the elements’ compareTo() method
binarySearch(array, value)	Returns the index of value in the array, or < 0 if not found. The array must be sorted.