Searching Algorithms

Lecture Objectives

- Learn how to implement the sequential search algorithm
- Learn how to implement the binary search algorithm
- To learn how to estimate and compare the performance of algorithms
- To learn how to measure the running time of a program

Searching Algorithms

- **Necessary components to search a list of fdata**
  - Array containing the list
  - Length of the list
  - Item for which you are searching

- **After search completed**
  - If item found, report "**success**," return location in array
  - If item not found, report "**not found**" or "**failure**"

Searching Algorithms (Cont'd)

- Suppose that you want to determine whether `27` is in the list
- First compare `27` with `list[0]`; that is, compare `27` with `35`
- Because `list[0] ≠ 27`, you then compare `27` with `list[1]`
- Because `list[1] ≠ 27`, you compare `27` with the next element in the list
- Because `list[2] = 27`, the search stops
- This search is successful!

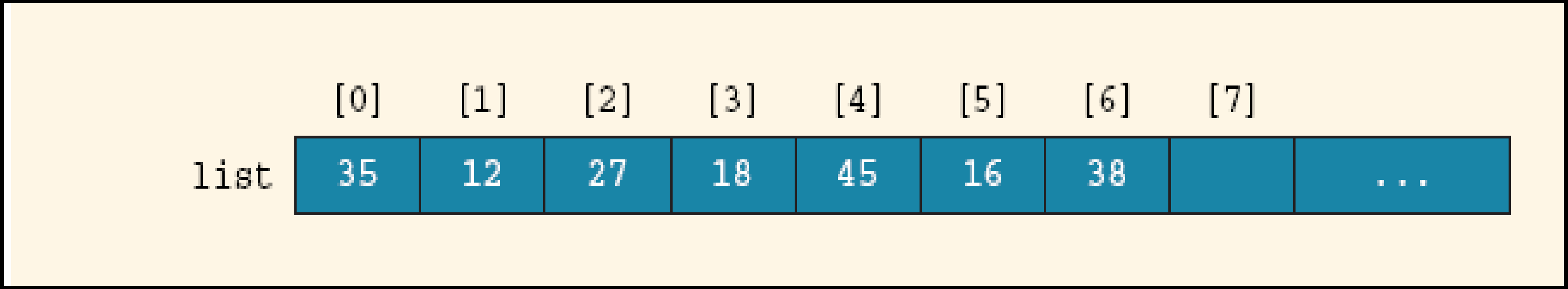

Figure 1: Array list with seven (07) elements

Searching Algorithms (Cont’d)

- Let’s now search for `10`
- The search starts at the first element in the list; that is, at `list[0]`
- Proceeding as before, we see that this time the search item, which is `10`, is compared with every item in the list
- Eventually, no more data is left in the list to compare with the search item; this is an unsuccessful search

Linear Search Algorithm

The previous could be further reduced to:

```
public static int linSearch(int[] list, int listLength, int key) {
   int loc;
   boolean found = false;

  for(int loc = 0; loc < listLength; loc++) {
     if(list[loc] == key) {
        found = true;
        break;
     }
  }
  if(found)
     return loc;
  else
     return -1;
}
```

Linear Search Algorithm (Cont'd)

```
public static int linSearch(int[] list, int listLength, int key) {
   int loc;

  for(int loc = 0; loc < listLength; loc++) {
     if(list[loc] == key)
          return loc;
  }
  return -1;
}
```

Linear Search Algorithm (Cont'd)

- Using a **`while`** (or a **`for`**) loop, the definition of the method `seqSearch` can also be written without the **`break`** statement as:

```
public static int linSearch(int[] list, int listLength, int key) {
   int loc = 0;
   boolean found = false;

  while(loc < listLength && !found) {
    if(list[loc] == key)
         found = true;
    else
         loc++
  }
  if(found)
    return loc;
  else
   return -1;
}
```

Performance of the Linear Search

- Suppose that the first element in the array *list* contains the variable *key*, then we have performed one comparison to find the *key*.

- Suppose that the second element in the array *list* contains the variable *key*, then we have performed two comparisons to find the key.

- Carry on the same analysis till the *key* is contained in the last element of the array *list*. In this case, we have performed N comparisons (N is the size of the array list) to find the *key*.

- Finally if the key is **NOT** in the array list, then we would have performed N comparisons and the key is NOT found and we would return -1.

Performance of the Linear Search (Cont'd)

- Therefore, the best case is: **1**
- And, the worst case is: **N**
- The average case is:

Worst case and *key* found at the end of the array *list*!

Best case

$$\text{Average Number of Comparisons} = \frac{1 + 2 + 3 + \ldots\ldots + N + N}{\underbrace{N+1}_{\text{Number of possible cases}}}$$

Worst case and *key* is NOT found!

Number of possible cases

Binary Search Algorithm

- Can only be performed on a sorted list !!!

- Uses *divide and conquer* technique to search list

Binary Search Algorithm (Cont’d)

- Search item is compared with middle element of list
- If search item < middle element of list, search is restricted to first half of the list
- If search item > middle element of list, search second half of the list
- If search item = middle element, search is complete

Binary Search Algorithm (Cont'd)

- Determine whether ***75*** is in the list

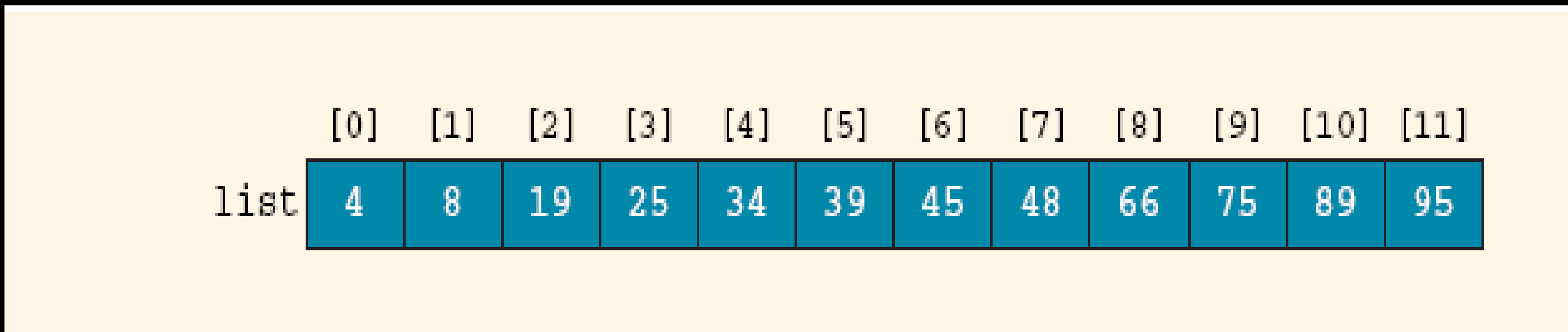

Figure 2: Array list with twelve (12) elements

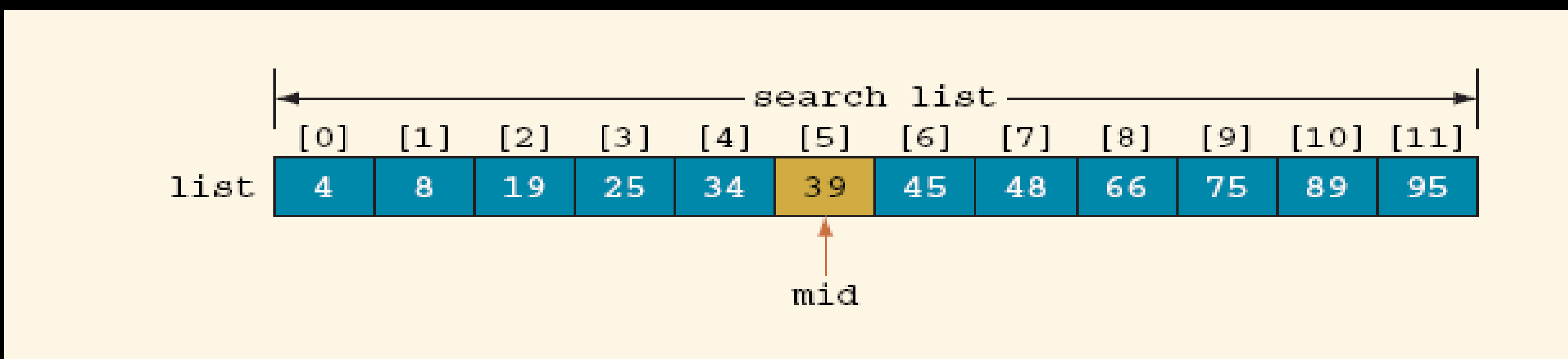

Figure 3: Search list, list[0] … list[11]

Binary Search Algorithm (Cont'd)

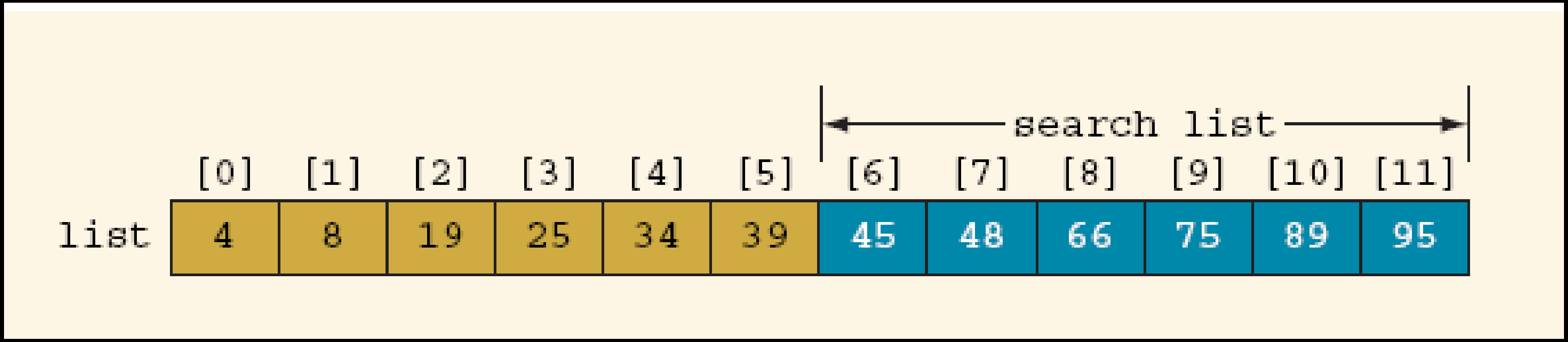

Figure 4: Search list, list[6] … list[11]

Binary Search Algorithm (Cont'd)

```
public static int binarySearch(int[] list, int listLength, int key) {
    int first = 0, last = listLength - 1;
    int mid;
    boolean found = false;

    while (first <= last && !found) {
        mid = (first + last) / 2;
        if (list[mid] == key)
            found = true;
        else
            if(list[mid] > key)
                last = mid - 1;
            else
                first = mid + 1;
    }
    if (found)
        return mid;
    else
        return –1;
} //end binarySearch
```

Binary Search Algorithm (Cont'd)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

Figure 5: Sorted list for binary search

key = 89

Iteration	first	last	mid	list[mid]	Number of key comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1 (found is true)

key = 34

Iteration	first	last	mid	list[mid]	Number of key comparisons
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	4	4	4	34	1 (found is true)

Binary Search Algorithm (Cont'd)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

Figure 6: Sorted list for binary search

key = 22

Iteration	first	last	mid	list[mid]	Number of key comparisons
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	3	2	the loop stops (because first > last) unsuccessful search		

Performance of Binary Search Algorithm

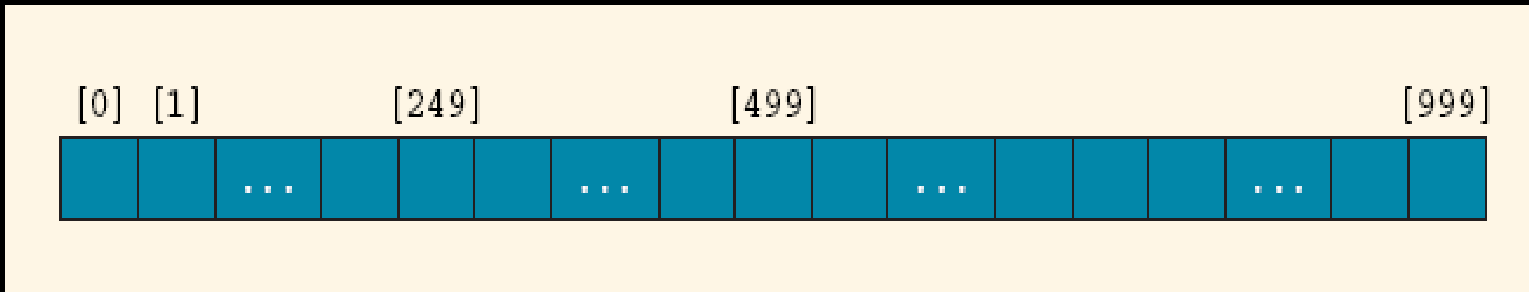

Figure 7: A Sorted list for binary search

key ≠ List[499]

key < List[499]

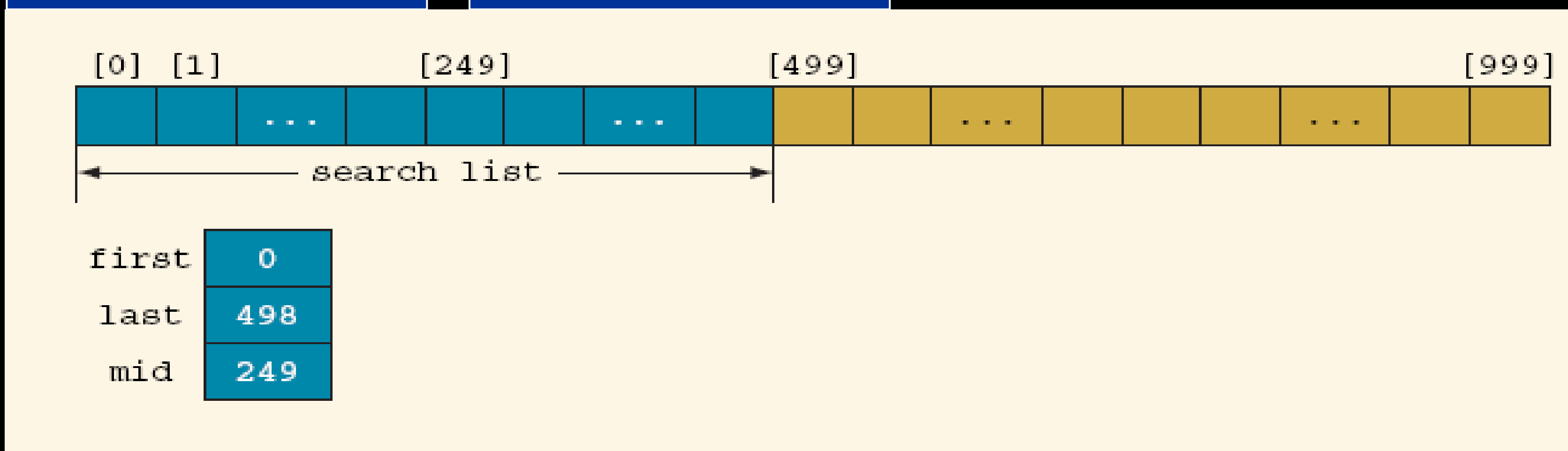

Figure 8: Search list after first iteration

Performance of Binary Search Algorithm (Cont'd)

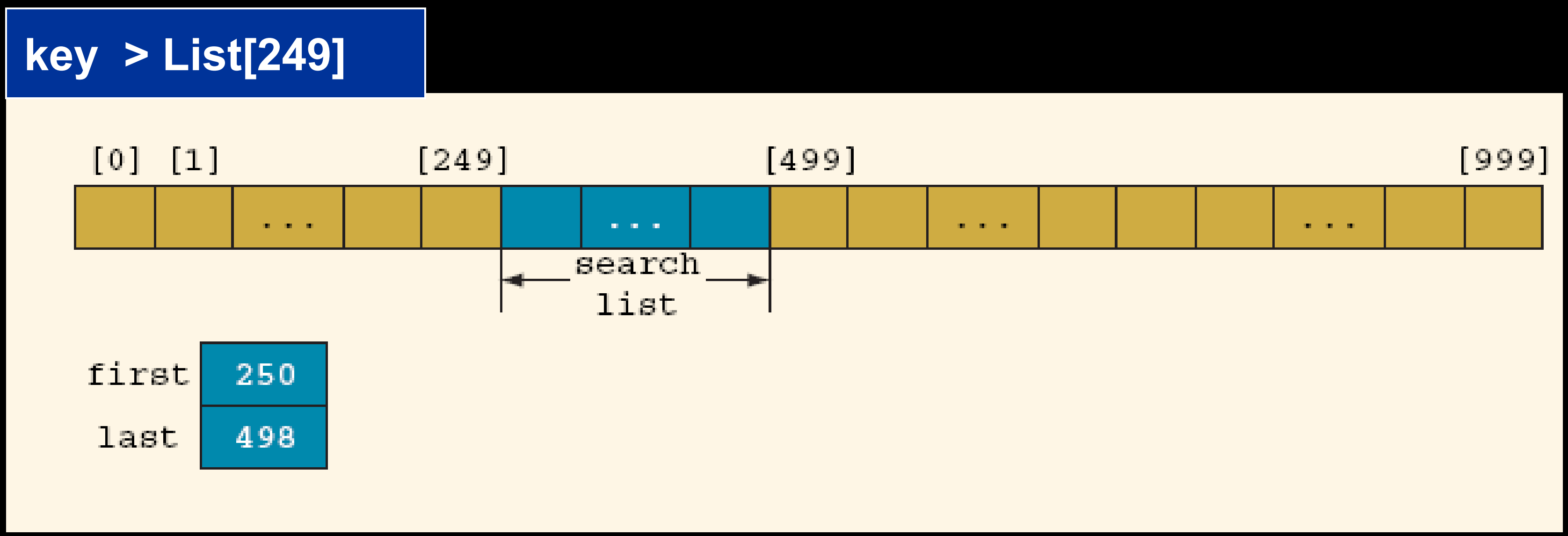

Figure 9: Search list after second iteration

Performance of Binary Search Algorithm (Cont'd)

- Suppose that ***L*** is a list of size 1000000
- Since 1000000 ≈ 1048576 = 220, it follows that the **`while`** loop in binary search will have at most 21 iterations to determine whether an element is in ***L***
- Every iteration of the **`while`** loop makes two key (that is, item) comparisons

Performance of Binary Search Algorithm (Cont'd)

- **To determine whether an element is in *L*, binary search makes at most *42* item comparisons**
  - On the other hand, on average, a linear search will make 500,000 key (item) comparisons to determine whether an element is in ***L***
- **In general, if *L* is a sorted list of size *N*, to determine whether an element is in *L*, the binary search makes at most $2\log_2 N + 2$ key (item) comparisons**

Searching a Sorted Array in a Program

- The `Arrays` class contains a static `binarySearch()` method
- The method returns either:
  - The index of the element, if element is found
  - Or **-k - 1** where **k** is the position before which the element should be inserted

```
int[] a = {1, 4, 9};
int v = 7;
int pos = Arrays.binarySearch(a, v);
   // Returns -3; v should be inserted before position 2
```

Searching Real Data

- `Arrays.binarySearch()` **sorts objects of classes that implement** `Comparable` **interface:**

```
public interface Comparable {
   int compareTo(Object otherObject);
}
```

- **The call** `a.compareTo(b)` **returns**
  - A negative number if ***a*** should come before ***b***
  - 0 if ***a*** and ***b*** are the same
  - A positive number otherwise

Searching Real Data (Cont'd)

- Several classes in Java (e.g. `String` and `Date`) implement `Comparable`
- You can implement `Comparable` interface for your own classes

```
public class Coin implements Comparable {
    . . .
    public int compareTo(Object otherObject) {
        Coin other = (Coin) otherObject;
        if (value < other.value) return -1;
        if (value == other.value) return 0;
        return 1;
    }
    . . .
}
```

The `CompareTo()` Method

- **The implementation must define a total ordering relationship**
  - Antisymmetric:
    If `a.compareTo(b) = 0`, then `b.compareTo(a) = 0`
  - Reflexive:
    `a.compareTo(a) = 0`

The `compareTo()` Method (Cont'd)

- **The implementation must define a total ordering relationship**
  - Transitive:
    If `a.compareTo(b) = 0` and `b.compareTo(c) = 0`,
    then `a.compareTo(c) = 0`

The compareTo() Method (Cont'd)

- Once your class implements `Comparable`, simply use the `Arrays.binarySearch()` method:

```
Coin[] coins = new Coin[n];
Coin aCoin = new Coin(…);
// Add coins
. . .
Arrays.binarySearch(coins, aCoin);
```