# Structural Programming and Data Structures

Winter 2000

**CMPUT 102: Searching**

Dr. Osmar R. Zaïane

University of Alberta

---

# Course Content

- Introduction
- Objects
- Methods
- Tracing Programs
- Object State
- Sharing resources
- Selection
- Repetition
- Vectors
- Testing/Debugging
- Arrays
- Searching
- Files I/O
- Sorting
- Inheritance
- Recursion

---

# Objectives of Lecture 21

## Searching

- Introduce two techniques for searching for an element in a collection;
- Learn sequential search algorithm;
- Learn the binary search algorithm for ordered collections.
- Learn how to evaluate the complexity of an algorithm and compare between algorithms.

---

# Outline of Lecture 21

- Review the simple array examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search
- Compare sequential search and binary search

# Array Example

```
// Find the largest element in an array of ints

int  markArray[] = {50, 37, 71, 99, 63};
int  index;
int  max;
index = 0;
max = markArray[index];
for (index = 1; index < markArray.length; index++)
        if (markArray[index] > max)
                max = markArray[index];
System.out.println(max);
```

markArray

| | |
|---|---|
| 50 | 0 |
| 37 | 1 |
| 71 | 2 |
| 99 | 3 |
| 63 | 4 |

index=5

max

99

---

# Array Example2

```
// Find the index of the largest element in an
array of ints
int  markArray[] = {50, 37, 71, 99, 63};
int  index;
int  indexOfMax;
index = 0;
indexOfMax = 0;
for (index = 1; index < markArray.length; index++)
  if (markArray[index] > markArray[indexOfMax])
     indexOfMax = index;
System.out.println(indexOfMax);
```

markArray

| | |
|---|---|
| 50 | 0 |
| 37 | 1 |
| 71 | 2 |
| 99 | 3 |
| 63 | 4 |

index = 5

indexOfMax

3

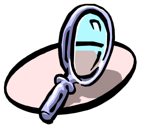
---

# Outline of Lecture 21

- Review the simple array examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search
- Compare sequential search and binary search

---

# The Search Problem



- Given a container, find the index of a particular element, called the key.
- Technique applies for vectors, arrays, files, etc.
- Applications: information retrieval, database querying, etc.

30

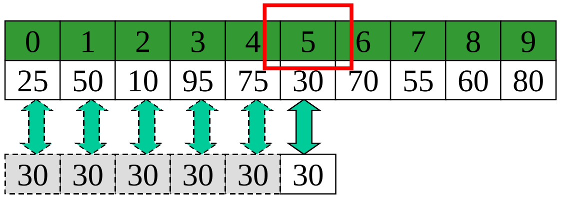
Element sought for

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 25 | 50 | 10 | 95 | 75 | 30 | 70 | 55 | 60 | 80 |

Collection

# Sequential Search

- Compare the key to each element in turn, until the correct element is found, and return its index.



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 25 | 50 | 10 | 95 | 75 | 30 | 70 | 55 | 60 | 80 |
| 30 | 30 | 30 | 30 | 30 | 30 | | | | |

---

# Sequential Search Code

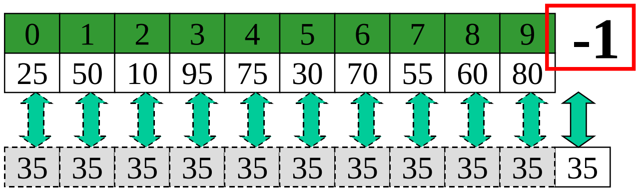
Compare all elements of the collection until we find the key.

```
/*   a sequential search code (first tentative)  */
  public static int sequential_search( int data[], int key ) {
    boolean found = false;
    int index = 0;

    while ( !found) {
      if ( key == data[index] )
        found = true;
      else
        index = index + 1;
    }
    return index;
  }
```

---

# Element not found

- We must take into account that the key we are searching for may not be in the array.
- In this case we must return a special index, say -1.



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | -1 |
|---|---|---|---|---|---|---|---|---|---|---|
| 25 | 50 | 10 | 95 | 75 | 30 | 70 | 55 | 60 | 80 | |
| 35 | 35 | 35 | 35 | 35 | 35 | 35 | 35 | 35 | 35 | 35 |

---

# Search Algorithm

**INPUT:** data: array of int; key: int;

**OUTPUT:** index : an int such that
data[index] == key if key is in data,
or -1 if key is not stored in data.

**Method**:

```
1. index = 0; found=false;
2. While ( not found  and index < data.length )
       check similarity data[index] and key
       index = index + 1
3. if  not found then  index = -1;
```

## Revised Sequential Search Code

```
/*   a sequential search method  */
 public static int sequential_search( int data[], int key ) {
   boolean found = false;
   int index = 0;

   while ( !found && index < data.length ) {
     if ( key == data[index] )
       found = true;
     else
       index = index + 1;
   }

   if  (!found)   index = -1;
   return index;
 }
```

---

## Outline of Lecture 21

- Review the simple array examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search
- Compare sequential search and binary search

---

## Complexity Analysis

- How efficient is this algorithm?
- In general if we have an algorithm that does something with ***n*** objects, we want to express the time efficiency of the algorithm as a function of ***n***.
- Such an expression is called the **time complexity** of the algorithm.
- In the case of search, we can count the number of comparison operations between the key and the elements.

---

## Worst, Best and Average cases

- In fact, we usually have multiple expressions:
  - the worst case complexity,
  - the best case complexity
  - the average case complexity.

# Complexity of Sequential Search

- How many comparison operations are required for a sequential search of an n-element container?
- In the worst case ➔ n.
- In the best case ➔ 1.
- In the average case:

$$\frac{1+2+3+...+n}{n}=\frac{n(n+1)}{2n}=\frac{(n+1)}{2}$$

- In this case, we say the complexity of Search is in the order of *n*, denoted as ***O(n)***.
- **Can we improve this algorithm?**

---

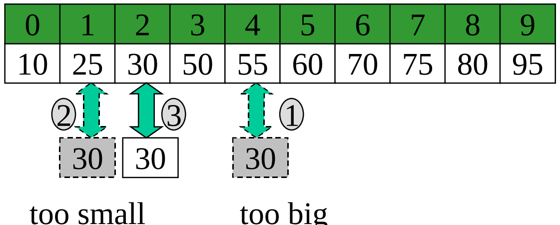
# Outline of Lecture 21

- Review the simple array examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search
- Compare sequential search and binary search

---

# Binary Search

- If the elements are <u>ordered</u>, we can do better.
- Guess the middle and adjust accordingly.

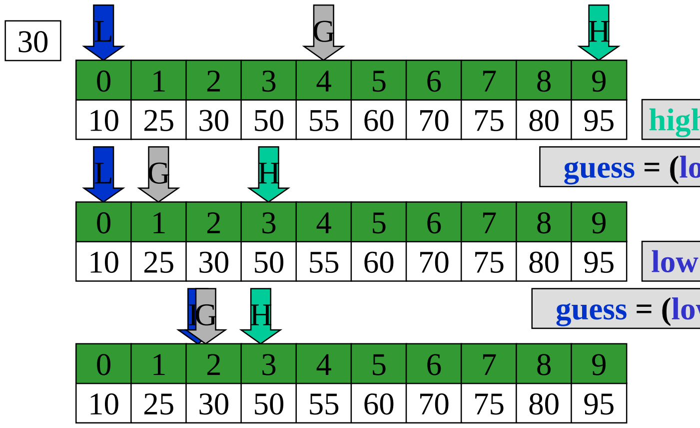


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 25 | 30 | 50 | 55 | 60 | 70 | 75 | 80 | 95 |

The order is important

---

# Binary Search Algorithm



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 25 | 30 | 50 | 55 | 60 | 70 | 75 | 80 | 95 |

**guess = (low + high) / 2**

**high = guess - 1**

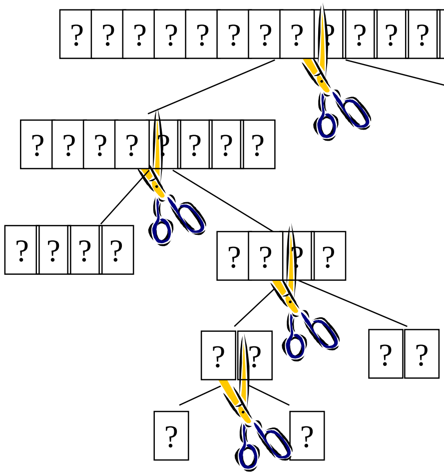
**guess = (low + high) / 2**

**low = guess + 1**

**guess = (low + high) / 2**

## Strategy of Binary Search:

Given an ordered array of integers, and a value of integer, search for the value in the array using an approach of **Divide and Conquer .**



Like searching in a dictionary, calendar or white pages.
Similar strategy as guess game in Lab exercise 6.

---

# Binary Search Code

Divide in 2 between lower and upper bounds until we find the key.

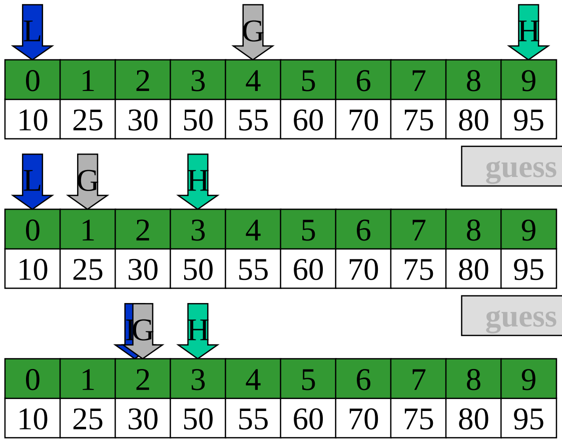
```
/*  a binary search code of ordered array (first tentative)  */
 public static int binary_search( int data[], int key ) {
   boolean found = false;
   int guess; int low = 0; int high=data.length-1;

   while ( !found) {
     guess = (high+low)/2;
     if ( key == data[guess] )   found = true;
     else if (key < data[guess])  high=guess-1;
     else low = guess+1;
   }
   return guess;
 }
```

---

# Element not found

**guess = (low + high) / 2**

35



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 25 | 30 | 50 | 55 | 60 | 70 | 75 | 80 | 95 |

**high = guess - 1**

**guess = (low + high) / 2**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 25 | 30 | 50 | 55 | 60 | 70 | 75 | 80 | 95 |

**low = guess + 1**

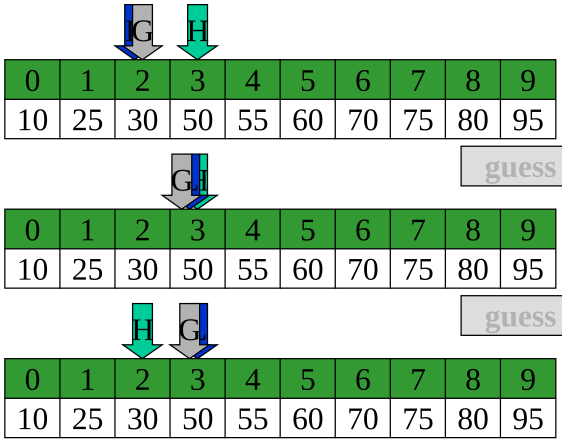
**guess = (low + high) / 2**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 25 | 30 | 50 | 55 | 60 | 70 | 75 | 80 | 95 |

---

# Element not found (con’t)

35



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 25 | 30 | 50 | 55 | 60 | 70 | 75 | 80 | 95 |

**low = guess + 1**

**guess = (low + high) / 2**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 25 | 30 | 50 | 55 | 60 | 70 | 75 | 80 | 95 |

**high = guess - 1**

**guess = (low + high) / 2**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 25 | 30 | 50 | 55 | 60 | 70 | 75 | 80 | 95 |

~~**low <= high**~~

# Binary Search Algorithm

```
INPUT:     data:   array of  ordered int;  key:  int;
OUTPUT:  index : an int such that
                  data[index] = = key if  key is in data,
                  or -1   if  key is not stored in data.
Method:
    1.  lower = 0;  upper = length;
    2.  While ( not found && low < =upper )
            index = (lower + upper) /2;
            check similarity data[index] and key
            if similar then found, otherwise
                if  key  < data[index]
                        upper =  index-1;
                else   lower = index +1;
    3.  If  ( data[index] != key )  index = -1;
```

---

```
/*   a binary search code of ordered array  */
 public static int binary_search( int data[], int key ) {
   boolean found = false;
   int guess; int low = 0; int high=data.length-1;

   while ( !found && low <= high) {
     guess = (high+low)/2;
     if ( key == data[guess] )   found = true;
     else if (key < data[guess])  high=guess-1;
     else low = guess+1;
   }
   if (! found) guess = -1;
   return guess;
 }
```

## Revised Binary Search Code

---

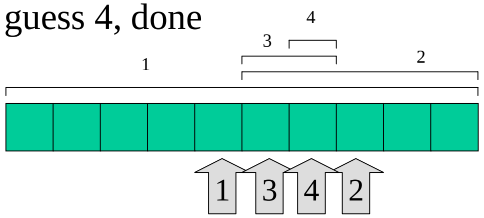
# Outline of Lecture 21

- Review the simple array examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search
- Compare sequential search and binary search

---

# Worst-case Binary Search

- Each time we guess, we divide the list in half:
- In the worst case:
  - – 10 elements, make guess 1, then
  - – 5 elements, make guess 2, then
  - – 2 elements, make guess 3, then
  - – 1 element, make guess 4, done



# Worst-case Binary Search (con't)

- If there were 15 elements:
    - 15 elements, make guess 1, then
    - 7 elements, make guess 2, then
    - 3 elements, make guess 3, then
    - 1 elements, make guess 4, done
- These results are the same, but if we have from 16 to 31 elements it takes 5 guesses.
- This formula is: $\lfloor \log_2(n)+1 \rfloor$
- $\log_2(n)$ is number of times you have to divide $n$ by 2 to get 1

---

# Average-case Binary Search

- If there were 15 elements:
    - 1 element takes 1 guess
    - 2 elements take 2 guesses
    - 4 elements take 3 guesses
    - 8 elements take 4 guesses
- The average is:

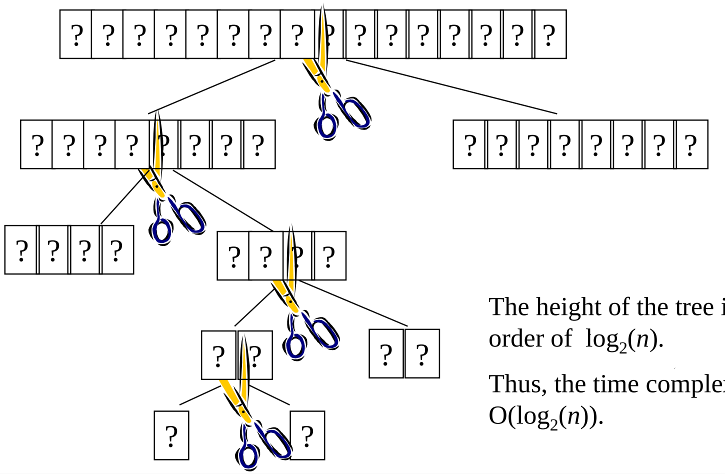
$$\frac{(1*1)+(2*2)+(4*3)+(8*4)}{15}=\frac{49}{15}\approx 3$$

- The average case is about one less than the worst case, so this is: $\lfloor \log_2(n) \rfloor$

---

## Time Complexity of Binary Search

The number of comparisons is proportional to the height of the following search tree:



The height of the tree is in the order of $\log_2(n)$.

Thus, the time complexity is $O(\log_2(n))$.

---

# Outline of Lecture 21

- Review the simple array examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search
- Compare sequential search and binary search

# Sequential and Binary Search

- For average and worst case sequential search, it takes: $\frac{(n+1)}{2}$ and $n$.
- For average and worst case binary search, it takes: $\lfloor \log_2(n) \rfloor$ and $\lfloor \log_2(n)+1 \rfloor$.

| list size | Sequential average | Sequential worst | Binary average | Binary worst | Ratio |
|---|---|---|---|---|---|
| 10 | 6 | 10 | 3 | 4 | 2 |
| 100 | 51 | 100 | 6 | 7 | 8 |
| 1000 | 501 | 1000 | 9 | 10 | 55 |
| 10000 | 5001 | 10000 | 13 | 14 | 384 |