



CMPUT 175

Introduction to Foundations of Computing

Searching

Objectives

- 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

- Review the simple list 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 a list of numbers

```
markList = [50, 37, 71, 99, 63]
```

```
max = markList[0]
```

```
for index in range(1,len(markList)):
    if (markList[index] > max):
        max = markList[index]
```

```
print("highest mark=",max)
```

markList

50	0
37	1
71	2
99	3
63	4

index=4

max

99

Array Example2

Find the index of the largest element in a list of numbers

```
markList = [50, 37, 71, 99, 63]
```

```
indexOfMax = 0
```

```
for index in range(1,len(markList)):
```

```
    if (markList[index] > markList[indexOfMax]):
```

```
        indexOfMax = index
```

```
print("index of highest mark=",indexOfMax)
```

markList

50	0
37	1
71	2
99	3
63	4

index = 4

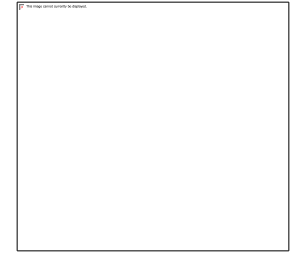
indexOfMax

3

Outline of Lecture

- Review the simple list 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 lists, arrays, files, etc.
- Applications: information retrieval, database querying, etc.

30

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

Element
sought for

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
----	----	----	----	----	----

The diagram illustrates the sequential search process. A table of 10 elements is shown with indices 0 to 9. The element 30 at index 5 is highlighted with a red box. Below the table, a row of six boxes, each containing the number 30, represents the key being searched for. Green double-headed arrows connect the first five elements of the table (25, 50, 10, 95, 75) to the first five boxes of the key row, indicating comparisons. A red box highlights the element 30 at index 5, and a green double-headed arrow connects it to the sixth box of the key row, indicating a successful match.



Sequential Search Code

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

```
# a sequential search code (first tentative)
```

```
def sequential_search(data, key):
```

```
    found = False
```

```
    index = 0
```

```
    while ( not 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 list.
- 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: list 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;

```
# a sequential search method
def sequential_search(data, key) :
    found = False
    index = 0

    while ( not found and index < len(data) ):
        if ( key == data[index] ):
            found = True
        else:
            index = index + 1

    if ( not found):
        index = -1
    return index
```

Revised Sequential Search Code

Outline of Lecture

- Review the simple list 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

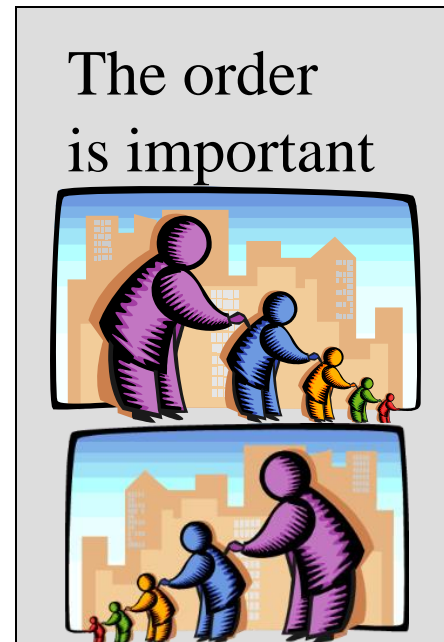
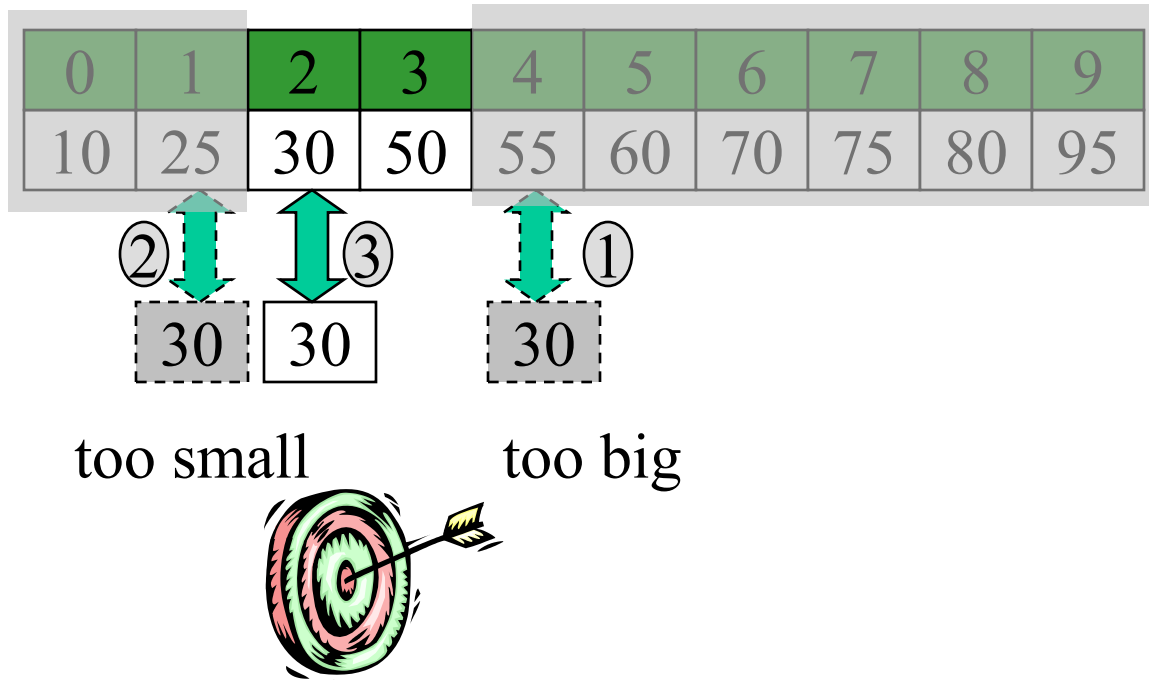
- How many comparison operations are required for a sequential search of an n -element container?
- In the worst case $\rightarrow n$. Key not in container
- In the best case $\rightarrow 1$. Key is in first position in container
- In the average case: $\frac{1+2+3+\dots+n}{n} = \frac{n(n+1)}{2n} = \frac{(n+1)}{2}$
- 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

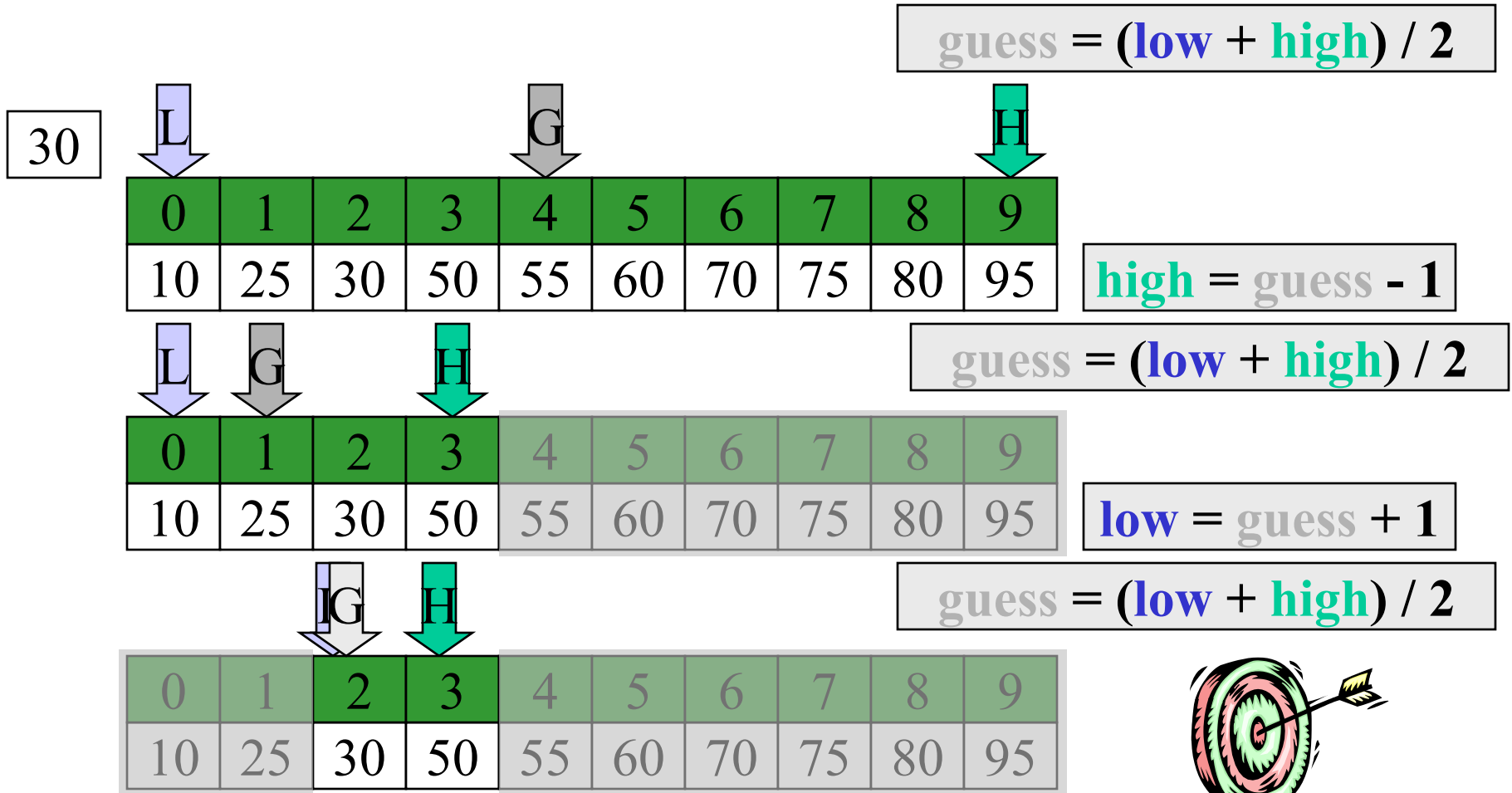
- Review the simple list examples
- Sequential search approach
- Complexity of sequential search
- Binary search approach
- Complexity of binary search
- Compare sequential search and binary search

Binary Search

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

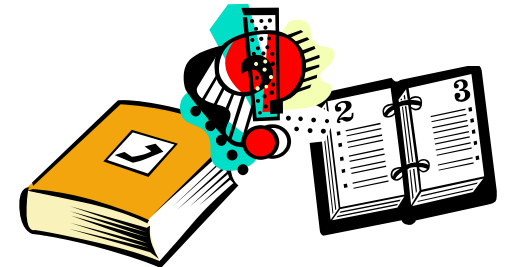
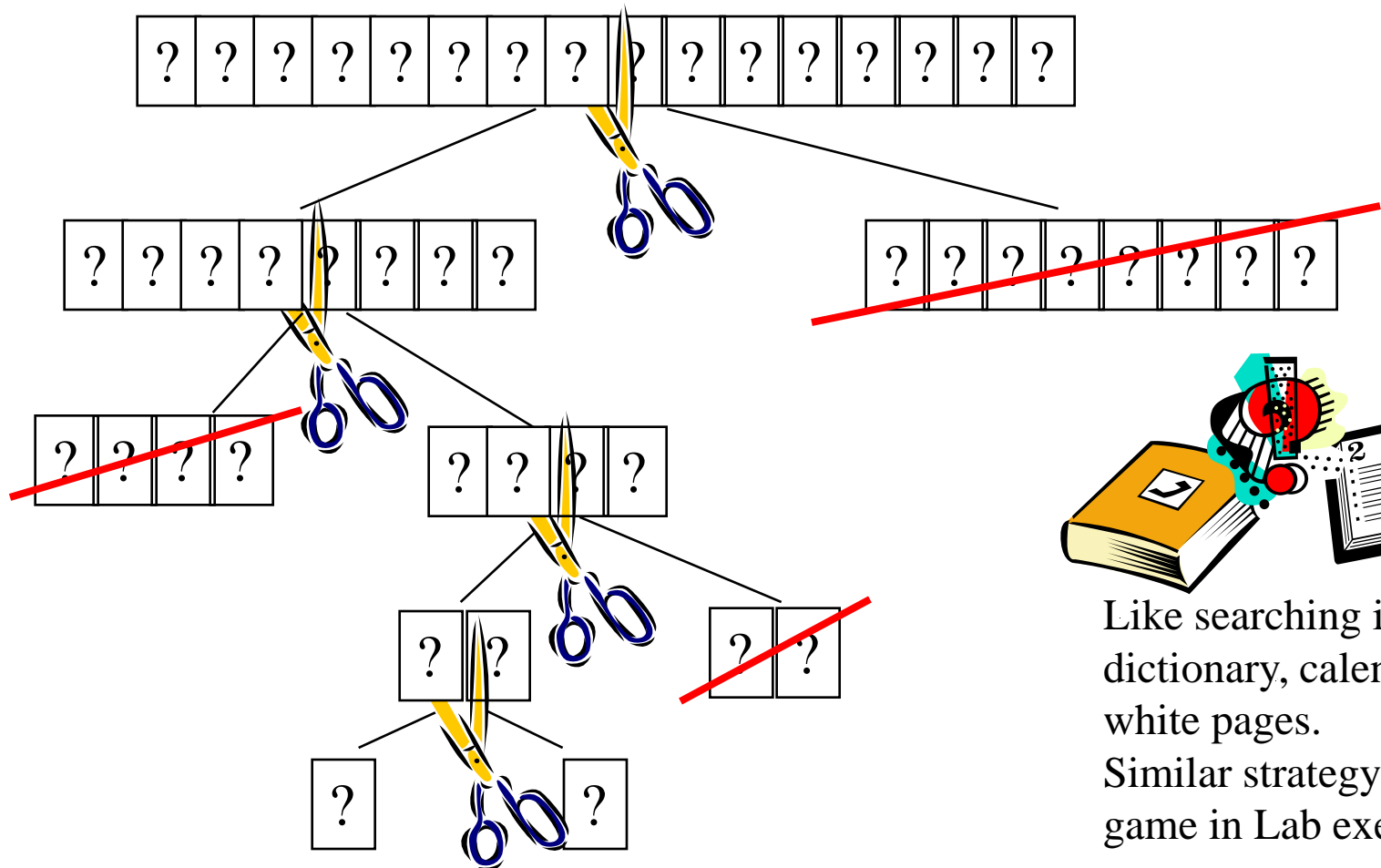


Binary Search Algorithm



Strategy of Binary Search:

Given an ordered list 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 8.

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)
```

```
def binary_search( data, key ) :
```

```
    found = False
```

```
    low = 0
```

```
    high=len(data)-1
```

```
    while ( not 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

35

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

$$\text{guess} = (\text{low} + \text{high}) / 2$$

$$\text{high} = \text{guess} - 1$$

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

$$\text{guess} = (\text{low} + \text{high}) / 2$$

$$\text{low} = \text{guess} + 1$$

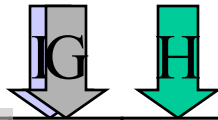
		L	G	H					
0	1	2	3	4	5	6	7	8	9
10	25	30	50	55	60	70	75	80	95

$$\text{guess} = (\text{low} + \text{high}) / 2$$

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



$$\text{low} = \text{guess} + 1$$

$$\text{guess} = (\text{low} + \text{high}) / 2$$

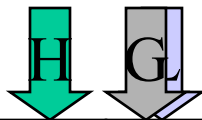
0	1	2	3	4	5	6	7	8	9
10	25	30	50	55	60	70	75	80	95



$$\text{high} = \text{guess} - 1$$

$$\text{guess} = (\text{low} + \text{high}) / 2$$

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



~~$$\text{low} < \text{high}$$~~

Binary Search Algorithm

INPUT: data: list of ordered int; key: int;

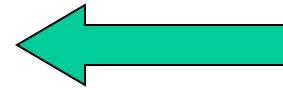
OUTPUT: index : an int such that
data[index] == key if key is in data,
or -1 if key is not stored in data.

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

```
def binary_search( data, key ) :  
    found = False  
    low = 0  
    high=len(data)-1  
    while ( not found and 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 (not found):  
        guess=-1  
    return guess
```



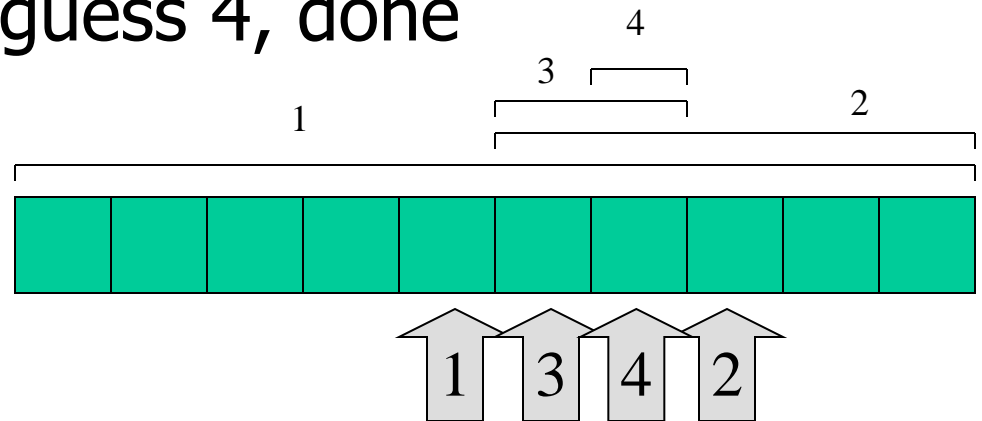
**Revised Binary
Search Code**

Outline of Lecture

- Review the simple list 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



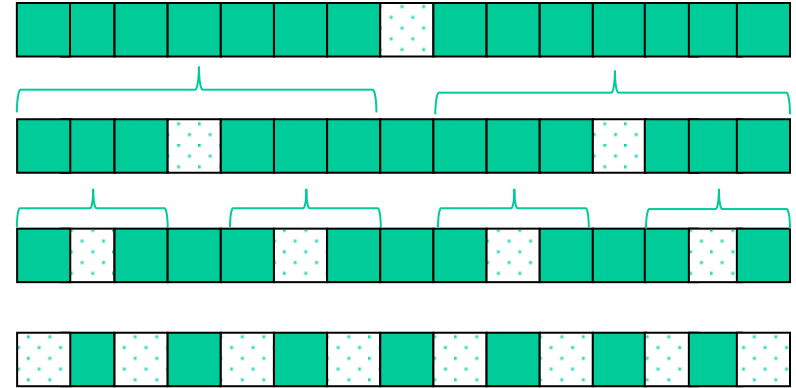
Worst-case Binary Search (con't)

- 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 the 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



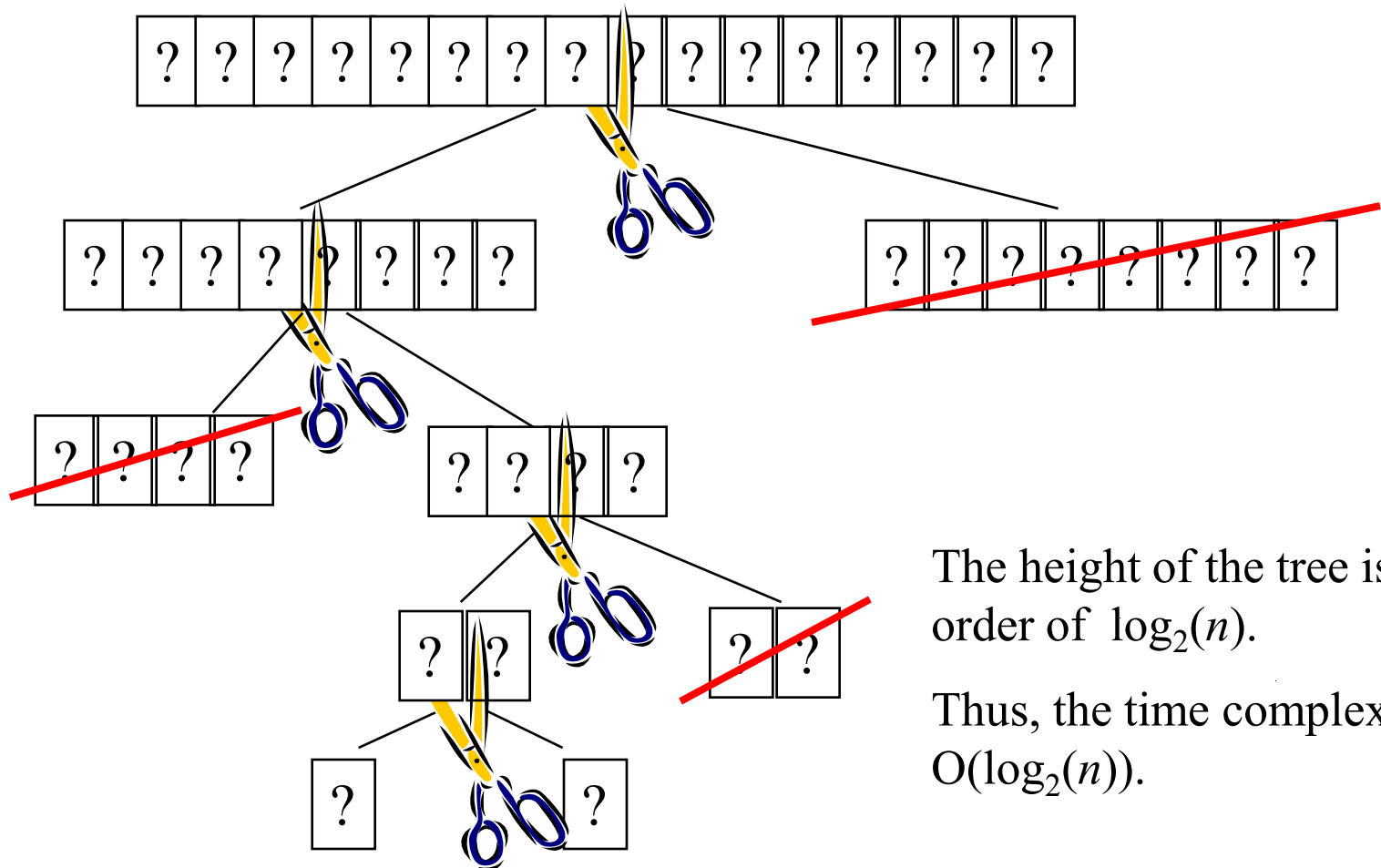
- The average is:

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

- 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

- Review the simple list 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: $\lceil \log_2(n) \rceil$ and $\lceil \log_2(n) + 1 \rceil$

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