BACS2063 Data Structures and Algorithms

Algorithms for Searching

Chapter 8a

Chapter 8 Part 1

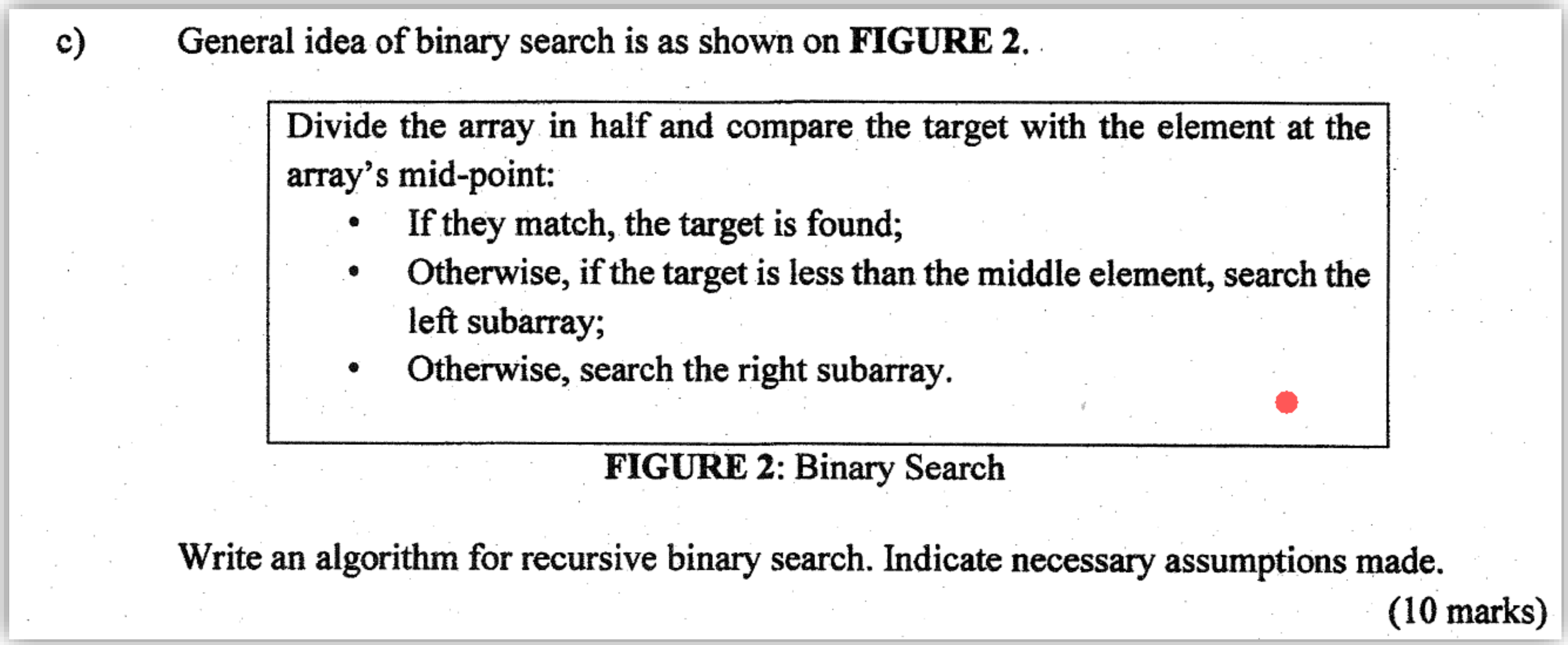
Algorithms for Searching

Learning Outcomes

At the end of this part, you should be able to

- Implement **sequential search** and **binary search** algorithms
- Assess the time efficiency of sequential search and binary search algorithms

PYQ Oct 2022 Q3c



c) General idea of binary search is as shown on **FIGURE 2**.

> Divide the array in half and compare the target with the element at the array's mid-point:
> - If they match, the target is found;
> - Otherwise, if the target is less than the middle element, search the left subarray;
> - Otherwise, search the right subarray.

FIGURE 2: Binary Search

Write an algorithm for recursive binary search. Indicate necessary assumptions made.

(10 marks)

The Problem

Fig. 16-1 Searching is an every day occurrence.

Searching: Introduction

- Searching is a common task in computing
- Given a collection of records or entries, searching is the process of locating a record (entry) whose key is equal to the target key value.
- A search algorithm accepts an argument **k** and tries to find a record whose key is **k**. A successful search is called a **retrieval**.
- Searching is time consuming, therefore different techniques are deployed to different data arrangement to achieve better performance.

Searching Techniques

1. **Sequential (linear) search** - for unsorted & sorted arrays, as well as unsorted & sorted linked lists.
2. **Binary search** - for sorted arrays only.

Sequential Search

- The sequential search algorithm searches through a list sequentially from the beginning, searching for a target value.
- a.k.a. *linear search*, *serial search*
- The method **`contains`** in **`ArrayList.java`** (from Chapter 4) and **`LinkedList.java`** (from Chapter 5) implements the sequential search algorithm.

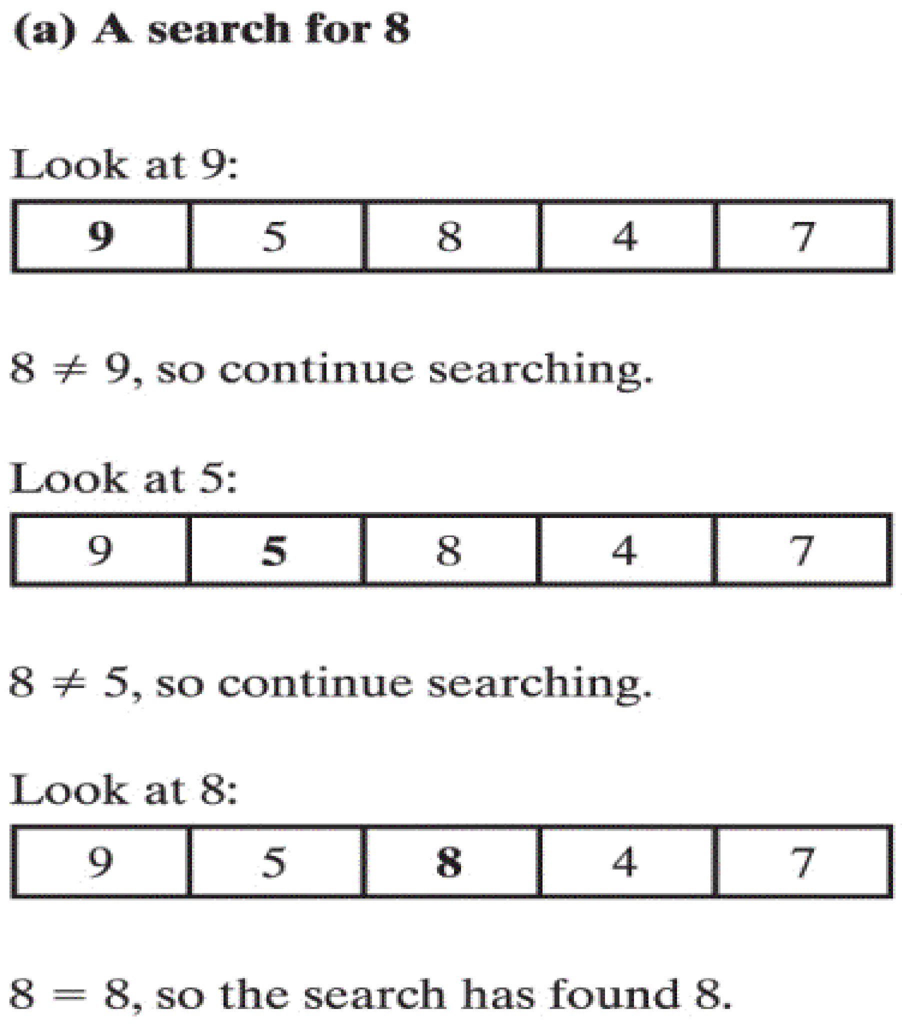
Iterative Sequential Search

Sequential Search - *Unsorted* Array (1)

- *Iterative* sequential search algorithm

```
public boolean contains(T anEntry) {
  boolean found = false;
  for (int index = 0;
           !found && (index < length); index++){
    if (anEntry.equals(array[index]))
      found = true ;
  }
  return found;
}
```

Sequential Search - *Unsorted* Array (2)



(a) A search for 8

Look at 9:

9	5	8	4	7

$8 \neq 9$, so continue searching.

Look at 5:

9	**5**	8	4	7

$8 \neq 5$, so continue searching.

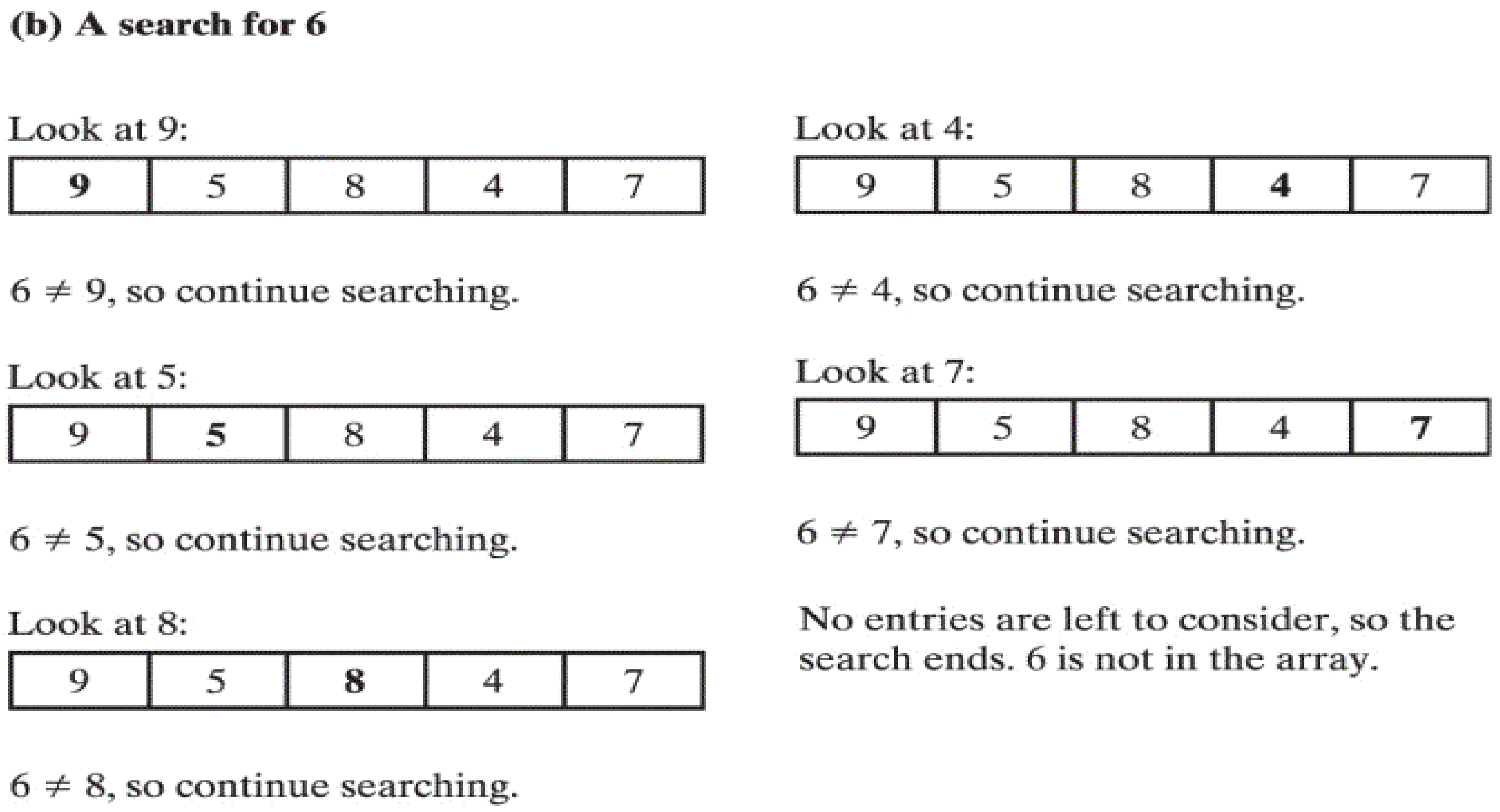
Look at 8:

9	5	**8**	4	7

$8 = 8$, so the search has found 8.

Fig. 16-2 An iterative sequential search of an array that (a) finds its target

Sequential Search - *Unsorted* Array (3)



(b) A search for 6

Look at 9:

9	5	8	4	7

6 ≠ 9, so continue searching.

Look at 5:

9	**5**	8	4	7

6 ≠ 5, so continue searching.

Look at 8:

9	5	**8**	4	7

6 ≠ 8, so continue searching.

Look at 4:

9	5	8	**4**	7

6 ≠ 4, so continue searching.

Look at 7:

9	5	8	4	**7**

6 ≠ 7, so continue searching.

No entries are left to consider, so the search ends. 6 is not in the array.

Fig. 16-2 An iterative sequential search of an array that (b) does not find its target

Recursive Sequential Search

Sequential Search on a Sorted List vs an Unsorted List

Target: 44

Sorted Data

10	20	30	40	50	60	70

Unsorted Data

50	70	80	10	60	30	40

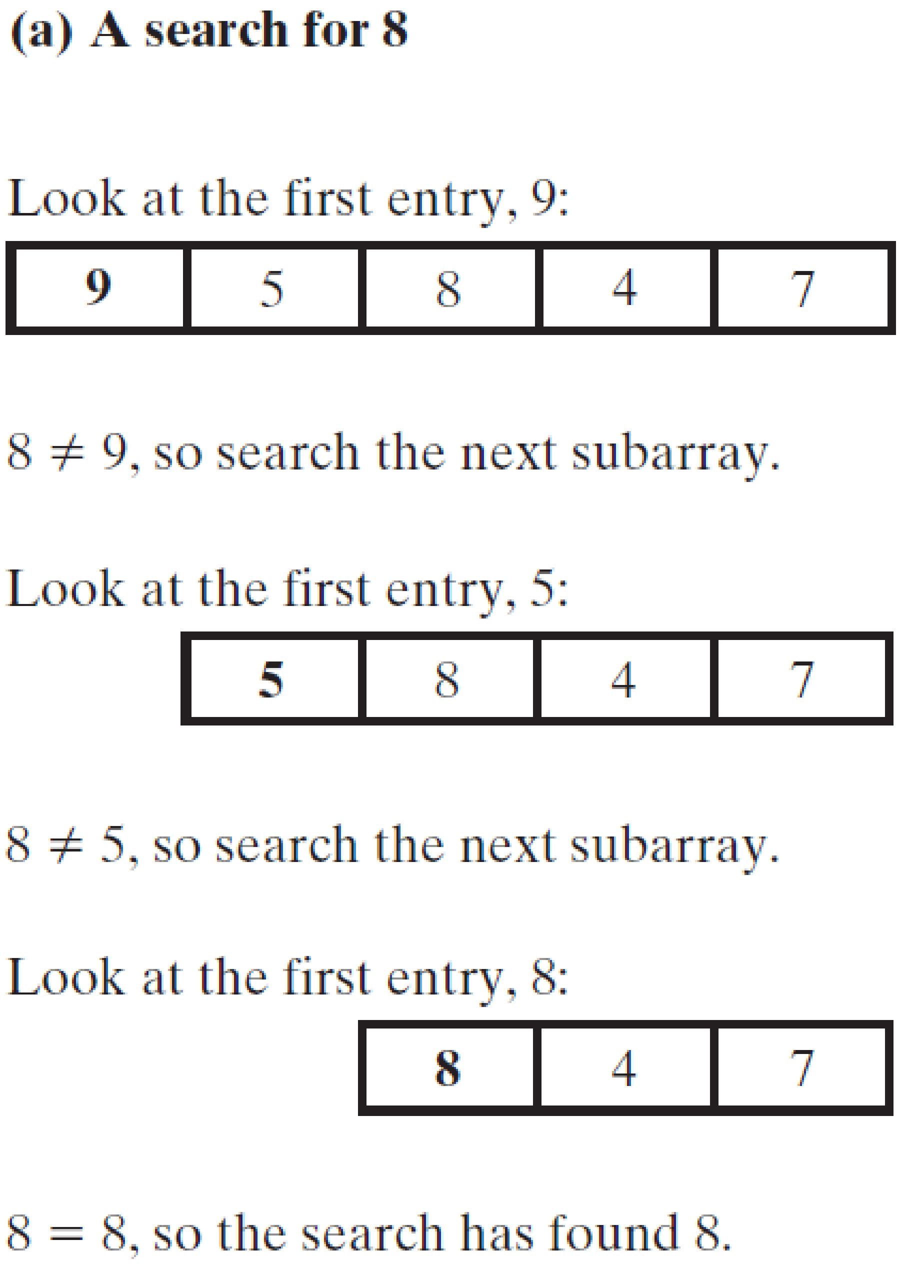
Recursive Algorithm for Sequential Search on *Unsorted Array*

```
Algorithm: Search a[first] through a[last] for desiredItem

  if (there are no elements to search)
    return false
  else if (desiredItem equals a[first])
    return true
  else
    return result of searching a[first+1] through a[last]
```

Recursive Method Sequential Search on *Unsorted Array*

```
public boolean contains(T anEntry) {
  return search(0, length - 1, anEntry);
}

private boolean search(int first, int last, T desiredItem){
  boolean found;

  if (first > last)
    found = false; // no elements to search
  else if (desiredItem.equals(array[first]))
    found = true;
  else
    found = search(first + 1, last, desiredItem);
  return found;
}
```

Recursive sequential search on *unsorted* array example: target found



(a) A search for 8

Look at the first entry, 9:

9	5	8	4	7

8 ≠ 9, so search the next subarray.

Look at the first entry, 5:

5	8	4	7

8 ≠ 5, so search the next subarray.

Look at the first entry, 8:

8	4	7

8 = 8, so the search has found 8.

Recursive sequential search on *unsorted* array example: target not found

(b) A search for 6

Look at the first entry, 9:

9	5	8	4	7

6 ≠ 9, so search the next subarray.

Look at the first entry, 5:

5	8	4	7

6 ≠ 5, so search the next subarray.

Look at the first entry, 8:

8	4	7

6 ≠ 8, so search the next subarray.

Recursive sequential search on *unsorted* array example: target not found (cont'd)

Look at the first entry, 4:

4	7

$6 \neq 4$, so search the next subarray.

Look at the first entry, 7:

7

$6 \neq 7$, so search an empty array.

No entries are left to consider, so the search ends. 6 is not in the array.

Efficiency of a Sequential Search

- Best case O(1)
  - Locate desired item first
- Worst case O(n)
  - Must look at all the items
- Average case O(n)
  - Must look at half the items
  - O(n/2) is still O(n)
- The time it takes (on average, and worst case) is *linear*, or O(n), in the length of the list.

Searching a *Sorted* Array

- A sequential search can be more efficient if the data is sorted

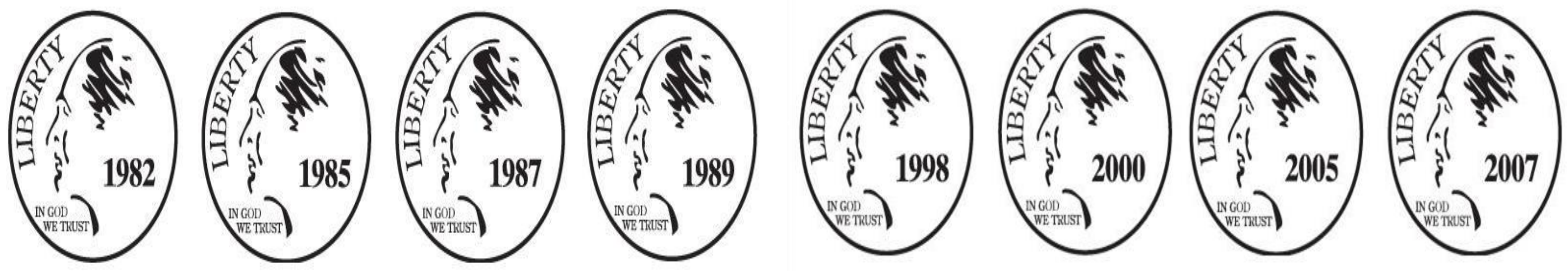
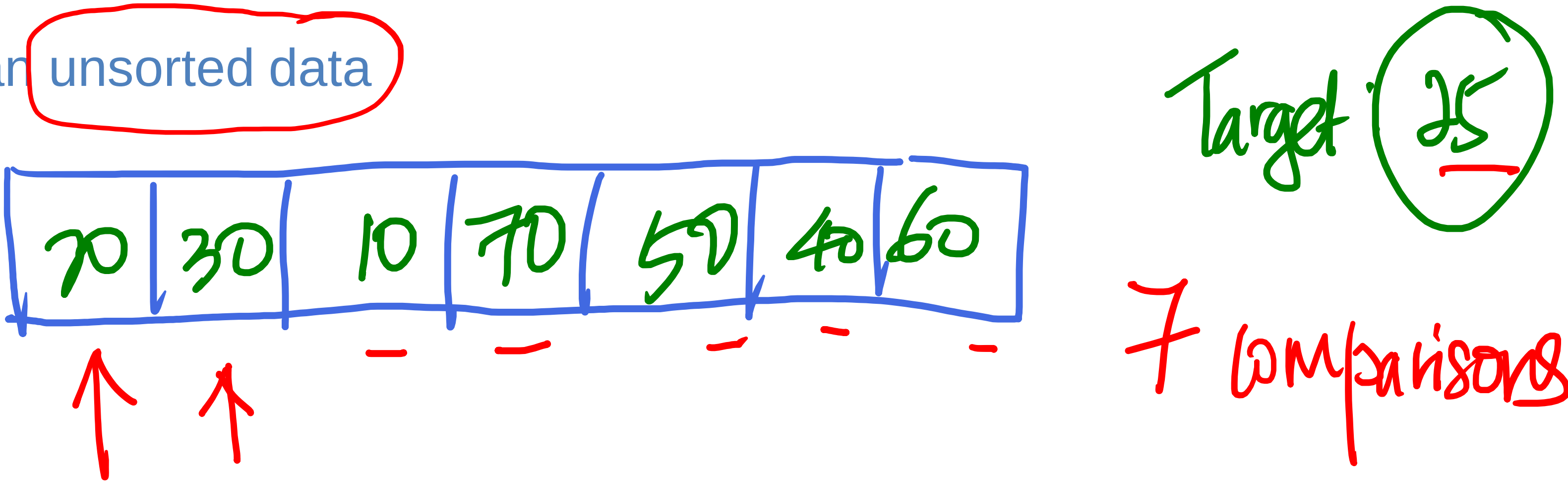


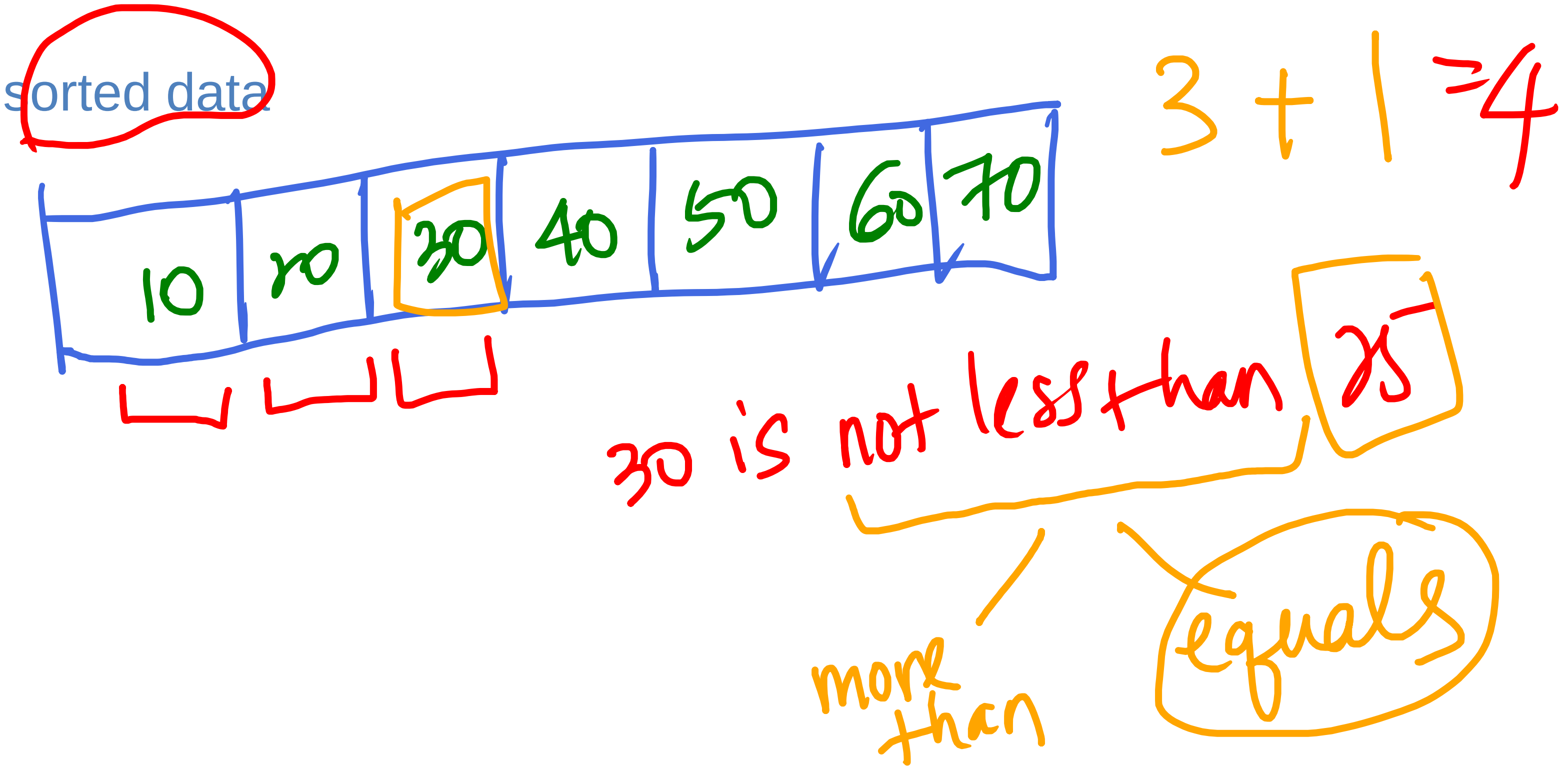
Fig. 16-4 Coins sorted by their mint dates.

- It is also more useful for the sequential search method to *return the location of the target* in the array.

Searching an unsorted data



Searching a sorted data



Binary Search

- Can only be applied to *sorted* arrays.
- Repeatedly divides the array in half until the element is found or there is nothing left to search

I don't need this half of the book. I'll just throw it away.

Fig. 16-5 Ignoring one-half of the data when the data is sorted.

Binary Search: General Idea

Divide the array in half and compare the target with the element at the array's mid-point:

- If they match, the target is found;
- Otherwise, if the target is less than the middle element, search the left subarray;
- Otherwise, search the right subarray.

Recursive Binary Search Algorithm

```
Algorithm binarySearch (a, first, last, desiredItem)
  mid = (first + last) / 2
  if (first > last)
     return false
  else if (desiredItem equals a[mid])
     return true
  else if (desiredItem < a[mid])
     return binarySearch(a, first, mid - 1, desiredItem)
  else       // desiredItem > a[mid]
     return binarySearch(a, mid + 1, last, desiredItem)
```

Binary Search Example: Target **Found**

(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

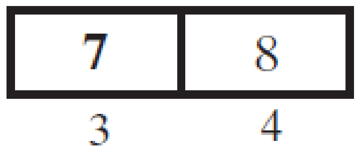
Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

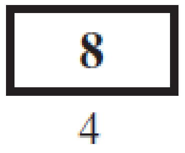
Binary Search Example: Target Found (cont’d)

Look at the middle entry, 7:



$8 > 7$, so search the right half of the array.

Look at the middle entry, 8:



$8 = 8$, so the search ends. 8 is in the array.

Binary Search Example: Target **Not Found**

(b) A search for 16

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

16 > 10, so search the right half of the array.

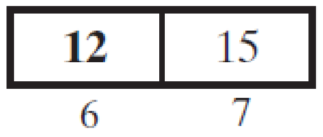
Look at the middle entry, 18:

12	15	**18**	21	24	26
6	7	8	9	10	11

16 < 18, so search the left half of the array.

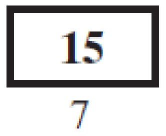
Binary Search Example: Target **Not Found** (cont'd)

Look at the middle entry, 12:



16 > 12, so search the right half of the array.

Look at the middle entry, 15:



16 > 15, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.

Efficiency of a Binary Search

- Best case $O(1)$ Locate desired item first
- Worst case $O(\log n)$ Must look at all the items
- Average case $O(\log n)$
- In the recursive binary algorithm, with each comparison we halve the size of the list under consideration. Since n is the size of the array, the consecutive sizes of the arrays in the succeeding comparisons will be

$$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, \ldots \quad \text{OR} \quad \frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, \frac{n}{2^4}, \ldots$$

Comparison #	Total Elements
0	n
1	$n/2 \rightarrow n/2^1$
2	$n/4 \rightarrow n/2^2$
3	$n/8 \rightarrow n/2^3$
$\vdots$	
k	$n/2^k = 1$

$$\frac{n}{2^k} = 1 \qquad \underline{O(\log_2 n)}$$

$$n = 2^k$$

$$\log_2 n = k \underbrace{\log_2 2}_{1}$$

$$\log_2 n = k$$

Iterative Sequential Search of an Unsorted Chain

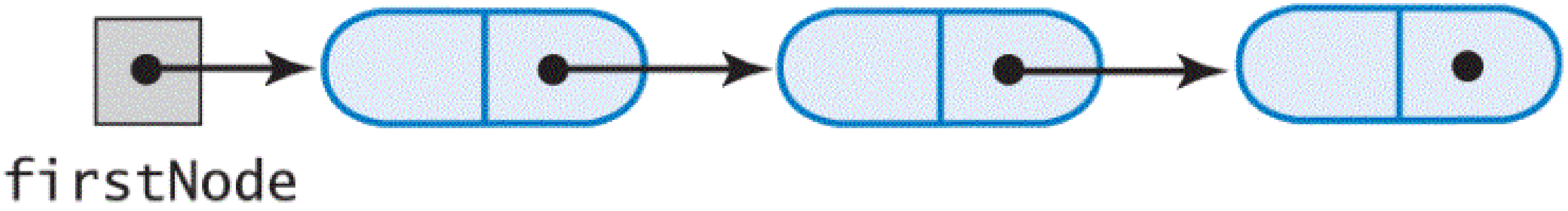
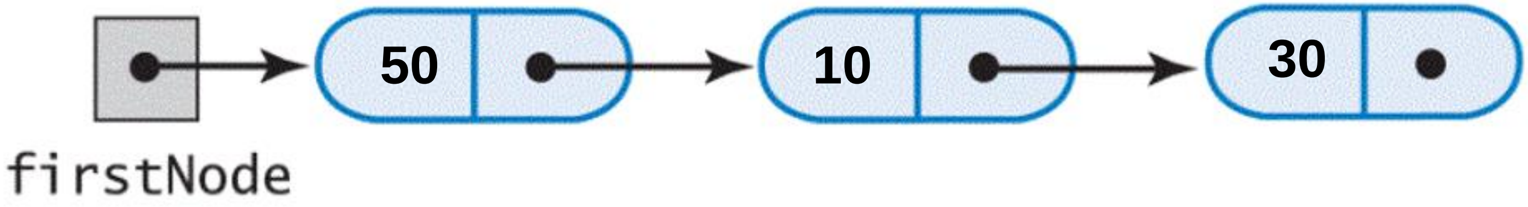


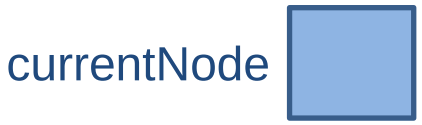
Fig. 16-7 A chain of linked nodes that contain the entries in a list.

Iterative Sequential Search of an Unsorted Chain

```
public boolean contains(T anEntry) {
  boolean found = false;
  Node currentNode = firstNode;

  while (!found && (currentNode != null)) {
    if (anEntry.equals(currentNode.data))
      found = true;
    else
      currentNode = currentNode.next;
  }
  return found;
}
```

Local reference variable `currentNode` moves from a node to other.



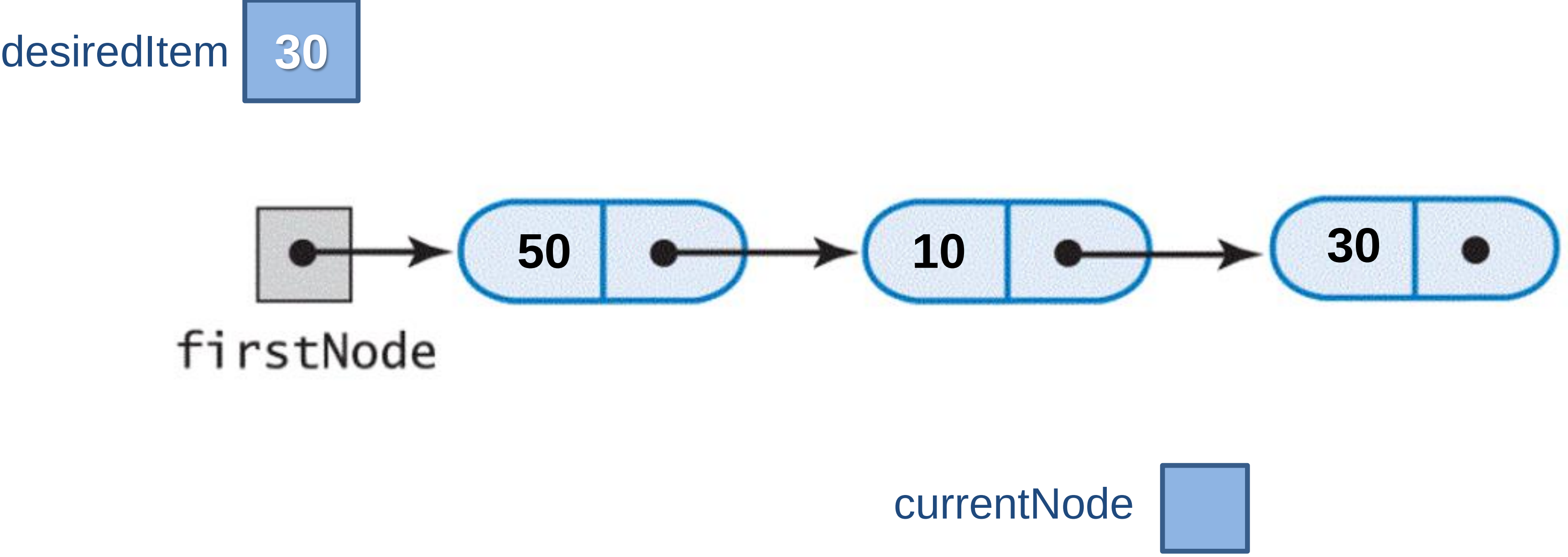


```
public boolean contains(T anEntry) {
  boolean found = false;
  Node currentNode = firstNode;

  while (!found && (currentNode != null)) {
    if (anEntry.equals(currentNode.data))
      found = true;
    else
      currentNode = currentNode.next;
  }
  return found;
}
```

Local refe

Recursive Sequential Search of an Unsorted Chain

```
private boolean search(Node currentNode, T desiredItem) {
  boolean found;
  if (currentNode == null)
   found = false;
  else if (desiredItem.equals(currentNode.data))
   found = true;
  else
   found = search(currentNode.next, desiredItem);
  return found;
}

public boolean contains(T anEntry){
  return search(firstNode, anEntry);
}
```

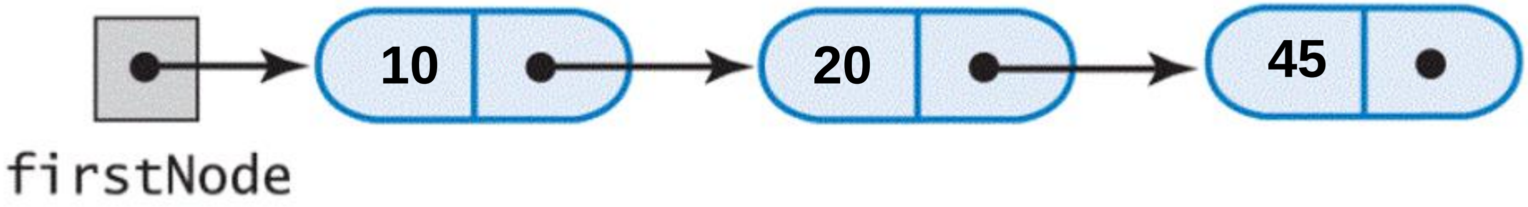


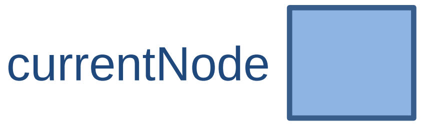
search(30)

```
private boolean search(Node currentNode, T desiredItem) {
  boolean found;
  if (currentNode == null)
   found = false;
  else if (desiredItem.equals(currentNode.data))
   found = true;
  else
   found = search(currentNode.next, desiredItem);
  return found;
}

public boolean contains(T anEntry){
  return search(firstNode, anEntry);
}
```

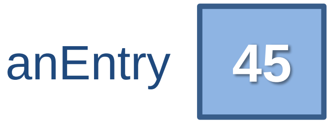
Sequential Search of a Sorted Chain

```
public boolean contains(T anEntry){
  Node currentNode = firstNode;
  while ((currentNode != null) &&
        (anEntry.compareTo(currentNode.data)> 0)) {
    currentNode = currentNode.next;
  }
   return (currentNode != null) &&
           anEntry.equals(currentNode.data);
}
```





```
public boolean contains(T anEntry){
  Node currentNode = firstNode;
  while ((currentNode != null) &&
          (anEntry.compareTo(currentNode.data)> 0)) {
    currentNode = currentNode.next;
  }
   return (currentNode != null) &&
            anEntry.equals(currentNode.data);
}
```



Exercise 8a.1

Compare and contrast how each of the following methods would be implemented in an *unsorted list* and a *sorted list* using linked implementation. Provide detailed explanations. Your answers may also include appropriate diagrams for illustration.

(ii) `boolean contains(T anEntry)` – returns true if the given entry is found in the list; false otherwise. (6 marks)

Unsorted List

(Iterative solution)
The search will start from the first node. The target value will be compared with the data in the node. If they are equal, the search ends here. If they are not equal, the next node will be processed. This will continue until the target value is found in the chain, or when the end of the chain has been reached.

<include your diagram here>

Sorted List

(Iterative solution)
The search will start from the first node. The target value will be compared with the data in the node. If the target value is larger, then the next node will be processed. This will happen iteratively, and the loop will stop when either the end of the chain has been reached (i.e. the node has a null value) or when the target value is not larger than the value of the data in the node. Next, an equality comparison will be made between the target value and the data in the node. If the comparison matches, the Boolean value TRUE will be returned, otherwise FALSE will be returned.

<include your diagram here>

Efficiency of a Sequential Search of a Chain

- Best case O(1)
  - Locate desired item first
- Worst case O(n)
  - Must look at all the items
- Average case O(n)
  - Must look at half the items
  - O(n/2) is just O(n)

Binary search of a Sorted Chain

- Binary search works perfectly if lists are implemented with arrays. But, it is **impractical** with linked implementations of lists.
- With a linked implementation we cannot find the middle of the list in constant time: it takes linear time and this takes away the speedup we get from binary search.

Choosing between a sequential search and a binary search

- Both of search algorithms are applicable to array.
- If the array size is small, use sequential search.
- If the array is large and already sorted, a binary search is much faster than a sequential search.
- A binary search of a chain of linked nodes is impractical.

Choosing a Search Method

	Best case	Average case	Worst case
Sequential search (unsorted data)	O(1)	O(n)	O(n)
Sequential search (sorted data)	O(1)	O(n)	O(n)
Binary search (sorted array)	O(1)	O(log n)	O(log n)

Fig. 16-8 The time efficiency of searching, expressed in Big Oh notation

Choosing between an iterative search and a recursive search

- The recursive sequential search is tail recursion, it can save some time and space using the iterative version of the search.
- The binary search is fast, so using recursion will not require much additional space for the recursive calls.
- For coding the binary search recursively is easier than coding it iteratively.

Exercise 8a.2

Given the array:

5	10	15	20	25	30	35	40	45	50

a. How will binary search for the target value 50.

b. What are the best-case and worst-case Big-O running times of the binary search algorithm.

$\text{mid}_1 = (0 + 9) / 2 = 4$. The value at index 4 is 25.

Since the target is **50** > 25, search the right subarray [5..9].

$\text{mid}_2 = (5 + 9) / 2 = 7$. The value at index 7 is 40.

Since the target is **50** > 40, search the right subarray [8..9].

$\text{mid}_3 = (8 + 9) / 2 = 8$. The value at index 8 is 45.

Since the target is **50** > 45, search the right subarray [9..9].

$\text{mid}_4 = (9 + 9) / 2 = 9$. The value at index 9 is 50.

Therefore, the target value has been located at index 9.

- **Best case**: target is at the array's mid. Only 1 comparison is required, i.e. O(1).
- **Worst case**: when the target is not found and its value is either smaller than the smallest value in the array or larger than the largest value in the array. As each comparison reduces the number of elements to be searched to half of the existing number of (sub)array elements, it gives running time i.e. $O(\log_2 n)$.

Review of Learning Outcomes

You should now be able to

- Implement sequential search and binary search algorithms
- Assess the time efficiency of sequential search and binary search algorithms

To Do

- Review the slides and source code for this chapter.
- Read up the relevant portions of the recommended text.
- Do the practical questions for this chapter.

References

- Carrano, F. M., 2019, Data Structures and Abstractions with Java, 5th edn, Pearson
- Liang, Y.D., 2018. Introduction to Java Programming and Data Structures.11th ed.United Kingdom:Pearson