CSC212
Data Structure

COMPUTER SCIENCE
CITY COLLEGE OF NEW YORK

Lecture 18
Searching

Instructor: George Wolberg
Department of Computer Science
City College of New York

Topics

- Applications
- Most Common Methods
  - Serial Search
  - Binary Search
  - Search by Hashing (next lecture)
- Run-Time Analysis
  - Average-time analysis
  - Time analysis of recursive algorithms

Applications

- Searching a list of values is a common computational task
- Examples
  - database: student record, bank account record, credit record...
  - Internet – information retrieval: Google, Yahoo
  - Biometrics –face/ fingerprint/ iris IDs

Most Common Methods

- Serial Search
  - simplest, O(n)
- Binary Search
  - average-case O(log n)
- Search by Hashing (the next lecture)
  - better average-case performance

Serial Search

- A serial search algorithm steps through (part of) an array one item a time, looking for a “desired item”

Pseudocode for Serial Search

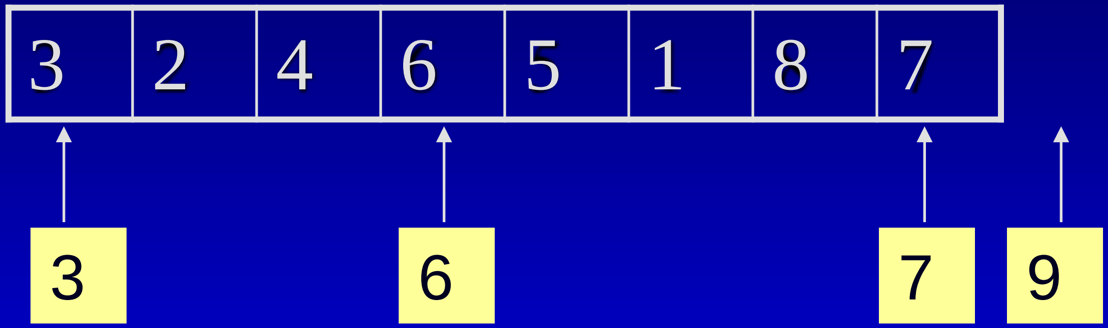
```
// search for a desired item in an array a of size n

set i to 0  and set found to false;

while (i<n && ! found)
{
   if (a[i] is the desired item)
         found = true;
   else
         ++i;
}

if (found)
      return i; // indicating the location of the desired item
else
      return –1; // indicating “not found”
```

Serial Search -Analysis



- Size of array: n
- Best-Case: O(1)
  - item in [0]
- Worst-Case: O(n)
  - item in [n-1] or not found
- Average-Case
  - usually requires fewer than n array accesses
  - But, what are the average accesses?

Average-Case Time for Serial Search

- A more accurate computation:
  - Assume the target to be searched is in the array
  - and the probability of the item being in any array location is the same
- The average accesses

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

When does the best-case time make more sense?

- For an array of **n** elements, the best-case time for serial search is just one array access.
- The best-case time is more useful if the probability of the target being in the [0] location is the highest.
  - or loosely if the target is most likely in the front part of the array

Binary Search

- If **n** is huge, and the item to be searched can be in any locations, serial search is slow on average
- But if the items in an array are sorted, we can somehow know a target's location earlier
  - Array of integers from smallest to largest
  - Array of strings sorted alphabetically (e.g. dictionary)
  - Array of students records sorted by ID numbers

Binary Search in an Integer Array

if target is in the array

2	3	6	7	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Items are sorted
  - target = 16
  - n = 8
- Go to the middle location i = n/2
- if (a[i] is target)
  - done!
- else if (target <a[i])
  - go to the first half
- else if (target >a[i])
  - go to the second half

Binary Search in an Integer Array

if target is in the array

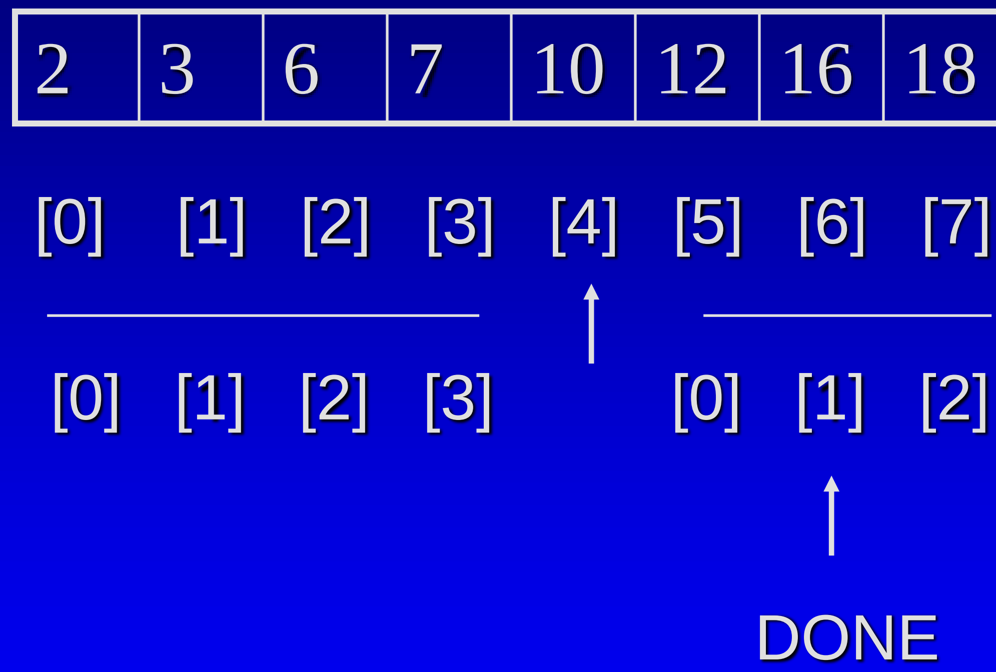
- Items are sorted
  - target = 16
  - n = 8
- Go to the middle location i = n/2
- if (a[i] is target)
  - done!
- else if (target <a[i])
  - go to the first half
- else if (target >a[i])
  - go to the second half

2	3	6	7	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	[1]	[2]	[3]	↑	[0]	[1]	[2]

Binary Search in an Integer Array

if target is in the array

- Items are sorted
  - target = 16
  - n = 8
- Go to the middle location i = n/2
- if (a[i] is target)
  - done!
- else if (target <a[i])
  - go to the first half
- else if (target >a[i])
  - go to the second half



Binary Search in an Integer Array

if target is in the array

- Items are sorted
  - target = 16
  - n = 8
- Go to the middle location i = n/2
- if (a[i] is target)
  - done!
- else if (target <a[i])
  - go to the first half
- else if (target >a[i])
  - go to the second half

2	3	6	7	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

[0] [1] [2] [3]　　　　[0] [1] [2]

DONE

recursive calls: what are the parameters?

Binary Search in an Integer Array

if target is in the array

- Items are sorted
  - target = 16
  - n = 8
- Go to the middle location i = n/2
- if (a[i] is target)
  - done!
- else if (target <a[i])
  - go to the first half
- else if (target >a[i])
  - go to the second half

2	3	6	7	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

[0] [1] [2] [3]   [0] [1] [2]

DONE

recursive calls with parameters:

array, start, size, target

found, location // reference

Binary Search in an Integer Array

if target is not in the array

- Items are sorted
  - target = 17
  - n = 8
- Go to the middle location i = n/2
- if (a[i] is target)
  - done!
- else if (target <a[i])
  - go to the first half
- else if (target >a[i])
  - go to the second half

2	3	6	7	10	12	16	18

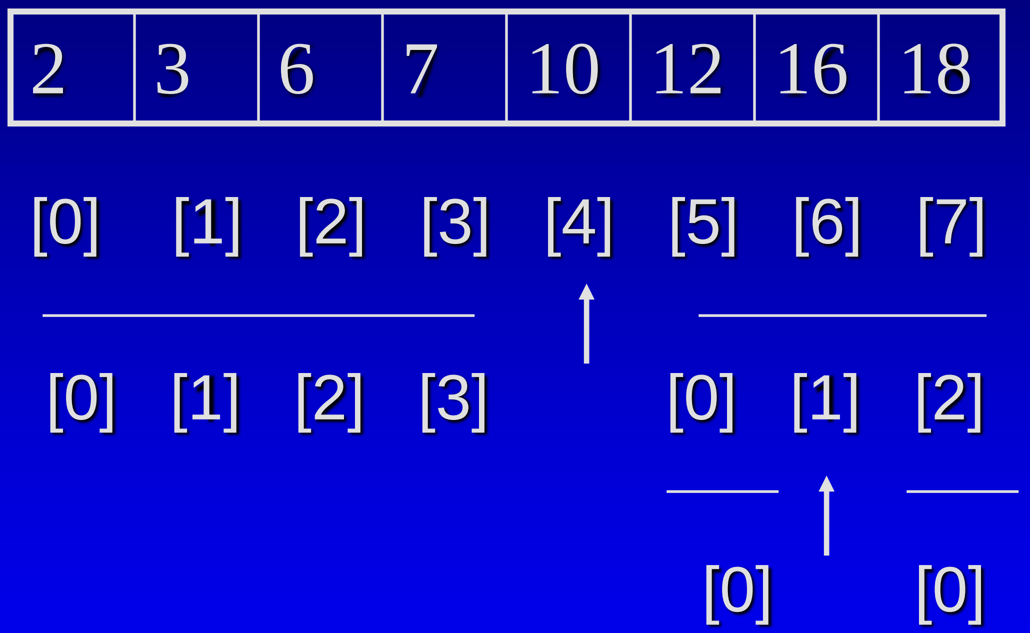
[0] [1] [2] [3] [4] [5] [6] [7]

[0] [1] [2] [3]   [0] [1] [2]

Binary Search in an Integer Array

if target is not in the array

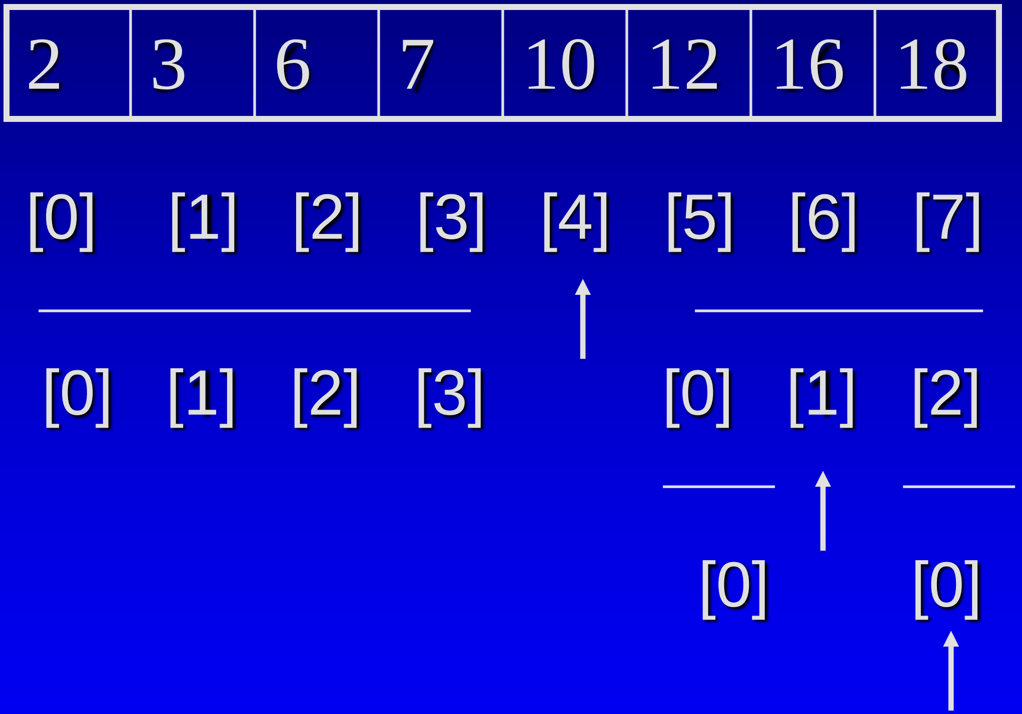
- Items are sorted
  - target = 17
  - n = 8
- Go to the middle location i = n/2
- if (a[i] is target)
  - done!
- else if (target <a[i])
  - go to the first half
- else if (target >a[i])
  - go to the second half



Binary Search in an Integer Array

if target is not in the array

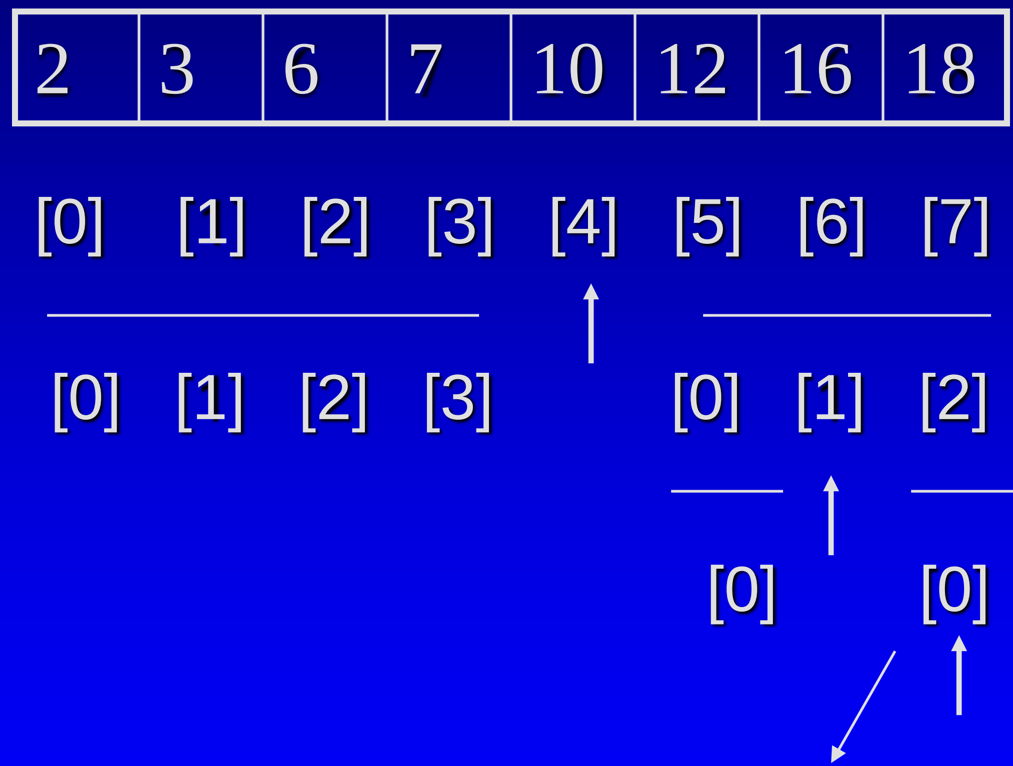
- Items are sorted
  - target = 17
  - n = 8
- Go to the middle location i = n/2
- if (a[i] is target)
  - done!
- else if (target <a[i])
  - go to the first half
- else if (target >a[i])
  - go to the second half



Binary Search in an Integer Array

if target is not in the array

- Items are sorted
  - target = 17
  - n = 8
- Go to the middle location i = n/2
- if (a[i] is target)
  - done!
- else if (target <a[i])
  - go to the first half
- else if (target >a[i])
  - go to the second half



the size of the first half is 0!

Binary Search in an Integer Array

if target is not in the array

- target = 17

2	3	6	7	10	12	16	18

[0] [1] [2] [3] [4] [5] [6] [7]

[0] [1] [2] [3] [0] [1] [2]

[0] [0]

- **If (n == 0)**
  - **not found!**
- Go to the middle location i = n/2
- if (a[i] is target)
  - done!
- else if (target <a[i])
  - go to the first half
- else if (target >a[i])
  - go to the second half

the size of the first half is 0!

Binary Search Code

- 6 parameters
- 2 stopping cases
- 2 recursive call cases

```
void search (const int a[ ], size_t first, size_t size,
                    int target,
                    bool& found,  size_t& location)
 {
   size_t middle;

   if (size == 0) // stopping case if not found
      found = false;
   else
   {
      middle = first + size/2;
      if (target == a[middle])  // stopping case if found
      {
         location = middle;
         found = true;
      }
      else if (target < a[middle]) // search the first half
         search(a, first, size/2, target, found, location);
      else  //search the second half
         search(a, middle+1, (size-1)/2, target, found, location);
   }
}
```

Binary Search - Analysis

- Analysis of recursive algorithms
- Analyze the worst-case
- Assuming the target is in the array
- and we always go to the second half

```
void search (const int a[ ], size_t first, size_t size,
                     int target,
                     bool& found,  size_t& location)
 {
    size_t middle;

    if (size == 0) // stopping case if not found
       found = false;
    else
    {
       middle = first + size/2;
       if (target == a[middle])  // stopping case if found
       {
          location = middle;
          found = true;
       }
       else if (target < a[middle]) // search the first half
          search(a, first, size/2, target, found, location);
       else  //search the second half
          search(a, middle+1, (size-1)/2, target, found, location);
    }
}
```

Binary Search - Analysis

- Analysis of recursive algorithms
- Define T(n) is the total operations when size=n

T(n) = 6+T(n/2)

T(1) = 6

```
void search (const int a[ ], size_t first, size_t size,
                    int target,
                    bool& found,  size_t& location)
{
   size_t middle;

   if (size == 0) // 1 operation
      found = false;
   else
   {
      middle = first + size/2; // 1 operation
      if (target == a[middle])  // 2 operations
      {
         location = middle; // 1 operation
         found = true; // 1 operation
      }
      else if (target < a[middle]) // 2 operations
         search(a, first, size/2, target, found, location);
      else  //  T(n/2) operations for the recursive call
         search(a, middle+1, (size-1)/2, target, found, location);
   } // ignore the operations in parameter passing
}
```

Binary Search - Analysis

- How many recursive calls for the longest chain?

$$
\begin{aligned}
&T(n) \\
&= 6 + T(n/2^1) \\
&= 6 + 6 + T(n/2^2) \\
&= \ldots \\
&= 6 + 6 + \ldots + 6 + T(n/2^m) \\
&= 6 + 6 + \ldots + 6 + 6 \\
&= 6(m+1) \\
&= 6\log_2 n + 6
\end{aligned}
$$

original call

1st recursion, 1 six

2nd recursion, 2 six

*m*th recursion, m six

and $n/2^m = 1$ – target found

depth of the recursive call
$m = \log_2 n$

Worst-Case Time for Binary Search

- For an array of n elements, the worst-case time for binary search is logarithmic
  - We have given a rigorous proof
  - The binary search algorithm is very efficient

- What is the average running time?
  - The average running time for actually finding a number is O(log n)
  - Can we do a rigorous analysis????

Summary

- Most Common Search Methods
  - Serial Search – O(n)
  - Binary Search – O (log n)
  - Search by Hashing (*) – better average-case performance (next lecture)
- Run-Time Analysis
  - Average-time analysis
  - Time analysis of recursive algorithms

Homework

- Review Chapters 10 & 11 (Trees), and
  - do the self_test exercises
- Read Chapters 12 & 13, and
  - do the self_test exercises
- Homework/Quiz (on Searching):
  - Self-Test 12.7, p 590 (binary search re-coding)