Binary Search

CS A250 – C++ Programming II

Search Algorithms

- Search algorithms are very common
  - They search a list
  - Look at each item in the list and compare to the search item
- We will consider two ways to search:
  - **Linear**
    - Also called **sequential** search
    - Can be done in an **ordered/unordered** list
  - **Binary**
    - Can be done ***only*** in an ***ordered*** list

Linear Search

- **Linear Search** searches through the elements of an arbitrary sequence (in this case, a **vector v**) until
  - The match is found **OR**
  - It reaches the end of the sequence

```
int count = 0;
bool found = false;
int size = static_cast<int>(v.size());
while (!found && count < size)
{
    if (v[count] == value)
        found = true;
    else
        ++count;
}
```

Assumption: All elements in the list are ***unique***.

How Many Comparisons?

- You always look at the **worst case**:
  - Assuming your list has 20 elements
    - You compare each element at most once with the given element
    - If the element you are searching for happens to be the last element in the list:
      - You have made **20 comparisons**
      - **Generalize:** 20 is denoted by ***n*** (a list of ***n*** elements)
- Therefore, the worst case in a sequential search is ***n* comparisons**.

Binary Search

- **Binary search**
  - Is faster than **linear search**
    - **BUT** assumes **array is sorted**
  - Breaks the list in **half**
    - Determines if item in 1st or 2nd half
    - Then searches again just that half
      - Can be done **recursively**.

Execution of Binary Search

- A **sorted** array of 10 elements
  - Search for **63**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Since indices are type **int** result will be truncated

Find the **middle** ➔ (0 + 9) / 2 = 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Find the **middle** ➔ (0 + 9) / 2 = 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Is [4] equal to **63**? No
Is **63** > or < than [4]? >
Check between [5] and [9]

Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Find the **middle** ➔ (5 + 9) / 2 = **7**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Find the **middle** ➔ (5 + 9) / 2 = **7**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Is [**7**] equal to **63**? No
Is **63** > or < than [**7**]? <
Check between [5] and [6]

Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Find the **middle** ➔ (5 + 6) / 2 = **5**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Find the **middle** ➔ (5 + 6) / 2 = **5**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

Is [**5**] equal to **63**? Yes
Stopping case → **63** found
Location = **mid**

Execution of Binary Search (cont.)

- A **sorted** array of 10 elements
  - Search for **63**

Number of **comparisons** ➔ **3**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	20	35	41	57	63	75	80	85	90

mid

IMPLEMENTATION

- **Binary search** can be implemented
  - **Iteratively**
    - Using a **while** loop
  - **Recursively**
    - **IF/ELSE** statement and **call to itself**

  - **Stopping case** for both implementations:
    - **if (first > last)** → no elements between them, so **key** cannot be there!
    - **if (key == a[mid])** → found!

Efficiency of Binary Search

- **Binary search** is ***very efficient***
  - Extremely ***fast,*** compared to **sequential search**
- Half of array eliminated at start!
  - Then a quarter, then 1/8, etc.
  - Essentially eliminate half with each call
- Example: Array of 100 elements
  - In this case, a binary search
    never needs more than 7 compares!

Recursive Solutions

- If done **recursively**, then the **binary search** algorithm actually solves "more general" problem
  - Original goal:
    - Design function to search an entire list
  - **BUT** you can search any ***interval of the list***
    - By specifying bounds *first* and *last*
    - Very common when designing recursive functions

Binary Search (end)