Binary Search

CMPT 125
Mo Chen
SFU Computing Science
27/1/2020
Lecture 12

Today

● Binary Search
What if the array was ordered?

Think of searching a dictionary for a word?

- Strategy: *Not* one word at a time in sequential order starting from aardvark, etc.

- Strategy: Jump to where you estimate the word to be based on what you know about the alphabet.
  Refine your jumps + hone in on the correct page quickly.

This is the main idea behind *binary search*.
Divide and Conquer

Generic Strategy (Paradigm):

1. **Divide:** Cut the array into 2 or more roughly equally sized pieces

2. **Conquer:** Use what you know about the pieces to solve the original problem
Binary Search

Strategy: Divide and Conquer

1. Examine the *middle* element of the array of candidates. This divides the array into two [roughly] equal halves.
2. Compare the middle element with the target.
   - If middle < target then throw out the first half.
   - But if middle > target then throw out second half.
3. Repeat 1-3 until middle == target (found!) or no candidates remain (fail!).

E.g., target = 42:  

- -8 -7 -5 -2 0 4 6 7 17 20 28 29 42 49 64
Binary Search

Strategy: Divide and Conquer

1. Examine the *middle* element of the array of candidates. This divides the array into two [roughly] equal halves.

2. Compare the middle element with the target.
   - If middle < target then throw out the first half.
   - But if middle > target then throw out second half.

3. Repeat 1-3 until middle == target (found!) or no candidates remain (fail!).

E.g., target = 42:
Binary Search

Strategy: Divide and Conquer

1. Examine the *middle* element of the array of candidates. This divides the array into two [roughly] equal halves.

2. Compare the middle element with the target.
   ○ If middle < target then throw out the first half.
   ○ But if middle > target then throw out second half.

3. Repeat 1-3 until middle == target (found!) or no candidates remain (fail!).

E.g., target = 42: -8 -7 -5 -2 0 4 6 7 17 20 28 29 42 49 64
Binary Search

Strategy: Divide and Conquer

1. Examine the middle element of the array of candidates. This divides the array into two [roughly] equal halves.
2. Compare the middle element with the target.
   - If middle < target then throw out the first half.
   - But if middle > target then throw out second half.
3. Repeat 1-3 until middle == target (found!) or no candidates remain (fail!).

E.g., target = 42:

```
-8 -7 -5 -2 0 4 6 7 17 20 28 29 42 49 64
```

return true or index=12
Binary Search

Requirements (Pre-Conditions):

- Candidate array must be sorted

How to keep track of the list of candidates?

- Use integers `first` and `last` for remaining candidates `arr[first..last]`
- Initially, `first=0; last=len-1`
- Middle element is at index `(first+last)/2`

E.g., target = 42:

```
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
-8 -7 -5 -2 0 4 6 7 17 20 28 29 42 49 64
```

first = 0
last = 14
mid = 7
Binary Search

Requirements (Pre-Conditions):

- Candidate array must be sorted

How to keep track of the list of candidates?

- Use integers first and last for remaining candidates arr[first..last]
- Initially, first=0; last=len-1
- Middle element is at index \((first+last)/2\)

E.g., target = 42:

\[
\begin{array}{ccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
-8 & -7 & -5 & -2 & 0 & 4 & 6 & 7 & 17 & 20 & 28 & 29 & 42 & 49 & 64 \\
\end{array}
\]

first = 0  
last = 14  
mid = 7
Binary Search

Requirements (Pre-Conditions):

- Candidate array must be sorted

How to keep track of the list of candidates?

- Use integers \( \text{first} \) and \( \text{last} \) for remaining candidates \( \text{arr}[\text{first}..\text{last}] \)
- Initially, \( \text{first}=0; \text{last}=\text{len}-1 \)
- Middle element is at index \( (\text{first}+\text{last})/2 \)

E.g., target = 42:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
-8 -7 -5 -2 0 4 6 7 17 20 28 29 42 49 64
```

\[
\begin{align*}
\text{first} &= 0 \\
\text{last} &= 14 \\
\text{mid} &= 7
\end{align*}
\]

8 12
14 14
11 13
Binary Search

Requirements (Pre-Conditions):

- Candidate array must be sorted

How to keep track of the list of candidates?

- Use integers `first` and `last` for remaining candidates `arr[first..last]`
- Initially, `first=0; last=len-1`
- Middle element is at index `(first+last)/2`

E.g., target = 42:

```
-8 -7 -5 -2 0 4 6 7 17 20 28 29 42 49 64
```

- `first = 0`
- `last = 14`
- `mid = 7`

```
8 12 12
14 14 12
11 13 12
```

return true or index=12
Binary Search Code

```c
int BinarySearch(int arr[], int len, int target) {
    int first = 0;
    int last = len - 1;
    while (first <= last) {
        int mid = first + (last - first) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] > target) {
            last = mid - 1;
        } else {
            first = mid + 1;
        }
    }
    return -1;  // Target not found
}
```

- **Search candidate array** `arr[first..last]` while not empty
- **Compare with the middle element**
- **Algorithm:**
  - found if equal to `target`, so return position
  - throw out second half if greater than `target` **OR**
  - throw out first half if less than `target`
- **No candidates, so return fail**
```c
int BinarySearch(int arr[], int len, int target) {
    int first = 0;
    int last = len-1;
    while (first <= last) {
        int mid = (first+last) / 2;
        if (target == arr[mid]) return mid;
        if (target < arr[mid]) last = mid-1;
        else first = mid+1;
    }
    return -1;
}
```

// Q. What’s a good assertion this time?
int BinarySearch(int arr[], int len, int target) {

  ● Search candidate array  \texttt{arr[0..len-1]}
  ● Algorithm:
    ○ return fail if empty

  ● Compare with the middle element + re-search
  ● Algorithm:
    ○ found if equal to \texttt{target}, so return true
    ○ throw out second half if greater than \texttt{target} OR
    ○ throw out first half if less than \texttt{target}

}
int BinarySearch(int arr[], int len, int target) {
    if (len <= 0) {
        return 0;
    }
}

- Compare with the middle element + re-search
- Algorithm:
  - found if equal to target, so return true
  - throw out second half if greater than target OR
  - throw out first half if less than target
int BinarySearch(int arr[], int len, int target) {
    if (len <= 0) {
        return 0;
    }
    int mid = len/2;
    if (target == arr[mid]) return 1;
    if (target < arr[mid]) return BinarySearch(arr,mid,target);
    else return BinarySearch(arr+mid+1,len-mid-1,target);
}

If we go from index 0 to len-1, there are len items
If we go from index a to b, there are b-a+1 items
If we go from index mid+1 to len-1, there are
    (len-1) - (mid+1) + 1 items
Analysis of Binary Search

What’s the worst case on an array of length $N$?

- After one iteration, the possible candidates are [roughly] cut in half.

After $k$ iterations, how many candidates remain?

- Roughly $N / 2^k$

When do you run out of candidates?

- when $2^k \geq N$
- i.e., after $k \geq \log_2 N$ iterations

Thus binary search runs in $O(\log N)$. 
Linear Search vs Binary Search

Even though the inner loop of binary search is more complex than linear search, we expect $O(\log N)$ to outperform $O(N)$ as $N$ gets large.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Linear Search $(3 + 4N)$</th>
<th>Binary Search $(4 + 12 \log_2(N+1))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>31</td>
<td>40</td>
</tr>
<tr>
<td>15</td>
<td>63</td>
<td>52</td>
</tr>
<tr>
<td>100</td>
<td>403</td>
<td>88</td>
</tr>
<tr>
<td>1000</td>
<td>4003</td>
<td>124</td>
</tr>
<tr>
<td>$10^6$</td>
<td>4000003</td>
<td>244</td>
</tr>
<tr>
<td>$10^9$</td>
<td>$4 \times 10^9$</td>
<td>364</td>
</tr>
</tbody>
</table>
Linear Search vs Binary Search

- Binary search has a fast running time.

- Disadvantages?
  - Harder to code
  - Requires the array be sorted

- Keeping the array sorted can be expensive!
  - Significantly more searching than update? Keep list sorted (slow) and use (fast) binary search
  - Significantly more update than search? Keep array unsorted (fast) and use (slow) linear search