COMP 250

Lecture 11

more recursion: binary search, call stack

Sept. 30, 2016
Converting to binary (iterative)

\[
\begin{align*}
  k &= 0 \\
  \text{while } m > 0 \{ \\
    b[k] &= m \% 2 \\
    m &= m / 2 \\
    k &= k + 1 \\
  \}
\end{align*}
\]

Recall that \( m \) in binary needs approximately \( \log m \) bits.
Converting to binary (recursive)

toBinary( m ){
    if m > 0 {
        print m % 2
        toBinary( m / 2 )
    }
}

// prints b[ k ], k = 0, 1, .....
Binary Search

Input:

• a list of \( n \) sorted values
• the value that we are searching for e.g. 17

Output:

If the value is in the list, return its index. Otherwise, return -1.
Example: Search for 17.

What is an efficient way to do this?
Think of how you search for a term in an index. Do you start at the beginning and then scan through to the end? (No.)
compare 17 to \( \rightarrow \)

\[
\begin{array}{c}
-75 \\
-31 \\
-26 \\
-4 \\
1 \\
6 \\
25 \\
26 \\
28 \\
39 \\
72 \\
141 \\
290 \\
300
\end{array}
\]

**low = 0**

**mid = \((\text{low} + \text{high}) / 2\)**

**high = size - 1**
search for 17 here

\[
\begin{array}{c}
-75 \\
-31 \\
-26 \\
-4 \\
1 \\
6 \\
25 \\
26 \\
28 \\
39 \\
72 \\
141 \\
141 \\
290 \\
300 \\
\end{array}
\]

low = 0

mid = (low + high) / 2

high = size - 1
compare 17 to →

<table>
<thead>
<tr>
<th></th>
<th>low = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-75</td>
<td></td>
</tr>
<tr>
<td>-31</td>
<td></td>
</tr>
<tr>
<td>-26</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td></td>
</tr>
<tr>
<td>290</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td></td>
</tr>
<tr>
<td>high</td>
<td></td>
</tr>
</tbody>
</table>

mid = (low + high) / 2
search for 17 here

\[
\begin{align*}
\text{low} &= 0 \\
\text{mid} &= (\text{low} + \text{high}) / 2 \\
\text{high} &= \text{low} + 75
\end{align*}
\]
compare 17 to →

low = 0
mid = (low + high) / 2
high

-75
-31
-26
-4
1
6
25
26
28
39
72
141
290
300
search for 17 here

low = high

{-75
 -31
 -26
 -4
 1
 6
 25
 26
 28
 39
 72
 141
 290
 300
compare 17 to \( \rightarrow \) 6

low = high

so return index -1

(value 17 not found)
binarySearch(list, value){
    low = 0
    high = list.size - 1
    while low <= high {
        // if high - low == 1, then mid == low
        ....

    }

    return -1  // value not in list
}
binarySearch(list, value ){
    low = 0
    high = list.size - 1
    while low <= high {
        mid = low + (high - low)/ 2
        if list[mid] == value
            return mid
        else{
            // modify low or high
        }
    }
    return -1 // value not in list
}
binarySearch(list, value ){
    low = 0
    high = list.size - 1
    while low <= high {  // if high - low == 1, then mid == low
        mid = low + (high - low)/ 2
        if list[mid] == value
            return mid
        else{
            if value < list[mid]
                high = mid - 1  // high can become less than low.
            else
                low = mid + 1
        }
    }
    return -1    // value not found
}
binarySearch(list, value ){
    low = 0
    high = list.size - 1
    while low <= high {
        mid = low + (high - low)/ 2
        if list[mid] == value
            return mid
        else{

            // how to make this recursive ?

            }
    }
    return -1 // value not in list
}
binarySearch( list, value, low, high ){ // pass as parameters
    // low = 0
    // high = list.size - 1
    if low <= high { // if instead of while
        mid = low + (high - low) / 2
        if value == list[mid]
            return value
        else if value < list[mid]
            return binarySearch(list, value, low, mid - 1 )
        else
            return binarySearch(list, value, mid+1, high)
    }
    else
        return -1
Observations about binary search

- The recursive version doesn’t shrink the list. Rather it shrinks the interval [low, high]

- Iterative and recursive versions are similar.

- Q: How many times through the while loop?
  (How many recursive calls?)

A: $O(\log n)$ where $n$ is size of the list. Why? Because each time we are approximately halving the size of the list.
ASIDE: “Tail Recursion”

When the recursive call is the absolute last thing that happens in your method, it is called ‘tail recursion’.

e.g. binarySearch()

This is discussed in COMP 302.
factorial( n ){
   // assume n >= 1
   if n == 1
      return 1
   else
      return n * factorial( n - 1 )
}

Q:  Is this tail recursive?
A:  No, because a multiplication is needed after the return from the recursive call.
COMP 250

Lecture 11

more recursion: binary search, call stack

Sept. 30, 2016
Recall (lecture 7): “call stack”

```c
void mA() {
    mB();
    mC();
}

void main() {
    mA();
}
```
Recursive methods & Call stack

factorial(n) {
    if n == 1
        return 1
    else
        return factorial(n - 1) * n
}

main
    factorial(3)
main
    factorial(2)
    factorial(2)
main
    factorial(2)
main
    factorial(2)
    factorial(2)
main
    factorial(3)
main
    factorial(1)
main
    factorial(3)
main
    factorial(3)
Call stack for TestFactorial
Stack frame (COMP 273)

The call stack consists of “frames” that contain:

- the parameters passed to the method
- local variables of a method
- information about where to return (“what was I doing before this method?”)
Call stack for TestTowerOfHanoi

parameters in current stack frame

```java
public class TestTowerOfHanoi {
    static void tower(int n, String start, String finish, String other) {
        if (n > 0) {
            tower(n - 1, start, other, finish);
            System.out.println("move from " + start + " to " + finish);
            tower(n - 1, other, finish, start);
        }
    }
}
```
void mA() {
    mB();
    mA();
    mC();
}

In a general situation, a method can make both recursive and non-recursive calls.

There is a single call stack for all methods.