Recursive Binary Search

Alexander Kalinowski

April 16, 2025

Recurrence and Recursion

Definition of Recurrence and Recursion

- Recurrence is an equation where a value depends on a function of the prior terms
- Recursion involves taking a larger problem and sub-dividing it into smaller instances of the same problem
- Recursion is a programming technique, while recurrence is a mathematical method

Example of a Recurrent Function

- The Fibonacci numbers are an example of a sequence that can be generated recurrently
- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$
- The next number is the sum of the prior two numbers
- We can write the above as a statement $F_n = F_{n-1} + F_{n-2}$
- This is recurrent since the function for the n-th number depends on the function value for the prior two

Example of Recursion

- An example of recursion is computing the factorial of a number
- Refresher: the factorial of an integer is the product of all intergers less than or equal to that number
- Ex: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$
- We can split this computation into smaller parts
- $5! = 5 \cdot 4!$
- This application of splitting helps us solve larger problems

Recursion in Code

- Splitting problems into smaller sub-problems also has an added benefit
- We can write one program (or function), then **re-use** that function to solve larger problems
- In this case, we can allow for a function to call itself directly
- Particularly useful in search

Recursive Binary Search

Recursive Binary Search

- We begin with a sorted array of integers
- Ex: $[1, 4, 7, 19, 37, 256, 364, 518, 999]$
- We have some target, or *key*, integer we are searching for
- We want to find the index in the array where that element exists
- For example, if our key is 364, we want to return the index 7
  - assuming indexing starts at 1

Recursive Splitting

- We have one large array we need to search through
- Idea: split this array into smaller pieces to make search more efficient
- One way: find the mid-point of the array
- Since the array is sorted, if the *key* we are looking for is larger than the mid-point, eliminate half the array where the numbers are smaller
- Similarly, if the *key* is smaller, eliminate the half of the array where the numbers are larger
- Either way, we now have a new array to search through
- **Recursion: we can apply the same strategy to the new array**

Illustrative Example

Key: 364

$[1, 4, 7, 19, 37, 256, 364, 518, 999]$

Start index: 1, end index: 9

Mid-point: $\frac{1+9}{2} = 5$

Value at mid-point (index 5): 37

Is $37 > 364$?

No: run recursive binary search on the new array: $[256, 364, 518, 999]$

Building the Algorithm

- Let's consider the necessary inputs
- We need to know the array we're searching through: a
- We need to know the *key* we're looking for: k
- We need to know the start and end indices of the array: start, end
- We also need to consider some edge cases

```
binarySearch(int[] a, int key, int start, int end)
```

Edge Cases

- What if we provide an end index that is smaller than the start index?
  - The array has no elements, throw an error or return -1
- What if the mid-point is equal to our key?
  - If this happens on the first pass, we got lucky!
  - If this happens on subsequent passes, we have found our answer

Putting it all together

```
public static int binarySearch(int[] a, int key, int start, int end)
    int mp = (start + end) / 2;
    if(end < start) {
        return -1;
    }

    if(key==a[mp]) {
        return mp;
    } else if(key<a[mp]) {
        return binarySearch(a, start, mp - 1, key);
    } else {
        return binarySearch(a, mp + 1, end, key);
    }
}
```

The Recursive Call

```
public static int binarySearch(int[] a, int key) {
    return binarySearch(a, key, 0, a.length-1);
}
```

Note here we're using the start as zero and the end as the length of the array minus one

Efficiency of Recursive Binary Search

- In the evaluation of algorithms, we use big-O notation to compare algorithms
- Big-O gives us a sense of how the run time of an algorithm performs, depending on the size of the input
- Can measure the time complexity (worst-case run time) or the space complexity (memory usage)
- The time complexity of recursive binary search is $\mathcal{O}(\log N)$
- Standard linear search has time complexity $\mathcal{O}(N)$
- Since $\log N < N$, recursive binary search tends to be faster
- BUT linear search can be run on unsorted arrays
- Important to consider the trade-offs depending on the application you're building!

References