Recursion and Searching

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

Outline

- Recursion examples
- Linear search
- Binary search

Recursion Example 1

```
int sum(int arr[], int len) {
    int total = 0;
    for (int i = 0; i < len; i++) {
        total += arr[i];
    }
    return total;
}
```

Recursion Example 1

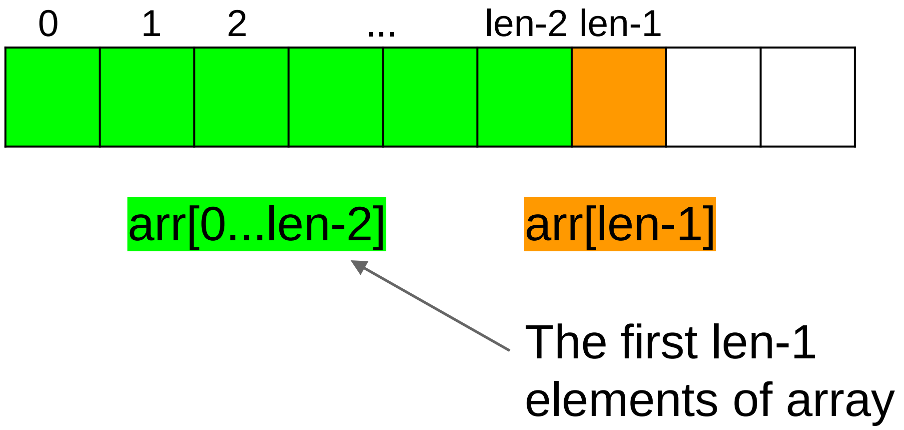
- Now, do it using recursion
- Steps:
  - 1) Base case: when array has length 1, just return the only element
  - 2) Assume your function can sum an array that has length len-1, and call the function inside itself
- New interpretation of “len”: number of elements you want to sum

Recursion Example 1

```
int sum(int arr[], int len) {
  // returns sum of first len elements of arr

  // base case
  if (len == 1) {
    return arr[0];
  }

  // recursion
  return arr[len-1] + sum(arr, len-1);
}
```

Recursion Example 1

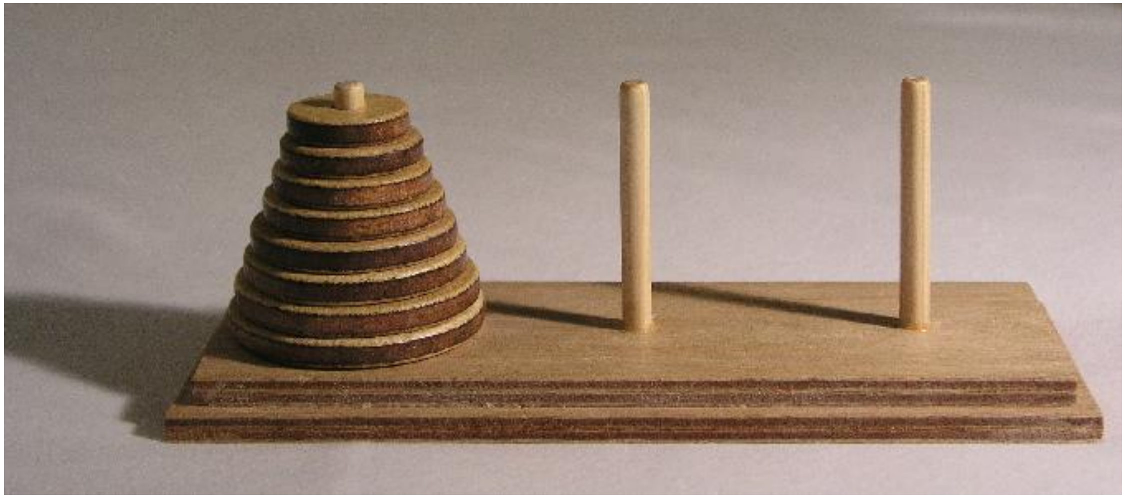
```
int sum(int arr[], int len) {
  // returns sum of first len elements of arr

  // base case
  if (len == 1) {
    return arr[0];
  }

  // recursion
  return arr[len-1] + sum(arr, len-1);
}
```



Returns sum of first len-1 elements of array

Recursion Example 2: Tower of Hanoi

- https://en.wikipedia.org/wiki/Tower_of_Hanoi



  - Move tower to the right slot
  - Move disks one by one
  - Bigger disks must **always** be below smaller disks

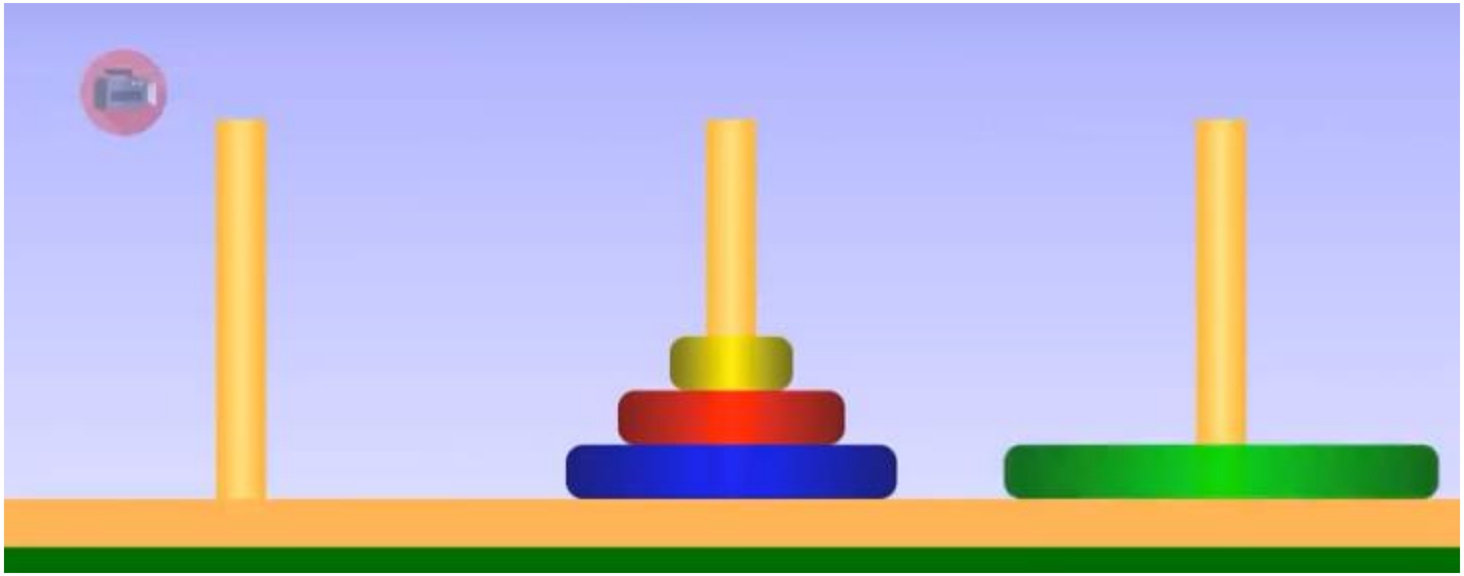
Recursion Example 2: Tower of Hanoi

Source: YouTube

- Move tower to the right slot
- Move disks one by one
- Bigger disks must **always** be below smaller disks

Recursion Example 2: Tower of Hanoi

- Recursive solution:
  a. Base case: If N is 1, then move the disk from A to C
  b. Otherwise:
     - Move (smallest) N-1 disks from A to B
     - Move (largest) 1 disk from A to C



Recursion Example 2: Tower of Hanoi

- Recursive solution:
  a. Base case: If N is 1, then move the disk from A to C
  b. Otherwise:
     - Move (smallest) N-1 disks from A to B
     - Move (largest) 1 disk from A to C
     - Move (smallest) N-1 disks from B to C

- Organization:
  a. At any time, A, B, and C may be labeled as “source”, “spare”, and “destination”

Recursion Example 2: Tower of Hanoi

- Recursive solution: **`solve_ToH(N, src, des)`**
  a. Base case: If N is 1, then move the disk from A to C
     - `move(A,C);`
  b. Otherwise:
     - Move (smallest) N-1 disks from A to B
       - `solve_ToH(N-1, A, B);`
     - Move (largest) 1 disk from A to C
       - `move(A,C);`
     - Move (smallest) N-1 disks from B to C
       - `solve_ToH(N-1, B, C);`

- Organization:
  a. At any time, A, B, and C may be labeled as “source”, “spare”, and “destination”

Searching Overview

- It is often useful to find out whether or not an array contains a particular item
  - E.g., “Is Alice among your Facebook friends?”
  - E.g., “Find Bob’s phone number.”
- Two possible specifications:
  - A search can either return true or false
  - OR . . . the position of the item in the array (-1 for fail)

Searching Variations

- There are many possible search algorithms
  - generally, want the one that finds the item the fastest
- Searching is one of those activities that can be done much more efficiently if the set is sorted ahead of time
  - Q. How does sorting make *your* searches easier?
- Best for unordered array is a *linear search*

Linear Search Algorithm

Strategy: Start with the first item and step through the array one element at a time, comparing each item with the target until either a match is found (return true / index) or all elements have been exhausted (return false / -1).

E.g., target = `"Saturn"`:

Neptune	Uranus	Saturn	Jupiter	Mars	Earth	Venus	Mercury

return true
or index=2

Q. What input results in the worst-case running time?

E.g., target = `"Mercury"`:

Neptune	Uranus	Saturn	Jupiter	Mars	Earth	Venus	Mercury

E.g., target = `"Pluto"`:

Neptune	Uranus	Saturn	Jupiter	Mars	Earth	Venus	Mercury

Linear Search in C

```
int LinearSearch(int arr[], int len, int target) {
```

- Repeat for all `i` from `0` to `len-1`
  - Check the next element, `arr[i]`
  - Algorithm:
    - found if equal to `target`, so return position
- Not found, so return fail

```
}
```

Linear Search in C

```
int LinearSearch(int arr[], int len, int target) {
      for (int i = 0; i < len; i++) {
      //  What's a good assertion?

          if (arr[i] == target) {
              return i;
          }
      }
   return -1;
}
```

Linear Search Analysis

Worst case for linear search is linear time $O(N)$

- Intuition: You have to check all elements to confidently return false.

Best case?

- You find the element at index 0

Q. What do you think is the average case?

Counting Comparisons

```
int LinearSearch(int arr[], int len, int target) {
        for (int i = 0; i < len; i++) {
                if (arr[i] == target) {
                        return i;
                }
        }
        return -1;
}
```

$N + 1$

N

Total Comparisons $= 2N + 1$

- Comparisons are *relatively* expensive elementary operations
- Use a *sentinel* to cut the comparisons in half
    - It's still $O(N)$, but with half the leading constant

Optimized Linear Search

```
int LinearSearch(int arr[], int len, int target) {
    arr[len] = target;
    int i = 0;
    while (arr[i] != target) {
            i++;
    }
    if (i != len) return i;
    return -1;
}
```

Sentinel value

Signals the end of the search

And yes, assignment to `arr[len]` is a side-effect that can have bad consequences

$N + 1$

1

Total Comparisons $= N + 2$

- Sentinel allows you to combine the element comparison and loop termination conditions

But is it really an improvement?

Big-*O* methods say that leading constants don't matter when comparing two algorithms

- they usually don't if the two algorithms have *different* Big-*O* running times
- E.g., $50000N + 300$ vs $2N^2 - 3N + 1$

But they *do* matter when their Big-*O* growth rates are the same

- E.g., optimized program vs unoptimized
- E.g., fast machine vs slow machine