

## 15-104 Introduction to Computing for Creative Practice

*Fall 2020*

### 33 Recursive Searching

Instructor: Tom Cortina, [tcortina@cs.cmu.edu](mailto:tcortina@cs.cmu.edu), GHC 4117, 412-268-3514

1




## Linear Search Revisited

- Recall that we could use iteration (loops) to implement a linear search on an array of data elements:

```
// search for element in array arr starting at index
function lin_search(arr, element, index) {
  if (index < 0 || index >= arr.length) return -1;
  var i = index;
  while (i < arr.length) {
    if (arr[i] == element) return i;
    i++;
  }
  return -1;
}
```

2




## Recursive Linear Search

Searching for element in array `arr` starting at `index`:

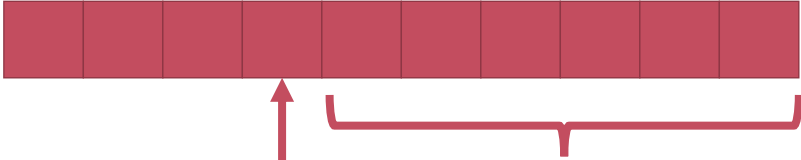
- Is `index` out of range (i.e. less than 0 or at least `arr.length`)?
  - If so, return what?
    - `-1` (can't possibly find it) **BASE CASE**
- Is `arr[index]` the element we're looking for?
  - If so, return what?
    - `index` (that's where it is) **BASE CASE**
- Otherwise, what do we do?
  - Return the result of a search for element in array `arr` starting at `index+1`. **RECURSIVE CASE**

3



## Another way to look at it

Searching for element in array `arr` starting at `index`:



`index`

If `arr[index]` is not the element, then the result of a search starting at `index+1` is also the answer to a search starting at `index`.

4



## Recursive Linear Search

```
// search for element in array arr starting at index,
// recursively
function lin_search(arr, element, index) {
  if (index < 0 || index >= arr.length) {
    return -1;
  }
  if (arr[index] == element) {
    return index;
  }
  return lin_search(arr, element, index+1);
}
```

5




## Trace

```
arr = [13, 83, 23, 90, 54, 47, 61]
```

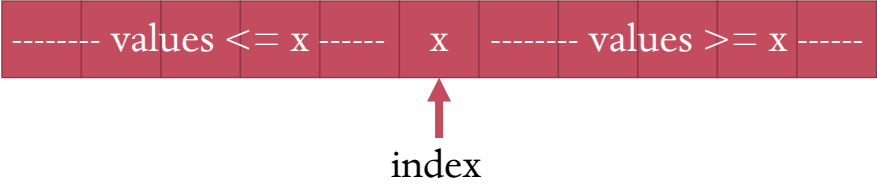
```
lin_search(arr, 90, 1)
  lin_search(arr, 90, 2)
    lin_search(arr, 90, 3)
      3
    3
  3
```

6




## A Faster Search: Binary Search

- What if we knew that the data in the array was sorted?
- We can take advantage of this fact to search much faster.
- General idea, look at the middle data value in the array:



If  $\text{arr}[\text{index}] = \text{element}$ , then we found it.  
 If  $\text{element} < \text{arr}[\text{index}]$ , then it could only be in the left half if it is there at all. So search that half the same way.  
 If  $\text{element} > \text{arr}[\text{index}]$ , then it could only be in the right half if it is there at all. So search that half the same way.

7



## A Faster Search: Binary Search

- To search for an element:
  - Find the index of the middle position of the array.
  - If the array has no elements, then return -1.
  - If the value at that index is the element we're looking for, then we're done. Return that index.
  - Otherwise:
    - If the element we're looking for is less than the value at the middle position, search the left half of the array recursively.
    - If the element we're looking for is greater than the value at the middle position, search the right half of the array recursively.

8



## Recursive Binary Search

```
function bin_search(arr, element, lo, hi) {
  // search for element in arr between indices lo and hi
  if (lo > hi) { return -1; } // not found
  var mid = int((lo + hi) / 2); // int in case even # of values
  if (arr[mid] == element) {
    return mid; // found
  } else if (arr[mid] > element) {
    // return the result of a search of the left "half"
    return bin_search(arr, element, lo, mid - 1);
  } else if (arr[mid] < element) {
    // return the result of a search of the right "half"
    return bin_search(arr, element, mid + 1, hi);
  }
}
```

To start a search: `bin_search(arr, element, 0, arr.length-1)`

9



## Trace

```
arr = [13, 23, 47, 54, 61, 83, 90] // sorted!
```

```
bin_search(arr, 61, 0, 6)
  lo = 0, hi = 6, mid = 3, arr[3] = 54 😞
  bin_search(arr, 61, 4, 6)
    lo = 4, hi = 6, mid = 5, arr[5] = 83 😞
    bin_search(arr, 61, 4, 4)
      lo = 4, hi = 4, mid = 4, arr[4] = 61 😊
    4
  4
4
```

10

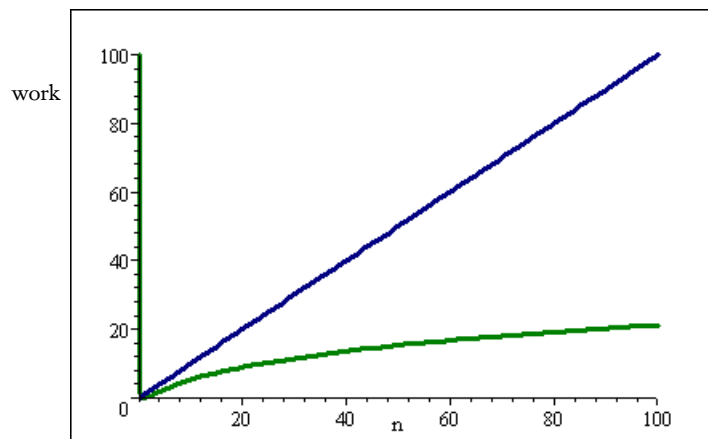


## Big O

- Binary Search is an  $O(\log n)$  algorithm.
- But it does require the data to be sorted before its use.  
(if the data needs to be sorted, this adds to the computational complexity of the search)
- If you have a sorted array with 1 million elements, you only need to examine about 20 elements!
  - Since  $1 \text{ million} \cong 2^{20}$ ,  $\log_2(2^{20}) = 20 \log_2(2) = 20$ .
- If you have a sorted array with 1 billion elements, you only need to examine about 30 elements! WOW!
  - Since  $1 \text{ billion} \cong 2^{30}$ ,  $\log_2(2^{30}) = 30 \log_2(2) = 30$ .

13

## Linear vs. Logarithmic



(source: Stack Overflow)

14