

Computer Science 210: Data Structures

Searching

Searching

- The problem: Given a sequence of elements, and a target element, find whether the target occurs in the sequence
- Variations:
 - find first occurrence
 - find all occurrences
 - find the number of occurrences, etc
- Searching is a fundamental problem
- For simplicity, let's assume we have an array of numbers
 - `double a[];`
 - `double target;`
- and we want to write a method
 - `//return the position of first occurrence or -1 if not found`
 - `int search (double a[], double target)`

Searching

//return the position of first occurrence or -1 if not found

int search (double a[], double target)

Searching

//return the position of first occurrence or -1 if not found

```
int search (double a[], double target) {  
    for (int i=0; i < a.length; i++)  
        if (a[i] == target) return i;  
    //if we got here, no element matched  
    return -1;  
}
```

linear search



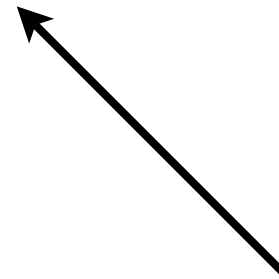
Searching

//return the position of first occurrence or -1 if not found

```
int search (double a[], double target) {  
    for (int i=0; i < a.length; i++)  
        if (a[i] == target) return i;  
    //if we got here, no element matched  
    return -1;  
}
```

- best-case (fastest) ?
- worst-case (slowest) ?

linear search



Searching

```
//return the position of first occurrence or -1 if not found
```

```
int search (double a[], double target) {  
    for (int i=0; i < a.length; i++)  
        if (a[i] == target) return i;  
    //if we got here, no element matched  
    return -1;  
}
```

linear search



- With linear search, in the worst case we have to examine the entire input
 - Can we do better? (that is, faster)?
 - Yes, if the input is sorted

Binary search

- Input: A target and a sequence of elements, sorted (in some order). For simplicity, we assume increasing (non-decreasing) order.

```
//return the position of occurrence or -1 if not found
```

```
//invariant: a is sorted in increasing order
```

```
int binarysearch (double a[], double target)
```

- Idea: searching in a phone book
 - open in the middle; if name comes before the "middle" name, search in the left half. if name comes after the middle name, search in the right half.
- Examples:
 - `double a[] = {1, 3, 4, 6, 7, 7, 9, 12, 14, 18, 56, 67, 89, 100};`
 - search for 6
 - search for 80

Binary Search

```
//return the position of occurrence or -1 if not found
```

```
//invariant: a is sorted in increasing order
```

```
int binarysearch (double a[], double target) {  
    int start, end, middle;  
    start = 0;  
    end = a.length-1;  
    while ...?... {  
        middle = (start + end)/2;  
        if (target == a[middle]) return middle;  
        if (target < a[middle]) end = middle-1;  
        if (target > a[middle]) start = middle +1;  
    }  
    //if we are here, not found  
    return -1;  
}
```


Binary Search

- Correctness

- Is it correct to throw away half of the input? Can you argue why?

- Analysis:

- at the first iteration through the loop, start and end delimit the entire array
- at the second iteration through the loop, start and end delimit one half of the array
- at the third iteration..... one quarter of the array
- at the fourth iteration..... one eighth of the array

- Notation: let n denote the size of the input array
- i^{th} iteration \implies a section of size $n/2^i$
- How many iterations can there be?

Logarithm review

Binary search

- Assume $n = 1,000,000$
 - How many elements does linear search compare?
 - How many elements does binary search compare?
- Intuitively, binary search is (much) more efficient than linear search
 - That is, in the worst case. We always think of the worst-case. Best-cases are irrelevant and offer no guarantees on the performance of an algorithm.
- We will analyze and compare them formally when we talk about algorithm analysis next week.

Recursive Binary Search

- It's easy to think of it recursively
- Searching in the first or second half are recursive problems
- We need to give the start and end to the recursive call

```
//invariant: a[] is sorted in increasing order
```

```
//return the position where target is found, or -1 if not found
```

```
int binarysearch (double a[], double target) {
```

```
    //this is the call to the recursive solver
```

```
    return binsearchRecursive(a, target, 0, a.length -1);
```

```
}
```

```
// invariant: a[] is sorted in increasing order
```

```
//search for target in a[start...end]; return the position where target is found, or -1 if not found
```

```
int binsearchRecursive(double a[], double target, int start, int end)
```

Binary Search

- It's easy to think of it recursively
- Searching in the first or second half are recursive problems
- We need to give the start and end to the recursive call

```
// invariant: a[] is sorted in increasing order
```

```
//search for target in a[start...end]; return the position where target is found, or -1 if not found
```

```
int binsearchRecursive(double a[], double target, int start, int end) {
```

```
//base case
```

```
if (start > end) return -1;
```

without base-case, infinite recursion



```
//otherwise
```

```
int middle = (start+end)/2; //note that it gets truncated
```

```
if (target == a[middle]) return middle;
```

```
if (target < a[middle]) return binsearchRecursive(a, target, start, middle -1);
```

```
return binSearchRecursive(a, target, middle+1, end);
```

```
}
```