CSE 142
Computer Programming I

Linear & Binary Search

Concepts This Lecture

Searching – what it means
Linear search
Binary search
Comparing algorithm performance

Searching

What is the name of the student with id 9933432? QUICK!

9932431 John Smith
9933431 Jay Smith
9935443 Jack Smythe
9793212 John Smythe
9328122 Jon Smith
9934431 Johan Schmitz
9923871 Jack Smith
9933432 Jon Smythe

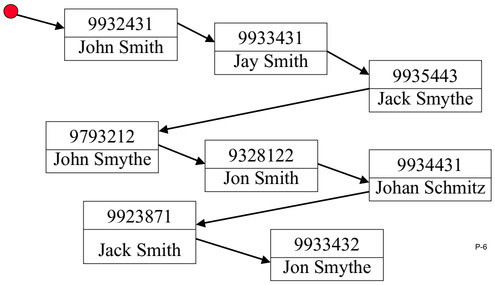
Searching

- Assume we have N items in some sort of collection
- Each item has a key field
- We're looking for a record matching a search key – it may or may not be in the collection
- "One operation" means comparing the search key with a single item's key (and possibly moving on to the next item in the collection)
- Goal: minimize the number of operations required to find an item (as a function of N, the collection size)

Data Structures

- Collections of items are organized into data structures that determine:
  - How expensive it is to get from item n to item n+1 (or from n to n+k)
  - The relationship of the key values among the items (e.g, "item n+1 is guaranteed to be larger than item n", or "there's no size relationship between n and n+1")
- Data structures is a gigantic topic that we'll touch on only a bit in 142; more in 143; more in all of Computer Science…

Example Data Structure: "Linked List"



Linked List: Example Implementation

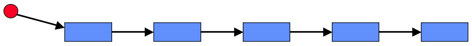
```
typedef struct {
      … // actual data
      int   nextElement;
} ElementType;

ElementType      linkedList[MAXELS];
int              listHead;
```

WARNING: This isn't how it's "really done" in most cases

0	9793212 John Smythe 2
1	9932431 John Smith 4
2	9328122 Jon Smith 18
3	9933431 Jay Smith 5
4	9935443 Jack Smythe 0
…	

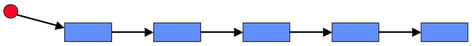
listHead
1

Linked List



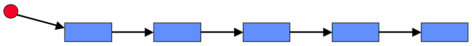
- N elements total
- Assume for now that the elements are in no particular order in the list (i.e., they're not sorted)
- "1 operation" is "compare for equality and move one element down the list if not equal"
- How long does it take to find the element matching the search key?

Linked List



- N elements total
- "1 operation" is "compare for equality and move one element down the list if not equal"
- How long does it take to find the element matching the search key?
  - We mean "how long in the worst case"
  - We state answer not in seconds or microseconds, but in terms proportionality to N; e.g.,
    - Proportional to N
    - Proportional to N^2
    - Proportional to Sqrt(N)

Linked List: Search Time



- If the elements are not sorted in the list, the search time is proportional to N
- Does it help to sort the elements?
  - No – in the worst case (the element is bigger than any element in the list) we still have to go all the way from the beginning to the end, which costs N
- Is there anything you can do about ordering the elements in the list that will reduce the (worst case) search time?

Example Data Structure II: Array

```
typedef struct {
      … // actual data
} ElementType;

ElementType      linkedList[MAXELS];
int              currentSize;
```

WARNING: This isn't how it's "really done" in most cases

0	9932431 John Smith
1	9933431 Jay Smith
2	9328122 Jon Smith
3	9935443 Jack Smythe
4	
…	

currentSize
4

Searching An Array: Linear Search

Q: If there are N elements stored in the array, how long does it take to perform the search for a specific key (worst case)?

A: If we search by first looking at element 0, then element 1, then element2, etc., it's just like a linked list, so time is proportional to N. (In fact, it is a linked list, but with "implicit pointers" from each element n to n+1 as its successor.)

Gee, that was boring…

Searching An Array: Can We Do Better?

Q: Suppose we know the array is sorted from smallest to largest element. How long does a search take?

A: Well, if there is no relationship among the key values stored in the array and their position in the array, no – proportional to N is the best we can do.

But, if the array is sorted…

Searching An Array: Can We Do Better?

Unsorted Array	Sorted Array
0	-19
22	0
-19	7
8	8
33	18
41	22
18	33
7	41

Searching A Sorted Array: Idea #1

Sorted Array
-19
0
7
8
18
22
33
41

Count By Two

- Compare to element 0
- Compare to element 2
- Compare to element 4
- …

Example: *Search key = 8*

Time is proportional to N/2

Hey, That Was Great!: Idea #2

Sorted Array
-19
0
7
8
18
22
33
41

Count By Five

- Compare to element 0
- Compare to element 2
- Compare to element 4
- …

Example: *Search key = 8*

Time is proportional to N/5

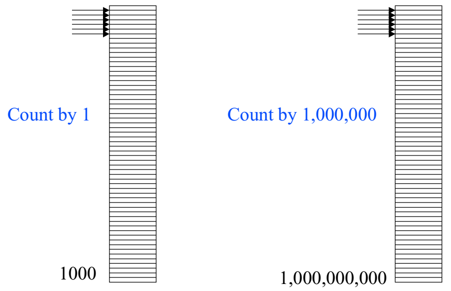
This is So Great!: The Problem

Q: If looking at every other element is faster than looking at elements one after another, and if…
Looking at every fifth element is faster still, …
Why not look at every millionth element (or every zillionth)?

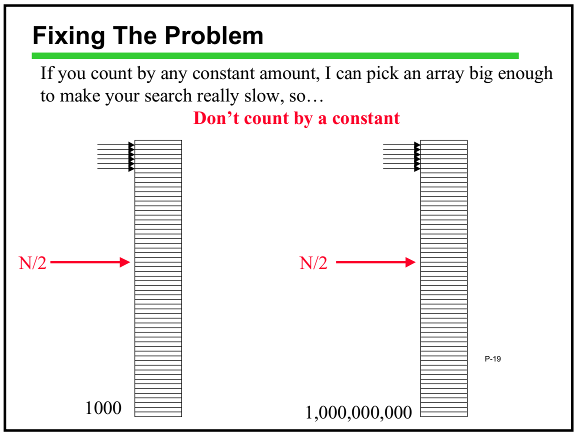
A: This will seem weird, but it's because it isn't really helping us very much

That is, N and N/1,000,000 are both really big numbers if N is big enough

This is So Great!: The Problem Visually



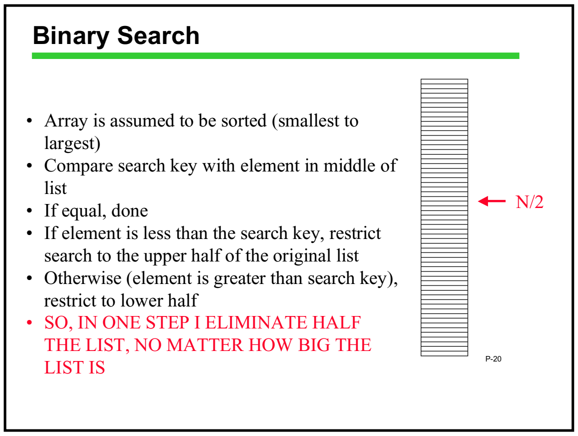
The Problem: You pick what you want to count by, and I'll pick my array big enough that your search will be really slow



Fixing The Problem

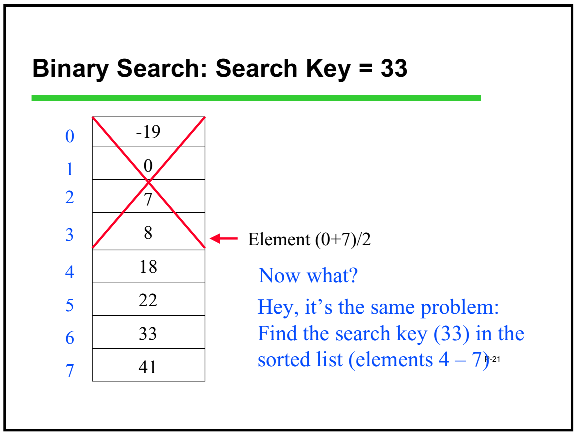
If you count by any constant amount, I can pick an array big enough to make your search really slow, so…

Don't count by a constant



Binary Search

- Array is assumed to be sorted (smallest to largest)
- Compare search key with element in middle of list
- If equal, done
- If element is less than the search key, restrict search to the upper half of the original list
- Otherwise (element is greater than search key), restrict to lower half
- SO, IN ONE STEP I ELIMINATE HALF THE LIST, NO MATTER HOW BIG THE LIST IS



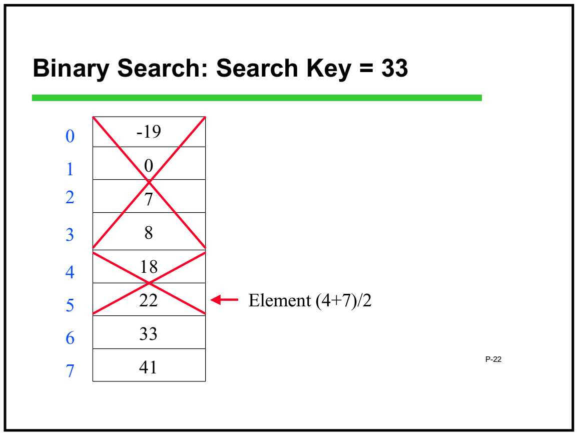
Binary Search: Search Key = 33

0	-19
1	0
2	7
3	8
4	18
5	22
6	33
7	41

Element (0+7)/2

Now what?

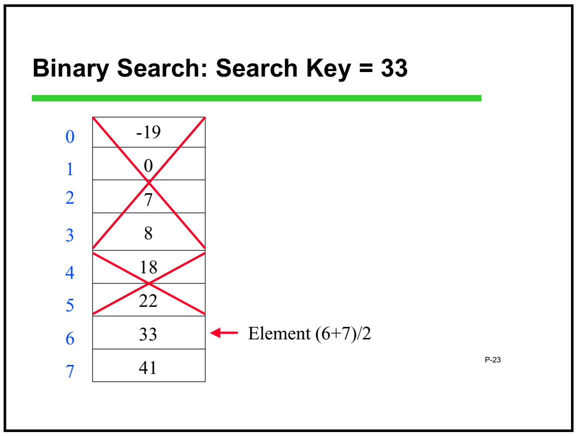
Hey, it's the same problem: Find the search key (33) in the sorted list (elements 4 – 7)



Binary Search: Search Key = 33

0	-19
1	0
2	7
3	8
4	18
5	22
6	33
7	41

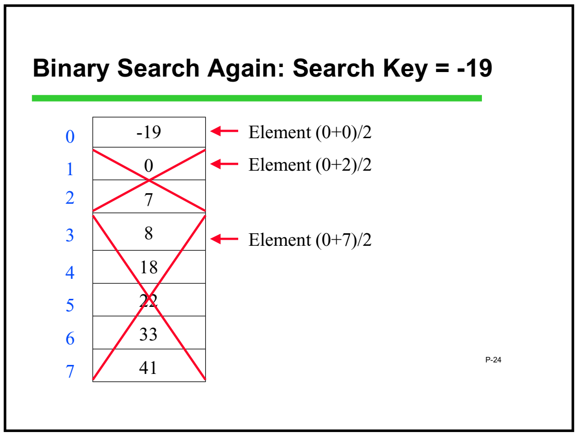
Element (4+7)/2



Binary Search: Search Key = 33

0	-19
1	0
2	7
3	8
4	18
5	22
6	33
7	41

Element (6+7)/2



Binary Search Again: Search Key = -19

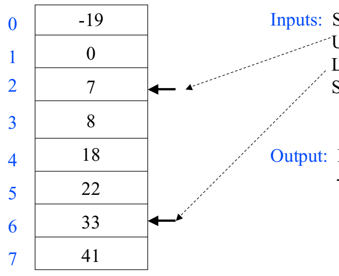
0	-19
1	0
2	7
3	8
4	18
5	22
6	33
7	41

Element (0+0)/2

Element (0+2)/2

Element (0+7)/2

Binary Search As a C Function



Inputs: Sorted array
Upper index
Lower index
Search key

Output: Index of element, or
-1 if not found

```
int BinarySearch (int array[], int lowerBound, int upperBound, int searchKey)
{
        int       midPoint;
        // this is the base case
        if (upperBound<lowerBound) {
                return -1;
        }

        midPoint = (lowerBound + upperBound)/2;
        if (array[midPoint] == searchKey) {
                return midPoint;
        } else if (array[midPoint] < searchKey) {
                return BinarySearch(array, midPoint+1, upperBound, searchKey);
        } else {
                return BinarySearch(array,lowerBound, midPoint-1, searchKey);
        }

}
```

Arguing That This Is Correct: Invariants

A program invariant is a (useful) property of the program that is "always true"

- Ideally, "always true" means just that, except for very brief moments when a variable's value is updated
- Can also mean "always true at the top of the loop" or "always true at the bottom of the loop" or just "always true at this particular point in the program"

Understanding your program's invariants can be a GIGANTIC help in writing error-free code.

Arguing That This Is Correct: Invariants

What is the important invariant in
int BinarySearch (int array[], int lowerBound, int upperBound, int searchKey)

If the searchKey is in the array, it's located in the range
of elements [lowerBound, upperBound] (inclusive)

Is that true? I.e., is our claim about the invariant correct?

- Show that it's true at the beginning
- Show that each step we take keeps it true

Arguing That This Is Correct: Invariants

If the searchKey is in the array, it's located in the range
of elements [lowerBound, upperBound] (inclusive)

Show that it's true at the beginning:
BinarySearch(array, 0, MAX, searchKey)

Show that each step we take keeps it true:
Assume it's true now: BinarySearch(array, lb, ub, searchKey)

Because the array is sorted, if array(midPoint)<searchKey, searchKey must in an element greater than midPoint ⇒
BinarySearch(array, midPoint+1, ub, searchKey)

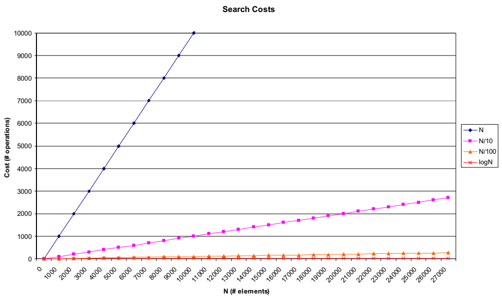
Time: How Many Comparisons Are Needed?

Key observation: for binary search: size of the array N that can be searched with k comparisons: $N \sim 2^k$

Number of comparisons k as a function of array size N: $k \sim \log_2 N$

This is fundamentally faster than linear search (where $k \sim N$) ⇒

$\log_2 N$ is much, much smaller than N for big enough N



Summary

- Linear search and binary search are two different algorithms for searching an array
- Binary search is vastly more efficient
  - But binary search works only if the array elements are sorted
- Looking ahead: we will study how to sort arrays, that is, place their elements in order