BINARY SEARCH

MSCI 240: Algorithms & Data Structures

lab 3 (program complexity)

- lab 3 next week (week of Oct 22)
- lab 4 the week after that (week of Oct 29)
- resume every other week (lab 5, week of Nov 12, etc.)

lecture/tutorial swap (Nov 5 + 7)

- Mon, Nov 5: two lectures—normal lecture + lecture in tutorial time
- Wed, Nov 7: tutorial in lecture time

lecture summary

logarithmic complexity

binary search

clicker questions

Topic	Building Java Programs	Algorithms (Sedgewick)
classes, ADTs	chapter 8	1.2
arrays	chapter 7	
ArrayList<T>	chapter 10	1.3
Stack/Queue	chapter 14, (11)	1.3
LinkedList	chapter 16	1.3
Complexity	chapter 13	1.4
Searching		pp. 46-47
Sorting		chapter 2.1-2.3
Recursion	chapter 12	1.1 (p. 25)
BSTs	chapter 17	chapter 3.1-3.2
Dictionaries	chapter 18.1	chapter 3.4
Graphs	N/A (Wikipedia good)	chapter 4.1
Heaps/Priority Queues	chapter 18.2	chapter 2.4

what is a logarithm?

$$x^y = z$$
$$\log_x z = y$$

$$n = 2^k$$
$$T(n) \sim C \cdot k = C \cdot \log_2 n$$

$n = 2^k$	k	$T(n)$
2	1	0.1s
4	2	0.2s
8	3	0.3s
16	4	0.4s
32	5	0.5s
64	6	0.6s
128	7	0.7s
256	8	0.8s
512	9	0.9s
1024	10	1.0s

logarithmic complexity: whenever the input size doubles, the running time increases by a constant amount how could an algorithm take logarithmic time?

binary search

recall: linear search

input:

n numbers, $A = < a_1, a_2, \ldots, a_n >$

a value, v

output:

index i, such that $v = a_i$

or, nil if $v \notin A$

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

i

input:

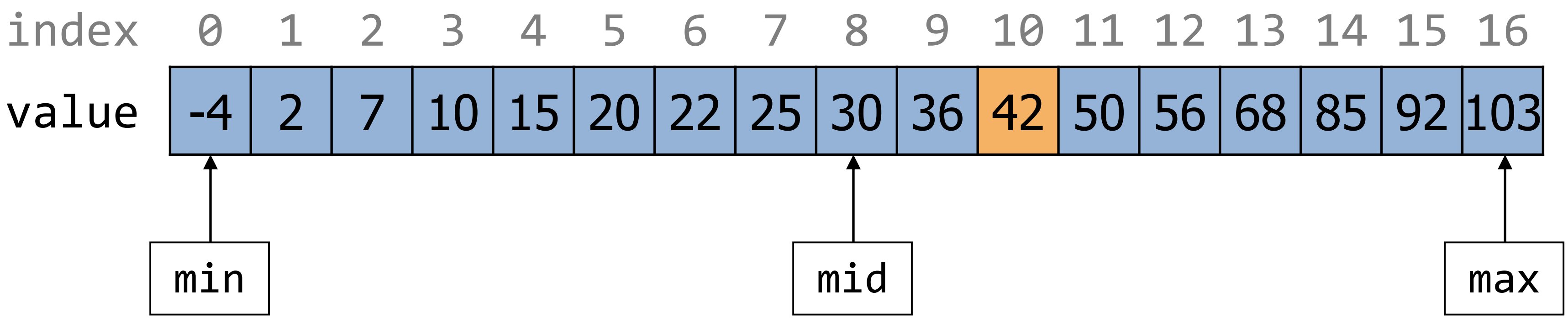
n sorted numbers, $A = < a_1, a_2, \ldots, a_n >$

a value, v (e.g., 42 above)

output:

index i, such that $v = a_i$ (e.g., 10 above)

or, nil if $v \notin A$



binary search: locates target value in a sorted array/list by successively eliminating half of the array from consideration

example: phonebook

how many steps does it take to complete?

(a.k.a. how many elements does it need to examine?)

binary search algorithm

examine the middle element of the array

- if it is too big, eliminate the right half of the array and repeat
- if it is too small, eliminate the left half of the array and repeat
- else it is the value we're searching for, so stop

binary search (pseudocode)

```
binarySearch(int[] a, int target) {

    while (a has remaining elements) {
        int mid = middle element index;
        if (a[mid] < target) {
            eliminate left half;
        } else if (a[mid] > target) {
            eliminate right half;
        } else {
            return mid; // target found
        }
    }
    return nil; // target not found
}
```

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103
	min								mid								max

```java
public static int binarySearch(int[] a, int target) {
    int min = 0;  int max = a.length - 1;
    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            min = mid + 1;
        } else if (a[mid] > target) {
            max = mid - 1;
        } else {
            return mid; // target found
        }
    }
    return -(min + 1); // target not found
}
```

binary search (code)

```java
public static int binarySearch(int[] a, int target) {
    int min = 0;  int max = a.length - 1;
    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            min = mid + 1;
        } else if (a[mid] > target) {
            max = mid - 1;
        } else {
            return mid; // target found
        }
    }
    return -(min + 1); // target not found
}
```

recall: class `Arrays` in package `java.util` has useful static methods for manipulating arrays:

syntax: `Arrays.`*`methodName`*`(parameters)`

Method name	Description
`binarySearch(`**array**`, `**value**`)`	returns the index of the given value in a *sorted* array (or < 0 if not found)
`binarySearch(`**array**`, `**minIndex**`, `**maxIndex**`, `**value**`)`	returns index of given value in a *sorted* array between indexes *min* / *max* - 1 (< 0 if not found)
`copyOf(`**array**`, `**length**`)`	returns a new copy of an array
`equals(`**array1**`, `**array2**`)`	returns `true` if the two arrays contain the same elements in the same order
`fill(`**array**`, `**value**`)`	sets every element to the given value
`sort(`**array**`)`	arranges the elements into sorted order
`toString(`**array**`)`	returns a string representing the array, such as `"[10, 30, -25, 17]"`

```java
// searches an entire sorted array for a given value
// returns its index if found;  a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, value)

// searches given portion of a sorted array for a given value
// examines minIndex (inclusive) through maxIndex (exclusive)
// returns its index if found;  a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, minIndex, maxIndex, value)
```

- the `binarySearch` method in the `Arrays` class searches an array very efficiently if the array is sorted
  - you can search the entire array, or just a range of indexes (useful for “unfilled” arrays such as the one in `ArrayListOfDouble`)
  - if the array is not sorted, you may need to sort it first

```java
// index    0  1  2  3   4   5   6   7   8   9  10  11  12  13  14  15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};
int index  = Arrays.binarySearch(a, 42);   // index1 is 10
int index2 = Arrays.binarySearch(a, 21);   // index2 is -7
```

`binarySearch` returns the index where the value is found

if the value is not found, `binarySearch` returns:

```java
-(insertionPoint + 1)
```

where `insertionPoint` is the index where the element would have been, if it had been in the array in sorted order

to insert the value into the array, negate `insertionPoint + 1`

```java
int indexToInsert21 = -(index2 + 1);  // 6
```

which indexes does the algorithm examine to find value `22`?

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

A. 0, 1, 2, 3, 4, 5, 6

B. 8, 12, 10

C. 8, 4, 6

D. 0, 16, 7, 4, 6

E. 8, 3, 5, 6

what is the runtime complexity of binary search?

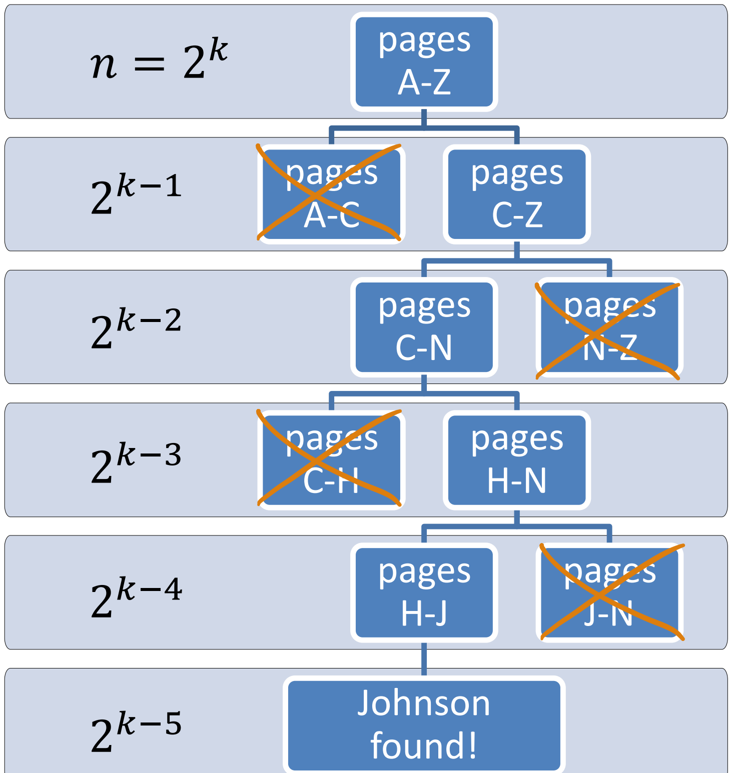
best case?

found at middle element (1 step)

worst case?

not found (? steps)

binary search (phonebook: “Johnson”)



if not found, how many steps (rows)?

k steps would be required

left with only $2^{k-k} = 1$ page

remember: $n = 2^k$, so $k = \log_2 n$

$\therefore \log_2 n$ steps would be required

# pages	# tests	$T(n)$?
2	1	0.1s
4	2	0.2s
8	3	0.3s
16	4	0.4s
32	5	0.5s
64	6	0.6s
128	7	0.7s
256	8	0.8s
512	9	0.9s
1024	10	1.0s

for an array of size n, it eliminates half until one element remains

$$n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \dots, 4, 2, 1$$

how many divisions does it take?

think of it from the other direction:

how many times do I have to multiply by 2 to reach n?

$$1, 2, 4, 8, \dots, \frac{n}{4}, \frac{n}{2}, n$$

call this number of multiplications k

$$2^k = n$$

$$k = \log_2 n$$

if you only have to search once through an unsorted array, which is better: linear or binary search?

what is the order of growth of binary search on a linked list?

does data ever actually get thrown away/deleted?

binary search and objects

can we `binarySearch` an array of `String`s?

- operators like < and > do not work with `String` objects
- but we do think of strings as having an alphabetical ordering

natural ordering: rules governing the relative placement of all values of a given type

comparison function: code that, when given two values a and b of a given type, decides their relative ordering:

$a < b, \quad a == b, \quad a > b$

the `compareTo` method (10.2)

the standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method

- example: in the `String` class, there is a method:

```java
public int compareTo(String other)
```

a call of `a.compareTo(b)` will return:

- a value < 0 if `a` comes "before" `b` in the ordering,
- a value > 0 if `a` comes "after" `b` in the ordering,
- or 0 if `a` and `b` are considered "equal" in the ordering

binary search summary

divides problem in half until trivial to solve

because it is ignoring these halves, requires at most $\log_2 n$ steps

requires a sorted array (sorting is expensive—will discuss later)

use `Arrays.binarySearch` in Java to search a sorted array

need to define `compareTo` instance method for objects to be able to use the `Arrays.binarySearch` method

clicker questions

You have an unsorted array $\boldsymbol{A}$. You want to know if it contains a value $\boldsymbol{v}$. Should you use binary or linear search to determine if $\boldsymbol{A}$ contains $\boldsymbol{v}$? You want the result as fast as possible.

A. binary search
B. linear search
C. neither will work

You have an unsorted array $\boldsymbol{A}$ that will not change. You need to test 10 billion values for their presence or absence from $\boldsymbol{A}$. Should you use binary or linear search to determine if $\boldsymbol{A}$ contains each of these values? You want the results as fast as possible.

A. binary search
B. linear search
C. depends on the size of A, but most likely binary search
D. depends on the size of A, but most likely linear search
E. neither will work

next:

big oh