CSCI 136
Data Structures & Advanced Programming
(Lecture 9)

Search

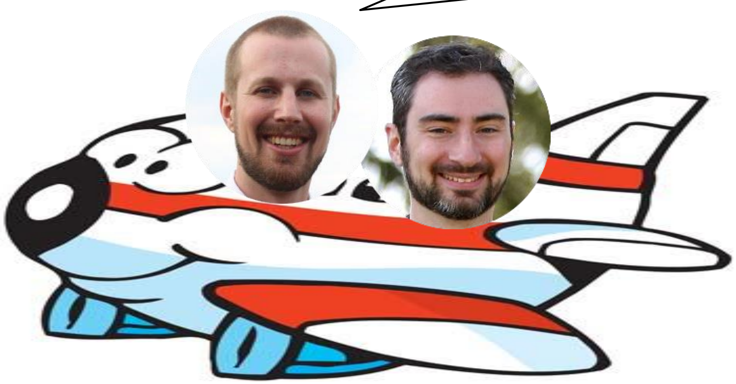
Jon Park

Feb 24, 2017

Announcements

- Lab 3 due on Monday!
- A message from Morgan and Bill:

"OHs will be canceled next week!"



Last Time

- Asymtotic analysis (Big-O)
- Recursion

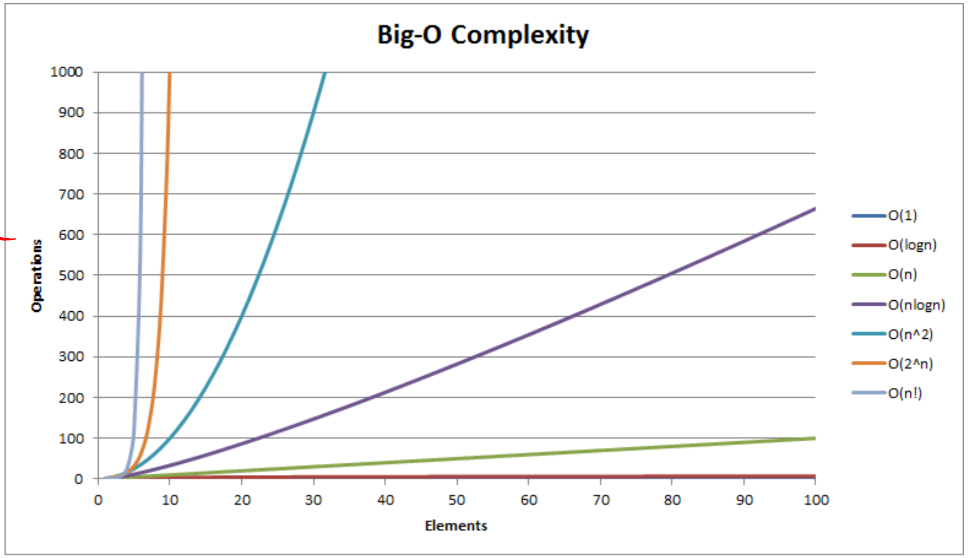
Today’s Outline

- Review
  - Asymtotic Analysis (Big-O)
- Search
  - Linear Search
  - Binary Search
- Defining Sortable Classes
  - Comparable
  - Comparator

Asymtotic Analysis (Big-O)

- “How scalable is the algorithm?”
- Commonly split into the following *classes*:
  - O(1) : “constant”
  - O(log n) : “logarithmic” or “log n”
  - O(n) : “linear”
  - O(n log n) : “n log n”
  - $O(n^c)$: “polynomial”
    - $O(n^2)$: “quadratic”
    - $O(n^3)$: “cubic”
  - $O(c^n)$: “exponential”

bad



Today's Outline

- Review
  - Asymtotic Analysis (Big-O)
- Search
  - Linear Search
  - Binary Search
- Defining Sortable Classes
  - Comparable
  - Comparator

Search

- What is search?
  - Locating an element in data
- Examples:

- documents
- phone #
- file

⋮

Searching for a Card

Time Complexity:

A. O(1) ← Best case

B. O(log n) $O(\frac{n}{2}) = O(n)$

C. O(n) ← worst case & ave.

D. O(n log n)

E. Not sure

Linear Search

```
public boolean linearSearch (int[] data, int num) {
    for (int i = 0; i < data.length; i++) {
        if (num == data[i])
            return true;        O(1)
    }                                          n   ⇒ O(n)
    return false;
}
```

Linear Search Summary

- Precondition:
  - None (The data can be in any order)
- Time complexity:
  - Best case:
    - O(1)
  - Worst case:
    - O(n)
  - Average case
    - O(n)

Searching for a Card (Again!)

Time Complexity:

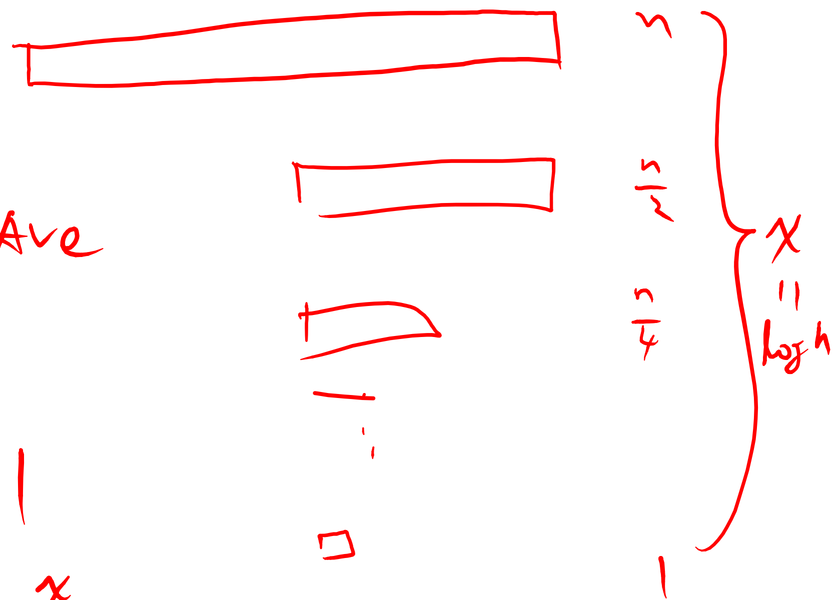
A. O(1) ← best case

B. O(log n) ← worst case + Ave

C. O(n)

D. O(n log n)

E. Not sure



$$\frac{n}{2^x} = 1$$

$$n = 2^x$$

$$\log_2 n = x$$

Binary Search

```
public boolean binarySearch (int[] data, int num) {
    int low = 0;
    int high = data.length - 1;
    while (high >= low) {
        int mid = (low + high)/2;
        if (data[mid]==num)
            return true;

        if (data[mid]< num)        ] check upper half
            low = middle + 1;

        else // data[mid] > num    ] check lower half
            high = middle - 1;
    }
    return false;
}
```

Binary Search Summary

- Precondition:
  - The data is sorted
- Time complexity:
  - Best case:
    - O(1)
  - Average case
    - O(log n)
  - Worst case:
    - O(log n)

Search Time Complexity

Algorithm	Best	Worst	Ave
Linear	$O(1)$	$O(n)$	$O(n)$
Binary	$O(1)$	$O(\log n)$	$O(\log n)$

- Caveat: Binary search only works on sorted data.
- Some classes are sortable: Integer, String, …
- But how do we define new sortable classes?

Today’s Outline

- Review
  - Asymtotic Analysis (Big-O)
- Search
  - Linear Search
  - Binary Search
- Defining Sortable Classes
  - Comparable
  - Comparator

2 Types of Sortable Classes

- One "obvious" way to compare/sort
  - Examples: Integer
  - Make the given class **Comparable** (=implement **Comparable**)<E>

- Multiple ways to compare/sort
  - Examples: PatientRecord
  - Make **Comparator**<E> classes
    - E.g. nameComparator, idComparator

Today's Outline

- Review
  - Asymtotic Analysis (Big-O)
- Search
  - Linear Search
  - Binary Search
- Defining Sortable Classes
  - Comparable
  - Comparator

Comparable<E> Interface

```
public interface Comparable<T> {
   public int compareTo(T o);
}
```

Comparable<E> Interface

- “Class X implement Comparable<X>”

  → “X contains **compareTo(X obj)** method”

  → “X can compare objects of class X”

- To compare object A to object B (both of type X), call A.compareTo(B), which returns an int:
  - Negative when A is “<” in rank
  - Zero A and B are “=” in rank
  - Positive when A is “>” in rank

Example: *Integer*

```
public class Integer … {
   ...

}
```

Example: *Integer*

```
public class Integer … implements Comparable<Integer> {
   ...
   // From Java6 Integer implementation (renamed variables)
   public int compareTo(Integer anotherInteger) {
        int x = this.value;
        int y = anotherInteger.value;
        return (x < y ? -1 : (x == y ? 0 : 1));
   }
}
```

ternary operator

Condition ? ____ : ____ ;
if true   if false

```
if(x < y) {
    return -1;
}
else {
    if (x==y)
        return 0;
    else
        return 1;
}
```

Exercise: *Integer* class

Integer A = new Integer(1);

Integer B = new Integer(2);

int val = A.compareTo(B);

Given the lines above, which of the following is true?

A. val < 0

B. val = 0

C. val > 0

D. Not sure

Today's Outline

- Review
  - Asymtotic Analysis (Big-O)
- Search
  - Linear Search
  - Binary Search
- Defining Sortable Classes
  - Comparable
  - Comparator

Comparator<E> Interface

```
public interface Comparator<T> {
    …
    int compare(T o1, T o2);
    …
}
```

Comparator<E> Interface

- “Class X implements Comparator<Y>”

  ➔ “X contains **compare(Y obj1, Y obj2)** method.”

  ➔ “X can compare objects of class Y”
- To compare object A to object B (both of type Y), create an object C (of type X) and call C.compare(A, B), which returns an int:
  - Negative when A is “<” in rank
  - Zero A and B are “=” in rank
  - Positive when A is “>” in rank

Example: *Comparator*

Patient.java

```
class Patient {
    protected int age;
    protected String name;
    public Patient (String s, int a) {name = s; age = a;}
    public String getName() { return name; }
    public int getAge() {return age;}
}
```

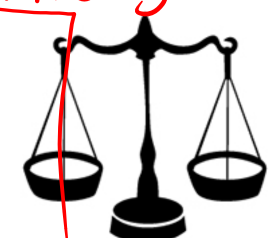
AgeComparator.java

```
class AgeComparator implements Comparator<Patient>{
     public int compare(Patient a, Patient b) {
          return a.getAge() - b.getAge();
     }
}
```

NameComparator.java

```
class NameComparator implements Comparator<Patient>{
     public int compare(Patient a, Patient b) {
          return a.getName().compareTo(b.getName());
     }
}
```

defined in String class



Exercise: *Patient* class

Patient A = new Patient("Deepak", 27);

Patient B = new Patient("Jenny", 52);

Given the lines above, how would you figure out the order between A and B?

// Name

NameComparator c = new NameComparator();

c.compare(A, B);

2 Types of Sortable Classes

- Classes with 1 “obvious” way to compare/sort (a) vs Classes with multiple ways to compare/sort (b)

Characteristic	a	b
Implements Comparable<E> (i.e. contains compareTo(E otherObj))	O	X
Need comparator classes containing compare(E obj1, E obj2)	X	O
The class itself supports comparison	O	X
Can be compared/sorted in multiple ways	X	O