[TAP:xxxxx] Search

- Which of the following is the efficient way to search for an element in an unsorted list?
  A. Perform linear search
  B. Perform binary search
  C. Sort the list, then perform linear search
  D. Sort the list, then perform binary search
  E. Whatever

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

- Defining Sortable Classes
  - Comparable
  - Comparator
- Sort
  - Bubble Sort
  - Selection Sort

2 Types of Sortable Classes

- One “obvious” way to compare/sort
  - Examples: Integer
  - Make the given class **Comparable** (=implement **Comparable**)

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

Today's Outline

- Defining Sortable Classes
  - Comparable
  - Comparator
- Sort
  - Bubble Sort
  - Selection Sort

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:
  - -1 Negative when A is "<" in rank
  - 0 Zero A and B are "=" in rank
  - 1 Positive when A is ">" in rank

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));
    }
}
```

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

"return x-y;"

[TAP] *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

- Defining Sortable Classes
  - Comparable
  - Comparator
- Sort
  - Bubble Sort
  - Selection Sort

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

the same as compareTo()

Example: *Comparator*

```
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;}
}

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

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

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?

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

Today's Outline

- Defining Sortable Classes
  - Comparable
  - Comparator
- Sort
  - Bubble Sort
  - Selection Sort

Sorting a Deck of Cards

- Come up with your own algorithm! (and let me know when one of the algorithms presented today is exactly like yours. ;))
- Hint: If you're stuck, think of it this way:
  - After *1st* iteration, at least *1* item is sorted.
  - After *ith* iteration, at least *i* items are sorted.
  - After *nth* iteration, all items are sorted. Done!
  - What needs to happen during each iteration?

Sorting a Deck of Cards

Time Complexity:

A. O(n) ← best

B. O(n log n)

C. O(n^2) ← Ave, worst

D. O(n^3)

E. Not sure

Bubble Sort

- [5 1 3 2 9]
- First Pass:
  - [5 1 3 2 9]
  - [1 5 3 2 9]
  - [1 3 5 2 9]
  - [1 3 2 5 9]
- Second Pass:
  - [1 3 2 5 9]
  - [1 3 2 5 9]
  - [1 2 3 5 9]
  - [1 2 3 5 9]
- Third Pass:
  - [1 2 3 5 9]
  - [1 2 3 5 9]
  - [1 2 3 5 9]
  - [1 2 3 5 9]

Bubble Sort

```
public static void bubbleSort(int[] data) {
    for (int curN = data.length-1; curN > 0; curN--) {
        boolean swapped = false;
        for (int i = 1; i <= curN; i++) {
            if (data[i-1] > data[i]) {
                swap(data, i, i-1);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Best

$O(n^2)$

$O(n)$

Bubble Sort Summary

- Overview
  - After *ith* iteration, at least *i* items are sorted.
  - During *ith* iteration, sweep through the unsorted portion of the list, swapping 2 adjacent elements if the right one is smaller.

    (End after iteration *i* if no swapping happens!)
- Time complexity:
  - Best case: O(n)
  - Worst case: $O(n^2)$
  - Average case: $O(n^2)$

Today’s Outline

- Defining Sortable Classes
  - Comparable
  - Comparator
- Sort
  - Bubble Sort
  - Selection Sort

Sorting a Deck of Cards

Time Complexity:

A. $O(n)$

B. $O(n \log n)$

C. $O(n^2)$ ← worst, ave, best

D. $O(n^3)$

E. Not sure

Selection Sort

- [11 3 27 5 16]
- [11 3 16 5 27]
- [11 3 5 16 27]
- [5 3 11 16 27]
- [3 5 11 16 27]

Selection Sort

```
public static void selectionSort(int[] data){
    for (int curN = data.length - 1; curN > 0; curN--) {
      int maxIdx = 0;
      for (int i = 1; i <= curN; i++){
        if (data[i] > data[maxIdx])
            maxIdx = i;
      }
      swap(data, maxIdx, curN);
    }
}
```

Selection Sort Summary

- Overview
  - After *ith* iteration, at least *i* items are sorted.

    ➔ the list is sorted at least after *n* iterations.
  - During *ith* iteration, **select** the max item in the unsorted portion of the list and move it to right-most location of the unsorted portion.
- Time complexity:
  - Best case: $O(n^2)$
  - Worst case: $O(n^2)$
  - Average case: $O(n^2)$