# Chapter 13, Sorting, Searching, and Shuffling

John M. Morrison

March 29, 2019

## Contents

# 0 Introduction

Two activities that occur a great deal on computers are those of sorting and searching. Sorting makes it easier to locate items in a collection; sorting makes searching far easier.

Searching is an activity that modern computers do with great speed. We often need to find a piece of data in a large collection. We can optimize this by sorting the data first if we know we are going to do a large number of retrievals.

Numerous algorithms exist for sorting. Also, Java has support for sorting. You will learn about sorting and searching algorithms in this chapter. We will use both Python and Java for implementing them. First we have to ask: *What do we mean by "sorting?"*

# 1 Ordering Data

To describe relationships among data we will avail ourselves of the notion of a set. We proceed naïvely; we will think of a set as any well-defined collection of objects. Examples include such things as phone book entries, prescription orders, or books recorded by title, author and ISBN.

If $S$ is a set, then a *relation* $R$ on $S$ is a set of ordered pairs of elements of $S$. If for a relation $R$ on $S$ we have $(x, y) \in R$, we will write $xRy$. Here are some basic properties relations can have.

The terms you see here are used by an international convention amongst computer scientists and mathematicians. They will help us to convey ideas in a succinct and precise manner. Here we show some properties of relations.

- A relation $R$ on a set $S$ is *reflexive* if for any $x \in S$, $xRx$.
- A relation $R$ on a set $S$ is *symmetric* if for any $x, y \in S$, if $xRy$ then $yRx$.
- A relation $R$ on a set $S$ is *antisymmetric* if for any $x, y \in S$, if $xRy$ and $yRx$ can only occur if $x = y$.
- A relation $R$ on a set $S$ is *transitive* if for any $x, y, z \in S$, if $xRy$ and $yRz$, then $yRz$.

An relation that is reflexive and transitive is called a *preorder*. An antisymmetric preorder is called a *partial order* If a preorder $R$ satisfies $xRy$ or $yRx$ for all $x, y \in S$, (i.e. any pair of elements is related) then it is called a *linear* or *total* ordering.

A relation that is reflexive, transitive and symmetric is called an *equivalence relation*. The standard design contract calls for an `equals` method in a Java class to be an equivalence relation.

Now let us come out of the clouds and look at some examples. Suppose we let $S$ be all US cities and we define $d(x, y)$ to be the distance from city $x$ to city $y$ in statute miles. Put

$$R = \{x, y | d(x, y) < 100\}.$$

Suppose you own an electric car that can go 100 miles; in this case, $R$ is the set of all city pairs you can drive between in your electric car.

Is this relation reflexive? Yes, because a city is 0 miles from itself. It is symmetric, since the distance from $x$ to $y$ is the same as the distance from $y$ to $x$.

It is not transitive. From Davenport, IA to Iowa City, IA the distance is 58 miles. From Iowa City, IA to Newton, IA the distance is 84 mi. The distance from Davenport to Newton is 142 mi. This breaks transitivity.

Now let $S$ be the set of all strings and let us say that $xRy$ if $x$ and $y$ are strings of the same length. This is an equivalence relation: it is reflexive, transitive, and symmetric.

Here is another important relation on strings. Let $xRy$ if $x$ precedes $y$ asciicographically. This is a linear ordering on strings. Similarly, case-insensitive comparsison of strings is also an equivalence.

Let $S$ be a set of all instances of a given class $C$. The design contract for the `equals` method stipulates that the relation $xRy$ if $x$.`equals`$(y)$ evaluates to `true` is an equivalence relation.

In both Python and Java, many types come with a "natural ordering." Java primitive numerical types and Python numerical types are linearly ordered by $\leq$, which in both cases, is implemented by `<=`. For Java, ordering of object types is achieved if they implement the interface `Comparable<T>` which looks like this.

```
@FunctionalInterface
public interface Comparable<T>
{
    public int compareTo(T t);
}
```

This interface is a functional interface specifying one method, `compareTo(T other)`. This provides a "natural" means of comparing objects that are instances of an implementing class. Python can impose order on objects via the hooks `__lt__`, `__gt__`, `__let__`, and `__ge__` ; implementing these defines behavior for the relational operators `<`, `>`, `<=`, and `>=`.

Here are some of the finer points of implementing the design contract. Suppose `C` is a class implementing `Comparable<T>`. Then we can impose an ordering on instances of `C` as follows. We say that `x`$R$`y` if `x.compareTo(y) <= 0`. The

design contract says that $R$ must be a linear preording on the instances of `C`. It also says that `x.compareTo(y)` and `y.compareTo(x)` must have opposite sign. In particular, if `x.equals(y)` then `y.comapreTo(x)` and `x.compareTo(y)` are both 0.

## 2 Sorting with Comparables

Let us now meet the class `Collections`. You will notice that this is a static service class. It is marked `final`, its constructor is private, and all of its methods are static. Let us look at its `sort` method. Here the method detail taken from `https://docs.oracle.com/javase/8/docs/api/`. Note that this does an in-place sort. This method changes the state of the list, hence the requirement that the list be of a mutable type.

```
public static <T extends Comparable<? super T>>
       void sort(List<T> list)}
```

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the Comparable interface. Furthermore, all elements in the list must be mutually comparable (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements e1 and e2 in the list). This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort.

The specified list must be modifiable, but need not be resizable.

Sorting a list using this method sorts the list according to the linear preording that is imposed by the `compareTo()` method.

To demonstrate this, consider the following program

```
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
public class CompareDemo
{
    public static void main(String[] args)
    {
        List<String> s = new ArrayList<>();
        List<Integer> t = new ArrayList<>();
        for(int k = 0; k <= 50; k++)
        {
            s.add("" + k);
            t.add(k);
        }
```

```
        Collections.shuffle(s);
        Collections.shuffle(t);
        System.out.println(s);
        System.out.println("");
        System.out.println(t);
    }

}
```

Now run this and see the shuffled lists.

```
$ java CompareDemo.java
$ java CompareDemo
[39, 12, 8, 38, 1, 16, 48, 40, 43, 18, 47, 50, 22, 6, 33,
15, 32, 37, 42, 35, 36, 2, 30, 34, 17, 23, 3, 0, 26, 31,
19, 44, 27, 29, 24, 49, 46, 7, 13, 45, 25, 28, 11, 41, 5,
14, 9, 21, 10, 20, 4]

[31, 9, 39, 13, 6, 10, 28, 27, 29, 19, 35, 32, 42, 48, 1,
2, 5, 15, 16, 44, 3, 8, 22, 47, 0, 11, 45, 49, 4, 24, 12,
46, 17, 38, 33, 40, 36, 34, 41, 20, 14, 18, 26, 50, 43, 30,
23, 7, 25, 37, 21]
$
```

Now let us call `Collections.sort()` on both lists. Add these lines of code to `CollectionsDemo.java`

```
        System.out.println("");
        Collections.sort(s);
        Collections.sort(t);
        System.out.println("");
        System.out.println(t);
```

Now run it.

```
java CompareDemo
[17, 4, 43, 22, 30, 36, 27, 5, 34, 6, 2, 45, 31, 25, 16,
19, 35, 7, 14, 40, 38, 8, 23, 49, 47, 41, 33, 3, 42, 0, 44,
32, 39, 26, 29, 10, 11, 28, 20, 13, 9, 50, 37, 1, 15, 21,
24, 12, 46, 48, 18]

[2, 23, 22, 49, 37, 38, 30, 40, 18, 19, 0, 48, 1, 42, 36,
43, 9, 11, 21, 14, 47, 20, 29, 25, 17, 35, 10, 4, 32, 3,
12, 46, 26, 15, 6, 50, 31, 24, 44, 27, 28, 41, 45, 13, 5,
7, 8, 34, 16, 33, 39]
```

```
[0, 1, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 3, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 4, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
5, 50, 6, 7, 8, 9]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
47, 48, 49, 50]
$
```

The sort of the list `s` of strings is done with `String`'s `compareTo()` method. As a result, the items in the list `s` is sorted asciicographically.

The sort of the list `t` of integers is done with `Integer`'s `compareTo()` method. This `comapreTo()` method imposes a linear order on the integers that is the usual linear order we learned from Mrs. Wormwood.

To use `Collections.sort()` in this way, the list involved must contain items that are instances of a class that implements `Comparable<T>`.

## 3 Methods of Sorting: the Bozo sort

It is well and good that the Collections Framework provides a sorting mechanism for us. However, we really should look under the hood and see how sorting works "from the inside." To this end, we will learn about a variety of algorithsm that sort sortable objects.

We will begin by implementing the worst possible sort: the Bozo sort. Let us do this silly thing in Python and leave an an exercise coding it in java. We begin with this shell. You will pass an integer as a command-line argument. The program will punt if you fail to do this.

```
import random
from sys import argv, stderr
def Bozo(x):
    """precondition: x is a mutable sequence of orderable items
    postcondition:  x is sorted according to the natural ordering on its items.
    """
    pass

try:
    n = int(argv[1])
except:
```

```
    print("Error: command line argument must be an integer", file = stderr)
x = list(range(n))
random.shuffle(x)
Bozo(x)
print("Sorted list:  %s" % x)
```

Here is the workings of the Bozo sort. Check if the list is sorted. If it is, return. If not, shuffle and repeat. We shall do this iteratively to avoid overfilling the stack. To begin we write a function to check if the list is in order. We also insert error handling code to avert a crash.

```
import random
from sys import argv, stderr
def isInOrder(x):
    """precondtion: x is a sequence of orderable items
    postcondition:  returns True if x is in order.
    """
    for k in range(len(x) - 1):
        if x[k] > x[k+1]:
            return False
    return True
def Bozo(x):
    """precondition: x is a mutable sequence of orderable items
    postcondition:  x is sorted according to the natural ordering on its items.
    """
    pass

if len(argv) < 2:
    print("Error: No command line argument is present", file = stderr)
    quit()
try:
    n = int(argv[1])
except:
    print("Error: command line argument must be an integer", file = stderr)
x = list(range(n))
random.shuffle(x)
Bozo(x)
print("Sorted list:  %s" % x)
```

Finally, we insert the main loop into our program.

```
import random
from sys import argv, stderr
def isInOrder(x):
    """precondtion: x is a sequence of orderable items
```

```
    postcondition:  returns True if x is in order.
    """
    for k in range(len(x) - 1):
        if x[k] > x[k+1]:
            return False
    return True
def Bozo(x):
    """precondition: x is a mutable sequence of orderable items
    postcondition:  x is sorted according to the natural ordering on its items.
    """
    while not isInOrder(x):
        random.shuffle(x)

if len(argv) < 2:
    print("Error: No command line argument is present", file = stderr)
    quit()
try:
    n = int(argv[1])
except:
    print("Error: command line argument must be an integer", file = stderr)
x = list(range(n))
random.shuffle(x)
Bozo(x)
print("Sorted list:  %s" % x)
```

Now we take it on a few runs on a MacBook Air with 8 gB of memory.

```
$ time python Bozo.py 4
Sorted list:  [0, 1, 2, 3]

real    0m0.056s
user    0m0.039s
sys 0m0.013s
Mon Apr 04:10:53:java>time python Bozo.py 5
Sorted list:  [0, 1, 2, 3, 4]

real    0m0.059s
user    0m0.040s
sys 0m0.013s
$ time python Bozo.py 6
Sorted list:  [0, 1, 2, 3, 4, 5]

real    0m0.056s
user    0m0.039s
sys 0m0.012s
$ time python Bozo.py 7
```

```
Sorted list:  [0, 1, 2, 3, 4, 5, 6]

real    0m0.092s
user    0m0.075s
sys 0m0.012s
$ time python Bozo.py 8
Sorted list:  [0, 1, 2, 3, 4, 5, 6, 7]

real    0m0.096s
user    0m0.080s
sys 0m0.012s
$ time python Bozo.py 9
Sorted list:  [0, 1, 2, 3, 4, 5, 6, 7, 8]

real    0m1.150s
user    0m1.121s
sys 0m0.017s
$ time python Bozo.py 10
Sorted list:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

real    0m1.170s
user    0m1.147s
sys 0m0.015s
$ time python Bozo.py 11
Sorted list:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

real    2m53.774s
user    2m49.006s
sys 0m0.990s
$
```

Once we get to 11 elements, we see that this process will take an eternity. Later we shall analyze this more precisely. Needless to say, this is an algorithm that works, but works poorly. Clearly this is not the way to go.

# 4 Methods of Sorting: The Bubble Sort

Now we rise up the product line to a faster, more practical sort. This is "better" but not necessarily best. It will sort 10,000 orderable items quickly on a modern computer.

Here is the idea. Suppose we have a list of orderable items. Begin by iterating through the neighboring pairs of items in the list. For each pair of items, swap the items if they are out of order; otherwise, leave them alone and move on.

After one pass, what is *guaranteed* to have happened? The largest item with respect to the ordering on the list is now at the end of the list. Hence, the name of this sorting procedure; note that the largest element "bubbles up" to the end of the list.

When we begin we should have a pointer at the end of the list. After each pass, we decrement this pointer. When we do this, we know the following.

1. The items to the right of the pointer are sorted in order.
2. The items to the right of the pointer have higher order rank than all of the items to the left of the pointer.
3. There is no particular order to the items on the left of the pointer.

Note that all of these conditions were vacuously true at the beginning, since there were no elements to the right of the pointer when we started.

When there is only one item to the left of the pointer, we are done.

Now let us get this into code. While we are here, we will see how to create a generic static service class, which we will call `Sordid.java`. In making this class, we will declare it `final` and we will mark its constructor `private`. This is customary practice with static service classes. You notice that this class is purported to be generic but that there is no type parameter yet.

```
public final class Sordid
{
    private Sordid(){}
}
```

## 4.1 A Brief but Important Aside

Let us discuss why we can't use the type parameter in the usual way. Suppose we were to make this class generic like so.

```
public class Sordid<E>
{
    public static void main(String[] args)
    {
        E tmp = null;
    }
}
```

Let us declare an object of the type of the type parameter in a static method.

When we compile we get an ugly error.

```
Tue Apr 05:11:20:BF>javac Sordid.java
Sordid.java:5: error: non-static type variable E cannot be
referenced from a static context
    E tmp = null;
      ^
1 error
```

Conclusion: *the type parameter of a generic class is instance data, so it cannot be used directly in static methods.*

We will make the sorting `methods` generic. For a model, let us look at the static service class `Collections`. Here is the method detail for `sort`.

`public static <T extends Comparable<? super T>> void sort(List<T> list)`

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be mutually comparable (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements e1 and e2 in the list).

This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort.

The specified list must be modifiable, but need not be resizable.

**Implementation Note:**
This implementation defers to the `List.sort(Comparator)` method using the specified list and a null comparator.

**Type Parameters:**
`T` - the class of the objects in the list **Parameters:**
`list` - the list to be sorted. **Throws:**
`ClassCastException` - if the list contains elements that are not mutually comparable (for example, strings and integers). `UnsupportedOperationException` - if the specified list's list-iterator does not support the set operation. `IllegalArgumentException` - (optional) if the implementation detects that the natural ordering of the list elements is found to violate the `Comparable` contract **See Also:**
`List.sort(Comparator)`

Let us write a simple example to illustrate this usage. What this does is to take an array of a given type and convert it to being an array list of the same type

```
import java.util.ArrayList;
public class Converter
{
    public static <E> ArrayList<E>  array2ArrayList(E[] array)
    {
```

```
        ArrayList<E> out = new ArrayList<>();
        for(E a: array)
        {
            out.add(a);
        }
        return out;
    }
    public static void main(String[] args)
    {
        String[] foo = {"aardvark", "bobcat", "capybara", "dingo", "eland"};
        ArrayList<String> ourList = array2ArrayList(foo);
        System.out.println(ourList);
        System.out.println(ourList.getClass());

    }
}
```

Look at the header. After `public static` comes `<E>`. This is the type parameter for this *method*. Next we see that this method will return an object of type `ArrayList<E>`, and that it accepts as input an object of type `E[]`, i.e. an array whose entries are of type `E`. The `main` method drives the class and converts an array of strings into an `ArrayList<String>`.

## 5 And Now Back to our Main Thread

So let us begin to frankencode. We will steal the tortured-looking function header to make our generic bubble sort and swap methods. Once we get this working, we will discuss the meaning of this seemingly convoluted syntax.

```
public final class Sordid
{
    private Sordid(){}
    public static <T extends Comparable<? super T>> void bubble(List<T> list)
    {
    }
}
```

We know we are going to be swapping, so let's add a swap method as well.

```
public final class Sordid
{
    private Sordid(){}
    public static <E extends Comparable<? super E>> void bubble(List<E> list)
```

```
    {
    }
    public static <E extends Comparable<? super E>>
         void swap(List<E> list, int m,  int n)
    {
    }
}
```

Now we code up `swap`. If `m != n`, we do nothing.

```
    public static <E extends Comparable<? super E>>
        void swap(List<E> list, int m,  int n)
    {
        if(m != n)
        {
            E tmp = list.get(m);
            list.set(m, list.get(n));
            list.set(n, tmp);
        }
    }
```

You should test this method. Now we begin to think about our `bubble` method. We repeat it here for your convenience.

When we begin we should have a pointer at the end of the list. After each pass, we decrement this pointer. When we do this, we know the following after each repetition of the loop. This is called the *loop invariant*.

1. The items to the right of the pointer are sorted in order.
2. The items to the right of the pointer have higher order rank than all of the items to the left of the pointer.
3. There is no particular order to the items on the left of the pointer.

Note that all of these conditions were vacuously true at the beginning, since there were no elements to the right of the pointer when we started.

When there is only one item to the left of the pointer, we are done.

```
    public static <E extends Comparable<? super E>>
        void bubble(List<E> list)
    {
        int end = list.size();  //pointer to end of the list
        while(end > 1)
        {
            //iterate through pairs
```

```
            for(int k = 0; k < end - 1; k++)
            {
                if(list.get(k).compareTo(list.get(k + 1)) > 0)
                {
                    swap(list, k, k + 1);
                }
            }
        end--;
        }
    }
```

Now let us add a main method and put this to the test.

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
public final class Sordid
{
    private Sordid(){}
    public static <E extends Comparable<? super E>>
        void swap(List<E> list, int m,  int n)
    {
        if(m != n)
        {
            E tmp = list.get(m);
            list.set(m, list.get(n));
            list.set(n, tmp);
        }
    }
    public static <E extends Comparable<? super E>> void bubble(List<E> list)
    {
        int end = list.size();  //pointer to end of the list
        while(end > 1)
        {
            //iterate through pairs
            for(int k = 0; k < end - 1; k++)
            {
                if(list.get(k).compareTo(list.get(k + 1)) > 0)
                {
                    swap(list, k, k + 1);
                }
            }
        end--;
        }
    }
    public static void main(String[] args)
```

```
    {
        if(args.length < 1)
        {
            System.err.println("Integer command line argument required.");
        }
        int n = 0;
        try
        {
            n = Integer.parseInt(args[0]);
        }
        catch(NumberFormatException ex)
        {
            System.err.printf("Command line argument %s must be an integer.\n", args[0]);
        }
        List<Integer> al = new ArrayList<>(n);
        for(int k = 0; k < n; k++)
        {
            al.add(k);
        }
        Collections.shuffle(al);
        System.out.println(al);
        bubble(al);
        System.out.println(al);
    }
}
```

Run this and see it sort.

```
java Sordid 100
[72, 86, 76, 24, 53, 42, 40, 31, 73, 17, 79, 20, 14, 96, 50, 77, 81,
75, 84, 91, 88, 93, 3, 85, 25, 1, 82, 34, 83, 38, 58, 80, 12, 41, 30,
67, 7, 21, 33, 52, 98, 54, 19, 56, 65, 63, 62, 60, 45, 92, 89, 66, 90,
74, 18, 99, 37, 70, 11, 4, 57, 47, 39, 78, 22, 13, 55, 8, 5, 27, 16,
49, 23, 97, 32, 10, 68, 0, 69, 2, 44, 43, 28, 51, 61, 87, 26, 35, 15,
9, 36, 94, 95, 48, 71, 59, 6, 29, 46, 64]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73,
74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
92, 93, 94, 95, 96, 97, 98, 99]
Tue Apr 05:14:52:java>
```

# 6 Quadratic Sorts

Let us begin here with an analysis of the bubble sort. Let $n$ denote the size or the list we are sorting. When we make the first pass, we visit $n-1$ pairs of neighbors and we decrement the pointer; this amounts to $n$ operations.

On the second pass, we visit $n-2$ pairs of neighbors and we decrement pointer. This has us doing $n-1$ operations. Continuing in this fashion we see that we wind up performing

$$n + (n-1) + (n-2) + \cdots + 2 + 1 = \sum_{k=0}^{n} k$$

operations.

Legend, or possibly fact, has it that Karl Freidrich Gauss had incurred the displeasure of one of his teacher in grade school. As punishment he was told to add up the numbers 1-100. About a minute later his hand went up with an answer: 5050. How did he do this? A little creative rearranging makes the answer clear.

```
  1    2   3   4   5          ....          47   48   49   50
100   99  98  97  96                        54   53   52   51
-------------------------------------------------------------
101  101 101 101 101          ....         101  101  101  101
```

All of the columns add to 101 and there are 50 of them, so the answer is $101 \times 50 = 5050$. Now consider this.

$$\sum_{k=1}^{100} k = 50 \cdot 101 = \frac{100 \cdot 101}{2}.$$

This just might lead you to guess that

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2};$$

This guess would be correct.

What does this mean for the bubble sort? The number of operations needed to do this sort is $O(n^2)$, which means that it is worst proportional to the square of the size of the list. So, if your list is twice as big, it will take roughly four times as long to sort it with the bubble sort. As a result, the bubble sort is called a *quadratic sort*.

In this section we will meet two other quadratic sorts. The logic involved for these sorts is very simple, as it was for the bubble sort.

For the first example suppose we have a list of orderable items. Place a pointer at the end of the list. Now traverse the list and find the biggest element in the list. Swap that element with the element just before the pointer, then decrement the pointer. Repeat. We see that this has exactly the same loop invariant as the bubble sort. This sort is called the *selection sort* since we are selecting the largest element in each pass. Now get out your `Sordid.java` and add this new method.

```
public static <E extends Comparable<? super E>> void selection(List<E> items)
{
}
```

Begin by inserting the pointer and the loop.

```
public static <E extends Comparable<? super E>> void selection(List<E> items)
{
    int ptr = items.size();
    while(ptr > 1)
    {
        //inner loop goes here
        ptr--;
    }
}
```

Now we need to code the procedure that traverses the list in search of the largest element. What we need to do here is to keep track as we go along of the largest element we have seen so far and its index. We need the index since we swap by index. We create the variables `candidate` and `maxIndex` to keep track of these two critical quantities.

```
public static <E extends Comparable<? super E>> void selection(List<E> items)
{
    int ptr = items.size();
    while(ptr > 1)
    {
        //find largest
        int maxIndex = 0;
        int k = 0;
        E candidate = items.get(0);
        while(k < ptr )
        {
            if(items.get(k).compareTo(candidate) > 0)
```

```
            {
                candidate = items.get(k);
                maxIndex = k;
            }
            k++;
        }
        //exchange with end
        //watch for the FENCEPOST.
        swap(items, maxIndex, ptr - 1);
    ptr--;
    }
}
```

Note that this is quite similar to the bubble sort when you analyze its run-time. It is also a quadratic sort.

Now we will look at a second sort. Here is how it works. Imagine you are being dealt cards with numbers on them and you are picking them up to make a sorted hand. As you pick up each card, you will insert it into its order-correct spot. Hence this sort will be known as the *insertion sort*. We would like to keep all of the sorting in the original memory of the array. To do this, let us begin by placing a pointer at the beginning of the array (index 0). Increment this pointer.

What is now true? Every item to the left of the pointer is sorted. All bets are off to the right.

Begin each cycle by incrementing the pointer. You have invited one new entry into the sorted area. Now trickle the entry down by swapping it with its neighbor to the left until this neighbor is equal to or smaller than the new invitee.

This maintains the loop invariant: after each cycle, the sublist to the left of the pointer is sorted. Let's look at an example. We will now carry this procedure out in painful detail so you can get a good understanding of how it works. Here we begin by inserting the pointer; it is represented by a —. Remember, indices of lists live *between* their entries.

```
unsorted:          9   2   3   1   6
insert pointer     9 | 2   3   1   6
```

Now invite the 2 in by incrementing the pointer.

```
unsorted:          9   2   3   1   6
insert pointer     9 | 2   3   1   6
invite:            9   2 | 3   1   6
```

Since $2 < 9$ swap. This completes the "trickle down" step.

```
unsorted:          9   2   3   1   6
insert pointer     9 | 2   3   1   6
insert pointer     9 | 2   3   1   6
invite:            9   2 | 3   1   6
swap:              2   9 | 3   1   6
```

Notice that the loop invariant is maintained. Next, invite 3.

```
unsorted:          9   2   3   1   6
insert pointer     9 | 2   3   1   6
insert pointer     9 | 2   3   1   6
invite:            2   9 | 3   1   6
invite:            2   9   3 | 1   6
```

Since $3 < 9$ we swap.

```
unsorted:          9   2   3   1   6
insert pointer     9 | 2   3   1   6
insert pointer     9 | 2   3   1   6
invite:            2   9 | 3   1   6
invite:            2   9   3 | 1   6
swap:              2   3   9 | 1   6
```

Since $3 > 2$, we stop the trickling process. This completes the loop execution. Again the loop invariant is unchanged. Now invite 1 by incrementing the pointer.

```
unsorted:          9   2   3   1   6
insert pointer     9 | 2   3   1   6
insert pointer     9 | 2   3   1   6
invite:            2   9 | 3   1   6
invite:            2   9   3 | 1   6
swap:              2   3   9 | 1   6
invite:            2   3   9   1 | 6
```

Now trickle the 1 down. All swaps are shown.

```
unsorted:          9   2   3   1   6
insert pointer     9 | 2   3   1   6
insert pointer     9 | 2   3   1   6
invite:            2   9 | 3   1   6
invite:            2   9   3 | 1   6
swap:              2   3   9 | 1   6
invite:            2   3   9   1 | 6
swap:              2   3   1   9 | 6
```

```
swap:              2   1   3   9 | 6
swap:              1   2   3   9 | 6
```

Finally, invite 6 and swap to get this

```
unsorted:          9   2   3   1   6
insert pointer     9 | 2   3   1   6
insert pointer     9 | 2   3   1   6
invite:            2   9 | 3   1   6
invite:            2   9   3 | 1   6
swap:              2   3   9 | 1   6
invite:            2   3   9   1 | 6
swap:              2   3   1   9 | 6
swap:              2   1   3   9 | 6
invite:            1   2   3   9   6 |
swap:              1   2   3   6   9 |
```

You now have a sorted list. Notice that, on average, we will only have to trickle the invitee halfway down the sorted region of the list. This makes the insertion sort work roughly twice as fast as the bubble sort.

Let us now write code. We see the basic step is as follows.

1. While the pointer is not at the end of the list, invite in a new entry by incrementing it.
2. Trickle the invitee down until it is in the correct place.

We begin by creating the pointer and inserting the outer loop that will govern the repetitions of the trickle down procedure.

```
public static <E extends Comparable<? super E>> void insertion(List<E> items)
{
    int ptr = 0;
    int n = items.size();
    while(ptr < n)
    {
        ptr++;
        //do the trickle down procedure.
    }
}
```

Here is how the trickle down procedure works.

1. Compare the invitee to its neighbor to the left; if no such neighbor exists, you are done.

2. If the neighbor to the left is smaller you are done.
3. Otherwise swap the invitee with its neighbor to the left. Repeat until the procedure halts for one of the two reasons specified above.

```
public static <E extends Comparable<? super E>> void insertion(List<E> items)
{
    int ptr = 0;
    int n = items.size();
    while(ptr < n)
    {
        ptr++;
        int k = ptr - 1;  //points at invitee
        //do the trickle down procedure.
            //while there is a neighbor to the left and
            //that neighbor is bigger than the invitee
            //swap and keep k pointing at the invitee
        while( k > 0 && items.get(k - 1).compareTo(items.get(k)) > 0)
        {
            swap(items, k, k - 1);
            k--;
        }
    }
}
```

You should do some experimenting with sorting lists of various sizes. You will see that the insertion sort is, in general, the fastest of these three algorithms.

# 7 Recursive Sorting Methods

Consider this wild proposition. Suppose we have a huge list we are sorting and that it takes time $T$ to sort it. Let's cut the list in half. Because we only know quadratic sorts, it will take only $T/4$ to sort each of the lists. Then we just zipper the lists together and the total time becomes

$$T/2 + \text{time to zipper.}$$

Let us now design a way to zipper two sorted lists. Say we two lists

```
x -> [1,2,4,6,9,12, 15, 18,22]
y -> [3,4,8,16, 22, 40, 48, 62]
```

To get started, make two pointers that point at the beginning of the two lists, `px` and `py`. Then create a list called `out` that is empty. Append the smaller of

`x[px]` and `y[py]` to `out` and increment the pointer for the variable that was used. Do this until you arrive at the end of one list. Then attach the tail of the remaining list (if any) to the end out and return it.

Let's write this in Python; you can also code in in Java as an exercise. We begin by creating the two pointers and `out`.

```
def zip(x,y):
    px = 0
    py = 0
    out = []
```

Now we will create a loop to zip the two lists until one runs out. Note the use of DeMorgan's law: `not(P and Q) == (not P) or (not Q)`. When the while loop is done executing, one of the pointers will be at the end of its list.

```
def zip(x,y):
    px = 0
    py = 0
    out = []
    while px < len(x) and py < len(y):
        if x[px] < y[py]:
            out.append(x[px])
            px += 1
        else:
            out.append(y[py])
            py += 1
```

Now for a dirty trick. Since one of the tails of the lists will be empty, we can just append both.

```
def zip(x,y):
    px = 0
    py = 0
    out = []
    while px < len(x) and py < len(y):
        if x[px] < y[py]:
            out.append(x[px])
            px += 1
        else:
            out.append(y[py])
            py += 1
    out.extend(x[px:])
    out.extend(y[py:])
    return out
```

Now add these lines and see it work.

```
x = [1,2,5,8,9,22, 60, 85]
y = [0,1,2,5,9,11,59,66,102, 109, 200]
print(zip(x,y))
```

Running we see that it behaves as specified.

```
$python zip.py
[0, 1, 1, 2, 2, 5, 5, 8, 9, 9, 11, 22, 59, 60, 66, 85, 102, 109, 200]
```

Observe that this zip procedure uses a loop that passes through the combined lists once. Therefore its running time is at worst proportional to the size of the list. Hence, this is an $O(n)$ procedure. Such procedures we call *linear time* procedures.

So, we could take a large unsorted list, cut it in half, sort each half, then zipper the two sorted halves together to get a sorted list. This indeed would result in a running time that is nearly twice as fast. But... if it pays to cut the list in half once, why not do it repeatedly?

Here is the idea. We know for sure that empty lists and lists with one element are sorted. So, we will keep snapping the list in half until it is empty or one-element, then start zipping the results together.

This sounds fanciful but the magic of recursion can be used to make it happen. So we will code a function called `merge` that achieves this feat. Begin as follows

```
def merge(x):
    if len(x) <= 1:
        return x
```

Now snap the list `x` in half.

```
def merge(x):
    if len(x) <= 1:
        return x
    half = len(x)//2
    first = x[:half]
    second = x[half:]
```

We must guarantee that `merge` returns a sorted list. If `merge(first` and `merge(second)` are both sorted, then `zip(merge(first), merge(second)` is also sorted. Let's return it!

```
def merge(x):
    if len(x) <= 1:
```

```
        return x
    half = len(x)//2
    first = x[:half]
    second = x[half:]
    return zip(merge(first), merge(second))
```

Now let us put this together and make a modest test.

```
import random
def zip(x,y):
    px = 0
    py = 0
    out = []
    while px < len(x) and py < len(y):
        if x[px] < y[py]:
            out.append(x[px])
            px += 1
        else:
            out.append(y[py])
            py += 1
    out.extend(x[px:])
    out.extend(y[py:])
    return out
def merge(x):
    if len(x) <= 1:
        return x
    half = len(x)//2
    first = x[:half]
    second = x[half:]
    return zip(merge(first), merge(second))
BIG = 10
x = list(range(BIG))
random.shuffle(x)
print(x)
print(merge(x))
```

Now run this and see it work.

```
$python zip.py
[4, 7, 3, 8, 9, 2, 1, 0, 5, 6]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
$
```

Change the main routine for `merge.py` as follows. We get rid of the IO because it slows things down.

```
BIG = 100000
x = list(range(BIG))
random.shuffle(x)
merge(x)
```

Running this on a MacBook Air with 8gB of memory yields this result.

```
Thu Apr 07:09:36:java>time python zip.py

real    0m1.393s
user    0m1.361s
sys 0m0.022s
Thu Apr 07:09:36:java>
```

Now set `BIG = 1000000` and do this again

```
$ time python zip.py

real    0m15.439s
user    0m15.190s
sys 0m0.146s
$
```

We sorted a list of a million elements in a reasonably short time. This task left the quadratic sorts wallowing like pigs in mud. The quadratic methods are saddled with the chore of dealing with a number of operations that is on the order of $10^{12}$. No wonder they are slow.

This sort is *not* an in-place sort. It returns a copy of the list sorted in order.

**Programming Exercises**

1. Code the `zip` method in Java using a generic method.
2. Go to `http://www.sorting-algorithms.com`. This has cool visualizations of the algorithms we have discussed here and quite a few more.
3. Code insertion sort in Python.
4. Modify `merge` sort as follows. If the list passed it has 10 or fewer elements sort it with `insertion` and then pass it to `merge`. Tinker with the number 10. Does that make your sort faster?

## 7.1 Quicksort

Quicksort enjoys the advantage of being an in-place sort. We shall code it in Java. The basic step we need to complete first is to do *pivoting* in a list. Here is how it works.

1. Use the first entry as the pivot entry. Later we will pick an entry randomly and exchange it with the first entry and pivot on it, but postponing this will make it easier to test and debug.
2. Place a pointer at the end of the list. Everything to the left of the pivot entry is order-smaller than the pivot entry and everything to the right of the end pointer is order-larger than the pivot entry.
3. Look at the neighbor just to the pivot entry's right. If it is order-larger than the pivot entry, exchange it with the last item on the list and decrement the pointer.
4. If the neighbor to the right is order-smaller than the pivot entry, swap it with the pivot entry.
5. This process ends when the pivot entry's index is equal to one less than the end pointer.

We shall begin by writing a version of this algorithm that acts on an entire list. Then we shall need a version that acts on a sublist. This is actually very simple to write.

Let us carry this out an example Suppose we have a list.

```
 6 3 2 5 1 7|
pivotEntry: 0
```

The — designates the end pointer. Since $3 < 6$, we just exchange and increment the pivot entry pointer and we have

```
 3 6 2 5 1 7|
pivotEntry: 1
```

Since $2 < 6$ we exchange to the left and increment the pivot entry pointer again.

```
 3 2 6 5 1 7|
pivotEntry: 2
```

This also happens with 5 and 1 so we have

```
 3 2 5 1 6 7|
pivotEntry: 4
```

Since $7 > 6$ we swap with the entry just to the left of the pointer (nothing happens) and decrement the pointer to see this.

```
 3 2 5 1 6|7
pivotEntry: 4
```

Observe that everything to the left of the pivot entry is no larger than the pivot entry, everything to the right is at least as large, and that the pivot entry pointer points to the pivot entry at all times. This is the desired outcome. Don't sweat ties: you can make them go either way. So let us begin by coding and testing this process. As the loop executes we will print out the partial results so we can see how we are doing.

Let us begin by creating the method header and setting up the pivot entry and end pointers.

```
private static <E extends Comparable<? super E>> int pivot(List<E> list)
{
    int pivotEntry = 0;
    int end = list.size();
    return pivotEntry;
}
```

Here is the loop invariant. After each pass of the loop the following will be true.

1. The index `pivotEntry` will point at the the pivot entry.
2. All items to the right of `end` will be at least as large as the pivot entry.
3. All items to the left of the pivot entry will be no larger than the pivot entry.

Notice these are true right at the start as well.

We will keep going as long as `pivotEntry < end + 1`. When this condition is false, we are done. So, we put in the main loop. We also insert comment telling what we are going to do.

```
private static <E extends Comparable<? super E>> int pivot(List<E> list)
{
    int pivotEntry = 0;
    int end = list.size();
    return pivotEntry;
    while(pivotEntry < end + 1)
    {
        //if the pivot entry's right neighbor is smaller
        //than the pivot entry, swap the pivot entry and its neighbor.
        //increment the pivot entry pointer

        //if the pivot entry's right neighbor is larger
        //than the pivot entry, swap the pivot entry and the entry
        //just before the end pointer
        //decrement the end pointer
```

```
    }
}
```

Now drop in the code.

```
private static <E extends Comparable<? super E>> int pivot(List<E> list)
{
    int pivotEntry = 0;
    int end = list.size();
    return pivotEntry;
    while(pivotEntry < end + 1)
    {
        //if the pivot entry's right neighbor is smaller
        //than the pivot entry, swap the pivot entry and its neighbor.
        //increment the pivot entry pointer
        if(list.get(pivotEntry).compareTo(list.get(pivotEntry + 1)) <  0)
        {
            swap(list, pivotEntry + 1, end - 1);
            end--;
        }

        //if the pivot entry's right neighbor is larger
        //than the pivot entry, swap the pivot entry and the entry
        //just before the end pointer
        //decrement the end pointer
        else
        {
            swap(list, pivotEntry, pivotEntry + 1);
            pivotEntry++;
        }
        //Use this code to test some cases.  You can see the loop's
        //progress
        System.out.printf("pivotEntry = %s, %s\n", pivotEntry, list);
    }
}
```

Notice that as the loop executes, the pivot entry pointer follows the pivot entry. Notice that, at the end the pivoted value 10 is pointed at by `pivotEntry`.

```
$ java Sordid
Original array:  [10, 1, 11, 8, 3, 14, 20]
pivotEntry = 1, [1, 10, 11, 8, 3, 14, 20]
pivotEntry = 1, [1, 10, 20, 8, 3, 14, 11]
pivotEntry = 1, [1, 10, 14, 8, 3, 20, 11]
pivotEntry = 1, [1, 10, 3, 8, 14, 20, 11]
```

```
pivotEntry = 2, [1, 3, 10, 8, 14, 20, 11]
pivotEntry = 3, [1, 3, 8, 10, 14, 20, 11]
pivotEntry = 3
Pivoted array:  [1, 3, 8, 10, 14, 20, 11]
$
```

Having seen this, comment out the print routine that executes in every loop.

```
System.out.printf("pivotEntry = %s, %s\n", pivotEntry, list);
```

Now for the next step. We are going to pivot on a sublist of our list. To do this, we will create an overloaded method that takes the bounds of the sublist and which pivots only within that sublist, leaving the rest of the list untouched. Here is the method header.

```
private static <E extends Comparable<? super E>> int
    pivot(List<E> list, int start, int endBefore)
{

}
```

You will see now that all of the real work is done.

```
private static <E extends Comparable<? super E>>
    int pivot(List<E> list, int start, int endBefore)
{
    List<E> sublist = list.subList(start, endBefore);
    return start + pivot(sublist);
}
```

You can write some test code in the main method and see that this works. If you wish, you can uncomment the output line in the loop and you get to see all of the gory details from start to finish.

What is next? We will pivot the list, and then call the pivoting procedure recursively on the subarrays to the left and to the right of the pivot entry. Notice that the pivot entry must be in the exact right spot it will occupy in the final sorted array. We will use a recursive helper function that acts on a subarray.

```
private static <E extends Comparable<? super E>> void
    quickSort(List<E> list, int start, int endBefore)
{
    if(endBefore - start <= 1)
    {
        return;
```

```
    }
    int divide = pivot(list, start, endBefore);
    quickSort(list, start, divide - 1);
    quickSort(list, divide + 1, endBefore);
}
```

We then do the public part that gets it all started.

```
public static <E extends Comparable<? super E>> void quickSort(List<E> list)
{
    quickSort(list, 0, list.size());
}
```

# 8 Shuffling: The Fisher-Yates Algorithm

We begin our exploration by performing a "naïve shuffle" as follows. For each element, exchange that element with a random choice of elements from the list. Seems reasonable, eh?

Implementing this algorithm is appealingly simple. All we need is a random number generator. Let us make a class for shuffling.

```
public final class Shuffle
{
    static Random r = new Random();
    private Shuffle()
    {
    }
    private static <E extends Comparable<? super E>>  void
        swap(List<E> items, int a, int b)
    {
        if(a != b)
        {
            E tmp = items.get(b);
            items.set(b, items.get(a));
            items.set(a, tmp);
        }
    }
    public static <E extends Comparable<? super E>>
        void naiveShuffle(List<E> items)
```

```
    {
        int n = items.size();
        for(int k = 0; k < n; k++)
        {
            int chosen = r.nextInt(n);
            swap(items, chosen, k);
        }
    }
}
```

We shall now test this algorithm using the familiar technique of creating a dictionary that tallies results. All shufflings of a list should be equiprobable. What we will do here is to run this shuffling procedure on the list `[1,2,3]` 600,000 times.

Here is the program we will run.

```
import java.util.Random;
import java.util.Arrays;
import java.util.Map;
import java.util.HashMap;
import java.util.List;
public final class Shuffle
{
    static Random r = new Random();
    private Shuffle()
    {
    }
    private static <E extends Comparable<? super E>>  void
        swap(List<E> items, int a, int b)
    {
        if(a != b)
        {
            E tmp = items.get(b);
            items.set(b, items.get(a));
            items.set(a, tmp);
        }
    }
    public static <E extends Comparable<? super E>>
        void naiveShuffle(List<E> items)
    {
        int n = items.size();
        for(int k = 0; k < n; k++)
        {
            int chosen = r.nextInt(n);
            swap(items, chosen, k);
```

```
        }
    }
    public static void main(String[] args)
    {
        Map<List<Integer>, Integer> tally = new HashMap<List<Integer>, Integer>();
        for(int k  = 0; k < 600000; k++)
        {
            List<Integer> x = Arrays.asList(1, 2, 3);
            naiveShuffle(x);
            //System.out.println(x);
            if(tally.containsKey(x))
            {
                tally.put(x, tally.get(x) + 1);
            }
            else
            {
                tally.put(x, 1);
            }
        }
        for(List<Integer> shuffling: tally.keySet())
        {
            System.out.printf("%s appeared %s times\n",
                shuffling, tally.get(shuffling));
        }
    }
}
```

```
$ java Shuffle
[3, 2, 1] appeared 88815 times
[1, 2, 3] appeared 89320 times
[2, 1, 3] appeared 111527 times
[3, 1, 2] appeared 88475 times
[1, 3, 2] appeared 110946 times
[2, 3, 1] appeared 110917 times
$
```

We now inject a note of statistical sobriety. For any of the shufflings we are seeing repeated statistically independent trials with probability of success equal to $1/6$. For $n$ independent trials with probably of success $p$, the standard deviation on the number of successes is $\sqrt{np(1-p)}$. In our case we have

$$\sqrt{np(1-p)} = \sqrt{600000(1/6)(5/6} = 288.7.$$

So, we expect the vast majority of our observations to be within 2 or 3 standard deviations of the mean, which in this case is $100,000$. We see that **none** of the

observations meet this criterion. Go ahead, run this a bunch of times. This procedure has a big problem. It has some kind of ugly bias built into it.

Three of the shufflings, `[2, 1, 3]`, `[1, 3, 2]`, and `[2, 3, 1]` are consistently over-represented. The over-represented shufflings appear about 18.5% of the time, and the under-represented shufflings occur about 14.8% of the time. This occurs consistently. In fact the exact probability of occurrence for the more likely shufflings is 5/27 and the less likely is 4/27.

For more details on this bias, we suggest Jeff Atwood's article `http://blog.codinghorror.com/the-danger-of-naivete/`. He performs a careful analysis of this algorithm and shows why it is fundamentally flawed.

We now present a much better algorithm, the *Fisher-Yates* algorithm. Here is the idea. Place a pointer at index 1. Swap the first item with a randomly-chosen item from the elements past the pointer. Then increment the pointer. Proceed in this manner until the pointer is at the end of the list.

```
public static <E extends Comparable<? super E>>  void fyShuffle(List<E> items)
{
    int entropy = 0;
    int n = items.size();
    while (entropy < n)
    {
        int chosen = entropy + r.nextInt(n - entropy);//pick entry
        swap(items, chosen, entropy);
        entropy++;
    }
}
```

Now run this.

```
$ java Shuffle
[3, 2, 1] appeared 99982 times
[1, 2, 3] appeared 100081 times
[2, 1, 3] appeared 100099 times
[3, 1, 2] appeared 99906 times
[1, 3, 2] appeared 100065 times
[2, 3, 1] appeared 99867 times
```

This is more like it. Let us run this 2.4 million times on a list of size 4. The standard deviation here is 577.

```
$java Shuffle
[2, 4, 1, 3] appeared 99948 times
[2, 3, 1, 4] appeared 99817 times
```

```
[1, 3, 2, 4] appeared 99793 times
[1, 4, 2, 3] appeared 100088 times
[3, 4, 1, 2] appeared 99818 times
[3, 2, 1, 4] appeared 99997 times
[1, 2, 3, 4] appeared 100307 times
[1, 4, 3, 2] appeared 100386 times
[2, 4, 3, 1] appeared 99947 times
[3, 4, 2, 1] appeared 99777 times
[3, 1, 2, 4] appeared 100185 times
[4, 2, 1, 3] appeared 99625 times
[2, 1, 3, 4] appeared 100076 times
[4, 3, 1, 2] appeared 99916 times
[1, 3, 4, 2] appeared 100335 times
[1, 2, 4, 3] appeared 99813 times
[4, 1, 2, 3] appeared 99976 times
[2, 1, 4, 3] appeared 99642 times
[2, 3, 4, 1] appeared 99922 times
[4, 3, 2, 1] appeared 100359 times
[4, 1, 3, 2] appeared 99928 times
[3, 2, 4, 1] appeared 100055 times
[3, 1, 4, 2] appeared 100063 times
[4, 2, 3, 1] appeared 100227 times
$
```

This is how both Python and Java's random libraries shuffle lists.

# 9 Searching

Suppose we are searching a list for a particular object. If the elements of the list are in no particular order, all we can really do is this.

1. Compare each item in the list using `.equals` to our object.
2. If one of these evaluations come up as `true`, return `true`.
3. If we go all through the list and we never find our object, return `false`.

Clearly, this is an $O(n)$ procedure, since we are making $n$ checks on the list.

Coding is is very simple. Here is the method; it is unspectacular and utterly straightforward.

```
public static <E extends Comparable<? super E>> boolean
    contains(List<E> items, Object o)
    {
```

```
        for(E item : items)
        {
            if(item.equals(o))
            {
                return true;
            }
        }
        return false;
    }
```

Why do we bother to sort things? The short answer: it makes it faster to find them. So, if a list is sorted, how can we take advantage of that?

Again the "divide and conquer" strategy works. Look at the entry half-way through the list. If it equals the object we seek, we return `true` and are done. If it order-precedes our the half-way entry we know that if our list contains the object we seek, the object is in the first half of the list. If the halfway entry order-precedes the object we seek, the object can only be present in the first half of the list. Finally, if the list itself is empty, the object cannot be present in the list.

So, let us code this procedure.

```
public static <E extends Comparable<? super E>> boolean
    containsSorted(List<E> items, E o)
{
    int n = items.size();
    int midpoint = n/2;   //compute midpoint
}
```

If the list is empty, return false.

```
public static <E extends Comparable<? super E>> boolean
    containsSorted(List<E> items, E o)
{
    if(items.isEmpty())
    {
        return false;
    }
    int n = items.size();
    int midpoint = n/2;   //compute midpoint
}
```

If the item is found at the midpoint, return `true`. Otherwise, narrow the search.

```
public static <E extends Comparable<? super E>> boolean
    containsSorted(List<E> items, E o)
```

```
{
    if(items.isEmpty())
    {
        return false;
    }
    int n = items.size();
    int midpoint = n/2;   //compute midpoint
    if(items.get(midpoint).equals(o)
    {
        return true;
    }
    if(items.get(midpoint).compareTo(o) < 0)
    {
        return containsSorted(items.sublist(items, midpoint + 1, n);
    }
    else
    {
        return containsSorted(items.sublist(items, 0, midpoint);
    }
}
```

Searching a sorted list is much faster. Since the search field shrinks by a factor of 2 on each go-round, this procedure is $O(log(n))$.

### Programming Exercises

1. Write a method `indexOf(List<E> list, E o)` that finds the first index of `o` in the list `list`.
2. Can you find a speedier version for sorted lists?

## 10 Heaps and Heapsort

Trees are near-cousins to linked lists. A link in a linked list can have only one pointer to a `next`, where a tree can have a potentially unlimited number of pointers to "descendants." Where do we see trees?

1. Your file system is hierarchically organized as a tree, where pointing is a containment relationship. A regular file has no descendants. A directory has descendants that are its immediate contents. In a UNIX system, the directory `/` is the root directory. As a tree, a UNIX file system has a single root.
2. The inheritance hierarchy in Java is a tree, with `Object` as its root.

3. The java package hierarchy is a tree.
4. A scene in JavaFX is a tree; descendants of a given node in the scene are just the items contained in that node.

You could implement a tree state like this.

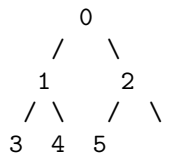
```
import java.util.ArrayList;
public class Tree<E>
{
    TreeNode root;

}
class TreeNode<E>
{
    E datum;
    ArrayList<E> children;
}
```

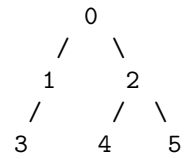
A tree is said to be *binary* if each node has at most two children. For a binary tree you might do this.

```
public BinaryTree<E>
{
    Node<T> head;
}
class Node<E>
{
    E datum
    Node<E> leftChild;
    Node<E> rightChild;
}
```

A binary tree is said to be *complete* if, when you do a level order traversal, there are no gaps in the nodes. We make this clear with an example. The tree below is complete.

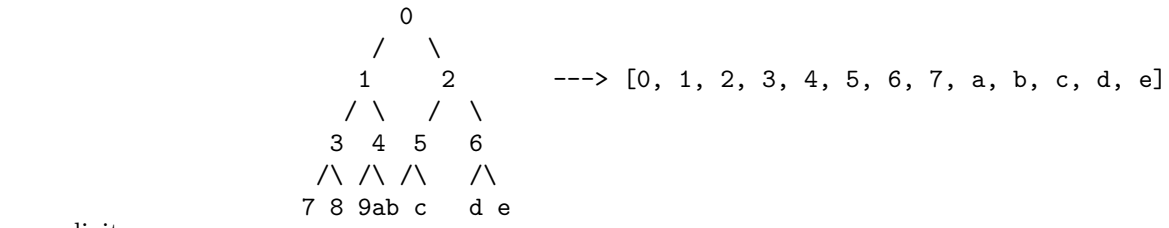



This next binary tree is not. You can see that the absence of a right child to the node containing the 1 spoils its completeness.



There is a dirty rotten trick that allows us to implement a complete binary tree using a list. This implementation will give us simple access, from any node to the parent node and the child nodes.

Consider extending the first tree. We store it in a list like so. Note that we are deliberately populating the three with list indices. Think of the letters as hex digits.




All we did was write the items in the tree in the order imposed by a level-order traversal. Let's do a little spelunking. Consider this table.

| Node | Left Child | Right Child |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 3 | 4 |
| 2 | 5 | 6 |
| 3 | 7 | 8 |
| 4 | 9 | 0xa |
| 5 | 0xb | 0xc |
| 6 | 0xd | 0xe |

Look at the relationship between the node and the left child. Each time the node goes up by 2, its left child index goes up by 2. We see that if we are at node $n$, the left child index is $2n + 1$. Similarly, the right child of node $n$ is $2n + 2$. It looks tricky to go the other way, but if you have a node $n > 1$, its parent is at index $(n - 1)//2$. We can implement a complete binary tree using a simple list as state.

What interests us now is a *heap*; this is a complete binary tree with the additional property that the order-largest element in any subtree is at the root of that subtree. We shall now implement a heap class using a list. We begin making this class by inserting state, and a constructor, and a `toString()` method so we can see what we are doing. Our new heap will be empty.

```
import java.util.ArrayList;
public class Heap<E extends Comparable<E>>
{
    private ArrayList<E> duhList;
    public Heap()
    {
        duhList = new ArrayList<>();
    }
    @Override
    public String toString()
    {
        return duhList.toString()
    }
}
```

The client programmer will have no clue that our internal representation is a list. We begin by adding functions that access, from any node, the parent and the two children, provided they exist. While we do this, let us add a method to check for emptiness of a heap and for getting its size.

```
import java.util.ArrayList;
public class Heap<E extends Comparable<E>>
{
    private ArrayList<E> duhList;
    public Heap()
    {
        duhList = new ArrayList<>();
    }
    @Override
    public String toString()
    {
        return duhList.toString()
    }
    public boolean isEmpty()
    {
        return duhList.isEmpty();
    }
    public boolean size()
    {
        return duhList.size();
    }
    public int yoMama(int n)
    {
        return (n - 1)/2;
    }
    public int leftChild(int n)
```

```
    {
        return 2*n + 1;
    }
    public int rightChild(int n)
    {
        return 2*n + 2;
    }
}
```

Next, let us add new nodes to our heap. Remember, when we are done adding, we need to restore our collection to being a heap. Let us demonstrate the "heapification" algorithm for adding an element.

1. Place the new item in the last position. The heap will still be a complete binary tree.
2. Check the value at the parent node. If there is no parent node, do nothing. If the value at the node is greater than that of the parent, swap the two items.
3. Do this until you get to the top of the tree (case 1) or until the parent node has a larger value than the child.

When this is over, our complete binary tree will be a heap again. Coding that looks like this.

```
public void add(E newItem)
{
    duhList.add(newItem);
    int nav = duhList.size() - 1;

    while(nav > 0 && duhList.get(nav).compareTo(duhList.get(yoMama(nav)))

    {
        swap(nav, yoMama(nav));
        nav = yoMama(nav);
    }
}
```

Note that this procedure is $O(log(n))$ procedure. The depth of a binary tree with $n$ nodes is at most $\lceil \log(n) \rceil$.

a b f c d g e h i j