Contents

0 Introduction

We have learned that data structures are containers that hold collections of related data under a single name. They have an interface and an underlying implementation. You saw in the last chapter that stacks and lists both can be implemented using arrays or links. This distinction is not cosmetic.

In this chapter we begin to think about verbs: *find, search, sort, compute*. These verbs are embodied by methods in a data structure. The efficiency with which certain chores can be done depends in part on the implementation that undergirds the interface.

How do we measure and think about this efficiency? What is the right data structure for a particular job? In this chapter, we start to address those questions. We begin with a little necessary mathematics.

1 Some Mathematical Preliminaries

It is important to understand how fast the cost of a computation grows as a function of its size. To describe this, we will use *Landau symbols*. We three of these now and more as we move along.

Suppose that f and g are functions whose domain contains the positive integers. We will write

$$f = O(g) \qquad \text{as } n \to \infty$$

if for some constant $M > 0$ there is some $N \in \mathbb{N}$, $|f(n)| \leq M|g(n)|$. In other words, the sequence $n \mapsto |f(n)/g(n)|$ is "bounded at infinity." Intuitively, this says that f is at most proportional to g as n gets large.

More strictly, we write $f(n) \sim g(n)$ as $n \to \infty$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

This means that f and g grow at almost the same rate at ∞.

We will write $f \approx g$ if $f = O(g)$ and $g = O(f)$ as $n \to \infty$. This says that f an g grow at comparable rates at ∞. In this case, neither grows faster than a rate proportional to the other.

Theorem *If f and g are non-negative sequences so that the limit*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists, then $f(n) = O(g(n))$ as $n \to \infty$.

Proof. Put

$$M = \lim_{n\to\infty} \frac{f(n)}{g(n)}.$$

Then by the definition of limit there is some $N \in \mathbb{N}$ so that for $n \geq N$,

$$\left|\frac{f(n)}{g(n)}\right| < M + 1.$$

Therefore $n \geq N \implies f(n) \leq (M+1)g(n)$.

Our result follows immediately.

Corollary *If f and g are non-negative sequences so that the limit*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists and is positive, then $f(n) \approx g(n)$ as $n \to \infty$.

2 Searching A List

This terminology we developed in the last sections will come in handy. Let's look at an example. Suppose you have a list of n items not sorted in any particular order. You are asked to search for an item in the list. Since the items are not sorted, we just check each item in succession. We will stop if we find the item we are seeking. If we traverse the entire list and do not find the item, we report it as missing.

The algorithm we have described, a sequential search requires a number of checks that is at worst proportional to the size of the list. Hence we say this is a $O(n)$, or *linear time* algorithm.

Now suppose that the list is sorted. We can search it via a divide-and-conquer method. Let us denote by `quarry` the item we seek and that items in this list are orderable; we use `<` to denote the ordering. Here is an outline of what we do.

1. Let n be the size of the list x. Retrieve the item `x.get(n/2)`.
   - If `x.get(n/2) > quarry`, we know that `quarry` is in the first half of the list. Put `left = 0` and `right = n/2`.
   - if `x.get(n/2) == quarry` then return `true`.
   - If `x.get(n/2) < quarry`, we know that `quarry` is in the second half of the list. Put `left = n/2` and `right = 0`.

Repeat this procedure until `left == right`. At that time, if we never returned `true`, we know we went on a goose chance. In that event return `false`.

How efficient is this procedure? It cuts the number of items in the search field by half each time. Suppose we start with a million items. The search field, at worst behaves as follows.

0	1000000
1	500000
2	250000
3	125000
4	62500
5	31250
6	15625
7	7813
8	3907
9	1953
10	977
11	489
12	245
13	123
14	62
15	31
16	16
17	8
18	4
19	2
20	1

So, by sorting a list one mega-element long, we can now search out an item in 20 steps instead of half of a million. This represents some serious improvement. We sort things to make them easier to find. This search technique is called *binary search*. To use it, the items must be orderable and they must be in order. In Java, this includes the numeric types, characters, and all classes implementing the `Comparable` interface.

All Logs are Created Equal . This is true in the eyes of the big-O notation. Suppose a and b are positive and that $c > 1$ is any other positive number. Then the change of base formula for logs says that

$$\log_a(b) = \frac{\log_c(b)}{\log_c(a)}.$$

Since $c > 1$, $\log_c(a)$ has a the same sign as $\log_c(b)$. Now we are concerned about the behavior of these log functions for large numbers, so we don't care about the nastiness that occurs in the open interval $(0, 1)$. Hence any two logs of base larger than 1 are proportional to one another.

Now the number of times it takes to grind n down to 1 by integer division is $\log_2(n)$. Binary search is a $O(\log_2(n))$ algorithm. Since all logs are proportional you will see us say that it is $O(\log(n)$, where by $\log(n)$, we mean the natural log of n.

Mathematical Problems

1. For what power s does the function $f(x) = x^3 - 3x - 2$ satisfy $f(n) \sim n^s$. For what powers s do we have $f(n) = O(n^s)$.
2. Can you frame a rule generalizing the result of the last problem in terms of $\deg(f)$, the degree of the polynomial f?
3. Show that if f_0 and f_1 are sequence and $f_0(n) = O(g(n))$ and $f_1(n) = O(g(n))$ as $n \to \infty$ then

   $$f_0(n) + f_1(n) = O(g(n))$$

   as $n \to \infty$.
4. A *rational function* is a function that is a ratio of polynomials. If $R(x) = P(x)/Q(x)$, and if P and Q are polynomials with degrees m and n respectively, for what powers r do we have $R(n) = n^r$? What about $R(n) \sim n^r$?

Programming Problems

1. Write a Python or Java method that searches an unsorted list of strings. For Python use the header

   ```
   def search(x, quarry):  ##x is the list to search
       return false        ## return true if the quarry is found
   ```

   For Java, use a method like this.

   ```
   public static boolean search(x, quarry):  ##x is the list to search
       return false        ## return true if the quarry is found
   ```

   Test it in a main method.
2. Implement the binary search method on Python and Java, that searches a sorted list of orderable items.
3. Write a routine to check if a list of strings is in order.

3 Methods of Sorting

Our approach here is to first look at data structures and algorithms from the inside, by creating our own examples from scratch, then from the outside by learning about the Java Collections Framework. This framework is a powerful library of tools that can accomplish serious tasks quickly. But, it is important to know how these tools work as well.

One of the most fundamental processes in computing is that of sorting a collection. We have three types of collection at our disposal now: arrays, `LinkedList`s and `ArrayList`s.

3.1 The Bozo Sort

The least efficient of the sorting algorithms is the *Bozo* sort, which is named after a long-running TV show featuring Bozo the Clown. Here is the procedure. We will think of our array as a deck of n cards numbered 0 through $n-1$. Shuffle the deck; now check if it is in order. If it is not, reshuffle and repeat.

We know that there are $n!$ shufflings of a deck of n cards, so the probability of seeing the deck in the right order on any given trial is $1/n!$.

It is known that if you make independent repeated trials of an experiment with probability $p > 0$ of success in any trials, the average number of trials to success is $1/p$. So we would expect a deck of n cards, to take on the average, $n!$ trials to be sorted. So the average time is $O(n!)$. The laws of large numbers say that this process will end with probability 1. However, we have no upper bound on the worst-case scenario.

What is awful is to note that the addition of one card makes the mean sort time much longer. We know we can do better. The Bozo sort works fine for sorting a list of two or three elements, but beyond that, forget it.

Implementing this algorithm in Python is a simple exercise, so we show it here. First we write a function that sees if a list in in order. All we do is to advance to each adjacent pair of items in the list. In each repetition, we check to see if the items are out of order. If they are we are done. If we make it all the way through, the items are in the correct order.

```
isInOrder(x):
    for k in range(len(x) -1):
        if x[k] > x[k+1]:
            return False
    return True
```

Now for the rest.

```
import random
from sys import argv
def isInOrder(x):
    for k in range(len(x) -1):
        if x[k] > x[k+1]:
            return False
    return True
def sort(x):
    while not isInOrder(x):
        random.shuffle(x)

x = range(1, int(argv[1]) + 1)
```

```
random.shuffle(x)
print x
sort(x)
print x
```

Note what is happening in the main routine. The user will run the program as follows.

`$ python bozo.py` *someNumber*

The command line argument *someNumber* is converted into an integer, say `n`. Then `range(1, n + 1)` is created and shuffled. Finally it is bozo sorted. At the unix command line, we can call `time` on it to see how long it takes.

Running it once looks like this.

```
time(python Bozo.py 1)
[1]
[1]

real    0m0.023s
user    0m0.019s
sys 0m0.003s
```

Now we show times 2-10, listing the timings and suppressing the rest of the output.

trial	real	user	sys
2	.0.023s	0.016s	0.006s
3	.0.024s	0.018s	0.004s
4	.0.023s	0.017s	0.005s
5	.0.024s	0.022s	0.001s
6	.0.025s	0.021s	0.003s
7	.0.057s	0.052s	0.003s
8	.0.358s	0.354s	0.001s
9	.3.164s	3.146s	0.004s
10	2m53.621s	2m51.808s	0.084s

Notice how the time to do the job rises dramatically beginning at about 7 trials. After 10 or 11 it becomes hopeless. This run was done on a fairly powerful machine.

3.2 The Bubble Sort

Now we will study a very simple sort, which for large data sets President Barack Obama has said, is not ideal. You can see his comments `http://www.youtube.`

com/watch?v=k4RRi_ntQc8 But it will work well enough for small arrays or lists.

Imagine you have a list like this one

```
4  5  1  3
```

Now insert a bookmark pointer `p` like so.

```
4  5  1  3
          p
```

This points off the end of the list. Next, we traverse the list, visiting each adjacent pair of entries. If the entries are in order, leave them undisturbed. Otherwise, swap them. We show this action. Notice that the first pair required no swapping, but the rest did.

```
4  5  1  3
4  5  1  3
4  1  5  3
4  1  3  5
```

Next, update the pointer by decrementing it.

```
4  1  3  5
        p
```

Here is the *loop invariant*, which is true after each pass of the loop.

1. Every item to the left of the pointer is smaller than those on the right. (this is vacuously true at the start)
2. Items to the right of the pointer are in order.

The next update of the pointer (check for yourself, Thomas) yields

```
1  3  4  5
    p
```

Now do this again to get

```
1  3  4  5
  p
```

The list is sorted. This sort is called the *bubble sort*, because the largest element in the list bubbles up to the top by the pointer on each pass of the loop. The loop invariants show why the list must be sorted at the end.

So let's see if we can code this algorithm in Python. We have a variable `p` that is an index or "pointer." It points off the end of the list so we begin like so. We will place this in file `Bub.py`

```
def sort(x):
    p = len(x)
```

We will do the swaps then decrement the pointer `p`. This will go on until `p==1` So we have

```
def sort(x):
    p = len(x)
    while p > 1:
        ##do the swaps
        p -= 1
```

Now we do the swaps. Let's fill that code in.

```
def sort(x):
    p = len(x)
    while p > 1:
        for k in range(p - 1):
            if x[k] > x[k + 1]:
                x[k], x[k+1] = x[k + 1], x[k]

        p -= 1
```

Now we add in some code to test our shiny new function.

```
import random
def sort(x):
    p = len(x)
    while p > 1:
        for k in range(p - 1):
            if x[k] > x[k + 1]:
                x[k], x[k+1] = x[k + 1], x[k]

        p -= 1
x = range(100)
random.shuffle(x)  ##shuffle  list
print x            #show unsorted
sort(x)
print x            #show sorted
```

This sorts the list. It happens quickly. You can see it is a big improvement over the hapless bozo sort.

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

Now let us conduct an analysis on this algorithm. Suppose the list has n elements in it. On the first pass, we do $n-1$ swaps on it, then we decrement the pointer, so there are n operations done. On the second pass, there are $n-1$ operations done. This continues until the pointer gets to the other end of the list. Altogether, we have

$$\sum_{k=1}^{n} k$$

operations we performed. There is a this well-known formula

$$\sum_{k-1}^{n} k = \frac{n(n+1)}{2} \sim \frac{n^2}{2}.$$

Therefore, this sort is $O(n^2)$; hence it is called a `quadratic sort`.

3.3 Can we do a Little Better?

Now we will look at another quadratic sort. We begin by illustrating the same first case as we did in the bubble sort.

```
 4  5  1  3
```

Now set a pointer to point at the zero index just before the first element.

```
 4  5  1  3
p
```

Everything to the left of the pointer is in order. All bets are off on the right. Begin by incrementing the pointer.

```
 4  5  1  3
  p
```

There is nothing to do. Increment again

```
4  5  1  3
    p
```

the 5 is in order on the left; we have no work to do. Increment again.

```
4  5  1  3
      p
```

The 1 is out of order. Swap it with the 5.

```
4  1  5  3
      p
```

The 1 remains out of order; swap it with the 4.

```
1  4  5  3
      p
```

The 1 is in place. We are done with this pass. Now increment the pointer.

```
1  4  5  3
      p
```

Now trickle the 3 down until it's in the right place

```
1  4  3  5
         p
```

then

```
1  3  4  5
        p
```

The pointer is at the end of the list. So our algorithm works as follows.

1. Have a pointer point immediately after the first item in the list.
2. Increment the pointer.
3. Do the "trickle-down" procedure on the element now to the left of the pointer by swapping it with its neighbor to the left until the neighbor to the left is smaller or until you arrive at the beginning of the list.
4. You are done when the list is all to the left of the pointer.

```java
public static void sortRWR(int[] x)
{
    int n = x.length;
    // control with a loop
    //   inside loop, increment pointer, trickle down
    for(int p = 0; p < n-1; p++)
    {
        for(int k = p + 1; k > 0 && x[k] < x[k-1]; k--)
        {
            swap(x, k-1, k);
        }
    }
}
```

Programming Problems

1. Perform a time profile on the bubble sort like we did on the bozo sort for various values of n. Make a table. Then plot log(n) on the x-axis and log(time) on the y-axis. What do you see?
2. Implement the second sorting algorithm using Python.
3. Perform a time profile on the second sorting algorithm like we did on the bubble sort for various values of n. Make a table. Then plot log(n) on the x-axis and log(time) on the y-axis. What do you see? Which of the two quadratic sorts works better.
4. If the list is already in order, compare the number of operations done by the bubble sort, the bozo sort, and second quadratic sort.

4 An Analysis of Methods in the Linked List and Array List Implementations

The standard java library package `java.util` contains both `ArrayList` and `LinkedList` classes. Having written little versions of these, we gain an understanding about their relative strengths and weaknesses. Likewise, we can examine our two stack implementations.

With our array-based stack, decreasingly periodic resizings must occur, but each resizing is much larger (either twice or 3/2 as large) as its predecessor.

For our link-based stack, when we push a new item on, we must do a finite series of operations; pushing a new item onto the stack is O(1) (constant cost), no matter how big the stack is. Popping is also O(1), as is peeking or checking if the stack is empty. Viewing the entire stack is $O(n)$, since we must do a print operation and a "nexting" for each link in the stack. However, this is basically

the same in the array-based stack. You must access each entry and then print it. It's a tie here.

The link based stack has no costly resizes, so it wins here. We do not need random access to the elements of the stack; we just interact with the top.

Now let us compare element access in the linked and array based lists. In an array based list, you are accessing an entry in an array. This is a short operation. Underneath you do this.

1. Fetch the memory address of the array.
2. move over the number of items indicated in the index
3. Fetch the integer there.

This is three operations. It's $O(1)$. In a link-based list, you must traverse as many elements in the list as your index indicates. This, typically will be about half-way. So this operation is $O(n)$. Access to elements is slower in link-based lists.

On the other hand, putting a new element at the head of a linked list is simple. It is a splicing operation with a few steps; it is $O(1)$. In the array-based list, you have to move every element over 1 to accommodate the new element. And that costs $O(n)$. So you can see here that there are no perfect tools and that there are design compromises.

An array-based implementation of a list has an array inside of it. When the list gets too big to fit in the array, we resize the array. This is not as simple as it sounds, as there will be problems caused by the type erasure mechanism in Java generics. Resizing is relatively costly because a lot of copying of elements is involved. Item access is $O(1)$, since we are just grabbing an item out of an array.

A link is a very simple data structure which holds a piece of data and which tells where to find the next piece of data. You will see that lists and stacks will be built of out links. This makes resizing unnecessary, but item access is $O(n)$, since we must traverse the whole list to hunt for an element. This, however, is no liability with a stack, since we only interact with the top of the stack.

Along the way, it will be necessary for us to gain a greater insight into the way Java generics work. This mechanism allows us to create classes that can contain any type of items of object type that we specify. We will begin this chapter by exploring the generic mechanism in enough detail for us to proceed.

5 The Merge Sort

The two sorts we have seen so far are called *quadratic sorts*, because they work in $O(n^2)$ time. As a result, sorting a list twice the size is takes four times as

long. There is something not quite right with this.

Imagine you have a list with n elements. Let's just make it into two mini-lists, each $n/2$ long. Now sort the two smaller lists. Each takes on the order of $n^2/4$ operations. We can then "zipper" them together. This is a quick operation; we shall see it is $O(n)$. So, our sorting process takes roughly

$$\frac{n^2}{4} + \frac{n^2}{4} + n + \frac{n^2}{2} + n$$

operations. So sorting the two half-lists and zippering saves nearly half of the work. So, we begin to think that we could even cut the halves in half and squeeze even more performance out of the procedure.

This is the idea behind the Merge Sort. Take note that a zippering procedure must occur that merges two sorted lists. Let us begin by developing that procedure.

Suppose we have two sorted lists such as these.

```
x -> [1, 2, 4, 8, 16, 32, 64, 128]
y -> [1, 3, 9, 27, 81, 243, 729, 2187]
```

We begin by pointing to the beginning of each list with pointers `p` and `q`. We will create a list `out` in which to store the combined list.

```
      p
x -> [1, 2, 4, 8, 16, 32, 64, 128]
y -> [1, 3, 9, 27, 81, 243, 729, 2187]
      q
out -> []
```

These pointers are list indices, so you should think of them as living between list elements. Now compare `x[p]` and `y[q]`; since they are equal, we will place `x[p]` on the combined list and then increment `p`. So we execute `out.append(x[p])` then `p += 1`. The situation now looks like this.

```
         p
x -> [1, 2, 4, 8, 16, 32, 64, 128]
y -> [1, 3, 9, 27, 81, 243, 729, 2187]
      q
out -> [1]
```

Next, notice that `x[p] < y[q]` so now we will add `y[q]` to the list and increment `q`.

```
         p
```

```
    x -> [1, 2, 4, 8, 16, 32, 64, 128]
    y -> [1, 3, 9, 27, 81, 243, 729, 2187]
            q
  out -> [1, 1]
```

Continuing in this fashion, we will arrive at the end of the list `x` first.

```
                                    p
    x -> [1, 2, 4, 8, 16, 32, 64, 128]
    y -> [1, 3, 9, 27, 81, 243, 729, 2187]
                          q
  out -> [1, 1, 2, 3, 4, 8, 9, 16, 27, 32, 64, 81, 128]
```

Once we hit the tail of one list, we can add in the tail of the other. In fact, since once list is empty, you can add in the tails of both. For one of the lists nothing will happen.

In this case, we add in the tail of `p`, and nothing happens. Then we chuck on the tail of `q`, and we are done. Things finish like this.

```
                                    p
    x -> [1, 2, 4, 8, 16, 32, 64, 128]
    y -> [1, 3, 9, 27, 81, 243, 729, 2187]
                          q
  out -> [1, 1, 2, 3, 4, 8, 9, 16, 27, 32, 64, 81, 128, 243, 729, 2187]
```

If we return `out` to the caller, we have successfully merged the two lists. Let us make a method for this in a Java class. You can try this in Python, too.

Let us begin by creating a class with a main method that includes a simple test.

```
import java.util.ArrayList;
public class Merge
{
    //x and y are sorted in order.  return a new array list that
    //zippers them together so the combined list is in order.
    public static void main(String[] args)
    {
        ArrayList<Integer> a = new ArrayList<Integer>();
        for(int k = 0; k < 8; k++)
            a.add((int) Math.pow(2,k));
        ArrayList<Integer> b = new ArrayList<Integer>();
        for(int k = 0; k < 8; k++)
            b.add((int) Math.pow(3,k));
        System.out.println(zip(a,b));
```

```
    }
    public static ArrayList<Integer>
        zip(ArrayList<Integer> x, ArrayList<Integer> y)
    {
        return null;
    }
}
```

Let us go to work on the zip method. Begin by creating an array list to hold the combined list. Let us specify the capacity in advance to avoid resizings. While we are it it, let's put in a return statement.

```
    ArrayList<Integer> out = new ArrayList<Integer>(x.size() + y.size());
    return out;
```

Now make `p` and `q`.

```
    ArrayList<Integer> out = new ArrayList<Integer>(x.size() + y.size());
    return out;
    int p = 0;
    int q = 0;
```

Now for the loop. Let us go until one list runs out.

```
    ArrayList<Integer> out = new ArrayList<Integer>(x.size() + y.size());
    return out;
    int p = 0;
    int q = 0;
    while(p < x.size() && q < y.size())
    {
        if(x[p] <= y[q])
        {
            out.add(x.get(p));
            p++;
        }
        else
        {
            out.add(y.get(q));
            q++;
        }
    }
    return out;
```

Finally, chuck on the tails for both lists. Note that one of these will be empty, because the predicate in the `while` loop is false.

```
    ArrayList<Integer> out = new ArrayList<Integer>(a.size() + b.size());
    return out;
    int p = 0;
    int q = 0;
    while(p < x.size() && q < y.size())
    {
        if(x[p] <= y[q])
        {
            out.add(x.get(p));
            p++;
        }
        else
        {
            out.add(y.get(q));
            q++;
        }
    }
    for(int k = p; k < x.size(); k++)
    {
        out.add(x.get(k));
    }
    for(int k = q; k < y.size(); k++)
    {
        out.add(y.get(k));
    }
    return out;

import java.util.ArrayList;
import java.util.Random;
public class Merge
{
    //x and y are sorted in order.  return a new array list that
    //zippers them together so the combined list is in order.
    public static void main(String[] args)
    {
        ArrayList<Integer> a = new ArrayList<Integer>();
        for(int k = 0; k < 8; k++)
            a.add((int) Math.pow(2,k));
        ArrayList<Integer> b = new ArrayList<Integer>();
        for(int k = 0; k < 7; k++)
            b.add((int) Math.pow(3,k));
        System.out.println(zip(a,b));
        a.clear();
        Random r = new Random();

        for(int k = 0; k < 200000; k++)
```

```
    {
        a.add(r.nextInt());
    }
    System.out.println(a);
    sort(a);
    System.out.println(a);

}
public static ArrayList<Integer> zip(ArrayList<Integer> x, ArrayList<Integer> y)
{
    ArrayList<Integer> out = new ArrayList<Integer>(x.size() + y.size());
    int p = 0;
    int q = 0;
    while (p < x.size() || q < y.size())
    {
        if(q >= y.size() || (p < x.size() && x.get(p)<y.get(q)))
        {
            out.add(x.get(p));
            p++;
        }
        else
        {
            out.add(y.get(q));
            q++;
        }
    }
    return out;
}
public static ArrayList<Integer> zipJM(ArrayList<Integer> x, ArrayList<Integer> y)
{
    ArrayList<Integer> out = new ArrayList<Integer>(x.size() + y.size());
    int p = 0;
    int q = 0;
    int combined = x.size() + y.size();
    for(int k = 0; k < combined; k++)
    {
        if(p < x.size() && x.get(p) < y.get(q))
        {
            out.add(x.get(p));
            p++;
        }
        if(q < y.size() && x.get(p) >= y.get(q))
        {
            out.add(y.get(q));
            q++;
        }
```

```
        }
        return out;
    }
    public static void sort(ArrayList<Integer> x)
    {
        //Idea: break list in two.  sort each half then zipper.
        //Note: if a list has length 0 or 1, it is SORTED.
        if(x.size() <= 1)
            return ;
        int n = x.size();
        ArrayList<Integer> first = new ArrayList<Integer>();
        ArrayList<Integer> second = new ArrayList<Integer>();
        int k;
        for(k = 0; k < n/2; k++)
            first.add(x.get(k));
        for(; k < n; k++)
            second.add(x.get(k));
        x.clear();
        sort(first);
        sort(second);
        for(Integer i: zip(first, second))
        {
            x.add(i);
        }
    }
}

import java.util.ArrayList;
import java.util.Random;
public class Merge
{
    //x and y are sorted in order.  return a new array list that
    //zippers them together so the combined list is in order.
    public static void main(String[] args)
    {
        ArrayList<Integer> a = new ArrayList<Integer>();
        for(int k = 0; k < 8; k++)
            a.add((int) Math.pow(2,k));
        ArrayList<Integer> b = new ArrayList<Integer>();
        for(int k = 0; k < 7; k++)
            b.add((int) Math.pow(3,k));
        System.out.println(zip(a,b));
        a.clear();
        Random r = new Random();

        for(int k = 0; k < 200000; k++)
```

```
        {
            a.add(r.nextInt());
        }
        System.out.println(a);
        sort(a);
        System.out.println(a);

    }
    public static ArrayList<Integer> zip(ArrayList<Integer> x, ArrayList<Integer> y)
    {
        ArrayList<Integer> out = new ArrayList<Integer>(x.size() + y.size());
        int p = 0;
        int q = 0;
        while (p < x.size() || q < y.size())
        {
            if(q >= y.size() || (p < x.size() && x.get(p)<y.get(q)))
            {
                out.add(x.get(p));
                p++;
            }
            else
            {
                out.add(y.get(q));
                q++;
            }
        }
        return out;
    }
    public static ArrayList<Integer> zipJM(ArrayList<Integer> x, ArrayList<Integer> y)
    {
        ArrayList<Integer> out = new ArrayList<Integer>(x.size() + y.size());
        int p = 0;
        int q = 0;
        int combined = x.size() + y.size();
        for(int k = 0; k < combined; k++)
        {
            if(p < x.size() && x.get(p) < y.get(q))
            {
                out.add(x.get(p));
                p++;
            }
            if(q < y.size() && x.get(p) >= y.get(q))
            {
                out.add(y.get(q));
                q++;
            }
```

```
        }
        return out;
    }
    public static void sort(ArrayList<Integer> x)
    {
        //Idea: break list in two.  sort each half then zipper.
        //Note: if a list has length 0 or 1, it is SORTED.
        if(x.size() <= 1)
            return ;
        int n = x.size();
        ArrayList<Integer> first = new ArrayList<Integer>();
        ArrayList<Integer> second = new ArrayList<Integer>();
        int k;
        for(k = 0; k < n/2; k++)
            first.add(x.get(k));
        for(; k < n; k++)
            second.add(x.get(k));
        x.clear();
        sort(first);
        sort(second);
        for(Integer i: zip(first, second))
        {
            x.add(i);
        }
    }
}
```