1.

```
INPUT: a, b, c                          From $ax^2+bx+c$

r1=0                                    These will ultimately form the
r2=0                                    solution, with r1 and i1 being
i1=0                                    the real and imaginary parts of
i2=0                                    the first root, and r2 and i2
                                        the parts of the second root.

if a=0 then                             First, deal with cases where a=0
       if b=0 then
              if c=0 then
                     OUTPUT "True"      0=0
              else
                     OUTPUT "False"     c=0 where c is nonzero constant
       else
              r1=-c/b                   Easy to find root.
              OUTPUT r1
       END                              Don't go to quadratic formula (it
                                        will be undefined).

d=b*b-4*a*c                             Find the determinant.

if d<0 then                             If the determinant is negative,
       r1=-b/(2*a)                      we get roots with imaginary
       i1=SQRT(ABS(d))/(2*a)            parts. We find the imaginary
       r2=-b/(2*a)                      parts without causing an error
       i2=-SQRT(ABS(d))/(2*a)           by using absolute value.
else
       r1=(-b+SQRT(d))/(2*a);           Otherwise, we can apply the
       r2=(-b-SQRT(d))/(2*a);           regular quadratic formula.

OUTPUT r1, i1, r2, i2                   Send answers in component form.
```

This algorithm deals with any real number inputs, even if they result in equations that are not really quadratic equations (like $2x-7=0$ and $3=4$). If the input is a quadratic equation, it returns the real and imaginary parts of both roots. If it is a linear equation, the one unique solution is returned. If a and b are both 0, which means the equation is in the form of $c=0$, then the equation is evaluated to true or false.

2.

```
INPUT: x, z, n                          x = number to find square root of
                                        z = initial guess
for i=1 to n do                         n = number of "improvements"—more
       z=0.5*(z + x/z)                  will result in a better guess.
```

At the end of this algorithm, *z* is the final guess. The algorithm really requires only one input: the number of which to find the square root. For instance, the algorithm could make an initial guess of *x*/2 and go through 20 iterations. However, the most efficient, most useful algorithm would ask the user for an initial guess and a number of iterations. The former allows the algorithm to get closer to the actual answer faster (assuming the initial guess is any good), and the latter allows one to get an idea of how accurate the final answer will be.

An alternative method uses *r* as a sort of cap on the error. In this algorithm, calculation continues until the improvement between two guesses is less than or equal to *r*. Thus, one can decide ahead of time the minimum benefit desired from another iteration of the algorithm.

```
INPUT: x, z, r                          x = number to find square root of
                                        z = initial guess
old=0                                   r = minimum amount of improvement
new=z                                   between guesses

while ABS(old-new)>r do
       old=new
       new=0.5*(new + x/new)
```

3. The simplest way to look at this problem is in terms of the number of black beans. Counting zero as even, the number must be either even or odd. Obviously, in order to change the parity of this number, we must add or remove an odd number of black beans. Ignoring the white beans, we now look at how each possible “grab” affects whether the number of black beans in the jar (we’ll call it *n*) is even or odd:

a) Two black beans: Two black beans are removed. *n*-2 black beans remain. Two is even. Parity is not affected.
b) Two white beans: Black beans are not affected, so *n* (or *n*-0) beans remain. Zero is even. Parity is not affected.
c) One of each: Black bean remains, so *n* (or *n*-0) beans remain. Zero is even. Parity is not affected.

Thus we see that there is no way for an even number of black beans to become an odd number of black beans, or vice versa.

Next, we consider the fact that at some point, there is one bean left. This bean must be either black or white. If it is black, then there is an odd number (1) of black beans left. Thus, we must have started with an odd number. If it is white, then there is an even number (0) of black beans left in the jar. Thus, we must have started with an even number. So, to summarize:

If there is an odd number of black beans in the jar to start, then a black bean will be the last one left. If there is an even number of black beans in the jar to start, then a white bean will be the last one left.

4. The following Java program achieves the lower bound of $\left\lceil \frac{3n}{2} - 2 \right\rceil$ comparisons to find the minimum and maximum of a list of numbers. It first divides the numbers into winners and losers, and then compares the winners to the winners and the losers to the losers. It eventually finds a minimum and a maximum. In fact it has a chance (50%, assuming the numbers are sorted randomly in the list) of making only $\left\lfloor \frac{3n}{2} - 2 \right\rfloor$ comparisons, if n is odd (see the comments for section [3] of the program).

The program also keeps track of how many comparisons are made, and displays that total along with the evaluation of $\left\lceil \frac{3n}{2} - 2 \right\rceil$.

Three examples of results can be found below the code.

```java
import java.io.*;

/** Class for finding the minimum number in an array, the maximum, or both at
once, all in the fewest possible comparisons */
class MinMax {

    //fields:
    private static int[] nums;
    private static int n;
    private static int c=0;

    //methods:
    public static void main(String[] args) throws IOException {
        BufferedReader stdin = new BufferedReader
            (new InputStreamReader(System.in));
        System.out.println("Enter the number of numbers:");
        String str = stdin.readLine();
        n = Integer.parseInt(str);
        nums = new int[n];

        if (n==0){
            System.out.println("Well, never mind then.");}
        else {
            System.out.println("Enter the numbers, one on each line:");

            for (int i = 0; i < n; i++){
                str = stdin.readLine();
                nums[i] = Integer.parseInt(str);}

            getBoth(nums, n);}
    }//method main()

    /** Finds and displays the largest and smallest integer from the array */
    public static void getBoth(int[] A, int size) {
        //init variables
        int i, max, min, maxListLength, minListLength, temp;
        int[] minList, maxList;
        double best;

        //[1] -- Definition of some variables
        temp = size/2;
        minList = new int[temp + 1]; //Add an extra element to MinList in case
        maxList = new int[temp];     //size is odd (see [3]).
        minListLength = size/2; //These variables will be updated later if
        maxListLength = size/2; //necessary (in the case of odd size).
```

```
//size of 1 will cause errors later, so deal with it separately
if (size==1)
{
    System.out.println("Max: " + A[0] + "\nMin: " + A[0]);
    System.out.println("No comparisons necessary");
    return;
}

//[2] -- Separate
//Main loop to separate into two groups ("winners" and "losers")
//Goes 2 at a time, and only to (size - 1) because it looks at the
//element after i, and there is no element after i=size. Thus,
//it skips the last element if size is odd. See [3].
for (i = 0; i < (size - 1); i += 2){
    if (greaterThan(A[i], A[i+1])){
        maxList[i/2] = A[i];
        minList[i/2] = A[i+1];
    }
    else {
        minList[i/2] = A[i];
        maxList[i/2] = A[i+1];
    }
}

//[3] -- Special Case: size is odd
//The following chunk of code deals with arrays containing an odd
//number of numbers.
if (size % 2 != 0){
    //If there is an odd number of numbers, grab the last number and
    //deal with it, because the loop didn't.

    //Check the last element against an element of maxList and copy
    //it to minList or maxList, as is appropriate.
    //This allows the previous for loop to ignore the last number.
    if (greaterThan(A[size - 1], maxList[0])){
        //The last element of A can replace the first of maxList
        //if the last element of A "wins," because it means that first
        //element of maxList is no longer a candidate for the maximum.
        //This will result in 3/2*n-2.5 comparisons by the end.
        maxList[0] = A[size - 1];
    }
    else {
        //If the last element of A loses, we add it to minList and
        //leave maxList alone. This will result in 3/2*n-1.5
        //comparisons.
        minList[minListLength] = A[size - 1];
        minListLength++;
    }
}
```

```
        //[4] -- Find Max and Min
        //Now, find the max of the "winners" and the min of the "losers":
        max = getMax(maxList, maxListLength);
        min = getMin(minList, minListLength);

        //[5] -- Display
        System.out.println("Max: " + max);
        System.out.println("Min: " + min);
        System.out.println("Comparisons: " + c);
        best = java.lang.Math.ceil(3*size/2.0-2);
        System.out.println("ceil(3/2*n-2): " + best);
    }//method getBoth()

    /** Essentially a wrapper for the ">" operation, but also keeps track of
    how many comparisons are formed via the class variable c */
    public static boolean greaterThan(int a, int b)
    {
        c++;
        return (a > b);
    }//method greaterThan()

    /** Returns the largest integer from the array */
    public static int getMax(int[] A, int size) {
        int i, max;
        max = A[0];

        for (i=1; i < size; i++){
            if (greaterThan(A[i], max)) max=A[i];}

        return max;
    }//method getMax()

    /** Returns the smallest integer from the array */
    public static int getMin(int[] A, int size) {
        int i, min;
        min = A[0];

        for (i=1; i < size; i++){
            if (greaterThan(min, A[i])) min = A[i];
        }

        return min;
    }//method getMin()

}//class MinMax
```

Below are three examples of the display after running the java program MinMax.

In the first (leftmost) example, ten numbers are compared. $\frac{3}{2}n-2$ works out to be an integer (13), so that is exactly how many comparisons the program makes.

In the remaining two examples, n is 7. Thus, $\frac{3}{2}n-2$ is not an integer, so there is some uncertainty as to how many comparisons it will take to find the maximum and minimum. In the center example, the "odd number out" compared in section [3] above is 13, and it is compared to 45, which is in the "never lost" group. Forty-five "wins," so it remains in its same group, and it just took $\frac{n-1}{2}+1=\frac{n}{2}+0.5$ comparisons to be left with still n numbers to eliminate, so it takes $\frac{3n}{2}-1.5=9=\left\lceil \frac{3n}{2}-2 \right\rceil$ comparisons total.

However, in the last example, the odd number out is 891, which is compared to 768 in the "never lost" group. This eliminates 768 (now it has lost), so after $\frac{n-1}{2}+1$ comparisons we have only $n-1$ numbers left to eliminate, so we end up with only $\frac{3n}{2}-2.5=8=\left\lfloor \frac{3n}{2}-2 \right\rfloor$ total comparisons. (For an explanation of this, see the last paragraph of answer 5).

```
[mimi] [~] java MinMax
Enter the number of
numbers:
10
Enter the numbers, one on
each line:
34
56
43
899
234
654
2
43
765
12
Max: 899
Min: 2
Comparisons: 13
ceil(3/2*n-2): 13.0
```

```
[mimi] [~] java MinMax
Enter the number of
numbers:
7
Enter the numbers, one
on each line:
12
45
76
390
99
65
13
Max: 390
Min: 12
Comparisons: 9
ceil(3/2*n-2): 9.0
```

```
[mimi] [~] java MinMax
Enter the number of
numbers:
7
Enter the numbers, one
on each line:
34
768
23
123
50
51
891
Max: 891
Min: 23
Comparisons: 8
ceil(3/2*n-2): 9.0
```

5. For this proof, we will call *A* the array of distinct, positive integers from which to find the largest and smallest and *n* the number of integers in the array. In any comparison, we consider the smaller number in the comparison to have "lost" and the larger number to have "won." This allows us to separate the numbers into three broad categories: "never won," "never lost," and "won and lost." Any number that has never been compared is in both the "never won" and "never lost" categories.

The first step in this proof is to establish the effect of any comparison between two numbers. A comparison counts as a win for one of the numbers and a loss for the other. Thus:

- A number that had never been compared has now either won or lost. It is moved to the "never won" category if it loses, or the "never lost" category if it wins.
- A number in the "never won" category stays in that category if it loses, and changes to the "won and lost" category if it wins.
- A number in the "never lost" category stays in that category if it wins, and changes to the "won and lost" category if it loses.
- A number in the "won and lost" category stays in the same category regardless of the outcome of the comparison.

Next we state that, at any point in the algorithm's execution, the maximum must be in the "never lost" category, and the minimum must be in the "never won" category (1). Proof: if the maximum has lost, it means some other number is greater than it, so it cannot be the maximum. If the minimum has won, then it is greater than some other number, so it cannot be the minimum.

We also state that any number still in the "never won" category is a candidate for the minimum, and any number still in the "never lost" category is a candidate for the maximum (2). Proof: two numbers in the "never won" category could not have been compared to each other (because comparing always results in a winner), and, if they were at any time both compared to some third number, they must have both lost, so no inferential conclusion can be drawn, either. Thus, neither can be eliminated as a candidate for minimum. The same thinking proves that any number in the "never lost" category is a candidate for maximum.

By statements (1) and (2), then, we know that in order to know for sure that it produces the maximum and minimum, an algorithm must continue comparisons until exactly one number has never lost, exactly one number has never won, and the rest have both won and lost.

At the beginning of any algorithm for finding the maximum and minimum of *A*, no comparisons have been made, so every number is in both the "never won" and "never lost" categories, and is thus a candidate for both maximum and minimum (Statement 2). Thus, every number must be compared at least once, or it will still be in both categories. Because one comparison compares two numbers, we know that it will require, at best, $\frac{n}{2}$ comparisons to compare every number once.

Any number that has only been compared once is in the "never won" or "never lost" category, and thus is still a candidate for either maximum or minimum (Statement 2). Thus, every number must be compared again. However, we keep in mind that we are trying to accomplish two separate tasks: finding the maximum and finding the minimum. After the initial comparison, these tasks may branch off for efficiency. This is because numbers in the "never won" or "never lost" category are each candidates for either the maximum or minimum, and the only other candidates for the same position are the other numbers in the same group (Statement 1). Thus, we need only compare numbers that have never lost to other numbers in the "never lost" group, and we need only compare numbers in the never won group to others in the "never won" group.

The algorithm's next step should then be to compare numbers in the same group. Each time this happens, one of the two numbers being compared stays in its current group, and the other number becomes a part of the "won and lost" group. From Statement 1, we know that we can discard (and never compare again) any number in the "won and lost" group, because it could be neither the minimum nor the maximum. Thus, provided the algorithm discards all "won and lost" numbers, every comparison results in one elimination, so x comparisons results in x eliminations. Thus, the number of numbers left (not discarded) is equal to n minus the number of comparisons (after the initial "phase").

Ultimately, all numbers except two must be eliminated. We just determined that at any point in the elimination phase $n - x$ numbers remain, where x is the number of comparisons. If $n - x = 2$, then $x = n - 2$. Thus, $n - 2$ comparisons must be performed to eliminate all numbers but two. Adding this to the best-case "separation phase," in which the algorithm compared every number once to group them into "never won" and "never lost," we get a minimum of $(n-2) + (\frac{n}{2}) = \frac{3n}{2} - 2$ comparisons.

One final note: in the case of n being odd, $\frac{3n}{2} - 2$ will not be an integer, so to have $\frac{3n}{2} - 2$ comparisons is impossible. Specifically, the initial grouping into winners and losers will take more than $\frac{n}{2}$ comparisons, for the following reason: At some point, there will be only one number left in the "never compared" group, so it must be compared to a number that is in the "never won" or "never lost" group. This results in $\frac{n-1}{2} + 1 = \frac{n}{2} + 0.5$ comparisons. However, this comparison may eliminate a number (for instance, if the "never compared" number loses to a "never won" number, the "never won" number is eliminated) or it may not (for instance, if the "never compared" number loses to a "never lost" number). Thus, there may remain $(n-1) - 2 = n - 3$ comparisons before the minimum and maximum are found, or there may remain $n - 2$ comparisons. This means that there may be $\left(\frac{n-1}{2} + 1\right) + (n-3) = \frac{3n}{2} - 2.5 = \left\lfloor \frac{3n}{2} - 2 \right\rfloor$ total comparisons, or there may be $\left(\frac{n-1}{2} + 1\right) + (n-2) = \frac{3n}{2} - 1.5 = \left\lceil \frac{3n}{2} - 2 \right\rceil$comparisons. There is no way to write an algorithm that is guaranteed to require one number of steps or the other, because it depends on what number the "odd number out" is compared to, and there is no way for an algorithm which doesn't know anything about that last number to pick which other number to compare it to. It can be guaranteed, though, not to be greater than the second case. Thus, we say that in general, the theoretical lower bound for being guaranteed to find the minimum and maximum of n numbers is $\left\lceil \frac{3n}{2} - 2 \right\rceil$ comparisons. QED.