CSCI 3501

Sorting competition

Groups 4 and 13

Sorting competition

Groups 4 and 13

Group 4

Joseph and Sean

Parsing in a sort of clever way

```java
private final static int[] reverseMod = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 9, 8, 7, 6, 5, 4, 3,
        2, 1, 0, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };

public static long parseLong(final String s) {
    long num = 0;
    num = (reverseMod[((s.charAt(2) - '0') + (s.charAt(3) - '0') + (s.charAt(4) - '0') + (s.charAt(5) - '0'))] + 10)
            * -1;
    num = num * 10 + '0' - s.charAt(2);
    num = num * 10 + '0' - s.charAt(3);
    num = num * 10 + '0' - s.charAt(4);
    num = num * 10 + '0' - s.charAt(5);
    num = num * 10 + '0' - s.charAt(6);
    num = num * 10 + '0' - s.charAt(7);
    num = num * 10 + '0' - s.charAt(8);
    num = num * 10 + '0' - s.charAt(9);
    num = num * 10 + '0' - s.charAt(10);
    return num * -1 + 10000000000L;
}
```

Examples:

“0.234890892” → 10^10 + (((-2-3-4-8)-1)%10)*10^9 + 234890892 == 12234890892
“0.916454238” → 10^10 + (((-9-1-6-4)-1)%10)*10^9 + 916454238 == 19916454238

Parsing in a sort of clever way

```java
private final static int[] reverseMod = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 9, 8, 7, 6, 5, 4, 3,
        2, 1, 0, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };

public static long parseLong(final String s) {
    long num = 0;
    num = (reverseMod[((s.charAt(2) - '0') + (s.charAt(3) - '0') + (s.charAt(4) - '0') + (s.charAt(5) - '0'))] + 10)
            * -1;
    num = num * 10 + '0' - s.charAt(2);
    num = num * 10 + '0' - s.charAt(3);
    num = num * 10 + '0' - s.charAt(4);
    num = num * 10 + '0' - s.charAt(5);
    num = num * 10 + '0' - s.charAt(6);
    num = num * 10 + '0' - s.charAt(7);
    num = num * 10 + '0' - s.charAt(8);
    num = num * 10 + '0' - s.charAt(9);
    num = num * 10 + '0' - s.charAt(10);
    return num * -1 + 10000000000L;
}
```

Examples:

“0.234890892” → 10^10 + (((-2-3-4-8)-1)%10)*10^9 + 234890892 == 12234890892
“0.916454238” → 10^10 + (((-9-1-6-4)-1)%10)*10^9 + 916454238 == 19916454238

The other part.

For both large and small data we implemented the caching of the previous value. This made it like 20 times faster.

```
    public static void sort(String[] data) {
        String prevString = "0.000000000";
        long prevNum = 10000000000L;

        long[] tosort = new long[data.length];
            for (int i = 0; i < data.length; i++) {
                tosort[i] = parseLong(data[i]);
            }
        Arrays.sort(tosort);

        if (data.length < 1500002) {
            for (int i = 0; i < data.length; i++) {
                if (!(prevNum == tosort[i])) {
                    prevString = "0." + Long.toString(tosort[i]).substring(2);
                    data[i] = prevString;
                    prevNum = tosort[i];
                } else {
                    data[i] = prevString;
                }
            }
        } else {
        Field f = null;
        try {
            f = String.class.getDeclaredField("value");
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
        }
        f.setAccessible(true);
        char[] value;
            for (int i = 0; i < data.length; i++) {
                if (prevNum == tosort[i]){
                data[i] = prevString;
                }else{
                data[i]=Long.toString(tosort[i]);
                try {
                    value = (char[])f.get(data[i]);
                    value[0]='0';
                    value[1]='.';
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
                prevString = data[i];
                prevNum = tosort[i];
                }
            }
        }
    }
```

```java
    public static void sort(String[] data) {
        String prevString = "0.000000000";
        long prevNum = 10000000000L;

        long[] tosort = new long[data.length];
            for (int i = 0; i < data.length; i++) {
                tosort[i] = parseLong(data[i]);
            }
        Arrays.sort(tosort);

        if (data.length < 1500002) {
            for (int i = 0; i < data.length; i++) {
                if (!(prevNum == tosort[i])) {
                    prevString = "0." + Long.toString(tosort[i]).substring(2);
                    data[i] = prevString;
                    prevNum = tosort[i];
                } else {
                    data[i] = prevString;
                }
            }
        } else {
        Field f = null;
        try {
            f = String.class.getDeclaredField("value");
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
        }
        f.setAccessible(true);
        char[] value;
            for (int i = 0; i < data.length; i++) {
                if (prevNum == tosort[i]){
                data[i] = prevString;
                }else{
                data[i]=Long.toString(tosort[i]);
                try {
                    value = (char[])f.get(data[i]);
                    value[0]='0';
                    value[1]='.';
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
                prevString = data[i];
                prevNum = tosort[i];
                }
            }
        }
    }
```

The other part.

For both large and small data we implemented the caching of the previous value. This made it about 20 times faster.

Small data handling

Next we looped back over the array doing “0.” + Long.toString().substring(2)

This was a little faster on small data, and almost the same speed on large data after garbage collection was forced. Our other approach was more consistently fast on larger data, and used a fraction of the memory.

```
for (int i = 0; i < data.length; i++) {
    if (!(prevNum == tosort[i])) {
        prevString = "0." + Long.toString(tosort[i]).substring(2);
        data[i] = prevString;
        prevNum = tosort[i];
    } else {
        data[i] = prevString;
    }
}
```

Large data handling

```java
Field f = null;
try {
    f = String.class.getDeclaredField("value");
} catch (NoSuchFieldException e) {
    e.printStackTrace();
}
f.setAccessible(true);
char[] value;
    for (int i = 0; i < data.length; i++) {
        if (prevNum == tosort[i]){
        data[i] = prevString;
        }else{
        data[i]=Long.toString(tosort[i]);
        try {
            value = (char[])f.get(data[i]);
            value[0]='0';
            value[1]='.';
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        prevString = data[i];
        prevNum = tosort[i];
        }
    }
}
```

Here we do the toString() then use reflections to change the first two chars in the Strings’ internal immutable char array.

Most of this code is just to make java not do errors as we abuse reflections.

Large data handling

```java
Field f = null;
try {
    f = String.class.getDeclaredField("value");
} catch (NoSuchFieldException e) {
    e.printStackTrace();
}
f.setAccessible(true);
char[] value;
    for (int i = 0; i < data.length; i++) {
        if (prevNum == tosort[i]){
        data[i] = prevString;
        }else{
        data[i]=Long.toString(tosort[i]);
        try {
            value = (char[])f.get(data[i]);
            value[0]='0';
            value[1]='.';
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        prevString = data[i];
        prevNum = tosort[i];
        }
    }
}
```

Here we do the toString() then use reflections to change the first two chars in the Strings' internal immutable char array.

Most of this code is just to make java not do errors as we abuse reflections.

Large data handling

```java
Field f = null;
try {
    f = String.class.getDeclaredField("value");
} catch (NoSuchFieldException e) {
    e.printStackTrace();
}
f.setAccessible(true);
char[] value;
    for (int i = 0; i < data.length; i++) {
        if (prevNum == tosort[i]){
        data[i] = prevString;
        }else{
        data[i]=Long.toString(tosort[i]);
        try {
            value = (char[])f.get(data[i]);
            value[0]='0';
            value[1]='.';
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        prevString = data[i];
        prevNum = tosort[i];
        }
    }
}
```

Here we do the toString() then use reflections to change the first two chars in the Strings' internal immutable char array.

Most of this code is just to make java not do errors as we abuse reflections.

Reflections

```
String s = "Hello World!";
char[] c = {'R','e','f','l','e','c','t','i','o','n','s','!'};
char[] c2 = {'H','e','l','l','o',' ','W','o','r','l','d','!'};
System.out.println(s);
editIntStr(s,c);
System.out.println(s);
```

```
Hello World!
Reflections!
```

```
System.out.println("Hello World!");
String s2 = "Reflections!";
editIntStr(s2,c2);
System.out.println(s2);
System.out.println("Reflections!");
System.out.println("Hello World!");
```

```
Reflections!
Hello World!
Hello World!
Reflections!
```

```
System.out.println(s2);
System.out.println("Reflections!");
System.out.println(s);
```

```
Hello World!
Hello World!
Reflections!
```

Reflections

```java
String s = "Hello World!";
char[] c = {'R','e','f','l','e','c','t','i','o','n','s','!'};
char[] c2 = {'H','e','l','l','o',' ','W','o','r','l','d','!'};
System.out.println(s);
editIntStr(s,c);
System.out.println(s);
```

```
Hello World!
Reflections!
```

```java
System.out.println("Hello World!");
String s2 = "Reflections!";
editIntStr(s2,c2);
System.out.println(s2);
System.out.println("Reflections!");
System.out.println("Hello World!");
```

```
Reflections!
Hello World!
Hello World!
Reflections!
```

```java
System.out.println(s2);
System.out.println("Reflections!");
System.out.println(s);
```

```
Hello World!
Hello World!
Reflections!
```

```java
System.out.println("false: " + "Reflections!".equals("Hello World!"));
System.out.println("true: " + "Hello World!".equals("Hello World!"));
System.out.println("false: " + "Hello World!".equals("Reflections!"));
System.out.println("false: " + "Reflections!".equals("Reflections!"));
System.out.println("false: " + "Reflections!".equals(s));
System.out.println("false: " + "Reflections!".equals(s2));
System.out.println("true: " + "Hello World!".equals(s));
System.out.println("true: " + "Hello World!".equals(s2));
```

```
false: false
true: true
false: false
false: true
false: false
false: true
true: true
true: false
```

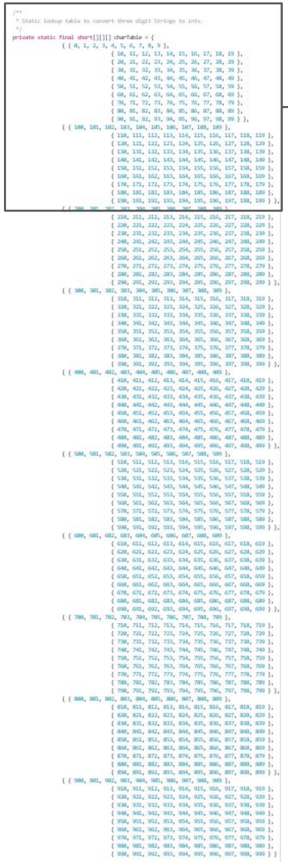
Efficiency

Our algorithm uses Arrays.sort() on an array of longs, which means that we’re effectively using a quicksort. This means that our worst case time is the same as quicksort: O(n^2). The times of all the data we sampled implies our algorithm has an average efficiency.

Group 13

Kevin Arhelger

Hard coding all the things



```java
/**
 * Static lookup table to convert three digit Strings to ints.
 */
private static final short[][][] charTable = {
                    { { 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 } },
                    { { 100, 101, 102, 103, 104, 105, 106, 107, 108, 109 },
                        { 110, 111, 112, 113, 114, 115, 116, 117, 118, 119 },
                        { 120, 121, 122, 123, 124, 125, 126, 127, 128, 129 },
                        { 130, 131, 132, 133, 134, 135, 136, 137, 138, 139 },
                        { 140, 141, 142, 143, 144, 145, 146, 147, 148, 149 },
                        { 150, 151, 152, 153, 154, 155, 156, 157, 158, 159 },
                        { 160, 161, 162, 163, 164, 165, 166, 167, 168, 169 },
                        { 170, 171, 172, 173, 174, 175, 176, 177, 178, 179 },
                        { 180, 181, 182, 183, 184, 185, 186, 187, 188, 189 },
                        { 190, 191, 192, 193, 194, 195, 196, 197, 198, 199 } },
                    { { 200, 201, 202, 203, 204, 205, 206, 207, 208, 209 },
```

Converting from strings to ints

```java
/**
 * Parse an int from a String of format.
 * 0.[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]
 *
 * @param s
 *            String to convert
 * @return integer value of the nine digits post decimal
 */
public static final int fastParseInt(final String s) {
        return charTable[s.charAt(2) - 48][s.charAt(3) - 48][s.charAt(4) - 48]
                        * 1000000
                        + charTable[s.charAt(5) - 48][s.charAt(6) - 48][s.charAt(7) - 48]
                        * 1000
                        + charTable[s.charAt(8) - 48][s.charAt(9) - 48][s.charAt(10) - 48];
}
```

Comparing our parsing

Comparing just the parts of our code that convert strings into longs that we sort on, ours was consistently faster than kevin's. (there's a good reason his is slower here)

(Times are the average of several runs)

Amount of data:	Kevin's time (ms)	Our time (ms)
1,000,000	69	36
5,000,000	258	121
10,000,000	536	250

Doing a fancy Radix Sort (using a fancy bucket sort).

```java
/**
 * More or less translated from this C example.
 * https://www.quora.com/What-is
 * -the-most-efficient-way-to-sort-a-million-32-bit-integers
 *
 * Generates a fair amount of garbage. 4*2^17 ints.
 *
 * 17 bit radix that we apply twice, swapping back and forth from temp.
 * We don't care about the bottom 19 bits so we just don't sort those.
 *
 * @param toSort
 *            the long array to sort
 * @param temp
 *            temporary array to store intermediate results.
 */
private static void rSort(long[] toSort, long[] temp) {
        int[] count = new int[0x20000];
        int[] bucket = new int[0x20000];
        int q = 0, i = 0, p = 0;

        for (p = 0; p < toSort.length; p++)
                count[((int) (toSort[p] >> 29) & 0x1FFFF)]++;
        for (i = 0; i < 0x20000; q += count[i++])
                bucket[i] = q;
        for (p = 0; p < toSort.length; p++) {
                temp[bucket[(int) ((toSort[p] >> 29) & 0x1FFFF)]++] = toSort[p];
        }

        count = new int[0x20000];
        bucket = new int[0x20000];
        q = 0;

        for (p = 0; p < toSort.length; p++)
                count[((int) (temp[p] >> 46) & 0x1FFFF)]++;
        for (i = 0; i < 0x20000; q += count[i++])
                bucket[i] = q;
        for (p = 0; p < toSort.length; p++) {
                toSort[bucket[(int) ((temp[p] >> 46) & 0x1FFFF)]++] = temp[p];
        }
}
```

Doing a fancy Radix Sort (using a fancy bucket sort).

```java
/**
 * More or less translated from this C example.
 * https://www.quora.com/What-is
 * -the-most-efficient-way-to-sort-a-million-32-bit-integers
 *
 * Generates a fair amount of garbage. 4*2^17 ints.
 *
 * 17 bit radix that we apply twice, swapping back and forth from temp.
 * We don't care about the bottom 19 bits so we just don't sort those.
 *
 * @param toSort
 *            the long array to sort
 * @param temp
 *            temporary array to store intermediate results.
 */
private static void rSort(long[] toSort, long[] temp) {
        int[] count = new int[0x20000];
        int[] bucket = new int[0x20000];
        int q = 0, i = 0, p = 0;

        for (p = 0; p < toSort.length; p++)
                count[((int) (toSort[p] >> 29) & 0x1FFFF)]++;
        for (i = 0; i < 0x20000; q += count[i++])
                bucket[i] = q;
        for (p = 0; p < toSort.length; p++) {
                temp[bucket[(int) ((toSort[p] >> 29) & 0x1FFFF)]++] = toSort[p];
        }

        count = new int[0x20000];
        bucket = new int[0x20000];
        q = 0;

        for (p = 0; p < toSort.length; p++)
                count[((int) (temp[p] >> 46) & 0x1FFFF)]++;
        for (i = 0; i < 0x20000; q += count[i++])
                bucket[i] = q;
        for (p = 0; p < toSort.length; p++) {
                toSort[bucket[(int) ((temp[p] >> 46) & 0x1FFFF)]++] = temp[p];
        }
}
```

Bringing it all together...

His comments explain the process pretty well.

basically what is happening

```java
/**
 * Bit manipulation FTW.
 *
 * Assign the bottom 29 bits to the location of the original string. Assign
 * the top bits to the inverse of group value. Assign the middle 30 bits to
 * the 0.XXXXXX value
 *
 * Sort (I doubt I can do much better than DualPivot Quicksort)
 *
 * Assign the original strings to sorted location using the bottom bits
 *
 * Copy
 *
 * This came from the thought, "Wouldn't it be nice not to build strings?"
 * Performance is slightly worse with the bin sorting I was using earlier, I
 * now just set the top bits to the bin number and run a standard sort.
 *
 * @param toSort
 *            String array to be sorted
 */
private static void ftlSort(final String[] toSort) {
        long[] temp = new long[toSort.length];
        long[] temp2 = new long[toSort.length];
        String[] trash = new String[toSort.length];
        int i, r, t;

        for (i = 0; i < toSort.length; i++) {
                t = fastParseInt(toSort[i]);
                r = t;
                // No assignments with the digits improves chances of optimization.
                // [%/] base 10 arithmetic is slow on most CPUs.
                t = (11 - (t / 100000 % 10 + t / 1000000 % 10 + t / 10000000 % 10 + t / 100000000 % 10) % 10);
                temp[i] = compose(t, r, i);
        }

        rSort(temp, temp2);

        for (i = 0; i < temp.length; i++) {
                trash[i] = toSort[(int) (temp[i] & 0x1FFFFFFF)];
        }
        System.arraycopy(trash, 0, toSort, 0, trash.length);
}
```

The fancy part where he does weird magic with bits.

```java
/**
 *
 * @param bin
 *            set the top bits to this value <= 15
 * @param value
 *            the XXXXXX int value ~999,999,999
 * @param index
 *            int ~500,000,000
 * @return composite of bin, value and index.
 */
private static long compose(final long bin, final long value,
        final long index) {
    return index + (value << 29) + (bin << 59);
}
```

```java
/**
 * Bit manipulation FTW.
 *
 * Assign the bottom 29 bits to the location of the original string. Assign
 * the top bits to the inverse of group value. Assign the middle 30 bits to
 * the 0.XXXXXX value
 *
 * Sort (I doubt I can do much better than DualPivot Quicksort)
 *
 * Assign the original strings to sorted location using the bottom bits
 *
 * Copy
 *
 * This came from the thought, "Wouldn't it be nice not to build strings?"
 * Performance is slightly worse with the bin sorting I was using earlier, I
 * now just set the top bits to the bin number and run a standard sort.
 *
 * @param toSort
 *            String array to be sorted
 */
private static void ftlSort(final String[] toSort) {
        long[] temp = new long[toSort.length];
        long[] temp2 = new long[toSort.length];
        String[] trash = new String[toSort.length];
        int i, r, t;

        for (i = 0; i < toSort.length; i++) {
                t = fastParseInt(toSort[i]);
                r = t;
                // No assignments with the digits improves chances of optimization.
                // [%/] base 10 arithmetic is slow on most CPUs.
                t = (11 - (t / 100000 % 10 + t / 1000000 % 10 + t / 10000000 % 10 + t / 100000000 % 10) % 10);
                temp[i] = compose(t, r, i);
        }

        rSort(temp, temp2);

        for (i = 0; i < temp.length; i++) {
                trash[i] = toSort[(int) (temp[i] & 0x1FFFFFFF)];
        }
        System.arraycopy(trash, 0, toSort, 0, trash.length);
}
```

Bringing it all together...

His comments explain the process pretty well.

basically what is happening

```
/**
 * Bit manipulation FTW.
 *
 * Assign the bottom 29 bits to the location of the original string. Assign
 * the top bits to the inverse of group value. Assign the middle 30 bits to
 * the 0.XXXXXX value
 *
 * Sort (I doubt I can do much better than DualPivot Quicksort)
 *
 * Assign the original strings to sorted location using the bottom bits
 *
 * Copy
 *
 * This came from the thought, "Wouldn't it be nice not to build strings?"
 * Performance is slightly worse with the bin sorting I was using earlier, I
 * now just set the top bits to the bin number and run a standard sort.
 *
 * @param toSort
 *            String array to be sorted
 */
private static void ftlSort(final String[] toSort) {
        long[] temp = new long[toSort.length];
        long[] temp2 = new long[toSort.length];
        String[] trash = new String[toSort.length];
        int i, r, t;

        for (i = 0; i < toSort.length; i++) {
                t = fastParseInt(toSort[i]);
                r = t;
                // No assignments with the digits improves chances of optimization.
                // [%/] base 10 arithmetic is slow on most CPUs.
                t = (11 - (t / 100000 % 10 + t / 1000000 % 10 + t / 10000000 % 10 + t / 100000000 % 10) % 10);
                temp[i] = compose(t, r, i);
        }

        rSort(temp, temp2);

        for (i = 0; i < temp.length; i++) {
                trash[i] = toSort[(int) (temp[i] & 0x1FFFFFFF)];
        }
        System.arraycopy(trash, 0, toSort, 0, trash.length);
}
```

LIES!!!

Bringing it all together...

His comments explain the process pretty well.

basically what is happening

```
/**
 * Bit manipulation FTW.
 *
 * Assign the bottom 29 bits to the location of the original string. Assign
 * the top bits to the inverse of group value. Assign the middle 30 bits to
 * the 0.XXXXXX value
 *
 * Sort (I doubt I can do much better than DualPivot Quicksort)
 *
 * Assign the original strings to sorted location using the bottom bits
 *
 * Copy
 *
 * This came from the thought, "Wouldn't it be nice not to build strings?"
 * Performance is slightly worse with the bin sorting I was using earlier, I
 * now just set the top bits to the bin number and run a standard sort.
 *
 * @param toSort
 *            String array to be sorted
 */
private static void ftlSort(final String[] toSort) {
        long[] temp = new long[toSort.length];
        long[] temp2 = new long[toSort.length];
        String[] trash = new String[toSort.length];
        int i, r, t;

        for (i = 0; i < toSort.length; i++) {
                t = fastParseInt(toSort[i]);
                r = t;
                // No assignments with the digits improves chances of optimization.
                // [%/] base 10 arithmetic is slow on most CPUs.
                t = (11 - (t / 100000 % 10 + t / 1000000 % 10 + t / 10000000 % 10 + t / 100000000 % 10) % 10);
                temp[i] = compose(t, r, i);
        }

        rSort(temp, temp2);

        for (i = 0; i < temp.length; i++) {
                trash[i] = toSort[(int) (temp[i] & 0x1FFFFFFF)];
        }
        System.arraycopy(trash, 0, toSort, 0, trash.length);
}
```

LIES!!!

Efficiency

Kevin’s algorithm uses a cool radix sort, using a bucket sort.

Both of these things are linear, making his algorithm’s worst case linear.