Problem D: Decoding EDSAC Data

Greater New York Region

October 30, 2011

There are two different approaches we might try for this problem. One uses an algorithm that works in any programming language. The other only works properly on certain types of hardware. We program the first in C++, the second in Java.

First Version: C++

The first thing to do is to set up the boilerplate: read P (the number of data sets), then read and process the data sets themselves.

1 ⟨*d.cpp* 1⟩≡

```
⟨includes [[C++]] 2c⟩
using namespace std;

⟨constants [[C++]] 3c⟩
⟨subsidiary function declarations [[C++]] 2e⟩
⟨subsidiary function definitions [[C++]] 2d⟩

int main()
{
    int p;
    cin >> p;
    for (int i = 0; i < p; i++) {
        ⟨read and process a single data set [[C++]] 2a⟩;
    }
    return 0;
}
```

This code is written to file `d.cpp`.

Each data set consists of N (the data set number) followed by an EDSAC instruction. The instruction consists of a single character prefix, `c`, followed by an unsigned decimal number, $0 \le \mathtt{d} < 2^{11}$, followed by a single character suffix, `s`, all on a single line. Fortunately, the `>>` operator skips leading whitespace, even when reading `char`'s.

2a ⟨*read and process a single data set* ⟦**C++**⟧ 2a⟩≡ (1)

```
int n; char c, s; unsigned d;
cin >> n;
cin >> c >> d >> s;
cin.ignore(100, '\n');  // skip rest of input line
⟨process data set #n consisting of c, d, and s ⟦C++⟧ 2b⟩
```

Processing a data set consists of printing the data set number, `n`, followed by a space, followed by the exact decimal fraction represented by the input instruction, We use a function, `e_convert`, to do the real work:

2b ⟨*process data set #*`n` *consisting of* `c`*,* `d`*, and* `s` ⟦**C++**⟧ 2b⟩≡ (2a)

```
cout << n << "␣" << e_convert(c, d, s) << endl;
```

Uses `e_convert` 2d.

2c ⟨*includes* ⟦**C++**⟧ 2c⟩≡ (1) 3b ▷

```
#include <iostream>
```

The `e_convert` function converts its three components (`prefix`, `address` and `suffix`) into the corresponding 17-bit EDSAC instruction, `e`, and uses `e_decimal` to turn `e` into a string representing the desired decimal fraction.

2d ⟨*subsidiary function definitions* ⟦**C++**⟧ 2d⟩≡ (1) 3d ▷

```
string e_convert(char prefix, unsigned address, char suffix)
{
    ⟨determine e, the equivalent EDSAC instruction ⟦C++⟧ 3a⟩;
    return e_decimal(e);
}
```

Defines:
`e_convert`, used in chunk 2.
Uses `e_decimal` 3d.

2e ⟨*subsidiary function declarations* ⟦**C++**⟧ 2e⟩≡ (1) 3e ▷

```
string e_convert(char, unsigned, char);
```

Uses `e_convert` 2d.

Converting a symbolic EDSAC instruction into its corresponding 17-bit integer value is simple: the 5-bit teleprinter code for `prefix` is concatenated to the 11-bit `address`, which is concatenated to a single bit (1 for `'D'`, 0 for `'F'`) `suffix`. Since `strchr` returns an address rather than an index, we subtract the string's initial address from its result to get the index we need.

3a ⟨*determine* `e`*, the equivalent* EDSAC *instruction* ⟦**C++**⟧ 3a⟩≡ (2d)

```
int e = ((strchr(COLLATING, prefix) - COLLATING) << 12)
    + (address << 1) + (suffix == 'D' ? 1 : 0);
```

3b ⟨*includes* ⟦**C++**⟧ 2c⟩+≡ (1) ◁2c

```
#include <cstring>
```

The 5-bit EDSAC teleprinter code gives a collating sequence of:

`PQWERTYUIOJ#SZK*?F@D!HNM&LXGABCV`

3c ⟨*constants* ⟦**C++**⟧ 3c⟩≡ (1)

```
#define COLLATING   "PQWERTYUIOJ#SZK*?F@D!HNM&LXGABCV"
```

The `e_decimal` function takes a 17-bit integer value, `e`, representing a binary fraction $-1 \leq$ `e` < 1, and converts it to a decimal string of the form *sb.ddd*..., where *s* is an optional minus sign, *b* is either a 1 or 0, and *d* is a decimal digit 0–9, with at most 16 digits after the decimal point. After dealing with the trivial cases 0.0 and −1.0, we append the prefix `"0."`* after a possible negative sign, followed by the digits of the fraction.

3d ⟨*subsidiary function definitions* ⟦**C++**⟧ 2d⟩+≡ (1) ◁2d

```
string e_decimal(int e)
{
    ⟨return results for trivial cases ⟦C++⟧ 4a⟩;
    string result = "";
    ⟨deal with possible negative prefix ⟦C++⟧ 4b⟩;
    result += "0.";
    ⟨append the appropriate decimal digits to result ⟦C++⟧ 4c⟩;
    return result;
}
```

Defines:
`e_decimal`, used in chunks 2d and 3e.

3e ⟨*subsidiary function declarations* ⟦**C++**⟧ 2e⟩+≡ (1) ◁2e

```
string e_decimal(int);
```

Uses `e_decimal` 3d.

*Note that only −1.0 (one of our trivial cases) can possibly have a digit other than 0 to the left of the decimal point.

The bit pattern `0x10000` represents the EDSAC value -1, and a zero bit pattern represents zero. Dealing with the former case here simplifies the code that follows it (as we can then assume that we have a zero to the left of the decimal point); dealing with the latter here insures that there will be at least one digit to the right of the decimal point.

4a ⟨*return results for trivial cases* ⟦**C++**⟧ 4a⟩≡ (3d)

```
if (e == 0) { return "0.0"; }
if (e == 0x10000) { return "-1.0"; }
```

Values with the 17th bit (`0x10000`) set are negative EDSAC values, and require a minus sign. To set the digits of a negative value properly, we need to negate the 16 fraction bits (which are currently the two's complement of the value we want).

4b ⟨*deal with possible negative prefix* ⟦**C++**⟧ 4b⟩≡ (3d)

```
if ((e & 0x10000) != 0) {   // negative #
    result = "-";
    e = -e & 0xffff;
}
```

It's simple to transform a binary fraction to a string of decimal digits: Multiplying the fraction by 10 puts the value of the fraction's most significant decimal digit to the left of the binary point (which we know is between the 16th and 17th bits). We simply convert the digit to a character and append it to our result. We then strip the decimal digit from our fraction, and repeat the process until we run out of digits.

4c ⟨*append the appropriate decimal digits to* `result` ⟦**C++**⟧ 4c⟩≡ (3d)

```
while (e != 0) {
    e *= 10;
    result += '0' + (e / 0x10000);
    e = e & 0xffff;
}
```

Second Version: Java

The second approach works only on floating point hardware with a binary mantissa and exponent. Of the allowable contest languages, only Java *guarantees* this to be true. Therefore, we code the second approach in Java.

The only real difference between the two versions (aside from contrastive naming conventions) is in the `eConvert` method, so we start with that. For this version, we simply convert the 17-bit EDSAC fraction into a Java `double`, and divide it by 2^{16}. The only tricky parts are "sign-extending" the 17-bit negative values to 32 bits, and using a `DecimalFormat` object to properly format the `double` value.

5a ⟨*subsidiary method definitions* **⟦Java⟧** 5a⟩≡ (6a)

```
private static String eConvert(char prefix, int address, char suffix) {
    ⟨determine e, the equivalent EDSAC instruction ⟦Java⟧ 5c⟩
    if ((e & 0x10000) != 0) { e |= 0xffff0000; }    // sign-extend
    double result = (double)e / 0x10000;
    return new DecimalFormat("0.0###############").format(result).trim();
}
```

Defines:
`eConvert`, used in chunk 6d.

5b ⟨*imports* **⟦Java⟧** 5b⟩≡ (6a) 6b ▷

```
import java.text.DecimalFormat;
```

From here on, everything works exactly as it does in the C++ version. Since the code that follows is (basically) just a Java translation of the C++ code, we don't bother to repeat the explanations.

5c ⟨*determine* `e`*, the equivalent* EDSAC *instruction* **⟦Java⟧** 5c⟩≡ (5a)

```
int e = (COLLATING.indexOf(prefix) << 12)
    + (address << 1) + (suffix == 'D' ? 1 : 0);
```

5d ⟨*constants* **⟦Java⟧** 5d⟩≡ (6a)

```
private final static String
    COLLATING = "PQWERTYUIOJ#SZK*?F@D!HNM&LXGABCV";
```

6a ⟨*D.java* 6a⟩≡

```
⟨imports [[Java]] 5b⟩
public class D {
     ⟨constants [[Java]] 5d⟩
     ⟨subsidiary method definitions [[Java]] 5a⟩

     public static void main(String[] args) {
          int p;
          Scanner in = new Scanner(System.in);
          p = in.nextInt();
          for (int i = 0; i < p; i++) {
               ⟨read and process a single data set [[Java]] 6c⟩
          }
     }
}
```

This code is written to file `D.java`.

6b ⟨*imports* **[[Java]]** 5b⟩+≡ (6a) ◁5b

```
import java.util.Scanner;
```

Here we have to do a little more work than in the C++ version to separate out `c`, `d`, and `s`, but it's pretty straightforward, nonetheless.

6c ⟨*read and process a single data set* **[[Java]]** 6c⟩≡ (6a)

```
int n; String data;
n = in.nextInt();
data = in.nextLine().trim();
int len = data.length();
char c = data.charAt(0);
int d = Integer.parseInt(data.substring(2, len - 2));
char s = data.charAt(len - 1);
⟨process data set #n consisting of c, d, and s [[Java]] 6d⟩
```

6d ⟨*process data set #*n *consisting of* `c`, `d`, *and* `s` **[[Java]]** 6d⟩≡ (6c)

```
System.out.println(n + "␣" + eConvert(c, d, s));
```

Uses `eConvert` 5a.

A Chunk Index

B Identifier Index