Math 374

Homework 1

Name: Gianluca Crescenzo

**Problem 1.** Compute the decimal (base 10) value for the following binary numbers.

(1) 10101100111000

(2) 0.110110110

(3) 100101110.01100101

*Solution.*

$$\begin{aligned} 10101100111000_2 &= 2^{13} + 2^{11} + 2^9 + 2^8 + 2^5 + 2^4 + 2^3 \\ &= 302.39453125_{10} \end{aligned}$$

$$\begin{aligned} 0.110110110_2 &= 2^{-1} + 2^{-2} + 2^{-4} + 2^{-5} + 2^{-7} + 2^{-8} \\ &= 0.85546875_{10} \end{aligned}$$

$$\begin{aligned} 100101110.01100101_2 &= 2^8 + 2^5 + 2^3 + 2^2 + 2^1 + 2^{-2} + 2^{-3} + 2^{-6} + 2^{-8} \\ &= 302.39453125_{10} \end{aligned}$$

**Problem 2.** Compute the binary form of the following decimal numbers. Write 10 digits to the right of the binary point.

(1) 1272025.3255

(2) $\frac{3141592}{65}$

(3) $\frac{1}{7}$

*Solution.* I used Mathematica code for this part, because division on pen/paper/calculator was frustrating.

(1) The integer part is:

```
In[1]:= Clear[n, f, q, r, bits, digit, s]
        n = 1272025;
        bits = "";
        While[n > 0, {q, r} = QuotientRemainder[n, 2];
            bits = ToString[r] <> bits;
        n = q;];
        bits

Out[1]= 100110110100011011001
```

The fractional part is:

In[2]:=
```
Clear[n, f, q, r, bits, digit, s]
f = 0.3255;
bits = "";
Do[s = 2*f;
    digit = Floor[s];
    bits = bits <> ToString[digit];
    f = s - digit;
    , {10}];
bits
```

Out[2]=
```
0101001101
```

Thus $1272025.3255_{10} = 100110110100011011001.0101001101_2$.

(2) Note that $\frac{3141592}{65} = 48332.1\overline{846153}$. It's probably safer to write this as a mixed fraction, since I'm not sure if truncating the fractional part will cause problems. Changing the code a bit gives us:

In[3]:=
```
Clear[n, ibits, fbits, q, r, f, s, digit]
n = 3141592/65;
i = IntegerPart[n];
f = FractionalPart[n];
ibits = "";
fbits = "";
While[i > 0,
    {q, r} = QuotientRemainder[i, 2];
    ibits = ToString[r] <> ibits;
    i = q;];
Do[s = 2*f;
    digit = Floor[s];
    fbits = fbits <> ToString[digit];
    f = s - digit;
    , {10}];
ibits <> "." <> fbits
```

Out[3]=
```
1011110011001100.0010111101
```

Whence $\frac{3141592}{65}_{10} = 1011110011001100.0010111101_2$.

(3) Computing $\frac{1}{7}$ isn't as frustrating. We can see that:

$$
\begin{array}{ll}
2 * \frac{1}{7} & \\
2 * \frac{2}{7} & r = 0 \\
2 * \frac{4}{7} & r = 0 \\
2 * \frac{1}{7} & r = 1 \\
2 * \frac{2}{7} & r = 0 \\
2 * \frac{4}{7} & r = 0 \\
2 * \frac{1}{7} & r = 1 \\
\vdots &
\end{array}
$$

This repeats forever. Only writing 10 digits to the right of the binary point, we see that $\frac{1}{7}_{10} = 0.0010010010_2$.

**Problem 3.** Determine the Binary16 (half precision), Binary32 (single precision), and Binary64 (double precision) bit patterns for the number $\pi$. Express your answers in both binary and hexadecimal form.

*Solution.* Note that $\pi \approx 3.14159265358979323846 = 2^1 \cdot 1.57079632679489661922$. I made a lot of changes to the above code to make it work with floating point precision.

```
In[4]:= ClearAll[exponentBits, mantissaBits];

        exponentBits[e_, bitLength_] :=
         Module[{ebits = "", q, r, localE = e},
          Do[{q, r} = QuotientRemainder[localE, 2];
           ebits = ToString[r] <> ebits;
           localE = q;, {bitLength}];
          ebits]

        mantissaBits[f_, bitLength_] :=
         Module[{mbits = "", digit, s, localF = f},
          Do[s = 2*localF;
           digit = Floor[s];
           mbits = mbits <> ToString[digit];
           localF = s - digit;, {bitLength}];
          mbits]

        ClearAll[Binary];
        Binary[totalBits_, val_] :=
         Module[{sbit, eBits, mBits, bias, e, f, n, counter},
          Switch[totalBits,
           16, {eBits = 5; mBits = 10; bias = 15;},
           32, {eBits = 8; mBits = 23; bias = 127;},
```

```
 64, {eBits = 11; mBits = 52; bias = 1023;},
 128, {eBits = 15; mBits = 112; bias = 16383;},
 256, {eBits = 19; mBits = 236; bias = 262143;},
 _, Return["unsupported size"]];

sbit = If[val < 0, "1", "0"];
n = Abs[val];
If[n < 2, e = bias;
 f = n - 1;,
 counter = 0;
 While[n >= 2, n = n/2;
  counter++;];
 e = counter + bias;
 f = n - 1;];
sbit <> exponentBits[e, eBits] <> mantissaBits[f, mBits]]
```

For Binary16, our exponent is going to be $e = 1 + 15 = 16_{10} = 10000_2$, our mantissa is going to be $.57079632679489661922_{10} \approx 1001001000_2$, and our sign bit is going to be 0. Binary32 and Binary64 follow similarly.

```
In[5]:= Binary[16, N[Pi, 23]]
        Binary[32, N[Pi, 23]]
        Binary[64, N[Pi, 23]]

Out[5]= 0100001001001000

Out[6]= 01000000010010010000111111011010

Out[7]= 0100000000001001001000011111101101010100010001000010110100011000
```

Splitting the bit string into groups of four allows us to express our answer in hexademical. So:

$$\begin{aligned} \text{Binary16} &: \pi_{10} \approx 4248_{16} \\ \text{Binary32} &: \pi_{10} \approx 40490\text{FDA}_{16} \\ \text{Binary64} &: \pi_{10} \approx 400921\text{FB}54442\text{D}18_{16} \end{aligned}$$

**Problem 4.** Determine the Binary64, Binary128, and Binary256 bit patterns for the number $\frac{127}{128}$. Express your answer in both binary and hexadecimal form.

*Solution.* Note that $\frac{127}{128} = 0.9921875 = 2^{-1} \cdot 1.984375$. I adjusted the previous code so that it can account for values less than 1.

```
In[8]:= sbit = If[val < 0, "1", "0"];
        n = Abs[val];
        counter = 0;
        While[n >= 2, n = n/2;
          counter++;];
        While[n < 1, n = 2*n;
          counter--;];
        e = counter + bias;
        f = n - 1;
        sbit <> exponentBits[e, eBits] <> mantissaBits[f, mBits]
```

```
In[9]:= Binary[64, 127/128]
        Binary[128, 127/128]
        Binary[256, 127/128]

Out[9]= 0011111111101111110000000000000000000000000000000000000000000000

Out[10]= 0011111111111110111111000000000000000000000000000000000000000000000000
             0000000000000000000000000000000000000000000000000000000000

Out[11]= 0011111111111111111011111100000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
```

We can now easily see that:

$$
\begin{aligned}
\text{Binary16}: \quad \frac{127}{128}_{10} &= \text{3FEFC}\,\underbrace{0...0}_{11} \\
\text{Binary32}: \quad \frac{127}{128}_{10} &= \text{3FFEFC}\,\underbrace{0...0}_{26} \\
\text{Binary64}: \quad \frac{127}{128}_{10} &= \text{3FFFEFC}\,\underbrace{0...0}_{57}
\end{aligned}
$$

**Problem 5.** Determine the largest positive double precision number $x_1$ and the next largest positive double precision number $x_2$. What is the difference between these two numbers?

*Solution.* Since $e = 11111111111_2$ is used to represent $\infty$, the largest positive double precision number is:

$$
\begin{aligned}
x_1 &= 0\ 11111111110\ 1111111111111111111111111111111111111111111111111111_2 \\
&= (-1)^0 \cdot 2^{2046-1023} \cdot \left(1 + \left(\frac{1}{2} + \frac{1}{4} + \ldots + 2^{-52}\right)\right) \\
&= 2^{1023} \cdot (1 + (1 - 2^{-53})) \\
&= 2^{1023} \cdot (2 - 2^{-53})_{10} \\
&\approx 1.7976931348623158\ldots \times 10^{308}.
\end{aligned}
$$

The next largest positive double precision number, $x_2$, would be:

$$
\begin{aligned}
x_2 &= 0\ 11111111110\ 1111111111111111111111111111111111111111111111111110_2 \\
&= 2^{1023} \cdot (2 - 2^{-52}) \\
&\approx 1.7976931348623157\ldots \times 10^{308}.
\end{aligned}
$$

So:

$$
\begin{aligned}
x_1 - x_2 &= 2^{1023} \cdot (2 - 2^{-53}) - 2^{1023} \cdot (2 - 2^{-52}) \\
&= 2^{1023} \cdot (2 - 2^{-53} - 2 + 2^{-52}) \\
&= 2^{1023} \cdot (2^{-52} - 2^{-53}) \\
&\approx 9.9792015476735990\ldots \times 10^{291}.
\end{aligned}
$$

**Problem 6.** Determine the smallest positive double precision number $x_1$ and the next smallest positive double precision number $x_2$. What is the difference between these two numbers?

*Solution.* Since $e = 00000000000_2$ is used to represent subnormal numbers, $x_1$ in base 10 is going to have a slightly different formula. Instead of the mantissa being preceded by a 1, it is instead preceded by a 0. So we have:

$$\begin{aligned} x_1 &= 0\ 00000000000\ 0000000000000000000000000000000000000000000000000001 \\ &= (-1)^0 \cdot 2^{1-1023} \cdot (0 + 2^{-52}) \\ &= 2^{-1074}. \end{aligned}$$

The next smallest positive double precision number, $x_2$, would be:

$$\begin{aligned} x_2 &= 0\ 00000000000\ 0000000000000000000000000000000000000000000000000010 \\ &= 2^{-1022} \cdot (0 + 2^{-51}) \\ &= 2^{-1073} \end{aligned}$$

So:

$$\begin{aligned} x_1 - x_2 &= 2^{-1074} - 2^{-1073} \\ &\approx -4.9406564584124659... \times 10^{-324} \end{aligned}$$

**Problem 7.** In class, we saw that the way two binary numbers are added, was that for each bit we employed the logic functions:

$$\begin{aligned} s_i &= x_i \oplus y_i \oplus \mathrm{cin}_i \\ \mathrm{cout}_i &= (x_i \wedge y_i) \vee (x_i \wedge \mathrm{cin}_i) \vee (y_i \wedge \mathrm{cin}_i). \end{aligned}$$

Create the logic functions $s_i(x_i, y_i, \mathrm{bin}_i)$ and $\mathrm{bout}_i(x_i, y_i, \mathrm{bin}_i)$, for performing bit by bit subtractions, where $\mathrm{bin}_i$ and $\mathrm{bout}_i$ are "borrow" bits.

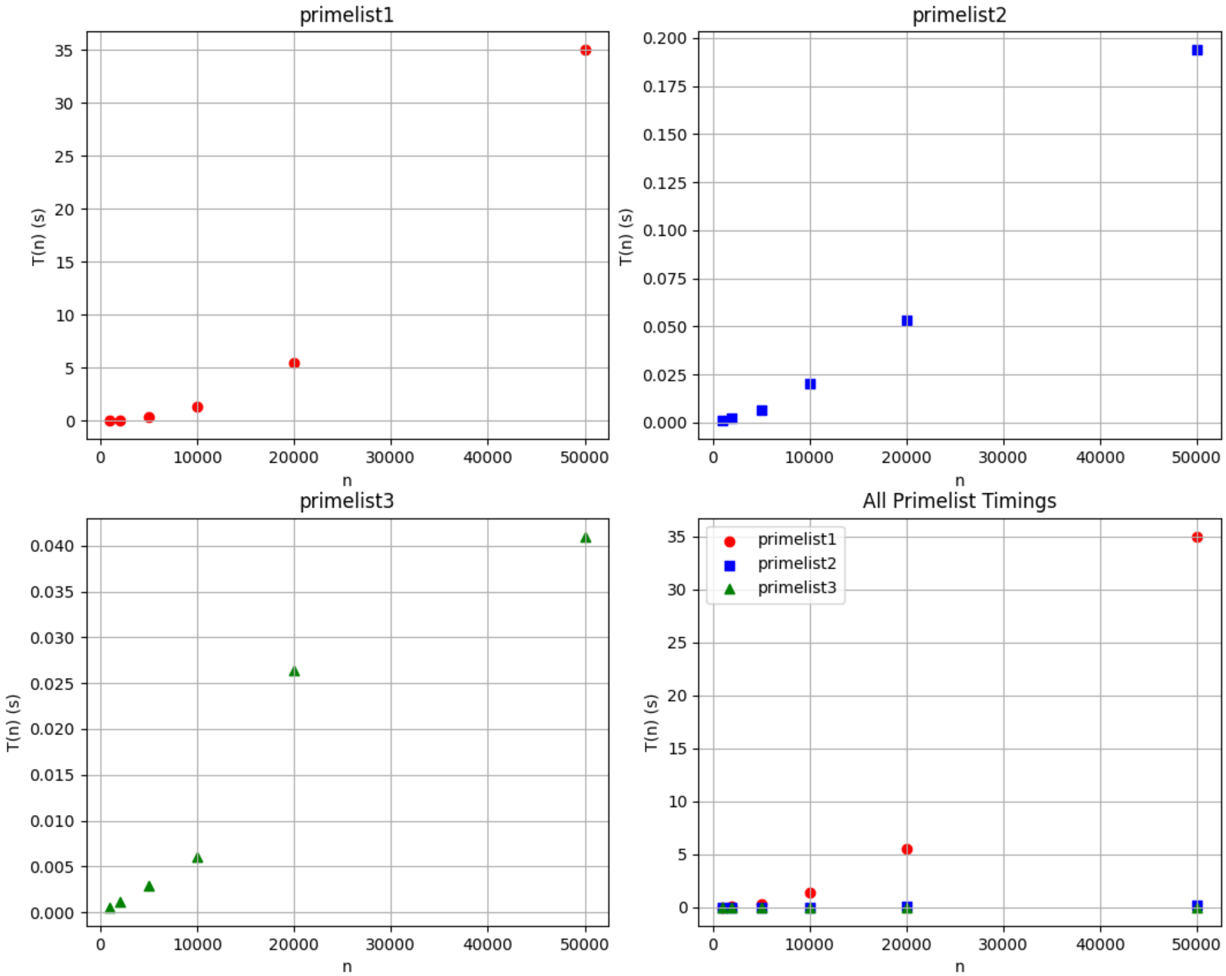
*Solution.* From the table:

| $x$ | $y$ | $b_{in}$ | $s_i$ | $b_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

We have that $s_i(x_i, y_i, b_{in}) = x_i \oplus y_i \oplus b_{in}$ and $b_{out}(x_i, y_i, b_{in}) = (\overline{x_i} \wedge y_i) \vee (\overline{x_i} \wedge b_{in}) \vee (y_i \wedge b_{in})$

**Problem 8.** Choose three prime number algorithms to locate all prime numbers from 2 to $n$. Run each of these for values of $n = 1000, 2000, 5000, 10000, 20000, 50000$. For each algorithm, plot the time $T(n)$ v.s. $n$ on a scatter plot. Comment on the relative efficiency of each of your three algorithms.

*Solution.* I switched to Python, since Mathematica was too slow. Primelist3 was the fastest, followed by primelist2. Primelist1 was much slower.


primelist1
primelist2
primelist3
All Primelist Timings
T(n) (s)
n
primelist1
primelist2
primelist3
0
10000
20000
30000
40000
50000
0
5
10
15
20
25
30
35
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
0.040

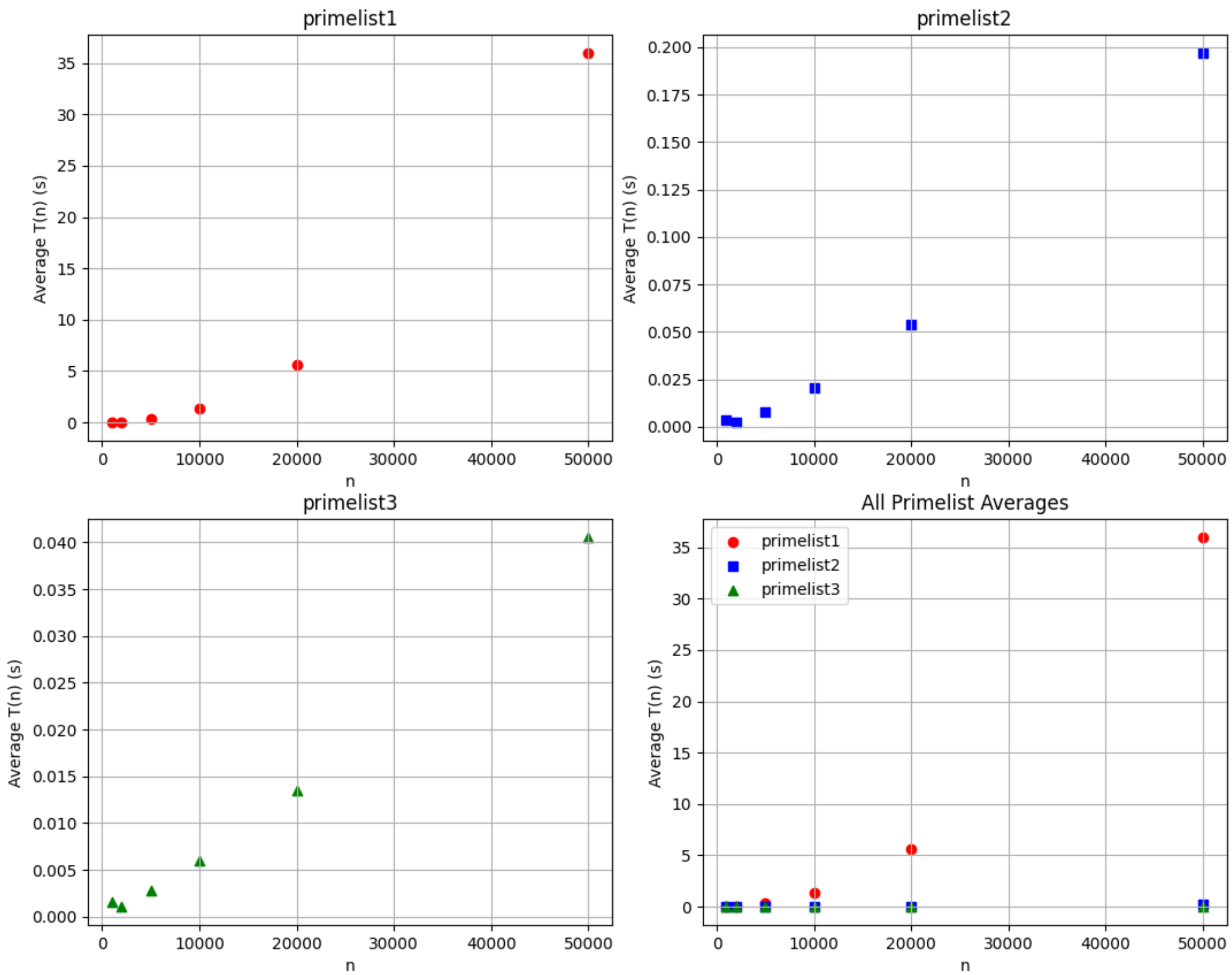
Here is the code that I used:

```
import time
import matplotlib.pyplot as plt

def primelist1(n):
    if n < 2:
        return
    print(2)
    for j in range(3, n):
        isprime = True
        for i in range(2, j-1):
            if j % i == 0:
                isprime = False
        if isprime:
            print(j)

def primelist2(n):
    if n < 2:
        return
    print(2)
    for j in range(3, n):
        isprime = True
        for i in range(2, int(j**0.5)):
            if j % i == 0:
                isprime = False
        if isprime:
            print(j)

def primelist3(n):
    if n < 2:
        return
    print(2)
    for j in range(3, n):
        isprime = True
        for i in range(2, int(j**0.5)):
            if j % i == 0:
                isprime = False
                break
        if isprime:
            print(j)

n_values = [1000, 2000, 5000, 10000, 20000, 50000]

timingData1 = []
timingData2 = []
timingData3 = []

for n in n_values:
    start = time.time()
    primelist1(n)
    end = time.time()
    timingData1.append(end - start)

    start = time.time()
    primelist2(n)
    end = time.time()
    timingData2.append(end - start)

    start = time.time()
    primelist3(n)
    end = time.time()
```

```
    timingData3.append(end - start)

fig, axs = plt.subplots(2, 2, figsize=(12, 10))

axs[0,0].scatter(n_values, timingData1, color='red', marker='o')
axs[0,0].set_title('primelist1')
axs[0,0].set_xlabel('n')
axs[0,0].set_ylabel('T(n) (s)')
axs[0,0].grid(True)

axs[0,1].scatter(n_values, timingData2, color='blue', marker='s')
axs[0,1].set_title('primelist2')
axs[0,1].set_xlabel('n')
axs[0,1].set_ylabel('T(n) (s)')
axs[0,1].grid(True)

axs[1,0].scatter(n_values, timingData3, color='green', marker='^')
axs[1,0].set_title('primelist3')
axs[1,0].set_xlabel('n')
axs[1,0].set_ylabel('T(n) (s)')
axs[1,0].grid(True)

axs[1,1].scatter(n_values, timingData1, color='red', marker='o', label='primelist1
                                          ')
axs[1,1].scatter(n_values, timingData2, color='blue', marker='s', label='
                                          primelist2')
axs[1,1].scatter(n_values, timingData3, color='green', marker='^', label='
                                          primelist3')
axs[1,1].set_title('All Primelist Timings')
axs[1,1].set_xlabel('n')
axs[1,1].set_ylabel('T(n) (s)')
axs[1,1].legend()
axs[1,1].grid(True)

plt.tight_layout()
plt.show()
```

**Problem 9.** Repeat the question above for three prime number algorithms ...except that now, each T(n) represents the average of times over five similar trials. Create the same three scatter plots as you did above. Do you notice any difference? If so, explain what just happened.

*Solution.* The averages look very similar to the previous question.




I made the following changes to the above code:

```
n_values = [1000, 2000, 5000, 10000, 20000, 50000]

num_trials = 5

for n in n_values:
total_time_1 = 0.0
total_time_2 = 0.0
total_time_3 = 0.0

for _ in range(num_trials):
    start = time.time()
    primelist1(n)
    end = time.time()
    total_time_1 += (end - start)

    start = time.time()
    primelist2(n)
```

```
    end = time.time()
    total_time_2 += (end - start)

    start = time.time()
    primelist3(n)
    end = time.time()
    total_time_3 += (end - start)

avg_time_1 = total_time_1 / num_trials
avg_time_2 = total_time_2 / num_trials
avg_time_3 = total_time_3 / num_trials

timingData1.append(avg_time_1)
timingData2.append(avg_time_2)
timingData3.append(avg_time_3)
```