Homework 2

Assigned: 9/14

Due: 9/20, 11:59pm

Possible Points: 100

Submit on Canvas a zip or zipped tar file containing your solution code for each problem (just the cpp files, please don't submit the solution/project files), turn in one cpp file for each problem. If a problem has multiple parts, it should still be one file. Each problem specifies how you should name the file containing your solution so it's easy for us to find.

Since the assignment will be graded using an automated grading system you should have your program's output match the sample output shown since you may lose points. If your program fails to compile using Visual Studio 2017 it will receive zero points. If your program compiles but fails to run properly (runtime errors or wrong output) points will be deducted based on how correct the program is. For example, if you have a small typo and the output does not match exactly, it will lead only to minor deduction from grade.

Students are expected to write their source code from scratch, those who copy code from each other or online (including from Canvas examples) will be reported for cheating.

Note: We will also look at your code so don't just hard-code the answer and print it.

1. Flip Flop: 10 points

File: `flip_flop.cpp`

We want to model the behavior of a strange sort of fish over some time. On seconds divisible by a it flips, on those divisible by b it flops and on those divisible by both it flips and flops. To simulate this behavior your program should print "flip" when it flips, "flop" when it flops and "flipflop" when it flips and flops. If the fish doesn't do any action then you should just print out the current second.

Input to your program for each case will be the second to start at, the number of seconds to simulate and the values for a and b. Your program should then log the behavior of the fish by printing its action or the current time as specified above for the desired number of seconds. Note that 0 counts as divisible by any number, recall that `0 % x` is always 0 for any value of `x`.

Example Input

```
3
0 16 3 5
```

```
10 10 2 7
4 20 12 4
```

Expected Output

```
Case 0:
flipflop
1
2
flip
4
flop
flip
7
8
flip
flop
11
flip
13
14
flipflop
Case 1:
flip
11
flip
13
flipflop
15
flip
17
flip
19
Case 2:
flop
5
6
7
flop
9
10
11
flipflop
13
14
15
flop
```

```
17
18
19
flop
21
22
23
```

---

2. Worth Every Penny: 30 points

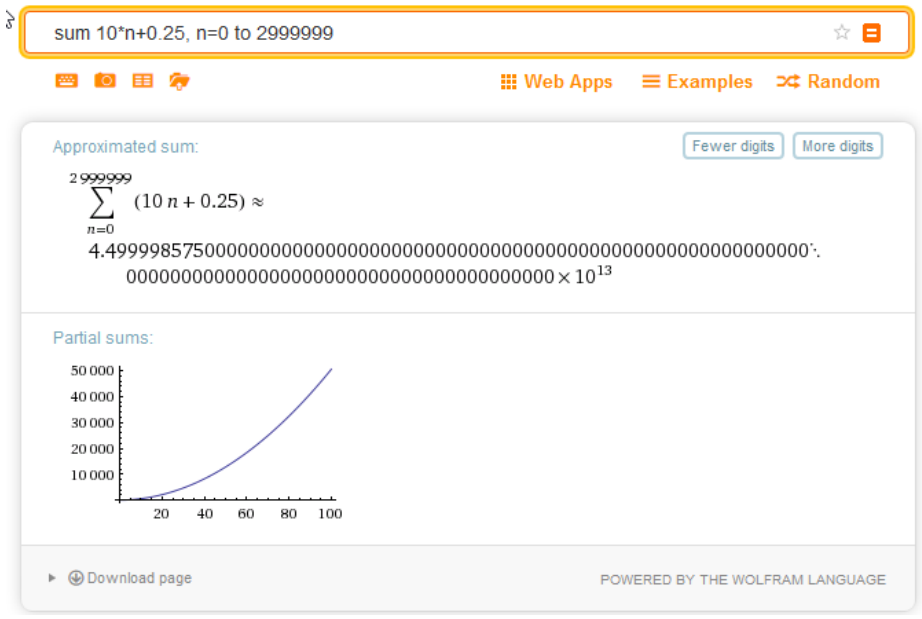
File: `worth_every_penny.cpp`

You are running a business of selling stainless-steel wedding rings. Bill, the software engineer you hired to program your business software is an experienced programmer, so he decided to use double-precision floating-point numbers (`double`) for all amounts of money in the software, fearing that one day a puny `float` would run out of precision to represent your billion-dollar fortune. Being a brilliant businessman yourself, you have come up with a unique pricing strategy: the first ring is sold for 25 cents ($0.25), the second one for $10.25, the third one $20.25 and so on. In other words, the price for each ring is $10 more than the last. This plan has proven to work quite well: over the years you have sold 30'000'000 (thirty million) rings, and every penny earned was recorded in the software. The profit from selling all these rings is $4499999857500000. This result can be verified using, for example, Wolfram Alpha. See the following screenshot for how to use Wolfram Alpha to evaluate this number.

Yet the software written by Bill surprisingly reports a different profit value. Obviously you are not happy and decide to take the matters into your own hands.

In this problem you will write a program to compute the exact total profit after the $n - 1$th ring is sold. n will be given to your program as input, and it will be an integer between 1 and 30000000, inclusively. For example, if $n = 3$, the profit should be $0.25 + 10.25 + 20.25 = 30.75$. In addition to the exact total profit, your program should also output the number computed by Bill's program (remember that he used double-precision floating-point numbers for all his calculations). Print your output in dollars, in fixed point notation, to up to 2 digits after the decimal point. You can use `std::fixed` and `std::setprecision` in the `<iomanip>` header file. For example, to output the variable `num` in fixed point notation, to 2 digits of accuracy after the decimal point, do as follows: `std::cout << std::fixed << std::setprecision(2) << num << "\n";`

Please note that each ring is sold independently, so the program needs to add the rings' prices to the sum (profit) one by one. In other words, you **cannot use**


sum 10*n+0.25, n=0 to 2999999

$$\sum_{n=0}^{2999999} (10\,n + 0.25) \approx$$
4.4999985750000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 $\times 10^{13}$

Figure 1: Evaluating sums using Wolfram Alpha

a closed-form formula such as the one shown in the Wolfram Alpha picture. You need to write a loop to compute the total profit.

Example input

```
5
1
100
4000
200000
30000000
```

Example output

```
Case 0:
1 rings were sold
Bill's program outputs 0.25
The exact profit is    0.25
Case 1:
100 rings were sold
Bill's program outputs 49525.00
```

```
The exact profit is    49525.00
Case 2:
4000 rings were sold
Bill's program outputs 79981000.00
The exact profit is    79981000.00
Case 3:
200000 rings were sold
Bill's program outputs 199999050000.00
The exact profit is    199999050000.00
Case 4:
30000000 rings were sold
Bill's program outputs 4499999855305422.00
The exact profit is    4499999857500000.00
```

Hint: you will need to use a type with enough precision to compute the exact profit. Below is a list of numeric types and their most common sizes in practice that we have discussed so far (`unsigned` types have been omitted to keep the table compact).

types	size	`#include`
char	8 bits	none
int8_t	8 bits	`<cstdint>`
short	16 bits	none
int16_t	16 bits	`<cstdint>`
int	32 bits	none
int32_t	32 bits	`<cstdint>`
long	32 bits	none
float	32 bits	none
long long	64 bits	none
int64_t	64 bits	`<cstdint>`
double	64 bits	none

---

3. Estimating π using Leibniz's formula: 20 points

File: `leibniz.cpp`

The German mathematican Leibniz (1646 - 1716) discovered the following formula to approximate π:

$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + ...$

Write a program to compute an approximation of π using the first n terms in Leibniz's series, where $1 \leq n \leq 10000000$ is input. Print your output in fixed-point notation, to up to 8 digits of accuracy. You can use `std::fixed` and `std::setprecision` from the `<iomanip>` header file. For example, to output the variable `num` in fixed point notation, to 4 digits of accuracy after the decimal point, do as follows: `std::cout << std::fixed << std::setprecision(4) << num << "\n";`.

Example input

```
6
1
100
1000
10000
100000
1000000
```

Example output

```
Case 0:
Pi estimated as: 4.00000000
Case 1:
Pi estimated as: 3.13159290
Case 2:
Pi estimated as: 3.14059265
Case 3:
Pi estimated as: 3.14149265
Case 4:
Pi estimated as: 3.14158265
Case 5:
Pi estimated as: 3.14159165
```

---

4. Root Finding: 40 points

File: `root_finding.cpp`

This assignment is to use Newton's method to find the roots of an equation, starting the search at some point and finding the root to some desired accuracy or number of iterations. The equation you will be computing roots of is:

$$x^5 + 6x^4 + 3x^3 - x - 50 \tag{1}$$

Newton's method works by starting from some (hopefully close) guess of the root and then iteratively updating it until we get close enough to the actual root of the equation. In other words, you iteratively compute the next best guess for the root using the following formula until convergence:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2}$$

Your program doesn't need to compute $f'(x)$, instead compute the derivative by hand (or using WolframAlpha or however you prefer) and write the expression in your program. You can test for convergence by checking if $x_{n+1} \approx x_n$, specifically you should stop the root finding when the relative error has reached some threshold:

$$\left| \frac{x_{n+1} - x_n}{x_{n+1}} \right| \leq \varepsilon \tag{3}$$

or you've executed n iterations of the root finding method. You can find more information about Newton's method on Wikipedia.

For computing absolute values, sine, cosine and powers you'll want to use some of the functions in the `cmath` header (`#include <cmath>`), documented here.

The input to your program for each case will be the x value to start the search at followed by the number of iterations n and the tolerance ε. Your program should then find the root of equation (1) starting the search at x until the root is found to the desired tolerance or you've run the maximum number of iterations. When your program has found the root to within the desired precision or run out of iterations you should print:

```
root at x = A with error E after C iterations
```

Where `A` is the root value you found, `E` is the remaining relative error and C is the number of iterations you took to find the root. Your program will need to run for at least one iteration to find the error between the guess and where we think the root is, even if the guess is exactly on.

Example Input

```
6
-5.2 100 1e-10
-1.0 100 1e-6
-5 1000 1e-10
1 10 1e-4
-7 100 1e-7
1 5 1e-10
```

Expected Output

```
Case 0:
root at x = -5.39065 with error 7.24956e-14 after 5 iterations
Case 1:
root at x = -5.39065 with error 2.08036e-09 after 5 iterations
Case 2:
root at x = -5.39065 with error 5.60193e-15 after 6 iterations
Case 3:
root at x = 1.52641 with error 9.20928e-07 after 6 iterations
Case 4:
root at x = -5.39065 with error 5.6942e-13 after 7 iterations
Case 5:
root at x = 1.52641 with error 0.000774228 after 5 iterations
```