Assignment 2 Function & Libraries

Part 1 - Debugging

In this task, you can get familiar with the debugger and use it to do what it's designed for - to debug a program.

For `m` tennis balls, put them in `n` buckets. All the tennis balls and buckets are identical, which means division 1, 1, 3 and 1, 3, 1 are the same. Plus, some of the buckets could be empty. Notice that there is at least one bucket for you. How many dividing plans can we have for specific `m` and `n`?

In `1_division/main.cpp`, we have provided you an **incorrect** implementation of a function `int countPlans(int m, int n)` to solve this problem. It's not that far from working correctly – in fact, there is a one-character mistake in that function – but a single character can make a big difference. Your task is to use the debugger to figure out what the error is and correct it.

Input Format

Two integers `m n` in one line. `m` is the number of tennis balls and `n` is th number of buckets.

Output Format

One integer in a line, which is the number of possible dividing plans.

Examples

```
Input: 7 3
Output: 8
```

```
Input: 5 2
Output: 3
```

To summarize, here's what you need to do:

> ***Milestone Requirements***
>
> *Use the debugger to find the error in our provided code. Revise it and submit the correct program named `division/main.cpp`.*
>
> *Hint: It is fine if you don't figure out how this recursive function works at the moment. You can use the debugger to try simple examples and see if the result of a recursive sub-problem is correct. There is only a one-character mistake in the code.*

Part 2 - GCD, and Loop Invariant

You have already seen Ecuclid's algorithm for finding the greatest common divisor in Chapter 2.1 of the textbook (Page 59), but it is not easy to see exactly why the algorithm gives the correct result. Now let us verify this code with the idea of loop invariant.

To avoid ambiguity about variable namings, we restructure the code from the textbook as follows:

```c
/*
 * Function: gcd(int x, int y)
 * Usage: int result = gcd(x,y);
 * -----------------------------------------------------
 * return the greatest common divisor of x and y
 * Pre-condition: x > 0, y > 0
 * Post-condition: result = GCD(x,y)
 */
int gcd(int x, int y) {
   int a = x;
   int b = y;
   int r = a % b;
   // Fill in a loop invariant INV here,
   // INV should hold before entering the loop
   // initially, x > 0 /\ y > 0 =>
   // INV[a|->x, b|->y, r|->x%y] should hold
   while (r != 0) {
       // If the loop goes on, we know that INV holds
       // and that the loop condition holds, which gives
       // us the premise (INV && r != 0)
       a = b;
       b = r;
       r = a % b;
       // After reassigning values to variables a, b and r
       // The loop invariant should still hold, i.e.
       // (INV && r != 0) => INV[a|->b, b|->r, r|->b%r]
   }
   // When the loop exits, we know that INV holds
   // and that the loop condition is not satisfied, so
   // INV && r == 0 should derive the post condition
   return b;
}
```

The restructured code is basically just copying the input into two variables `a` and `b`, so that the input `x` and `y` keeps unchanged during the whole process. It is not hard to come up with a specification for this `gcd(x,y)` function

```
Pre condition: x > 0, y > 0
Post condition: b = GCD(x,y)
```

Note that `GCD(x,y)` is a mathematical function that represents the greatest common divisor of `x` and `y`. We write it in capital letters to distinguish it from the function `gcd(x,y)` in the code.

To verify this specification, we should provide a loop invariant `INV` to ensure the functional correctness of this function. We have already provided you with several proposals. Your task is to choose one among them so that the loop invariant is **satisfied** and **strong enough to ensure the correctness of the function**.

To check your invariant is correct, you should ensure that:

1. Invariant holds before executing the loop:
   `x > 0 && y > 0 => INV[a|->x, b|->y, r|->x%y]`
2. Invariant holds for every loop:
   `(INV && r != 0) => INV[a|->b, b|->r, r|->b%r]`
3. After exiting the loop, the invariant can derive the post condition we want:
   `INV && r = 0 => b = GCD(x,y)`

`INV[a|->x]` means replacing the variable `a` with `x` in the proposition `INV`. We need this substitution in the verification condition because the values of certain variables are modified during the program execution.

During the verification, you may find the following properties about the mathematical function `GCD(m,n)` useful:

1. GCD(x,y) = GCD(y,x)
2. GCD(x,x) = x
3. GCD(0,x) = x
4. **Euclid's theorem**: If m, n are positive integers, then GCD(m,n) = GCD(m%n, n)

To summarize, here's what you need to do:

> ***Milestone Requirements***
>
> *Uncomment one of the answers in the function `gcdAnswer` in `2_gcd_answer/main.cpp` such that the program prints the correct loop invariant to our iterative `gcd` function. Submit the program that prints out your choice.*
>
> *Hint: You can use the debugger to test the `gcd` function with various inputs to help you identify the correct choice.*

Part 3 - GCD, Factorial and Tail Recursion

Recall that iteration and recursion are related. In this part, let us try to use recursion to find the greatest common divisor. Now, please figure out what should be filled into `3_tail_rec/main.cpp` in the starter files.

If you have already filled in the blank, let us take a closer look at the pattern of such recursive function. We note that the last execution of this function is either to return the result or to call itself again. We call this function a **tail recursive function**.

What is the use of tail recursive (or tail call)? Tail calls can be implemented without adding a new stack frame to the call stack. Most of the frame of the current procedure is no longer needed, and can be replaced by the frame of the tail call, modified as appropriate. The program can then jump to the called subroutine. Producing such code instead of a standard call sequence is called tail-call elimination or tail-call optimization, which is done by some optimizing compilers. With tail call elimination, the call stack is no longer needed, and tail recursion can thus compare to the efficiency of iteration.

Now, recall the factorial function we have seen in class. Is it a tail recursive function?

```
/*
 * Function: factorial(int n)
 * Usage: int result = factorial(n);
 * ------------------------------------------------------
 * return the factorial of n
 */
int factorial(int n) {
   if (n == 0) {
       return 1;
   } else {
       return n * factorial(n-1);
   }
}
```

This is **NOT** a tail recursive function. We can see that the last execution of this function is not to call itself, but to do a multiplication on `n` and the result of a recursive call. So with this implementation, tail call optimization cannot be performed, since we need to save `n` in the current frame, and wait until the call to `factorial(n-1)` is finished, so that the program can continue to compute the result of `n * factorial(n-1)`.

The good news is that, indeed, we can use tail recursion to implement `factorial`. This is left for you as an exercise for this part. We have provided you with a template of the tail recursive version in `3_tail_rec/main.cpp`. Try to fill in the blanks in the function.

As an extension, you can also think of whether there is a general way to convert a program implemented in tail recursion into that implemented in iteration. Furthermore, how about recursion into iteration?

To summarize, here's what you need to do:

> ***Milestone Requirements***
>
> *Implement a recursive function `int gcd(int x, int y)` that returns the greatest common divisor of `x` and `y`. You should keep the implementation in a tail recursive way. Your implementation should be based on the provided template in `starter_files`.*
>
> *Implement the function `int factorial(int n)` that returns the factorial of `n` in a **tail recursive way**. Your implementation should be based on the provided template in `starter_files`.*
>
> *Write a main function that demonstrates your two functions. The main function takes three inputs `x`, `y` and `n`, and prints out the result of `gcd(x,y)` and `factorial(n)` in two lines sequentially. The testcases are guaranteed to be valid and not produce results that overflow the `int` range. Submit the program in `3_tail_rec/main.cpp`*

Part 4 - Permutations and Combinations, more efficiently

You have already seen how to write a function that computes the number of combinations of `n` items taken `k` at a time in the textbook (Chapter 2.4, Figure 2-2).

The combinations function $C(n, k)$ we have seen determines the number of ways you can choose k values from a set of n elements, ignoring the order of the elements. If the order of the value matters - so that, in the case of the coin example, choosing a quarter first and then a dime is seen as distinct from choosing a dime and then a quarter-you need to use a different function, which computes the number of permutations, which are all the ways of ordering k elements taken from a collection of size n. This function is denoted as $P(n, k)$, and has the following mathematical formulation:

$$P(n,k) = \frac{n!}{(n-k)!}$$

Although this definition is mathematically correct, it is not well suited to implementation in practice because the factorials involved can get much too large to store in an integer variable, even when the answer is small. For example, if you tried to use this formula to calculate the number of ways to select two cards from a standard 52-card deck, you would end up trying to evaluate the following fraction:

$$\frac{80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,883,277,824,000,00}{30,414,093,201,713,378,043,612,608,166,064,768,844,377,641,568,960,512,000,000,}$$

even though the answer is the much more manageable $2652 (52 \times 51)$.

Your task is to write a function `permutations(n,k)` that computes the $P(n, k)$ function *without calling the fact function*. Part of your job in this problem is to figure out how to compute this value efficiently. To do so, you will probably find it useful to play around with some relatively small values to get a sense of how the factorials in the numerator and denominator of the formula behave.

In this task, you are also required to rewrite the code for the `combinations` function in the textbook so that it uses the efficiency enhancements suggested for permutations in the above text.

To summarize, here's what you need to do:

> ***Milestone Requirements***
>
> *Implement a function `int permutations(int m, int k)` that returns the number of permutations of `m` items taken `k` at a time. Your implementation should not use the factorial function. The function will be tested against various inputs that do not produce results larger than `INT_MAX`*

> *Implement a function `int combinations(int m, int k)` that returns the number of combinations of `m` items taken `k` at a time. Your implementation should not use the factorial function. The function will be tested against various inputs that do not produce results larger than `INT_MAX`*
>
> *Write a main function that demonstrates your two functions. The main function takes two integer inputs `m`, `k`, and prints out the result of `permutations(m,k)` and `combinations(m,k)` in two lines sequentially. The testcases are guaranteed to be valid and not produce results that overflow the `int` range. Submit the program in `4_perm_comb/main.cpp`*

Part 5 - GCD, LCM, Factorial, and together as a library

Now that you have C++ implementations for many handy mathematical functions, it might be worth putting them in a library so that you can use them in many different applications.

To summarize, here's what you need to do:

> ***Milestone Requirements***
>
> *Implement a function `int lcm(int m, int n)` that returns the least common multiple of `m` and `n`.*
>
> *Write a library that exports the **tail recursive** function `gcd`, `factorial` you write in part 3 and the `lcm` function you write in this part. Name the library as `myMath.h` and `myMath.cpp`.*
>
> *In `main.cpp`, implement a function named `gcd_lcm` that computes the greatest common divisor and the least common multiple of `m` and `n` in a single function call. Use the knowledge we have learnt in recent lectures to figure out how to pass two result numbers from this function call to the main function.*
>
> *The main program should demonstrate the function of `gcd_lcm`, and the tail recursive `factorial`, Write a main function that calls `gcd_lcm` and `factorial` and then prints the result. The testcases are guaranteed not to produce results larger than `INT_MAX`.*

The input/output format of `main.cpp` is as follows:

Input: Three positive integers `m`, `n` and `k`, separated by space
Output: The greatest common divisor of `m` and `n`, the least common multiple of `m` and `n`, the factorial of `k`, sequentially in three lines
Remark: When you write a library, note that implementation details **should not be exposed** to the user in the header file.

Part 6 - Balls and Buckets, revisited

Recall the balls and buckets problem we have encountered in part 1, you have already seen a recursive solution for that problem, which divides the problem into smaller subproblems. For that setting, where the balls and buckets are both indistinguishable, you need certain iterative/recursive strategies to solve the problem.

Now, with the `permutations` and `combinations` ready, some other situations may have a one-line straightforward solution. Consider the following three settings:

1. Scenario A: Distribute `n` distinguishable balls into `k` buckets of different colors, and each bucket should at most contain one ball
2. Scenario B: Divide `n` indistinguishable balls into `k` buckets of different colors, and each bucket should contain at least one ball.
3. Scenario C: Divide `n` indistinguishable balls into `k` buckets of different colors, and buckets can be empty.

The number of plans for each of these three settings has straightforward solution, i.e., you can write a simple function that calls `permutations(n,k)` or `combinations(n,k)` you have implemented before (without calling `factorial` version) to get the result. Your task is to first organize these two utility functions into a library, and write a program that imports this library to solve the "balls and buckets" problem.

If you have difficulty in thinking of a solution for the latter two problems, you may want to refer to this material: Stars and Bars - Wikipedia

To summarize, here's what you need to do:

> ***Milestone Requirements***
>
> *Write the files `combinatorics.h` and `combinatorics.cpp` for a library that exports functions `permutations` and `combinations` that **do not use factorial**.*
>
> *In `main.cpp`, implement three functions `int countPlanA(int n, int k)` , `int countPlanB(int n, int k)`, `int countPlanC(int n, int k)` that compute the number of ways to divide `n` balls into `k` buckets, for the three scenarios above, respectively.*
>
> *The main function should call `countPlanA`, `countPlanB` and `countPlanC` and print the result accordingly*
>
> ***Input Format**. One character (`A`, `B`, or `C`) indicating the scenario, two integers `n` and `k`, The testcases are guaranteed to have at least one valid plan and not to produce results greater than `INT_MAX`.*

> ***Output Format.*** *One integer, the number of plans for* `n` *balls and* `k` *buckets in the designated scenario.*

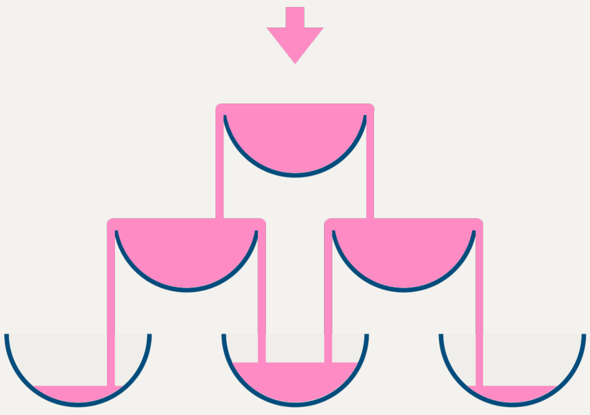
Part 7 - Champagne Problem: recursive problem solving

As a final part in this assignment, let us see how to use recursion to solve pratical problems.

We stack glasses in a pyramid, where the first row has 1 glass, the second row has 2 glasses, and so on until the 100th row. Each glass holds one cup of champagne.

Then, some champagne is poured into the first glass at the top. When the topmost glass is full, any excess liquid poured will fall equally to the glass immediately to the left and right of it. When those glasses become full, any excess champagne will fall equally to the left and right of those glasses, and so on. (A glass at the bottom row has its excess champagne fall on the floor.)

For example, after one cup of champagne is poured, the top most glass is full. After two cups of champagne are poured, the two glasses on the second row are half full. After three cups of champagne are poured, those two cups become full - there are 3 full glasses total now. After four cups of champagne are poured, the third row has the middle glass half full, and the two outside glasses are a quarter full, as pictured below.



Now after pouring some non-negative integer cups of champagne, return how full the `j`th glass in the `i`th row is (both `i` and `j` are 0-indexed.)

Examples

```
Input: poured = 1, query_row = 1, query_glass = 1
Output: 0.00000
Explanation: We poured 1 cup of champange to the top glass of the
tower (which is indexed as (0, 0)). There will be no excess liquid so
all the glasses under the top glass will remain empty.
```

```
Input: poured = 2, query_row = 1, query_glass = 1
Output: 0.50000
Explanation: We poured 2 cups of champange to the top glass of the
tower (which is indexed as (0, 0)). There is one cup of excess
liquid. The glass indexed as (1, 0) and the glass indexed as (1, 1)
will share the excess liquid equally, and each will get half cup of
champange.
```

```
Input: poured = 100000009, query_row = 17, query_glass = 13
Output: 1.00000
```

Constraints

- `0 <= poured <= 10^9`
- `0 <= query_glass < 20`, `0 <= query_row < 20`

Hint

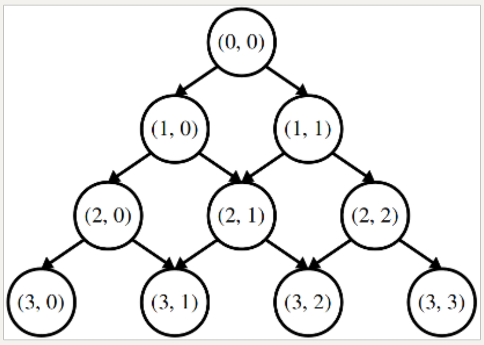
Observe the pattern of champgane flows (that is, the amount of champagne that will flow into a glass during the whole process of pouring), as your first milestone for this problem, and write a recursive function

```
double champagneFlow(double to_pour, int row, int col)
```

that takes as input the row and column number of a glass in the champagne tower, along with the amount of champagne that is poured into the top glass, and returns the amount of flow.

You might need to write a wrapper function to transform the result of `champagneFlow` into the result we want: the amount of champagne in the glass.

The coordinate system we're using for the champagne tower works as follows. Rows increase from top to bottom. The topmost row is row 0, the row below that is row 1, then row 2, etc. Columns increase from left to right, with the leftmost bottle in each row being in column 0, the bottle to its right being in column 1, etc.



If the provided `(row, col)` position passed into `champagneTower` is out of bounds you should return `-1` as an indication of error. For example, if you're asked to look up position (-1, 0), or position (2, 3), you'd always return `-1`.

Your implementation of `champagneFlow` must be implemented recursively. Test your solution thoroughly! You should write at least one test before moving on.

To summarize, here's what you need to do:

> ***Milestone Requirements***
>
> *Implement the `champagneFlow` function from `champagne.cpp` to report the flow into the indicated glass in a champagne tower. Test your solution thoroughly before proceeding, noting that your implementation will be slow if you look deep in the pyramid. This is fine for this milestone. Use the `champagneFlow` function to implement `champagneTower`.*
>
> *Don't forget to check the validity of input coordinates and report error using `-1` as return value in `chanmpagneTower`.*

Some notes on this milestone:

- Remember that the `int` and `double` types are distinct in C++, and be careful not to mix them up. Your flow calculations should be done with `double`s.
- Due to the way that the `double` type works in C++, the values that you get back from your program in large pyramids might contain small rounding errors that make the value close to, but not exactly equal to, the true value (we're talking about very small errors – less than a part in a thousand). The output format in your console may also vary. Don't worry if this happens, as long as you `cout` the correct result.

Submission Guideline

We have provided the codes that have been mentioned in this handout in `starter_files.zip`. Feel free to copy-and-paste them into your own files. Your submission should be organized in the same structure as `submission_example` and follow the specifications in the comment.

We have summarized the task you need to do in each part. Make sure to follow those milestones in corresponding files. There will be **a reduction of scores** if you do not implement your functions in specified ways. Make sure that your interface matches the specification in the handout (if any). You will have **zero points** if your function interface in the header file mismatches the specification in the handout (if any).

Your submission should be a zip file with the following strucutre:

```
Your_student_number.zip
|- 1_division      // Part 1
|  |- main.cpp
|
|- 2_gcd_answer    // Part 2
|  |- main.cpp
|
|- 3_tail_rec      // Part 3
|  |- main.cpp
|
|- 4_perm_comb     // Part 4
|  |- main.cpp
|
|- 5_myMath        // Part 3, 5
|  |- myMath.cpp
|  |- myMath.h
|  |- main.cpp
|
|- 6_combinatorics // Part 3, 4, 6
|  |- combinatorics.cpp
|  |- combinatorics.h
|  |- main.cpp
|
|- 7_champagne     // Part 7
   |- main.cpp
```

We have also prepared a `judger` program that can be used to test the functional correctness your submission (i.e. the corretness of input/output behavior). You still need to check by yourself whether your submission meets the requirements (e.g. interfaces, tail recursive styles, etc...) of this document, which will also be marked by TAs after your submission.

To make `judger` easier to run on various machines, this time the program is shipped as a python script. We presume that after CS1602, Python is already installed on your machine and added to the PATH environment variable. If you have problem using the judger, please reach out to TA.

The basic usage of `judger` is similar to Assignment 1. Just change `judger.exe` into `python judger.py` in the command line and run it.

```
usage:
python judger.py -h   // See help message

python judger.py [-T
{1_division,3_tail_rec,4_perm_comb,5_myMath,6_combinatorics,7_champag
ne}] -I INPUT_FILE -O STANDARD_FILE -S SOURCE_DIR [--timeout TIMEOUT]
```

```
example:

python judger.py -T 1_division -I testcases/1_division/example1.txt -
O testcases/1_division/example1_out.txt -S
submission_sample/1_division

python judger.py -T 6_combinatorics -I
testcases/6_combinatorics/a_example.txt -O
testcases/6_combinatorics/a_example_out.txt -S
submission_sample/6_combinatorics
```