Exercise 0.1. Representing numbers in memory.

Note: You can assume that int and float both occupy 32 bits in memory.

(a) How are integers represented in memory?

(b) Implement a function void int_to_bits(int x) which prints the bit representation of x.
   
   Hints: Use the bitwise shift operator << and the bitwise AND operator &.\(^1\)

(c) How are floating point numbers represented in memory?

(d) Implement a function void float_to_bits(float x) which prints the bit representation of x.
   
   Hint: Casting a float to an int truncates the fractional part, but no information is lost casting a float pointer to an int pointer.

---

\(^1\)Given char x and char n, x \(\ll n\) shifts by n positions to the left the bits which represent x, while x & n applies bit per bit the AND operator. If for example the bits of x are 10100101, then x \(\ll 1\) is 01001010. If y is 00100000 then x \& y is 00100000. Also, recalling that any non-zero value cast to bool is converted to 1, (bool) (x & y) is 1. The same considerations hold for int, only in this case the bit representations have length 32 instead of 8.
Exercise 0.2. Fast and accurate powers.

In this exercise use only elementary functions (+, -, *, /, %, &, ...) or std::exp, std::log from cmath.

(a) Implement a function double power(double a, int b) which returns \(a^b\).

(b) Implement a function double fast_power(double a, int b) which returns \(a^b\) in \(O(\log b)\).
   
   *Hint*: It might be useful to rewrite \(a^b\) as \((a^2)^{b/2}\) when \(b\) is even and as \(a \cdot a^{b-1}\) when \(b\) is odd.

(c) Implement a function double fast_power(double a, double b) which returns \(a^b\) in \(O(1)\).
   
   *Hint*: Use std::exp and std::log. You can assume that both run in \(O(1)\).

(d) Does any of your functions give exactly the same result of std::pow(a,b) for all inputs \(a\) and \(b\)? Why?

(e) How big is the relative error between your implementations and std::pow(a,b) on average if \(a\) is an integer in \([1,100)\) and \(b\) is an integer in \([0,300)\)? (discard the cases when either of your functions or std::pow overflow; you can use std::isinf to check if a double has overflowed). Which one of your implementations minimizes the error? Why?
Exercise 0.3.  
Templates and factorials.

(a) Implement a template function `factorial` which returns the factorial of an integer (of type `char`, or `int`, or `long`, ...) with the same return type as the input. Implement an iterative solution (no recursive calls).

(b) Implement a template function `dbl_factorial` which returns the factorial of any number as a `double` (with the convention that the factorial of a non-integer number is the factorial of its closest integer).

*Hint*: Use `std::round`.

(c) For which input values will your implementation of `factorial` return an accurate value if we pass an `int`? What if we pass a `long`? What if we pass a `double`?

*Hint*: If you include `climits` and `float.h` you can access the maximum `int` / `long` / `double` with the macro `INT_MAX` / `LONG_MAX` / `DBL_MAX`. 

Exercise 0.4. (Optional) Dynamic array implementation.

Implement a barebone struct vec to represent a dynamic array. The struct should have members:

- capacity, the maximum number of elements which can fit in the current instance of the array;
- size, the number of filled slots in the array;
- double* data, a pointer to an array of length capacity.

(a) Implement a default constructor which sets capacity to 10, size to 0, and allocates in data a new array of length 10.

(b) Implement a method void push_back(double x) which appends the element x to the dynamic array in amortized $O(1)$ time.

Hint: If size < capacity simply insert x at position size in data, and increase size by 1. However if size $\geq$ capacity, first allocate a new array of doubled capacity, copy in it all the old values and reassign it to data (remember to delete[] the old array).