Exercise 1 ....

Empirical analysis of 3-SUM. Repeat the experiments from the slides to find the running time of brute-force 3Sum algorithm in the form an^b where a and b are constants. Run the experiments for $n = 500, 1,000, 2,000, 4,000, 8,000, 16,000$. Estimate the value of a and b.

Note: The code from the slides and `stopwatch.hpp` is attached with this lab. Use optimization flag `-O2` while compiling the code, e.g., `g++ -O2 3sum.cpp -o 3sum`.

Exercise 2 ....

Throwing eggs from a building. Suppose that you have an n-story building and plenty of eggs. Suppose also that an egg is broken if it is thrown off floor m or higher, and unbroken otherwise.

Here we use a function object to simulate the process of throwing eggs from a building. The value of m is randomly chosen between 1 and n.

```
class EggDrop {
public:
    EggDrop(int n) : m(std::random_device()() % n + 1) {}
    bool operator()(int x) {
        if (x >= m) return true;
        return false;
    }
private:
    int m;
};
```

We can use binary search to determine the value of m using $\sim \log_2 n$ throws (the number of broken eggs is also $\sim \log_2 n$)

```
int main() {
    int n = 10000;
    EggDrop eggdrop(n);
    int lo = 1, hi = n;
    while (lo < hi) {
        int mid = lo + (hi-lo)/2;
        if (eggdrop(mid)) hi = mid;
        else lo = mid+1;
    }
    std::cout << "The value of m is " << lo << std::endl;
    return 0;
}
```

Devise and implement a strategy to reduce the cost to $\sim 2\log_2 m$ when n is much larger than m.

Hint: Probe at height $2^0, 2^1, 2^2, \ldots, 2^k, \ldots$ and find the value of k such that $2^k \leq m < 2^{k+1}$. Then do a binary search between $lo = 2^k$ and $hi = 2^{k+1}$.

Exercise 3 . . . .

Anagrams. In this exercise, we design a $O(n \log n)$ algorithm to read in a list of words and print out all anagrams. For example, the strings `"comedian"` and `"demoniac"` are anagrams of each other. Assume there are n words and each word contains at most 20 letters.

(a) In this part you generate a list containing artificial data. (A list of $224,714$ words is also provided in the file `wordlist.txt`.)

The following code generates a random word of length m:

```
std::string random_word(int m) {
    std::string s;
    for (int i = 0; i < m; ++i) {
        s.push_back('a' + std::random_device()() % 26);
    }
    return s;
}
```

For each word generated, we can shuffle the characters to generate an anagram. The following code shuffles the characters of a string `s`:

```
shuffle(s.begin(), s.end(),
   std::default_random_engine(std::random_device()()));
```

Generate a list of $n = 1000$ words, where each word has length at most 10. Ensure that the list contains anagrams.

(b) Design a $O(n^2)$ algorithms to find all anagrams in a given list of words.

(c) Improve the above algorithm to $O(n \log n)$.

We will make use of `std::sort` function from the C++ Standard Library. To sort a container, we can use the following syntax:

```
std::sort(container.begin(), container.end());
```

Now, do the following steps to solve the problem:

1. sort each word in the list of words.
2. sort the list of words.
3. After the above steps, all anagrams will be next to each other and can be counted in linear time.