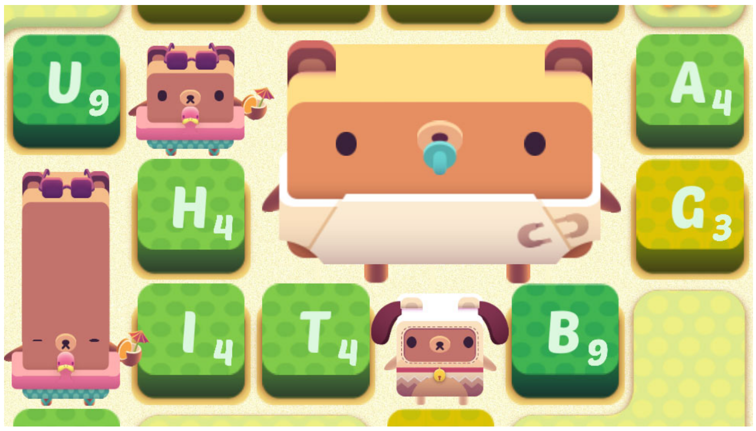

Figure 1: Alphabear gameplay, `https://patricktay.files.wordpress.com/2015/09/alphabear1.jpg`.

Problem 1 (25 points)

Alphabear[1] is a mobile game in which the player is given a series of letters, each of which has a "timer" (see Figure 1[2]). The player then forms a word based on the available letters, and for the letters not chosen, the timer for those letters decreases by 1. If any letter's timer decreases to 0, then it will be unavailable from that point on. At the end of the game, either there are no possible words that can be formed, or all available letters have been used. Bonuses are awarded based on what bears the player has chosen (each has specific bonuses, such as one uses certain letters, or a percent score increase at the end of the game). The game awards a very large bonus for being able to clear the entire board without having any letters unavailable.

We will be writing a "partial solver" for Alphabear. We cannot create a full solver because when letters are chosen, others appear in a random way (i.e., the characters themselves are random), so we cannot necessarily predict what the best strategy is overall. Nevertheless, we will write a solver for *one turn* in the game; as you will see, it can be extended to an arbitrary number of turns using a loop. For this reason, we won't be doing calculations for bonuses at the end of the game. What we want to accomplish here is to find the "best" word at the current turn based on what is visible.

Suppose a visible letter on the board c, has a current timer value of t (we assume that the timer is always a positive integer). Then the *urgency* of c is $1/t$. In other words, c has higher urgency if the timer is lower. The Alphabear game loves words that are longer, so we want our model of what the best word is to factor this into consideration. Let $x_1 x_2 \cdots x_n$ be an English word, and has corresponding timer values $t_1, t_2, \cdots, t_n$ on the board. Then the urgency of this word is:

$$(1/t_1 + 1/t_2 + \cdots + 1/t_n) \times 1.1^n.$$

The urgency of an English word, with letters $x_1 \cdots x_n$, is the sum of the urgencies of $x_1, x_2, \cdots, x_n$. For example, if *ahh* is the considered word, and a has an timer of 1, one occurrence of h has a timer of 2, and another occurrence of h has a timer of 7, then the urgency of *ahh* is $(1+1/2+1/7) \times 1.1^3 \approx 2.186$.

Therefore, we want to find any English word of highest urgency. **We will assume that if there are more occurrences of a given character visible than are in the considered word, the**

[1] `http://spryfox.com/our-games/alphabear/`
[2] Yes, it is extremely adorable.

urgency of the word will be calculated based on the timers *in sorted, increasing order.* This assumes that the player will make the "obvious" choice about which letters to pick.

Note that if we want to consider an English word, all of its letters must be usable by the board. For instance, in the figure above, *hit* can be chosen (since all three letters are visible), whereas *hunt* is not (assuming n is not visible anywhere else).

All English words will be provided next to this PDF on the course webpage. The file has one English word per line (i.e., separated by a newline character). You will have to research how to read a file line-by-line (or all at once) in `C++`, but a good resource is here:

`https://stackoverflow.com/a/7868998/3232207`.

Of course, since there are so many English words, it would be impractical to print *all* valid words. So, you will proceed as follows:

- Add the file of English words to your CLion project, or to your workspace (depending on what you use).
- `cin` a positive integer n from the user (this is the number of words you will print eventually).
- Read from the user (until the user enters -1) a character, along with its corresponding timer. **Note: a letter can appear any number of times here, and there is no upper limit on how many characters will be entered.** (Hint: what structure helps with doing this?). You can assume that the user will enter everything correctly here.
- Read through the file word-by-word. (Hint: check that your program can do this first before tackling all the other parts). **You must treat the file as if it is in the same directory as the `main` function** (as this is how we will grade it).
- For each word, if it is compatible with the letters the user entered, then calculate its urgency. You can loop over a `std::string` as follows:

```
string s = ...;
for (char ch : s) {
    // process character ch
}
```

  Or if you want to use an index:

```
string s = ...;
for (int idx=0; idx < s.length(); idx++) {
    char ch = s[idx];
}
```

- Maintain a structure of compatible words along with their urgencies (Hint: which one would work?).
- Output to the user a sorted list, by urgency in decreasing order, of the top n words in your list. (If there are fewer words than n in your list, output them all).

You may use parallel arrays, but the issue comes at the last step in trying to sort across two arrays (it is possible with some thought); or create a custom class for tying an individual character with its urgency, and then a custom class corresponding to a word. If you do this, you may have to look up how to define a custom comparison function for a user-defined object. Or, you can write your own custom sorting function that handles the work for you.

As always, sprinkle your program with helpful comments throughout, as well as if you use any functions not discussed during class.

Solution: below is the code. The high-level ideas are:

1. Reading a file,
2. Getting user inputs and storing them in a `vector`,
3. Finding some method to calculate the urgencies. Remember that the user will pick “the best” letter if there are multiple of them. What I did was to store the letters and timers in two `vector`s, and then sort the timers, while also sorting the letters. We don’t need to sort the letters wrt the timers here because if two identical letters have the same timer, it doesn’t matter. Crucially, when we “use” a letter, we set its value *in a copy of the timers* `vector` to -1 (or any invalid value) so that it marks that we cannot use it again. We then just calculate the urgency in the straightforward way.

   Another way to do this is to use *operator overloading* (which is in the next set of slides, so is not required), but has the disadvantage that you have to implement a class as well as the comparison operators yourself. However, it has the advantage of being more maintainable.
4. At this point, we have urgencies along with valid words, so we do the same trick as before where we simultaneously sort the urgencies and valid words. This time I flipped the comparison so as to make the urgencies go in descending order.
5. Then just print the urgencies and valid words.

```cpp
#include <algorithm>
#include <cmath>
#include <fstream>
#include <iostream>
#include <numeric>
#include <vector>

using namespace std;

bool is_compatible(string word_to_check, vector<char> inputted_chars
   ) {
        for (char ch : word_to_check) {
                // get the counts of each character
                auto num_occurrences_in_word = count(word_to_check.
                   begin(), word_to_check.end(), ch);
                auto num_occurrences_in_input_chars = count(
                   inputted_chars.begin(), inputted_chars.end(), ch)
                   ;
                if (num_occurrences_in_word >
                   num_occurrences_in_input_chars) {
                        return false;
                }
        }
        return true;
}

int main() {

        int n;
        cin >> n;

        vector<char> input_chars;
        vector<int> timers;

        while (true) {
                char ch;
                int timer;
                cin >> ch >> timer;
                if (timer == -1) {
                        break;
                }
                input_chars.push_back(ch);
                timers.push_back(timer);
        }

        ifstream input_file("./words.txt");
        vector<string> compatible_words;
        string line;
        cout << "\nReading file\n";
        while (getline(input_file, line)) {
                if (is_compatible(line, input_chars)) {
                        compatible_words.push_back(line);
                }
        }

        // sort timers along with letters using bubble sort
        bool swapped;
        cout << "\nSwapping timers + input chars\n";
        do {
                swapped = false;
                for (int ct = 0; ct < timers.size() - 1; ct++) {
                        if (timers[ct] > timers[ct + 1]) {
                          swap(timers[ct], timers[ct+1]);
                          swap(input_chars[ct], input_chars[ct+1]);
                          swapped = true;
                        }
                }
        } while (swapped);  // Loop again if a swap occurred on this pass.



        vector<double> urgencies;
        cout << "\nCalculating urgencies\n";
        for (auto& word : compatible_words) {
                double urgency = 0.0;
                vector<int> timers_copy(timers); // so we don't modify the original
                for (int char_idx = 0; char_idx < word.size(); char_idx++) {
                        int idx_input = 0;
                        bool keep_going = true;
                        while (keep_going) {
                          if (word[char_idx] == input_chars[idx_input] && timers_copy[idx_input] != -1) {
                            break;
                          }
                          idx_input++;
                        }
                        // guarantee idx_in_inputs is valid because we verified it earlier
                        auto timer_value = timers_copy[idx_input];
                        timers_copy[idx_input] = -1; // so we don't use this instance of the letter again
                        urgency += (1.0/timer_value);
                }
                urgency *= pow(1.1, word.size());
                urgencies.push_back(urgency);
                // the indices line up here, so we are good!
        }

        // sort urgencies along with valid words using bubble sort
        cout << "\nSorting Urgencies + Valid Words\n";
        do {
                swapped = false;
                for (int ct = 0; ct < urgencies.size() - 1; ct++) {
                        if (urgencies[ct] < urgencies[ct + 1]) { // flipped here to be in ascending order
                                swap(urgencies[ct],urgencies[ct+1]);
                                swap(compatible_words[ct], compatible_words[ct+1]);
                                swapped = true;
                        }
                }
        } while (swapped);  // Loop again if a swap occurred.

        // need the minimum of the inputted # of words and the size of valid words
        auto m = (compatible_words.size() > n) ? n : compatible_words.size();

        for (int i=0; i<m; i++) {
                cout << compatible_words[i] << "," << urgencies[i] << "\n";
        }

        return 0;
}
```