Lab_dict: Devious Dictionaries

Welcome to Lab Dictionaries!

Overview

You have already seen how to implement many interesting data structures, in this lab we'll be using dictionary types in the Standard Template Library (STL) to solve some puzzle problems that might come up in your future technical interviews!

Memoization Dynamic Programming:

Memoization is a dynamic programming technique that refers to the process of caching previously calculated results in some sort of data structure, so that when the computation needs to be run again with the same inputs, the value can simply be looked up. This process results in a large speedup for problems that have expensive computation and that can be broken down into subproblems that could be reused to compute the larger problem.

Exercise 1.1: Suppose you are given the set of positive integers: E = {1, 2, 5, 7, 11, 18, 25, 300}
Your client gives you an input positive integer S, and asks you to select one or more numbers from the above set such that their sum = S. ***You may select the same number from your set multiple times***. Notice that since we have a 1 in our set, the client's request can always be satisfied by choosing 1 S times; although this isn't the most efficient/clever way. Write pseudocode for a function expectedSum(S) that returns a list of numbers from E (with repetition) that sum to S. This can be as naive or sophisticated as you like:

```
One possible solution:
Let R=S, and result = []
Start from the largest number in E (300) going down the
list, and repeat until R=0:
Find the largest coefficient c such that: c*300 <= R
Append (c, 300) to result, and update R -= c*300
Move to next largest number: 25
```

Exercise 1.2: Now suppose your client gives you a sequence of 10 million such inputs: $s_1 < s_2 < s_3 < \ldots < s_{10\ mill}$ (for simplicity, assume $s_1 > max(E)$)

How can we speed up our calculations? Since the input sequence is increasing,

```
What we want to do here is slowly build up a virtual "memory" where
we keep track of the sums already calculated and how they were
calculated.

One possible way to do this is: Keep an expanding list of these sums
memory=[] , appending each s_i to it after its been calculated. We
can keep a corresponding list C =[] to save "how" each s_i in memory
was obtained i.e. C=[...,[(2, 300), (1, 25)],...] where [(2, 300),
(1, 25)] is the entry for how to obtain 625. We initialize memory
and C to: memory = [1, 2, 5, 7, 11, 18, 25, 300] C =
[(1,1),(1,2),(1,5),(1,7),(1,11),(1,18),(1,25),(1,300)]

The way that this will speed up our solution for 1.2 is that for any
s_i we will be able to consult memory and use the entries there to
try to build up the sum for it; and not be restricted by just the
numbers in E. For example, since we already know how to make 625:
[(2, 300), (1, 25)] and since we save it in memory, if we have an
input 1250 we can easily see that that is: 2*625 without redoing the
work we did for 625.
```

can we use the solution for s_i to calculate the solution for s_{i+1}?

Dictionaries:

You have already seen how to implement dictionaries using hash tables in **lab_hash,** when we stored different <*Key, Value*> pairs in the hash table. We also used tree-based structures to implement dictionaries. In C++, **`std::map`** is a tree-based implementation of a dictionary, while **`std::unordered_map`** uses a hash table implementation as the underlying structure.

main.cpp

```
1  struct Student {
2        string name;
3        int uin;
4        int year;
5        string major;
6  };
```

Exercise 2: Suppose that there are 40 000 undergraduate students in our UIUC database. Each student object has a name (first last), uin, year (1 through 4), and major ("CS", "ME", "ECE", etc.). Look at the Student struct above for reference.
The student objects are currently stored in the database in no particular order. We would like to be able to answer questions such as: "How many sophomores are in CS?" or "How many ECE students are graduating this year?"
How would you use dictionaries to answer such questions quickly and efficiently without having to comb through all 40 000 records in the database?

One possible solution:

We organize all 40 000 students in a nested dictionary structure, the first key can be the student's major; so students who have the same major are grouped under the same dictionary entry. The value of each entry is another dictionary, this time with the student's year as key; thus students with the same major AND year will be grouped together. The value for this inner dictionary can be a list of the student names. To visualize this:

```
{CS: {1: [Alice, Bob, Charlie, ...];
      2: [Dawn, Ema, ...];
      3: [Genna, ...];
      4: [Hana, ...];}
ECE: {1: [Jordi, ...]; 2: [Mariam, ...]; 3: ...;  4: ...;}
ME: ... ... ...
}
```

In the programming part of this lab, you will complete the following functions/classes:

- In fib.cpp, implement fib() and memoized_fib()
  - Look at fac.cpp for inspiration.
- In the CommonWords class implement the following functions:
  - init_file_word_maps() - **file_word_maps** holds a map for each file. Each map associates a word in that file to the number of times it has been seen in that file.
  - init_common() - **common** maps a word to the number of documents that word appears in
  - get_common_words()
    - There are several ways to solve the get_common_words() problem
    - Try to discuss your approach with your TA/CA!
- For the Pronunciation Puzzler class:
  - The purpose is to find words such that the word itself, the word with its first character, and the word with its second character removed are all homophones.
    - Example: wrack, and rack
  - homophones()
    - Determines whether two words are homophones.
  - cartalk_puzzle()
- For the Anagram class:
  - Implement both constructors.
  - get_anagrams() and get_all_anagrams()
    - Think about how you would set anagrams to map to the same location in a dictionary.
    - What do two words that are anagrams have in common?

As your TA and CAs, we're here to help with your programming during the Virtual Office Hours! ☺