CS 106B Section 4 Handout (Week 5)

1. CHeMoWiZrDy

Some words in the English language can be spelled out using just symbols from the Periodic Table. For example, the word "began" can be spelled out as BeGaN (beryllium, gallium, nitrogen) and the word "feline" can be spelled out as FeLiNe (iron, lithium, neon). Not all words have this property, though; the word "interesting" cannot be made out of the element symbols, nor can the word "chemistry".

Suppose you are given a Lexicon containing all the element symbols in the periodic table. Write a function:

```
bool isElementSpellable(string text, Lexicon& symbols);
```

that accepts as input a string, then returns whether the string can be written using only element symbols. If you'd like, you can use the fact that all element symbols are at most three letters.

2. Shrinkable Words

Having solved the problem of finding ladders between words, you may now become interested in what other silly things you can do with sequences of valid English words. Let's introduce the concept of a "shrinkable" word:

> A word is considered shrinkable if it can be reduced to a single-letter word by removing letters one at a time, leaving a valid English word at each step.

For example, the word "startling" is startlingly shrinkable:

```
startling → starting → starting → string → sting → sing → sin → in → i
```

Can you use your recursive backtracking skills to find other shrinkable words? Try writing a function:

```
bool isShrinkableWord(string word, Lexicon& words, Vector<string>& path);
```

Which returns true if the first parameter is a shrinkable word (using the words given in the lexicon). If the word is shrinkable, return the path taken to shrink that word in the output parameter called path.

3. **printSquares**

Write a recursive function `printSquares` that uses backtracking to find all ways to express an integer as a sum of squares of unique positive integers. For example, the call of `printSquares(200);` should produce the following output:

```
1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 8^2 + 9^2
1^2 + 2^2 + 3^2 + 4^2 + 7^2 + 11^2
1^2 + 2^2 + 5^2 + 7^2 + 11^2
1^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2
1^2 + 3^2 + 4^2 + 5^2 + 7^2 + 10^2
2^2 + 4^2 + 6^2 + 12^2
2^2 + 14^2
3^2 + 5^2 + 6^2 + 7^2 + 9^2
6^2 + 8^2 + 10^2
```

Some numbers (such as 128 or 0) cannot be represented as a sum of squares, in which case your function should produce no output. Keep in mind that the sum has to be formed with unique integers. Otherwise you could always find a solution by adding `1^2` together until you got to whatever `n` you are working with.

To help you solve this problem, assume there already exists a function `printHelper` that accepts any C++ collection of integers (such as a vector, set, stack, queue, etc.) and prints the collection's elements in order.

CS 106B Section 4 Handout (Week 5)

4. makeChange

Write a recursive function `makeChange` that uses backtracking to find all ways to make change for a given amount of money using pennies (1 cent), nickels (5 cents), dimes (10 cents), and quarters (25 cents). For example, when making change for 37 cents, you could use: 1 quarter, 1 dime and 2 pennies; 3 dimes and 7 pennies.

You should accept a single parameter: the amount of cents for which to make change. Your output should be a sequence of all combinations of each type of coin that add up to that amount, one per line. For example, if the client code contained the following call:

```
cout << " P  N  D  Q"  << endl;
cout << "------------" << endl;
makeChange(28);
```

The overall output generated should be the following:

```
 P  N  D  Q
------------
{3, 0, 0, 1}
{3, 1, 2, 0}
{3, 3, 1, 0}
{3, 5, 0, 0}
{8, 0, 2, 0}
{8, 2, 1, 0}
{8, 4, 0, 0}
{13, 1, 1, 0}
{13, 3, 0, 0}
{18, 0, 1, 0}
{18, 2, 0, 0}
{23, 1, 0, 0}
{28, 0, 0, 0}
```

You may assume that the amount of change passed is non-negative, but it could exceed 100.

Hints: A key insight toward solving this problem is the notion of looking at each denomination of coin (penny, nickel, etc.) and trying all possible numbers of that coin (1 penny, 2 pennies, ..., 28 pennies) to see what combinations can be made starting with that choice. For example, in the output above, first all the combinations that begin with 3 pennies are shown, then all combinations with 8 pennies, and so on.

Since backtracking involves exploring a set of choices, you should represent the coin denominations in some way in your code. We suggest keeping a vector of all coin denomination values to process. As you process coin values, you can modify the contents of the vector. The template below is a starting point:

```
void makeChange(int amount) {
    Vector<int> coinValues;
    coinValues += 1, 5, 10, 25;    // penny, nickel, dime, quarter

    // ... your code goes here …
}
```

CS 106B Section 4 Handout (Week 5)

5. TimeSpan

Define a class named `TimeSpan`. A `TimeSpan` object stores a span of time in hours and minutes (for example, the time span between 8:00am and 10:30am is 2 hours, 30 minutes). Each `TimeSpan` object should have the following public methods:

`TimeSpan(int hours, int minutes)`
Constructs a TimeSpan object storing the given time span of minutes and seconds.

`void add(int hours, int minutes)`
Adds the given amount of time to the span. For example, (2 hours, 15 min) + (1 hour, 45 min) = (4 hours). Assume that the parameters are valid: the hours are non-negative, and the minutes are between 0 and 59.

`void add(TimeSpan timespan)`
Adds the given amount of time (stored as a time span) to the current time span.

`double getTotalHours()`
Returns the total time in this time span as the real number of hours, such as `9.75` for (9 hours, 45 min).

`string toString()`
Returns a string representation of the time span of hours and minutes, such as `"28h46m"`.

`bool operator<(TimeSpan other)`
Returns true if this TimeSpan represents a shorter time than another TimeSpan.

The minutes should always be reported as being in the range of 0 to 59. That means that you may have to "carry" 60 minutes into a full hour. For example, consider the following client code:

```
int main() {
    TimeSpan t1(3, 45);
    cout << t1.toString() << " is " <<  t1.getTotalHours() << " hours";

    t1.add(2, 30);
    cout << t1.toString() << " is " <<  t1.getTotalHours() << " hours";

    TimeSpan t2(0, 51);
    t1.add(t2);
    cout << t1.toString() << " is " <<  t1.getTotalHours() << " hours";
}
```

This code creates a `TimeSpan` object and adds three different times to it. The output should be:

```
3h45m is 3.75 hours
6h15m is 6.25 hours
7h6m is 7.1 hours
```

Notice that the second time is *not* 5 hours, 75 min, although that's what you'd get by just adding field values.