CISS362: Introduction to Automata Theory, Languages, and Computation Assignment 1

Q1. [Pancake-flipping for digits]

Write a program that accepts integer `n` and perform (and print) pan-cake-flipping on the digits of `n` until the digits are in ascending order. For instance if `n = 123152`, your program should print

```
123152
123152
513212
212315
321215
121235
211235
112235
```

[The first line is your input. Subsequent lines are output from your program.] Pan-cake-flipping here means you take a contiguous chunk of digits starting with the leftmost and reversing the digits. For instance the following pancake-flip:

`123152 -> 513212`

reverses the leftmost 5 digits. The above example illustrates the (obvious) algorithm. Make sure your pancake-flipping follow that idea.

The following are some test cases.

TEST 1:

```
123152
123152
513212
212315
321215
121235
211235
112235
```

TEST 2:

```
12345678
12345678
```

TEST 3:

```
87654321
87654321
12345678
```

Q2. [Computational Molecular Biology]

If you're given an array of integers in ascending order starting with 0, for instance

```
{0, 4, 5, 8}
```

you can compute the pairwise differences:

$$\begin{aligned}&4-0, 5-0, 8-0\\&5-4, 8-4\\&8-5\end{aligned}$$

i.e.,

```
{4, 5, 8, 1, 4, 3}
```

which when sorted becomes

```
L = {1, 3, 4, 4, 5, 8}
```

We call this function `Delta`:

```
Delta({0, 4, 5, 8}) = {1, 3, 4, 4, 5, 8}
```

In computational molecular biology, one is interested in going the opposite way. In other words if you're given

```
{1, 3, 4, 4, 5, 8}
```

can you find all `X` (containing 0 as first value) such that

```
Delta(X) = {1, 3, 4, 4, 5, 8}
```

1. Implement the `Delta` function with the following prototype:

```
std::vector< int > Delta(std::vector< int >);
```

2. Write a function `SOLVE` that accepts an vector of integers `L` and prints all `X` such that

```
Delta(X) = L
```

3. Finally here's the program: Write a program that allows the user to enter integer values for `L`, terminating the input with `-1`, and then print the solutions `X` such that `Delta(X) = L`. Here's a screenshot just to fix the format:

```
1 3 4 4 5 8 -1
0 4 5 8
```

(If there are more than one solution, then you should print them on separate lines.)

Here's a hint: In the above example, the largest value in `L = {1, 3, 4, 4, 5, 8}` is 8. The array `X` must have values in `Z = {0, 1, 2, 3, ..., 8}`. (In general, if the largest value in `L` is `M`, then `X` must have values in `{0, 1, 2, ..., M}`. Think about it.)

All you need to do is to generate all subsets `X` of `Z` and test if `Delta(X) = L`. In fact, you don't have to test all subsets. If `L` has 6 elements, then `X` must have 4 elements. In other words there's a relationship between `|X|` and `|L|`. (Discrete 1)

[Connection with computational molecular biology: The `X` are points on a DNA where "cuts" occurs due to an enzyme acting on the DNA. The resulting lengths are DNA fragments. The reconstruction of the places where the "cuts" occur is important to molecular biologists. Of course the problem is completely computational and you do not need to understand biology to understand this problem.]

NOTES ON `std::vector`. The class `std:vector` allows you to create dynamic arrays. The following code shows you how to create a vector of integers using the constructor, get the number of values in the vector using the `size()` method, insert a value into the end of the vector using the `push_back()` method, and how to access a value in the vector using `operator[]()`. There are other methods/operators available in the `std::vector` class. Make sure you run the following example:

```
#include <iostream>
#include <vector>

template< class T >
std::ostream & operator<<(std::ostream & cout, const std::vector< T > & v)
{
    for (unsigned int i = 0; i < v.size(); i++)
    {
        cout << v[i] << ' ';
    }
    return cout;
}

int main()
{
    std::vector< int > v;                           // constructor
    std::cout << "size: " << v.size() << std::endl; // number of values in v
    std::cout << "v: " << v << std::endl;
    v.push_back(5);
    std::cout << "v: " << v << std::endl;
    v.push_back(3);
    std::cout << "v: " << v << std::endl;
    v.push_back(5);
    std::cout << "v: " << v << std::endl;
    v.push_back(0);
    std::cout << "v: " << v << std::endl;
    std::cout << "size: " << v.size() << std::endl; // number of values in v
    std::cout << "v[1]: " << v[1] << std::endl;     // random access
    return 0;
}
```

You can of course perform assignment using `operator=()`. You can easily find everything you want to know about `std::vector` if you google.

NOTES ON GENERATING POWERSETS. The set of all subsets of X is called the powerset of X. So this exercise is really about the computation of powersets. (The calculation of `Delta` is easy – that's not the main point of this exercise.)

If you have a set `X` say

```
    X = {a, b, c, d}
```

Here's a subset `Y` of `X`:

```
    X = {a, b, c, d}
    Y = {a, c}
```

Write it this way:

```
    X = {a, b, c, d}
    Y = {a,    c   }
```

and you see that forming subsets is (of course) just a matter selecting elements from X. If you indicate whether a particular value in `X` is selected or not and say I call this selection array `Z`:

```
    X = {a, b, c, d}
    Z = [1, 0, 1, 0]
    Y = {a,    c   }
```

where `1` means selected and `0` means not selected. In fact this *is* the method used to prove that the number of sets in the powerset of `X` is 2^n where n is the size of `X`.

This means that as long as you can compute all the arrays of 0s and 1s of length n where n is the size of `X`, you should be able to form all subsets. For the X above this means that we need to compute this sequence of arrays:

```
[0,0,0,0], [0,0,0,1], [0,0,1,0], [0,0,1,1], [0,1,0,0], ...
```

(not necessarily in this order.)

Finally let me say that the sequence of 0s and 1s look very much like binary numbers. There's a translation between binary numbers and integers:

```
[0,0,0,0] <----> 0
[0,0,0,1] <----> 1
[0,0,1,0] <----> 2
[0,0,1,1] <----> 3
[0,1,0,0] <----> 4
...
[1,1,1,1] <----> 15
```

Obviously we can loop from 0 to 15!!! This means that the strategy is looks like this:

```
for i = 0, 1, 2, ..., 15:
    Z = compute binary sequence from i
    Y = form subset of X using Z and X
    ... do whatever you need to do with Y ...
```

Of course this is only for `X` where `X` has 4 elements. However it's easy to handle the case where `X` has `n` elements.

This is an example where knowing the proof of a theorem, i.e. the number of subsets of a set with n elements is 2^n, turns out useful for generating sets.

Q3. [Adapting `std::set`.]

This exercise is optional but ... highly recommended for CS students.

The C++ STL comes with a template set class, `std::set`. This exercise involves getting to know this class by giving it more "features". Adding more features to a class is done by subclassing or by composition (or adapting). The standard practice of extending the features of STL classes is by adapting. So let's begin ...

Adapting a class say `X` to another class `Y` is pretty simple. Suppose `X` has a method `m()`. Then each `Y` object contains an `X` object. Furthermore, you simple create a method `m()` in `Y` that delegates the task to the `m()` of `X`:

```
class Y
{
public:
    void m()
    {
        x.m();
    }
private:
    X x;
};
```

Now every `Y` object is really pretty much the same as an `X` object. You can then add new methods to class `Y`:

```
class Y
{
public:
    void m()
    {
        x.m();
    }
    void n()
    {
        // new method
    }
private:
    X x;
};
```

Enough said. Let's go to the real problem.

You are given the following the following skeleton code:

```
// File: Set.h
#ifndef MYSET_H
#define MYSET_H

#include <iostream>
#include <set>

template < class T >
class Set
{
public:
    Set() {}

    void insert(const T & x) { set.insert(x); }
    int size() const { return set.size(); }

    const typename std::set< T >::iterator begin() const
    {
        return set.begin();
    }

    const typename std::set< T >::iterator end() const
    {
        return set.end();
    }

private:
    std::set< T > set;
};


template < class T >
std::ostream & operator<<(std::ostream & cout, const Set< T > & X)
{
    cout << '{';
    if (X.size() > 0)
    {
        typename std::set< T >::iterator p = X.begin();
        cout << (*p);
        typename std::set< T >::iterator end = X.end();
        for (p++; p != end; p++)
        {
            cout << ", " << (*p);
        }
    }
    cout << '}';
    return cout;
}

#endif
```

(skeleton = not complete!) with the following test code:

```
#include <iostream>
#include "Set.h"

int main()
{
    Set< int > X;
    X.insert(1);
    std::cout << "|X| = " << X.size() << ", X = " << X << std::endl;

    X.insert(1);
    std::cout << "|X| = " << X.size() << ", X = " << X << std::endl;

    X.insert(5);
    std::cout << "|X| = " << X.size() << ", X = " << X << std::endl;

    X.insert(0);
    std::cout << "|X| = " << X.size() << ", X = " << X << std::endl;

    return 0;
}
```

Run the above.

Add methods so that you can do the following:

`X.clear()`	`Make X the empty set`
`X.empty()`	`True iff X is the empty set`
`X == Y`	`True iff X and Y are the same sets`
`X != Y`	`opposite of ==`
`X = Y`	`The obvious assignment`
`X.has_member(x)`	`True iff x is a member of X`
`X <= Y`	`True iff X is a subset or is equal to Y`
`X < Y`	`True iff X is a subset and is not equal to Y`
`X + Y`	`Union of X and Y`
`X * Y`	`Intersection of X and Y`
`X - Y`	`Set difference of X and Y`
`X.erase(p)`	`Erase element iterator p points to`
`X.swap(Y)`	`Swap the contents of X and Y`
`X.powerset()`	`Powerset of X (optional)`
`prod(X, Y)`	`Product set of X and Y (optional)`

You may add whatever method you need to achieve the above. Make sure you test your code thoroughly.

I will only test the case where `X` and `Y` in the above are sets of `int` values.

[Do *not* use other C/C++ provided functions that does the work for you. For instance C++ does provides `set_union` – do not use it. Likewise for `set_intersection`, etc. You should only use what is provided by `std::set`. If in doubt, just ask.]