4.1 STL Algorithms I

Summary and Goals

The goal of the exercises in this section is to examine the ADTS and some of the algorithms in STL and build new generic abstractions on top of them. In particular, you design and implement reusable classes and functionality in C++11.
In a sense, this is a transition from syntax-based exercises to ones that involve more analysis, thought and design**. This trend will continue in coming sections**.

1. (Adapter Classes for Bitsets)

In this exercise we create a class that models *propositions* in Boolean algebra (for this exercise you may need to review this topic). To this end, we create a class called `Proposition` that models propositions and the associated Boolean operators such as (A, B and C are propositions):

- Equality, inequality (A == B, A != B)
- Conjunction (AND) (A & B)
- Disjunction (OR) (A | B)
- Exclusive OR (XOR) (A ^B)
- Conditional (->) (A % B – operator% is overloaded, as modulo is inappropriate in this context)
- Biconditional (<->) (aka (A->B)&(B->A))
- Negation (!A)
- Assign a proposition to a `bool`

The basic class interface is:

```
class Proposition
{ // A class representing true/false

private:
    std::bitset<1> data;
public:

// TBD

};
```

Answer the following questions:

a) Create the following constructors: default, bool as argument and a `std::bitset<1>` as argument.
b) Implement the above Boolean operators as members with the exception of conditional and biconditional which should be non-member friends. Test each operator separately.
c) Check that your code satisfies *De Morgan's Laws*:
   - NOT (A OR B) == NOT(A) AND NOT(B)
   - NOT (A AND B) == NOT(A) OR NOT(B).
d) Check that your code satisfies the *Distributive Laws*:
   - A AND(B OR C) == (A AND B) OR (A OR C)
   - A OR(B AND C) == (A OR B) AND (A OR C).
e) Let A, B and C be propositions. Check that the statement form:
   - [A -> (B -> C)] <-> [(A & B) -> C]

Is a *tautology* (that is, it always returns true for all assignments of truth values to its statement letters A,B and C). There are eight options to be checked.

In all code delegate to `std::bitset<1>` and use as much of the underlying mathematics as possible.

2. (Compile-Time Vector Class)

In the next two exercises we create fixed-sized vector and matrix classes, that is classes whose instances reside on the stack rather than on the heap. There are many applications of these classes where there is a need for *tiny vectors* and *tiny matrices*. The scope and requirements for both classes are:

- Constructors: default, with value as argument and copy constructor.
- Accessing elements (read/write) using [].
- Adding and subtraction; unary minus.
- Premultiplication by a scalar quantity.

The classes are generic:

```
template<typename T, int N> class Vector;
template <typename T, int NR, int NC> class Matrix;
```

Answer the following questions:

a) Given the above operations to be implemented you need to choose a suitable data structure ADT to hold the data, for example:

```
// Type == V
Type arr[N];
std::array<Type, N> arr;
```

The choice depends on how efficient the ADT is and how much functionality can be delegated to the ADT.

b) Create some instances of `Vector` and print them.

c) Implement the above operation pertaining to the properties of a vector space as explained in the introduction.

d) Implement the *scalar multiplication* operation as a *template member function* having an extra template parameter:

```
template <typename Type, int N, typename F> Vector<Type, N>
friend operator * (const F& a, const Vector<Type, N>& pt);
```

e) Create a member function that modifies all the elements of a vector using `std::function <T (const T&)>`. Internally, the member function should call `std::transform` or use the internal array directly. Be careful: *data is on the stack and not on the heap*. The signature of the function is:

```
void modify(const std::function < Type (Type&)>& f);
```

The power of this member function is that we can modify the elements of a vector by providing different kinds of functions such as scaling, offsetting and so on (in fact, can be seen as a very simple example of a *Strategy* or even *Visitor* pattern).

3. (Compile-Time Matrix Class)

This exercise is similar in structure to exercise 2 except that we now wish to create a compile-time matrix class:

- Constructors: default, with value as argument and copy constructor.
- Accessing elements (read/write) using (,).
- Adding and subtraction; unary minus.
- Premultiplication by a scalar quantity.

Answer the following questions:

a) Given the above operations to be implemented you need to choose a suitable data structure ADT to hold the data, for example:

```
template <typename V, int NR, int NC>
using NestedMatrix
          = std::array<std::array<V, NC>, NR>;
```

```
//  V mat[NR][NC];
NestedMatrix<V, NR, NC> mat;
```

The choice depends on how efficient the ADT is and how much functionality can be delegated to the ADT.

b) Execute the equivalent of the steps b) to e) from exercise 2 but you now do it for matrices instead of vectors.

4. (A Bitmap Class)

We now wish to create a useful class that has many applications. We call it a *bitmap* and it can be created as a partial specialization of the matrix class that we discussed in exercise 3 in which the specialised class is the `Proposition` class of exercise 1. The possible specifications are:

```
using value_type = Proposition;
template <int NR, int NC>
    using BitMap = Matrix<value_type, NR, NC>;


template <int NR, int NC>
using BitMap2 = std::array<std::bitset<NC>, NR>;
```

You can see that there is a high degree of reusability. Extra functionality will be created using free functionality. Hence there is no need to create class hierarchies or extra classes (at least not for the moment).

Answer the following questions:

a) Create instances of `BitMap` and of `BitMap2` as well as the corresponding print functions.

b) We wish to superimpose two bitmaps on top of each other in some way in order to create a third bitmap. For each, we could AND the individual cells of the bitmaps. To this end, we first create a generic function (essentially a *binary operator*) that we apply to the cells of two bitmaps:

```
using BitFunction = std::function <Proposition(const Proposition&, const
Proposition&)>;
```

Second, create a free function to 'merge' two bitmaps in some way. The signature of the function to be created is:

```
template <int NR, int NC>
    BitMap<NR, NC> mask(const BitMap<NR, NC>& bm1, const BitMap<NR, NC>& bm2,
                                                   const BitFunction& masker);
```

Implement this function and test it using the Boolean operators from `Proposition`, for example:

```
BitMap<NR,NC> bm(true);
bm(4, 3) = false;


BitMap<NR, NC> bm2(true);
bm2(4, 3) = true;

auto COND = []
    (const Proposition& p1,const Proposition& p2) { return p1 % p2; };
auto bmA = mask(bm, bm2, XOR);
```

Test all the other Boolean operators as well. Examine the output.

c) One of the shortcomings of `BitMap` is the drudgery of setting its rows, for example:

```
BitMap<NR, NC> bitblt(false);
bitblt(1, 2) = bitblt(1, 3) = bitblt(1, 4) = bitblt(1, 6) = true;
```

Now use `BitMap2` to make life easier:

```
const int P = 8; const int Q = 8;
BitMap2<P, Q> bitblt2;

initialise(bitblt2, 0, std::string("01111100"));
```

Create two bitmaps (one from each class) and compare the solutions in terms of ease of use.

5. (Numeric Algorithms for Vectors and Matrices)

In this exercise we develop a number of questions concerning the classes that we have created in the first 4 exercises in the current section. In particular, we focus on the following issues:

- Delegating to the functionality in STL.
- Some functionality similar to Boost uBLAS.
- Numeric properties of compile-time vectors and matrices.
- Testing containers with `double` and `complex` underlying types.

The main objective is to build the functionality by writing as little code as possible and using as much of C++11 as possible.

Answer the following questions:

a) Create user-friendly wrappers for the STL `inner_product` functions. In both cases you use instances of `Vector` as arguments. You will need to add iterator functionality to `Vector`:

```
template <typename T, int N>
T inner_product(const Vector<T, N>& v1, const Vector<T, N>& v2,T initValue);

template <typename T>
    using BinaryFunction = std::function < T (const T& t1, const T& t2)>;

template <typename T, int N>
    T inner_product(const Vector<T, N>& v1, const Vector<T, N>& v2,
    T initValue, const BinaryFunction<T>& op1, const BinaryFunction<T>& op2)
```

b) Test the first variant.

c) For the second variant, test it using addition and multiplication operators (you get 4 options for lambda binary functions `ADD` and `MUL`), for example:

```
std::cout << inner_product(vec1, vec2, 0.0, ADD, MUL);
```

Create small arrays so that you can check the results by hand.

d) Write code to compute the *outer product* of two vectors:

$$u \otimes v = uv^T = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{bmatrix} u_1v_1 & u_1v_2 & u_1v_3 \\ u_2v_1 & u_2v_2 & u_2v_3 \\ u_3v_1 & u_3v_2 & u_3v_3 \\ u_4v_1 & u_4v_2 & u_4v_3 \end{bmatrix}$$

e) Specialise the code in part d) for complex numbers because the product has to take the *complex conjugate transpose* of vector v into account:

$$u \otimes v = uv^H$$

4.2 STL Algorithms II

Summary and Conclusions

The main goals of the exercises in this section are:

- Awareness of what is in STL (don't reinvent the wheel).
- Hands-on coding experience with STL algorithms (101 examples in the main).

In later sections and applications it should be possible to use these algorithms more regularly because you know what is on offer.

1. (Searching in Numeric Arrays)

An important topic in some applications is the ability to search for data in sorted (and unsorted) arrays (for example, one-dimensional vectors and two-dimensional matrices), for example in numerical analysis, science and engineering. In order to scope the exercise, we focus exclusively on `std::vector`. Some common scenarios are:

- S1: Does a sorted range contain a given value (`std::binary_search`)?
- S2: Does a sorted range contain another sorted range (`std::includes`)?
- S3: Locating first or last positions in a sorted range (`std::lower_bound`, `std::upper_bound`).
- S4: Return a range containing all elements corresponding to a value in a sorted range (`std::equal_range`).
- S5: Find the first occurrence of a value in a sorted range (`std::find()`, `std::find_if()`, `std::find_if_not()`). The last algorithm is since C++11.
- S6: Some algorithms need sorted ranges as input, hence we need algorithms to test if they are sorted (`std::is_sorted`, `std::is_sorted_until`) (since C++11).

Most of the STL algorithms return an iterator to the target value that we wish to locate. In this exercise however, we are interested in finding the (integral) index in the sorted range corresponding to that value. This entails that we must create user-friendly *wrapper functions* for the above STL algorithms. You also need to realise that several of these algorithms may take extra parameters to test values against each other.

The design rationale in this set of exercises is:

- Well-defined error-handling procedures.
- Efficiency of the algorithms (Complexity of algorithms).
- Choosing the most appropriate algorithm in a given context.

In general, we are interested in searching for values and indexes in sorted instances (usually sorted in ascending order) of instances of `std::vector`.

Answer the following questions:

a) All the search algorithms search for the location of a value **x** in a vector **v**. The return type is a nested tuple consisting of a range that encloses **x** and a Boolean value that indicates success or failure:

```
using Range = std::tuple<std::size_t, std::size_t>;
using Result = std::tuple<Range, bool>;
```

Implement the following function:

```
Result find_sequential_greater(const Vector& v, value_type x)
{
    for (std::size_t j = 0; j < v.size(); ++j)
    {
        if (v[j] <= x && v[j+1] > x)
        {
            return std::make_tuple(std::make_tuple(j, j+1), true);
        }
```

```
        }

        return std::make_tuple(std::make_tuple(999, 999), false);
    }
```

Now implement the same functionality using `std::find_if`. Remember that the output is in index space and you will need `std::distance` to convert iterators to indexes. Furthermore, create a wrapper for `std::find` to find the index corresponding to a given value in the vector.

b) Implement the *O (log n)* algorithms `std::lower bound` and `std::upper bound` to effect the same functionality as in part a). Again, you need (as always) to create a wrapper function.
c) Implement the *O (log n)* algorithm `stud::equal range` to effect the same functionality as in part b). Again, you need (as always) to create a wrapper function.
d) Test the algorithm `stud::find_if_not()` using an example of your choice.

2. (Predicates for Ranges)

Some STL algorithms only work properly with sorted containers. Since C++11 we have functionality to test if all (part of) a range is sorted. There are four algorithms, one of which is:

```
Vector vec8{ 1.0, 2.0, 3.0, -4.0, 10.0 }; // Not ordered
auto pos = std::is_sorted_until(vec8.begin(), vec8.end());
std::cout << "bad value: " << *pos << std::endl;
```

Answer the following questions:

a) Apply the other three algorithms to this container while experimenting with various binary predicate arguments.
b) Test the efficiency of the code in this question.

3. (Classifying and Implementing Algorithms)

Recall STL algorithm categories:

- C1: Nonmodifying.
- C2: Modifying.
- C3: Removing.
- C4: Mutating.
- C5: Sorting.
- C6: Sorted-Range.
- C7: Numeric.

Consider a general sequence container for example, `std::list`. To which categories do the following algorithms belong and which specific algorithm would you use in each case?

a) Scale all values by a given factor.
b) Count the number of elements whose values are in a given range.
c) Find the average, minimum and maximum values in a container.
d) Find the first element that is (that is not) in a range.
e) Search for all occurrences of '3456' in the container.
f) Determine if the elements in two ranges are equal.
g) Determine if a set is some permutation of '12345'.
h) Is a container already sorted?
i) Copy a container into another container.
j) Move the last 10 elements of a container to the front.
k) Swap two ranges at a given position.
l) Generate values in a container based on some formula.
m) Replace all uneven numbers by zero.
n) Remove all elements whose value is less than 100.
o) Shuffle a container randomly (pre- and post-C++11 versions).
p) Compute one-sided divided differences of the values in a container.

Now take specific containers for example, `std::vector` and implement each of the above operations. The challenge here is in finding the most appropriate STL algorithm to use and then to implement it.

For each of the above questions a) to b) ask yourself the following questions:
- What is algorithm input? What is the output?
- Is the input modified on output?
- Which category does the algorithm belong to?
- Choose the algorithm and implement it.
- Check and understand the output.
- Think about (other examples) where you can use the algorithm.

You will know that you have successfully completed the exercises when you have executed these steps. Understand the output.

After having completed these exercises you will have gained a good panoramic view of STL algorithms.

4. (Nonmodifying Algorithms)

We include a number of exercises that process STL containers and produce some output, usually a number or a data structure. For convenience (in order to scope the problem), use STL vectors as input container and note that there are several possible solutions to a given problem, some more efficient and/or readable than others. It is up to you to decide how to design the algorithm and the structure/type of the output.

a) Count the frequency of each value in a container. For example, for {1,2,4,5,4,1} we get the output {{1,2}, {2,1}, {4,2}, {5,1}}. There are different ways to model the output depending on your requirements.

b) Write a function to compute the minimum, maximum, sum and average of the values of the elements in a container with numeric values. The output is a tuple with four fields.

c) Consider the container S = {1,2,-3,4,5,5,5,6}. Use an STL algorithm to find the first occurrence of the value 5. Now use:
   - *std::bind2nd()*
   - *std::bind()*
   - *Lambda expression*

   to find the position of the first even number in S.

d) Consider the container `S = {1,2,5,5,-3,4,5,5,5,6,3,4,5}`. Determine how to do the following:
   - Return the position of the first three consecutive elements having the value 5.
   - Return the position of the first element of the first subrange matching {3,4,5}.
   - Return the position of the first element of the last subrange matching {3,4,5}.

e) Consider the container `S = {1,2,5,5,-3,4,5,5,5,6,3,4,5}`. Find the first element in S that has a value equal to the value of the following element.

f) Consider the container `S = {1,2,5,5,-3,4,5,5,5,6,3,4,5}` and `S1 = {1,2,5}`. Determine whether the elements in S1 are equal to the elements in S.

5. (Which Style to use?)

Some STL algorithms need *unary* and *binary predicates*. Both types return a Boolean. A unary predicate has one input argument while a binary predicate has two input argument. We can model these predicates and other kinds of functions in a number of ways:
- User-defined function objects.
- Predefined STL function objects (for example, `std::multiplies<T>()`).
- Using lambda functions (possibly with captured variables).

Answer the following questions:

a) Compare these three solutions against each other with regard to quality issues such as readability, understandability and maintainability.
b) Consider the case of transforming a vector of integers into a set of integers. Only those elements whose absolute value is strictly greater than a given threshold value. An example is the vector `{1,2,1,4,5,5,-1}` . If the threshold value is 2 then the output set will be `{4,5}`. Implement this problems using the three bespoke methods.
c) Having developed and debugged the code in part b) review the three solutions from the perspective of understandability, maintainability and efficiency.

6. (Modifying the Elements of a Container)

We now give some exercises on applying the modifying, replacing and removing algorithms in STL. The initial focus is more on functionality and less on efficiency. Initially, the underlying data type can be `int` and you can always generalise it later. Most of the exercises use specific examples and it is your task to use them as the test cases for your code.

Answer the following questions.

a) Consider the set S1 = {a,b,c,d,e,k} and the set S2 = {a,e}. Remove those elements from S1 that are not in S2. The output set is {b,c,d,k}.
b) Create a class `Point` that models two-dimensional points. Provide constructors, member functions to access the coordinates of a point and a member function to compute the distance between two points. You also need to define a binary predicate that tests two points for equality (they have the same values for their x and y coordinates). Now create an array of points (duplicates allowed). Transform this array to a set of points with no duplicates. Finally, filter this set (that is, remove points) of those points that are not within a distance from some predefined point.
c) We wish to create some functions that process strings in some way (in a sense, we are emulating a simple version of the *Boost C++ String Algorithm* library). A string can be seen as a special kind of vector whose elements are characters. Create functions to do the following (in all cases the input is a string):
   - Trim all leading and trailing blanks (space, tabs etc.) from the string.
   - Trim all leading and trailing blanks based on a unary predicate, e.g. is a digit, is a member of some set of characters (you could call the function `trim_if()`).
   - Produce a vector of strings from a character-separated string.
   - Join two strings.

7. (Mutating Algorithms)

Mutating algorithms rearrange the elements of a container in some way without modifying the values. We consider the container `S = {1,-1,7,8,9,10}` for convenience and to keep focused on a specific case.
Answer the following questions:

a) Reverse S as a modifier option and copy to a second container.
b) Rotate S so that the value 8 is the beginning of the container.
c) Write a function to compute the *power set* of S (that is, the set of all subsets of S containing $2^6 = 64$ elements).
d) Move the subset `{8,9,10}` to the front of the container.

8. (Classifying Algorithms)

Consider a stack ADT (*First-In-First-Out*) and a selection of its operations:

- *max(), min()* : leaves on the stack the larger and lesser of the two top values, respectively.
- *over()* : duplicates the second stack value on top of the stack.
- *rot()* : rotate the stack's third data value to the top of the stack.
- *swap()* : interchange the top two values on the stack.
- *drop()* : discard the value on the top of the stack.

If we consider these operations as simple algorithms, in which categories do they belong?
Implement a simple version of this stack ADT showing how these operations work.

4.3 Random Number Generators and Distributions

Summary and Goals

In this set of exercises we motivate the random number generation and distributions library in C++11. Prior to C++11 developers had to rely on the infamous C library function `rand()` or some proprietary library (for example, Boost Random) in order to generate pseudo-random numbers.

The focus in this section lies in getting hands-on experience with C++11 Random and applying it to some small yet interesting examples. There are many applications of random numbers and these are unfortunately outside the scope of this course. Some possible applications are:

- Statistics.
- Monte Carlo simulation (for example, option pricing).
- Genetic programming and genetic algorithms.
- Generating random values of input parameters to algorithms (white-box testing).

Of particular relevance when using random number generators in multi-threaded code is that we must take race conditions into account (we address this issue in exercise 5).

It would seem that C++11 Random has been influenced by Boost Random which contains more functionality than C++11, for example various random number engines (such as lagged Fibonacci, ranlux8 and ecuyer) as well as the following distributions:

- Hyperexponential (service time of k parallel servers).
- Beta and Laplace distributions.
- Non-central chi-squared.
- Triangle and uniform_on_sphere distributions.
- Utilities (for example, producing random floating point values with a given precision).

An overview (possibly somewhat outdated but still useful) is
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf

See also “The Art of Computer Programming” Volumes 1,2 and 3 by Donald E. Knuth Addison-Wesley 1998. In particular, Volume 2 discusses all things random in detail.
A related library is the Boost Math Toolkit (functions for distributions).

1. (First 101 Example)

This exercise is meant to get you acquainted with the syntax of random number generation in C++11. It is an improvement on Boost Random syntax as the classes are now function objects. We first give some representative code. In general, the steps are usually the same in all cases:

- Create a random number engine `eng`.
- Create a distribution `d`.
- Use the function call operator () to create a variate of the distribution, `d(eng)`.

The corresponding representative code is:

```
// Choose the engine
std::default_random_engine eng;

// Generate uniform random variates in interval [A, B]
double A = 0.0;
double B = 1.0;
std::uniform_real_distribution<double> dist(A, B);

int nTrials = 30;
for (int i = 1; i <= nTrials; ++i)
{ // Produce a number of uniform variates
```

```
        std::cout << dist(eng) << ", ";
    }
    std::cout << "end\n\n";
```

Answer the following questions:

a) Copy and adapt this code by using the engine `std::linear_congruential_engine` and the engine adapter with predefined parameters `std::mt19937` and `std::mt19937_64`.
b) Adapt the code in a) so that it works with the following distributions: Normal, chi-squared, Bernoulli and Cauchy.
c) Write a *generic function* to generate a list of random numbers for a generic engine and a generic distribution. Test the examples from parts a) and b) on this new function.

2. (Categories of Distributions)

In this exercise we write a generic function that works with any distribution and random number engine. We also generate a large number of variates of the distribution and we store these values in a map:

```
// Key = bucket/value; Value = number of occurrences
std::map<long long, int> counter;
```

The signature of the function to be created is given by:

```
template <typename Dist, typename Eng>
    void GenerateRandomNumbers(Dist d, Eng eng,
                  int NTrials, const std::string& s)
{
    // TBD by student
}
```

Answer the following question:

a) Design and implement the above code to generate what is essentially a histogram of values of a distribution.
b) Test the code by choosing the distributions: geometric, uniform and poisson.
c) Examine the generated output in each case; does it look like the probability of these distributions?

3. (Do the STL Shuffle)

This is a straightforward exercise on what is usually called *shuffling*. Old, global C functions (such as `rand()`, for example) store their local state in static variables. This implies that these random-number generators are inherently thread-safe which means that producing independent streams of random numbers is not possible. Using function objects however, is a better solution by encapsulating their local states as one or more member variables. The end-result is that the state of the passed generator is changed while generating a new random number.

We discuss the following;

- Shuffling the order of elements in a range.
- Adapting a base engine so that generated numbers are delivered in a different sequence.

Answer the following questions:

a) Create a large vector of integers. Shuffle the order of elements using `random_shuffle()` which has two forms; the first shuffles the elements using an implementation-dependent random-number generator (typically `rand()`); the second form has an extra argument that is a function (with a single input parameter) that returns a random number, using rand() or one of the C++11 random number engine such as:
   - User-defined random number generator.
   - `mt19937`.
   - `default_random_engine`.
   - `linear_congruential_engine`.

Implement the second form using each of these generators.

b) Implement the C++11 `shuffle()` using each of the generators in part a).

4. (Computing pi = 3.14159265358979323846)

We wish to compute pi using a random number generator to compute pi, which we know is the area of a circle of radius 1. We now inscribe a quarter of a circle in a unit square, the former having area equal to pi/4 and the latter having area equal to one.
In order to compute pi using random numbers, we proceed as follows in a manner similar to throwing darts at a board:

a) Create a random number engine and set its seed (for example, using `std::random_device`).
b) Create two instances of `std::uniform_real_distribution<double>` on the unit interval.
c) Create a loop, generate uniform random values x and y. Determine the Pythagorean distance to the origin is greater than 1 and if so, increment the counter.
d) Compute the value of pi based on the number of trials and the final value of the counter. How many trials are needed to compute pi ~ 3.14159?

5. (Generating 'random' Bitsets)

In this section we examine some of the possibilities for generating random numbers and experimenting with their ability to generate random bits in compile-time and dynamic bitsets.
We use the C++11 class (random number engine adapter):

```
template<class Engine, std::size_t W, class UIntType>
class independent_bits_engine;
```

to generate random numbers with a different number of bits `W` than that of the wrapped engine `Engine`.

Answer the following questions:

a) Create an instance of `independent_bits_engine` with a given random number engine, number of desired bits and integer types. Call the constructor with a suitable seed as argument. For example, the seed could be user-defined or from the `std::chrono` high-resolution clock. Display the generated value as an integer and as a bitset.
b) Create a function to test this functionality in the following way; take the width = 2, generate a random number using `independent_bits_engine` and then use this number to create a bitset. Determine how many bits have been set in this bitset (use member function `count()`) and add the number to a running counter. Then call the function a large number of times. Do you get a value in the region of 50%?
c) Experiment with width = 8,16 and 64. How many trials are needed to arrive at 50% of outcomes equal to 1?

We thus see how to automatically generate bitsets. And we might have the good fortune that they are random.

6. (Boost Random 101)

This is a small exercise to test some functionality in Boost Random. We include it for completeness.

Answer the following questions:

a) Create a variate of the triangle distribution with lagged Fibonacci as random number engine.
b) Use Boost function template `generate_canonical()` to generate a value uniformly in the range [0,1) with at least `bits` random bits:

```
template<typename RealType, std::size_t bits, typename URNG>
                  RealType generate_canonical(URNG & g);
```

Experiment with various values of `bits` (for example, 8,16,32).

7. (The Chi-Squared Test)

This exercise can be seen as a small application that uses some of the functionality of C++11 Random. To this end, we discuss the *Chi-squared test* (also known as the *Pearson goodness-of-fit test*) that determines where observed frequencies differ significantly from expected frequencies in an experiment. This test is well-documented in the literature.

Answer the following questions:

a) Write a function to compute the chi-squared statistic:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} = N \sum_{i=1}^{n} p_i \left(\frac{O_i/N - p_i}{p_i} \right)^2$$

where:

χ^2 = Pearson's cumulative test statistic, which asymptotically approaches a χ^2 distribution.

O_i = the number of observations of type i.

N = total number of observations.

$E_i = N p_1$ = the expected (theoretical) frequency of type i, asserted by the null hypothesis that the fraction of type i in the population is p_i.

n = the number of cells in the table.

b) Check the accuracy of this formula on examples by inspecting the output histogram. Precise answers are not needed.

c) Now choose an integer k and create uniform integers in the range $(0; k)$. Generate N draws and count the number of occurrences if each integer j $(0 <= j <= k)$ and compute the frequency. Note that the expected frequency is $N=k$. Compute the chi-squared statistic.

Final remark: there are many applications of *C++11 Random*. It is unfortunately not possible to discuss them here. We hope that the exercises here will serve as *building blocks* for your own applications.