NAME: . . ..

STUDENT NUMBER: . . ..

MARK: /50 = %.

Question 1

A sequence of pseudo random numbers can be generated using the recurrence relation:

$$x_{i+1} = (a * x_i + c) \bmod m$$

where x_0 is the `seed` for the sequence and is often chosen as some integer representation of the current time.

Such a sequence will have uniform probability characteristics provided the parameters a, c and m are chosen expeditiously. For example in the software package, *Numerical Recipes*,

$$m = 2^{32} \qquad a = 1664525 \qquad c = 1013904223$$

Write a `C++` function that, given x_0, m, a, c and n, returns a sequence of n pseudo random numbers. [5]

```
vector<int> rng(int seed,int m,int a,int c, int n){
    int r=seed;
    vector<int> rn;
    for(int i=0; i<n; ++i){
        r= (a * r + c) % m;
        if(r<0)r=r+m;
        rn.push_back(r);
    };
    return rn;
}
```

Question 2

Construct a `C++` function, `decimalDigits`, that given an integer n returns a list of n's decimal digits. For example `decimalDigits(4525063)` should generate the list {`4, 5, 2, 5, 0, 6, 3`}. [5]

```
vector<char> decimalDigits(int n){
    vector<char> dd;
    while(n > 0){
        dd.push_back(char('0'+n%10));
        n=n/10;
    }
    vector<char> rdd=dd;
    copy(dd.rbegin(),dd.rend(),rdd.begin());
    return rdd;
}
```

Question 3

The randomness of a sequence of pseudo random number can be tested by decomposing each number in the sequence into a list of its decimal digits and then counting how often each of the 10 possible digits occur. One can then produce a χ^2 statistic for the pseudo random sequence as follows:

$$\chi^2 = \sum_{i=0}^{9} \frac{(O_i - E_i)^2}{E_i}$$

where O_i is the observed count for decimal digit i and E_i is the expected count for decimal digit i.

Construct a C++ program that tests the randomness of the random number generator from question 1 above. You may make use of the code from questions 1 and 2 in your construction. [15]

```
int main() {

    vector<int> myRandomNumbers;

    // Numerical Recipes parameters
    int m = pow(2,20);
    cout << "modulus = " << m << endl;
    int a = 1664525;
    cout << "multiplier = " << a << endl;
    int c = 1013904223;
    cout << "increment = " << c << endl;

    // get seed from current time
    int s = time(NULL) % m;
    cout << "seed = " << s << endl;

    // generate random sequence
    myRandomNumbers = rng(s,m,a,c,10000);
    copy(myRandomNumbers.begin(),myRandomNumbers.end(),
            ostream_iterator<int>(cout, "\n"));

    // testing the rng
    map<char,int> myDigitCounts;
    for(int i=0; i<myRandomNumbers.size(); ++i){
        vector<char> decDigs=decimalDigits(myRandomNumbers[i]);
        for(int j=0; j<decDigs.size(); ++j){
            ++myDigitCounts[decDigs[j]];
        }
    }
```

```
    double myTotal=0;
    for(map<char,int>::iterator mitr=myDigitCounts.begin();
        mitr!=myDigitCounts.end(); ++mitr){
            cout << mitr->first << " = " << mitr->second << endl;
            myTotal += mitr->second;
    }
    cout << endl;
    double myExpected = myTotal / 10.0;
    double myChiSquare = 0.0;
    for(map<char,int>::iterator mitr=myDigitCounts.begin();
        mitr!=myDigitCounts.end(); ++mitr){
            myChiSquare += pow(mitr->second-myExpected,2);
    }
    myChiSquare = myChiSquare / myExpected;
    cout << "chiSquare for decimal digits = " <<
        myChiSquare << endl << endl;

    return 0;
}
```

Question 4

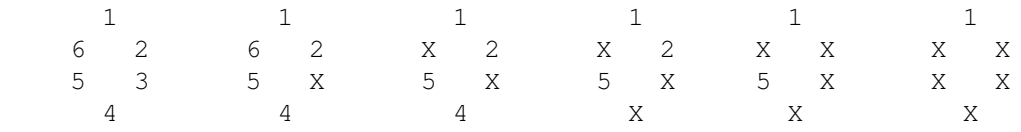
Construct a C++ function that, given a `string` representing a valid `filePath` to a text, makes use of `map<char, int>` from the Standard Template Library to compute and return the most commonly used alphabetic character in the text. [10]

```
char mostCommonChar(string filePath){

        // load the file into vector of chars
        ifstream book(filePath);
        vector<char> chars;
        copy(istream_iterator<char>(book), istream_iterator<char>(),
            inserter(chars, chars.begin()));

        // throw away non-alpha chars
        chars.erase(remove_if(chars.begin(), chars.end(), IsNotAlpha),
        chars.end());

        // convert remainder to lower case
        transform(chars.begin(), chars.end(), chars.begin(), ::tolower);

        // count occurence of each lower case letter using map
        map<char,int> cMap;
        for (int i=0; i<chars.size(); ++i) {
            ++cMap[chars[i]];
        };

        // mind char with max count
        int max = cMap['a'];
        char best = 'a';
        for (map<char,int>::iterator mi = cMap.begin(); mi != cMap.end();
            if (mi->second > max) {
                max = mi->second;
                best = mi->first;
            };
        };

        // return most pop char
        return best;
    }
```

Question 5

Suppose that m tokens are arranged in a circle and are enscribed clockwise with the itegers 1 to m. Starting with token number 1 and counting each token in turn around the circle every k^{th} token is removed from the circle. After $m-1$ removals have been carried out there is only one token left. Denote the number on this token by $P(m,k)$.

For example to compute $P(6,3)$ consider the following sequence of removals:



So $P(6,3)=1$.

Write a C++ function that computes $P(m,k)$ for strictly positive integers m and k. [15]

```
int pFunc(int m, int k){
    deque<int> dq;
    for(int i=1; i<=m; ++i){
        dq.push_back(i);
    }
    while(dq.size()>1){
        for(int j=1; j<k; ++j){
            int t=dq.front();
            dq.pop_front();
            dq.push_back(t);
        }
    dq.pop_front();
    }
    return dq.front();
}
```

vector API

Constructor: `vector<T> ()`	`vector<int> myVector;` Constructs an empty vector.
Constructor: `vector<T> (size_type size)`	`vector<int> myVector(10);` Constructs a vector of the specified size where all elements use their default values (for integral types, this is zero).
Constructor: `vector<T> (size_type size, const T& default)`	`vector<string> myVector(5, "blank");` Constructs a vector of the specified size where each element is equal to the specified default value.
`size_type size() const;`	`for(int i = 0; i < myVector.size(); ++i) { ... }` Returns the number of elements in the vector.
`bool empty() const;`	`while(!myVector.empty()) { ... }` Returns whether the vector is empty.
`void clear();`	`myVector.clear();` Erases all the elements in the vector and sets the size to zero.

`T& operator [] (size_type position);`
`const T& operator [] (size_type position) const;`
 	
`T& at(size_type position);`
`const T& at(size_type position) const;`
	`myVector[0] = 100;`
	`int x = myVector[0];`
	`myVector.at(0) = 100;`
	`int x = myVector.at(0);`
	
	Returns a reference to the element at the specified position. The bracket notation [] does not do any bounds checking and has undefined behavior past the end of the data. The `at` member function will throw an exception if you try to access data beyond the end. We will cover exception handling in a later chapter.

`iterator erase(iterator position);`
`iterator erase(iterator start, iterator end);`
	`myVector.erase(myVector.begin());`
	`myVector.erase(startItr, endItr);`
	
	The first version erases the element at the position pointed to by `position`. The second version erases all elements in the range [`startItr`, `endItr`). Note that this does **not** erase the element pointed to by `endItr`. All iterators after the remove point are invalidated. If using this member function on a `deque` (see below), all iterators are invalidated.

`void resize(size_type newSize);`
`void resize(size_type newSize, T fill);`
	`myVector.resize(10);`
	`myVector.resize(10, "default");`
	
	Resizes the vector so that it's guaranteed to be the specified size. In the second version, the `vector` elements are initialized to the value specified by the second parameter. Elements are added to and removed from the end of the vector, so you can't use `resize` to add elements to or remove elements from the start of the vector.

`void push_back();`	`myVector.push_back(100);` Appends an element to the `vector`.

`T& back();`
`const T& back() const;`
	`myVector.back() = 5;`
	`int lastElem = myVector.back();`
	
	Returns a reference to the last element in the `vector`.

`T& front();`
`const T& front() const;`
	`myVector.front() = 0;`
	`int firstElem = myVector.front();`
	
	Returns a reference to the first element in the vector.

`void pop_back();`	`myVector.pop_back();` Removes the last element from the vector.
`iterator begin();` `const_iterator begin() const;`	`vector<int>::iterator itr = myVector.begin();` Returns an iterator that points to the first element in the vector.
`iterator end();` `const_iterator end() const;`	`while(itr != myVector.end());` Returns an iterator to the element *after* the last. The iterator returned by `end` does not point to an element in the vector.

`iterator insert(iterator position,`
`const T& value);`
`void insert(iterator start,`
`size_type numCopies,`
`const T& value);`
	`myVector.insert(myVector.begin() + 4, "Hello");`
	`myVector.insert(myVector.begin(), 2, "Yo!");`
	
	The first version inserts the specified value into the vector, and the second inserts `numCopies` copies of the value into the vector. Both calls invalidate all outstanding iterators for the vector.

STL algorithms

`Type accumulate(InputItr start, InputItr stop, Type value)`	Returns the sum of the elements in the range [`start`, `stop`) plus the value of `value`.
`bool binary_search(RandomItr start, RandomItr stop, const Type& value)`	Performs binary search on the sorted range specified by [`start`, `stop`) and returns whether it finds the element `value`. If the elements are sorted using a special comparison function, you must specify the function as the final parameter.
`OutItr copy(InputItr start, InputItr stop, OutItr outputStart)`	Copies the elements in the range [`start`, `stop`) into the output range starting at `outputStart`. `copy` returns an iterator to one past the end of the range written to.
`size_t count(InputItr start, InputItr end, const Type& value)`	Returns the number of elements in the range [`start`, `stop`) equal to `value`.
`size_t count_if(InputItr start, InputItr end, PredicateFunction fn)`	Returns the number of elements in the range [`start`, `stop`) for which `fn` returns true. Useful for determining how many elements have a certain property.
`bool equal(InputItr start1, InputItr stop1, InputItr start2)`	Returns whether elements contained in the range defined by [`start1`, `stop1`) and the range beginning with start2 are `equal`. If you have a special comparison function to compare two elements, you can specify it as the final parameter.
`pair<RandomItr, RandomItr> equal_range(RandomItr start, RandomItr stop, const Type& value)`	Returns two iterators as a `pair` that defines the sub-range of elements in the sorted range [`start`, `stop`) that are equal to `value`. In other words, every element in the range defined by the returned iterators is equal to `value`. You can specify a special comparison function as a final parameter.
`void fill(ForwardItr start, ForwardItr stop, const Type& value)`	Sets every element in the range [`start`, `stop`) to `value`.
`void fill_n(ForwardItr start, size_t num, const Type& value)`	Sets the first `num` elements, starting at `start`, to `value`.
`InputItr find(InputItr start, InputItr stop, const Type& value)`	Returns an iterator to the first element in [`start`, `stop`) that is equal to `value`, or `stop` if the value isn't found. The range doesn't need to be sorted.

`InputItr find_if(InputItr start, InputItr stop, PredicateFunc fn)`	Returns an iterator to the first element in [`start`, `stop`) for which `fn` is true, or `stop` otherwise.
`Function for_each(InputItr start, InputItr stop, Function fn)`	Calls the function `fn` on each element in the range [`start`, `stop`).
`void generate(ForwardItr start, ForwardItr stop, Generator fn);`	Calls the zero-parameter function `fn` once for each element in the range [`start`, `stop`), storing the return values in the range.
`void generate_n(OutputItr start, size_t n, Generator fn);`	Calls the zero-parameter function `fn` `n` times, storing the results in the range beginning with `start`.
`bool includes(InputItr start1, InputItr stop1, InputItr start2, InputItr stop2)`	Returns whether every element in the sorted range [`start2`, `stop2`) is also in [`start1`, `stop1`). If you need to use a special comparison function, you can specify it as the final parameter.
`Type inner_product(InputItr start1, InputItr stop1, InputItr start2, Type initialValue)`	Computes the inner product of the values in the range [start1, stop1) and [start2, start2 + (stop1 – start1)). The inner product is the value $\sum_{i=1}^{n} a_i b_i + initialValue$, where a_i and b_i denote the ith elements of the first and second range.
`bool lexicographical_compare(InputItr s1, InputItr s2, InputItr t1, InputItr t2)`	Returns whether the range of elements defined by [`s1`, `s2`) is lexicographically less than [`t1`, `t2`); that is, if the first range precedes the second in a "dictionary ordering."
`InputItr lower_bound(InputItr start, InputItr stop, const Type& elem)`	Returns an iterator to the first element greater than or equal to the element `elem` in the sorted range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.
`InputItr max_element(InputItr start, InputItr stop)`	Returns an iterator to the largest value in the range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.
`InputItr min_element(InputItr start, InputItr stop)`	Returns an iterator to the smallest value in the range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.

`void random_shuffle(RandomItr start, RandomItr stop)`	Randomly reorders the elements in the range [`start`, `stop`).
`ForwardItr remove(ForwardItr start, ForwardItr stop, const Type& value)`	Removes all elements in the range [`start`, `stop`) that are equal to `value`. This function will **not** remove elements from a container. To shrink the container, use the container's `erase` function to erase all values in the range [`retValue`, `end()`), where `retValue` is the return value of `remove`.
`ForwardItr remove_if(ForwardItr start, ForwardItr stop, PredicateFunc fn)`	Removes all elements in the range [`start`, `stop`) for which `fn` returns true. See `remove` for information about how to actually remove elements from the container.
`void replace(ForwardItr start, ForwardItr stop, const Type& toReplace, const Type& replaceWith)`	Replaces all values in the range [`start`, `stop`) that are equal to `toReplace` with `replaceWith`.
`void replace_if(ForwardItr start, ForwardItr stop, PredicateFunction fn, const Type& with)`	Replaces all elements in the range [`start`, `stop`) for which `fn` returns true with the value `with`.
`ForwardItr rotate(ForwardItr start, ForwardItr middle, ForwardItr stop)`	Rotates the elements of the container such that the sequence [`middle`, `stop`) is at the front and the range [`start`, `middle`) goes from the new middle to the end. `rotate` returns an iterator to the new position of `start`.
`ForwardItr search(ForwardItr start1, ForwardItr stop1, ForwardItr start2, ForwardItr stop2)`	Returns whether the sequence [`start2`, `stop2`) is a subsequence of the range [`start1`, `stop1`). To compare elements by a special comparison function, specify it as a final parameter.

`InputItr set_difference(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in the sorted range [`start1`, `stop1`) but not in the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`InputItr set_intersection(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in both the sorted range [`start1`, `stop1`) and the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`InputItr set_union(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in either the sorted range [`start1`, `stop1`) or in the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`InputItr set_symmetric_difference(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in the sorted range [`start1`, `stop1`) or in the sorted range [`start2`, `stop2`), but not both, in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`void swap(Value& one, Value& two)`	Swaps the values of `one` and `two`.

`ForwardItr`
`swap_ranges(ForwardItr start1,`
`  ForwardItr stop1,`
`  ForwardItr start2)`
	Swaps each element in the range [`start1`, `stop1`) with the correspond elements in the range starting with `start2`.

`OutputItr transform(InputItr start,`
`  InputItr stop,`
`  OutputItr dest,`
`  Function fn)`
	Applies the function `fn` to all of the elements in the range [`start`, `stop`) and stores the result in the range beginning with `dest`. The return value is an iterator one past the end of the last value written.

`RandomItr`
`  upper_bound(RandomItr start,`
`  RandomItr stop,`
`  const Type& val)`
	Returns an iterator to the first element in the sorted range [`start`, `stop`) that is strictly greater than the value `val`. If you need to specify a special comparison function, you can do so as the final parameter.