NAME: . . ..

STUDENT NUMBER: . . ..

MARK: /50 = %.

Question 1

Assuming that `vector<int> vec` is an STL vector of integers and assuming that the integers currently stored in `vec` are unique and stored in ascending order, write a C++ code segment that will insert a new integer `newInt` into `vec` retaining the uniqueness and ordering properties.

For example if `newInt = 7` and if `vec` contains `3, 6, 13, 15, 19` before the code segment is executed then after the segment is executed `vec` should contain `3, 6, 7, 13, 15, 19`. [7]

```
// assuming.....
vector<int> myInts;
int newInt = rand();
// we can inset newInt and keep myInts orded using.............
vector<int>::iterator p=lower_bound(myInts.begin(),myInts.end(),newInt);
if( p == myInts.end() || *p != newInt) myInts.insert(p,newInt);
```

What other STL container would make this task easier? Explain why! [3]

```
// a set<int> is an ordered collection of ints
// The STL maintains the ordering for us
// assuming.....
set<int> ints;
int newInt = rand();
// we can inset newInt and keep myInts orded using.............
ints.insert(newInt);
```

Question 2

In the game of *snakes* a snake like object moves on a two dimensional grid eating food and growing longer. The snake must not bump into walls or itself while moving.

What STL container would you choose to represent a snake? Explain why your choice is appropriate. [5]

```
\\ use a double ended queue of grid coordinates

struct pointT {
    int row, col;
};

deque<pointT> snake;

\\ this allows you to move the snake by pushing a new
\\ grid position on the front of the snake and poping
\\ the last position off the tail of the snake
```

Give a code segment that indicates how you would move your snake to a new tile in the *snakes* world assuming that the tile in question is empty (NOT food wall or snake) and borders the current snake head. [5]

```
// assume that the world is an array of strings
// each char in a string is drawn from a set of game tiles
vector<string> world;

// assume that we want to move the head to a new coordinate
pointT nextHead = snake.front();
nextHead.row += dy;
nextHead.col += dx;

// update the world and snake queue with the new head position
world[nextHead.row][nextHead.col] = snakeTile;
snake.push_front(nextHead);

// update the world and snake queue by removing the tail
world[snake.back().row][snake.back().col] = emptyTile;
snake.pop_back();
```

Question 3

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(); ++ mi ) {
        if (mi->second > max) {
            max = mi->second;
            best = mi->first;
        };
    };

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

Question 4

The median of a collection of data is the value that is greater than half the elements in the collection and less than half the elements in a collection. For data collections with an odd number of elements, this is the middle element when the elements are sorted, and for data collections with an even number of elements it is the average of the two middle elements when the elements are sorted. Write a `C++` function that computes the median of a `vector<double>` collection. [5]

```
double median(vector<double> vec){
    sort(vec.begin(),vec.end());
    if (vec.size()%2 == 1) return vec[vec.size()/2];
    else return  (vec[vec.size()/2-1]+vec[vec.size()/2])/2.0;
}
```

Question 5

If $(x(t), y(t))$ with $a \leq t \leq b$ is a parametrization of a simple closed curve C then **Green's** theorem tells us that the area enclosed by the curve can be computed via:

$$area(C) = \| \int \int_G dA \| = \frac{1}{2} \| \oint_C (xdy - ydx) \| = \frac{1}{2} \| \int_a^b (x(t)y'(t) - y(t)x'(t))dt \|.$$

Use this result to show that if a simple closed polygon P has n vertices in the xy plane given by (x_i, y_i) with $i = 1 \ldots n$ and $(x_1, y_1) = (x_n, y_n)$ then the area enclosed by P can be computed via:

$$area(P) = \frac{1}{2} \| \sum_1^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \|$$

[5]

Consider the segment from (x_i, yi) to (x_{i+1}, y_{i+1}). Parameterize this segment as:

$$(x(t), y(t)) = (x_i, y_i) + t(x_{i+1} - x_i, y_{i+1} - y_i) \qquad \texttt{with} \qquad \texttt{0} \leq \texttt{t} \leq \texttt{1}$$

then the contribution of this segment to the area can be calculated by:

$$\int_0^1 (x(t)y'(t) - y(t)x'(t))dt = \int_0^1 (x_i + t(x_{i+1} - x_i))(y_{i+1} - y_i) - (y_i + t(y_{i+1} - y_i))(x_{i+1} - x_i)dt$$

and after a little algebra this reduces to

$$\frac{1}{2}(x_i y_{i+1} - x_{i+1} y_i).$$

Summing over all segments the result follows.

Question 6

Make use of the result from the previous question to construct a `C++ STL` program that reads pairs of xy coordinates from the keyboard and then computes the area enclosed by a polygon whose vertices are at those coordinates. You may assume that the polygon so described is simple (no intersections) and closed (last point = first point)

For example if the input to the program is:

```
1.0     1.0
5.0     1.0
5.0     4.0
1.0     1.0
```

then the area is calculated as:

$$area = \frac{1}{2}\|(1 \times 1) - (1 \times 5) + (5 \times 4) - (5 \times 1) + (5 \times 1) - (4 \times 1)\| = 6$$

as expected [10]

Question 6, more space .....

```
#include <iostream>
#include <algorithm>
#include <numeric>
#include <iterator>
#include <vector>
#include <fstream>
#include <cctype>

using namespace std;


int main() {

    // Reading coords ...
    vector<double> coords;
    copy(istream_iterator<double>(cin), istream_iterator<double>(),
        inserter(coords, coords.begin()));

    // split into x and y components
    // let me know if you can do this using STL (ie without loops)
    vector<double> xC;
    vector<double> yC;
    for (int k=0; k<coords.size(); ++k){
        if ( k%2 == 0 ) xC.push_back(coords[k]);
        else yC.push_back(coords[k]);
    };

    // rotate x and y components 1 position
    vector<double> xRc = xC;
    vector<double> yRc = yC;
    rotate(xRc.begin(), xRc.begin() + 1, xRc.end());
    rotate(yRc.begin(), yRc.begin() + 1, yRc.end());

    // drop last element from all 4 vectors
    xC.pop_back();
    yC.pop_back();
    xRc.pop_back();
    yRc.pop_back();

    // compute the two inner products
    double pa = inner_product(xC.begin(),xC.end(),yRc.begin(),0.0);
    double na = inner_product(yC.begin(),yC.end(),xRc.begin(),0.0);

    // comput and display area
    double area = (pa-na)/2.0;
    cout << "area = " << area << endl;
}
```

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.