CS106X
Winter 2017

Chris Gregg
Feb 23rd, 2017

CS106X Midterm Exam B (KEY)

This is an open-note, open-book exam. You can refer to any course handouts, textbooks, handwritten lecture notes, and printouts of any code relevant to any CS106X assignment. You may not use any laptops, cell phones, or internet or electronic devices of any sort. You will be graded on functionality—but good style helps graders understand what you were attempting. You do not need to `#include` any libraries and you do not need to forward declare any functions. You have two (2) hours to complete the exam. We hope this exam is an exciting journey.

All answers to this exam must go in the space provided. Think ahead before writing anything down. You may use scratch paper, but we will only count answers put in the space provided.

Your name: ____

Your Stanford SUNet ID (e.g., cgregg): ___________________________

Honor Code Pledge:

I accept the letter and the spirit of the honor code. I have neither given nor received aid on this exam. I pledge to write more neatly than I ever have in my entire life.

(signed)____

Question	Points
1. Radix Sort	12
2. Complex Number Class	12
3. Mandelbrot Set	6
4. Mystery Functions	8
5. Big O	6
6. Linked Lists	8
7. Generic Game Engine	10
8. Bonus	3
Total	62

Question 1: Radix Sort (12 Points) (3-part question)

In class, we have discussed a number of "comparison sorts," which have a provable O(n log n) worst case performance, even for the best comparison sort algorithms (e.g., merge sort, quicksort, etc.). However, there are sorts that do not compare individual elements. Instead, they use other properties inherent in the data to perform the sort. One such sort is called a "Radix" sort, because it uses the base of a value's number representation (the "radix") to sort the values.

The idea behind a decimal-based radix sort for a Vector of *integers* is as follows:

1. Each integer is evaluated first by its least-significant digit (e.g., for the number 54296, the algorithm first evaluates the "6", because this is the ones digit).
2. Based on the value of the least significant digit, the integer is placed into a temporary structure, and there are ten such structures (one for each of the digits 0-9).
3. Once all the integers are evaluated in this matter, empty out each temporary structure from 0 to 9 in a first-in-first-out order (starting from the structure that holds the ints placed into the 0s structure, and progressing up to the structure that holds the integers placed into the 9s structure) to replace the values into the original Vector.
4. The process is then repeated for each subsequent digit (i.e., the tens digit, then the 100s digit, etc.).
5. After the algorithm finishes, the numbers will be sorted, from lowest at the beginning of the original Vector to the highest at the end of the Vector.

Part A (8 points):

On the **next page**, write the following function:

```
void radixSort(Vector<int> &numbers, int maxDigits);
```

where

`numbers` is a `Vector` of `int`s, and

`maxDigits` is the maximum number of digits any of the numbers in the `Vector` holds (e.g., if `maxDigits` is `4`, the maximum integer in the entire `Vector` is `9999`).

You may use any collection we have discussed in class so far, and you may write helper functions if desired. You do not have to `#include` any libraries.

You may assume you have the following function, which calculates the power of a number, e.g., `power(10,2)` returns `100`.

```
int power(int a, int b) {
   if (b == 0) {
         return 1;
   }
    return a * power(a, b-1);
}
```

Write your answer on the following page

Please answer Question 1 on this page.

```
void radixSort(Vector<int> &numbers, int maxDigits) {
    Vector<Queue<int>> tempQueues;

    // create the queues
    for (int i=0; i < 10; i++) {
        Queue<int> q;
        tempQueues.add(q);
    }
    for (int i=0; i < maxDigits; i++) {
        // perform a pass over the numbers
        for (int number : numbers) {
            tempQueues[(number /
                         (int)power(10,i)) % 10].enqueue(number);
        }

        // return the numbers to the vector
        int numbersPos = 0;
        for (int i=0; i < 10; i++) {
            while (!tempQueues[i].isEmpty()) {
                numbers[numbersPos] = tempQueues[i].dequeue();
                numbersPos++;
            }
        }
    }
}
```

Part B (2 points):

Using *n* as the number of elements in `Vector<int> numbers`, and using *m* as the maximum digits (`maxDigits`), what is the worst-case complexity of Radix Sort?

Answer: O (m * n)

Part C (2 points): In *one sentence*, justify your answer for part B:

(Any reasonable response, e.g., "You must loop through each of *n* numbers *m* number of times")

Question 2: Complex Number Class (12 points)

(*Note:* do not let this question and the next one scare you, especially if you don't remember or have never studied complex numbers–the entire problem is explained such that you will be able to complete it without prior knowledge other than basic algebra).

The following are details about complex numbers:

1. A complex number is defined as *a* + *b*i, where *i* is equal to √-1, (the square root of negative 1). "*a*" refers to the coefficient of the *real* part of the number, and "*b*" refers to the coefficient of the *imaginary* part of the number. Both *a* and *b* should be represented as `double`s.
2. The absolute value of a complex number is its distance from zero on the complex plane. To calculate *n,* the absolute value of a complex number *a* + *b*i, use the following formula:

   $$|n| = \sqrt{a^2 + b^2}$$

3. To add two imaginary numbers, simply add the real parts together, and add the imaginary parts together.
4. To multiply two imaginary numbers, $(a_1 + b_1 i) * (a_2 + b_2 i)$, multiply as follows: $(a_1 * a_2 - b_1 * b_2) + (a_1 * b_2 + a_2 * b_1)i$. The result is also a complex number, because i^2 reduces to -1.

As an example of multiplication, (4 + 5i) ∗ (2 – 3i)

= (4 ∗ 2 - 5∗-3) + (4 ∗ -3 + 5 ∗ 2)i

= (8 +15) + (-12i+10)i

= 23 – 2i

On the following page, write the *six* functions necessary to define a Complex number class, as defined by the following header file, `complex.h`:

```
#pragma once

class Complex {
public:
    Complex(double a, double b);  // constructor
    double abs();                 // returns the absolute value of this number
    double realPart();            // returns the real part of this number
    double imagPart();            // returns the coefficient of the imaginary part of this number
    friend Complex operator+(Complex m, Complex n);   // returns the sum of two Complex numbers
    friend Complex operator*(Complex m, Complex n);   // returns the product of two Complex numbers

private:
    double a; // the real part of the complex number
    double b; // the coefficient of the imaginary part of the complex number
};
```

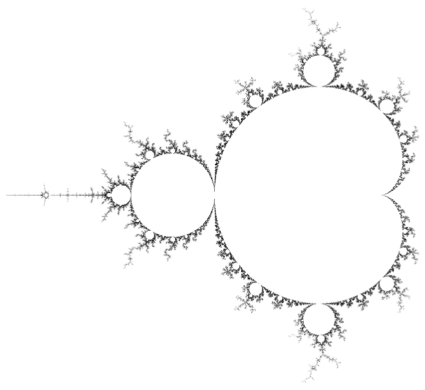
You may assume you have the following function, which calculates the positive square root of a number, e.g., sqrt`(25)` returns 5:

```
double sqrt(double x); // returns the positive square root of x
```

Please answer Question 2 on this page. You should use proper class syntax for your function definitions.

```
#include "complex.h"
#include <cmath>     // provides the double sqrt(double x) function

Complex::Complex(double a, double b) { // constructor      // 2 points
    this->a = a;
    this->b = b;
}

double Complex::abs() {                                    // 1 point
   return sqrt(a*a + b*b);
}

double Complex::realPart() {                               // 1 point
    return a;
}

double Complex::imagPart() {                               // 1 point
    return b;
}

// multiplies two complex numbers and returns              // 3 points
// the result as a new complex number
Complex operator*(Complex m, Complex n) {
   double realP = m.a * n.a - m.b * n.b;
   double imagP = m.b * n.a + m.a * n.b;
   return Complex(realP, imagP);
}

// adds two complex numbers and returns                    // 3 points
// the result as a new complex number
Complex operator+(Complex m, Complex n) {
   double realP = m.a + n.a;
   double imagP = m.b + n.b;
   return Complex(realP, imagP);
}
```

Question 3: Mandelbrot Set (6 Points)



The fractal to the right is a depiction of the "Mandelbrot Set," in tribute to Benoit Mandelbrot, a mathematician who investigated the phenomenon. The Mandelbrot Set is recursively defined as the set of complex numbers, *c*, for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from z=0. In other words, a complex number, *c*, is in the Mandelbrot Set if it does not grow without bounds when the recursive definition is applied.

Formally, the Mandelbrot set is the set of values of *c* in the complex plane that is bounded by the following recursive definition:

$$z_{n+1} = z_n^2 + c$$

$$z_0 = 0, \quad n \rightarrow \infty$$

In order to test if a number, *c*, is in the Mandelbrot Set, we recursively update *z* until either of the following conditions are met:

A. The absolute value of *z* becomes greater than 4 (it is diverging), at which point we determine that *c* is not in the Mandelbrot Set.

B. We exhaust a number of iterations, defined by a parameter (usually 1000 is sufficient), at which point we declare that *c* is in the Mandelbrot Set.

In the space below, write the recursive *helper* function that returns true if a Complex number, *c*, is in the Mandelbrot Set. You should use the functions you created in Question 2 to answer this question, and you can assume that you have written them correctly.

```
bool inMandelbrot(Complex c, int maxIterations) {
    Complex z0(0, 0);
    return inMandelbrot(z0, c, maxIterations);
}
// you write this helper function
bool inMandelbrot(Complex z, Complex c, int maxIterations) {
    // base case
    if (z.abs() > 4.0) {                            //  2 points
        return false;
    }
    if (maxIterations == 0) {                       // 1 point
        return true;
    }
    else {
        return inMandelbrot(z * z + c, c, maxIterations - 1); // 3 pts
    }
}
```

Question 4: Mystery Function (8 Points)

A. The following program prints out eight lines of text.

```
#include<iostream>
using namespace std;

int mysteryFunction(int a, int &b, int *c) {
    b++;
    *c = 2 * a + b;
    a--;
    cout << a + b << endl;
    cout << b + *c << endl;
    cout << *c + 1 << endl;
    return a - b;
}

int main() {
    int d = 3;
    int e = 5;
    int *f = &d;

    if (d == *f) {
         d = mysteryFunction(d, e, f);
    }

    if (!(d == e)) {
        cout << "false" << endl;
        mysteryFunction(*f, e, &d);
    }
    cout << d + e + *f << endl;

    return 0;
}
```

What is the output of the above program?

(1 point each)

```
8
18
13
false
2
6
0
5
```

Question 5: Big O (6 Points)

Give a tight bound of the nearest runtime complexity class for each of the following code fragments in Big-Oh notation, in terms of variable N. (In other words, the algorithm′s runtime growth rate as N grows.) Write a simple expression that gives only a power of N, not an exact calculation like O(2N^3 +4N +14). Write your answer in the blanks on the right side.

Question | Answer: (2pts each)

a.

```
  float some_math(float n) {
     if (n <= 1) return 0;
     return 1.0 + some_math(n / 2.0);
 }
```

O(____log n______)

b.

```
  int calculate(int n,int last) {
     if (n==0) {
        return 0; // base case
    }
     int temp = 0;
     for (int i=0; i < n+last; i++) {
         temp++;
     }
     return temp+calculate(n−1, last+1);
  }
```

O(____n^2______)

c.

```
  int numARecursive(Grid<char> &board) {
      int total = 0;
      for (int i = 0; i < board.numRows(); i++){
          if (board[i][0] == 'a') {
              total += 1;
          }
      }
      if (board.numCols() == 1) {
          return total;
      }
      Grid<char> newBoard(board.numRows(), board.numCols() - 1);
      for (int i = 0; i < board.numRows(); i++) {
          for (int j = 0; j < board.numCols() - 1; j++) {
              newBoard[i][j] = board[i][j + 1];
          }
      }
      return total + numARecursive(newBoard);
  }
```

Assumptions:
1. **n is the number of elements, i * j**
2. **i == j**

O(____$n^{3/2}$______)

Assumptions:
1. **n is the number of rows and cols**
2. **Answer:** O(n^3)

Question 6: Linked Lists (8 Points)

Given the `Node` definition below, write the following function that inserts a new `Node` with a given name into a linked list before another named node in a linked list. You may assume that the other name appears at most once in the list. Note that the function returns the head of the list:

```
struct Node {
    string value;
    Node *next;
};

// Insert a new node with the value of "newName", into a a linked list
// directly before the node with the value "otherName".
// If otherName does not exist in the list, does not add to the list.
// Returns the head of the list
Node *insertBefore(Node *head, string newName, string otherName) {
    // keep a prev pointer and a curr pointer
    Node *prev = NULL;
    Node *curr = head;
    while (curr != NULL) {
        if (curr->value == otherName) {
            // found it
            Node *newNode = new Node;
            newNode->value = newName;
            newNode->next = curr;

            if (prev == NULL) {
                // at head
                head = newNode;
            } else {
                prev->next = newNode;
            }
            break; // no need to continue
        }
        prev = curr;
        curr = curr->next;
    }
    return head;
}
```

Question 7: Generic Dice Game (10 Points)

Consider a game where players get points for rolling a single, six-sided die. To play the game, a player keeps rolling the 6-sided die until the game ends, at which point the player collects a certain number of points. For this problem, you will write a function that determines the optimal list of rolls for a player that results in the highest number of points, based on a particular starting state.

You will be given a `State` object that holds the current state of the game, and you will only need to concern yourself with two public functions in the State class:

```
class State {
public:
    // returns true if the game is over
    // bool isGameOver();

    // returns the number of points the player has
    // (only valid when the game is over)
    int getPoints();

private:
    ...
}
```

You may also call the following function, which takes a `State` object, and returns a new `State`, based on the roll:

```
// returns the next State in a game, based on a roll of the die (1-6)
State getNextState(State currentState, int dieRoll);
```

On the next page, write the following function, which returns a `Vector` of `int`s that represents the optimal set of rolls for a player:

```
Vector<int> optimalRolls (State startState);
```

Notes:

1. You should not call the `getPoints()` function until the game is over.
2. Your answer should investigate all possible solutions to the problem.
3. You do not need to return the number of points a player would receive for the optimal `Vector` of rolls.

Please answer Question 7 on this page.

```
Vector<int> optimalRolls (State startState) {
      Vector<int> rolls;
      getOptimal(startState, rolls);
    return rolls;
}

int getOptimal(State s, Vector<int> &bestRolls) {
      if(s.isGameOver()) {
            return s.getPoints();
      }

      // ASSUMPTION: score cannot be negative
      int max = -1;

      // alternate: assumption: score can be negative
      int max = INT_MIN;

      Vector<int> argMax;
      for(int i = 1; i <= 6; i++) {
            State next = getNextState(s, i);
            Vector<int> rolls;
            int tempScore = getOptimal(next, rolls);
            if(tempScore > max) {
                  max = tempScore;
                  argMax = []
                  argMax.add(i);
                  argMax.addAll(rolls);
            }
      }
      bestRolls = argMax;
      return max;
}
```

Alternate solution:

```
Vector<int> optimalRolls(State startState) {
  Vector<int> rolls;
  int bestScore = -1; // or INT_MIN
  getOptimal(startState, rolls, bestScore);
  return rolls;
}

void getOptimal(State s, Vector<int> &rolls, Vector<int> &bestRolls, int
&bestScore) {
  if (s.isGameOver()) {
    if (s.getPoints() >= bestScore) {
      bestScore = s.getPoints();
      bestRolls = rolls;
    }
    return;
  }

  for (int i = 1; i <= 6; i++) {
    State next = getNextState(s, i);
    rolls.add(i);
    getOptimal(next, rolls, bestRolls, bestScore);
    rolls.remove(rolls.size() - 1);
  }
}
```

8. Bonus (3 points)

8a. Supposedly, a professor once told a class that they could “bring in anything they could carry” into an exam. What did one student carry into the exam that probably helped significantly? (1 point)

The student carried in a graduate student.

8b. Chris has a couple of fun engineering projects sitting on his desk in his office. What piece of equipment do the projects focus on? (1 point)

Typewriter

8c. What is the name of the famous computer science professor at Stanford with the title “Professor Emeritus of The Art of Computer Programming”? (1 point)

Donald (Don) Knuth