COP 3223H Fall 2017 Final Exam

Last Name: _____ , First Name: _____

Directions: Please answer questions 1 - 5 in C, and questions 6 - 10 in C++.

Part I - C

1) (10 pts) What is the output of the following C program?

```
#include <stdio.h>

void print(int *arr, int len);
int f(int* arr, int len, int* ptr, int a);

int main() {
    int vals[] = {2, 6, 3, 9, 4};
    vals[3] = f(vals, 5, &vals[1], 4);
    print(vals, 5);
    vals[1] = f(vals, 5, &vals[2], 0);
    print(vals, 5);
    return 0;
}

void print(int *arr, int len) {
    int i;
    for (i=0; i<len; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int f(int* arr, int len, int* ptr, int a) {
    int b = a+1;
    if (a == len-1) b = a-1;
    int tmp = arr[a];
    int tmp2 = tmp + arr[b];
    arr[a] = *ptr;
    arr[b] = tmp2;
    *ptr = tmp;
    return arr[a] - arr[0];
}
```

_____

_____

2) (10 pts) Consider the situation of trying to pay off a credit card with a current balance. One such plan would involve making the same monthly payment regardless of expenditures for the month while hoping to keep those expenditures under the payment you are making. Eventually, you might catch up and have no balance. In particular, here is how the process works. Say the initial balance is $1,000.00 and the monthly interest rate is 0.01 (this is 12% a year, which is comparable to many credit cards). Let's say that the expenditures for the month were $500.00 and the monthly payment you had planned was $600.00. Here is how one month would be processed:

1) Balance goes to $1000 + $500 = $1500 for expenditures.
2) Balance goes to $1500 + $15 = $1515 due to 1% monthly interest.
3) Balance goes to $1515 - $600 = $915 after monthly payment is made.

Write a function in C that, given the current balance, monthly interest rate, monthly payment, an array of monthly expenditures, and the length of the array, calculates the balance remaining after processing the expenditures and payments indicated by the input parameters. You may assume that the balance never dips below 0.

```
double amtOwed(double balance, double monthIR, double payment,
               double* spent, int length) {









}
```

3) (10 pts) A number in binary (base 2) is such that only digits 0 and 1 are used (called bits) and the place value of the i^{th} location from the right is 2^i, where the right most location is location 0. The value of the lowest one bit of a number is simply, 2^i, where location i is the right-most location in the binary representation of the number with a 1. For example, the number 84, represented in binary is 1010100, since $2^6 + 2^4 + 2^2 = 84$. The lowest one bit of 84 is simply 4, since the right most bit set to 1 is in location 2 and $2^2 = 4$. Write a ***recursive*** function that takes in a positive integer n and returns its lowest one bit. (Hint: use mod to determine if the least significant bit of n is 1.)

```
int lowestOneBit(int n) {









}
```

Questions 4 and 5 will involve the following struct that stores information about a ficticious UCFCard which stores both a monetary balance (in cents) and points:

```
typedef struct UCFCard {
    int UCFID;
    int balanceCents;
    int points;
} UCFCard;
```

4) (5 pts) Write a function add that takes in a pointer to a UCFCard, ptrCard, and a number of cents, addCents, (guaranteed to be positive) and adds addCents to the balance of the UCFCard pointed to by ptrCard. Your function should contain a single line of code:

```
void add(UCFCard* ptrCard, int addCents) {



}
```

5) (15 ps) Write a function purchase that takes in a pointer to a UCFCard, ptrCard, and a number of cents, costCents, and attempts to make a purchase of that value with the Card. All purchases on the card must either entirely be made with points (one point equals one cent) OR the balance. We always try to make the purchase with points first. If we have enough points to do it, then we just deduct the appropriate number of points. If we don't, then we try to make the purchase with our balance. If we have enough balance, then we deduct from the balance accordingly AND add the appropriate number of points (to be discussed later). If we don't have enough points or balance to make the purchase, we don't make it. The function returns 1 if the purchase was made and 0 if it wasn't. We add 1 point per 20 cents of purchase. For example, a purchase of 499 cents would add 24 points (since $20 \times 24 = 480 \leq 499$ and $20 \times 25 = 500 > 499$.)

```
int purchase(UCFCard* ptrCard, int costCents) {









}
```

Part II - C++

6) (10 pts) A valid star arrangement is where we have $2k+1$ rows of stars, for some ***positive integer*** k, where the odd rows have a stars each and the even rows have a-1 stars each. (For example, the United States flag has k = 4, or 9 total rows, with a = 6 stars on the odd rows and 5 stars on the even rows.) Write a **complete program in C++** that asks the user for the total number of stars they want in their design and prints out each valid option of k and a that has n stars. Print the solutions in increasing value of k. You may assume that the input value is 10^6 or less. (This means you can try each value of k, but when considering a single value of k, you can't loop through values of a as this would take too long.) Note: Partial credit will be given if you do a double for loop structure. (Hint: use mod!)

For example, if the user entered 50, then your program should output:

```
1 17
4 6
5 5
16 2
```

These correspond to the designs (17,16,17), (6,5,6,5,6,5,6,5,6), (5,4,5,4,5,4,5,4,5,4,5), and (2,1,2,1,2,1,2,1, 2,1,2,1,2,1,2,1, 2,1,2,1,2,1,2,1, 2,1,2,1,2,1,2,1,2), respectively.

7) (10 pts) Gnome Sort is yet another $O(n^2)$ sorting algorithm that works as follows:

1. Set your initial position, p, in the array to 0.
2. While your position, p, isn't equal to n, the length of the array, do the following:
    a. Check if arr[p] ≥ arr[p-1]. If so, add 1 to your position p.
    b. If not, then do
        b1. Swap arr[p] and arr[p-1].
        b2. Subtract one from the position p.

Write a C++ function to implement Gnome Sort.

```
void gnomesort(int array[], int len) {









}
```

Questions 8 through 10 will utilize the rock class written for you below:

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

class rock {
    public:
        rock(string n, int h);
        void addCount(int c);
        friend ostream& operator <<(ostream& outstr, rock r);
        string name;
        int freq;
        int hardness;
};

rock::rock(string n, int h) {
    name = n;
    freq = 0;
    hardness = h;
}
```

```
void rock::addCount(int c) {
    freq += c;
}

ostream& operator <<(ostream& outstr, rock r) {
    outstr << r.name << " Hardness = " << r.hardness << " Quantity = "
           << r.freq;
    return outstr;
}
```

Imagine creating a Collection of Rocks class that manages a collection of rock objects as just defined. The instance variables for the class would be a vector of rock, an integer storing the total number of rocks, an integer storing the sum of hardness of every rock, and a double storing the average hardness of all of the rocks in the collection. Here is the listing of functions and instance variables in the class:

```
class rockcollection {
    public:
        rockcollection();
        friend rockcollection operator +(const rockcollection& c1,
                                         const rockcollection& c2);
        friend rockcollection operator +(const rockcollection& c, const rock& r);
        friend rockcollection operator +(const rock& r, const rockcollection& c);
        friend ostream& operator <<(ostream& outstr, rockcollection c);
    private:
        vector<rock> rocklist;
        int rockcount;
        int sumhardness;
        double avghard;
        void add(const rock& r);
        int find(const string myname);
};
```

Most of the functions in the class are given to you in a separate handout. For the following questions, you'll write the find method, and write two functions overloading the + operator.

8) (9 pts) The find function takes in a single string, myname, and determines if there is a rock in the collection with that name. If there is, it returns the index of the vector rocklist that matches myname. If there isn't, it returns -1. Complete the function below (it should be 4 lines long):

```
int rockcollection::find(const string myname) {









}
```

9) (10 pts) The + operator with a collection and a rock works by creating a new empty collection, adding the rock r to it, and then going through each rock in the collection c and adding it into the new collection, one by one. After this, the new collection is returned. This function should be five lines long. Complete it:

```
rockcollection operator +(const rockcollection& c, const rock& r) {









}
```

10) (10 pts) The + operator with a two collections works by creating a new empty collection, going through each rock in the collection c1 and adding it into the new collection, one by one. Then, going through each rock in the collection c2 and adding it into the new collection, one by one. After this, the new collection is returned. This function should be 6 lines long. Complete it:

```
rockcollection operator +(const rockcollection& c1, const rockcollection& c2)
{









}
```

11) (1 pt) The UCF Knights Football team earned a berth to the Peach Bowl with its victory last Saturday. After what fruit commonly found in Georgia is the Bowl named? _____

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

class rock {
    public:
        rock(string n, int h);
        void addCount(int c);
        friend ostream& operator <<(ostream& outstr, rock r);
        string name;
        int freq;
        int hardness;
};

class rockcollection {
    public:
        rockcollection();
        friend rockcollection operator +(const rockcollection& c1,
const rockcollection& c2);
        friend rockcollection operator +(const rockcollection& c,
const rock& r);
        friend rockcollection operator +(const rock& r, const
rockcollection& c);
        friend ostream& operator <<(ostream& outstr, rockcollection
c);
    private:
        vector<rock> rocklist;
        int rockcount;
        int sumhardness;
        double avghard;
        void add(const rock& r);
        int find(const string myname);
};

rock::rock(string n, int h) {
    name = n;
    freq = 0;
    hardness = h;
}

void rock::addCount(int c) {
    freq += c;
}

ostream& operator <<(ostream& outstr, rock r) {
    outstr << r.name << " Hardness = " << r.hardness << " Quantity = "
<< r.freq;
    return outstr;
}

rockcollection::rockcollection() {
```

```
    rockcount = 0;
    sumhardness = 0;
    avghard = 0;
}

rockcollection operator +(const rockcollection& c, const rock& r) {
    /*** Question 9 ***/
}

rockcollection operator +(const rock& r, const rockcollection& c) {
    /*** Same exact code as question 9 works here ***/
}

rockcollection operator +(const rockcollection& c1, const
rockcollection& c2) {
    /*** Question 10 ***/
}

int rockcollection::find(const string myname) {
    /*** Question 8 ***/
}

ostream& operator <<(ostream& outstr, rockcollection c) {
    outstr << "Total = " << c.rockcount << " Avg Hardness = " <<
c.avghard << endl;
    for (int i=0; i<c.rocklist.size(); i++)
        outstr << c.rocklist[i] << endl;
    return outstr;
}

void rockcollection::add(const rock& r) {

    int index = find(r.name);

    if (index == -1)
        rocklist.push_back(r);
    else
        rocklist[index].addCount(r.freq);

    rockcount += r.freq;
    sumhardness += (r.freq*r.hardness);
    avghard = (double)sumhardness/rockcount;
}
```