CS 3353 Fall 2023

Program 3 – Making a tough decision on <fill in the blanks>

Due: 4/18 (Thu) 11:59pm. Late pass deadline 4/21 (Sun) 11:59pm

Human just discovered information about a inhibited planet call “Quad-Solaria”. In this country, they don’t have a president/king/emperor etc. Their country is ruled by a counil of 500 Quad-Solarians. Every decision/choice about their planet have to done by a vote. And a decision/choice can only be enacted if at least a clear majority (in this case, 251) of them vote for it. For example, if there is choice of 3 options, and the vote is (240, 160, 100), further discussion will be needed to ensure one option has at least 251 votes before it can be chosen.

Class Stuff3

For this program, you are to sort the object of a class called Stuff3. The class has the following definition:

- Members:
  - int a, b, c: integers
  - static int compareCount: return the number of times the == operator (see below) is called.

  (Notice that all members (except compareCount) is private).

- Constructor
  - Stuff3(bool israndom = true, int x = 0, int y = 0, int z = 0): If isRandom is false, the three integers that passed to a, b, and c. However, if isRandom is true, then x, y, z will be ignored, and random values will be generated for a, b and c.
  - Stuff3(const Stuff&): copy constructor
- Overloaded operators:
  - bool operator==(const Stuff3& s): return true if the two objects have the same (this.a == s.a, this.b == s.b, this.c == s.c) and false if otherwise.
  - Stuff3& operator=(const Stuff3& s): assignment operator.
  - friend ostream& operator<<(ostream& os, Stuff& s): output the content of a stuff object
  - friend istream& operator>>(istream& os, Stuff& s): read the value of a stuff object

You are NOT allowed to add change the class in anyway.

Problem and solution

Your goal is to implement two versions of the following function

```
pair<bool, Stuff3> Decision(vector<Stuff3>& vec)
```

The input is a vector of Stuff3 objects. Your program should determine whether there is one object in vec that appears in STRICTLY MORE THAN half the location in the vector. (So if the vector is on length 50, the item must appear at least 26 times; if the vector is of length 49, it must appear at least 25 times). Your function should return a pair. If such an object exists, the first item of the pair should be true, and the second item of the pair should be such object. If no such object exists, the first item of the pair should be false, and the second object can be anything (it will be ignored).

There is a variety of algorithms that solves the problem. However, in this program you are to implement two versions of the solution, both using recursion (you will get 0 credit if your solution is non-recursive.

Version 1

You are given the following function

```
pair<bool, Stuff3> Decision1(vector<Stuff3>& vec)
{
        return DecisionRecur1(….);
}
```

Your recursion should be based answering the following questions:

a. Suppose you have break vec into two equal parts (if v has an odd number of objects, one part will have one more object), and suppose you obtain the answer to the problem for each part. Let say the answer is <t1, e1>, <t2, e2> respectively (where t1 and t2 can be true or false, and e1 and e2 are defined accordingly as above). Can you infer from that piece of information, what is the possible solution for the whole vector?

Based on this, implement DecisionRecur1, which will be the recursive function that will end up return the right answer. It must return the same thing as DescisionRecur1. DecisionRecur1 MUST be recursive (that calls itself). You are to determine what parameters should be used.

Your function must run in O(n log n) time, where n is the number of objects in vec,

Version 2

You should implement the following function:

```
pair<bool, Stuff3> Decision2(vector<Stuff3>& vec)
{
        return DecisionRecur2()
}
```

Your recursion should be based on the following result:

b. Assume vec has n element (assuming n is even). Split the elements into n/2 pairs. An object (o) can only be the solution for the whole vec if it there is a pair where both objects are o. (You need to consider what is the answer when n is odd. You only need to make a slight addition/adjustment)

Based on this, implement DecisionRecur2, which will be the recursive function that will end up return the right answer. It must return the same thing as Descision2. DecisionRecur2 MUST be recursive (that calls itself). You are to determine what parameters should be used.

Your function must run in O(n) time, where n is the number of objects in vec.

Other requirements for both programs.

- Under NO circumstances can you attempt to sort the objects. If your code do ANY sorting (either by calling a method in the standard library or you code it yourself), you will get 0 points.
- Your functions MUST follow the logic that is described.
- You are NOT allowed to modify the content of vec, but you are allowed to create new vector<Stuff3> objects inside your function, if it makes your life easier.

Grading and What to hand in

Your grade is a combination of four things:

1. Implement DecisionRecur1() correctly and with time complexity O(n log n)
2. Implement DecisionRecur2() correctly and with time complexity O(n)
3. Answer question (a)
4. Modify result (b) for n = odd.

If you either do (1) or (2) correctly, you get 75 points. If you do both correctly, you get 90 points.

(3) and (4) are worth 10 points each.

(Full mark is considered 100, so there is a 10 point bonus build in)

You are given four files:

- Stuff3.h, Stuff3.cpp: implementation of the Stuff3 class.
- Prog3.cpp: contain the skeleton code for Decision1() and Decision2()
- Prog3_test.cpp: contains a driver program that will call Decision1() and Decision2()

You need to send me 2 things:

- The modified version of Prog3.cpp containing your implementation
- A pdf file contains the answer of (3) and (4) above.

You should zip them and upload the zip file to Canvas

Extra Bonus:

If you can prove that your program runs in the complexity required, you get 15 points bonus for each program (30 points max). You should include that into your pdf file.

Notice that I am talking about proving, not just running tests and plotting graphs and tell me the graph show that the complexity is correct.