Project 1

Part 1: Report

(a) Show that there is always a stable assignment of users to servers.

i. For the first kind of instability given, suppose there are users u and u' and a server s. Suppose s is paired with u and u' is free. Now assume s prefers u' to u. If s prefers u' to u then s would have offered a slot to to u' before it offered one to u, and from then on u' would be assigned a slot in some server because according to the algorithm, once assigned a slot, you can only switch slots, you cannot be unassigned. Thus u' could not be free at the completion of the algorithm based on our initial assumptions and hence we have a contradiction.

ii. For the second kind of instability, suppose that there is an unstable matching produced by the algorithm. Suppose there exists pairs (s, u) and (s', u'). Then suppose s actually prefers u' to u and u actually prefers s' to s. Then s offered a slot to u' before offering one to u since u' is higher on its preference list. But a user can only reject a server if it receives an offer from a server it prefers. So if u' rejected s then it must prefer its final match, s to s but that contradicts our initial assumption.

(b) Give an algorithm in pseudocode (either an outline or paragraph works) to find a stable
assignment that is server optimal. Hint: it should be very similar to the Gale-Shapley algorithm, with servers taking the role of the men, and users of the women.

```
for each server
  while some server $s_i$ has available slots
    $s_i$ offers a slot to the next user $u_j$ on its preference list
    if $u_j$ is free then
      $u_j$ accepts the offer
    else $u_j$ is already committed t some other server $s_k$
      the number of available server slots in $s_k$ increases by one
      the number of available server slots in $s_k$ decreases by
one
```

(c) Give the runtime complexity of your algorithm in Big O notation and explain why.

- Let m be the number of servers and n be the number of users. The algorithm will run in $O(nm)$ steps because each server can only offer a slot to one user at a time and in each iteration of the algorithm, a server offers a single slot to a single user. Since $n \geq m$ we can further propose the algorithm will run in $O(n^2)$ time.

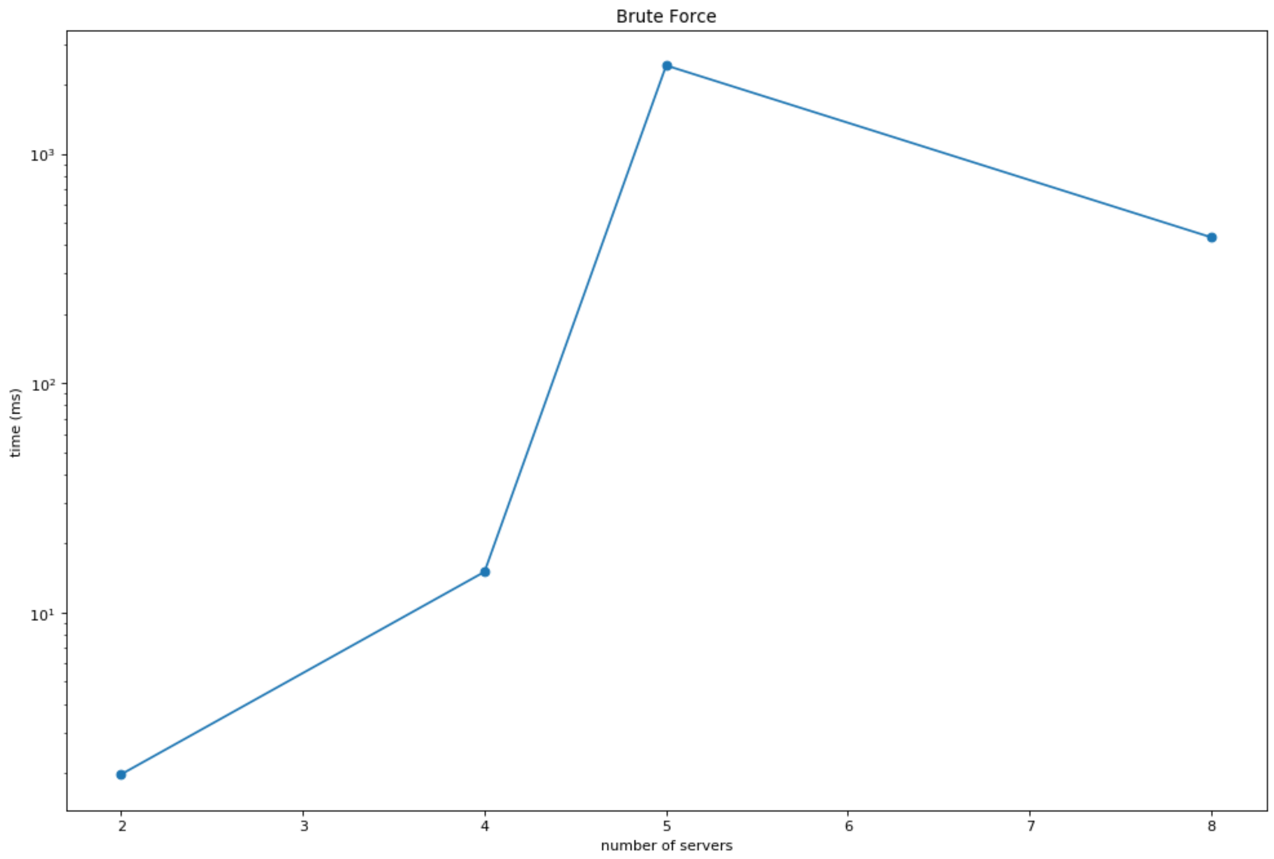
(d) Give a proof of your algorithm's correctness. Remember that you must prove both that your algorithm terminates and gives a correct result.

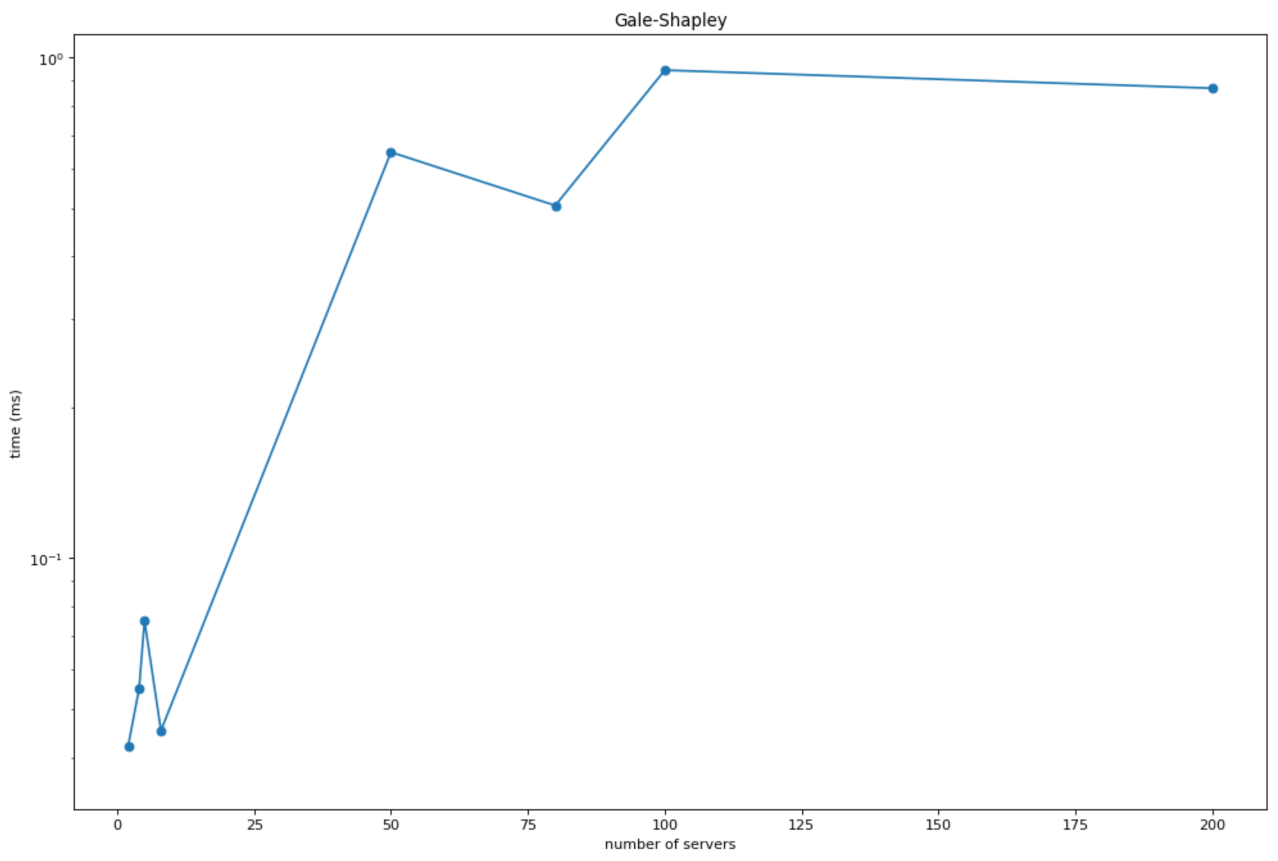
- The algorithm terminates upon completion of the while loop which means as soon as every slot in a server is filled by a user. We assume in the problem description that the total number of slots is less than or equal to the number of students. Thus for every iteration of the while loop, a slot will be filled by a user. Since there are either the same number of users or more than the number of slots, each slot will be filled and thus the while loop will terminated once the server runs out of slots.
- Suppose the algorithm returns a set of pairs S and that S is a stable matching. Assume there is an instability. Then there exist two pairs (s, u) and (s', u') in S such that s prefers u' and u prefers s'. During execution, s's last proposal must have been to u. Thus either s proposed to u' earlier which means u' rejected s in favor of some other server s'' or u is higher than u' on s's preference list. Thus either $s'' = s'$ or u' prefers s' to s'' which are both contradictions.

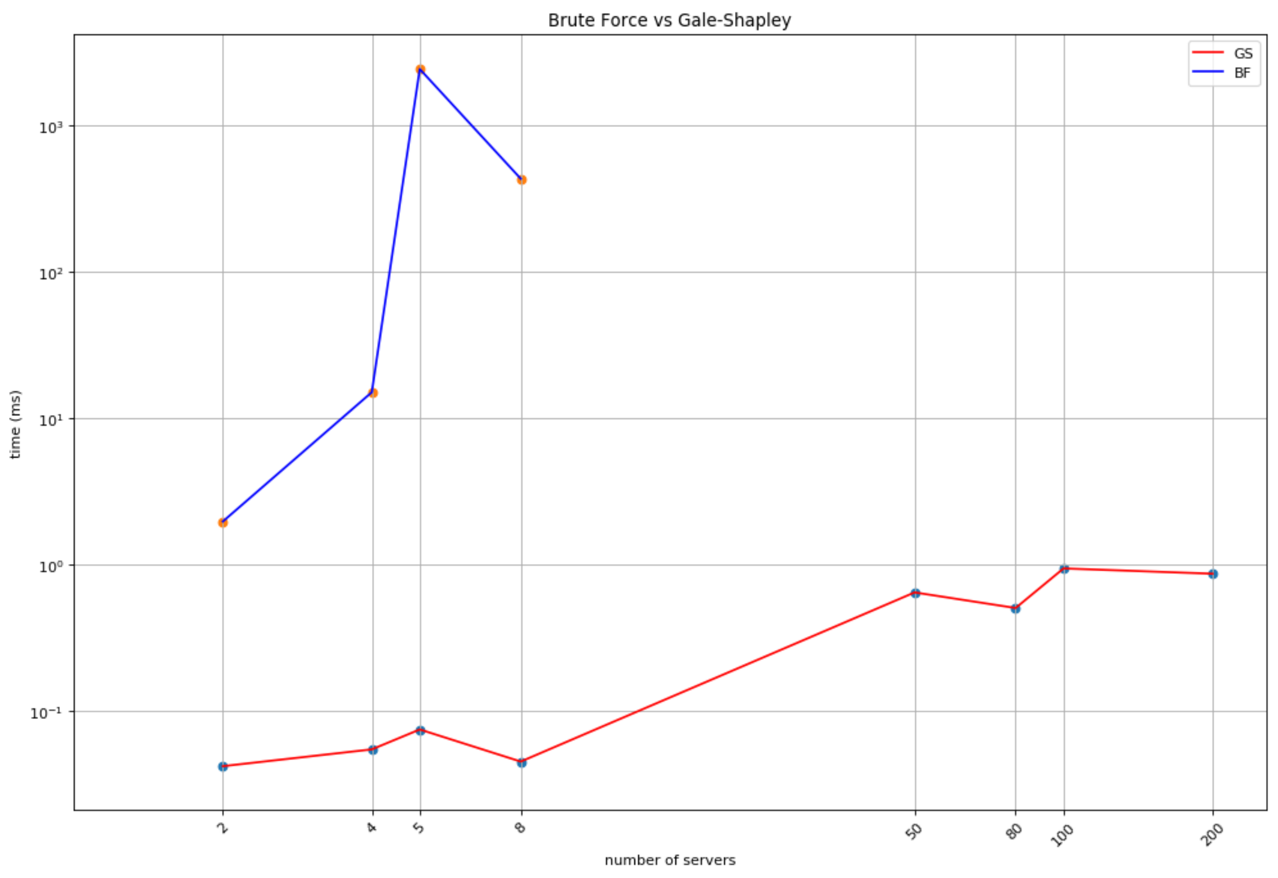
(e) Consider a Brute Force Implementation of the algorithm where you find all combinations of possible matchings and verify if they are a stable marriage one by one. Give the runtime
complexity of this brute force algorithm in Big O notation and explain why.

- The runtime complexity of the Brute Force Implementation is $O(n!)$ because there $n!$ total possible permutations of server and user matchings.

(f) In the following two sections you will implement code for a brute force solution and an efficient solution. In your report, use the provided data files to plot the number of servers (x-axis) against the time in ms it takes for your code to run (y-axis).


Brute Force
time (ms)
10³
10²
10¹
number of servers
2
3
4
5
6
7
8


Gale-Shapley
time (ms)
10^0
10^{-1}
number of servers
0
25
50
75
100
125
150
175
200



Part 2: Implement a Brute Force Solution

```
public boolean isStableMatching(Matching allocation) {
    int m = allocation.getServerCount();
    int n = allocation.getUserCount();
    ArrayList<Integer> user_matching = allocation.getUserMatch
ing();
    ArrayList<ArrayList<Integer>> user_preference = allocatio
n.getUserPreference();
    ArrayList<ArrayList<Integer>> server_preference = allocati
on.getServerPreference();

    int slots_filled = 0;
    for(int u=0; u < n; u ++) {
        if(user_matching.get(u) != -1) {
            slots_filled++;
        }
    }
    if(slots_filled != allocation.totalServerSlots()) return false;

    for(int u=0; u < n; u++) { //for each user
        if(user_matching.get(u) == -1) continue; // make sure u is assigned
        int s = user_matching.get(u); // s
        // instability 1
        for(int up=0; up < n; up++) {
            if(up == u) continue; // make sure u' is different from u
            int sp = user_matching.get(up); // s'
            if(sp == -1) {
                ArrayList<Integer> s_pref = server_preference.get(s);
                for(int i=0; i < s_pref.size(); i++) {
                    if(s_pref.indexOf(u) > s_pref.indexOf(up)) { //if s prefers up to u then instability
                        return false;
                    }
                }
            }
        }
        // instability 2
        for(int up=0; up < n; up++) {
            int sp = user_matching.get(up); // s'
            if(up == u || sp == -1) continue; // make sure u'
is different from u and u' is assigned
            ArrayList<Integer> s_pref = server_preference.get
(s);
            for(int i=0; i < s_pref.size(); i++) {
                if(s_pref.indexOf(u) > s_pref.indexOf(up)) {
//if s prefers u' to u
                    ArrayList<Integer> up_pref = user_preferen
ce.get(up);
                    for(int j=0; j < up_pref.size(); j++) {
                        if(up_pref.indexOf(s) < up_pref.indexO
f(sp)) { // and u' prefers s to s'
                            return false;
                        }
                    }
                }
            }
        }
    }
    return true;
}
```

Part 3: Implement an Efficient Algorithm

```
public Matching stableMarriageGaleShapley(Matching allocation)
{
        //long startTime = System.nanoTime();


        int[] serverSlots = new int[allocation.getServerCount
()];
        int m = allocation.getServerCount();
        int n = allocation.getUserCount();
        ArrayList<Integer> user_matching = new ArrayList<>();
        ArrayList<ArrayList<Integer>> user_preference = alloca
tion.getUserPreference();
        ArrayList<ArrayList<Integer>> server_preference = allo
cation.getServerPreference();


        for(int i = 0; i < allocation.getServerCount(); i++) {
            serverSlots[i] = allocation.getServerSlots().get
(i);
        }


        for(int i = 0; i < n; i++) {
            user_matching.add(i, -1);
        }
        while(allocation.totalServerSlots() > 0) {
        for (int s = 0; s < m; s++) {
                                              // for each serv
er
                for (int k = 0; k < n && allocation.getServerS
lots().get(s) > 0; k++) {                     // offer a slot
to each user while there are slots available
                    int u = allocation.getServerPreference().g
et(s).get(k);
                    if (user_matching.get(u) < 0) {
                        user_matching.set(u, s);
                        allocation.getServerSlots().set(s, all
ocation.getServerSlots().get(s) - 1);         // number of slo
ts at s decreases (one just filled)
                    } else {
                        int sp = user_matching.get(u); // s'
                        if (user_preference.get(u).indexOf(s)
< user_preference.get(u).indexOf(sp)) {       // if u does not
prefer sp to s
```

```
                            user_matching.set(u, s);
                         // match u and s
                            allocation.getServerSlots().set(s
p, allocation.getServerSlots().get(sp) + 1);   // number of sl
ots at sp increases (one just removed)
                            allocation.getServerSlots().set(s,
allocation.getServerSlots().get(s) - 1);     // number of slot
s at s decreases (one just filled)
                        }
                    }
                }
            }
        }
        allocation.setUserMatching(user_matching);
        for(int i = 0; i < allocation.getServerCount(); i++) {
            allocation.getServerSlots().set(i, serverSlots
[i]);
        }
        /*
        long endTime   = System.nanoTime();
        long totalTime = endTime - startTime;
        System.out.println(totalTime);
        */
        return allocation;
    }
}
```