(1)

ALGORITHM

Inputs: H[n][n] is a 2D array representing n Hospital preferences of n Residents, and R[n][n] is a 2D array ranking n Resident's preferences of n Hospitals.

Output: String, either "single match," if the input has one stable match, or "more than one" if more than one match exists for input.

//in this pseudocode, we will use the Gale-Shapley algorithm discussed in lecture, so include the algorithm here. Assume the algorithm takes two 2D arrays outlined above as inputs and returns an array M[i], where every index i represents a Hospital and every matching element M[i] represents the hospital's resident match produced by the algorithm.

Initialize:
M[n] ← array of length n with zeroes // M[n] will represent the first matches produces by the GS algorithm outlined above

M2[n] ← array of length n with zeroes //this will represent possible other stable matching and follows the same rules as M[n] outlined above.

do:
// call GS s.t. M[n] gets matches where Hospitals propose to Residents.
//call GS s.t. M2[n] gets matches where Residents propose to Hospitals

//compare matches
For i=1 to n+1 **do**:
    If M[i]!= M2[i] **do** :
        Return "more than one" string

Return "single match" string

PROOF OF CORRECTNESS

By GS, there must be a stable matching, GS is order-independent, the proposer gets their best possible stable matching, and the receiver gets their worst possible stable matching.

When Hospitals propose to residents, this implies Hospitals get their best possible stable matching, regardless of which Hospital starts. Additionally, Residents will get their worst possible stable matching.

CS 330 HW2
Chandini Toleti

When Residents propose to Hospitals, Residents get their best possible stable matching, regardless of which Hospital starts. In addition, Hospitals get their worst possible stable matching.

Therefore, if both sides proposing leads to the same matching, the following is true:
R has worst matching = R has best matching, and H has worst matching = H has best matching.
For Rw = Rb and Hw=Hb, there must be only one unique stable matching.

TIME ANALYSIS

This algorithm has a worst-case time efficiency of O(n^2). Let's observe each part

(1) Initializing M[n] will take space O(n)
(2) Initializing M2[n] will take space O(n)
(3) G-S Algorithm has worst-case space efficiency of O(n^2) with an input of n length
(4) G-S Algorithm has worst-case efficiency O(n^2)
(5) Comparing matches will take constant time O(n)

Since these events are **not** nested we say our worst efficiency is the polynomial created by adding all parts → O($2n^2$+ $3n$). Since we only care about the highest degree in polynomials we say our worst-case time efficiency is O(n^2).

SPACE ANALYSIS

The inputs of this algorithm are two 2D Arrays. Therefore, the take up O(n^2) space in memory. Additionally, we allocated O(2n) space for both arrays of matches. My algorithm uses these structures to complete its task. Therfore, the overall space complexity is O(n^2+2n). Since we only consider the highest order for polynomials, the overall space complexity is O(n^2).

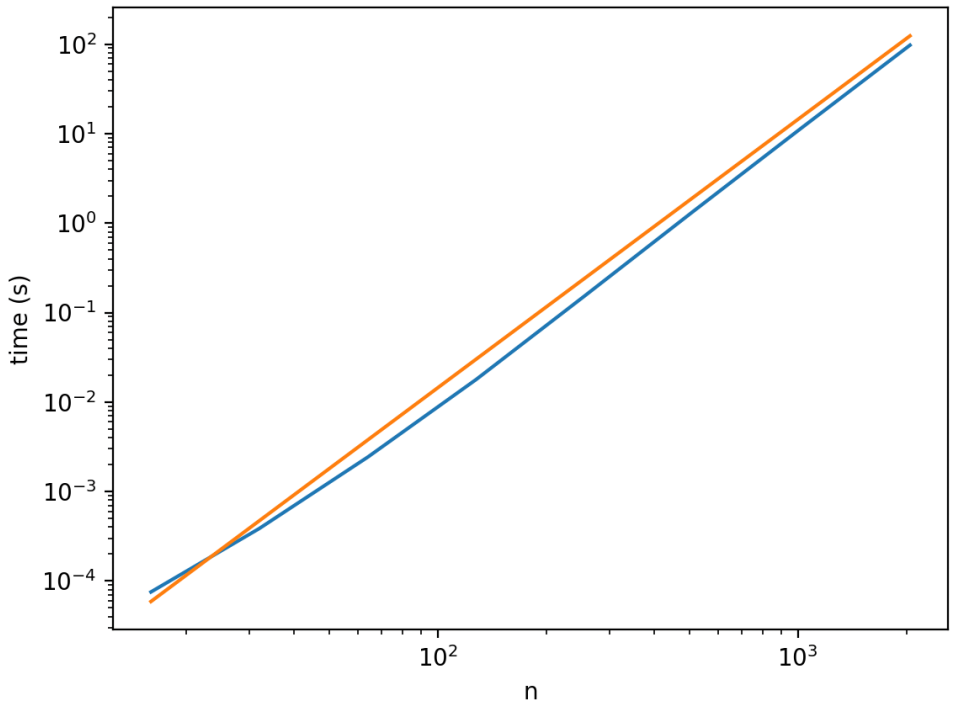
(2)

[A] Running the given algorithm through VS-Code on my MacBook Pro, my runtimes are as follows:

n	running time (seconds)
16	8.04E-05
32	0.000415
64	0.002751292
128	0.019267
256	0.155508792
512	1.846722208
1024	23.81248371
2048	152.1756135

CS 330 HW2
Chandini Toleti

By plotting different *w* values on a log-log graph, I determined my ***w* to be 1.44×10^{-8}**. The following plot shows my runtimes (blue) and the line y= wn^3(orange) with my *w*:



[B] ALGORITHM. Here is my implementation of the algorithm with better runtime:

```
def coinChoices(a, b, c, n):
    assert type(a) == type(b) == type(c) == type(n) == int
    assert a > 0 and b > 0 and c > 0 and n > 0

    count = 0
    max_1s = min(a, n) //we can't make n with more than n single bills! We include the
given number for all min() calls if it's less than the max, so it becomes our new max!

    for i in range(max_1s + 1):
        max_2s = min(b, (n - i) // 2) //(n-i) is remaining with 1s added, we can't add
this to n with more than (n-i)//2 2 dollar bills!
        for j in range(max_2s + 1): //loop through 2s
            remaining_3 = n - (i + 2 * j) //everything left without threes
             //check if threes can fill the rest (will also work if no threes since
0%3 == 0!)
            if remaining_3 % 3 == 0 and remaining_3 // 3 <= c: //must also check if we
have enough 3s!
                count += 1

    return count
```

CS 330 HW2
Chandini Toleti

PROOF OF CORRECTNESS

The algorithm begins by calculating the minimum between a (the number of 1-dollar coins) and n (the total we're trying to achieve). If we had more than n 1 dollar coins, there is no way to use more than n of those coins to achieve n. Therefore, we only consider the minimum of those two values.

Next, for every number of 1 dollar coins (starting with 0), we calculate the minimum between b(the number of 2 dollar coins) and (n-i)//2. (n-i) represents the money amount we have left to add, but we must integer divide by 2 to turn this amount into a number of 2 dollar coins. If we had more 2 dollar coins than the number of 2 dollar coins needed to get to n, we don't account for them. Therefore, we take the minimum of these two numbers and loop through the possible combinations of 2 dollar coins for every dollar coin.

Finally, instead of creating another loop for 3 dollar coins, we simply check during each 2 dollar iteration if the remainder of n after adding a number of 2 coins and 1 coins can be filled with 3 dollar coins[n-(I + 2*j)]! Next, we use an if-statement to account for 2 special cases. Case 1: no 3 coins are needed (since 3%3==0), and if we accidentally use more 3 coins than we have remaining_3//3>=c(checks that the number we're using is less than or equal to the number we have).

Since we're counting every possible combination that can lead to the sum being n, this algorithm accomplishes the task assigned correctly.

TIME ANALYSIS

This algorithm has a worst-case time efficiency of O(n^2). Let's observe each part

(1) Looping through all max_2s for each max_1 has a worst-case time efficiency of O(max_2s.length * max_1s.length), we'll use n to denote this worst-case number. So O(n^2)
(2) All other operations have constant, or O(1) time efficiency

We say our worst efficiency is the polynomial created by adding all parts → O(n^2+1). Since we only care about the highest degree in polynomials we say our worst-case time efficiency is O(n^2).

SPACE ANALYSIS

This algorithm has a worst-case space efficiency of O(1). Our inputs are all of the type "int" that each takes O(1) space in memory. We perform comparisons and calculations with the values stored in these variables and store them occasionally in other variables with space complexity O(1). Consequently, our total space efficiency still remains as O(1)!

CS 330 HW2
Chandini Toleti

[C] Running my improved Algorithm through VS-Code on my MacBook Pro, my runtimes are now as follows:

n	running time (seconds)
16	1.50E-05
32	3.44E-05
64	0.000103125
128	0.000357416
256	0.001334917
512	0.005535084
1024	0.023285666
2048	0.095557167

At first, I plotted my runtimes with the original line y= wn^3 and tried to find a *w*. However, no such *w* exists since the logarithmic growth of my implementation is not n^3. Instead, it is n^2. When I changed F(n) to n^2 I easily determined my *w* to be $\mathbf{2.82 \times 10^{-8}}$. Here is my new log-log graph with both my runtimes (blue) and the line y = wn^2 (orange)with my w: