Gale-Shapley Matching Algorithm:

Findings from Experimental Simulations in R

Nicholas Hall

Programming and Computation in R

December 2022

Introduction

The Gale-Shapley algorithm is a matching algorithm created in 1962 by David Gale and Lloyd Shapley. It was defined originally in the context of a stable-marriage problem. The problem asks how, given a list of n men and n women, as well as their preferences, one create stable marriages for all participants. Stable, in this case, refers to an outcome wherein no man and woman who are not married to each other would both prefer to be married to each other than their respective spouses. In other words, though it is not necessary for every man and woman to be matched to their first preference, there will not be a reciprocation of preference between a given individual and his or her more-preferred men or women (Roth 2008).

Generally speaking, the Gale-Shapley algorithm is a stable, one-to-one matching algorithm between two groups of n entities based on rank-order preferences. The algorithm accepts as input an ordinal list of preferences from each entity of both groups, which ranks each entity of the other group from most preferred to least preferred. Using an iterative process, the algorithm produces n pairs, each pair consisting of one entity from each group. Though tentative pairings are formed throughout the algorithm, they are not final until it finishes running. These resulting pairs are stable (Roth 2008).

The benefits of this algorithm are best demonstrated through contrast with market-based outcomes. The National Internship Matching Program (NIMP) separately discovered and implemented this algorithm in the 1950s to match medical school graduates with internships at hospitals. The previous, market-based system, wherein graduates apply directly to hospitals, which conduct interviews and send offers in a decentralized manner, was failing. In order to secure their top candidates over their competitors, hospitals began to send offers earlier and earlier, as early as the end of the students' second year of medical school. With this practice, the hospitals did not have a full understanding of the students' performance in the medical program, and students had to focus on securing an internship rather than their coursework. Furthermore, students would delay committing to an internship if waitlisted by a more preferred hospital. In some cases, even after committing to a less preferred internship, students would renege when accepted to their more preferred hospital. Attempts to correct these market failures also did not work; one such attempt was standardizing the time that hospitals sent offers, and requiring acceptance within mere hours. Eventually, however, the NIMP system was implemented, wherein hospitals were the "males" or proposers, and students the "females" or proposees (Wu 2020).

The purpose of this paper is to analyze the computational efficiency of the Gale-Shapley algorithm under different conditions. This is done via simulations with controlled numbers of entities and preferences. It will demonstrate the effects of correlation between the "male" or proposer preferences and the number of entities to be matched on the computations necessary to produce the stable matchings, on average. Furthermore, the code used for the match generation and simulations is versatile and can be used as a jumping off point for further experimentation.

Data and Methodology

The data used in this paper is generated via simulations, rather than sourced from real-world data sets. This allows clean manipulation of different factors, including the number of pairs and the exact correlation between lists. Doing so required three sets of functions to accomplish the following goals: generation of preference lists with a given correlation, generation of matches according to the Gale-Shapley algorithm, and generating graphs from repeated iterations of the two aforementioned function sets.

Generating Matches

The core research question of this paper depends on the ability to generate matches in accordance with the Gale-Shapley algorithm. Though invented long before computers were widely available, it is relatively easy to automate. The process is run by a single function, `generate_matches`, which takes two inputs: `malePreferences` and `femalePreferences`. Both parameters should be in the form of matrix type, with the same number of rows and columns. Each column refers to the preferences of a single individual, with the first number indicating the number of the most preferred individual in the other group. If the size of the preference matrices do not match, the function immediately returns -1 to indicate an error.

Assuming the preferences match the above conditions, the function will then create several vector (`unmarriedMales`, `maleSpouse`, `femaleSpouse`, and `nextMaleChoice`) and one numeric (`iterations`) variable. `unmarriedMales`, at the beginning, contains all males in the simulation. `nextMaleChoice` begins as 1 for all males (representing the index of the female in his preference to whom he will propose next), and `maleSpouse` and `femaleSpouse` begin as NA for all participants. Finally, `iterations` begins at 0.

With those foundation variables established, the rest of the function operates within a while loop, conditional on any males remaining in the `unmarriedMales` vector. Each run increments the `iterations` variable. Within the loop, the function selects the first male in the `unmarriedMales` vector (note: the order of selection does not matter, as the algorithm will always produce the same matches given a set of preferences), and has him propose to his most-preferred choice that he has not yet proposed to, as stored by `nextMaleChoice`. If that female has not yet received a proposal, she will tentatively accept it; this will remove the male from `unmarriedMales`, place the matches into the appropriate indices of `maleSpouse` and `femaleSpouse`, and increment `nextMaleChoice` for that male. If the female has already received a proposal, and prefers that tentative match to the new proposal, she will reject the new male's proposal; he will be left in `unmarriedMales` and his value in `nextMaleChoice` will be incremented. Finally, should the female already have a tentative match but prefer the new male, she will reject the previous tentative match. This will add the previous male back into, and remove the new male from, `unmarriedMales`, and update the `maleSpouse` and `femaleSpouse` vectors to reflect the new tentative match. The algorithm will then proceed to the next male in `unmarriedMales`.

When the loop exits (i.e. there are no more unmatched individuals), the function returns a list containing the following variables: `iterations`, `maleSpouse`, and `femaleSpouse`. Of primary interest to this paper is `iterations`, but the inclusion of the other variables allows future research, as detailed in the discussion section.

Preference Generation

In order to analyze the impact of preference correlation, preference lists must be created in a controlled manner. To this end, I wrote several functions to generate preferences. The most simple is `generate_random_preferences`, which takes an argument, `n`, to indicate how many preferences should be generated (i.e. how many males or females are in the simulation). The function then loops that many times, each loop creating two vectors randomly shuffling numbers between one and the number of preferences to be generated, and appending those separate vectors to vectors of male and female preferences. As a result, the male and female preferences are independently random from each other and independently random from other male or female preferences. Both of those vectors are then converted into matrices, `malePreferences` and `femalePreferences`, of size `n`. Throughout all simulations, this is the only preference generation function used for female preferences, while the male preference generation function varies. This is intended to isolate the effects of the changes in male preference correlation.

At one extreme of the preference spectrum, the function `generate_perfectly_different_lists`, with argument `num_lists`, generates a set of preferences for `num_lists` men which are each unique. The process for this generation is to first shuffle a set of numbers between one and the number of preferences to be generated. Then, this vector is iteratively offset to generate a unique preference vector for each male. For example, `generate_perfectly_different_lists(3)` may produce the following set of preferences: `[(1,2,3), (2,3,1), (3,1,2)]`.

The other extreme is generated with the function `generate_perfectly_correlated_lists`, with argument `num_lists`. Similar to the perfectly different function, `num_lists` represents the number of men in the simulation. The function generates one vector of preferences by shuffling all numbers between one and `num_lists`. It then repeats that vector `num_lists` times, and converts the repeated vector to a matrix with `num_lists` rows and columns, each column being identical.

In order to generate intermediary preferences, with a correlation between perfectly different and perfectly correlated, I first had to define correlation for ordinal lists. The method that I chose was the average correlation between all combinations of preferences, without comparing any preference to itself. This can be represented with the following mathematical notation:

$$mean\left(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} cor\left(i,\ j\right)\right) \tag{1}$$

With the ability to measure the correlation, I then had to create a system to generate correlated lists; as far as I am aware, there is not an existing method to do so in R. To do so, I defined a function, `generate_correlated_lists`, with two arguments, `num_lists` and `corr`. The former is the number of men in the simulation, while the latter indicates the desired level of correlation between the preferences with three supported options: 0.3, 0.5, and 0.8. These were selected to represent low, moderate, and high levels of correlation in preferences. In order to generate preferences with a low level of correlation, I created a random chi-squared distribution using the `rchisq` function in base R. I used the parameters of `n = 100, df = 30` after experimentation to find which parameters most consistently elicited a correlation of 0.3. After generating 100 random values in the distribution, I divided the range into `num_lists` equally-sized bins. I then ordered those bins by frequency of occurrences using the `order()` function. This returns the bin numbers (as its index between 1 and `num_lists`) in order of smallest to largest, which was then used as the rank order preference of the male. This was repeated for each male, generating a new set of values using the `rchisq(100, 30)` function each time.

The same process was followed to generate moderate correlation preferences. To generate moderate correlation preferences, I once again used random values from a chi-squared distribution. In order to increase the correlation from approximately 0.3 to 0.5, I generated these values using the parameters of `n = 100, df = 10`. This allowed for less freedom in the dispersion of the 100 generated values, creating more consistent patterns of frequencies across the `num_lists` bins. With these 100 values following the new parameters, I followed the same procedure of range splitting and frequency ordering to create male preferences.

Finally, to generate high correlation preferences, I used random exponential distributions in the same manner. The parameters I used for the `rexp` function were `n = 100, rate = 1`. I then applied the same procedure as in the generation low and moderate correlation, splitting into evenly-sized bins and ordering by frequency. This yielded a very tight distribution pattern, meaning that the preferences generated were very similar.

As will be discussed in the next subsection, I added additional checks to the generation of preferences to ensure that the returned lists were within a reasonable range of the desired correlation. With any random distribution, there is the possibility of outliers which do not conform to expected results.

Fixed-Correlation Tests

The first set of tests that I ran were to examine how the number of matches impacts the number of iterations, given a fixed correlation. For all simulations, the number of matches ranged from 1 to 50. This was, in part, based on limitations of computing power; in order to find a meaningful average with minimal influence of outliers, each number of pairs for each level of correlation was run 100 times. This means that there were 5,000 simulated match generations per level of correlation. Every such match requires hundreds to thousands of computations to complete the generation of the correlated preferences and the solving of the matches in accordance with the Gale-Shapley matching algorithm. Given those practical limitations, I settled on a range of 1 to 50 pairs.

The first test function was `test_num_iterations_random`, which takes an argument `n` to indicate the maximum number of pairs to solve (50) It creates vector with value 0 repeated `n` times. This function begins at 1 pair, and calls the `generate_random_preferences(1)` to generate the preferences. It then runs the preferences through `generate_matches()`, and adds the number of iterations taken to solve the match to the first position of the aforementioned vector. This occurs 100 times before incrementing to 2 pairs, repeating the process through 50 pairs. Once this is complete, it divides the entire interactions vector by 100 to find the average number of iterations to solve each number of pairs.

The remaining test functions, `test_num_iterations_correlated_male`, `test_perfectly_correlated_lists`, `test_perfectly_different_lists`, operate in the same manner. Each function takes an input for the maximum number of pairs to solve, `n`, which was 50 for the purposes of this paper. Additionally, `test_num_iterations_correlated_male` required a second parameter, `corr`, which was passed through to `generate_correlated_lists`. In order to keep the correlation near the desired value, this function also rejects any set of preferences which falls more than 0.05 above or below the specified correlation. This condition is ignored when comparing preferences of 10 or fewer males because I found reasonable estimates of correlation with low numbers of participants to be difficult to generate. Each of these tests totaled the number of iterations required for 100 trials of each number of pairs (between 1 and 50), then calculated the average.

Fixed-Pair Tests

The fixed-pair tests took a different approach. In order to demonstrate the effect of changes in correlation on the number of iterations required for a solution, the number of pairs are held constant while correlation is allowed to vary. I selected three numbers of pairs: 10, 25, and 50. These tests depend on two functions, `generate_correlated_lists_unfixed` and `test_iterations_vs_correlation_fixed`.

The former takes heavily from `generate_correlated_lists`. However, instead of requiring an input of a desired correlation, it only accepts the parameter `num_lists`, representing the number of males for which it should generate preferences. It then generates three parameters, `a`, `b`, and `c`, and chooses a distribution type from amongst exponential, F-distribution, and chi-squared. It then creates a random distribution of the selected type with the following parameters: `rexp(n = a, rate = b)`, `rf(n = a, df1 = b, df2 = c)`, or `rchisq(n = a, df = b)`. With the values from this distribution, it follows the procedure outlined in the correlated male preferences section: divide the range into `num_lists` even bins, sort by frequency, and use this order as the male preferences. This was repeated with the same type of distribution and parameters, but new calls to that selected random distribution function, for `num_lists` total males.

The second function, `test_iterations_vs_correlation_fixed`, loops the first function 2,000 times. With each loop, it matches the generated male preferences with randomly generated female preferences, records the number of iterations taken to solve, and calculates the correlation of male preferences. The correlation is recorded in one vector (x), while the number of iterations is recorded in another (y). Once those loops have completed, the function excludes outliers of the y variable, and plots the remaining pairs of correlation and iterations. The function accepts one parameter, `n`, to indicate how many pairs of preferences should be generated and matched.

Results

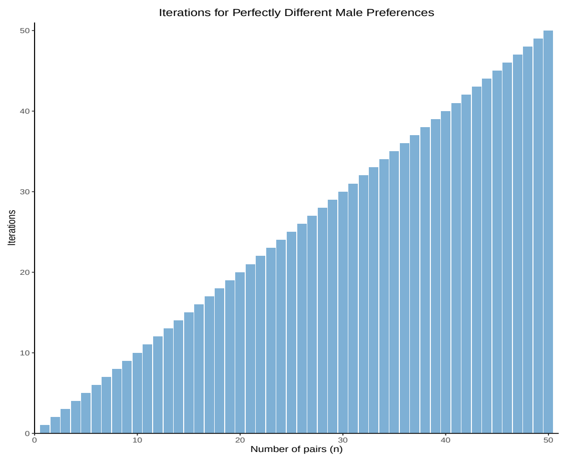

Figure A. Perfectly Different Preferences

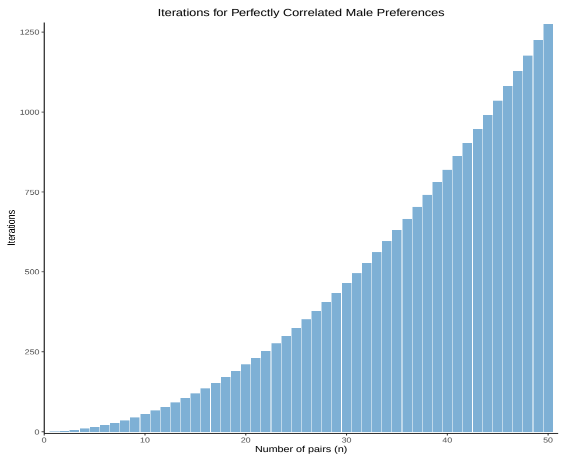

Figure B. Perfectly Correlated Preferences

The results from the fixed-correlation tests make intuitive sense: as the number of pairs increases, so does the number of iterations required to solve the matches. For every level of correlation, generating a single pair took one iteration. Beyond one pair, the different levels of correlation diverged. The perfectly different preferences (Figure A) yielded a linear increase in the number of iterations required for solution, which can be expressed as $f(x) = x$, where $f(x)$ is the number of iterations required for solution and x is the number of pairs. Thus, to create 50 pairs, it required 50 iterations. On the opposite end of the extremes, the perfectly correlated preferences (Figure B) required 1275 iterations to solve 50 pairs. The number of iterations required can be expressed as $f(x) = \frac{x^2+x}{2}$. Of note, for these two distributions of preferences, a given number of pairs will always require the same number of iterations to solve; there is no variation. Both graphs are continuously increasing as the number of pairs increase.

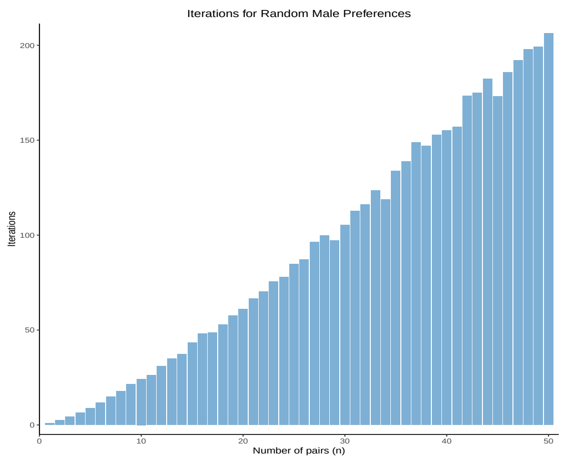

Figure C. Random Preferences

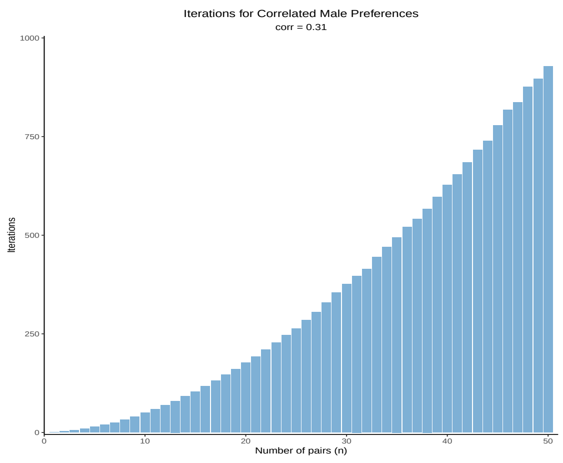

Figure D. Correlated Preferences, corr = 0.31

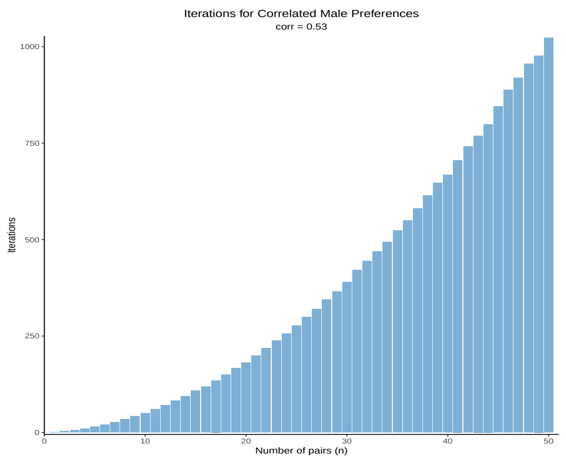

Figure E. Correlated Preferences, corr = 0.53

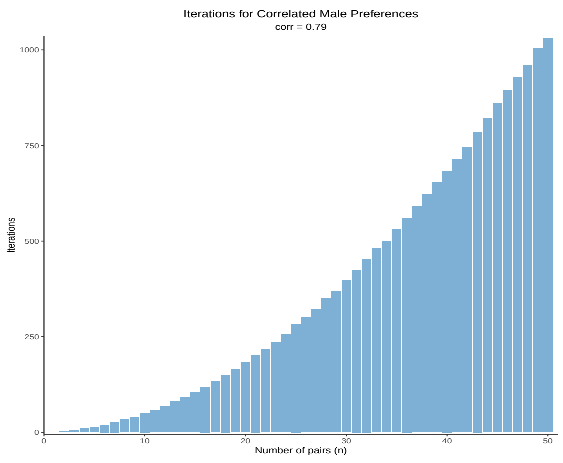

Figure F. Correlated Preferences, corr = 0.79

Variation was present in the other four distributions of preferences. The most noisy distribution was random male preferences (Figure C), which required an average of 206.43 iterations to solve 50 pairs. As the number of pairs increases, the number of iterations at first has a curved shape but becomes more linear. As it becomes more linear, it also becomes more noisy, with some some higher number of pairs requiring fewer iterations than lower numbers of pairs. This is likely due to the range of possibilities that random male preferences allows for: a random set of preferences could be perfectly different, perfectly correlated, or anything in between. As such, there can be a large variation in how many iterations it takes to solve a given number of pairs of random preferences. By conducting 100 trials of each number of pairs and averaging the result, I controlled for some of the variation, which is why the graph does have a generally-consistent shape.

The target values for low (Figure D), medium (Figure E), and high (Figure F) preference correlation were 0.3, 0.5, and 0.8, respectively. Using the previously discussed methods of preference generation and correlation measurement, the actual values were 0.31, 0.53, and 0.79, respectively. In each case, as the number of pairs increased, so too did the average number of iterations required to solve the matches. Unlike the random male preferences, there was little to no noise in these values. There was never a case where a greater number of pairs required, on average, fewer iterations than an instance of fewer pairs; in other words, all three plots are continuously increasing. The low, medium, and high preference correlations required 928.18, 1022.89, and 1031.00 iterations to solve 50 pairs, respectively. All three plots have a curved shape, with the high correlation having the most steep curve.

These findings indicate two conclusions. First, as the number of pairs increases, given a fixed male preference distribution, the number of iterations increases on average. This is shown by the continuously increasing shape of five of the distributions, with the random preference distribution containing noise. Second, the total number of iterations required, and the growth rate of the number of iterations, varies with the distribution of male preferences. This is demonstrated in the shape of the plots, with a linear shape for perfectly different preferences and a curved shape for perfectly correlated preferences.

For the data set used in the fixed-correlation findings, see Appendix A.

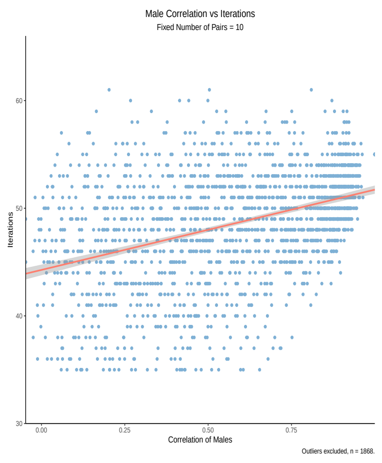

Figure G. 10 Pairs

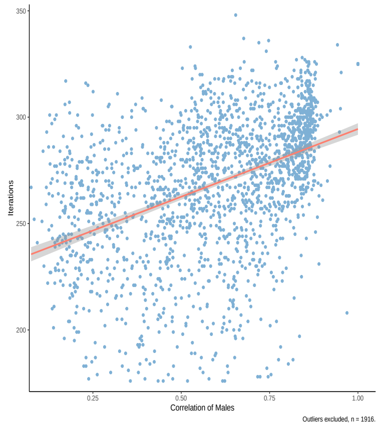

Figure H. 25 Pairs

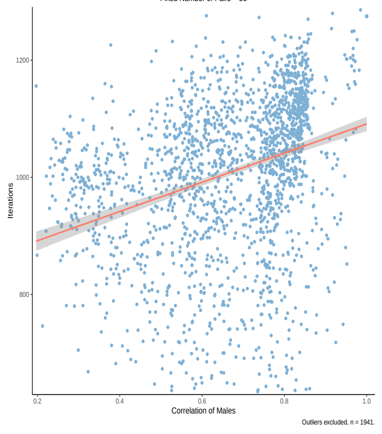

Figure I. 50 Pairs

The results from the fixed-pair tests strongly indicate that the correlation of a set of male preferences has a positive effect on the number of iterations required to solve the matches. For all three fixed-pair tests, with 10 (Figure G), 25 (Figure H), and 50 (Figure I) pairs, the trend line indicates that as the correlation of male preferences increased, so did the number of iterations, on average. There was significant noise across all three graphs, primarily with values far below the average number of iterations for a given level of correlation. True outliers were excluded from these graphs, reducing the number of points from 2,000 to 1,868, 1,916, and 1,941 for the plots of 10, 25, and 50 pairs, respectively. This could indicate that, as the number of pairs increases, there is less variation in the number of iterations required regardless of the correlation; however, to definitively prove that conclusion, further research would be necessary. It should also be noted that, in the

50 pair plot, no point fell below 0.2 correlation. This may represent an issue with the method of measurement for correlation that I selected or simply a quality of large lists.

Conclusion

This project has completed several objectives: automating the Gale-Shapley algorithm in R; generation and measurement of various distributions of correlated and uncorrelated preferences; and conducting both fixed-correlation and fixed-pair simulations to determine the effects on computational efficiency, as measured by iterations required to solve matches. The fixed-correlation simulations yielded two primary conclusions:

1. As the number of pairs increases, given a fixed male preference distribution, the number of iterations increases on average.
2. The total number of iterations required, and the growth rate of the number of iterations, varies with the distribution and correlation of male preferences.

The fixed-pair simulations yielded one conclusion:

1. The correlation of a set of male preferences has a positive effect on the number of iterations required to solve the matches.

The combination of these three findings may indicate problems of inefficiency when using the Gale-Shapley algorithm for large numbers of pairings with highly correlated preferences. Other, perhaps non-stable, matching algorithms may be better suited for such situations at the present time.

Beyond efficiency, the code produced in this project can be adapted to study other properties of the Gale-Shapley algorithm. For example, welfare analysis based on the resulting pairs can easily be computed from the output of the matches. Another example is the effect of the correlation of female preferences, or intercorrelation of male and female preferences. All three topics would require minimal adjustments to the existing code.

Appendix A: Data Set

Number of Pairs	Perfectly Different	Perfectly Correlated	Random Preferences	Corr = 0.31	Corr = 0.53	Corr = 0.79
1	1.00	1.00	1.00	1.00	1.00	1.00
2	2.00	3.00	2.48	2.94	3.00	3.00
3	3.00	6.00	4.37	5.78	5.80	5.99
4	4.00	10.00	6.57	9.51	9.72	9.86
5	5.00	15.00	8.89	14.21	14.54	14.37
6	6.00	21.00	11.82	19.46	20.09	19.65
7	7.00	28.00	14.91	25.16	26.38	26.21
8	8.00	36.00	17.70	32.58	33.95	33.01
9	9.00	45.00	21.45	40.34	42.10	40.01
10	10.00	55.00	24.26	49.84	50.52	49.73
11	11.00	66.00	26.24	58.70	60.11	57.82
12	12.00	78.00	30.93	69.19	70.22	68.54
13	13.00	91.00	34.84	79.94	82.26	80.77
14	14.00	105.00	37.39	91.99	93.75	92.30
15	15.00	120.00	43.29	103.38	108.96	105.38
16	16.00	136.00	48.24	117.08	119.31	117.86
17	17.00	153.00	48.68	131.09	135.00	132.76
18	18.00	171.00	52.83	146.56	150.32	150.37
19	19.00	190.00	57.70	160.18	167.19	165.24
20	20.00	210.00	60.93	176.79	180.67	182.63
21	21.00	231.00	66.69	192.79	199.35	201.52
22	22.00	253.00	70.23	210.31	218.49	217.73
23	23.00	276.00	75.67	227.66	237.59	235.00
24	24.00	300.00	78.02	247.02	256.38	257.06
25	25.00	325.00	84.65	263.00	276.89	282.51
26	26.00	351.00	87.20	284.47	299.35	300.96
27	27.00	378.00	96.33	304.58	319.97	322.85
28	28.00	406.00	99.82	329.31	344.66	351.31
29	29.00	435.00	97.28	355.04	365.73	368.19
30	30.00	465.00	105.40	376.65	390.27	397.86
31	31.00	496.00	112.66	397.12	420.62	423.76
32	32.00	528.00	116.18	414.45	443.91	452.51
33	33.00	561.00	123.48	444.54	468.55	481.03
34	34.00	595.00	118.87	469.95	494.04	500.13
35	35.00	630.00	133.72	494.79	524.02	529.87
36	36.00	666.00	138.75	520.84	549.52	561.03
37	37.00	703.00	148.78	541.51	580.31	592.17
38	38.00	741.00	147.01	567.10	614.08	622.05
39	39.00	780.00	152.81	596.55	647.36	652.87
40	40.00	820.00	155.25	627.67	667.85	683.91
41	41.00	861.00	157.08	654.02	705.86	714.36
42	42.00	903.00	173.45	684.01	741.82	745.58
43	43.00	946.00	174.95	716.62	769.45	784.68
44	44.00	990.00	182.24	739.61	799.28	820.97
45	45.00	1035.00	173.03	778.52	845.67	860.68
46	46.00	1081.00	185.82	817.76	888.50	894.84
47	47.00	1128.00	192.06	837.16	919.70	928.29
48	48.00	1176.00	197.89	876.40	955.09	959.52
49	49.00	1225.00	199.14	896.12	977.03	1004.13
50	50.00	1275.00	206.43	928.18	1022.89	1031.00

Bibliography

Roth, Alvin E. 2008. "Deferred acceptance algorithms: history, theory, practice, and open questions" [in en]. *International Journal of Game Theory* 36, no. 3 (March): 537–569. Accessed December 7, 2022. https://doi.org/10.1007/s00182-008-0117-6.

Wu, Yue. 2020. "Game Theoretic Consequences of Resident Matching," accessed December 7, 2022. https://doi.org/10.48550/ARXIV.2003.07205.