# Executive Summary

## Gale-Shapley Algorithm

Artie Humphreys, Ridge Redding, Arthur Wu, Ayaan Rahman

April 24, 2025

## 1 Introduction

Modern marketplaces, ranging from the college admissions process to online dating platforms, rely on matching algorithms that pair two equally sized sets of agents, each with ranked preferences over the other side. Beyond matching two sets of agents together, a core requirement that should be considered when matching these two sets together should be stability. This means that there are no two agents that prefer each other over their assigned partners. If stability wasn't met in these matches, it could lead to dissatisfaction, churn, or in more extreme situations, market failure. This is where the Gale-Shapley algorithm comes in.

While other alternative approaches exist for matching algorithms, they have their own respective drawbacks. For instance, the brute-force enumeration of all $n!$ matchings is exponential, becoming infeasible beyond $n \approx 10$. The greedy approach for pairing ("best available") may terminate quickly, but it can leave blocking pairs, resulting in outcomes unraveling. Gale-Shapley's algorithm successfully circumvents the aforementioned drawbacks with other matching algorithms and produces a stable and optimal matching for the proposing side in $O(n^2)$ time. This can be done using simple data structures such as arrays and hashmaps. In their 1962 paper, Gale and Shapley proved that, for any two equally-sized sets with strict preference orderings, a stable matching will always exist, meaning that the procedure will terminate in a finite number of steps.

## 2 Approach

The approach of the Gale-Shapley algorithm can be simplified into three steps: a proposal, holding, and potentially "trading up" if a better preference is found. There are two equally sized sets of proposers and acceptors, and each agent ranks all members of the opposite set based on preference. For the sake of simplicity, we will refer to these sets as Set $A$ (proposers) and Set $B$ (acceptors). To begin, all of the proposers begin as free and available, allowing them to be matched with an acceptor that they will propose to. Then, iterate while there exists a proposer, let's call it $p$, who is free and still has acceptors to approach. Within this iteration, the free proposer $p$ proposes to the most-preferred acceptor $a$ on $p$'s list that hasn't already taken another proposer over them.

Acceptor $a$ then has several choices: if $a$ is free, they provisionally accept $p$. If $a$ is engaged to $p'$ but prefers $a$, they will trade up and cut off their matching with $p'$, who becomes free again. Otherwise, $a$ rejects $p$, who will later propose to the next-highest acceptor in their preference list.

The algorithm terminates when no proposer is free. All provisional engagements become final, forming the matching.

## 3 Implementation

To implement the Gale–Shapley algorithm, first begin by reading the size $n$ and the two $n \times n$ preference tables (proposer and acceptor), then construct a rank-lookup matrix `acceptorRank`. For every acceptor $a$ and proposer $p$, the entry `acceptorRank[`$a$`][`$p$`]` stores the position of $p$ in $a$'s preference list. This preprocessing step, taking $O(n^2)$ time, where $n$ is the number of agents in both sets, ensures that future comparisons of acceptors' preferences are done in constant time.

After the preferences are fixed, four arrays capture the evolving state of the matching. Boolean arrays `proposerFree` and `acceptorFree` mark which agents are still unattached; `matchForProposer[`$p$`]` records the acceptor currently paired with proposer $p$; the symmetric array `matchForAcceptor[`$a$`]` records the proposer holding acceptor $a$. All four start in the obvious way (everyone is free, and every match slot is set to $-1$).

The main loop will continually call the helper method `findFreeProposer`, which returns the index of any free proposer or $-1$ when none remain, in which case we terminate. For the chosen proposer $p$, the algorithm walks down $p$'s preference list from the top. If the first candidate acceptor $a$ is still free, the pair `(`$p$`,`$a$`)` becomes engaged and both `proposerFree[`$p$`]` and `acceptorFree[`$a$`]` change to `false`. If $a$ already holds another proposer $p_2$, the rank matrix tells us instantly whether the acceptor prefers the new proposal. If `acceptorRank[`$a$`][`$p$`]` is smaller than `acceptorRank[`$a$`][`$p_2$`]`, meaning the acceptor values $p$ higher than $p_2$, the acceptor "trades up," which makes $p_2$ free again and updates the match arrays by setting `matchForProposer[`$p_2$`]` $= -1$ and `matchForAcceptor[`$a$`]` $= p$. If $a$ rejects, $p$ simply moves on to the next acceptor on his list in the same iteration. Because $p$ advances through their list of preference exactly once, each ordered pair (`proposer`, `acceptor`) is examined at most once, resulting in a worst-case time complexity of $O(n^2)$.

To help visualize this, imagine three proposers (`p_0`, `p_1`, `p_2`) and three acceptors (`a_0`, `a_1`, `a_2`). Each proposer ranks the acceptors in the order `a_0` $\rightarrow$ `a_1` $\rightarrow$ `a_2` while the acceptors' lists differ. For instance, `a_0` prefers `p_1` $\rightarrow$ `p_2` $\rightarrow$ `p_0`, `a_1` prefers `p_2` $\rightarrow$ `p_0` $\rightarrow$ `p_1`, and `a_2` prefers `p_0` $\rightarrow$ `p_1` $\rightarrow$ `p_2`. At the start of the algorithm, the boolean array `proposerFree` is [`true`, `true`, `true`], so the helper `findFreeProposer` returns index 0. Proposer `p_0` consults its list and proposes to `a_0`, who is free; both `proposerFree[0]` and `acceptorFree[0]` change to `false` and the match tables record (`p_0`, `a_0`). The algorithm then scans `proposerFree` again, finds the next free proposer index 1, and processes that. Proposer `p_1` also proposes to `a_0`

and, since a_0 prefers p_1 over its current match, trades up, so the pairing becomes (p_1, a_0), freeing p_0. Scanning again finds p_2 free; it proposes to a_0 (rejected) then to a_1 (accepted). Next, p_0 proposes to a_1; since a_1 prefers p_0, it trades up to (p_0, a_1), freeing p_2. Finally, p_2 skips past a_0 and a_1 and is accepted by its third choice, a_2. No `true` entries remain in `proposerFree`, so the loop terminates with a stable, proposer-optimal matching.

### 3.1 Psuedocode

The pseudocode for the algorithm can be found below:

```
Initialize the 2 input sets to a "X" set and a "Y" set

While there exists an unmatched member of X, denoted as x:

    For each element of Y, denoted as y, in the order of x's preferences:

        Move beyond y in x's preference list, for future use.

        If y is free:

            Match y and x, mark y and x as matched

            Exit the for-loop

        Else:

            If y prefers x to y's existing partner p:

                Un-match y and p, and mark p as unmatched.

                Match y and x and mark x as matched.

                Exit the for-loop.
```

## 4 Summary of Programming Challenge

One of the real-world applications of the Gale-Shapley algorithm is in residency matching programs. However, every year, thousands of graduating students don't get matched to a specific program. Partially, this is because there are less open spots than applicants. However, another reason is that not every hospital and resident would be willing to match with each other, unlike in the standard Gale-Shapley implementation. A specific applicant may not be willing to relocate to a far program, and a specific program may not accept an applicant with limited experience in their field. Additionally, real-life examples have the added complexity of a one-to-many or proposer relationship. A hospital can host many residents, and the amount each hospital accepts varies wildly.

This example and these two differences served as inspirations for our programming challenge and twist. In our challenge, you play the part of an admissions officer who is evaluating the class of tentatively admitted students. Your goal is to assign each student a lab such that no pair of lab and student prefers each other over their current match. We hope that pairing two groups like this should be an obvious use case of the Gale-Shapley algorithm. However, we impose two twists to this problem. First, each lab can accept up to n students, which may not be the same for every lab. Second, every lab may not be willing to admit every student, and every student may not be willing to join every lab. If you find that a stable matching does not exist because of this, you must report the labs and their number of unfilled slots so that the university can look for students in that field.

In regards to our learning objectives for the challenge, one thing we wanted to stress was how and why the algorithm terminates and in quadratic time. It can be hard to visualize this idea through common pseudo-code or implementations, because they often employ an outer while loop which runs until all proposers are taken, or not "free." It may be easy to misinterpret this as a non-polynomial runtime or a possible infinite loop. However, the key insight is that every proposer only needs to propose at most n times before finding a stable match. This is because a stable matching is proven to exist regardless of preferences, and if a proposer is rejected once there is no use proposing again since the acceptors match at the second proposal cannot be worse than before. Thus, if we propose at most n times for every proposer, it follows that the runtime is $O(n^2)$.

We designed our second twist to help students better understand this by breaking the paradigm that there always exists a stable matching, while still requiring students to find the stable matching for the remaining labs. This forces students to "catch" the case where proposers and acceptors go unmatched, helping them understand why this case does not exist in the simple algorithm. In the original version, there will always be one unmatched acceptor willing to accept our proposal. In this updated version, this is not the case as they may not want to join us, or we may not want to accept them.. We also wanted to stress the idea that the stable matching is optimal for the proposers and suboptimal for the acceptors. This idea was implemented by forcing them to return the optimal matching for labs, not students. Lastly, we wanted students to better understand how this algorithm could be applied more in the real world, like in hospital matching, where you don't necessarily have a one to one mapping. Thus, we imposed the condition that labs can host multiple students, forcing students to rethink what it means for a proposer to be "free" and augment the algorithm beyond its direct pseudocode.

## 5 Key Ideas for Solving Programming Challenge

To successfully address the fact that some students or labs may not accept a pairing, students must understand the way that the algorithm iterates through potential pairings. Since the algorithm iterates through every proposer's student preferences, we don't have to account for students the labs refuse to take, as these students will not be present in the preference list and thus never be proposed too. However, we do have to account for the labs a student would refuse

to join. A naive way of doing this would be to check if the student's preference list contains this lab by iterating through the list. Yet, this introduces an additional nested for loop and increases time complexity to $O(n^3)$. Students should instead realize that we can facilitate membership checks in constant time by adding valid student lab pairings to a set or matrix in a preprocessing step. We can then quickly check whether this lab pairing is allowed by the student before accepting or issuing a proposal, still allowing for $O(n^2)$ time. This strategy should take thought but not be too hard to decipher, as using hashing or arrays to store membership is stressed in many algorithms and classes.

Additionally, this twist brings up the problem of breaking out of the while loop if certain labs are never matched. We know that after a lab has proposed to all students without an acceptance that it must be unpaired in the final solution. However, we should not return early if we find this case since we require that the algorithm find the labs with unfilled spots. Instead, students should notice that we can artificially set "free" to false, allowing the loop to eventually break once no free labs are left. This should not affect the results of any other lab, as we know that no student would be paired to it. To ensure that this information is still recorded, students should update a list or other data structure mapping labs to their number of unpaired slots. While we think this is slightly more difficult than before, it should not be difficult to determine if you understand the structure of the original Gale-Shapley algorithm.

Ensuring that results are most beneficial to the lab rankings is trivial given a cursory understanding of Gale-Shapley's results. In any case, the output matching is optimal with respect to the proposers, meaning students must use the labs as proposers in their implementation. This was mainly added, alongside the requirements that outputs be ordered in terms of lab preferences, to impose structure to the results. Since many stable matchings are usually possible, this would otherwise introduce issues where two implementations could return different but equally correct results.

To employ a unique amount of students, you can treat each spot in the lab as its own proposer, reducing the problem back into one-to-one mappings. The updated algorithm now iterates through the spots of each lab, marking a lab as not "free" when there are no spots available. This introduces possible issues with preferences, making it more than an extremely quick fix, as if strict comparisons are not used to break off previous pairings, we introduce instances where students can circularly iterate through spots in the same lab. In our solution, we imposed the additional condition that, if a lab spot proposes to a student in the same lab, the student will only accept if the index of this spot is lower than the current. This makes better students bubble up to earlier indices, removing the need to sort these students in post-processing. We also must now update which spots are open in the lab, and monitor whether a lab is "free" based on an integer of open spots, which is incremented or decremented after successful proposals. While treating each spot as its individual proposers is easier to see, the many small changes that are needed as a result can be slightly tricky.

Overall, the time complexity of the augmented algorithm still remains $O(n^2)$ if we treat n

as the overall number of students. Added preprocessing steps are all $O(n^2)$, and thus do not drive up asymptotic runtime. Although we add an additional nested for loop to iterate over open spots within the lab, this is equivalent to iterating n times once, as the total number of spots across all labs is equal to $n$. Our space complexity is similarly also $O(n^2)$. We additionally store a mapping from each of the n students to the at most l labs it refuses to participate in. Since the number of labs is always less than or equal to the number of students, this is $O(n^2)$ space, and it does not increase the overall complexity.

# 6 Conclusion

The Gale-Shapley algorithm demonstrates a solution that is guaranteed to provide stable matchings, which is especially useful in countless real-world allocation problems. First, the idea of stability was reinforced (pairings with no incentive to unravel), and this showed why brute-force or greedy heuristics could not meet the goal. The "Approach" section then distilled the algorithm into three intuitive actions: propose, hold, and (when it makes sense) trade up. The "Implementation" section translated those actions into constant-time array operations, proving that every proposer-acceptor pair is examined once and establishing the canonical $O(n^2)$ bound. A step-by-step three-agent scenario illustrated each state change, while the programming-challenge extension demonstrated how the same ideas stretch to capacity constraints and incomplete preference lists without sacrificing correctness or asymptotic efficiency.

Taken together, these pieces show a seamless arc: a practical problem leads to a principled algorithm; the algorithm maps cleanly onto basic data structures; and minor, well-reasoned modifications adapt it to messier domains such as residency placement or many-to-one lab assignments. Because every proposer advances down a finite preference list and each acceptor only improves or keeps an existing partner, the process must terminate, and it always terminates with a matching that is both stable and optimal for the proposing side. That guarantee, and the clarity with which we can explain and implement it, is precisely why Gale-Shapley remains a cornerstone of modern matching theory.