Lecture 2

The stable matching algorithm

Chinmay Nirkhe | CSE 421 Spring 2025

Previously in CSE 421…

The propose and reject algorithm

Gale & Shapley 1962

The group P proposes and the group R receives

```
Initialize each person to be free.
while (some p in P is free) {
    Choose some free p in P
    r = 1st person on p's preference list to whom p has not yet proposed
    if (r is free)
        tentatively match (p,r)    //p and r both engaged, no longer free
    else if (r prefers p to current tentative match p′)
        replace (p′,r) by (p,r)    //p now engaged, p′ now free
    else
        r rejects p
}
```

The propose and reject algorithm

What have we learned?

- Proof of termination in n^2 iterations. ✓
- Proof of perfection: everyone gets matched. ✓
- Proof of stability: the output matching is stable for all pairs. ✓
- What have we not talked about?
  - Is it fair? Is it better to be a proposer or a receiver? Does the first proposer or the last proposer have it better?
  - Is there a faster algorithm?
  - How do we extend to n proposers and n' receivers?

Today

Gale-Shapley walkthrough

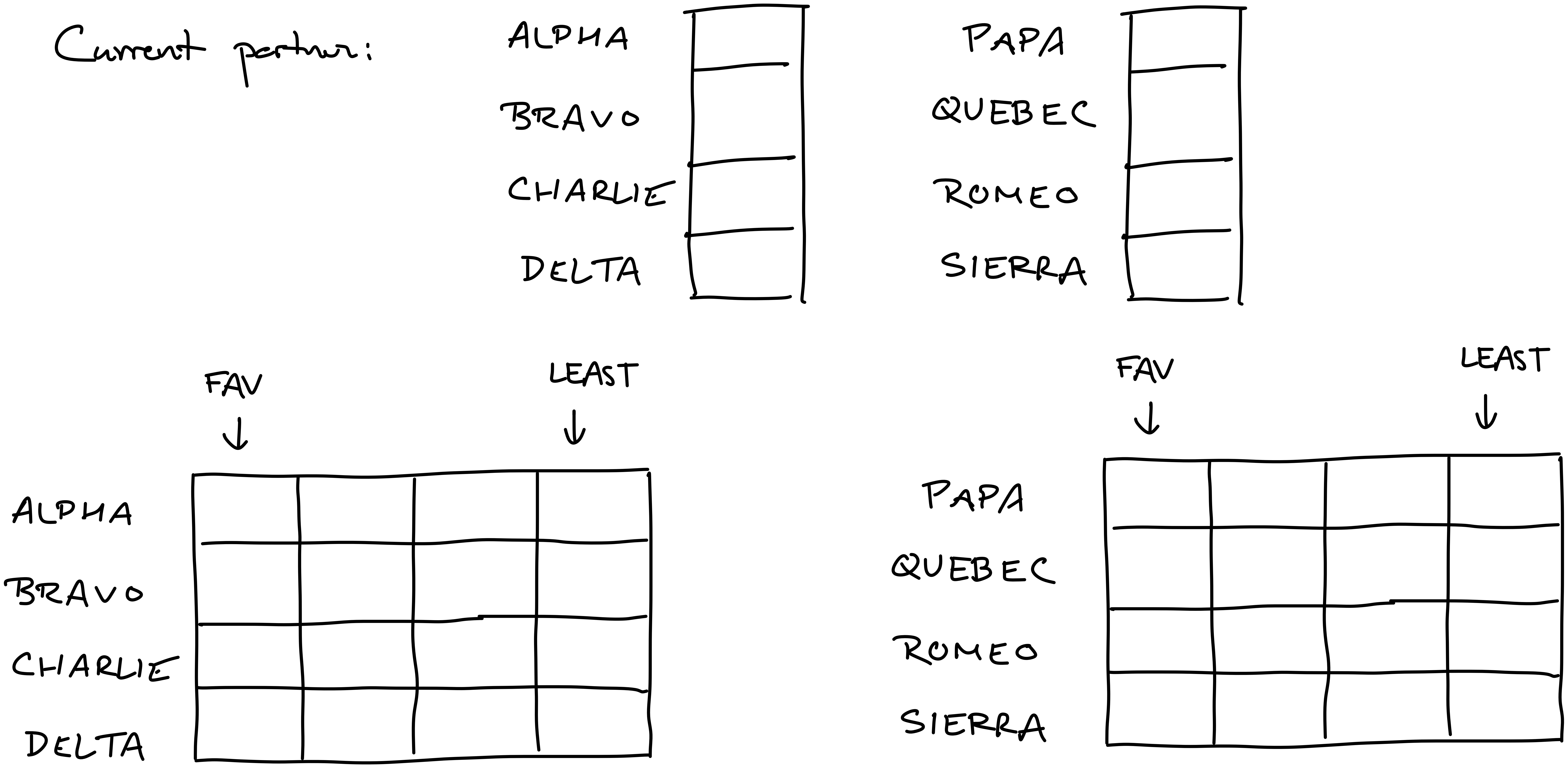
$n = 4$.

We will walkthrough alg, staying blind to the remainder of the input until we have queried it.

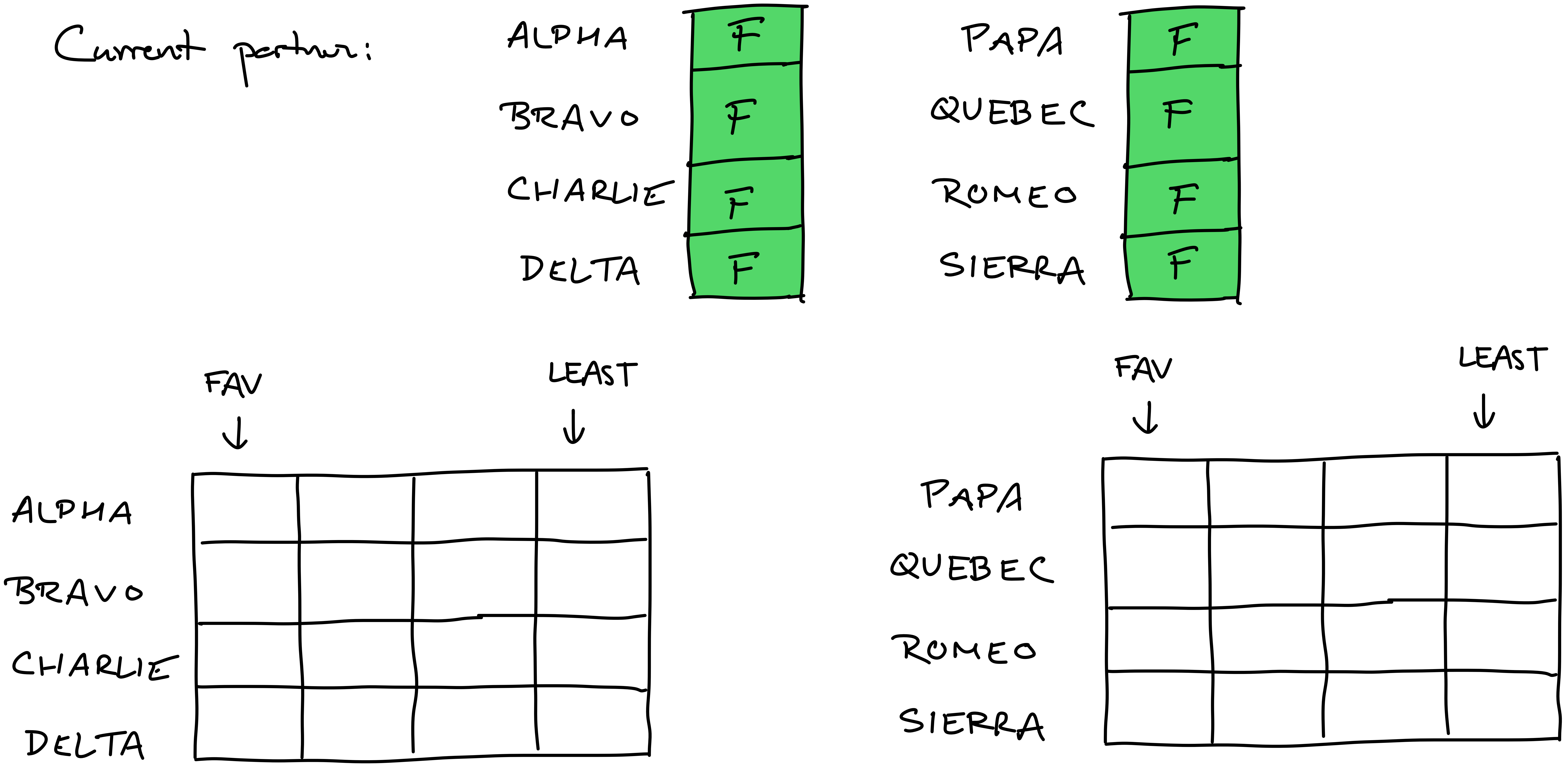
	FAV ↓			LEAST ↓
ALPHA				
BRAVO				
CHARLIE				
DELTA				

	FAV ↓			LEAST ↓
PAPA				
QUEBEC				
ROMEO				
SIERRA				

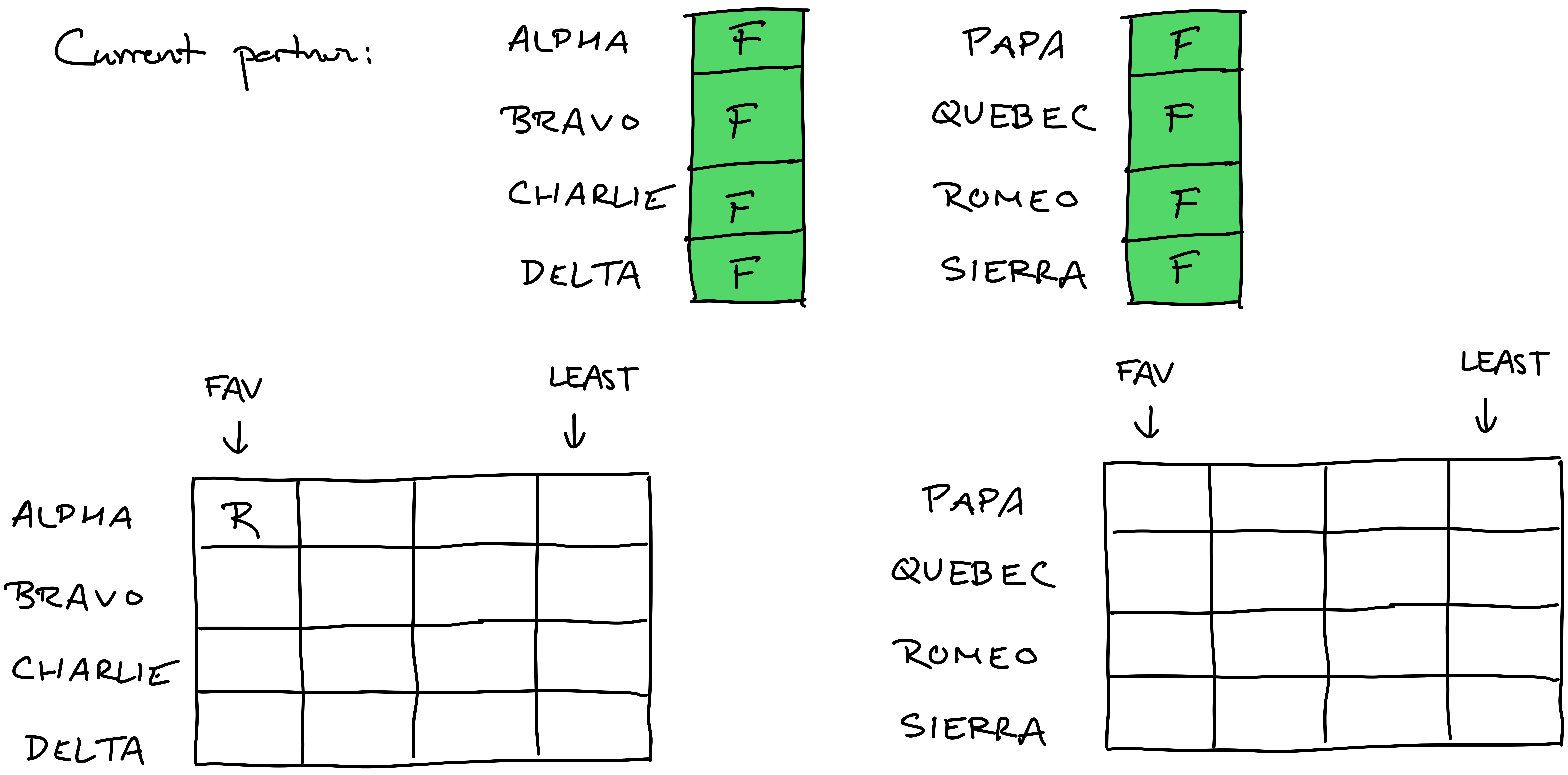
Gale-Shapley walkthrough



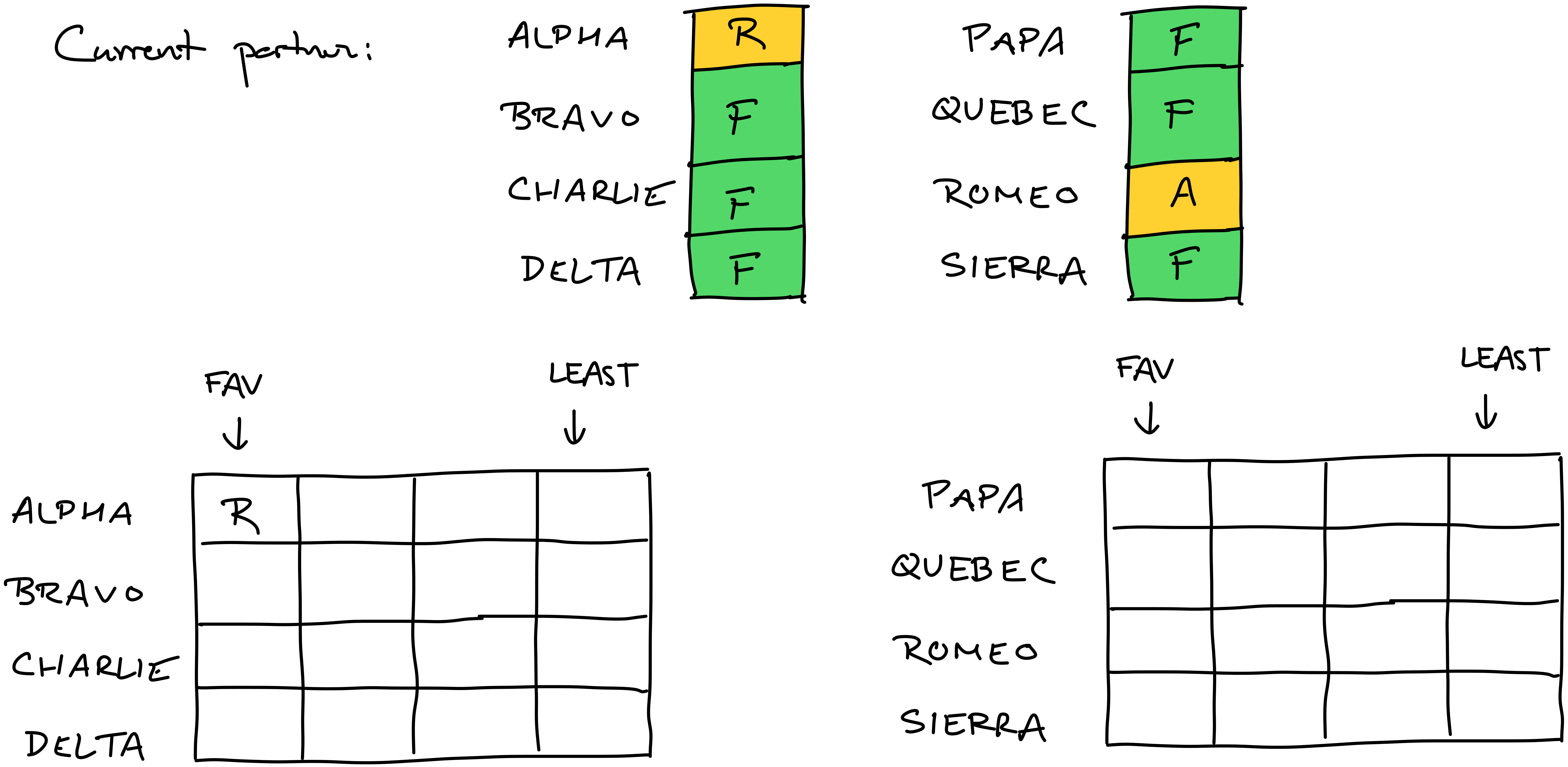
Gale-Shapley walkthrough



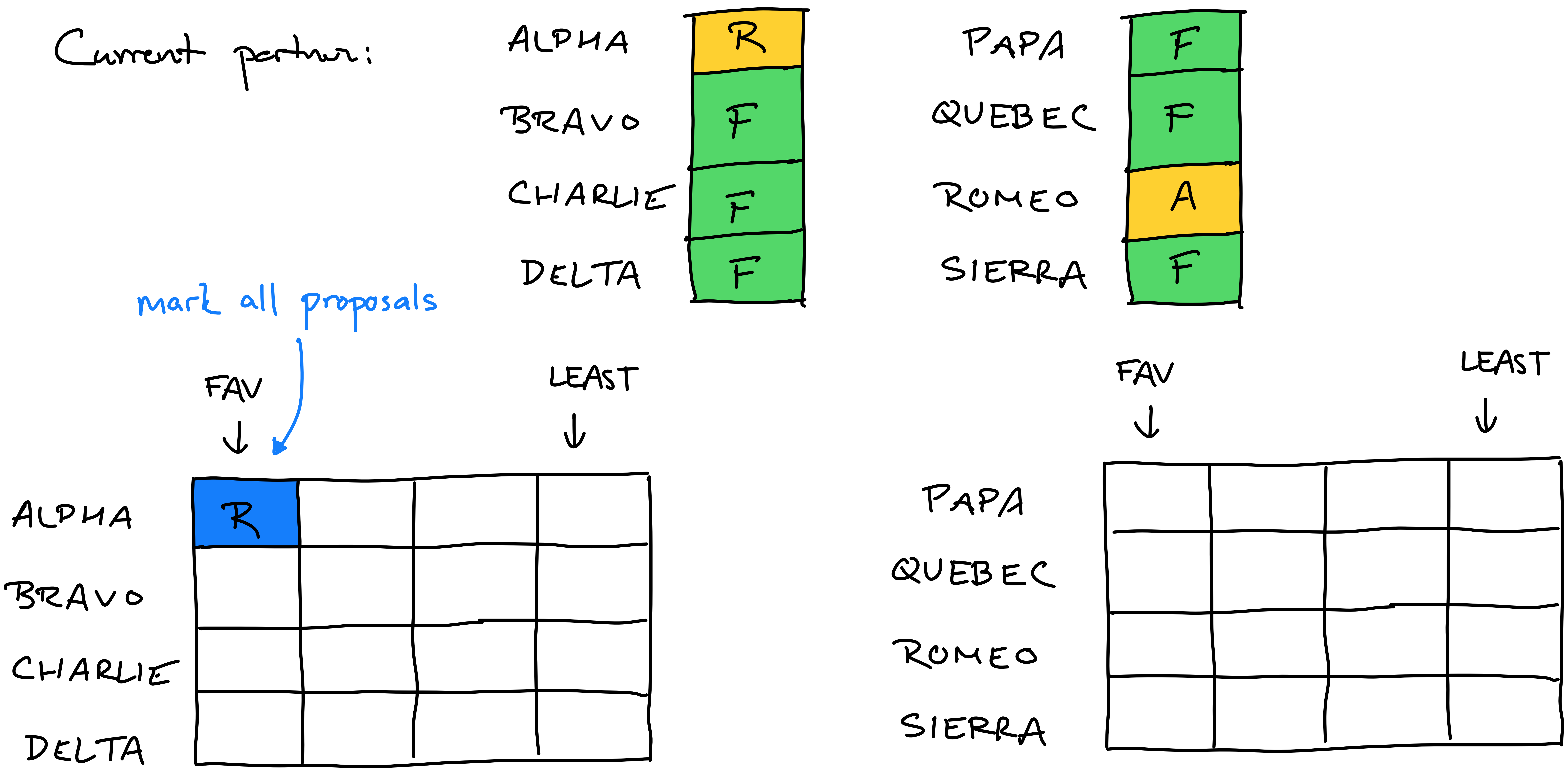
Gale-Shapley walkthrough



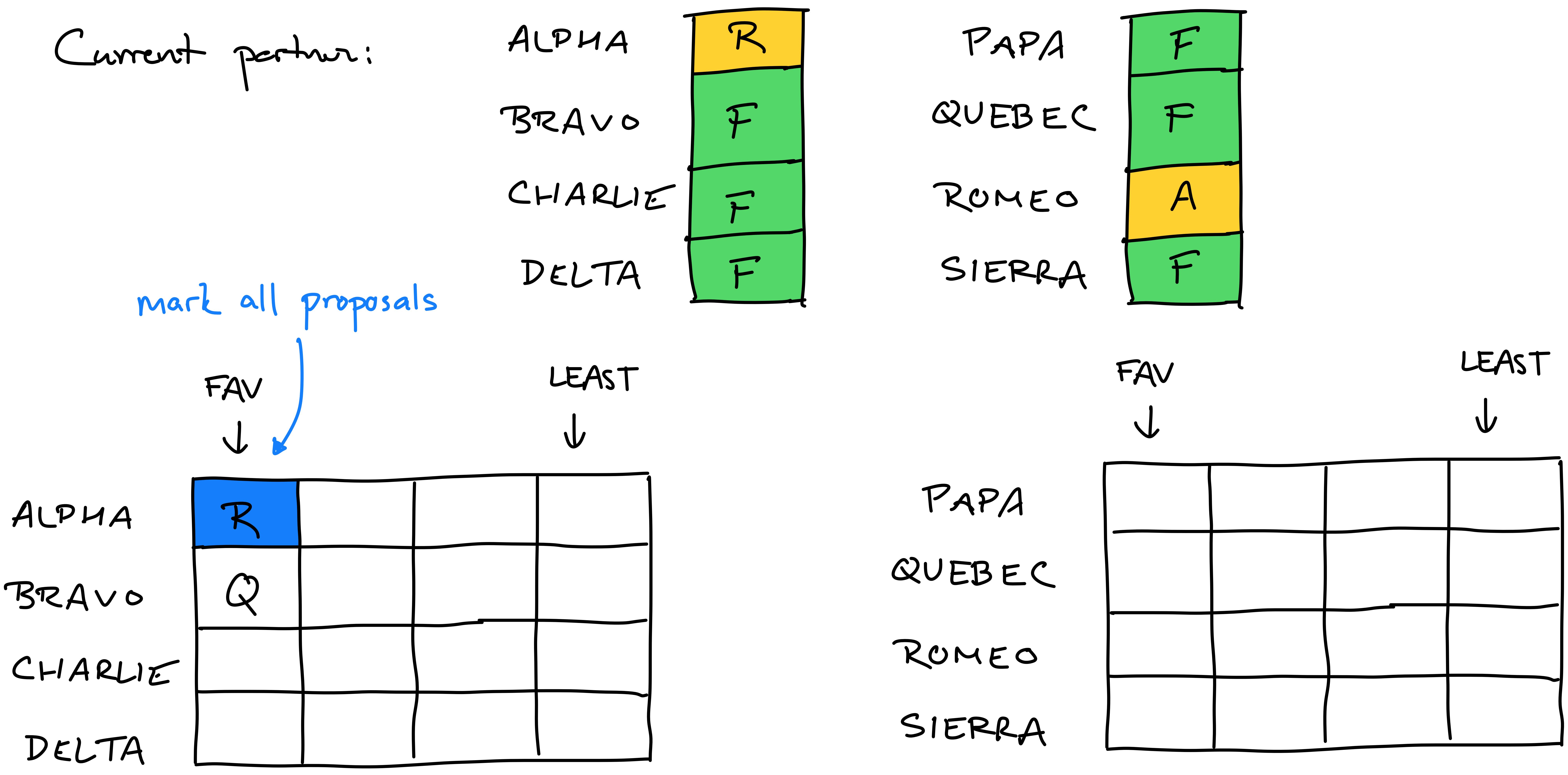
Gale-Shapley walkthrough



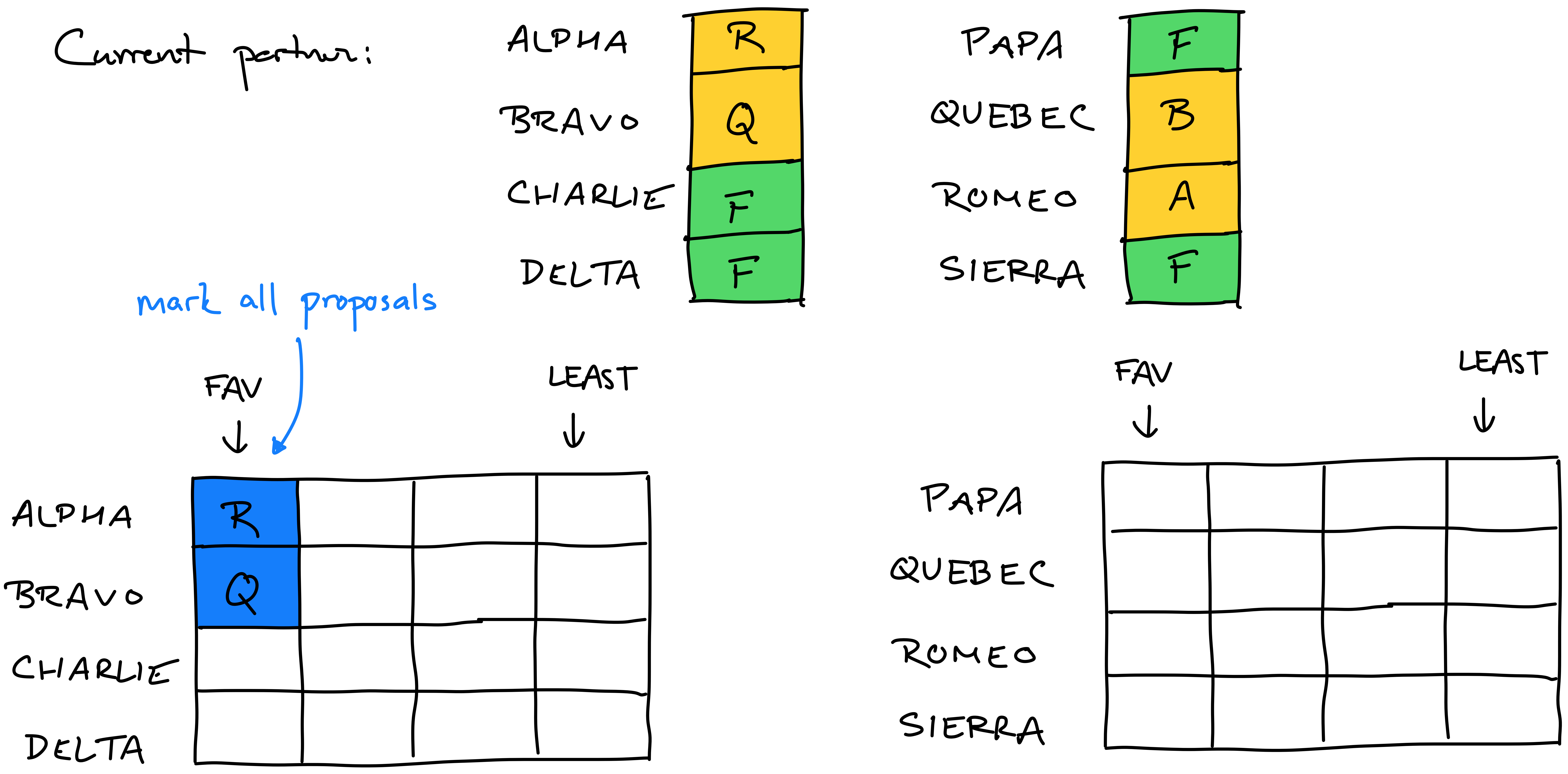
Gale-Shapley walkthrough



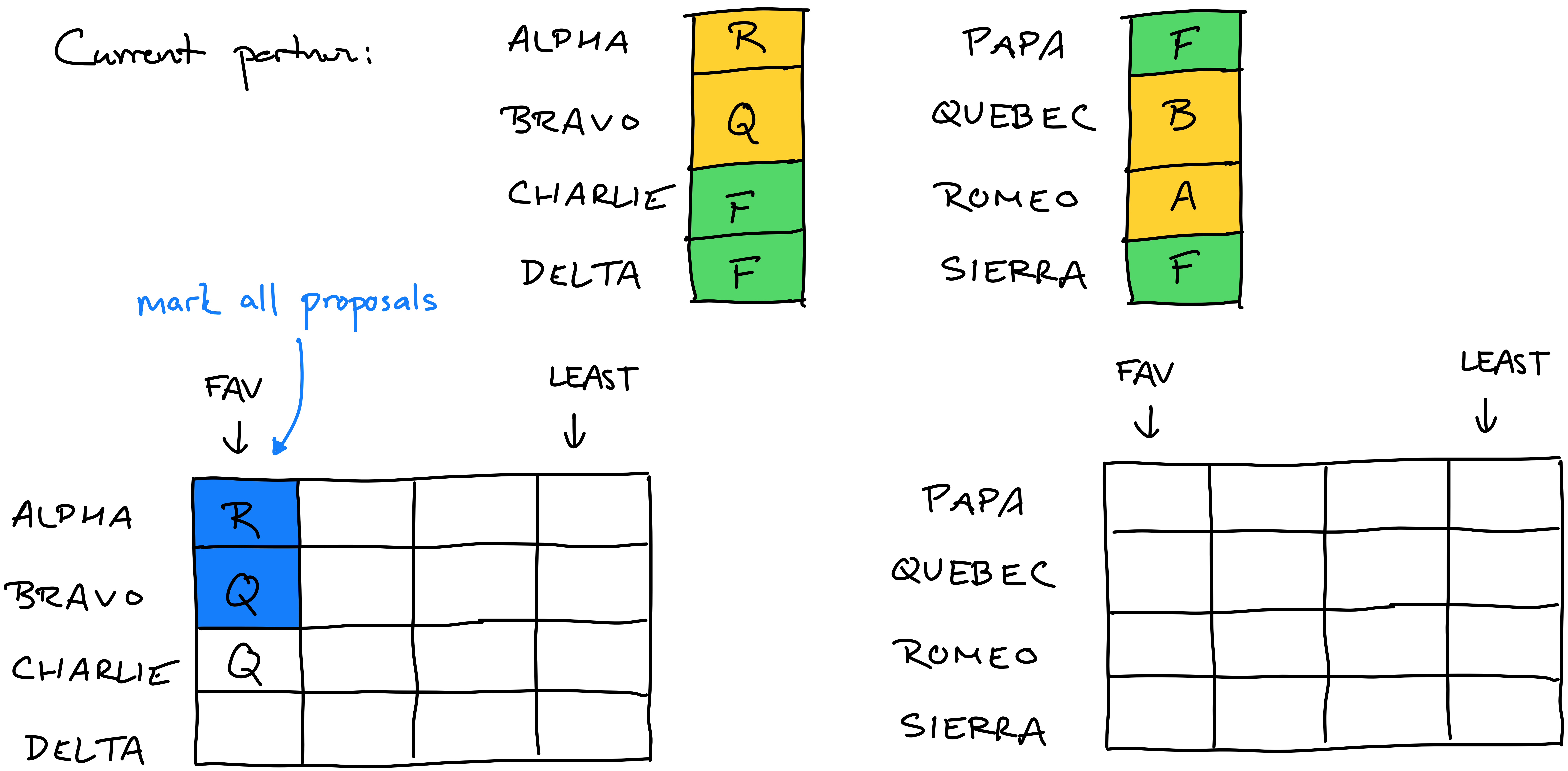
Gale-Shapley walkthrough



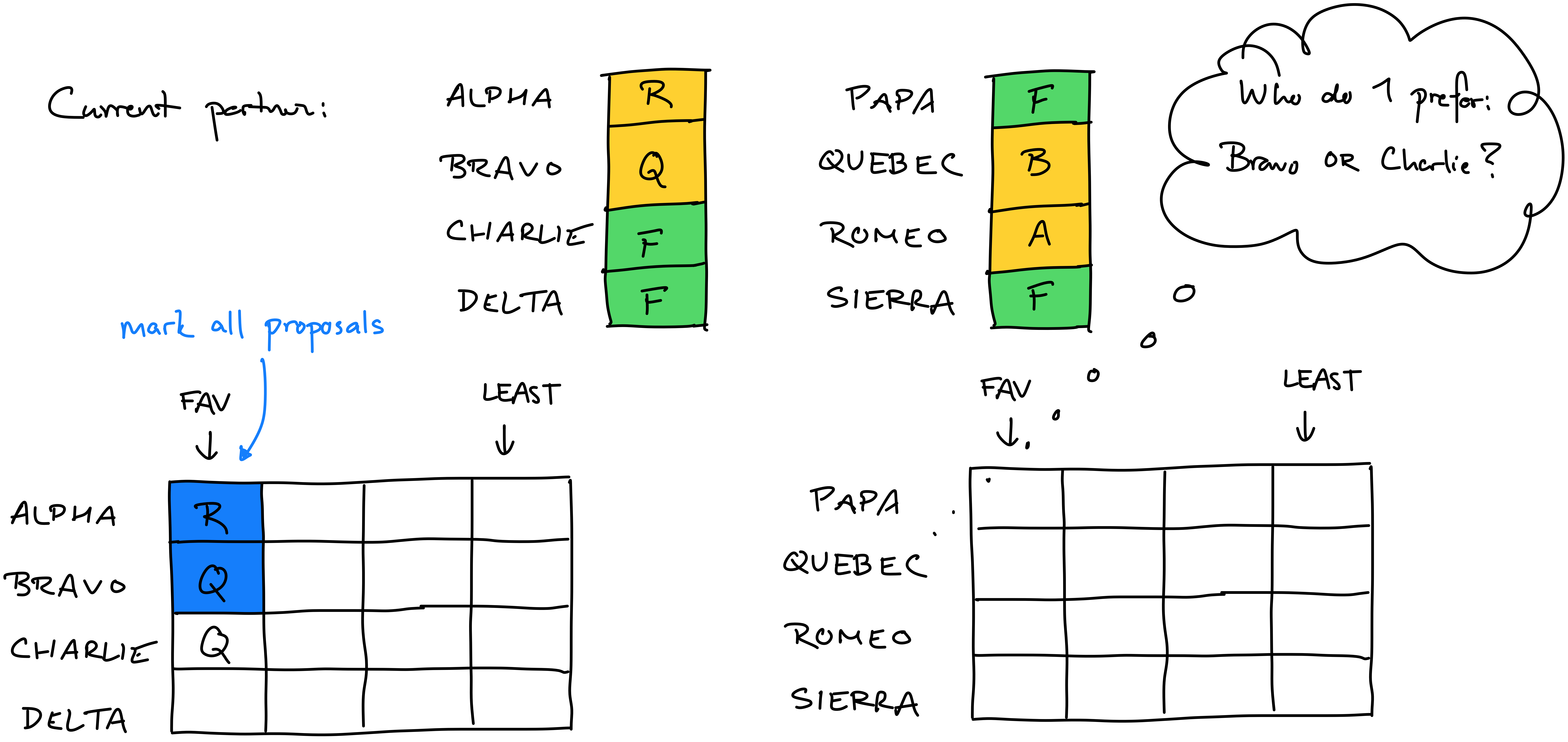
Gale-Shapley walkthrough



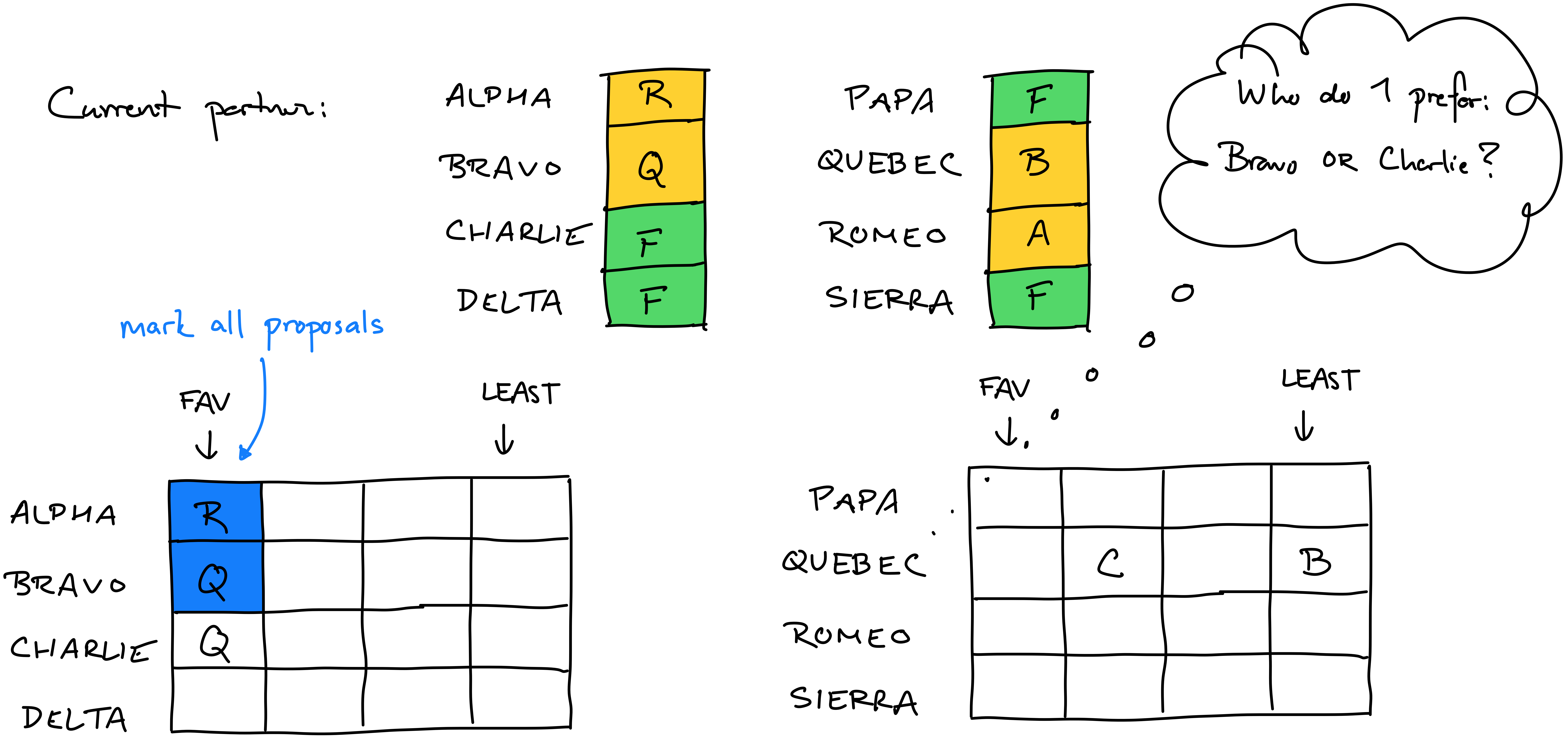
Gale-Shapley walkthrough



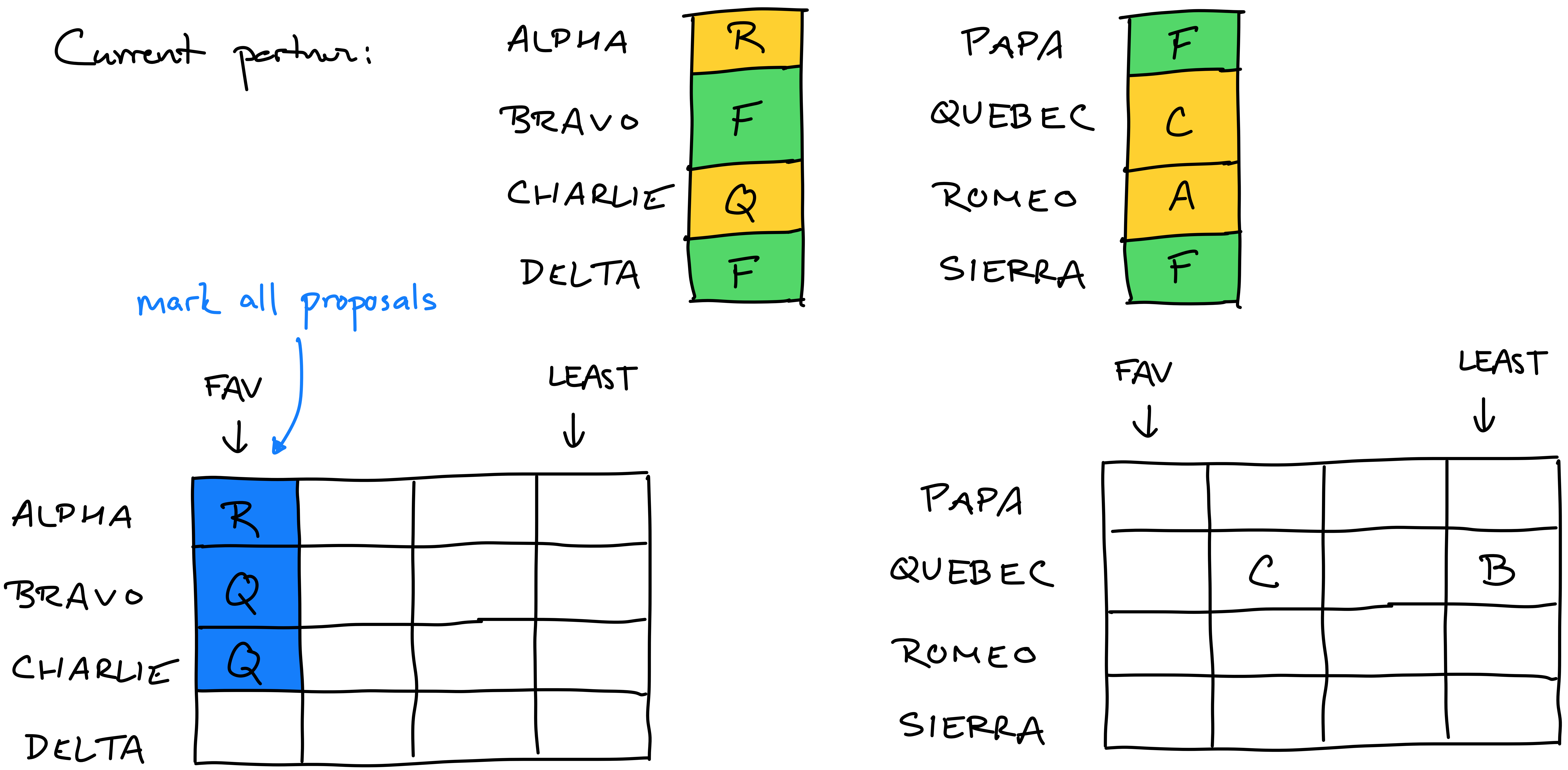
Gale-Shapley walkthrough



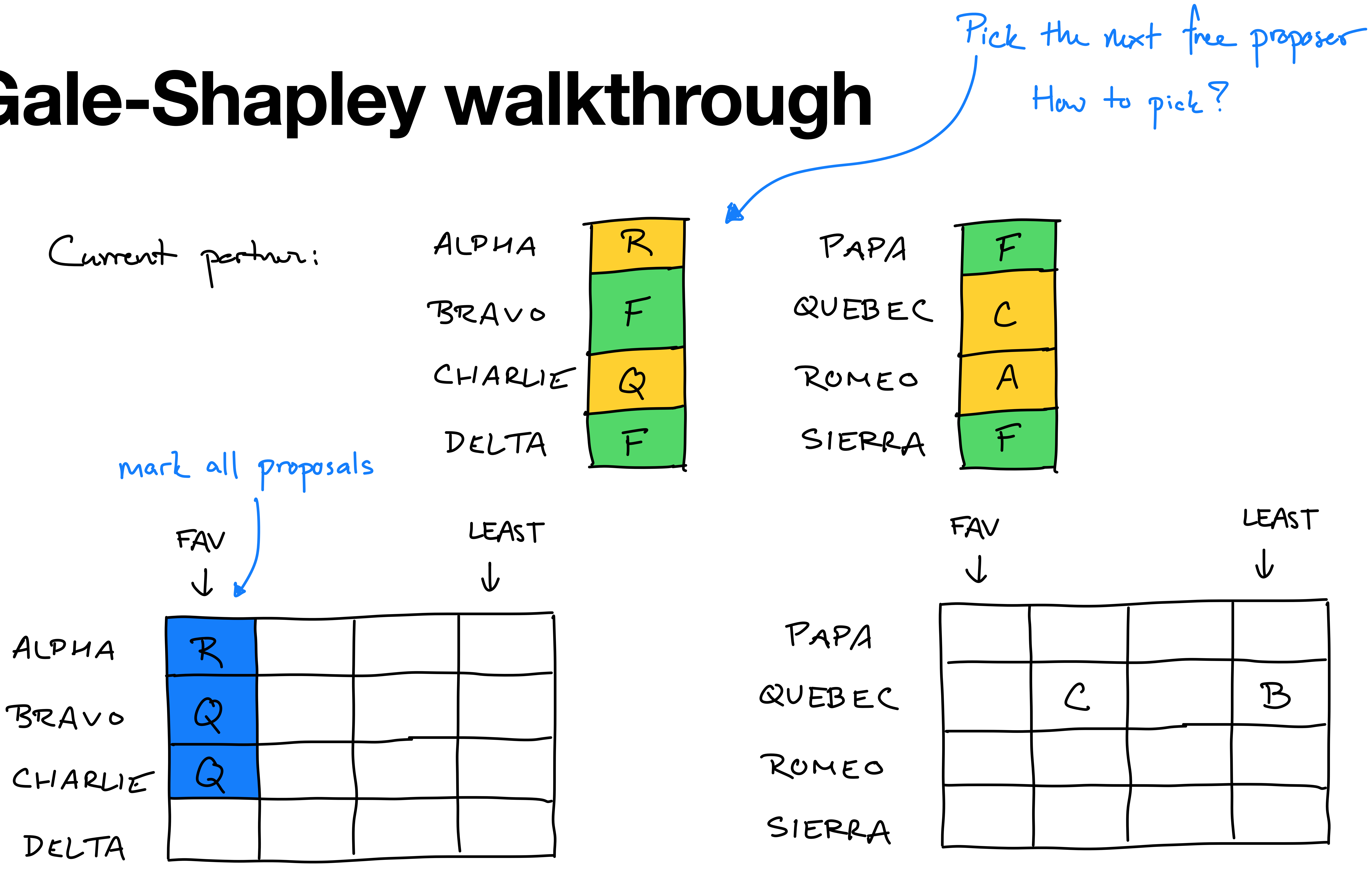
Gale-Shapley walkthrough



Gale-Shapley walkthrough

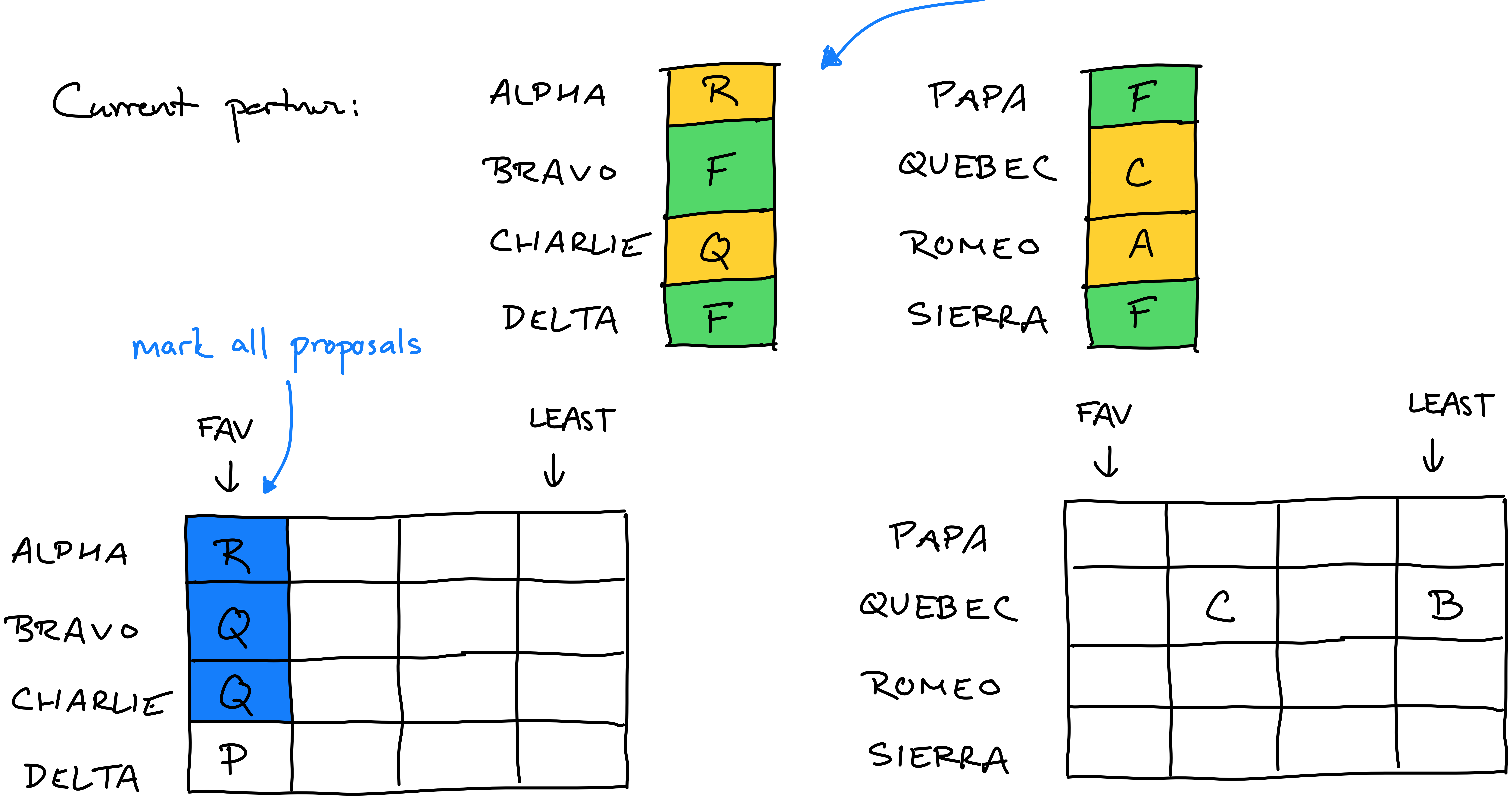


Gale-Shapley walkthrough

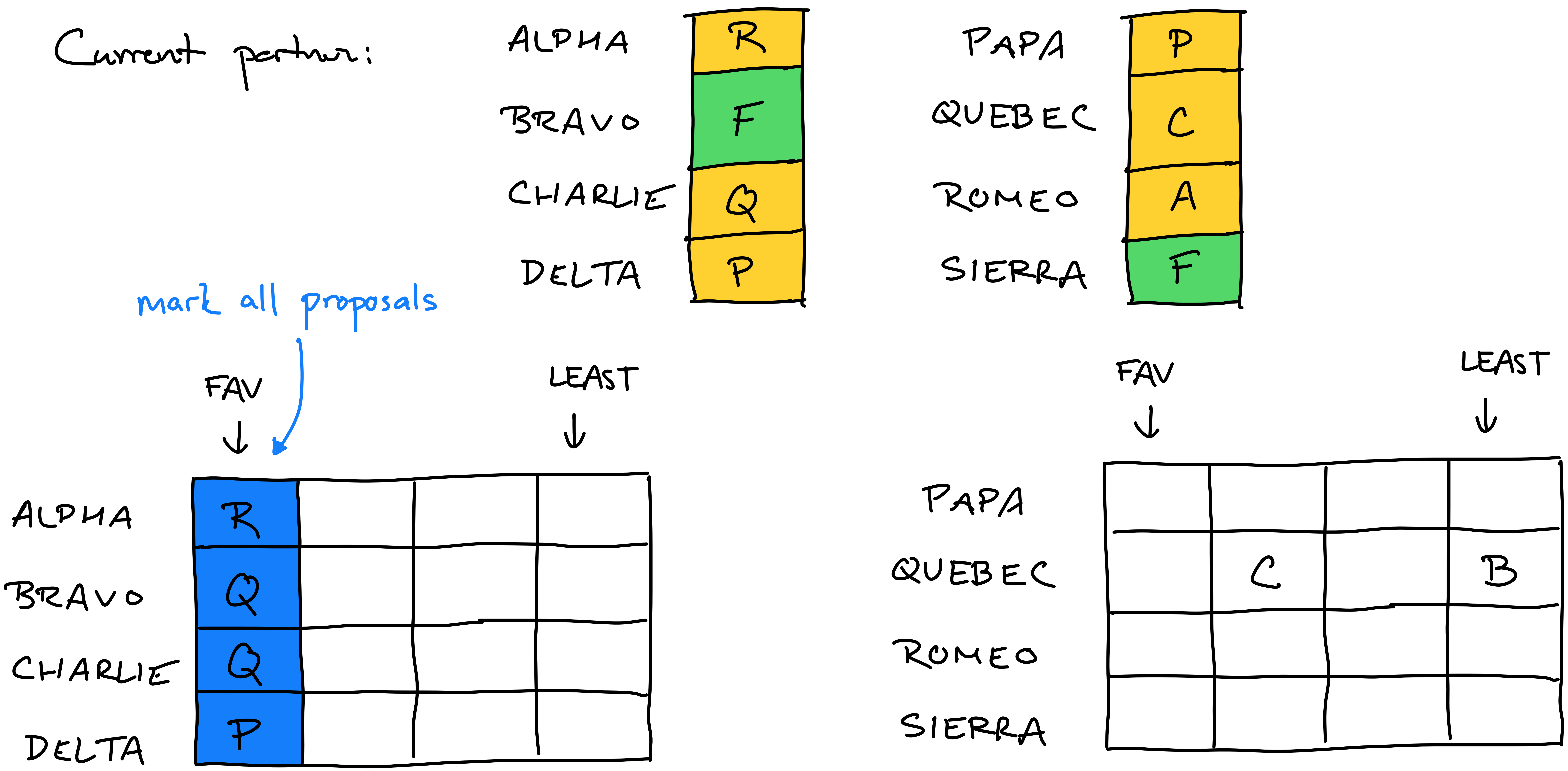


Gale-Shapley walkthrough

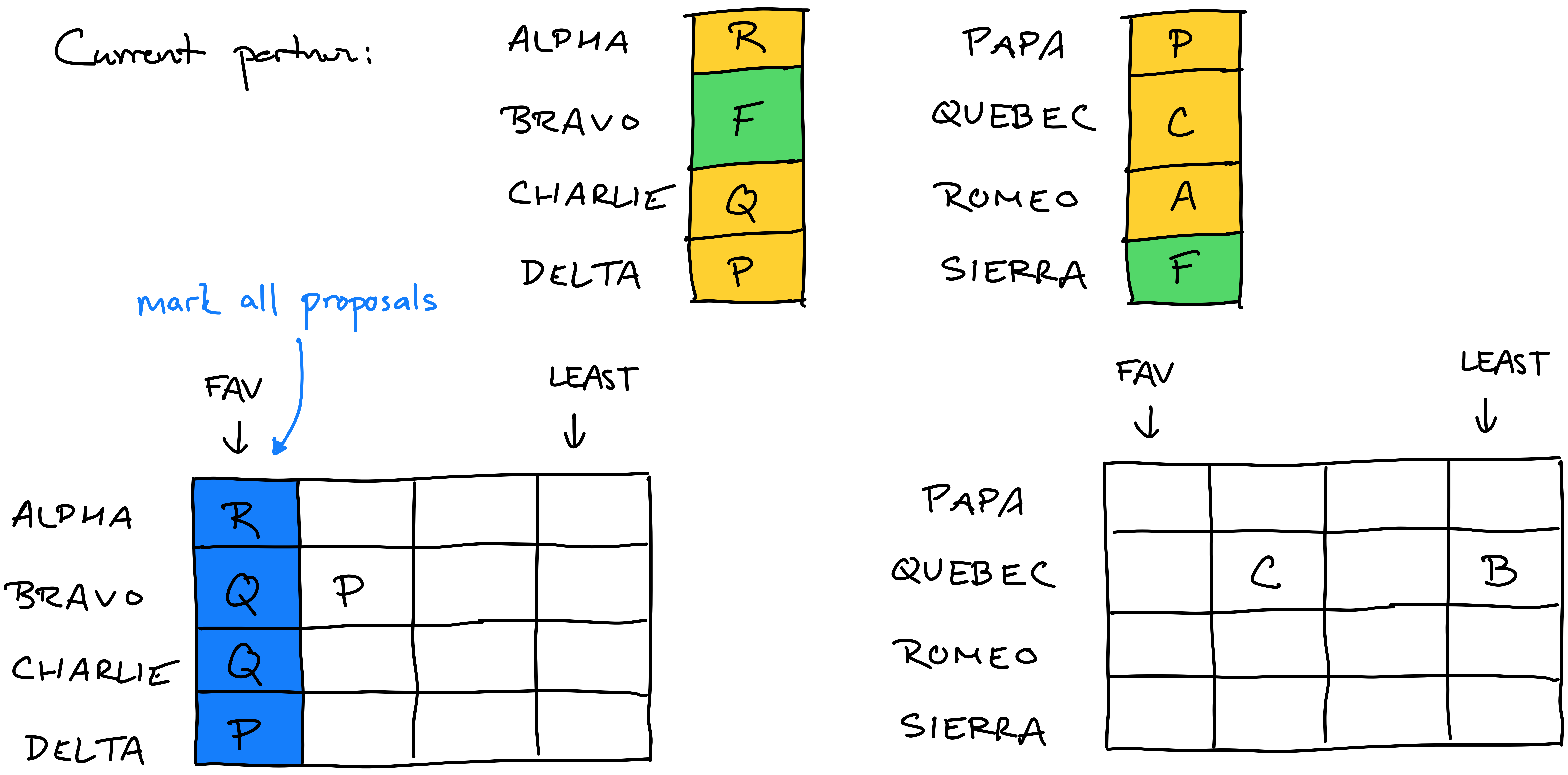
Pick the next free proposer
How to pick?



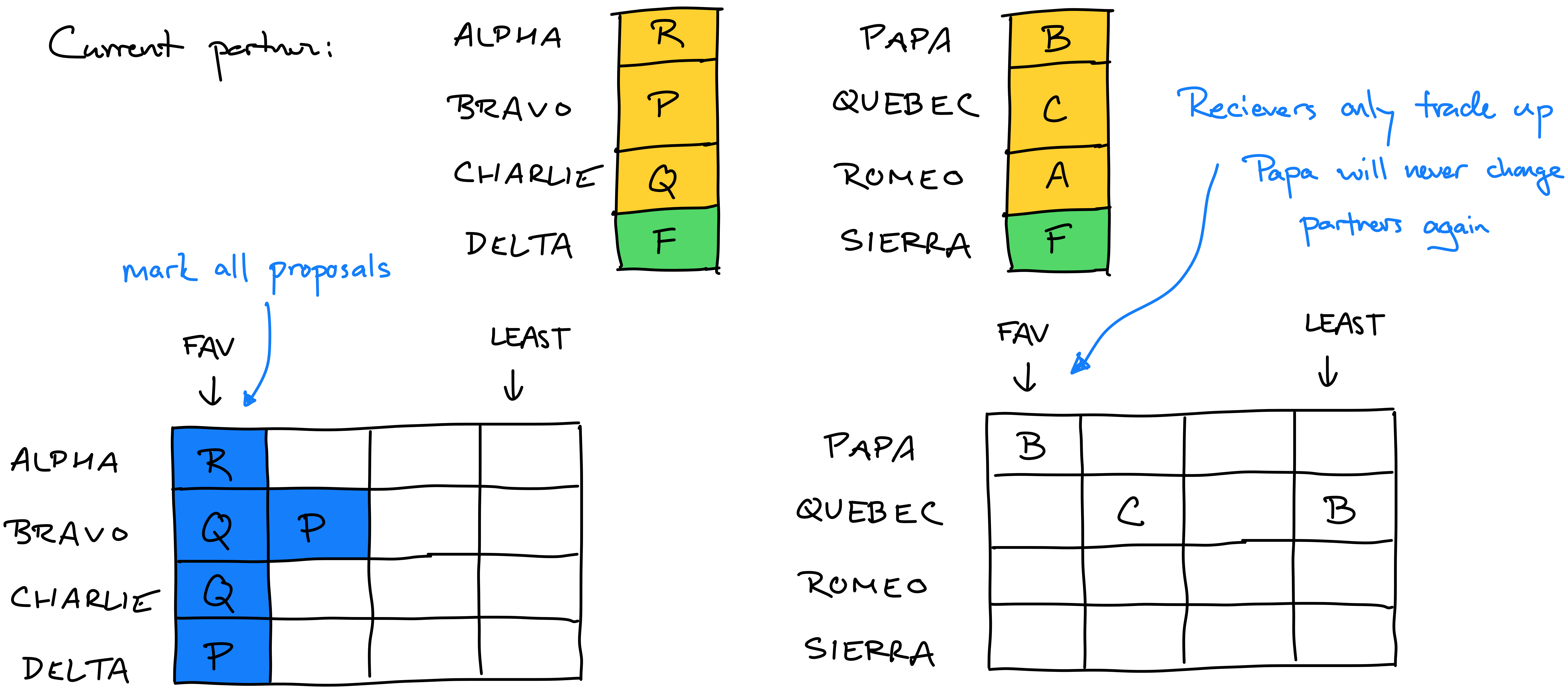
Gale-Shapley walkthrough



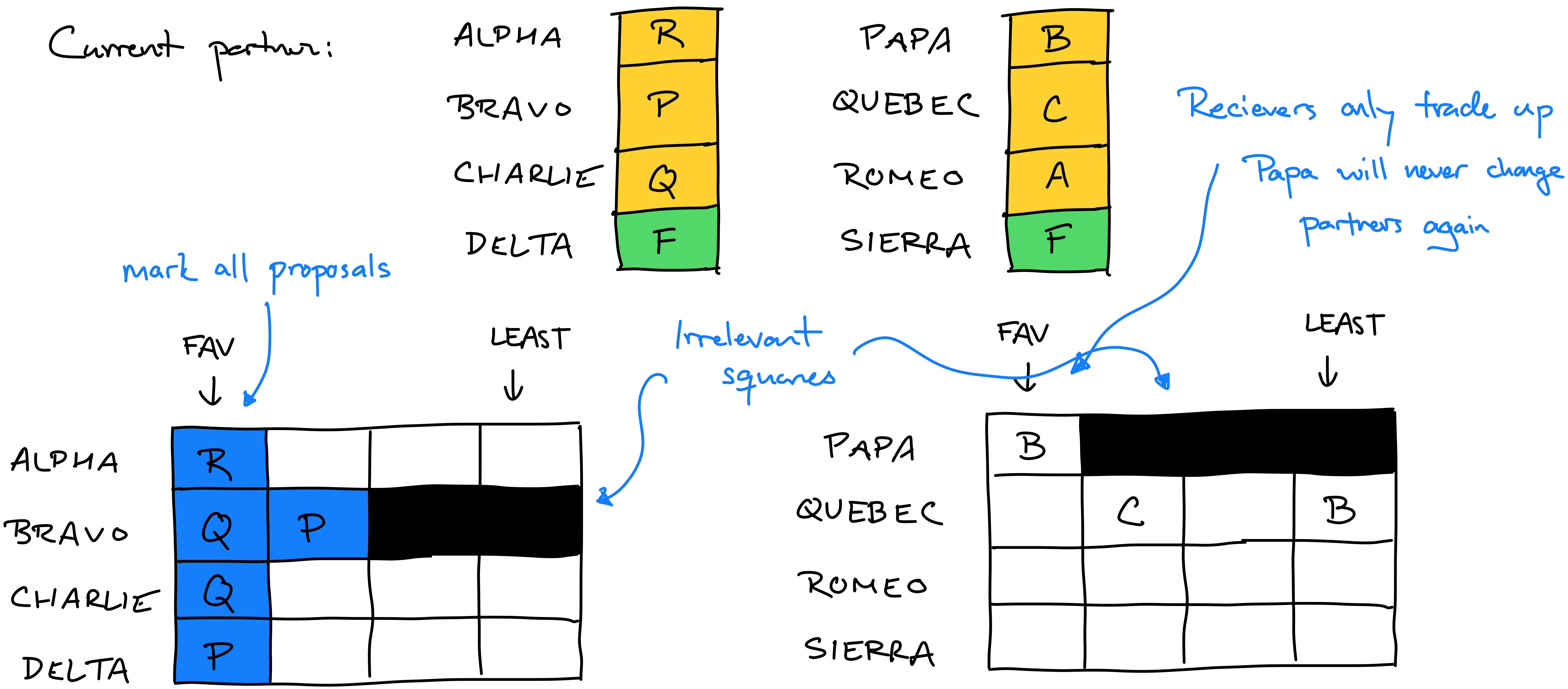
Gale-Shapley walkthrough



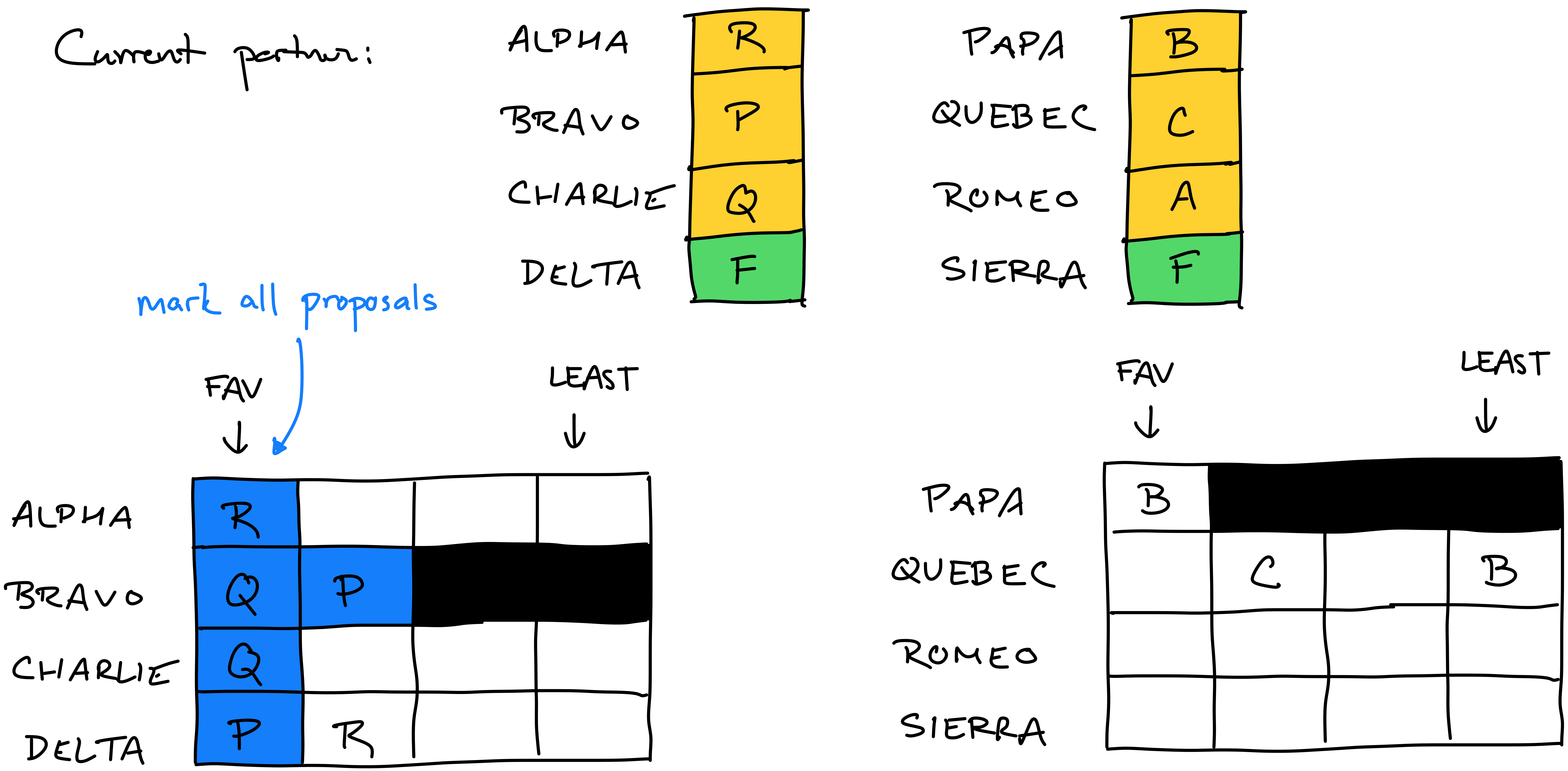
Gale-Shapley walkthrough



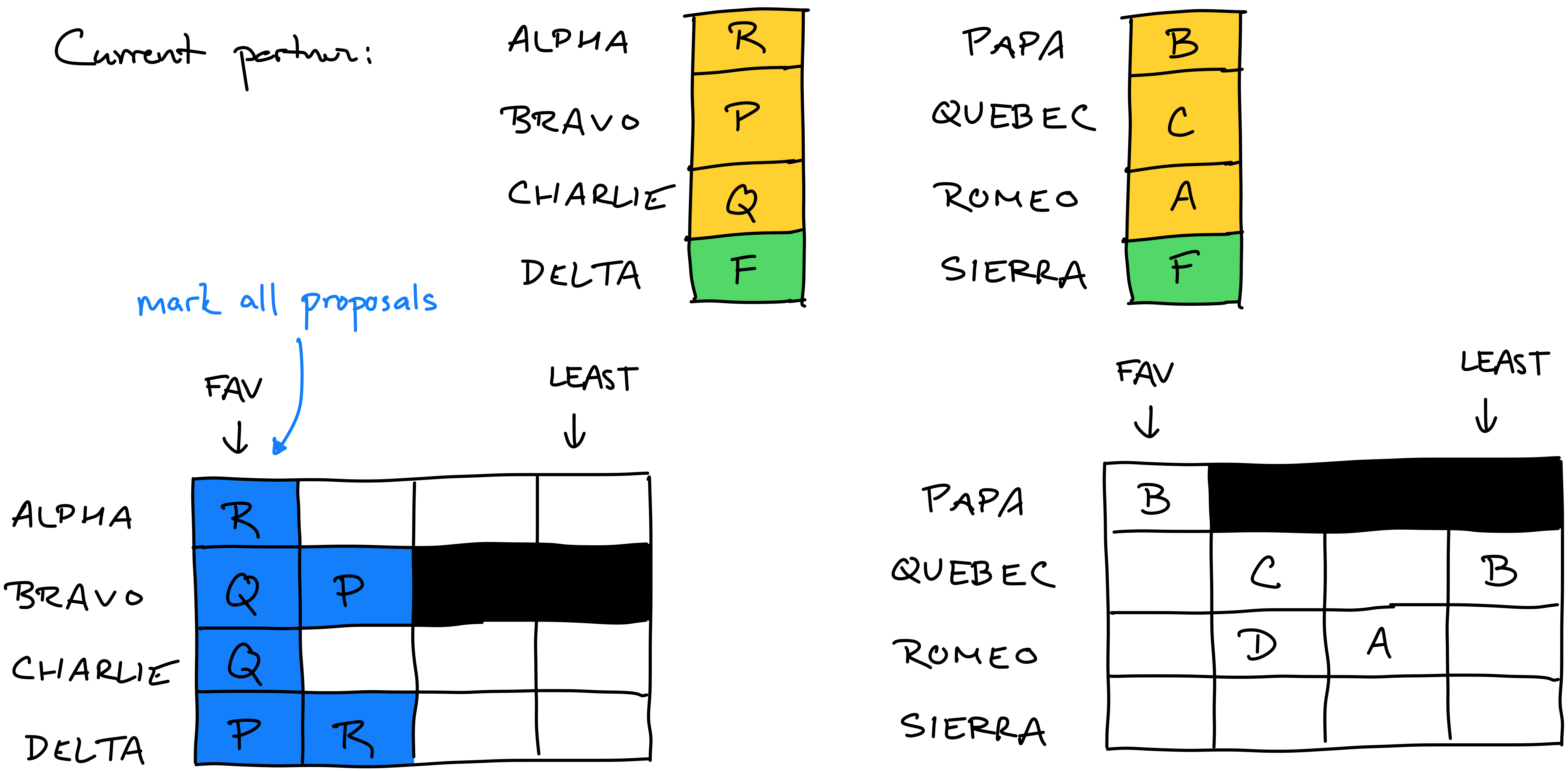
Gale-Shapley walkthrough



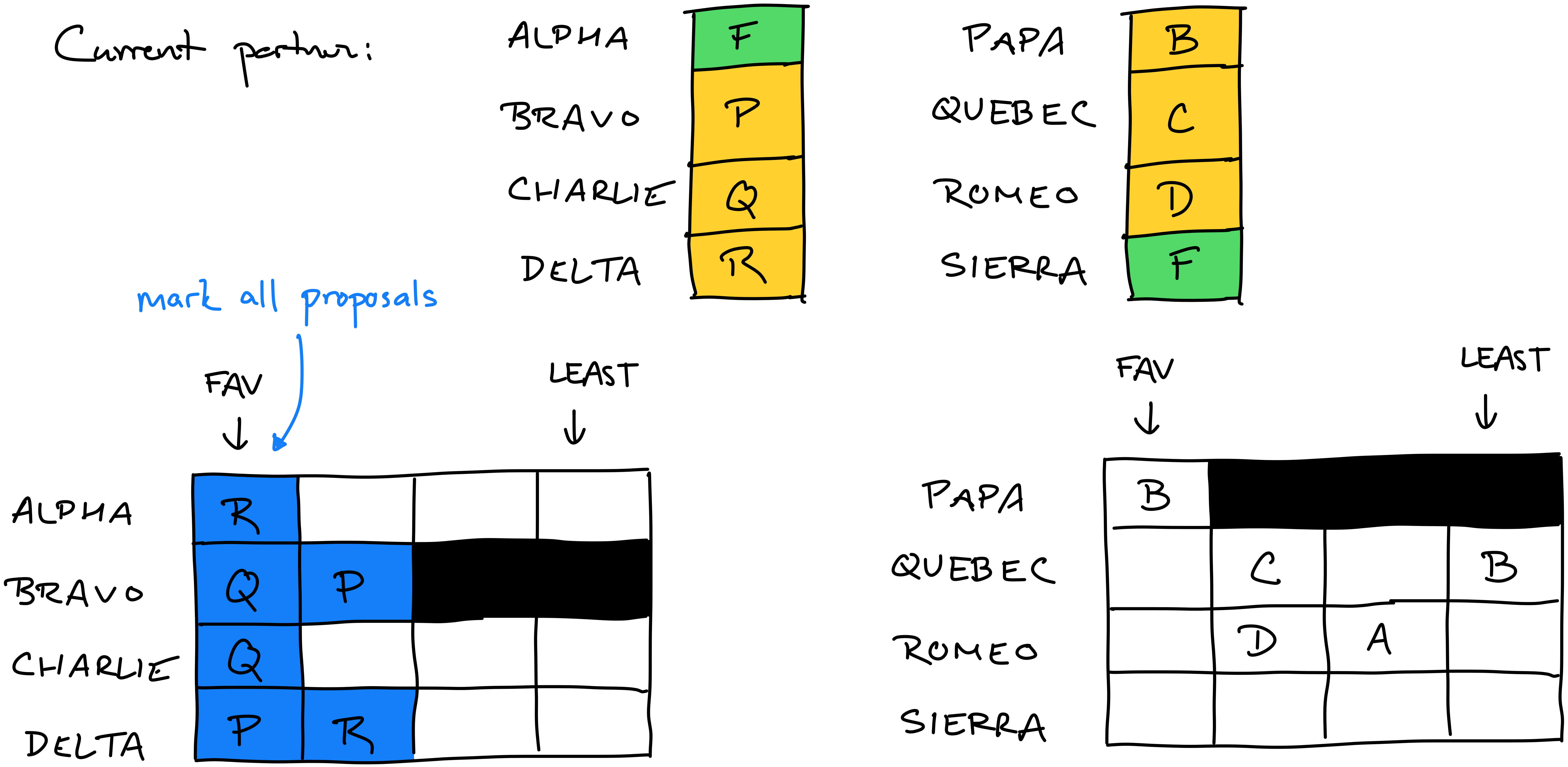
Gale-Shapley walkthrough



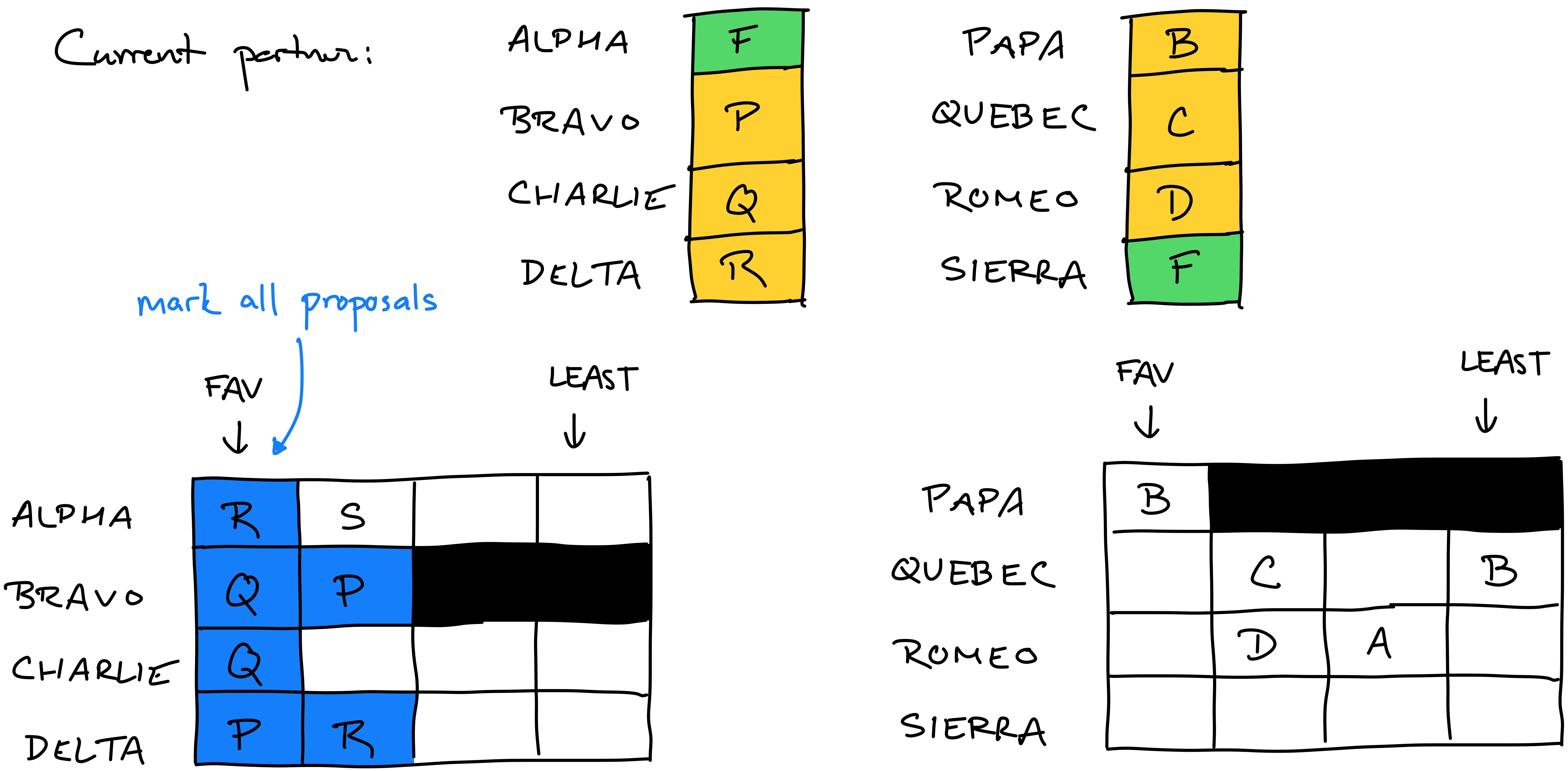
Gale-Shapley walkthrough



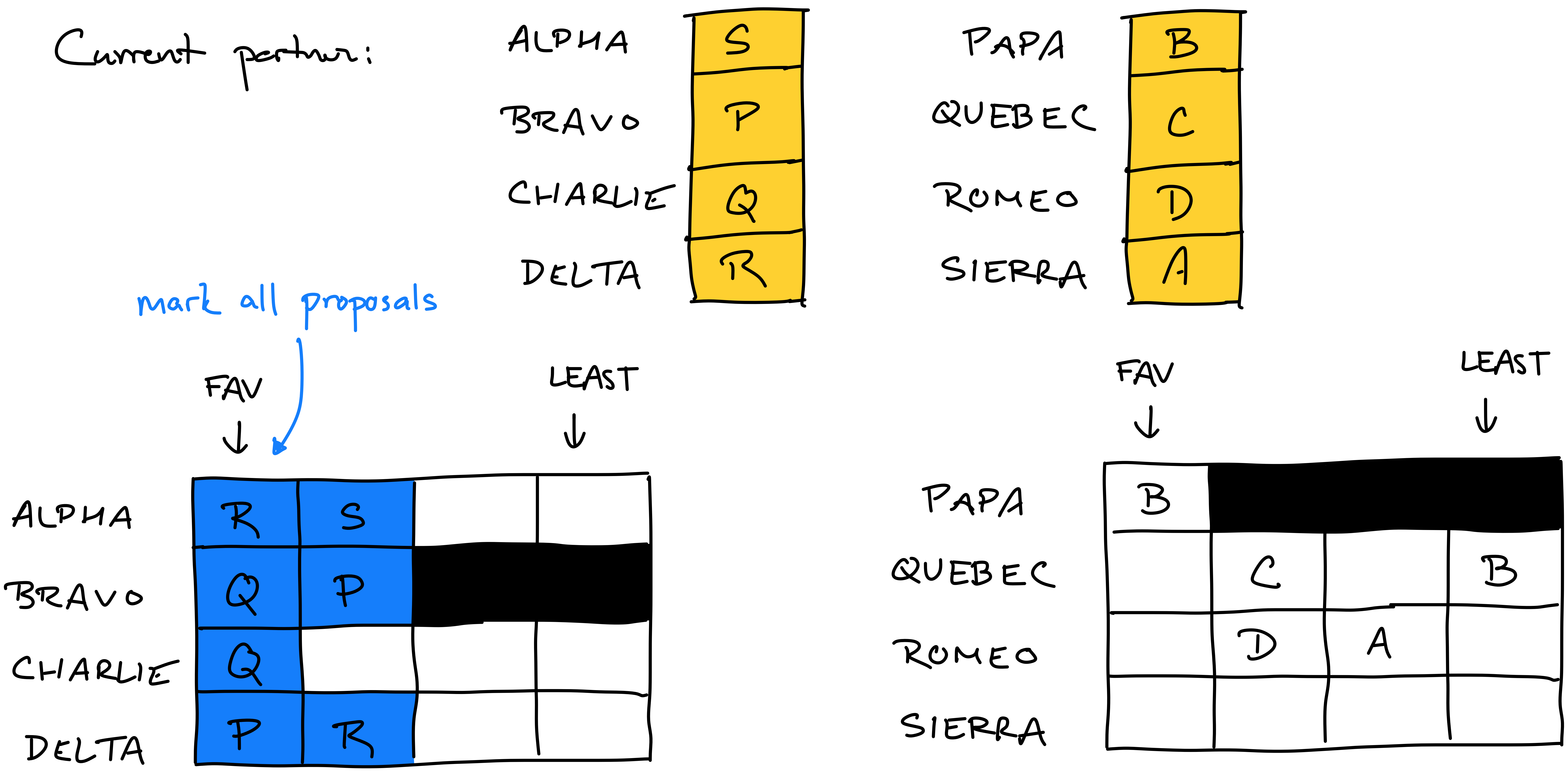
Gale-Shapley walkthrough



Gale-Shapley walkthrough

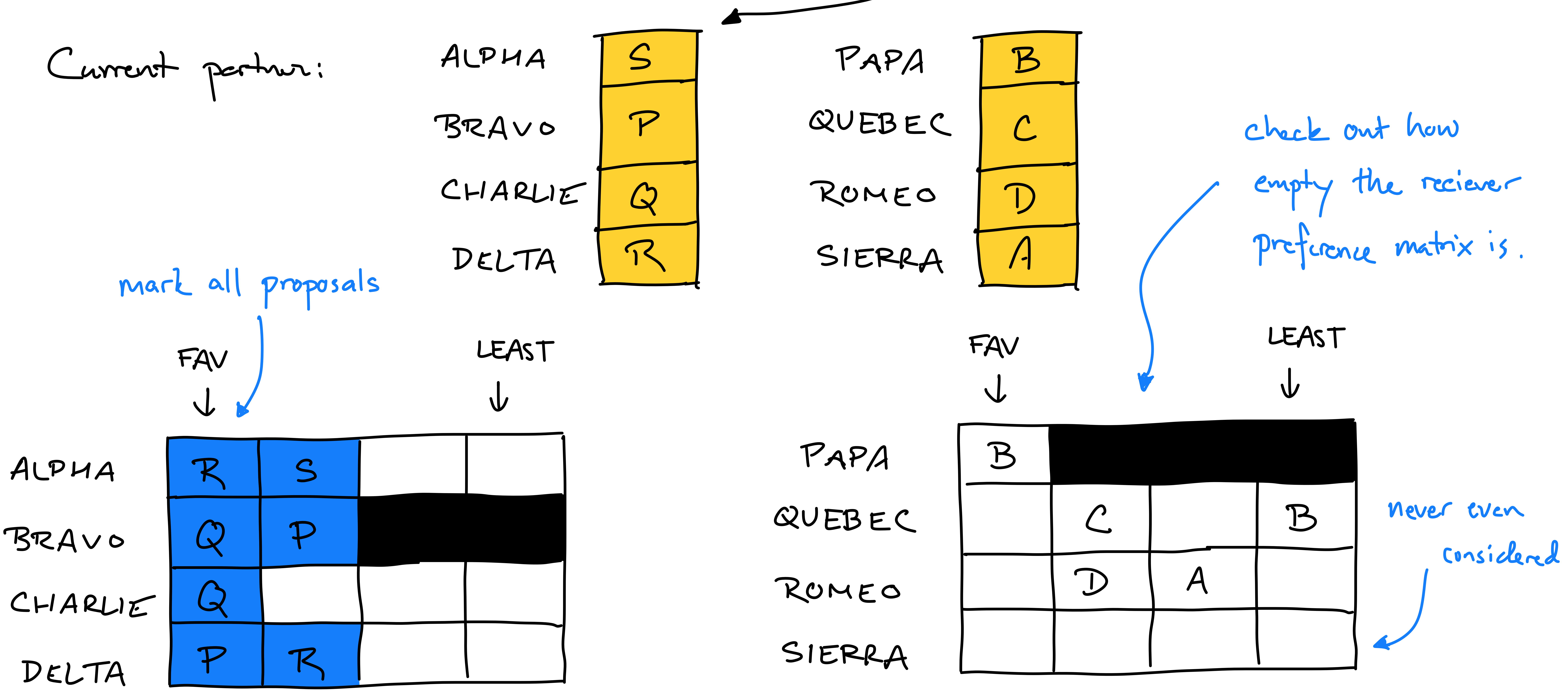


Gale-Shapley walkthrough



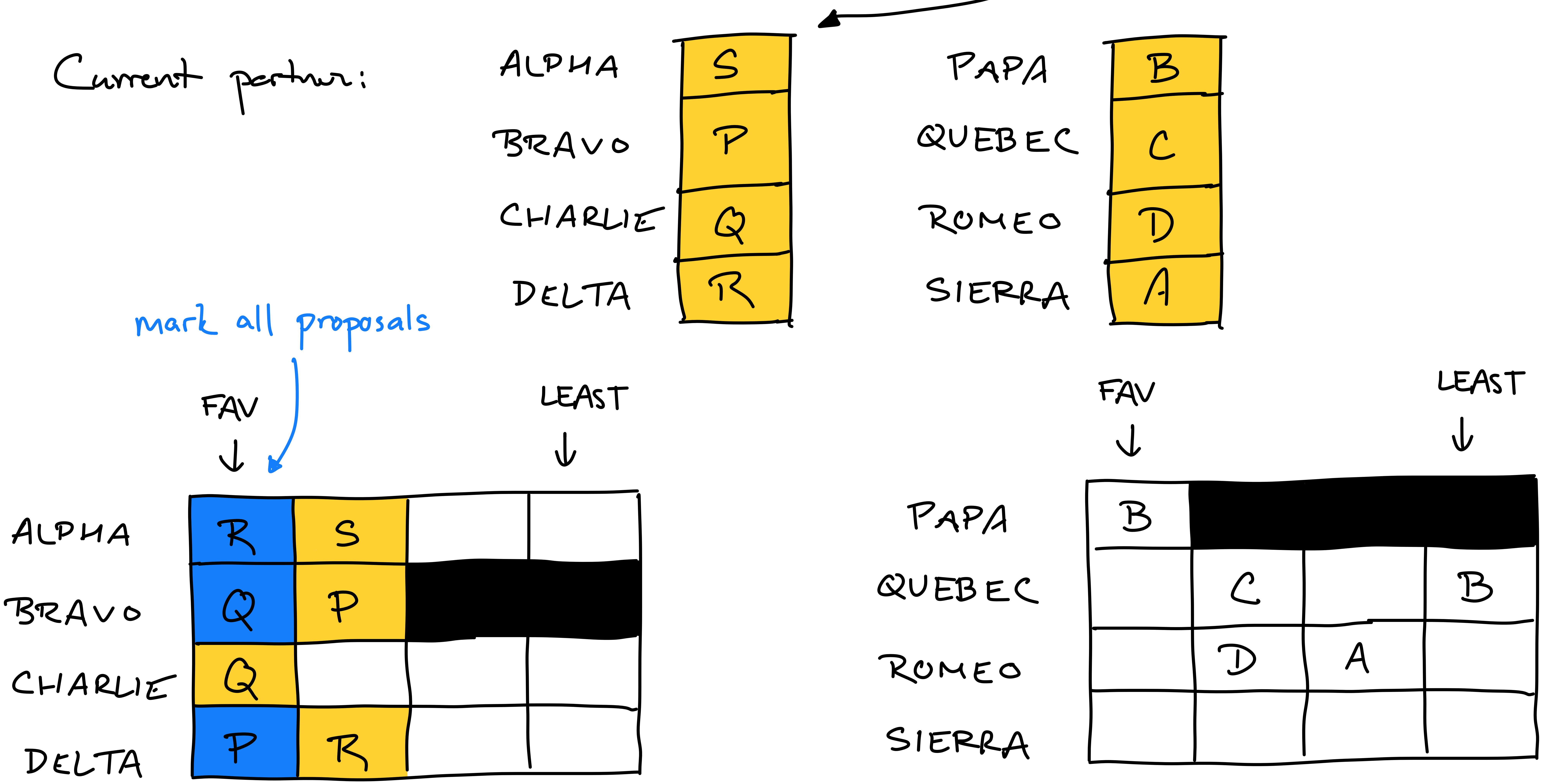
Gale-Shapley walkthrough

no free proposers.
Alg terminates and everyone is matched.

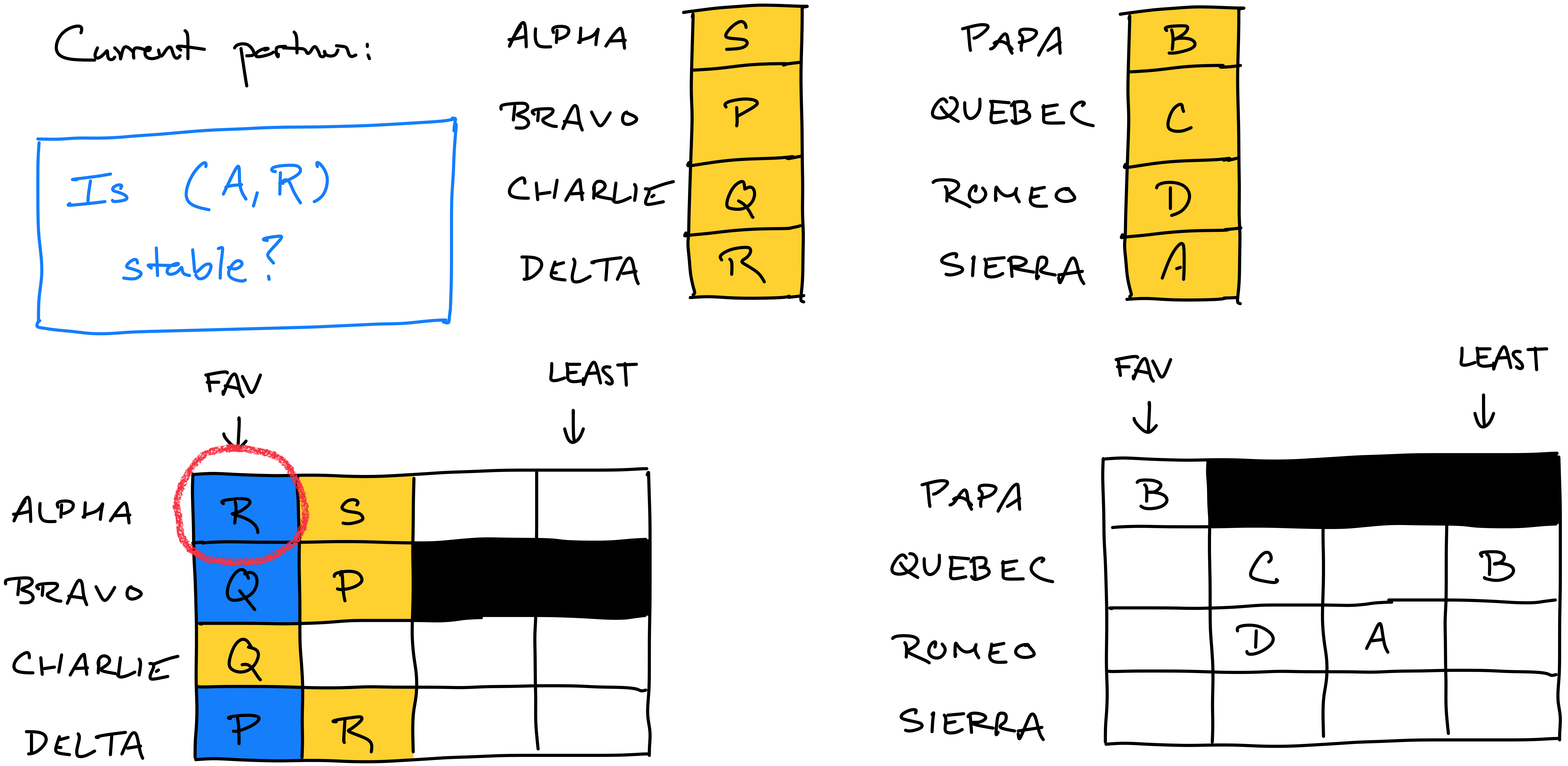


Gale-Shapley walkthrough

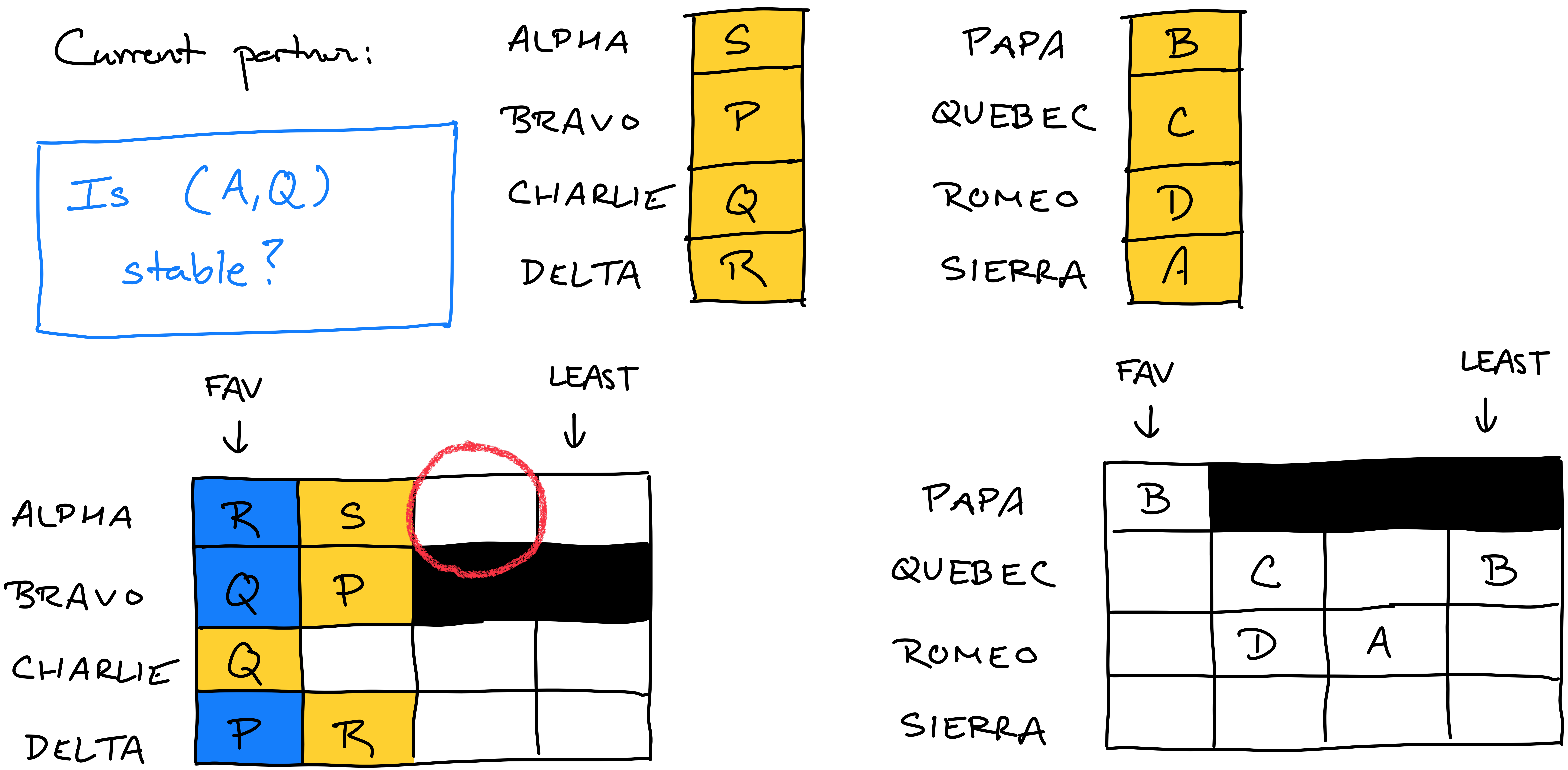
no free proposers.
Alg terminates and everyone is matched.



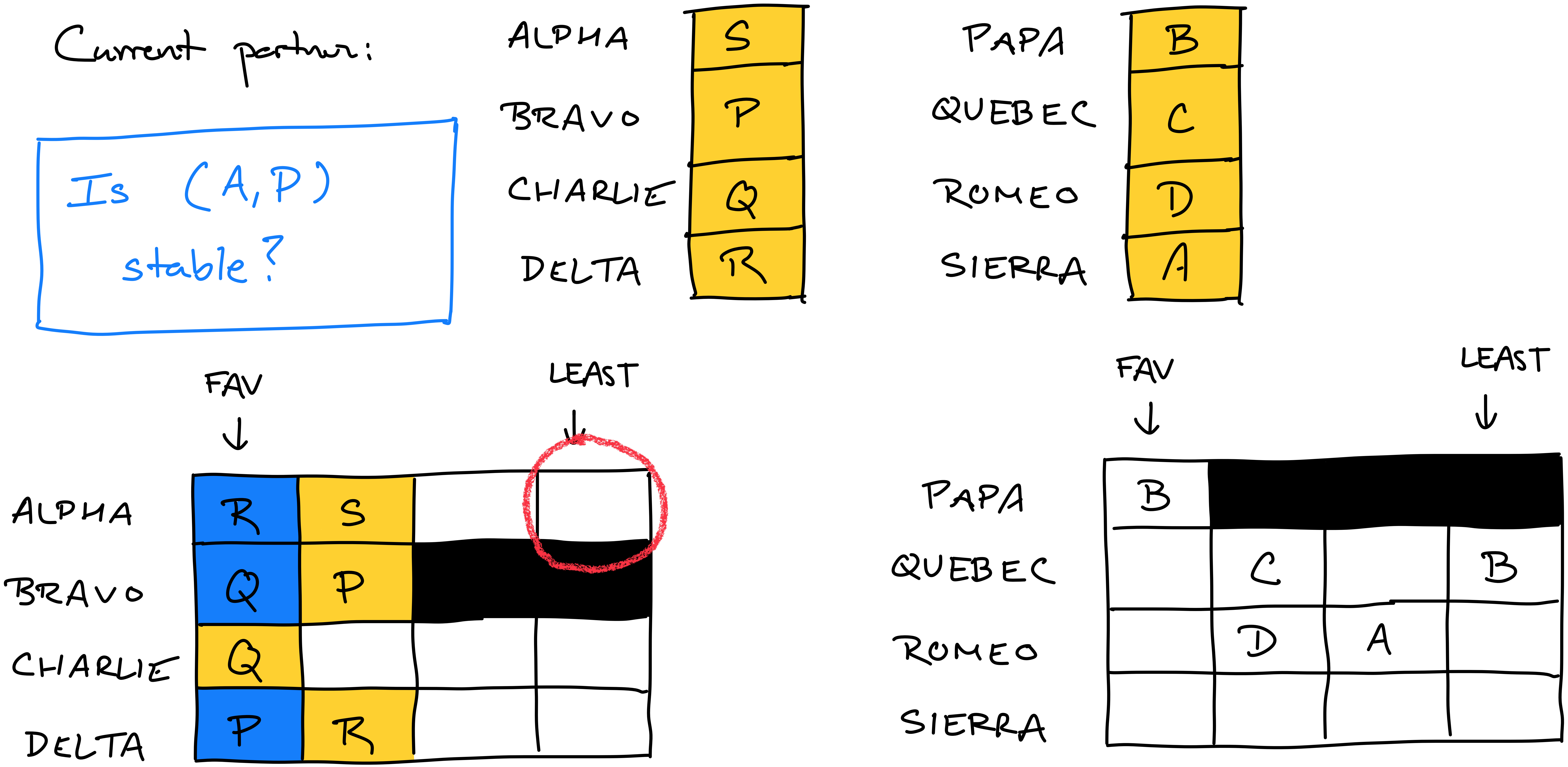
Gale-Shapley walkthrough



Gale-Shapley walkthrough



Gale-Shapley walkthrough



Implementing stable matching

- Input length
  - $N := 2n^2$ words in length because $2n$ people × preference list of length n.
  - A "word" here is a number $\in [n] = \{1,2,\ldots,n\}$. Takes $\lceil \log_2 n \rceil$ bits to represent.
  - Input length of $2n^2 \lceil \log_2 n \rceil$ bits.
- **Brute force algorithm**: Try all $n!$ possible matchings. Testing if a matching is stable requires testing if each of the n^2 pairs (p, r) is stable.
- **Gale-Shapley algorithm**: takes $\leq n^2$ iterations. How long does each iteration take to run?

Implementing Gale-Shapley in $O(n^2)$ time

Comparing

- **Input**: 2 $n \times n$ representing the preferences of P and R:
  - $\text{pref}_P[p][j], \text{pref}_R[r][j]$
  - Assume the proposers and receivers are numbers $1,2,\ldots,n$
  - Each preference array is a *permutation* of $\{1,2,\ldots,n\}$
- **Data structure for the matching**:
  - Maintain two arrays $M_P[p]$ and $M_R[r]$ denoting match of p and r
  - Initialize both arrays to all $\perp$, a symbol denoting that the match isn't set
  - If during the algorithm, (p, r) is matched, set $M_P[p] \leftarrow r, M_R[r] \leftarrow p$
- **Making proposals**:
  - Maintain a queue Q of all the free proposers. Initially Q contains all n proposers.
  - Maintain an array $\text{count}[p]$ which counts how many proposals p has made so far. Initially all entries are 0.

```
Initialize each person to be free.
while (some p in P is free) {
    Choose some free p in P
    r = 1st person on p's preference list to whom p has not yet proposed
    if (r is free)
        tentatively match (p,r)   //p and r both engaged, no longer free
    else if (r prefers p to current tentative match p')
        replace (p',r) by (p,r)   //p now engaged, p' now free
    else
        r rejects p
}
```

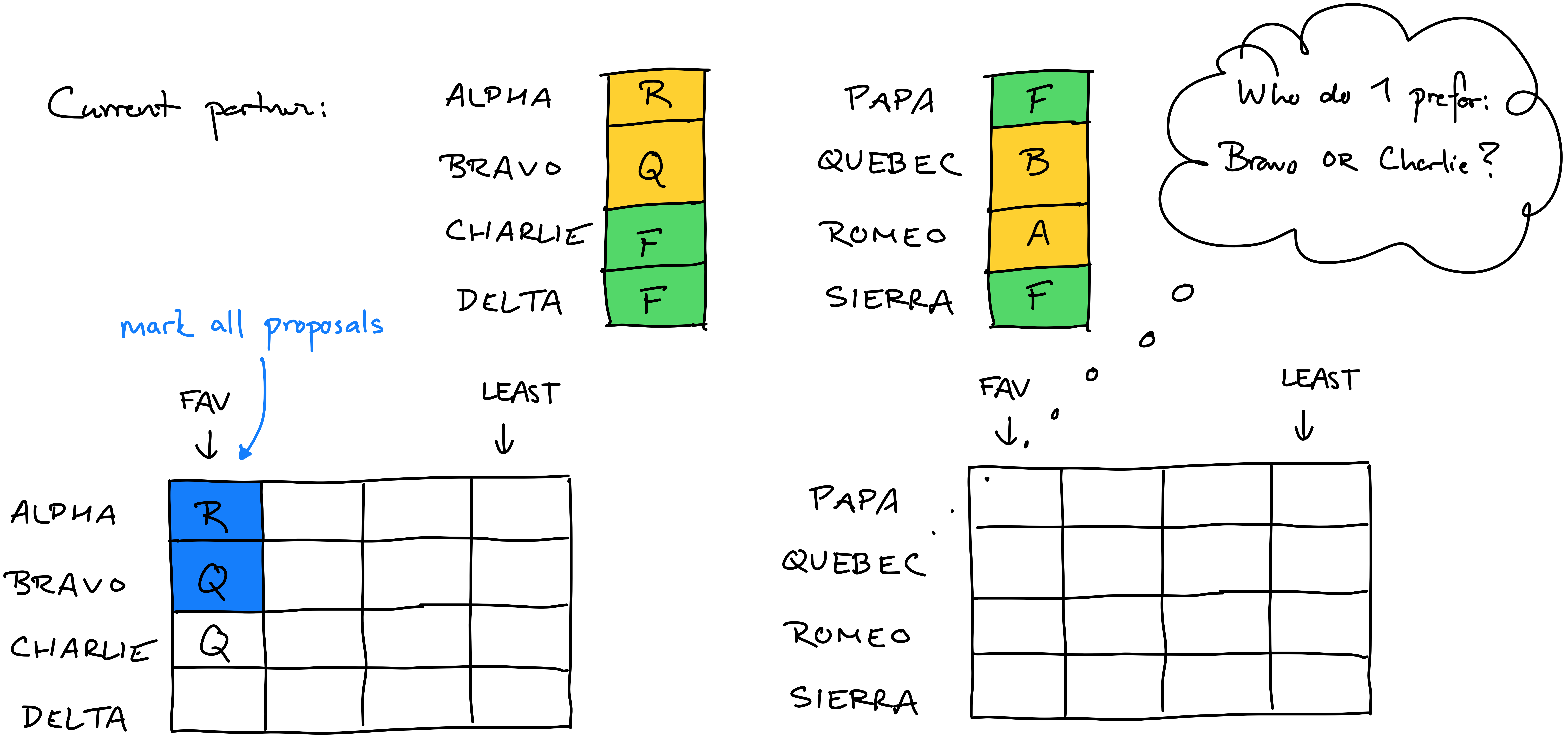
Implementing Gale-Shapley in $O(n^2)$ time

Rejecting & accepting proposals

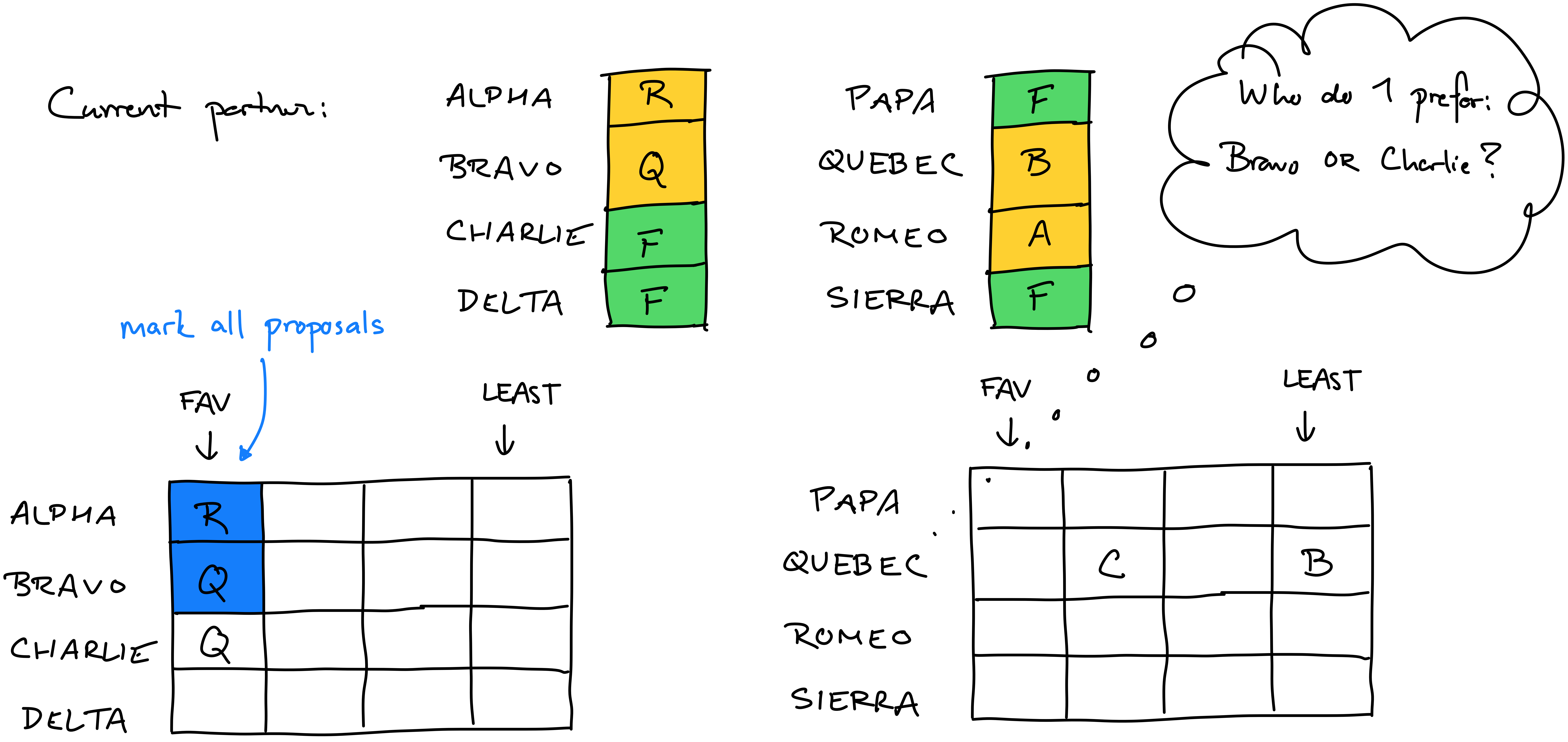
- How do we decide efficiently if receiver r prefers proposer p to proposer p'?
- Naïvely would take $O(n)$ queries to read through $\mathrm{pref}_R[r][\cdot]$ to find both p and p'

```
Initialize each person to be free.
while (some p in P is free) {
    Choose some free p in P
    r = 1st person on p's preference list to whom p has not yet proposed
    if (r is free)
        tentatively match (p,r)    //p and r both engaged, no longer free
    else if (r prefers p to current tentative match p')
        replace (p',r) by (p,r)    //p now engaged, p' now free
    else
        r rejects p
}
```

Gale-Shapley walkthrough



Gale-Shapley walkthrough



Implementing Gale-Shapley in $O(n^2)$ time

Rejecting & accepting proposals

- How do we decide efficiently if receiver r prefers proposer p to proposer p'?
- Naïvely would take $O(n)$ queries to read through $\text{pref}_R[r][\,\cdot\,]$ to find both p and p'
- Instead, *precompute* the inverse list of preferences: $\text{invpref}_R[r][p]$.
- Property: $j = \text{invpref}_R[r][p]$ if and only if $p = \text{pref}_R[r][j]$.
- Takes $O(n^2)$ time to precompute inverse list. Once computed, each comparison takes time $O(1)$.

```
Initialize each person to be free.
while (some p in P is free) {
    Choose some free p in P
    r = 1st person on p's preference list to whom p has not yet proposed
    if (r is free)
        tentatively match (p,r)   //p and r both engaged, no longer free
    else if (r prefers p to current tentative match p')
        replace (p',r) by (p,r)   //p now engaged, p' now free
    else
        r rejects p
}
```

r	1st	2nd	3rd	4th	5th	6th	7th	8th
pref	8	3	7	1	4	5	6	2

r	1	2	3	4	5	6	7	8
inverse	4th	8th	2nd	5th	6th	7th	3rd	1st

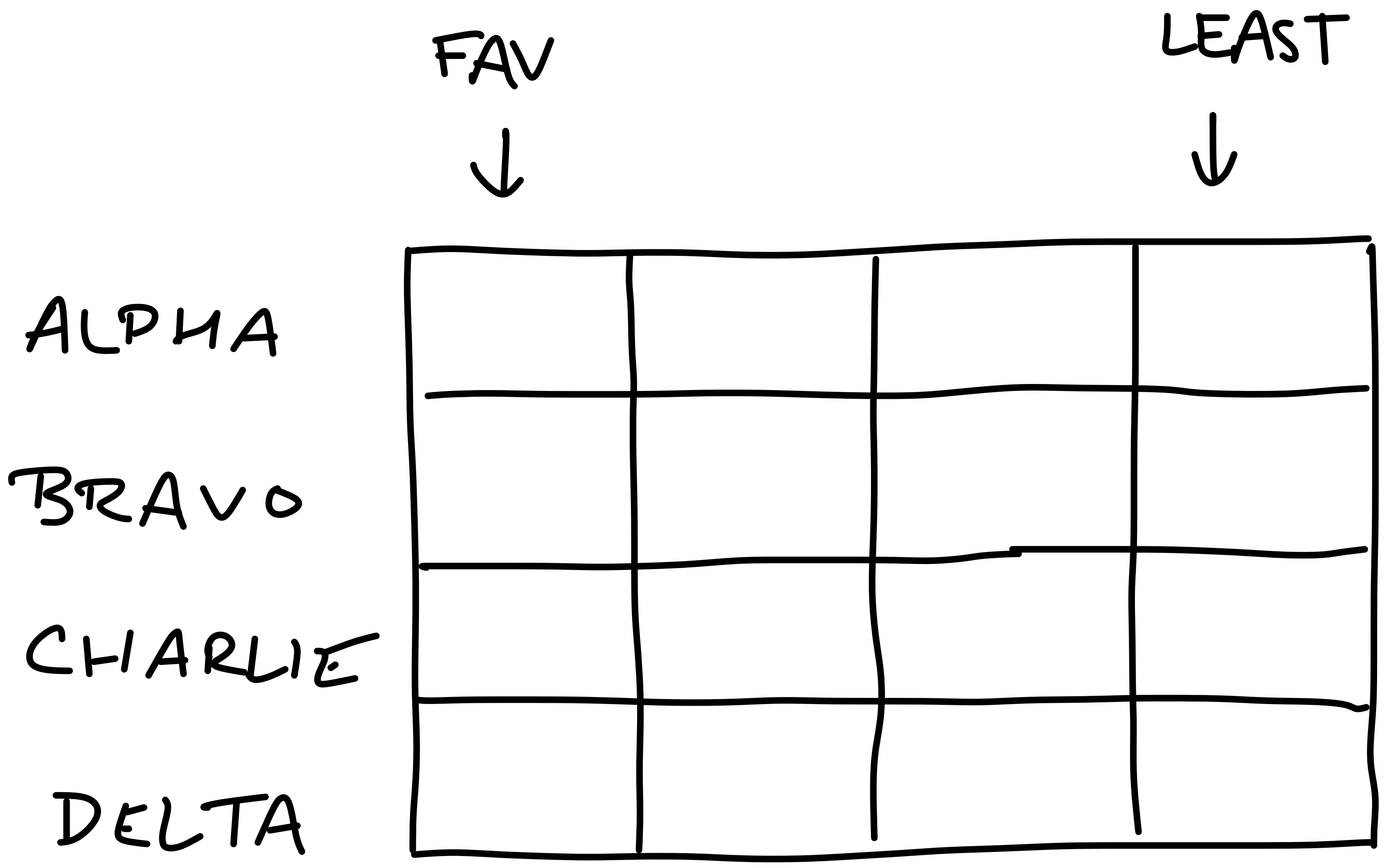
```
for i = 1 to n
   invpref[r][pref[r][i]] = i
```

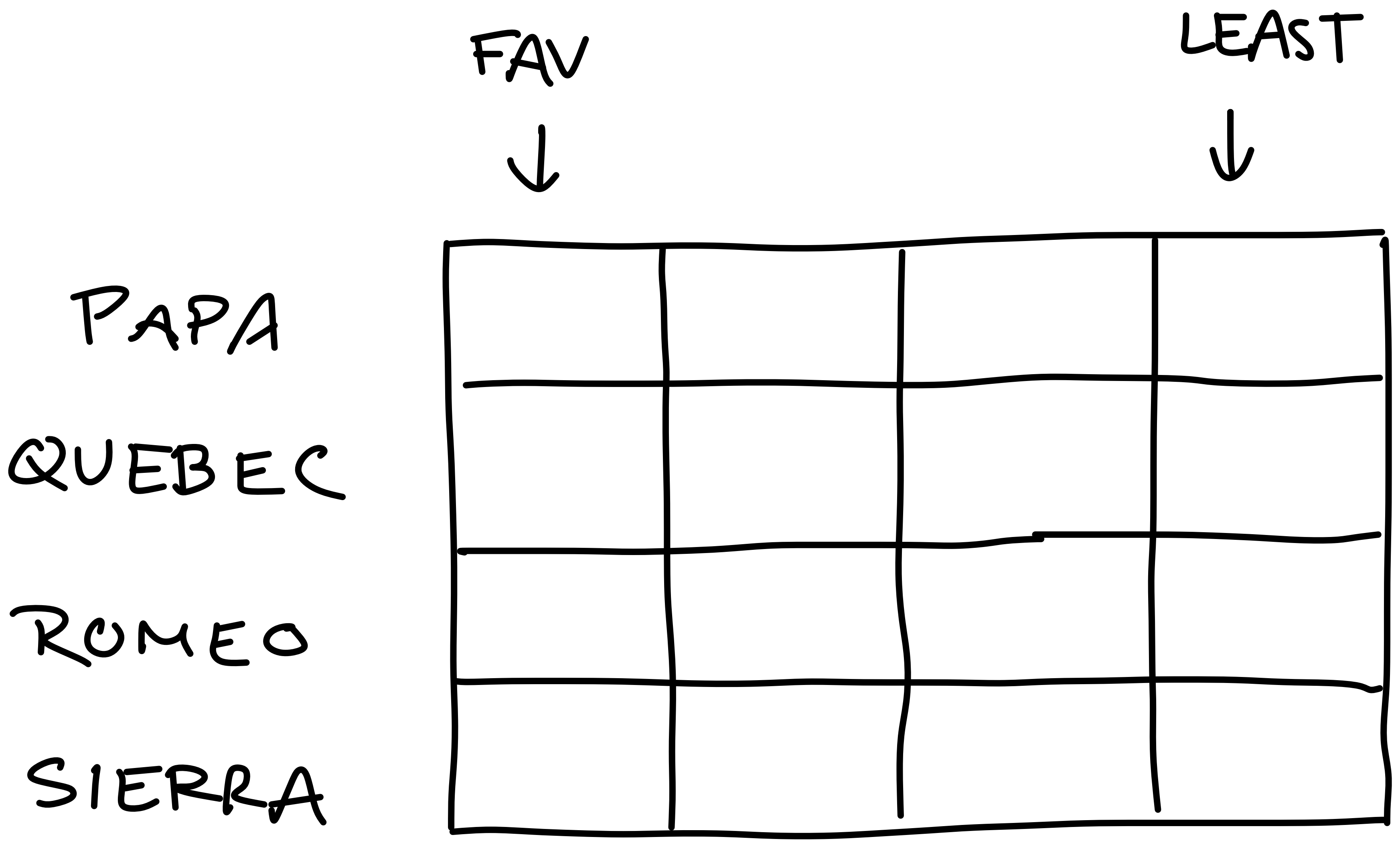
Implementing Gale-Shapley in $O(n^2)$ time

- When a proposer p becomes free, p starts proposing to new receivers starting from $\text{count}[p]$. All previous receivers have been proposed to in previous steps of the algorithm. Update $\text{count}[p]$ as rejections occur.
- Combined with the inverse list pre computation, we achieve that every proposer-receiver pair (p, r) is considered in $O(1)$ computational steps and there are a total n^2 possible pairs.
- This completes the entire time complexity argument of $O(n^2)$. More details can be covered in section.

Does the ordering of the people matter?

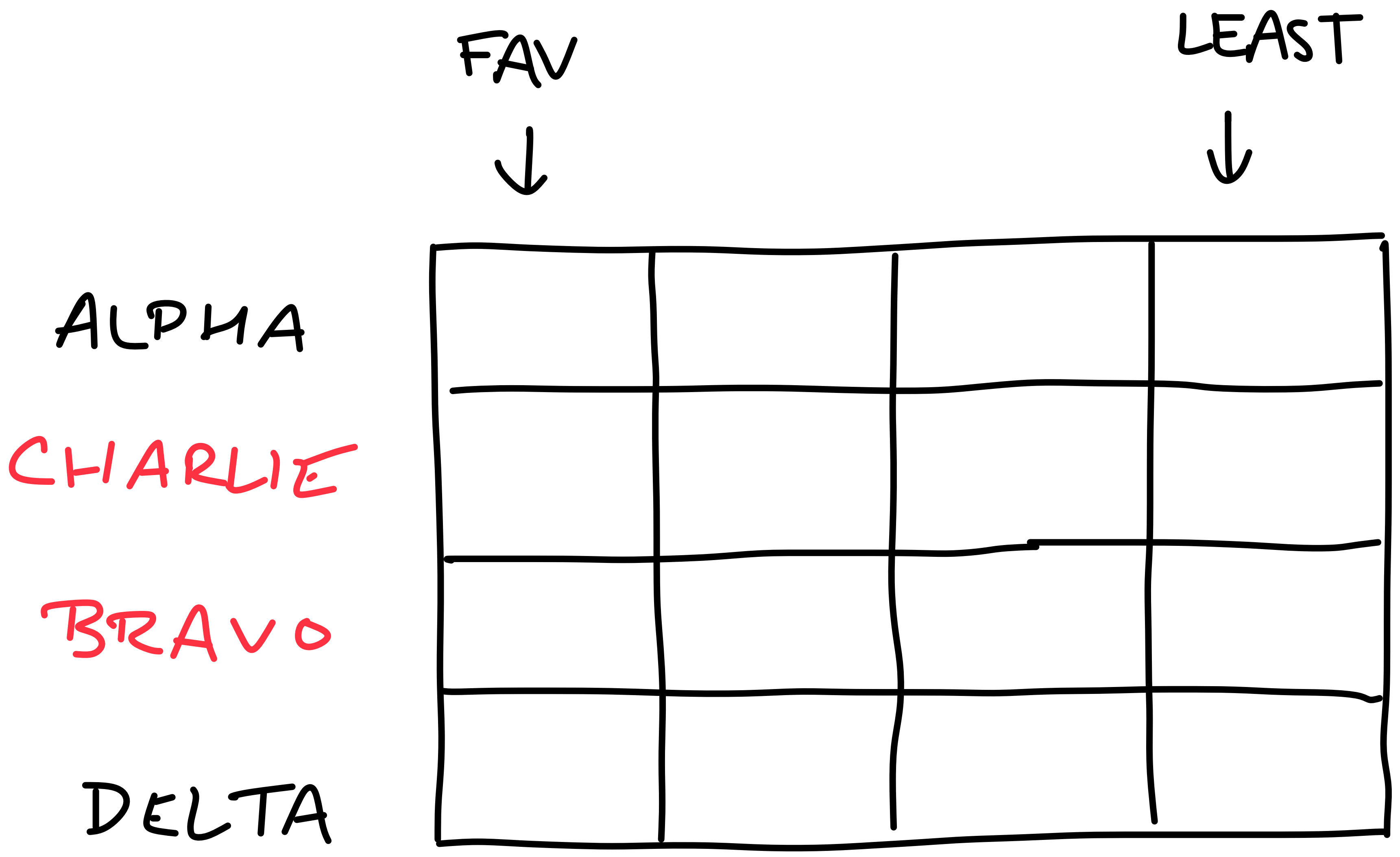
- We arbitrarily assigned the proposers and receivers indexes $1 \ldots n$.
- Would a different assignment have occurred under a different ordering?
- Multiple stable matchings can exist!

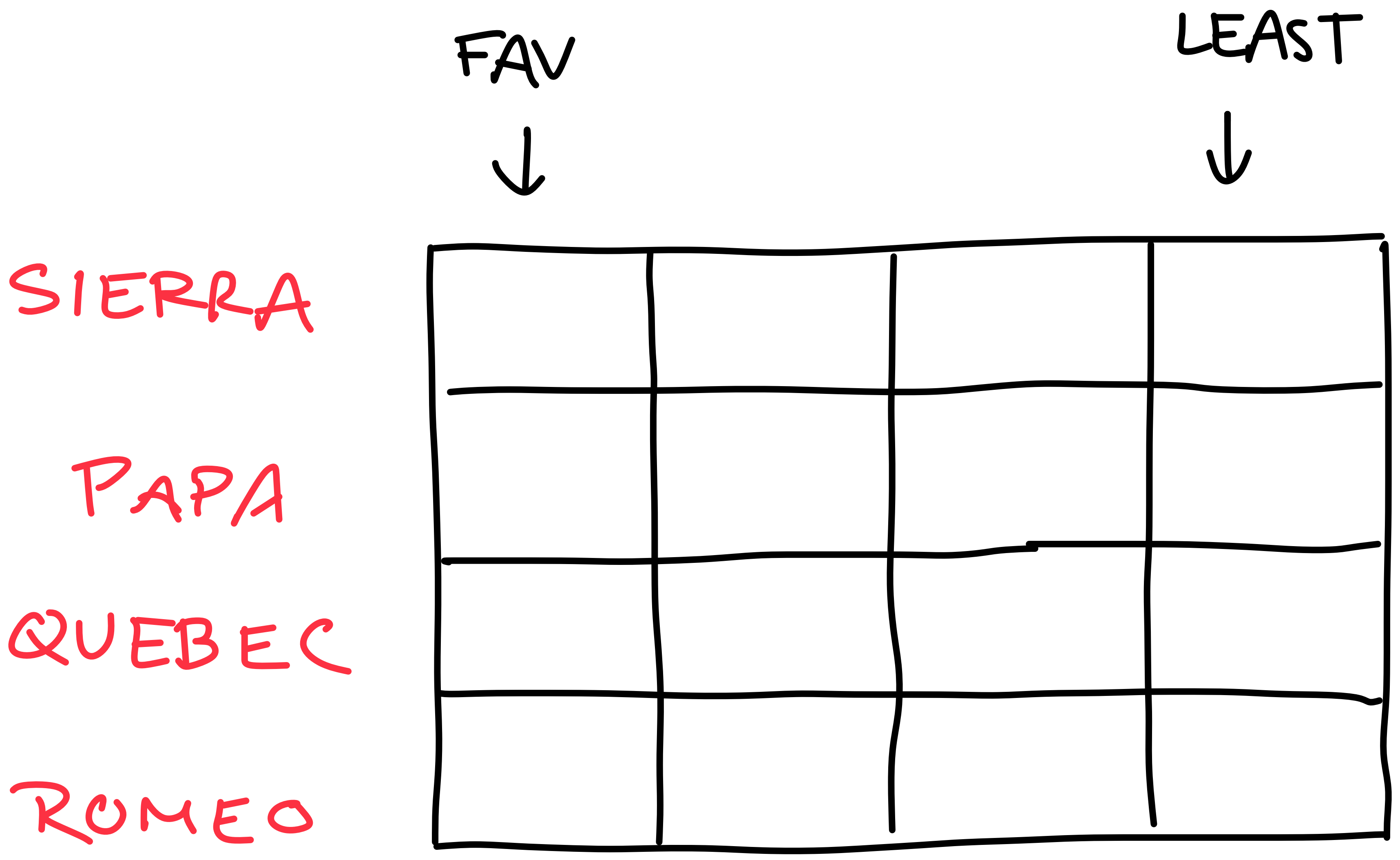




Does the ordering of the people matter?

- We arbitrarily assigned the proposers and receivers indexes $1 \ldots n$.
- Would a different assignment have occurred under a different ordering?
- Multiple stable matchings can exist!





It's good to be a proposer

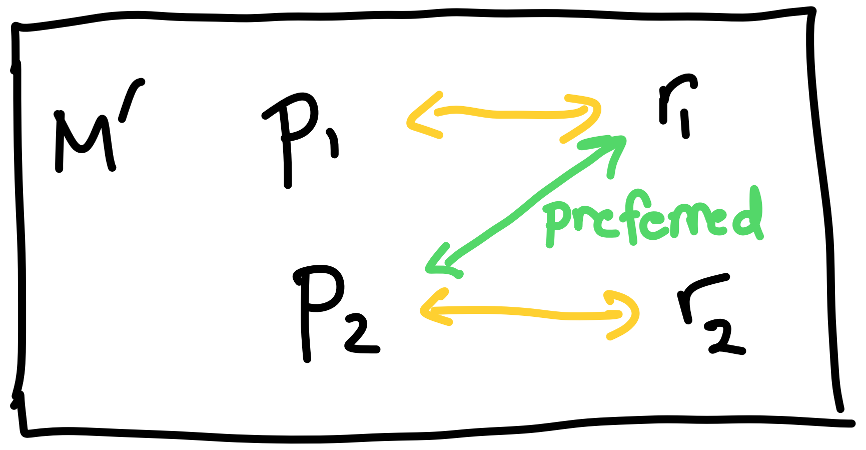
Proposer-optimality of Gale-Shapley

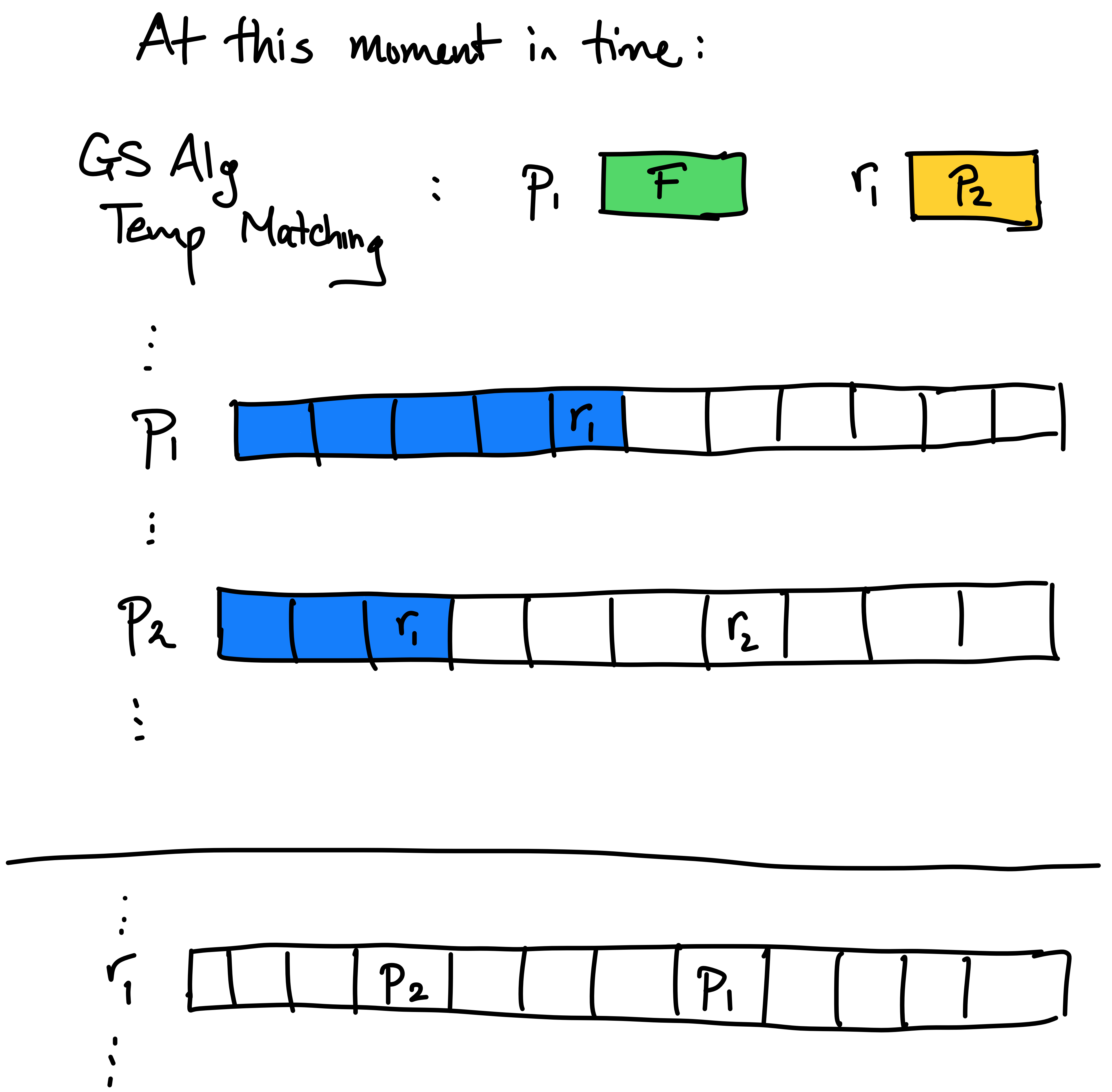
- **Proposer-optimal:** The proposer-optimal assignment is one in which *every* proposer p is matched with their best *valid* partner
- **Valid partnership:** p and r is a valid partnership if there exists some stable matching containing (p, r)
- **Lemma:** Gale-Shapley always produces a proposer-optimal stable matching.
  - **Corollary:** Gale-Shapley always produces the same assignment. I.e. ordering does not matter!
  - **Proof:** There is at most one proposer-optimal stable matching. Since Gale-Shapley always outputs a proposer-optimal stable matching, it always outputs the same assignment.

Proof of proposer-optimality

there is some stable matching M' containing (p_1, r_1).

- A proof by contradiction. Assume M is not proposer-optimal then there is some ***first time in running GS*** that a proposer p_1 is rejected by a valid partner r_1 since proposers propose in order of preference.
- Since r_1 rejected p_1, let p_2 be the partner r_1 prefers: either (p_2 was engaged to r_1) or (p_2 replaced p_1). And in M', let r_2 be the partner of p_2: valid partnership (p_2, r_2).
- Since r_1 rejecting p_1 is the ***first*** rejection by a valid partner, at that moment in the algorithm, r_2 cannot have rejected p_2. Only possibility, p_2 hasn't proposed to r_2 yet.
    - So p_2 prefers r_1 to r_2.
    - And, we said that r_1 prefers p_2 to p_1.
    - So (p_2, r_1) is unstable for M'. A contradiction to its stability of M'.





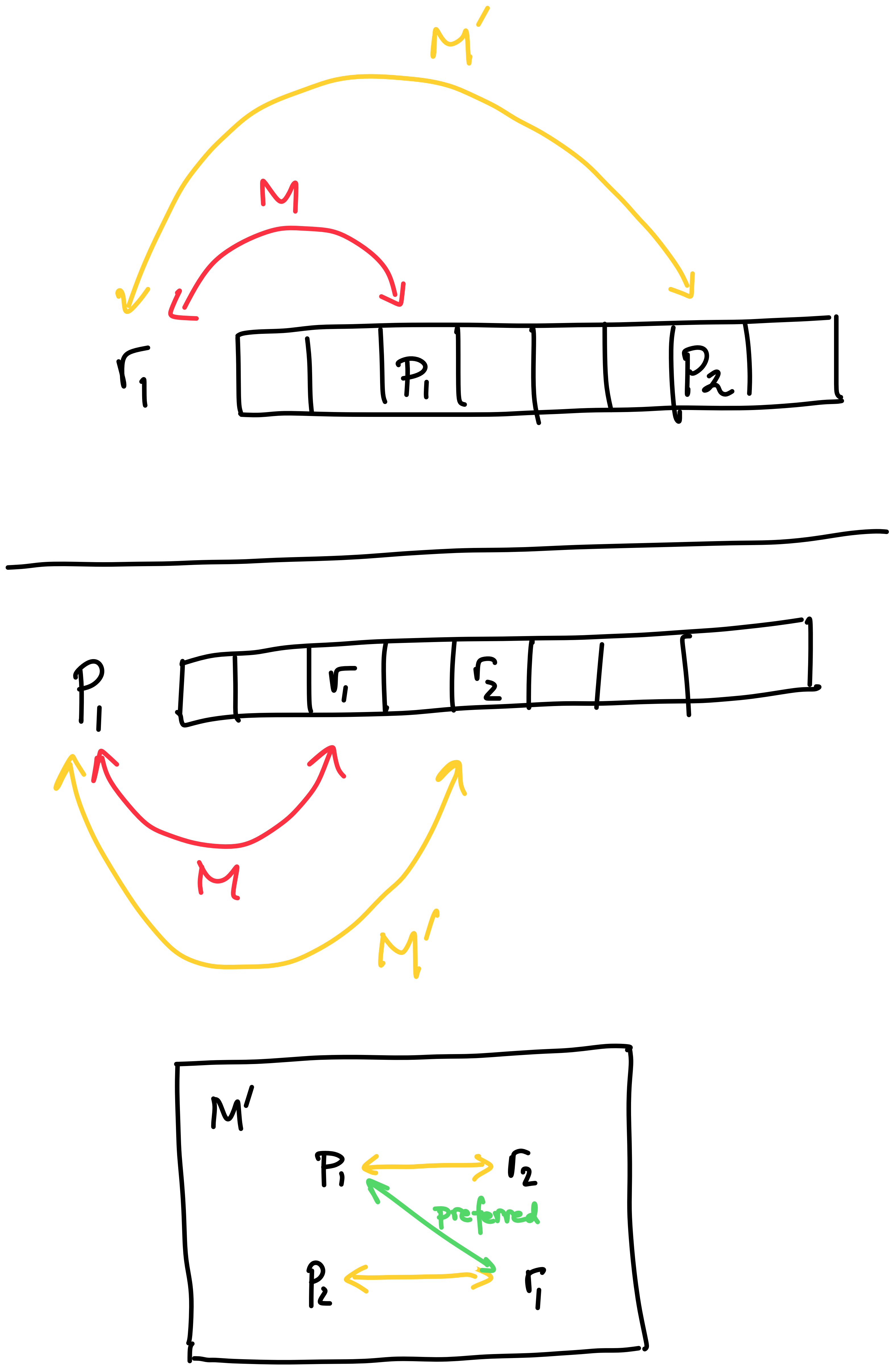
It's bad to be a receiver

Receiver-pessimality of Gale-Shapley

- **Receiver-pessimal:** The receiver-pessimal assignment is one in which *every* receiver r is matched with their worst *valid* partner
- **Valid partnership:** p and r is a valid partnership if there exists some stable matching containing (p, r)
- **Lemma:** Gale-Shapley always produces a receiver-pessimal stable matching.

Proof of receiver-pessimality

- A proof by contradiction. Assume M is not receiver-pessimal i.e. some receiver r_1 is matched to p_1 but p_1 is not the worst valid partner
  - There exists a M' stable matching in which r_1 is matched to p_2 but p_2 is lower ranked by r_1
  - Let r_2 be the match of p_1 in M'
- Proposer-optimality of M gives that p_1 prefers r_1 to r_2
  - (p_1, r_1) is unstable for M', a contradiction.



Natural extensions

Example: Matching residents to hospitals

- Original form: proposers are hospitals and receivers are med. school residents
- Variations that make the problem different:
  - Some participants could declare some partners as unacceptable. (Rank = ∞).
  - Unequal number of proposers and receivers.
  - Participants can participate in more than one matching.
  - A different notion of “stability”.
  - Residents may want to perform “couples matching”.
- Many natural variants turn out to be NP-complete! A topic we will discuss in depth later in the course.

Actual implementation

- NRMP (National Resident Matching Program)
  - 23,000+ residents legally bound by the outcome
  - Pre-1995 NRMP had the hospitals as proposers (recall, proposer optimality)
  - Post-1995 has the hospitals as receivers (recall, receiver pessimality)
- Rural hospital dilemma
  - How to get residents to unpopular (often rural hospitals)?
  - Rural hospitals were often undersubscribed in matchings.

Meta-lessons from stable matching

- To design and analyze algorithms, isolate the underlying structure of the problem.
- Algorithms can have deep social ramifications that need to be understood. Algorithm design can have unintended consequences.
- Technique for study algorithms: Find the first time the “bad event” might happen in the running of the algorithm and prove it doesn’t occur.
    - Variant of proof by contradiction.

Are you incentivized to lie?

- Should stable matching players lie about their preferences to get better outcomes?
  - By proposer optimality, a proposer has no incentive to lie.
  - Receivers are incentivized to lie.
- No mechanism can guarantee stable matchings and incentivize honesty. (Not proven in this class).

	1st	2nd	3rd
X	A	B	C
Y	B	A	C
Z	A	B	C

Group P Preference List

	1st	2nd	3rd
A	Y	X	Z
B	X	Y	Z
C	X	Y	Z

Group R True Preference List

	1st	2nd	3rd
A	Y	Z	X
B	X	Y	Z
C	X	Y	Z

A pretends to prefer Z to X

Algorithmic complexity

Measuring algorithmic efficiency
The RAM model

- RAM Model = “Random Access Machine” Model
- Each simple operation (arithmetic, evaluating if loop criteria, call, increment counter, etc.) takes one time step
- Accessing any one arithmetic number in memory takes one time step
- Measuring algorithm efficiency
    - Let input be $(x_1, \ldots, x_n)$ with each x_i representing one arithmetic number
    - Runtime of algorithm is the number of “simple operations” taken to compute algorithm in RAM model.

Complexity analysis

- Input $(x_1, \ldots, x_n)$ of length n.
- Multiple measures of complexity.
  - Worst-case: **maximum** # of steps taken on *any* input of length n
  - Best-case: **minimum** # of steps taken on *any* input of length n
  - Average-case: **average** # of steps taken over *all* input of length n

Complexity analysis

- The complexity of an alg. is a function $T(n)$ for each input size $n \in \mathbb{N}$.
- i.e. $T_{\text{worst}}(n)$ or $T_{\text{avg}}(n)$ could be two different functions.
- $T : \mathbb{N} \to \mathbb{N}$
- We are interested in understanding the overall behavior/shape of T, not the exact function.
- Sometimes there is more than one size parameter. $T(n, m)$ for a n vertex and m edge graph.

Polynomial time

A notion of efficiency

- A function $T(n)$ is **polynomial time** if $T(n) \leq cn^k + d$ for some constants $c, k, d > 0$.
  - Let k be the minimal such value. This is the degree of the *dominating* polynomial.
  - Polynomial time is known as “efficient” in theoretical CS.

Polynomial time

A notion of efficiency

- A function $T(n)$ is **polynomial time** if $T(n) \leq cn^k + d$.
- Why **polynomial time**?
  - Scaling the instance by a constant factor so does the runtime.
  - **Church-Turing thesis**: Any function computable in polynomial time by a physically realizable model of computation can also be computed in polynomial time a *different* physically realizable model.
    - I.e. polynomial-time is a notion independent of model of computation.
    - Ideal for theoretical study of what problems are efficient and which are not.
  - Problem size grows by constant, then running time also grows by constant.
  - If $T(n) = cn^k + d$ then $T(2n) = c(2n)^k + d \leq 2^k(cn^k + d) = 2^kT(n)$.
  - Typically, polynomials for common algorithms are small polynomials cn, cn^2, cn^3, cn^4. Rarely anything higher.

Big-O notation

Let $T, g : \mathbb{N} \rightarrow \mathbb{N}$. Then

- $T(n)$ is $O(g(n))$ if $\exists\ c, n_0 > 0$ such that $T(n) \leq cg(n)$ when $n \geq n_0$.
- $T(n)$ is $o(g(n))$ if $\lim_{n\rightarrow\infty} \frac{T(n)}{g(n)} = 0$.
- $T(n)$ is $\Omega(g(n))$ if $\exists\ \epsilon, n_0 > 0$ such that $T(n) \geq \epsilon g(n)$ when $n \geq n_0$.
- $T(n)$ is $\Theta(g(n))$ if $T(n)$ is $O(g(n))$ **and** $T(n)$ is $\Omega(g(n))$.