Comparison Query Model

- Input is an array X of size n, indexed from 0 to n-1
- Assume all array elements are distinct.
- The algorithm can choose two indices (i,j), and ask
  "is X[i] < X[j]?", and will get a true/false answer.
  This is called a comparison. Each comparison has cost 1.
- The input array can be modified by swapping elements. Swaps have cost 0.
- All operations that don't involve the input array have cost 0.

Lecture Outline

1. Model: Comparison Queries
2. **Review: Find the maximum element**
3. Problem: Find the 2nd largest element
4. Problem: Find maximum and minimum element
5. Problem: Sorting an array
6. Technique: Decision Trees

Problem: Find the Maximum

- We consider the problem of finding the maximum in an array of n distinct integers.
- The task: output the index where the maximum element can be found
- The "Scan Algorithm" with cost n-1:

  Idea: keep track of the index of the largest element seen so far, call it maxIndex.

  Initialize maxIndex = 0.

  For i = 1,…,n-1, perform comparison (i,maxIndex),

  If X[i] > X[maxIndex], set maxIndex = i.

  Output maxIndex

Winners vs. Losers

- For every comparison made, we call the larger element the “winner” and the smaller element the “loser”.
- We can show that every non-maximum element must “lose” at least one comparison. Let’s call this the “Loser Lemma”.

Proof of the “Loser Lemma”

(see previous lecture)

The Lower Bound for Finding Max

- We proved the "Loser Lemma": every non-maximum element must lose at least once.
- There are n-1 non-maximum elements, and each comparison has exactly one loser. This means there must be at least n-1 comparisons.

Lecture Outline

1. Model: Comparison Queries
2. Review: Find the maximum element
3. **Problem: Find the 2nd largest element**
4. Problem: Find maximum and minimum element
5. Problem: Sorting an array
6. Technique: Decision Trees

Problem: Find the 2nd largest element

- Task: output the index where the 2nd largest element can be found.
- First, let's think about some algorithms.
- The most obvious is just mergesort the array in ascending order and output n-2.

  This has cost O(n log n).
- Next obvious idea is to:

  1. Find the maximum, put it at X[n-1],
  2. Then find the maximum among X[0..n-2].

  - This has cost (n-1)+(n-2) = 2n-3 using the "Scan Algorithm" from earlier.
- Can you think of something more clever? or is 2n-3 the optimal cost?

What next?

- This happens often in Algorithms research:
  - We have a decent algorithm which gives an upper bound on complexity. We can't think of anything better, we ask "is this the best?"
  - So we try to prove a lower bound that matches the cost of our algorithm. If it matches, then we know we can stop looking for a better algorithm.
  - If it doesn't match, maybe the insight we gain from the lower bound proof will help us find a better algorithm!

A Lower Bound

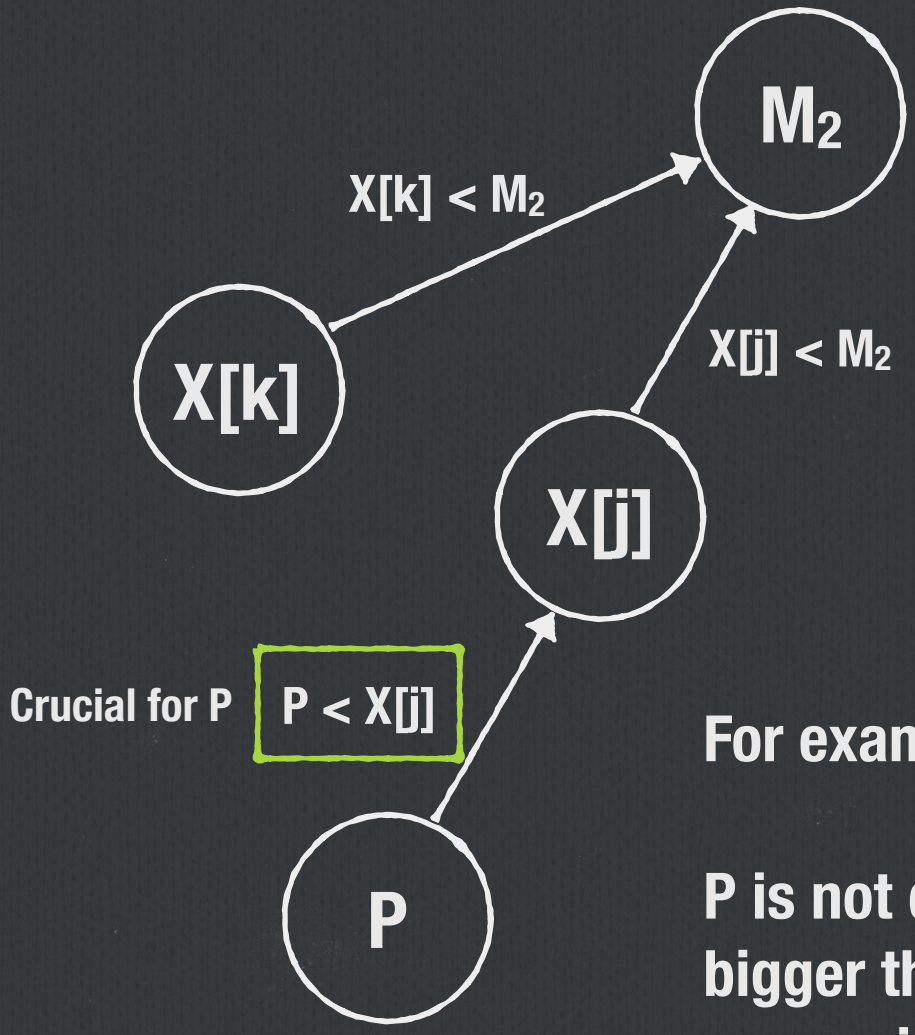
- First, observe that the proof of the Loser Lemma also applies for this problem is applicable here.
  (You should read it again later to verify this)
  - This immediately implies a lower bound of n-1 comparisons.
- But this isn't good enough, we are trying to show that 2n-3 comparisons are needed.

A Lower Bound

- Here's an overview of a better idea. Let M_1 be the max, let M_2 be 2nd largest.
  - A certain number of comparisons are needed to find M_1. We can't be sure of the 2nd largest if we don't know the largest.
  - But finding the maximum isn't enough.
    Of all the remaining non-maximum elements, we need to use some comparisons to figure out the maximum among them, which is M_2.
  - The key will be to find lower bounds for these two cases separately and add them together, but making sure that none of the comparisons for one case are useful in the other case (i.e., no double-counting)

"Crucial" Comparisons

- Let P be any element that is smaller than M_2.
- During an execution of an algorithm A, we say that P is dominated by M_2 if P ever loses to M_2, or if P ever loses to an element Q and Q is dominated by M_2.
- Essentially, it means that there is a chain of losses that we can draw with P below M_2.
- P's loss in this chain below M_2 is called a crucial comparison for P.



Start with a node for each element.
For each comparison performed, draw an arrow from loser to winner.

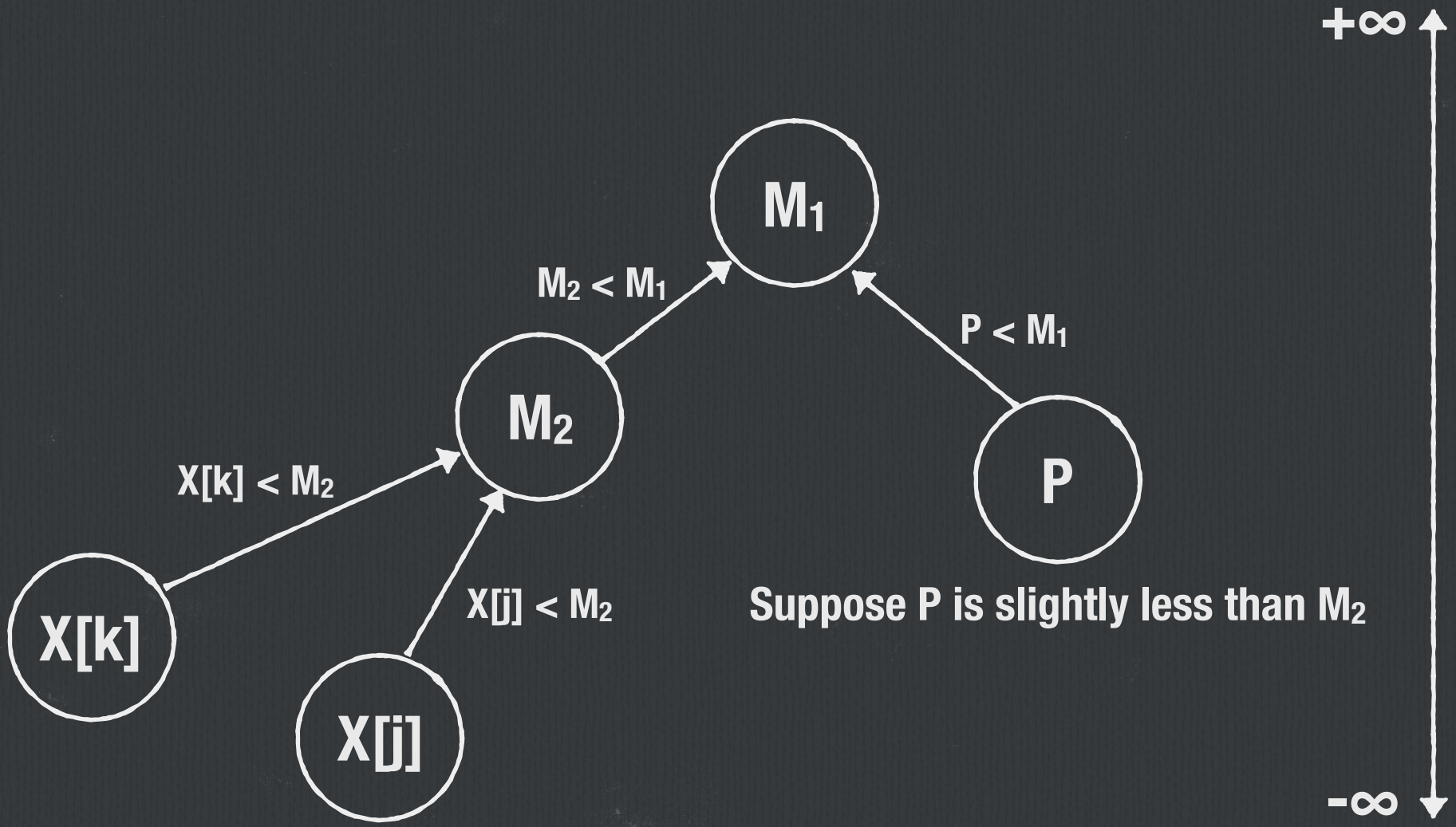
For example, P is dominated by X[j] and M_2.

P is not dominated by X[k], even if X[k] is bigger than P, because there is no chain of comparisons with P below X[j].

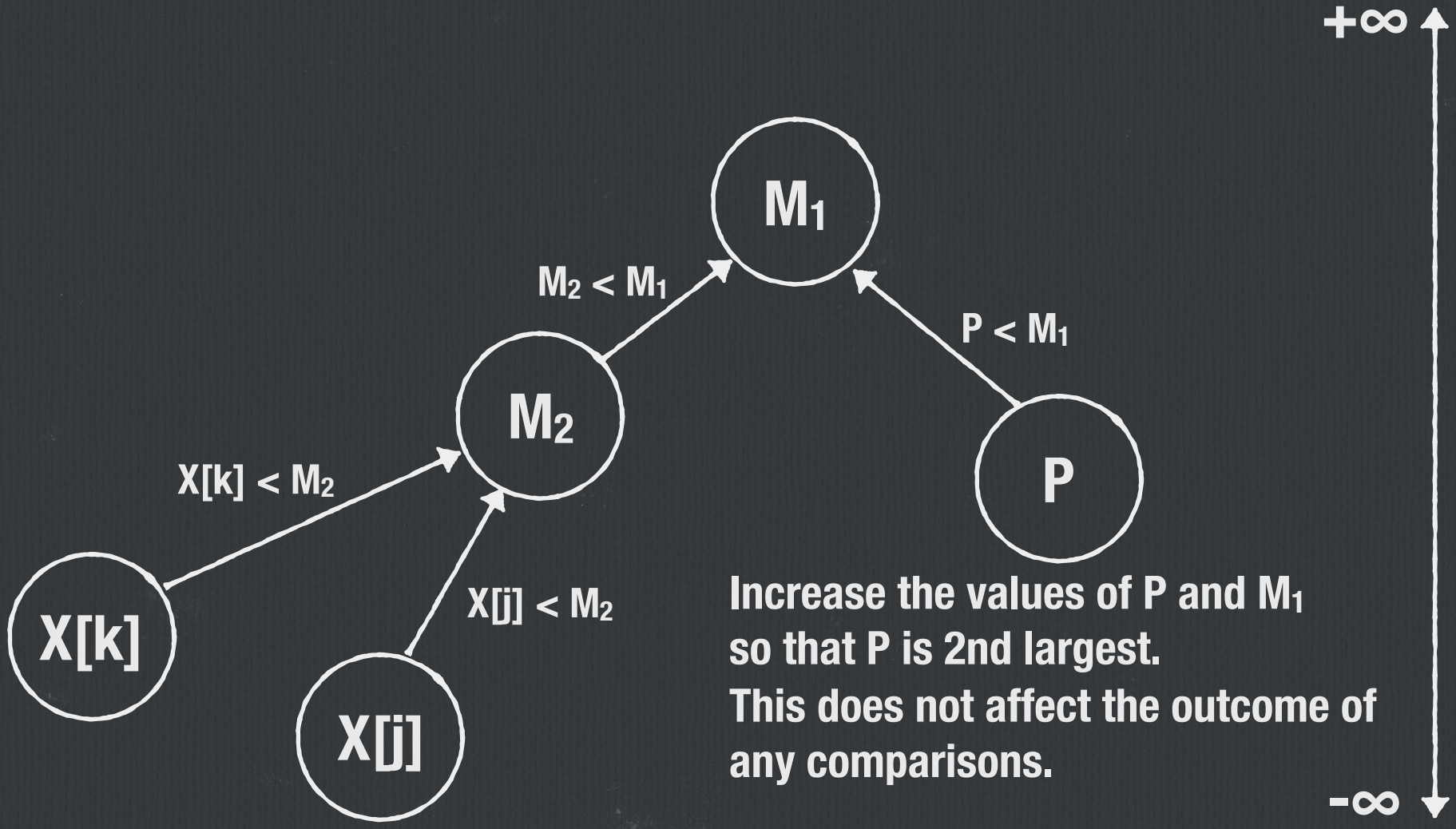
Crucial Comparisons are Crucial

- Claim: for every element P that is less than M_2, P is involved in at least one crucial comparison.
- Since there are n-2 elements that are less than M_2, it follows from the above Claim that there are at least n-2 crucial comparisons in every execution of a correct algorithm.
- We will not prove the Claim today. Perhaps on an assignment…
- This is the end of part 1. Essentially these crucial comparisons are necessary to find M_2.

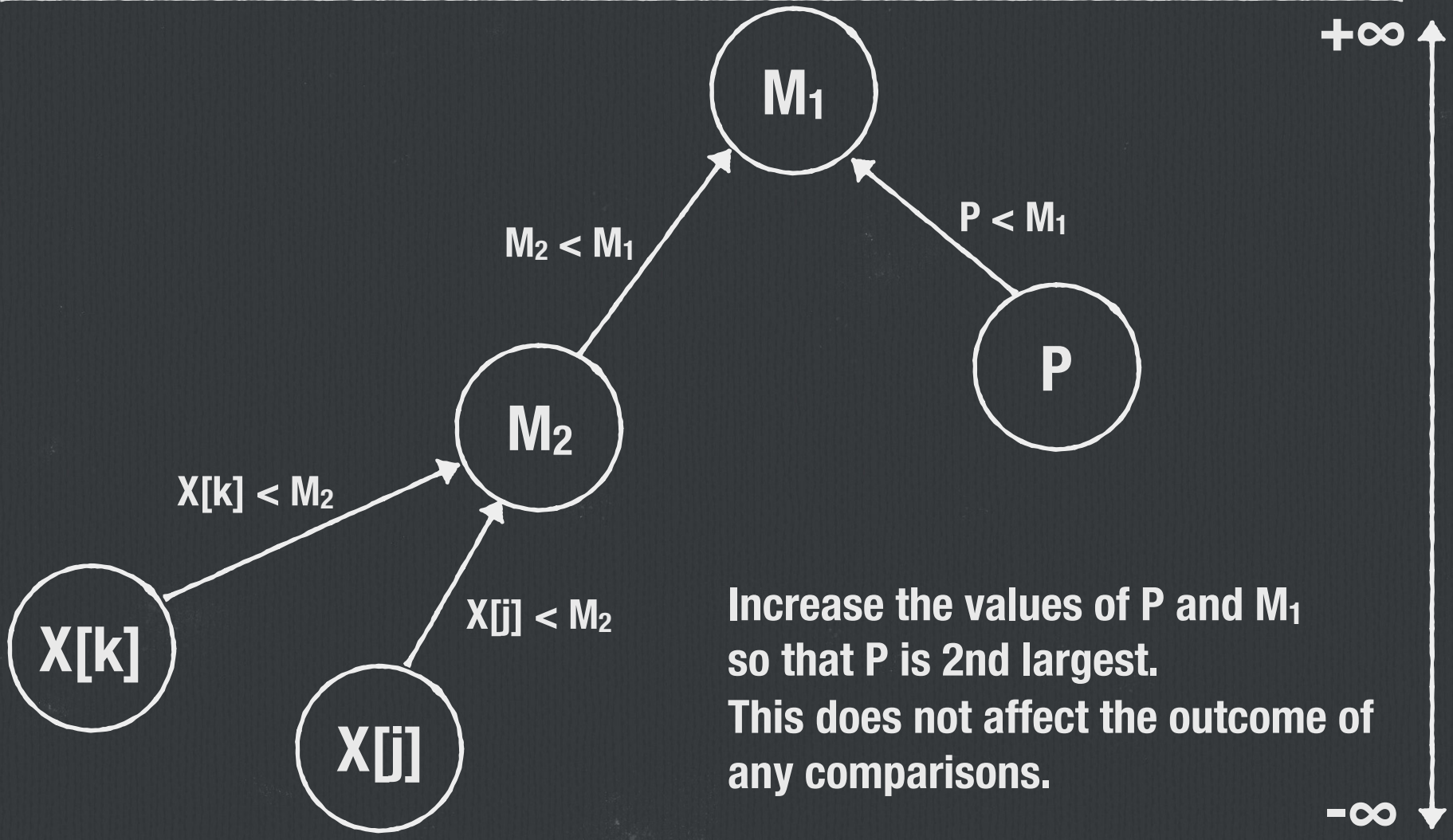
Informal Diagram of Claim



Informal Diagram of Claim



Informal Diagram of Claim



Increase the values of P and M_1 so that P is 2nd largest.
This does not affect the outcome of any comparisons.

Now find M_1

- The next idea is to show that a certain number of comparisons must involve M_1, and add them to the number of crucial comparisons.
- Why can we do this? Are we sure there is no double-counting?
  - No comparisons with M_1 are crucial, so we can be sure we haven't counted them yet.

    By definition, a crucial comparison either involves a loss to M_2, or a loss to an element that is smaller than M_2.

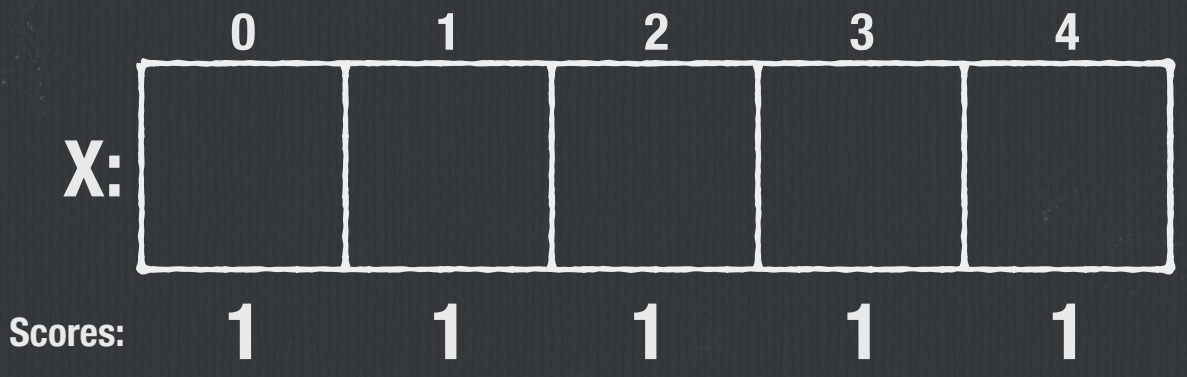
    A comparison involving M_1 is neither of these.

An adversary

- We'll use an adversary argument to force lots of comparisons involving M_1.
- The adversary is going to associate a "score" with each position in the array.
- Remember, the array elements don't have values yet, the adversary is making things up as it goes along.
- Initially, each position has a score of 1.
- When the algorithm performs a comparison (i,j), the adversary has to decide whether i or j wins (i.e., whether X[i] > X[j] or X[i] < X[j])

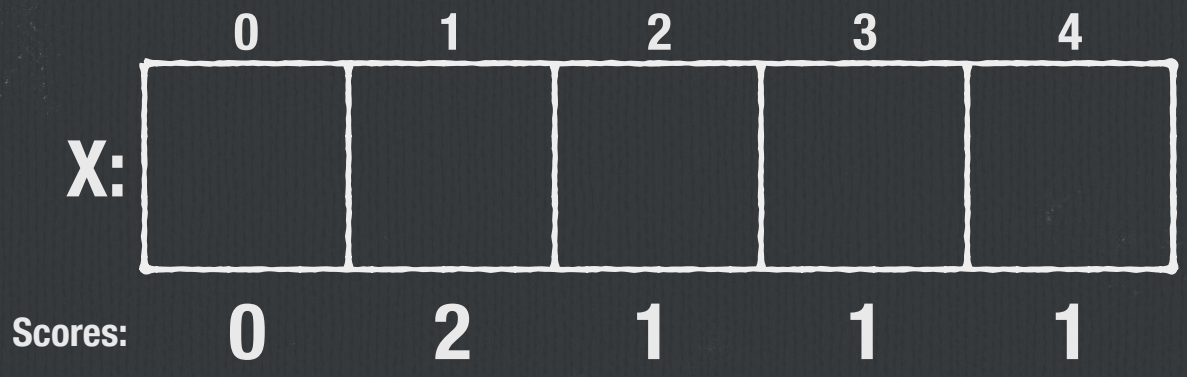
An adversary

- The adversary definition:
  - Choose as winner the position with the higher score (ties go to larger index, except 0-0 comparisons that must be answered consistently as before).
  - Then reduce the loser's score down to 0, and transfer all of the loser's points to the winner's score.
  - Tell the algorithm who the winner was.
- Note: any position that has a non-zero score has not lost yet.



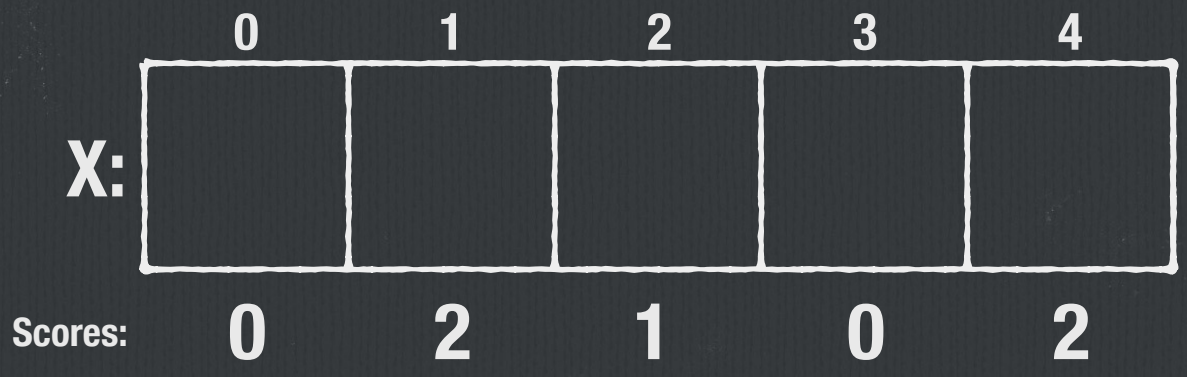
Algorithm's comparisons:

(0,1)
(3,4)
(2,3)
(1,4)
(1,3)
(2,4)



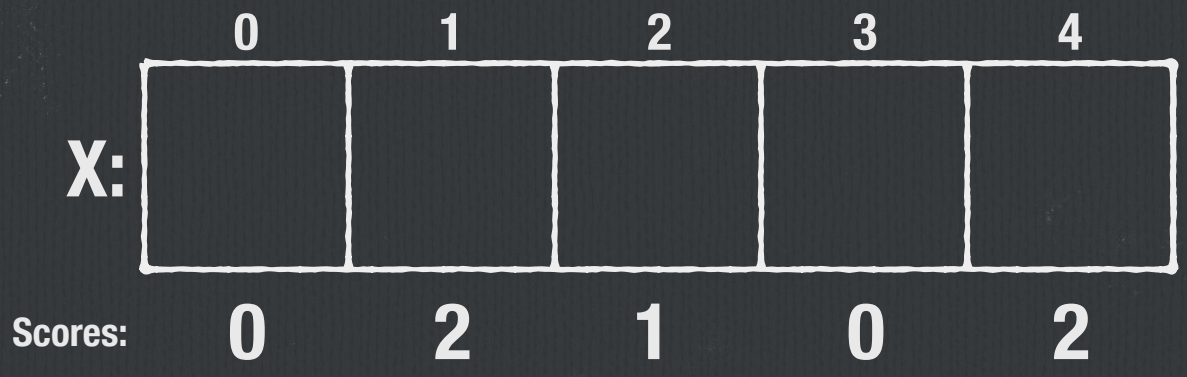
Algorithm's comparisons:

(0,1)
(3,4)
(2,3)
(1,4)
(1,3)
(2,4)



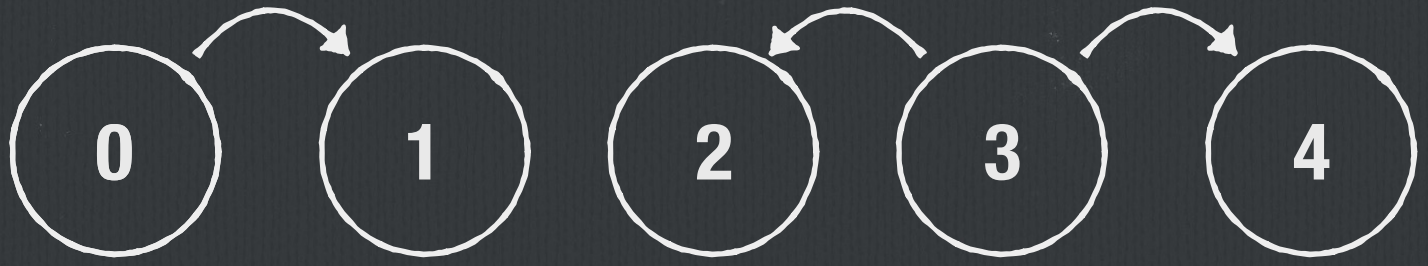
Algorithm's comparisons:

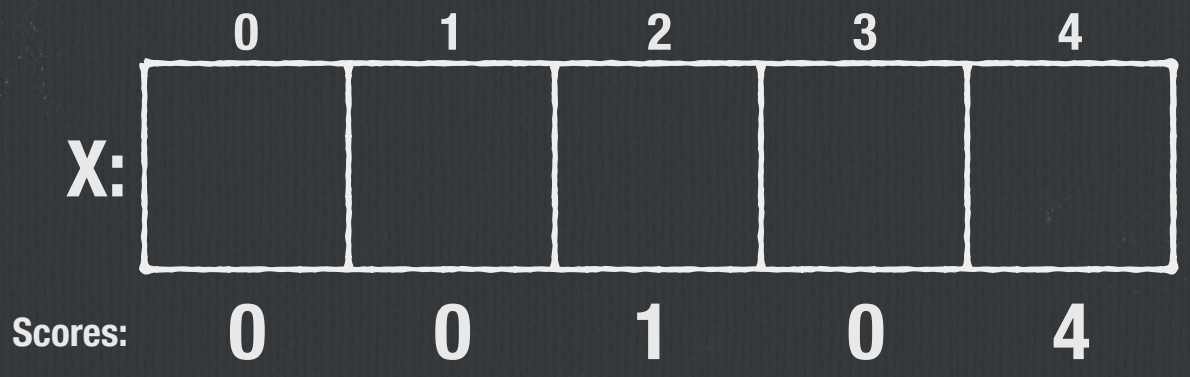
(0,1)
(3,4)
(2,3)
(1,4)
(1,3)
(2,4)



Algorithm's comparisons:

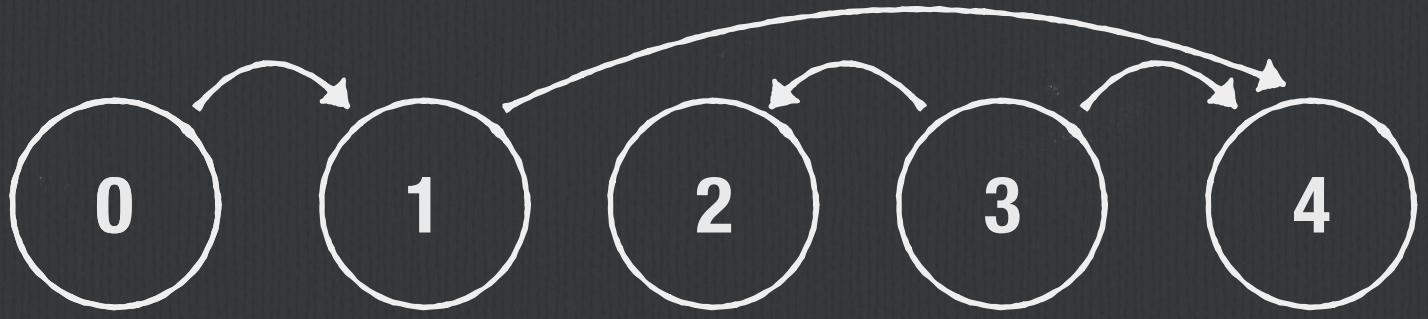
(0,1)
(3,4)
(2,3)
(1,4)
(1,3)
(2,4)

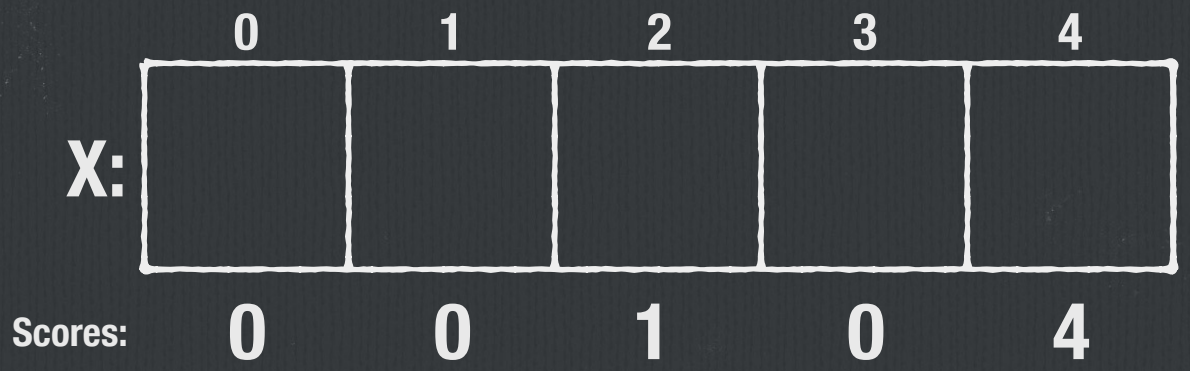




Algorithm's comparisons:

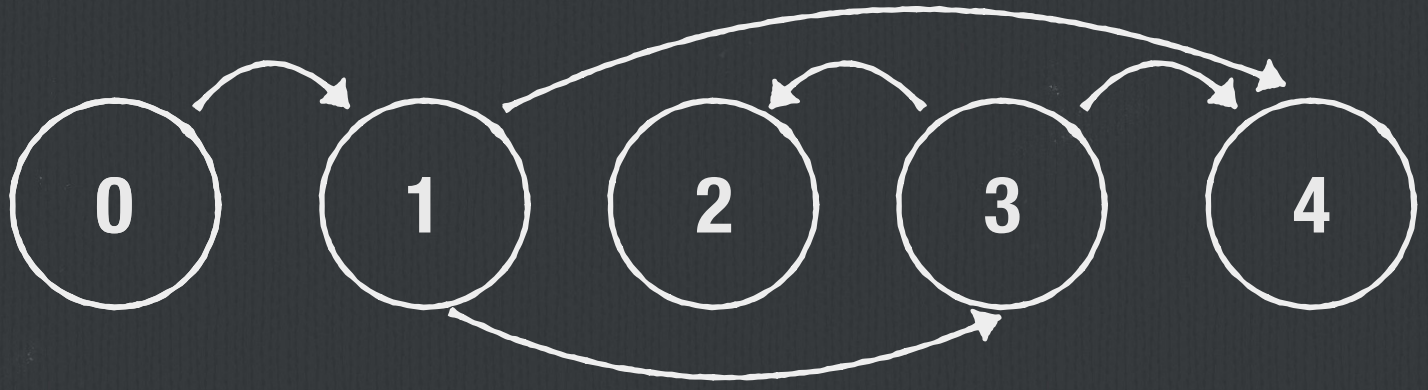
(0,1)
(3,4)
(2,3)
(1,4)
(1,3)
(2,4)

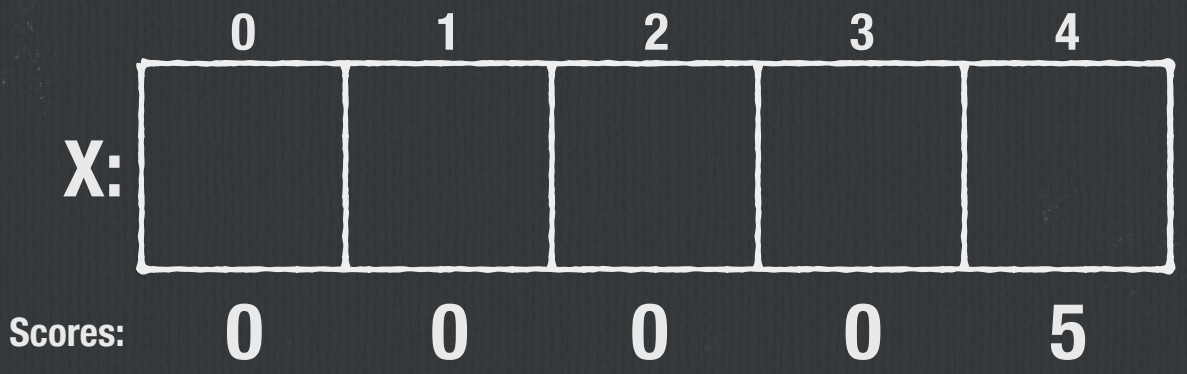




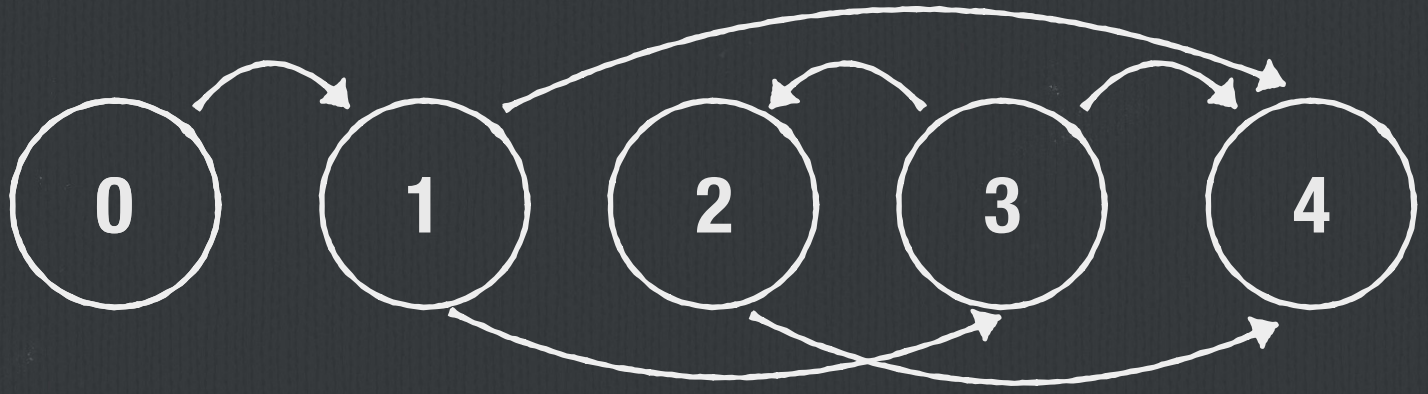
Algorithm's comparisons:

(0,1)
(3,4)
(2,3)
(1,4)
(1,3)
(2,4)





Algorithm's comparisons:
(0,1)
(3,4)
(2,3)
(1,4)
(1,3)
(2,4)

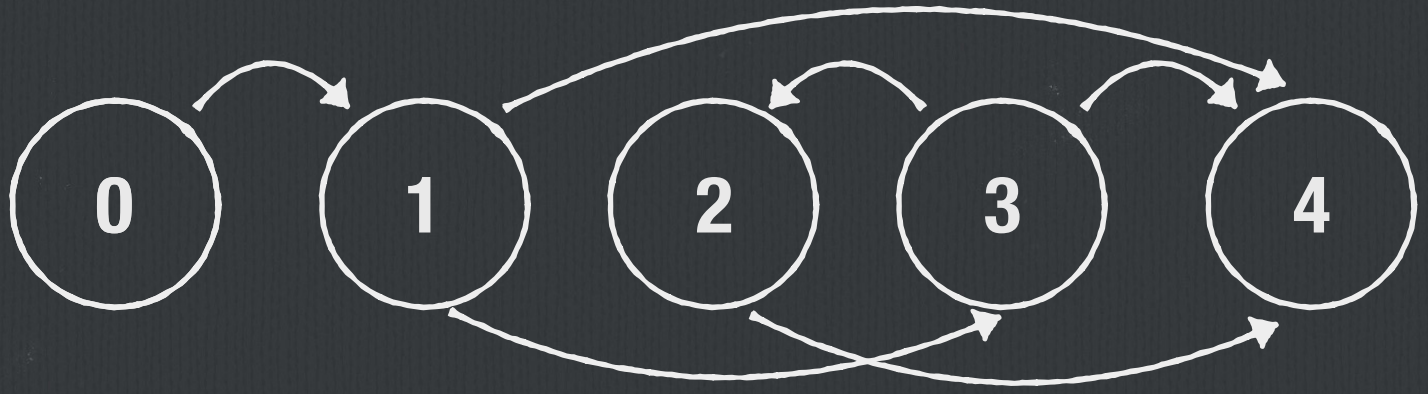


We need this diagram for two reasons:

1. If the algorithm then performs a comparison like (0,3), the adversary needs to answer consistently with previous answers, e.g., 0 < 3
2. At the end, we need to provide actual array values that are consistent with all answers. Use the diagram as constraints.

	0	1	2	3	4
X:	10	20	40	30	50
Scores:	0	0	0	0	5

Algorithm's comparisons:
(0,1)
(3,4)
(2,3)
(1,4)
(1,3)
(2,4)



We need this diagram for two reasons:

1. If the algorithm then performs a comparison like (0,3), the adversary needs to answer consistently with previous answers, e.g., 0 < 3
2. At the end, we need to provide actual array values that are consistent with all answers. Use the diagram as constraints.

Why does this work?

- Notice: As long as there is more than one non-zero score, there are at least two non-losers, and one of them is not the maximum.
  - But then, the "Loser Lemma" proves that the algorithm cannot accurately output the 2nd largest element.
- So when the algorithm terminates, there must be exactly one position with a non-zero score, and this score will equal n (since that's how many points were assigned at the beginning). This position must contain M_1.

Why does this work?

- But, how quickly can a position's score go up?
- Looking at any single comparison, the winner's score at most doubles.
  - This is because the loser's score before the comparison is at most the winner's.
- So the minimum number of comparisons involving M_1 is at least the number of times M_1's score must double-up to go from 1 to n.
  - This is exactly $\lceil \log_2(n) \rceil$

Putting it together

- So we showed:
  - n-2 crucial comparisons are needed
  - $\lceil \log_2(n) \rceil$ comparisons involving M_1 are needed, which aren't crucial
- So we have a lower bound of cost $n-2+\lceil \log_2(n) \rceil$

That isn't 2n-3…

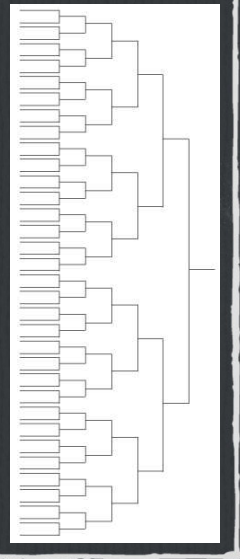
- … but maybe 2n-3 isn't optimal!
- Let's think: we know that each non-maximum element must lose at least once.
- Any element that loses to M_1 could be the 2nd largest.

  But any element that loses to any other element <u>can't</u> be the 2nd largest.
- So let's try to minimize the number of elements that lose to M_1.
  - Why? Because these are the only candidates for 2nd largest.

    So if there aren't many, we can run the simple/inefficient algorithm to find the largest among them.
- Looking at the second part of the lower bound proof, we found the minimum number of comparisons involving M_1 is at least $\lceil \log_2(n) \rceil$. Can we match that?

Use the same scoring idea to find a good algorithm!

- The Tournament:
  1. Initially assign to each position a score of 1.
  2. Keep comparing elements that have equal score.
  3. After each comparison, transfer all the points from the loser to the winner.
- If you want to visualize it: this is just a binary tree with all competitors starting at the leaves, winners move up until one arrives at the root.
  (If n isn't a power of 2, things are a little less clean, but not a huge deal)
- There is exactly one element that will win all of its comparisons: M_1.
  Its score will double with every win, until it has score n, so there are $\lceil \log_2(n) \rceil$ comparisons involving M_1



The Algorithm

1. Run the binary tree tournament (as described on previous slide)
2. Take the $\lceil \log_2(n) \rceil$ elements that lost to M_1 and find the maximum among them. This is the 2nd largest element.

- Cost is $n-2+\lceil \log_2(n) \rceil$ (which matches the lower bound):
  - n-1 comparisons to run the tournament (since there are n-1 internal nodes of the tournament tree)
  - $\lceil \log_2(n) \rceil - 1$ comparisons to find the largest among $\lceil \log_2(n) \rceil$ elements (using the "Scan Algorithm" from earlier)

Lecture Outline

1. Model: Comparison Queries
2. Review: Find the maximum element
3. Problem: Find the 2nd largest element
4. **Problem: Find maximum and minimum element**
5. Problem: Sorting an array
6. Technique: Decision Trees

Problem: Find the maximum and minimum element

- Task: output the two indices where the maximum and minimum can be found.
- First, let's think about some algorithms.
- The most obvious is just mergesort the array in ascending order and output 0 and n-1. This has cost O(n log n).
- Next obvious idea is to
  1. Find the maximum, put it at X[n-1]
  2. Then find the minimum among X[0..n-2].
  - This has cost (n-1)+(n-2) = 2n-3 using the "Scan Algorithm" (and a simple variant for minimum) from earlier.
- Can you think of something more clever? or is 2n-3 the optimal cost?

Lower Bound Idea

- As before, it's not obvious how to do better, so let's see if we can find a matching lower bound.
- Let's think about winners and losers again.
- Initially, no element has won or lost, so let's put them in a set called "Unknowns".
- The Unknowns set will be the elements that the algorithm knows nothing about. Any of them could be the maximum or minimum.

Lower Bound Idea

- Once an element e is involved in a comparison, the algorithm has gained some information about it.
  - if e was a winner, it's possible to conclude that e can't be the minimum
  - if e was a loser, it's possible to conclude that e can't be the maximum
  - We can't assume that the algorithm will come to these conclusions! But they are motivating the idea behind the proof.
- So let's make two more sets:
  1. Winners: elements that have won at least once, and have never lost
  2. Losers: elements that have lost at least once, and never won

Lower Bound Idea

- What about elements that have lost and won? They are pretty useless, and a smart algorithm might conclude that, so let's put them in a set called Useless.

- To recap, we have Unknowns, Winners, Losers, Useless. Looking at their definitions, we see that each element must belong to exactly one of the four sets at any given time.

Lower Bound Idea

- Let's make some observations about what an algorithm must accomplish:
  - At termination, the Unknowns set must be empty. Why?
    - This is because an element that hasn't been in a comparison could be the maximum or minimum. (Use an indistinguishability proof)
  - At termination, the Winners and Losers sets must have exactly one element each. Why?
    - Again, by indistinguishability, if there are two elements that never lost, either can be the maximum. Similarly, either of two elements that never won could be the minimum.
- So we define an adversary to slow down progress towards these accomplishments.

Definition of the Adversary

- The algorithm chooses a comparison (i,j), and the adversary looks at which sets contains them.
- If exactly one element is in Winners, then make that element the winner of the comparison.
- If exactly one element is in Losers, then make that element the loser of the comparison.
- (These two cases prevent elements from leaving the Winners and Losers sets, which slows down progress.)

Definition of the Adversary

- When both elements are in Unknowns, both in Winners, or both in Losers: it doesn't matter what we say, just answer X[i] > X[j]
- However, when an element moves from Losers to Useless, assign it the largest negative integer that hasn't been assigned yet.
- When an element moves from Winners to Useless, assign it the smallest positive integer that hasn't been assigned yet.
- When both elements are in Useless, answer consistently using the integers we assigned when they were moved to Useless.

Analysis

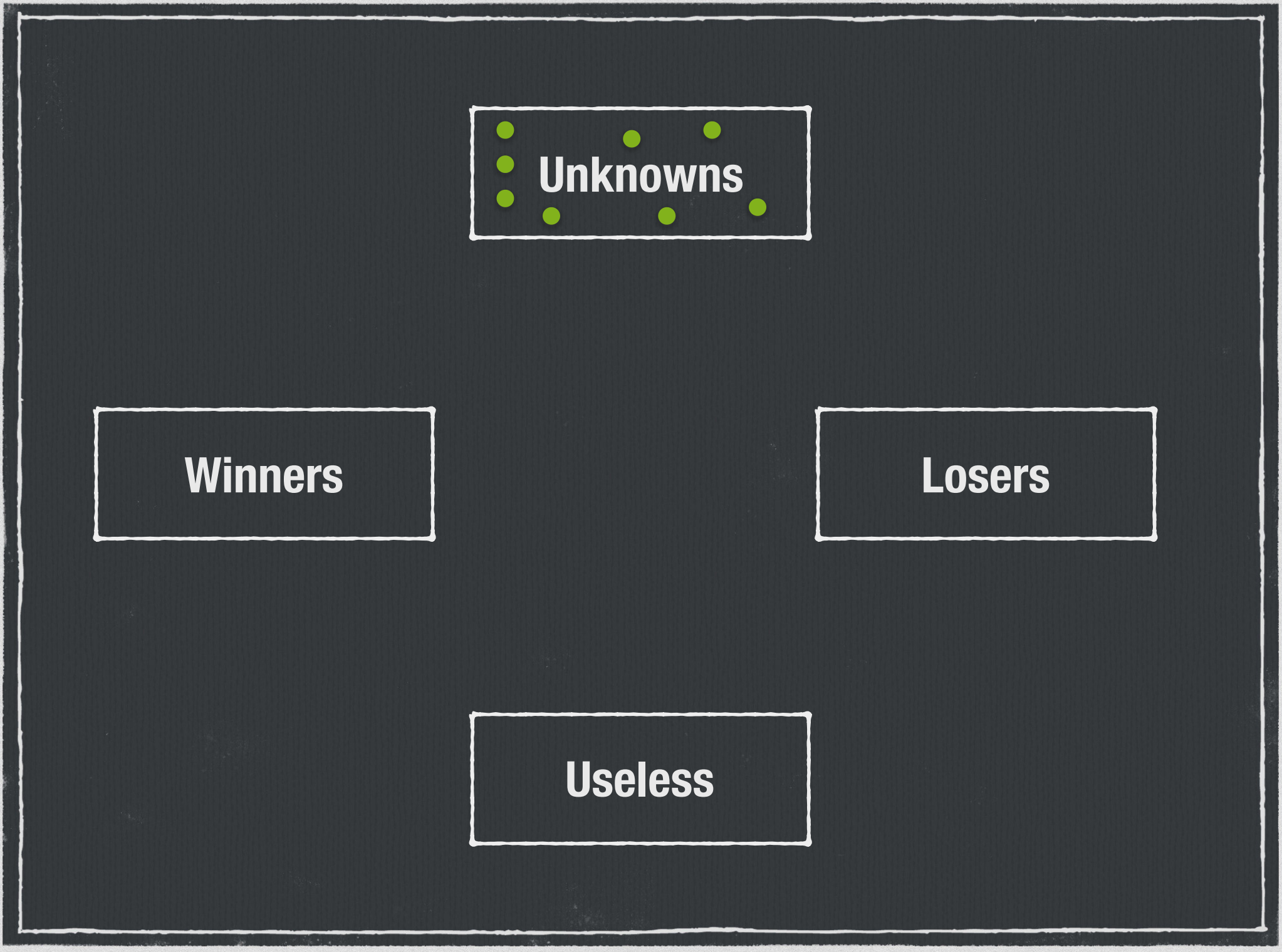
- So we've defined an adversary on the previous two slides. What remains is to analyze how many comparisons it forces the algorithm to perform.
- Initially, we have the following set sizes:
  - Unknowns = n
  - Winners = Losers = Useless = 0
- From our earlier observation, when the algorithm terminates:
  - Unknowns = 0
  - Winners = 1, Losers = 1
  - Useless = n-2

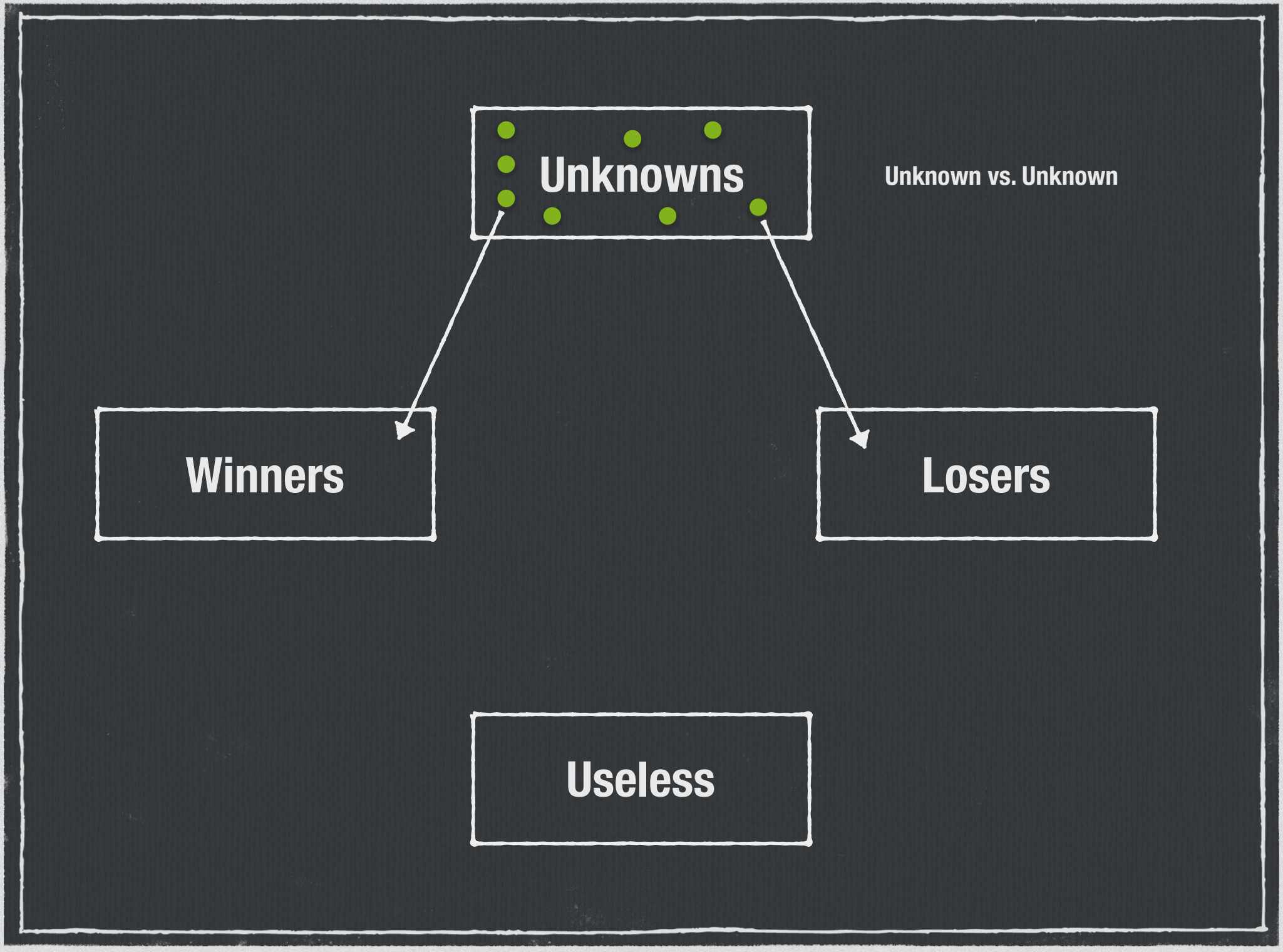
Analysis

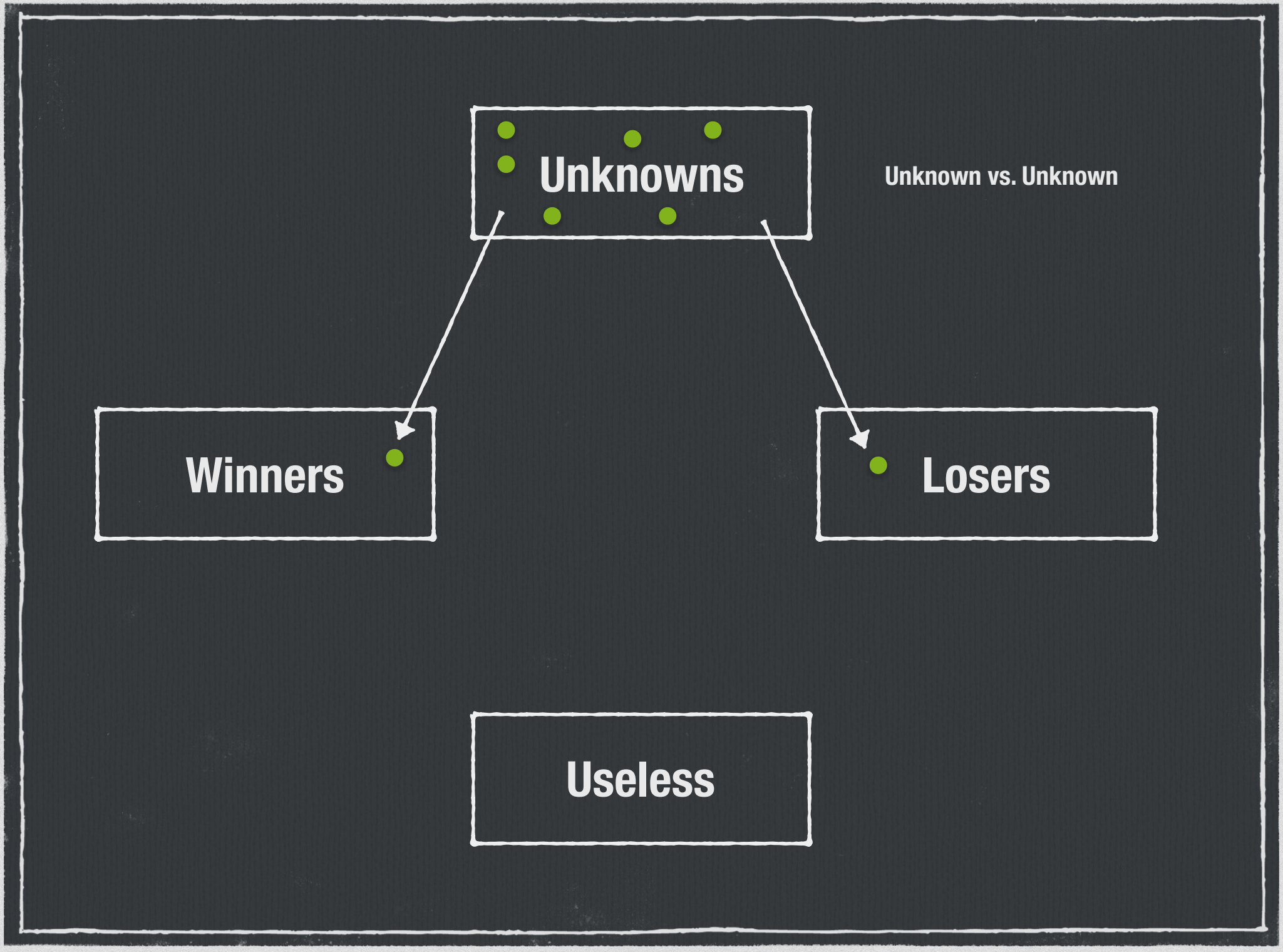
- Initially, we have the following set sizes:
  - Unknowns = n
  - Winners = Losers = Useless = 0
- From our earlier observation, when the algorithm terminates:
  - Unknowns = 0
  - Winners = 1, Losers = 1
  - Useless = n-2
- Notice: an element e cannot move directly from Unknowns to Useless.

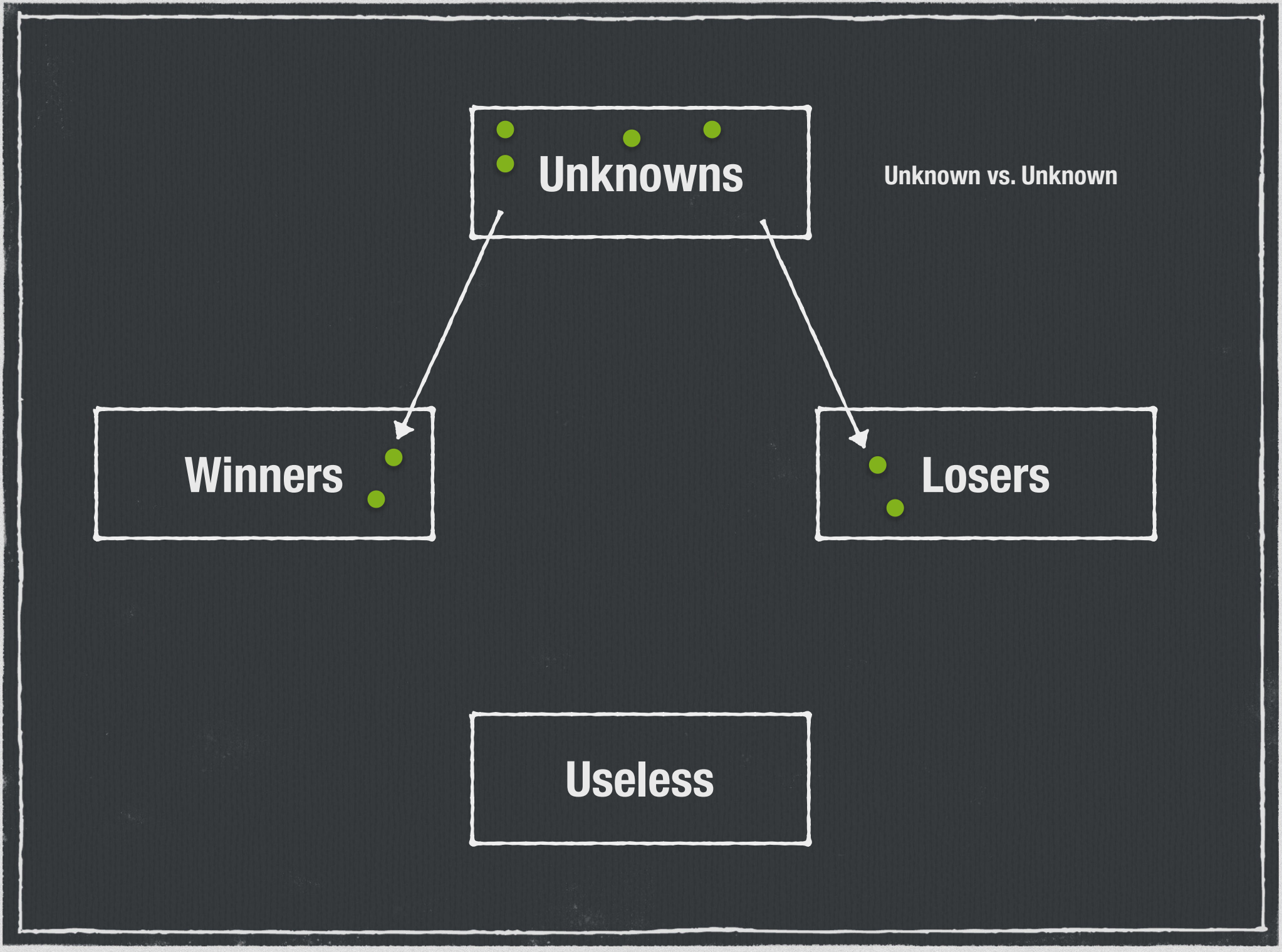
  The first comparison involving e moves it to Winners or Losers.
- So each element goes from

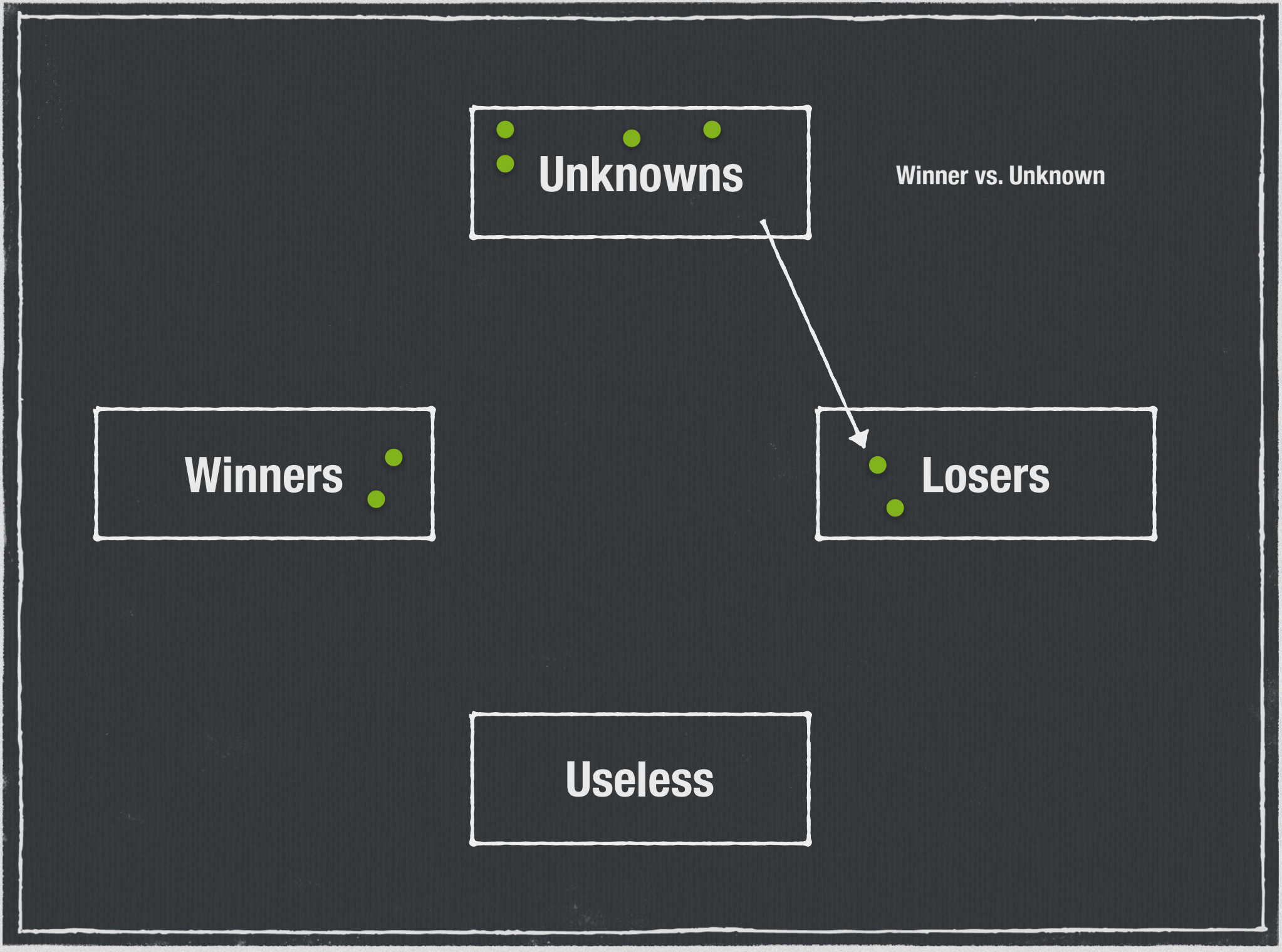
  Unknowns $\longrightarrow$ Winners $\cup$ Losers $\longrightarrow$ Useless

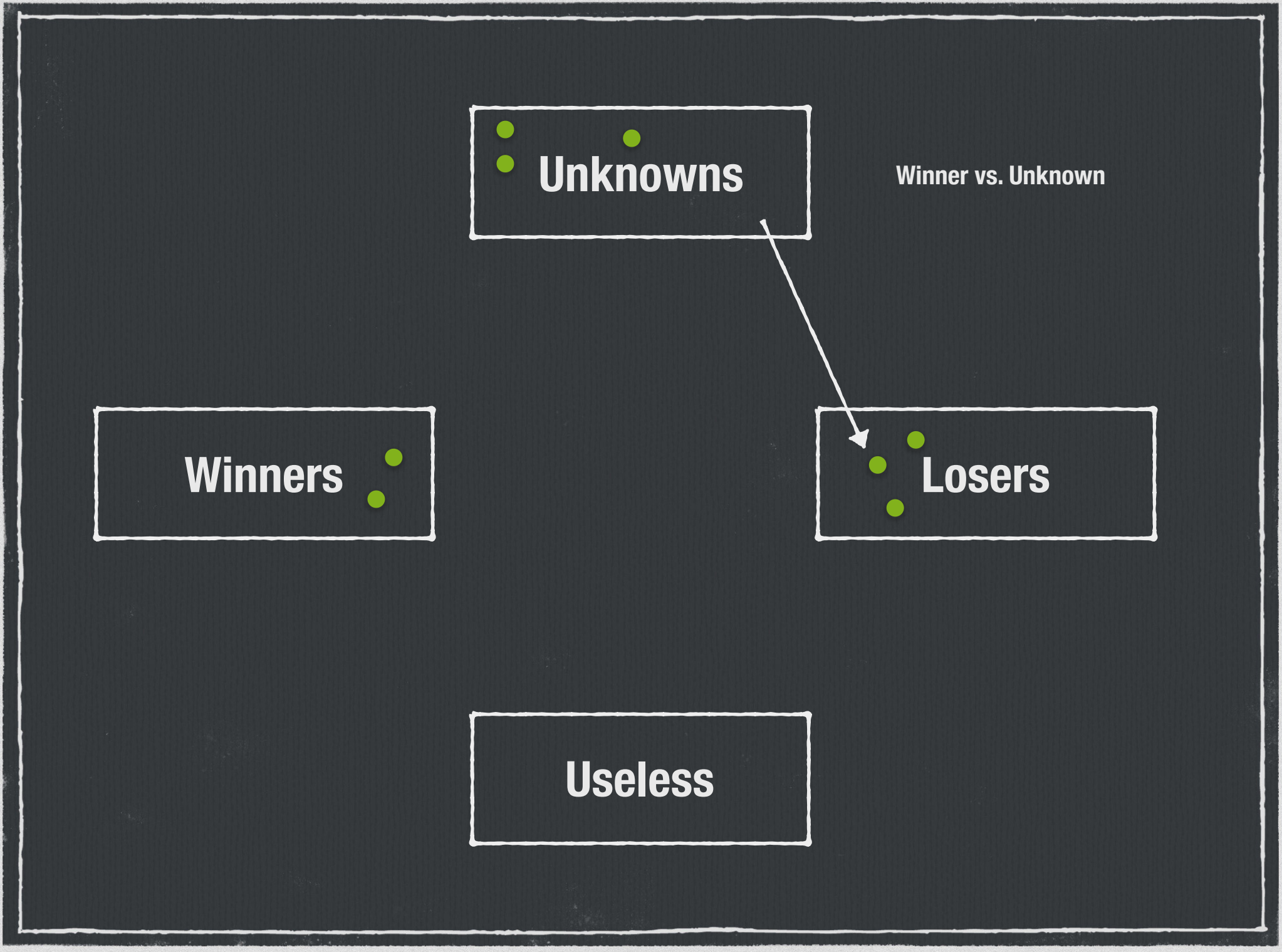

Unknowns
Winners
Losers
Useless

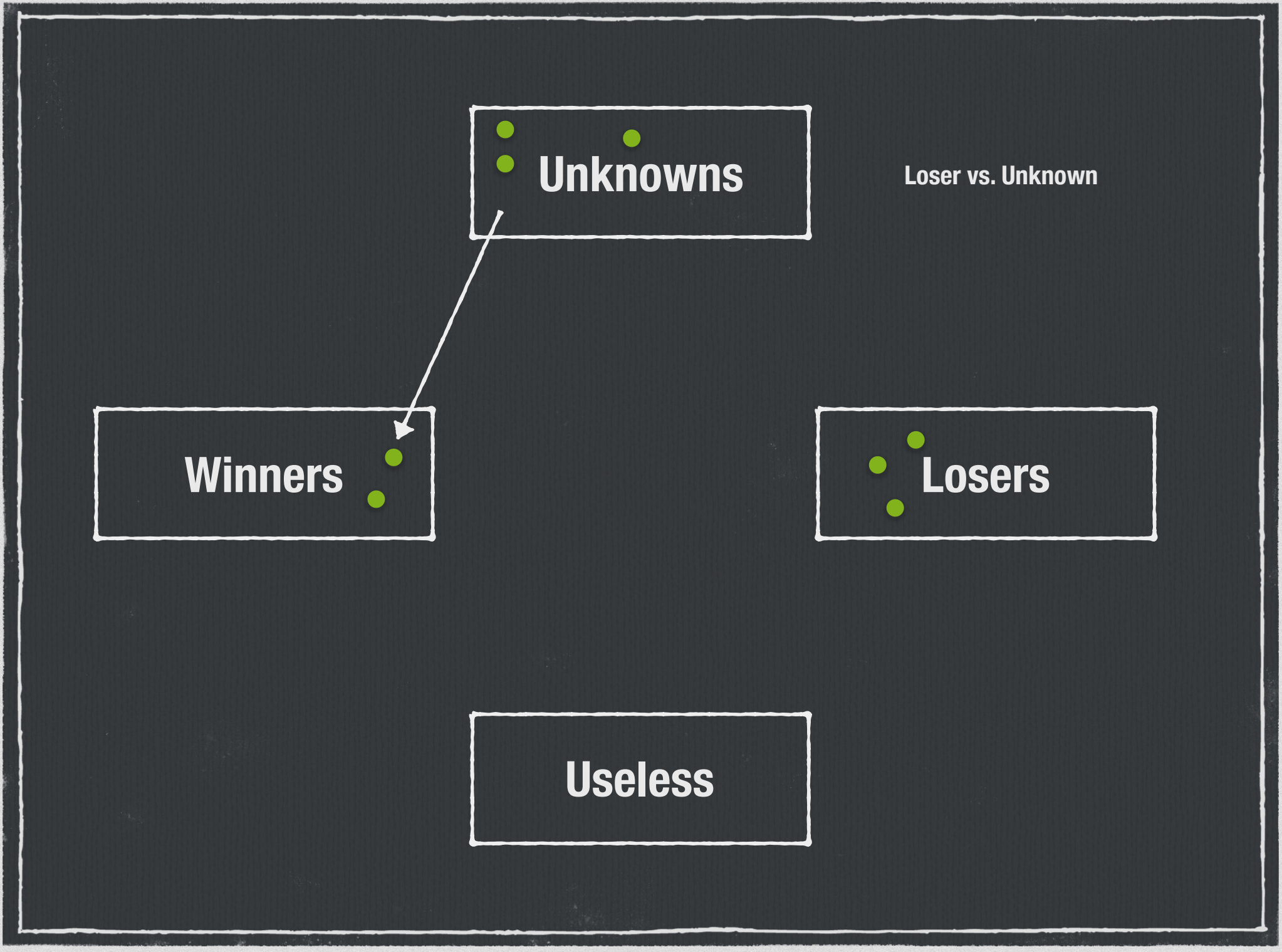

Unknowns
Unknown vs. Unknown
Winners
Losers
Useless

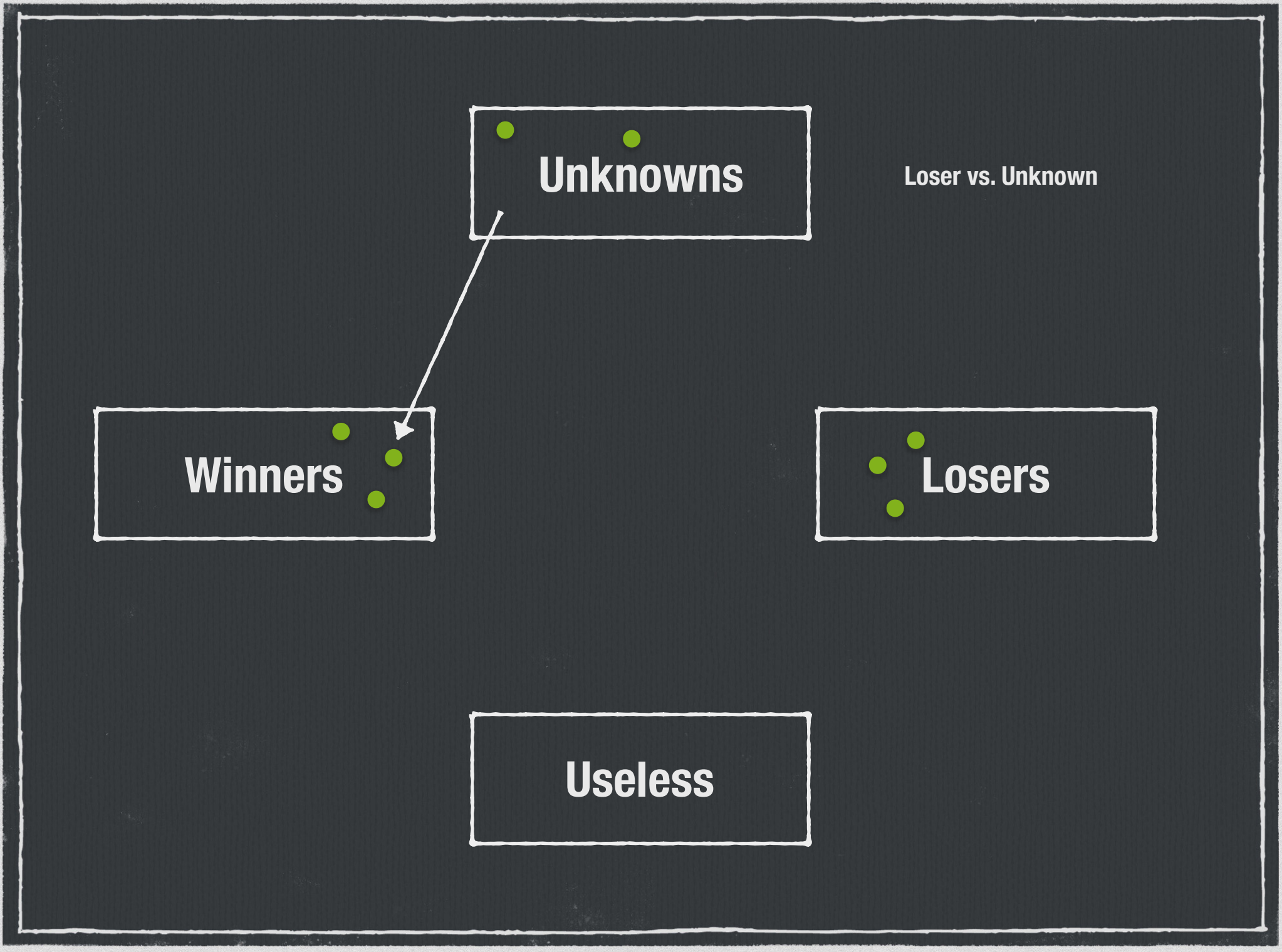

Unknowns
Unknown vs. Unknown
Winners
Losers
Useless

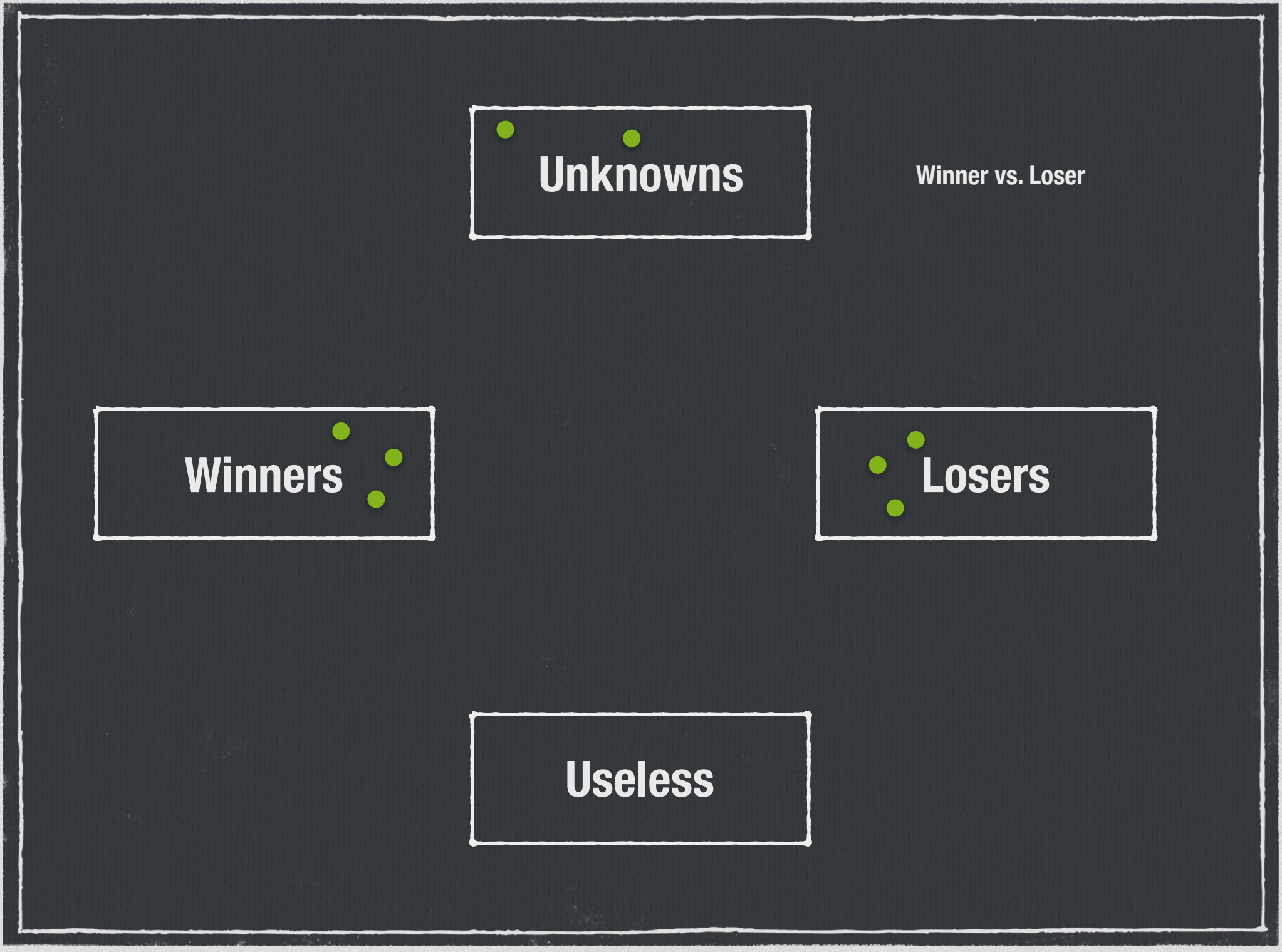

Unknowns
Unknown vs. Unknown
Winners
Losers
Useless

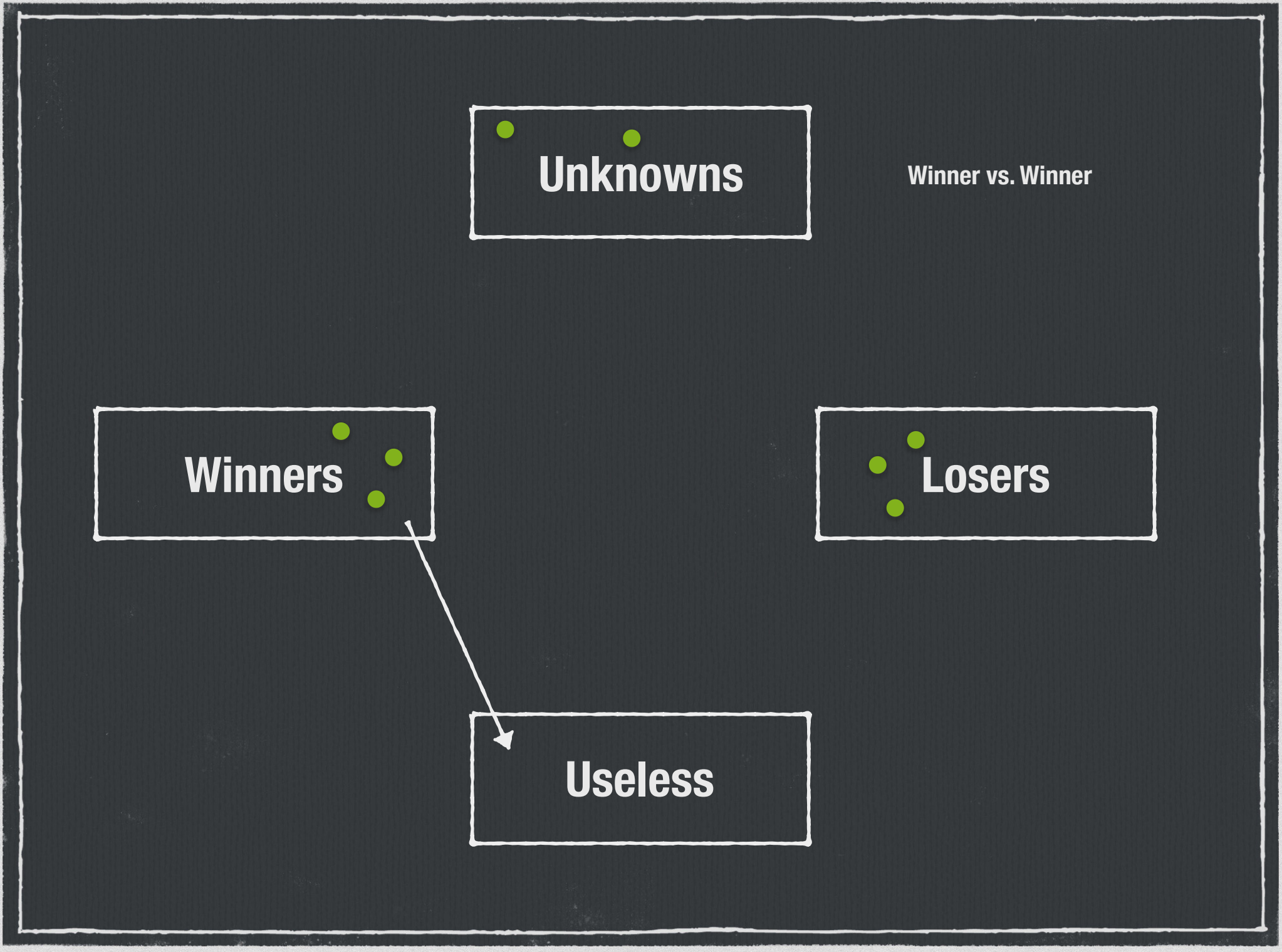

Unknowns
Winner vs. Unknown
Winners
Losers
Useless

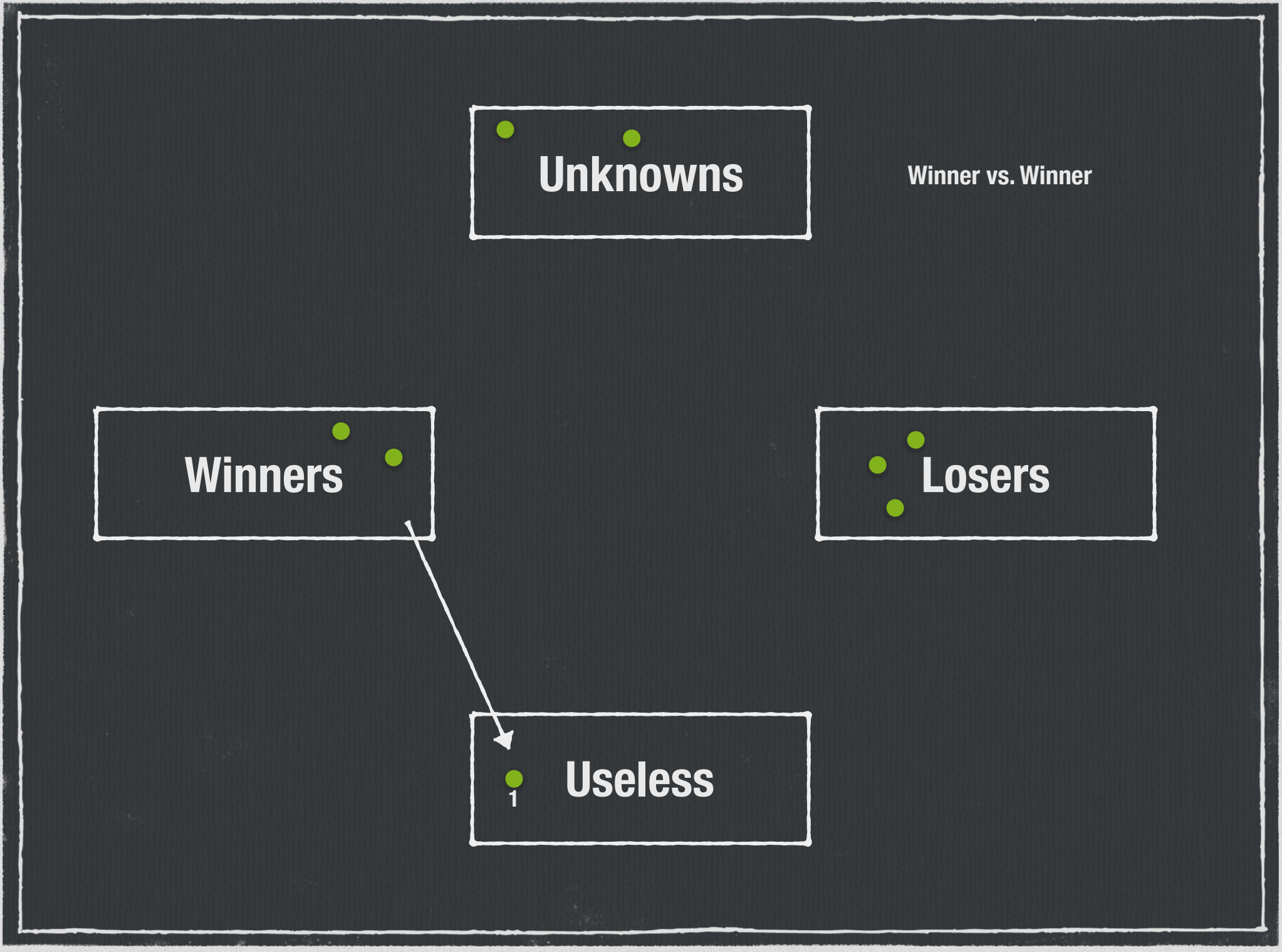

Unknowns
Winner vs. Unknown
Winners
Losers
Useless

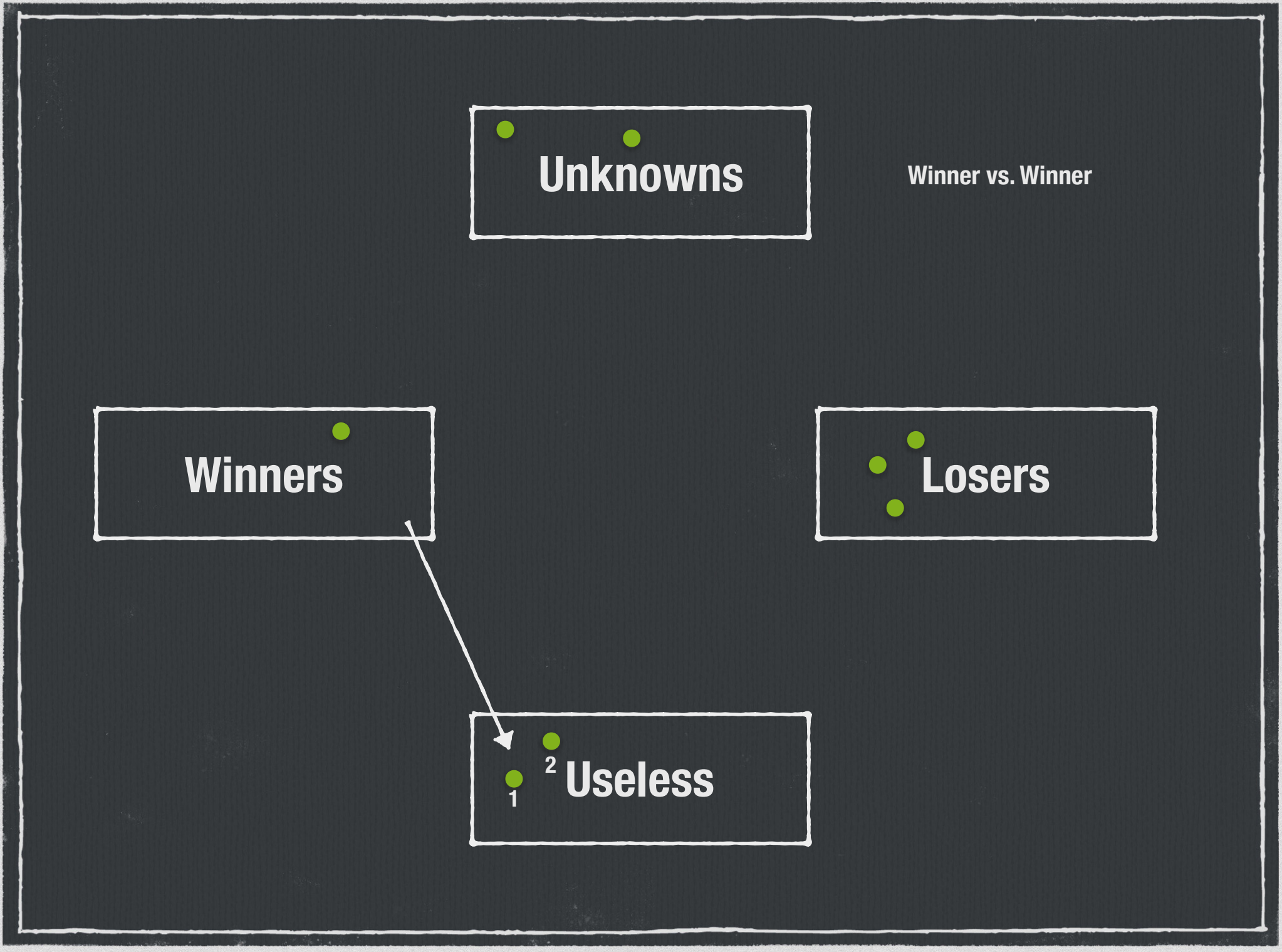

Unknowns
Loser vs. Unknown
Winners
Losers
Useless

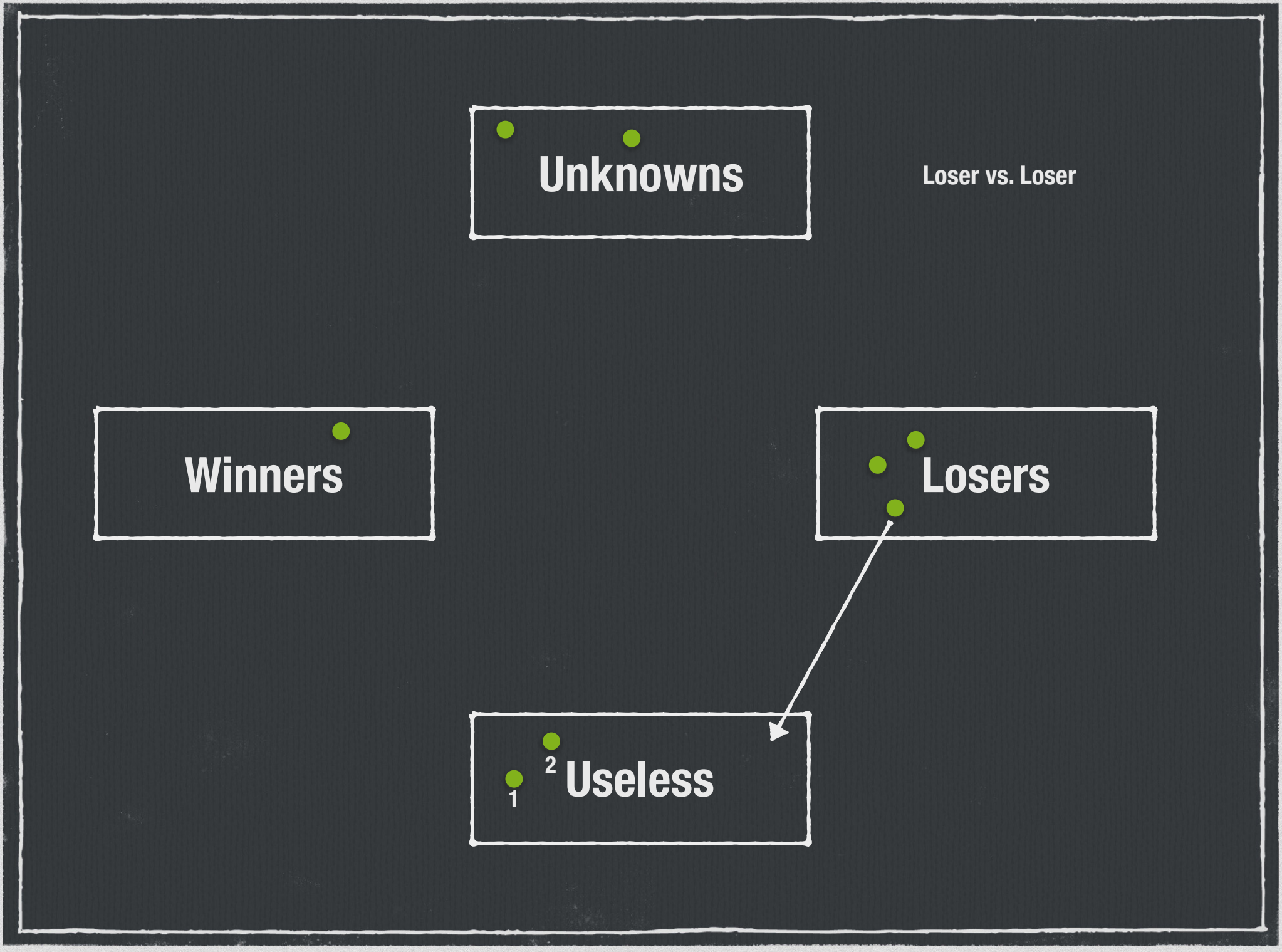

Unknowns
Loser vs. Unknown
Winners
Losers
Useless

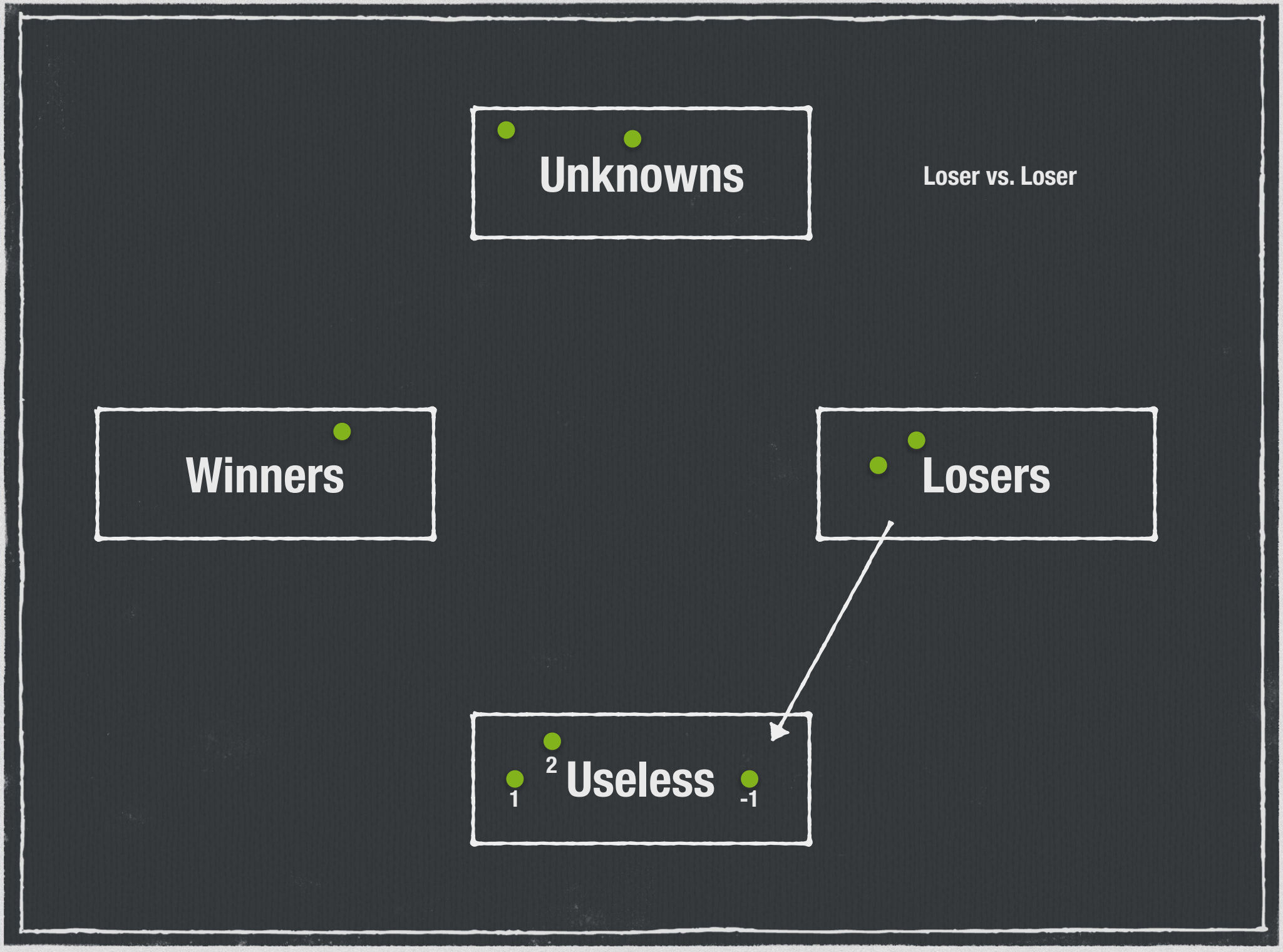

Unknowns
Winner vs. Loser
Winners
Losers
Useless


Unknowns
Winner vs. Winner
Winners
Losers
Useless

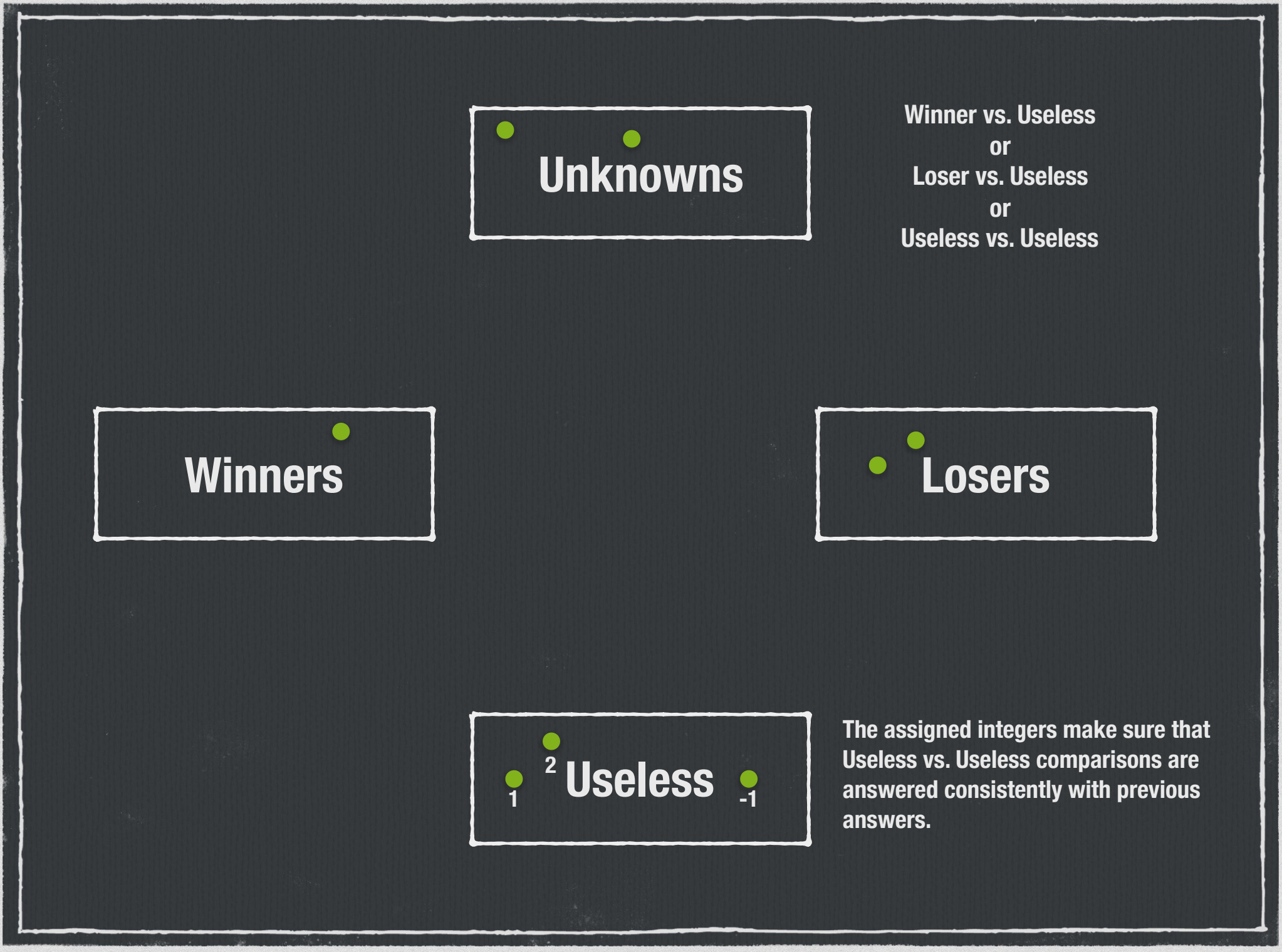

Unknowns
Winner vs. Winner
Winners
Losers
1
Useless


Unknowns
Winner vs. Winner
Winners
Losers
2
Useless
1


Unknowns
Loser vs. Loser
Winners
Losers
2
Useless
1


Unknowns
Loser vs. Loser
Winners
Losers
2
Useless
1
-1


Unknowns
Winner vs. Useless
or
Loser vs. Useless
or
Useless vs. Useless
Winners
Losers
2
Useless
1
-1


Unknowns
Winner vs. Useless
or
Loser vs. Useless
or
Useless vs. Useless
Winners
Losers
2
1
Useless
-1
The assigned integers make sure that Useless vs. Useless comparisons are answered consistently with previous answers.


Unknowns
Eventually…
Winners
Losers
Useless
-2
2
1
-1
3
-3

Analysis

- Eventually all n elements must leave Unknowns, and no more than two elements can leave Unknowns in a single comparison.
  - So at least ⌈n/2⌉ comparisons are required. All of these comparisons involve elements leaving Unknowns and going to Winners ∪ Losers.
- n-2 elements must move from (Winners ∪ Losers) to Useless.
  - Referring to the adversary's strategy (and diagrams on previous slides), this only happens when we compare two elements in Winners or two elements in Losers. As a result, only one element moves to Useless with each such comparison.
- Adding these two up, we see that at least ⌈n/2⌉+(n-2) comparisons must occur, which gives ⌈3n/2⌉ - 2.

Matching Upper Bound:
An algorithm using ⌈3n/2⌉ - 2 comparisons

- Thanks to this lower bound proof, an optimal algorithm pretty much falls into our lap.
- Do exactly what the proof did: define sets Unknowns, Winners, Losers, Useless.

  Initially, all array elements are considered Unknowns.

  1. While (# of Unknowns) ≥ 2, compare two elements from Unknowns.

     Put the larger one in Winners and put the smaller one in Losers.
  2. If (# of Unknowns) == 1, compare the e ∈ Unknown with any other element.

     If e is smaller, put in Losers. If e is bigger, put in Winners. (this is special case if n is odd)
  3. While (# of Losers) ≥ 2, compare two elements from Losers. Put bigger one in Useless.
  4. While (# of Winners) ≥ 2, compare two elements from Winners. Put smaller one in Useless.
  5. Output the last x in Losers as min, output the last y in Winners as max.