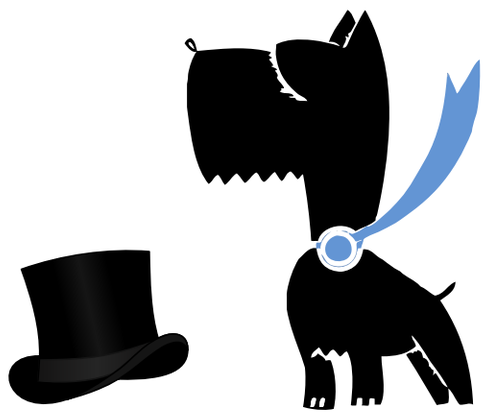
Schrödinger's hats

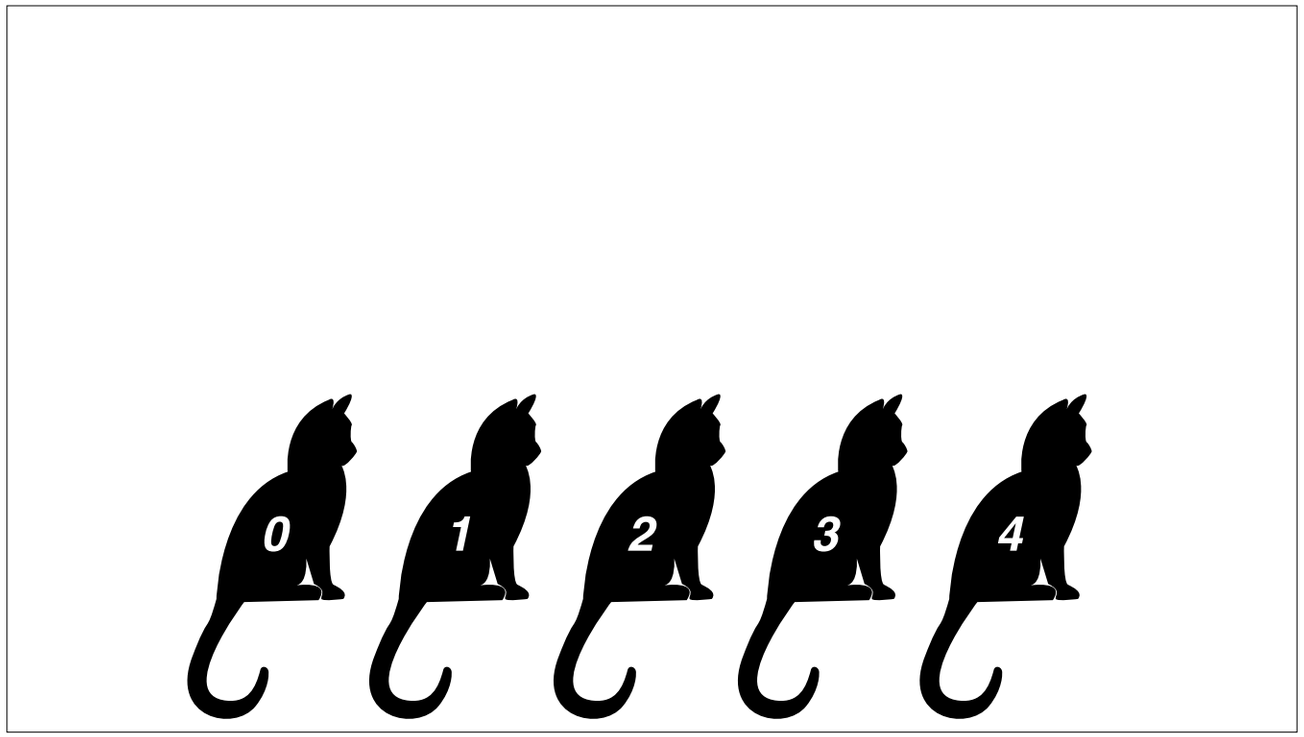
A puzzle about parities and permutations

`m.brck.nl/ylj17`

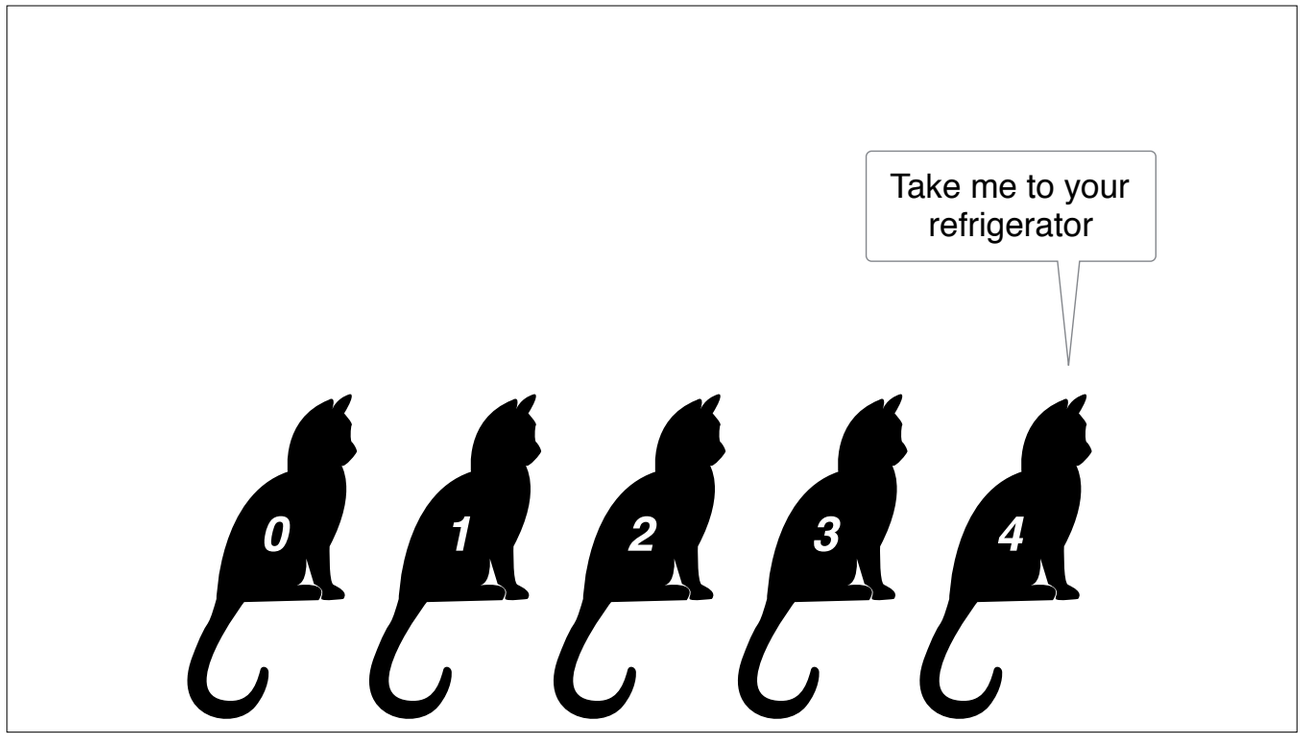
Matthew Brecknell
Proof Engineer



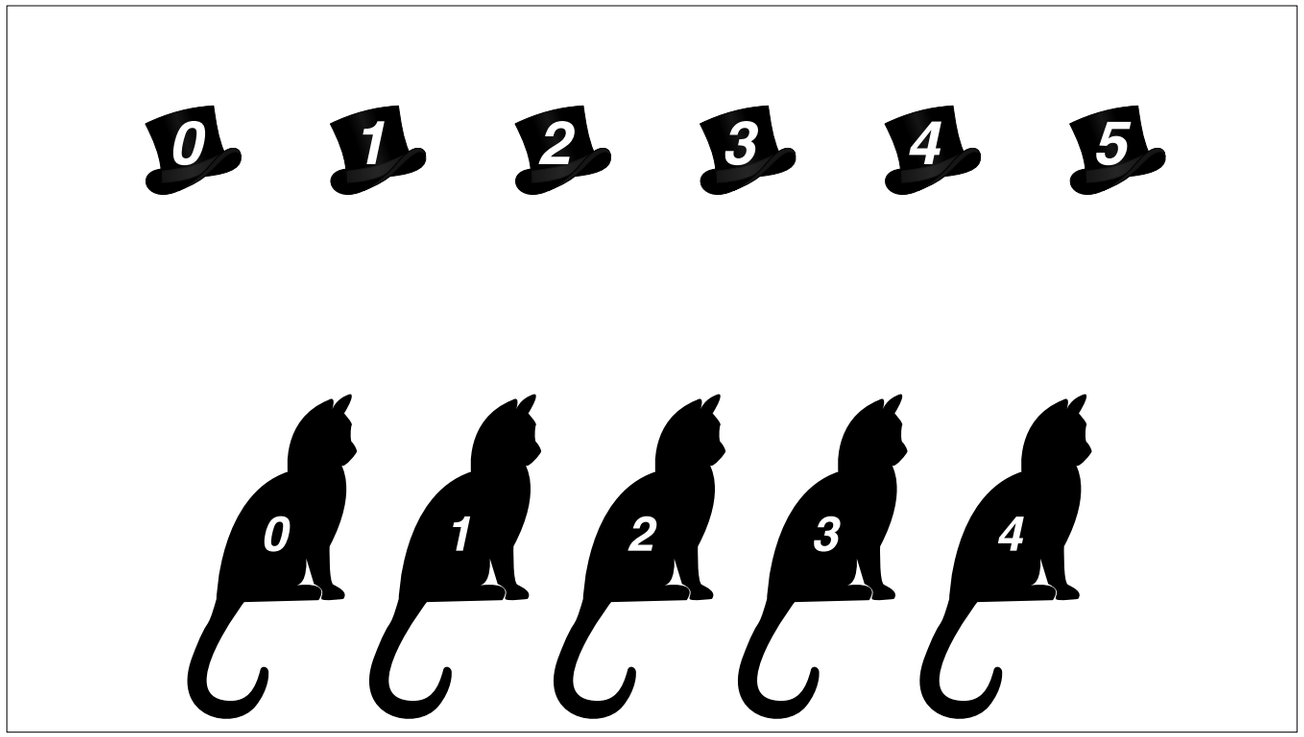
Meet Schrödinger…



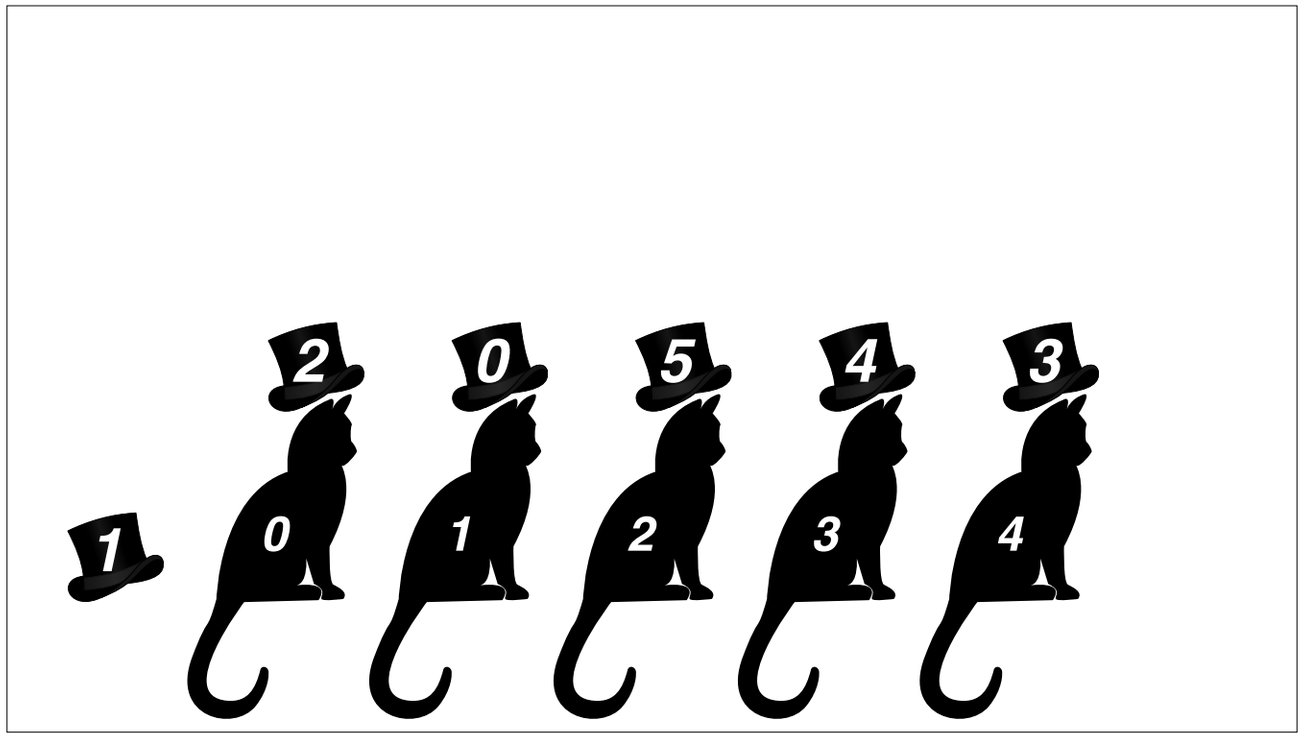
…who travels the world with an unusually clever clowder of *n* talking cats. In their latest show, the cats sit in a line.



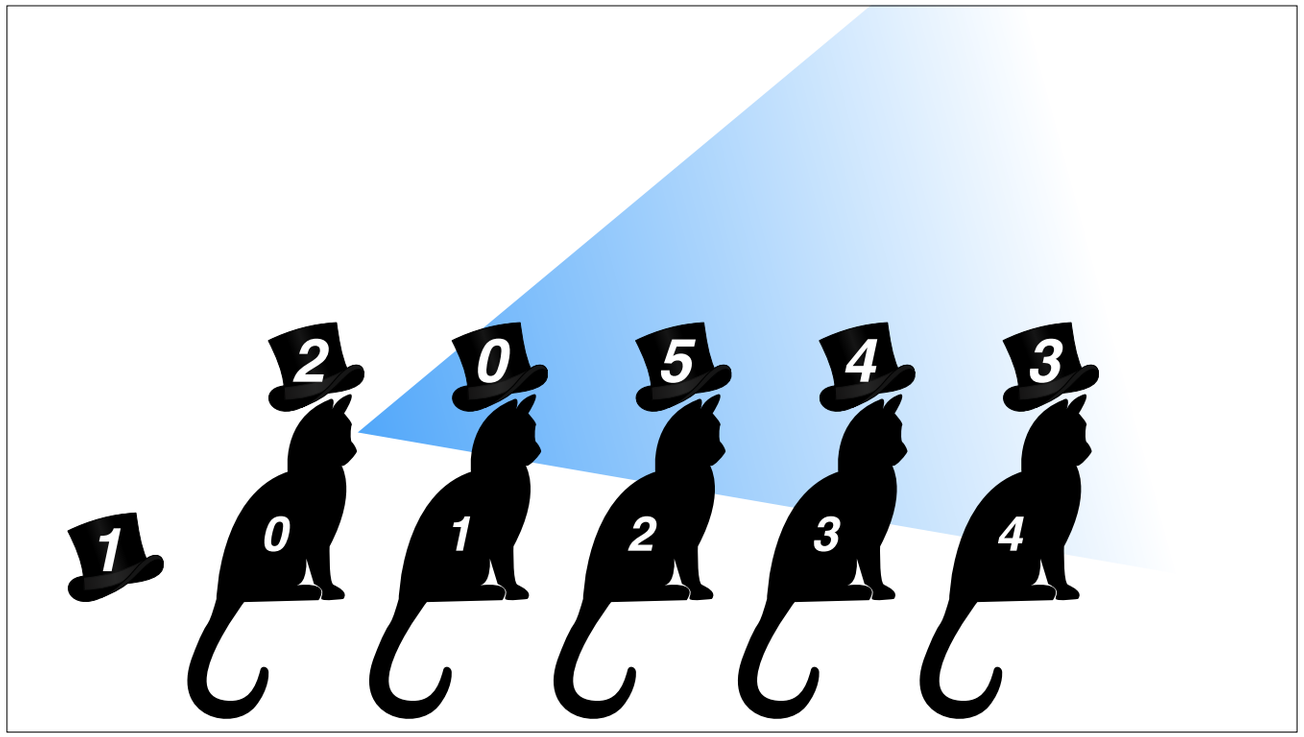
…who travels the world with an unusually clever clowder of *n* talking cats. In their latest show, the cats sit in a line.



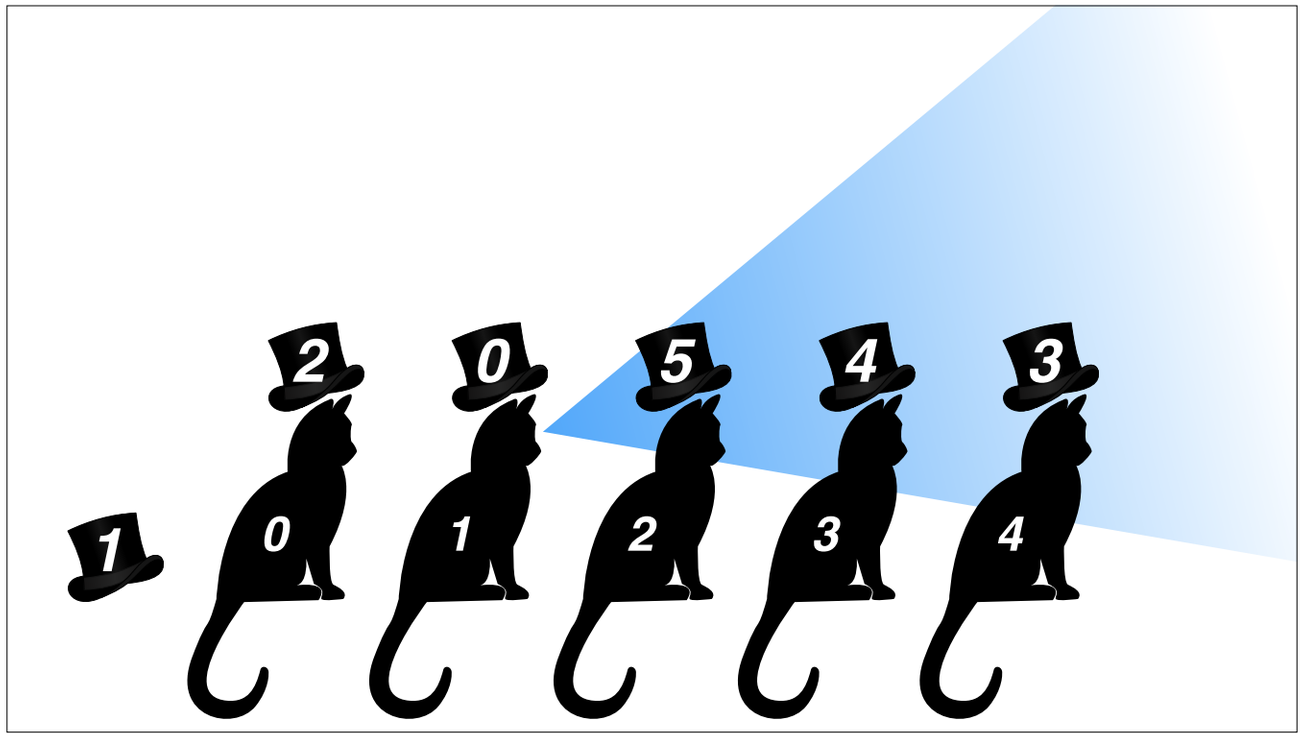
Schrödinger asks a volunteer to take *n* hats, numbered 0 to *n*…



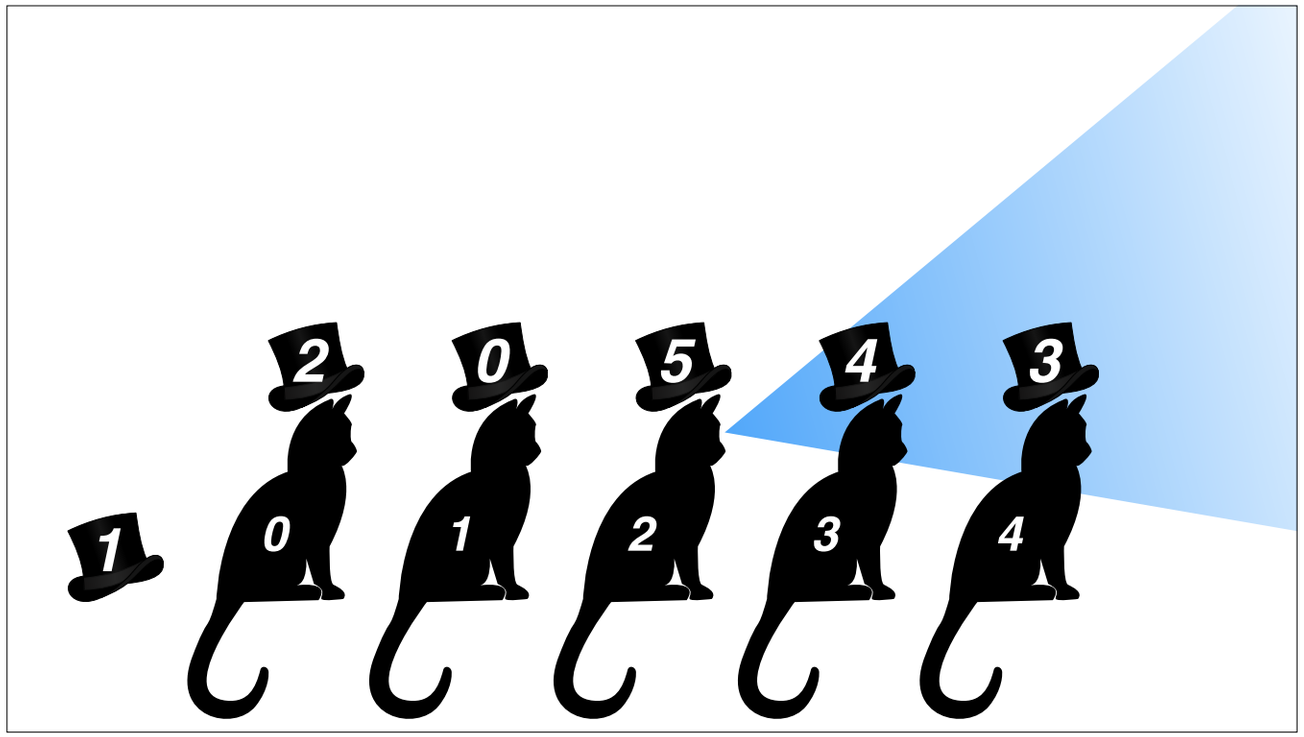
…and randomly assign one to each cat, so that there is one spare.



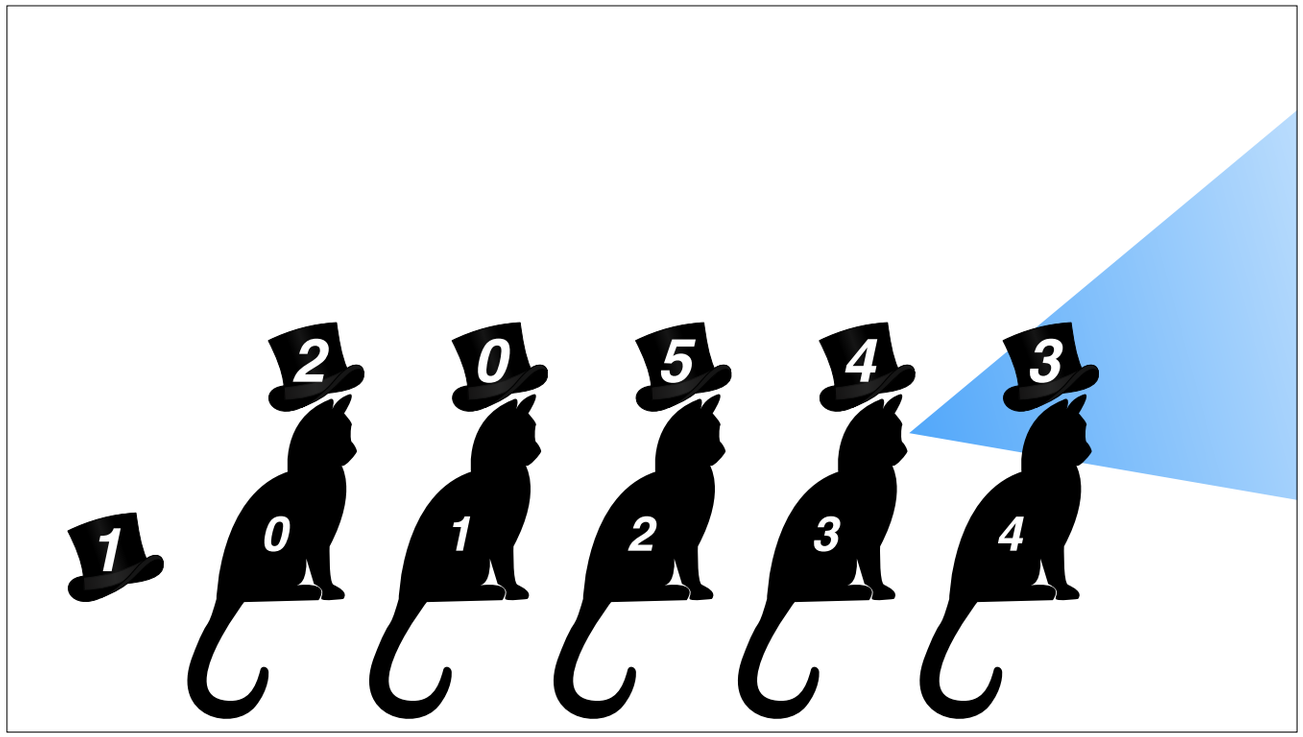
Each cat can see all of the hats in front of it…



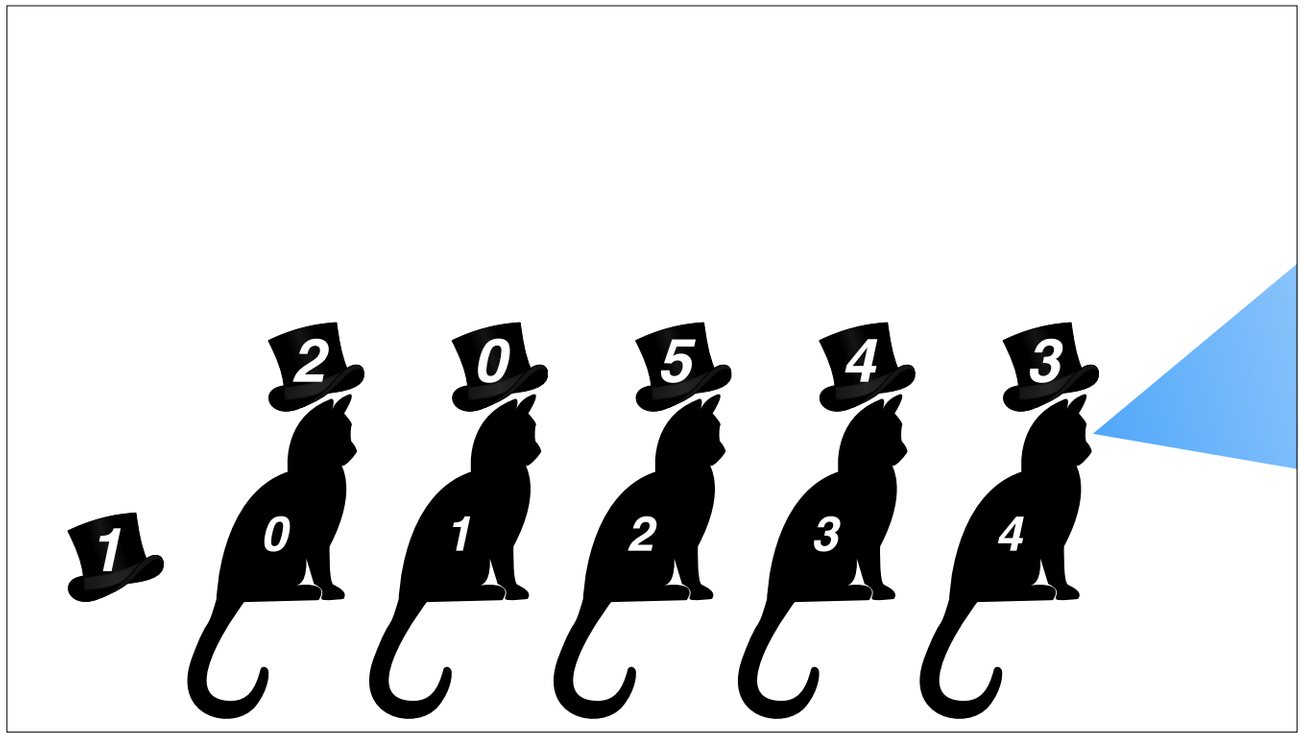
…but not its own hat…

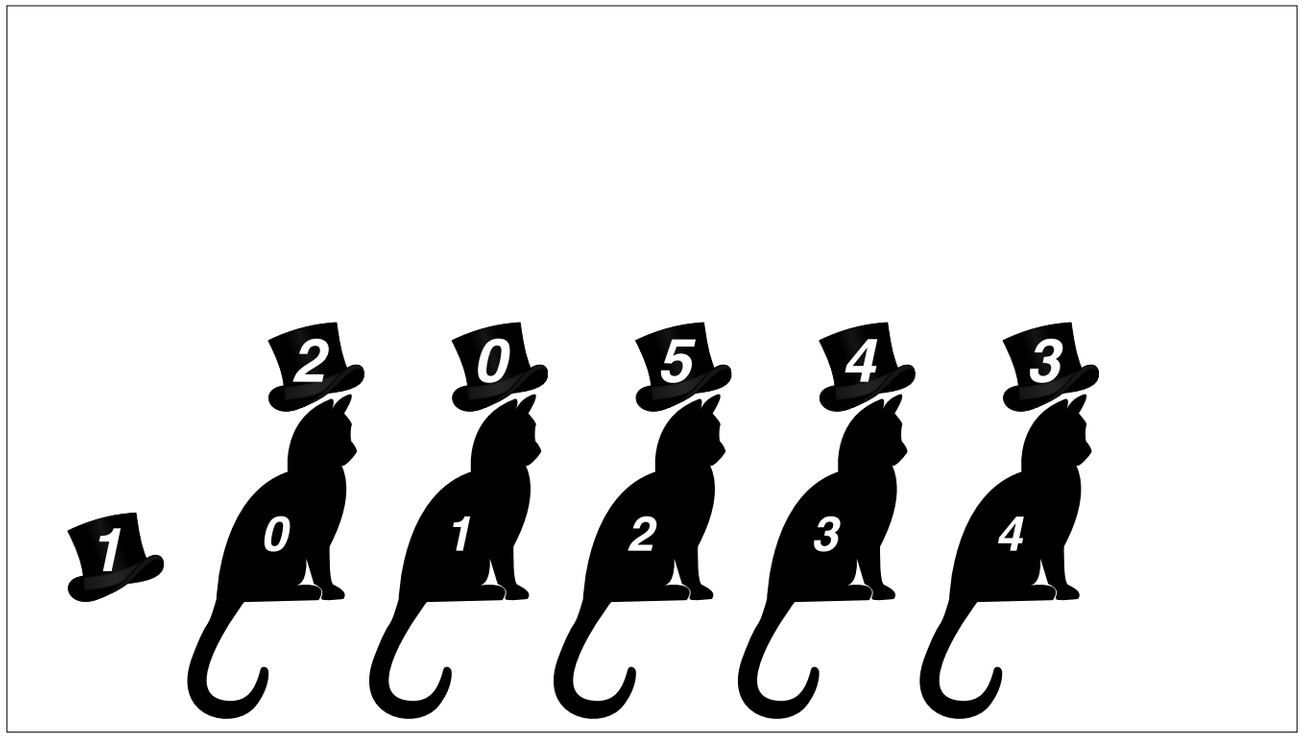


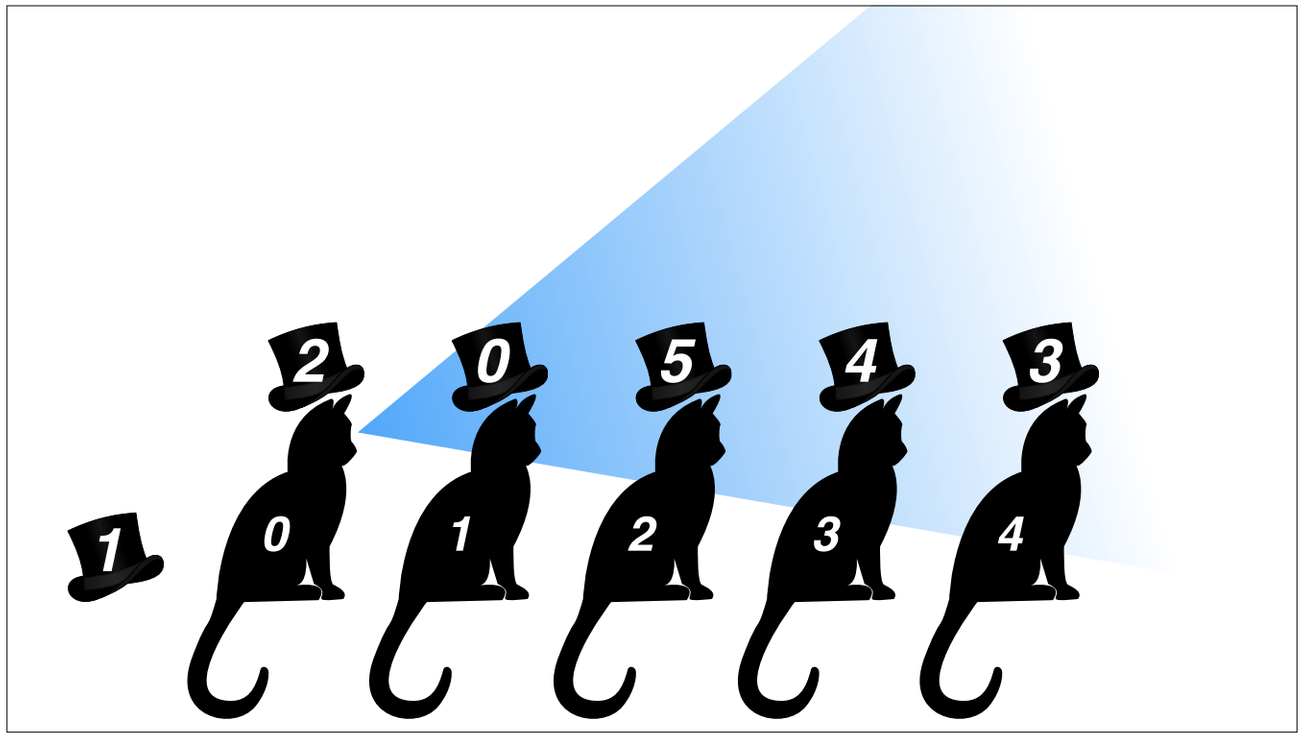
…nor any of the hats behind it…



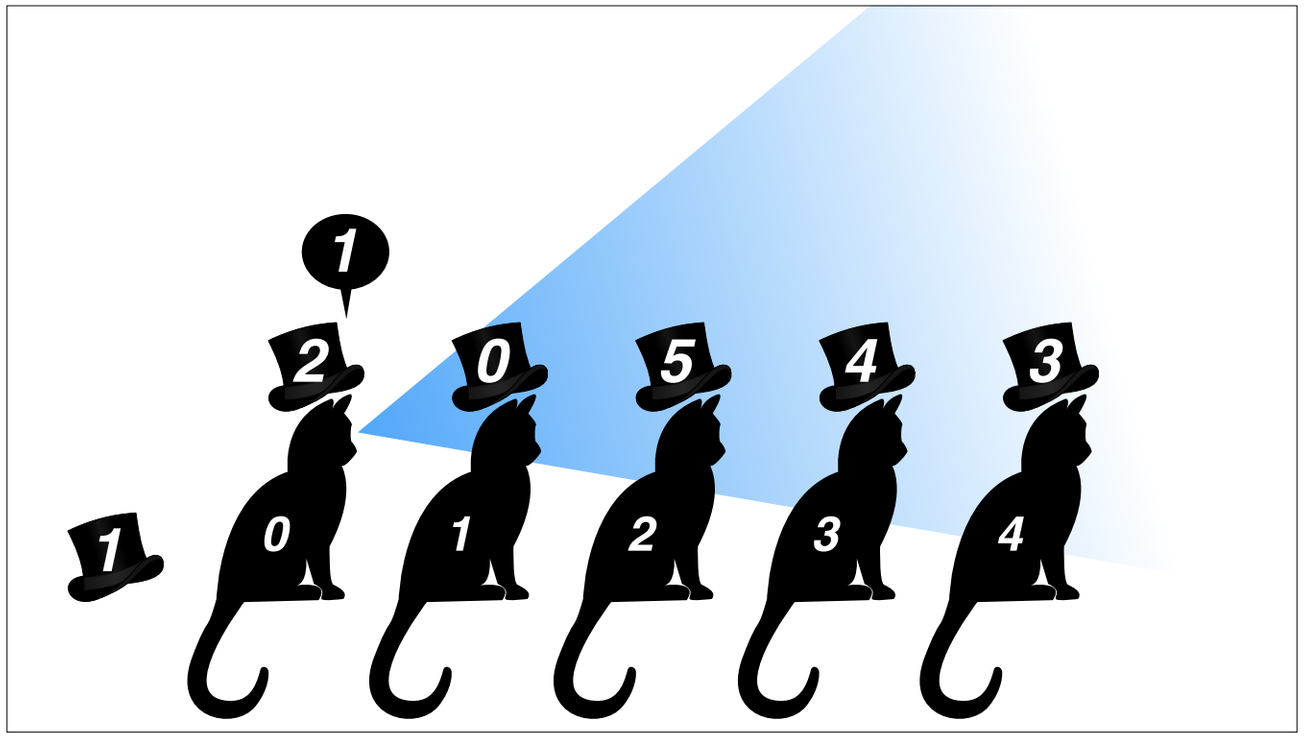
…nor the spare hat.


2
0
5
4
3
1
0
1
2
3
4

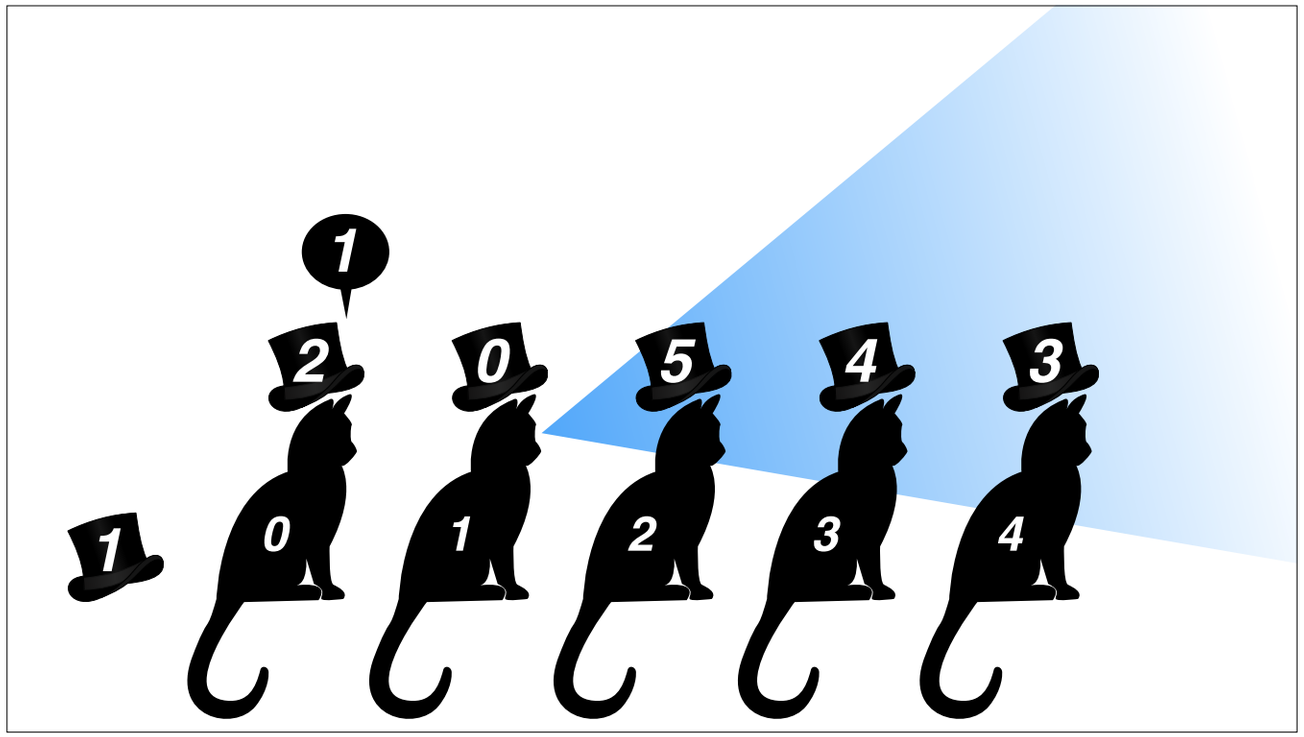

1
2
0
5
4
3
0
1
2
3
4



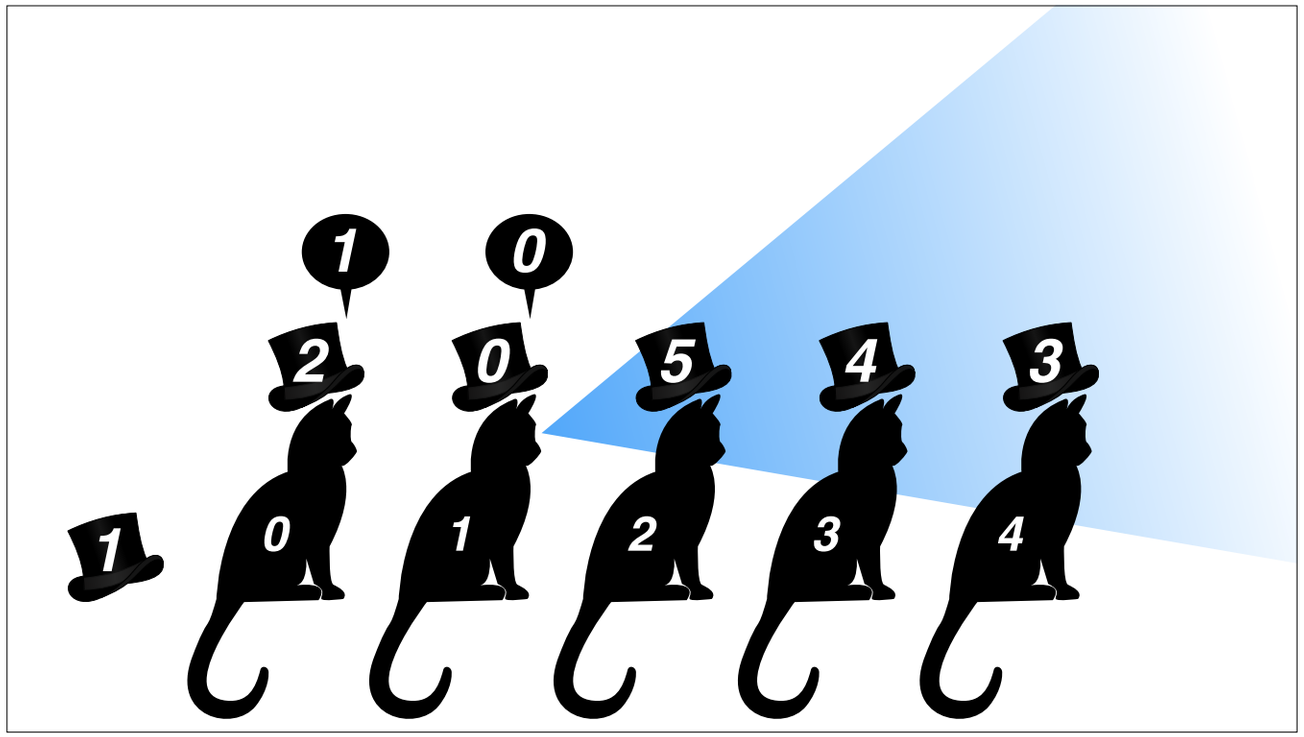
The cats then take turns…



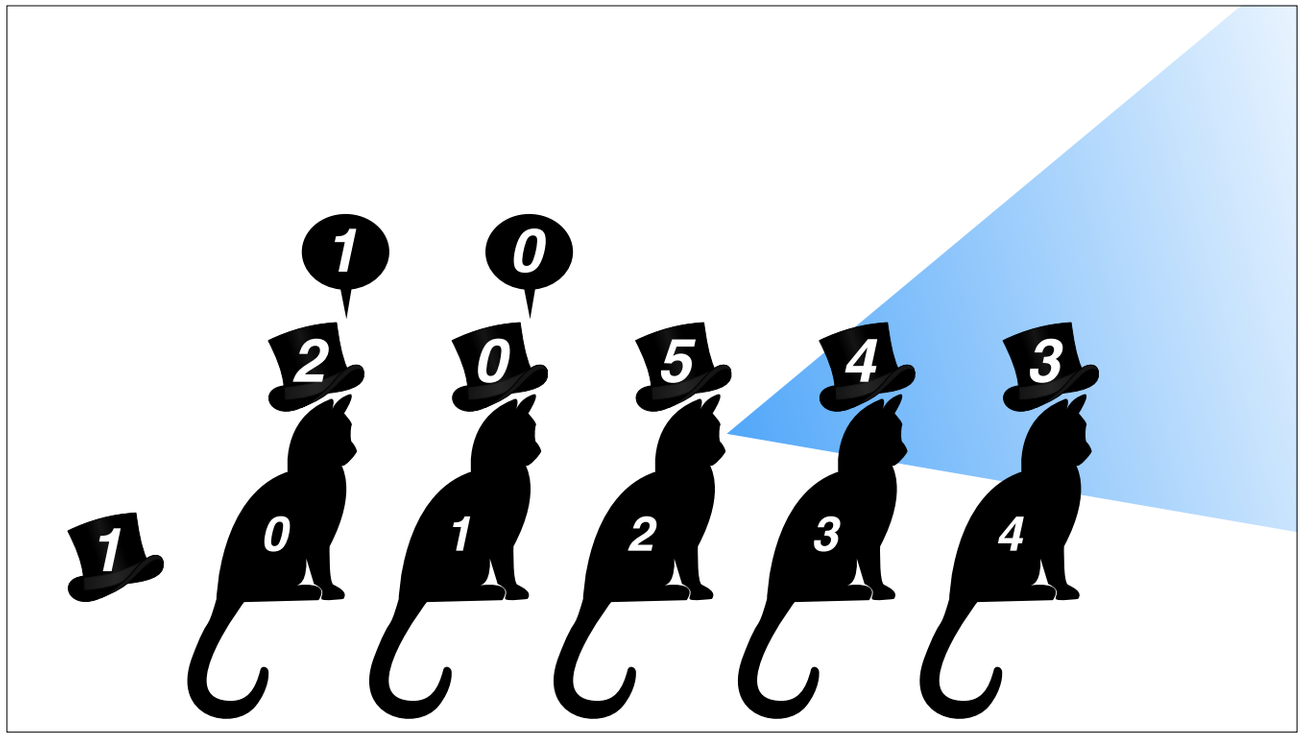
The cats then take turns…



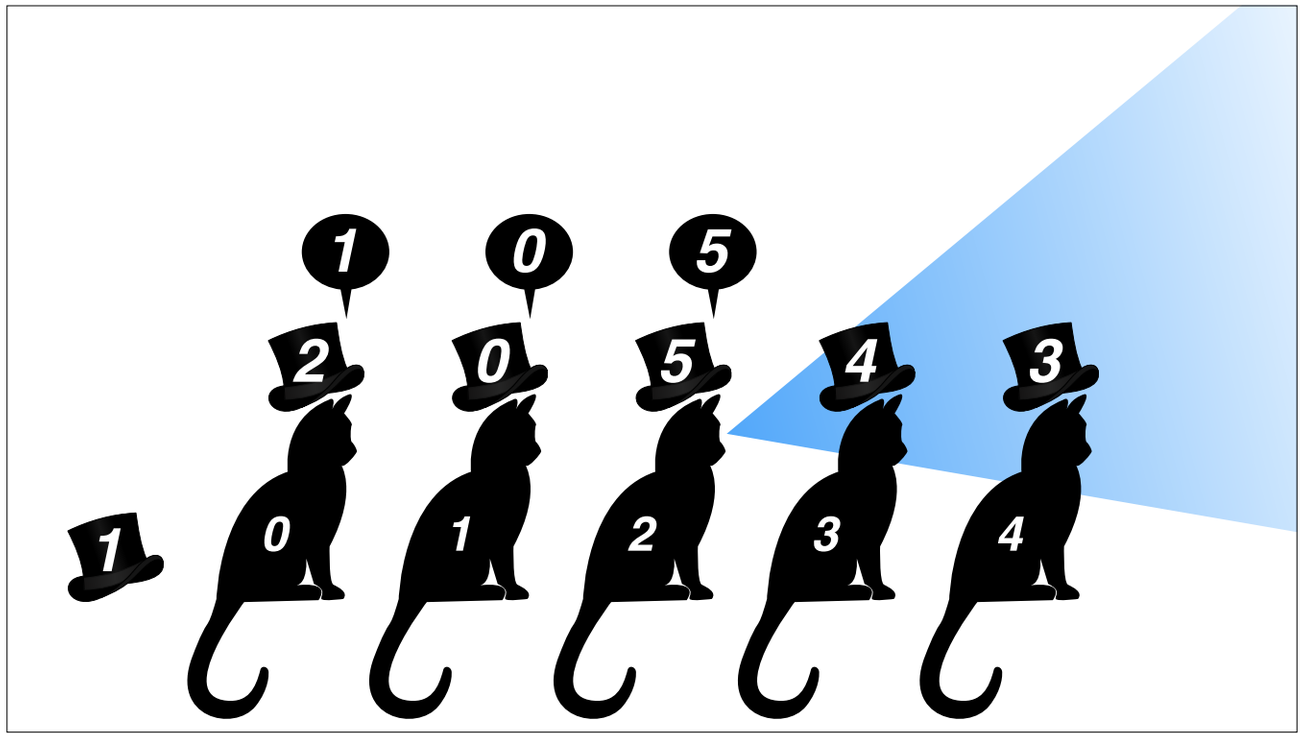
…each calling out a single number in the range 0 to *n*…



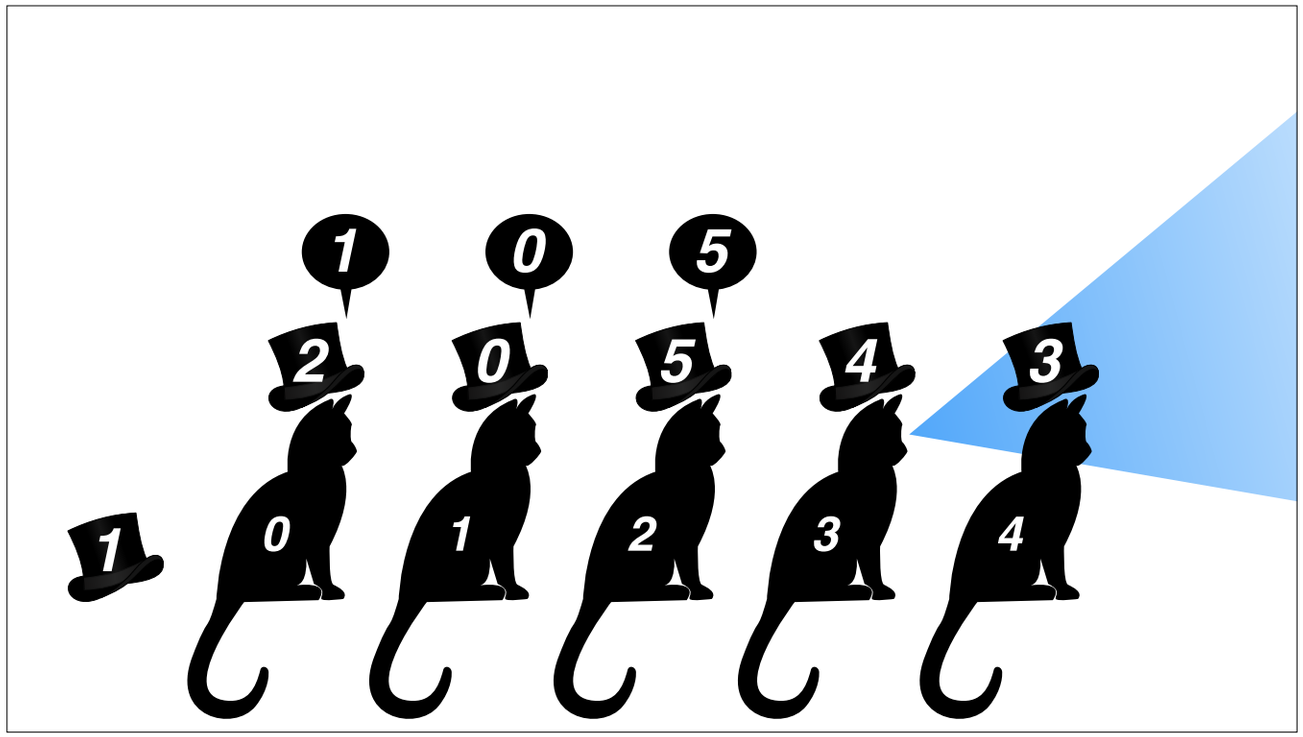
…each calling out a single number in the range 0 to *n*…



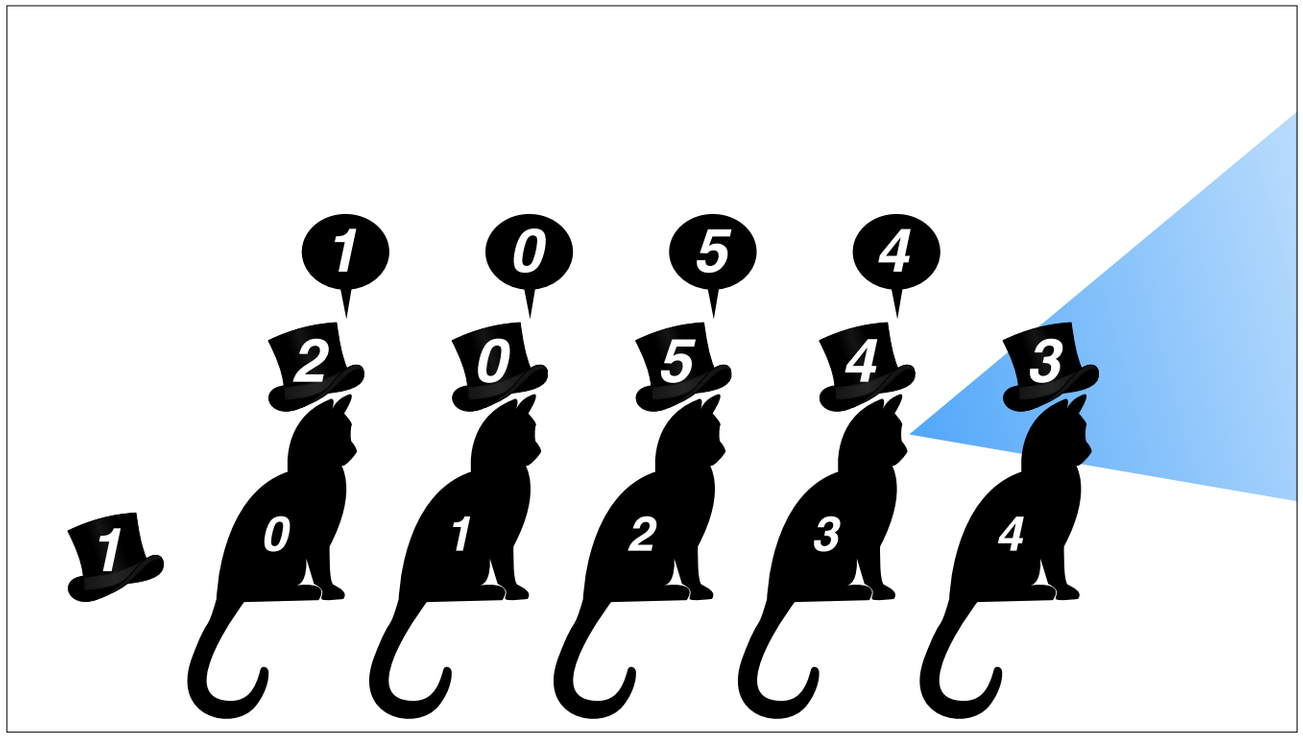
…without repeating any number previously called…



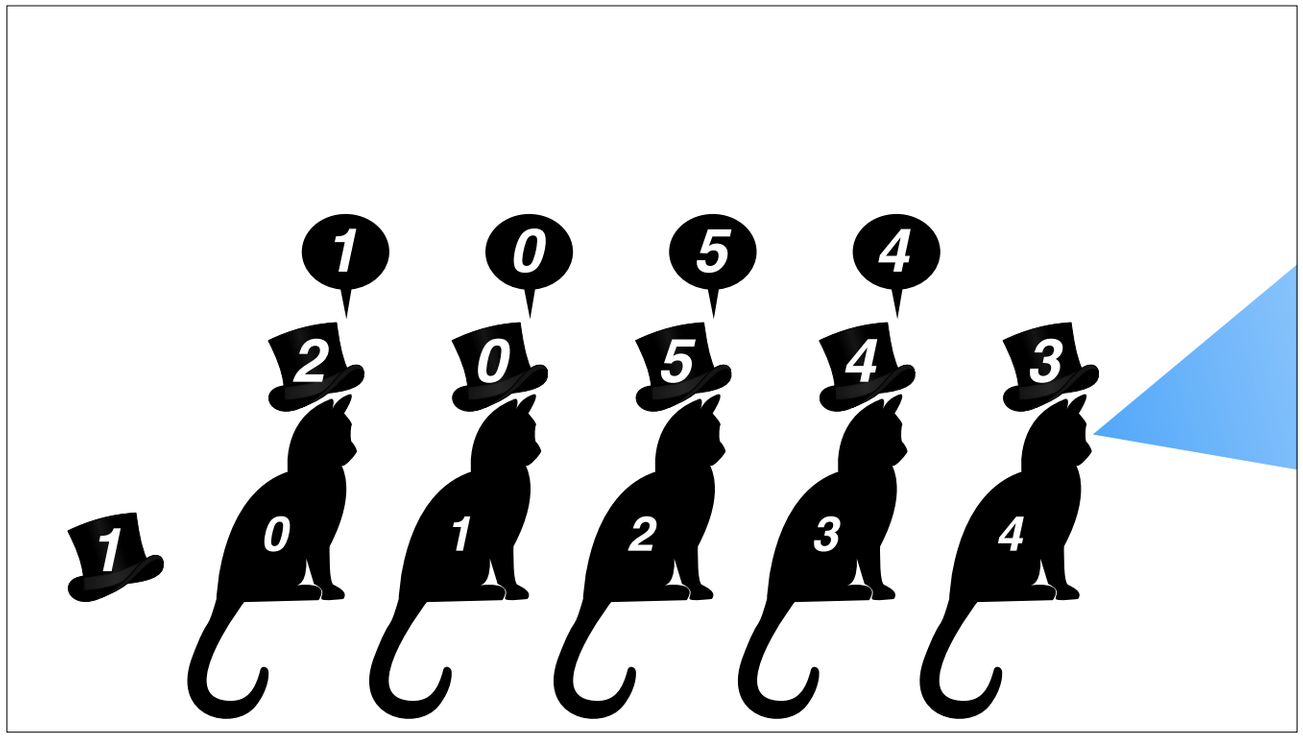
…without repeating any number previously called…

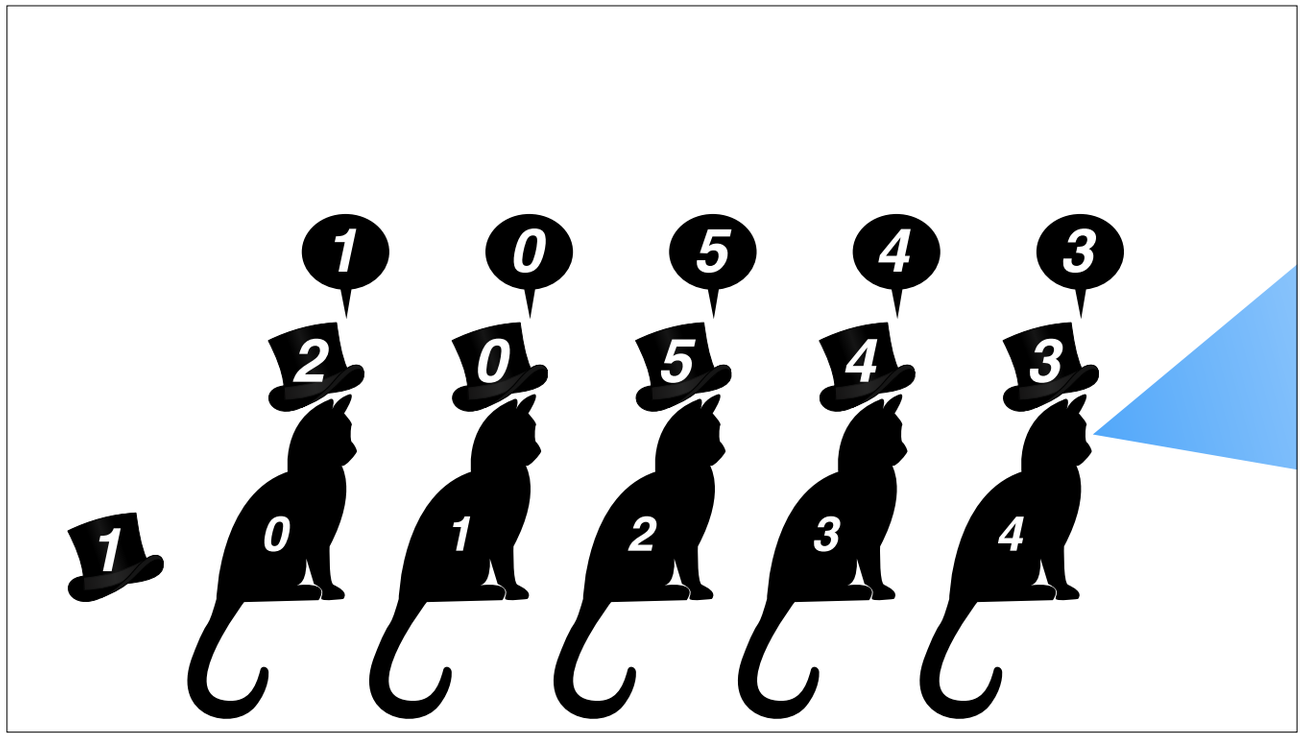


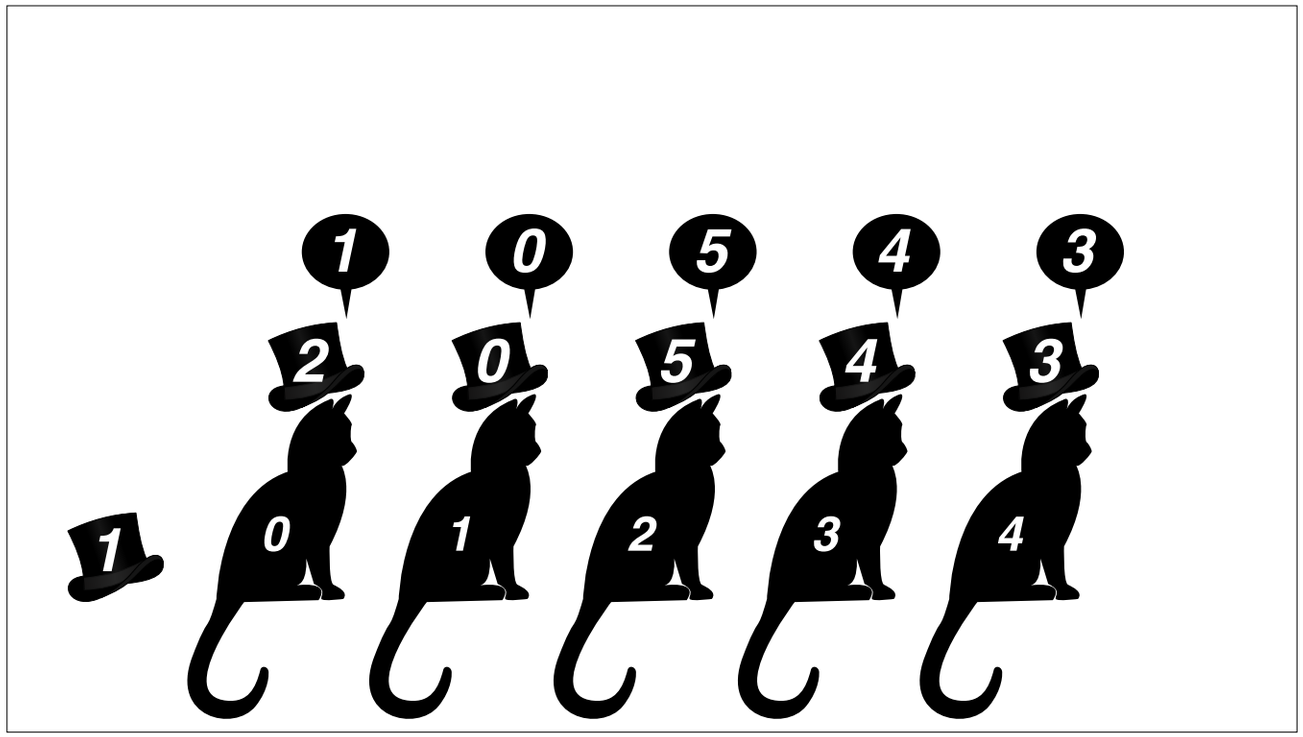
…and without any other communication.



…and without any other communication.


1
0
5
4
2
0
5
4
3
1
0
1
2
3
4


1
0
5
4
3
2
0
5
4
3
1
0
1
2
3
4


1 0 5 4 3
2 0 5 4 3
1 0 1 2 3 4

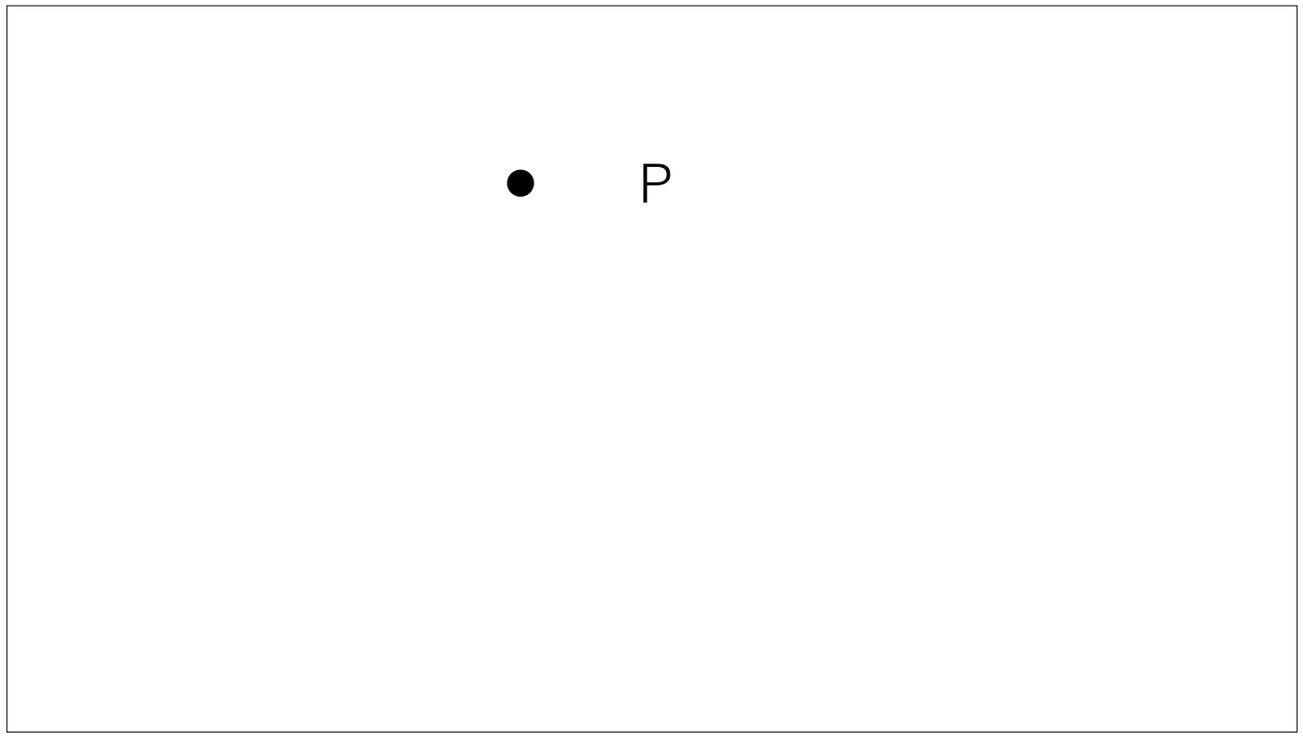
The plan is to:

- Figure out how they do this, by a methodical process of logical deduction.
- Formally prove that our solution is correct.

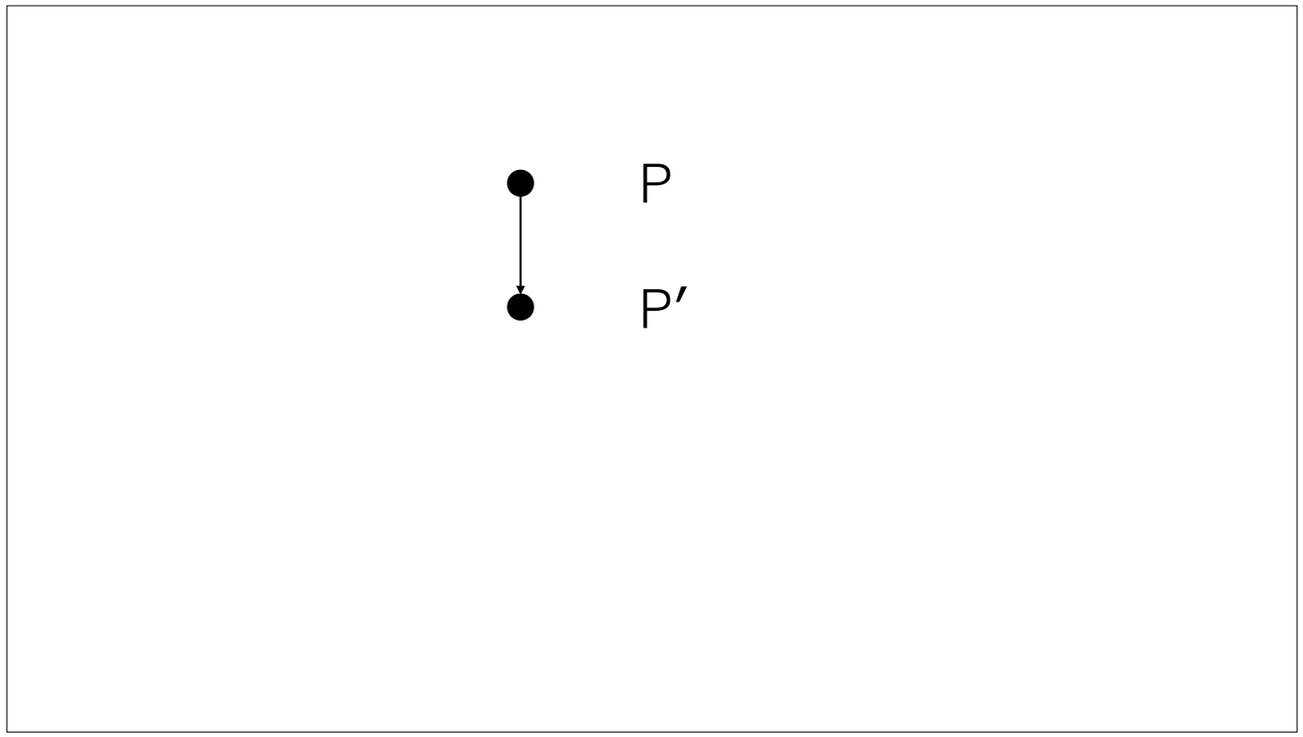
I hope to convince you that mechanical theorem proving provides:

- a language
- a tool
- a personal discipline

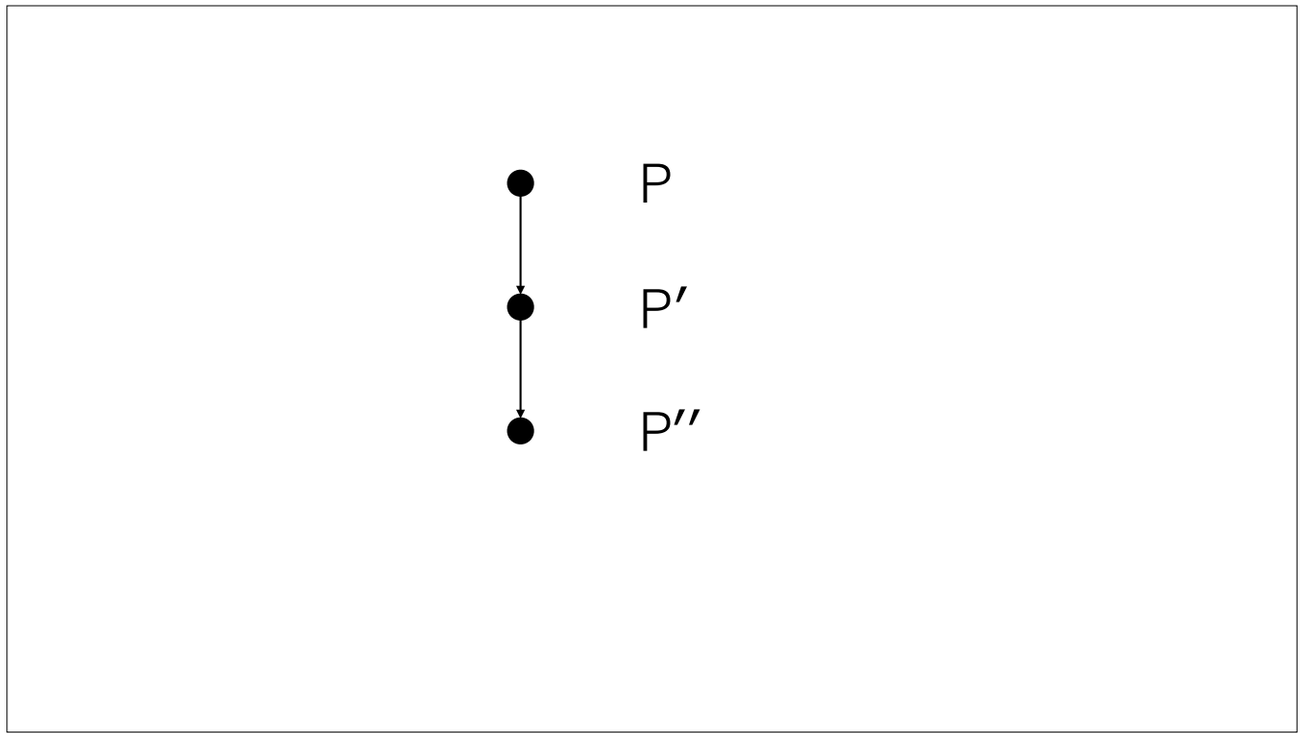
for making your thinking more precise.



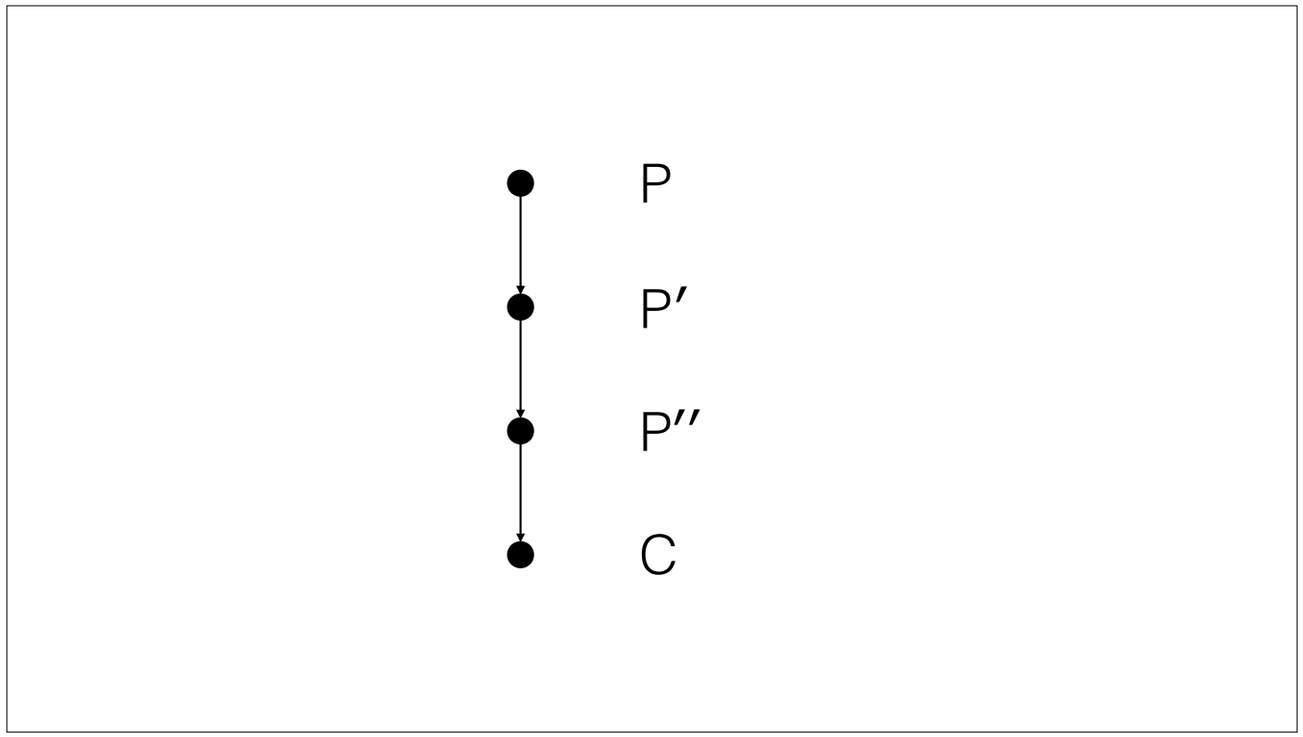
Informal refinement is top-down. Proof is bottom-up.



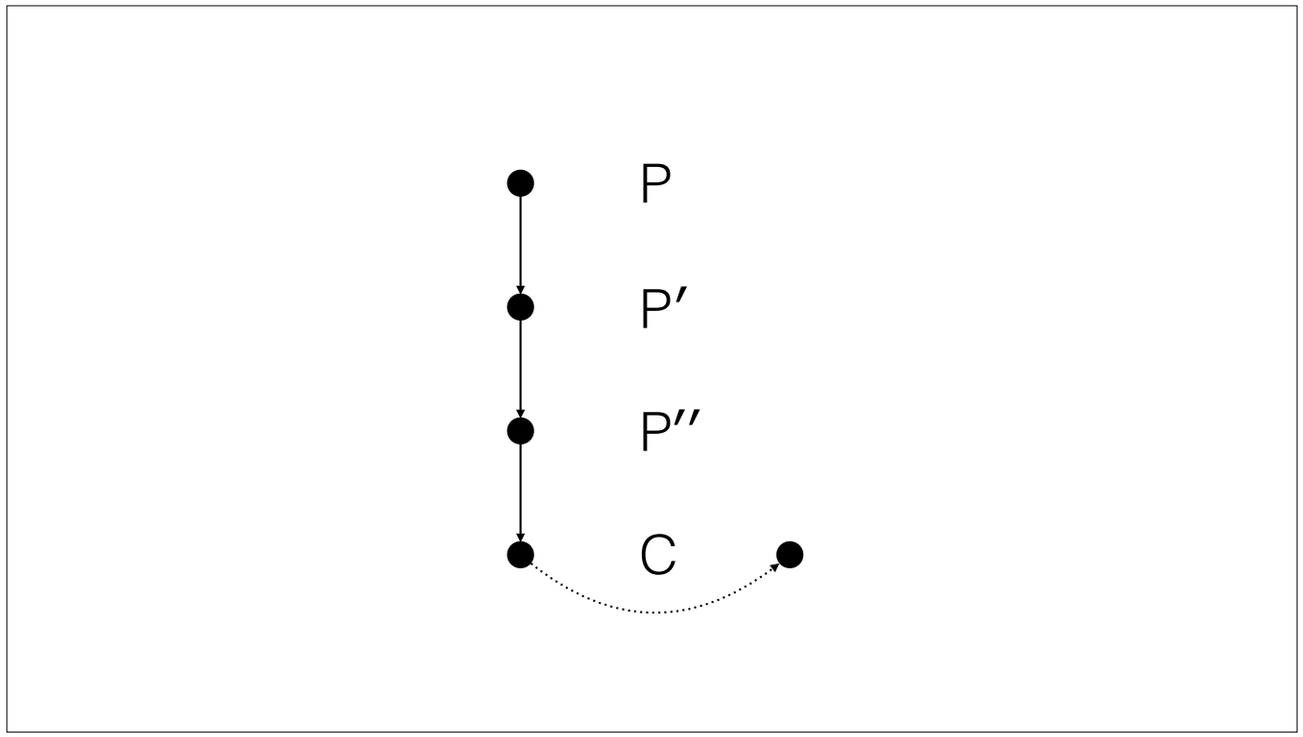
Informal refinement is top-down. Proof is bottom-up.



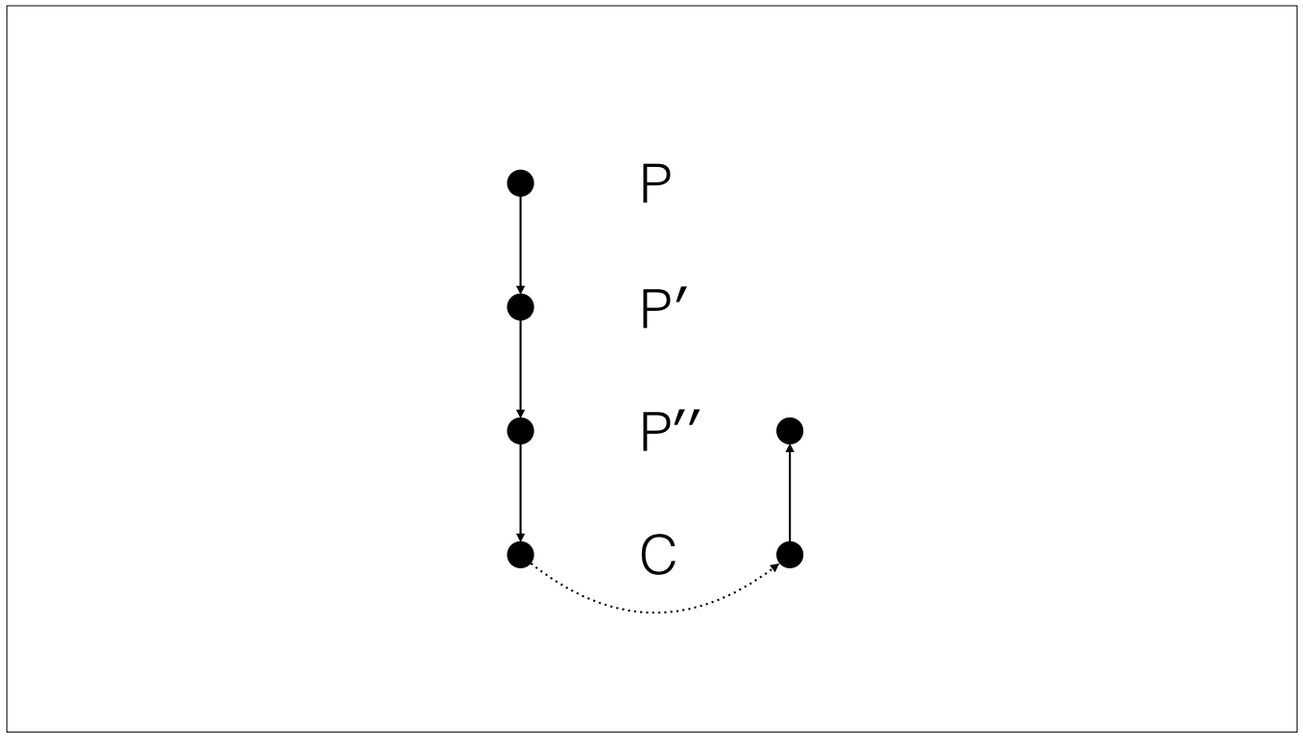
Informal refinement is top-down. Proof is bottom-up.



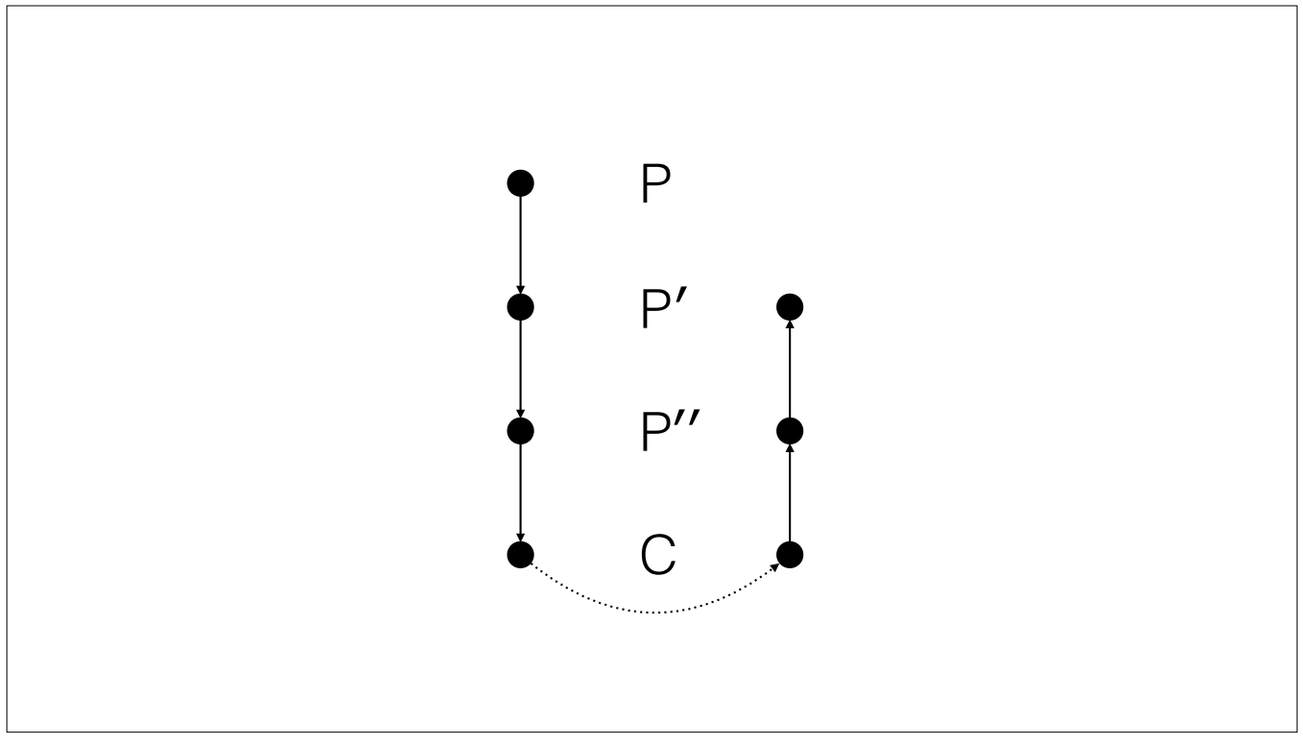
Informal refinement is top-down. Proof is bottom-up.



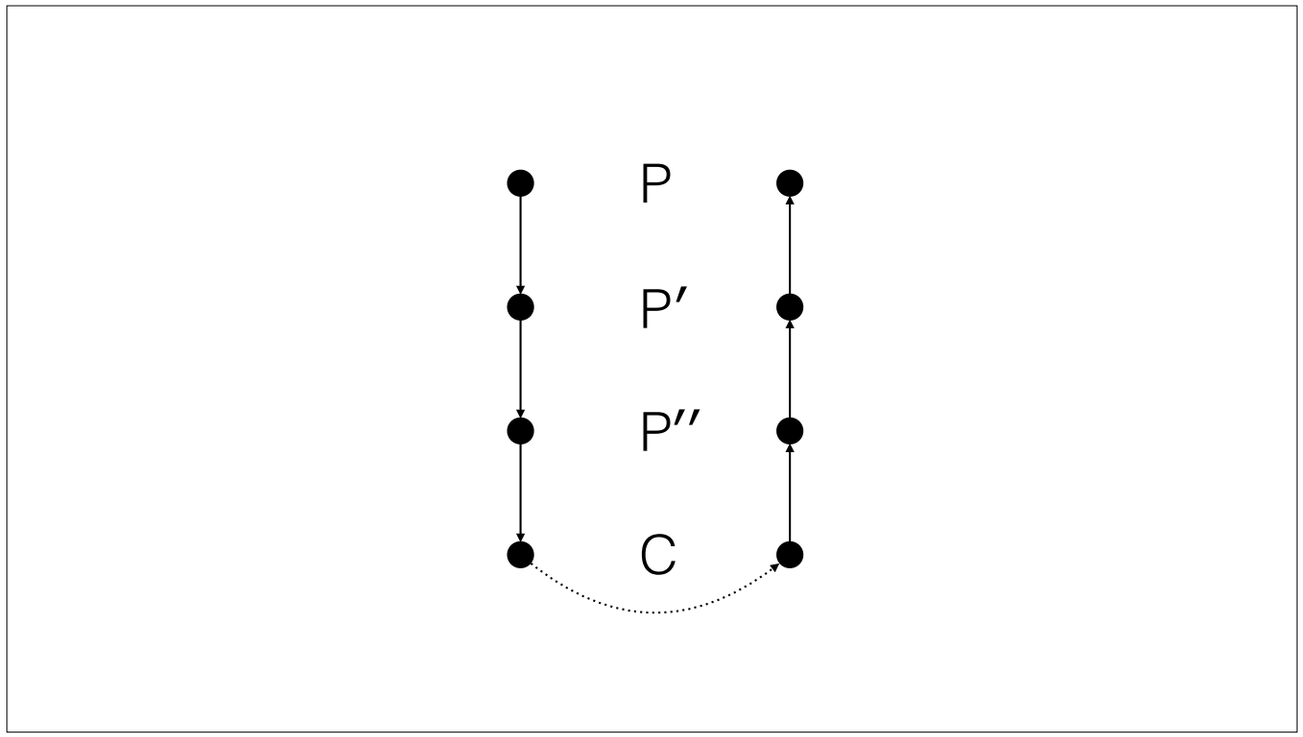
Informal refinement is top-down. Proof is bottom-up.



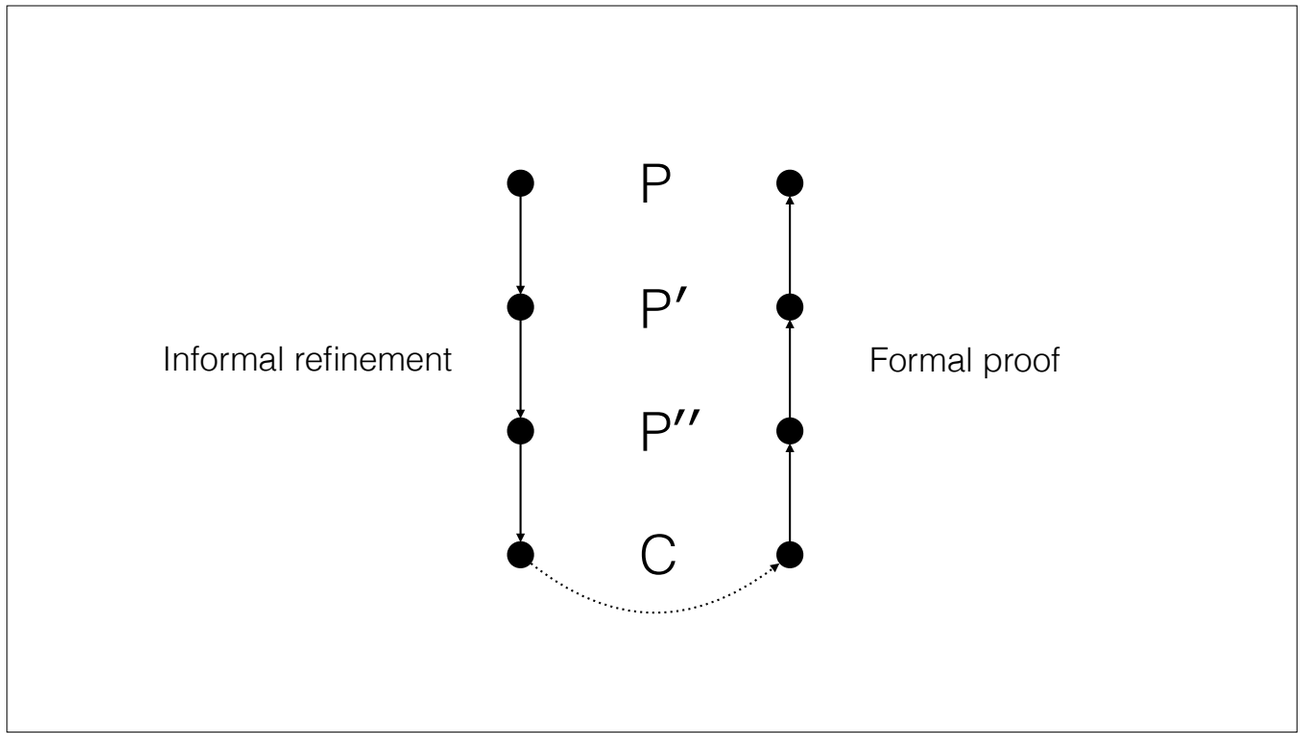
Informal refinement is top-down. Proof is bottom-up.



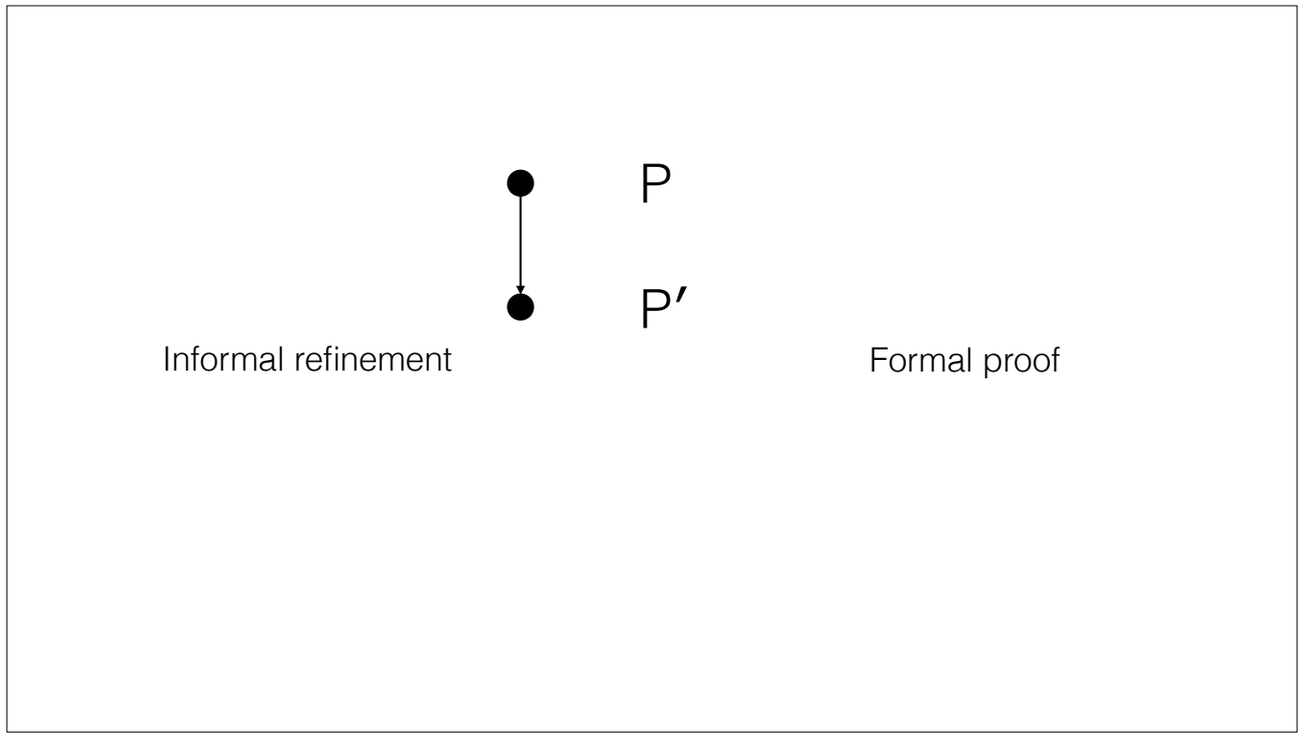
Informal refinement is top-down. Proof is bottom-up.



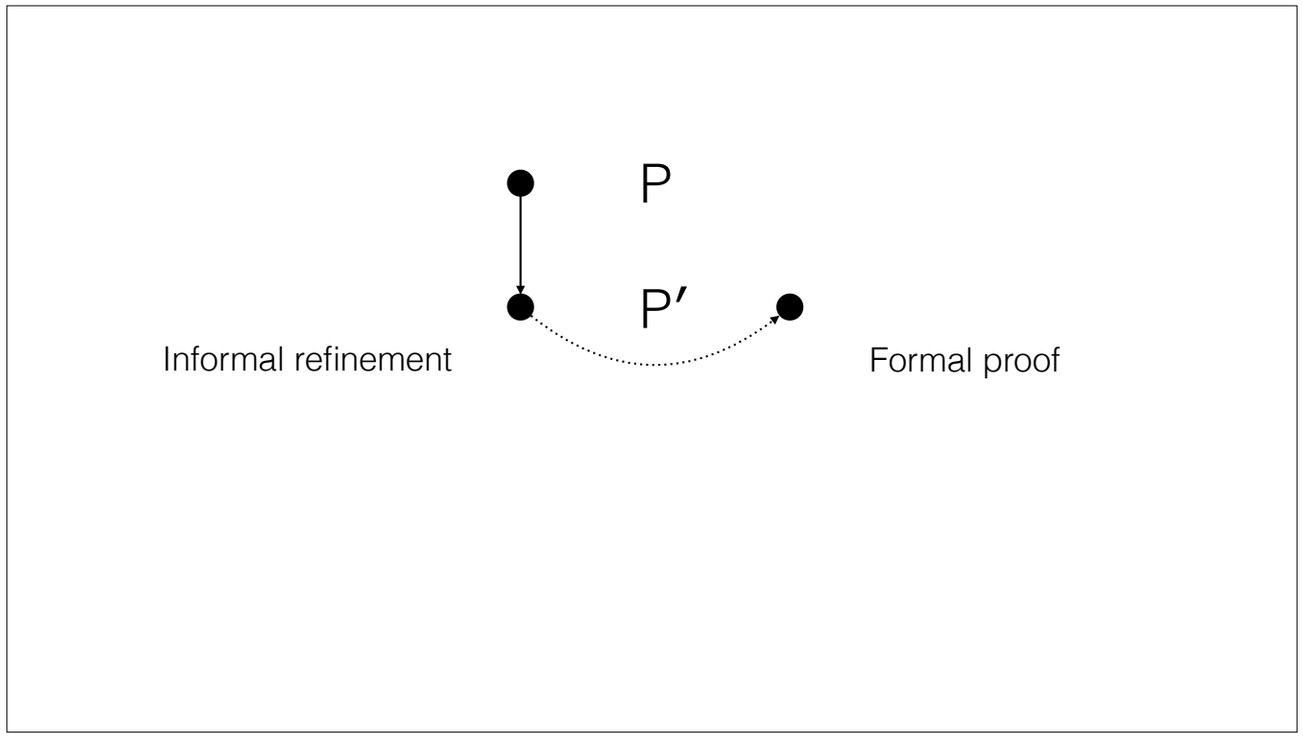
Informal refinement is top-down. Proof is bottom-up.



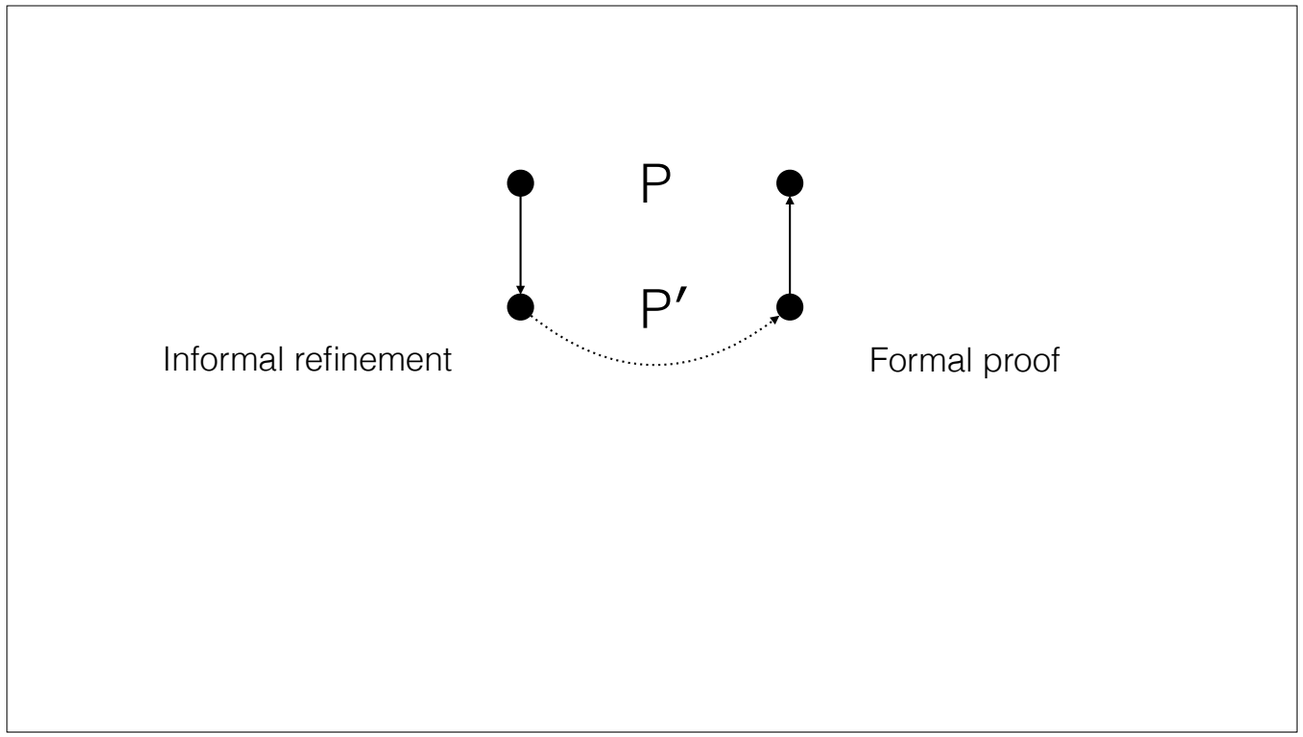
Informal refinement is top-down. Proof is bottom-up.



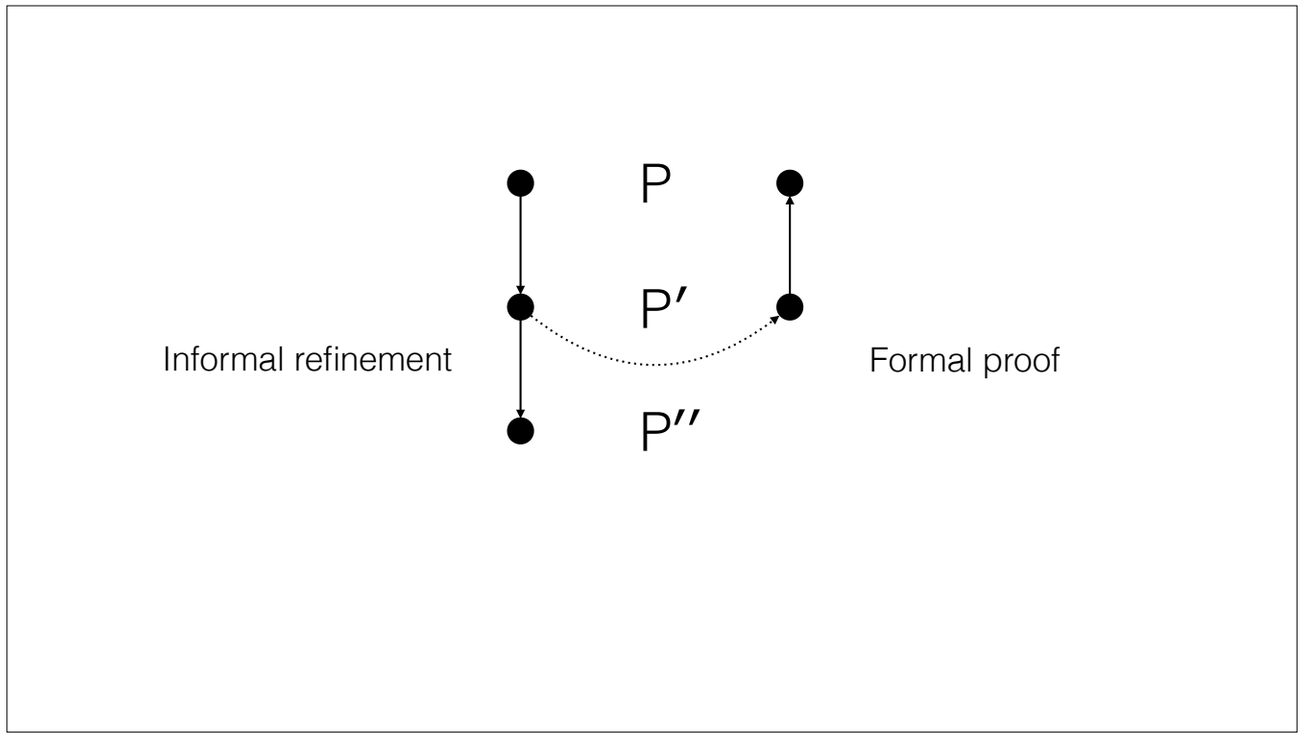
Formalise as we go, by assuming the more refined property, and proving that the original property holds as a result.



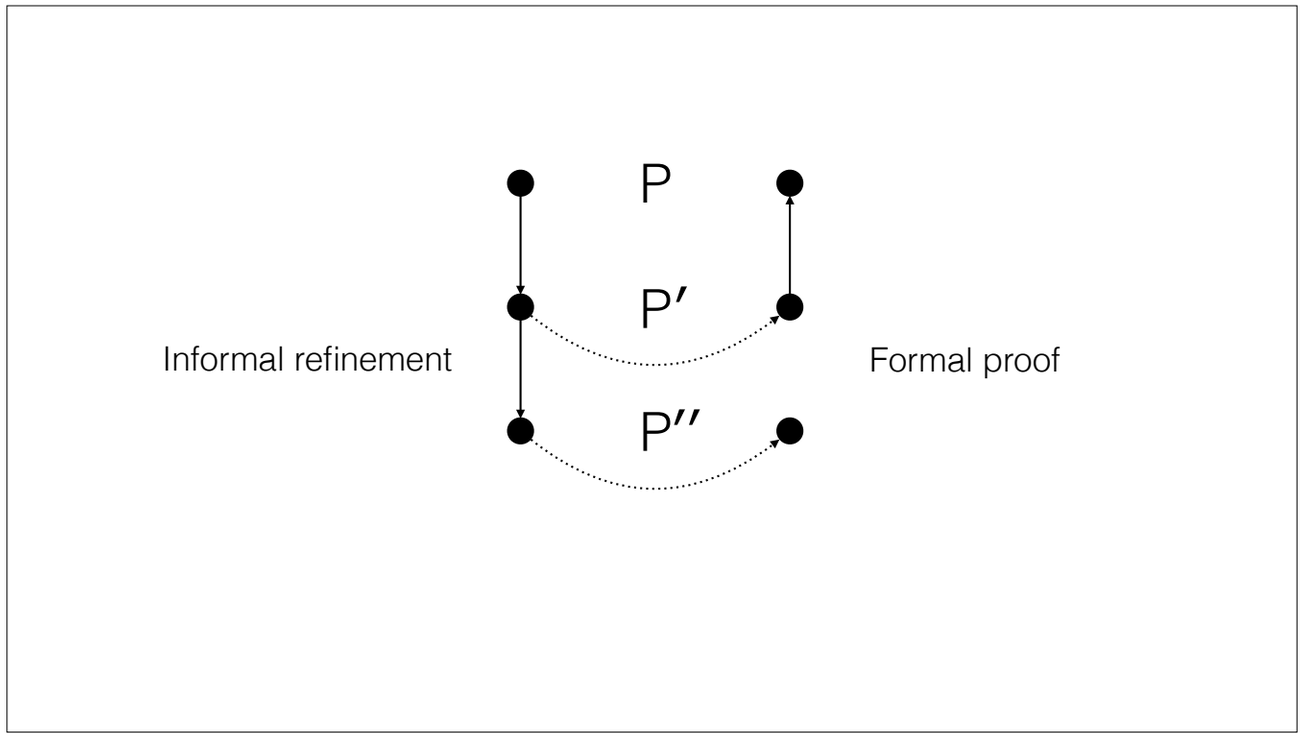
Formalise as we go, by assuming the more refined property, and proving that the original property holds as a result.



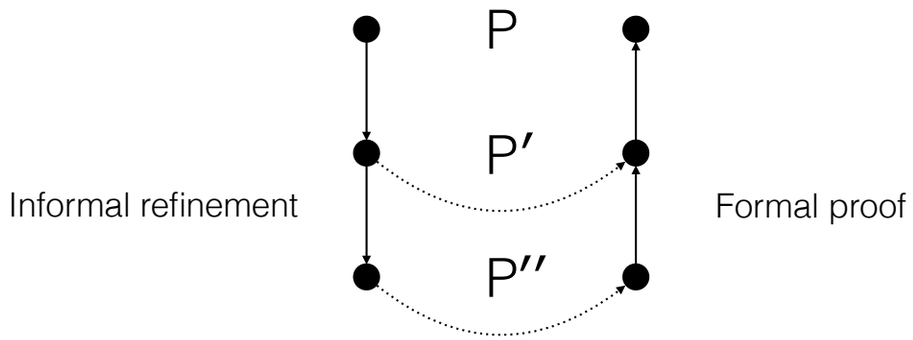
Formalise as we go, by assuming the more refined property, and proving that the original property holds as a result.



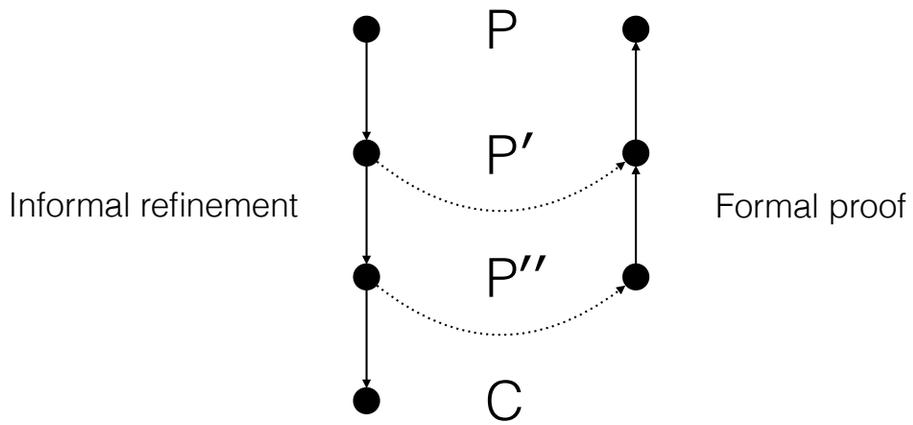
Formalise as we go, by assuming the more refined property, and proving that the original property holds as a result.



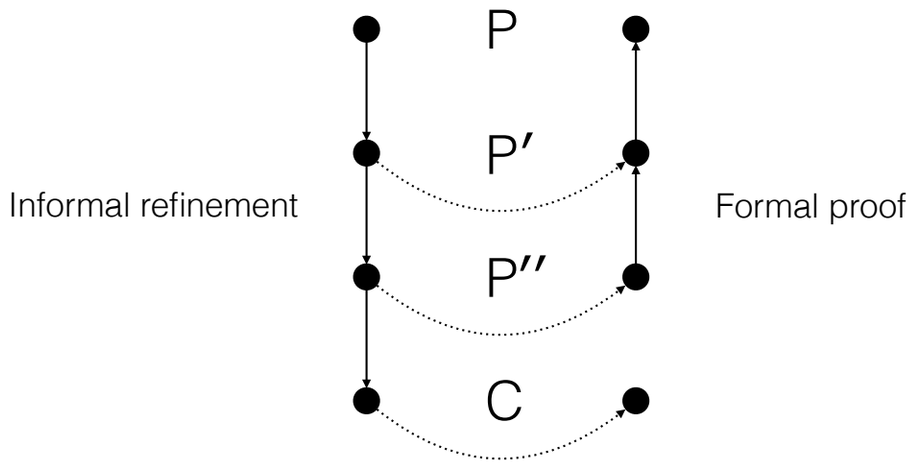
Formalise as we go, by assuming the more refined property, and proving that the original property holds as a result.



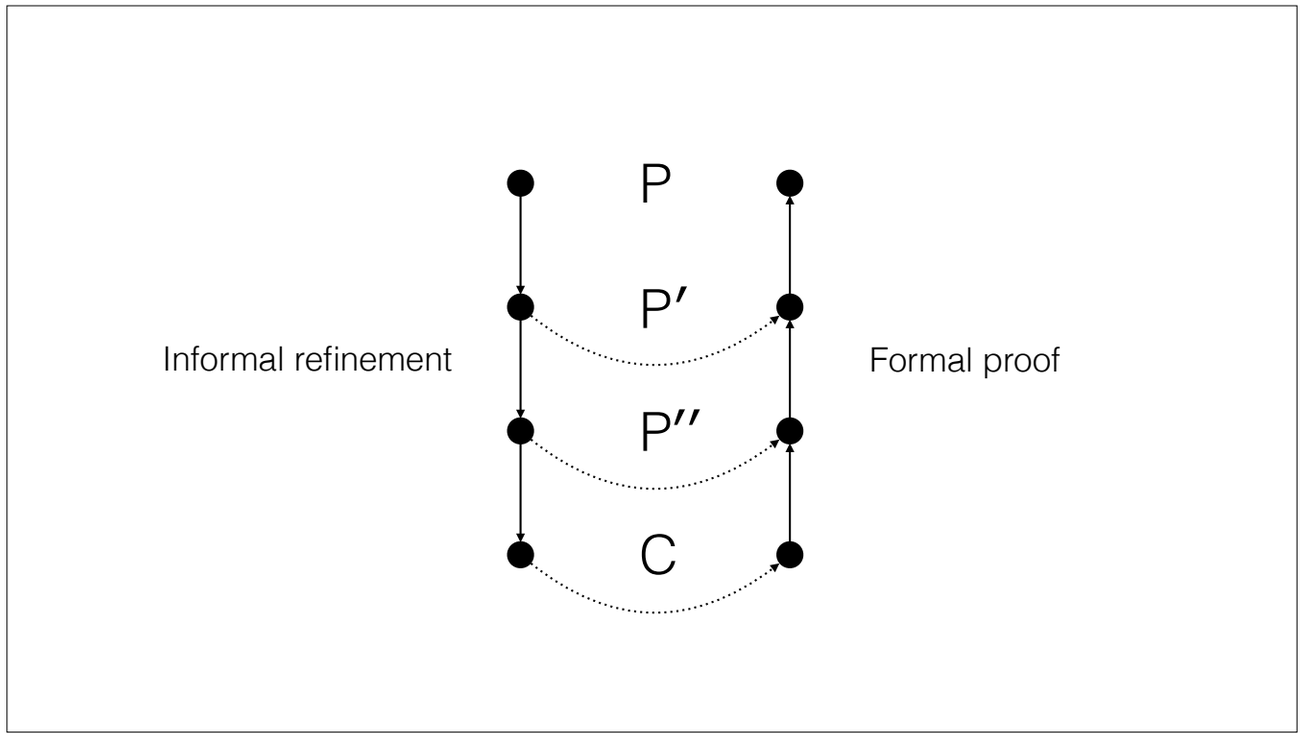
Formalise as we go, by assuming the more refined property, and proving that the original property holds as a result.



Formalise as we go, by assuming the more refined property, and proving that the original property holds as a result.



Formalise as we go, by assuming the more refined property, and proving that the original property holds as a result.



Formalise as we go, by assuming the more refined property, and proving that the original property holds as a result.

Formalising as we go:

- Checks the validity of informal refinements.
- Provides clarity for identifying the next refinement.

We'll use locales to keep track of assumptions:

- A locale is a named bundle of assumptions about some parameters.

```
locale hats =
  fixes spare :: "nat"
  fixes assigned :: "nat list"
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
```

Informally: there is a spare hat and a list of hats assigned to cats, such that the complete set of hat numbers is the set containing 0 up to the number of assigned hats.

Explain: double quotes, type constructor application, set, cons, significance of set equality.

```
locale hats =
  fixes spare :: "nat"
  fixes assigned :: "nat list"
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
```

Informally: there is a spare hat and a list of hats assigned to cats, such that the complete set of hat numbers is the set containing 0 up to the number of assigned hats.

Explain: double quotes, type constructor application, set, cons, significance of set equality.

```
locale hats =
  fixes spare :: "nat"
  fixes assigned :: "nat list"
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
```

Informally: there is a spare hat and a list of hats assigned to cats, such that the complete set of hat numbers is the set containing 0 up to the number of assigned hats.

Explain: double quotes, type constructor application, set, cons, significance of set equality.

```
locale hats =
  fixes spare :: "nat"
  fixes assigned :: "nat list"
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
```

Informally: there is a spare hat and a list of hats assigned to cats, such that the complete set of hat numbers is the set containing 0 up to the number of assigned hats.

Explain: double quotes, type constructor application, set, cons, significance of set equality.

```
locale hats =
  fixes spare :: "nat"
  fixes assigned :: "nat list"
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
```

Informally: there is a spare hat and a list of hats assigned to cats, such that the complete set of hat numbers is the set containing 0 up to the number of assigned hats.

Explain: double quotes, type constructor application, set, cons, significance of set equality.

```
locale hats =
  fixes spare :: "nat"
  fixes assigned :: "nat list"
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
```

Informally: there is a spare hat and a list of hats assigned to cats, such that the complete set of hat numbers is the set containing 0 up to the number of assigned hats.

Explain: double quotes, type constructor application, set, cons, significance of set equality.

```
locale hats =
  fixes spare :: "nat"
  fixes assigned :: "nat list"
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
```

Informally: there is a spare hat and a list of hats assigned to cats, such that the complete set of hat numbers is the set containing 0 up to the number of assigned hats.

Explain: double quotes, type constructor application, set, cons, significance of set equality.

```
locale hats =
  fixes spare :: "nat"
  fixes assigned :: "nat list"
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
```

Informally: there is a spare hat and a list of hats assigned to cats, such that the complete set of hat numbers is the set containing 0 up to the number of assigned hats.

Explain: double quotes, type constructor application, set, cons, significance of set equality.

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

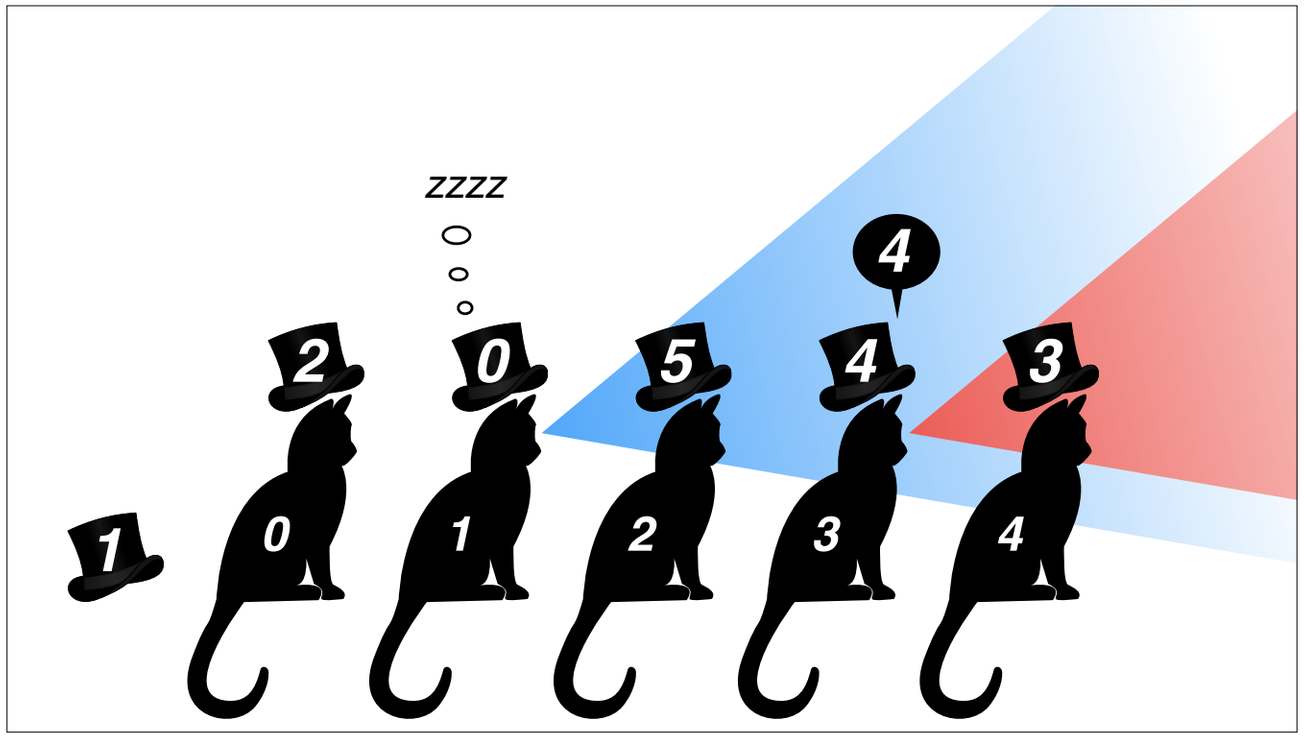
```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

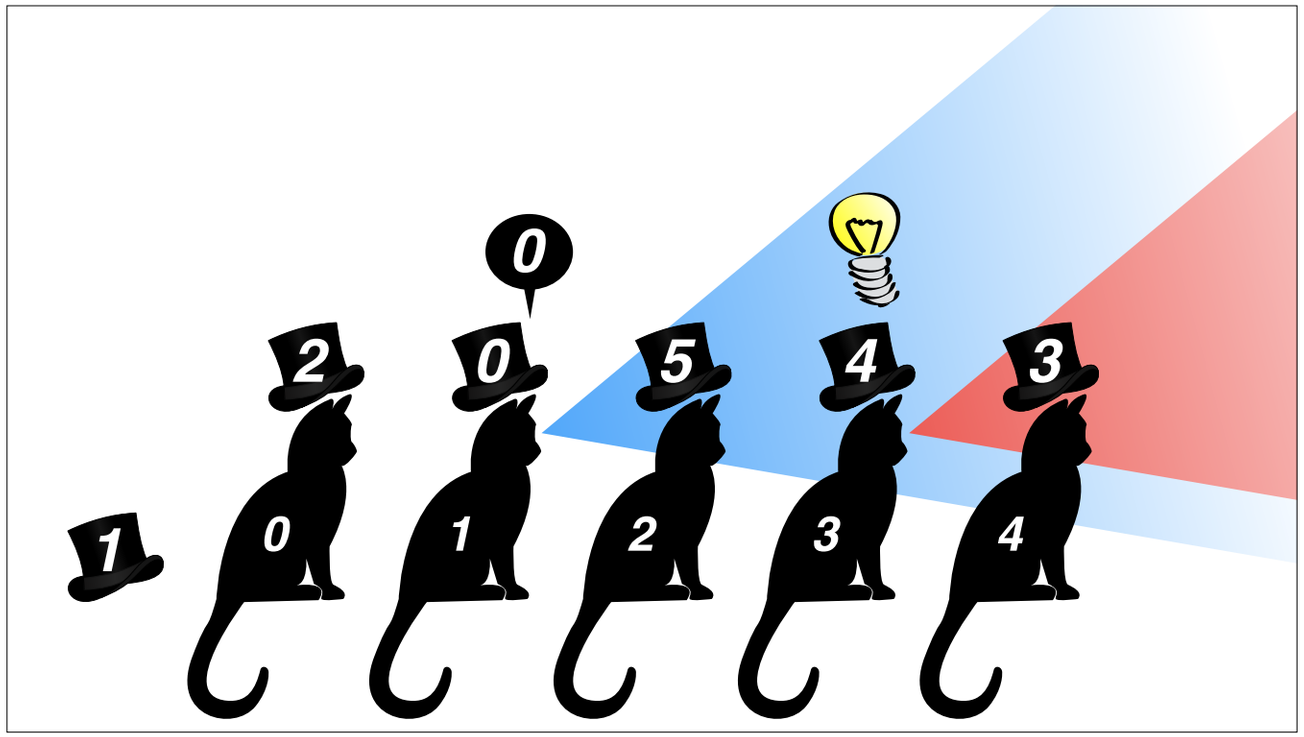
```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  proof -
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by simp
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

```
lemma (in hats) distinct_hats': "distinct (spare # assigned)"
  by (rule card_distinct, subst assign, simp)
```



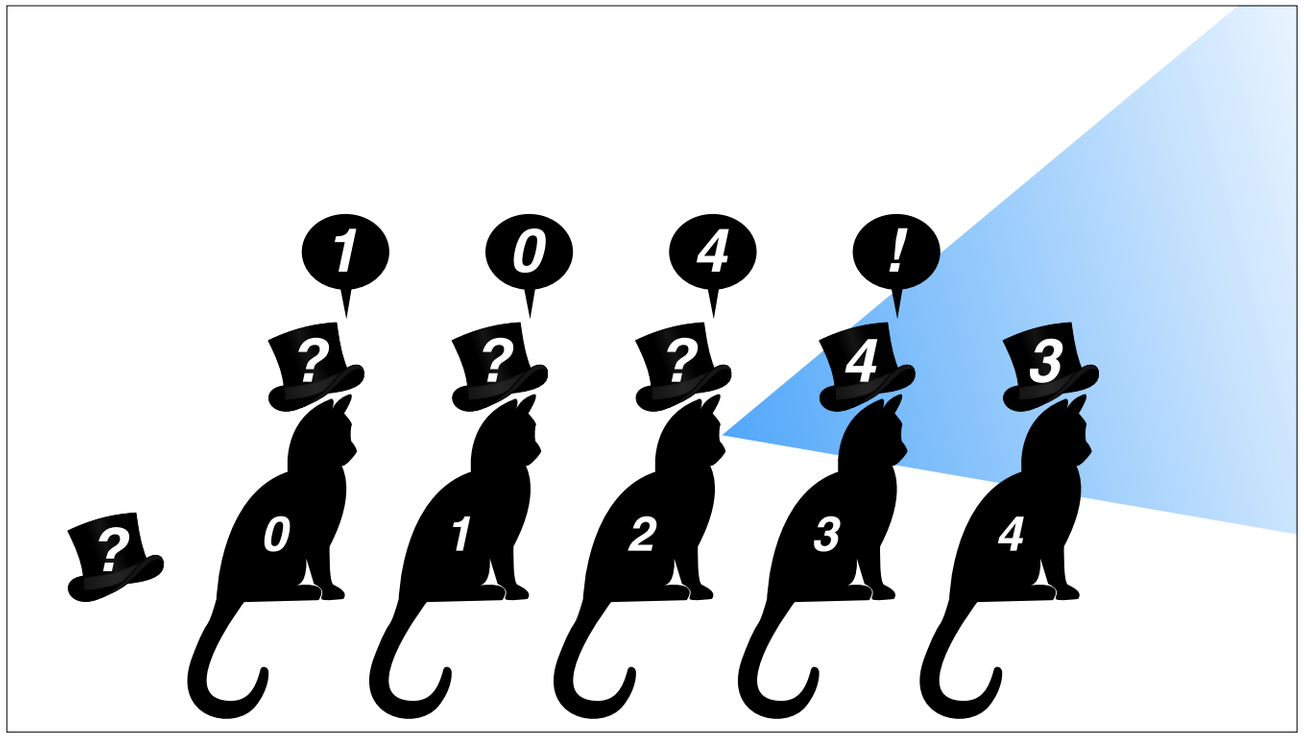
The order in which the cats make their calls is unspecified. We work from back to front, because cats towards the rear always have at least as much information as cats towards the front. Therefore, cats towards the rear cannot learn anything from choices made by cats towards the front.



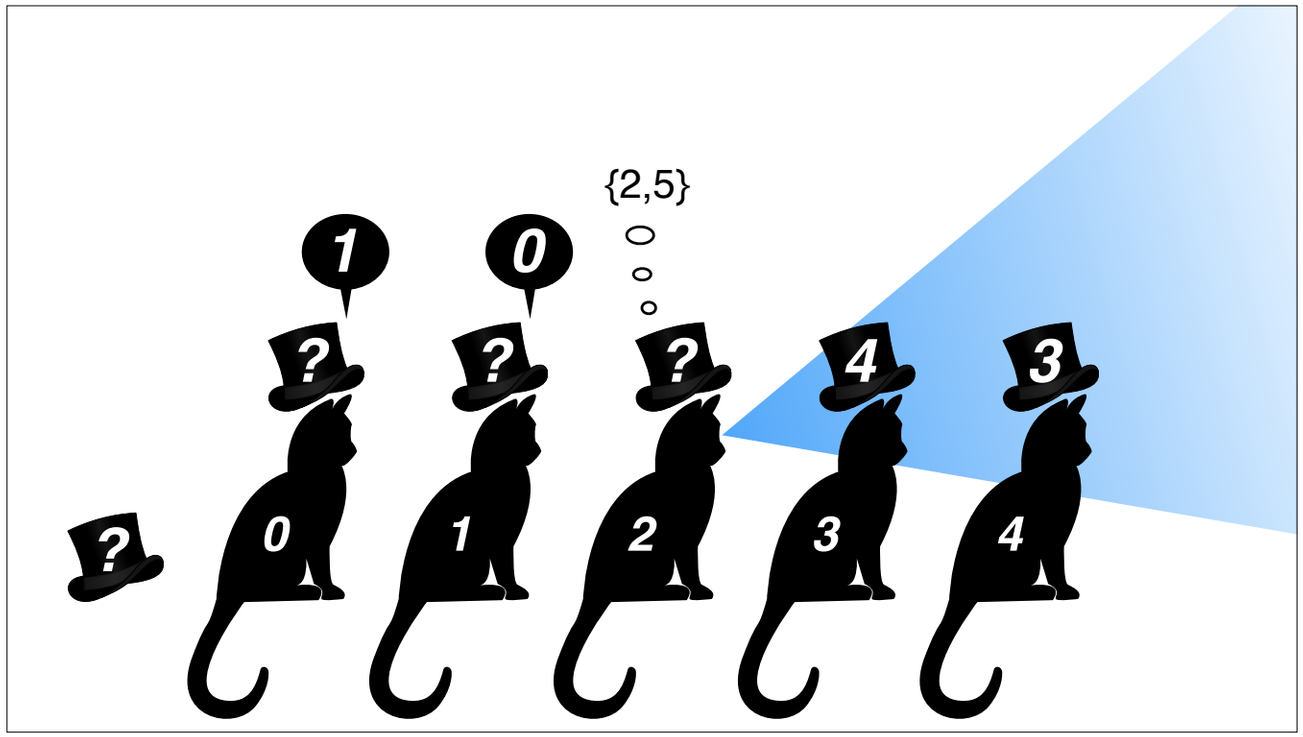
However, cats towards the front *can* learn things from choices made by cats towards the rear, because those choices might encode knowledge of hats that are not visible from the front.

```
locale cats = hats +
  fixes spoken :: "nat list"
  assumes length: "length spoken = length assigned"

definition (in cats) "heard k ≡ take k spoken"
definition (in cats) "seen k ≡ drop (Suc k) assigned"
```

Explain: locale extension, separate locale so we can discharge assumptions separately.



According the rules, a cat may not repeat a number that was already said by a cat behind it. What happens if it says one that it sees in front? Then *both* cats get it wrong. So we can’t allow that.



If a cat excludes the numbers it has heard and seen, it will have a choice between exactly two numbers. Let’s formalise this.

```
definition
  candidates_excluding :: "nat list ⇒ nat list ⇒ nat set"
where
  "candidates_excluding heard seen ≡
    let excluded = heard @ seen in
      {0 .. 1 + length excluded} - set excluded"

definition (in cats)
  "candidates i ≡ candidates_excluding (heard i) (seen i)"
```

```
definition
  candidates_excluding :: "nat list ⇒ nat list ⇒ nat set"
where
  "candidates_excluding heard seen ≡
    let excluded = heard @ seen in
      {0 .. 1 + length excluded} - set excluded"

definition (in cats)
  "candidates i ≡ candidates_excluding (heard i) (seen i)"
```

```
definition
  candidates_excluding :: "nat list ⇒ nat list ⇒ nat set"
where
  "candidates_excluding heard seen ≡
    let excluded = heard @ seen in
      {0 .. 1 + length excluded} - set excluded"

definition (in cats)
  "candidates i ≡ candidates_excluding (heard i) (seen i)"
```

```
definition
  candidates_excluding :: "nat list ⇒ nat list ⇒ nat set"
where
  "candidates_excluding heard seen ≡
    let excluded = heard @ seen in
      {0 .. 1 + length excluded} - set excluded"

definition (in cats)
  "candidates i ≡ candidates_excluding (heard i) (seen i)"
```

```
definition
  candidates_excluding :: "nat list ⇒ nat list ⇒ nat set"
where
  "candidates_excluding heard seen ≡
    let excluded = heard @ seen in
      {0 .. 1 + length excluded} - set excluded"

definition (in cats)
  "candidates i ≡ candidates_excluding (heard i) (seen i)"
```

```
definition
  candidates_excluding :: "nat list ⇒ nat list ⇒ nat set"
where
  "candidates_excluding heard seen ≡
    let excluded = heard @ seen in
      {0 .. 1 + length excluded} - set excluded"

definition (in cats)
  "candidates i ≡ candidates_excluding (heard i) (seen i)"
```

```
lemma (in cats) candidates_i:
  fixes a b i
  defines "view ≡ (a # heard i @ b # seen i)"
  assumes "i < length assigned"
  assumes "set view = {0..length assigned}"
  assumes "distinct view"
  shows "candidates i = {a,b}"
```

```
lemma (in cats) candidates_i:
  fixes a b i
  defines "view ≡ (a # heard i @ b # seen i)"
  assumes "i < length assigned"
  assumes "set view = {0..length assigned}"
  assumes "distinct view"
  shows "candidates i = {a,b}"
```

```
lemma (in cats) candidates_i:
  fixes a b i
  defines "view ≡ (a # heard i @ b # seen i)"
  assumes "i < length assigned"
  assumes "set view = {0..length assigned}"
  assumes "distinct view"
  shows "candidates i = {a,b}"
```

```
lemma (in cats) candidates_i:
  fixes a b i
  defines "view ≡ (a # heard i @ b # seen i)"
  assumes "i < length assigned"
  assumes "set view = {0..length assigned}"
  assumes "distinct view"
  shows "candidates i = {a,b}"
```

```
lemma (in cats) candidates_i:
  fixes a b i
  defines "view ≡ (a # heard i @ b # seen i)"
  assumes "i < length assigned"
  assumes "set view = {0..length assigned}"
  assumes "distinct view"
  shows "candidates i = {a,b}"
```

```
lemma (in cats) candidates_i:
  fixes a b i
  defines "view ≡ (a # heard i @ b # seen i)"
  assumes "i < length assigned"
  assumes "set view = {0..length assigned}"
  assumes "distinct view"
  shows "candidates i = {a,b}"
```

```
lemma (in cats) candidates_i:
  fixes a b i
  defines "view ≡ (a # heard i @ b # seen i)"
  assumes "i < length assigned"
  assumes "set view = {0..length assigned}"
  assumes "distinct view"
  shows "candidates i = {a,b}"
```

```
lemma (in cats) candidates_i:
  fixes a b i
  defines "view ≡ (a # heard i @ b # seen i)"
  assumes "i < length assigned"
  assumes "set view = {0..length assigned}"
  assumes "distinct view"
  shows "candidates i = {a,b}"
```

```
lemma (in cats) candidates_i:
  fixes a b i
  defines "view ≡ (a # heard i @ b # seen i)"
  assumes "i < length assigned"
  assumes "set view = {0..length assigned}"
  assumes "distinct view"
  shows "candidates i = {a,b}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

abbreviation (in cat_0) (input)
  "view_0 ≡ spare # assigned ! 0 # seen 0"

lemma (in cat_0) set_0: "set view_0 = {0..length assigned}"
lemma (in cat_0) distinct_0: "distinct view_0"

lemma (in cat_0) candidates_0:
  "candidates 0 = {spare, assigned ! 0}"
```

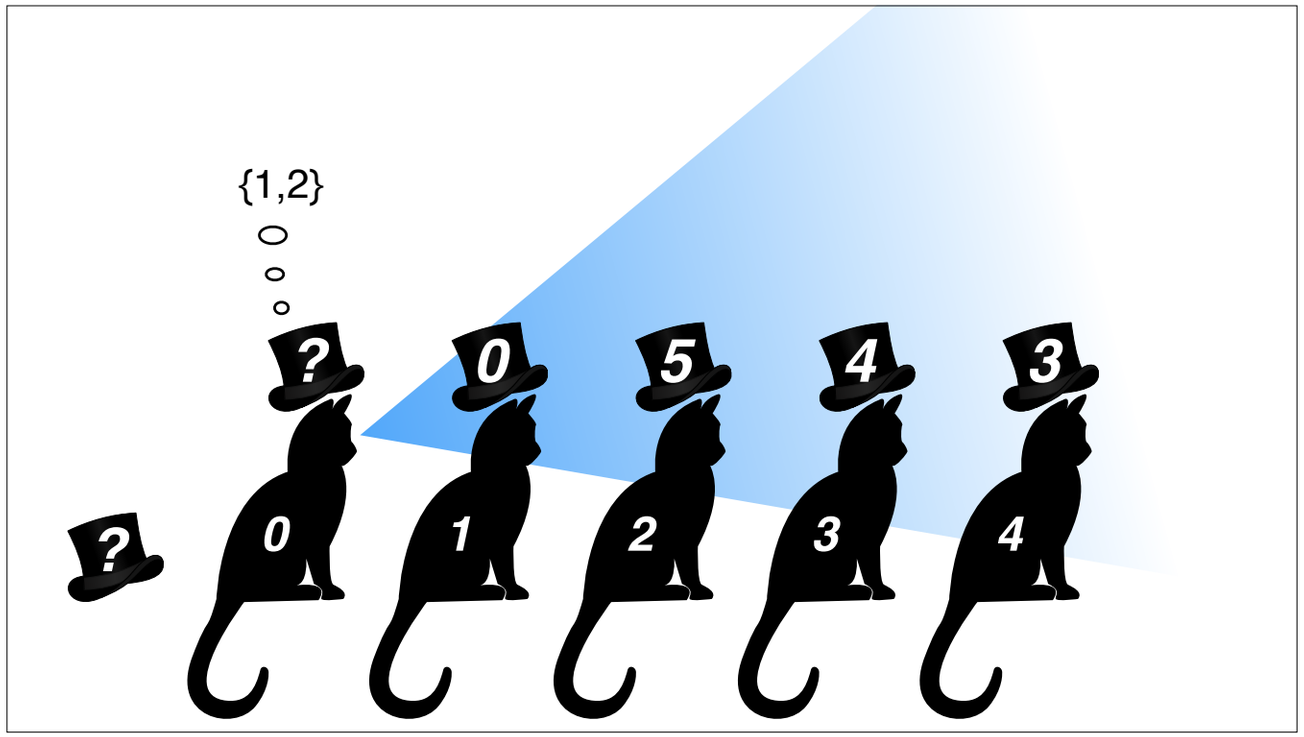
```
definition (in cat_0)
  "rejected ≡ if spoken ! 0 = spare then assigned ! 0 else spare"

abbreviation (in cat_0) (input)
  "view_r ≡ rejected # spoken ! 0 # seen 0"

locale cat_0_spoken = cat_0 +
  assumes spoken_candidate_0: "spoken ! 0 ∈ candidates 0"

lemma (in cat_0_spoken) set_r: "set view_r = {0..length assigned}"
lemma (in cat_0_spoken) distinct_r: "distinct view_r"
```

```
definition (in cat_0)
  "rejected ≡ if spoken ! 0 = spare then assigned ! 0 else spare"

abbreviation (in cat_0) (input)
  "view_r ≡ rejected # spoken ! 0 # seen 0"

locale cat_0_spoken = cat_0 +
  assumes spoken_candidate_0: "spoken ! 0 ∈ candidates 0"

lemma (in cat_0_spoken) set_r: "set view_r = {0..length assigned}"
lemma (in cat_0_spoken) distinct_r: "distinct view_r"
```

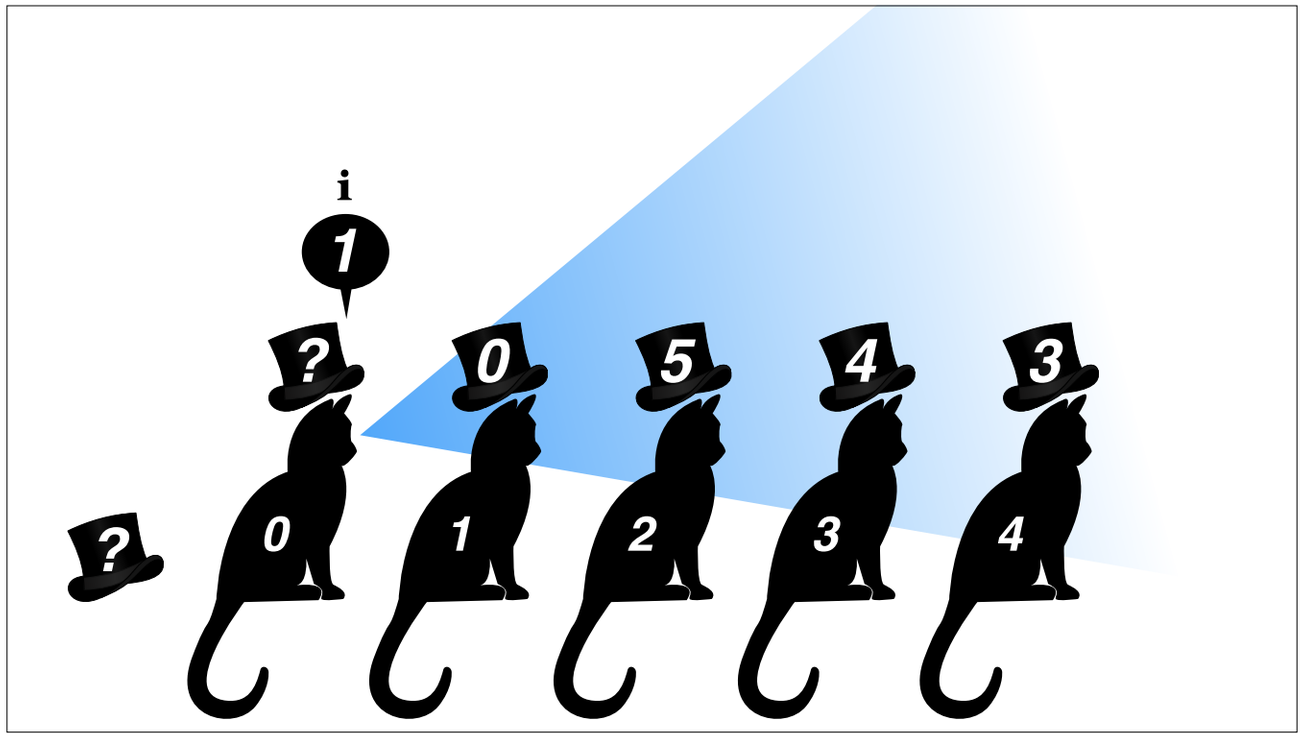
```
definition (in cat_0)
  "rejected ≡ if spoken ! 0 = spare then assigned ! 0 else spare"

abbreviation (in cat_0) (input)
  "view_r ≡ rejected # spoken ! 0 # seen 0"

locale cat_0_spoken = cat_0 +
  assumes spoken_candidate_0: "spoken ! 0 ∈ candidates 0"

lemma (in cat_0_spoken) set_r: "set view_r = {0..length assigned}"
lemma (in cat_0_spoken) distinct_r: "distinct view_r"
```

```
definition (in cat_0)
  "rejected ≡ if spoken ! 0 = spare then assigned ! 0 else spare"

abbreviation (in cat_0) (input)
  "view_r ≡ rejected # spoken ! 0 # seen 0"

locale cat_0_spoken = cat_0 +
  assumes spoken_candidate_0: "spoken ! 0 ∈ candidates 0"

lemma (in cat_0_spoken) set_r: "set view_r = {0..length assigned}"
lemma (in cat_0_spoken) distinct_r: "distinct view_r"
```

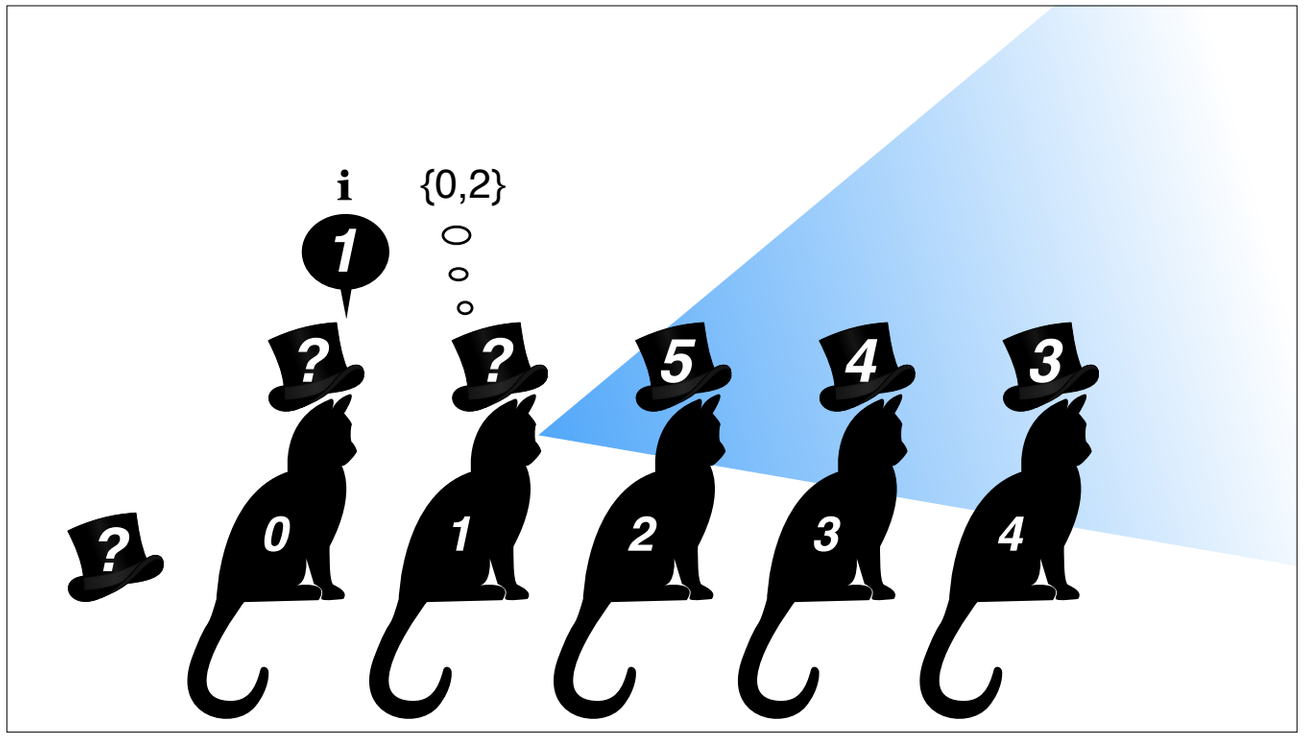
```
definition (in cat_0)
  "rejected ≡ if spoken ! 0 = spare then assigned ! 0 else spare"

abbreviation (in cat_0) (input)
  "view_r ≡ rejected # spoken ! 0 # seen 0"

locale cat_0_spoken = cat_0 +
  assumes spoken_candidate_0: "spoken ! 0 ∈ candidates 0"

lemma (in cat_0_spoken) set_r: "set view_r = {0..length assigned}"
lemma (in cat_0_spoken) distinct_r: "distinct view_r"
```

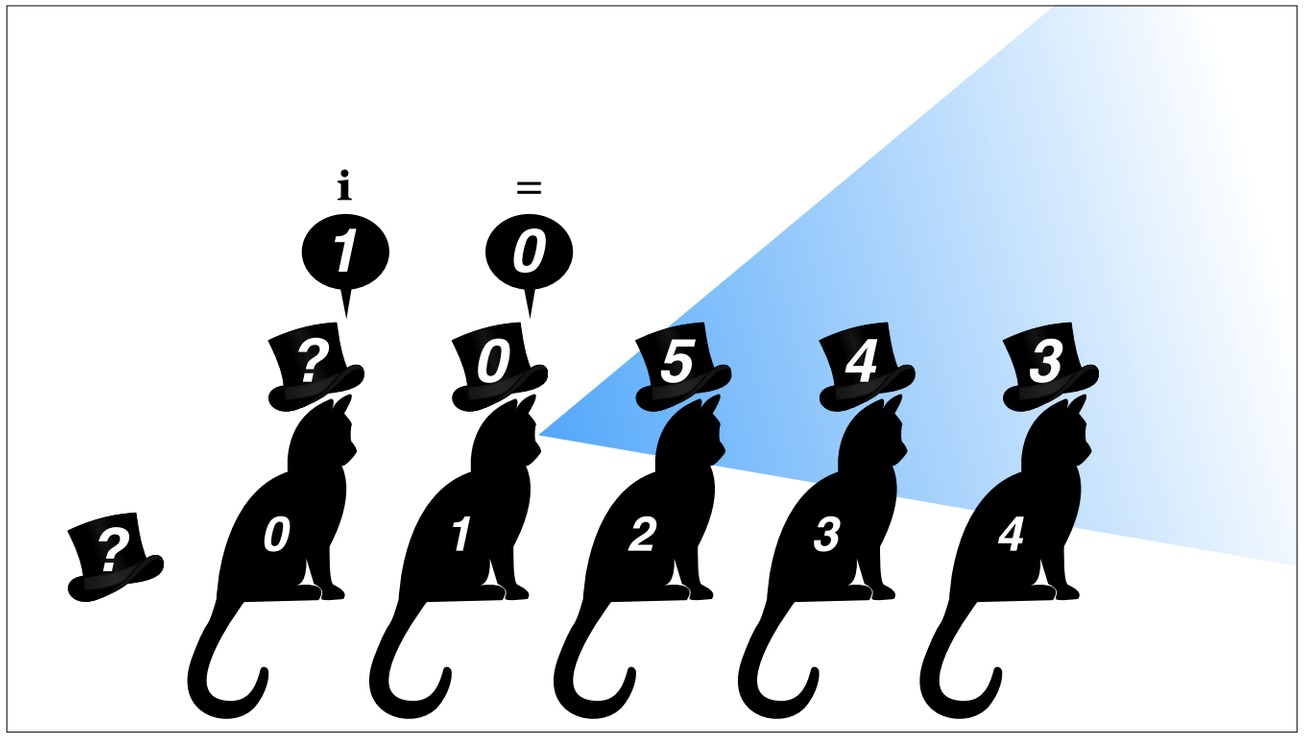


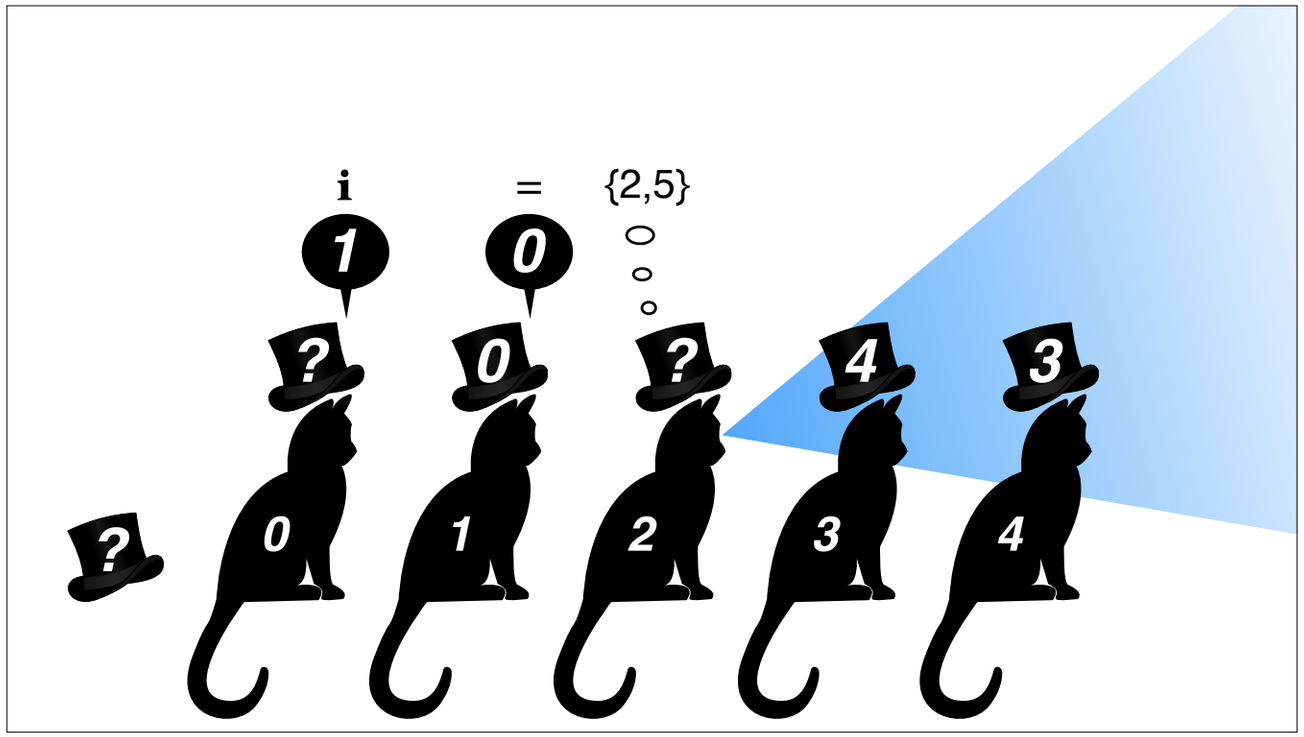
No cat knows the rearmost cat’s number. We must assume the worst: that cat 0 got it wrong. Therefore, we must ensure that all the other cats get it right.

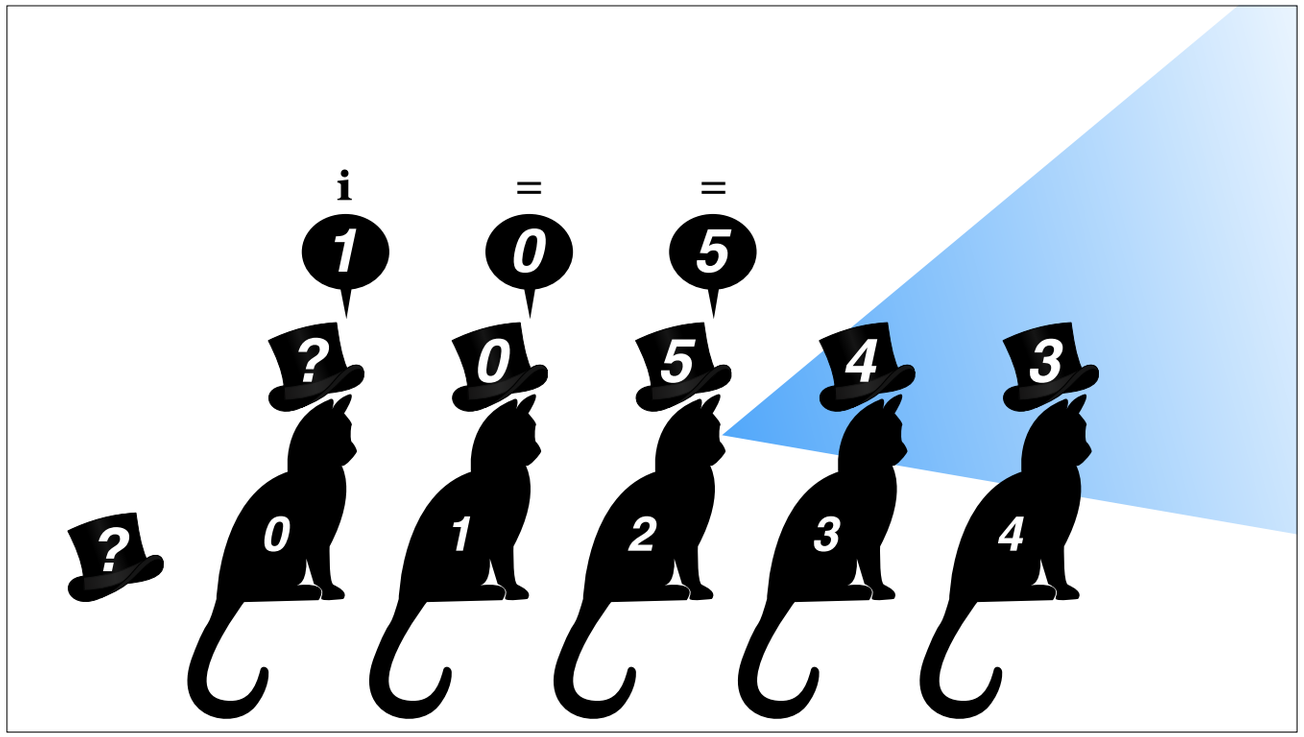


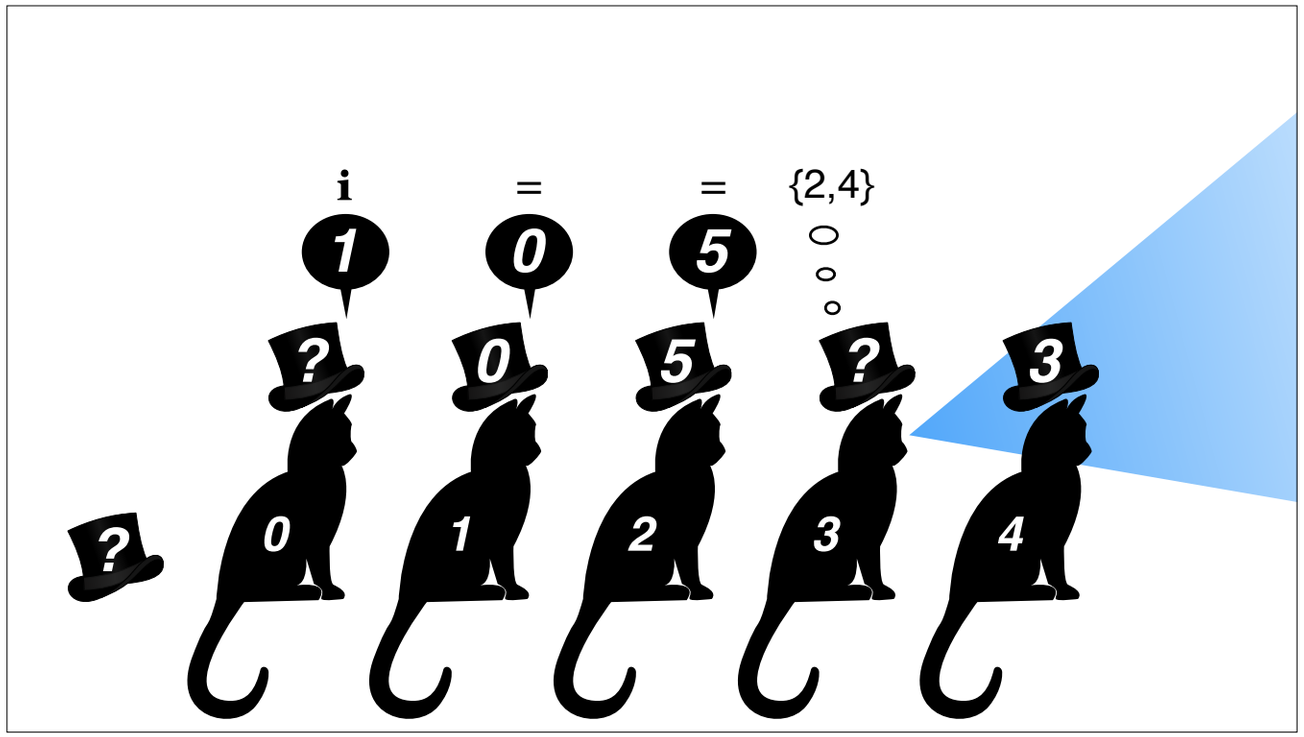
The job of the rearmost cat is not to guess it’s assigned hat number, but to pass information to the other cats.

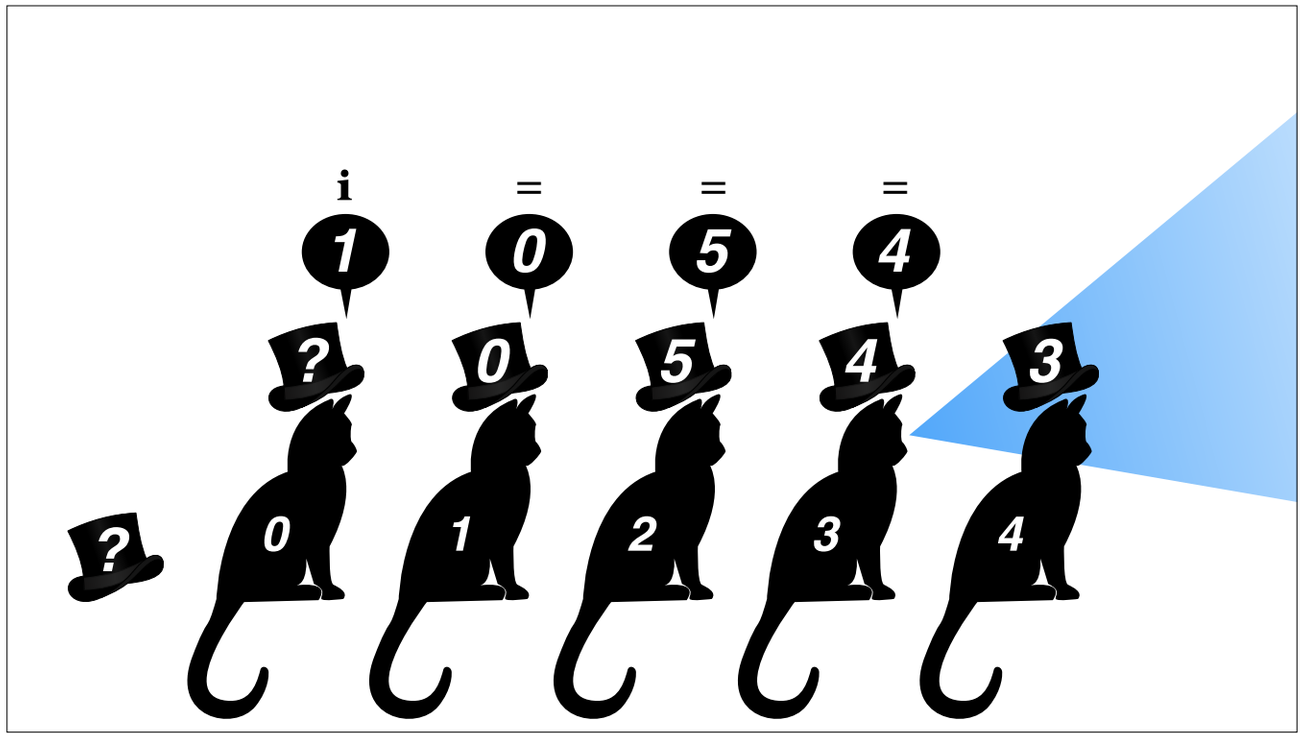

i
{0,2}
1
?
?
5
4
3
?
0
1
2
3
4

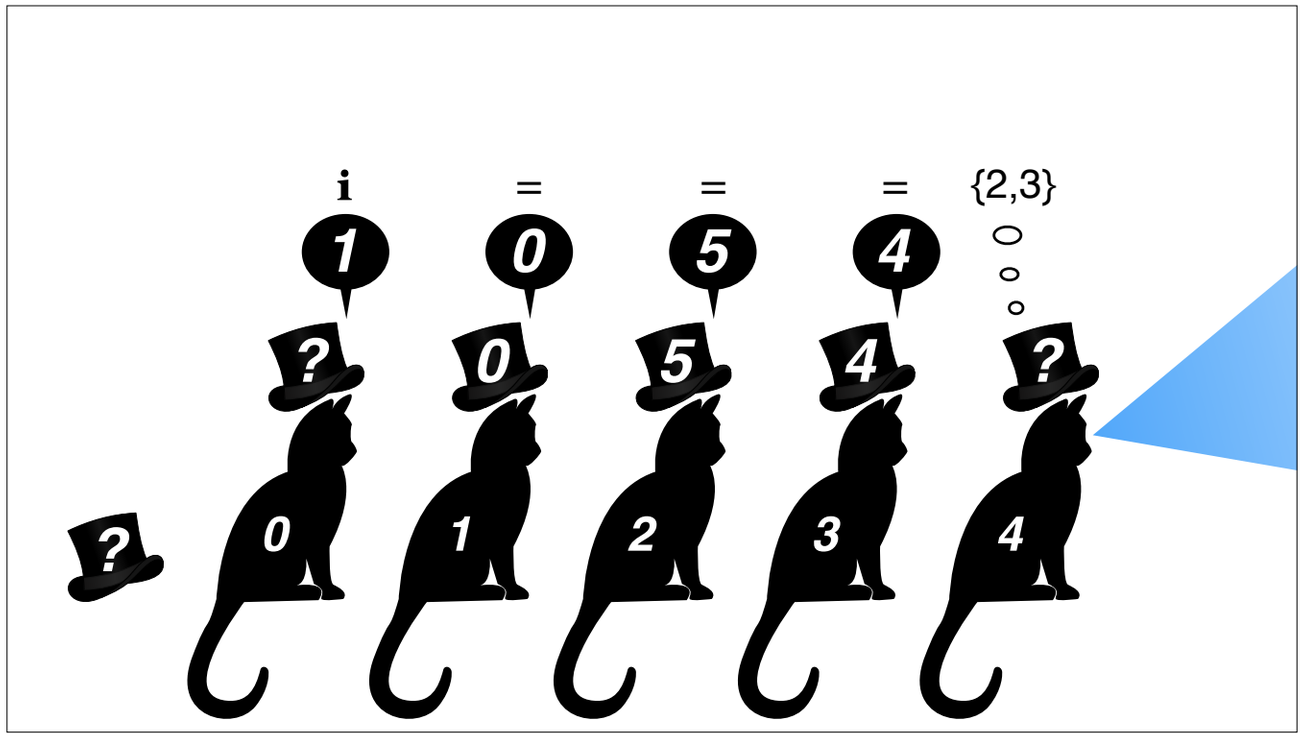

i
=
1
0
?
0
5
4
3
?
0
1
2
3
4

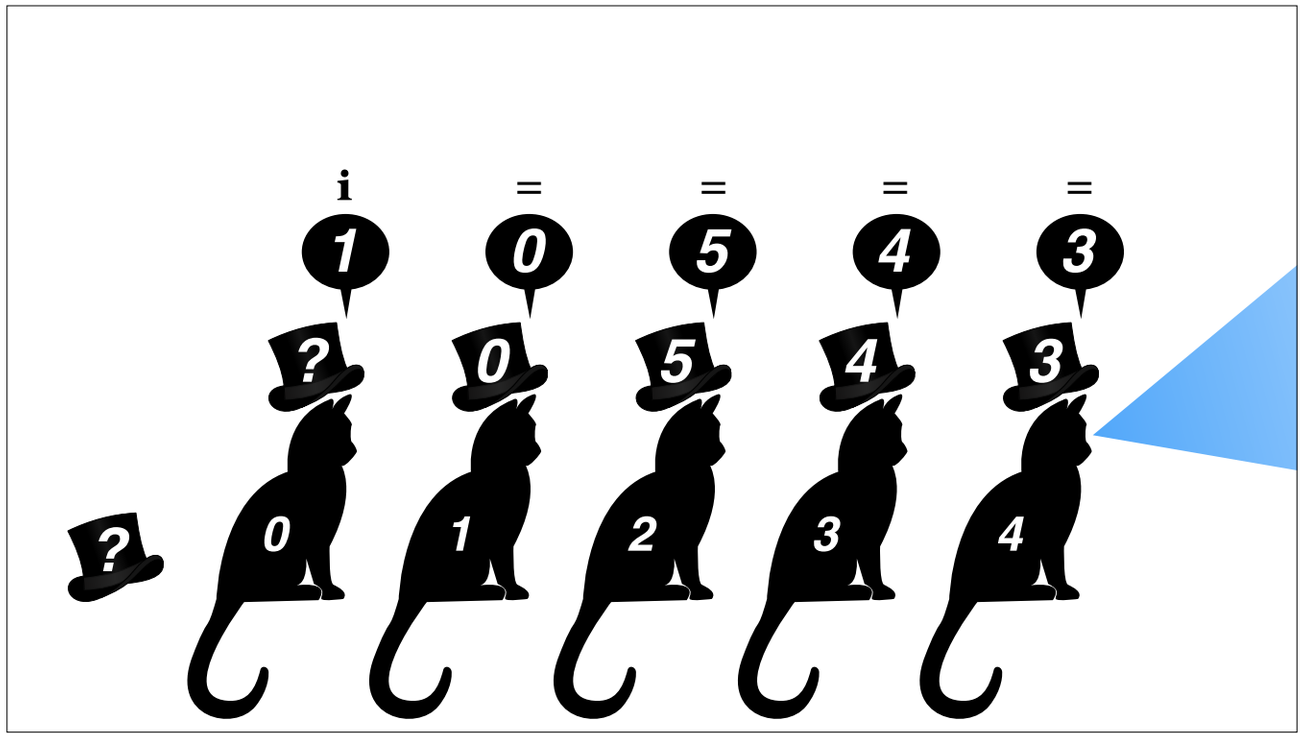

i
=
{2,5}
1
0
?
0
?
4
3
?
0
1
2
3
4


i
=
=
1
0
5
?
0
5
4
3
?
0
1
2
3
4


i
=
=
{2,4}
1
0
5
?
0
5
?
3
?
0
1
2
3
4


i
=
=
=
1
0
5
4
?
0
5
4
3
?
0
1
2
3
4


i
=
=
=
{2,3}
1
0
5
4
?
0
5
4
?
?
0
1
2
3
4


i
=
=
=
=
1
0
5
4
3
?
0
5
4
3
?
0
1
2
3
4

Mathematical induction

- Because all these steps are the same, we can do them all at once.
- To prove that all cats do the right thing, prove that an arbitrary cat *k* does the right thing *if* all preceding cats have done the right thing.

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
```

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀i ∈ {1 ..< k}. spoken ! i = assigned ! i"

lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
  apply (induct k rule: nat_less_induct)
  apply (rule assms)
  apply (unfold_locales)
  by auto
```

```
lemma (in cat_k) heard_k:
  "heard k = spoken ! 0 # map (op ! assigned) [Suc 0 ..< k]"
```

```
lemma (in cat_k) heard_k:
  "heard k = spoken ! 0 # map (op ! assigned) [Suc 0 ..< k]"
```

In any solution, every cat's candidates include:

- The hat which cat 0 rejected.
- The cat's own assigned hat.

To prove this for cat k, we need to talk about the rejected hat in the ***`cat_k`*** locale.

But we defined it in the ***`cat_0`*** locale.

Locale interpretation

- Make the *consequences* of some locale available in some other context.
- Must *prove* that the *assumptions* of the locale hold in that context.

```
sublocale cat_k < cat_0
  using k_max by unfold_locales auto
```

```
abbreviation (in cat_k) (input)
  "view_k ≡ rejected # heard k @ assigned ! k # seen k"

lemma (in cat_k) view_eq: "view_k = view_r"

locale cat_k_view = cat_k + cat_0_spoken

lemma (in cat_k_view) set_k: "set view_k = {0..length assigned}"
lemma (in cat_k_view) distinct_k: "distinct view_k"

lemma (in cat_k_view) candidates_k:
  "candidates k = {rejected, assigned ! k}"
```

```
abbreviation (in cat_k) (input)
  "view_k ≡ rejected # heard k @ assigned ! k # seen k"

lemma (in cat_k) view_eq: "view_k = view_r"

locale cat_k_view = cat_k + cat_0_spoken

lemma (in cat_k_view) set_k: "set view_k = {0..length assigned}"
lemma (in cat_k_view) distinct_k: "distinct view_k"

lemma (in cat_k_view) candidates_k:
  "candidates k = {rejected, assigned ! k}"
```

```
abbreviation (in cat_k) (input)
  "view_k ≡ rejected # heard k @ assigned ! k # seen k"

lemma (in cat_k) view_eq: "view_k = view_r"

locale cat_k_view = cat_k + cat_0_spoken

lemma (in cat_k_view) set_k: "set view_k = {0..length assigned}"
lemma (in cat_k_view) distinct_k: "distinct view_k"

lemma (in cat_k_view) candidates_k:
  "candidates k = {rejected, assigned ! k}"
```

```
abbreviation (in cat_k) (input)
  "view_k ≡ rejected # heard k @ assigned ! k # seen k"

lemma (in cat_k) view_eq: "view_k = view_r"

locale cat_k_view = cat_k + cat_0_spoken

lemma (in cat_k_view) set_k: "set view_k = {0..length assigned}"
lemma (in cat_k_view) distinct_k: "distinct view_k"

lemma (in cat_k_view) candidates_k:
  "candidates k = {rejected, assigned ! k}"
```

```
abbreviation (in cat_k) (input)
  "view_k ≡ rejected # heard k @ assigned ! k # seen k"

lemma (in cat_k) view_eq: "view_k = view_r"

locale cat_k_view = cat_k + cat_0_spoken

lemma (in cat_k_view) set_k: "set view_k = {0..length assigned}"
lemma (in cat_k_view) distinct_k: "distinct view_k"

lemma (in cat_k_view) candidates_k:
  "candidates k = {rejected, assigned ! k}"
```

```
abbreviation (in cat_k) (input)
  "view_k ≡ rejected # heard k @ assigned ! k # seen k"

lemma (in cat_k) view_eq: "view_k = view_r"

locale cat_k_view = cat_k + cat_0_spoken

lemma (in cat_k_view) set_k: "set view_k = {0..length assigned}"
lemma (in cat_k_view) distinct_k: "distinct view_k"

lemma (in cat_k_view) candidates_k:
  "candidates k = {rejected, assigned ! k}"
```

```
abbreviation (in cat_k) (input)
  "view_k ≡ rejected # heard k @ assigned ! k # seen k"

lemma (in cat_k) view_eq: "view_k = view_r"

locale cat_k_view = cat_k + cat_0_spoken

lemma (in cat_k_view) set_k: "set view_k = {0..length assigned}"
lemma (in cat_k_view) distinct_k: "distinct view_k"

lemma (in cat_k_view) candidates_k:
  "candidates k = {rejected, assigned ! k}"
```

```
abbreviation (in cat_k) (input)
  "view_k ≡ rejected # heard k @ assigned ! k # seen k"

lemma (in cat_k) view_eq: "view_k = view_r"

locale cat_k_view = cat_k + cat_0_spoken

lemma (in cat_k_view) set_k: "set view_k = {0..length assigned}"
lemma (in cat_k_view) distinct_k: "distinct view_k"

lemma (in cat_k_view) candidates_k:
  "candidates k = {rejected, assigned ! k}"
```

```
abbreviation (in cat_k) (input)
  "view_k ≡ rejected # heard k @ assigned ! k # seen k"

lemma (in cat_k) view_eq: "view_k = view_r"

locale cat_k_view = cat_k + cat_0_spoken

lemma (in cat_k_view) set_k: "set view_k = {0..length assigned}"
lemma (in cat_k_view) distinct_k: "distinct view_k"

lemma (in cat_k_view) candidates_k:
  "candidates k = {rejected, assigned ! k}"
```

Our goal reduces to this:

- Ensure cat *k* rejects the same hat as cat 0.

How to do this:

- Stop thinking about choosing particular numbers for particular cats.
- Start thinking about classifying orderings of the complete set of hats.

Good:

```
rejected # heard k @ assigned ! k # seen k
rejected # spoken ! 0 @ seen 0
```

Bad:

```
assigned ! k # heard k @ rejected # seen k
spoken ! 0 # rejected @ seen 0
```

Idea:

- Assume a bool function which inverts when we swap anything with the first element.

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
     parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
locale classifier =
  fixes parity :: "nat list ⇒ bool"
  assumes parity_swap_first:
    "⋀a heard b seen. distinct (a # heard @ b # seen) ⟹
      parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

definition (in classifier)
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates_excluding heard seen) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

```
primrec (in classifier)
  choices' :: "nat list ⇒ nat list ⇒ nat list"
where
  "choices' heard [] = []"
| "choices' heard (_ # seen)
    = (let c = choice heard seen in c # choices' (heard @ [c]) seen)"

definition (in classifier) "choices ≡ choices' []"

lemma (in classifier) choices:
  assumes "i < length assigned"
  assumes "spoken = choices assigned"
  shows "spoken ! i = choice (take i spoken) (drop (Suc i) assigned)"
```

```
primrec (in classifier)
  choices' :: "nat list ⇒ nat list ⇒ nat list"
where
  "choices' heard [] = []"
| "choices' heard (_ # seen)
    = (let c = choice heard seen in c # choices' (heard @ [c]) seen)"

definition (in classifier) "choices ≡ choices' []"

lemma (in classifier) choices:
  assumes "i < length assigned"
  assumes "spoken = choices assigned"
  shows "spoken ! i = choice (take i spoken) (drop (Suc i) assigned)"
```

```
primrec (in classifier)
  choices' :: "nat list ⇒ nat list ⇒ nat list"
where
  "choices' heard [] = []"
| "choices' heard (_ # seen)
    = (let c = choice heard seen in c # choices' (heard @ [c]) seen)"

definition (in classifier) "choices ≡ choices' []"

lemma (in classifier) choices:
  assumes "i < length assigned"
  assumes "spoken = choices assigned"
  shows "spoken ! i = choice (take i spoken) (drop (Suc i) assigned)"
```

```
primrec (in classifier)
  choices' :: "nat list ⇒ nat list ⇒ nat list"
where
  "choices' heard [] = []"
| "choices' heard (_ # seen)
    = (let c = choice heard seen in c # choices' (heard @ [c]) seen)"

definition (in classifier) "choices ≡ choices' []"

lemma (in classifier) choices:
  assumes "i < length assigned"
  assumes "spoken = choices assigned"
  shows "spoken ! i = choice (take i spoken) (drop (Suc i) assigned)"
```

```
locale hats_parity = hats + classifier

sublocale hats_parity < cats spare assigned "choices assigned"

locale cat_0_parity = hats_parity spare assigned parity
  + cat_0 spare assigned "choices assigned"
  for spare assigned parity

locale cat_k_parity = cat_0_parity spare assigned parity
  + cat_k spare assigned "choices assigned" k
  for spare assigned parity k
```

```
locale hats_parity = hats + classifier

sublocale hats_parity < cats spare assigned "choices assigned"

locale cat_0_parity = hats_parity spare assigned parity
  + cat_0 spare assigned "choices assigned"
  for spare assigned parity

locale cat_k_parity = cat_0_parity spare assigned parity
  + cat_k spare assigned "choices assigned" k
  for spare assigned parity k
```

```
locale hats_parity = hats + classifier

sublocale hats_parity < cats spare assigned "choices assigned"

locale cat_0_parity = hats_parity spare assigned parity
  + cat_0 spare assigned "choices assigned"
  for spare assigned parity

locale cat_k_parity = cat_0_parity spare assigned parity
  + cat_k spare assigned "choices assigned" k
  for spare assigned parity k
```

```
lemma (in cat_0_parity) choice_0:
  "choices assigned ! 0 = (if parity view_0 then assigned ! 0 else spare)"

lemma (in cat_0_parity) parity_r: "parity view_r"
lemma (in cat_k_parity) parity_k: "parity view_k"

sublocale cat_k_parity < cat_k_view spare assigned "choices assigned" k

lemma (in cat_k_parity) choice_k: "choices assigned ! k = assigned ! k"

lemma (in hats_parity) choices_correct:
  "k ∈ {1..<length assigned} ⟹ choices assigned ! k = assigned ! k"
```

```
lemma (in cat_0_parity) choice_0:
  "choices assigned ! 0 = (if parity view_0 then assigned ! 0 else spare)"

lemma (in cat_0_parity) parity_r: "parity view_r"
lemma (in cat_k_parity) parity_k: "parity view_k"

sublocale cat_k_parity < cat_k_view spare assigned "choices assigned" k

lemma (in cat_k_parity) choice_k: "choices assigned ! k = assigned ! k"

lemma (in hats_parity) choices_correct:
  "k ∈ {1..<length assigned} ⟹ choices assigned ! k = assigned ! k"
```

```
lemma (in cat_0_parity) choice_0:
  "choices assigned ! 0 = (if parity view_0 then assigned ! 0 else spare)"

lemma (in cat_0_parity) parity_r: "parity view_r"
lemma (in cat_k_parity) parity_k: "parity view_k"

sublocale cat_k_parity < cat_k_view spare assigned "choices assigned" k

lemma (in cat_k_parity) choice_k: "choices assigned ! k = assigned ! k"

lemma (in hats_parity) choices_correct:
  "k ∈ {1..<length assigned} ⟹ choices assigned ! k = assigned ! k"
```

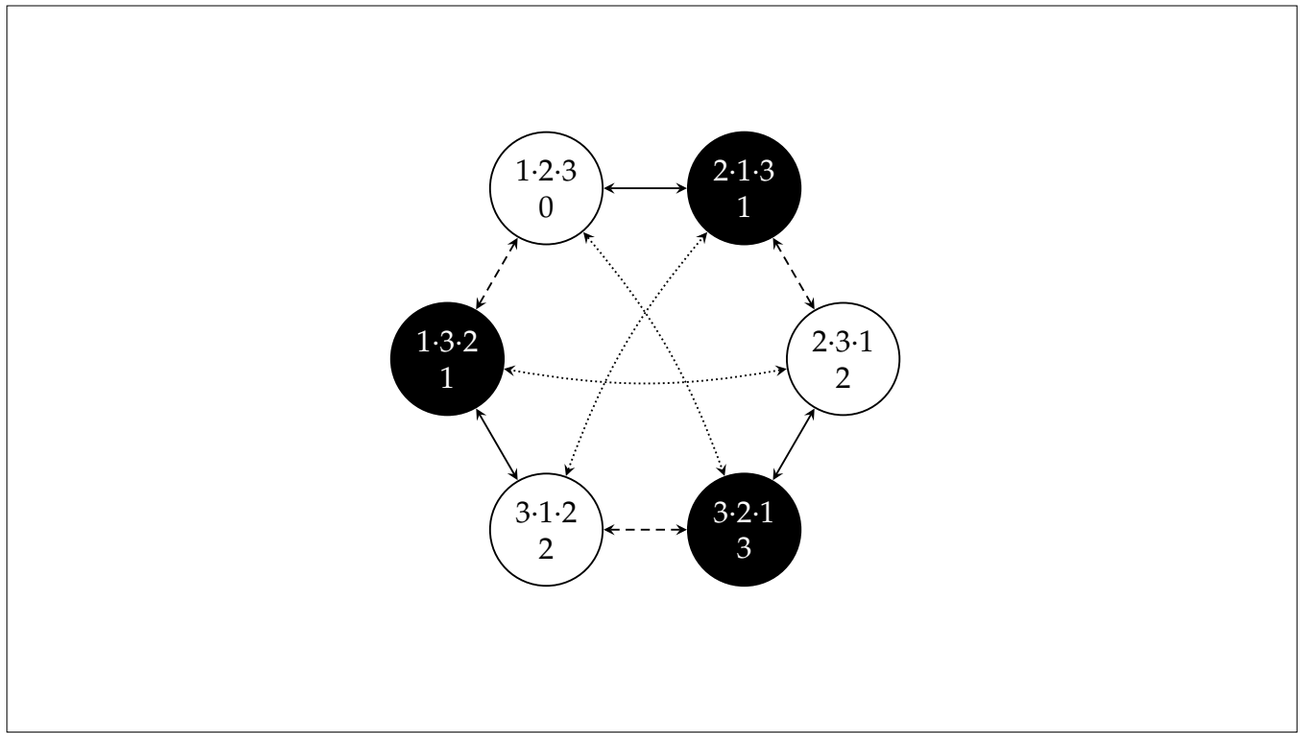
```
lemma (in cat_0_parity) choice_0:
  "choices assigned ! 0 = (if parity view_0 then assigned ! 0 else spare)"

lemma (in cat_0_parity) parity_r: "parity view_r"
lemma (in cat_k_parity) parity_k: "parity view_k"

sublocale cat_k_parity < cat_k_view spare assigned "choices assigned" k

lemma (in cat_k_parity) choice_k: "choices assigned ! k = assigned ! k"

lemma (in hats_parity) choices_correct:
  "k ∈ {1..<length assigned} ⟹ choices assigned ! k = assigned ! k"
```

```
lemma (in cat_0_parity) choice_0:
  "choices assigned ! 0 = (if parity view_0 then assigned ! 0 else spare)"

lemma (in cat_0_parity) parity_r: "parity view_r"
lemma (in cat_k_parity) parity_k: "parity view_k"

sublocale cat_k_parity < cat_k_view spare assigned "choices assigned" k

lemma (in cat_k_parity) choice_k: "choices assigned ! k = assigned ! k"

lemma (in hats_parity) choices_correct:
  "k ∈ {1..<length assigned} ⟹ choices assigned ! k = assigned ! k"
```

```
lemma (in cat_0_parity) choice_0:
  "choices assigned ! 0 = (if parity view_0 then assigned ! 0 else spare)"

lemma (in cat_0_parity) parity_r: "parity view_r"
lemma (in cat_k_parity) parity_k: "parity view_k"

sublocale cat_k_parity < cat_k_view spare assigned "choices assigned" k

lemma (in cat_k_parity) choice_k: "choices assigned ! k = assigned ! k"

lemma (in hats_parity) choices_correct:
  "k ∈ {1..<length assigned} ⟹ choices assigned ! k = assigned ! k"
```

```
lemma (in cat_0_parity) choice_0:
  "choices assigned ! 0 = (if parity view_0 then assigned ! 0 else spare)"

lemma (in cat_0_parity) parity_r: "parity view_r"
lemma (in cat_k_parity) parity_k: "parity view_k"

sublocale cat_k_parity < cat_k_view spare assigned "choices assigned" k

lemma (in cat_k_parity) choice_k: "choices assigned ! k = assigned ! k"

lemma (in hats_parity) choices_correct:
  "k ∈ {1..<length assigned} ⟹ choices assigned ! k = assigned ! k"
```

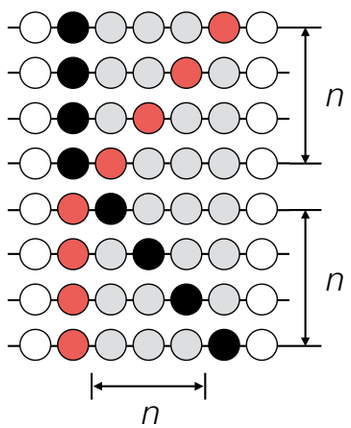
```
lemma (in cat_0_parity) choice_0:
  "choices assigned ! 0 = (if parity view_0 then assigned ! 0 else spare)"

lemma (in cat_0_parity) parity_r: "parity view_r"
lemma (in cat_k_parity) parity_k: "parity view_k"

sublocale cat_k_parity < cat_k_view spare assigned "choices assigned" k

lemma (in cat_k_parity) choice_k: "choices assigned ! k = assigned ! k"

lemma (in hats_parity) choices_correct:
  "k ∈ {1..<length assigned} ⟹ choices assigned ! k = assigned ! k"
```


1·2·3
0
2·1·3
1
1·3·2
1
2·3·1
2
3·1·2
2
3·2·1
3

— Do we have an even number of inversions?

```
primrec
  parity :: "nat list ⇒ bool"
where
  "parity [] = True"
| "parity (x # ys) = (parity ys = even (length [y ← ys. x > y]))"

lemma parity_odd: "parity [1,2,0,5,4,3] = False"
lemma parity_even: "parity [2,1,0,5,4,3] = True"

lemma parity_swap_adj:
  "b ≠ c ⟹ parity (as @ b # c # ds) ⟷ ¬ parity (as @ c # b # ds)"
  by (induct as) auto
```

lemma *parity_swap:*
  assumes *"b ≠ d ∧ b ∉ set cs ∧ d ∉ set cs"*
  shows *"parity (as @ b # cs @ d # es) ⟷ ¬ parity (as @ d # cs @ b # es)"*



```
global_interpretation classifier parity

sublocale hats < hats_parity spare assigned parity

context hats begin
  lemma example_odd: "choices [2,0,5,4,3] = [1,0,5,4,3]"
  lemma example_even: "choices [1,0,5,4,3] = [1,0,5,4,3]"
end
```

m.brck.nl/ylj17