NETS 213: CROWDSOURCING AND HUMAN COMPUTATION

Programming the Crowd

Penn Engineering
UNIVERSITY of PENNSYLVANIA

Algorithms for human computation

- MTurk provides an on-demand source for human computation
- Potential opportunities for exploring algorithms that use people as a function call
- However, MTurk isn’t set up to support algorithms

MTurk limitations

- MTurk requesters can post batches of independent jobs
- Perfect for tasks that can be done in parallel like labeling 1000 images
- But poorly suited for tasks that build on each other
- **What is MTurk missing that is essential in algorithms or programming languages?**

TurKit: A programming language for the crowd

```
ideas = []
for (var i = 0; i < 5; i++) {
      idea = mturk.prompt(
          "What’s fun to see in New York City? Ideas so far: " + ideas.join(", "))
      ideas.push(idea)
}
ideas.sort(function (a, b) {
      v = mturk.vote("Which is better?", [a, b]) return v == a ? -1 : 1
})
```

What new concerns exist for crowd programming?

- After a HIT is posted to MTurk, it can take hours before Turkers complete it and so latency could cause algorithms to take days
- What is the behavior if your program crashes?
- What if this happens after you have already spent money on a bunch of HITs?

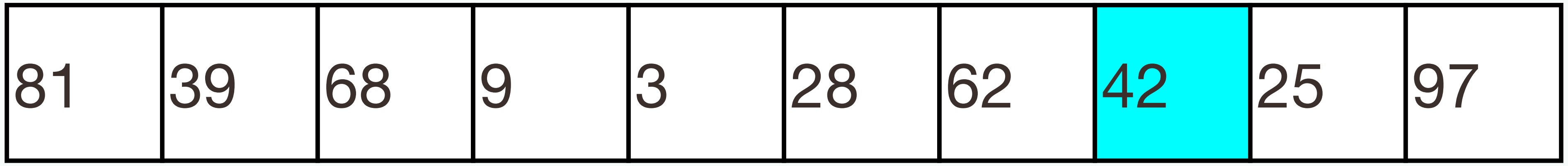
Crash and re-run

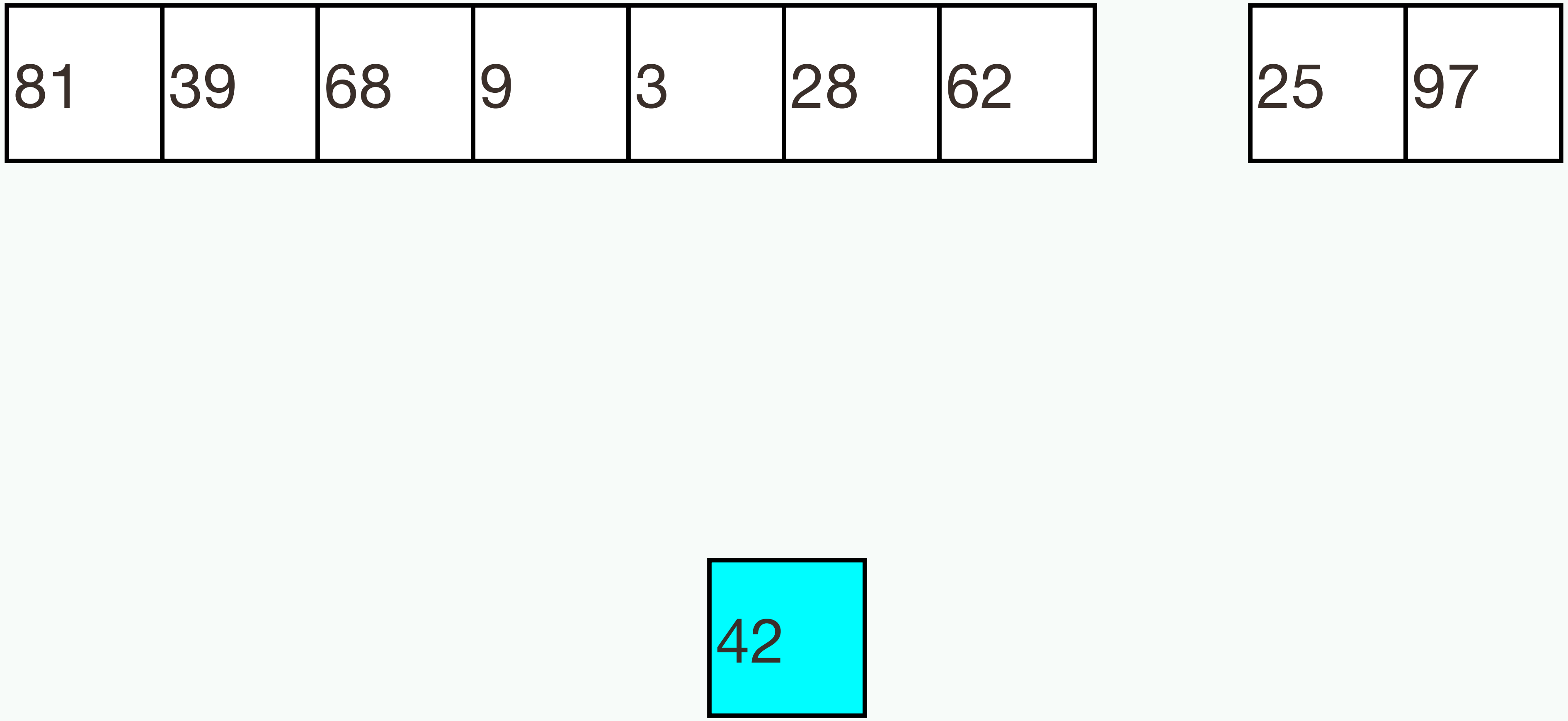
- TurKit introduces a new programming paradigm called crash and rerun
- Designed for long running processes where local computation is cheap, and remote work is costly
- (~~Crash~~) Cache and re-run

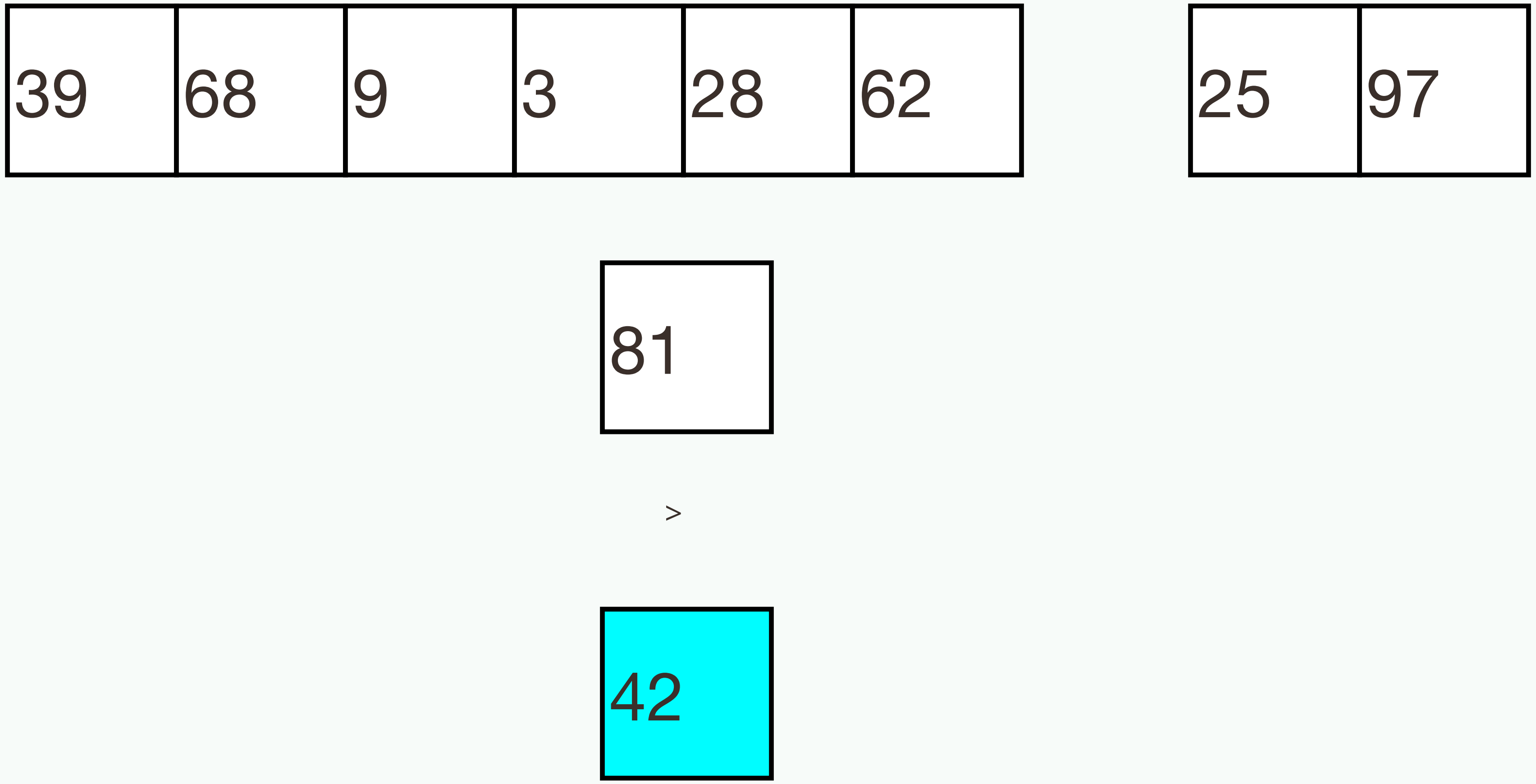
Quicksort

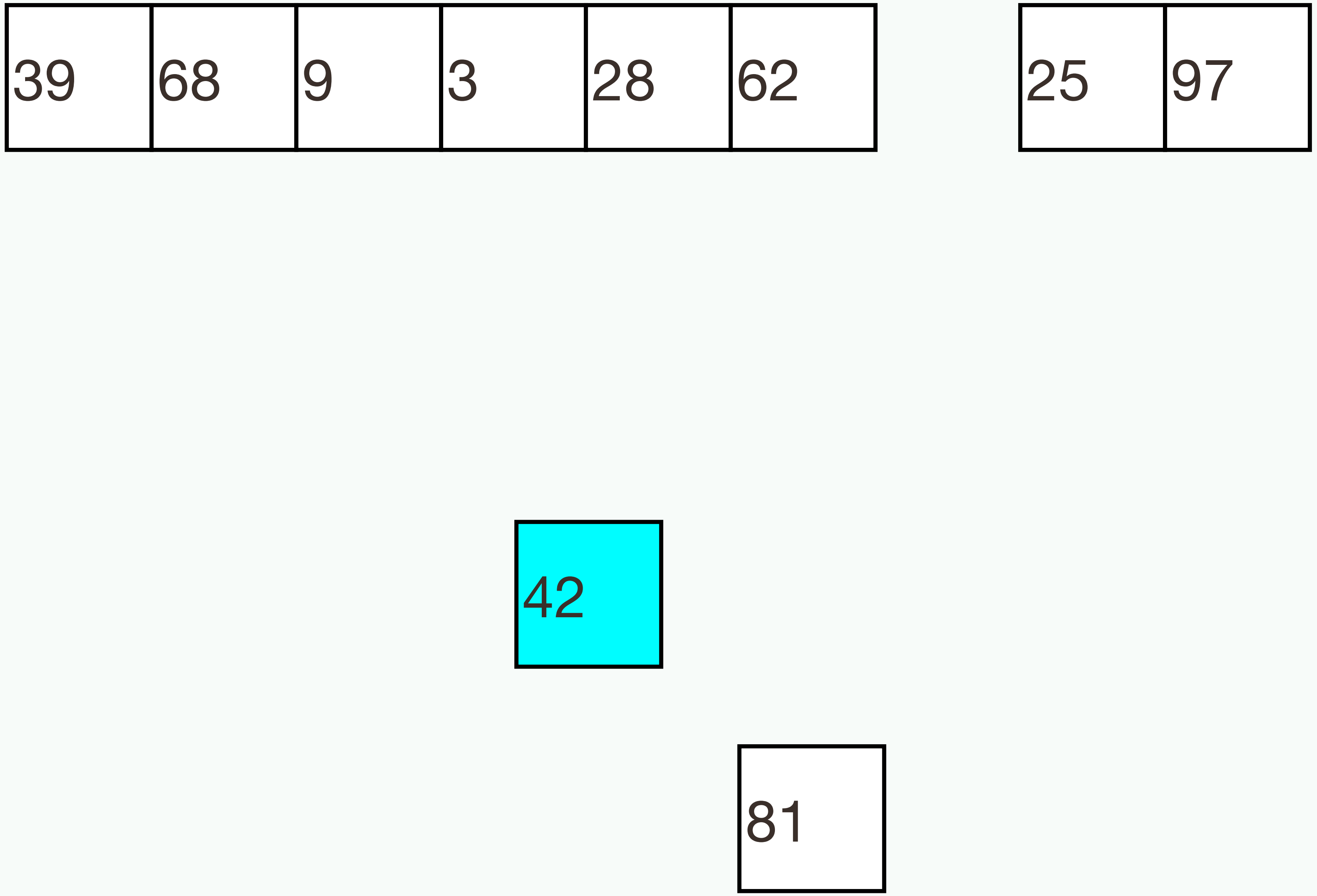
```
quicksort(A):
  if A.length > 0:
    pivot ← A.remove(A.randomIndex())
    left ← new array; right ← new array
    for x in A:
      if **compare**(x, pivot):
        left.add(x)
      else:
        right.add(x)
    quicksort(left)
    quicksort(right)
    A.set(left + pivot + right)
```

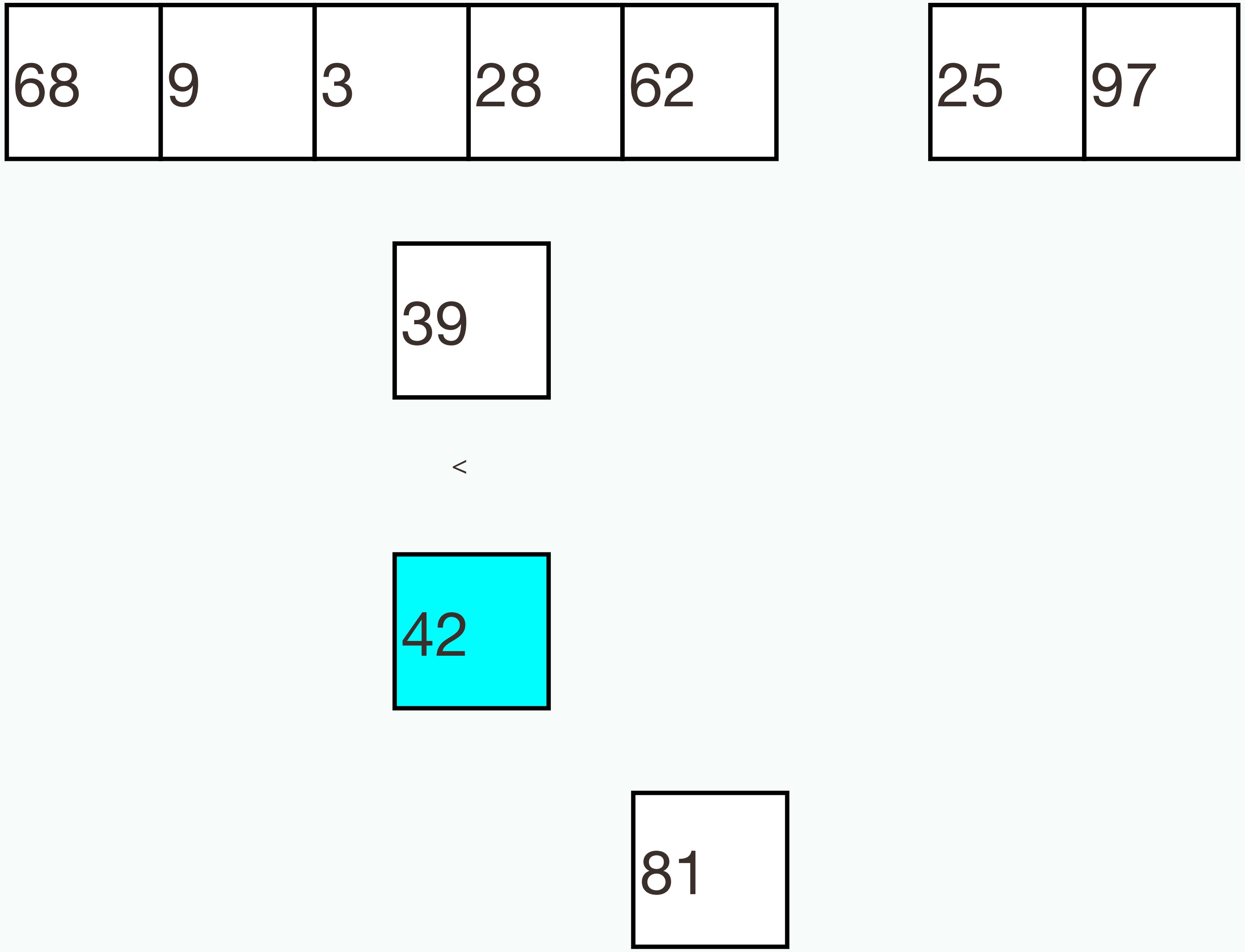
81	39	68	9	3	28	62	42	25	97

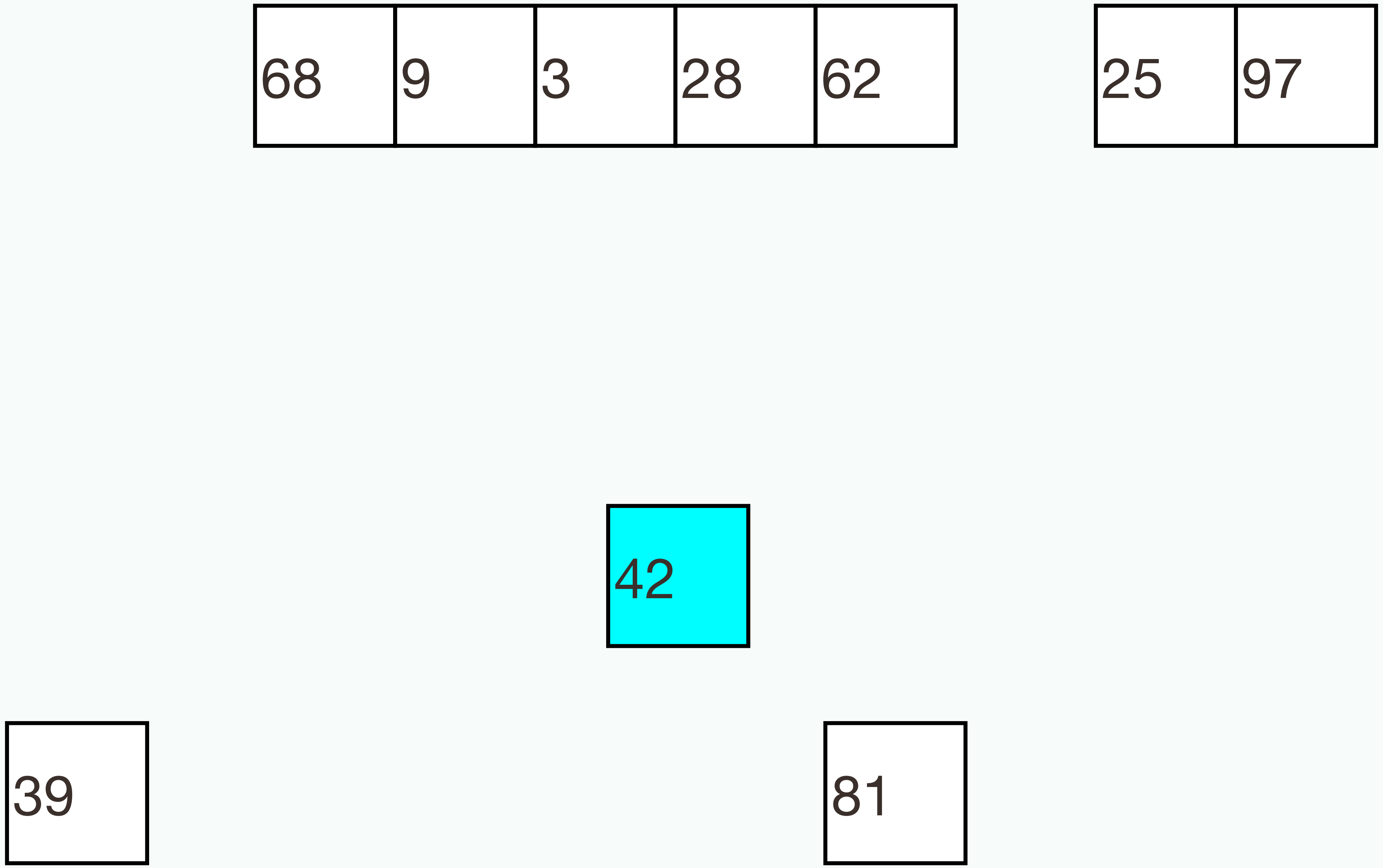

81
39
68
9
3
28
62
42
25
97

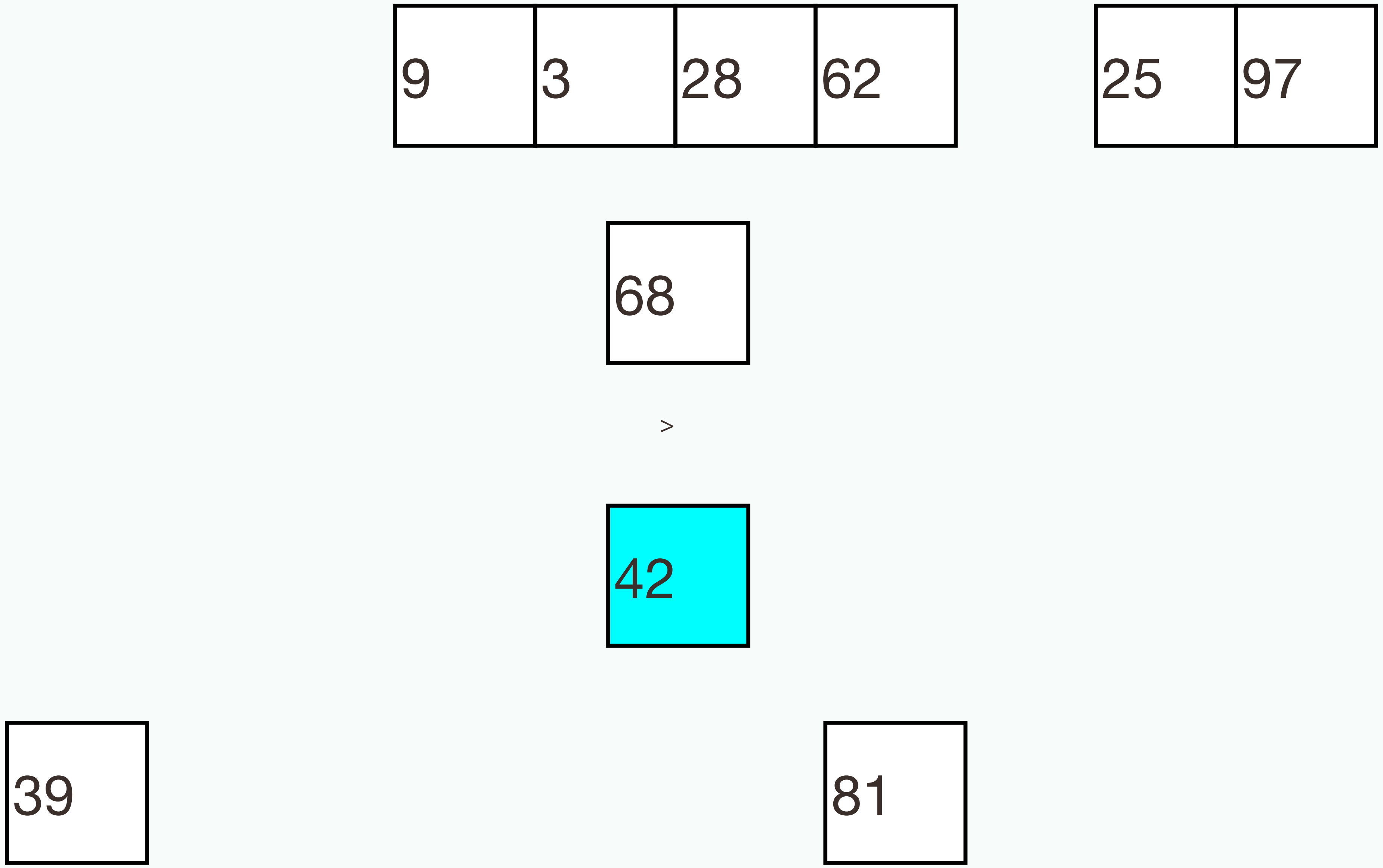

81
39
68
9
3
28
62
25
97
42

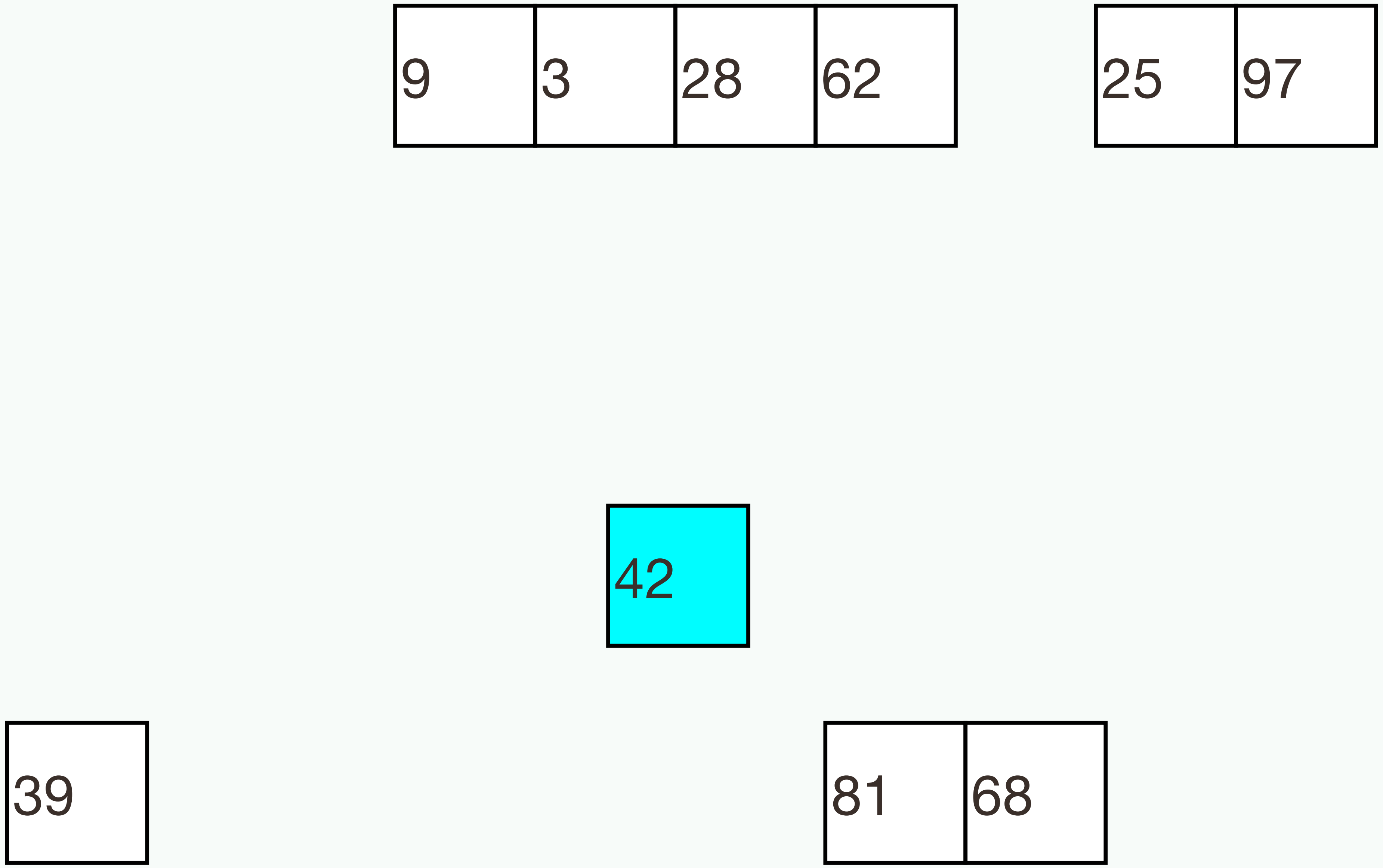

39
68
9
3
28
62
25
97
81
>
42

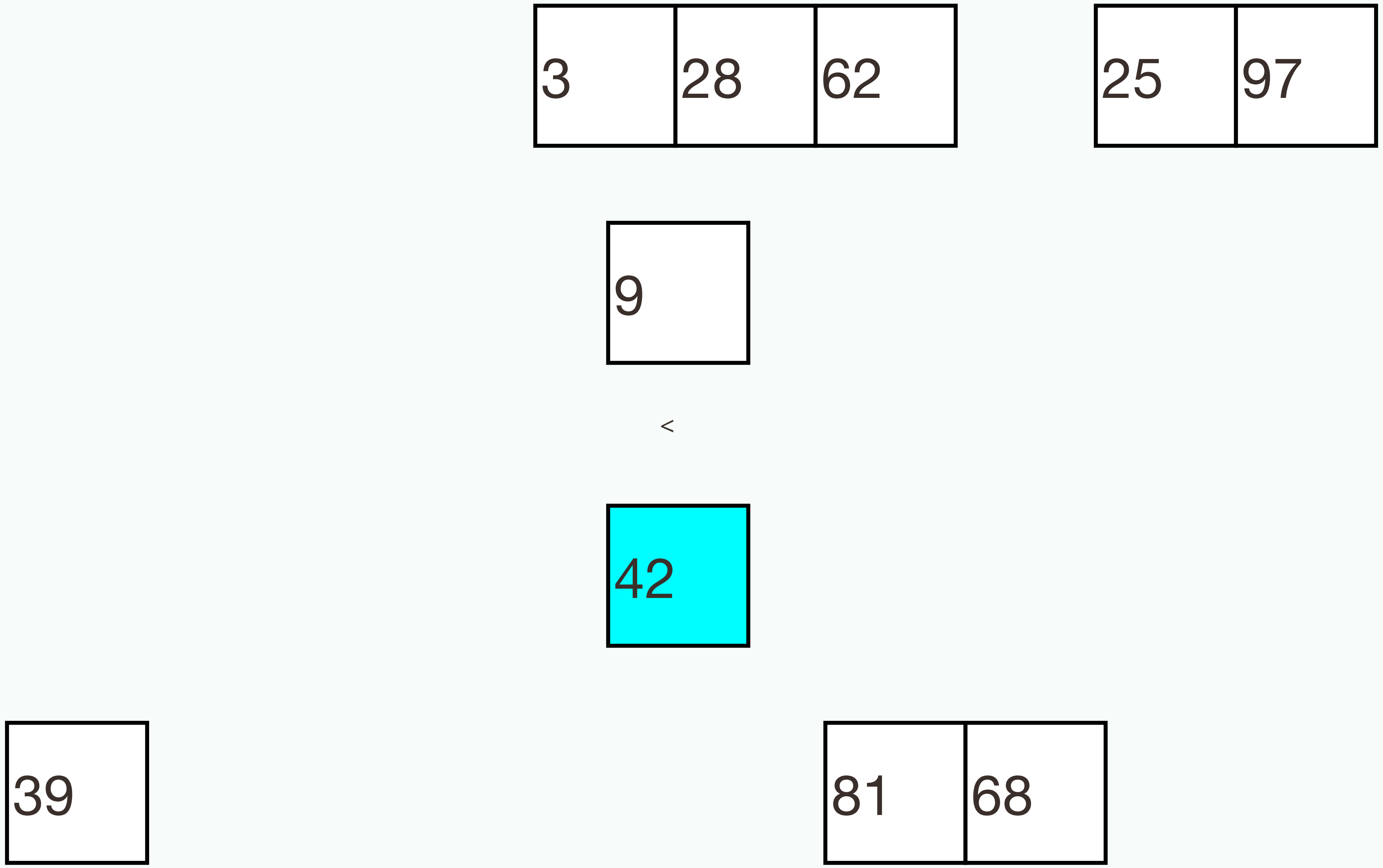

39
68
9
3
28
62
25
97
42
81

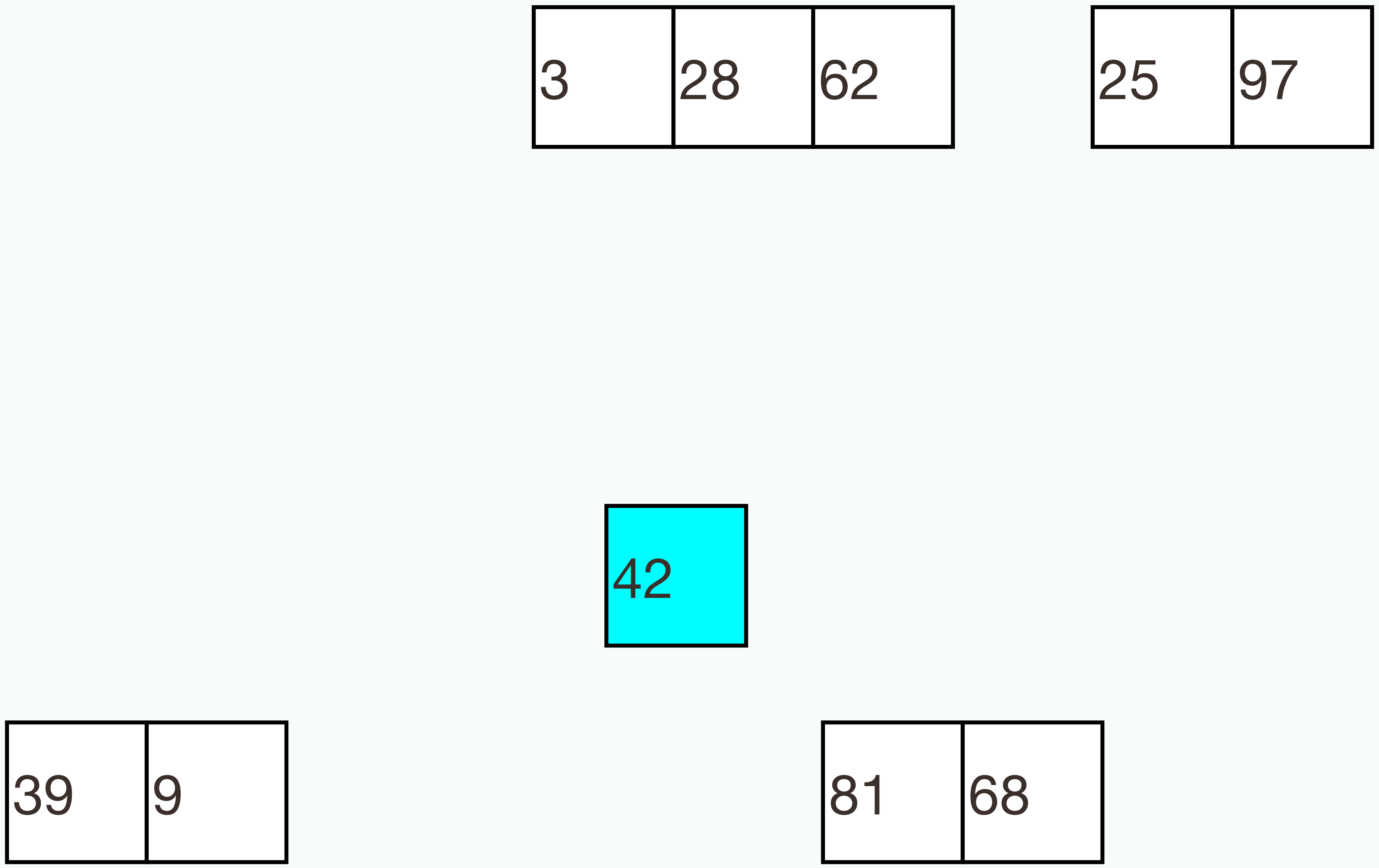

68
9
3
28
62
25
97
39
<
42
81

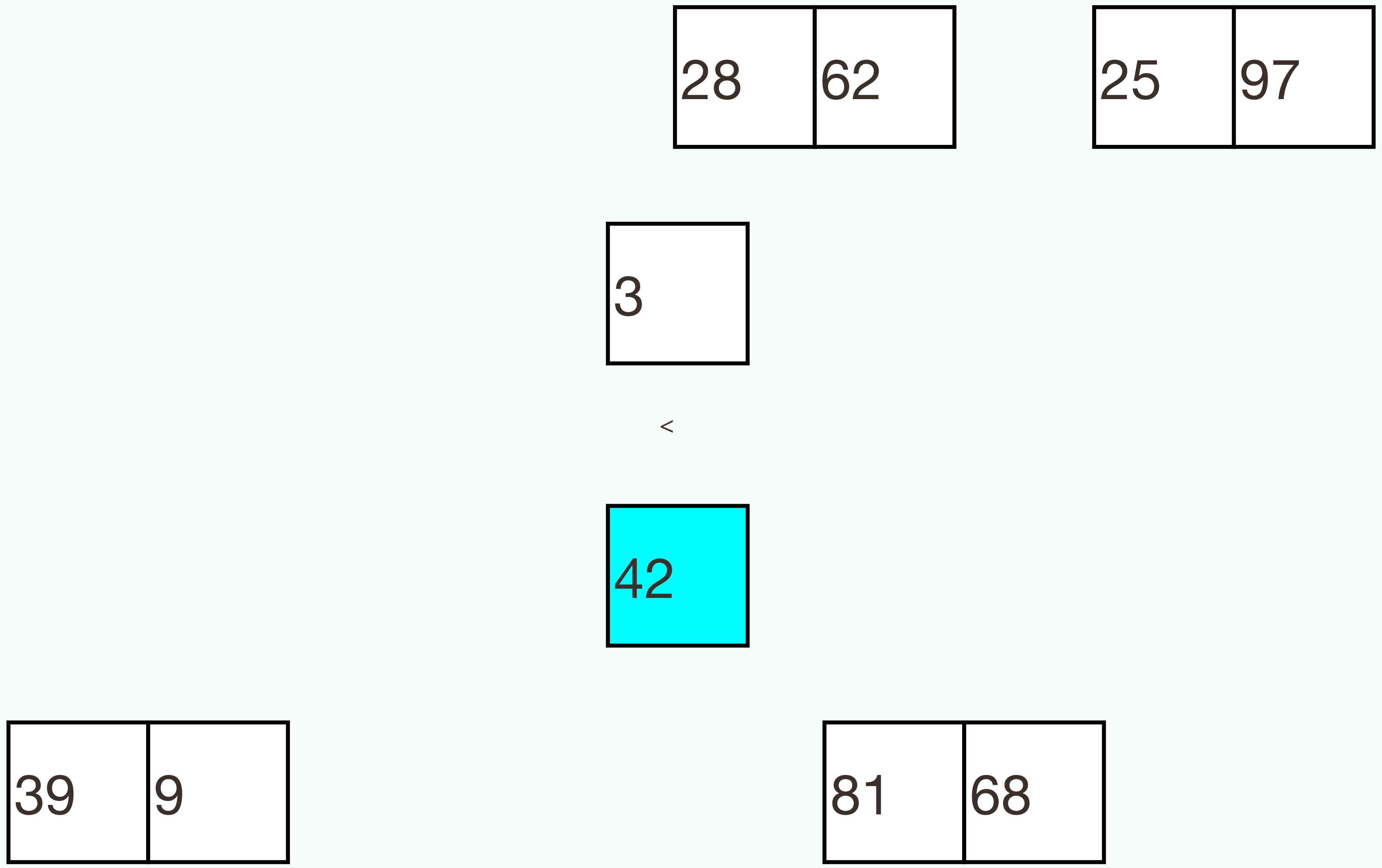

68
9
3
28
62
25
97
42
39
81

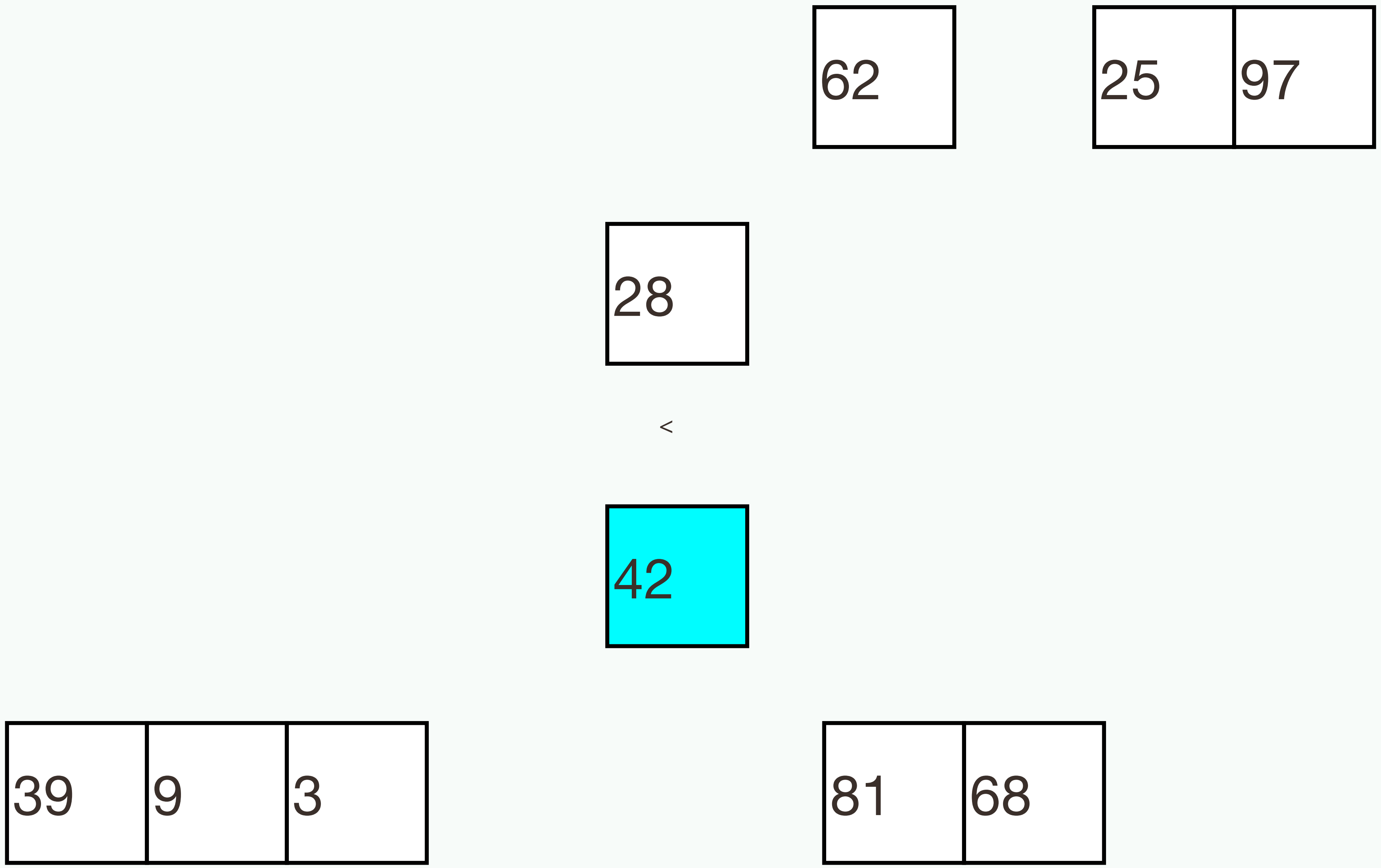

9
3
28
62
25
97
68
>
42
39
81

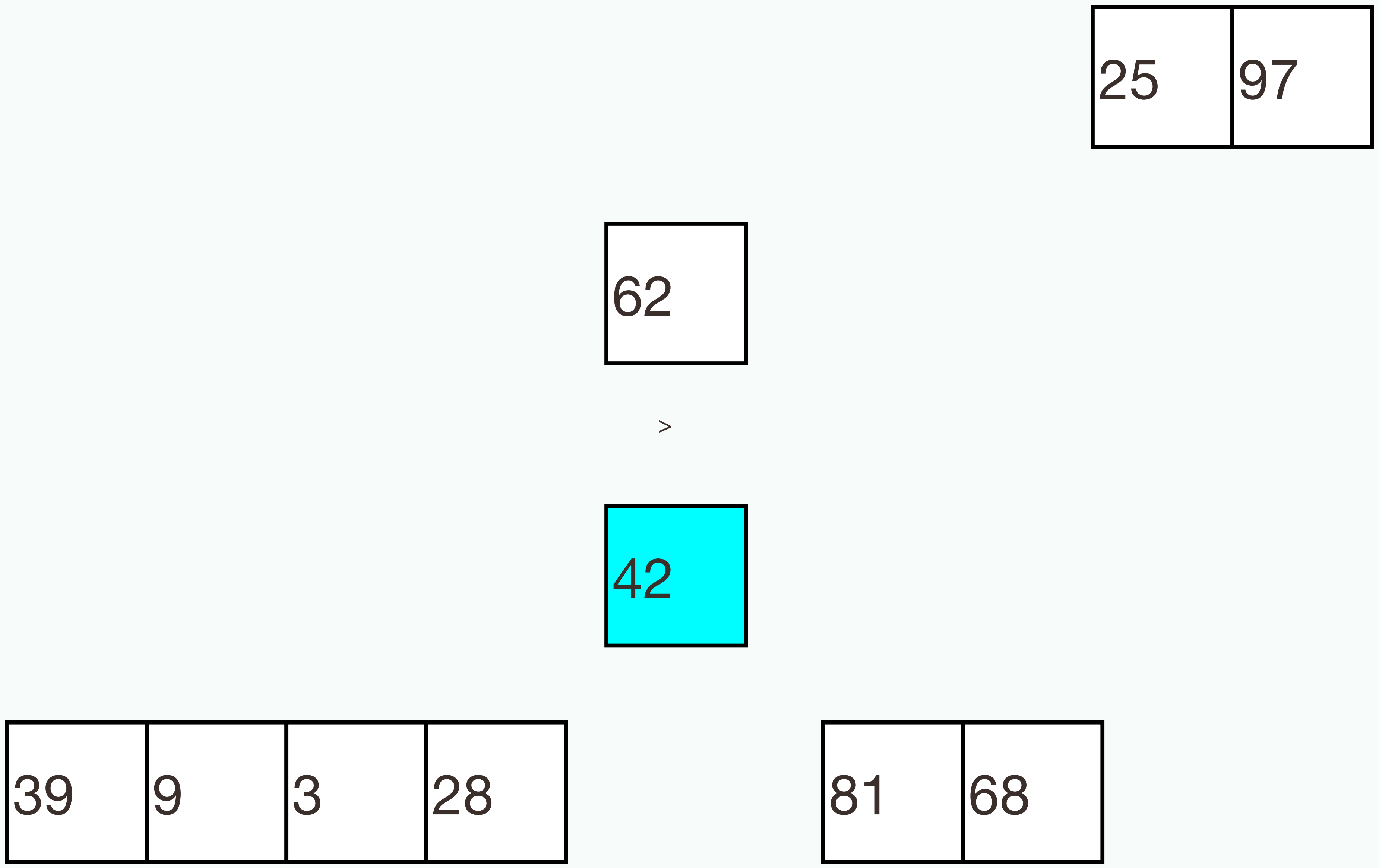

9
3
28
62
25
97
42
39
81
68

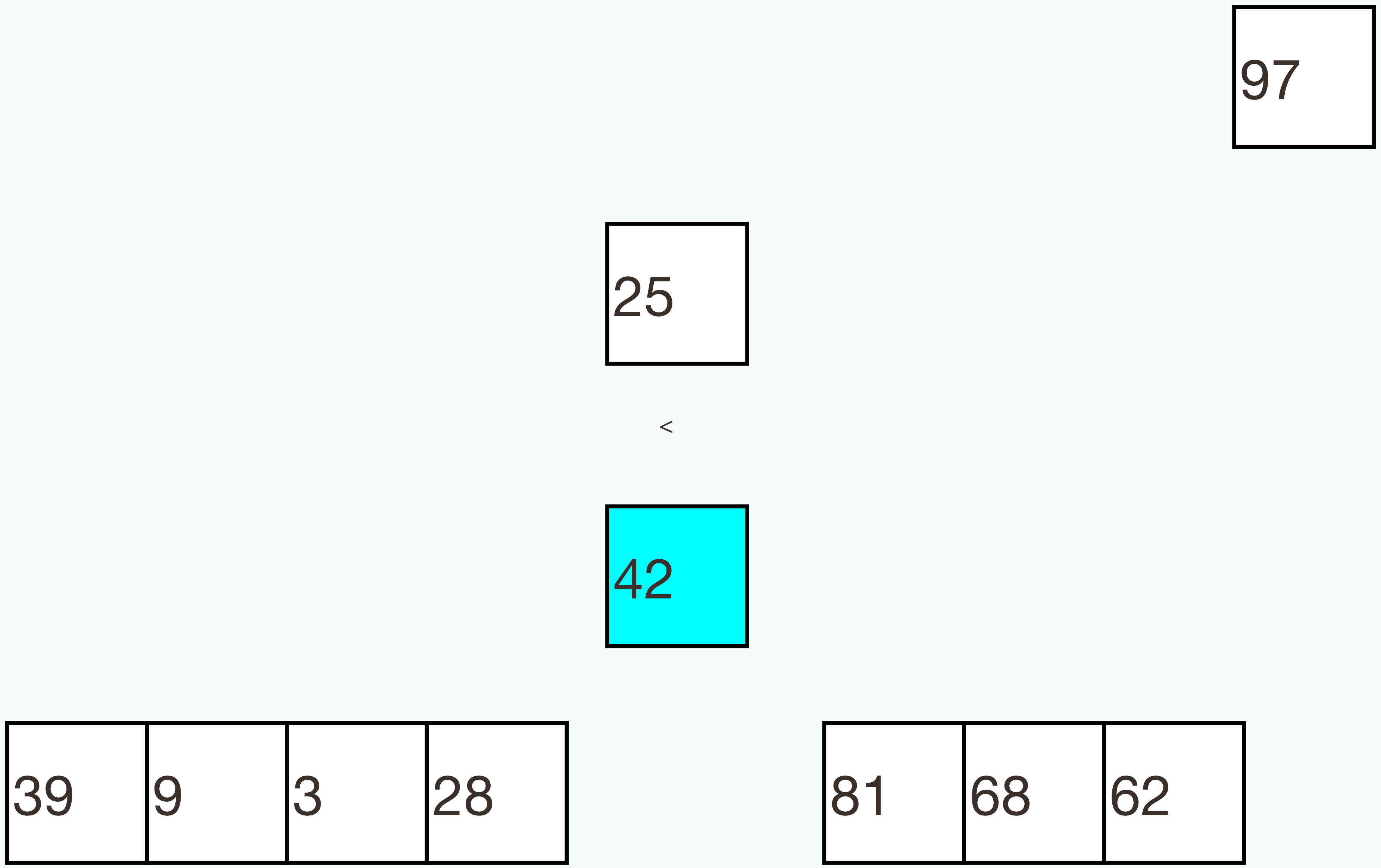

3
28
62
25
97
9
<
42
39
81
68

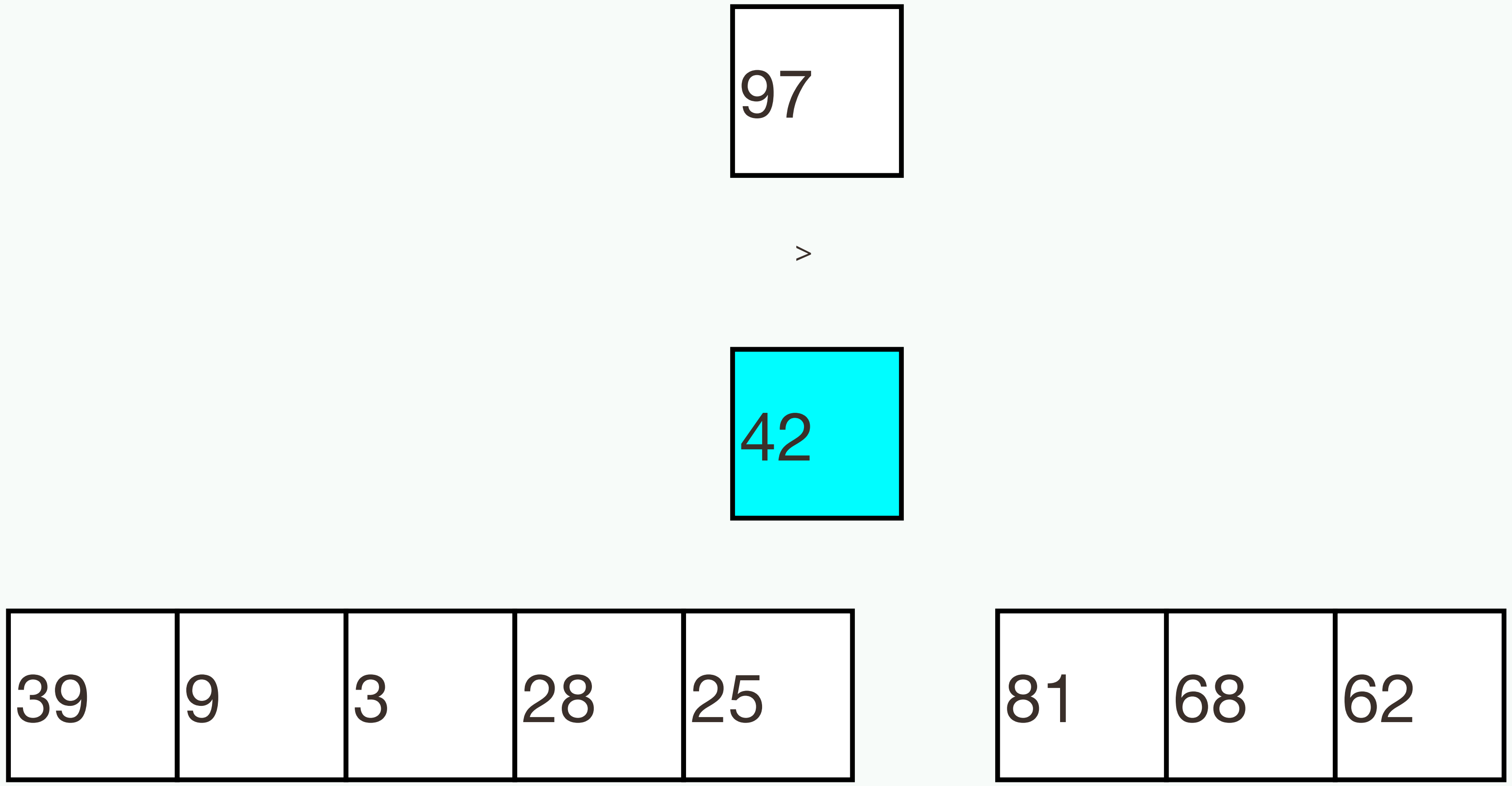

3
28
62
25
97
42
39
9
81
68

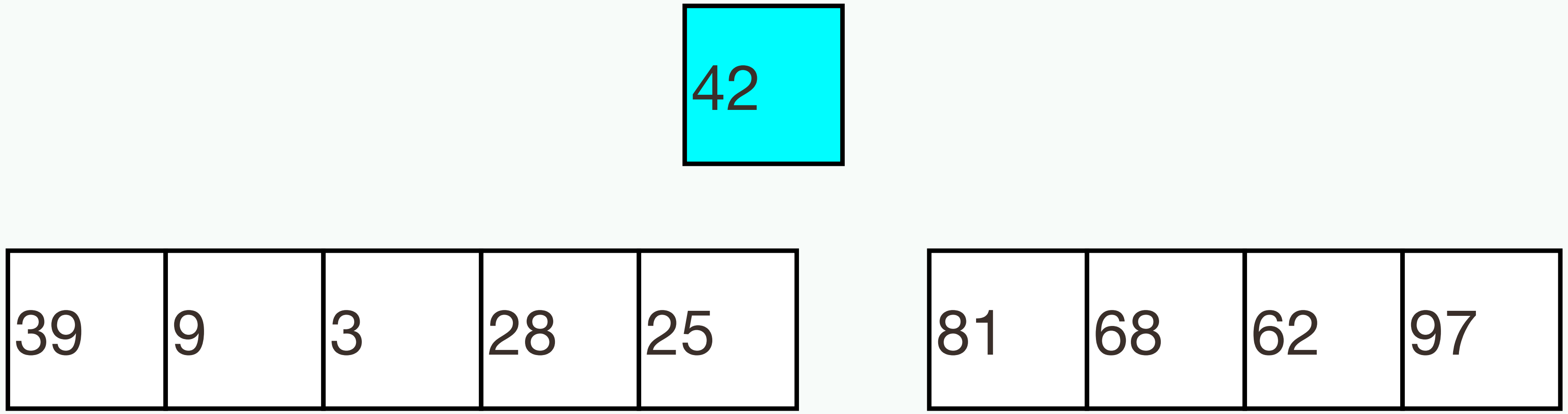

28
62
25
97
3
<
42
39
9
81
68

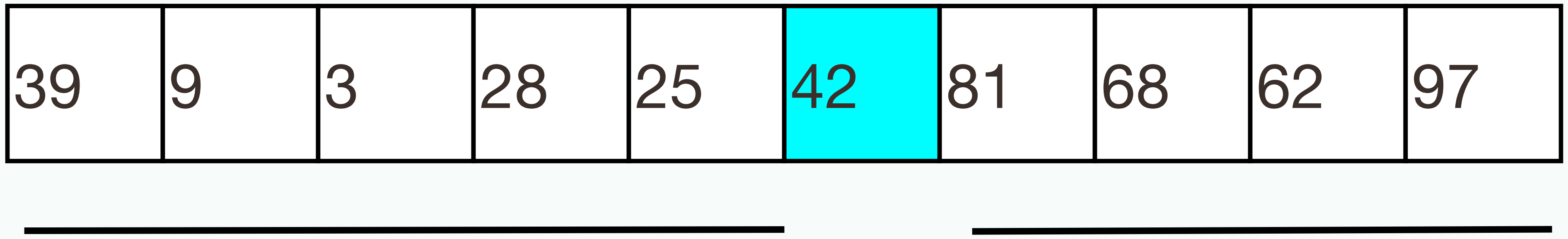

62
25
97
28
<
42
39
9
3
81
68

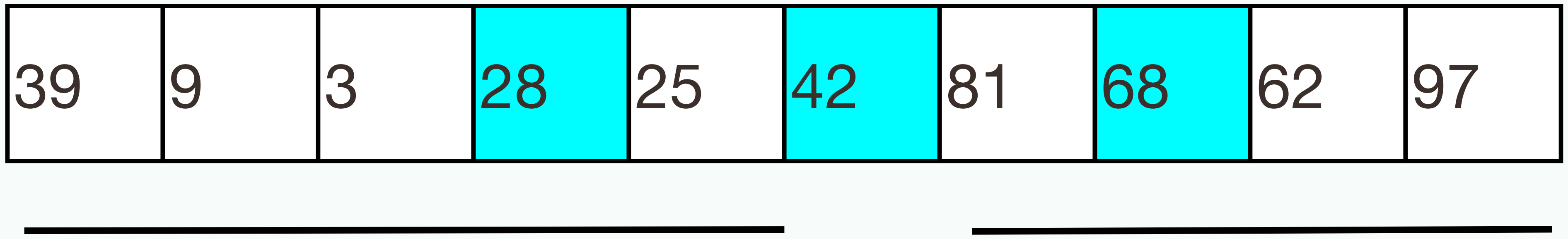

25
97
62
>
42
39
9
3
28
81
68

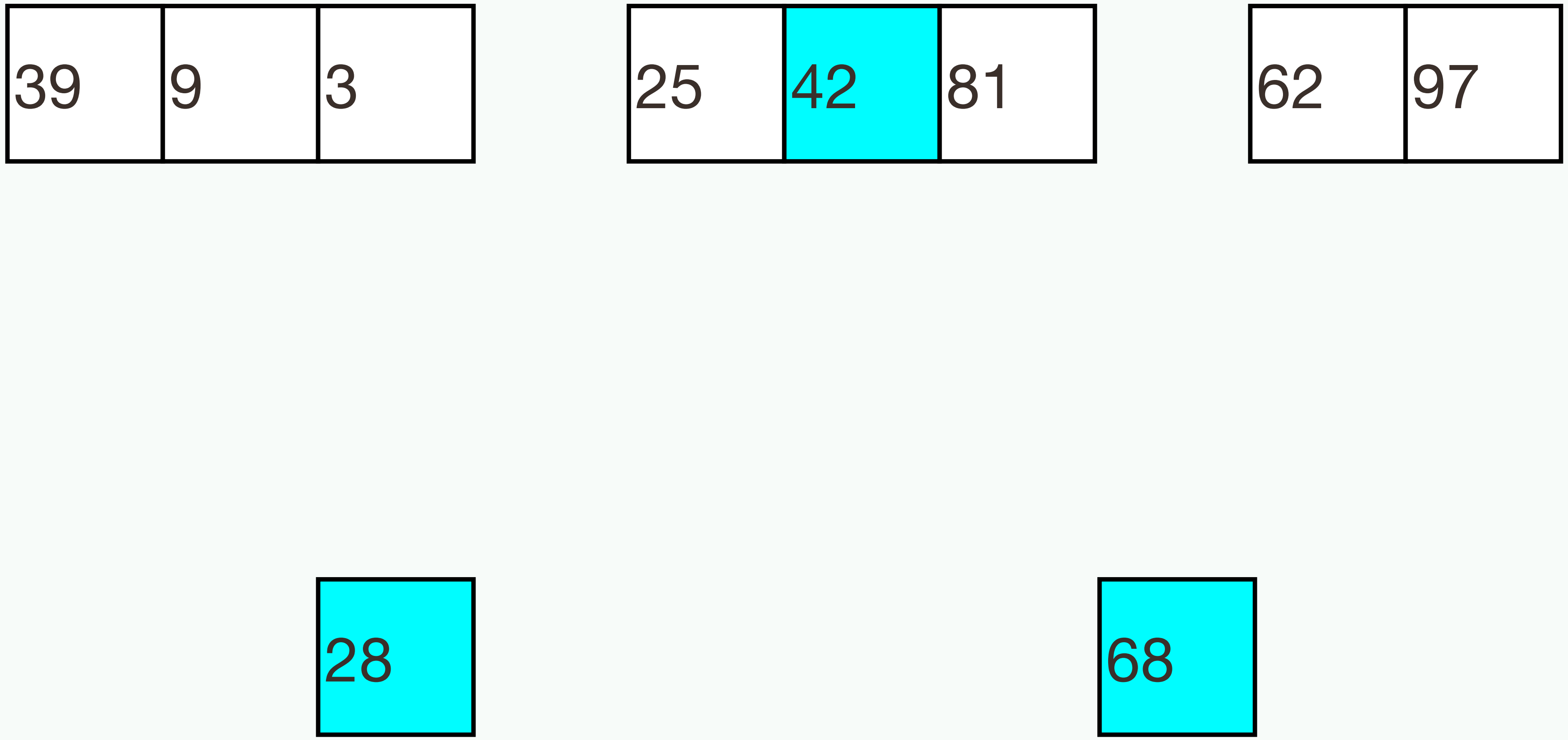

97
25
<
42
39
9
3
28
81
68
62

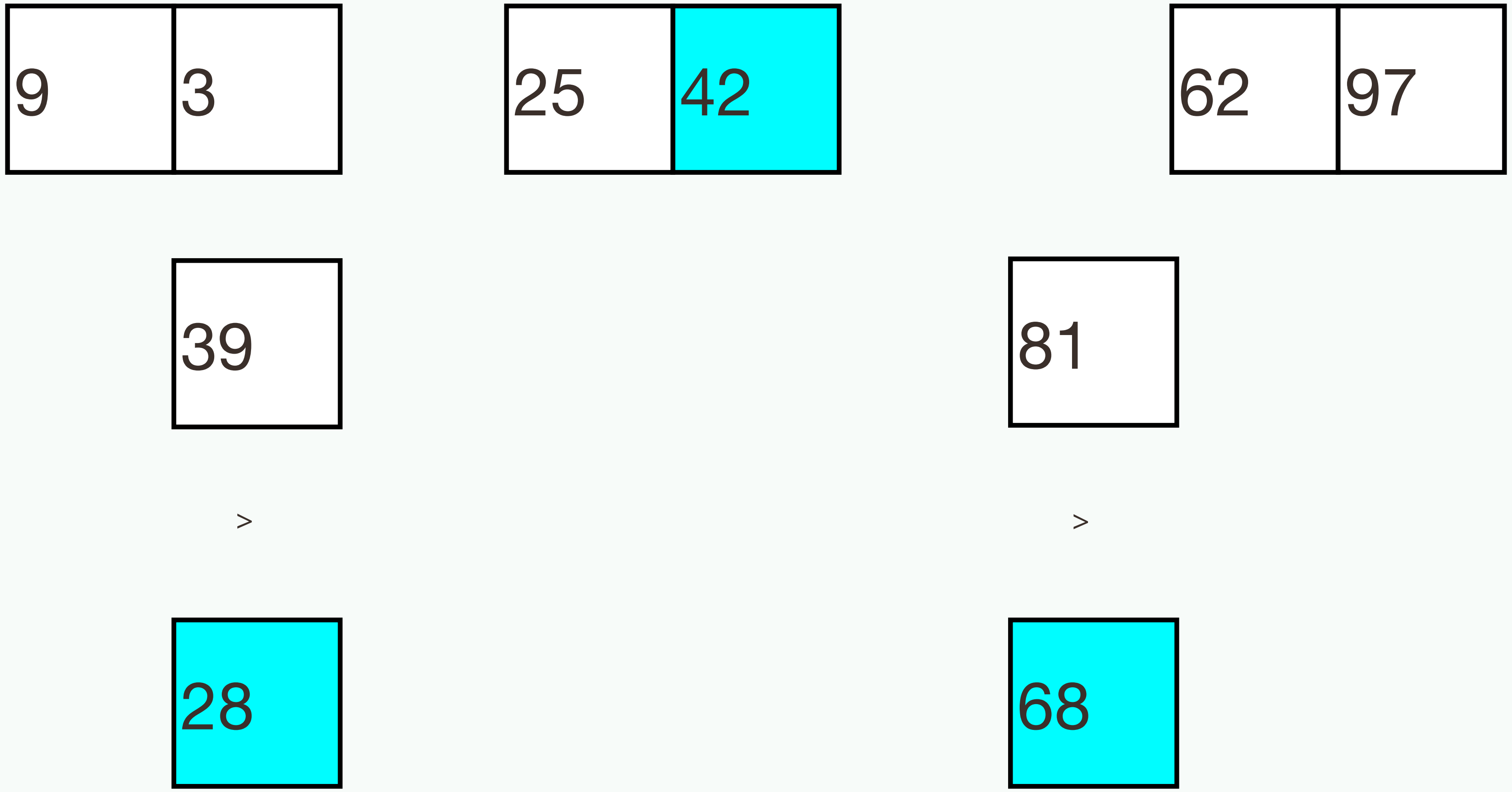

97
>
42
39
9
3
28
25
81
68
62

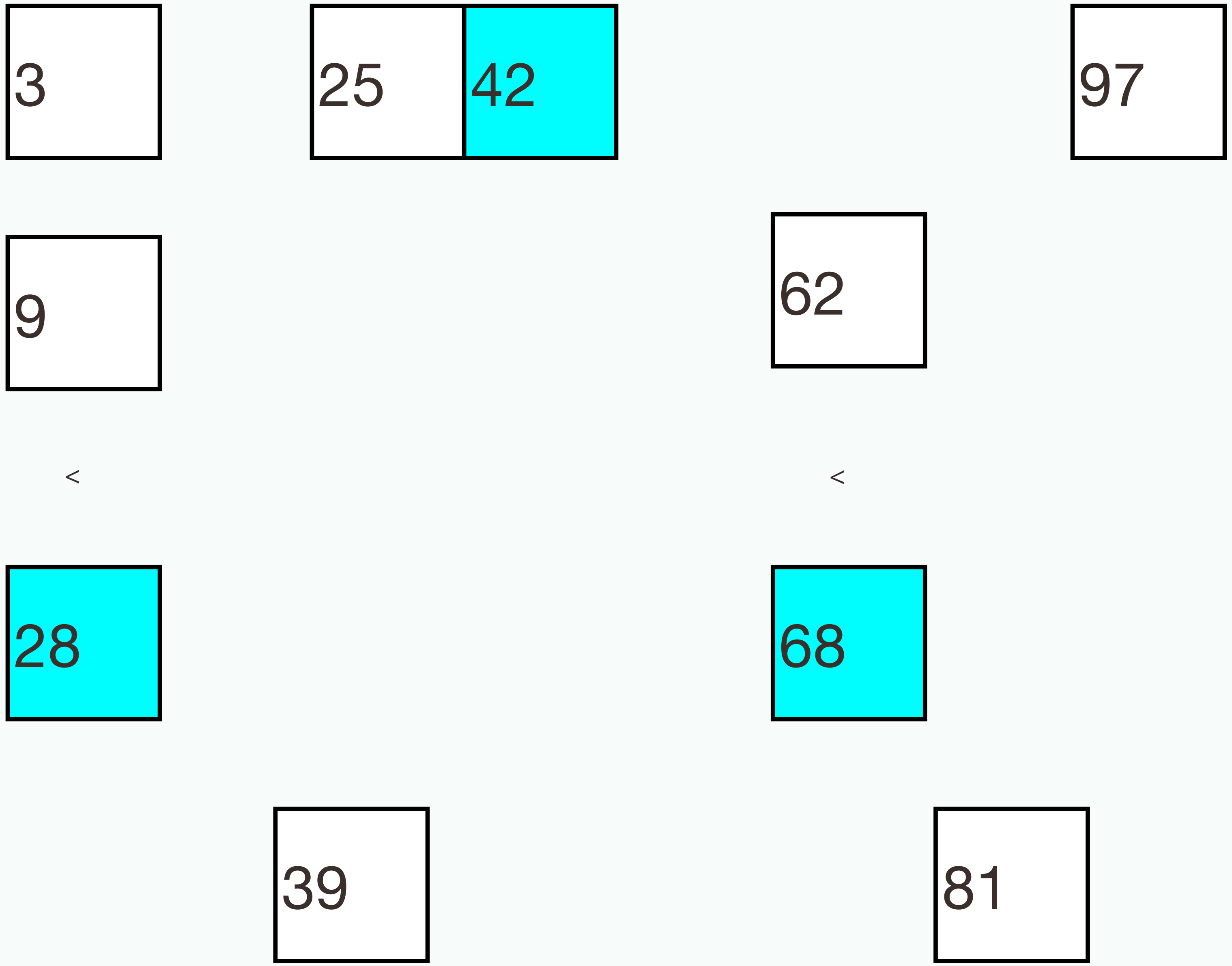

42
39
9
3
28
25
81
68
62
97

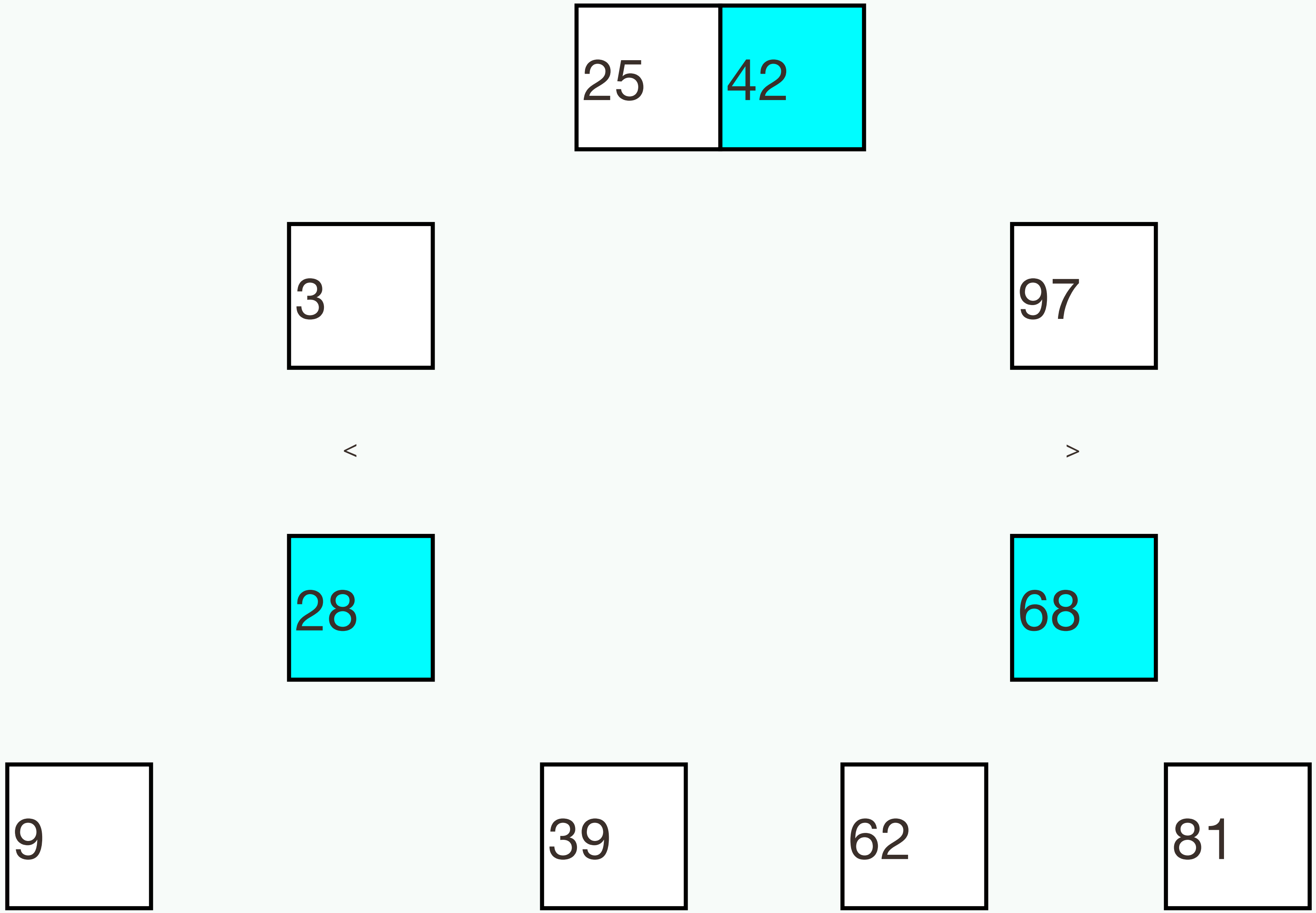

39
9
3
28
25
42
81
68
62
97

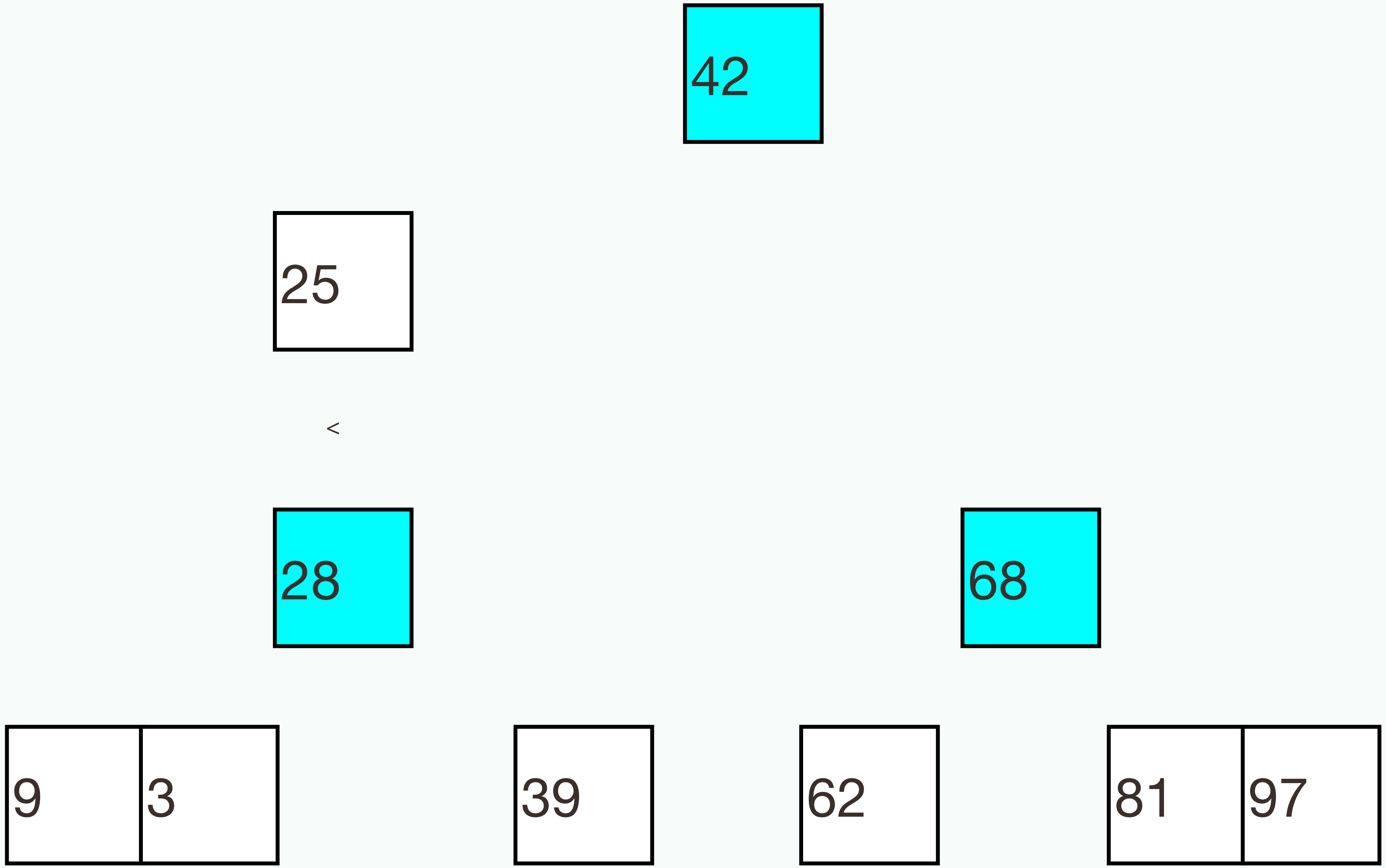

39
9
3
28
25
42
81
68
62
97

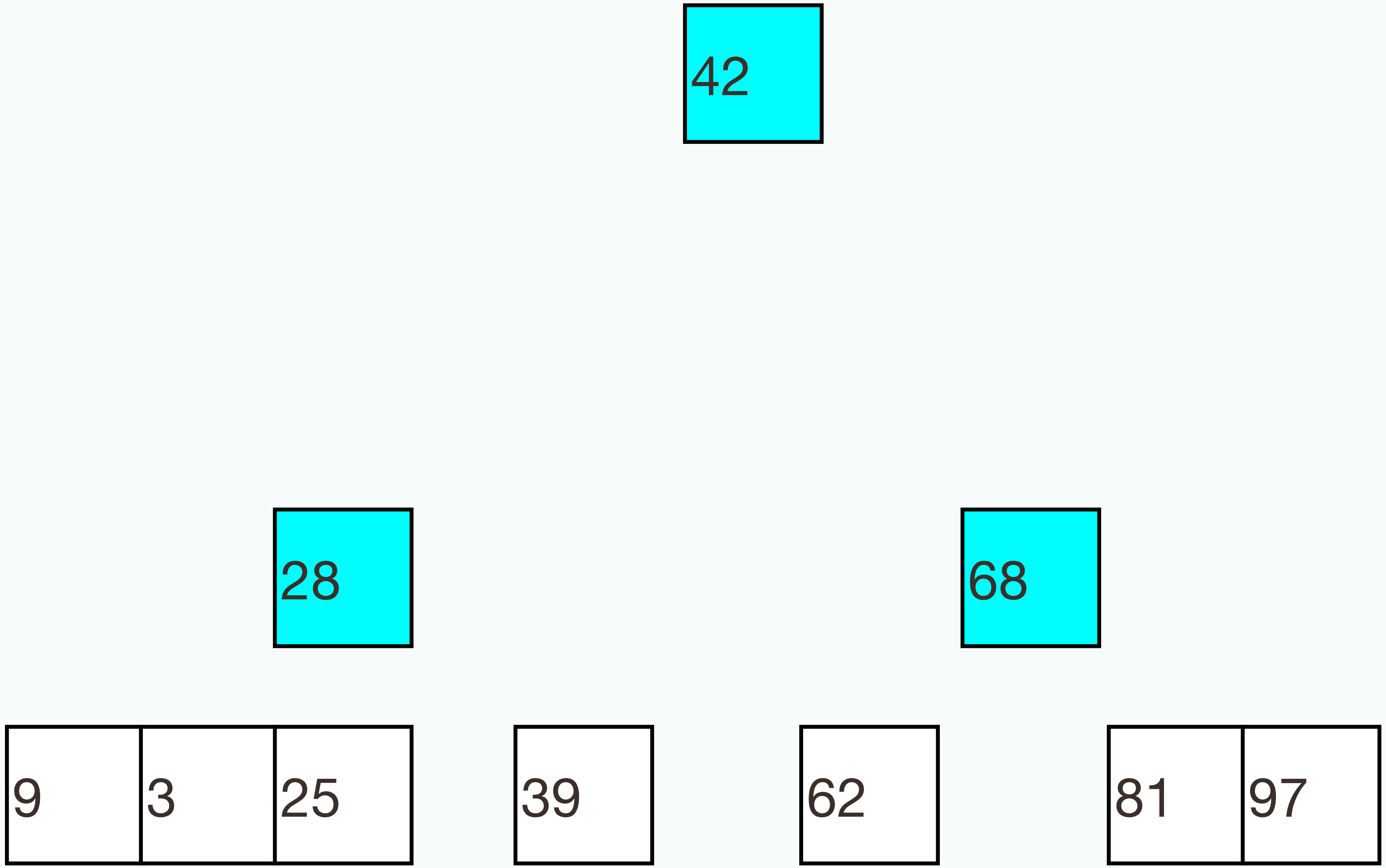

39
9
3
25
42
81
62
97
28
68

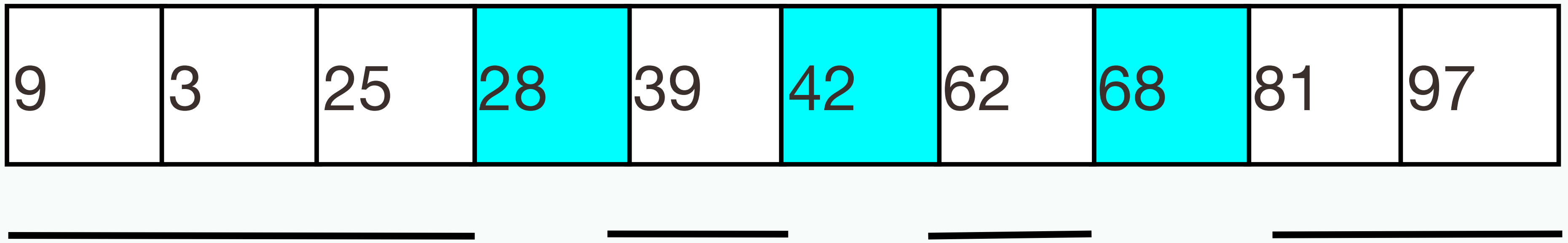

9
3
25
42
62
97
39
81
>
>
28
68

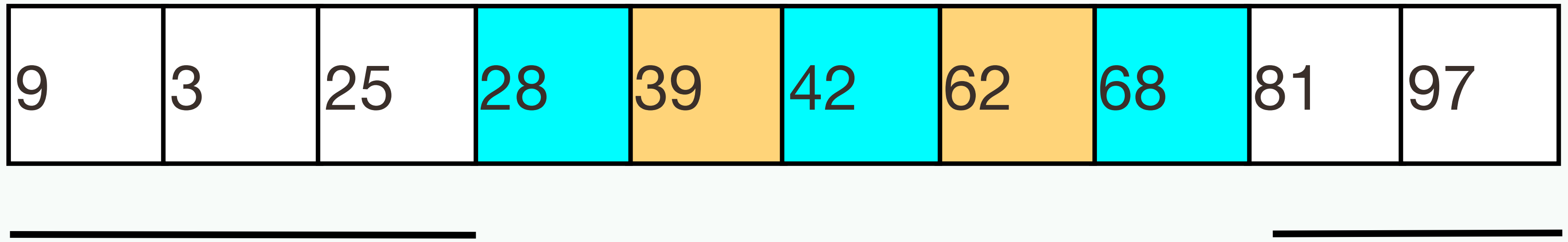

3
25
42
97
9
62
<
<
28
68
39
81

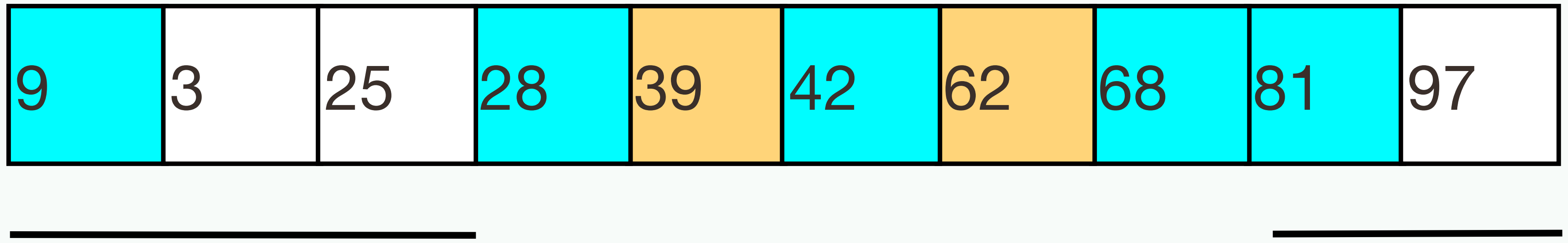

25
42
3
97
<
>
28
68
9
39
62
81

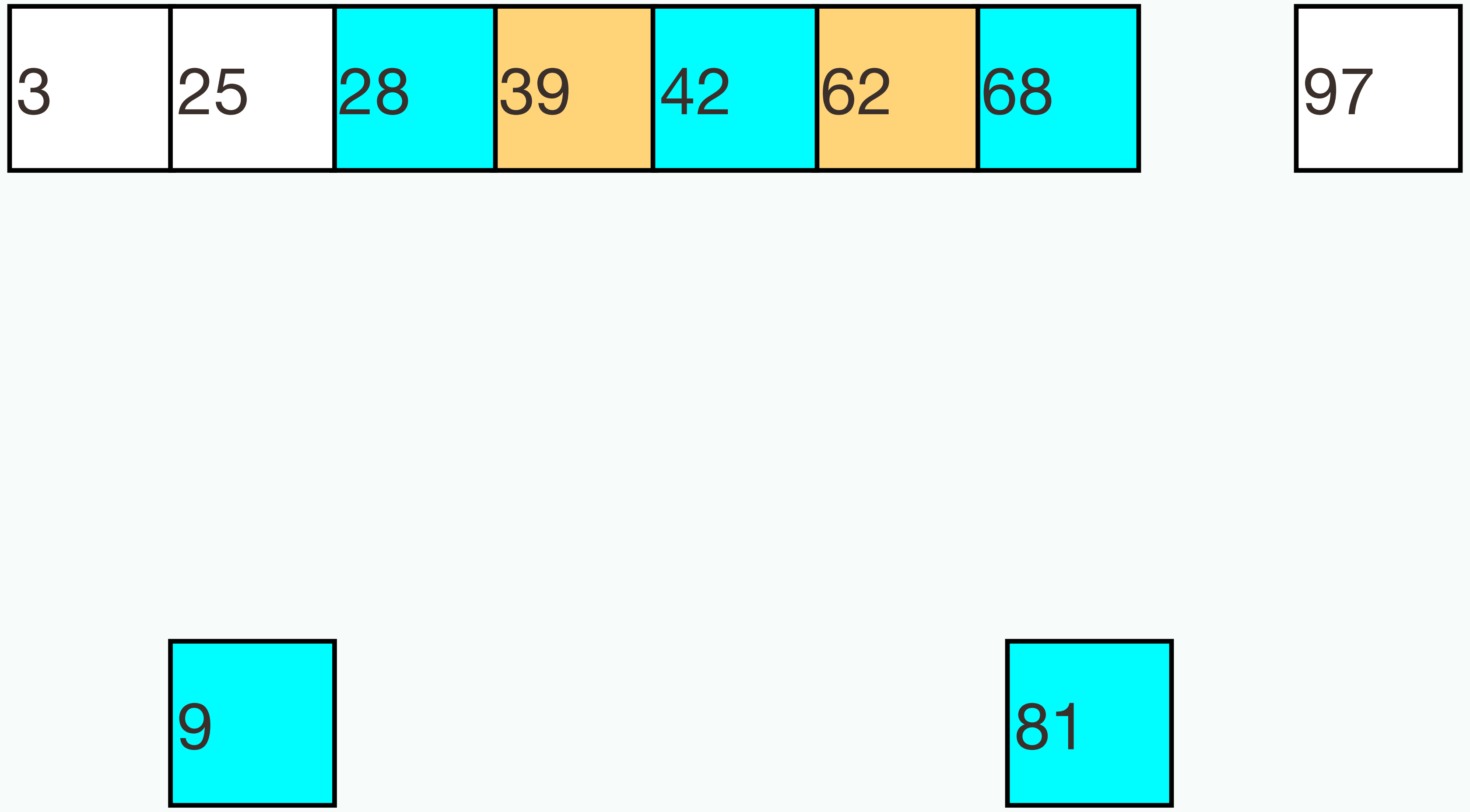

42
25
<
28
68
9
3
39
62
81
97

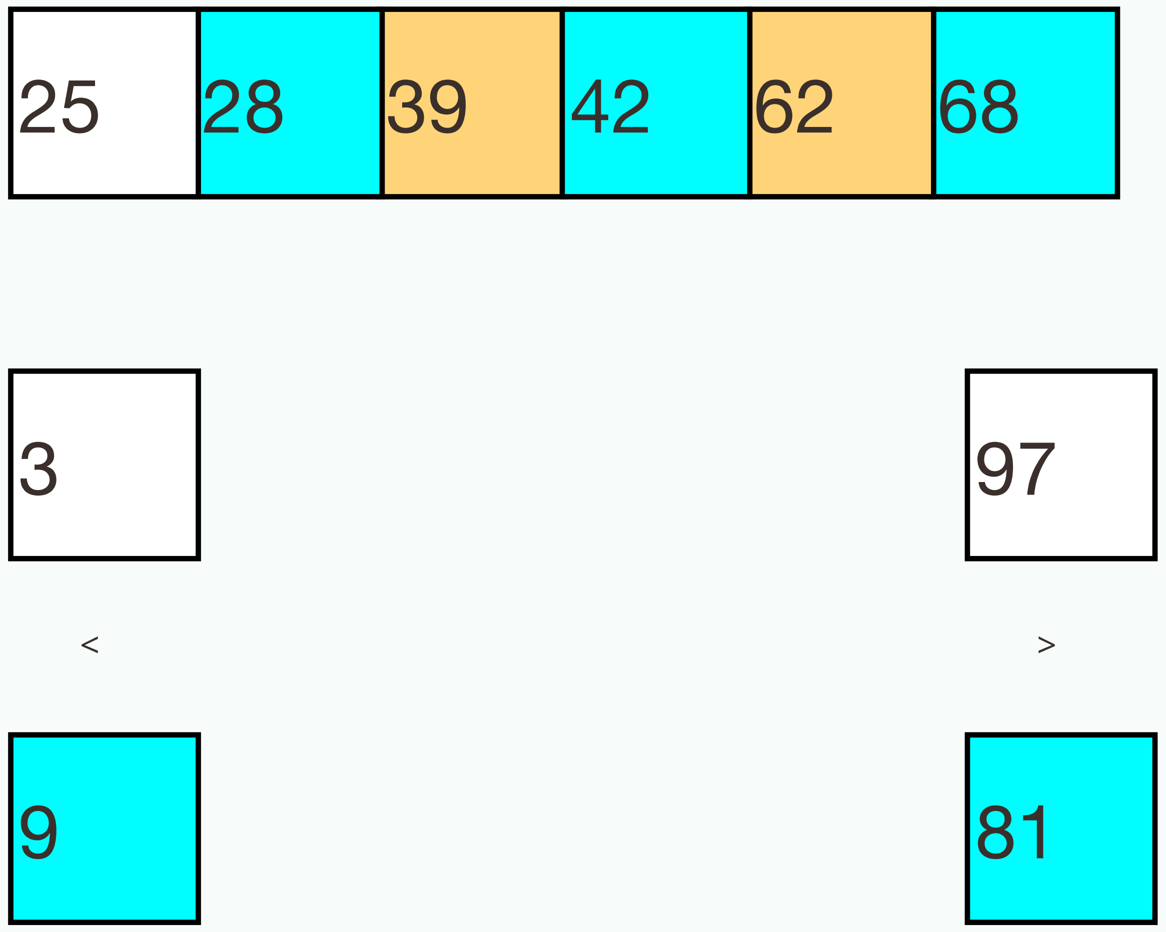

42
28
68
9
3
25
39
62
81
97

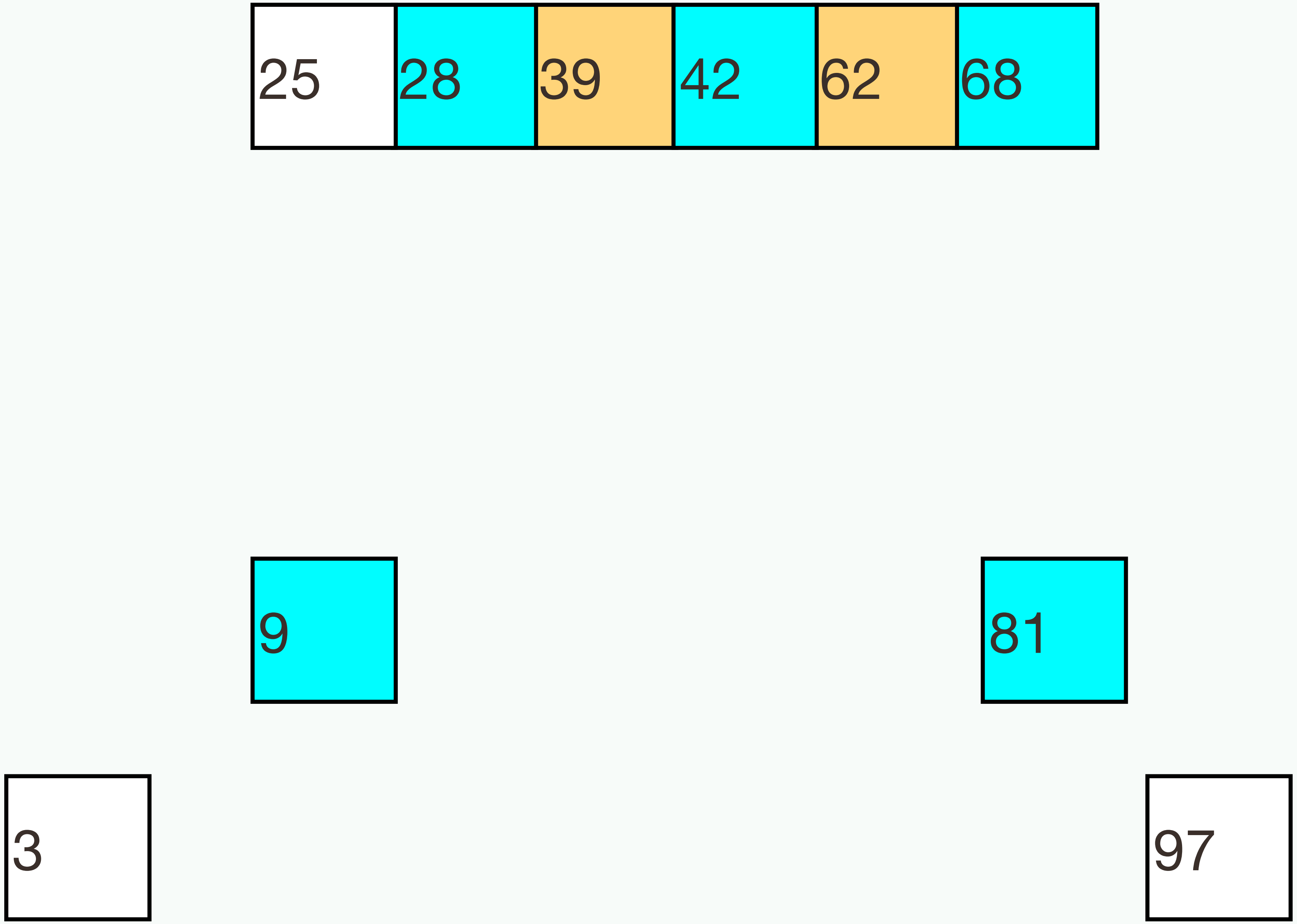

9
3
25
28
39
42
62
68
81
97

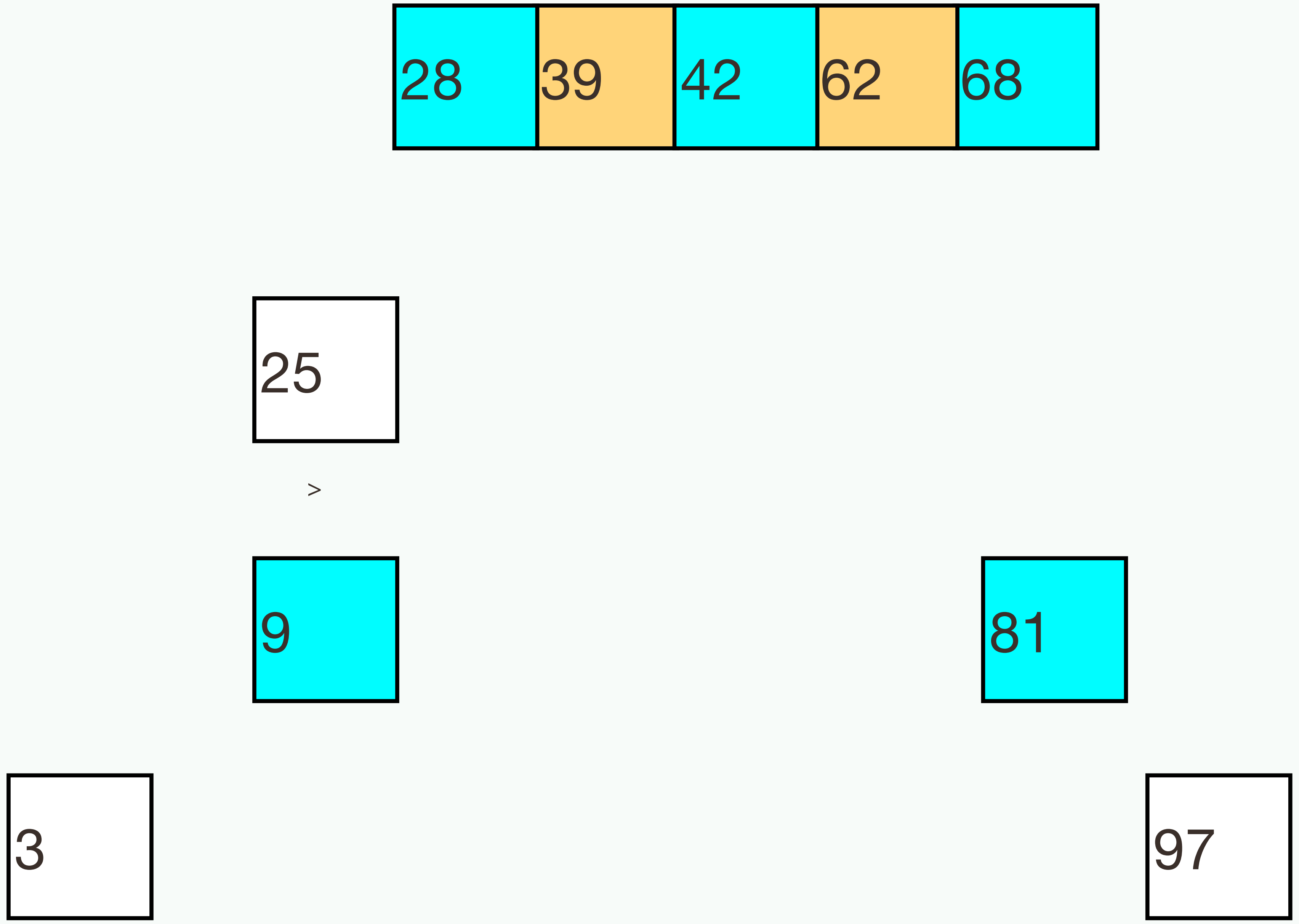

9
3
25
28
39
42
62
68
81
97

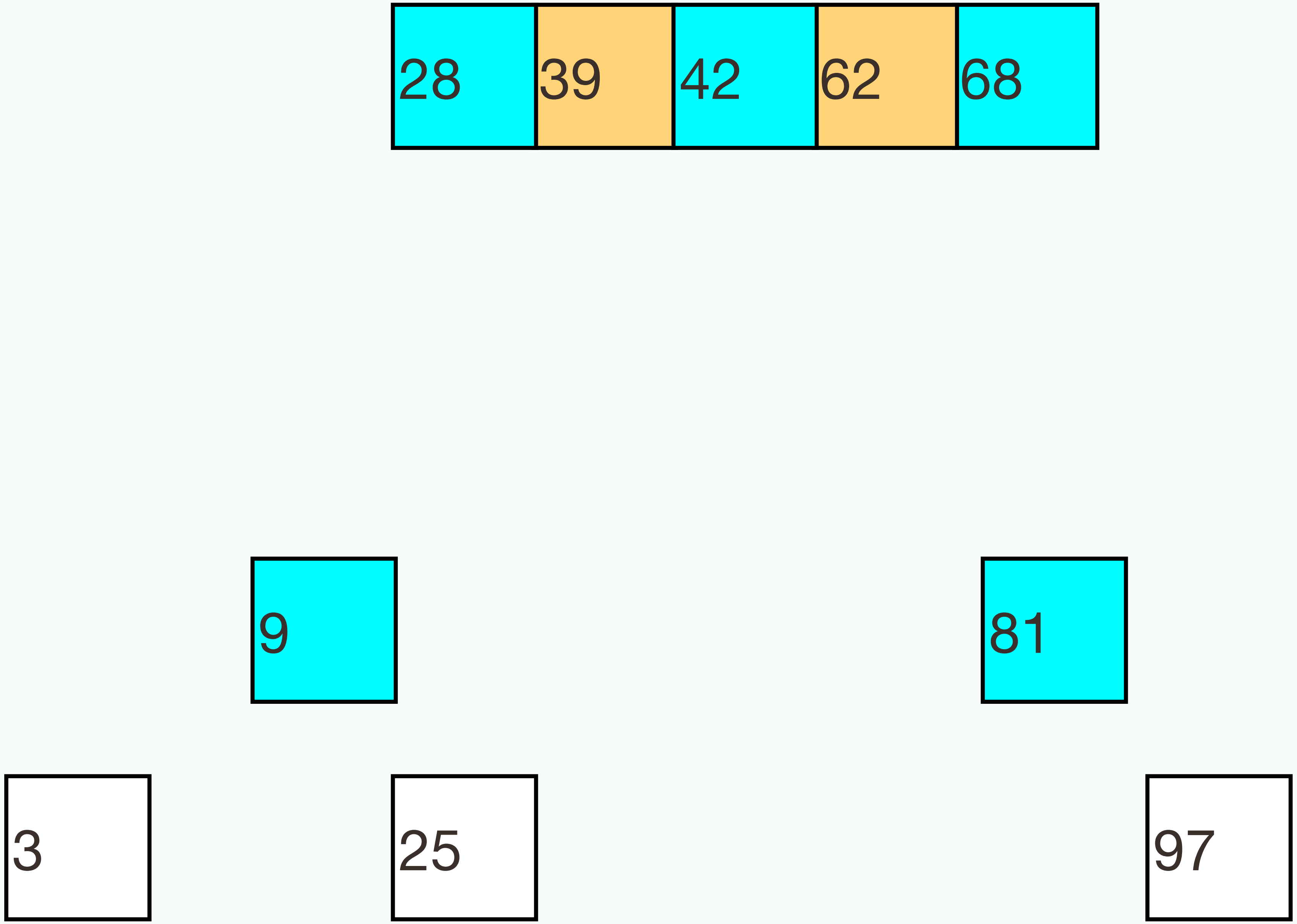

9
3
25
28
39
42
62
68
81
97


3
25
28
39
42
62
68
97
9
81

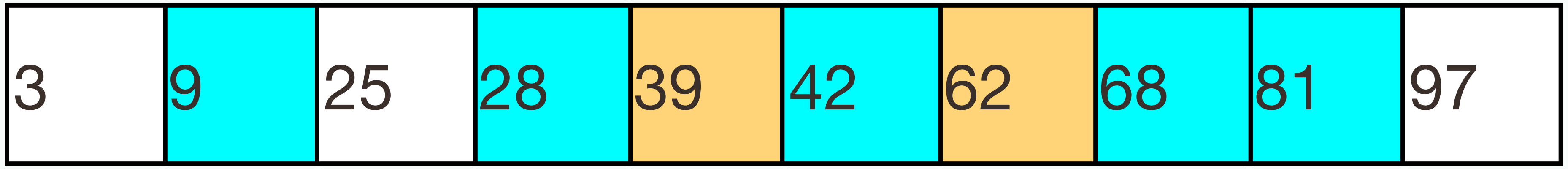

25
28
39
42
62
68
3
97
<
>
9
81


25
28
39
42
62
68
9
81
3
97


28
39
42
62
68
25
>
9
81
3
97



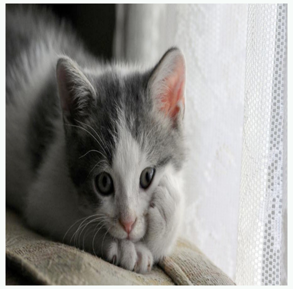
28
39
42
62
68
9
81
3
25
97

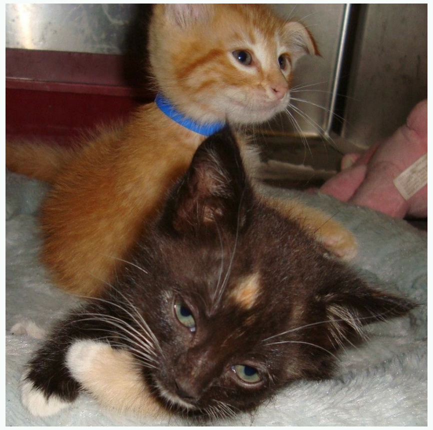

3
9
25
28
39
42
62
68
81
97

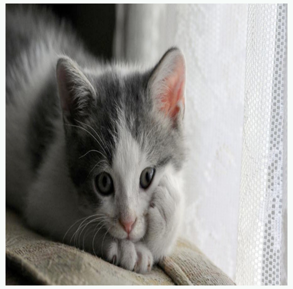
3	9	25	28	39	42	62	68	81	97

Quicksort on MTurk

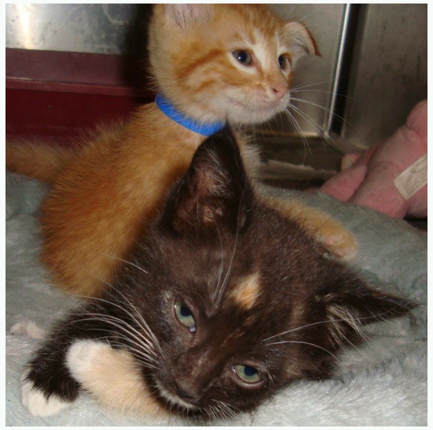
```
compare(a, b):
    hitId ← createHIT(...a...b...)
    result ← getHITResult(hitId)
    return (result says a < b)
```

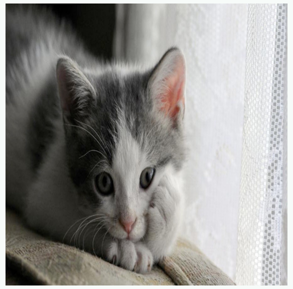


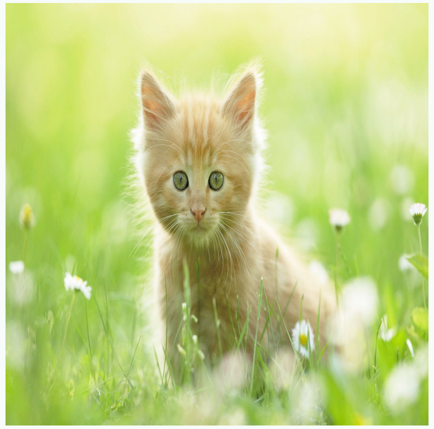




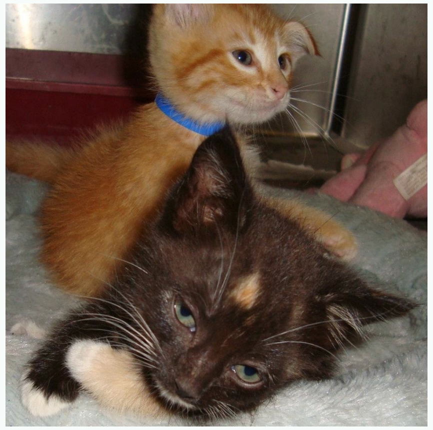
>

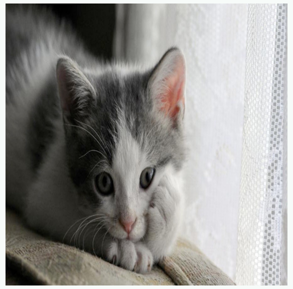




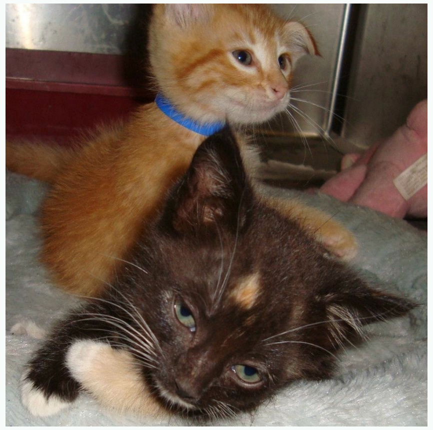


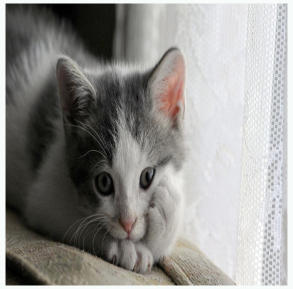
>



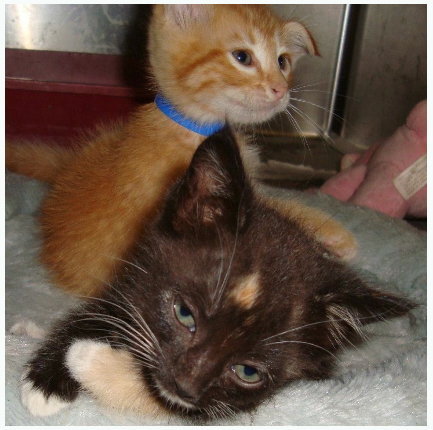


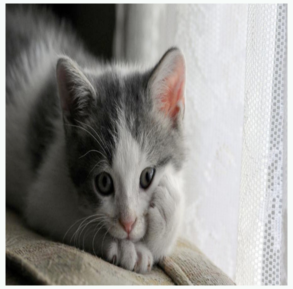
$<$



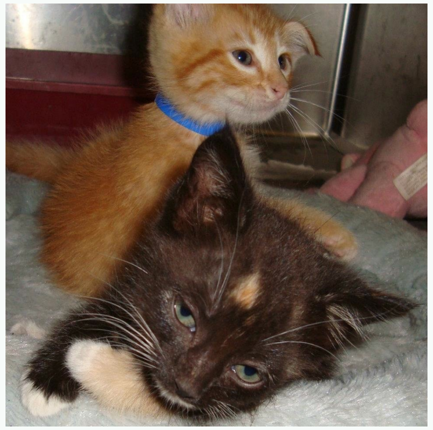


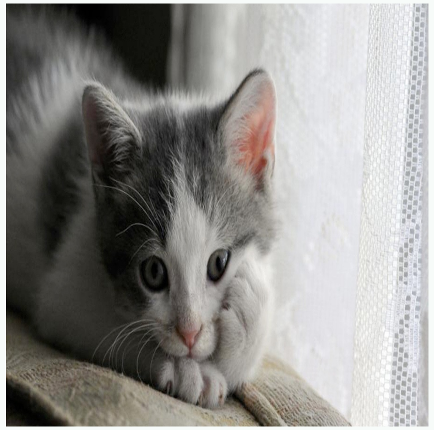
<



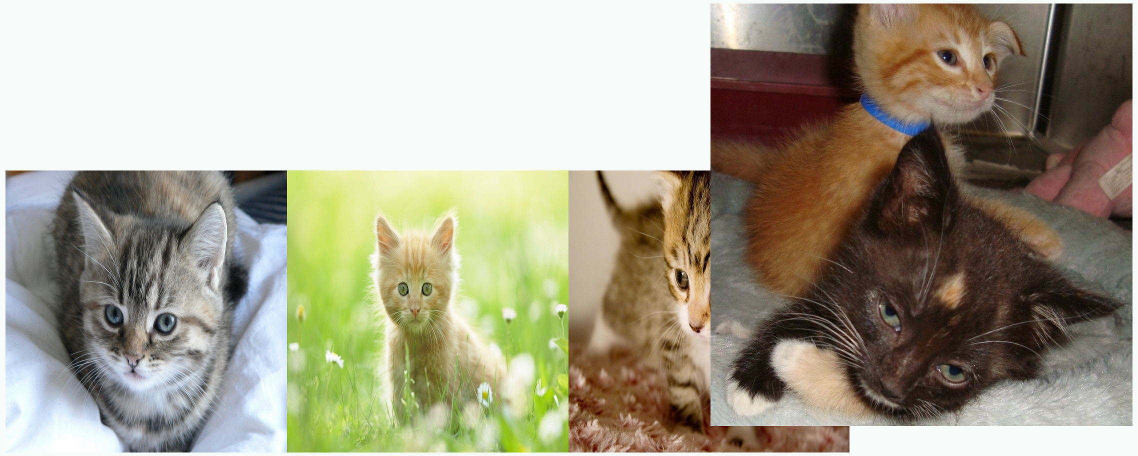


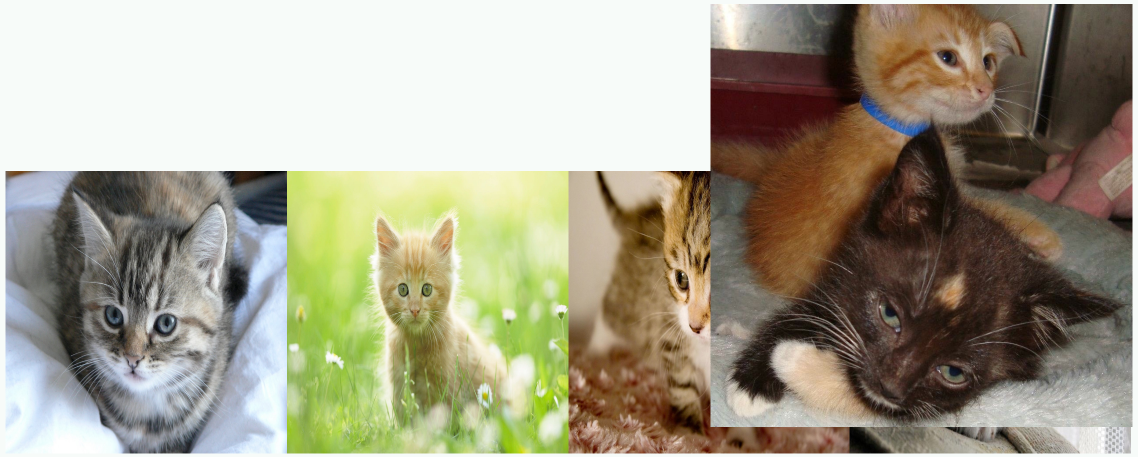
>

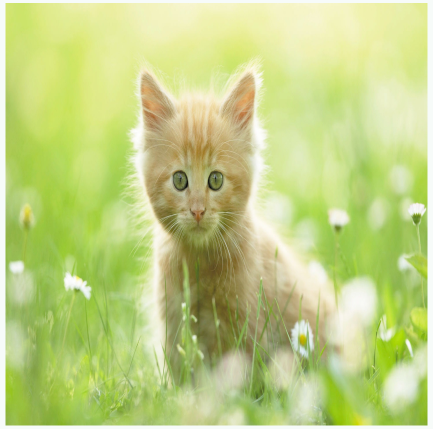




>



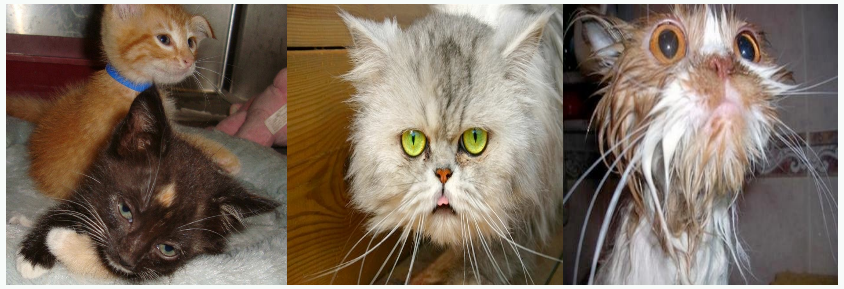




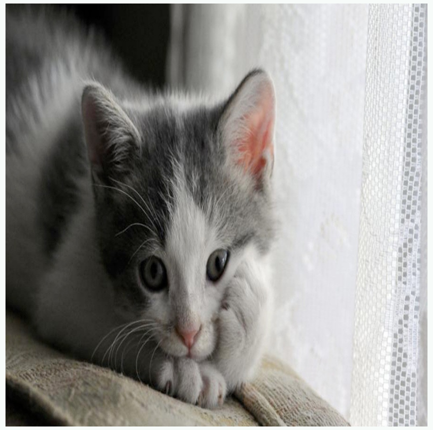
>

<

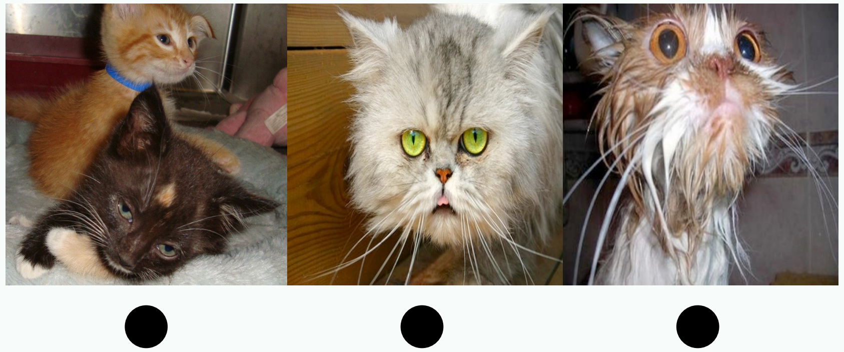
>

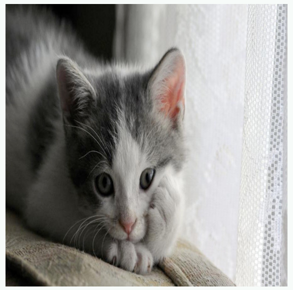


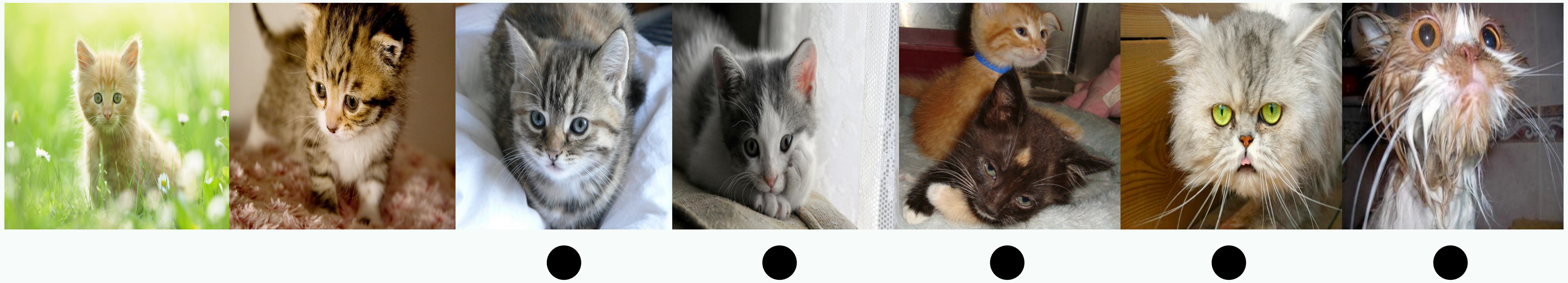
● ● ●

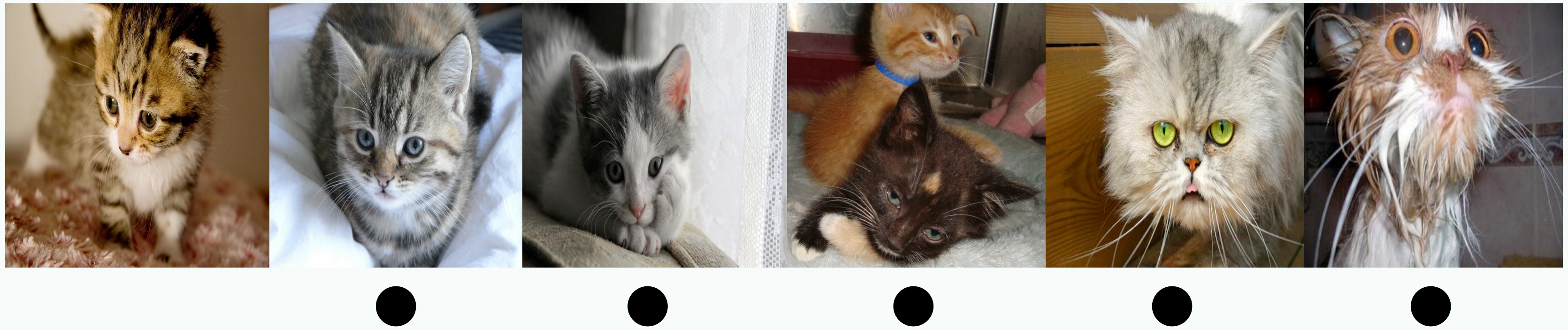


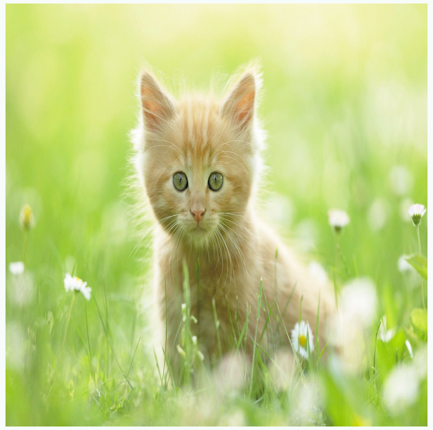
<

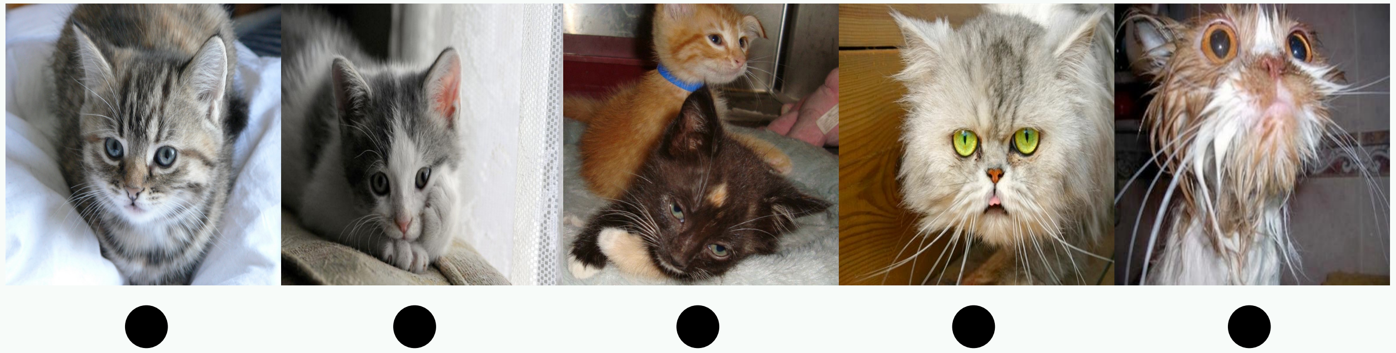




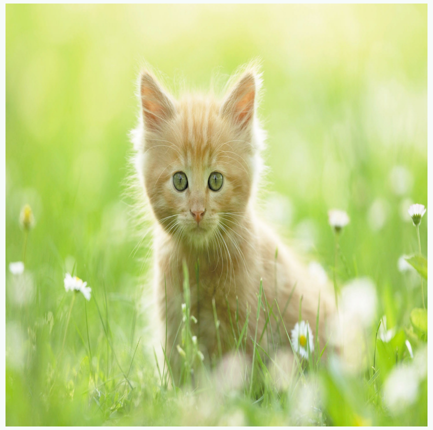


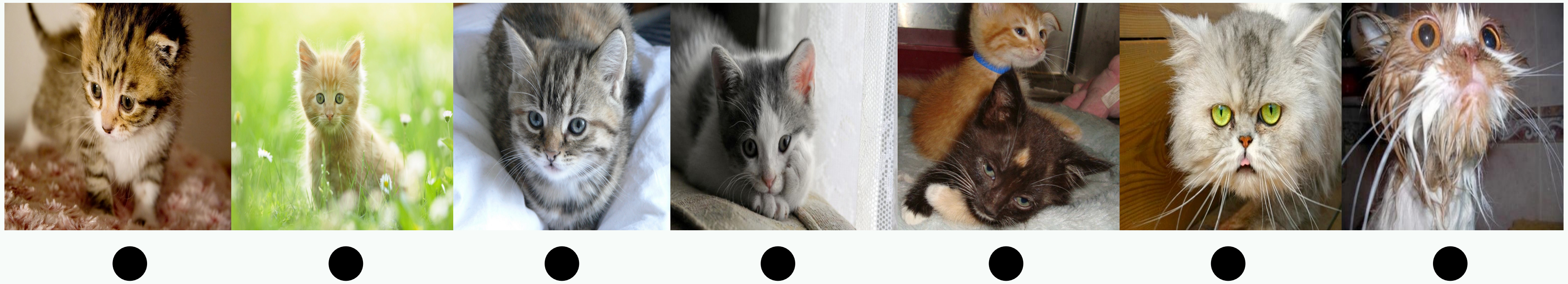






>





Quicksort as a long-running process

- With this implementation we must wait for people to complete their judgments
- The algorithm may need to run for a very long time while waiting
- Challenge: How to maintain state

Quicksort as a long-running process

- Normally quicksort maintains its state in the heap or the stack
- These are normally dynamically allocated in memory, and used by all of the programs running on a computer
- Memory isn’t typically used for hours or days
- If the computer reboots, then our program’s state would be lost and we would lose $$$

Store results in a DB

- Insight of crash-and-rerun paradigm is that if the program crashes, it should be cheap to re-run
- Use a database to store all of the results up to the place that it crashed
- Since local computation is cheap, calling DB and re-executing code with store results is cheap

New keyword once

- Costly operations can be marked in a TurKit program with keyword **once**
- **once** denotes that an operation should only be executed once across all runs of a program

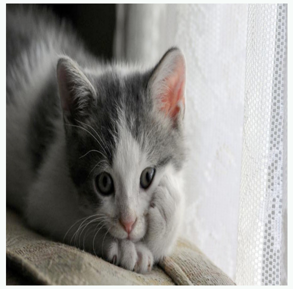
Quicksort on MTurk

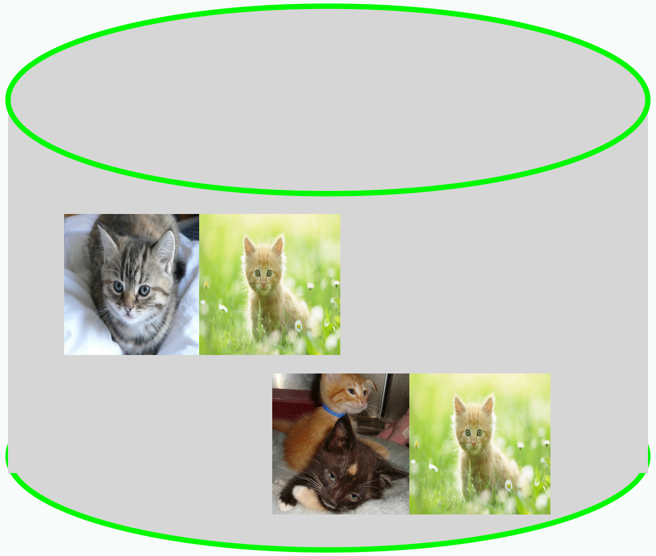
```
compare(a, b):
  hitId ← once createHIT(...a...b...)
  result ← once getHITResult(hitId)
  return (result says a < b)
```

- Subsequent runs of the program will check the database before performing these operations

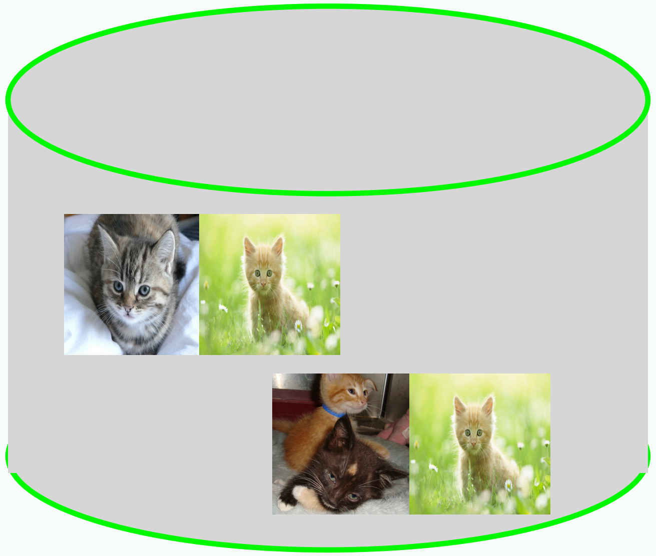
When should you mark a function with once?

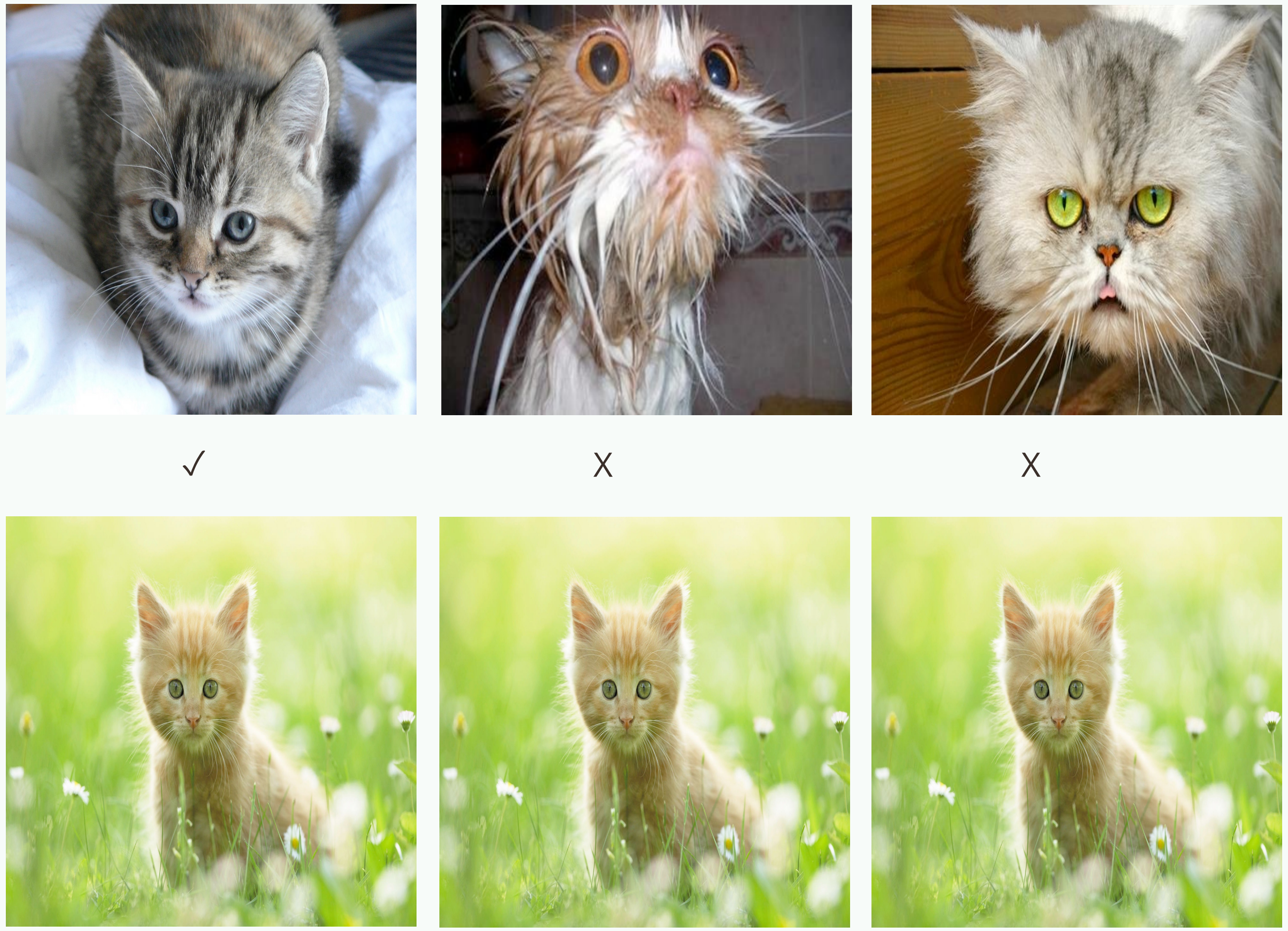
- **High cost**: This is its main usage; whenever a function is high-cost in terms of money or time, once saves the day
- **Non-determinism**: Storing results in DB assumes that the program executes in a deterministic way

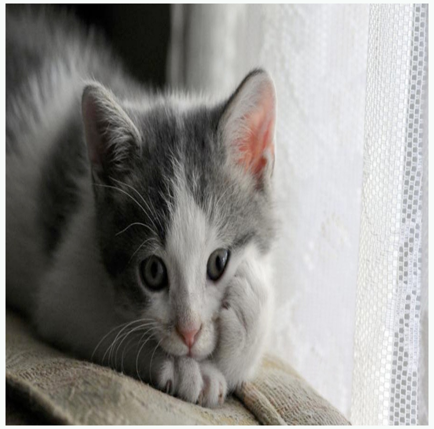




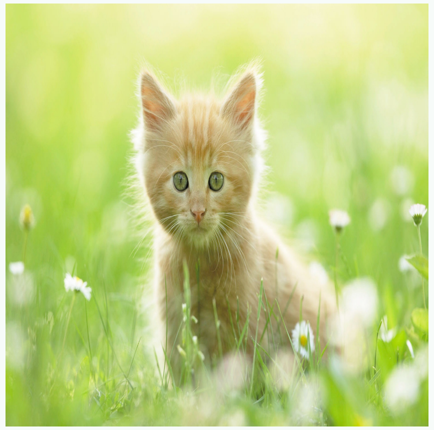
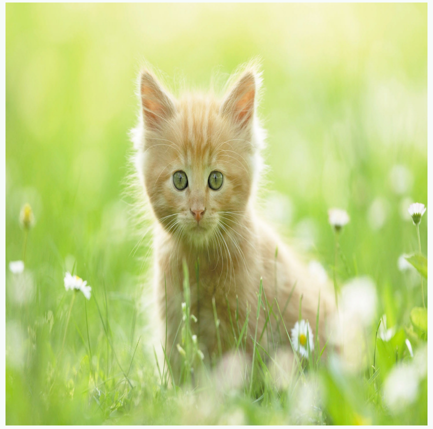
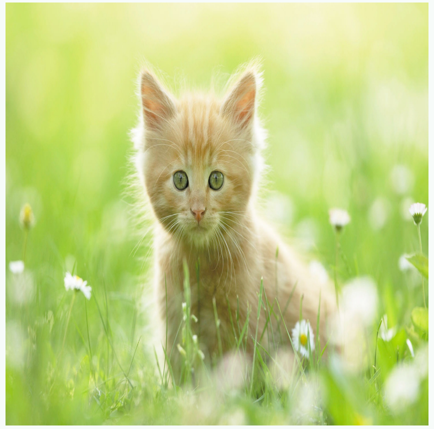
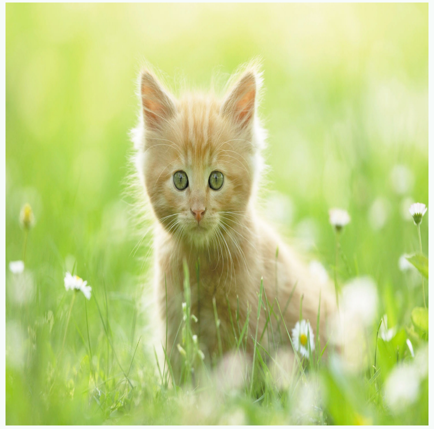
✓




✓
X
X



X X X ✓ X

Quicksort

```
quicksort(A):
  if A.length > 0:
    pivot ← A.remove(**once** A.randomIndex())
    left ← new array; right ← new array
    for x in A:
      if compare(x, pivot):
        left.add(x)
      else:
        right.add(x)
    quicksort(left)
    quicksort(right)
    A.set(left + pivot + right)
```

When should you mark a function with once?

- **Side-effects**: If a function has side effects during repeated calls, then wrap it in once

Other benefits of once

- **Incremental programming**: You can write part of an algorithm, test it, view the results, modify it, and rerun.
- **Retroactive print-line debugging**: If your program behaves in an unexpected fashion, you can put in debugging print statements after the fact
    - This also lets you print data to a file if you decide that you want to analyze it

TurKit script

- TurKit is built on top of JavaScript
- Users have full access to JavaScript
- Plus a set of APIs built around MTurk and the crash-and-rerun programming paradigm

TurKit keywords

- once
- crash
- fork / join

The crash keyword

- Why in the hell would you want to tell your program to crash?
- Since we cache results in a DB, **crash** is an alternate to **wait**
- Most common use for **crash** is waiting for results to be returned from MTurk
- TurKit automatically re-runs program after a set interval

fork allows for parallel execution

- TurKit allows multiple branches to be run in parallel via **fork**
- Calling **crash** from within a **fork**ed branch resumes the execution of the former branch
- This allows you to post multiple jobs on MTurk simultaneously
- The script can make progress on whatever path gets a result first

One HIT at a time

```
a = createHITAndWait()          // HIT A
b = createHITAndWait(...a...)   // HIT B
c = createHITAndWait()          // HIT C
d = createHITAndWait(...c...)   // HIT D
```

- B depends on A
- D depends on C
- They don’t depend on each other. Why wait?

Multiple HITs at a time

```
fork(function() {
   a = createHITAndWait()        // HIT A
   b = createHITAndWait(...a...) // HIT B
})
fork(function() {
   c = createHITAndWait()        // HIT C
   d = createHITAndWait(...c...) // HIT D
})
```

The join keyword

```
fork(...b = ...)
fork(...d = ...)
join()
e = createHITAndWait(...b...d...)
```

- **join** waits for all previous forks for finish

Calling Mechanical Turk

- TurKit adds several simple commands for interacting with MTurk:
    - **prompt**
    - **vote**
    - **sort**

Calling MTurk: prompt

```
print(mturk.prompt("When did Colorado become a state?"))
```

- **prompt** optionally allows a second argument with the number of responses

```
a = mturk.prompt("What is your favorite color?", 100)
```

Calling MTurk: vote

```
v = mturk.vote("Which is better?", [a, b])
// returns the list item with the most votes
```

- Optional 3rd argument to specify how many votes to collect

Calling MTurk: vote

```
function vote(message, options) {
   // create comparison HIT
   var h = mturk.createHITAndWait({
      ...message...options...
      assignments : 3})
   // get enough votes
   while (...votes for best option < 3...) {
      mturk.extendHIT(...add assignment...)
      h = mturk.waitForHIT(h)
   }
   return ...best option...
}
```

Calling MTurk: sort

```
ideas.sort(function (a, b) {
   v = mturk.vote("Which is better?", [a, b])
   return v == a ? -1 : 1
})
```

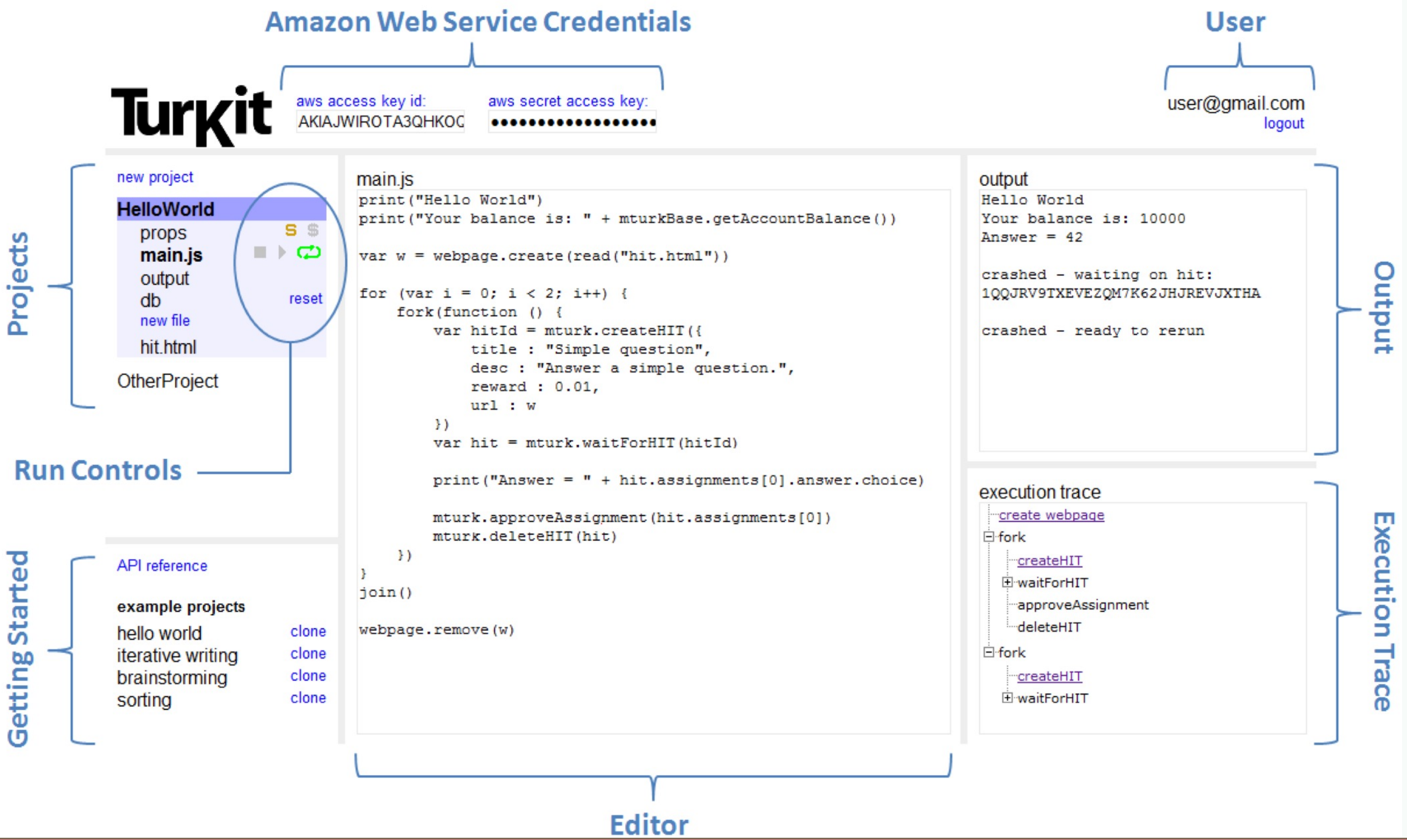
- This version just uses JavaScript’s built-in sorting function
- Defines a comparator using mturk.vote
- Negative: Comparisons are done serially

Under the hood

- TurKit is handles the MTurk API
- It generates web pages and CSS and hosts them on Amazon’s S3 server
- Nice additional features, like disabling of form elements while in preview mode
- DB is serialized using JSON

TurKit

- IDE for writing TurKit scripts, running them, and automatically rerunning them
- TurKit “crashes” after publishing a HIT; re-running polls MTurk to check for result
- Provides controls for switching from sandbox into normal MTurk, clearing DB

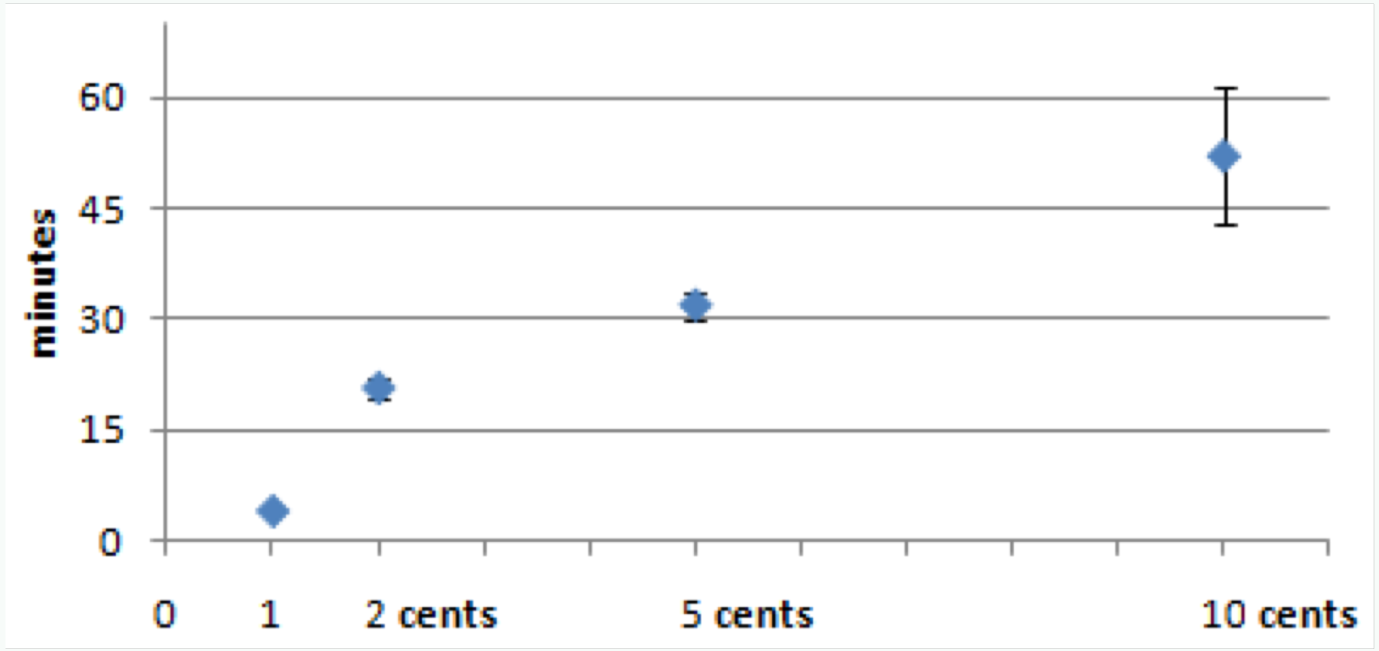


```
print("Hello World")
print("Your balance is: " + mturkBase.getAccountBalance())

var w = webpage.create(read("hit.html"))

for (var i = 0; i < 2; i++) {
    fork(function () {
        var hitId = mturk.createHIT({
            title : "Simple question",
            desc : "Answer a simple question.",
            reward : 0.01,
            url : w
        })
        var hit = mturk.waitForHIT(hitId)

        print("Answer = " + hit.assignments[0].answer.choice)

        mturk.approveAssignment(hit.assignments[0])
        mturk.deleteHIT(hit)
    })
}
join()

webpage.remove(w)
```

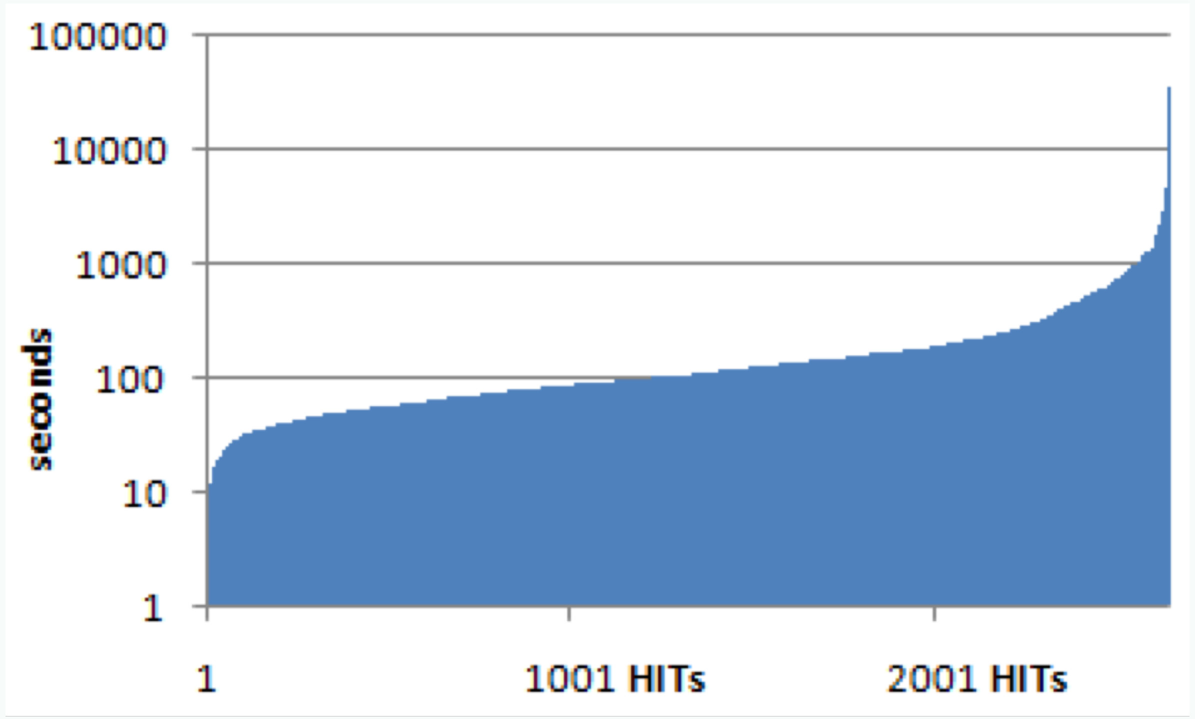
```
Hello World
Your balance is: 10000
Answer = 42

crashed - waiting on hit:
1QQJRV9TXEVEZQM7K62JHJREVJXTHA

crashed - ready to rerun
```

```
create webpage
⊟ fork
    createHIT
  ⊞ waitForHIT
    approveAssignment
    deleteHIT
⊟ fork
    createHIT
  ⊞ waitForHIT
```

Time for results to come back, by reward amount

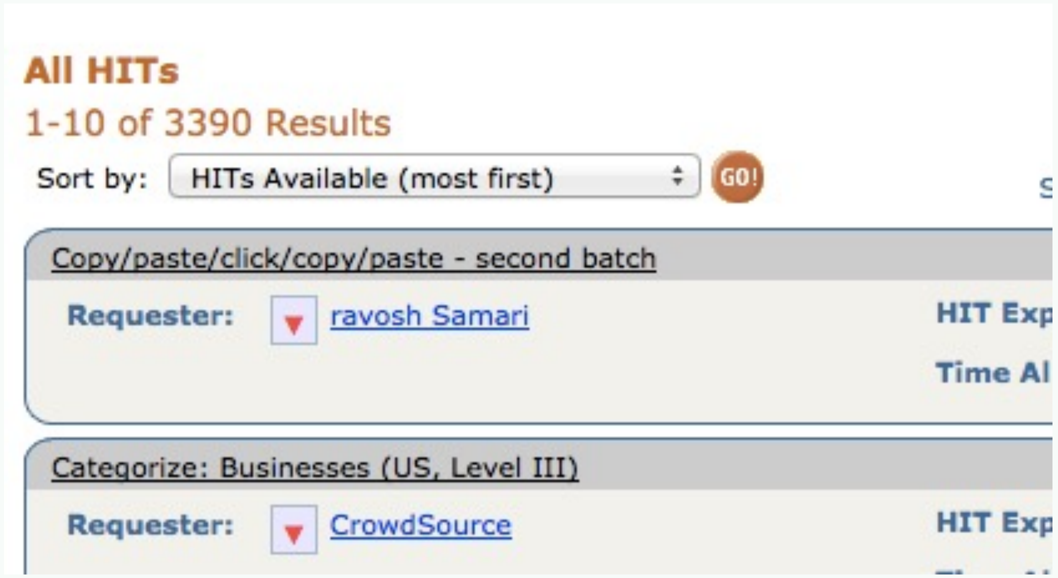


Time for first $0.01 assignment to complete



Dealing with latency

- Build the programming language to deal with high-latency operations
- Do something to optimize throughput on MTurk
- One (nefarious) example: Artificially inflate number of assignments in your HIT to get front-page placement

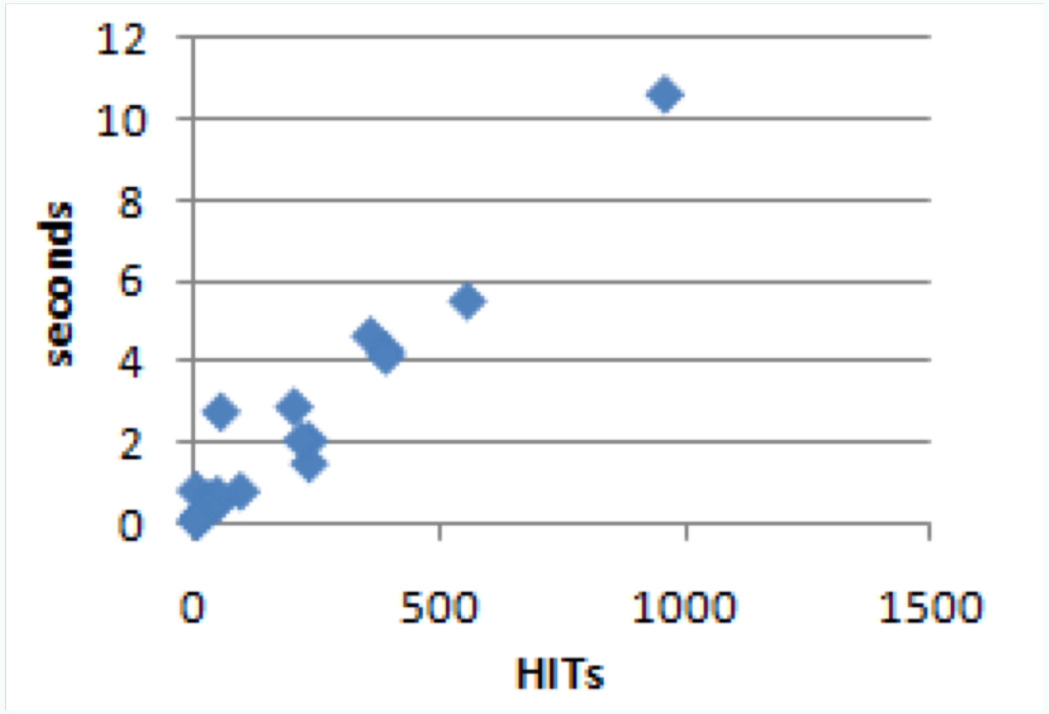


All HITs
1-10 of 3390 Results
Sort by: HITs Available (most first)

Copy/paste/click/copy/paste - second batch
Requester: ravosh Samari
HIT Exp
Time Al

Categorize: Businesses (US, Level III)
Requester: CrowdSource
HIT Exp

Time to execute once all HITs have been cached



Pros and cons of TurKit

- **Con**: Scalability – assumes local computation is minimal. Rerunning after each HIT might be tedious if task is large
- **Con**: Parallel programming – not completely general in TurKit. **once**, **fork** and **join** do not give enough state.
- **Con**: Experimental replicability – usually one downside of human computation is that results with differ each time. Not so with TurKit!

What experiments would you run?