# Algorithms

## Introduction

Morgan Ericsson

# Today

- Introduction
- Introduction to algorithms and data structures

# About the course

# Welcome

# Expectations

- We expect that you:
  - Stay up to date with material posted on the Moodle room (and resources linked from it) and the Slack channel
  - Do you best and ask for help if you get stuck
  - Treat teachers and other students with respect
- You can expect that we:
  - Do our best to support your learning

# Course management

- Course reponsible and examiner:
  - Morgan Ericsson, @morganericsson, morgan.ericsson@lnu.se
- Administration:
  - Ewa Püschl, eva.puschl@lnu.se

# Registration

- If you have not registered, please do
- If you cannot register:
  - Check that you passed the prerequisities
  - If you have, contact me
- The activity control after three weeks will be based on the first assignment
  - You need to have submitted something that “could” work
  - And passes a plagiarism check

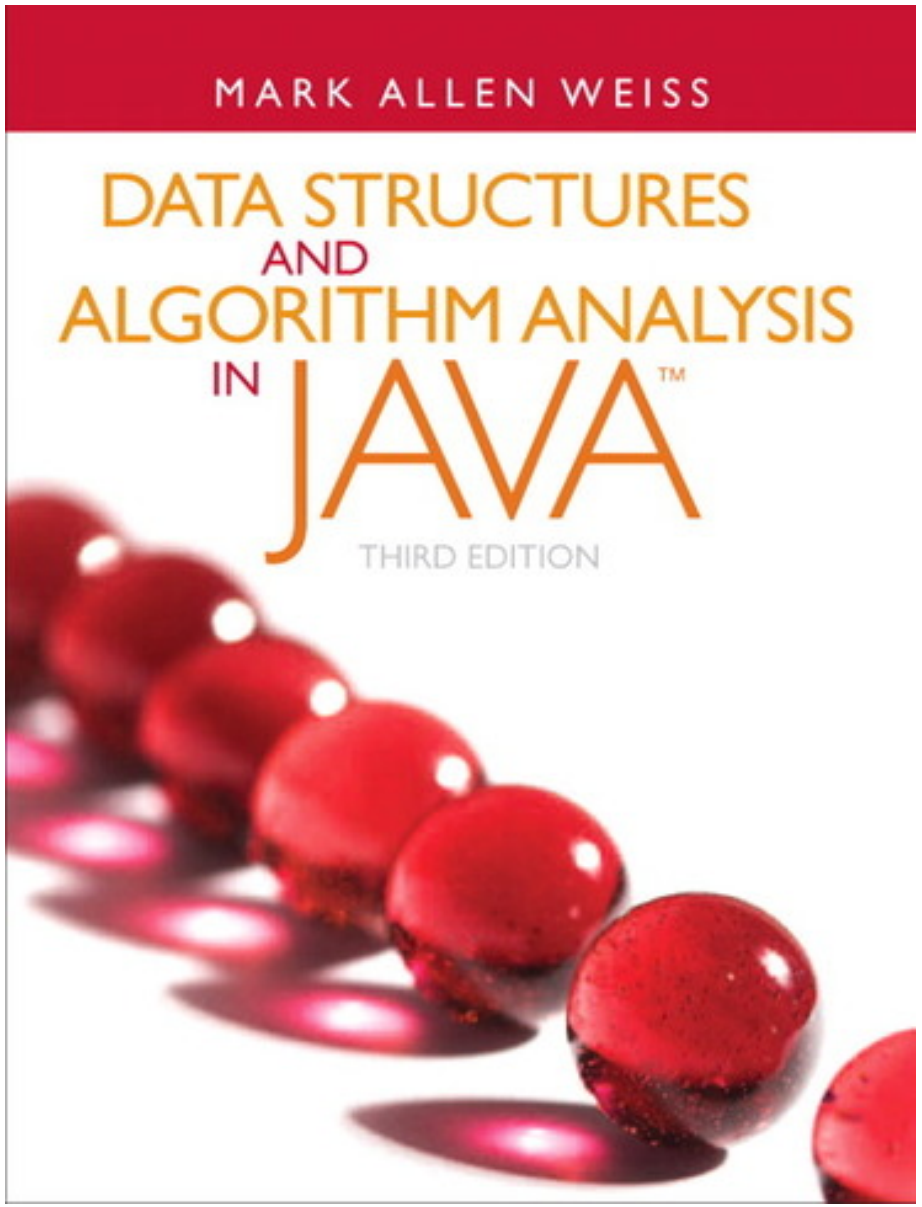
# Communications

- All static information is available via Moodle
- Assignments are submitted via Moodle
- Questions and discussion on Slack
  - Use the course channel as much as possible
  - No DMs unless you need to ask or tell something private
- Try to avoid email. If you must, see policy for DMs

# Examination

- The examination is in two parts:
  - Oral exam, 3 hp
  - Programming assignments, 2 hp
- You need to pass both to pass the course
- Final grade 60% exam and 40% assignments

# Learning resources



# Course evaluation

- There will be a course evaluation at the end of the course
- It is important that you give honest feedback …
- … and help improve future versions of the course
- Feedback during the course is also welcome

# Practical details

# Code examples

- Most code examples will be in Python
  - Might introduce new features
  - So, ask if you do not understand
- You will submit your assignments in Java
- All code from the slides is available on GitLab (link on Moodle)

# Plagiarism

- All submissions will be automatically checked for plagiarism
  - If you are flagged, you will failed the assignment
  - And maybe reported to the disciplinary board
- You are here to learn, so do not copy code from the Internet
  - Instead, understand and adapt!

# Difficult topic

- Algorithms can be difficult!
- Read the book
- Check the slides
- Ask questions
- Start early!

# Introduction to algorithms

# What?

- An algorithm is a method for solving a problem
  - E.g., sorting or searching
- A data structure is a method to store information
  - E.g., a tree or a linked list

# Why?

- Algorithms and data structures are everywhere!
  - Networking and web search
  - AI and machine learning
  - Physics simulations
  - Video compression
  - Security and encryption
  - …

# Pure problem solving

- The design of an algorithm or a data structure is about creating a solution to a problem
- Designing a good and efficient solution can be challenging
  - So, study to avoid repeating
- *“great algorithms are the poetry of computation”*

# To be a good programmer

*I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important.*

*Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*

# New concepts

- Old roots
  - Euclid’s algorithm (GCD) is an old example
  - Formalized by Church and Turing in the 1930s
- New concepts …
  - Scalability, computatbility, …
- … that make you a better programmer

# What will you learn?

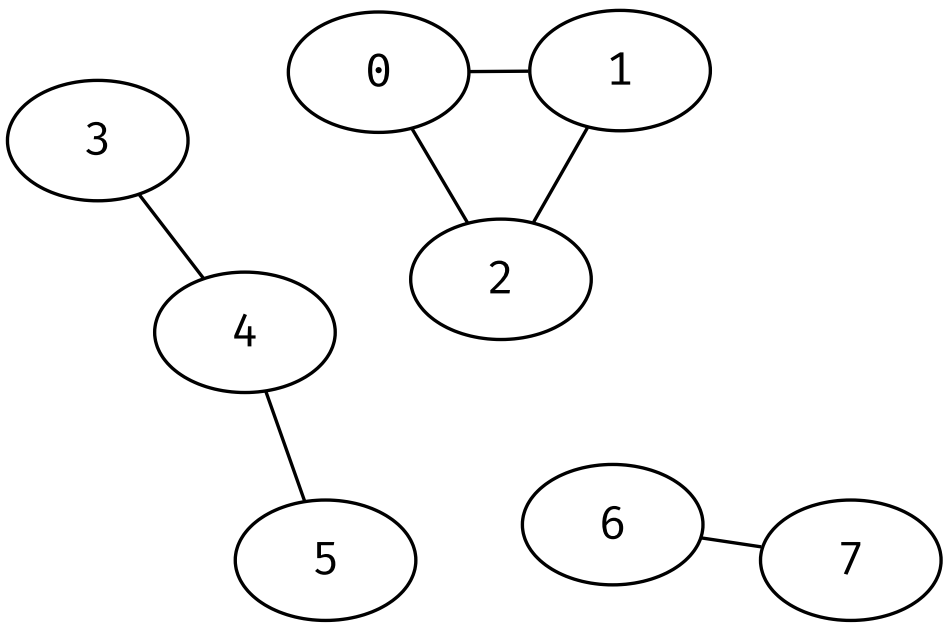
- List, stacks, and queues
- Trees
- Hashing
- Sorting
- Graphs
- Algorithm design, e.g., divide and conquer
- Algorithm analysis, e.g., Big-Oh
- N, P, and NP

# An example

# Note

- This example will introduce some new ideas
- We will return to a lot of what we discuss
- So, try to keep up, but do not expect to understand everything
- Pay attention, you will implement it!

# The problem




Given a set of objects, is there a path that connects two specific objects, e.g., 0 and 5.

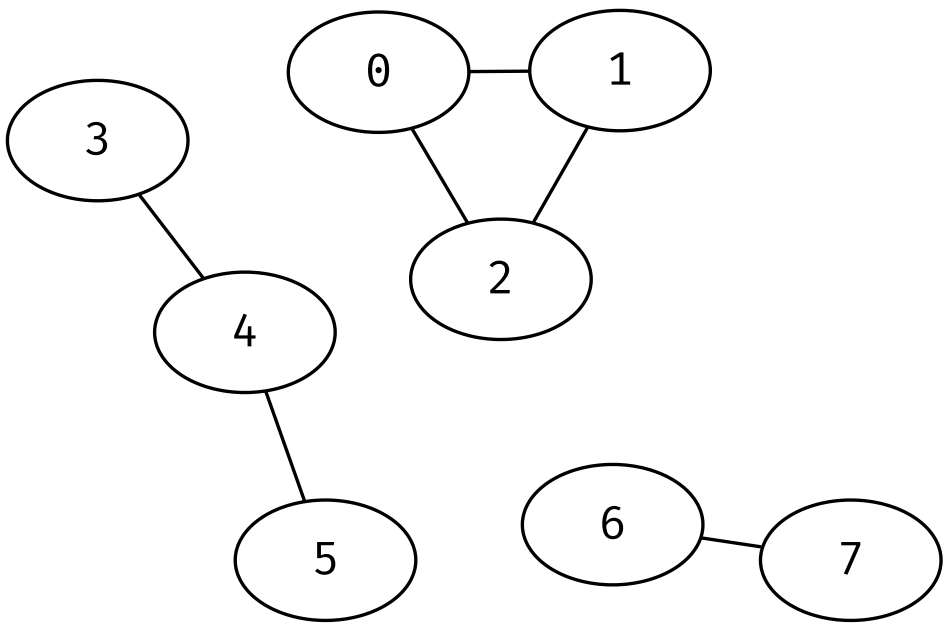
# Solution / API

- Data structure with two operations:
  - Union connects two objects:
    - `union(a:int, b:int) -> None`
  - Connected determins whether two objects are connected:
    - `connected(a:int, b:int) -> bool`
- To simplify, we use an integer index to identify the objects

# Connections

- We model the connection as an equivalence relation
  - Reflective, i.e., $a$ is connected to $a$
  - Symmetric, i.e., if $a$ is connected to $b$, then $b$ is connected to $a$
  - Transitive, i.e., if $a$ is connected to $b$ and $b$ is connected to $c$ then $a$ is connected to $c$.
- A Connected component is a maximal set of objects that are mutually connected

# Example




There are three connected components: `{0,1,2}`, `{3,4,5}`, and `{6,7}`.

# A first attempt

- We use the connected components to formulate a solution
- Each connected component has an id that identifies it
  - When two objects are connected, they get the same id
  - If two objects have the same id, they are connected

# A first attempt

```
1  def init(N:int) -> list[int]:
2    return list(range(N))
3
4  def connected(d:list[int], a:int, b:int) -> bool:
5    return d[a] == d[b]
```

# A first attempt

```
1  def union(d:list[int], a:int, b:int) -> None:
2    a_id = d[a]
3    b_id = d[b]
4
5    for ix, v in enumerate(d):
6      if v == a_id:
7        d[ix] = b_id
```

# A first attempt

```
1 uf = init(8)
2 print(uf)
3 union(uf, 0, 1)
4 union(uf, 6, 7)
5 print(uf)
6 union(uf, 1, 2)
7 print(uf)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
[1, 1, 2, 3, 4, 5, 7, 7]
[2, 2, 2, 3, 4, 5, 7, 7]
```

# What if?

```
1 # uf = [2, 2, 2, 3, 4, 5, 7, 7]
2 union(uf, 0, 6)
3 print(uf)
```

```
[7, 7, 7, 3, 4, 5, 7, 7]
```

# Using a class

```
1  class UnionFind:
2    def __init__(self, N:int) -> None:
3      self.d = list(range(N))
4
5    def connected(self, a:int, b:int) -> bool:
6      return self.d[a] == self.d[b]
```

# Using a class

```
 1  from fastcore.basics import patch
 2
 3  @patch
 4  def union(self:UnionFind, a:int, b:int) -> None:
 5    a_id = self.d[a]
 6    b_id = self.d[b]
 7
 8    for ix, v in enumerate(self.d):
 9      if v == a_id:
10        self.d[ix] = b_id
```

# Adding a nice print

```
 1  @patch
 2  def __str__(self:UnionFind) -> str:
 3    tmpd = {}
 4    for ix, v in enumerate(self.d):
 5      if v not in tmpd:
 6          tmpd[v] = []
 7      tmpd[v].append(ix)
 8
 9    s = ''
10    for k, v in tmpd.items():
11      s += f'{{{",".join(map(str, v))}}} '
12    return s
```

# Example

```
 1  uf = UnionFind(8)
 2  uf.union(0, 1)
 3  uf.union(1, 2)
 4  uf.union(3, 4)
 5  uf.union(4, 5)
 6  uf.union(6, 7)
 7
 8  assert uf.connected(1, 2)
 9  assert not uf.connected(0, 5)
10  print(uf)
```

```
{0,1,2} {3,4,5} {6,7}
```

# Good enough?

- How do we evaluate?
  - Correct? Yes
  - Speed?
  - Memory use?

# Good enough?

```
1 uf = UnionFind(1_000_000)
2 %timeit uf.union(0, 1)
```

```
46.9 ms ± 2.11 ms per loop (mean ± std. dev. of 7 runs, 10
loops each)
```

# Good enough?

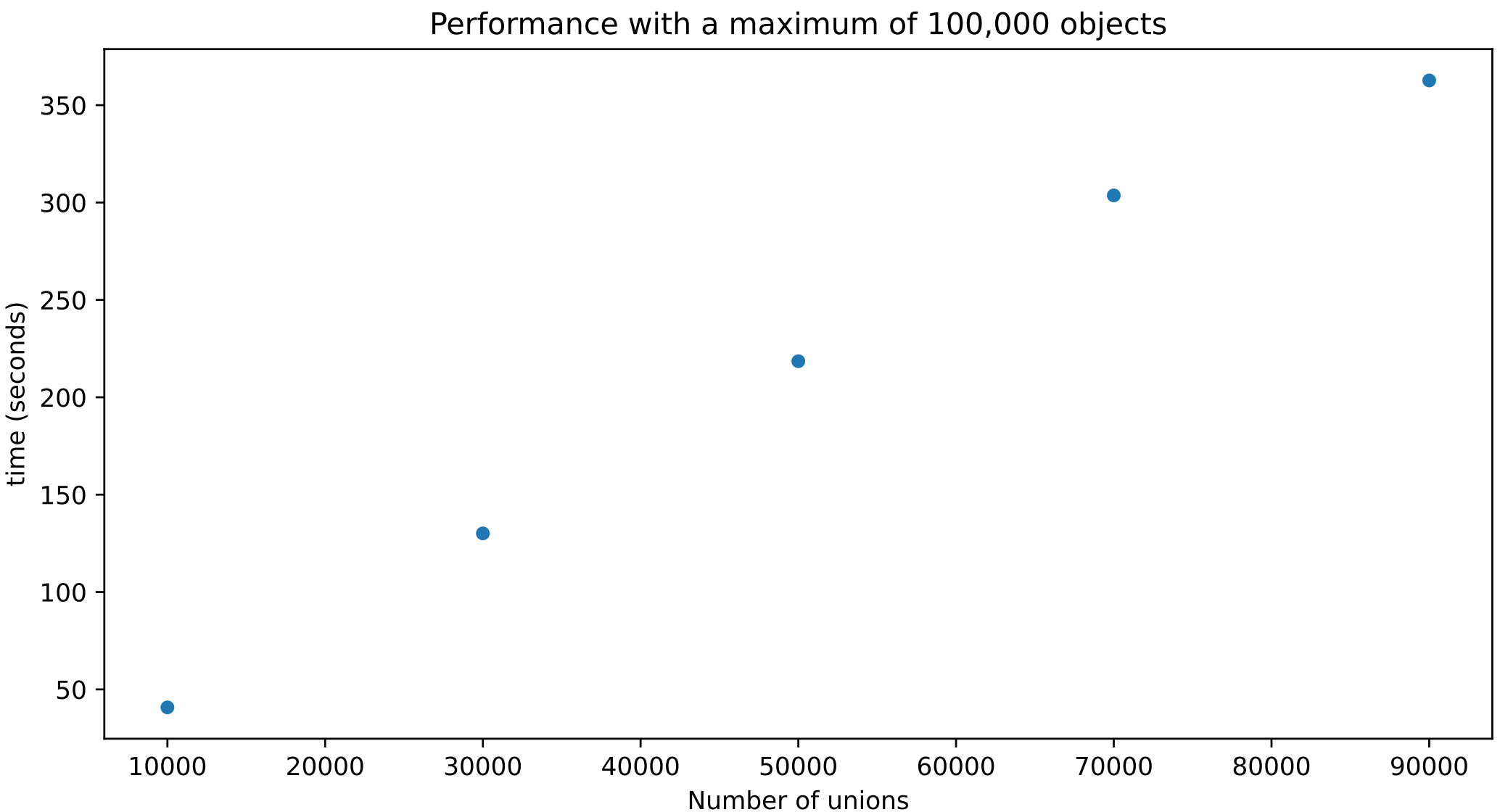
```
1  %timeit uf.connected(0, 1)
```

```
108 ns ± 8.23 ns per loop (mean ± std. dev. of 7 runs,
10,000,000 loops each)
```

# Why?

```
 1  # Fast
 2  def connected(self, a:int, b:int) -> bool:
 3    return self.d[a] == self.d[b]
 4
 5  # Slower
 6  def union(self:UnionFind, a:int, b:int) -> None:
 7    a_id = self.d[a]
 8    b_id = self.d[b]
 9
10    for ix, v in enumerate(self.d):
11      if v == a_id:
12        self.d[ix] = b_id
```

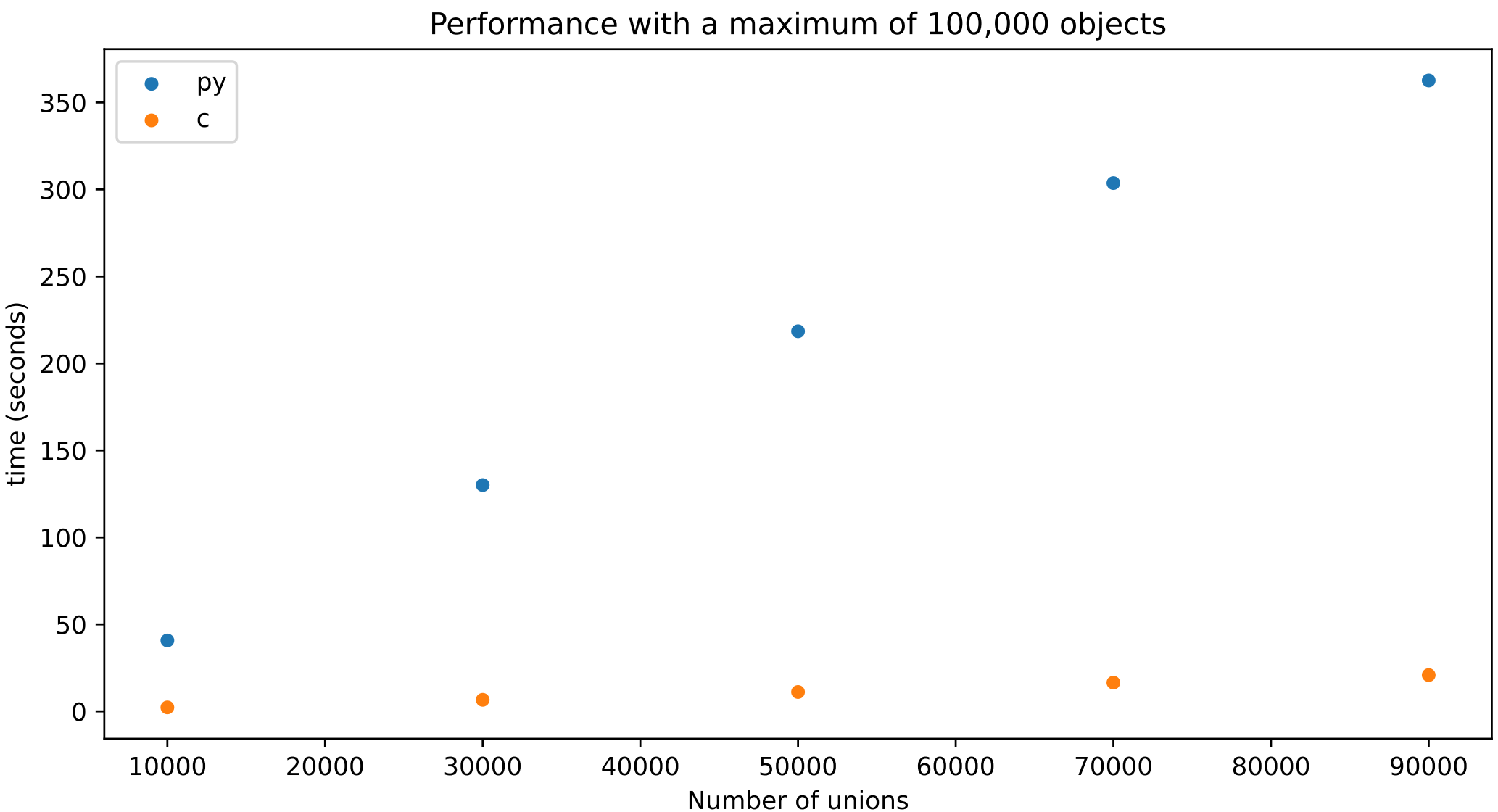
# How bad?


Performance with a maximum of 100,000 objects
time (seconds)
350
300
250
200
150
100
50
10000
20000
30000
40000
50000
60000
70000
80000
90000
Number of unions

# Python is slow?

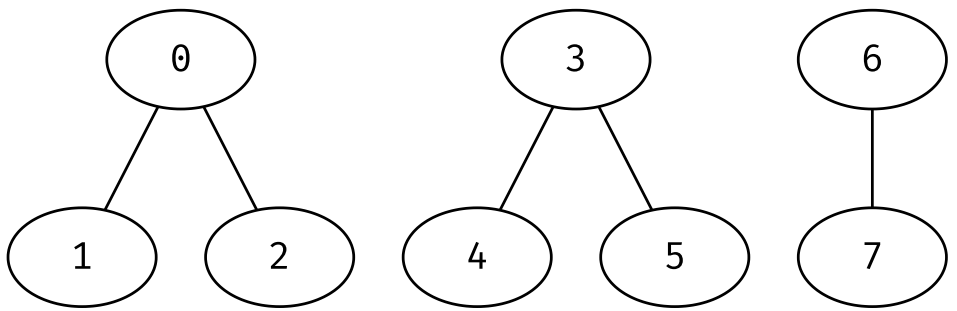
```c
int *init(int n) {
  int *res = calloc(n, sizeof(int));
  for(int i=0; i<n; i++) res[i] = i;
  return res;
}

int connected(int *uf, int a, int b) {
  return (uf[a] == uf[b]);
}

void union_f(int *uf, int sz, int a, int b) {
  int id_a = uf[a];
  int id_b = uf[b];

  for(int i=0; i<sz; i++)
    if(uf[i] == id_a)
      uf[i] = id_b;
}
```

# Python is slow?



# We can do better

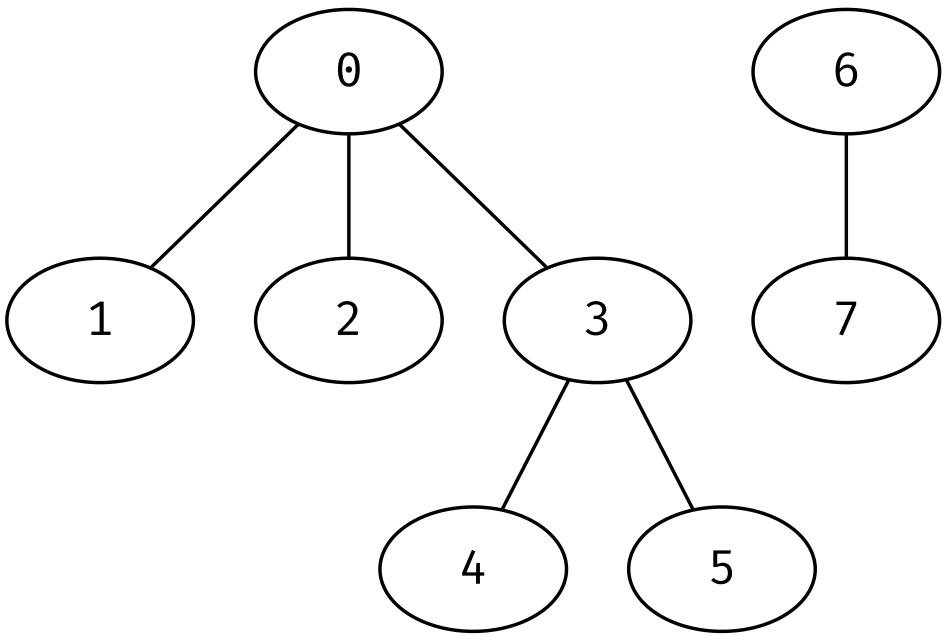
# Another approach (quick union)




- Each component is represented as a tree
- Same root means same component
- When adding an object to a component, its root is connected to the root of the component

# Another approach

» After `union(0, 3)`



# Quick Union

```
 1  def init(N:int) -> list[int]:
 2    return list(range(N))
 3
 4  def connected(d:list[int], a:int, b:int) -> bool:
 5    return root(d, a) == root(d, b)
 6
 7  def union(d:list[int], a:int, b:int) -> None:
 8    ra = root(d, a)
 9    rb = root(d, b)
10    d[ra] = rb
```

# Quick Union

```
1 def root(d:list[int], a:int) -> int:
2   while a != d[a]:
3     a = d[a]
4   return a
```

# Quick Union

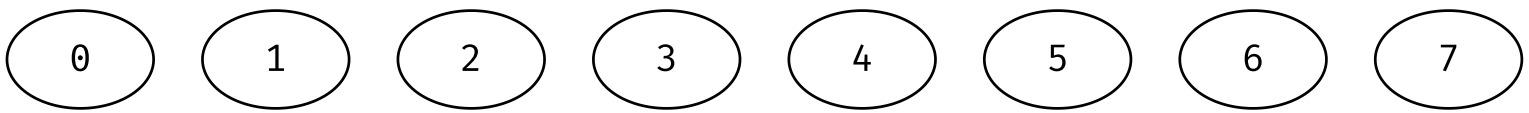
```
1 uf = init(8)
2 union(uf, 0, 1)
3 union(uf, 1, 2)
4 union(uf, 3, 4)
5 union(uf, 4, 5)
6 union(uf, 6, 7)
7
8 assert connected(uf, 1, 2)
9 assert not connected(uf, 0, 5)
```

# How does it work?

```
1 uf = init(8)
2 print(uf)
3 union(uf, 0, 1)
4 union(uf, 1, 2)
5 print(uf)
6 union(uf, 3, 4)
7 print(uf)
```

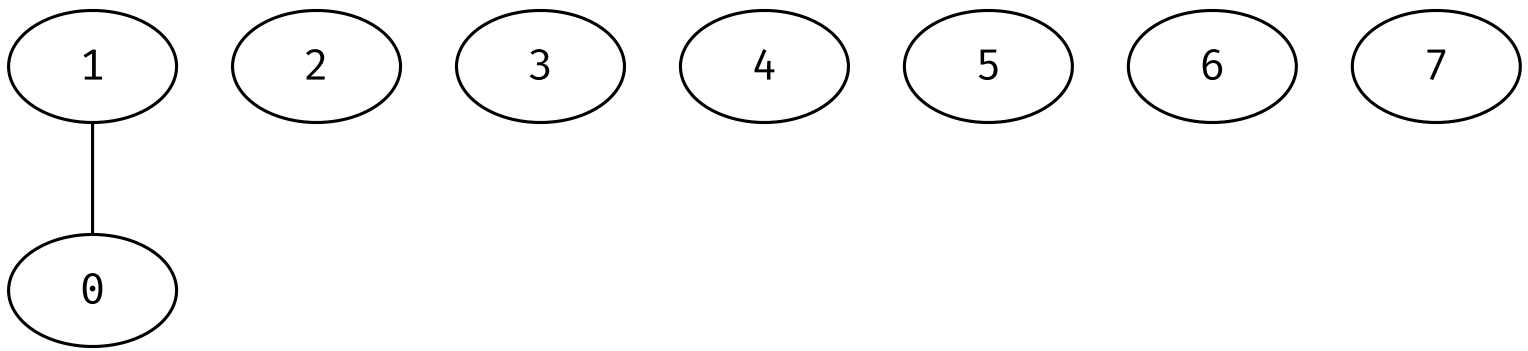
```
[0, 1, 2, 3, 4, 5, 6, 7]
[1, 2, 2, 3, 4, 5, 6, 7]
[1, 2, 2, 4, 4, 5, 6, 7]
```

# How does it work?



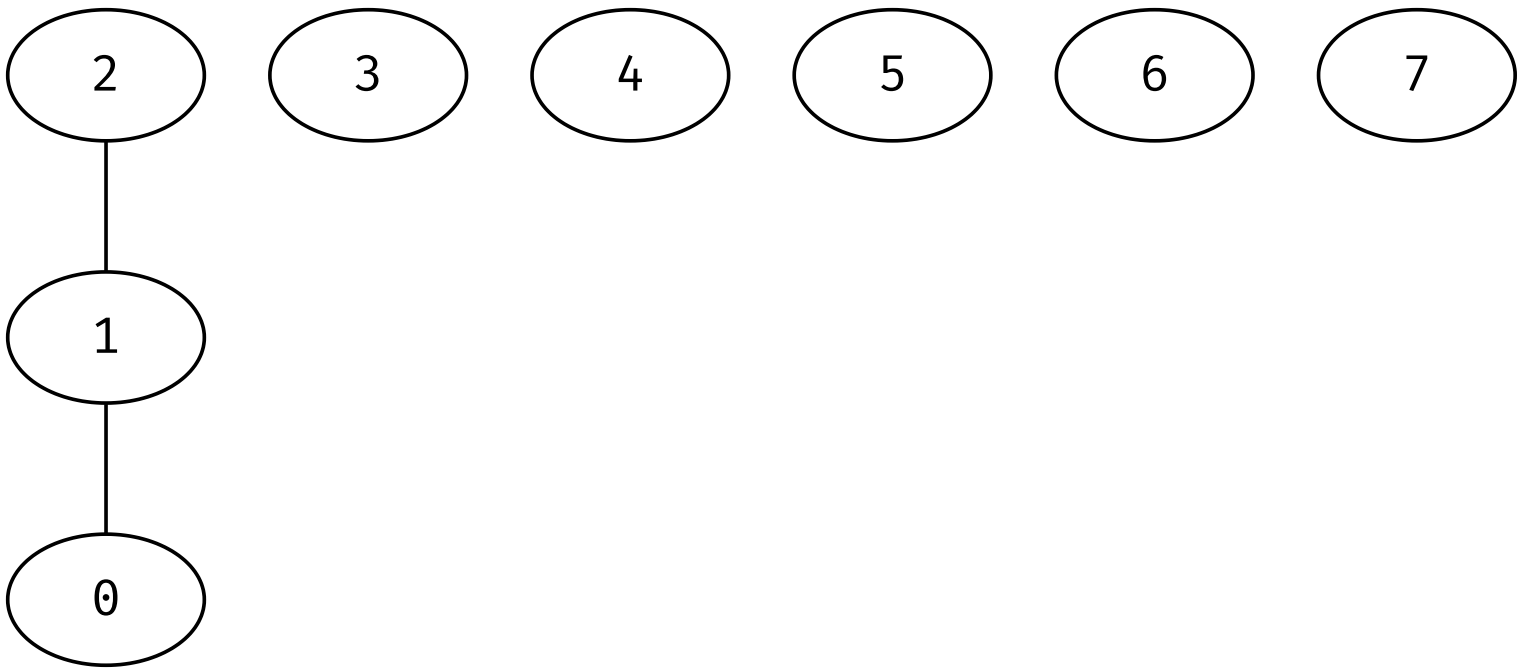

```
init(8)
```

# How does it work?



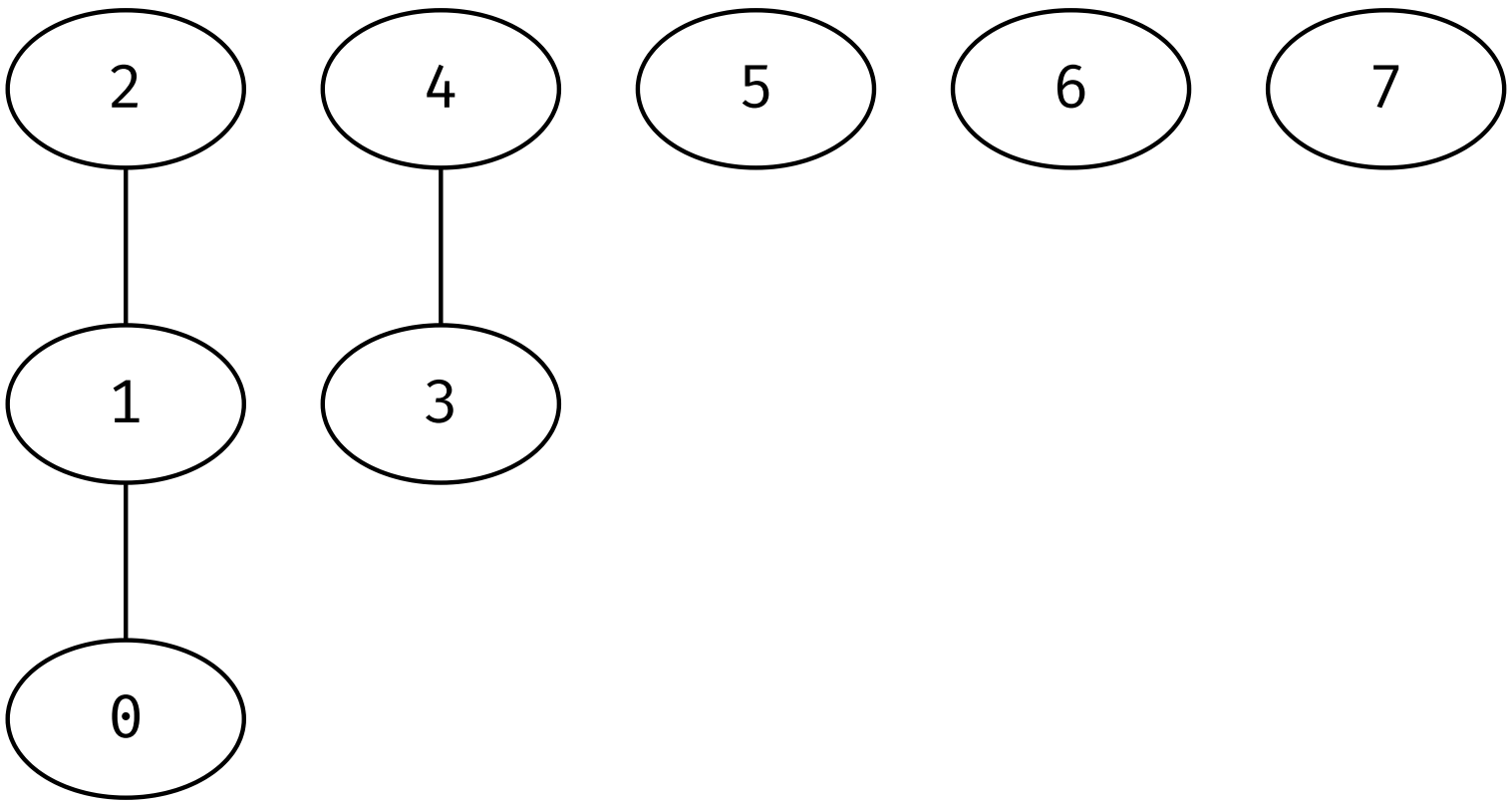

```
init(8),union(0, 1)
```

# How does it work?



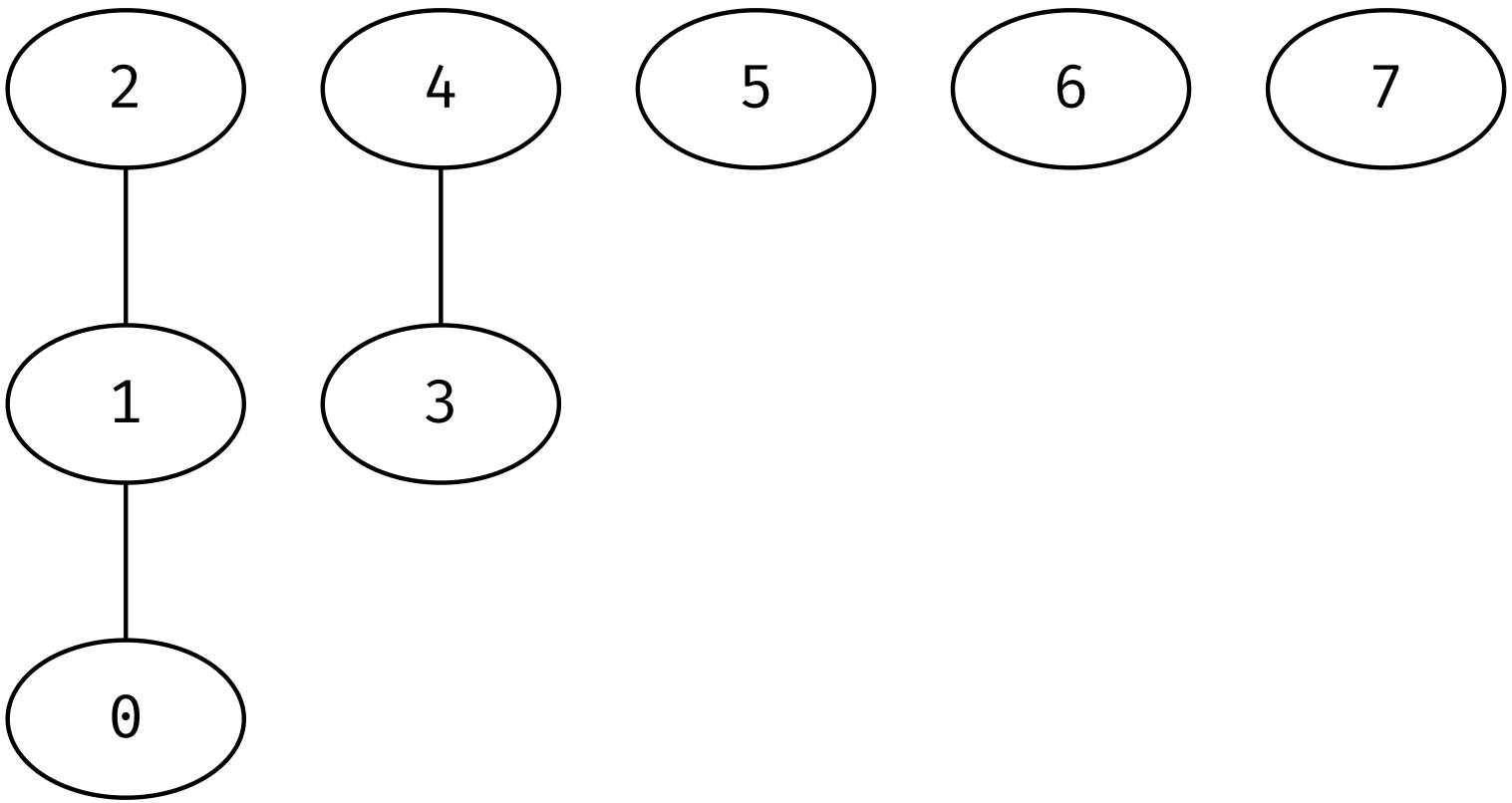

```
init(8),union(0, 1),union(1, 2)
```

# How does it work?




```
init(8),union(0, 1),union(1, 2),union(3, 4)
```

# How does it work?



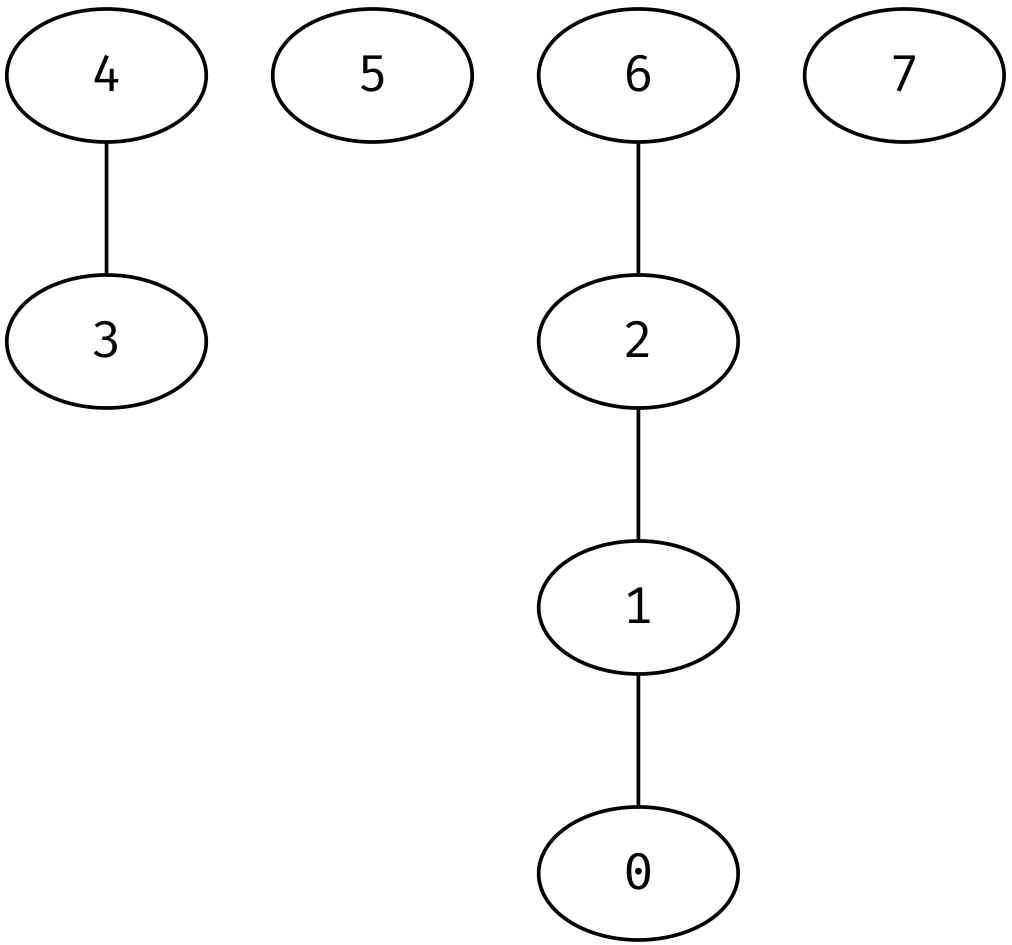

`connected(1, 3)`? No.

# How does it work?

```
1 # uf = [1, 2, 2, 4, 4, 5, 6, 7]
2 union(uf, 0, 6)
3 print(uf)
```

```
[1, 2, 6, 4, 4, 5, 6, 7]
```

# How does it work?



# Moving to a class

```
 1  class QUnionFind:
 2    def __init__(self, N:int) -> None:
 3      self.d = list(range(N))
 4
 5    def connected(self, a:int, b:int) -> bool:
 6      return self.root(a) == self.root(b)
 7
 8    def union(self, a:int, b:int) -> None:
 9      ra = self.root(a)
10      rb = self.root(b)
11      self.d[ra] = rb
```

# Moving to a class

```
1  @patch
2  def root(self:QUnionFind, a:int) -> int:
3    while a != self.d[a]:
4      a = self.d[a]
5    return a
```

# Better?

```
1  quf = QUnionFind(1_000_000)
2  %timeit quf.union(0, 1)
```

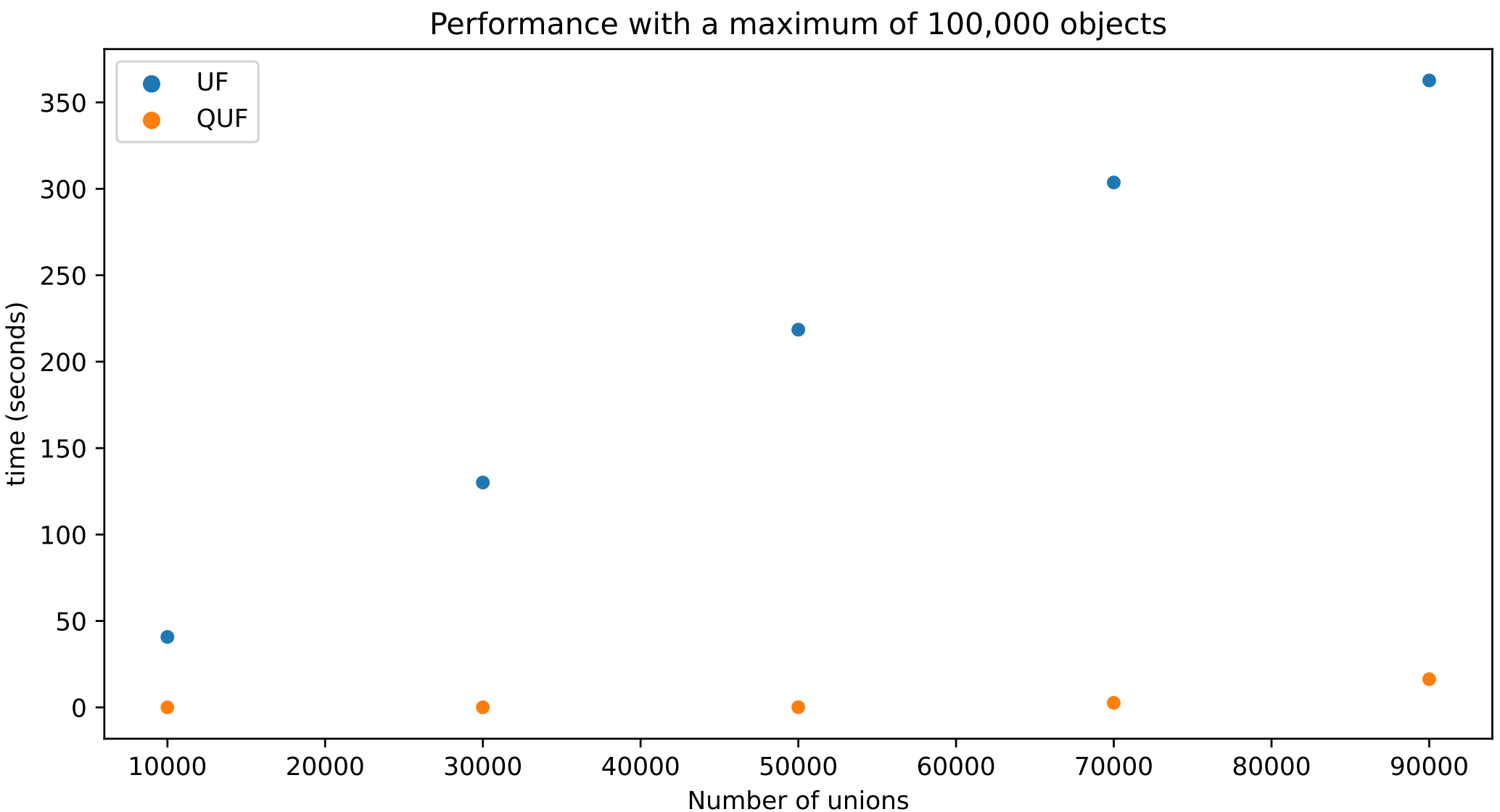
```
246 ns ± 5.87 ns per loop (mean ± std. dev. of 7 runs,
1,000,000 loops each)
```

# Better?

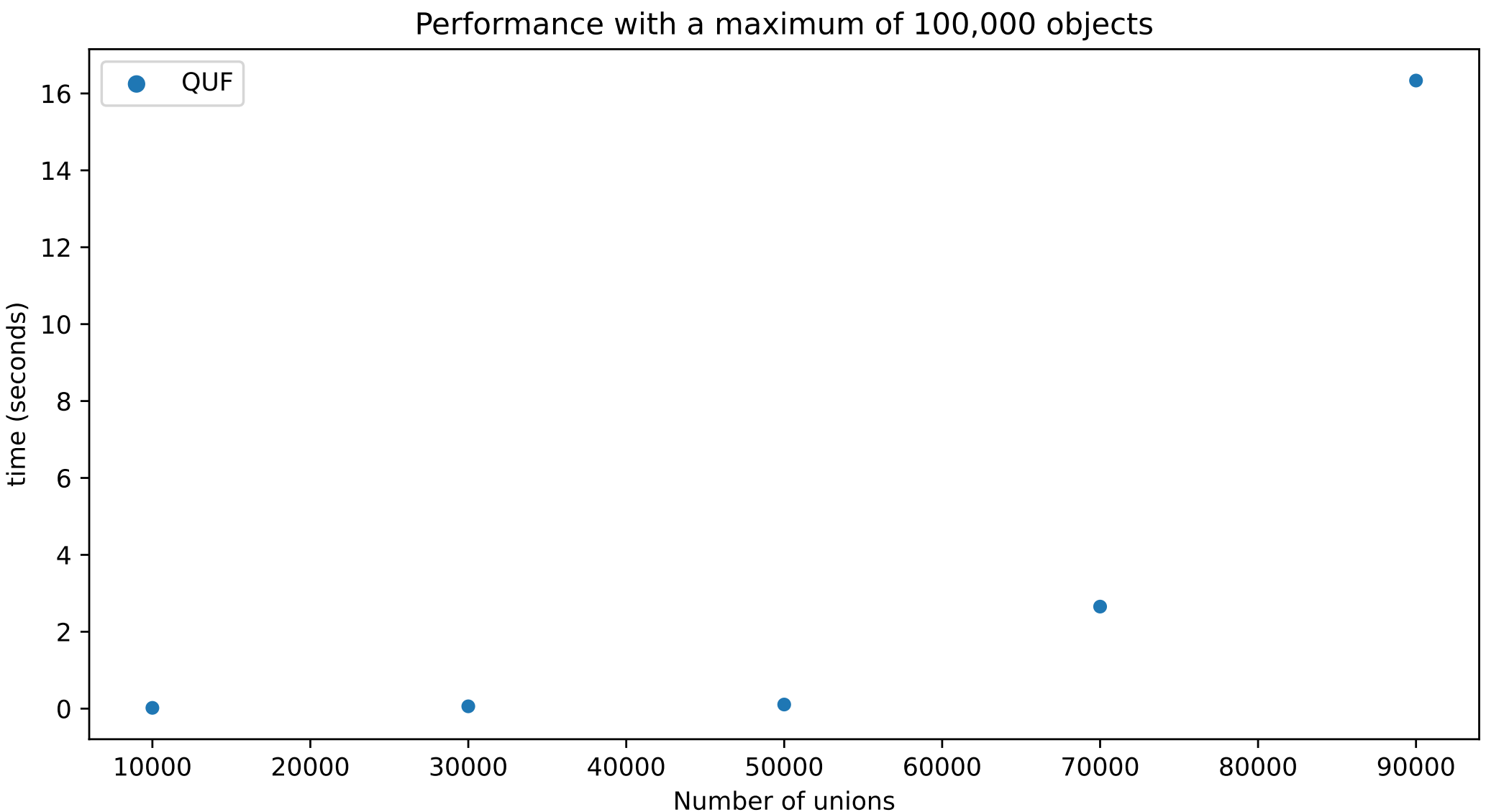
```
1  %timeit quf.connected(0, 1)
```

```
220 ns ± 3.07 ns per loop (mean ± std. dev. of 7 runs,
1,000,000 loops each)
```

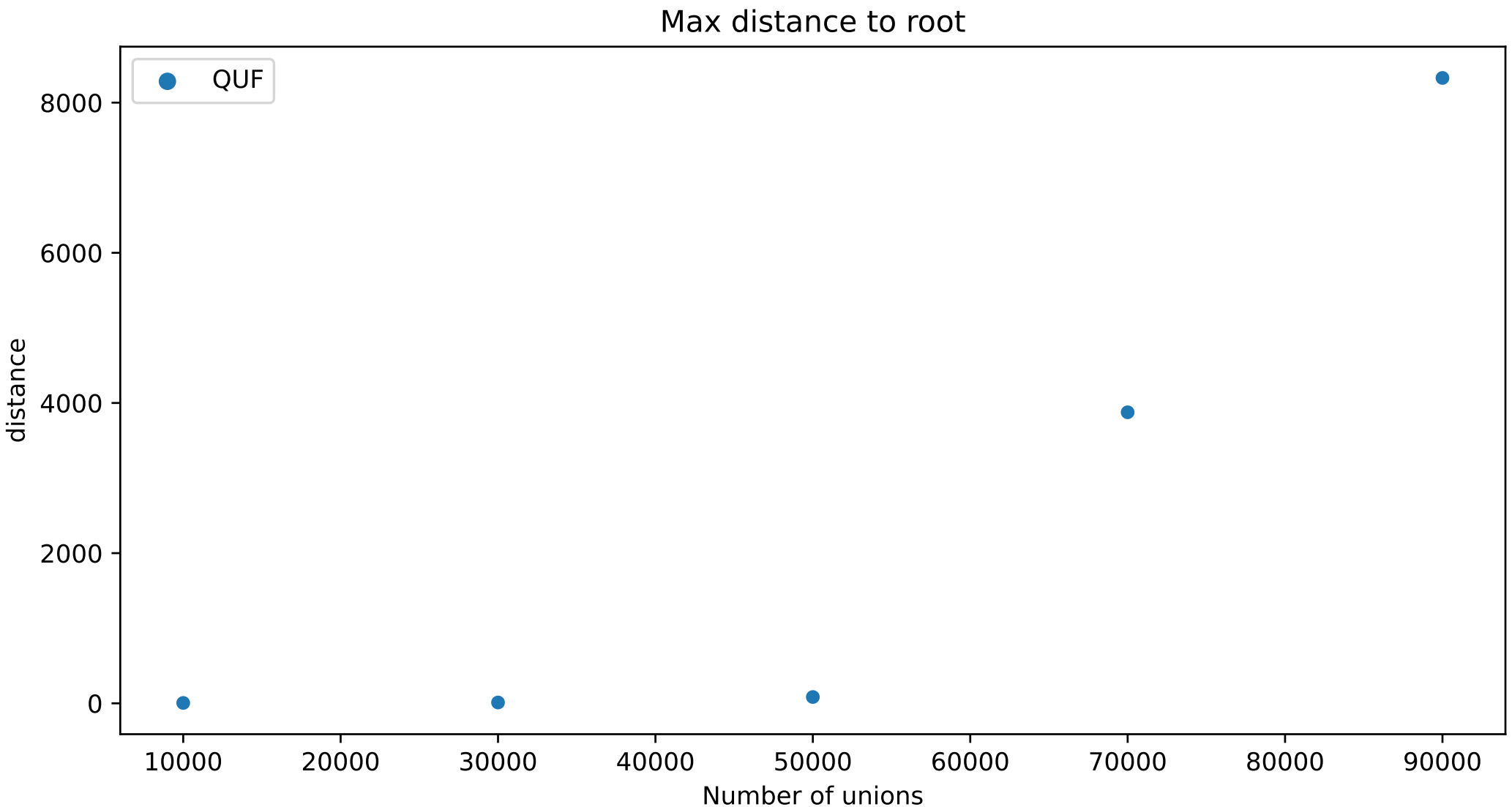
# Compared to the first attempt



# Looks good?


Performance with a maximum of 100,000 objects
QUF
time (seconds)
0
2
4
6
8
10
12
14
16
10000
20000
30000
40000
50000
60000
70000
80000
90000
Number of unions

# Why the quick increase in time?



# Can we do better?

```
 1  class WQUnionFind:
 2    def __init__(self, N:int) -> None:
 3      self.d = list(range(N))
 4      self.sz = [1] * N
 5
 6    def connected(self, a:int, b:int) -> bool:
 7      return self.root(a) == self.root(b)
 8
 9    def root(self, a:int) -> int:
10      while a != self.d[a]:
11        a = self.d[a]
12      return a
```

# Can we do better?

```
 1  @patch
 2  def union(self:WQUnionFind, a:int, b:int) -> None:
 3    ra = self.root(a)
 4    rb = self.root(b)
 5
 6    if self.sz[ra] < self.sz[rb]:
 7      self.d[ra] = rb
 8      self.sz[rb] += self.sz[ra]
 9    else:
10      self.d[rb] = ra
11      self.sz[ra] += self.sz[rb]
```

# Can we do better?

```
1 wquf = WQUnionFind(8)
2 print(wquf.sz, wquf.d)
3 wquf.union(0, 1)
4 print(wquf.sz, wquf.d)
5 wquf.union(1, 2)
6 print(wquf.sz, wquf.d)
7 wquf.union(3, 4)
8 print(wquf.sz, wquf.d)
```

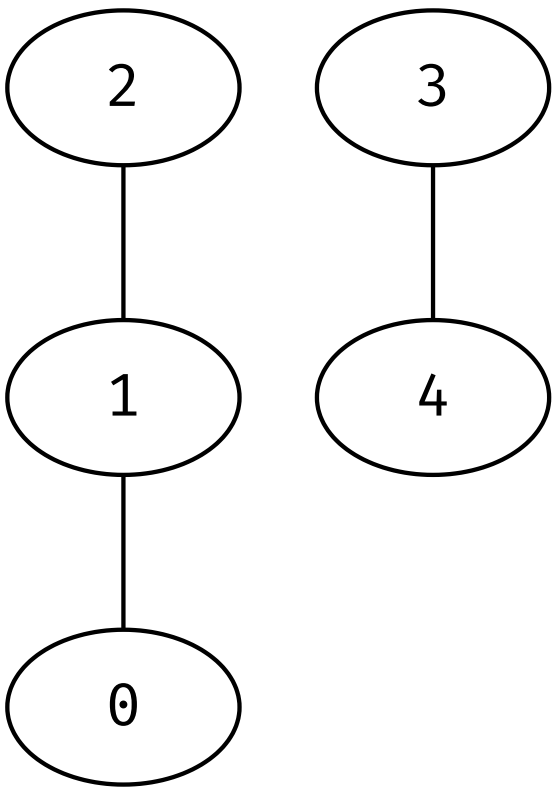
```
[1, 1, 1, 1, 1, 1, 1, 1] [0, 1, 2, 3, 4, 5, 6, 7]
[2, 1, 1, 1, 1, 1, 1, 1] [0, 0, 2, 3, 4, 5, 6, 7]
[3, 1, 1, 1, 1, 1, 1, 1] [0, 0, 0, 3, 4, 5, 6, 7]
[3, 1, 1, 2, 1, 1, 1, 1] [0, 0, 0, 3, 3, 5, 6, 7]
```

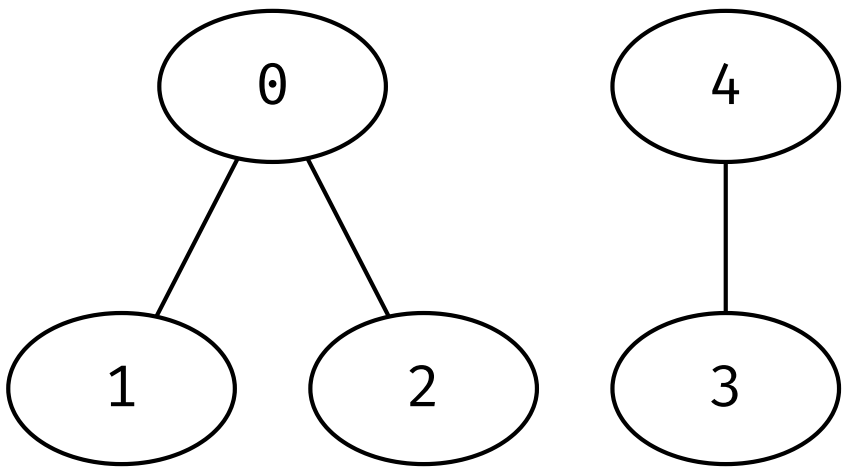
# Spot the difference

```
 1  quf = QUnionFind(8)
 2  quf.union(0, 1)
 3  quf.union(1, 2)
 4  quf.union(3, 4)
 5
 6  wquf = WQUnionFind(8)
 7  wquf.union(0, 1)
 8  wquf.union(1, 2)
 9  wquf.union(3, 4)
10
11  print(quf.d)
12  print(wquf.d)
```

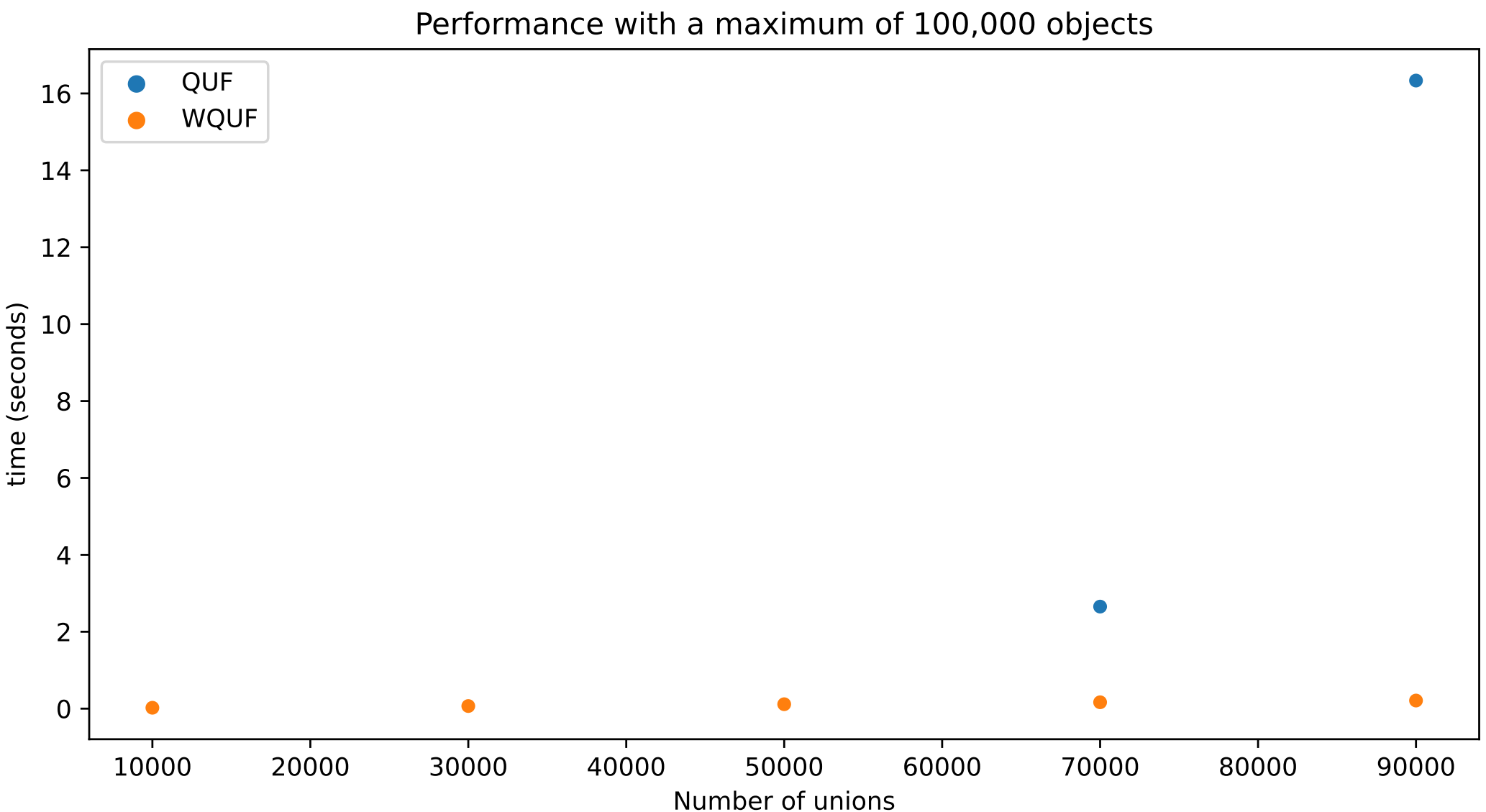
```
[1, 2, 2, 4, 4, 5, 6, 7]
[0, 0, 0, 3, 3, 5, 6, 7]
```

# Spot the difference

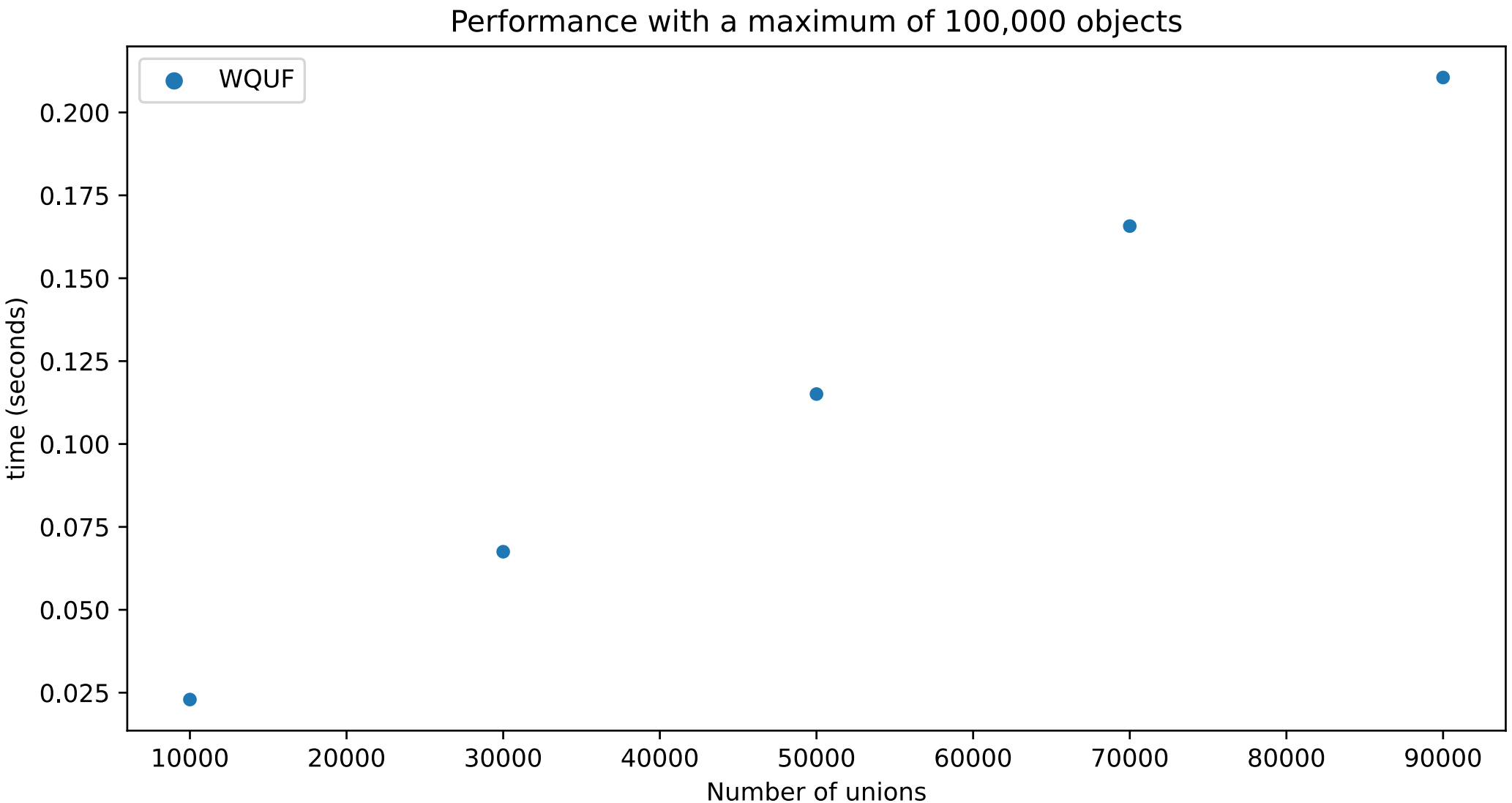


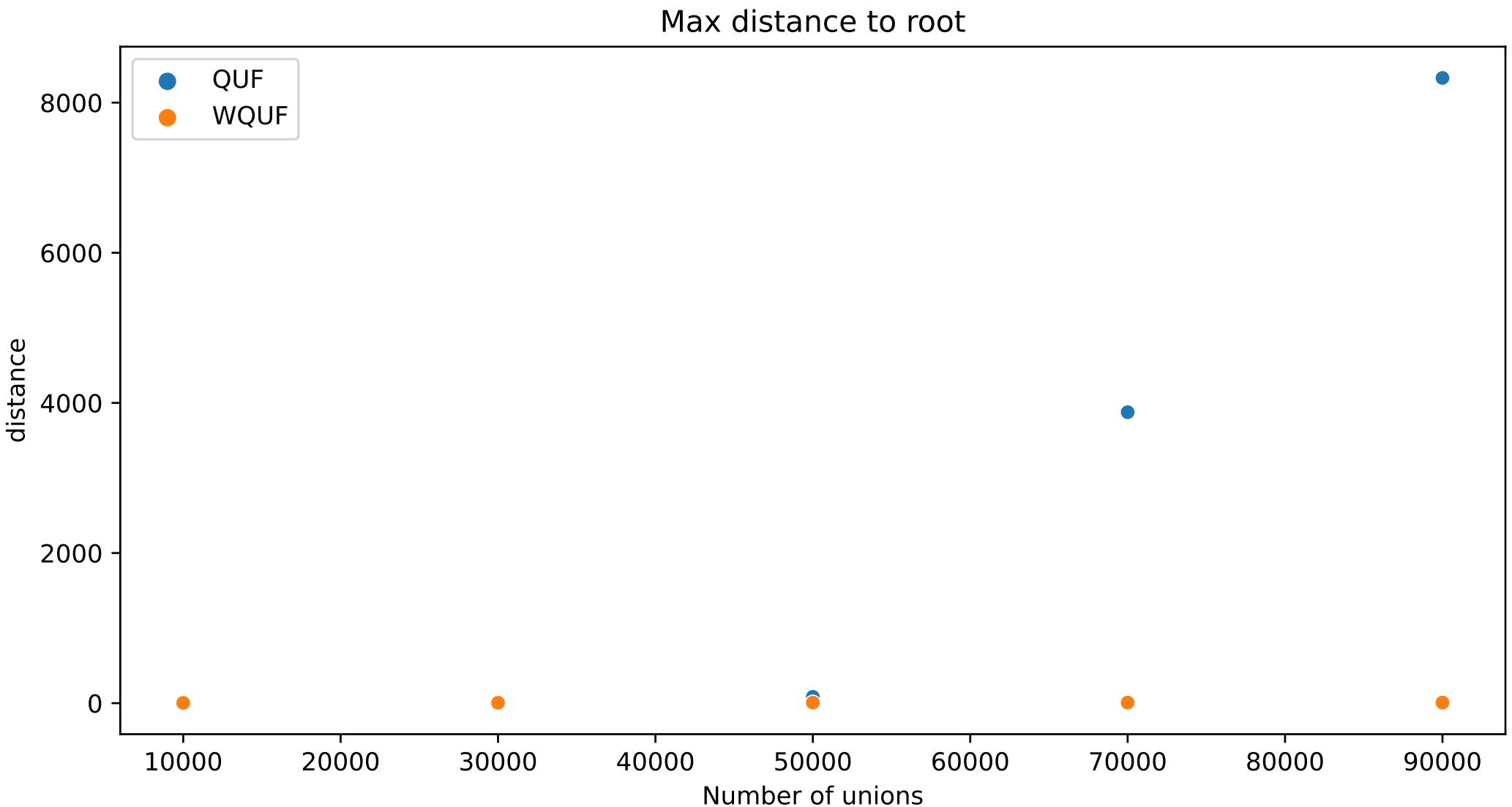
# Performance compared to previous



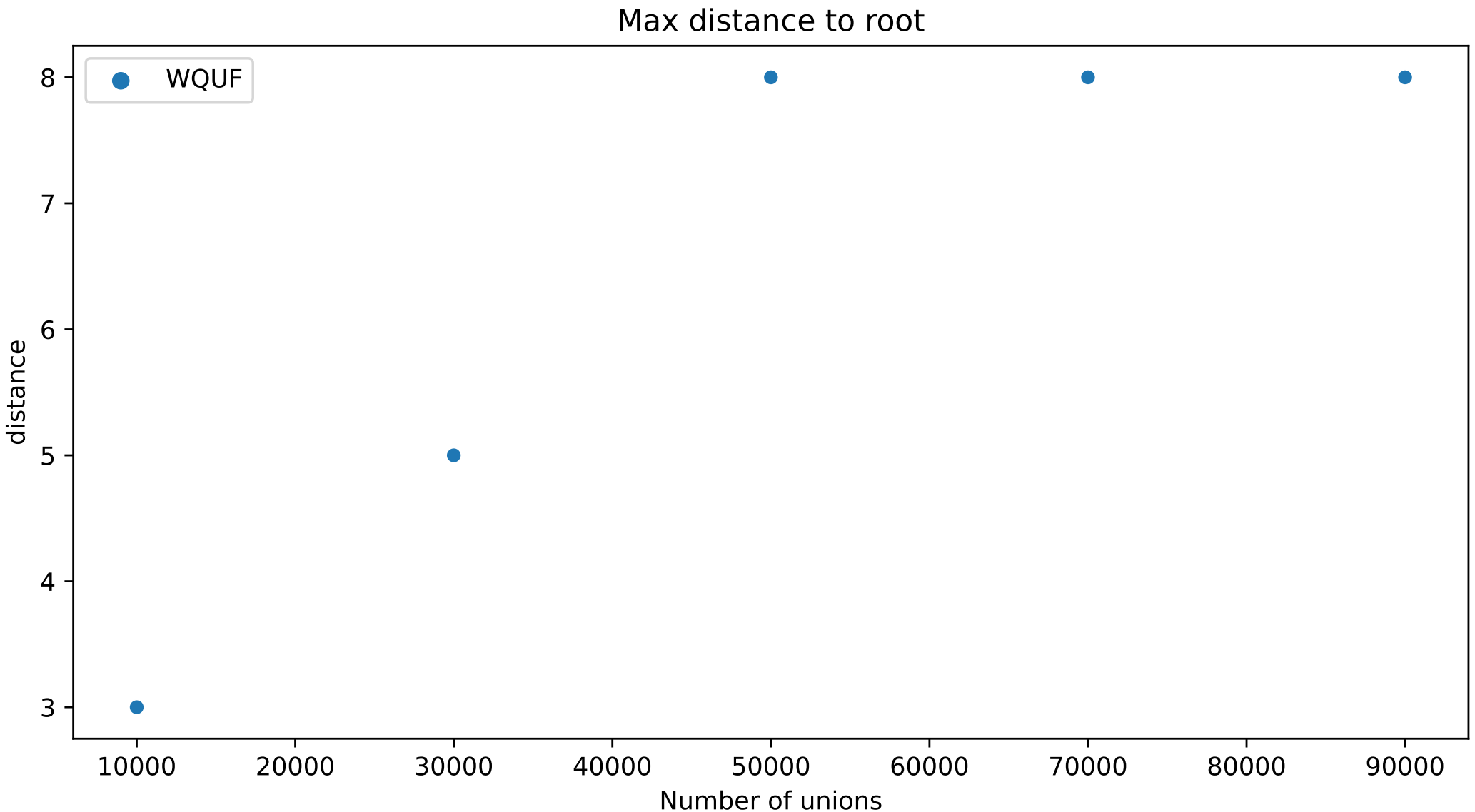
# Nicer growth



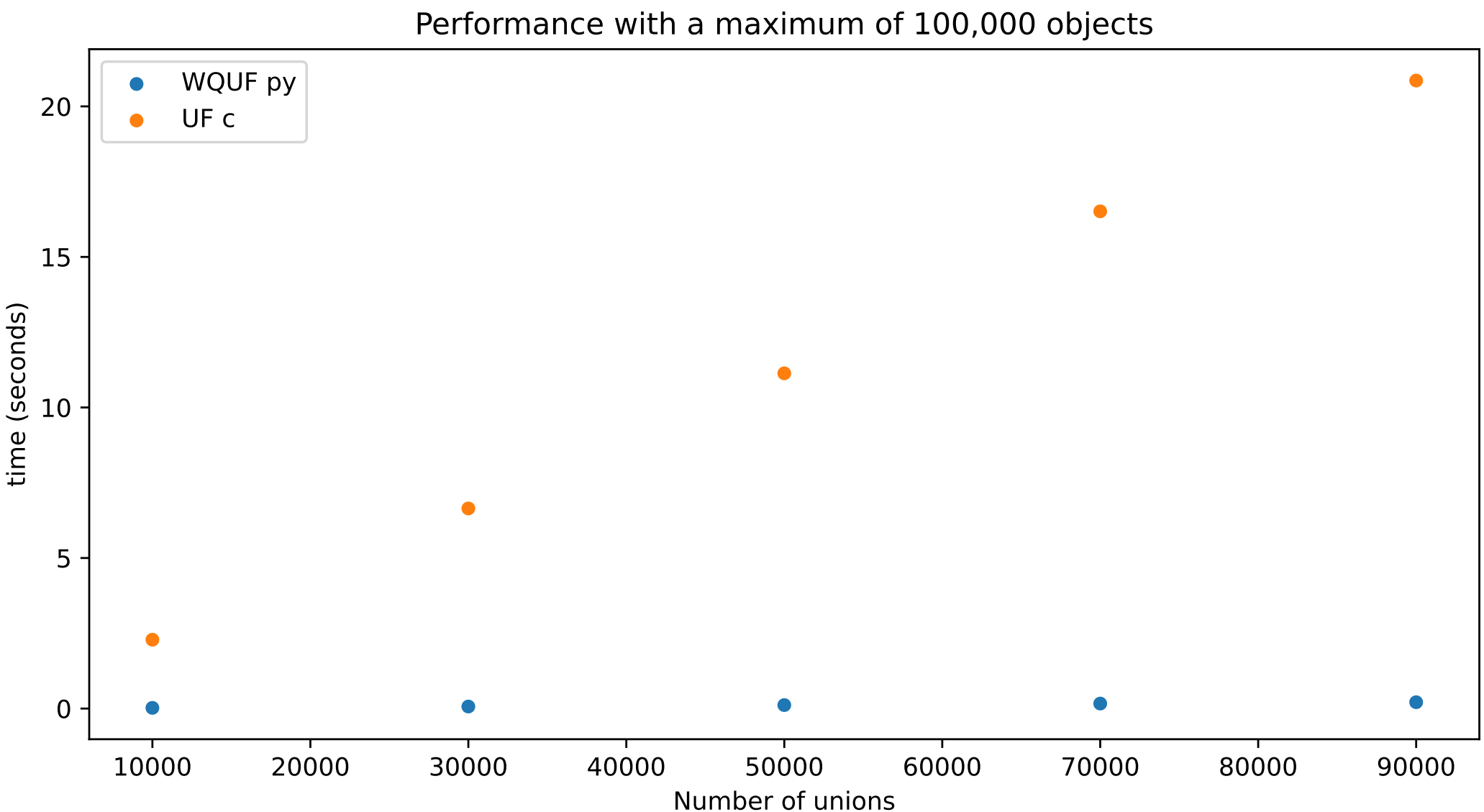
# Tree depth


Max distance to root
QUF
WQUF
8000
6000
4000
2000
0
distance
10000
20000
30000
40000
50000
60000
70000
80000
90000
Number of unions

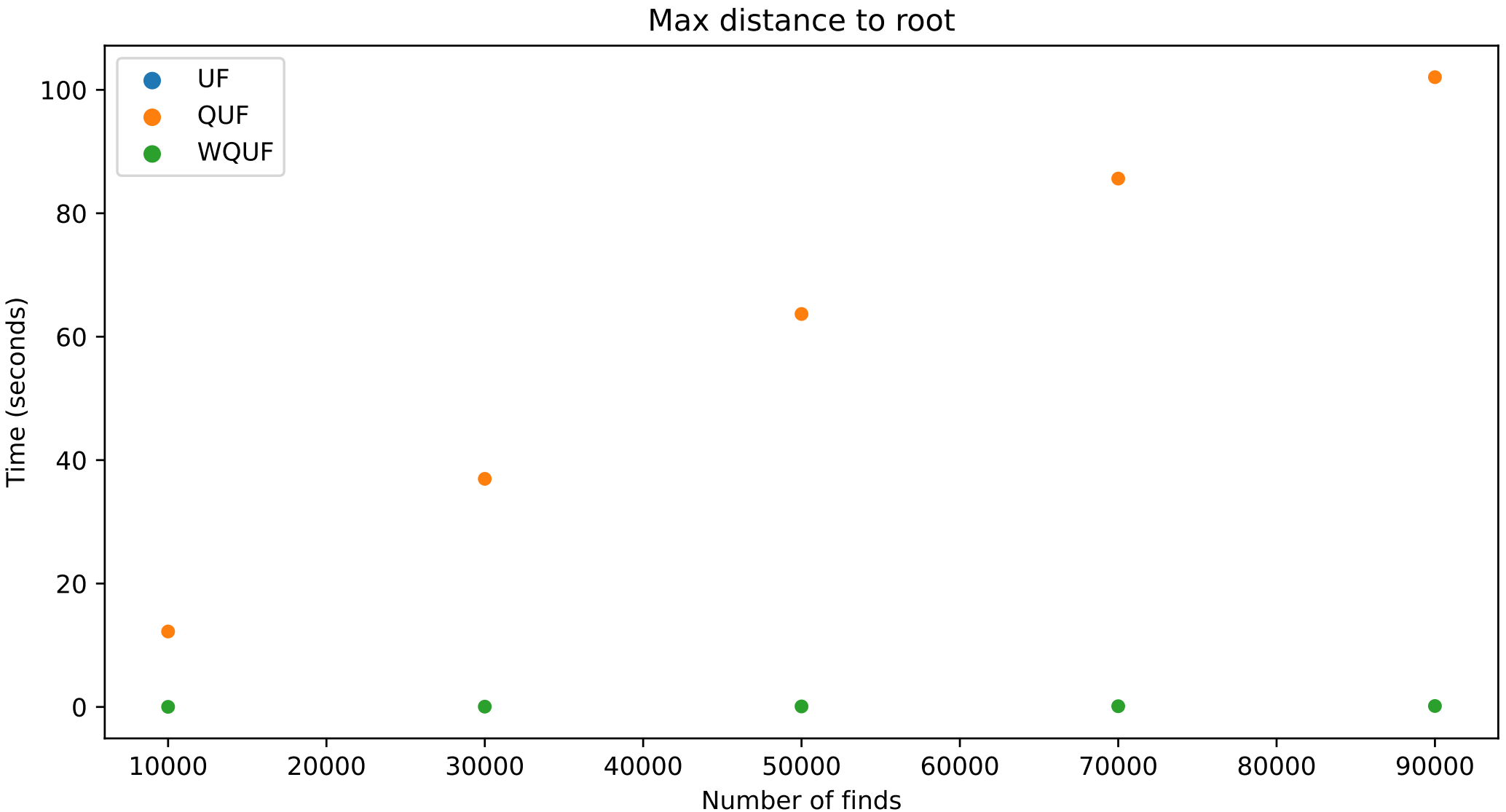
# Much shorter distances



# Python is slow?



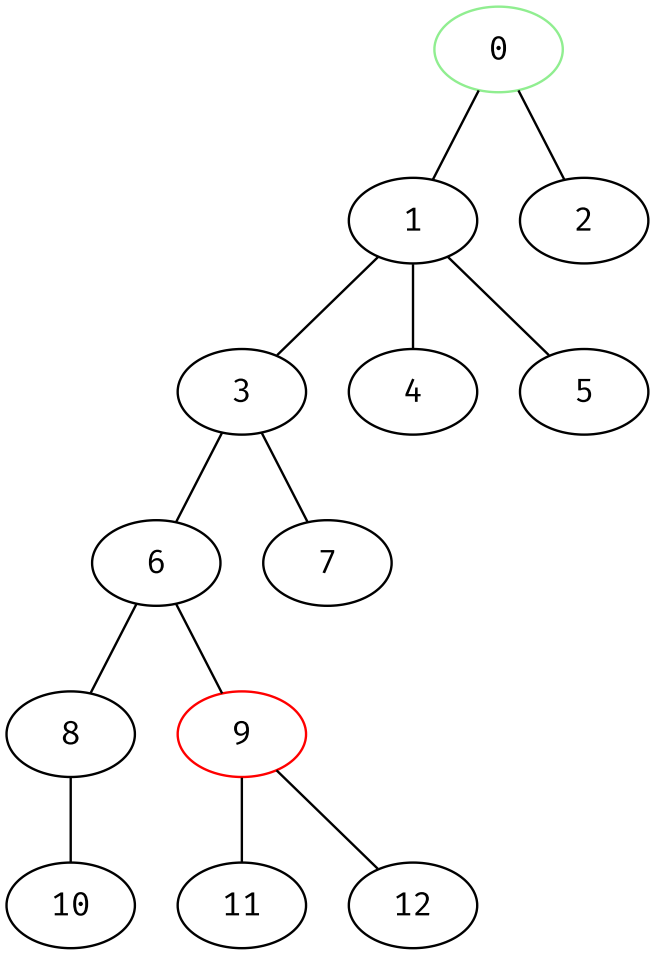
# Drawbacks?


Max distance to root
UF
QUF
WQUF
Time (seconds)
0
20
40
60
80
100
10000
20000
30000
40000
50000
60000
70000
80000
90000
Number of finds

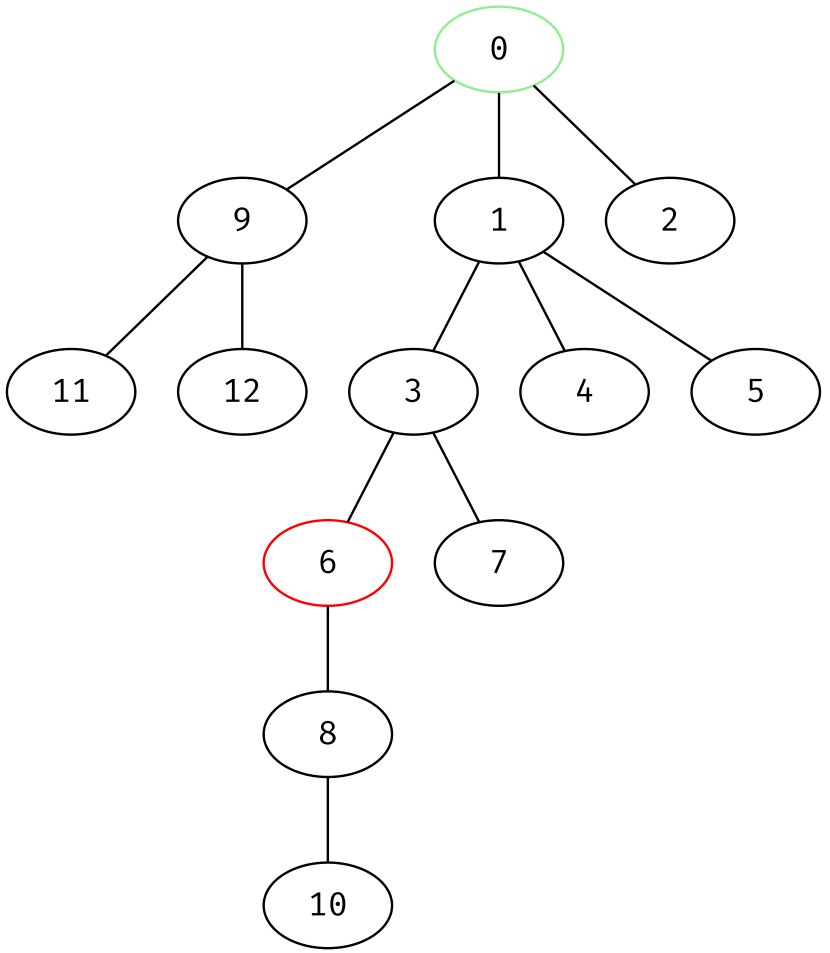
# Path compression

- The time to find the root depends on the height of the tree
- Merging to smaller helped
- What if we can make the trees even “flatter” (but wider)
- Idea: move subtrees when we are looking for the root?

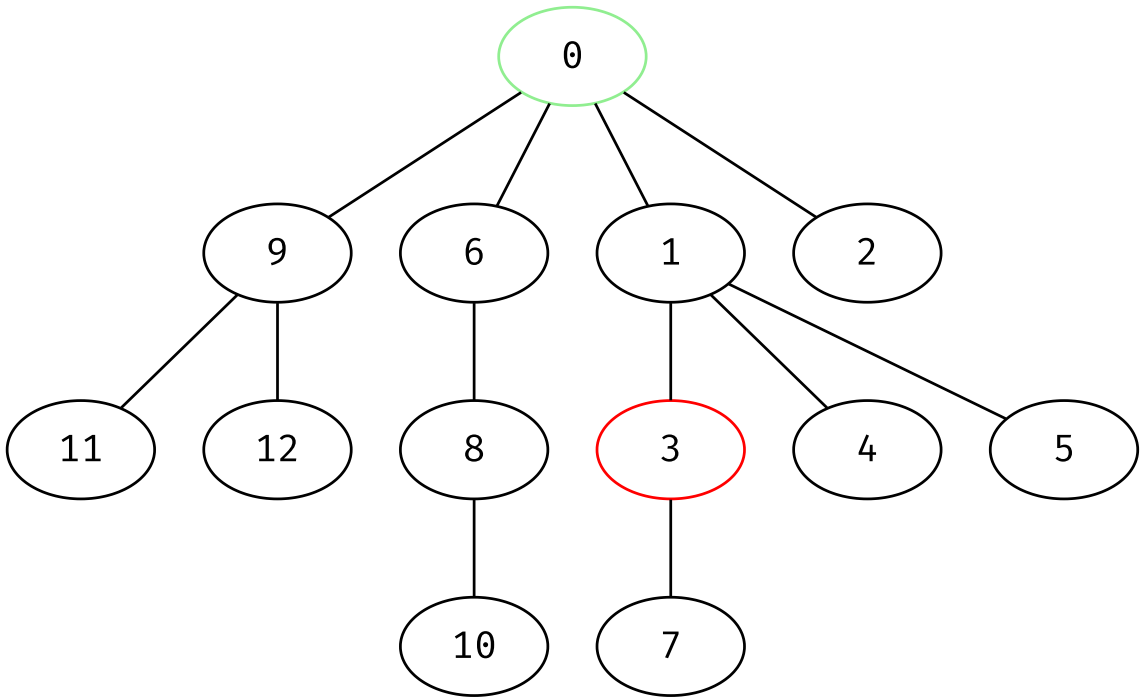
# Path compression



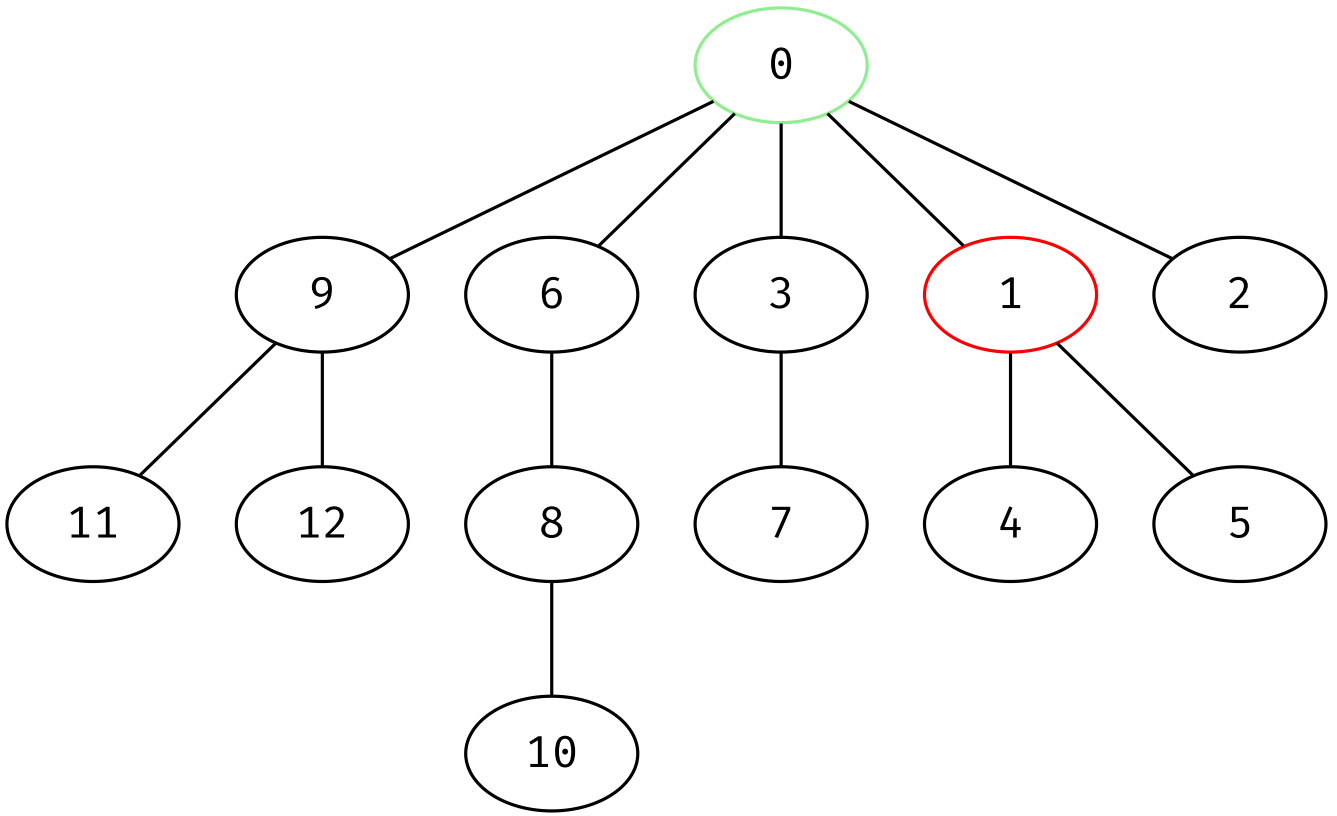
# Path compression



# Path compression



# Path compression



# Simple change

```
1  @patch
2  def root(self:WQUnionFind, a:int) -> int:
3    while a != self.d[a]:
4      self.d[a] = self.d[self.d[a]]
5      a = self.d[a]
6    return a
```

# Reading instructions

# Reading instructions

» Ch. 1

» Ch. 8.1 - 8.5, 8.7