Algorithms for Programming Contests - Week 02

Prof. Dr. Javier Esparza,
Philipp Czerner, Martin Helfrich, Christoph Welzel,
Mikhail Raskin,
conpra@in.tum.de

October 29, 2021

Theme of this Week

- Data structures + simple operations on them
- Basic algorithms

Graphs

- Known from practically all theoretical courses: $\mathcal{G} = (V, E)$.
- Numerous variants: Directed, weighted, labelled, ...
- Different underlying structures for different tasks!

Important properties / considerations

- Is the graph sparse or dense?
- Directed or undirected?
- Are node/edge labels/meta-information important?
- Transitive closure or predecessor relation relevant?
- Do we need the whole set of states?

Example approaches

Usually represented as some form of adjacency matrix:

- Dense, unweighted graph: `boolean[][]`
- Sparse, unweighted graph: `Set(int)[]`, `int -> Set(int)`
- Sparse digraph with labels: `Node -> Set(Edges)`
- Dense, weighted graph: `double[][]`
- ...

Other possibilities (no access to whole state set):

- `Node` object with set of successors
- Successor function (on-demand computation)
- ...

Summary

- Representation highly depends on the input format and task!
- Usually, adjacency lists are sufficient
  ...Adjacency hashtables are very nice
- Sometimes, having objects as graph nodes is really helpful
  might not happen in the course

Binary Search

- Widely known approach to search for specific items in $O(\log(n))$ time
- Simple example: Search in an ordered array

Idea

- Input: Sorted array `a` and object of interest `needle`
- Naive approach: Linearly scan through array – $O(n)$.
- Array sorted $\Rightarrow$ if `needle` $<$ `a[4]`, then `needle` $<$ `a[i]` for all $i \geq 4$!
- Idea: Check the middle element, compare it to `needle`:
  - Element is larger: Search on left side
  - Element is equal: Found it!
  - Element is smaller: Search on right side

Example

a:	3	5	7	12	13	20	21	40	50	90	100

needle: 12

Example

`needle`: 12

Example

Example

Example

Example

Summary

- Very efficient method to check whether an object is in an array
- Java has a dedicated function (`Arrays.binarySearch`)
- *Important:* Easily generalized to monotonous properties, e.g. searching for a bug with `git bisect`

Union Find

- Extremely fast approach to maintain a partition of a set
- Useful to model equivalence relations, e.g., graph partitions
- Theoretically interesting: Inverse Ackermann complexity[1] (when implemented with weighted union and path compression)

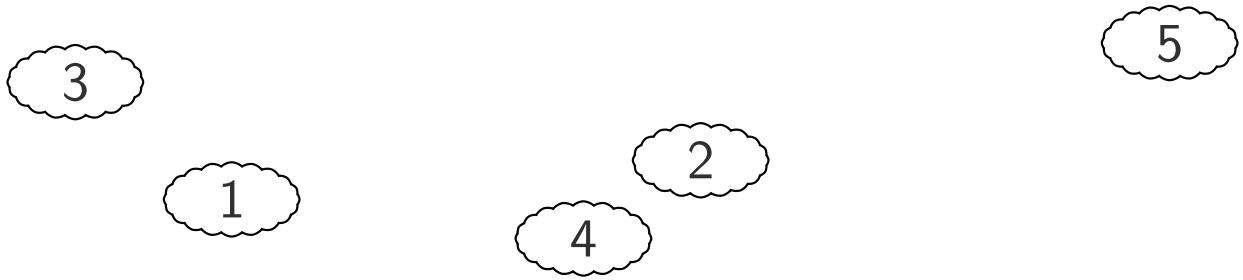
[1] very very close to constant: $\alpha(61) = 3$, $\alpha(2^{2^{2^{2^{16}}}}) \approx 4$

Features

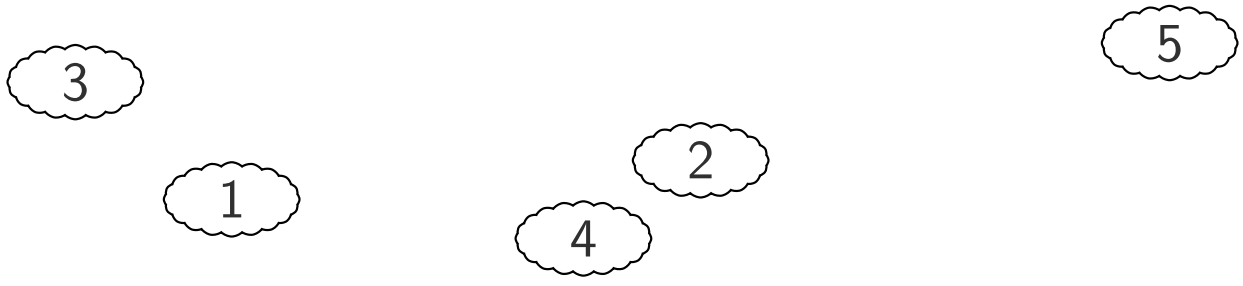
- Two core operations:
  - `union(a, b)`: Merge the sets of `a` and `b`
  - `find(a)`: Find the "root" of `a` (the representative of the set)

  ... and potentially `make`, adding a new element
- Idea: `find(a)` = `find(b)` iff `a` and `b` are in the same set
- Extensions: Size of each subset, amount of subsets, ...

Example

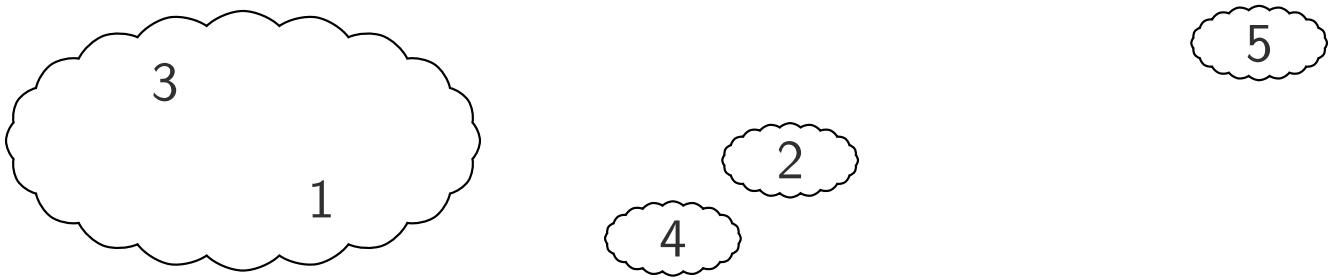


Example



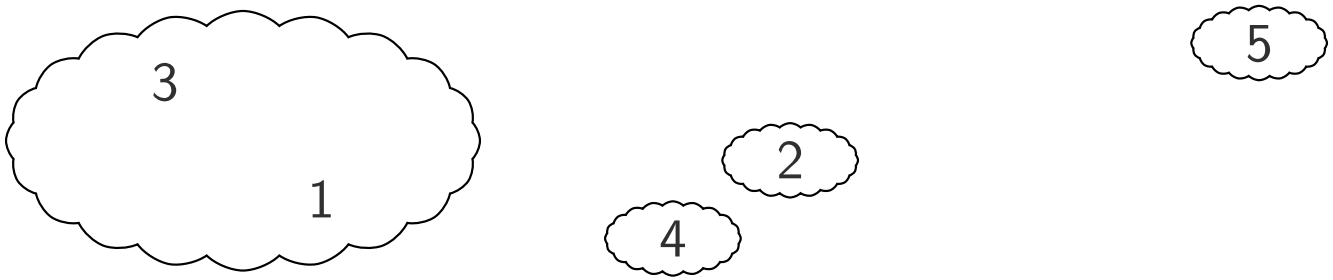
1. `union(1, 3)`

Example



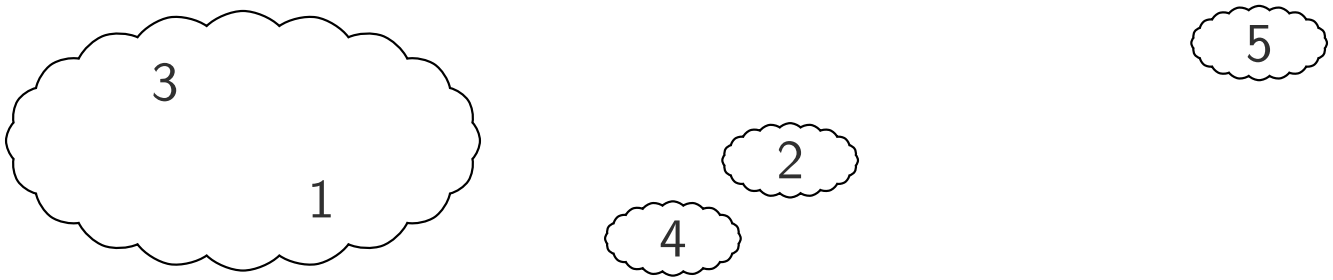
1. `union(1, 3)`

Example



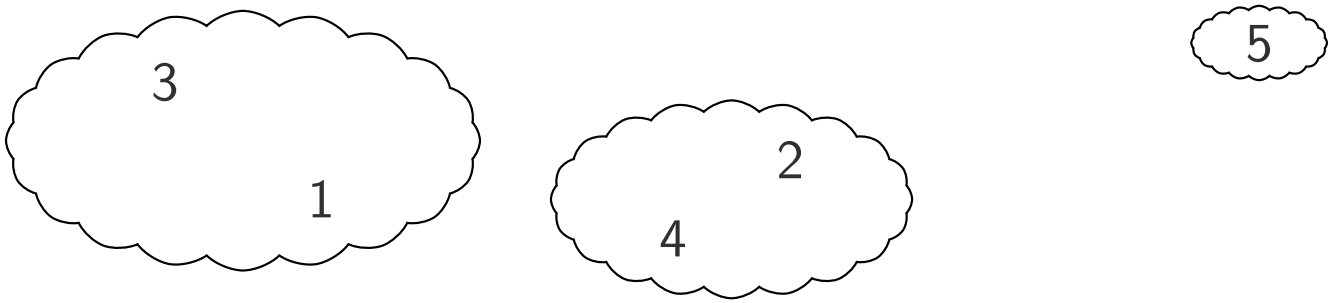
1. `union(1, 3)`
2. `find(1)` $\neq$ `find(4)`

Example



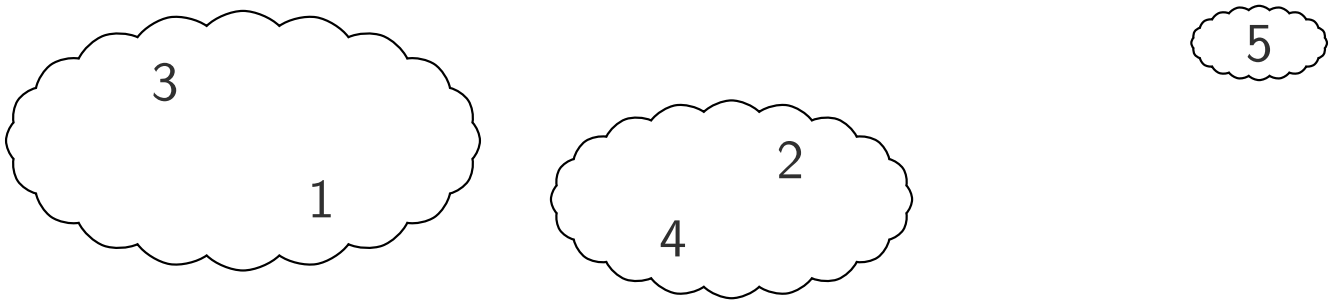
1. `union(1, 3)`
2. `find(1)` $\neq$ `find(4)`
3. `union(4, 2)`

Example



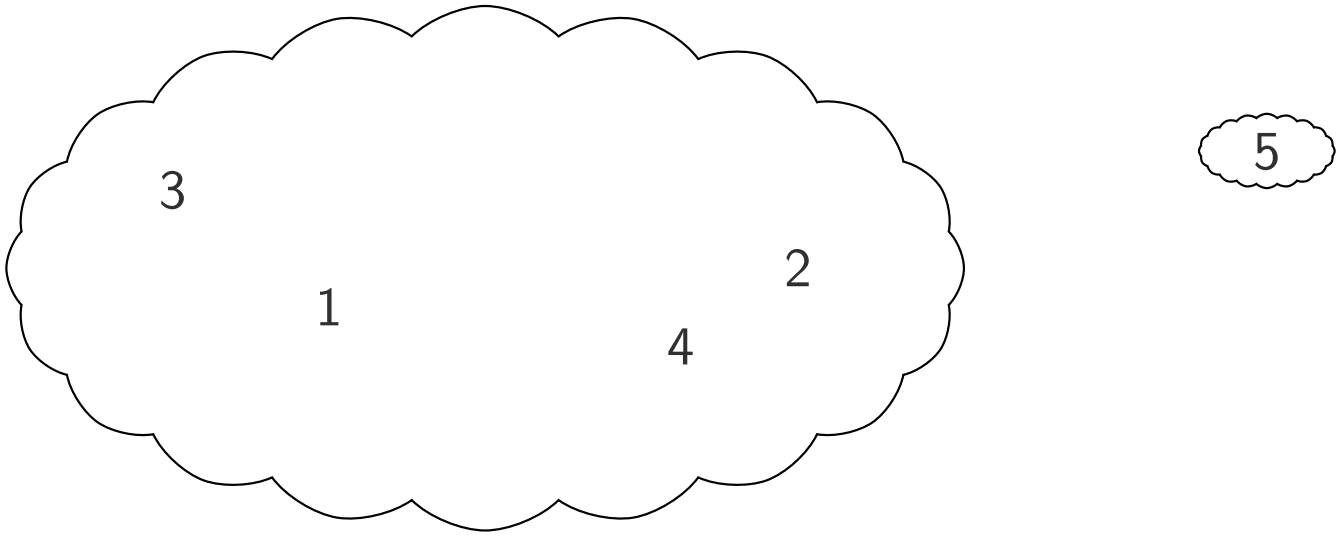
1. `union(1, 3)`
2. `find(1)` $\neq$ `find(4)`
3. `union(4, 2)`

Example



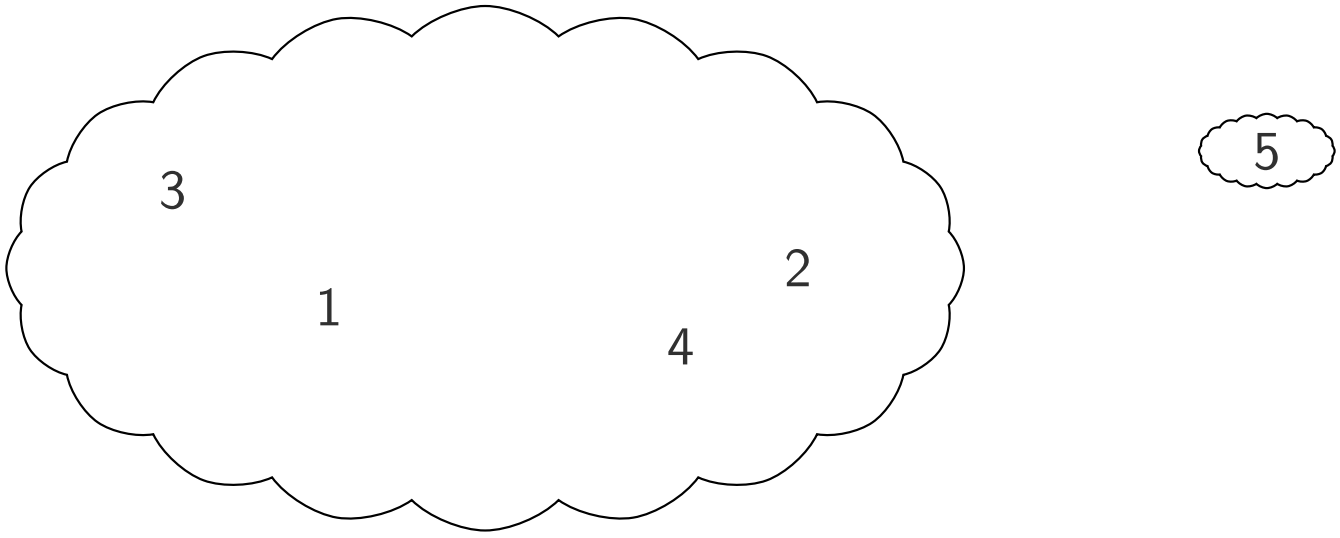
1. `union(1, 3)`
2. `find(1)` $\neq$ `find(4)`
3. `union(4, 2)`
4. `union(2, 1)`

Example



1. `union(1, 3)`
2. `find(1)` $\neq$ `find(4)`
3. `union(4, 2)`
4. `union(2, 1)`

Example



1. `union(1, 3)`
2. `find(1)` $\neq$ `find(4)`
3. `union(4, 2)`
4. `union(2, 1)`
5. `find(1)` $=$ `find(4)`

Implementation overview

- Here: Base set integers from `0` to `n`
- Underlying structures:
  - `parent : int[n]`: Parent of an element
  - `size : int[n]`: Size of the set (needed for weighted union)
- Initialize: `parent[i] = i`, `size[i] = 1`

Implementation overview

- Here: Base set integers from `0` to `n`
- Underlying structures:
  - `parent : int[n]`: Parent of an element
  - `size : int[n]`: Size of the set (needed for weighted union)
- Initialize: `parent[i] = i`, `size[i] = 1`
- `find(a)`:
  - Follow `parent[i]` until element is its own parent ($\hat{=}$ it's the root)
  - Path compression: Set `parent[i] = root` for all `i` along the way.

Implementation overview

- Here: Base set integers from `0` to `n`
- Underlying structures:
  - `parent : int[n]`: Parent of an element
  - `size : int[n]`: Size of the set (needed for weighted union)
- Initialize: `parent[i] = i`, `size[i] = 1`
- `find(a)`:
  - Follow `parent[i]` until element is its own parent ($\hat{=}$ it's the root)
  - Path compression: Set `parent[i] = root` for all `i` along the way.
- `union(a, b)`:
  - Find roots of `a` and `b`
  - Set the larger (according to `size`) of the two as the parent of the smaller one (weighted union)
  - Update `size` accordingly

Implementation - find(a)

```
# Find root
root = a
while True:
    parent = parent[root]
    if parent == root:
        break
    root = parent

# Compress path
current = a
while current != root:
    next_elem = parent[current]
    parent[current] = root
    current = next_elem

return root
```

Implementation - union(a, b)

```
a, b = find(a), find(b)
if a == b: # a and b are already merged
    return a

# Weighted Union: Update smaller component
a_size, b_size = size[a], size[b]
if a_size < b_size:
     a, b = b, a

parent[b] = a
# Update size accordingly
size[a] = a_size + b_size
```

Summary

- Practically constant time data structure for (merging) disjoint sets
- Simple to understand and implement