Algorithms for Programming Contests - Week 02

Prof. Dr. Javier Esparza,
Vincent Fischer,
Jakob Schulz,
conpra@model.cit.tum.de

October 21, 2025

Contents of this Week

- Data structures: general considerations
- Binary Search
- Union Find

Data structures: Graphs as example

- Known from practically all theoretical courses: $\mathcal{G} = (V, E)$.
- Numerous variants: Directed, weighted, labelled, ...
- Implement using different data structures depending on task!

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/list:

- Dense, unweighted graph: `bool[][]`
- Sparse, unweighted graph: `set<int>[]`, `int -> set<int>`
- Sparse digraph with labels: `Node -> set<Edge>`
- 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
- 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

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

needle: 12

Example

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

`needle`: 12

Example

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

`needle`: 12

Example

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

`needle`: 12

Example

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

`needle`: 12

Adaptions

- So far: given `x` and a sorted array `a`, return whether `x` is in `a`
- Algorithm can easily be adapted to return
  - (first) index `i` where `a[i] == x`, if such an index exists
  - index `i` where `x` would need to be inserted to maintain the order of the array, otherwise (just return the last pivot index)

Generalizations

- Simple, but powerful generalization: instead of finding the first index `i` where `a[i] == x`, find the first index `i` where `f(a[i]) == x`, where `f` is a monotonically increasing function (actually, only needed that `i` $\mapsto$ `f(a[i])` is monotonically increasing)
  - e.g. `git bisect` to find a bug
- Yet another generalization: instead of searching in an explicitly stored array `a`, search in some interval $[b, c]$
  - e.g. to find the square root of an `f64`
  - return once current search interval is small enough (or result is accurate enough)

Running Time

- Search interval halved every time $\rightarrow$ at most $\lceil \log_2 n \rceil + 1$ iterations, where n is the size of the search interval
- For `f64`, this means at most 65 iterations (and one bit of precision per iteration!)

$\Rightarrow$ fast for such a generic algorithm!

Summary

- Very efficient method to check whether an object is in an array
- Implementations in all major programming languages
  - E.g. C++: `std::binary_search` for `std::vector`
  - E.g. Rust: `binary_search` for `Vec`
- Easily generalized to search in an interval

Union Find

- Extremely fast approach to maintain a partition of a set
  - Given a set X, a *partition* of X is a set of pairwise disjoint, non-empty subsets of X, whose union is X
  - Partitions can be used to represent equivalence relations, e.g. on nodes of a graph
- Theoretically interesting: Amortized running time described by Inverse Ackermann function α[1] (when implemented with weighted union and path compression)

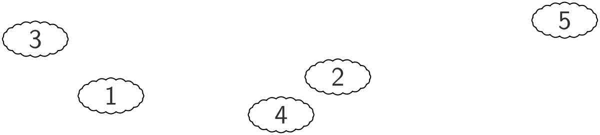
[1] very very close to constant: $\alpha(61) = 3$, $\alpha(2^{2^{2^{2^{2^{16}}}}}) \approx 4$

Features

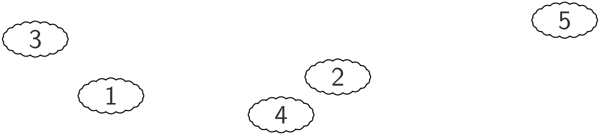
- Two core operations:
  - `union(a, b)`: Merge the sets of `a` and `b`
  - `find(a)`: Find the "root" of `a` (the representative of the set)

  ... and potentially `make`, adding a new element
- Idea: `find(a)` $=$ `find(b)` iff `a` and `b` are in the same set
- Extension: Size (or other property that behaves well under union) of each class, number of classes, ...

Example

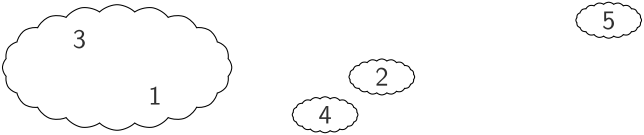


Example



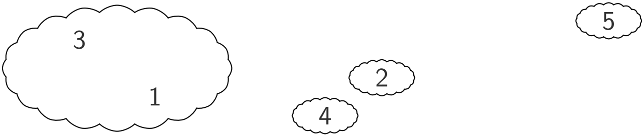
1. `union(1, 3)`

Example



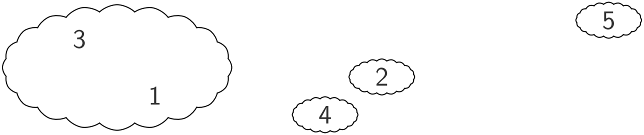
1. `union(1, 3)`

Example



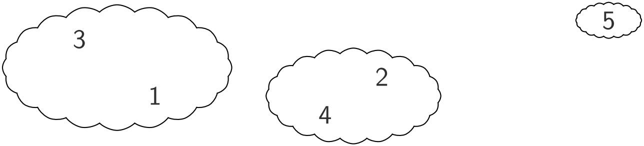
1. `union(1, 3)`
2. `find(1)` $\neq$ `find(4)`

Example



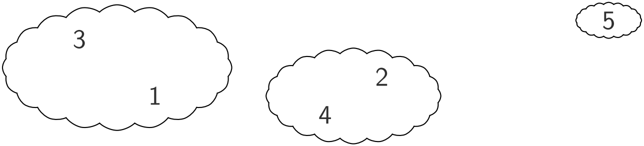
1. `union(1, 3)`
2. `find(1)` $\neq$ `find(4)`
3. `union(4, 2)`

Example



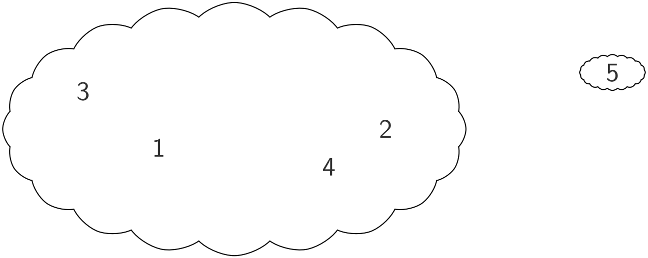
1. `union(1, 3)`
2. `find(1)` $\neq$ `find(4)`
3. `union(4, 2)`

Example



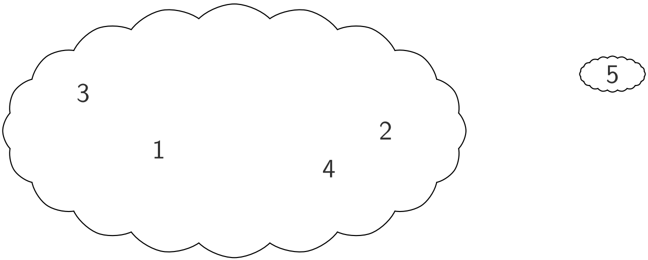
1. `union(1, 3)`
2. `find(1)` $\neq$ `find(4)`
3. `union(4, 2)`
4. `union(2, 1)`

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