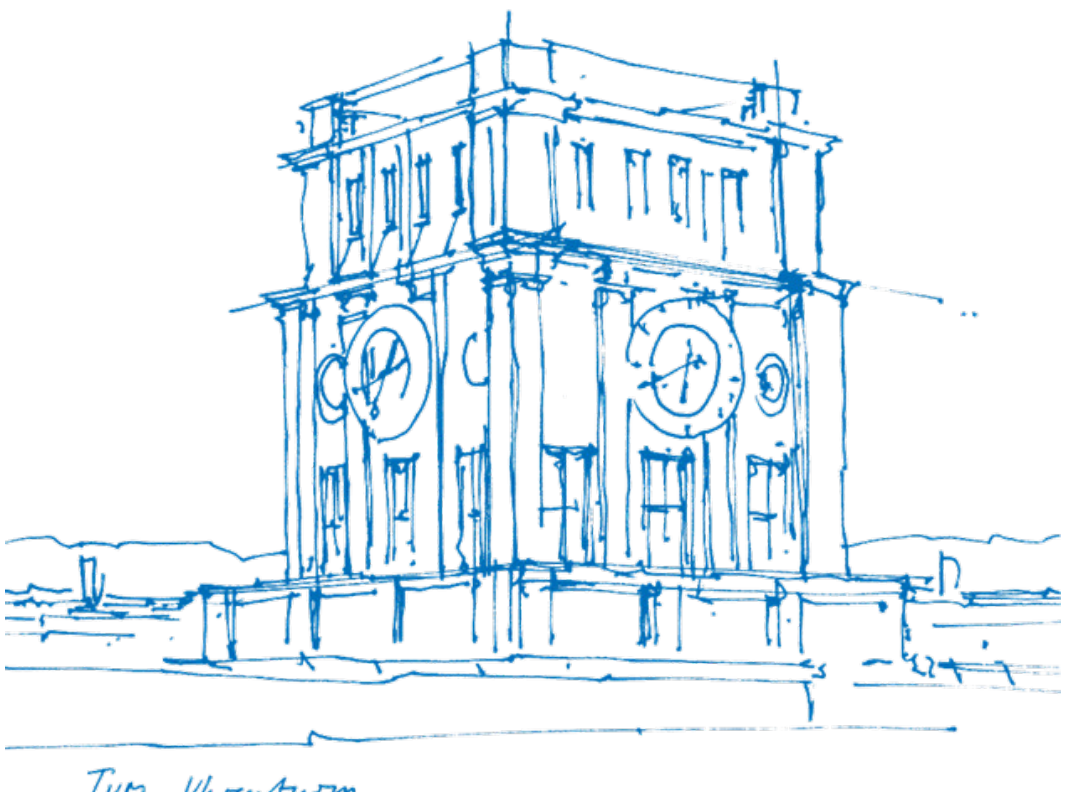
School of Computation, Information and Technology
Technical University of Munich

AACPP WiSe 2025/26

Class 3: Graphs



Mateusz Gienieczko

School of Computation, Information and Technology
Technical University of Munich

2025.11.18

Assignment 2 Rewind

Assignment 2 proved too difficult

- Getting few points with suboptimal solutions was not very feasible.

As this year started with two very hard assignments, and low scoring, here is what we can do:

- Give base 9 points to everyone and present solutions.
- Extend the assignment for another week (?) and give more hints.

Third problem

Third deadline – 03.12.2025, 12:00 PM.

CCC – Cat Care Coordination

We have $n \leq 18$ people and an interval $[0, L)$ ($L \leq 100\,000$). For each unit interval $[i, i+1)$ we know who is **busy** and who is not. We need to choose a T and for each person an interval of length T for **sleeping**. In each moment in $[0, L)$ someone has to be free (not **busy** and not **sleeping**). We want to maximise T.

CCC – existence

Since T can be zero, the answer is negative if and only if there already exists an interval where everyone is busy – easy to check.

If there is an interval where only one person is not busy then they have to care for Dexter and cannot sleep.

So we are left with a schedule where at every point at least two people are free.

CCC – result constraints

The result can be an arbitrary rational number.

But we can constrain the denominator of the simplified fraction by n.

The observation is that in an optimal solution we need to tightly pack the sleep intervals, i.e. there cannot be a situation like:

CCC – Result binsearch

With a finite solution space we can binary search over T between 0 and L.

Here, n is so small that we can just go through all possible denominators from 1 to n and binsearch the numerator.

In the future, we will assume the denominator is fixed to some q and we are dealing with $[0, L)$ divided into $\frac{L}{q}$ intervals of length $\frac{1}{q}$.

Farey Sequences

For an integer N, the Farey sequence of order N is: all reduced fractions a/b with $0 \le a \le b \le N$, gcd(a,b)=1, sorted by size.

Example for N=4:

$F4$={0/1, 1/4, 1/3, 1/2, 2/3, 3/4, 1/1}

Those are exactly the possible "**simple rational**" values with denominator $\le N$.

Stern–Brocot tree (same fractions, but as a search tree)

The **Stern–Brocot** tree is an infinite binary tree that contains every positive rational number exactly once, in reduced form:

- Start with two “sentinel” fractions: left L=0/1 and right R=1/0 (“infinity”).
- The mediant of L=a/b and R=c/d is $M=(a+c)/(b+d)$
- Insert M between L and R.
- Recurse on [L,M] and [M,R].
- If you do an inorder traversal of this tree, you see fractions in increasing order.

So: Farey sequence sorted list ↔ Stern–Brocot tree in-order traversal.

CCC – Aside on mediants

In general, for binsearching fractions one can use **Farey sequences**.[1] A *mediant* of $\frac{a}{b}$ and $\frac{c}{d}$ is defined as:

$$\frac{a}{b} \oplus \frac{c}{d} := \frac{a+c}{b+d}$$

It is obvious that if $\frac{a}{b} \leq \frac{c}{d}$ then $\frac{a}{b} \leq \frac{a}{b} \oplus \frac{c}{d} \leq \frac{c}{d}$. A harder, but crucial, property, is that if $bc - ad = 1$ then $\frac{a}{b} \oplus \frac{c}{d}$ is the fraction with the smallest denominator in the range $\left(\frac{a}{b}, \frac{c}{d}\right)$.

[1] We had a very interesting task PHR for mediants and Stern-Brocot tree. The task can still be submitted in the Suse 2025 contest and the solution is available here.

CCC – Aside on mediants

$$\frac{a}{b} \oplus \frac{c}{d} := \frac{a+c}{b+d}$$

If $bc - ad = 1$ then $\frac{a}{b} \oplus \frac{c}{d}$ is the fraction with the smallest denominator in $\left(\frac{a}{b}, \frac{c}{d}\right)$.

We can binsearch by choosing the mediant and not allowing the denominator to exceed n.

The precondition is maintained:

$$\begin{aligned} b(a+c) - a(b+d) &= ba + bc - ba - ad \\ &= bc - ad \\ &= 1 \end{aligned}$$

CCC – Finding sleep

We have $T = \frac{p}{q}$ and want to check if everyone can get a valid interval of sleep of length T. First, in $\mathcal{O}(nqL)$, we find all intervals in which a given person *could* sleep. They can be longer than T, but shorter than T are skipped.

Brute-force: fix some order of people sleeping, try to assign intervals with a backtrack. Time complexity hard to analyse, but it gets 2 points.

CCC – algorithm sketch

The idea in the model solution is to greedily assign to each person the first possible moment to sleep. In case of a conflict we will backtrack and try to fix it.

The number of decisions will be potentially exponential, but n is small.

Brute force

```
def assign(i):
    for every slot [a, a+T) where caretaker i is free:
        update(a, a+T, -1)        // they sleep → fewer awake
        if segment_tree.min > 0:  // cat never alone
            if i is the last caretaker → return true
            else if assign(next caretaker): return true
        update(a, a+T, +1)        // undo if failed
    return false
```

If assign(0) ever returns true, the given sleep length T is feasible.

BUT ⟹ There are n caretakers ⟹ potentially **$O(L^n)$ branches.**

What we can then do

Maintain **cnt[j]** = number of caretakers available at hour j.

Precompute **nextGood[i][j]** — for caretaker i, the next possible starting hour ≥ j where that caretaker can sleep uninterrupted for sleep hours.

Caretakers already assigned earlier have been assigned a **permanent interval.**
Caretakers that remain to be assigned have a **tentative interval** (nextGood), which may change depending on other assignments and availability.

Maintain in each recursion:

- mask = bitmask of which caretakers already have a sleep assigned.
- from = starting hour to search from.
- tree = tracks availability, receives updates and tracks permanent intervals
- nextGood = tentative intervals (may change), next possible starting hours for caretakers

CCC – algorithm sketch

Each person has either a *tentative* interval that may change, or a *permanent* interval that we fix and it will never move again.

A *conflict* occurs when during a given person's sleep interval there is a moment where no one takes care of Dexter, i.e. everyone is busy or asleep.

Take the person x with earliest tentative interval. If in their interval there is no conflict then there will never be a conflict there, so we make it permanent.

If there is one, we need to resolve it.

CCC – resolving conflicts

Assume there is a conflict at $\left[\frac{t}{q}, \frac{t+1}{q}\right)$. There are at least two people in the conflict, including x. Assume there's k conflicting sleep intervals.

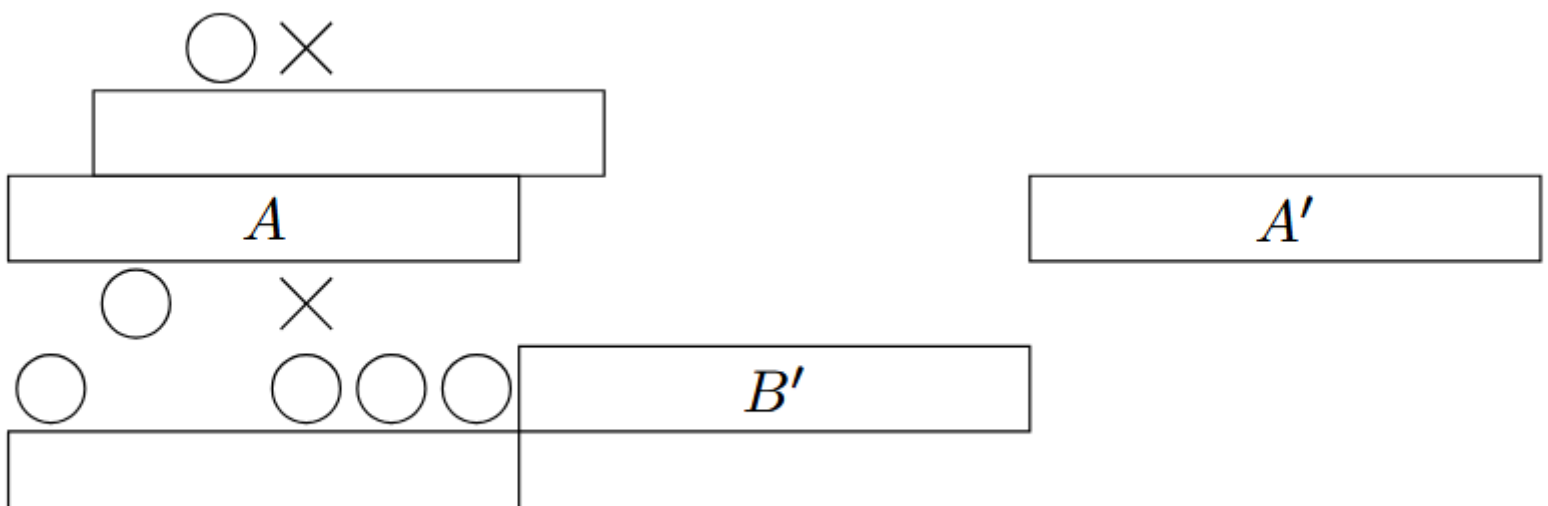
Main observation: we can fix $k-1$ of those intervals.

First, if in some correct solution at $\frac{t}{q}$ we have $k-1$ people sleeping, then we can shift them maximally "to the left" and get our tentatives. So fix those.

Otherwise, two of the tentative intervals conflict and both must be moved outside of $\frac{t}{q}$. Call them $A = [a, a+T)$, $B = [b, b+T)$ for $a \leq b$.

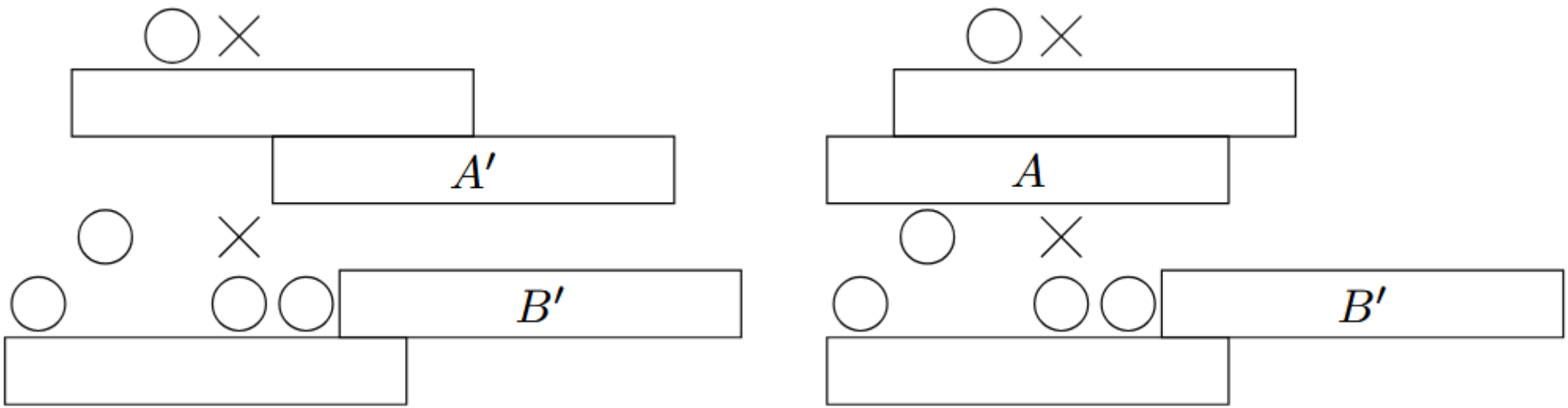
CCC – observation proof

Call the intervals in some correct solution $A' = [a', a' + T), B' = [b', b' + T)$.



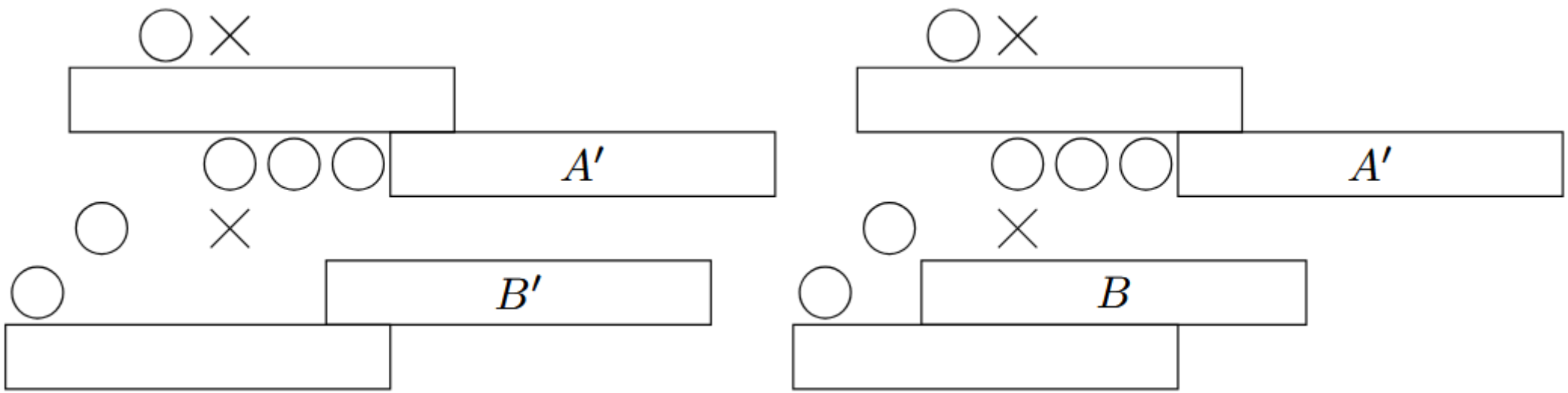
Case: $b' \geq a + T$ – exchange A' for A

CCC – observation proof



Case: $a' \leq b' < a + T$ – exchange A' for A

CCC – observation proof



Case: $b' < a', a + T$ – exchange B' for B

CCC – algorithm

On a conflict, branch into k possibilities (which interval is *not* fixed).

In each branch the number of tentative intervals decreases by $k - 1$.

Need to find the next tentative interval for the chosen person. We can find the last moment where only one person is not busy/sleeping (according to our permanent choices).

We must check at most three intervals (it starts in $T + \frac{t+1}{q}$ and must not collide), so $\mathcal{O}(1)$.

Fixing the next permanent schedule (no conflict)

We learned: If at some time t there are k tentative sleep intervals overlapping and causing a conflict (no one awake), then in some correct schedule we can keep k–1 of those intervals exactly where they are and only move one of them to the right.

What this means for our DFS recursion

- At each recursion step we place the “next” caretaker’s sleep interval greedily (as far left as possible under current choices) and then:
- If the new interval creates no conflict
  - It will never need to move: if there exists any valid global schedule, we can always transform it so that this caretaker sleeps exactly where we just put them (shift them left until they hit our position).
  - So we can safely mark this interval as permanent and recurse.

Fixing the next permanent schedule (conflict)

If the new interval creates a conflict

- Suppose k tentative intervals overlap there (including the one we just placed). By the lemma, in some correct solution: k–1 of these intervals can stay exactly where they are now (become permanent),
- only 1 of them needs to be shifted further right (stays tentative).
- So in the recursion we only need to explore branches that differ in which interval is the "special" one that's allowed to move:
- Branch A: assume the newly placed interval is the one that will move later (keep it tentative; the other k–1 can be fixed);
- Branch B / C / …: assume some other interval in the conflict is the one that moves, and our new interval is actually permanent.

Therefore, each recursive step explores at most **2 promising branches**.

- If no solution is found by trying to fix the next two tentative intervals, then we can safely backtrack.

CCC – analysis

Number of branches is described by:

$$T(n) = \max_k(k \cdot T(n-k))$$

Worst case is $k = 2$, $T(n) = \mathcal{O}(2^n)$.

For managing intervals we need a data structure.

We can maintain two segment trees over $[0, Lm)$: one for permanent, one for tentative intervals. We keep the number of people available for caretaking.

Conflict is "$\min = 0$", finding the next interval on a branch is finding the last 1 (ignoring the skipped intervals). Each operation takes $\mathcal{O}(\log(Lq))$.

CCC – analysis

Overall we have $\mathcal{O}(n \log(Ln))$ for the outer binsearch, $\mathcal{O}\left(n^2 L\right)$ for preprocessing, $\mathcal{O}(2^n)$ branches, and $\mathcal{O}(Ln \log(Ln))$ for processing one branch. In total:

$$\mathcal{O}\left(n^3 L \log(Ln) + Ln^2 2^n \log^2(Ln)\right)$$

Trees

Foundational data structure.

Many equivalent definitions:

- n nodes, each except for one has a single parent node;
- connected graph on n vertices with $n - 1$ edges;
- recursive: empty tree or a node connected to other trees;
- graph on n vertices where there is exactly one path between each pair of vertices.

Trees – Root, Parent, Child, Depth

Mathematically trees don't have to be rooted, but as a data structure they are.

Root – unique distinguished vertex.

Parent – unique node "up"; root has none.

Child – direct neighbour other than the parent.

Depth – root has depth 0, children of a node with depth d have depth $d + 1$.

Leaves – nodes with no children.

Trees – Ancestor, Descendant

Transitive closure of the parent relation is called *ancestor*.

Of the child relation – *descendant*.

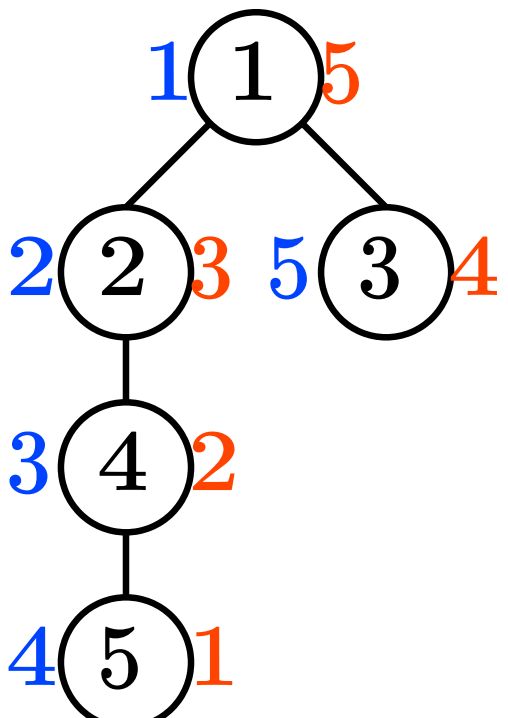
Trees – Forests

Many separate trees are called a *forest*.

Trees – DFS preorder and postorder

Preorder – give a number when entering.

Postorder – give a number when exiting.



Trees – Lowest Common Ancestor

Each two nodes have one path that joins them.

This path always goes through the *Lowest Common Ancestor* (LCA).

Trees – Lowest Common Ancestor

Each two nodes have one path that joins them.

This path always goes through the *Lowest Common Ancestor* (LCA).

Data structure for LCA: log-jumps.

1. Compute depth.
2. Compute jump$[v][k]$ as the ancestor 2^k higher in the tree. jump$[v][0]$ = parent$[v]$.
3. LCA of two vertices can be done by iterating largest possible jump until depth equalises.

Trees – Centroid Decomposition

Method for running divide-and-conquer algorithms on trees.

Idea – take a vertex and split the problem across its children.

Subproblems happen entirely in the subtrees, or involve the pivot vertex.

Effective if the pivot splits into “small” subtrees.

Trees – Centroid Decomposition

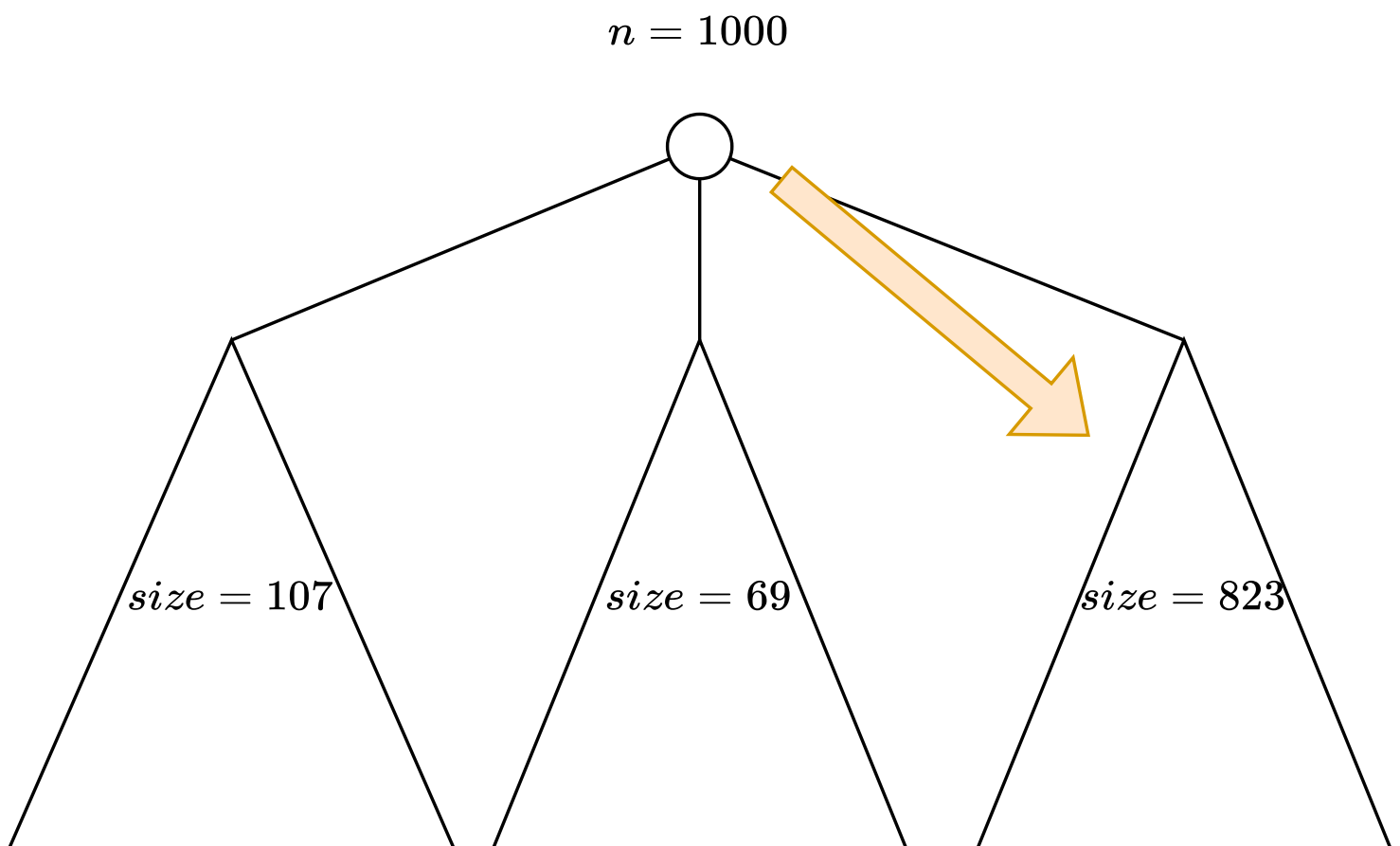
Centroid of a tree on n vertices is a vertex whose subtrees rooted at children are all at most size $\frac{n}{2}$.

Provably there is exactly one or two such vertices.

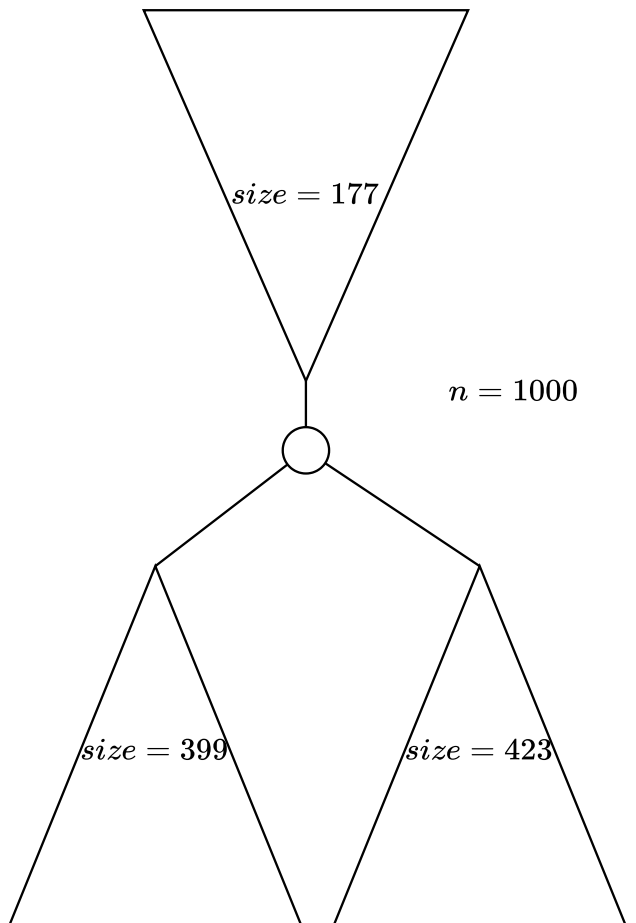
Algorithm to find – root arbitrarily, then compute subtree sizes.

Iteratively move the root to the heaviest subtree until we reach the centroid.

Trees – Centroid Decomposition



Trees – Centroid Decomposition



Graphs

Widespread data structure.

$G = (V, E)$

Network of vertices and edges between them.

Surprisingly many things can be modelled as a graph.

Usually no *multiedges* (E is a set).

Most of the time no *self-loops* ((v, v)).

Graphs

Multitude of different interpretations:

- travelling between vertices using edges, e.g. a map of cities and connections between them, road network where vertices are intersections, etc.
- edges are dependencies between vertices, e.g. this task has to be completed before this one;
- social network, e.g. this person follows this person.

Literally any binary relation can be a graph if you're brave enough.

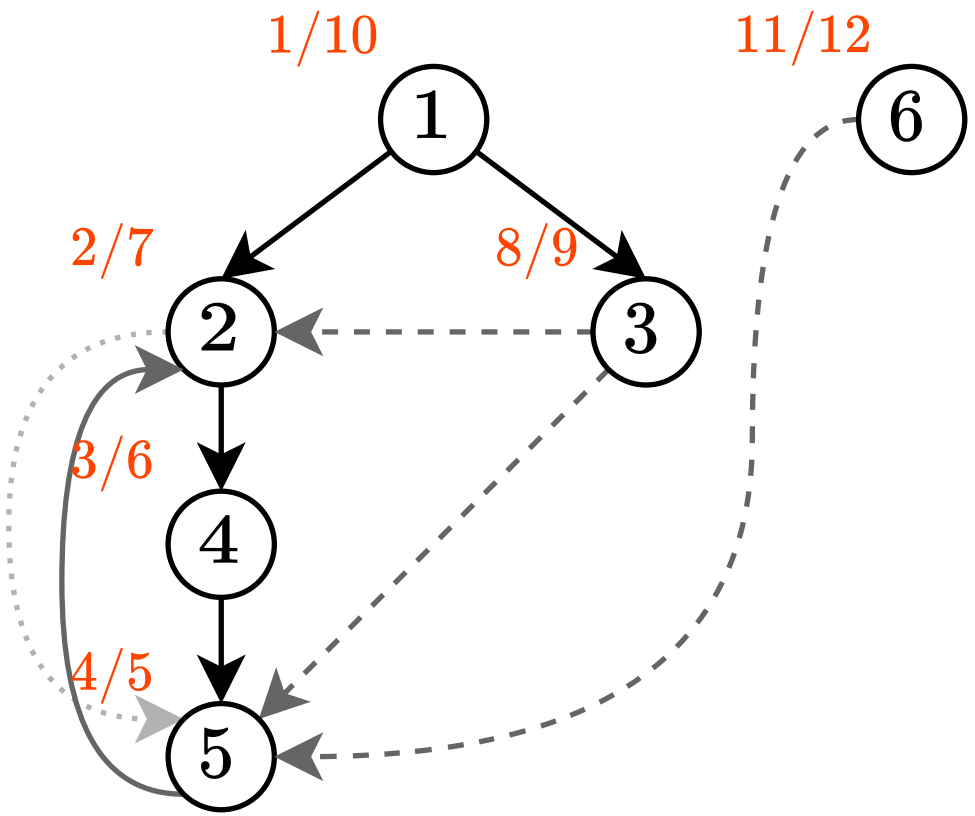
DFS tree structure

Tree edges – solid black.

Forward edges – dotted light grey.

Back edges – solid dark grey.

Cross edges – dashed dark grey.



Detecting cycles

Cycles are always back edges.

In a directed graph – any edge back to an *active* vertex.

In an undirected graph – any edge back to any visited vertex.

Graphs – bridges and articulation points

A *bridge* is any edge in a connected graph whose removal would disconnect it.

An *articulation point* is any vertex whose removal would do so.

If (v, u) is a bridge then v and u are articulation points (unless v and u have no other edges).

The other direction is not true.

Graphs – biconnected components

A *biconnected component* is a maximal subgraph that has no articulation points.

Any graph can be decomposed into a tree of biconnected components.

Easy to find articulation points and bridges in such a tree.

Graphs – biconnected components

```
fn ArticulationPoints() {
  time = 0;
  for u in V
    if pre[u] is None
      ArtDFS(u, u)
      is_art[u] = dfs_deg[u] > 1
}
```

Graphs – biconnected components

```
fn ArtDFS(v, p) {
  low[v] = pre[v] = time
  dfs_deg[v] = 0
  time += 1
  for u in N[v] {
    if u == p { continue }
    if pre[u] is None
      dfs_deg[v] += 1
      dfsAP(u, v);
      if pre[v] <= low[u] { is_art[v] = true }
      low[v] = min(low[v], low[u]);
    else
```

```
        low[v] = min(low[v], pre[u])
}
```

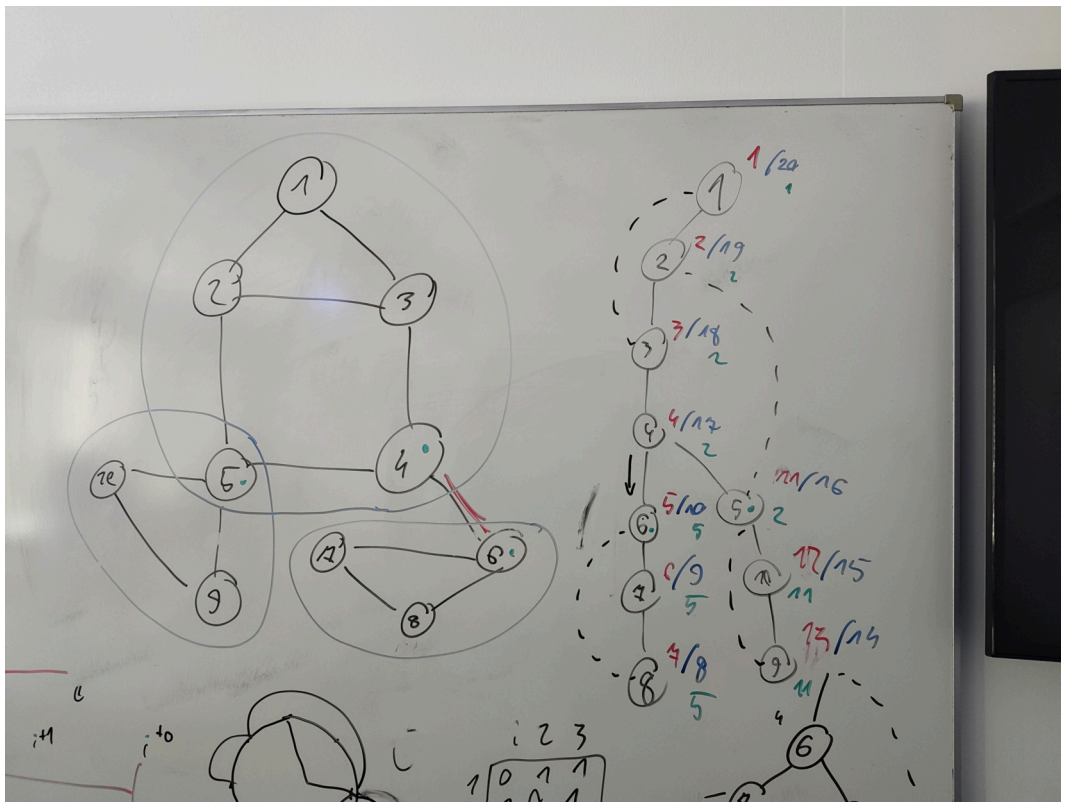
Graphs – biconnected components

How to find bridges?

One line change – the condition is `pre[v] < low[u]` for (v, u) to be a bridge.

The same algorithm works with depth instead of preorder times.

Graphs – biconnected components



Graphs – biconnected components

Some properties:

- any biconnected component can be decomposed to a cycle and then additional paths that connect vertices from the cycle or earlier paths;[2]
- if an articulation point belongs to two biconnected components, any path between two vertices in different components passes through that vertex;

[2] This is called an *ear decomposition*. See: https://en.wikipedia.org/wiki/Ear_decomposition

Graphs – strongly connected components

In a *directed* graph a connected component is not really coherent.

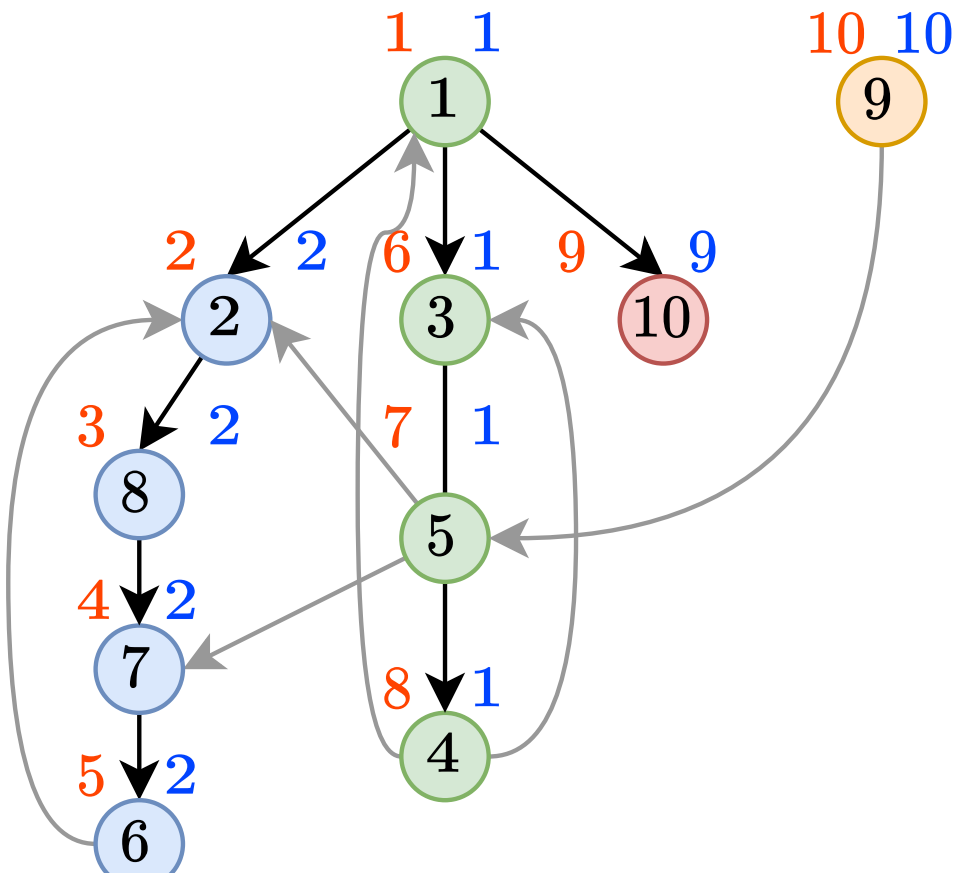
We define *strongly connected components* as maximal sets of vertices where there exists a pair between each pair of vertices (bidirectional).

Graphs – strongly connected components

Any directed graph can be decomposed into a graph of its SCCs.

An SCC graph is always a DAG (Directed Acyclic Graph).

Graphs – finding SCCs



Graphs – DAGs and DP

DAGs are very nice for DP.

Information flows along the edges – obvious “direction” of recurrence.

See you next week

LOO: 19.11.2025, 12:00 AM Good luck!