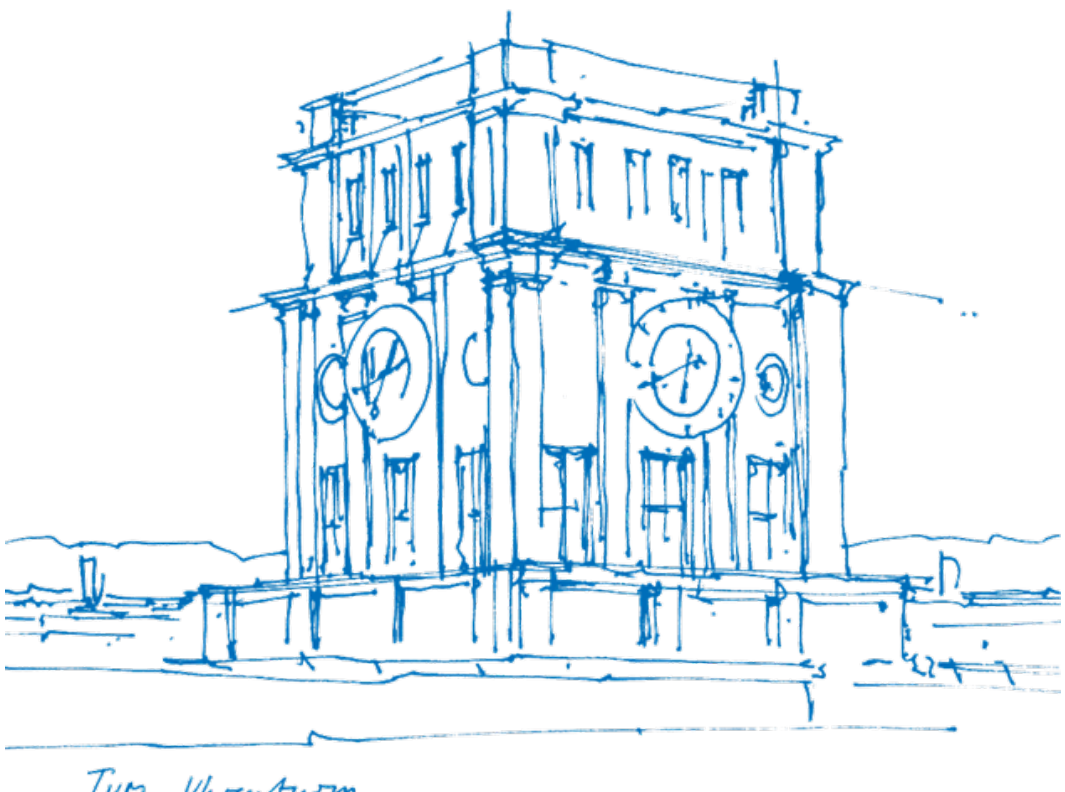
School of Computation, Information and Technology
Technical University of Munich

AACPP WiSe 2025/26

Final class



Mateusz Gienieczko

School of Computation, Information and Technology
Technical University of Munich

2026.02.04

ACT – Accelerated Command Training

Given n strings and m patterns calculate for each pattern how many strings contain it as a substring.

Library usage forces *online* answering, i.e. each pattern query has to be answered immediately – no pre-processing of patterns!

Alphabet $\Sigma = [a\text{-}z]$.

Let S_{cmd} be the sum of lengths of strings and S_{pat} of patterns.

Obviously $S_{\text{cmd}} \geq n$ and $S_{\text{pat}} \geq m$.

ACT – brute force

For each pattern go through all strings and match naively (e.g. using `.contains` in Rust or `std::search` in C++).

Worst-case: $\mathcal{O}\left(S_{\text{cmd}} \cdot S_{\text{pat}}\right)$. 2 points.

ACT – smarter brute force

Once a pattern comes we can spend some time pre-processing to make the substring checks faster.

Compute the prefix-suffix table and use KMP for matching.

$\mathcal{O}\left(m \cdot \left(S_{\text{pat}} + S_{\text{cmd}}\right)\right)$. 4 points.

ACT – model solution

How would we solve the task offline?

Take all the patterns, compute Aho-Corasick.

Each string s would match in $\mathcal{O}(|s| \cdot |\Sigma|)$.

ACT – model solution

If we had a single string we could compute its **suffix tree** and then quickly check each pattern p in $\mathcal{O}(|p|)$.

Generalising this to n strings:

- How to build a suffix tree of many strings?
- How to extract the number of unique strings matched, not just a boolean?

ACT – suffix tree of a dictionary

We can create a suffix tree of many strings by concatenating all of them interspersed with special "end-of-string" tokens.

Need to be able to tell from the end token which original string it refers to – use distinct tokens or have a map.

We increase the total build time and size of the tree to $\mathcal{O}(n + S_{\text{cmd}})$, but it's OK.

ACT – getting unique matches

We can augment the suffix tree similarly to how we augmented BSTs.

In each node keep some information about which strings we are in the middle of.

In other words, the set of first end tokens encountered on paths down from this node.

ACT – getting unique matches

Since n is small this can be a bitset.

Recursively computed from the root. Let `which` be a map from indices of the "megastring" to IDs of source strings.

```
if which(node.l) == which[node.r]
    // This is an internal node with no word switch.
    node.parent.bitmask |= node.bitmask
else
    node.parent.bitmask.set(which(node.l))
```

ACT – getting unique matches

When searching for the pattern we either end in a node or in the middle of an edge.

Edge case – if in the middle of an edge and the edge contains a terminator, only that word counts.

ACT – complexity and memory

Complexity is $\mathcal{O}\left(S_{\text{cmd}} \cdot \frac{n}{B} + S_{\text{pat}} + m \cdot \frac{n}{B}\right)$ where B is the bitwidth of a machine word (on our judging system it's 32).

This is fine – but how much memory?

Need the "megastring" (10^7 characters), the suffix tree ($10^7 + 2 \cdot 10^3$ nodes), each node holds a bitset ($\left\lceil \frac{10^3}{8} \right\rceil = 125$ bytes.

In the model a node has $192B$ and the string takes $8B$ per char.

This is about $8 \cdot 10^7 + 192 \cdot \left(10^7 + 2 \cdot 10^3\right) \approx 1.91$ GiB.

Grading scale

grade	points
1.0	95
1.3	84
1.7	70
2.0	60
2.3	56
2.7	52
3.0	48
3.3	44
3.7	40
4.0	30
4.3	20
4.7	10

Carefully constructed using cutting-edge algorithms to deliver maximum motivation for second-round submissions.

You can submit until **16th of March, 12:00PM**.

Closing words

This is the end!

What did we *not* cover?

- More detailed analysis of greedy vs DP.
- Number Theory
- NP-hardness

Number theory lightning

- Modular arithmetic
  - Add, Sub, Mul, Power easy
  - Division hard (multiplicative inverse, FLT, Euler's)
  - Discrete log harder
  - Discrete roots even harder
- FFT

NP-hardness

Some problems are too hard to solve in polynomial time.

Approximations exist (sometimes).

Heuristics exist (sometimes good).

Clever backtracking.

Closing words

We’re from the database chair – come and do projects with us!

A lot of interesting algorithms and performance-focused work.

Some Other Fields are so boring in comparison...

Closing words

Query optimisation relies on Dynamic Programming.

Cloud environments make it harder (more parameters).

DP approaches to find Pareto-optimal query plans, latency vs cost.

Closing words

Plenty of innovation in how we optimise and represent plans - see link/cut tree representation.

B-trees, mainstay of indexing. Many variants, requires understanding of complexity models.

Indexing in general. GIN indexes. JSON acceleration (rsonpath-lut).

Closing words

Sooo much to do with string processing (Adrian).

Geometry for processing geographical data.

Graphs, graphs everywhere (graph dbs, task DAGs).

Closing words

Go to our website!

https://db.in.tum.de

See you and good luck!