Algorithm Engineering **Summer 2026**

Dr. Phillip Keldenich
Dr. Dominik Krupke

Submission due: 23.04.2026 16:45

Homework 1

Work in teams of two. Homework is due every two weeks.

Submission format: Host your solution in a private Git repository on one of the following platforms and add Rouven as a collaborator: IBR GitLab (`kniep`), TU GitLab (`r.kniep`), GitHub (`rkniep`), or Codeberg (`rkniep`). If you want to use a different platform, please ask first.

Before the deadline, send the link to your submission commit by e-mail to `kniep@ibr.cs.tu-bs.de`.

Exercise 1 (Setup): **(5 points)**

You will need to have a working C++ compiler and python environment for almost all programming exercises in this course. We will typically provide docker files for the exercises to allow you to easily build the provided examples in a somewhat reproducible manner and also to allow us to easily run your solutions. On typical Linux or Mac OS machines with a working python installation (such as anaconda), a C++ compiler and CMake, the provided examples can also be built and run without relying on the docker files.

Under `https://github.com/tubs-alg/algorithm-engineering-material-ss2026`, you can find a publicly readable git repository. In this repository, you can find a `sheet1/` folder with a subdirectory for each exercise (`exercise01`, `exercise02`, `exercise03`), each containing the programs or program skeletons, a `Dockerfile`, and a `CMakeLists.txt`.

The program for this task need not be modified in any way; the goal of this exercise is to get you ready to modify, build and run our examples. Each exercise directory contains its own `Dockerfile`. To build and run this exercise using Docker, `cd` into the `exercise01` directory and run: `docker build -t ae26_ex1 .` followed by `docker run --rm -it ae26_ex1`. To enter the container interactively, override the entrypoint: `docker run --rm -it --entrypoint /bin/bash ae26_ex1`. Note that since the container is ephemeral (`--rm`), any changes you make inside it are lost when you exit. To modify and rebuild programs without losing your edits, edit the source files on your host machine and rebuild the container, or work natively on your machine as described below.

If you have installed CMake and are working under Linux or Mac OS, you can also build the example via CMake, by running the commands `mkdir cmake-build-release`, followed by `cmake -S . -B cmake-build-release -DCMAKE_BUILD_TYPE=Release` to configure and `cmake --build cmake-build-release` in the `exercise01` directory.

Running the program produces a string consisting of two parts that should be unique for each participant; the second part is a special hash of the first that allows us to validate the output. For this task, you only have to specify the output; we expect a unique output for each participant.

If you encounter problems that you cannot solve by internet research, please contact us sooner rather than later.

Exercise 2 (Bad Programs): **(10+15 points)**

The `exercise02` subdirectory for this exercise contains two programs with suboptimal performance.

a) The program `problem_program_a` takes a single integer N as input and contains a problematic function that runs into a high number of branch mispredictions; running it under `perf stat` reveals that, depending on compiler and CPU, up to 14 % of branches in it end up being mispredicted, including a considerable number of predictable branches in the test input generation.

   Identify and describe the reason for the mispredictions. Are these mispredictions unavoidable? If so, argue why. Otherwise, fix the program to avoid the mispredictions without changing the result it outputs for any N. Report the running time of the routine for $N = 1024$, $N = 8192$ and $N = 32768$ before and after your changes; if you cannot manage to run $N = 32768$ without exhausting your RAM, use the largest $N = 2^k$ you can manage instead (the program needs slightly more than $8N^2$ bytes of memory).

b) The program `problem_program_b` reads a graph from a JSON file and computes a clique on it by repeatedly going through the vertices in random order and computing a greedy maximal clique in that order. It suffers from poor data structure selection leading to poor memory performance and lots of cache misses. Your task is to optimize the implementation without changing the underlying algorithm: after your optimization, assuming the random number generator behaves truly randomly, the probability with which any clique is selected should be the same as before.

   You are otherwise free to change the program in any way: you can change the graph data structure, the way possible extensions are tracked, or any other data structure to your liking. Before changing the code, describe at least two different alternative implementations and argue why they may have better performance than the current implementation. Also write down the running times printed by the program for each of the 9 example graphs before and after your optimization.

Exercise 3 (Traveling Salesman Problem – 2-Opt): **(5+5+4+6 points)**

Background. Given n points in the plane with coordinates (x_i, y_i), the *Euclidean Traveling Salesman Problem (TSP)* asks for the shortest tour that visits every point exactly once and returns to the starting point. A tour can be represented as a permutation $\pi = (\pi_0, \pi_1, \ldots, \pi_{n-1})$ of the point indices, where the tour visits $\pi_0 \to \pi_1 \to \cdots \to \pi_{n-1} \to \pi_0$.

The *2-opt* heuristic is a simple local search that iteratively improves a tour. A *2-opt move* picks two non-adjacent edges (π_i, π_{i+1}) and (π_j, π_{j+1}) in the tour (indices modulo n), removes them, and reconnects the tour by reversing the segment $\pi_{i+1}, \ldots, \pi_j$. This move is *improving* if the new tour is shorter, i.e.,

$$d(\pi_i, \pi_j) + d(\pi_{i+1}, \pi_{j+1}) < d(\pi_i, \pi_{i+1}) + d(\pi_j, \pi_{j+1}),$$

where $d(a, b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$ is the Euclidean distance.

Starting from a random tour, we repeatedly scan all pairs (i, j) for improving 2-opt moves until no more improvements can be found (the tour is *2-optimal*).

Below are three variants of this idea that differ in a single detail: what happens after an improving move is found?

Variant 1 — First-improvement (classical): Apply the first improving move found, then restart the scan from the beginning.

function FirstImprovementTwoOpt(π, points)
  improved $\leftarrow$ **true**
  while *improved* **do**
    improved $\leftarrow$ **false**
    for $i \leftarrow 0$ **to** $n-2$ **do**
      for $j \leftarrow i+2$ **to** $n-1$ **do**
        if $i = 0$ **and** $j = n-1$ **then**
          continue ▷ adjacent edges in the cyclic tour
        $\delta \leftarrow d(\pi_i, \pi_j) + d(\pi_{i+1}, \pi_{j+1}) - d(\pi_i, \pi_{i+1}) - d(\pi_j, \pi_{j+1})$
        if $\delta < 0$ **then**
          reverse $\pi[i{+}1 \ldots j]$
          improved $\leftarrow$ **true**
          restart scan ▷ break both loops, begin next pass
  return π

Variant 2 — Full-scan (continue scanning): Apply each improving move immediately and keep scanning; only restart when a full pass finds no improvement.

function FullScanTwoOpt(π, points)
  repeat
    improved $\leftarrow$ **false**
    for $i \leftarrow 0$ **to** $n-2$ **do**
      for $j \leftarrow i+2$ **to** $n-1$ **do**
        if $i = 0$ **and** $j = n-1$ **then**
          continue ▷ adjacent edges in the cyclic tour
        $\delta \leftarrow d(\pi_i, \pi_j) + d(\pi_{i+1}, \pi_{j+1}) - d(\pi_i, \pi_{i+1}) - d(\pi_j, \pi_{j+1})$
        if $\delta < 0$ **then**
          reverse $\pi[i{+}1 \ldots j]$
          improved $\leftarrow$ **true** ▷ continue scanning
  until $\neg$ *improved*
  return π

Variant 3 — Best-improvement: Scan all pairs, record the single most improving move, apply it, then restart.

function BestImprovementTwoOpt(π, points)
  repeat
    $\delta^* \leftarrow 0$; $i^* \leftarrow -1$; $j^* \leftarrow -1$
    for $i \leftarrow 0$ **to** $n-2$ **do**
      for $j \leftarrow i+2$ **to** $n-1$ **do**
        if $i = 0$ **and** $j = n-1$ **then**
          continue

$\delta \leftarrow d(\pi_i, \pi_j) + d(\pi_{i+1}, \pi_{j+1}) - d(\pi_i, \pi_{i+1}) - d(\pi_j, \pi_{j+1})$
if $\delta < \delta^*$ **then**
$\delta^* \leftarrow \delta$; $i^* \leftarrow i$; $j^* \leftarrow j$
if $i^* \geq 0$ **then**
reverse $\pi[i^*{+}1 \ldots j^*]$
until $i^* < 0$
return π

Note: When $i = 0$ and $j = n - 1$, the edges (π_0, π_1) and (π_{n-1}, π_0) share vertex π_0 and are adjacent in the cyclic tour — this pair must be skipped. All three variants take the initial tour as input; the benchmark generates a random permutation and passes the same starting tour to every variant so results are directly comparable.

In the `exercise03` subdirectory, you will find a Python/C++ project with functions to complete. A `benchmark.py` script generates random instances (sizes $n = 100$, 500, 1 000, 5 000, 20 000), runs all implementations on the same points, and prints a comparison table per size. Each run has a 10-second timeout, so even slow variants on large instances will not block the benchmark. See the `README.md` in that directory for setup instructions. All single-run functions take a list of (x, y)-tuples and an `initial_tour` (a permutation of $0, \ldots, n{-}1$), and return an improved tour as a list of indices.

Parts a)–c) are implementation tasks; submit your code and verify it passes the provided tests (`pytest tests/ -v`). Part d) asks you to run the benchmark and report your findings.

a) **Python implementations.** Implement all three Python 2-opt variants:
   - `first_improvement_two_opt` in `first_improvement.py` (Variant 1),
   - `full_scan_two_opt` in `full_scan.py` (Variant 2),
   - `best_improvement_two_opt` in `best_improvement.py` (Variant 3).

b) **C++ implementations via pybind11.** Implement all three 2-opt variants in C++ in `_core.cpp`: `cpp_first_improvement` (Variant 1), `cpp_full_scan` (Variant 2), and `cpp_best_improvement` (Variant 3). The project skeleton already contains the pybind11 bindings; you only need to fill in the algorithm logic. Respect the timeout parameter by checking the deadline periodically. A coarse-grained check is sufficient: for Variant 1, check the deadline after each restart round; for Variants 2 and 3, check it after each full scan over all candidate 2-opt pairs. If the timeout is reached, return the current tour immediately. There is no need to check the deadline in every inner-loop iteration.

c) **Parallel 2-Opt.** 2-opt converges to a local optimum that depends on the random starting tour. A simple way to improve solution quality is to run multiple independent searches with different random seeds in parallel and keep the best result. Implement `parallel_two_opt` in `_core.cpp`: launch `num_threads` threads using `std::thread`, each running the full-scan 2-opt with a different seed, and return the shortest tour found.

d) **Benchmark analysis.** Run the benchmark (`python benchmark.py`) and write a short report (at most one page) addressing the following:

- How do the runtimes and solution qualities of the three 2-opt variants compare in Python? Do they always converge to the same tour when starting from the same initial tour? Why or why not?
- How large is the Python-to-C++ speedup? How does the speedup vary across instance sizes?
- How do tour quality and wall-clock time change for the parallel variant as the number of threads increases from 1 to 8?

Note: the parallel variant generates its own random starting tours internally, so its results are not directly comparable to the single-threaded variants.