Challenge 1 - Primality test on 512-bit numbers

Course: 095946 – Advanced Algorithms and Parallel Programming
Student: Stiliyan Andreev (ID: 11158007) - stiliyan.andreev@mail.polimi.it

I implemented the "Randomized primality test 2" from Lecture 10. The core idea is to compute $a^{n-1} mod n$with fast exponentiation and, during the recursion, watch for a non-trivial square root of 1: if at some step $x^2 \equiv 1 (\mathrm{mod} n)$but $x \notin \{1, n-1\}$, we mark the candidate as composite immediately. This logic follows slides 9–14: the definition of non-trivial square roots, the recursive `power` routine that halves the exponent, the check right after squaring, and the outer `primalityTest` that draws a random base $a \in [2, n-1]$and accepts only if the final residue is 1 and no bad square root was seen.

To satisfy the 512-bit requirement, I used Boost.Multiprecision with a fixed 512-bit unsigned backend, so arithmetic is exact on the required width without writing a custom bignum. The driver repeats the one-shot test $k$times on independent bases; this reduces the one-sided error as in the lecture statement about failure probability.

The code is organized as a small CMake project: a header-only `primality.hpp` (API + implementation), a tiny `main.cpp` for validation, and a `benchmark.cpp` using Google Benchmark. This matches the challenge brief, which also asks us to measure asymptotic behavior.

Experimental setup. I ran everything on a Google Colab VM (Linux, system g++ with C++17). The benchmark tool reported two vCPUs at ~2.2 GHz and showed cache sizes; these runs are single-threaded. Inputs included a synthetic 512-bit odd candidate $n = 2^{511} + 159$(to exercise 512-bit math), small sanity cases (17, 18), and a few Carmichael numbers (561, 1105, 1729, 2465, 2821, 6601). All were processed through the same fixed-width `uint512` type.

Correctness sanity checks. The driver prints that 17 is classified as prime and 18 as composite. The Carmichael numbers-classic Fermat liars-are rejected here thanks to the inner "non-trivial $\sqrt{1}$" check prescribed by the lecture, so the algorithm does not accept them as probably prime.

Performance measurements. I timed the procedure with Google Benchmark for bit-lengths 256, 384, and 512 (all represented inside the same 512-bit type) and for $k \in \{1,5,10\}$. Reporting medians over five repetitions, a concise sample is:
256 bits: $k = 1$→ 109,780 ns; $k = 10$→ 147,636 ns.
512 bits: $k = 1$→ 87,201 ns; $k = 10$→ 103,878 ns.
The scaling with $k$is roughly linear, as expected for $k$independent trials. Bit-length trends can wobble slightly because arithmetic is always carried out in a 512-bit backend and the VM is noisy, but the measurements are consistent with fast exponentiation doing $O(\log n)$modular squarings/multiplications.

Complexity discussion. The recursive exponentiation halves the exponent at each step, so the number of modular multiplications grows like $O(\log\ n)$(slide 11). The additional comparisons performed only when the squared residue equals 1 do not change the asymptotic order; the lecture summary places the routine with checks in the same polylogarithmic regime. Repeating the whole iteration $k$times multiplies the work by $k$and reduces the one-sided error, exactly as stated in the course notes.

Design choices. I kept the implementation header-only to make inclusion in tests and benchmarks trivial. I used `std::mt19937_64` to pick bases uniformly within $[2, n-2]$, in the spirit of the lecture's `random(2, n-1)`; the API exposes `primalityTest_k(n, k)` so the confidence can be increased by raising k. Finally, I validated with tiny primes/composites and with Carmichael numbers to show the practical effect of the inner check.

Conclusion. The implementation follows the lecture exactly: fast exponentiation plus the non-trivial $\sqrt{1}$check inside `power`. It rejects Carmichael numbers in the quick tests and the benchmark results show the expected behavior: runtime grows with $k$and is consistent with the $O(\log\ n)$structure of exponentiation. This meets the challenge requirements-512-bit arithmetic, validation, and asymptotic measurements-within a compact, well-commented codebase.