0. Przygotowanie repozytorium i środowiska (zadanie startowe)

- Utwórz repozytorium Git (lokalnie i opcjonalnie GitHub/GitLab).
- Plik README.md z krótkim opisem projektu i instrukcją uruchomienia.
- Konfiguracja budowania:
    - użyj CMake (CMakeLists.txt) lub prostego Makefile.
- Dodaj .gitignore (np. dla build/, *.o, bin/).
- Struktura proponowana:

```
css

rsa-project/
├─ src/
│  ├─ main.cpp
│  ├─ rsa.cpp
│  ├─ rsa.h
│  ├─ bigint.cpp (opcjonalnie)
│  ├─ bigint.h (opcjonalnie)
│  ├─ math_utils.cpp
│  └─ math_utils.h
├─ tests/
│  ├─ test_math.cpp
│  └─ test_rsa.cpp
├─ examples/
│  └─ sample_run.txt
├─ CMakeLists.txt
└─ README.md
```

- Zależności:
    - jeśli używasz Boost.Multiprecision: zapisz to w README i w CMakeLists.txt.
    - opcjonalnie: Catch2/GoogleTest dla testów — dodaj instrukcję instalacji.

Kryterium akceptacji: repo zawiera szkielet projektu, działający build (empty main kompiluje).

---

1. Podstawowe narzędzia matematyczne (fundament)

Cel: poprawne, przetestowane implementacje GCD, rozszerzonego Euklidesa, modular inverse, modular pow.

1.1. GCD (największy wspólny dzielnik)

- Funkcja: cpp_int gcd(cpp_int a, cpp_int b)
- Testy:
  - gcd(48,18) == 6
  - gcd(101, 10) == 1

1.2. Rozszerzony algorytm Euklidesa

- Funkcja: std::tuple<cpp_int, cpp_int, cpp_int> extended_gcd(cpp_int a, cpp_int b) zwraca (g, x, y) gdzie ax + by = g.
- Testy:
  - Dla (a,b) = (240, 46) sprawdź g=2 i że 240*x + 46*y == 2.

1.3. Modular inverse

- Funkcja: cpp_int modinv(cpp_int a, cpp_int m) — użyj extended_gcd; jeśli gcd != 1, zwróć błąd/wyjątek.
- Testy:
  - modinv(3,11) == 4 (bo 3*4 mod 11 = 1).

1.4. Fast modular exponentiation

- Funkcja: cpp_int modexp(cpp_int base, cpp_int exp, cpp_int mod) — metoda potęgowania przez kwadraty.
- Testy:
  - modexp(2,10,1000) == 24 (2^10=1024 mod 1000 = 24).

- Test z dużymi wykładnikami (porównanie z boost pow / ręcznym wynikiem dla małych modów).

Kryterium akceptacji: wszystkie funkcje mają jednostkowe testy i przechodzą je.

---

2. Generowanie i testowanie liczb pierwszych

Cel: generowanie dużych liczb prawdopodobnie pierwszych z użyciem testu Miller–Rabin.

2.1. RNG (bezpieczny generator)

- Funkcja do losowania bitów/ciągów bajtów: użyj std::random_device + std::mt19937_64 jako minimum; w README zaznacz, że to demonstracja (dla produkcji użyć /dev/urandom lub biblioteki kryptograficznej).
- Funkcja: cpp_int random_odd_kbit(int k) — generuje losową nieparzystą liczbę o k bitach.

2.2. Implementacja testu Miller–Rabin

- Funkcja: bool is_probable_prime(cpp_int n, int rounds = 10)
- Wybór podstaw losowych (a) w zakresie [2, n-2].
- Testy:
  - Sprawdź is_probable_prime dla małych znanych liczb pierwszych i złożonych (np. 2,3,5,7,11, 561 (Carmichael) — powinien wykryć złożoność przy wystarczającej liczbie rund).
  - Porównaj wyniki z bibliotekową funkcją (jeśli dostępna) dla małych zakresów.

2.3. Generowanie p i q

- Funkcja: cpp_int generate_prime(int bits):
  - generuj random_odd_kbit(bits),

    - testuj Miller–Rabin, jeśli nieprzechodzi — inkrementuj o 2 i testuj dalej (lub generuj nowe).
- Testy:
    - Generuj 16-bitowe liczby i sprawdź, że są pierwsze.
    - Generuj kilkukrotnie, sprawdź różnorodność.

Kryterium akceptacji: generate_prime zwraca liczbę, która przechodzi testy primości i jest zadeklarowanej długości bitów.

---

3. Generowanie kluczy RSA

Cel: poprawna generacja pary (n, e) i (n, d).

3.1. Algorytm

- Wybierz p = generate_prime(bits/2), q = generate_prime(bits/2) (upewnij się, że p != q).
- Oblicz n = p * q.
- Oblicz phi = (p - 1)*(q - 1).
- Wybierz e:
    - typowo e = 65537, sprawdź gcd(e, phi) == 1, jeśli nie — wybierz inny e (np. kolejne nieparzyste).
- Oblicz d = modinv(e, phi).
- Zapisz klucze do plików:
    - public.key (np. JSON lub prosty tekst: e\nn\n)
    - private.key (d\nn\n)

3.2. Funkcje/Pliki

- rsa.h/.cpp:
    - struct PublicKey { cpp_int e, n; };
    - struct PrivateKey { cpp_int d, n; };

- KeyPair generate_keys(int bits)

3.3. Testy

- Test na małej konfiguracji z ręcznie dobranymi p,q (np. p=61,q=53) — porównaj obliczone e,d,n z znanymi wartościami.
- Sprawdź, że e * d mod phi == 1.

Kryterium akceptacji: klucze się generują, zapisują poprawnie, i spełniają relację modułową.

---

4. Szyfrowanie i deszyfrowanie RSA

Cel: poprawne szyfrowanie i odzyskiwanie wiadomości.

4.1. Kodowanie wiadomości

- Prosty sposób (dla demonstracji):
  - zamień bajty wiadomości na wielką liczbę (big-endian) — m.
  - wymaganie: m < n. Jeśli m >= n, rozbij wiadomość na bloki tak, by każdy blok < n.
- Alternatywa: mapowanie przez konwersję UTF-8 → hex → cpp_int.

4.2. Funkcje

- cpp_int rsa_encrypt_block(cpp_int m, PublicKey pub) → c = modexp(m, e, n)
- cpp_int rsa_decrypt_block(cpp_int c, PrivateKey priv) → m = modexp(c, d, n)
- Dla wieloblockowej wiadomości: implementuj encrypt_message(string plain, PublicKey pub) / decrypt_message(...).

4.3. Padding (informacja)

- Dla projektu możesz użyć prostego schematu (zero-padding, albo prefiks długości), ale w dokumentacji opisz, że produkcyjne systemy używają PKCS#1 / OAEP.

4.4. Testy

- Test z przykładami:
    - p=61, q=53 → n=3233, e=17, d=2753 (klasyczny przykład). Szyfruj m=65 → c=2790 (znane wartości) i deszyfruj z powrotem.
    - Szyfruj krótki tekst, porównaj po deszyfrowaniu.

Kryterium akceptacji: szyfrowanie i deszyfrowanie dla bloków działa odwrotnie: decrypt(encrypt(m)) == m dla testowanych bloków.

---

5. Interfejs użytkownika / scenariusz użycia

Cel: prosty CLI do demonstracji.

5.1. Funkcjonalności CLI

- ./rsa genkeys --bits 1024 --out pub.key priv.key
- ./rsa encrypt --pub pub.key --in message.txt --out cipher.bin
- ./rsa decrypt --priv priv.key --in cipher.bin --out message_out.txt
- Opcje dodatkowe: --force, --format hex|bin|base64

5.2. Pliki wejścia/wyjścia

- message.txt — zwykły tekst (UTF-8)
- cipher.bin — zapis binarny (każdy blok jako długość + big-int) albo base64

5.3. Przykładowy run (w README)

- Zamieść krok po kroku przykład: generacja kluczy, szyfrowanie, deszyfrowanie, porównanie oryginału i wyniku.

Kryterium akceptacji: demonstracyjny scenariusz działa i jest opisany w README.

---

6. Testy jednostkowe i walidacja

Cel: pełne testy matematyczne i integracyjne.

6.1. Testy matematyczne (tests/test_math.cpp)

- gcd, extended_gcd, modinv, modexp.
- Miller–Rabin na zestawie wartości (w tym Carmichael numbers).

6.2. Testy RSA (tests/test_rsa.cpp)

- Generowanie kluczy (dla małych bitów) i test: decrypt(encrypt(m)) == m dla kilku m.
- Test z ręcznie znanym przykładem p,q,e,d.

6.3. Automatyzacja

- Dodaj target make test lub ctest w CMake.
- Opcjonalnie: GitHub Actions z prostym CI, uruchamiającym build i testy przy każdym push.

Kryterium akceptacji: wszystkie testy jednostkowe przechodzą w repo.

---

7. (Opcjonalnie) Hybrydowe szyfrowanie z AES

Cel: zasymulować realny protokół — RSA do wymiany klucza AES, AES do treści.

7.1. Wybór implementacji AES

- Możliwości:
  - użyć biblioteki (np. OpenSSL, Crypto++), lub
  - zaimplementować uproszczony AES (większy nakład pracy).
- Dla demo zalecam użyć biblioteki i wyjaśnić działanie w raporcie.

7.2. Schemat

- Nadawca:
  - generuje losowy aes_key (128/256 bitów),
  - szyfruje wiadomość AES(aes_key),
  - szyfruje aes_key RSA(pub),

    - wysyła rsa_encrypted_key + aes_ciphertext.
- Odbiorca:
    - odszyfrowuje aes_key RSA(priv),
    - odszyfrowuje AES ciphertext.

7.3. Testy

- Test integracyjny: pełny cykl — wynik plaintext == oryginalny.

Kryterium akceptacji: hybrydowy scenariusz przy użyciu RSA do klucza i AES do danych daje poprawny rezultat.

---

8. Pomiary wydajności i profilowanie

Cel: zebrać dane o skalowaniu (bez prognoz czasowych).

8.1. Co mierzyć

- czas generowania kluczy (dla różnych rozmiarów bitów).
- czas szyfrowania/deszyfrowania jednego bloku.
- czas szyfrowania całej wiadomości (jeśli rozbijasz na bloki).
- (jeśli hybrydowo) czas AES vs RSA dla tych samych wielkości danych.

8.2. Narzędzia

- użyj std::chrono::high_resolution_clock do zmierzenia czasu w programie.
- albo time ./rsa ... w shellu.

8.3. Raportowanie

- zapisz wyniki w CSV (tests/perf_results.csv) i pokaż w raporcie wykresy/wnioski (np. jak rosną czasy z rozmiarem klucza).

Kryterium akceptacji: masz plik z wynikami pomiarów dla kilku rozmiarów kluczy i przykładów.

---

9. Dokumentacja techniczna i raport

Cel: przygotować wymagany ~10-stronicowy raport + krótką instrukcję użytkownika.

9.1. Zawartość raportu

- Wprowadzenie i cel projektu.
- Matematyczne podstawy RSA (p, q, n, ϕ(n), e, d).
- Implementacja: opis algorytmów, kluczowe fragmenty kodu lub pseudokod (extended_gcd, modexp, Miller–Rabin, generate_keys).
- Testy: przykłady wejść/wyjść; przykład p=61,q=53.
- Wyniki wydajności.
- Analiza bezpieczeństwa: dlaczego RSA działa, konieczność paddingu, zagrożenia.
- Wnioski i dalsze prace.

9.2. User Guide (w README lub oddzielny USER_MANUAL.md)

- Jak zbudować projekt (CMake/Make).
- Przykład użycia CLI i pliki wej/wyj.
- Jak uruchomić testy i pomiary wydajności.

Kryterium akceptacji: raport i user guide są kompletne i umożliwiają replikację wyników.

---

10. Końcowe zadania i kontrolna lista jakości

- Code review: wzajemne sprawdzenie kodu zespołowego.
- Statyczna analiza kodu (np. clang-tidy) i sprawdzenie pamięci (Valgrind) dla krytycznych modułów.
- Upewnij się, że nie wypisujesz prywatnego klucza w logach.
- Dodaj licencję i plik CONTRIBUTING.md jeśli wymagane.

Ostateczne kryteria akceptacji (checklista):

- Project builds cleanly.
- Podstawowe funkcje matematyczne przetestowane.
- Generowanie kluczy RSA poprawne i zapisywane do plików.
- Szyfrowanie / deszyfrowanie bloków poprawne.
- CLI / przykładowy scenariusz udokumentowany i działający.
- Testy jednostkowe i integracyjne przechodzą.
- Raport techniczny i instrukcja użytkownika przygotowane.
- (Opcjonalnie) Hybrydowe szyfrowanie z AES działa i jest przetestowane.
- Wyniki pomiarów zebrane i zapisane.

Dodatkowe wskazówki implementacyjne i dobre praktyki

- Trzymaj separację warstw: math_utils (gcd, modexp), primes (Miller–Rabin, gen prime), rsa (keygen, encrypt/decrypt), cli.
- W testach używaj małych bitów (np. 16/32), by szybko weryfikować logiczne błędy; dla demonstracji generuj też 512/1024 bity i rejestruj wyniki.
- Nie używaj double w obliczeniach kryptycznych — wszystko w integerach wielkiej precyzji.
- Loguj tylko niekrytyczne informacje (unikaj wypisywania prywatnych kluczy).
- W README wyraźnie zaznacz ograniczenia projektu (brak OAEP, demo, RNG nieprodukcyjny jeśli użyty mt19937_64).