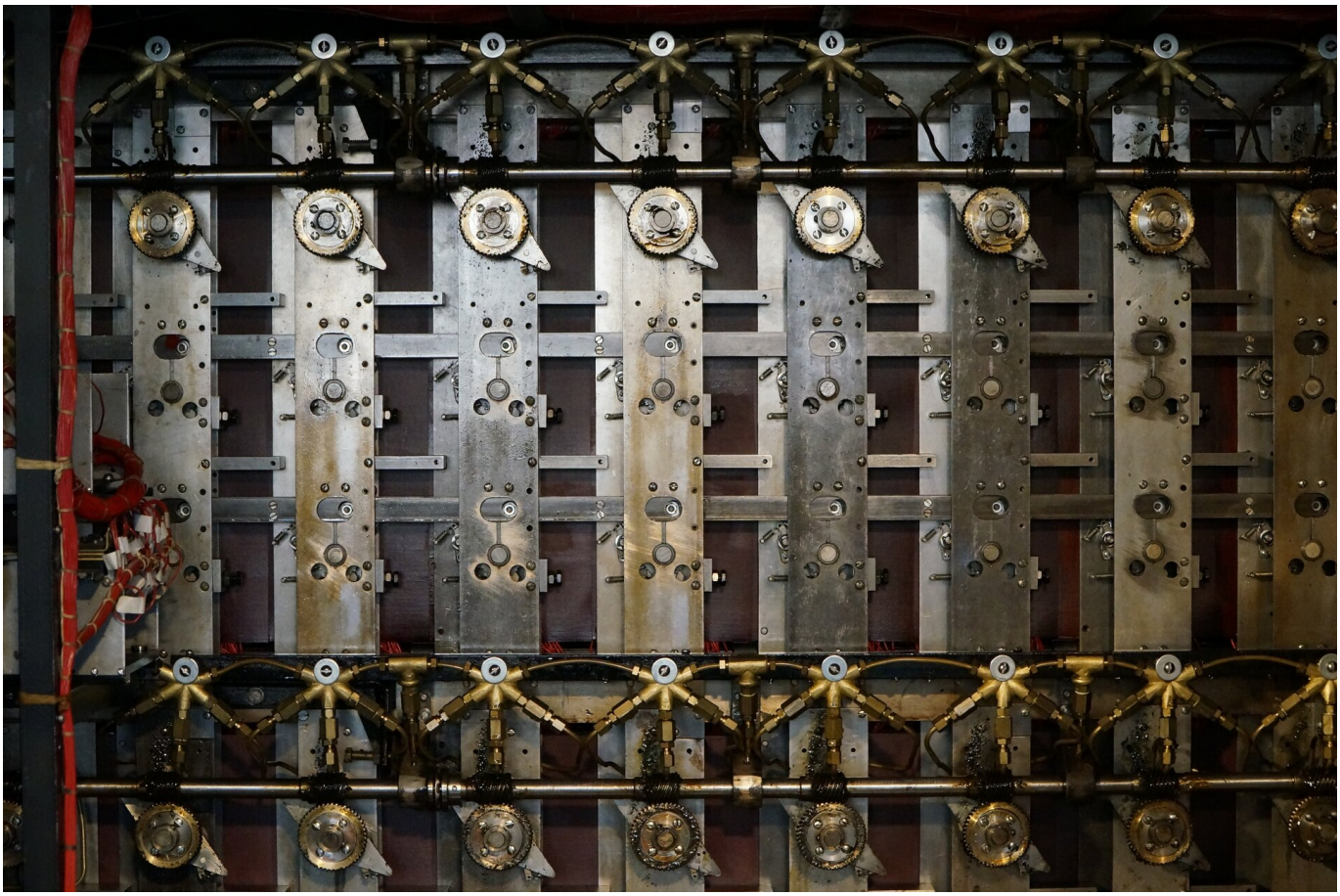


Szyfry symetryczne i asymetryczne w języku C++

- Wprowadzenie
- Przeczytaj
- Prezentacja multimedialna
- Sprawdź się
- Dla nauczyciela

Źródło: awsloley, domena publiczna.

Poznaliśmy już definicję szyfrów symetrycznych i asymetrycznych. Wiemy, czym się różnią, jak się dzielą oraz jakie jest ich zastosowanie. Kolejny krok to napisanie przykładowych programów szyfrujących. Ten e-materiał poświęcony jest ich implementacji w języku C++.

Ciekawi cię, jak wyglądają implementacje w innych językach programowania? Możesz się z nimi zapoznać w dwóch pozostałych e-materiałach z tej serii:

- Szyfry symetryczne i asymetryczne w języku Java,
- Szyfry symetryczne i asymetryczne w języku Python.

Więcej zadań? Sięgnij do: Szyfry symetryczne i asymetryczne – zadania maturalne.

> **Twoje cele**
>
> - Porównasz implementację szyfru strumieniowego i szyfru blokowego w języku C++.
> - Prześledzisz, jak komputer generuje kolejne liczby pseudolosowe.
> - Przeanalizujesz, dlaczego szyfry asymetryczne wykorzystywane są w komunikacji internetowej.

Przeczytaj

Szyfry symetryczne w języku C++

Szyfry symetryczne dzielimy na dwie grupy: **szyfry strumieniowe** i **szyfry blokowe**.
W szyfrze blokowym wiadomość dzieli się na bloki o określonej długości, a następnie fragmenty te są szyfrowane z użyciem klucza. W szyfrze strumieniowym natomiast klucz używany jest do wygenerowania strumienia bitów, który wykorzystuje się do zaszyfrowania jednego bitu/bajtu informacji na raz. W tym e-materiale zaimplementujemy oba rodzaje szyfru symetrycznego.

Szyfr strumieniowy

Stosując szyfry strumieniowe, należy pamiętać, że okres generowanego strumienia powinien być możliwie długi (termin okres w tym kontekście odnosi się do okresu ciągu). Dlatego ważny jest algorytm generowania strumienia.

W implementacjach szyfru strumieniowego stosowane są najczęściej generatory liczb pseudolosowych, np. funkcja `rand()`, dostępna w języku C++ w bibliotece standardowej.

Przeanalizujemy przykładową implementację szyfru strumieniowego w języku C++. Zaszyfrujemy, a później – w ramach sprawdzenia – odszyfrujemy wybrany ciąg znaków.

Specyfikacja problemu:

Dane:

- `kluczGlowny` – wykorzystywany w procesie szyfrowania klucz, na podstawie którego generowany jest strumień bitów; liczba całkowita
- `slowoDoZaszyfrowania` – tekst jawny przekazany do zakodowania; ciąg znaków

Wynik:

- `zaszyfrowaneSlowo` – zaszyfrowany tekst jawny; ciąg znaków
- `odszyfrowaneSlowo` – odszyfrowany szyfrogram; ciąg znaków

W celu szyfrowania bitów wiadomości możemy posłużyć się różnymi operacjami. Wykorzystamy operator XOR.

> **Ważne!**
>
> W procesie szyfrowania wykorzystamy operator XOR, czyli operator różnicy symetrycznej. W języku C++ odpowiada mu znak `^`, natomiast w zapisie matematycznym wykorzystuje się zazwyczaj symbol $\veebar$.

> Opisywany operator opiera się na następującej logice:
>
> - $0_{(2)} \veebar 0_{(2)} = 0_{(2)}$
> - $0_{(2)} \veebar 1_{(2)} = 1_{(2)}$
> - $1_{(2)} \veebar 0_{(2)} = 1_{(2)}$
> - $1_{(2)} \veebar 1_{(2)} = 0_{(2)}$
>
> Wynikiem operacji $1001_{(2)} \veebar 1111_{(2)}$ będzie $0110_{(2)}$ – należy pamiętać, że operacje wykonujemy tylko na odpowiadających sobie bitach. Warto tutaj dodać, że wynikiem operacji $0110_{(2)} \veebar 1111_{(2)}$będzie $1001_{(2)}$, a wynikiem operacji $0110_{(2)} \veebar 1001_{(2)}$ będzie $1111_{(2)}$.

Do szyfrowania pojedynczych bajtów informacji wykorzystamy funkcję `zaszyfrujZnak()`, która jako parametry przyjmuje zmienne `znak` oraz `klucz`. Zdefiniujemy również tablicę o nazwie `ustring` do przechowywania zmiennych typu `unsigned char`, co pozwoli na zapisywanie wartości w zakresie <0; 255>.

> Domyślnie zmienna typu `basic_string` pozwala przechowywać znaki typu `signed char`, co oznacza, że maksymalną wartością jest 127. Dzięki nim można kodować siedmiobitowy standard ASCII. W omawianym przykładzie szyfrowania potrzebny jest standard Extended ASCII, w którym wartością maksymalną jest 255, czyli ośmiobitowa liczba binarna `11111111`.

```
typedef basic_string<unsigned char> ustring;

unsigned char zaszyfrujZnak(unsigned char znak, unsigned char klu
    return znak ^ klucz;
}
```

Funkcja ta będzie przydatna w implementacji szyfru blokowego i strumieniowego.

> **Przykład 1**
>
> Generator bitów generuje kolejne wyrazy ciągu Fibonacciego. Wiadomością, którą chcemy zaszyfrować, jest `DodaDodaDodatki`. Każdej literze szyfrowanego tekstu odpowiada pewien kod ASCII. Konwertując go na system binarny, dla każdego znaku otrzymamy ciągi składające się z ośmiu bitów.
>
> Znaki wiadomości szyfrujemy kolejnymi wyrazami ciągu Fibonacciego w następujący sposób:
>
> 1. $\texttt{D\^{}0}_{(10)} = \texttt{01000100}_{(2)} \texttt{ \^{} } \texttt{00000000}_{(2)} = \texttt{01000100}_{(2)} = \texttt{D}$
> 2. $\texttt{o\^{}1}_{(10)} = \texttt{01101111}_{(2)} \texttt{ \^{} } \texttt{00000001}_{(2)} = \texttt{01101110}_{(2)} = \texttt{n}$
> 3. $\texttt{d\^{}1}_{(10)} = \texttt{01100100}_{(2)} \texttt{ \^{} } \texttt{00000001}_{(2)} = \texttt{01100101}_{(2)} = \texttt{e}$

4. $a\hat{}2_{(10)} = 01100001_{(2)} \hat{} 00000010_{(2)} = 01100011_{(2)} = c$
5. $D\hat{}3_{(10)} = 01000100_{(2)} \hat{} 00000011_{(2)} = 01000111_{(2)} = G$
6. $o\hat{}5_{(10)} = 01101111_{(2)} \hat{} 00000101_{(2)} = 01101010_{(2)} = j$
7. $d\hat{}8_{(10)} = 01100100_{(2)} \hat{} 00001000_{(2)} = 01101100_{(2)} = l$
8. $a\hat{}13_{(10)} = 01100001_{(2)} \hat{} 00001101_{(2)} = 01101100_{(2)} = l$
9. $D\hat{}21_{(10)} = 01000100_{(2)} \hat{} 00010101_{(2)} = 01010001_{(2)} = Q$
10. $o\hat{}34_{(10)} = 01101111_{(2)} \hat{} 00100010_{(2)} = 01001101_{(2)} = M$
11. $d\hat{}55_{(10)} = 01100100_{(2)} \hat{} 00110111_{(2)} = 01010011_{(2)} = S$
12. $a\hat{}89_{(10)} = 01100001_{(2)} \hat{} 01011001_{(2)} = 00111000_{(2)} = 8$
13. $t\hat{}144_{(10)} = 01110100_{(2)} \hat{} 10010000_{(2)} = 11100100_{(2)} = ń$
14. $k\hat{}233_{(10)} = 01101011_{(2)} \hat{} 11101001_{(2)} = 10000010_{(2)} = é$
15. $i\hat{}377_{(10)} = 01101001_{(2)} \hat{} 101111001_{(2)} = 00010000_{(2)} = 16_{(10)}$
    (wynikiem ostatniej operacji jest liczba dziewięciobitowa, z której odrzucamy najbardziej znaczący bit, otrzymujemy bajt o dziesiętnej wartości 16 – jest to kod znaku kontrolnego bez wizualnej reprezentacji w kodzie ASCII)

Każda litera została zaszyfrowana jako osobny znak, a złamanie szyfrogramu byłoby problematyczne, ponieważ występujące w nim znaki są nieodróżnialne od losowego ciągu symboli.

Proces szyfrowania rozpoczynamy od przygotowania klucza szyfrującego oraz inicjalizacji zmiennej przechowującej tekst jawny ciągiem znaków `DodaDodaDodatki`. Następnie tworzymy kolejną zmienną `zaszyfrowaneSlowo`, która zawierać będzie szyfrogram złożony jedynie z liter „x”.

Ważne!

Funkcja `srand()` jako parametr przyjmuje tak zwane ziarno (z ang. *seed*), czyli pewną wartość, na podstawie której generator będzie obliczał kolejną losową liczbę.
W przypadku analizowanego szyfru strumieniowego ziarno będzie stanowić klucz szyfrujący.

Jeśli spróbujemy użyć funkcji `rand()` bez jawnego wywoływania funkcji `srand()`, to wyniki losowania będą identyczne dla każdego uruchomienia programu, ponieważ funkcja `srand()` wykona się z argumentem 1. Dlatego najczęściej wywołuje się funkcję `srand()` z argumentem `time(NULL)`, który odczytuje liczbę sekund, jaka upłynęła od godziny 00:00:00 1 stycznia 1970 r. – to gwarantuje, że dla każdego uruchomienia programu ziarno będzie unikalne.

Funkcja `srand()` potrzebuje ziarna, ponieważ współczesne komputery są deterministyczne i z poziomu programu nie jesteśmy w stanie wprowadzić prawdziwej losowości.

```
1 int main() {
```

```
    int kluczGlowny = 1234567890; //przykładowy klucz, za pomocą
    srand(kluczGlowny); //ustawienie ziarna i "wyzerowanie genera
    string slowoDoZaszyfrowania = "DodaDodaDodatki";
    string zaszyfrowaneSlowo(slowoDoZaszyfrowania.size(), 'x');
    cout << "Slowo do zaszyfrowania: " << endl;

    for(int i = 0; i < slowoDoZaszyfrowania.size(); i++) {
        cout << slowoDoZaszyfrowania[i];
    }

    cout << endl;
    cout << "Tablica, w ktorej bedzie zaszyfrowane slowo:" << end

    for(int i = 0; i < zaszyfrowaneSlowo.size(); i++) {
        cout << zaszyfrowaneSlowo[i];
    }

    cout << endl;
}
```

Po implementacji opisanych operacji możemy przejść do samego szyfrowania. Wykorzystujemy przygotowaną wcześniej funkcję `zaszyfrujZnak`.

```
for(int i = 0; i < zaszyfrowaneSlowo.size(); i++) {
    zaszyfrowaneSlowo[i] = zaszyfrujZnak(slowoDoZaszyfrowania[i],
}

cout << "Zaszyfrowane slowo:" << endl;

for(int i = 0; i < zaszyfrowaneSlowo.size(); i++) {
    cout << zaszyfrowaneSlowo[i];
}

cout << endl;
```

Dodatkowo program możemy uzupełnić o deszyfrowanie utworzonego szyfrogramu. Ponownie wymagane będzie wykorzystanie funkcji `zaszyfrujZnak`.

```
string odszyfrowaneSlowo(zaszyfrowaneSlowo.size(), 'x');
srand(kluczGlowny); //Ponowne ustawienie generatora liczb pseudol
```

```
3
4  for(int i = 0; i < odszyfrowaneSlowo.size(); i++) {
5      odszyfrowaneSlowo[i] = zaszyfrujZnak(zaszyfrowaneSlowo[i], ra
6  }
7
8  cout << "Odszyfrowane slowo:" << endl;
9
10 for(int i = 0; i < odszyfrowaneSlowo.size(); i++) {
11     cout << odszyfrowaneSlowo[i];
12 }
```

Gotowy program wygląda następująco:

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <string>
4
5  using namespace std;
6
7  typedef basic_string<unsigned char> ustring;
8
9  unsigned char zaszyfrujZnak(unsigned char znak, unsigned char klu
10     return znak ^ klucz;
11 }
12
13 int main() {
14     int kluczGlowny = 1234567890; //przykładowy klucz, za pomocą
15     srand(kluczGlowny); //ustawienie ziarna i "wyzerowanie genera
16     string slowoDoZaszyfrowania = "DodaDodaDodatki";
17     ustring zaszyfrowaneSlowo(slowoDoZaszyfrowania.size(), 'x');
18     cout << "Slowo do zaszyfrowania: " << endl;
19
20     for(int i = 0; i < slowoDoZaszyfrowania.size(); i++) {
21         cout << slowoDoZaszyfrowania[i];
22     }
23
24     cout << endl;
25     cout << "Tablica, w ktorej bedzie zaszyfrowane slowo:" << end
26
27     for(int i = 0; i < zaszyfrowaneSlowo.size(); i++) {
28         cout << zaszyfrowaneSlowo[i];
```

```
    }

    cout << endl;

    for(int i = 0; i < zaszyfrowaneSlowo.size(); i++) {
        zaszyfrowaneSlowo[i] = zaszyfrujZnak(slowoDoZaszyfrowania
    }

    cout << "Zaszyfrowane slowo:" << endl;

    for(int i = 0; i < zaszyfrowaneSlowo.size(); i++) {
        cout << zaszyfrowaneSlowo[i];
    }

    cout << endl;
    ustring odszyfrowaneSlowo(zaszyfrowaneSlowo.size(), 'x');
    srand(kluczGlowny); //Ponowne ustawienie generatora liczb pse

    for(int i = 0; i < odszyfrowaneSlowo.size(); i++) {
        odszyfrowaneSlowo[i] = zaszyfrujZnak(zaszyfrowaneSlowo[i]
    }

    cout << "Odszyfrowane slowo:" << endl;

    for(int i = 0; i < odszyfrowaneSlowo.size(); i++) {
        cout << odszyfrowaneSlowo[i];
    }

    return 0;
}
```

A oto przykładowy wynik pracy programu:

```
Slowo do zaszyfrowania:
DodaDodaDodatki
Tablica, w ktorej będzie zaszyfrowane slowo:
xxxxxxxxxxxxxxx
Zaszyfrowane slowo:
Ů%ďj╠╣5■ýN├Uą5
Odszyfrowane slowo:
DodaDodaDodatki
```

W przedstawionym przykładzie szyfrowaliśmy całe bajty informacji. Przypomnijmy, że każdy znak w Extended ASCII kodowany jest na jednym bajcie, czyli na zmiennej typu `char`.

Warto zwrócić uwagę, że choć w szyfrowanej wiadomości występują:

- trzy znaki `D`,
- trzy znaki `o`,
- trzy znaki `d`,
- trzy znaki `a`,
- jeden znak `t`,
- jeden znak `k`,
- jeden znak `i`,

nie ma to jednak przełożenia na zaszyfrowaną wiadomość – wszystkie znaki występujące w szyfrogramie wydają się losowe.

Otrzymany ciąg znaków może różnić się w zależności od użytego standardu dekodowania. Ten z kolei zależy między innymi od wykorzystywanego systemu operacyjnego. Jeżeli w ramach swojej implementacji otrzymasz inne rozwiązanie, nie oznacza to, że jest ono błędne. W przypadku wątpliwości warto sprawdzić uzyskane kody w systemie dziesiętnym.

Szyfr blokowy

Szyfrowanie blokowe zaprezentujemy z wykorzystaniem trybu szyfrowania ECB (ang. *Electronic Codebook*). Tryb ten polega na szyfrowaniu po kolei bloków tekstu jawnego w kolejne bloki szyfrogramu. W ramach przedstawionej implementacji najpierw zaszyfrujemy, a później odszyfrujemy wybrany ciąg znaków.

Specyfikacja problemu:

Dane:

- `kluczGlowny` – klucz wykorzystywany w procesie szyfrowania; tablica trzech liczb całkowitych (składających się na klucz), które można przedstawić na ośmiu bitach
- `slowoDoZaszyfrowania` – tekst jawny przekazany do zakodowania; ciąg znaków

Wynik:

- `zaszyfrowaneSlowo` – zaszyfrowany tekst jawny; ciąg znaków
- `odszyfrowaneSlowo` – odszyfrowany szyfrogram; ciąg znaków

Podczas implementacji algorytmu dzielimy tekst jawny na bloki. Następnie każdy z bloków szyfrujemy kluczem, dzięki operatorowi XOR.

> **Przykład 2**

Mamy przekazać wiadomość `DodaDodaDodatki`. Podczas szyfrowania znaków posługujemy się kodem Extended ASCII.
Kluczem są kody ASCII trzech znaków: a, b, c, co oznacza, że blok będzie miał długość 24 bitów, a klucz w postaci binarnej będzie
następujący: $01100001\ 01100010\ 01100011_{(2)}$. Należy więc podzielić wiadomość na pięć bloków, zawierających po trzy znaki:

`Dod`, `aDo`, `daD`, `oda`, `tki`.

Operacja XOR polega na porównywaniu bitów dwóch liczb binarnych, a następnie ustaleniu na ich podstawie wartości bitów w liczbie wynikowej. Opisaną operację będziemy wykonywać jednocześnie na dwóch pojedynczych znakach: znaku bloku oraz znaku klucza, przekonwertowanych na wartości binarne.

$$
\begin{aligned}
D \wedge a &= 01000100_{(2)} \wedge 01100001_{(2)} = 00100110_{(2)} \\
o \wedge b &= 01101111_{(2)} \wedge 01100010_{(2)} = 00001110_{(2)} \\
d \wedge c &= 01100100_{(2)} \wedge 01100011_{(2)} = 00000110_{(2)} \\
a \wedge a &= 01100001_{(2)} \wedge 01100001_{(2)} = 00000000_{(2)} \\
D \wedge b &= 01000100_{(2)} \wedge 01100010_{(2)} = 00100110_{(2)} \\
o \wedge c &= 01101111_{(2)} \wedge 01100011_{(2)} = 00001110_{(2)} \\
d \wedge a &= 01100100_{(2)} \wedge 01100001_{(2)} = 00000110_{(2)} \\
a \wedge b &= 01100001_{(2)} \wedge 01100010_{(2)} = 00000000_{(2)} \\
D \wedge c &= 01000100_{(2)} \wedge 01100011_{(2)} = 00100110_{(2)} \\
o \wedge a &= 01101111_{(2)} \wedge 01100001_{(2)} = 00001110_{(2)} \\
d \wedge b &= 01100100_{(2)} \wedge 01100010_{(2)} = 00000110_{(2)} \\
a \wedge c &= 01100001_{(2)} \wedge 01100011_{(2)} = 00000000_{(2)} \\
t \wedge a &= 01110100_{(2)} \wedge 01100001_{(2)} = 00010110_{(2)} \\
k \wedge b &= 01101011_{(2)} \wedge 01100010_{(2)} = 00001010_{(2)} \\
i \wedge c &= 01101001_{(2)} \wedge 01100011_{(2)} = 00001011_{(2)}
\end{aligned}
$$

W tym przypadku nie zapiszemy ich odpowiedników w kodzie ASCII, ponieważ większość z uzyskanych znaków ich nie ma.

Warto jednak zwrócić uwagę, że symbole są za każdym razem szyfrowane przez inny fragment kodu.

Jak pokazaliśmy w przykładzie, szyfr blokowy jest w stanie zaszyfrować jeden znak na wiele różnych znaków. Jednak aby tak się stało, klucz musi być dłuższy od jednego bajta.

Istnieje również możliwość szyfrowania blokowego z użyciem klucza o liczbie bitów niepodzielnej przez osiem. Jego implementacja jest jednak znacznie bardziej skomplikowana i nie będziemy się nią teraz zajmować.

Przykład 3

> Zaszyfrujmy ponownie wiadomość: `DodaDodaDodatki`, tym razem używając jako klucza wartości `234`, która mieści się na jednym bajcie.
>
> $D \text{ ^ } 234_{(10)} = 01000100_{(2)} \text{ ^ } 11101010_{(2)} = 10101110_{(2)} = «$
> $o \text{ ^ } 234_{(10)} = 01101111_{(2)} \text{ ^ } 11101010_{(2)} = 10000101_{(2)} = ů$
> $d \text{ ^ } 234_{(10)} = 01100100_{(2)} \text{ ^ } 11101010_{(2)} = 10001110_{(2)} = Ä$
> $a \text{ ^ } 234_{(10)} = 01100001_{(2)} \text{ ^ } 11101010_{(2)} = 10001011_{(2)} = ő$
> $D \text{ ^ } 234_{(10)} = 01000100_{(2)} \text{ ^ } 11101010_{(2)} = 10101110_{(2)} = «$
> $o \text{ ^ } 234_{(10)} = 01101111_{(2)} \text{ ^ } 11101010_{(2)} = 10000101_{(2)} = ů$
> $d \text{ ^ } 234_{(10)} = 01100100_{(2)} \text{ ^ } 11101010_{(2)} = 10001110_{(2)} = Ä$
> $a \text{ ^ } 234_{(10)} = 01100001_{(2)} \text{ ^ } 11101010_{(2)} = 10001011_{(2)} = ő$
> $D \text{ ^ } 234_{(10)} = 01000100_{(2)} \text{ ^ } 11101010_{(2)} = 10101110_{(2)} = «$
> $o \text{ ^ } 234_{(10)} = 01101111_{(2)} \text{ ^ } 11101010_{(2)} = 10000101_{(2)} = ů$
> $d \text{ ^ } 234_{(10)} = 01100100_{(2)} \text{ ^ } 11101010_{(2)} = 10001110_{(2)} = Ä$
> $a \text{ ^ } 234_{(10)} = 01100001_{(2)} \text{ ^ } 11101010_{(2)} = 10001011_{(2)} = ő$
> $t \text{ ^ } 234_{(10)} = 01110100_{(2)} \text{ ^ } 11101010_{(2)} = 10011110_{(2)} = ×$
> $k \text{ ^ } 234_{(10)} = 01101011_{(2)} \text{ ^ } 11101010_{(2)} = 10000001_{(2)} = ü$
> $i \text{ ^ } 234_{(10)} = 01101001_{(2)} \text{ ^ } 11101010_{(2)} = 10000011_{(2)} = â$
>
> W wyniku otrzymaliśmy następującą kombinację symboli szyfrogramu:
>
> `«ůÄő«ůÄő«ůÄő×üâ`
>
> odpowiadającą wiadomości:
>
> `DodaDodaDodatki`.
>
> Po przeanalizowaniu przykładu możemy stwierdzić, że każdemu znakowi tekstu jawnego odpowiada tylko jeden konkretny znak, co powoduje, że szyfr ten jest – przynajmniej teoretycznie – prostszy do złamania i nie zapewnia praktycznie żadnego bezpieczeństwa.

Po analizie szyfrogramu z przykładu można zauważyć, że w wiadomości występuje kilka powtarzających się znaków. Podobnego efektu nie uzyskaliśmy w szyfrze strumieniowym oraz w szyfrze blokowym, wykorzystującym klucz dłuższy niż jeden bajt.

Trzy bajty klucza (przykład nr 2) w przypadku wiadomości wystarczyły do bezpiecznego jej zaszyfrowania. Przy szyfrowaniu dłuższych tekstów mogłoby się jednak okazać, że wybrany klucz jest zbyt krótki, by zapewnić pożądany poziom bezpieczeństwa (prawdopodobieństwo złamania szyfru byłoby wysokie).

ECB jest najprostszym koncepcyjnie algorytmem blokowym, którego jednak z tego właśnie powodu nie używa się w praktycznych zastosowaniach.

Przedstawiony program jest implementacją szyfru blokowego, opartego na szyfrowaniu tekstu jawnego kluczem o długości trzech bajtów (rozdzielonym na trzy liczby całkowite, gdzie każdą z nich możemy przedstawić za pomocą ośmiu bitów). Działa on podobnie do przedstawionego wcześniej algorytmu szyfrowania strumieniowego, z tą różnicą, że każdy blok wiadomości szyfrowany jest tym samym kluczem (a nie losowym strumieniem danych).

```
#include <iostream>
#include <cstdlib>
#include <string>

using namespace std;

typedef basic_string<unsigned char> ustring;

unsigned char zaszyfrujZnak(unsigned char znak, int klucz) {
    return znak ^ klucz;
}

int main() {
    int kluczGlowny[3] = {234, 255, 165}; //przykładowy klucz, za
    string slowoDoZaszyfrowania = "DodaDodaDodatki";
    ustring zaszyfrowaneSlowo(slowoDoZaszyfrowania.size(), 'x');
    cout << "Slowo do zaszyfrowania: " << endl;

    for(int i = 0; i < slowoDoZaszyfrowania.size(); i++) {
        cout << slowoDoZaszyfrowania[i];
    }

    cout << endl;
    cout << "Tablica, w ktorej bedzie zaszyfrowane slowo:" << end

    for(int i = 0; i < zaszyfrowaneSlowo.size(); i++) {
        cout << zaszyfrowaneSlowo[i];
    }

    cout << endl;

    for(int i = 0; i < zaszyfrowaneSlowo.size(); i++) {
        zaszyfrowaneSlowo[i] = zaszyfrujZnak(slowoDoZaszyfrowania
    }

```

```
    cout << "Zaszyfrowane slowo:" << endl;

    for(int i = 0; i < zaszyfrowaneSlowo.size(); i++){
        cout << zaszyfrowaneSlowo[i];
    }

    cout << endl;
    ustring odszyfrowaneSlowo(zaszyfrowaneSlowo.size(), 'x');

    for(int i = 0; i < odszyfrowaneSlowo.size(); i++) {
        odszyfrowaneSlowo[i] = zaszyfrujZnak(zaszyfrowaneSlowo[i]
    }

    cout << "Odszyfrowane slowo:" << endl;

    for(int i = 0; i < odszyfrowaneSlowo.size(); i++) {
        cout << odszyfrowaneSlowo[i];
    }

    return 0;
}
```

Program na wyjściu standardowym wypisze:

```
Slowo do zaszyfrowania:
DodaDodaDodatki
Tablica, w ktorej bedzie zaszyfrowane slowo:
xxxxxxxxxxxxxxx
Zaszyfrowane slowo:
«É┴ő╖╩Ä×ßůŤ—×ö╠
Odszyfrowane slowo:
DodaDodaDodatki
```

Podobnie jak w przypadku szyfru strumieniowego, otrzymany wynik może się różnić w zależności od przyjętego typu kodowania.

> **Dla zainteresowanych**
>
> Wyszukaj w internecie frazy „ECB Penguin” – czyli przykładu obrazka linuksowego pingwina, który został zaszyfrowany za pomocą ECB. Jest to świetny dowód ilustrujący, dlaczego szyfr ten jest nieskuteczny.

Szyfr asymetryczny

Omówimy koncepcję szyfrów asymetrycznych, korzystając z protokołu Diffiego-Hellmana. Jest on używany w protokole TLS/SSL, służącym do szyfrowania wiadomości wysyłanych drogą internetową, w celu asymetrycznego negocjowania sekretnego klucza.

W trakcie omawiania działania algorytmu użyjemy imion: Alicja i Bob – są to fikcyjne postacie, przywoływane podczas omawiania wielu algorytmów asymetrycznych. Z tymi imionami skojarzone są role dwóch osób, które chcą wymienić się wiadomością lub kluczem.

Algorytm można uprościć do następującej listy kroków:

1. Użytkownicy uzgadniają liczbę pierwszą p i bazę g – są to elementy publiczne i teoretycznie każdy może mieć do nich dostęp, co dla bezpieczeństwa szyfru nie stanowi żadnego problemu.
2. Alicja wybiera liczbę całkowitą a, a następnie przesyła Bobowi liczbę: $A = g^a \bmod p$.
3. Bob wykonuje identyczną operację i również wybiera dowolną liczbę całkowitą b, a następnie wysyła liczbę $B = g^b \bmod p$.

> **Ważne!**
>
> Liczby a oraz b są tajne i znane odpowiednio jedynie Alicji i Bobowi.

4. Następnie Alicja oblicza liczbę $s = B^a \bmod p$. Bob oblicza tę samą liczbę: $s = A^b \bmod p$.

Liczba s jest wynegocjowanym sekretnym kluczem (znanym Alicji i Bobowi), który może być następnie wykorzystany do szyfrowania symetrycznego. Ponieważ szyfrowanie asymetryczne jest zbyt zasobo- i czasochłonne, wykorzystamy jedynie asymetryczny schemat do negocjacji klucza, używanego do szyfrowania symetrycznego. Możemy więc stwierdzić, że używamy schematów asymetrycznych do wynegocjowania klucza, aby był on tajny. Ale sam klucz stosujemy w algorytmie symetrycznym, aby ograniczyć czas procesu szyfrowania.

Równe wartości wyznaczonych tajnych kluczy możemy udowodnić matematycznie. Kluczowy jest fakt, że dzielenie $(g^a)^b$ i $\left(g^b\right)^a$ modulo p zwróci w obu przypadkach tę samą wartość.

Alicja odbierająca wiadomość ma klucz publiczny składający się z trzech elementów: p, g oraz $g^a \bmod p$. Dysponuje ona również kluczem prywatnym a oraz wyznaczonym tajnym kluczem s.
Bob wysyłający wiadomość używa klucza publicznego, składającego się z trzech elementów: p, g oraz $g^b \bmod p$. Ma on również klucz prywatny b oraz wyznaczony tajny klucz s.

Przyjrzyjmy się implementacji opisanego algorytmu.

Specyfikacja problemu:

Dane:

- `p` – pierwszy element klucza publicznego Alicji; liczba pierwsza
- `g` – drugi element klucza publicznego Alicji – baza; liczba całkowita
- `a` – klucz prywatny Alicji; liczba całkowita
- `b` – klucz prywatny Boba; liczba całkowita

Wynik:

- `s` – obliczony sekretny klucz; liczba całkowita

Program rozpoczyna się od utworzenia trzech zmiennych: `p`, `g` oraz `a`, przechowujących opisane wcześniej wartości, czyli odpowiednio: liczbę pierwszą, bazę oraz liczbę całkowitą wybraną przez Alicję. Następnie obliczamy wartość zmiennej `A` według wzoru przedstawionego w drugim kroku przebiegu algorytmu.

> **Ważne!**
>
> Operacja modulo (w języku C++ reprezentowana za pomocą znaku `%`) jest operacją wyznaczania reszty.
> Przykładowo: `21 % 2 = 1`, ponieważ `21 / 2 = 10`, reszty `1`.

```cpp
#include <iostream>
#include <cmath>

using namespace std;

int main(int argc, char** argv) {
    int p = 29; //element klucza publicznego Alicji
    int g = 9; //element klucza publicznego Alicji
    int a = 7; //klucz prywatny Alicji
    unsigned int A = ((unsigned int)pow(g, a)) % p; //element klu

    cout << "A Alicji wynosi: "<< A << endl;
}
```

Następnie tworzymy kolejną zmienną `b`, która przechowywać będzie liczbę całkowitą wybraną przez Boba. Wyliczamy także wartość zmiennej `B` (wzór w kroku trzecim opisu algorytmu).

```
int b = 3; //klucz prywatny Boba
unsigned int B = ((unsigned int)pow(g, b)) % p; //element klucza

cout << "B Boba wynosi: "<< B << endl;
```

Ostatnim krokiem algorytmu jest wyznaczenie wartości zmiennej `s`, zarówno według wzoru wykorzystywanego przez Alicję, jak i przez Boba.

```
unsigned int s = ((unsigned int)pow(B, a)) % p; //Alicja oblicza

cout << "s obliczone przez Alicje wynosi: " << s <<endl;

s = ((unsigned int)pow(A, b)) % p; //Bob oblicza s

cout << "s obliczone przez Boba wynosi: " << s << endl;
```

Gotowy program wygląda następująco:

```
#include <iostream>
#include <cmath>

using namespace std;

int main(int argc, char** argv) {
    int p = 29; //element klucza publicznego Alicji
    int g = 9; //element klucza publicznego Alicji
    int a = 7; //klucz prywatny Alicji
    unsigned int A = ((unsigned int)pow(g, a)) % p; //element klu

    cout << "A Alicji wynosi: "<< A << endl;

    int b = 3; //klucz prywatny Boba
    unsigned int B = ((unsigned int)pow(g, b)) % p; //element klu

    cout << "B Boba wynosi: "<< B << endl;

    unsigned int s = ((unsigned int)pow(B, a)) % p; //Alicja obli

    cout << "s obliczone przez Alicje wynosi: " << s <<endl;

```

```
    s = ((unsigned int)pow(A, b)) % p; //Bob oblicza s

    cout << "s obliczone przez Boba wynosi: " << s << endl;

    return 0;
}
```

Wynikiem działania programu byłoby wypisanie następujących wiadomości:

```
A Alicji wynosi: 28
B Boba wynosi: 4
s obliczone przez Alicje wynosi: 28
s obliczone przez Boba wynosi: 28
```

> **Ważne!**
>
> Ze względu na sposób typowania zmiennych w języku C++ szyfrowanie asymetryczne nie jest proste do zaimplementowania, a na potrzeby tego e-materiału korzystaliśmy z małych wartości `p`, `g`, `a` i `b`. Duże wartości, wykorzystywane w realnych sytuacjach, doprowadziłyby do przekroczenia zakresu zmiennej typu `unsigned int`.

Słownik

> **determinizm**
>
> algorytm deterministyczny jest całkowicie określony przez warunki początkowe (parametry); oznacza to, że kilkukrotne uruchomienie algorytmu doprowadzi za każdym razem do takiego samego wyniku
> **generator liczb pseudolosowych**
>
> funkcja (lub program) generująca na podstawie ziarna (z ang. *seed*) deterministyczny ciąg bitów, który wydaje się losowy
> **okres ciągu**
>
> liczba k taka, że dla wszystkich indeksów n prawdziwe jest stwierdzenie, że $a_n = a_{n+k}$ – oznacza to, że wartości ciągu powtarzają się co k pozycji

Prezentacja multimedialna

Polecenie 1

Zapoznaj się z prezentacją multimedialną. Poszukaj informacji na temat wykorzystania omówionych w niej szyfrów.

Polecenie 2

Poszukaj informacji na temat metod łamania przedstawionych w prezentacji szyfrów. Który z nich jest najtrudniejszy do złamania?

1

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCpIj6rj

Znasz już zasadę działania algorytmu ECB oraz protokołu Diffiego-Hellmana. Potrafisz również wymienić różnice między kluczem prywatnym oraz publicznym. Wiesz też, na czym polega działanie szyfru blokowego, asymetrycznego oraz strumieniowego. Przygotujmy po jednym przykładzie wymienionych szyfrów i porównajmy ich skuteczność.

2

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCpIj6rj

Najpierw zaimplementujmy funkcję, która będzie służyła do kodowania pojedynczych znaków. Funkcję tę możemy wykorzystywać we wszystkich trzech szyfrowaniach.

```
unsigned char zaszyfrujZnak(unsigned char znak, unsigned char klucz)
{
    return znak^klucz;
}
```

3

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Znak ^ jest operatorem XOR w języku C++.

A	B	XOR
1	0	PRAWDA
1	1	FAŁSZ
0	0	FAŁSZ
0	1	PRAWDA

4

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Najpierw implementujemy szyfr blokowy.

```

string szyfrBlokowy(string kodowanyCiagZnakow, unsigned char klucz)
{
    for (int i = 0; i < kodowanyCiagZnakow.size(); i++)
    {
```

```
6
    kodowanyCiagZnakow[i] =
    zaszyfrujZnak(kodowanyCiag
    Znakow[i], klucz);
7       }
8       return
    kodowanyCiagZnakow;
9   }
10
```

Każdy ciąg znaków zostaje zakodowany podanym kluczem.

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

5

Przetestujmy ten kod.

```
1
2   #include <iostream>
3   using namespace std;
4
5   unsigned char
    zaszyfrujZnak(unsigned
    char znak, unsigned char
    klucz)
6   {
7       return znak ^ klucz;
8   }
9
10  string szyfrBlokowy(string
    kodowanyCiagZnakow,
    unsigned char klucz)
11  {
12      for (int i = 0; i <
    kodowanyCiagZnakow.size();
    i++)
13      {
14
    kodowanyCiagZnakow[i] =
```

```
    zaszyfrujZnak(kodowanyCiag
    Znakow[i], klucz);
15      }
16      return
    kodowanyCiagZnakow;
17  }
18  int main()
19  {
20      string ciagZnakow =
    "MaturaZInformatyki";
21      ciagZnakow =
    szyfrBlokowy(ciagZnakow,
    300);
22      cout << ciagZnakow;
23  }
24
```

6

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Wartość zmiennej `klucz` została podana jedynie jako przykład. W zaprezentowanej implementacji nie ma sensu podawać większych jej wartości, ponieważ kodujemy pojedyncze znaki, które zapisują się na maksymalnie jednym bajcie. Warto zaznaczyć, że w rzeczywistych zastosowaniach klucze mogą być wykorzystywane do kodowania całych słów.

7

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

W przypadku tego szyfrowania, jeżeli znamy kod, za którego pomocą wiadomość była zaszyfrowana, możemy użyć tego samego kodu

oraz funkcji szyfrującej do zdeszyfrowania wiadomości.

```
int main()
{
    string ciagZnakow = "MaturaZInformatyki";
    ciagZnakow = szyfrBlokowy(ciagZnakow, 300);
    cout << ciagZnakow << endl;
    ciagZnakow = szyfrBlokowy(ciagZnakow, 300);
    cout << ciagZnakow << endl;
}

```

8

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Funkcje szyfrująca i deszyfrująca mają taką samą postać, ponieważ podwójne wykonanie operacji XOR na danym bicie tym samym bitem powoduje odwrócenie operacji.

Wartość bitu w pamięci	Drugi argument operacji XOR	Wynik operacji
1	1	0
0	1	**1**

Materiał audio dostępny pod adresem:

9

https://zpe.gov.pl/b/PMCplj6rj

Szyfr strumieniowy i szyfr blokowy są do siebie pod pewnymi względami podobne. Różni je głównie fakt, że w szyfrze blokowym szyfrujemy kluczem poszczególne części wiadomości, a w szyfrze strumieniowym klucz używany jest do wygenerowania bitów, które posłużą do szyfrowania. W języku C++ stosujemy funkcję `srand()`. Jej argumentem jest wartość (tak zwane ziarno), na podstawie której generator liczb pseudolosowych, dostępny pod funkcją `rand()`, generuje kolejne losowe liczby.

```
srand(300);
rand();
```

10

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Wartość 300 jest w omawianym wypadku kluczem. Warto dodać, że używamy tutaj domyślnych funkcji losujących jedynie w ramach uproszczenia, w rozbudowanym programie wielowątkowym napisalibyśmy własną funkcję generującą liczby na podstawie klucza.

11

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Szyfr strumieniowy w swojej najprostszej postaci jest podobny do szyfru blokowego.

```
1  string szyfrBlokowy(string
   kodowanyCiagZnakow,
   unsigned char klucz)
2  {
3      for (int i = 0; i <
   kodowanyCiagZnakow.size();
   i++)
4      {
5
   kodowanyCiagZnakow[i] =
   zaszyfrujZnak(kodowanyCiag
   Znakow[i], klucz);
6      }
7      return
   kodowanyCiagZnakow;
8  }
9  string
   szyfrStrumieniowy(string
   kodowanyCiagZnakow,
   unsigned char klucz)
10 {
11     srand(klucz);
12     for (int i = 0; i <
   kodowanyCiagZnakow.size();
   i++)
13     {
14
   kodowanyCiagZnakow[i] =
   zaszyfrujZnak(kodowanyCiag
   Znakow[i], rand());
15     }
16     return
   kodowanyCiagZnakow;
17 }
```

12

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Jednak rezultaty ich działania znacząco się różnią.

Wynik szyfru blokowego:

```
aMXY^MveBJC^AMXUGE
```

Wynik szyfru strumieniowego:

```
űŔ)  ťű┤«ç-%°×˘T#ŕ
```

13

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Możemy zauważyć, że w szyfrze blokowym każdy znak został zakodowany w sposób jednoznaczny, przez co jest możliwy do złamania metodą analizy częstotliwości liter dla dłuższych wiadomości.

Każda litera jest bowiem szyfrowana w jeden konkretny sposób, przynajmniej dla uproszczonego kodowania ECB, które tutaj zastosowaliśmy. Dla kodów blokowych, szyfrujących kilkadziesiąt bajtów na raz, byłoby to dużo trudniejsze, ale również możliwe.

Natomiast w przypadku szyfru strumieniowego nie ma takiej możliwości, o ile cykl generatora bitów jest wystarczająco długi.

14

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Cały kod szyfru strumieniowego możemy zapisać w następujący sposób:

```
1 #include <iostream>
2 using namespace std;
3
```

```
4  unsigned char
   zaszyfrujZnak(unsigned
   char znak, unsigned char
   klucz)
5  {
6      return znak ^ klucz;
7  }
8
9  string
   szyfrStrumieniowy(string
   kodowanyCiagZnakow,
   unsigned char klucz)
10 {
11     srand(klucz);
12     for (int i = 0; i <
   kodowanyCiagZnakow.size();
   i++)
13     {
14
   kodowanyCiagZnakow[i] =
   zaszyfrujZnak(kodowanyCiag
   Znakow[i], rand());
15     }
16     return
   kodowanyCiagZnakow;
17 }
18 int main()
19 {
20     string ciagZnakow =
   "MaturaZInformatyki";
21     ciagZnakow =
   szyfrStrumieniowy(ciagZnak
   ow, 300);
22     cout << "Wynik szyfru
   strumieniowego:" << endl
   << ciagZnakow << endl;
23     ciagZnakow =
   szyfrStrumieniowy(ciagZnak
   ow, 300);
24     cout << "Po ponownym
   wykorzystaniu funkcji,
   wiadomosc wraca do postaci
25 pierwotnej" << endl <<
   ciagZnakow << endl;
26 }
```

15

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCpIj6rj

Szyfr asymetryczny różni się od szyfrów omówionych wcześniej. Cechuje go występowanie dwóch kluczy – publicznego i prywatnego. Używa się ich podczas komunikacji internetowej, ponieważ teoretycznie jedynie właściwy odbiorca wiadomości powinien być w stanie daną wiadomość odczytać. Oczywiście kryptografia jest rozwijającą się dziedziną, która pokonuje kolejne szyfry, więc to, czy dany szyfr jest złamany czy nie, może zmieniać się z roku na rok. Na potrzeby tego e-materiału skorzystamy z protokołu Diffiego-Hellmana, który pozwala bezpiecznie ustalić wspólny klucz do przesyłania sobie informacji. Warto jednak dodać, że istnieją inne szyfry asymetryczne, jak RSA, które służą bezpośrednio do szyfrowania i deszyfrowania wiadomości, a nie jedynie do ustalania wspólnego klucza.

16

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCpIj6rj

Robert i Alicja muszą uzgodnić dwie liczby: liczbę pierwszą p i bazę g. Liczby te są powszechnie znane.

Znane potencjalnie wszystkim użytkownikom sieci	**Znane jedynie Alicji**	**Znane jedynie Robertowi**

Znane potencjalnie wszystkim użytkownikom sieci	Znane jedynie Alicji	Znane jedynie Robertowi
p = 509		
g = 17		

```
1 long long p = 509;
2 long long g = 17;
```

17

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCpIj6rj

Załóżmy, że p = 509 i g = 17 są relatywnie małymi liczbami. Wybraliśmy je, ponieważ bez wprowadzania dodatkowych typów danych w języku C++ niemożliwe jest operowanie na dużo większych potęgach. Z problemem tym możemy sobie do pewnego stopnia poradzić, stosując zamiast `int` typ danych `long long`. Jednak nawet to rozwiązanie wiąże się z ograniczeniami.

18

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCpIj6rj

W kolejnym kroku algorytmu obydwoje bohaterów wybiera znane jedynie sobie dwie liczby: a dla Alicji i b dla Roberta.

Znane potencjalnie wszystkim użytkownikom sieci	Znane jedynie Alicji	Znane jedynie Robertowi
p = 509	a = 7	b = 3
g = 17		

Aktualizuje się też kawałek kodu:

```
long long p = 509;
long long g = 17;
long long a = 7;
long long b = 3;
```

19

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCpIj6rj

Następnie Alicja oblicza `A`, a Robert oblicza B, zgodnie ze wzorem g^n `modulo p`, gdzie n to a dla Alicji i b dla Roberta.

Znane potencjalnie wszystkim użytkownikom sieci	Znane jedynie Alicji	Znane jedynie Robertowi
p = 509	a = 7	b = 3
g = 17	A = 179	B = 332

```
long long p = 509;
long long g = 17;
long long a = 7;
long long b = 3;
unsigned long long A =
((unsigned long
long)pow(g, a)) % p;
```

```
6 unsigned long long B =
  ((unsigned long
  long)pow(g, b)) % p;
```

20

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Robert i Alicja przesyłają sobie A oraz B. Warto zaznaczyć, że może dojść do przechwycenia tych wartości przez osobę nieupoważnioną, nie jest to jednak problem dla bezpieczeństwa szyfrowania.

Znane potencjalnie wszystkim użytkownikom sieci	**Znane jedynie Alicji**	**Znane jedynie Robertowi**
p = 509	a = 7	b = 3
g = 17		
A = 179		
B = 332		

21

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Teraz Robert i Alicja mogą obliczyć s na podstawie przesłanych sobie wzajemnie A i B.

Alicja obliczy s zgodnie ze wzorem: B^a `modulo` p.

Robert obliczy s zgodnie ze wzorem: A^b `modulo p`.

22

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Podstawmy do wzorów wartości liczbowe i zweryfikujmy ich poprawność.

23

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Alicja obliczy s zgodnie ze wzorem: 332^7 `modulo 509 = 436.`

Robert obliczy s zgodnie ze wzorem: 179^3 `modulo 509 = 436.`

24

Znane potencjalnie wszystkim użytkownikom sieci	**Znane jedynie Alicji**	**Znane jedynie Robertowi**	**Znane Alicji i Robertowi**
p = 509	a = 7	b = 3	s = 436
g = 17			
A = 179			
B = 332			

25

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Całość kodu, pomijając przesyłanie informacji, wyglądała w następujący sposób:

```
#include <iostream>
#include <stdlib.h>
#include <string>
#include <math.h>
using namespace std;
int main(int argc, char**
argv)
{
    long long p = 509;
    long long g = 17;
    long long a = 7;
    long long b = 3;
    unsigned long long A =
((unsigned long
long)pow(g, a)) % p;
    unsigned long long B =
((unsigned long
long)pow(g, b)) % p;
    unsigned long long
sAlicji = ((unsigned long
long)pow(B, a)) % p;
    unsigned long long
sRoberta = ((unsigned long
long)pow(A, b)) % p;
    cout << "s Alicji to:
" << sAlicji << endl << "s
Roberta to: " << sRoberta;
    return 0;
```

26

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PMCplj6rj

Alicja i Robert ustalili klucz s, którym będą szyfrowali wiadomości przesyłane między sobą. Klucz ten może zostać użyty w szyfrze

blokowym, strumieniowym lub w dowolnym innym szyfrze.

Sprawdź się

Pokaż ćwiczenia:

Ćwiczenie 1

Napisz program szyfrujący podaną wiadomość, wykorzystując szyfr blokowy typu ECB oraz klucz o długości trzech bajtów. Działanie swojego programu przetestuj dla wiadomości `Animos labor nutrit` oraz dla klucza składającego się z trzech liczb całkowitych: 12, 24 oraz 31.

Specyfikacja problemu:

Dane:

- `kluczGlowny` – informacja wykorzystywana do szyfrowania i deszyfrowania podanej wiadomości; tablica trzech liczb całkowitych, gdzie każdą z nich można przedstawić na ośmiu bitach

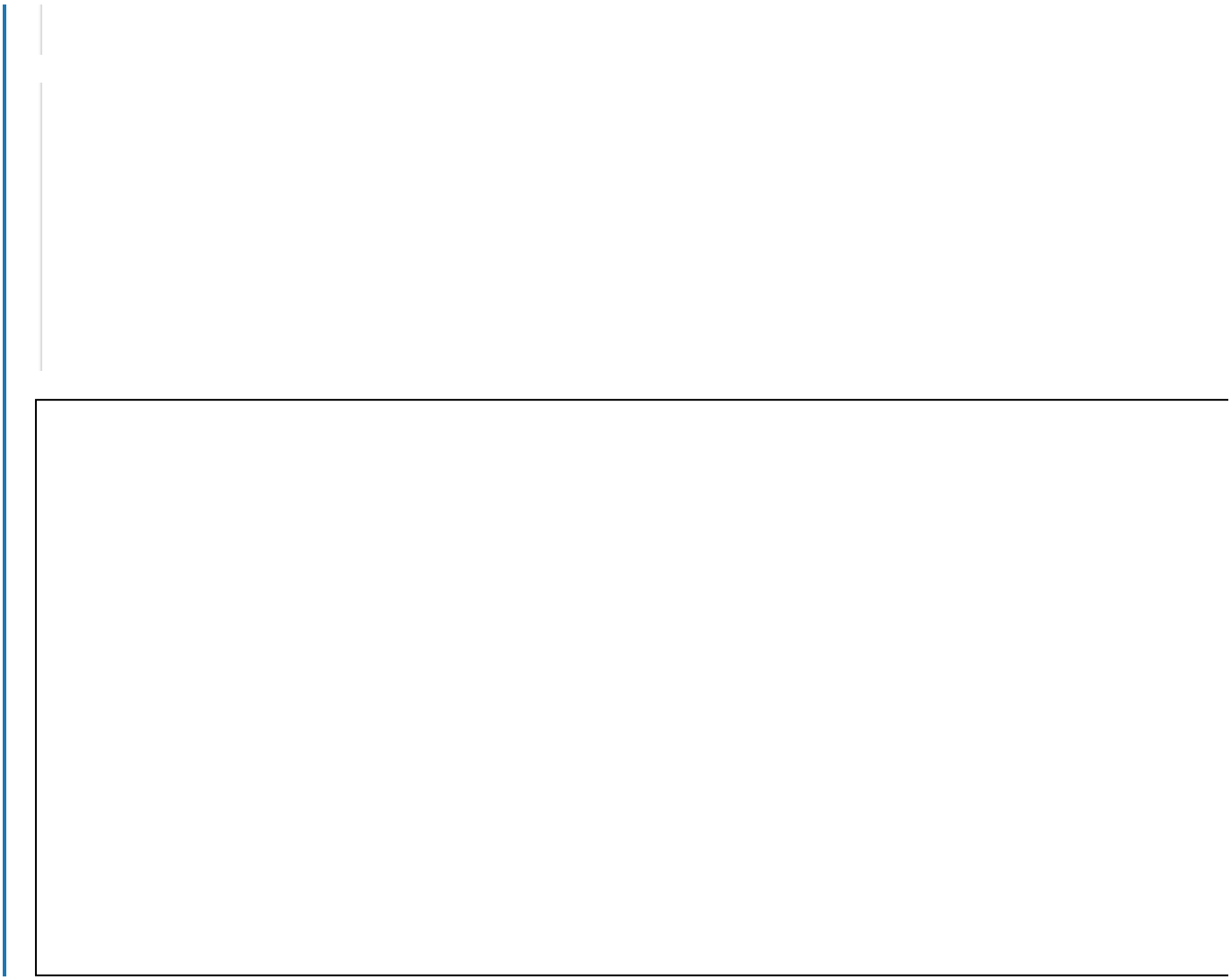
- `wiadomoscDoZakodowania` – tekst jawny przekazany do zaszyfrowania; ciąg znaków

Wynik:

- `zakodowaneSlowo` – zaszyfrowana wiadomość; ciąg znaków

Twoje zadania

1. Napisz program szyfrujący poprawnie podaną wiadomość, korzystając z szyfru blokowego i wybranego klucza.



Ćwiczenie 2

Napisz program szyfrujący wiadomość z wykorzystaniem szyfru strumieniowego. Jego działanie przetestuj dla klucza o wartości 7178300 oraz wiadomości `Animos labor nutrit`. Strumień wygeneruj za pomocą funkcji `rand()`, a otrzymaną wartość podziel modulo przez 128, przed wykorzystaniem jej w procesie szyfrowania. Do generowania bitów szyfrogramu wykorzystaj operację XOR.

Specyfikacja problemu:

Dane:

- `klucz` – informacja wykorzystywana do generowania strumienia bitów; liczba całkowita
- `wiadomoscDoZakodowania` – tekst jawny przekazany do zaszyfrowania; ciąg znaków

Wynik:

- `zakodowaneSlowo` – zaszyfrowana wiadomość; ciąg znaków

Twoje zadania

1. Napisz program szyfrujący poprawnie podaną wiadomość, korzystając z szyfru strumieniowego i wybranego klucza.

Ćwiczenie 3

Napisz program obliczający na podstawie algorytmu Diffiego-Hellmana klucz prywatny s, za pomocą którego Alicja i Robert będą mogli szyfrować wiadomości. Działanie swojego programu przetestuj dla następujących wartości: $p = 37$ i $g = 6$ oraz kluczy prywatnych Alicji i Roberta równych odpowiednio 3 i 5.

Specyfikacja problemu:

Dane:

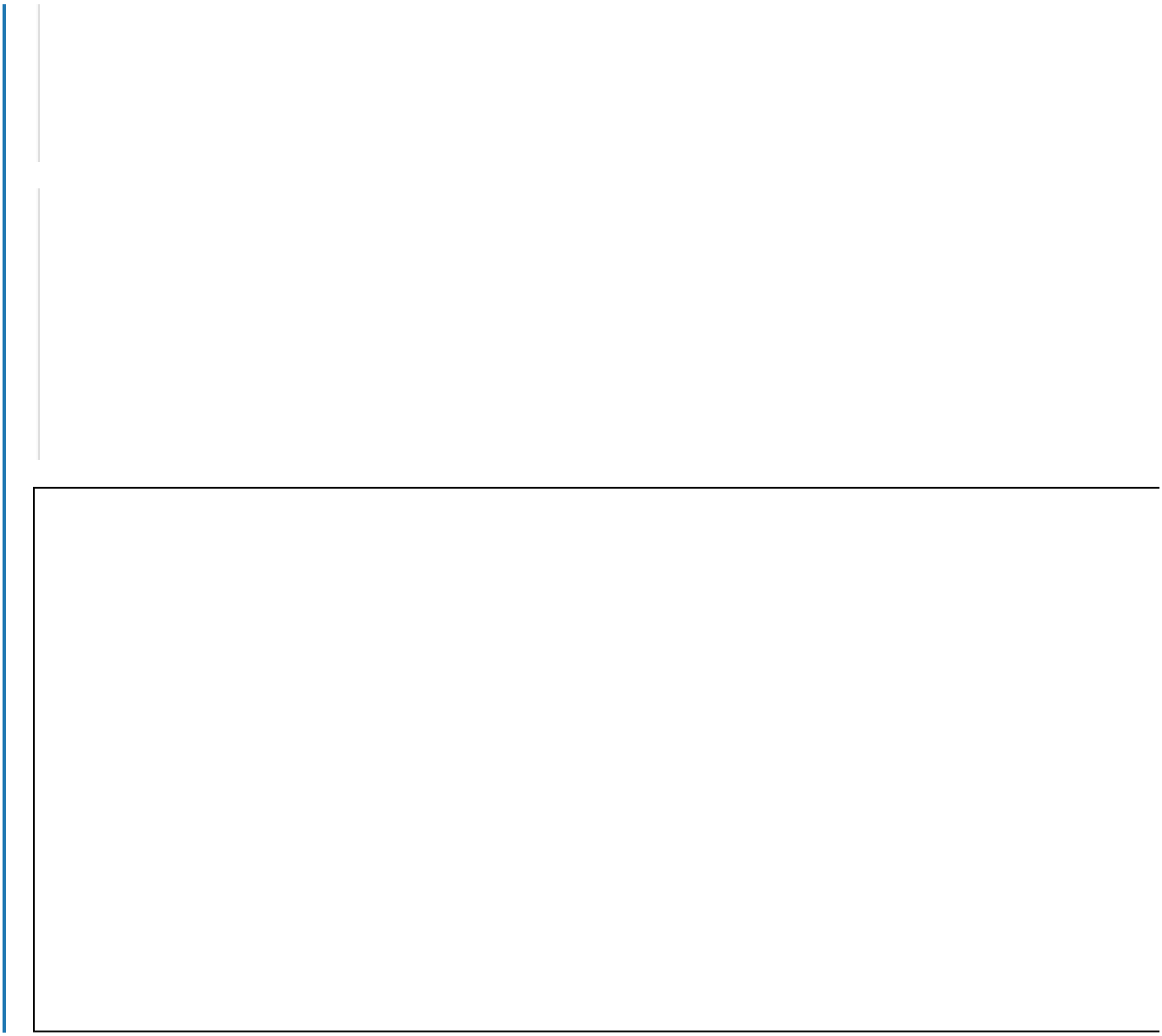
- p – pierwszy element klucza publicznego Alicji; liczba pierwsza
- g – drugi element klucza publicznego Alicji – baza; liczba całkowita
- a – klucz prywatny Alicji; liczba całkowita
- b – klucz prywatny Roberta; liczba całkowita

Wynik:

- s – obliczony sekretny klucz; liczba całkowita

Twoje zadania

1. Program generuje klucz prywatny z wykorzystaniem algorytmu Diffiego-Hellmana dla zdefiniowanych wartości kluczy publicznych komunikujących się osób.



Dla nauczyciela

Autor: Maurycy Gast
Przedmiot: Informatyka

Temat: Szyfry symetryczne i asymetryczne w języku C++

Grupa docelowa:

Szkoła ponadpodstawowa, liceum ogólnokształcące, technikum, zakres podstawowy i rozszerzony

Podstawa programowa:

Cele kształcenia – wymagania ogólne

I. Rozumienie, analizowanie i rozwiązywanie problemów na bazie logicznego i abstrakcyjnego myślenia, myślenia algorytmicznego i sposobów reprezentowania informacji.

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

V. Przestrzeganie prawa i zasad bezpieczeństwa. Respektowanie prywatności informacji i ochrony danych, praw własności intelektualnej, etykiety w komunikacji i norm współżycia społecznego, ocena zagrożeń związanych z technologią i ich uwzględnienie dla bezpieczeństwa swojego i innych.

Treści nauczania – wymagania szczegółowe

I. Rozumienie, analizowanie i rozwiązywanie problemów.

Zakres podstawowy. Uczeń:

2) stosuje przy rozwiązywaniu problemów z różnych dziedzin algorytmy poznane w szkole podstawowej oraz algorytmy:

b) na tekstach: porównywania tekstów, wyszukiwania wzorca w tekście metodą naiwną, szyfrowania tekstu metodą Cezara i przestawieniową,

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

3) objaśnia dobrany algorytm, uzasadnia poprawność rozwiązania na wybranych przykładach danych i ocenia jego efektywność;

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera i innych urządzeń cyfrowych.

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

3) sprawnie posługuje się zintegrowanym środowiskiem programistycznym przy pisaniu, uruchamianiu i testowaniu programów;

I + II. Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

3) objaśnia, a także porównuje podstawowe metody i techniki algorytmiczne oraz struktury danych, wykorzystując przy tym przykłady problemów i algorytmów, w szczególności:

f) metodę szyfrowania z kluczem publicznym i jej zastosowanie w podpisie elektronicznym,

V. Przestrzeganie prawa i zasad bezpieczeństwa.

Zakres podstawowy. Uczeń:

3) stosuje dobre praktyki w zakresie ochrony informacji wrażliwych (np. hasła, pin), danych i bezpieczeństwa systemu operacyjnego, objaśnia rolę szyfrowania informacji;

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

1) objaśnia rolę technik uwierzytelniania, kryptografii i podpisu elektronicznego w ochronie i dostępie do informacji;

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;
- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Porównasz implementację szyfru strumieniowego i szyfru blokowego w języku C++.
- Prześledzisz, jak komputer generuje kolejne liczby pseudolosowe.

- Przeanalizujesz, dlaczego szyfry asymetryczne wykorzystywane są w komunikacji internetowej.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych;
- ćwiczenia praktyczne.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda;
- oprogramowanie dla języka C++, w tym kompilator GCC/G++ 4.5 (lub nowszej wersji) i Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C++ 5.11 (lub nowszej wersji) lub Microsoft Visual Studio.

Przebieg lekcji

Przed lekcją:

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Szyfry symetryczne i asymetryczne w języku C++". Uczniowie zapoznają się z treściami w sekcji „Przeczytaj" w kontekście programowania.

Faza wstępna:

1. Nauczyciel wyświetla i odczytuje temat lekcji oraz cele zajęć. Prosi uczniów o sformułowanie kryteriów sukcesu.
2. **Rozpoznanie wiedzy uczniów.** Nauczyciel prosi wybranego ucznia lub uczniów o przedstawienie sytuacji problemowej związanej z tematem lekcji.

Faza realizacyjna:

1. **Praca z tekstem.** Nauczyciel ocenia, na podstawie informacji na platformie, stan przygotowania uczniów do zajęć. Jeżeli jest ono niewystarczające, prosi wybraną osobę o przedstawienie najważniejszych informacji z sekcji „Przeczytaj". Następnie na forum klasy uczniowie analizują przedstawione w niej rozwiązania Przykładów 1, 2 i 3. W kolejnym kroku indywidualnie implementują omawiane algorytmy na swoich komputerach.
2. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Prezentacja multimedialna". Uczniowie wspólnie zapoznają się z jego treścią. Zapisują ewentualne problemy i pytania. Po czym następuje dyskusja, w trakcie której nauczyciel wyjaśnia niezrozumiałe treści.
3. **Ćwiczenie umiejętności.** Uczniowie wykonują indywidualnie ćwiczenie nr 1 z sekcji „Sprawdź się", a następnie porównują swoje odpowiedzi z kolegą lub koleżanką.
4. Nauczyciel dzieli uczniów na grupy czteroosobowe. Uczniowie wykonują ćwiczenie nr 2 z sekcji „Sprawdź się", a następnie każda grupa wyznacza jedną osobę, którą wymienia się z inną grupą. W nowych zespołach uczniowie porównują swój kod i wybierają najbardziej efektywne rozwiązanie.

Faza podsumowująca:

1. Nauczyciel ponownie wyświetla na tablicy temat i cele lekcji zawarte w sekcji „Wprowadzenie". W kontekście ich realizacji następuje omówienie ewentualnych problemów z rozwiązaniem ćwiczeń z sekcji „Sprawdź się".
2. Na koniec zajęć z programowania w C++ nauczyciel prosi uczniów o rozwinięcie zdania: „Na dzisiejszych zajęciach nauczyłam/łem się jak...".

Praca domowa:

1. Uczniowie wykonują ćwiczenie nr 3 z sekcji „Sprawdź się".

Materiały pomocnicze:

- Oficjalna dokumentacja techniczna dla języka C++.
- Oficjalna dokumentacja techniczna dla kompilatora GCC/G++ 4.5 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C++ 5.11 (lub nowszej wersji) lub Microsoft Visual Studio.

Wskazówki metodyczne:

- Treści w sekcji „Przeczytaj" można wykorzystać jako podsumowanie i utrwalenie wiedzy uczniów.