Wstęp do informatyki

Marcin Orchel

Spis treści

Rozdział 1

Podstawy programowania

1.1 Wprowadzenie do algorytmiki, instrukcja warunkowa

1.1.1 Wstęp teoretyczny

Standardy C

- ANSI C, inaczej Standard C, inaczej C89, inaczej ANSI X3.159-1989.
- ISO/IEC 9899: 1990, inaczej C90 (praktycznie to samo co C89)
- IOS/IEC 9899: 1999, inaczej C99

Standardy C++

- ISO/IEC 14882:1998
- 2003 poprawiona wersja standardu C++
- 2005 technical report: Library Technical Report 1 (TR1)

Przykład z sumowaniem liczb

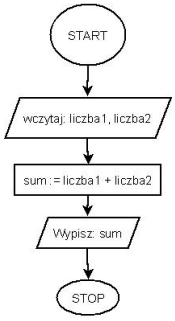
Zapis w języku C++.

```
1 /* program sumujący dwie liczby */
2 void sum() //definicja funkcji
3 {
4   int liczba1 = 20; //definicja zmiennej
5   int liczba2 = 30; //definicja zmiennej
6   int suma = liczba1 + liczba2; //definicja zmiennej suma
7   printf("suma liczb: %d", suma ); //wypisanie sumy
8 }
```

Zapis w pseudojęzyku:
Na rysunku Rys. 1.1 pokazany jest schemat blokowy dla algorytmu sumowania liczb.

Alg. 1 Suma dwóch liczb

Warunki początkowe: $liczba1 \in \mathbb{Z}$, $liczba2 \in \mathbb{Z}$
Warunki końcowe: $sum = liczba1 + liczba2$
1: $sum \Leftarrow liczba1 + liczba2$
2: **Wypisz:** sum



Rysunek 1.1: Schemat blokowy dla algorytmu sumowania.

Język programowania C++

Algorytmika

Pseudojęzyk Elementy pseudojęzyka:

- Nazwa algorytmu
- Opis algorytmu
- Warunki początkowe
- Warunki końcowe
- instrukcje specjalne, jak np. instrukcja warunkowa
- inne instrukcje

Schemat blokowy Elementy schematu blokowego:

- początek algorytmu
- koniec algorytmu
- strzałki
- blok operacyjny
- bloki wejścia/wyjścia
- blok decyzyjny

Alg. 2 Instrukcja warunkowa

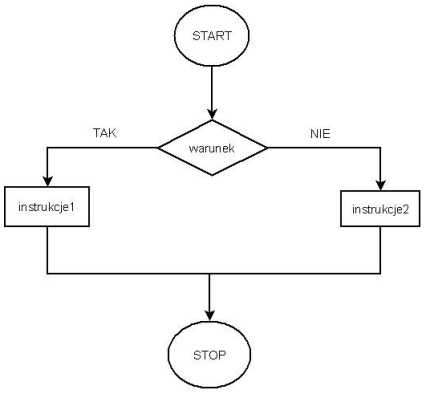
```
1: if warunek then
2:    {instrukcje jesli warunek jest spelniony}
3: else
4:    {instrukcje jesli warunek jest spelniony}
5: end if
```

Instrukcja warunkowa

Instrukcja warunkowa pozwala na wykonanie odpowiedniej listy instrukcji w zależności od wartości warunku logicznego. W języku C++ wygląda następująco:

```
1  if (warunek)
2  {
3    //lista instrukcji, ktore zostana wykonane
4    //jesli warunek jest prawdziwy
5  } else
6  {
7    //lista instrukcji, ktore zostana wykonane
8    //jesli warunek nie jest prawdziwy
9  }
```

Zapis instrukcji warunkowej w pseudojęzyku został przedstawiony w Alg. 2, zaś w schemacie blokowym na Rys. 1.2.



Rysunek 1.2: Schemat blokowy z instrukcją warunkową.

Istnieje możliwość łączenia kilku instrukcji warunkowych, w języku C++ wygląda to następująco:

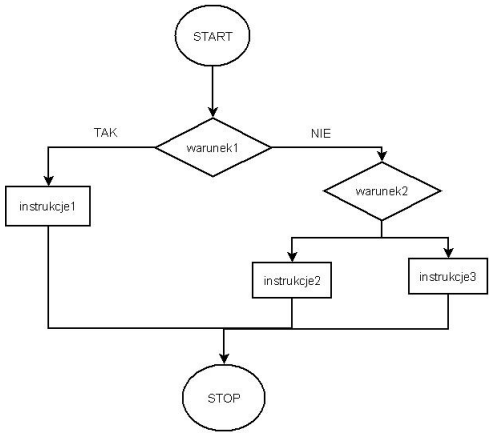
```
1  if (warunek1)
2  {
3    //lista instrukcji, ktore zostana wykonane
4    //jesli warunek jest prawdziwy
```

Alg. 3 Instrukcja warunkowa

```
1: if warunek then
2:     {lista instrukcji, ktore zostana wykonane,
       jesli warunek jest prawdziwy}
3: else if warunek2 then
4:     {lista instrukcji, ktore zostana wykonane,
       jesli warunek1 nie jest prawdziwy
       oraz warunek2 jest prawdziwy}
5: else
6:     {lista instrukcji, ktore zostana wykonane,
       jesli zaden z warunkow nie jest prawdziwy}
7: end if
```

```
5   } else if (warunek2)
6   {
7     //lista instrukcji, ktore zostana wykonane
8     //jesli warunek1 nie jest prawdziwy
9     //oraz warunek2 jest prawdziwy
10  } else
11  {
12    //lista instrukcji, ktore zostana wykonane
13    //jesli zaden z warunkow nie jest prawdziwy
14  }
```

W pseudojęzyku łączenie instrukcji warunkowych przedstawione jest w Alg. 3, zaś w schemacie blokowym na Rys. 1.3.



Rysunek 1.3: Schemat blokowy z dwoma połączonymi instrukcjami warunkowymi.

Przykład z nierównością trójkąta

Zapis w języku C++.

```
1  /* program sprawdzający warunek trójkąta*/
2  void warunekTrojkata()
3  {
4      int bokA = 4; //definicja zmiennej
5      int bokB = 3; //definicja zmiennej
6      int bokC = 6; //definicja zmiennej
7      if (bokA >= bokB + bokC
8              || bokB >= bokA + bokC
9              || bokC >= bokA + bokB) //warunek trojkata
10     {
11         printf(warunek trójkąta nie spełniony");
12 ␣␣␣␣}
13 ␣␣␣␣else
14 ␣␣␣␣{
15 ␣␣␣␣␣␣␣␣printf("warunek trójkąta spełniony");
16 ␣␣␣␣}
17 }
```

Jeśli wiadomo, że jeden z boków jest większy lub równy od dwóch pozostałych to sprawdzanie warunku trójkąta sprowadza się do sprawdzenia zachodzenia jednej nierówności: jeśli założymy, że $bokA >= bokB$ i $bokA >= bokC$ to wystarczy sprawdzenie warunku $bokA >= bokB + bokC$.

Techniki programowania

- umieszczanie zawsze w instrukcji warunkowej bloku *else*
- w niektórych przypadkach łączenie dwóch instrukcji warunkowych w jedną za pomocą operatora koniunkcji, przykładowo w języku C++ kod postaci:

```
1    if (warunek1)
2    {
3      if (warunek2)
4      {
5        instrukcja1;
6      } else
7      {
8        //empty
9      }
10   } else
11   {
12     //empty
13   }
```

możemy zapisać w postaci:

```
1    if (warunek1 && warunek2)
2    {
3      instrukcja1;
4    } else
5    {
6      //empty
7    }
```

1.1.2 Zadania

Zadania na 3.0

Napisać algorytm sprawdzający położenie punktu $A(x_A, y_A)$ względem danej prostej

$$y = ax + b$$

Algorytm zapisać w pseudojęzyku i schemacie blokowym.

Zadania na 4.0

Zapisać algorytm wyliczania rozwiązania układu dwóch równań liniowych z dwoma zmiennymi za pomocą pseudojęzyka i schematu blokowego.

Zadania na 5.0

Mamy dany odcinek $\overline{AB}$ w kartezjańskim układzie współrzędnych określony przez dwa punkty $A(x_A, y_A)$ i $B(x_B, y_B)$ oraz punkt $C(x_C, y_C)$. Napisać algorytm w pseudojęzyku, który sprawdza czy punkt C należy do odcinka AB.

1.2 Algorytm Euklidesa, instrukcje sterujące

1.2.1 Wstęp teoretyczny

Pętla *while*

Przykład w C++:

```
1    int i = 20;
2    while (i > 10)
3    {
4      printf("liczba␣i:␣%d\n", i);
5      i--;
6    }
```

Zapis algorytmiczny:

Alg. 4 Pętla *while*

```
1: while warunek do
2:    {instrukcje}
3: end while
```

Alg. 5 Pętla *repeat – until*

```
1: repeat
2:    {instrukcje}
3: until warunek
```

Pętla *do – while*

Przykład w C++:

```
1   int i = 20;
2   do
3   {
4      printf("liczba i: %d\n", i);
5      i--;
6   } while (i > 10);
```

Zapis algorytmiczny:

Pętla *for*

Przykład w C++:

```
1   for (int i = 20; i > 10; i--)
2   {
3      printf("liczba i: %d\n", i);
4   }
```

Pętla *for* może być zapisana za pomocą pętli *while*.

Pętla nieskończona

Język C++:

```
1   while(true)
2   {
3      printf("petla");
4   }
```

Zapis algorytmiczny:

Instrukcja break

Alg. 6 Pętla nieskończona

```
1: loop
2:    {instrukcje}
3: end loop
```

```
1   for (int i = 20; i > 10; i--)
2   {
3     if (i == 15)
4     {
5       break;
6     }
7     printf("liczba␣i:␣%d\n", i);
8   }
```

Pętle z użyciem instrukcji *break* można zapisać równoważnie bez tej instrukcji w następujący sposób:

```
1   bool done = false;
2   for (int i = 20; i > 10 && !done; i--)
3   {
4     if (i == 15)
5     {
6       done = true;
7     } else
8     {
9       printf("liczba␣i:␣%d\n", i);
10    }
11  }
```

Instrukcja continue

```
1   for (int i = 20; i > 10; i--)
2   {
3     if (i == 15)
4     {
5       //inne instrukcje
6       continue;
7     }
8     printf("liczba␣i:␣%d\n", i);
9   }
```

Pętle z instrukcją *continue* można zapisać bez tej instrukcji następująco:

```
1   for (int i = 20; i > 10; i--)
2   {
3     if (i == 15)
```

```
4       {
5         //inne instrukcje
6       } else
7       {
8         printf("liczba i: %d\n", i);
9       }
10    }
```

1.2.2 Algorytm Euklidesa

Algorytm Euklidesa służy do obliczenia największego wspólnego dzielnika dwóch liczb (oznaczenie: NWD(a,b) lub $\gcd(a, b)$). Algorytm podstawowy obliczania $\gcd(a, b)$ polega na znalezieniu wszystkich dzielników liczby a, później b, a następnie wybraniu największego wspólnego dzielnika. Algorytm Euklidesa opiera się na własności, że:

$$\gcd(a, b) = \gcd(a - b, b)$$

dla $a > b$ oraz

$$\gcd(a, b) = \gcd(a, b - a)$$

dla $b > a$. Jeśli $a = b$ to $\gcd(a, b) = a$.

Istnieje również druga wersja algorytmu Euklidesa, gdzie korzystamy z własności:

$$\gcd(a, b) = \gcd(a \bmod b, b - (a \bmod b))$$

Implementacja obu wersji dla $a, b > 0$ została przedstawiona poniżej:

```
1     int tempNumber1 = a;
2     int tempNumber2 = b;
3     while (tempNumber1 != tempNumber2)
4     {
5       if (tempNumber1 > tempNumber2)
6       {
7         tempNumber1 = tempNumber1 - tempNumber2;
8       } else
9       {
10        tempNumber2 = tempNumber2 - tempNumber1;
11      }
12    }
13    printf("NWD = %d\n", tempNumber1);
```

```
1     int tempNumber1 = a;
2     int tempNumber2 = b;
3     while (tempNumber1 > 0)
4     {
5       tempNumber1 = tempNumber1 % tempNumber2;
6       tempNumber2 = tempNumber2 - tempNumber1;
```

```
7     }
8     printf("NWD = %d\n", tempNumber2);
```

Można pokazać, że oba powyższe algorytmy zawsze zakończą swoje działanie.

1.2.3 Zadania

Zadania na 3.0

Sprawdzić, czy podana liczba jest liczbą pierwszą.

Zadania na 4.0

Sprawdzić, czy podana liczba jest liczbą doskonałą. Liczba doskonała to liczba naturalna, która jest sumą swoich dzielników właściwych. Przykład liczby doskonałej: $28 = 2 \cdot 2 \cdot 7 = 1 + 2 + 7 + 4 + 14$.

Zadania na 5.0

Wyznaczyć metodą iteracyjną kolejne wartości liczb Fibonacciego.

1.3 Reprezentacja liczba stało i zmienno-przecinkowych w komputerze

1.3.1 Wstęp teoretyczny

Specyfikacja możliwych typów w języku C znajduje się w Tablica 1.1. To czy typ *int* (lub *char*) jest typem *signed int* (*signed char*) czy *unsigned int* (*unsigned char*) jest zależne od implementacji.

Zakres liczb całkowitych 1 bajtowych: 0 - 255.
Zakres liczb całkowitych 2 bajtowych: 0 - 65535.
Zakres liczb całkowitych 4 bajtowych: 0 - 4294967295.
Zakres liczb całkowitych 1 bajtowych ze znakiem: -128 - 127.
Zakres liczb całkowitych 2 bajtowych ze znakiem: -32768 - 32767.
Zakres liczb całkowitych 4 bajtowych ze znakiem: -2147483648 - 2147483647.

Stałe. Stałe znakowe. Liczby całkowite. Stałe typu *long* zapisujemy z l na końcu. Stałą typu *unsigned* zapisujemy z literką u na końcu. Zaś stałą typu *unsigned long* z literkami ul. Liczby całkowite można przedstawiać w postaci ósemkowej za pomocą 0 na początku, lub w postaci szesnastkowej za pomocą $0x$ na początku. Stałe zmiennopozycyjne zapisujemy z kropką dziesiętną lub w notacji naukowej, jeśli na końcu jest L to są typu *long double*. Stała zmiennopozycyjna jest domyślnie typu *double*, z literką f na końcu jest typu $float$, a z literką l jest typu *long double*.

W języku C występuje również typ wyliczeniowy deklarowany za pomocą słowa kluczowego *enum*.

Deklaracje zmiennych mogą być poprzedzone słowem kluczowym języka C *const*.

Liczby całkowite

Liczby całkowite są zapisane w komputerze w kodzie uzupełnień do dwóch (U2). Liczby dodatnie są reprezentowane binarnie w standardowy sposób, i dodawany jest bit 0 na początku tej liczby. Liczby przeciwne powstają przez odjęcie liczby od dwukrotnej wagi najstarszego bitu $2 \cdot 2^{n-1} = 2^n$. W kodzie uzupełnień do jednego (U1) liczby przeciwne powstają przez odjęcie liczby od liczby składającej się z samych jedynek, np $-1 = 11111110$. Przykład dla dwóch bitów znalezienie liczby przeciwnej do 1 w u2:

$$1_{10} = 01_2$$

$$-1 = 100_2 - 01_2 = 11_2$$

Operacja wyszukiwania liczby przeciwnej może być przeprowadzona również w systemie dziesiętnym: Przykład: znaleźć reprezentację 4-bitową liczby -5.

$$-5_{10} = 2^4 - 5_{10} = 11_{10} = 1011_2$$

Na n-bitach można zapisać liczby z zakresu:

$$\left[-2^{n-1}, 2^{n-1} - 1\right]$$

Przykład zapisu liczb na 4 bitach:
0111, 7
0110, 6
0101, 5
0100, 4
0011, 3
0010, 2
0001, 1
0000, 0
1111, -1
1110, -2
1101, -3
1100, -4
1011, -5
1010, -6
1001, -7
1000, -8

W porównaniu do zapisu znak-moduł kolejność zapisu liczb ujemnych jest odwrócona i są przesunięte o 1, tak że nie ma podwójnego zera, co oznacza, że zero jest zapisane tylko za pomocą jednego kodu. Operacje dodawania i odejmowania są wykonywane tak samo jak dla liczb binarnych bez znaku. W metodzie tej liczby ujemne mają pierwszy bit równy 1.

Sposób konwersji liczby w kodzie u2 do systemu dziesiętnego w kodzie znak-moduł polega na wstawieniu znaku minus przy najwyższej potędzie 2:

$$1101_{u2} = -2^3 + 2^2 + 2^0 = -3_{zm}$$

Alternatywna metoda zamiany na liczbę przeciwną w kodzie u2: dokonujemy inwersji bitów, a następnie wynik zwiększamy o 1.

Przykład:

$$-5 = -\left(0101_{u2}\right) = 1010_2 + 1_2 = 1011_{u2} = 5$$

Dodawanie pisemne liczb całkowitych w systemie u2. Przykład dla liczb 8 bitowych:

$$15_{zm} = 00001111_{u2}$$

$$-5_{zm} = 11111011_{u2}$$

15 + (- 5)
11111111
00001111
+11111011
– – – – –
00001010

Ostatnie przeniesienie do 9 bitu jest ignorowane.

Liczby rzeczywiste

Konwersja liczby 0,1 do postaci binarnej:

$$0,1 = 2^0 \cdot 0 + 0,1$$

$$2^{-1} = 0,5$$

$$0,1 = 2^0 \cdot 0 + 2^{-1} \cdot 0 + 0,1$$

$$2^{-2} = 0,25$$

$$0,1 = 2^0 \cdot 0 + 2^{-1} \cdot 0 + 2^{-2} \cdot 0 + 0,1$$

$$2^{-3} = 0,125$$

$$0,1 = 2^0 \cdot 0 + 2^{-1} \cdot 0 + 2^{-2} \cdot 0 + 2^{-3} \cdot 0 + 0,1$$

$$2^{-4} = 0,0625$$

$$0,1 = 2^0 \cdot 0 + 2^{-1} \cdot 0 + 2^{-2} \cdot 0 + 2^{-3} \cdot 0 + 2^{-4} \cdot 1 + 0,0375$$

$$0,1_{10} = 0,0001_2$$

$$\ldots$$

Skończona reprezentacja liczb dziesiętnych w systemie binarnym. Przykład:

$$\frac{3}{8} = 0,375$$

$$0,375 = 2^0 \cdot 0 + 2^{-1} \cdot 0 + 2^{-2} \cdot 1 + 2^{-3} \cdot 1$$

Liczby dziesiętne, które można przedstawić w postaci sumy dowolnych potęg dwójki mają skończoną reprezentację binarną.

Znormalizowana reprezentacja liczb zmiennoprzecinkowych w komputerze:

$$x = \pm 0, b_1 b_2 \dots b_n \cdot 2^E$$

gdzie $b_1 \neq 0$; $0, b_1 b_2 \dots b_n$ - *mantysa*; E - liczba całkowita zwana *cechą*.

Przykład:

$$0,0001_{10} = 0,1 \cdot 2^{-3}$$

Sumowanie liczb Założenia mantysa 5 pozycji, cecha 1 pozycja. Przykład:

$$a = 0,1234 \cdot 10^3 = 123,4$$

$$b = 0,2345 \cdot 10^2 = 23,45$$

$$a + b = 0,1234 \cdot 10^3 + 0,02345 \cdot 10^3 = 0,14685 \cdot 10^3 = 146,85$$

$$a + b = 0,1234 \cdot 10^3 + 0,0234 \cdot 10^3 = 0,1468 \cdot 10^3 = 146,8$$

Błąd bezwzględny:

$$146,85 - 146,8 = 0,05$$

Błąd względny:

$$\frac{0,05}{146,85} \cdot 100\% = 0,034\%$$

1.3.2 Zadania

Zadania na 3.0

Wykonać dodawanie przy pomocy znormalizowanej reprezentacji liczb zmiennoprzecinkowych następujących liczb:

$$123456.7$$

$$101.7654$$

dla reprezentacji z 1 pozycją na cechę i z 6 pozycjami na mantysę. Podać wartość błędu bezwzględnego i względnego.

Zadania na 4.0

Po zamianie podanych liczb w systemie u2 do precyzji 8 bitowej również w systemie u2 pomnożyć je: 110_{u2} i 011_{u2}.

Zadania na 5.0

Przeanalizować, w którym wyrażeniu

$$x^2 + y^2$$

$$(x - y)(x + y)$$

błędy zaokrągleń są mniejsze, podać co najmniej 3 różne przykłady potwierdzające wnioski.

1.4 Operatory

1.4.1 Wstęp teoretyczny

- Dwuargumentowe operatory arytmetyczne: +, -, *, /, %.
- Operatory relacji: <, >=, <, <=.
- Operatory przyrównania: ==, !=.
- Operatory logiczne: &&, ||. Wyrażenia z tymi operatorami obliczamy od lewej do prawej, jeśli w trakcie analizy kolejnych operandów jest wiadome, jaką wartość będzie miało całe wyrażenie to kolejne operandy nie są obliczane.
- Jednoargumentowy operator negacji !.
- operatory zwiększania i zmniejszania: ++, - - mogą być przedrostkowe jak i przyrostkowe. W przypadku przedrostkowym wartość wyrażenia zmienia się przed jego użyciem, a w przypadku przyrostkowym wartość wyrażenia zmienia się po jego użyciu.

```
int a1 = 1;
int a2 = 1;
int l1 = ++a1;
int l2 = a2++;
```

Wynikiem jest $l1 = 2$, $l2 = 1$.

Wyrażenia logiczne w C mają wartość niezerową, jeśli są prawdziwe i 0, jeśli są fałszywe.

Konwersja typów. Automatyczne przekształcenie argumentu „ciaśniejszego" na „obszerniejszy" bez utraty informacji. Przykład:

```
int i = 2 + 0.6 + 0.6;
```

Liczba całkowita 2 zostanie przekształcona na liczbę typu *double* 2.0. Zmienna i będzie miała wartość 3. Możliwa jest również konwersja stratna np. w przypadku przypisania wyrażenia zmiennopozycyjnego do zmiennej całkowitej:

```
int i = 0.3;
```

W przypadku dzielenia liczby całkowitej przez całkowitą należy pamiętać, aby dzielna była liczbą zmiennopozycyjną lub dzielnik.

```
double d = 3/4;
```

Zmienna d będzie miała wartość 0.0. Jednak zazwyczaj chodzi nam o otrzymanie wartości 0,75:

```
double d = 3.0/4;
```

Dobrym zwyczajem jest zapisywanie stałych w docelowych typach, ze względu na czytelność programu, a więc w powyższym przypadku możemy zapisać:

```
double d = 3.0/4.0;
```

Operator rzutowania. *(nazwa typu) wyrażenie.* W przypadku rzutowania zmiennej całkowitej 2 bajtowej int do 1 bajtowej char, traci się najbardziej znaczący bajt (ten od lewej). Przykład: $(unsigned\ char)10000001 = 1$.

Operatory przypisania =, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^= . Przypisanie ma wartość o typie lewego argumentu i wartości przypisanej.

Wyrażenia warunkowe: $wyr1\ ?\ wyr2 : wyr3$.

Operatory bitowe.

- & - bitowa koniunkcja (AND). Wartości 1 i 0 traktowane są jako wartości logiczne. Dokonywana jest koniunkcja na każdym bicie. Służy do zerowania poszczególnych bitów w danej zmiennej. Jeden z argumentów można rozumieć jako maskę, ponieważ dla tych bitów, które ustawione są w nim na 1 wartości bitów pierwszego argumentu nie zmieniają się. Przykład: 111101 & 000111 = 000101.

- | - bitowa alternatywa (OR). Dokonywana jest alternatywa na każdym bicie. Służy do ustawiania 1 na poszczególnych bitach. Również tutaj można traktować jeden z argumentów jako maskę, ponieważ dla tych bitów, które ustawione są na 0 wartości bitów pierwszego argumentu nie zmieniają się. Przykład: 111101 | 000111 = 111111

- ^- bitowa różnica symetryczna (XOR). Dokonywana jest różnica symetryczna na każdym bicie, polegająca na ustawianiu 1 wszędzie gdzie bity w obu argumentach są różne, w przeciwnym razie 0. Wartość w masce 0 nie zmienia bitu, a 1 przełącza bit. Możemy zamienić każdy bit 1 na 0 i odwrotnie za pomocą bitowej różnicy symetrycznej z maską z samymi 1, jakkolwiek dla tej operacji istnieje osobny operator dopełnienia jedynkowego. Bitowa różnica symetryczna ma również tą właściwość, że zastosowanie dwukrotnie maski przywraca poprzednią wartość zmiennej, dlatego może być z łatwością użyta do kodowania i rozkodowywania. Przykład:
  111101^000111 = 111010
  111010^000111 = 111101.

- << - przesunięcie w lewo. Przesunięcie bitów w lewo w argumencie stojącym po lewej stronie operatora o liczbę pozycji określoną przez argument po prawej stronie operatora. Zwolnione bity są wypełniane zerami. Przykład: 111 << 2 = 11100. Przesunięcie o jedną pozycję powoduje dodanie jednego zera i odpowiada mnożeniu liczby przez 2.

- >> - przesunięcie w prawo. Przesunięcie bitów w prawo w argumencie stojącym po lewej stronie operatora o liczbę pozycji określoną przez argument po prawej stronie operatora. Zwolnione bity są wypełniane zerami, jednakże dla liczb ze znakiem zwolnione bity na niektórych maszynach wypełniane są bitem znaku. Przykład: 00000111 >> 2 = 1, przy wypełnianiu bitem znaku: 11111111 >> 2 = 11111111. Przesunięcie w prawo o jedną pozycję odpowiada dzieleniu przez 2:
  0110 - 6
  0011 - 3
  0001 - 1.
  Dla wypełniania bitem znaku otrzymujemy również poprawne dzielenie przez 2 dla liczb ujemnych:
  1010 - -6
  1101 - -3
  1110 - -2.

- ~ - dopełnienie jedynkowe (operator jednoargumentowy). Zamienia każdy bit 1 na 0 i odwrotnie. Przykład: wyzerowanie ostatnich 6 bitów w dowolnej liczbie całkowitej. A więc trzeba zrobić bitową koniunkcję z liczbą, która ma same jedynki i sześć ostatnich bitów 0. Gdy chcemy używać taką stałą z liczbami całkowitymi o dowolnej długości możemy ją zapisać jako: ~111111.

Proste operacje bitowe (najmniej znaczący bit ma indeks 0):

- ustawienie n-tego bitu na 1

  ```
  unsigned char c |= (1 << n)
  ```

- ustawienie n-tego bitu na 0

  ```
  unsigned char c &= ~(1 << n)
  ```

- przełączenie n-tego bitu

  ```
  unsigned char c ^= (1 << n)
  ```

- testowanie n-tego bitu, czy n-ty bit jest ustawiony na 1,

  ```
  unsigned char e = d & (1 << n)
  ```

  Testowanie to można również zrealizować następująco:

  ```
  unsigned char e = 1 & (d >> n)
  ```

Przykłady wykorzystania operacji bitowych.

Przykład 1. Zadanie polega na zwróceniu n bitów wyciętych ze zmiennej x od pozycji p, dosunięte do prawej strony wyniku. Zmienna x jest typu *unsigned*, wynik jest również typu *unsigned*. Nie zakładamy nic o rozmiarze tych zmiennych.

```
return (x>>(p-n)) & ~(~0<<n);
```

Rozwiązanie składa się z dwóch kroków, dosunięcie pola do prawego końca, a następnie wyzerowanie bitów znajdujących się przed tym polem. Przykład: 11000101, chcemy wyciąć 100, czyli $n = 3$, $p = 7$. Zmienna x zostaję najpierw przesunięta o 4 pozycje i otrzymujemy: 00001100. Teraz trzeba wyzerować wszystkie bity poza polem, a więc trzeba stworzyć maskę postaci 00000111, i dokonać bitowej koniunkcji. Maska powstaje przez dopełnienie jedynkowe maski 11111000. Maska ta powstaje za pomocą przesunięcia $\tilde{}0 << n$.

Przykład 2. Jeśli chcemy przetestować czy na danych bitach znajdują się określone wartości, to możemy to zrobić następująco: najpierw tworzymy maskę, taką, że ma same 0 na bitach, które nie są interesujące, a na pozostałych ma 1. Po zastosowaniu koniunkcji zerujemy nieistotne bity w zmiennej. Następnie możemy zastosować różnicę symetryczną z maską, która ma same zera na nieistotnych bitach i testowe wartości na bitach istotnych. Jeśli wynik jest zerowy oznacza to, że na określonych bitach wartości testowe są te same co w zmiennej.

Przykład 3. Mnożenie przez stałą da się wykonać za pomocą wielu dodawań i przesunięć. Przykład: $12 = 2^3 + 2^2$

$$a \cdot 12 = a\left(2^3 + 2^2\right) = a2^3 + a2^2$$

```
return (a<<3) + (a<<2);
```

Przykład 4. Znalezienie minimum dwóch liczb:

```
r = y + ((x - y) & -(x < y));
```

Przykład: Gdy pierwsza liczba jest mniejsza od drugiej, to drugi składnik sumy będzie równy modułowi różnicy między tymi liczbami ze znakiem ujemnym, ponieważ drugi człon koniunkcji będzie liczbą składającą się z samych bitowych jedynek. Natomiast gdy pierwsza liczba jest większa od drugiej, to drugi składnik jest 0, ponieważ drugi człon koniunkcji jest równy zero i zeruje pierwszy człon koniunkcji.

Optymalizacje bitowe:

- dzielenie przez potęgi 2 może być zapisane za pomocą przesunięcia,
- reszta z dzielenia przez potęgi dwójki może być zapisana za pomocą koniunkcji bitowej.

1.4.2 Zadania

Zadania na 3.0

Napisać program zliczający liczbę bitów 1 liczby całkowitej typu unsigned o dowolnym rozmiarze.

Zadania na 4.0

Napisać program bez użycia pętli, z użyciem operatorów bitowych, który dokonuje zaokrąglenia liczby do najbliższej od góry potęgi dwójki. Do programu dołączyć omówienie. Zadanie dodatkowe nieobowiązkowe: Napisać program bez użycia pętli, z użyciem operatorów bitowych, który dokonuje zaokrąglenia liczby do najbliższej potęgi dwójki.

Zadania na 5.0

Napisać program bez użycia pętli, z użyciem operatorów bitowych, który sprawdza czy liczba jest potęgą dwójki. Do programu dołączyć omówienie.

Zadania na 6.0

Napisz program bez użycia pętli, z użyciem operatorów bitowych zwracający informacje czy w liczbie znajduje się parzysta liczba bitów 1, czy nieparzysta. Do programu dołączyć omówienie.

1.5 Funkcje

1.5.1 Wstęp teoretyczny

Deklaracja funkcji:

```
zwracany_typ nazwa_funkcji(lista argumentów)
```

Przekazywanie parametrów do funkcji

W C parametry do funkcji przekazywane są przez wartość. Oznacza to, że kopiowane są wartości argumentów aktualnych do argumentów formalnych utworzonych na stosie. Przykład:

```
void foo(int foobar)
{
  foobar += 4;
  printf("%d\n", foobar);
}
```

Jeśli podaną funkcję wywołamy następująco:

```
int main()
{
  int bar = 1;
  foo(bar);
  printf("%d\n", bar);
  return 0;
}
```

to zostaną wyświetlone liczby 5 i 1, zatem zmienna bar z funkcji main() nie zostanie zmodyfikowana we funkcji foo(int foobar).

W C++ możliwe jest przekazywanie parametrów przez referencję (przezwisko). Przykład referencji:

```
void foo()
{
  int bar1 = 1;
  int &bar2 = bar1;
  bar1 = 3;
  printf("%d\n", bar2);
}
```

W powyższym przykładzie zostanie wyświetlona liczba 3.

Przekazywanie parametrów przez referencję oznacza to, że kopiowany jest adres zmiennej na stos. W podanym przykładzie funkcja main() pozostaje nie zmieniona, zaś funkcja foo wygląda następująco:

```
void foo(int &foobar)
{
  foobar += 4;
  printf("%d\n", foobar);
}
```

Zostaną wyświetlone liczby 5 i 5. Konsekwencją jest zatem zmiana wartości zmiennej bar we funkcji foo(int &foobar). Czytamy zmienna foobar jest referencją (przezwiskiem) do zmiennej typu int.

Referencja ma tę cechę, że jest stałym przezwiskiem, a więc nie można spowodować, że dana referencja stanie się przezwiskiem innej zmiennej. Przykład:

```
void foo()
{
  int bar1 = 1;
  int bar2 = 2;
  int &bar3 = bar1;
  int &bar4 = bar2;
  bar3 = bar4;
  printf("%d␣%d␣%d␣%d\n", bar1, bar2, bar3, bar4);
}
```

Powyższa funkcja wyświetli cztery razy 2. Zmienna bar3 jest referencją do zmiennej bar1. Zmienna bar4 jest referencją do zmiennej bar2. Instrukcja przypisania nie może spowodować, że bar3 stanie się referencją do tego samego co bar4, ze względu na wspomnianą wyżej cechę referencji. Powyższa funkcja poprawnie się skompiluje, a operacja przypisania będzie polegała na skopiowaniu wartości zmiennej bar2 do zmiennej bar1. Ten sam efekt uzyskalibyśmy również poprzez przypisanie bar3 = bar2, w tym wypadku do zmiennej wskazywanej przez bar3 kopiujemy wartość zmiennej bar2.

Stała referencja. Stała referencja oznacza, że nie życzymy sobie zmiany wartości zmiennej, na którą wskazuje referencja. Przykład:

```
void foo()
{
  int bar1 = 1;
  int bar2 = 2;
  const int &bar3 = bar1;
  int &bar4 = bar2;
  //bar3 = bar4;
  //bar3 = bar2;
}
```

Obydwie zakomentowane instrukcje nie powiodą się, ponieważ nie możemy zmieniać wartości zmiennej wskazywanej przez bar3.

Możemy również tworzyć referencje do stałej, ale pod jednym warunkiem, ze względu na to, że nie możemy zmieniać stałej, referencja również powinna być stała:

```
  const int &bar2 = 3;
```

Porównywanie referencji. Porównywane są zmienne, na które wskazują referencje. Przykład:

```
\void foo5()
{
  int bar1 = 1;
  int bar2 = 1;
  int &bar3 = bar1;
  int &bar4 = bar2;
  if (bar3 == bar4)
  {
    printf("Takie same\n");
  }
  if (bar3 == bar2)
  {
    printf("Takie same\n");
  }
}
```

W C++ możliwe jest stosowanie argumentów domniemanych, co oznacza możliwość pominięcia podawania argumentów aktualnych w wywołaniu funkcji. Przykład:

```
void foo(int foobar = 0)
{
  foobar += 4;
  printf("%d\n", foobar);
}
```

Możliwe jest wywołanie powyższej funkcji następująco: foo(). Wtedy zostanie przyjęte, że foobar ma wartość 0.

Techniki programowania

- Zmienne małych rozmiarów mogą być w praktyce przekazywane przez wartość, zmienne większych rozmiarów powinny być przekazywane przez referencję.
- sprawdzanie poprawności argumentów funkcji
- stosowanie const wszędzie gdzie spodziewamy się, że zmienna nie będzie modyfikowana

1.5.2 Zadania

Zadania na 3.0

Napisz funkcję, która modyfikuje dwa parametry aktualne tej funkcji, w ten sposób, że zamienia wartości miejscami. Napisać dwie funkcje, jedna dla argumentów typu int, a druga dla argumentów typu double.

Zadania na 4.0

Czy poprawna jest następująca funkcja:

```
int& foo()
{
  int bar = 9;
  return bar;
}
}
```

Co robi powyższa funkcja? Czy używanie powyższych konstrukcji jest błędem? Jeśli tak, to dlaczego?

Zadania na 5.0

Czy poprawna są funkcje:

```
void foo()
{
}

void bar()
{
  return foo();
}
```

Uzasadnić odpowiedź.

Zadania na 6.0

Co oznacza podany zapis? Do czego mógłby się przydać?

```
typedef int (&rifii) (int, int);
```

1.6 Tablice

1.6.1 Wstęp teoretyczny

Tablice jednowymiarowe

Elementy tablicy zajmują obszar ciągły w pamięci.

Definicja tablicy:

```
int nazwaTablicy[3];
```

Definicja tablicy wraz z inicjalizacją:

```
int tab1[] = {1, 2, 3, 4, 1, 1, 1};
```

Tablica może być również zainicjalizowana za pomocą pętli:

```
int rozmiar = 4;
int tab3[rozmiar];
for (int i = 0; i < rozmiar; i++)
{
  tab3[i] = 0;
}
```

Począwszy od standardu C99 obsługiwane są tablice o rozmiarze ustalanym w trakcie wykonywania programu, a nie podczas kompilacji.

W C zazwyczaj nie jest możliwe przypisanie tablicy do tablicy. Porównywanie tablic jest porównaniem adresu początku tablicy.

Przekazywanie tablicy w parametrach funkcji. Przekazanie tablicy o dowolnym rozmiarze:

```
void funkcja1(int tab1[], int rozmiarTab1)
{
  tab1[0] = 10;
}
```

Można również wpisać konkretny rozmiar tablicy do argumentu formalnego. W przypadku tablic nie ma potrzeby przekazywania tablicy przez referencję.

Stała tablica oznacza, że nie mogą być zmieniane wartości elementów.

Zwracanie tablicy. W C nie jest możliwe zwrócenie tablicy w rezultacie funkcji. Sposobem na zwrócenie tablicy jest podanie tablicy wynikowej w argumentach funkcji.

```
void funkcja1(int tab1[], int sizeTab1,
  int tabResult[], int sizeTabResult)
{
  tabResult[0] = tab1[0];
}
```

Tablice wielowymiarowe

Inicjalizacja tablicy dwuwymiarowej:

```
int tab[][3] = {{1, 1, 3}, {2, 3, 3}};
```

Inicjalizacja tablicy trójwymiarowej:

```
int tab[][1][1] = {{{1}}, {{2}}};
```

Przy inicjalizacji nie jest konieczne podawanie ilości wierszy tablicy, ponieważ wystarczy podać te rozmiary aby można było później dostać się do elementu tablicy [i][j]. Dlatego musimy podać liczbę kolumn, a nie musimy liczby wierszy.

Inicjalizacja tablicy dwuwymiarowej za pomocą pętli:

```
int iloscWierszy = 2;
int iloscKolumn = 3;
int tab[iloscWierszy][iloscKolumn] = {{1, 1, 3}, {2, 3, 3}};
for (int i = 0; i < iloscWierszy; i++)
{
  for (int j = 0; j < iloscKolumn; j++)
  {
    tab[i][j] = 0;
  }
}
```

Stałe napisowe. „Napis” jest tablicą znaków zakończoną znakiem '\0'. Do biblioteki standardowej należą funkcje, które operują na tablicach znaków, np. funkcja strlen zwracająca długość napisu. Porównywanie napisów zależne od implementacji.

Sito Eratostenesa

Za pomocą sita Eratostenesa znajdujemy liczby pierwsze z przedziału od 2 do określonej liczby.

```
int tabSize = 100;
_Bool tab[tabSize];
tab[0] = false;
tab[1] = false;
for (int i = 2; i < tabSize; i++)
{
  tab[i] = true;
}
int upperLimit = sqrt(tabSize);
for (int i = 2; i <= upperLimit; i++)
{
  if (tab[i] == true)
  {
    for (int j = i + i; j <= tabSize; j += i)
    {
      tab[j] = false;
    }
  }
}
for (int i = 0; i <= tabSize; i++)
{
  if (tab[i] == true)
  {
    printf("%d ", i);
  }
}
```

1.6.2 Zadania

Zadania na 3.0

a) Odczytac ze standardowego wejscia kilka napisów i umieścić je w tablicy dwuwymiarowej. Utworzyć drugą tablicę dwuwymiarową, w której należy umieścić palindromy do podanych na wejściu napisów. Wyświetlić tablicę wynikową.

Zadania na 4.0

Napisać funkcję, która przyjmuje tablicę wypełnioną 0 lub 1, i dodaje binarnie liczbę 1 i zwraca tablicę z cyframi wyniku dodawania. Napisać drugą funkcję, która dla każdej

liczby podanej w tablicy w postaci binarnej wypisuje ją na ekran. Parametrem tej funkcji jest rozmiar tablicy.

Zadania na 5.0

Tablica dwuwymiarowa może być zaimplementowana za pomocą tablicy jednowymiarowej. Zaimplementować funkcję get przyjmującą tablicę jednowymiarową, w której znajdują się elementy tablicy dwuwymiarowej, która ponadto przyjmuje współrzędne x, y tablicy dwuwymiarowej i zwraca wartość komórki x, y tej tablicy. Zadanie dodatkowe: to samo dla tablicy trójwymiarowej.

1.7 Wyszukiwanie i sortowanie

1.7.1 Wstęp teoretyczny

1.7.2 Algorytmy wyszukiwania

Sprawdzenie czy element znajduje się w tablicy:

```
_Bool sprawdzElement(int tab[], int size, int element)
{
  for (int i = 0; i < size; i++)
  {
    if (tab[i] == element)
    {
      return true;
    }
  }
  return false;
}
```

Znajdowanie najmniejszego i największego elementu w zbiorze.

```
void findExtrema(int tab[], int size)
{
  int max = tab[0];
  int maxIndex = 0;
  int min = tab[0];
  int minIndex = 0;
  for (int i = 1; i < size; i++)
  {
    if (tab[i] > max)
    {
      max = tab[i];
      maxIndex = i;
    }
```

```
    if (tab[i] < min)
    {
      min = tab[i];
      minIndex = i;
    }
  }
  printf("min: %d minIndex: %d\n", min, minIndex);
  printf("max: %d maxIndex: %d\n", max, maxIndex);
}
```

Powyższy algorytm w przypadku pesymistycznym wykonuje maksymalnie $2\,(n+1)$ ($n = size$) operacji przypisania oraz $3\,(n-1)$ operacji porównania.

1.7.3 Algorytmy sortowania

Sortowanie bąbelkowe

```
void babelkowe(int tab[], int size)
{
  for (int j = 0; j < size - 1; j++)
  {
    for (int i = 0; i < size - 1; i++)
    {
      if (tab[i] > tab[i+1])
      {
        int temp = tab[i];
        tab[i] = tab[i+1];
        tab[i+1] = temp;
      }
    }
  }
}
```

Sortowanie bąbelkowe polega na sortowaniu par elementów leżących obok siebie. Analizujemy kolejno po kolei wszystkie pary. Czynność tą powtarzamy. Można zauważyć, że po każdym pojedynczym przejściu będziemy mieć o jeden więcej element posortowany patrząc od końca tablicy. A więc w kolejnych przejściach nie trzeba sortować końcowych elementów:

```
void babelkowe(int tab[], int size)
{
  for (int j = size - 1; j > 0; j--)
  {
    for (int i = 0; i < j; i++)
    {
      if (tab[i] > tab[i+1])
```

```
        {
          int temp = tab[i];
          tab[i] = tab[i+1];
          tab[i+1] = temp;
        }
      }
    }
}
```

Liczba porównań wynosi:

$$\frac{n^2 - n}{2} - 1$$

Dla pesymistycznego przypadku liczba zamian wynosi tyle samo. Przypadek pesymistyczny zachodzi, gdy elementy tablicy są posortowane wcześniej w odwrotnej kolejności.

Sortowanie przez wstawianie

```
void wstawianie(int tab[], int size)
{
  int element;
  for (int i = 1; i < size; i++)
  {
    element = tab[i];
    int j;
    for (j = i - 1; j >= 0 && tab[j] > element; j--)
    {
      tab[j + 1] = tab[j];
    }
    tab[j + 1] = element;
  }
}
```

Algorytm wstawiania polega na podziale tablicy na dwie części, gdzie pierwsza część jest posortowana. Na początku pierwsza część zawiera tylko pierwszy element. Następnie wstawiamy kolejny element z drugiej części do pierwszej w odpowiednie miejsce, przesuwając odpowiednio elementy za wstawianym miejscem z pierwszej części, itd. Podany wyżej algorytm zawiera taką optymalizację, że wyszukiwanie miejsca gdzie należy wstawić element i przesuwanie elementów jest wykonywane w tej samej pętli. Maksymalna liczba operacji porównywania elementów tablicy:

$$\frac{n^2 - n}{2}$$

Maksymalna liczba operacji przypisań

$$\frac{n^2 - n}{2} + n - 1$$

Sortowanie metodą wstawiania połówkowego

```
int znajdzMiejsce(int limitGorny, int tab[], int size, int element)
{
  int lewy = 0;
  int prawy = limitGorny;
  while (lewy <= prawy)
  {
    int m = (lewy + prawy) / 2.0;
    if (element < tab[m])
    {
      prawy = m - 1;
    } else
    {
      lewy = m + 1;
    }
  }
  return lewy;
}

void wstawianiePolowkowe(int tab[], int size)
{
  int element;
  for (int i = 1; i < size; i++)
  {
    element = tab[i];
    int j;
    int miejsce = znajdzMiejsce(i - 1, tab, size, element);
    for (j = i - 1; j >= miejsce; j--)
    {
      tab[j + 1] = tab[j];
    }
    tab[miejsce] = element;
  }
}
```

Sortowania metodą wstawiania połówkowego jest modyfikacją sortowania przez wstawianie, polegającą na wstawianiu elementu za pomocą algorytmu połówkowego. Algorytm połówkowy polega na tym, że porównujemy element, który chcemy wstawić z elementem leżącym w połowie przedziału. Przez to porównanie wiemy, w której połowie znajduje się miejsce wstawienia. Następnie wybraną połowę jeszcze raz dzielimy na pół, itd. Aż otrzymamy miejsce, gdzie należy wstawić element.

Sortowanie przez wybór

```
void wybor(int tab[], int size)
{
  for (int i = 0; i < size - 1; i++)
  {
    int minIndex = i;
    for (int j = i + 1; j < size; j++)
    {
      if (tab[j] < tab[i])
      {
        minIndex = j;
      }
    }
    if (minIndex != i)
    {
      int temp = tab[i];
      tab[i] = tab[minIndex];
      tab[minIndex] = temp;
    }

  }
}
```

W sortowaniu przez wybór również tablica podzielona jest na dwie części, pierwsza część jest posortowana. Wyszukujemy minimum z drugiej części i zamieniamy to minimum z pierwszym elementem drugiej części i ten element staje się elementem pierwszej części. Maksymalna liczba porównań elementów tablicy:

$$\frac{(n-1)\,n}{2}$$

Maksymalna liczba zamian wynosi:

$$n-1$$

1.7.4 Zadania

Zadania na 3.0

Zrób zestawienie liczby operacji w przypadku najbardziej optymistycznym dla metod sortowania przedstawionych na zajęciach. Wskaż wszystkie przypadki optymistyczne dla każdej metody.

Zadania na 4.0

Przeanalizuj dla jakiego rodzaju danych każda z metod jest najszybsza i dlaczego.

Zadania na 5.0

Dodaj do sortowania bąbelkowego dwa udoskonalenia: 1. zapamiętanie czy dokonywano w poprzednim kroku jakiejkolwiej zmiany, jeśli nie dokonano to algorytm może się zakończyć, 2. zapamiętać pozycję ostatniej zmiany, i wykorzystać ją do pominięcia sprawdzania par, które występują wcześniej.

Zadania na 6.0

Zaimplementuj sortowanie mieszane, które jest modyfikacją sortowania bąbelkowego, która polega na zmiany kierunku kolejnych przejść.

1.8 Złożoność obliczeniowa

1.8.1 Wstęp teoretyczny

Złożoność obliczeniowa algorytmu to ilość zasobów komputerowych potrzebnych do jego wykonania. Podstawowymi rozważanymi zasobami są czas działania i ilość zajmowanej pamięci. Złożoność obliczeniowa to funkcja rozmiaru danych wejściowych, np dla algorytmów sortowania za rozmiar przyjmuje się liczbę elementów w ciągu wejściowym.

Na złożoność obliczeniową składa się **złożoność pamięciowa** i **złożoność czasowa**. Jednostką złożoności pamięciowej będzie słowo pamięci maszyny. Jednostką złożoności czasowej będzie jedna **operacja dominująca**. Łączna liczba operacji dominujących jest proporcjonalna do liczby wykonań wszystkich operacji jednostkowych. Dla algorytmów sortowania przyjmuje się za operację dominującą zwykle porównanie dwóch elementów w ciągu wejściowym, a czasem też przestawienie elementów w ciągu. Dla algorytmów numerycznych operacjami dominującymi są dodawanie i mnożenie.

Wyróżniamy złożoność pesymistyczną i oczekiwaną. **Złożoność pesymistyczna** definiowana jest jako ilość zasobów komputerowych potrzebnych przy „najgorszych" danych wejściowych. **Złożoność oczekiwana** definiowana jest jako ilość zasobów komputerowych potrzebnych przy „typowych" danych wejściowych.

Dokładna definicja złożoności pesymistycznej:
D_n - zbiór zestawów danych wejściowych rozmiaru n,
$t(d)$ - liczba operacji dominujących dla zestawu danych wejściowych d,
Pesymistyczna złożoność czasowa algorytmu to funkcja:

$$W(n) = \sup\{t(d) : d \in D_n\}$$

Oczekiwana złożoność czasowa w przypadku gdy wszystkie dane są jednakowo prawdopodobne (z prawdpodobieństwem niezerowym) wyraża się wzorem:

$$A(n) = \frac{\sum_{d \in D_n} t(d)}{|D_n|}$$

Przykład. Dla algorytmu wyszukiwania elementu w tablicy z poprzednich zajęć operacją dominującą jest porównanie, złożoność pesymistyczna wynosi n. Jeśli element

zostanie znaleziony na k-tym miejscu to liczba operacji dominujących wynosi k. Zatem liczba operacji dominujących waha się w granicy od 1 do n. A zatem złożoność oczekiwana wynosi:

$$A(n) = \frac{1 + 2 + \ldots + n}{n} = \frac{n+1}{2}$$

Notacja „O"

Faktyczna złożoność czasowa algorytmu w chwili jego użycia różni się od wyliczonej teoretycznie współczynnikiem proporcjonalności, który zależy od konkretnej realizacji tego algorytmu. Istotną informacją zawartą w pesymistycznej i oczekiwanej złożoności czasowej jest rząd wielkości, czyli zachowanie asymptotyczne, gdy n dąży do nieskończoności. Zwykle staramy się podać jak najprostszą funkcję charakteryzującą rząd wielkości $W(n)$ i $A(n)$, na przykład n, $n \log n$, n^2, n^3. Niech:

$$f, g, h : N \to R_+ \cup \{0\}$$

Funkcja f jest **co najwyżej rzędu g**, co zapisujemy $f(n) = O(g(n))$, jeśli istnieją stała rzeczywista $c > 0$ i stała naturalna n_0 takie, że nierówność $f(n) \leq cg(n)$ zachodzi dla każdego $n \geq n_0$.
Przykład: $n^2 + n = O(n^2)$, ponieważ $n^2 + 2n \leq 3n^2$.
Funkcja f jest **co najmniej rzędu g**, co zapisujemy $f(n) = \Omega(g(n))$, jeśli $g(n) = O(f(n))$.
Funkcja f jest **dokładnie rzędu g**, co zapisujemy $f(n) = \Theta(n)$, jeśli zarówno $f(n) = O(g(n))$, jak i $f(n) = \Omega(g(n))$. Poprawny jest też termin f jest asymptotycznie równoważne g i oznaczenie $f(n) \cong g(n)$. Przykład: $n^2 + 2n \cong n^2$, ponieważ $n^2 + 2n \leq 3n^2$ jak i $n^2 + 2n \geq n^2$ dla każdego $n > 0$.

Uproszczenie funkcji opisującej rząd wielkości polega na pominięciu wszystkich składników funkcji oprócz tego, który ma największy wpływ, pominięciu wszelkich stałych współczynników.

Popularne rzędy wielkości:

- złożoność stała $O(1)$ - stała liczba operacji dominujących bez względu na rozmiar danych wejściowych Przykład sprawdzenia czy liczba jest parzysta.
- złożoność logarytmiczna $O(\log n)$ - Np algorytmy w których problem postawiony dla danych rozmiaru n da sie sprowadzić do problemu z rozmiarem danych o połowę mniejszym. Przykład: znalezienie elementu w posortowanej tablicy za pomocą binarnego wyszukiwania.
- złożoność liniowa, $O(n)$ - dla każdej danej wykonywana jest stała liczba operacji, np znalezienie elementu w nieposortowanej liście
- złożoność liniowo-logarytmiczna, $O(n \log n)$, Np. gdy problem o rozmiarze n da się sprowadzić do problemu o rozmiarze n/2 w n krokach
- złożoność kwadratowa: np algorytm sortowanie przez wstawianie

- złożoność wykładnicza 2^n: sprawdzenie czy dla danej formuły logicznej istnieje takie podstawienie zmiennych zdaniowych, żeby formuła była prawdziwa
- złożoność wykładnicza $n!$: np. znalezienie optymalnego rozwiązania problemu komiwojażera(znaleźć najkrótszą drogę z miasta A do B, przechodząc przez każde miasto tylko raz)

1.8.2 Zadania

Zadania na 3.0

Podaj pesymistyczną złożoność algorytmów sortowania rozpatrywanych na poprzednich zajęciach w notacji duże O.

Zadania na 4.0

Określić liczbę operacji dominujących w zwykłym wyszukiwaniu i wyszukiwaniu z wartownikiem, polegającym na dodatniu na końcu tablicy elementu poszukiwanego i pominięciu sprawdzania, czy doszliśmy do końca tablicy. Podać złożoność obliczeniową tych dwóch algorytmów w notacji duże O. O ile skraca się czas wyszukiwania?

Zadania na 5.0

Udowodnij, że dolnym ograniczeniem przeszukiwania binarnego jest $\log_2 n$.

1.9 Złożoność obliczeniowa, cd

1.9.1 Wstęp teoretyczny

Theorem 1.9.1. *Gdy*

$$f(n) = an^2 + bn + c$$

gdzie a,b,c są stałymi, przy czym a > 0 zachodzi:

$$f(n) = \Theta\left(n^2\right)$$

Theorem 1.9.2. *Gdy*

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

gdzie a_i są stałymi i $a_d > 0$, mamy:

$$p(n) = \Theta\left(n^d\right)$$

Theorem 1.9.3. *Gdy*

$$f(n) = an + b$$

to

$$f(n) = O\left(g\left(n^2\right)\right)$$

Dowód.

$$c = a + |b|$$

$$an + b <= (a + |b|) n^2$$

dla $n_0 = 1$ □

Uwaga: Dla dwóch funkcji $f(n)$ i $g(n)$ może nie wystąpić żaden z dwóch przypadków: ani $f(n) = O(g(n))$, ani $f(n) = \Omega(g(n))$. Na przykład funkcji n oraz $n^{1+\sin n}$ nie można w tym sensie porównać. Wartość wykładnika waha się od 0 do 2.

Analiza czasu działania algorytmu przez wstawianie

Pesymistyczny czas działania wynosi $O(n^2)$. Optymistyczny czas działania wynosi $\Omega(n)$. Oszacowania te są najlepsze z możliwych.

Notacja o

$$o(g(n)) = \{\forall_{c>0}\exists_{n_0>0}\forall_{n\geq n_0}\ 0 \leq f(n) < cg(n)\}$$

Ograniczenie to nazywamy asymptotycznie dokładnym.

$$2n = o\left(n^2\right)$$

$$2n^2 \neq o\left(n^2\right)$$

Notacja ω

$$\omega(g(n)) = \{\forall_{c>0}\exists_{n_0>0}\forall_{n\geq n_0}\ 0 \leq cg(n) < f(n)\}$$

Przykładowo:

$$\frac{n^2}{2} = \omega(n)$$

$$\frac{n^2}{2} \neq \omega\left(n^2\right)$$

Własności notacji asymptotycznych

Przechodniość, zwrotność, symetria w przypadku notacji Θ, symetria transpozycyjna.

1.9.2 Zadania

Zadania na 3.0

Wyjaśnij dlaczego stwierdzenie „Czas działania algorytmu A wynosi co najmniej $O(n^2)$ nie ma sensu.

Zadania na 4.0

Pokaż, że dla dowolnych rzeczywistych stałych a i b, gdzie b > 0, zachodzi

$$(n+a)^b = \Theta\left(n^b\right)$$

Zadania na 5.0

Sprawdź czy

$$2^{n+1} = O(2^n)$$

oraz

$$2^{2n} = O(2^n)$$

Zadania na 6.0

Uporządkuj następujące funkcje ze względu na ich rząd wielkości.

$$\log_2(log_2{*n}), 2^{\log_2 *n}, \left(\sqrt{2}\right)^{\log_2 n}, n^2, n!, (\log_2 n)!$$

$$\left(\frac{3}{2}\right)^n, n^3, \log_2^2 n, log_2(n!), 2^{2^n}, n^{\frac{1}{\log_2 n}}$$

$$\ln\ln n, \log_2 *n, n2^n, n^{\log_2 \log_2 n}, \ln n, 1$$

$$2^{\log_2 n}, (\log_2 n)^{\log_2 n}, e^n, 4^{\log_2 n}, (n+1)!\frac{1}{\sqrt{\log_2 n}}$$

$$\log_2 * (\log_2 n), 2^{\sqrt{2\log_2 n}}, n, 2^n, n\log_2 n, 2^{2^{n+1}}$$

Wyrażenie $\log_2 *n$ oznacza tzw. logrytm iterowany i jest zdefiniowane następująco:

$$\log_2 *n = \min\left\{i \geq 0 : log_2^{(i)} n \leq 1\right\}$$

przy czym zapis $f^{(i)}n$ oznacza funkcję $f(n)$ zastosowaną iteracyjnie i razy do wartości początkowej n.

1.10 Wskaźniki

1.10.1 Wstęp teoretyczny

Wskaźnik to zmienna, której wartością jest adres pewnego obszaru pamięci, a zadaniem - wskazywanie na inne zmienne. Przykład deklaracji zmiennej wskaźnikowej:

```
int *wsk1;
```

Zmienna wskaźnikowa nazwa1 wskazuje na zmienne typu int. Wyświetlenie wartości wskaźnika oznacza wyświetlenie adresu, który jest zapisany w systemie szesnastkowym:

```
printf("%p\n", wsk1);
```

Operator jednoargumentowy & zwraca adres argumentu.

Operator jednoargumentowy * jako argument przyjmuje adres zmiennej i zwraca wartość zmiennej dostępnej pod tym adresem. Np. wyrażenie $*\&i$ zwróci wartość zmiennej i.

Przykład programu ze zmienna wskaźnikową:

```
int i = 5;
int *wsk = &i;
printf("%d\n", *wsk);
return 0;
```

Wskaźnik może zmieniać swoją wartość, co oznacza, że wskaźnik może wskazywać na inną zmienną o tym samym typie. Przykład:

```
int i = 5;
int j = 8;
int *wsk = &i;
printf("%d\n", *wsk);
wsk = &j;
printf("%d\n", *wsk);
```

Za pomocą operatora * dostajemy również pełny dostęp do wskazywanych zmiennych, tzn. możemy np zmienić wartość zmiennej wskazywanej w następujący sposób:

```
int i = 5;
int *wsk = &i;
*wsk = 8;
printf("%d\n", i);
```

W tym przypadku *wsk traktowane jest jako miejsce w pamięci do którego wpisywana jest liczba 8.

Operator przypisania dla wskaźników. Przypisanie wskaźnika do wskaźnika skutkuje skopiowaniem wartości wskaźnika, a więc adresu. Przykład:

```
int i = 5;
int j = 8;
```

```
  int *wsk1 = &i;
  int *wsk2 = &j;
  wsk1 = wsk2;
  printf("%d %d\n", *wsk1, *wsk2);
  printf("%d %d\n", i, j);
```

Podobnie do operatora przypisania działa inicjalizacja wskaźników i tak możemy mieć np taką inicjalizację:

```
int *wsk2 = wsk1;
```

Powyższe oznacza, że wskaźnik wsk2 będzie wskazywał na to samo co wsk1.

Operator porównywania dla wskaźników. Operator ten porównuje wartości wskaźników, a więc adresy. Przykład:

```
int i = 5;
  int j = 5;
  int *wsk1 = &i;
  int *wsk2 = &j;
  int *wsk3 = &i;
  printf("%d %d\n", wsk1 == wsk2, wsk1 == wsk3);
```

Program powyższy zwróci 0 i 1.

Przekazywanie argumentów funkcji przez wskaźniki

Przekazanie wskaźnika do funkcji oznacza kopiowanie adresu na stos. Przykład:

```
void funkcja(int * liczba1)
{
  *liczba1 = 5;
}
int main()
{
  int liczba = 1;
  funkcja(&liczba);
  printf("%d\n", liczba);
  return 0;
}
```

Stały wskaźnik. Stały wskaźnik to taki wskaźnik, że nie można zmienić wskazywanego miejsca na inne. Przykład

```
void funkcja(int * const liczba1)
{
  *liczba1 = 5;
  //int i = 3;
  //liczba1 = &i;
}
```

Druga zakomentowana instrukcja powoduje błąd kompilacji. Do powyższej funkcji może zostać przekazany wskaźnik, który niekoniecznie jest stały, ale we funkcji będzie traktowany jako stały.

Oprócz stałego wskaźnika stałą może być zmienna, na którą wskazuje wskaźnik, co oznacza, że nie możemy zmieniać jej wartości. Przykład:

```
void funkcja3(const int * liczba1)
{
  //*liczba1 = 5;
}
```

Instrukcja zakomentowana jest zabroniona. Obie powyższe stałe mogą być ze sobą łączone, co oznacza, że możemy np przekazać do funkcji parametr formalny postaci:

```
void funkcja(const int * const liczba1);
```

Porównanie przekazywania parametrów do funkcji przez wskaźniki i przez referencję

Referencje to stałe wskaźniki, a więc są zaimplementowane jako int * const p. Nie można zatem zmienić wskazywanego miejsca na inne. Stałe wskaźniki, a więc również referencje muszą być inicjalizowane, a więc można powiedzieć, że są bezpieczniejsze od wskaźników.

1.10.2 Zadania

Zadania na 3.0

Zaimplementuj dwie wersje algorytmu Euklidesa, tak aby obliczenia były oparte na wskaźnikach.

Zadania na 4.0

a) Czy poniższa instrukcja jest poprawna? Co powoduje?

```
int *wsk3 = 4;
```

b) Czy poprawne są poniższe instrukcje? Kiedy możemy ich używać? Co zostanie wypisane na ekran?

```
int liczba = 1;
int *wsk1 = &liczba;
int *wsk2 = &wsk1;
printf("%d\n", wsk1);
printf("%d\n", *wsk2);
```

Zadania na 5.0

Czy referencje w C++ zabezpieczają w 100% przed błędami pracy na nieznanym obszarze pamięci, tak jak to może mieć miejsce w przypadku wskaźników?

1.11 Wskaźniki, cd.

1.11.1 Wstęp teoretyczny

W języku C nazwa tablicy jest jednocześnie wskaźnikiem do jej pierwszego elementu.

```
int tab[4] =
{ 3, 0, 1, 2 };
int *wsk;
wsk = tab;
printf("%d\n", *wsk);
```

Powyższy fakt oznacza, że

`tab == &tab[0].`

Drugim wnioskiem jest to, że podczas przekazywania tablicy do funkcji przekazujemy tak naprawdę wskaźnik na pierwszy element.

Dodanie do wskaźnika 1 oznacza przesunięcie wskaźnika o jedną pozycję do przodu, przykładowo jeśli wskaźnik wskazuje na liczby typu int, dodanie do wskaźnika 1 oznacza przesunięcie wskaźnika w pamięci o jedną liczbę int do przodu, czyli przy reprezentacji int na 4 bajtach o 4 bajty. Podobnie zdefiniowana jest operacja dodawania lub odejmowania dowolnej liczby całkowitej do wskaźnika.

Tablice w C zajmują jednolity obszar pamięci. Dlatego, za pomocą operacji dodawania liczby całkowitej do wskaźników możemy przechodzić pomiędzy elementami tablicy. Przykład wypełnienie tablicy liczbą:

```
wsk = tab;
for (int i = 0; i < 4; i++)
{
  *wsk = 7;
  wsk++;
}
```

W przypadku zwyczajnego wypełnienia za każdym razem obliczany jest adres wypełnienia za pomocą dodania do adresu pierwszego elementu odpowiedniej wielokrotności wielkości przechowywanych zmiennych w tablicy. W przypadku użycia wskaźników adres wypełnienia jest już znany od razu, choć w każdym obiegu pętli musi być aktualizowany za pomocą inkrementacji.

W związku z tym, że zmienna tablicowa jest adresem pierwszego elementu tablicy, to możemy do takiej zmiennej przekazać dowolny element tablicy, który we funkcji traktowany jest jako pierwszy:

```
void function(int tab[4])
{
  //empty
}
int main()
```

```
{
  function(&tab[2]);
  return 0;
}
```

Tablice mogą również przechowywać wskaźniki. Zastosowaniem takiej konstrukcji może być np. przechowywanie napisów o dowolnym rozmiarze. Tablica dwuwymiarowa pozwalała na przechowywanie napisów o tym samym rozmiarze. Przykład:

```
char *napisy[5] = {"napis1", "napis2", "napis3",
"napis4", "napis5"};
printf("%s\n", napisy[2]);
printf("%c\n", napisy[2][2]);
printf("%c\n", *napisy[2]);
char **wsk2;
wsk2 = &napisy[2];
printf("%c\n", **wsk2);
```

W powyższym przykładzie występuje typ wskaźnik do wskaźnika.

Wskaźnik na typ void. Wskaźnik taki nie ma informacji na jaki typ danych wskazuje, konsekwencją jest to, że możemy za jego pomocą zapamiętywać adresy, ale nie możemy odczytać tego na co wskazuje. Przykład:

```
void *wsk3;
int i = 3;
double d2 = 1.0;
wsk3 = &i;
wsk3 = &d2;
printf("%f\n", *(double *)wsk3);
```

Wskaźnik na adres pusty NULL. Kiedy chcemy aby wskaźnik nie wskazywał na żadne miejsce w pamięci, możemy mu przypisać wartość NULL.

1.11.2 Zadania

Zadania na 3.0

Zdefiniuj tablicę napisów, w której napisy zawierają nazwy miesięcy. Wydrukuj te napisy. Przekaż tablicę do funkcji, która wydrukuje te napisy.

Zadania na 4.0

Zrobić eksperyment, jak działa odejmowanie od siebie wskaźników w przypadku tablic. Czy dodawanie wskaźników ma sens?

Zadania na 5.0

Gdzie można wykorzystać wskaźniki typu void? Podaj konkretny przykład.

1.12 Pliki tekstowe. Przeładowanie nazw funkcji

1.12.1 Wstęp teoretyczny

Pliki tekstowe

Struktura zawierająca informacje o pliku: FILE. Funkcja fopen pozwala na otwarcie pliku. Funkcja fopen zwraca wskaźnik do struktury FILE. Jako argumenty przyjmuje ścieżkę do pliku, oraz parametry otwarcia takie jak: „r" w trybie czytania pliku, „w" w trybie pisania, „a" w trybie dopisywania. Podczas uruchamiania programu otwierane są 3 pliki: stdin, stdout, stderr. Funkcja fclose służy do zamknięcia pliku.

```
FILE *fopen(char *name, char *mode)
int fclose(FILE *fp)
```

Funkcja getc służy do przeczytania jednego znaku z pliku. Jako znak końca pliku zwraca stałą EOF.

```
int getc(FILE *fp)
```

Funkcja putc natomiast zapisuje znak c do pliku, i zwraca zapisany znak lub EOF jesli wystąpił błąd.

```
int putc(int c, FILE *fp)
```

Możemy do czytania z pliku i zapisania w nim danych w postaci odpowiednio sformatowanej wykorzystać funkcję fscanf i fprintf.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Gdy chcemy odczytać lub zapisać kolejny wiersz do pliku możemy skorzystać z funkcji fgets i fputs. Funkcja fgets czyta co najwyżej $maxline - 1$ znaków. Funkcja fgets czyta również znak końca wiersza. Funkcja fputs nie powinna zawierać znaku nowego wiersza, ponieważ jest on automatycznie dopisywany.

```
char *fgets(char *line, int maxline, FILE *fp)
int fputs(char *line, FILE *fp)
```

Przeciążanie nazw funkcji w C++

Przeciążanie nazw funkcji jest możliwe w C++ i nie jest możliwe w C. Możliwe jest deklarowanie funkcji o tych samych nazwach ale różnych argumentach formalnych. Podczas podawania argumentów aktualnych do funkcji kompilator musi rozpoznać którą funkcję wywołać. W tym celu stosuje w podanej kolejności zestaw kryteriów:

1. Ścisła zgodność.
2. zgodność z zastosowaniem promowania w zakresie typów całościowych: bool na int, char na int, short na int, i odpowiedniki bez znaku, oraz float na double

3. zgodność z zastosowaniem standardowych konwersji: int na double, double na int, double na long double, T* na void*, int na unsigned int

4. zgodność z zastosowaniem konwersji zdefiniowanych przez użytkownika

5. zgoność z zastosowaniem wielokropka w definicji funkcji

Przykład:

```
void drukuj(int)
void drukuj(const char*)
void drukuj(double)
void drukuj(long)
void drukuj(char)

void h(char c, int i, short s, float f)
{
  drukuj(c); //drukuj(char)
  drukuj(i); //drukuj(int)
  drukuj(s); //promocja w zakresie typów całościowych drukuj(int)
  drukuj(f); //promocja z float na double drukuj(double)
  drukuj('a'); //drukuj(char)
  drukuj(49); //drukuj(int)
  drukuj(0); //drukuj(int)
  drukuj("'a"'); //drukuj(const char*)
}
```

1.12.2 Zadania

Zadania na 3.0

Zaimplementować łączenie dwóch plików tekstowych.

Zadania na 4.0

Zaimplementować funkcję getline, tak aby czytala z pliku wiersz i zwracała jego długość.

Zadania na 5.0

Zaimplementować porównywanie dwóch plików. Program powinien rozpoznawać zdania i zwracać te zdania, które różnią się na więcej niż 90% znaków zdania.

1.13 Różnice między językami C i C++

1.13.1 Wybrane konstrukcje poprawne w C, a niepoprawne w C++.

W C wskaźnik void może być przyporządkowany do dowolnego wskaźnika bez konieczności konwersji jawnej, w C++ nie może.

```
void *ptr;
int *i = ptr;
```

lub

```
int *j = malloc(sizeof(int)*5);
```

Aby skompilować powyższy kod w C++ należy dokonać jawnej konwersji:

```
void *ptr;
int *i = (int *) ptr;
int *j = (int *) malloc(sizeof(int)*5)
```

Konstrukcje używające słów kluczowych języka C++ nie występujących w C, np:

```
struct template
{
  int new;
  struct template *class;
}
```

Zwalnianie tablic jest odmienne w C++ Przykład C:

```
int *x = malloc(sizeof(int));
int *x_array = malloc(sizeof(int)*10);
free(x);
free(x_array)
```

Natomiast w C++:

```
int *x = new int;
int *x_array = new int[10];
delete x;
delete[] x;
```

1.13.2 Konstrukcje poprawne w C i C++, ale zachowujące się inaczej

Stałe znakowe w C są typu int, a w C++ są typu char, co powoduje, że sizeof('a') zwróci różne wyniki.

Zarówno C w wersji C99 jak i C++ posiadają typ bool ze stałymi true i false. W C++ typ bool jest typem wbudowanym i zarezerwowanym słowem języka, w C99 został wprowadzony typ _Bool, jest to typ unsigned int ale rzutowany do wartości 0 i 1. w nagłówku stdbool.h jest zdefiniowany typ bool, do którego jest zmapowany typ _Bool.

1.13.3 Konstrukcje poprawne w C++, a niepoprawne w C

Przeciążanie nazw funkcji dostępne jest jedynie w C++.

We funkcji main w C++ nie musi pojawić się intstrukcja return 0;, jest ona domniemana automatycznie, w C musi.

Definiowanie struktur. W C++ zmienna typu strukturalnego nie ma słowa kluczowego struct przy podaniu typu zmiennej:

```
struct a_struct
{
  int x;
};
a_struct struct_instance;
```

W C musimy zapisać:

```
struct a_struct struct_instance;
```

Podobna sytuacja zachodzi dla enum.

Tablica 1.1: Możliwe proste typy danych w C

typ	rozmiar w bajtach
void	1
char	1
signed char	1
unsigned char	1
short	2
signed short	2
short int	2
signed short int	2
unsigned short	2
unsigned short int	2
int	4
signed	4
signed int	4
unsigned	4
unsigned int	4
long	4
signed long	4
long int	4
signed long int	4
unsigned long	4
unsigned long int	4
long long	8
signed long long	8
long long int	8
signed long long int	8
unsigned long long	8
unsigned long long int	8
float	4
double	8
long double	12
_Bool	1
float _Complex	8
double _Complex	16
long double _Complex	24

Rozdział 2

Zaawansowane programowanie

2.1 Programowanie obiektowe

2.1.1 Paradygmaty programowania

Programowanie proceduralne to paradygmat programowania bazujący na koncepcji wywoływania procedur (funkcji). Procedury zawierają listę instrukcji, które będą wykonane po wywołaniu procedury. Każda procedura może być wywołana w każdym momencie w trakcie wykonywania programu przez inne procedury, lub przez nią samą. Zalety programowania proceduralnego:

- możliwość re-używania tego samego kodu, bez konieczności kopiowania go
- łatwiejsze śledzenie wykonywania programu niż programu z instrukcją GOTO

Programowanie proceduralne związane jest z dwoma paradygmatami: *z modularnością*, która oznacza podział programu na oddzielne części, w tym wypadku są to procedury. Każda z tych części ma wyspecyfikowanee wejście jako listę argumentów, oraz wyjście jako zwracaną wartość; *z zakresowością* oznaczającą w tym przypadku brak dostępu do lokalnych zmiennych procedury przez inne procedury, albo nią samą w kolejnych wywołaniach.

Programowanie obiektowe to paradygmat programowania, w którym używa się "obiektów", czyli struktur danych składających się z atrybutów i metod, oraz interakcji pomiędzy nimi. Program obiektowy może być widziany jako kolekcja obiektów, które ze sobą kooperują, w przeciwieństwie do programowania proceduralnego, w którym program widziany jest jako kolekcja procedur do wykonania. Każdy obiekt może odbierać wiadomości, może wykonywać zadania i wysyłać wiadomości do innych obiektów. Obiekt enkapsuluje dane oraz metody, procedura nie zawiera na stałe danych (po wywołaniu procedury zmienne lokalne są tracone).

2.1.2 Fundamentalne pojęcia programowania obiektowego

Klasa definiuje abstrakcyjną charakterystykę przedmiotu, która zawiera atrybuty przedmiotu, czyli jego właściwości, oraz zachowanie przedmiotu (metody, które mogą być

wykonane przez przedmiot). Przykładowo klasa Pies może mieć atrybuty takie jak imie psa, rasa, kolor futra, wzrost, waga, oraz metody takie jak szczekaj, siad. Atrybuty i metody nazywane są *składnikami klasy*.

Obiekt jest to egzemplarz klasy. Przykładowo dla klasy Pies egzemplarzem będzie pies o imieniu Lara, z kolorem futra bialym, z konkretnymi wartościami wzrostu i wagi. Zbiór wartości atrybutów konkretnego obiektu jest nazywany *stanem obiektu*. *Instancja* jest obiektem, który już został stworzony w trakcie działania programu.

Metody są to możliwości działania obiektu. Metody definiowane są w klasie, ale wywoływane są na obiektach tej klasy, np. wywołanie metody szczekaj na konkretnym obiekcie odpowiadającym psu o imieniu Lara.

Przekazywanie wiadomości to przesłanie nazwy metody wraz z parametrami do drugiego obiektu i wywołanie tej metody przez drugi obiekt wraz ze zwróceniem wyniku do pierwszego obiektu.

Abstrakcja to uproszczenie złożonej rzeczywistości poprzez modelowanie klas odpowiednich dla danego problemu. Przykładowo w naszym programie stworzyliśmy klasę Pies o odpowiednich atrybutach, ponieważ posługiwanie się dokładnie tą klasą jest najbardziej odpowiednie dla zamodelowania naszego problemu. Abstrakcja jest realizowana również przez kompozycję. Przykładowo klasa Samochód zawiera Silnik, Skrzynię biegów, Kierownicę, i inne. Klasa Silnik jest abstrakcją, ponieważ interesują nas składowe tej klasy wymienione wyżej, wraz z metodami, które udostępniają, nie interesują nas z poziomu klasy Silnik szczegóły działania skrzyni biegów.

Enkapsulacja to ukrywanie szczegółów metod danego obiektu przed innymi obiektami. Inne obiekty do komunikacji z danym obiektem potrzebują jedynie interfejs, a więc deklaracje metod. Enkapsulacja powoduje, że w przyszłości może być zmienione działanie metod bez konieczności zmiany kodu innych obiektów wywołujących te metody.

2.1.3 Zadanie lekcyjne

Dla wybranych przedmiotów wymyśleć zastosowanie w systemie informatycznym. Zapisać klasę odpowiadającą danemu przedmiotowi wraz z listą atrybutów oraz metod. Atrybuty powinny składać się z typu oraz nazwy. Metody powinny składać się z nazwy, argumentów metody, oraz typu zwracanego. Ponadto należy w ten sam sposób opisać klasy składowe danej klasy, oraz stwierdzić ile obiektów danej klasy składowej będzie potrzebne dla wybranych zastosowań.

2.1.4 Zadania domowe

Zadanie obowiązkowe

Przepisać do zeszytu rozwiązanie zadania lekcyjnego, uzupełnić w miare potrzeby listę klas, atrybutów i metod. Przykład musi zawierać co najmniej jedną klasę składową, każda klasa powinna zawierać co najmniej jeden atrybut i co najmniej jedną metodę.

Zadanie na 6.0

Zadanie z XVI olimpiady informatycznej 2008/2009 z I etapu o nazwie Gaśnice.

Uwaga. Zadanie na 6.0 powinno być wysłane na maila wraz z opisem rozwiązania.

2.2 Programowanie obiektowe, cd.

2.2.1 Elementy programowania obiektowego w C++

- Definicje klas

```
class Auto
{
  private:
    int liczbaDrzwi;
  public:
    int podajLiczbeDrzwi() const;
};
int Auto::podajLiczbeDrzwi() const
{
  return liczbaDrzwi;
}
```

- Metody stałe. Dla obiektów stałych nie możemy wywoływać metod nie-stałych.

- Konstruktory. Przykład:

```
class Auto
{
  int liczbaDrzwi;

  public:
    Auto(int liczbaDrzwi)
    {
      this->liczbaDrzwi = liczbaDrzwi
    }
}
```

  Inicjatory pól. Konstruktor wywołuje niejawnie konstruktory domyślne składowych klasy.

- Składowe statyczne. Przykład:

```
class Auto
{
  public:
  static const int liczbaKol = 4;
```

```
};
```

- Samoodwołanie za pomocą wskaźnika this. Przykład metody zwracającej referencję do obiektu:

```
class Auto
{
  private:
    int liczbaDrzwi;
  public:
    Auto & ustawLiczbeDrzwi(int liczbaDrzwi = 0)
    {
      this->liczbaDrzwi = liczbaDrzwi;
      return *this;
    }
};
```

  W metodzie stałej wskaźnik this jest wskaźnikiem stałym (const X*).

- Wartości domyślne argumentów metod.
- Atrybut metod inline.

2.2.2 Operacje na obiektach

- tworzenie obiektów
- kopiowanie obiektów za pomocą operatora kopiowania, przeciążenie operatora kopiowania
- kopiowanie obiektów za pomocą konstruktora kopiującego, przeciążenie konstruktora kopiującego
- Porównywanie obiektów, operator porównywania

2.2.3 Dziedziczenie

Modelowanie związków hierarchicznych między przedmiotami, tzn. wspólnych cech klas za pomocą hierarchii klas. Przykładowo okrąg i trójkąt są pokrewne w tym sensie, że są figurami, a więc możemy zdefiniować klasę Figura, po której dziedziczą klasy Okrąg i Trójkąt.

2.2.4 Zadania domowe

Zadanie obowiązkowe

Skompilować kod z zadania lekcyjnego. Przepisać do zeszytu rozwiązanie zadania lekcyjnego.

Zadanie na 6.0

Zadanie z XVI olimpiady informatycznej 2008/2009 z I etapu o nazwie Straż pożarna.

Uwaga. Zadanie na 6.0 powinno być wysłane na maila wraz z opisem rozwiązania.

2.3 Programowanie obiektowe, cd.

2.3.1 Operacje na obiektach

- Destruktory wywoływane są podczas niszczenia obiektu, czyli przy opuszczaniu zasięgu przez tą zmienną. Destruktor służy do robienia porządków i zwalniania zasobów. Destruktory są wywoływane w porządku odwrotnym do konstruktorów.

2.3.2 Dziedziczenie

Metody wirtualne. Jeśli dana metoda ma taką samą nazwę w klasie podstawowej i pochodnej, to która metoda zostanie wywołana w kodzie operującym klasami podstawowymi? Zostanie wywołana metoda z klasy podstawowej. Istnieje możliwość wywołania metody z klasy danego obiektu, jeśli metoda jest wirtualna.

Polimorfizm (wielopostaciowość)- używanie wartości, zmiennych, podprogramów na kilka różnych sposobów, uzyskanie właściwego zachowania metody, niezależnie od tego który dokładnie typ z hierarchii zostanie użyty. *Polimorfizm* (wielopostaciowość)- używanie wartości, zmiennych, podprogramów na kilka różnych sposobów, uzyskanie właściwego zachowania metody, niezależnie od tego który dokładnie typ z hierarchii zostanie użyty.

- polimorfizm statyczny - czasu kompilacji, np przeciążanie operatorów
- polimorfizm dynamiczny - w trakcie wykonywania programu, metody wirtualne

2.3.3 Zadanie lekcyjne

Klasy nadrzędne i podrzędne wymyślone na poprzedniej lekcji zapisać w notacji C++. Dodać przeładowania przynajmniej dwóch wybranych dowolnie operatorów.

2.3.4 Zadanie domowe

Zadanie obowiązkowe

Skompilować kod z zadania lekcyjnego. Przepisać go do zeszytu.

Zadanie na 6.0

Zadanie z XVI olimpiady informatycznej 2008/2009 z I etapu o nazwie Kamyki.

Uwaga. Zadanie na 6.0 powinno być wysłane na maila wraz z opisem rozwiązania.

2.4 Programowanie obiektowe, cd.

2.4.1 Elementy programowania obiektowego w C++

- klasa abstrakcyjna
- metoda czysto wirtualna

2.4.2 Zadanie domowe

Zadanie obowiązkowe

Skompilować kod z kartkówki. Przepisać kod do zeszytu.

Zadanie na 6.0

Zadanie z XVI olimpiady matematycznej 2008/2009 z I etapu o nazwie Przyspieszenie algorytmu.

Uwaga. Zadanie na 6.0 powinno być wysłane na maila wraz z opisem rozwiązania.

2.5 Programowanie obiektowe, cd.

2.5.1 Zadanie domowe

Zadanie na 6.0

Zadanie z XVI olimpiady informatycznej 2008/2009 z II etapu o nazwie Wyspy na trójkątnej sieci.

Uwaga. Zadanie na 6.0 powinno być wysłane na maila wraz z opisem rozwiązania.

2.6 Wzorce w C++

2.6.1 Wstęp

Programowanie uogólnione - programowanie z użyciem typów jako parametrów.

2.6.2 Wzorce funkcji

- Przykład 1:

```
//T - parametr wzorca
template <typename T>
class List
{
...
};
List <int> lista1;
```

```
List <std::string> lista2;
List <MojaKlasa> lista_mojej_klasy;
```

Przykład 2:

```
#include <iostream>

template <typename T>
const T& max(const T& x, const T& y)
{
  if(y < x)
    return x;
  return y;
}

int main()
{
  // wywołanie max <int>
  std::cout << max(3, 7) << std::endl;
  // wywołanie max<double>
  std::cout << max(3.0, 7.0) << std::endl;
  // niejednoznaczność; bezpsrednie wywołanie max<double>
  std::cout << max<double>(3, 7.0) << std::endl;
  return 0;
}
```

- bezpośrednie wskazanie wzorca funkcji

2.6.3 Wzorce klasy

- Przykład:

```
template <typename T> class Foo2
{
public:
  void g(T & arg);
};
//definicja poza klasa
template <typename T> void Foo2<T>::g(T & arg)
{
  arg.getNumber();
}
```

- bezpośrednia specjalizacja wzorca - specjalizacja zdefiniowana przez użytkownika

```
template<typename T> void foo(const T& x)
{
  ... T nie jest int
}

template<> void foo<int>(const int& x)
{
  ... T jest int
}
```

- konkretyzacja wzorca podczas kompilacji
- zamiennik dla makr preprocesora
- trudności kompilatora z wygenerowaniem prawidłowych, czytelnych komunikatów diagnostycznych w przypadku błędnego użycia poprawnego szablonu
- parametrem wzorca może być nie tylko typ, lecz również wartość typu porządkowego
- obliczenie wartości silni podczas kompilacji

```
#include <iostream>

template <int N>
struct Silnia {
    enum { wart = N * Silnia<N-1>::wart };
};

template <>
struct Silnia<1> {
    enum { wart = 1 };
};

int main() {
    const int sil7 = Silnia<7>::wart;
    std::cout << sil7 << std::endl;
    return 0;
}
```

2.6.4 Wzorzec projektowy strategii

- Implementacja w C++

```
#include <iostream>
using namespace std;
```

```
class StrategyInterface
{
    public:
        virtual void execute() const = 0;
};

class ConcreteStrategyA: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called␣ConcreteStrategyA␣execute␣method" << endl;
        }
};

class ConcreteStrategyB: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called␣ConcreteStrategyB␣execute␣method" << endl;
        }
};

class ConcreteStrategyC: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called␣ConcreteStrategyC␣execute␣method" << endl;
        }
};

class Context
{
    private:
        StrategyInterface * strategy_;

    public:
        explicit Context(StrategyInterface *strategy):strategy_(strategy)
        {
        }
```

```
        void set_strategy(StrategyInterface *strategy)
        {
            strategy_ = strategy;
        }

        void execute() const
        {
            strategy_->execute();
        }
};

int main(int argc, char *argv[])
{
    ConcreteStrategyA concreteStrategyA;
    ConcreteStrategyB concreteStrategyB;
    ConcreteStrategyC concreteStrategyC;

    Context contextA(&concreteStrategyA);
    Context contextB(&concreteStrategyB);
    Context contextC(&concreteStrategyC);

    contextA.execute();
    contextB.execute();p
    contextC.execute();

    contextA.set_strategy(&concreteStrategyB);
    contextA.execute();
    contextA.set_strategy(&concreteStrategyC);
    contextA.execute();

    return 0;
}
```

- Implementacja w C:

```
#include <stdio.h>

void print_sum(int n, int *array) {
  int total = 0;
  for (int i=0; i<n; i++) total += array[i];
  printf("%d", total);
}
```

```
void print_array(int n, int *array) {
  for (int i=0; i<n; i++) printf("%d ", array[i]);
}

int main(void) {
  typedef struct {
    void (*submit_func)(int n, int *array);
    char *label;
  } Button;

  // Create two instances with different strategies
  Button button1 = {print_sum, "Add 'em"};
  Button button2 = {print_array, "List 'em"};

  int n = 10, numbers[n];
  for (int i=0; i<n; i++) numbers[i] = i;

  button1.submit_func(n, numbers);
  button2.submit_func(n, numbers);

  return 0;
}
```

- klasy parametryzowane wytycznymi, realizacja wzorca strategy pattern w czasie kompilacji, użycie wielokrotnego dziedziczenia w klasie parametryzowanej wytycznymi

```
template <class T>
struct NoChecking
 {
  void Check(T * _ptr) { /* pusto, brak sprawdzania poprawności */ }
 }

template <class T>
struct StrictChecking
 {
  void Check(T * _ptr) { if (0 == ptr) { /* obsłuż błąd */ } }
 }

// Opis klasy:
//  Wytyczna ErrorHandling musi posiadać metodę void Check(T*)
template <class T, class ErrorHandling>
class SmartPtr : public ErrorHandling
 {
```

```
    public:

    T * Get() const
     {
      Check(ptr);  // <------ !!! zmienia się w zależności od ErrorHandling
      return ptr;
     }

    private:

     T * ptr;
};

// Używanie:
struct Foo
 {
  void Bar() { }
 };


SmartPtr<Foo, NoChecking<Foo> > ptrNoCheck = new Foo();

ptrNoCheck.Get()->Bar();  // Get() nie sprawdzane

SmartPtr<Foo, StrictChecking<Foo> > ptrChecked = new Foo();

ptrChecked.Get()->Bar();  // Get() sprawdzane
```

Przykład 2:

```
template
<
    typename output_policy,
    typename language_policy
>
class HelloWorld : public output_policy, public language_policy
{
    using output_policy::Print;
    using language_policy::Message;
public:
    //behaviour method
    void Run()
    {
        //two policy methods
```

```
        Print( Message() );
    }
};


#include <iostream>

class HelloWorld_OutputPolicy_WriteToCout
{
 protected:
    template< typename message_type >
    void Print( message_type message )
    {
        std::cout << message << std::endl;
    }
};


#include <string>

class HelloWorld_LanguagePolicy_English
{
 protected:
    std::string Message()
    {
        return "Hello, World!";
    }
};

class HelloWorld_LanguagePolicy_German
{
 protected:
    std::string Message()
    {
        return "Hallo Welt!";
    }
};


int main()
{
    /* example 1 */
    typedef
```

```
        HelloWorld
        <
            HelloWorld_OutputPolicy_WriteToCout,
            HelloWorld_LanguagePolicy_English
        >
            my_hello_world_type;

    my_hello_world_type hello_world;
    hello_world.Run(); // Prints "Hello, World!"


    /* example 2
     * does the same but uses another policy, the language has changed
     */
    typedef
        HelloWorld<
            HelloWorld_OutputPolicy_WriteToCout,
            HelloWorld_LanguagePolicy_German
        >
            my_other_hello_world_type;

    my_other_hello_world_type hello_world2;
    hello_world2.Run(); // Prints "Hallo Welt!"
}
```

2.6.5 Zadanie domowe

Zadanie na 6.0

Zadanie z XVI olimpiady informatycznej 2008/2009 z II etapu o nazwie Konduktor.

Uwaga. Zadanie na 6.0 powinno być wysłane na maila wraz z opisem rozwiązania.

2.7 wzorce w C++ cd.

2.7.1 Wstęp teoretyczny

- przeciążanie wzorca funkcji
- domyślne parametry wzorca
- specjalizacja, cd.
- częściowa specjalizacja

2.7.2 Zadanie domowe

Zadanie na 6.0

Napisać program w C++, który wypisuje sam siebie.

2.8 Biblioteka standardowa C++

2.8.1 Biblioteka standardowa

Biblioteka standardowa zawiera m.in. algorytmy, kolekcje i iteratory w formie szablonów. Biblioteka standardowa jest biblioteką generyczną - jej składniki współpracują z typami wbudowanymi, typami biblioteki i typami użytkownika. Elementy biblioteki standardowej:

1. Kolekcje - obiekty zbiorcze. Wektor, lista, zbiór, mapa.
2. Iteratory.
3. Algorytmy dostępne jako wzorce funkcji.

Rozdział 3

Język SQL

3.1 Relacyjne bazy danych wprowadzenie

3.1.1 Relacyjne bazy danych projektowanie

3.1.2 Zadania

Zadanie obowiązkowe: zaprojektować bazę danych z co najmniej jedną relacją jeden do wielu, z co najmniej trzema tabelami, zaprojektować klasy dla tego modelu danych.

3.1.3 Laboratoria

1. Zadania na 3.0
   (a) utworzyć w mysql bazę danych z dwoma tabelami połączonymi relacją jeden do wielu
   (b) wpisać do tabel przykładowe dane
   (c) przetestować podstawowe zapytania select na stworzonych tabelach
2. Zadania na 4.0
   (a) utworzyć bazę danych z tabelami odzwierciedlającymi relację wiele do wielu
3. Zadania na 5.0
   (a) utworzyć bazę danych z tabelą zawierającą kilka kluczy obcych

3.2 SQL, podstawowe zapytania

Podstawowe operacje bazodanowe

3.2.1 SELECT

Polecenie SELECT jest używane do pobierania wierszy z jednej lub więcej tabeli. Składnia najczęściej używana:

```
SELECT select_expr [, select_expr ...]
[FROM table_references]
[WHERE where_condition]
[GROUP BY {col_name|expr|position}]
[HAVING where_condition]
[ORDER BY {col_name|expr|position} [ASC|DESC]]
```

Każde select_expr wskazuje na kolumnę, którą chcemy pobrać. Musi być co najmniej jedno select_expr. Table_references wskazuje jedną tabelę lub wiele z których będą pobierane wiersze.

Klauzula WHERE określa warunki jakie muszą spełniać wiersze, aby być wybranymi. Jest to dowolne wyrażenie SQL, które dla danego rekordu jest prawdziwe lub nie.

Przy specyfikowaniu select_expr możemy użyć gwiazdki co oznacza wszystkie kolumny we wszystkich tabelach. Przykład

```
SELECT * FROM t1
```

Do tabeli można odwoływać się z nazwy tabeli tbl_name, albo db_name.tbl_name, a odwoływać do kolumny możemy się jako col_name, tbl_name.col_name, albo db_name.tbl_name.col_nam
Kiedy istnieje wiele kolumn o tej samej nazwie, wtedy musimy wyspecyfikować nazwę kolumny wraz z nazwą tabeli.

Przykład z klauzulą ORDER_BY

```
SELECT college, region, seed FROM tournament
ORDER BY region, seed
```

Klauzula ORDER BY oznacza sortowanie zwracanych rekordów.

3.2.2 INSERT

INSERT dodaje nowe wiersze do tabeli. Składnia: INSERT [INTO] tbl_name [(col_name,...)] {VALUES |VALUE } ({expr | DEFAULT},...), (...),... . Przykład

```
INSERT INTO tbl_name (a,b,c) VALUES (1,2,3);
```

3.2.3 UPDATE

UPDATE zmienia dane w istniejących wierszach tabeli. Składnia:

```
UPDATE table_references SET col_name1={expr|DEFAULT}
[col_name2={expr2|DEFAULT}]...
[WHERE where_condition] [ORDER BY ...]
```

Where_condition to wyrażenie, które jest rozwijane do true lub false dla każdego wiersza.

Przykład

```
UPDATE Persons SET Address='Nissestien 67', City='Sandnes'
WHERE LastName='Kowalski' AND FirstName='Jan'
```

Kiedy jest przydatne ORDER BY np nie możemy uruchomić zapytania: UPDATE t SET id = id + 1, ponieważ jeśli kolumna id jest unikalna, to możemy dostać błąd duplicate-key error, rozwiązaniem jest UPDATE t set id = id + 1 ORDER BY id DESC

3.2.4 DELETE

Klauzula DELETE usuwa wiersze z tabeli. Składnia

```
DELETE FROM tbl_name [WHERE where_condition]
[ORDER BY ...] [LIMIT row_count].
```

Przykład użycia

```
DELETE FROM somelog WHERE user='user1'
ORDER BY timestamp_column LIMIT 1
```

3.2.5 Zadania

Zadanie obowiązkowe: Do modeli baz danych skonstruowanych na poprzednich zajęciach zaprojektować model serwisu www korzystającego z tej bazy, oraz skonstruować po 3 zapytania każdego rodzaju, które będą wykorzystywane w systemie.

3.3 SQL, GROUP BY, HAVING

3.3.1 GROUP BY

Przykład z GROUP BY

```
SELECT a, COUNT(b) FROM test_table GROUP BY a
```

Klauzula GROUP BY jest używana razem z funkcjami agregującymi. GROUP BY oznacza kolumnę, według której agregujemy. Zostaną zwrócone rekordy dokładnie jeden dla każdej wartości kolumny GROUP BY z zastosowaną funkcją agregującą. Gdy nie podamy GROUP BY grupujemy po wszystkich wierszach. Funkcje agregujące

1. COUNT(expr) zwraca liczbe nie nullowych wartości expr, COUNT(*) zwraca liczbę wartości niezależnie od tego czy jest null czy nie, np

   ```
   SELECT student.student_name, COUNT(*)
   FROM student, course
   WHERE student.student_id=course.student_id
   GROUP BY student_name
   ```

2. AVG (expr) zwraca średnią wartość, np

```
SELECT student_name, AVG(test_score) FROM student
GROUP BY student_name;
```

3. MAX(expr) zwraca maksymalną liczbę, np

   ```
   SELECT student_name, MIN(test_score), MAX(test_score)
   FROM student GROUP BY student_name
   ```

4. SUM(expr) sumuje expr

5. VAR_POP(expr) population variance, odchylenie standardowe expr, liczone ze wzoru

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

   gdzie

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

   Przykład:

   ```
   SELECT YEAR(ShipDate) AS Year,
   Quarter(ShipDate) As Quarter,
   AVG(Quantity) AS Average, VAR_POP(quantity) As Variance
   FROM SalesOrderItems GROUP BY Year, Quarter
   ORDER BY Year, Quarter;
   ```

   Result: Year, Quarter, Average, Variance
   2000, 1, 25.7, 203
   2000, 2, 27, 225

6. VAR_SAMP(expr) sample variance, liczone ze wzoru

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \hat{y})^2$$

3.3.2 HAVING

Przykład z HAVING

```
SELECT CustomerName, SUM(OrderPrice) FROM Sales
GROUP BY CustomerName HAVING SUM(OrderPrice) > 1200
```

HAVING jest używana z GROUP BY. HAVING działa podobnie jak WHERE ale jest aplikowana do zgrupowanych rekordów.

3.3.3 Zadania

Zadanie obowiązkowe: Do własnych modeli baz danych zaprojektować 3 różne zapytania GROUP BY (z różnymi funkcjami agregującymi), które będą wykorzystywane w systemie.

3.4 SQL, JOIN

Wyrażenie table_references może zawierać klauzulę JOIN o następującej składni:
table_reference: table_factor| join_table,
table_factor: tbl_name [[AS] alias],
join_table: table_reference [INNER|CROSS] JOIN table_factor [join_condition]
| table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_condition.
Wyrażenie join_condition ma składnię: ON conditional_expr | USING (column_list).

3.4.1 INNER JOIN

Przykład:

```
SELECT t1.name, t2.salary FROM employee AS t1
INNER JOIN info AS t2 ON t1.name=t2.name
```

AS jest opcjonalne. INNER JOIN porównuje każdy wiersz pierwszej tabeli z każdym wierszem drugiej tabeli i sprawdza czy warunek ON jest spełniony.

Przykład

```
SELECT * FROM t1 INNER JOIN t2
```

Dwa poniższe zapytania zwracają to samo

```
SELECT * FROM t1, t2 WHERE t1.id = t2.id
```

```
SELECT * FROM t1 INNER JOIN t2 ON t1.id=t2.id
```

Dwa poniższe zapytania zwracają to samo:

```
SELECT a,b,c
FROM Foo, Bar, Flub
WHERE Foo.y BETWEEN Bar.x AND Flub.z
```

```
SELECT *
FROM Foo
INNER JOIN
BAR ON Foo.y >= Bar.x
INNER JOIN
Flub ON Foo.y <= Flub.z;
```

Przykład

```
SELECT płeć, COUNT(*), AVG(wiek),
(MAX(wiek) - MIN(wiek)) AS
zakres_wiekowy
FROM studenci, Rejestr_ocen
WHERE ocena = '5'
AND Studenci.nr_stud = Rejestr_ocen.nr_stud
GROUP BY płeć
HAVING COUNT(*) > 3;
```

3.4.2 LEFT JOIN

Inne przykłady:

```
SELECT * FROM table1 LEFT JOIN table2
ON table1.id=table2.id
```

LEFT JOIN oznacza zwrócenie wszystkich rekordów z tabeli 1, nawet jeśli warunek ON nie jest spełniony, wtedy w kolumnach drugiej tabeli będą wartości NULL. Jeśli jest więcej dopasowań z drugiej tabeli, to każde z nich będzie oddzielnym rekordem. A więc LEFT JOIN zwróci co najmniej tyle rekordów ile jest w tabeli 1.

3.4.3 RIGHT JOIN

Przykład:

```
SELECT * FROM t1 RIGHT JOIN t2 ON (t1.a = t2.a)
```

Klauzula RIGHT_JOIN działa odwrotnie do LEFT_JOIN, to znaczy zwraca wszystkie rekordy z tabeli 2, nawet jeśli warunek ON nie jest spełniony, wtedy w kolumnach pierwszej tabeli będą wartości NULL. Jeśli jest więcej dopasowań z pierwszej tabeli, to każde z nich będzie oddzielnym rekordem. A więc RIGHT_JOIN zwróci co najmniej tyle rekordów ile jest w tabeli 2.

3.4.4 FULL OUTER JOIN

Złączenie FULL OUTER JOIN jest mieszanką złączeń zewnętrznych prawego i lewego, które zachowuje utworzone wiersze z obydwu tabel.

Przykład

```
SELECT id_dost, nazwa_dost, kwota_zam
FROM Zamówienia FULL OUTER JOIN Dostawcy
ON (Dostawcy.id_dost = Zamówienia.id_dost);
```

3.4.5 Self join

Jest możliwy join z tą samą tabelą (self join). Przykład które części mają rożnych dostawców

```
SELECT DISTINCT a.pno
FROM sp a, sp b
WHERE a.pno = b.pno
AND a.sno <> b.sno
```

3.4.6 Zadania

Zadanie obowiązkowe: Do własnych modeli baz danych zaprojektować 3 różne zapytania JOIN, które będą wykorzystywane w systemie.

3.5 SQL, podzapytania

Podzapytania inaczej zapytania zagnieżdżone. Podzapytania są używane najczęściej w klauzuli WHERE kiedy wiemy jak znaleźć jakieś wartości, ale ich nie znamy. Przykład podzapytania:

```
SELECT * FROM t1 WHERE column1=(SELECT column1 FROM t2);
```

Podzapytanie może zwrócić pojedynczą wartość, pojedynczy wiersz lub tabelę (jeden lub więcej wierszy z jedną lub więcej kolumn)

Przykład

```
SELECT SUM(Sales) FROM Store_information
WHERE Store_name IN (SELECT store_name FROM Geography
WHERE region_name='West');
```

Zadanie zrealizować to zapytanie za pomocą join.

Występują również podzapytania skorelowane z nadzapytaniem. Przykład

```
SELECT SUM(a1.Sales) FROM Store_information a1
WHERE a1.Store_name IN (SELECT store_name FROM Geography a2
WHERE a2.store_name=a1.store_name);
```

W klauzuli WHERE podzapytania występuje odwołanie do tabeli z nadzapytania.

Przykład

```
SELECT id, first_name FROM student_details
WHERE first_name IN (SELECT first_name FROM student_details
WHERE subject='Science');
```

Przykład

```
SELECT p.product_name, p.supplier_name, (SELECT order_id
FROM order_items WHERE product_id=101)
AS order_id FROM product p WHERE p.product_id = 101;
```

Przykład skorelowanego podzapytania

```
SELECT p.product_name FROM product p
WHERE p.product_id=(SELECT o.product_id FROM order_items o
WHERE o.product_id = p.product_id);
```

Są 3 typy podzapytań: podzapytanie w klauzuli WHERE i HAVING, podzapytanie zwracające pojedynczą wartość i podzapytanie występujące w klauzuli FROM.

1. Podzapytanie musi zwracać jedną kolumnę w SELECT (oprócz EXISTS) Przykład z IN

   ```
   SELECT *
   FROM p
   WHERE pno IN (SELECT pno FROM sp);
   ```

   Realizacja self join za pomocą podzapytań

   ```
   SELECT DISTINCT pno
   FROM sp a
   WHERE pno IN (SELECT pno FROM sp b WHERE a.sno <> b.sno);
   ```

   Przykład z ANY, ALL, SOME

   ```
   SELECT *
   FROM p
   WHERE pno = ANY (SELECT pno FROM sp);
   ```

   Przykład self joina

   ```
   SELECT *
   FROM sp a
   WHERE qty > ALL (SELECT qty FROM sp b
   WHERE a.pno=b.pno
   AND a.sno <> b.sno
   AND qty IS NOT NULL;
   ```

   Przykład z EXISTS wylistowanie wszystkich części, które posiadają dostawcy

   ```
   SELECT *
   FROM p
   WHERE EXISTS(SELECT * FROM sp WHERE p.pno = sp.pno);
   ```

2. Podzapytania skalarne, podzapytanie musi zawierać jedną kolumnę i zwracać tylko jeden wiersz. Przykład

   ```
   SELECT pno, qty, (SELECT city FROM s WHERE s.sno=sp.sno)
   FROM sp;
   ```

   Przykład

```
SELECT *
FROM sp
WHERE 'Blue' = (SELECT color FROM p
WHERE p.pno = sp.pno)
```

3. Podzapytania we FROM. Dwa poniższe zapytania zwracają to samo

```
SELECT p.*, qty
FROM p, sp
WHERE p.pno=sp.pno
AND sno='S3'

SELECT p.*, qty
FROM p, (SELECT pno, qty FROM sp WHERE sno = 'S3')
WHERE p.pno=sp.pno
```

Dwa poniższe zapytania zwracają to samo

```
SELECT nr_autora
FROM Autorzy
WHERE nr_autora
IN (SELECT nr_autora
FROM AutorzyArtykulu
WHERE tantiemy < 0.50)

SELECT DISTINCT Autorzy.nr_autora
FROM Autorzy, AutorzyTytułu
WHERE (Autorzy.nr_autora = AutorzyTytułu.nr_autora)
AND (wielkość_tantiemów < 0.50);
```

Przykład:

```
SELECT *
From Studenci AS S1
WHERE wiek < (SELECT MAX(wiek)
FROM Studenci AS S2
WHERE S1.płeć=S2.płeć);
```

3.5.1 Zadania

Zadanie obowiązkowe: Do własnych modeli baz danych zaprojektować 3 różne zapytania z podzapytaniami, które będą wykorzystywane w systemie.

3.6 SQL, LIKE, IN, CASE, EXISTS

3.6.1 LIKE

Predykat LIKE jest testem dopasowującym wzorzec łańcucha. Składnia

```
<predykat like> ::=
<pasująca wartość> [NOT] LIKE <wzorzec>
[ESCAPE <znak escape>]
<pasująca wartość> ::= <wyrażenie wartości znakowej>
<wzorzec> ::= <wyrażenie wartości znakowej>
<znak escape> ::= <wyrażenie wartości znakowej>
```

W łańcuchu <wzorzec> dopuszczalne są dwa znaki wieloznaczne: '%' i '_'. Pierwszy z nich reprezentuje dowolny podłańcuch, również o zerowej długości. Drugi z nich reprezentuje dowolny pojedynczy znak.

Przykład, wyszukanie imion zaczynających się od 'b'

```
SELECT * FROM pet WHERE name LIKE 'b%'
```

Wyszukanie imion kończących się na 'fy'

```
SELECT * FROM pet WHERE name LIKE '%fy'
```

Znalezienie imon zwierających literkę 'w'

```
SELECT * FROM pet WHERE name LIKE '%w%'
```

Znalezienie imon zawierających dokładnie 5 liter

```
SELECT * FROM pet WHERE name LIKE '_____'
```

Dwa poniższe zapytania różnią się:

```
SELECT * FROM osoby WHERE nazwisko LIKE 'John_%'

SELECT * FROM osoby WHERE nazwisko LIKE 'John%'
```

Pierwsze zapytanie nie zaakceptuje nazwiska John. '_%' określają wzorzec jeden lub więcej znaków. Można również stosować operator konkatenacji do napisów:

```
SELECT * FROM osoby WHERE nazwisko LIKE '%' || imię || '%'
```

Tutaj są rozróżniane małe i duże litery, aby nie były modyfikujemy zapytanie

```
SELECT * FROM osoby WHERE UPPER(nazwisko)
LIKE '%' || UPPER(imię) || '%'
```

W Mysql konkatenacja jest zapisywana jako funkcji CONCAT(). Z reguły testowanie za pomocą '_' jest szybsze niż z '%'. Dlatego wyszukiwanie maksymalnie siedmioliterowego nazwiska zaczynającego się od Mar może wyglądać następująco

```
SELECT * FROM osoby WHERE nazwisko LIKE 'Mac'
OR nazwisko LIKE 'Mac_'
OR nazwisko LIKE 'Mac__'
OR nazwisko LIKE 'Mac___'
OR nazwisko LIKE 'Mac____'
```

Jest możliwość w SQL definiowania bardziej rozbudowanych wzorców za pomocą SIMILAR TO. W mysql jest alternatywa RLIKE.

3.6.2 BETWEEN

```
<wyrażenie wartości> [NOT] BETWEEN <wyrażenie wartości dolnej>
AND <wyrażenie wartości górnej>
```

Przykłady: x BETWEEN 12 AND 15 – zależy od wartości x, x BETWEEN 15 AND 12 – zawsze false, x BETWEEN NULL AND 15 – zawsze UNKNOWN, NULL BETWEEN 12 AND 15 – zawsze UNKNOWN, x BETWEEN 12 AND NULL – zawsze UNKNOWN, x BETWEEN x AND x – zawsze TRUE.

3.6.3 IN

Pobiera wartość i sprawdza czy wartość ta jest na liście porównywalnych wartości. Składnia

```
<predykat in> ::=
<konstruktor wartości wiersza> [NOT] IN <wartość predykatu in>
<wartość predykatu in> ::= <podzapytanie tabelaryczne>
| (<lista wartości predykatu in>)
<lista wartości predukatu in> ::= <wyrażenie wartości wiersza>
{<przecinek> <wyrażenie wartości wiersza >}...
```

Wyrażenie <konstruktor wartości wiersza> IN <wartość predykatu in> daje taki sam rezultat jak <konstruktor wartości wiersza> = ANY <wartość predykatu in>.

Można zastąpić operatory OR przedykatem IN(), przykład

```
SELECT * FROM RaportKontroliJakosci
WHERE test_1='zaliczony'
OR test_2 = 'zaliczony'
OR test_3 = 'zaliczony'
OR test_4 = 'zaliczony

SELECT * FROM RaportKontroliJakosci
WHERE zaliczony IN (test_1, test_2, test_3, test_4)
```

3.6.4 EXISTS

Jest to test dla zbioru niepustego. Składnia

```
<predykat exists> ::= EXISTS <podzapytanie tabelaryczne>
```

Przykład

```
SELECT P1.nazw_prac, ' urodził się tego samego
dnia co znana osoba!'
FROM Personel AS P1
WHERE EXISTS
(SELECT * FROM Osobistości AS C1
WHERE P1.urodziny = C1.urodziny)
```

3.6.5 WYRAŻENIA CASE

Wyrażenie CASE pozwala programiście wybrać wartość na podstawie wyrażenia logicznego. Składnia

```
<specyfikacja case> ::= <warunek prosty> | <warunek szukany>
<warunek prosty> ::=
CASE <argument case>
  <prosta klauzula when>...
        [<klauzula else>]
END
<warunek szukany> ::=
CASE
  <przeszukiwana klauzula when>...
  [<klauzula else>]
END
<prosta klauzula when> ::= WHEN <argument when> THEN <wynik>
<przeszukiwana klauzula when> ::= WHEN <warunek wyszukiwania>
THEN <wynik>
<klauzula else> ::= ELSE <wynik>
<argument case> ::= <wyrażenie wartości>
<argument when> ::= <wyrażenie wartości>
<wynik> ::= <wyrażenie wynikowe> | NULL
<wyrażenie wynikowe> ::= <wyrażenie wartości>
```

<Proste wyrażenie CASE> jest zdefiniowane jako przeszukiwane wyrażenie CASE, w którym wszystkie klauzule WHEN sprawdzają równość z <argument CASE>. Przykład

```
CASE kod_płci_iso
WHEN 0 THEN 'Nieznana'
WHEN 1 THEN 'Mężczyzna'
WHEN 2 THEN 'Kobieta'
WHEN 9 THEN 'N/D'
ELSE NULL
END
```

lub

```
CASE
WHEN kod_płci_iso = 0 THEN 'Nieznana'
WHEN kod_płci_iso = 1 THEN 'Mężczyzna'
WHEN kod_płci_iso = 2 THEN 'Kobieta'
WHEN kod_płci_iso = 9 THEN 'N/D'
ELSE NULL
END
```

Przykład ile pracowników każdej płci w podziale na wydziały mamy w tabeli Personel

```
SELECT nr_wydziału, SUM(CASE WHEN płeć='M' THEN 1 ELSE 0)
AS mężczyźni,
SUM(CASE WHEN płeć = 'F' THEN 1 ELSE 0) AS kobiety
FROM Personel
GROUP BY nr_wydziału
```

Przykład

```
SELECT store_name, CASE store_name
WHEN 'Los␣Angeles' THEN Sales*2
WHEN 'San␣Diego' THEN Sales*1.5
ELSE Sales
END "New␣Sales",
Date
FROM store_information
```

3.6.6 Zadania

Zadanie obowiązkowe: Do własnych modeli baz danych zaprojektować 3 różne zapytania z LIKE, CASE które będą wykorzystywane w systemie.

3.7 SQL, UNION

3.7.1 UNION

Operator UNION służy do łączenia poleceń SELECT. Obydwa polecenia SELECT powinny mieć kolumny tego samego typu. W sql domyślnie usuwane są duplikaty po takim połączeniu. Aby nie były usuwane należy użyć UNION ALL.

Przykład użycia z UNION i ORDER BY i LIMIT

```
(SELECT a FROM t1 WHERE a=10 AND B=1)
UNION
(SELECT a FROM t2 WHERE a=11 AND B=2)
ORDER BY a LIMIT 10;
```

UNION zwraca nieuporządkowane wiersze. Aby najpierw były zwrócone z pierwszego selecta a później z drugiego można użyć dodatkowej kolumny

```
(SELECT 1 AS sort_col, col1a, col1b, ... FROM t1)
UNION
(SELECT 2, col2a, col2b, ... FROM t2)
ORDER BY sort_col;
```

Aby posortować dodatkowo wiersze z obu selectów modyfikujemy ORDER BY

```
(SELECT 1 AS sort_col, col1a, col1b, ... FROM t1)
UNION
(SELECT 2, col2a, col2b, ... FROM t2)
ORDER BY sort_col, col1a;
```

Przykład

```
SELECT czynsz, usługi, telefon
FROM
(SELECT a, b, c FROM StaraLokalizacja WHERE city='Boston'
UNION
SELECT x, y, z FROM NowaLokalizacja WHERE city='Nowy York')
AS Cities (czynsz, usługi, telefon);
```

Przykład unii z tą samą tabelą

```
SELECT nazwa_miasta, 'Zachodnie'
FROM Miasta
WHERE kod_rynku='t'
UNION ALL
SELECT nazwa_miasta, 'Wschodnie'
FROM Miasta
WHERE kod_rynku='v';
```

Można to zapisać inaczej

```
SELECT nazwa_miasta,
CASE kod_rynku
WHEN 't' THEN 'Zachodnie'
WHEN 'v' THEN 'Wschodnie' END
FROM Miasta
WHERE kod_rynku IN ('v', 't');
```

3.7.2 INTERSECT, EXCEPT

INTERSECT - część wspólna, EXCEPT - odejmowanie zbiorów Przed dokonaniem tych operacji usuwane są duplikaty z obydwu zbiorów. Dla INTERSECT ALL i EXCEPT ALL nie są usuwane duplikaty. Przykład: mamy dwa wielozbiory: S1={a,a,b,b, c },

S2={a, b,b,b,c,d}. Po usunięciu duplikatów S1={a,b,c }, S2={a,b,c,d}. S1 INTERSECT S2={a,b,c}. S2 EXCEPT S1={d}. S1 EXCEPT S2={}. S1 INTERSECT ALL S2={a,b,b,c}. S2 EXCEPT ALL S1={b,d}.

Dwa poniższe zapytania robią to samo

```
SELECT *
FROM Personel
WHERE praca='Nowy Jork'
UNION
SELECT *
FROM Personel
WHERE dom='Chicago';
```

```
SELECT DISTINCT *
FROM Personel
WHERE praca='Nowy Jork' OR
dom='Chicago';
```

Rozdział 4

Statystyka w SQL

4.1 Podstawowe statystyki

Dla danej kolumny tabeli obliczyć częstotliwość występowania liczb oraz częstotliwość procentową.

```
SELECT F1.x, COUNT(F1.x),
CAST(ROUND(100.*(COUNT(F1.x))
/(SELECT COUNT(*) FROM Foobar),0) AS INTEGER)
FROM Foobar AS F1
GROUP BY F1.x;
```

```
SELECT F2.x, F2.bzwgl_czest,
```

Wartości modalne rozkładu, czyli wartości najbardziej prawdopodobne można znaleźć za pomocą zapytania:

```
SELECT pobory, COUNT(*) AS czestotliwosc
FROM ListaPlac
GROUP BY pobory
HAVING COUNT(*)
>= ALL (SELECT COUNT(*) FROM ListaPlac GROUP BY pobory)
```

Możemy zapisać to zapytanie bez GROUP BY

```
SELECT DISTINCT pobory
FROM Personel AS P1
WHERE NOT EXISTS
(SELECT *
FROM Personel AS P2
WHERE EXISTS
(SELECT COUNT(*) FROM Personel AS P3
WHERE P1.pobory=P3.pobory
< (SELECT COUNT(*) FROM Personel AS P4
WHERE P2.pobory=P4.pobory)))
```

Gdy chcemy zwrócić wartości modalne z 5% możliwym odchyleniem to

```
SELECT AVG(pobory), COUNT(*) AS wart_modalna
FROM ListaPlac
GROUP BY pobory
HAVING COUNT(*)
>= ALL (SELECT COUNT(*)*0.95 FROM ListaPlac GROUP BY pobory)
```

Średnia harmoniczna:

$$\frac{n}{\frac{1}{a_1}+\frac{1}{a_2}+\ldots+\frac{1}{a_n}}$$

Może być obliczona w SQL za pomocą następującego polecenia

```
SELECT COUNT(*)/SUM(1.0/x) AS srednia_harmoniczna
FROM Foobar;
```

Średnia geometryczna

$$\sqrt[n]{a_1 a_2 \ldots a_n}$$

Inny wzór

$$s_g = \exp\left(\frac{1}{n}\sum_{i=1}^{n}\ln a_i\right)$$

```
SELECT EXP(AVG(LOG(nr))) AS srednia_geometryczna
FROM TabelaLiczb;
```

4.1.1 Zadania

Zadanie obowiązkowe:

1. Obliczyć wartość średnią i wariancję dla ocen z poprzedniego semestru.
2. Obliczyć sumę wartość średnia - r*wariancja, gdzie r to ryzyko dla kilku wybranych wartości r.
3. Przetestować obliczanie wariancji w SQL

4.2 Wariancja

Wariancja jest zdefiniowana jako:

$$\sigma^2 = var(X) = E(X-\mu)^2$$

gdzie $\mu = E(X)$. Zachodzi

$$var(X+a) = var(X)$$
$$var(aX) = a^2 var(X)$$

$$var\left(aX+bY\right)=a^2var\left(X\right)+b^2var\left(Y\right)+2abcov\left(X,Y\right)$$

Dla rzutu kostką wartość oczekiwana wynosi

$$\left(1+2+3+4+5+6\right)/6=3.5$$

Wartość średnia odchyleń bezwzględnych $(3.5-1, 3.5-2, 3.5-3, 4-3.5, 5-3.5, 6-3.5)$

$$\left(2.5+1.5+0.5+0.5+1.5+2.5\right)/6=1.5$$

Wariancja średnia kwadratów odchyleń

$$\left(2.5^2+1.5^2+0.5^2+0.5^2+1.5^2+2.5^2\right)/6=17.5/6\approx 2.9$$

Dla dwukrotnego rzutu monetą mamy: liczba orzełków 0 z prawd. 0.25, 1 z prawd. 0.5, 2 z prawd. 0.25. Średnia wynosi:

$$0.25*0+0.5*1+0.25*2=1$$

A wariancja wynosi:

$$0.25*\left(0-1\right)^2+0.5\left(1-1\right)^2+0.25*\left(2-1\right)^2=0.25+0+0.25=0.5$$

Relacja między wariancją populacji, a wariancją próbki. Mamy populację liczb. Liczba 1 występuje 1/3 czasu, 2 występuje 1/3 czasu, 4 występuje 1/3 czasu. Wartość średnia populacji

$$1/3\left(1+2+4\right)=7/3$$

Odchylenia wynoszą: 1 - 7/3, 2 - 7/3, 4 - 7/3. Wariancja populacji wynosi

$$1/3\left(\left(-4/3\right)^2+\left(-1/3\right)^2+\left(5/3\right)^2\right)=14/9$$

Weźmy pod uwagę wszystkie możliwe dwie obserwacje, jest 9 możliwych równie prawdopodobnych możliwości: (1,1), (1, 2), (1, 4), (2, 1), (2, 2), (2, 4), (4, 1), (4, 2) and (4, 4). Wariancja próbki wynosi odpowiednio: $0, 1/4, 9/4, 1/4, 0, 4/4, 9/4, 4/4, 0$. A wartość oczekiwana wynosi

$$1/9\left(0+1/4+9/4+1/4+0+4/4+9/4+4/4+0\right)=7/9$$

Jest to dużo mniej niż 14/9, czyli prawdziwa wartość wariancji. Zauważmy, że jeśli pomnożymy wariancję próbek przez $n/\left(n-1\right)=2/\left(2-1\right)=2$, to wtedy estymacje wariancji populacji wynoszą: 0, 1/2, 9/2, 1/2, 0, 4/2, 9/2, 4/2 i 0. Średnia wynosi 14/9.

Przykład 2: Średnia populacji wynosi 2050, ale tego nie wiemy, więc estymujemy to z próbki: 2051, 2053, 2055, 2050, 2051. Średnia wynosi

$$\left(2051+2053+2055+2050+2051\right)/5=2052$$

Teraz musimy wyestymować wariancję. Jeśli znalibyśmy średnią populacji to

$$\left((2051-2050)^2+(2053-2050)^2+(2055-2050)^2+(2050-2050)^2+(2051-2050)^2\right)/5$$
$$=36/5=7.2$$

Natomiast gdy użyjemy 2052, czyli średniej z próbki to otrzymamy wariancję

$$\left((2051-2052)^2+(2053-2052)^2+(2055-2052)^2+(2050-2052)^2+(2051-2052)^2\right)/5$$
$$16/5=3.2$$

Jest to dużo mniej. Czy średnio zawsze estymacja będzie mniejsza? Tak, powód:

$$(a+b)^2=a^2+2ab+b^2$$

Po lewej odchylenie od średniej populacji do kwadraty

$$(2053-2050)^2=((2053-2052)+(2052-2050))^2$$

Pierwszy człon po prawej odchylenie od średniej próbki. Rozpisując ze wzoru

$$(2053-2052)^2+2\,(2053-2052)\,(2052-2050)+(2052-2050)^2$$

Rozpisując tym sposobem wszystkie wartości próbki

$$(2051-2052)^2+2\,(2051-2052)\,(2052-2050)+(2052-2050)^2$$
$$(2053-2052)^2+2\,(2053-2052)\,(2052-2050)+(2052-2050)^2$$
$$(2055-2052)^2+2\,(2055-2052)\,(2052-2050)+(2052-2050)^2$$
$$(2050-2052)^2+2\,(2050-2052)\,(2052-2050)+(2052-2050)^2$$
$$(2051-2052)^2+2\,(2051-2052)\,(2052-2050)+(2052-2050)^2$$

Suma środkowych wartości będzie zerem ponieważ suma odchyleń od średniej próbki jest zero. Suma wszystkich pierwszych członów jest sumą kwadratów odchyleń od średniej próbki. Suma wszystkich pierwszych i trzecich członów jest sumą kwadratów odchyleń od średniej populacji. Suma wszystkich tych członów musi być większa od sumy pierwszych członów, ponieważ ostatni człon jest dodatni (jeśli średnia próbki jest różna od średniej populacji). Dlatego suma kwadratów odchyleń ze średniej populacji będzie większa of sumy kwadratów odchyleń ze średniej próbki. Dlatego, suma kwadratów odchyleń z średniej próbki jest za mała aby dać estymatę nieobciążoną wariancji populacji.

4.2.1 Zadania

Zadanie obowiązkowe:

1. Obliczyć wartość średnią i wariancję dla ocen z poprzedniego semestru.
2. Obliczyć sumę wartość średnia - r*wariancja, gdzie r to ryzyko dla kilku wybranych wartości r.
3. Przetestować obliczanie wariancji w SQL

4.3 Kowariancja

Kowariancja jest miarą zależności liniowej między dwoma zmiennymi losowymi:

$$cov\left(X,Y\right)=E\left(\left(X-EX\right)\left(Y-EY\right)\right)$$

lub inaczej

$$cov\left(X,Y\right)=E\left(XY\right)-EXEY$$

Jeśli zmienne losowe X i Y są niezależne, a więc

$$E\left(XY\right)=EXEY$$

to

$$cov\left(X,Y\right)=0$$

Uwaga: zmienne mogą mieć kowariancję 0, a mimo to być zależne nieliniowo.

Wariancja jest szczególnym przypadkiem kowariancji gdy X=Y.

Macierz kowariancji jest uogólnieniem pojęcia wariancji na przypadek wielowymiarowy.

Macierz kowariancji z próbki obliczamy ze wzoru:

$$q_{ij}=\frac{1}{N-1}\sum_{k=1}^{N}\left(x_{ik}-\hat{x}_i\right)\left(x_{jk}-\hat{x}_j\right)$$

Jeśli znane jest $E\left(X\right)$ (nie obliczane z danej próbki) to wzór wygląda następująco

$$q_{ij}=\frac{1}{N}\sum_{k=1}^{N}\left(x_{ik}-E\left(X_i\right)\right)\left(x_{jk}-E\left(X_j\right)\right)$$

Kowariancja między dwoma zmiennymi o jednym wymiarze (odpowiedzią będzie jedna liczba) może być wyliczona w SQL za pomocą zapytania

```
SELECT (1.0/n)*SUM((x - xbar)*(y - ybar)) as Covariance
FROM stu,
(SELECT COUNT(*) AS n, AVG(x) AS xbar,
AVG(y) AS ybar
FROM stu) A
GROUP BY n
```

Inną miarą siły zależności liniowej między dwoma zmiennymi jest współczynnik Pearsona przyjmujący wartości od -1 do 1 zdefiniowany jako

$$\rho_{X,Y}=\frac{cov\left(X,Y\right)}{\sigma_X\sigma_Y}=\frac{E\left(\left(X-\mu_X\right)\left(Y-\mu_Y\right)\right)}{\sigma_X\sigma_Y}$$

Można go obliczyć z próbki za pomocą wzoru

$$r=\frac{\sum_{i=1}^{n}\left(X_i-\hat{X}\right)\left(Y_i-\hat{Y}\right)}{\sqrt{\sum_{i=1}^{n}\left(X_i-\hat{X}\right)^2}\sqrt{\sum_{i=1}^{n}\left(Y_i-\hat{Y}\right)^2}}$$

4.3.1 Zadania

Zadanie na 6.0 Wyznaczyć za pomocą zapytania SQL współczynnik kowariancji Pearsona dla danej próbki i przetestować na przykładowej tabeli.

Bibliografia

[1] http://cplusplus.com/reference/