Termin Egzaminu (Język C):

>> PIĄTEK, 5 LUTEGO 2027 <<

GODZ. 11.00

[*POPRAWKA:* PIĄTEK, 19 LUTEGO]

!!!

Forma zaliczenia kursu: Egzamin pisemny – test wyboru *)

*) Warunkiem przystąpienia do egzaminu jest ***zaliczenie ćwiczeń***
(w uzasadnionych przypadkach: *zgoda prowadzącego ćwiczenia*)

[OCENA KOŃCOWA: *0.5*ocena z ćwiczeń + 0.5*wynik egzaminu* **]**

Co to jest algorytm?

Algorytm, *przepis postępowania prowadzący do rozwiązania ustalonego problemu, określający ciąg czynności elementarnych, które należy w tym celu wykonać.*

[http://encyklopedia.pwn.pl/haslo/algorytm;3867807.html]

Algorytm – *skończony ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadań. Sposób postępowania prowadzący do rozwiązania problemu.*

[https://pl.m.wikipedia.org/wiki/Algorytm]

>> A czy może istnieć ***algorytm nieskończony***? <<

Po co stworzono komputery, czyli od *Logicyzmu* do *Maszyny Turinga* ...



Münchhausen O. Herrfurth pinx

Komputery służą realizacji algorytmów, zaś algorytmy wymyślono, aby programować komputery ...

Logicyzm (*inaczej:* **empiryzm logiczny**) — program redukcji całej matematyki do logiki, sformułowany na przełomie XIX i XX w.

W pierwszej fazie: postulował sprowadzenie twierdzeń z tzw. „wyższych" działów matematyki do twierdzeń dot. arytmetyki liczb naturalnych (Bertrand Russell, Georg Cantor, Ernest Zermelo)

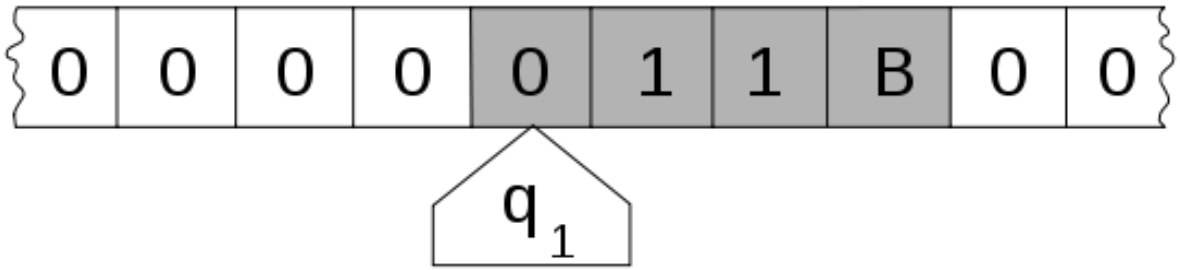
Następnie (lub *równolegle*) – planowano wykazać, że twierdzenia arytmetyki wynikają, po odpowiednim zdefiniowaniu pojęć, wprost z praw logiki (tzw. program *sformalizowania logiki* matematycznej sformułowany przez **Dawida Hilberta**).

Twierdzenie Gödela (1931) — *w ramach każdej teorii matematycznej (zawierającej pojęcie liczb naturalnych) istnieje zdanie, którego nie da się ani udowodnić, ani obalić.*

[**Umowny koniec programu *sformalizowania* matematyki …**]

Alan Turing (1936) — W pracy: *On computable numbers, with an application to the Entscheidungsproblem,* wprowadza pojęcie abstrakcyjnej maszyny (***maszyna Turinga***) oraz liczby obliczalnej, tj. liczby będącej wynikiem (na ogół: *nieskończonego*) ciągu operacji maszyny, wykonywanych wg *skończonego* zestawu instrukcji (dzisiaj powiedzielibyśmy: ***programu***).

***Prawie wszystkie* l.rzeczywiste to tzw. *liczby nieobliczalne* …**



Maszyna Turinga w stanie q_1 nad zerem na taśmie; zob: pl.wikipedia.org

Formalnie, **maszynę Turinga** definiuje *odwzorowanie jednoznaczne:*

$$\delta:\ \Gamma \times Q\ \ \text{—>}\ \ Q \times \Gamma \times \{\ L, P, -\ \}$$

gdzie:

Q — zbiór możliwych stanów maszyny (*skończony!*)

Γ — zbiór dopuszczalnych symboli na taśmie (*skończony!*)

$\{\ L, P, -\ \}$ — przesunięcie głowicy (*lewo, prawo, bez przesunięcia*)

[***Komputer to maszyna … ale nie Turinga!***]

=> *Ograniczenia maszyn Turinga (jak np. tzw.* ***„problem stopu”*** *)*
tym bardziej dotyczą realnych komputerów (**!**)

Przykładowy algorytm (*skończony ...*)

ZADANIE: Oblicz *n!* dla zadanego *n* wg wzoru:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n.$$

[*Rozwiązanie: wymnażamy od końca* ($n \cdot (n-1) \cdot \ldots \cdot 2$).]

1. Zapamiętaj n. (Zakładam: n>1)
2. Zmniejsz n o 1.
3. Sprawdź, czy n==1? Jeśli TAK -> **Koniec.**
   Jeśli NIE:
4. Pomnóż zapamiętaną liczbę przez n.
5. Idź do kroku 2.

Program obliczający *n!* (Wersja 1)

(*A zarazem przykład, jak **nie należy** programować …*)

```
main()
{
   int w, n;
   scanf("%d", &n);
   w = n;
 krok2:
   n -= 1;    /*  n = n - 1;  */
   if (n == 1) goto koniec;
   w *= n;    /*  w = w*n;  */
   goto krok2;
 koniec:
   printf("%d\n", w);
}
```

Program obliczający *n!* (Wersja 2: *tym razem **jak należy!***)

```
int main()
{
   int w, n;
   scanf("%d", &n);
   w = n;
   while (n > 2) {
      n--;   /* n = n-1; nieco krócej … */
      w *= n;  /* w = w*n; */
   }
   printf("%d\n", w);
   return 0;
}
```

[**Inne możliwe zapisy pętli**: `while (n > 2) w *= --n;`

lub np.: `while (n-- > 2) w *= n;`]

Instrukcja *goto*

Instrukcja skoku (`goto etykieta;`) na ogół powoduje zauważalny *wzrost długości kodu* binarnego; **prawie zawsze** pogarsza czytelność programu i utrudnia wprowadzanie zmian.

Tam, gdzie to możliwe – *należy jej unikać!*

Na przykład, do *niestandardowego opuszczenia* bloku pętli doskonale nadaje się instrukcja (*strukturalna!*) `break;`

[*Na ogół w formie:* `if (warunek) break;`].

A co jeśli chcę ***opuścić dwie pętle naraz***?

Przykład - opuszczanie 2 pętli z *goto*:

Sprawdamy, czy w tablicach `a` i `b` występuje taki sam element:

```
for (i=0; i<n; i++) {
   for (j=0; j<m; j++) {
      if (a[i] == b[j])
         goto found;  /* element znaleziony */
   }
}
/* Instrukcje, gdy nie ma elementu */
…
found:  /* znaleziono a[i]==b[j] */
;  /* Po etykiecie - zawsze intrukcja! */
```

Program zawierający instrukcje skoku (`goto`) **ZAWSZE** można napisać bez niej, zwykle kosztem kilku dodatkowych instrukcji; w tym przypadku możemy np. *zastąpić 2 pętle jedną:*

```
found=0;
for (ix = 0; ix < n*m; ix++) {
   j = ix%m;
   i = (ix-j)/m;
   /* A można też:  i = (ix-(j=ix%m))/m;  :-0 */
   if (a[i] == b[j]) {
      found = 1;
      break;
   }
}
if (!found) { /* Instrukcje, gdy nie ma elementu */ }
else { /* znaleziono a[i]==b[j] */ }
```

Program można jeszcze nieco skrócić zapisując pętlę tak:

```
for (found=0,ix=0; ix<n*m && !found; ix++) {
   …
   if (a[i]==b[j])
      found=1;
}
```

Nie nie potrzeba wówczas break w warunku: `if (a[i]==b[j])`.

[Frazy `"&& !found"` można też *użyć w 2 pętlach*, tj. dla `i` oraz `j`]

Kernighan & Ritchie, 1988:

„Nie jesteśmy w tej sprawie dogmatyczni, wydaje się jednak, że jeśli instrukcja `goto` *ma być w ogóle stosowana, to powinna być stosowana* ***rzadko****.”*

[*Na pocieszenie — w C mamy* ***wskaźniki na funkcje …***]

Algorytmy nieskończone (1)

Dwa problemy do samodzielnego przemyślenia:

(a) Oblicz liczbę π wg wzoru Wallisa (1655)

$$\prod_{n=1}^{\infty}\left(\frac{2n}{2n-1}\cdot\frac{2n}{2n+1}\right)=\frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\cdot\frac{8}{9}\cdots=\frac{\pi}{2}$$

(b) Oblicz liczbę e (tj. *podstawę logarytmu naturalnego*)

$$e=\sum_{n=0}^{\infty}\frac{1}{n!}=\frac{1}{1}+\frac{1}{1}+\frac{1}{1\cdot 2}+\frac{1}{1\cdot 2\cdot 3}+\cdots$$

W obu przypadkach, możliwe jest stworzenie algorytmu, którego wynik różni się od dokładnego o dowolnie małe ε>0.

Po ***nieskończenie długim czasie*** otrzymamy zatem ***wynik dokładny*** (!)

[**nieskończoność potencjalna** | *nieskończoność aktualna*]

Algorytmy nieskończone (2)

PROSTY problem do samodzielnego rozwiązania:

Oblicz wartość numeryczną $\sqrt{2}$ korzystając z działań elementarnych `+, -, *, /`

Wskazówka: Zdefiniuj pomocniczo funkcję $y = x^2 - 2$ i poszukaj jej miejsca zerowego w przedziale x ∈ [0,2] stosując np. **algorytm równego podziału (*bisekcji*)** [lub *metodę Newtona*].

Obliczenia należy prowadzić do momentu, gdy poprawa przybliżenia nie będzie już możliwa w ramach arytmetyki ustalonej precyzji (`float, double, long double`)

Algorytmy probabilistyczne

W sytuacji, gdy nie znamy *deterministycznego* przepisu na obliczenie interesującej nas wielkości (X) niekiedy można pokazać, że istnieje ***proces losowy*** (np. przebiegający wg schematu *prób Bernouliego*), dla którego X (zakładamy, że $0 < X < 1$) jest **prawdopodobieństwem sukcesu**.

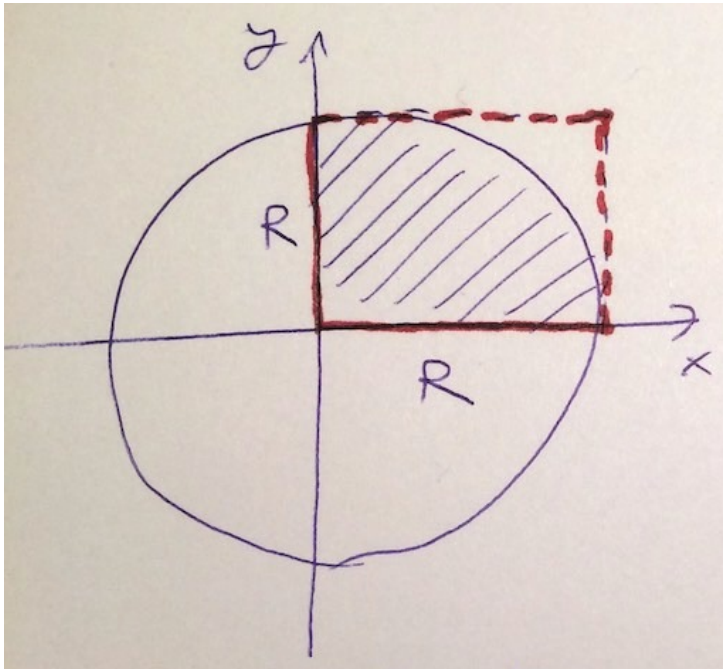
Wówczas — co prawda nie potrafimy X obliczyć — możemy jednak zaprząc komputer do symulacji, podczas której *średnia częstość* występowania interesującego nas zdarzenia będzie równa X.

[=> **Metody *Monte-Carlo***, *generacja* ***liczb pseudolosowych***]

Czas obliczeń: T ~ log(1/ε) dla aa. deterministycznych,

T ~ $1/\varepsilon^2$ dla metod Monte-Carlo.

Przykład: Obliczanie π metodą Monte-Carlo



Mamy:

$$\frac{\text{Pole } 1/4 \text{ koła}}{\text{Pole kwadratu}} = \frac{\frac{1}{4}\pi R^2}{R^2} = \frac{\pi}{4}$$

$\Rightarrow$ Zarazem, $\pi/4$ to prawdopodobieństwo, że dla **losowo wybranych** x i y

[zakładam: $0 \leq x < R$, oraz $0 \leq y < R$]

punkt o współrzędnych (x,y) leży ***wewnątrz okręgu***.

A czy można obliczyć e metodą Monte Carlo?

>> MOŻNA ! <<

Permutacje bez punktu stałego (*„problem sekretarki”*).
P-stwo, że losowo wybrana (spośród $n!$ permutacji zbioru n elementów) *nie pozostawia żadnego elementu na swoim miejscu*:

$$P_0(n) = 1 + \frac{-1}{1!} + \frac{(-1)^2}{2!} + \cdots + \frac{(-1)^n}{n!}$$

W granicy n —> ∞ dostajemy: $P_0 \to e^{-1} = 1/e \approx 0.3679$.

Ćwiczenie: Napisz program, który generuje *losową, reprezentatywną próbę* np. 10^6 spośród 100! permutacji 100 elementów, i sprawdza, jaki jest udział (w tej próbie) permutacji bez punktu stałego.

Wskazówka: *Każdą permutację można otrzymać dokonując skończonej liczby transpozycji sąsiednich elementów permutacji identycznościowej.*

Poprzedni wykład:

- Stałe i *wyrażenia stałe* w języku C
- Deklaracje/definicje (=> zasięg zmiennych, zmienne zewnętrzne i *zmienne automatyczne*)
- Operatory arytmetyczne; relacje i operatory logiczne
- Interpretacja logiczna liczb całkowitych
- ***Przekształcenia typów***

Przekształcenia typów

Jeśli argumenty operatora są różnych typów, dokonuje się przekształcenia typu „ciaśniejszego” do „obszerniejszego”.

[*Zawsze w ramach **jednego** wyrażenie składowego **!!!***
=> wyrażenie: `1/3*x == 0.0` *dla* `x` *typu **double***]

<u>Praktyczny zestaw reguł</u> (nie dotyczy unsigned …):

- Jeśli którykolwiek z argumentów jest typu `long double`, drugi zostanie przekształcony do `long double`.
- W przeciwnym przypadku, jeśli typem któregokolwiek argumentu jest `double`, to drugi argument będzie przekształcony do `double`.

- W przeciwnym przypadku, jeśli typem któregokolwiek z argumentów jest `float`, to drugi argument będzie przekształcony do `float`.
- W przeciwnym przypadku, obiekty typu `char` i `short` są przekształcane do `int`.
- Ostatecznie, jeśli którykolwiek z argumentów ma kwalifikator `long`, to drugi zostanie przekształcony do `long`.

Reguły przekształceń ***komplikują się*** dla argumentów `unsigned`.

Odpowiedź na pytanie, który z dwóch typów (`unsigned` i *coś ze znakiem*) jest *ciaśniejszy*, a który *obszerniejszy,* może zależeć od maszyny. Jeśli int ma 16 bitów a long — 32, wówczas `-1L<1U` (a jednocześnie `-1L>1UL`); jeśli oba mają 32 bity: `-1L>1U`.

Rzutowanie

Aby **wymusić przekształcenie** typu (*w pożądanym kierunku!*), używamy operatora rzutowania (ang. *cast*), np.

`sqrt((double)n)`, albo `(float)i/n`.

Podobnie, możliwe jest rzutowanie typu *obszerniejszego* na *ciaśniejszy*, np. `(int)x` dla zmiennej typu float (lub double).

W takim przypadku, wynik jest *obcinany* (do l.całkowitej *mniejszej lub równej* x) jedynie dla x – **nieujemnego**; dla ujemnego – „obcięcie w górę"; nie jest to zatem *matematycznie* „*część całkowita* x".

[*Istnieją funkcje biblioteczne:* `double floor(double x);` `double ceil(double x);` zdefiniowane w `<math.h>`]

Ważne: *Rzut produkuje wartość argumentu w odpowiednim typie; sam argument pozostaje niezmieniony.*

Operatory zwiększania i zmniejszania

W C mamy specyficzne dla tego języka, jednoargumentowe operatory *zwiększania* i *zmniejszania* wartości zmiennych:

> Operator ++ **dodaje** 1 do swojego argumentu;
> operator `--` **odejmuje** 1.

Każdy z nich może występować w wersji *przedrostkowej* (`++n`) lub *przyrostkowej*: `n++`. [Podobnie, mamy: `--n` lub `n--`]

[*W pierwszym przypadku, wyrażenie* `++n` *zwiększa wartość* `n` ***przed*** *jej użyciem; zaś* `n++` *zwiększa* `n` ***po*** *użyciu poprzedniej wartości.*]

Jeśli np. `n == 5`; instrukcja:

```
x=n++;
```

nadaje zmiennej x wartość 5, zaś n — wartość 6, natomiast instrukcja:

```
x=++n;
```

nadaje obu zmiennym, tj. x i n, wartość 6.

Operatory bitowe

W C mamy *sześć operatorów* pozwalających na *manipulacje bitami*, 5 dwuargumentowych i 1 jednoargumentowy; można je stosować jedynie do argumentów całkowitych, tj. typów

`char`, `short`, `int`, lub `long`,

zarówno w wersjach ze znakiem jak i bez znaku. Są to:

- `&` bitowa koniunkcja (AND),
- `|` bitowa alternatywa (OR),
- `^` bitowa *różnica symetryczna* (ang. exclusive-or, XOR),
- `<<` przesunięcie w lewo,
- `>>` przesunięcie w prawo,
- `~` dopełnienie bitowe (*operator 1–argumentowy*).

Bitowy **operator koniukcji** (&) służy do *„zasłaniania”* (*zerowania*) wybranego zbioru bitów; z kolei operator alternatywy (|) służy do *„ustawiania”* (zastępowania jedynkami) wybranych bitów.

Przykładowo, instrukcja

```
x = x & ~077;       /*  lub:  x &= ~077;  */
```

spowoduje wyzerowanie *sześciu ostatnich* bitów x
[*A wynik zupełnie* ***nie zależy od tego, ile bitów ma x!***].

Np, dla 8 bitów (unsigned char) stała 077 (dziesiątkowo: 63) reprezentowana jest przez ciąg: `00111111` zaś ~077 to `11000000`; w przypadku *obszerniejszych* typów przybywa jedynek z przodu.

Spostrzeżenie: Instrukcja `x ^= MASKA;` **jest odwracalna,** tzn. jeśli wykonać ją powtórnie — zmienna x odzyska początkową wartość.

Operatory i wyrażenia przypisania

Wyrażania podobne do: `i = i + 2`

można zastępować krótszymi: `i += 2`

[Np. `x *= y + 1` jest odpowiednikiem: `x = x * (y + 1)`]

W C istnieją zatem tzw. ***operatory przypisania***:

`+=` `-=` `*=` `/=` `%=` `<<=` `>>=` `&=` `^=` `|=`

Funkcja **zliczająca bitowe jedynki** argumentu x:

```
int bitcount(unsigned x)  /* NIE MA bitu znaku! */
{  int b;
   for (b = 0; x != 0; x >>= 1)
      if (x & 01u)  b++;
   return b;
}
```

Wyrażenia warunkowe (*czyli* operator "?:")

Instrukcja:

```
if (a > b)  z = a;
else z = b;
```

może zostać zastąpiona przez:

```
z = (a > b) ? a : b;    /* Nawiasy ( ) można pominąć */
```

[**Operator „?:" to JEDYNY *trójargumentowy* operator w C !**]

Po prawej stronie przypisania mamy **wyrażenie warunkowe** [TUTAJ: (a > b) ? a : b], które jest *pełnoprawnym wyrażeniem,* ma zatem swoją *wartość* i może zostać użyte wszędzie tam, gdzie inne wyrażenia. ***Przykład:***

```
printf("Mamy %d część%s.\n", n, n==1 ? "ć" : "ci");
```

UWAGI o wyrażeniu warunkowym „?:”

Operator ?: wiąże ***silniej*** niż przypisanie (=), czy operator przecinkowy (,) ale ***słabiej*** niż np. operatory relacji (< , > , ==).

Przykładowo, w warunku zakończenia pętli:

```
for (x = 0.0; x < (y>1.0 ? y : 1.0); x += 0.1)
```

nie można pominąć nawiasów „ ( ) ” , w przeciwnym wypadku wyrażenie ulegnie „rozerwaniu” przez pierwszy „<”.

Jest to szczególnie istotne w ***makrodefinicjach*** zawierających „?:”

```
#define MAX(a,b) (((a)>(b))?(a):(b))
#define ROUND(x) ((x)>=0?(int)((x)+0.5):(int)((x)-0.5))
```