

Sortowanie – prosty przykład

Dany jest ciąg wartości liczbowych, które należy uporządkować w kolejności rosnącej. Algorytm, który zostanie zastosowany do rozwiązania tego zadania nazywa się *sortowaniem bąbelkowym*, z uwagi na fakt, że w oryginalnej wersji polega on na tym, żeby po kolei przemieszczać najmniejsze wartości na początek ciągu, co ma pewną analogię z pęcherzykami powietrza (lekkie!) zmiernymi ku górze w kieliszku wypełnionym musującym napojem. W poniższym opisie dokonamy pewnej modyfikacji tego algorytmu: zamiast przemieszczania lekkich elementów ‘ku górze’, będziemy powodować opadanie ‘na dno’, czyli przemieszczanie na koniec ciągu, elementów o największych wartościach.

Przed precyzyjnym opisaniem algorytmu zwracam uwagę na fakt, że wartości kolejnych elementów ciągu najlepiej jest przechowywać w wektorze.

Zadanie formułujemy następująco:

Dany jest n -elementowy wektor x zawierający wartości rzeczywiste, które należy uporządkować w kolejności rosnącej.

Idea: Będziemy porównywać kolejne elementy parami. W przypadku gdy element pierwszy jest większy od drugiego, należy zamienić je miejscami. Następnie porównuje się element drugi z trzecim, dokonując ewentualnej zamiany. Po dokonaniu ostatniego porównania elementu przedostatniego i ostatniego, największy element wektora znajdzie się na ostatnim miejscu.

Tę procedurę można powtórzyć, ponownie zaczynając od porównania elementu pierwszego i drugiego, drugiego i trzeciego, itd., tym razem jednak nie ma potrzeby dokonywać porównania z ostatnim elementem wektora (bowiem wiadomo, że już znajduje się tam największy element). W wyniku na miejscu przedostatnim powinien znaleźć się drugi co do wielkości element.

Przy każdym kolejnym obiegu powyższej procedury liczba porównywanych elementów zmniejsza się o jeden, aż do zakończenia sortowania.

Realizacja:

Rozbijemy powyższy szkic algorytmu na czynności elementarne. Rozpocznijmy od najbardziej podstawowej, a mianowicie:

- **zamiany miejscami dwóch elementów wektora.** Fragment realizującego to zadanie programu może wyglądać następująco:

```
if (x(i)>x(i+1)) then ! zamiana miejscami
  c = x(i)
  x(i) = x(i+1)
  x(i+1) = c
endif
```

Przypominamy, że zamiana dwóch zmiennych wartościami wymaga wykorzystania zmiennej pomocniczej, tu rolę tę pełni zmienna c .

- **Powtórzenie tego ciągu czynności** (porównanie i ewentualna zamiana miejscami) dla całego wektora. Przypomnijmy, że zgodnie z ideą algorytmu liczba porównywanych ze sobą elementów wektora zmniejsza się w miarę zapełniania końcowych pozycji wektora elementami największymi. Załóżmy że, porównanie i zamiana będą wykonywane kolejno dla wszystkich elementów wektora o długości oznaczonej zmienną k .

Porównywać będzie się elementy dla indeksu i zmieniającego się od 1 (co oznacza porównanie elementu 1 (i) z elementem 2 ($i+1$), z krokiem 1 (bo rozważamy po kolei wszystkie elementy, aż do $k-1$ (co oznacza porównanie elementu $k-1$ (i) z ostatnim elementem k ($i+1$)). Osiągnięto się to ‘obudowując’ powyższe porównanie i zamianę instrukcją cyklu:

```
do i = 1, k-1
  if (x(i)>x(i+1)) then ! zamiana miejscami
    c = x(i)
```

```

        x(i) = x(i+1)
        x(i+1) = c
    endif
enddo

```

- Ostatnim krokiem jest udzielenie odpowiedzi na pytanie: jak zmienia się wartość k , czyli liczba elementów, które są porównywane i zamieniane miejscami. Przy pierwszym przebiegu należy rozważyć cały wektor (czyli $k=n$), przy drugim liczba elementów maleje o 1 ($k=n-1$), itd. W ostatnim kroku procedury porównuje się ze sobą tylko dwa elementy ($k=2$), bowiem cała reszta ciągu jest już uporządkowana. A zatem k zmienia się według schematu: $k=n, 2, -1$, co od razu wykorzystujemy dla zdefiniowania zewnętrznej pętli algorytmu:

```

do k = n, 2, -1
  do i = 1, k-1
    if (x(i)>x(i+1)) then ! zamiana miejscami
      c = x(i)
      x(i) = x(i+1)
      x(i+1) = c
    endif
  enddo
enddo

```

Pętla ta mówi, że długość rozważanego wycinka wektora zmniejsza się o jeden przy każdym obiegu algorytmu.

Na zakończenie podajemy cały program:

```

program babelki
implicit none
real, dimension (100) :: x
real :: c
integer :: n, k, i
write (*,*) 'liczba elementow wektora'
read (*,*) n
! write (*,*) 'podaj elementy wektora'
! read (*,*) (x(i), i=1,n)
! LOSOWANIE
call random_number (x)
! sortowanie babelkowe: kolejne przemieszczanie najciezszeo
! (czyli największego) elementu na koniec
do k = n, 2, -1
  do i = 1, k-1
    if (x(i)>x(i+1)) then ! zamiana miejscami
      c = x(i)
      x(i) = x(i+1)
      x(i+1) = c
    endif
  enddo
enddo
write (*,*) '===== '
do i = 1, n
  write (*,*) x(i)
enddo
end program babelki

```

Uwaga na temat danych do testowania zadania:

W powyższym programie podano dwa sposoby zdefiniowania danych dla programu. Jeden z nich to kolejno wczytanie liczby elementów wektora

```
read (*,*) n  
oraz następnie wartości jego elementów  
read (*,*) (x(i), i=1,n).
```

Ten fragment programu znalazł się w komentarzu. Zamiast niego aktywna jest instrukcja:

```
call random_number (x)
```

w której następuje wywołanie funkcji standardowej Fortranu będącej generatorem liczb pseudolosowych z przedziału (0,1). Jest to funkcja bardzo przydatna do generowania dużych zestawów danych testowych, z czego właśnie korzystamy. Zwrócić też należy uwagę na sposób wywołania (frazą `CALL`) oznaczający, że funkcja `random_number` jest podprogramem typu `SUBROUTINE`. Temat ten zostanie omówiony później.