Podstawowe koncepcje w programowaniu (4)

Problem V: sortowanie tablicy (porządkowanie elementów tablicy)

var a: **array** [1..n] **of** integer;

Uporządkować zawartość tablicy niemalejąco, tzn. tak by:

a[1] <= a[2] <= . . . . <= a[n]

Jeden z najważniejszych problemów algorytmicznych.
Istnieje wiele algorytmów sortowania.

Podstawowe koncepcje w programowaniu (4)

Sortowanie bąbelkowe zwane też jest **sortowaniem przez prostą zamianę** (bubble sort)

Pojedyncza faza: porównywanie kolejnych sąsiednich par i zamiana, jeśli lewy mniejszy niż prawy: a[1] z a[2], i ewentualna zamiana, a[2] z a[3], a[3] z a[4], itd.

indeksy:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
wartości:	**5**	**2**	**4**	**8**	**3**	**6**	**7**	**1**
po fazie 1:	**2**	**4**	**5**	**3**	**6**	**7**	**1**	**8**
po fazie 2:	**2**	**4**	**3**	**5**	**6**	**1**	**7**	**8**
po fazie 3:	**2**	**3**	**4**	**5**	**1**	**6**	**7**	**8**
po fazie 4:	**2**	**3**	**4**	**1**	**5**	**6**	**7**	**8**
po fazie 5:	**2**	**3**	**1**	**4**	**5**	**6**	**7**	**8**
po fazie 6:	**2**	**1**	**3**	**4**	**5**	**6**	**7**	**8**
po fazie 7:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Podstawowe koncepcje w programowaniu (4)

Algorytm pojedynczej fazy (uproszczony, gdyż nie bierze pod uwagę że kolejne fazy mogą obejmować coraz krótszą część tablicy)

```
for i:= 1 to n-1 do
    if a[i] > a[i+1] then
      begin // zamiana
         tmp := a[i];
         a[i] := a[i+1];
         a[i+1] := temp
      end;
```

Podstawowe koncepcje w programowaniu (4)

Wystarcza powtórzyć fazę n-1 razy by otrzymać algorytm sortowania:

```
// sortowanie bąbelkowe – wersja uproszczona
for k := 1 to n-1 do
 for i:= 1 to n-1 do
     if a[i] > a[i+1] then
  begin // zamiana
         tmp := a[i];
         a[i] := a[i+1];
         a[i+1] := tmp
       end;
```

../Przykłady/BubbleSortv1.pascal

Teraz metodę nieco usprawnimy. Zauważmy, że w kolejnych fazach można pomijać coraz dłuższy obszar końcowy tablicy, zawierający już ostateczne wartości. Wystarczy pętlę ustawić tak, by:

Faza 1: ostatnia porównywana para	a[n-1], a[n]
Faza 2: ostatnia porównywana para	a[n-2], a[n-1]
itd.	
Faza ostatnia:	a[1], a[2]

W każdej fazie pierwsza porównywana para to a[1], a[2].

Niech zmienna ost zawiera indeks lewego elementu ostatniej pary porównywanej w danej fazie. W fazie 1 ost=n-1, w fazie końcowej ost=1.

```
// sortowanie bąbelkowe – wersja poprawiona
for ost:= N-1 downto 1 do
  for i:= 1 to ost do
    if  a[i] > a[i+1] do
         begin // zamiana
                tmp := a[i];
                a[i] := a[i+1];
          a[i+1] := tmp
          end;
```

Liczba wykonywanych porównań:
Wersja uproszczona: (N–1)*(N–1) porównań (około N^2)
Wersja poprawiona: (N–1) + (N–2) + . . . + 2 + 1 = 0.5*N*(N–1) (około połowę mniej).

../Przykłady/BubbleSortv2.pascal

Podstawowe koncepcje w programowaniu (4)

Sortowanie bąbelkowe można jeszcze usprawnić wprowadzając przeglądanie ciągu w obie strony, zarówno w kierunku „ciężkiego” jak i „lekkiego” końca i ograniczając zakres przebiegów poprzez rejestrację punktu ostatniej zamiany.

Zastosowanie sortowania tablic

Wyszukiwanie połówkowe (wyszukiwanie binarne, bisekcja)

```
const  n=1000;
                    var a : array [1..n] of integer;
  // wygenerowano uporządkowaną zawartość tablicy
                     // wczytano wartość zmiennej x
                     // PYTANIE: CZY x WYSTĘPUJE W
TABLICY a  ?
 // lewy, prawy, srodek – zmienne robocze (indeksy w
 //tablicy a)
```


Wyszukiwanie połówkowe (wyszukiwanie binarne, bisekcja)

```
lewy :=0; prawy :=n; znal := false;
while prawy-lewy >= 0 and not znal do
  begin
    srodek := (lewy + prawy) div 2       // (*)
    // zał. wynik dzielenia całkowitych liczb jest
    // całkowity, reszta z dzielenia jest obcięta
    if x = a[srodek] then znal := true
    else if x < a[srodek] then prawy := srodek - 1
                          else lewy := srodek + 1
  end;
  if znal then begin
                  write('znaleziono, indeks = ');
                  write( srodek )
               end
            else
            write('nie ma takiego elementu w tablicy');
```

Test algorytmu wyszukiwania połówkowego

Tablica a:	**2,**	**3,**	**6,**	**8,**	**9,**	**12,**	**14,**	**15,**	**17,**	**20**
Indeksy:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

x=14

Wartości zmiennych po wykonaniu instrukcji (*)

	lewy,	prawy	srodek	
iteracja 1:	**1**	**10**	**5**	
iteracja 2:	**6**	**10**	**8**	
iteracja 3:	**6**	**9**	**7**	**KONIEC**

Porównywane elementy (wartości) kolejno: 9,15,14

PYTANIE: Jakie wartości są porównywane dla x=5?
UWAGA: Wyszukiwanie połówkowe jest szybkie
– ma złożoność około $\log_2 n$.

Obliczanie wartości wielomianu: schemat Hornera

Wielomian k-tego stopnia:

$P(x) = a_k x^k + a_{k-1} x^{k-1} + ... + a_1 x + a_0$

można przedstawić w postaci (schemat, lub reguła, Hornera):

$P(x) = ((\ldots ((a_k x + a_{k-1}) x + a_{k-2}) x + \ldots + a_2) x + a_1) x + a_0$

Algorytm obliczania wartości wielomianu dla argumentu x wg schematu Hornera:

```
const       k= … ;
var a : array [0..k]of integer;    // współczynniki
                                  //  wielomianu
                       // w kolejności od a(0) do a(k)
.....
read(x);
y := a[k];      // współczynnik a(k)
for i:= k-1 to 0 do y := y*x + a[i];
write(y);
```

Przykład: $P(x)=2x^4 - x^3 + 3x + 6$, $k=4$,

Tablica a:	6	3	0	-1	2
Indeksy tablicy a:	0	1	2	3	4

przed iteracją: $y = 2$

po iteracji dla i=3: $y = 2x - 1$

po iteracji dla i=2: $y = (2x-1)x + 0 = 2x^2 - x$

po iteracji dla i=1: $y = (2x^2-x)x + 3 = 2x^3-x^2+3$

po iteracji dla i=0: $y = (2x^3-x^2+3)x + 6$

$= 2x^4 - x^3 + 3x^2 + 6$

Algorytm Hornera wykonuje 2 razy mniej operacji mnożenia niż algorytm zaprojektowany wg klasycznej definicji wielomianu.

