Vyhľadávanie

Vyhľadávanie

Vstupy:
- N prvkov identifikovateľných kľúčom
- kľúč K, ktorý charakterizuje prvok, ktorý chceme nájsť

Výstupy:
- Úspech : prvok sa v zadanej množine našiel
- Neúspech: prvok sa nenašiel

Rozdelenie vyhľadávaní

- vnútorné, vonkajšie
- statické, dynamické
- uporiadaná, neusporiadaná postupnosť
- lineárne, binárne, pomocou rozptylovej funkcie

Jednoduché vyhľadávanie

Vstup: pole prvkov, kľúč
Výstup: pozícia nájdeného prvku v poli (ak sa prvok nenájde, tak sa vráti 0)

```
Procedure SEARCH(POST:pole, K: kľúč, var KDE: integer )
VAR NAJDENY : BOOLEAN;
    I  : 1..VELKOST;
BEGIN
   NAJDENY := FALSE;
   I := 1;
   WHILE((I <= VELKOST) AND (NOT NAJDENY)) DO
        IF K = POST[I].KLUC THEN
            BEGIN
                KDE := I;
                NAJDENY := TRUE;
            END
        ELSE
            I ++;
    IF NOT NAJDENY THEN
        KDE := 0;
END;
```


Jednoduché vyhľadávanie - lineárne

```
// vracia true akk existuje integer I také,
// že arr[i] == target a 0 <= i < arr.length

 boolean linearSearch(int[] arr, int target)
 {
        int i = 0;
        while (i < arr.length) {
          if (arr[i] == target) {
                return true;
          } // if
          ++i;
        }
        return false;
 }
```


Jednoduché vyhľadávanie - lineárne

```
type IntList is array(1..Max) of Integer;

function Search(List: IntList; Key: Integer)
return Integer
begin
   for I in List.Range loop
        if List(I) = Key then
               return I;
        end if;
   end loop;
return 0;
end Search;
```

popodmienka:
(List (I) = *Key*) $\vee$
$((I = 0) \wedge \forall J(1 \leq J \leq Max \mid (List\ (J) \neq Key))$

Jednoduché vyhľadávanie – lineárne so zarážkou

```
type IntList is array(0..Max) of Integer;

function Search(List: IntList; Key: Integer)
return Integer
I: Integer := Max;
begin
   List(0) := Key;
   while List(I) ≠ Key loop
            I := I - 1;
   end loop;
return I;
end Search;
```

invariant cyklu:
$\forall J(I < J \leq Max \mid List\ (J) \neq Key)$

Binárne vyhľadávanie

- Vstupom je usporiadané pole.
- Algoritmus porovná prvok nachádzajúci sa v strede poľa so zadaným kľúčom. Ak sa zhoduje, vráti jeho pozíciu. Ak sa nezhoduje, rozdelí pole na 2 polovice a pokračuje rovnakým hľadaním v tej polovici v ktorej sa prvok nachádza.

- Rekurzívna verzia algoritmu:

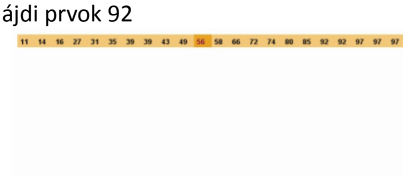
```
BinarySearch(A[0..N-1], value, low, high)
{
   if (high < low)
     return not_found
   mid = (low + high) / 2
   if (A[mid] > value)
     return BinarySearch(A, value, low, mid-1)
   else if (A[mid] < value)
     return BinarySearch(A, value, mid+1, high)
   else
     return mid
}
```


Binárne vyhľadávanie s cyklom

```
function Search(List: IntList; Key: Integer)
return Integer
          Low: Integer := List'First;
          High: Integer := List'Last;
          Mid: Integer;
          begin
          loop
                    Mid := (Low + High) / 2;
                    if Key = A[Mid] then return Mid;
                              elsif Key < A[Mid] then High := Mid - 1;
                              else Low := Mid + 1;
                    end if;
                    if Low > High then return 0;
          end loop;
end Search;
```


Binárne vyhľadávanie

- Nájdi prvok 92



Usporadúvanie

Usporadúvanie

- Postupnosť: a_1, a_2, a_3...a_n
- Položka:
  - a_i
  - Kľúč k_i položky a_i $k(a_i)$

Usporiadanie: binárna relácia

Úlohou je preusporiadať všetky položky postupnosti tak, aby platilo:

K1 <= K2 <= K3 <= ... <= Kn

Usporadúvanie

- Nech K je nekonečná lineárne usporiadaná množina (ďalej skratka LUM), t.j.
- množina, na ktorej je definovaná relácia < taká, že sú splnené tieto 2 podmienky:
  - zákon trichotómie
    - Pre ľubovoľné dva prvky K1, K2 ∈ K platí práve jedna z relácií : K1 < K2, K1 = K2, K1 > K2.
  - tranzitívny zákon
    - Pre ľubovoľné tri prvky K1, K2, K3 ∈ K platí : Ak K1 < K2 a K2 < K3, tak K1 < K3.
- Pre ľubovoľné K1, K2 ∈ K budeme písať, že K1 ≤ K2 akk K1 < K2 alebo K1 = K2.

13

Usporadúvanie

- Nech D je nekonečná množina a nech je na nej definovaná funkcia k : D → K.
- Definícia (problém usporadúvania). Nech je daná postupnosť a1, ..., an, kde n ∈ N; ai ∈ D. Potom všeobecný problém usporadúvania tkvie v určení takej permutácie π čísel 1,...,n, že platí $k(a_{\pi(1)}) \leq k(a_{\pi(2)}) \leq \ldots \leq k(a_{\pi(n)})$
- D – množina údajov
- k – funkcia usporiadania, daná zvyčajne explicitne pre každý prvok ako jeho zložka tzv. kľúč.
- Definícia. Permutácia n-prvkovej množiny je ľubovoľné usporiadanie prvkov tejto množiny do postupnosti.

14

Stabilný algoritmus

- Usporadúvací algoritmus je stabilný, ak vždy zachová originálne poradie elementov s rovnakými kľúčmi
- Ak elementy s rovnakými kľúčmi sú neodlíšiteľné, tak nie je potrebné sa zaoberať stabilitou algoritmu (napr. ak kľúčom je samotný element)
- Zachovať originálne poradie elementov je dôležité napr. pri viacnásobnom usporiadaní – najprv podľa priezviska a potom podľa mena.

15

Stabilný algoritmus

- Každý nestabilný algoritmus sa dá implementovať ako stabilný tým, že sa zapamätá originálne poradie elementov a pri zhodných kľúčoch sa berie do úvahy toto poradie
- Viacnásobné usporiadanie je možné obísť vytvorením jedného kľúča, ktorý je zložený z primárneho, sekundárneho, atď. kľúča usporiadania
  - Takéto úpravy nestabilných algoritmov majú negatívny vplyv na výpočtovú zložitosť.

16

Stabilný algoritmus

- Príklad – dvojice (kľúč, element):
  - (4, 5) (2, 7) (2, 3) (5, 6)
- Dve možné usporiadania:
  - **(2, 7) (2, 3)** (4, 5) (5, 6) – zachované poradie elementov s kľúčmi 2 – stabilné usporiadanie
  - **(2, 3) (2, 7)** (4, 5) (5, 6) – zmenené poradie elementov s kľúčmi 2 – nestabilné usporiadanie
- Príklad na viacnásobné usporiadanie – dvojice (kľúč 1, kľúč 2):
  - (4, 5) (2, 7) (2, 3) (4, 6)
- Usporiadanie najprv podľa kľúča 2, potom podľa kľúča 1:
  - (2, **3**) (4, **5**) (4, **6**) (2, **7**) – podľa kľúča 2
  - (**2**, 3) (**2**, 7) (**4**, 5) (**4**, 6) – podľa kľúča 1
- Usporiadanie najprv podľa kľúča 1, potom podľa kľúča 2:
  - (**2**, 7) (**2**, 3) (**4**, 5) (**4**, 6) – podľa kľúča 1
  - (2, **3**) (4, **5**) (4, **6**) (2, **7**) – podľa kľúča 2 – narušené poradie
- Pre zachovanie stability viacnásobného usporadúvania je potrebné usporadúvať postupne podľa kľúčov so zvyšujúcou sa prioritou.

17

Porovnávací algoritmus

- usporadúvací algoritmus, ktorý prechádza vstupné kľúče a na základe operácie porovnávania rozhoduje, ktorý z dvoch elementov sa má v usporiadanom poli objaviť ako prvý.
- Operácia porovnávania musí mať tieto vlastnosti:
  - Ak $a \leq b$ a $b \leq c$, tak potom $a \leq c$
  - Pre všetky a a b, buď $a \leq b$ alebo $b \leq a$
- Základným limitom je dolné ohraničenie počtu porovnávania Ω(n log n), ktoré je potrebné na usporiadanie postupností. Preto aj tie najlepšie algoritmy usporadúvania založené na porovnávaní, majú priemernú časovú zložitosť O (n log n) – na rozdiel od neporovnávacích algoritmov, kde sa môže dosiahnuť časová zložitosť aj O (n).

18

Výhody porovnávacích algoritmov

- Použiteľné pre rôzne dátové typy
- Jednoduchá implementácia porovnávania n-tíc v lexikografickej postupnosti
- Reverzná funkcia porovnávania = reverzne usporiadaná postupnosť

Najznámejšie algoritmy usporadúvania

- založené na porovnávaní:
  - usporadúvanie výberom,
  - vkladaním,
  - výmenou,
  - zlučovaním,
  - quicksort,
  - heapsort, ...
- neporovnávacie algoritmy:
  - radix sort,
  - counting sort,
  - bucket sort, ...

Usporadúvanie vkladaním

- algoritmus, ktorý realizuje usporadúvanie priamym vkladaním.
- vychádza z predpokladu, že do už usporiadanej postupnosti sa na správne miesto vloží ďalší prvok.
- ak sa miesto, na ktoré sa nový prvok vkladá, zisťuje binárnym vyhľadávaním, hovoríme o usporadúvaní binárnym vkladaním

Usporiadanie vkladaním

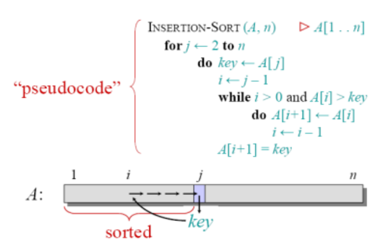
- Príklad:
  - usporiadajte vkladaním pole **6 4 5 2 3 1 7**

  - 1. krok 6 | **4** 5 2 3 1 7 –> **4** 6 | 5 2 3 1 7
  - 2. krok 4 6 | **5** 2 3 1 7 –> 4 **5** 6 | 2 3 1 7
  - 3. krok 4 5 6 | **2** 3 1 7 –> **2** 4 5 6 | 3 1 7
  - 4. krok 2 4 5 6 | **3** 1 7 –> 2 **3** 4 5 6 | 1 7
  - 5. krok 2 3 4 5 6 | **1** 7 –> **1** 2 3 4 5 6 | 7
  - 6. krok 1 2 3 4 5 6 | **7** –> 1 2 3 4 5 6 **7** |

  - 7. krok 1 2 3 4 5 6 7 | –> 1 2 3 4 5 6 7 |

Usporiadanie vkladaním

- Časová zložitosť závisí od vstupného poľa
  - pre takmer usporiadané vstupné pole algoritmus prebehne rýchlo – vtedy sa akoby vymieňali len dva susedné prvky (najpravejší z usporiadanej časti s najľavejším z neusporiadanej časti) a nedochádza tak k posunu ostatných prvkov.



Usporadúvanie vkladaním

```
Procedure InsertionSort(var A: pole)
Var i, j, x : Integer;

BEGIN
  for i:= 2 to Dlzka(A) do begin
     x := A[i];
     a[0] := x;
     j := i - 1;
     while x < A[j] do
          begin
               a[j+1] := a[j];
               j := j - 1;
          end;
     a[j+1] := x;
  end
```


Insertion Sort

INSERTION-SORT(A)

1. **for** j = 2 to length[A]
2. **do** *key* ← A[j]
3. //insert A[j] to sorted sequence A[1..j-1]
4. i ← j-1
5. **while** i >0 and A[i]>*key*
6. **do** A[i+1] ← A[i] //move A[i] one position right
7. i ← i-1
8. A[i+1] ← *key*

správnosť algoritmu Insertion Sort

- invariant cyklu
  - na začiatku každej iterácie obsahuje podpole A[1..j-1] pôvodné hodnoty z A[1..j-1] ale v usporiadanom poradí.
- dôkaz:
  - inicializácia : j=2, A[1..j-1]=A[1..1]=A[1], je usporiadané.
  - udržiavanie: každý krok cyklu udržiava platnosť invariantu.
  - ukončenie: j=n+1, takže A[1..j-1]=A[1..n] je usporiadané.

analýza algoritmu Insertion Sort

INSERTION-SORT(A)	cena	počet opakovaní
1. **for** j = 2 to length[A]	c_1	n
2. **do** *key* ← A[j]	c_2	n-1
3. //insert A[j] to sorted sequence A[1..j-1]	0	n-1
4. i ← j-1	c_4	n-1
5. **while** i >0 and A[i]>*key*	c_5	$\sum_{j=2}^{n} t_j$
6. **do** A[i+1] ← A[i]	c_6	$\sum_{j=2}^{n}(t_j-1)$
7. i ← i-1	c_7	$\sum_{j=2}^{n}(t_j-1)$
8. A[i+1] ← *key*	c_8	$n-1$

(t_j udáva, koľkokrát sa vykoná test cyklu while v riadku 5 pre danú hodnotu j)

Celková cena $T(n)$ = suma *cena* × *počet opakovaní* pre každý riadok

$$=c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

analýza algoritmu Insertion Sort

- cena v najlepšom prípade: prvky sú už usporiadané
  - t_j=1, a riadky 6 a 7 sa vykonajú 0 krát
  - $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$
    $=(c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) = cn + c'$
- cena v najhoršom prípade: prvky sú už usporiadané, ale v opačnom poradí
  - t_j=j,
  - so $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = n(n+1)/2-1$, a $\sum_{j=2}^{n}(t_j-1) = \sum_{j=2}^{n}(j-1) = n(n-1)/2$, a
  - $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n(n+1)/2 -1) + + c_6(n(n-1)/2 -1) + c_7(n(n-1)/2) + c_8(n-1) = ((c_5 + c_6 + c_7)/2)n_2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n - (c_2 + c_4 + c_5 + c_8) = an^2+bn+c$
- cena v priemernom prípade: čísla sú náhodné
  - v priemere, $t_j = j/2$. $T(n)$ bude aj v priemernom prípade stále rádu n^2, rovnako ako v najhoršom prípade.

časová zložitosť algoritmu Insertion Sort

- časová zložitosť:
  - v najlepšom prípade: T(n) = Θ(n),
  - v najhoršom prípade: T(n) = Θ(n^2),
  - v priemernom prípade: T(n) = Θ(n^2)
- je to rýchly algoritmus?
  - pre malé n celkom prijateľný
  - pre veľké n vonkoncom nie.

Usporadúvanie výmenou - bublinkové (bubble)

- bubble sort usporadúva prvky priamou výmenou
- je implementačne jednoduchý, ale neefektívny
- pri usporadúvaní porovnáva dva susedné prvky a ak nie sú v správnom poradí, vymenia sa
- procedúra sa opakuje, až kým nie sú potrebné žiadne výmeny

31

Usporadúvanie výmenou - bublinkové (bubble)

- Príklad:
  - Usporiadajte bublinkovou výmenou pole **5 1 4 2 8**
    - 1. fáza
      - » **5 1** 4 2 8 –> **1 5** 4 2 8
      - » 1 **5 4** 2 8 –> 1 **4 5** 2 8
      - » 1 4 **5 2** 8 –> 1 4 **2 5** 8
      - » 1 4 2 **5 8** –> 1 4 2 **5 8**
    - 2. fáza
      - » **1 4** 2 5 8 –> **1 4** 2 5 8
      - » 1 **4 2** 5 8 –> 1 **2 4** 5 8
      - » 1 2 **4 5** 8 –> 1 2 **4 5** 8
      - » 1 2 4 **5 8** –> 1 2 4 **5 8**

32

Usporadúvanie výmenou - bublinkové (bubble)

- Príklad:
  - na začiatku sme mali pole **5 1 4 2 8**
    - 3. fáza
      - » **1 2** 4 5 8 –> **1 2** 4 5 8
      - » 1 **2 4** 5 8 –> 1 **2 4** 5 8
      - » 1 2 **4 5** 8 –> 1 2 **4 5** 8
      - » 1 2 4 **5 8** –> 1 2 4 **5 8**
  - na konci máme usporiadané pole **1 2 4 5 8**

33

Bublinkové usporadúvanie

```
Procedure BubbleSort(var A: pole)
Var i, j, t : integer;
Begin
  for i := Dlzka(A) downto 1 do
     for j := 1 to Dlzka(A)-1 do
          if A[j] > A[j+1] then
          begin
               T := A[j];
               A[j] := A[j+1];
               A[j+1] := T;
          end;
end;
```

34

Bublinkové usporadúvanie

```
void bubbleSort (Array A) {
   n = dĺžka(A);
   boolean isSorted = false;
   while(!isSorted) {
        isSorted = true;
        for(i = 0; i<n; i++) {
           if(A[i] > A[i+1]) {
                       int T = A[i];
                       A[i] = A[i+1];
                       A[i+1] = T;
                       isSorted = false;
                 }
        }
}
```

35

časová zložitosť bublinkového usporadúvania

- 1 prechod = presun najväčšieho prvku na koniec
- i-tý prechod: n-i+1 operácií
- čas:
  $(n – 1) + (n – 2) + … + 1 = (n – 1)\, n / 2 = O(n^2)$
- ktorý prípad je najlepší?
- ktorý prípad je najhorší?

36

Usporadúvanie výberom

- algoritmus realizuje usporadúvanie priamym výberom
- vychádza z prepokladu, že najmenší prvok môžeme zaradiť priamo na začiatok vstupného poľa, najmenší prvok zo zvyšku poľa zase na jeho začiatok atď.
- podľa toho, či usporadúva prvky vzostupne/zostupne, sa môže označovať ako MinSort/MaxSort

Usporadúvanie výberom

- Príklad:
  - usporiadajte výberom pole **6 4 5 2 3 1 7**
    - 1. krok — | 6 4 5 2 3 **1** 7 –> **1** | 6 4 5 2 3 7
    - 2. krok — 1 | 6 4 5 **2** 3 7 –> 1 **2** | 6 4 5 3 7
    - 3. krok — 1 2 | 6 4 5 **3** 7 –> 1 2 **3** | 6 4 5 7
    - 4. krok — 1 2 3 | 6 **4** 5 7 –> 1 2 3 **4** | 6 5 7
    - 5. krok — 1 2 3 4 | 6 **5** 7 –> 1 2 3 4 **5** | 6 7
    - 6. krok — 1 2 3 4 5 | **6** 7 –> 1 2 3 4 5 **6** | 7
    - 7. krok — 1 2 3 4 5 6 | **7** –> 1 2 3 4 5 6 **7** |
- Miera usporiadanosti vstupného poľa nemá vplyv na časovú zložitosť – vždy sa vykoná maximálny počet krokov.

Usporadúvanie výberom

```
procedure SelectionSort(var A: pole)
var i, j, t : integer;
begin
  for i:= 1 to Dlzka(A) – 1 do
     for j:= Dlzka(A) downto i+1 do
         if A[i] > A[j] then
         begin
              T := A[i];
              A[i] := A[j];
              A[j] := T;
         end;
end;
```


Donald Shell

1.3.1924 –

1959 PhD Uni of Cincinnati

Shell, D.L. (1959). "A high-speed sorting procedure". Communications of the ACM 2 (7): 30–32.

Shellovo usporadúvanie

- algoritmus, ktorý realizuje usporadúvanie priamym vkladaním so zmenšovaním prírastku
- je to vlastne zlepšenie usporadúvania vkladaním a bublinkového
- táto metóda je jedna z najrýchlejších pre usporiadanie menších postupností (menej ako 1000 prvkov)

Shellovo usporadúvanie

- Príklad, prírastky n = n/2, atď (pôvodný návrh Shella):
  - Usporiadajte pole **6 4 5 2 8 3 1 7** pomocou algoritmu shell sort, n = 8/2 = 4
    - 1. krok, krokovanie 4, vyznačené čísla sa usporiadajú vkladaním
      - » **6** 4 5 2 **8** 3 1 7 –> **6** 4 5 2 **8** 3 1 7
      - » 6 **4** 5 2 8 **3** 1 7 –> 6 **3** 5 2 8 **4** 1 7
      - » 6 3 **5** 2 8 4 **1** 7 –> 6 3 **1** 2 8 4 **5** 7
      - » 6 3 1 **2** 8 4 5 **7** –> 6 3 1 **2** 8 4 5 **7**
    - 2. krok, krokovanie 2
      - » **6** 3 **1** 2 **8** 4 **5** 7 –> **1** 3 **5** 2 **6** 4 **8** 7
      - » 1 **3** 5 **2** 6 **4** 8 **7** –> 1 **2** 5 **3** 6 **4** 8 **7**
    - 3. krok, krokovanie 1
      - » **1 2 5 3 6 4 8 7** –> 1 2 3 4 5 6 7 8

Shellovo usporadúvanie

```
procedure ShellSort(var f: pole)
var i, j, h, v, N : integer;
begin
   N := Dlzka(f);
   h := 1;
   repeat // priprav prirastky podla Knutha
        h := (3 * h) + 1;
   until h > N;

   repeat
        h := (h div 3);
        for i:= (h + 1) to N do begin
            v := f[i];
            j := i;
            while ( (j > h) and ( f[j-h] > v ) ) do begin
                 f[j] := f[j - h];
                 dec(j, h);
            end;
            f[j] := v;
        end;
   until h = 1;
end
```


Shellovo usporadúvanie

- používa postupnosť prírastkov $h_1, h_2, \ldots, h_t$
- môže byť ľubovoľná postupnosť, len musí byť $h_1 = 1$ a $h_1 < h_2 < \ldots < h_t$
- rôzne voľby postupnosti prírastkov vedú k rôzne efektívnym verziám algoritmu.

Shellovo usporadúvanie

urči postupnosť prírastkov $h_1, h_2, \ldots, h_t$

urob t prechodov cez postupnosť prvkov

prechod 1: h_t – usporiadaná postupnosť, t.j.
pre $\forall$i a[i] ≤ a[i + h_t]

prechod 2: h_{t-1} – usporiadaná postupnosť, t.j.
pre $\forall$i a[i] ≤ a[i + h_{t-1}]

...

prechod t: h_1 – usporiadaná postupnosť, t.j.
pre $\forall$i a[i] ≤ a[i + h_1]

- v každom prechode dosiahni h_k – usporiadanie pomocou usporadúvania vkladaním

Shellovo usporadúvanie

- po každom prechode pri nejakom prírastku h_k, pre všetky i máme a[i] ≤ a [i + h_k] t.j. všetky prvky umiestnené od seba o h_k miest sú usporiadané.
- posledný prechod sa robí s prírastkom $h_1 = 1$
  $\Rightarrow$ pre $\forall$i a[i] ≤ a[i + 1]
  $\Rightarrow$ postupnosť a[ ] je usporiadaná.

Shellovo usporadúvanie: prírastky

- Shell: $\lfloor n/2 \rfloor, \lfloor n/2^2 \rfloor, \lfloor n/2^3 \rfloor, \ldots$, 1 alebo (ešte horšie) postupnosť prírastkov mocniny 2: $O(n^2)$
- Knuth: 1, 4, 13, 40, 121, 364,1093, 3280, 9841,... t.j. $h_1 = 1$, $h_{i+1} = 3 * h_i + 1$
- Knuth: blízko $O(n \log^2 n)$ a $O(n^{1.25})$
- Hibbard: 1, 3, 7,...2^{k-1}: $O(n^{3/2})$
- Sedgewick: 1, 8, 23, 77, 281, 1073, 4193, 16577..., ($4^{i+1} + 3 \cdot 2^i + 1$) pre i > 0, má byť lepšia než Knuth
- Pratt: $\log^2 n$ prírastkov $2^i 3^j < \lfloor n/2 \rfloor$

Shellovo usporadúvanie

- najlepší prípad: postupnosť je už usporiadaná – bude treba menej porovnaní
- najhorší prípad (pre postupnosť prírastkov podľa Pratta): $O(n \log^2 n)$
- priemerný prípad (pre postupnosť prírastkov podľa Pratta): $\Theta(n \log^2 n)$

jedna varianta inšpirovaná Shellovým usporadúvaním: Shaker-sort*

h-utrasenie postupnosti a[1], ..., a[n]:
porovnávanie a prípadná výmena týchto dvojíc prvkov v uvedenom poradí:
(1; 1 + h), (2; 2 + h),..., (N - h - 1; N - 1), (N - h; N),
(N - h - 1; N - 1), (N - h - 2; N - 2), ..., (2; 2 + h); (1; 1 + h)
všimnime si, že medzi prvkami každej dvojice je rovnaká vzdialenosť h.

usporadúvanie utrasením:
urči postupnosť prírastkov h_1, h_2, …, h_t
urob t prechodov cez postupnosť prvkov
prechod 1: h_1 – utrasenie
prechod 2: h_2 – utrasenie
...
prechod t: h_t – utrasenie.
usporiadaj vkladaním // alebo opakuj 1-utrasenie dovtedy, kým nie je postupnosť usporiadaná

*Incerpi, J. and R. Sedgewick, Practical Variations on Shellsort, Inform. Process. Lett. 26 (1987), 37-43.

49