Algorytmy i struktury danych I
Sortowanie
IO, WIMiIP

Danuta Szeliga

AGH Kraków

2021/2022

Spis treści I

Spis treści II

Sortowanie

Wstęp

Sortowanie

Sortowanie

- Jeden z podstawowych problemów informatycznych
- Polega na uporządkowaniu zbioru danych względem pewnych cech charakteryzujących każdy elementu tego zbioru, biorąc pod uwagę wartość klucza elementu

Algorytmy sortowania są stosowane dla uporządkowania danych, umożliwienia stosowania wydajniejszych algorytmów (np. wyszukiwania), prezentacji danych w czytelny sposób

Klasyfikacja metod sortowania I

- Według rodzaju sortowanej struktury:
  - sortowanie tablic liniowych
  - sortowanie list
- Według miejsca sortowania (rodzaju pamięci)
  - zewnętrzne
  - wewnętrzne
- Według zużycia pamięci
  - intensywne (in situ) – nie potrzebują (w zasadzie) dodatkowej pamięci
  - ekstensywne – potrzebują pamięci pomocniczej
- Według stabilności
  - czy dokonuje zbędnych przestawień, czy utrzymują kolejność występowania dla elementów o tym samym kluczu
- Według ilości etapów algorytmu sortującego:
  - jednoetapowe (bezpośrednie) – w zasadzie nie potrzebują dodatkowej pamięci
  - dwuetapowe (pośrednie)

Klasyfikacja metod sortowania II

- etap logiczny – nie przestawia rekordów, ale zdobywa informacje nt. ustawienia rekordów i zapisuje je w pewien sposób
- etap fizyczny (nie zawsze potrzebny)

- Według efektywności
  - proste (do krótkich plików) - $\mathcal{O}(n^2)$
  - szybkie - $\mathcal{O}(n \log n)$
- Czy używają wyłącznie relacji porównania $<, >, \leq, \geq$?
  - używa jedynie relacji porównania (porządek liniowy) $\rightarrow$ złożoność najlepszego przypadku $\Omega(n \log n)$
  - używa także innych własności sortowanego ciągu $\rightarrow$ złożoność najlepszego przypadku $\Omega(n)$

Klasyfikacja metod sortowania III

- Dlaczego nie może być lepszego algorytmu bazyjącego na porównaniach niż algorytm o złożoności $\Omega(n \log n)$?
  Ciąg n różnych elementów — tylko jedna z $n!$ permutacji odpowiada posortowanemu ciągowi
  Jeżeli algorytm posortuje ciąg po $f(n)$ krokach, to nie może rozróżnić więcej niż $2^{f(n)}$ przypadków (porównanie daje w wyniku jedną z dwóch odpowiedzi). A zatem:

$$2^{f(n)} \geq n! \Rightarrow f(n) \geq \log(n!) = \Omega(n \log n)$$

  Ostatnie oszacowanie na podstawie wzoru Stirlinga:

$$n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

Sortowanie

Metody proste

Sortowanie bąbelkowe

Bubble sort

- Algorytm kontroli monotoniczności: sprawdzanie monotoniczności kolejnych sąsiednich $n-1$ par
- Po jednym przebiegu wyprowadza rekord z najwyższym (najniższym) kluczem na właściwą pozycję
- W kolejnych przebiegach brana pod uwagę tylko nieposortowana część ciągu

```
4|2|5|7|1  ->  4|2|5|1|7  ->  4|2|1|5|7  ->  4|1|2|5|7  ->  1|4|2|5|7
      ^_^          ^_^          ^_^          ^_^
1|4|2|5|7  ->  1|4|2|5|7  ->  1|4|2|5|7  ->  1|2|4|5|7
      ^_^          ^_^          ^_^
1|2|4|5|7  ->  1|2|4|5|7  ->  1|2|4|5|7
      ^_^          ^_^
1|2|4|5|7  ->  1|2|4|5|7
      ^_^
```

Sortowanie bąbelkowe

Modyfikacje

- Ciągła kontrola monotoniczności → zmienna logiczna
  - Przed każdym przebiegiem `false`, po przestawieniu `true`
  - Jeśli `true`, to sortuj dalej, jeśli nie — zakończ
- Wariant wahadłowy (cocktail-sort) – ze "zderzakami"
  - Przebiegi na przemian: raz od lewej, raz od prawej
  - W zderzakach informacja, gdzie nastąpiło przestawienie — przeglądanie w przeciwnym kierunku zaczynamy od tego miejsca i tak aż do dotknięcia się zderzaków

Procedura pomocnicza:

```
swap (<type> x, <type> y){
   <type> tmp=x;
   x=y;
   y=tmp;
}
```

Sortowanie bąbelkowe

Ciągła kontrola monotoniczności

```
bubble_sort(<type> arr[n]){
   int i = -1, j;
   bool if_swap;
   do {
      if_swap = false;
      for(++i, j = n-1; j > i; --j)
         if(arr[j] < arr[j - 1]) {
            swap(arr[j], arr[j-1]); // zamiana
            if_swap = true;
         }
   }while(if_swap);
}
```

Sortowanie bąbelkowe

Wariant wahadłowy

```
coctail_sort (int arr[], int n) {
  int lz = 0; int pz = n - 1;
  while (lz < pz) {
    int pom = lz;
    for (int i = lz; i < pz; i++)
      if (arr[i] > arr[i + 1]) {
        swap(arr[i], arr[i + 1]);
        pom = i;
      }
    pz = pom;
    for (int i = pz; i > lz; i--)
      if (arr[i] < arr[i - 1]) {
        swap(arr[i], arr[i - 1]);
        pom = i;
      }
    lz = pom;
  }
}
```

Sortowanie bąbelkowe

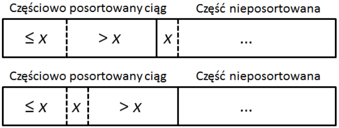
Analiza złożoności

- Złożoność pamięciowa: $\mathcal{O}(1)$
- Złożoność obliczeniowa:
  - najlepszego przypadku: $\mathcal{O}(n)$ gdy tablica jest wstępnie posortowana
  - najgorszego przypadku: $\mathcal{O}(n^2)$
    ($n^2/2$ porównań, $3/2n^2$ podstawień)
    gdy tablica jest odwrotnie posortowana

Sortowanie przez wstawianie

insert sort (Steinhaus,1958)

- Pierwszy krok: podział na część posortowaną i nieposortowaną
  - pusta i cała tablica lub
  - pierwszy rekord i reszta lub
  - pewna naturalnie posortowana część i reszta
- Kolejne kroki $\rightarrow$ pętla
  - pobranie rekordu (dowolnego) i wstawienie go do posortowanej części ciągu



- Sposób przeglądania części posortowanej
  - sekwencyjne przeglądanie części posortowanej od prawej strony (stabilność, natychmiastowe przestawienie rekordu)
  - przeszukiwanie binarne

Sortowanie przez wstawianie

Zalety:

- użyteczny dla tablic do 10 elementów
- wydajny dla danych wstępnie posortowanych: $\mathcal{O}(n + d)$, gdzie d to liczba zamian
- stabilny

(1) 4 | 2 | 5 | 7 | 1 | 8 | 3 | 2 | 7 | 9   4 | 4 | 5 | 7 | 1 | 8 | 3 | 2 | 7 | 9 ⟶ 2 | 4 | 5 | 7 | 1 | 8 | 3 | 2 | 7 | 9

(2,3) 2 | 4 | 5 | 7 | 1 | 8 | 3 | 2 | 7 | 9 ⟶ 2 | 4 | 5 | 7 | 1 | 8 | 3 | 2 | 7 | 9

(4) 2 | 4 | 5 | 7 | 1 | 8 | 3 | 2 | 7 | 9 ⟶ 2 | 4 | 5 | 7 | 7 | 8 | 3 | 2 | 7 | 9 ⟶ 2 | 4 | 5 | 5 | 7 | 8 | 3 | 2 | 7 | 9

⟶ 2 | 4 | 4 | 5 | 7 | 8 | 3 | 2 | 7 | 9 ⟶ 2 | 2 | 4 | 5 | 7 | 8 | 3 | 2 | 7 | 9 ⟶ 1 | 2 | 4 | 5 | 7 | 8 | 3 | 2 | 7 | 9

Sortowanie przez wstawianie

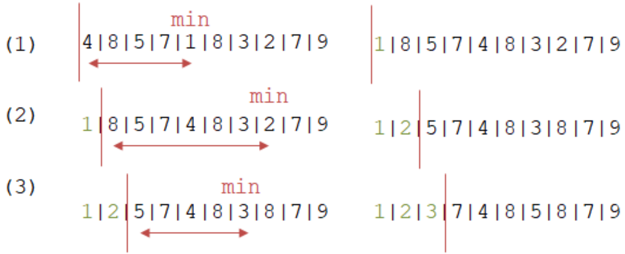
```
insert_sort(<type> t[N]){
   for(i = 1,j; i < N; ++i){
      x=t[i];
      for(j=i-1; j>=0 && t[j]>x; --j)
         t[j+1]=t[j];
      t[j+1]=x;
   }
}
```

- Złożoność pamięciowa: $\mathcal{O}(1)$
- Złożoność obliczeniowa:
  - przypisania: średnia $\mathcal{O}(n+d)$, najgorszego przypadku $\mathcal{O}(n^2)$
  - porównania: średnia $\mathcal{O}(n+d)$, najgorszego przypadku $\mathcal{O}(n^2)$

Sortowanie przez wybieranie (select sort)

- Pierwszy krok: podział na część posortowaną i nieposortowaną
- Kolejne kroki: $\rightarrow$ pętla:
  - wyszukujemy w części nieposortowanej element o najmniejszym kluczu (jeśli sortujemy w porządku niemalejącym)
  - zamieniamy ten element z lewym elementem w części nieposortowanej (*uwaga*: narusza to stabilność algorytmu)
- Stosowane do ciągów o długości do 20 elementów
- Algorytm niestabilny. Stabilność można uzyskać poprzez zamianę w warunku nierówności ostrej na nieostrą

Sortowanie przez selekcję



Sortowanie przez selekcję (wybieranie)

Select sort

```
select_sort(<type> t[N]){
   for(int i=0; i<N-1; i++){
      min=t[i];
      imin=i;
         for(int j=i+1; j<N; j++)
            if(t[j]<min){
               min=t[j];
               imin=j;
            }
        if(i!=imin) swap(t[i],t[imin]);
    }
}
```

- Liczba operacji porównania: zawsze $\Theta(n^2)$
- Liczba operacji podstawienia: zawsze $3(n-1)$

Sortownie grzebieniowe (comb sort)

- Uogólenie metody sortowania bąbelkowego
- za rozpiętość r przyjmij długość tablicy n, podziel rozpiętość przez $1.24733095\cdots$ (w praktyce 1.3), odrzuć część ułamkową
- badaj kolejno wszystkie pary obiektów odległych rozpiętość r (jeśli są ułożone niemonotonicznie - zamień miejscami)
- wykonuj powyższe działania w pętli dzieląc rozpiętość przez 1.3 do czasu, gdy rozpiętość osiągnie wartość 1
- Gdy rozpiętość spadnie do 1, metoda zachowuje się tak jak sortowanie bąbelkowe
- Tylko wtedy możemy określić, czy dane są już posortowane czy nie
- W tym celu można użyć zmiennej typu logicznego, która jest ustawiana po zamianie elementów tablicy miejscami (tak, jak przy ciągłej kontroli monotoniczności)

Sortowanie grzebieniowe

- Combsort11
  - najkorzystniejsza wartość rozpiętości to 11
  - jeżeli obliczona rozpiętość jest równa 9 lub 10 - zamieniamy ją na 11
  - zysk ok. 20%,
  - potrzebna specjalna uwaga dla tablic wstępnie posortowanych
  - w praktyce metoda nieznacznie wolniejsza od quick-sort (!)
- Złożoność obliczeniowa: prawdopodobnie $\mathcal{O}(n \log n)$
- Metoda niestabilna

Sortowanie grzebieniowe

n=10

-> r=7

4|8|5|7|1|8|3|2|7|9   2|8|5|7|1|8|3|4|7|9   2|7|5|7|1|8|3|4|8|9   2|7|5|7|1|8|3|4|8|9

-> r=5

2|7|5|7|1|8|3|4|8|9   2|7|5|7|1|8|3|4|8|9   2|3|5|7|1|8|7|4|8|9   2|3|4|7|1|8|7|5|8|9

-> r=3

2|3|4|7|1|8|7|5|8|9   2|3|4|7|1|8|7|5|8|9   2|1|4|7|3|8|7|5|8|9

Sortowanie grzebieniowe

Comb sort

```
int newGap(int gap){
   gap = (gap*10)/13;
   if(gap==9 || gap==10) gap=11;
   if(gap<1) gap=1;
   return gap;
}

comb_sort(<type> t[N]) {
   gap=N;
   while(true){
      gap = newGap(gap);
      if(gap==1) break;
      for(i=0; i<N-gap; ++i) {
         j = i+gap;
         if(t[i]>t[j])
            swap(t[i], t[j]);
      }
    }
    bubble_sort(t);
}
```

Sortowanie Shella

Shell sort, 1959

- Uogólnienie sortowania przez wstawianie
  Obserwacje:
  - sortowanie przez wstawianie jest efektywne dla tablic "prawie" posortowanych
  - sortowanie przez wstawianie jest nieefektywne ponieważ przesuwa elementy o jedną pozycję
- Idea:
  - sortowany zbiór dzielimy na podzbiory, których elementy są odległe od siebie w sortowanym zbiorze o pewien odstęp h
  - każdy z tych podzbiorów sortujemy algorytmem przez wstawianie, następnie odstęp zmniejszamy - powoduje to powstanie nowych podzbiorów (ich liczba będzie w kolejnych krokach malała, a liczność pojedynczego podzbioru wzrastała)
  - sortowanie powtarzamy i znów zmniejszamy odstęp aż osiągnie on wartość 1 i wtedy algorytm działa jak insert-sort

Sortowanie Shella

n=10

-> r=6

4 | 8 | 5 | 7 | 1 | 8 | 3 | 2 | 7 | 9   3 | 8 | 5 | 7 | 1 | 8 | 4 | 2 | 7 | 9   3 | 2 | 5 | 7 | 1 | 8 | 4 | 8 | 7 | 9

-> r=4

3 | 2 | 5 | 7 | 1 | 8 | 4 | 8 | 7 | 9   1 | 2 | 5 | 7 | 3 | 8 | 4 | 8 | 7 | 9   1 | 2 | 4 | 7 | 3 | 8 | 5 | 8 | 7 | 9

1 | 2 | 4 | 7 | 3 | 8 | 5 | 8 | 7 | 9

-> r=3

1 | 2 | 4 | 7 | 3 | 8 | 5 | 8 | 7 | 9   1 | 2 | 4 | 5 | 3 | 8 | 7 | 8 | 7 | 9

-> r=2

1 | 2 | 4 | 5 | 3 | 8 | 7 | 8 | 7 | 9   1 | 2 | 3 | 5 | 4 | 8 | 7 | 8 | 7 | 9

-> r=1

1 | 2 | 3 | 5 | 4 | 8 | 7 | 8 | 7 | 9   1 | 2 | 3 | 4 | 5 | 8 | 7 | 8 | 7 | 9   1 | 2 | 3 | 4 | 5 | 7 | 8 | 8 | 7 | 9   1 | 2 | 3 | 4 | 5 | 7 | 7 | 8 | 8 | 9

Sortowanie Shella

Generacja ciągu odstępów h_i

- Wg Knuth'a $\rightarrow \mathcal{O}(n^{1.15})$ (?)
  - $h_1 = 1$, następnie $h_k = 3h_{k-1} + 1$. Kolejne wyrazy są generowane do momentu, aż $h_k \geq n$, wtedy $h = h_k/9$
  - odstęp w iteracji i jest obliczany jako $h_i = h_{i-1}/3$
  - 1, 4, 13, 40, 121, 364, ...
- wg Hibbarda $\rightarrow \mathcal{O}(n^{1.5}) = \mathcal{O}(n\sqrt{n})$
  - $h_i = h^i - 1$, czyli: 1,3,7, 15, 31, ...
- wg Pratta $\rightarrow \mathcal{O}(n \log^2 n)$
  - $h_i = 2^p 3^q$, czyli 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81, ...
- wg Segedwicka $\rightarrow \mathcal{O}(n^{4/3})$
  - $h_i = 8(4^i - 2^i) + 1$ lub
  - $h_i = 4^{i+1} + 3 \cdot 2^i + 1$, czyli: 1, 8, 23, 77, 281, 1073, ...
- wg Incerpi-Segedwicka
  - $h_i = 5 \cdot h^i - 7$ dla $i = 2, 3$ oraz $h_i = 5 \cdot h^i - 45$ dla $i = 4, \ldots, 9$ $\rightarrow \mathcal{O}(n^{1+1/i})$
  - $h_i = \alpha^p (1+\alpha)^q \rightarrow \mathcal{O}(1 + \varepsilon/\log n)$
- wg Marcina Ciury (najlepszy znany ciąg)
  - 1, 4, 10, 23, 57, 132, 301, 701, kolejne $\times 2.3$

Sortowanie Shella

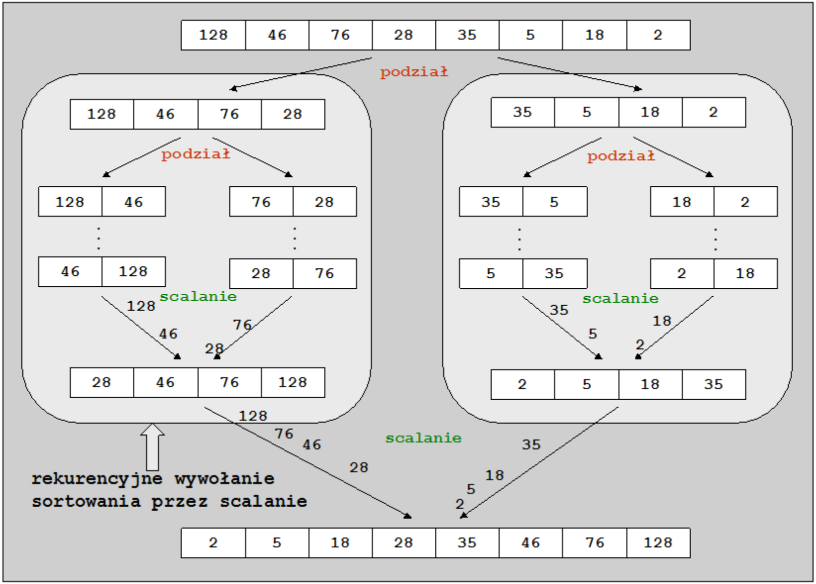
Shell sort

```
//wg Knutha
Shell_sort(<type> t[N]) {
   for(h = 1; h < N; h = 3 * h + 1);
   h /= 9;
   if(!h) h++;
   for(; h > 0; h /= 3) {
       for(j = h; j<N; ++j){
          x = t[j];
          for(i = j-h; i >= 0 && x<t[i]; i-=h)
             t[i+h] = t[i];
          t[i+h] = x;
       }
   }
}
```

Sortowanie

Algorytmy sortowania szybkiego

Sortowanie przez scalanie



Sortowanie przez scalanie

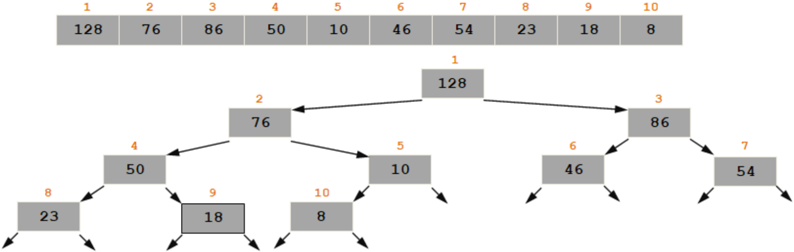
Merge sort

```
Merge_sort(<type> t[N], int l, int r){
   g=l+(r-l)/2;
   if(l<g) Merge_sort(t,l,g);
   if(g+1<r) Merge_sort(t,g+1,r);
   Merge(t,l,g,r);
}

Merge(<type> t[N],int l,int g,int r){
   h=g-l+1; //rozmiar tablicy u
   m=r-g;   //rozmiar tablicy v
   u=new_vector <type>[h]; //tablica pomocnicza
   v=new_vector <type>[m]; //tablica pomocnicza
   copy(u[0..h-1], t[l..l+(h-1)]; //kopiuj do tablicy u z tablicy t
   copy(v[0..m-1], t[g+1..g+m)];  //kopiuj do tablicy v z tablicy t
   i=0;j=0;
   for(k=0;i<h && j<m; k++){ //scalaj wyniki z tablic u i v to t
      if(u[i]<v[j]){
         t[l+k]=u[i]; i++;
      }
      else{
         t[l+k]=v[j]; j++;
      }
   }
   //skopiuj pozosta a czesc z tablicy u/v do t
   if(i<h) copy(t[l+k..r)],u[i..(h-1)]);
   else    copy(t[l+k..r)],v[j..(m-1)]);
}
```

Sortowanie kopcowe

Heap sort



Sortowanie kopcowe

Heap sort

Budowa kopca z tablicy

```
shiftdown(<type>*t, int i, int n) {
   for(int l=2*i; l <= n; i=l, l=2*i) {
      if(l+1 <= n && t[l+1] > t[l])  ++l;
      if(t[i] >= t[l]) return;
      swap(t[i], t[l]);
   }
}

buildheapBU(<type>*t, int n) {
   for(long i=n/2; i; --i)
      shiftdown(t, i, n);
}

shiftup(<type>*t, int i) {
   for(int f; i > 1; i=f) {
      f = i/2; // father
      if(t[f] >= t[i]) return;
      swap(t[f], t[i]);
      }
}

buildheapBD(<type>*t, int n) {
   for(i=2; i <= n; ++i)
      shiftup(t, i);
}
```

Sortowanie kopcowe

Heap sort

Sortowanie

```
heap_sort(<type>*t, int n) {
   --t; // teraz indeks 1 pokazuje na element 0 !
   buildheapXX(t, n); //shiftdown/shifup
   while(n > 1) {
      // przenosimy max. elem. na koniec kopca
      swap(t[1], t[n]);
      // przywracamy w asno    kopca dla mniejszego kopca
     shiftdown(t, 1, --n);
   }
}
```

Sortowanie szybkie

Quick sort

```
Quick_sort(<type> t[N], int l, int r){
   if(l>=r) return;
   p=partition_XX(t,l,r); //Hoare's, Lomuto
   Quick_sort(t,l,p-1);
   Quick_sort(t,p+1,r);
}

int partition_Lomuto(<type> t[N], int l, int r){
   x=t[r]; //podzia  wzgl dem prawego elementu
   i=l-1;
   for(j=l;j<r;j++)
      if(t[j]<x) { i++; swap(t[i],t[j]);}
   swap(t[i+1],t[r]);
   return i+1;
}

int partition_Hoare(<type> t[N], int l, int r){
   x=t[l]; //podzia  wzgl dem lewego elementu
   i=l, j=r+1;
   while(true){
      do{ i++;} while(i<=r && t[i]<x);
      do{ j--;} while(t[j]>x);
      if(i>j) break;
      swap(t[i],t[j]);
   }
   swap(t[j],t[l]);
   return j;
```

Algorytmy hybrydowe

Sortowanie hybrydowe

- Cel: modyfikacja metody Quick Sort
- Spostrzeżenia:
  - Bardzo dużo rekurencyjnych wywołań algorytmu Quick Sort wykonywanych jest dla małych tablic
  - W przypadku tablic o niewielkich rozmiarach instrukcje wykonywane w funkcji Partition są dość kosztowne w stosunku do rozmiaru samej tablicy
  - Samo wywołanie rekurencyjne jest czasochłonne i zajmuje miejsce na stosie (np. dla 5-elementowej tablicy mogą być potrzebne nawet aż 3 wywołania)
- Idea: wywoływana jest procedura Quick Sort, aż to otrzymania podzbiorów o małej liczebności, a następnie te małe zbiory o rozłącznych wartościach są sortowane jednym z prostych algorytmów sortowania (np. Insert Sort), które chociaż mają złożoność obliczeniową rzędu $O(n^2)$, to dla zbiorów o niewielkim rozmiarze działają relatywnie szybko
- Tego rodzaju technika nosi nazwę *metody odcinania małych podzbiorów*

Sortowanie introspektywne

- Cel: modyfikacja algorytmu Quick Sort tak, aby zapewnić złożoność $O(n \log n)$, czyli eliminacja zdegenerowanych podziałów funkcji Partition
- Idea: badanie głębokości drzewa wywołań rekurecyjnych
  - na początku algorytmu obliczana jest wartość $M = 2 \log_2 n$, która określa maksymalną, dozwoloną głębokość wywołań rekurencyjnych
  - w przypadku, gdy $M > 0$, uruchamiana jest metoda Quick Sort lub Quick Sort z odcinaniem małych podzbiorów, która przyjmuje dodatkowo parametr M. Każde rekurencyjne wywołanie kolejnej procedury Quick Sort jest uruchamiane z parametrem $M - 1$.
  - w przypadku, gdy $M = 0$, uruchamiana jest procedura Heap Sort (sortowanie przez kopcowanie).

Sortowanie w czasie liniowym

- Wszystkie do tej pory przedstawione algorytmy sortowania działały tylko w oparciu o porównania elementów $\Leftrightarrow$ porządek elementów w tablicy jest oparty jedynie na relacji porównania
- Algorytmy te w przypadku pesymistycznym musiały zawsze wykonać przynajmniej $\Omega(n \log n)$ porównań

Przedstawione dalej algorytmy działają w czasie liniowym $\Rightarrow$ do sortowania wykorzystują inne operacje niż porównanie

Sortowanie pozycyjne (radix-sort) I

- Kluczami są liczby naturalne lub łańcuchy znaków
- Stosowany jest zapis pozycyjny

Dodatkowa informacja o kluczach

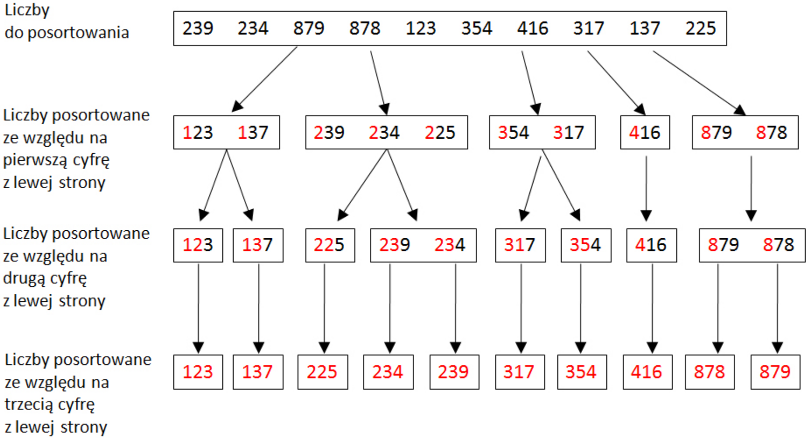
- Wszystkie klucze składają się z takiej samej liczby cyfr
- Znaczenie ma pozycja cyfry ⇒ najmniejsze klucze mają zawsze najmniejszą skrajnie lewą cyfrę , itd.

Sposób wykorzystania

- Najpierw sortujemy ze względu na pierwszą cyfrę klucza ⇒ dzielimy klucze na grupy
- Potem (rekurencyjnie) sortujemy każdą grupę ze względu na kolejną cyfrę znaczącą

⇒ MSD-radix-sort (Most Significant Digit radix-sort)

Sortowanie pozycyjne (MSD-radix-sort) II



Sortowanie pozycyjne

Trudności

- Sortowanie pozycyjne typu MSD było używane na początku w maszynach sortujących karty dziurkowane

Trudności

- Klucze muszą składać się z tej samej liczby cyfr/znaków
- Algorytm nie zachowuje oryginalnego porządku dla kluczy o tej samej wartości
- W pierwszym kroku algorytmu, dla kluczy o d cyfrach/znakach, klucze dzielone są pomiędzy d różnych zbiorów, w następnym kroku algorytm jest wykonywany dla pierwszego zbioru, pozostałe $d - 1$ zbiorów musi być w jakiś sposób zapamiętane $\Rightarrow$ zajmuje to pamięć i komplikuje sam algorytm

Sortowanie pozycyjne (LSD-radix-sort)

Jak rozwiązać trudności?

- Zaczynamy sortowanie od najmniej znaczącej cyfry

$\Rightarrow$ LSD-radix-sort (Least Significant Digit radix-sort)

- Teraz zawsze potrzeba jedynie d zbiorów w każdym kroku algorytmu
- Musimy jedynie spełnić warunek: sortowanie względem danej cyfry musi być stabilne, tzn. że klucze posortowane w kolejnym kroku algorytmu, które znajdują się w nowym zbiorze, jeśli w poprzednim kroku znajdowały się w zbiorze i, to dalej muszą znajdować się przed kluczami, które znajdowały się w zbiorze $i + 1$ w poprzednim kroku
- Jeśli długość klucza jest równa k, to algorytm wykona k iteracji

Sortowanie pozycyjne (LSD-radix-sort) II

Liczby do posortowania

239	234	879	878	123	354	416	317	137	225

Liczby posortowane ze względu na pierwszą cyfrę z prawej strony

0	1	2	3	4	5	6	7	8	9
			123	234 354	225	416	317 137	878	239 879

Liczby posortowane ze względu na drugą cyfrę z prawej strony

0	1	2	3	4	5	6	7	8	9
	416 317	123 225	234 137 239		354		878 879		

Liczby posortowane ze względu na trzecią cyfrę z prawej strony

0	1	2	3	4	5	6	7	8	9
	123 137	225 234 239	317 354	416				878 879	

Sortowanie pozycyjne LSD-radix-sort (Seward, 1954)

```
LsdRadixSort(T arr[1..n], k) {
   for i := 1 to k do
      StableSort(arr[1..n], i); // sortowanie stabilne
                                // wzgl dem pozycji i
}
```

Sortowanie pozycyjne LSD-radix-sort

- Ilość elementów każdego zbioru kluczy zmienia się z iteracji na iterację $\Rightarrow$ dobrym rozwiązaniem jest zastosowanie list
- Mamy tyle list, ile jest zbiorów, czyli d, gdzie d to liczba różnych cyfr/znaków
- Po każdej iteracji klucze są usuwane z list i łączone w jedną listę główną $\Rightarrow$ klucze są uporządkowane na tej liście zgodnie z kolejnością łączonych list
- W kolejnej iteracji lista główna jest przeglądana od początku, a każdy klucz jest umieszczany na końcu listy, do której ma być w bieżącej iteracji dołączony

Sortowanie pozycyjne LSD-radix-sort

```
// d - liczba r   nych cyfr, T - zbi r liczb zapisanych na k
     pozycjach w systemie o podstawie d
struct node { T key; node* next; };
LsdRadixSort(node* list, k) {
   node* list_d[1..d];
   for i := 1 to k do
     distribute(list, list_d, i);
     merge(list, list_d);
   end for
}
```

- `distribute()`: przegląda listę `list` (n elementów) i w zależności od wartości v i-tej cyfry dołącza ten element na koniec listy `list_d[v]`
- `merge()`: scala listy `list_d` w jedną listę `list` — wymaga d operacji
- efektywna implementacja wymaga przechowywania wskaźnika do ostatniego elementu każdej listy

Sortowanie pozycyjne LSD-radix-sort

- Złożoność obliczeniowa
$$\mathcal{O}(k(n+d))$$
- Przykłady:
  - sortowanie 10 liczb 10-cyfrowych: $n = 10$, $d = 10$, $k = 10$ $\Rightarrow \mathcal{O}(n^2)$
  - sortowanie 10^6 numerów PESEL: $n = 10^6$, $d = 10$, $k = 11$ $\Rightarrow \mathcal{O}(n)$
- Złożoność pamięciowa: wykorzystujemy listy, więc potrzebujemy liniowej ze względu na n ilości dodatkowej pamięci na wskaźniki

Sortowanie przez zliczanie (counting-sort)

Założenie: kluczami są liczby całkowite

Dodatkowa informacja o kluczach

- Wszystkie klucze należą do znanego, skończonego zbioru, tzn. znany jest zakres kluczy
- Zakres ten obejmuje k różnych kluczy (np. $[1, \ldots, k]$)

Sposób wykorzystania

- Idea algorytmu polega na sprawdzeniu ile wystąpień danego klucza znajduje się w sortowanej tablicy
- Tworzymy pomocniczą tablicę `C` o rozmiarze równym zakresowi kluczy k
- i-ty element tablicy `C` zawiera liczbę wystąpień klucza o wartości i w sortowanej tablicy

Sortowanie przez zliczanie (counting-sort)

Klucze do posortowania

0	4	2	2	0	0	1	1	0	1	0	2	4	2

Tablica zliczająca C

0	1	2	3	4
5	3	4	0	2

Posortowane klucze

0	0	0	0	0	1	1	1	2	2	2	2	4	4

Sortowanie przez zliczanie (counting-sort)

```
CountingSort(T arr[1..n], 1..k){
   integer C[1..k]; //tab., rozm.=zakr. kluczy [1..k]
   B:=map(T, 1..k); //mapuje [1..k] na zb.kluczy T,O(k)
   for i:=1 to k do C[i]:=0;  //O(k)
   for i:=1 to n              //O(n)
      C[arr[i].key] := C[arr[i].key] + 1;
   l:=0;
   for i:=1 to k do
      for j:=1 to C[i] do
         l:=l+1;
         arr[l]=B[i];
}
```

- Złożoność obliczeniowa: $\mathcal{O}(n+k)$
- Złożoność pamięciowa: $\mathcal{O}(k)$

Sortowanie kubełkowe, bucket-sort

Założenie: kluczami są liczby rzeczywiste

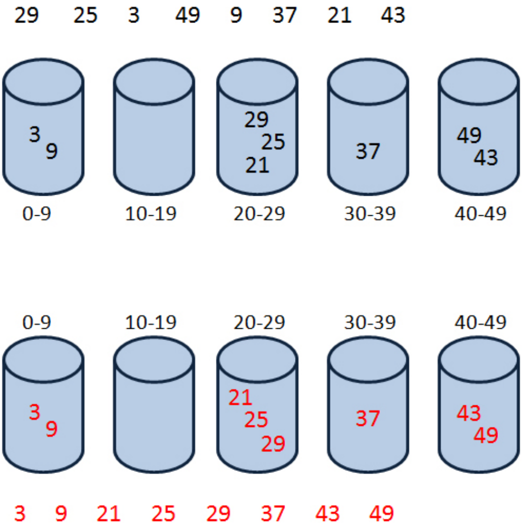
Dodatkowa informacja o kluczach

- Wszystkie klucze należą do znanego skończonego przedziału, np. $[0, m]$
- Jednostajny rozkład kluczy

Sposób wykorzystania

- Podział przedziału $[0, m]$ na l podprzedziałów, które odpowiadają liczbie kubełków (bucket)
- Dystrybucja elementów n-elementowej tablicy do odpowiednich kubełków
- Oczekujemy, że dzięki jednostajnemu rozkładowi w każdym kubełku będzie niewiele liczb
- Sortujemy liczby w każdym kubełku i scalamy rozwiązanie

Sortowanie kubełkowe, bucket-sort



Sortowanie kubełkowe, bucket-sort

```
BucketSort(real arr[1..n], integer l, real max){
   list_of_real_element bucket[1..l];
   real dx = max/l;
   for i:=1 to n do //O(n)
      add(bucket[⌊arr[i]/dx⌋ + 1], arr[i]);
   for i:=1 to l do //O(cl)
      sort(bucket[i]);
   j := 1;
   for i:=1 to l do //O(n)
      copy(arr[j..(j+size(bucket[i])-1)], bucket[i]);
      j:=j+size(bucket[i]);
}
```

- Złożoność obliczeniowa: optymistyczna $\mathcal{O}(n)$, pesymistyczna $\mathcal{O}(n^2)$
- Złożoność pamięciowa: $\Theta(n)$

Porównanie metod sortowania

Algorytm	Złożoność			Stabilny	Metoda
	średnia	najgorsza	pamięciowa		
bubble-sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	tak	zamiana
insert-sort	$\mathcal{O}(n + \mathrm{inv})$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	tak	wstawianie
select-sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	nie	selekcja
comb-sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	nie	zamiana
shell-sort		$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(1)$	nie	wstawianie
merge-sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	tak	scalanie
heap-sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	nie	selekcja
quick-sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$	nie	podział
intro-sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	nie	hybrydowy
radix-sort	$\mathcal{O}(k(n + d))$	$\mathcal{O}(k(n + d))$	$\mathcal{O}(n)$	tak	
counting-sort	$\mathcal{O}(n + k)$	$\mathcal{O}(n + k)$	$\mathcal{O}(n[+k])$	tak/nie	
bucket-sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	tak	