Źródła

- N.Wirth „*Algorithms and Data Structures*”, 1985
- D.E.Knuth „*The Art of Computer Programming. Vol. 3*”, 1973

Sortowanie

- Szukamy uporządkowania; mamy wiele algorytmów o różnych zaletach i różnych stopniach skomplikowania
- Sortowanie tablic (*internal* – wszystkie elementy dostępne) vs. plików (*external* – jeden element dostępny w danej chwili).
- Sortowanie n elementów $a_0,...,a_{n-1}$ to znalezienie takiej permutacji $a_{k0},...,a_{k[n-1]}$ że spełniona jest zależność $f(a_{k0}) \leq ... \leq f(a_{k[n-1]})$ dla funkcji porządkującej $f(x)$.

Sortowanie

- Zakładamy, że dane są uporządkowane w struktury zawierające pole klucza (np. Numer konta, NIP, itp.). Dla ułatwienia będziemy rozważać tablice złożone z samych kluczy – liczb naturalnych.
- Sortujemy *in situ* – bez wykorzystania pomocniczych tablic.
- Mówimy, że sortowanie jest **stabilne** jeśli zachowuje porządek elementów o tej samej wartości klucza – ważne przy sortowaniu po wielu kluczach.

Sortowanie przez proste wstawianie (*sorting by straight insertion*)

- Często stosowany przy sortowaniu kart do gry.
- Ciąg dzielimy w kolejnych krokach na część posortowaną i nieposortowaną. Element a_i w kroku i wstawiamy w odpowiednie miejsce.

Sortowanie przez proste wstawianie (*sorting by straight insertion*)

```
PROCEDURE StraightInsertion;
VAR i, j: INTEGER; x: Item;
BEGIN
   FOR i := 1 TO n-1 DO
      x := a[i]; j := i;
      // wstaw x w odpowiednie miejsce ciągu
      a0,...,ai
      WHILE (j > 0) & (x < a[j-1]) DO
         a[j] := a[j-1];
         DEC(j)
      END ;
      a[j] := x
   END
END StraightInsertion
```

Sortowanie przez proste wstawianie (*sorting by straight insertion*)

```
     44 55 12 42 94 18 06 67
i=1  44 55 12 42 94 18 06 67
i=2  12 44 55 42 94 18 06 67
i=3  12 42 44 55 94 18 06 67
i=4  12 42 44 55 94 18 06 67
i=5  12 18 42 44 55 94 06 67
i=6  06 12 18 42 44 55 94 67
i=7  06 12 18 42 44 55 67 94
```

Sortowanie przez proste wstawianie (*sorting by straight insertion*)

- C – comparisons M - moves
- $C_{min} = n-1$ $M_{min} = 3*(n-1)$
- $C_{max} = (n^2+n-4)/4$ $M_{max} = (n^2+3n-4)/2$
- Stabilny.
- W najgorszym przypadku (tablica posortowana odwrotnie) $O(n^2)$ porównań i $O(n^2)$ przypisań (i-1 w kroku i, n-1 kroków).
- W najlepszym przypadku (tablica już posortowana) O(n) porównań i O(n) przypisań.

Sortowanie przez binarne wstawianie (*binary insertion*)

- Ponieważ ciąg $a_0,...,a_i$ jest już posortowany możemy zastosować binarne wyszukiwanie (*binary search*).
- O(nlog n) porównań, jednak operacje przypisania są zwykle kosztowniejsze, więc mało opłacalne.

Sortowanie przez prosty wybór (*sorting by straight selection*)

- Wybieramy najmniejszy element a następnie zamieniamy z elementem na pozycji i (w kroku i).

Sortowanie przez prosty wybór (*sorting by straight selection*)

```
PROCEDURE StraightSelection;
VAR i, j, k: INTEGER; x: Item;
BEGIN
   FOR i := 0 TO n-2 DO
      k := i;
      x := a[i];
      FOR j := i+1 TO n-1 DO
         IF a[j] < x THEN
            k := j; x := a[k]
         END
      END ;
      a[k] := a[i]; a[i] := x;
   END
END StraightSelection
```

Sortowanie przez prosty wybór (*sorting by straight selection*)

```
44 55 12 42 94 18 06 67
06 55 12 42 94 18 44 67
06 12 55 42 94 18 44 67
06 12 18 42 94 55 44 67
06 12 18 42 94 55 44 67
06 12 18 42 44 55 94 67
06 12 18 42 44 55 94 67
06 12 18 42 44 55 67 94
```

Sortowanie przez prosty wybór (*sorting by straight selection*)

- $C = (n^2-n)/2$
- $M_{min} = 3*(n-1)$ $M_{max} = n^2/4+3*(n-1)$
- Nie jest stabilny. Niezależnie od początkowego uporządkowania tablicy wykonujemy $O(n^2)$ porównań. Ilość przypisań od $O(n)$ do $O(n^2)$ (oprócz zamiany elementów musimy zapamiętać element minimalny).
- Zwykle nieco lepsze od sortowania przez wstawianie dla nieuporządkowanych tablic, dla uporządkowanych gorsze.

Sortowanie przez prostą zamianę (*sorting by straight exchange*)

- Zwane też sortowaniem bąbelkowym (bubble sort).
- Zamieniamy elementy na sąsiadujących pozycjach (mniejsze, niczym lżejsze bąbelki idą w górę).

Sortowanie przez prostą zamianę (*sorting by straight exchange*)

```
PROCEDURE BubbleSort;
VAR i, j: INTEGER; x: Item;
BEGIN
   FOR i := 1 TO n-1 DO
      FOR j := n-1 TO i BY -1 DO
         IF a[j-1] > a[j] THEN
            x := a[j-1];
            a[j-1] := a[j];
            a[j] := x
         END
      END
   END
END BubbleSort
```

Sortowanie przez prostą zamianę (*sorting by straight exchange*)

I =	1	2	3	4	5	6	7	8
	44	06	06	06	06	06	06	06
	55	44	12	12	12	12	12	12
	12	55	44	18	18	18	18	18
	42	12	55	44	42	42	42	42
	94	42	18	55	44	44	44	44
	18	94	42	42	55	55	55	55
	06	18	94	67	67	67	67	67
	67	67	67	94	94	94	94	94

Sortowanie przez prostą zamianę (*sorting by straight exchange*)

- $C = (n^2-n)/2$
- $M_{min} = 0$ $M_{max} = 3*(n^{2-}n)/4$
- Stabilny.
- Praktycznie zawsze gorszy od sortowania przez wstawianie i wybór.

Sortowanie przez prostą zamianę (*sorting by straight exchange*)

- Można zauważyć, że jeśli nie wykonano żadnej zamiany, to znaczy iż tablica jest posortowana (ostatnie trzy kroki w powyższym przykładzie). Ponadto, jeśli ostatnia zamiana miała miejsce dla indeksu k, to wszystkie elementy o niższych indeksach są już posortowane i można ich już nie rozpatrywać. Najlepiej więc zapamiętać indeks ostatniej zamiany.

Sortowanie przez prostą zamianę (*sorting by straight exchange*)

- Pojedynczy „lekki” (mały) bąbelek na końcu tablicy przesunie się na jej początek w jednym kroku. Jednakże, pojedynczy „ciężki” bąbelek na początku tablicy będzie bardzo powoli przesuwał się „na dół” (jedną pozycję na krok), więc można prowadzić sortowanie w dwóch kierunkach (algorytm *Shakersort*).
- Po usprawnieniu C_{min}=n-1, ilość ruchów bez zmian.

Insertion sort by diminishing increment (Shellsort)

- Opisane w 1959 przez D.L.Shella
- Dzielimy tablicę na części i sortujemy oddzielnie przez proste wstawianie, następnie sortujemy całość. Korzystamy z własności, iż sortowanie mocno nieuporządkowanych tablic ma złożoność kwadratową, a prawie uporządkowanych w przybliżeniu liniową.
- Dzielimy kolejno na mniejszą ilość części, np. co czwarty element, co drugi element, całość.

Insertion sort by diminishing increment (Shellsort)

```
PROCEDURE ShellSort;
CONST T = 4;
VAR i, j, k, m, s: INTEGER;
   x: Item;
   h: ARRAY T OF INTEGER;
BEGIN h[0]:=9; h[1]:=5; h[2]:=3; h[3]:=1;
   FOR m := 0 TO T-1 DO
      k := h[m];
      FOR i := k+1 TO n-1 DO
         x := a[i]; j := i-k;
         WHILE (j >= k) & (x < a[j]) DO
            a[j+k] := a[j]; j := j-k
         END;
         a[j+k]:=x
      END
   END
END ShellSort
```

Insertion sort by diminishing increment (Shellsort)

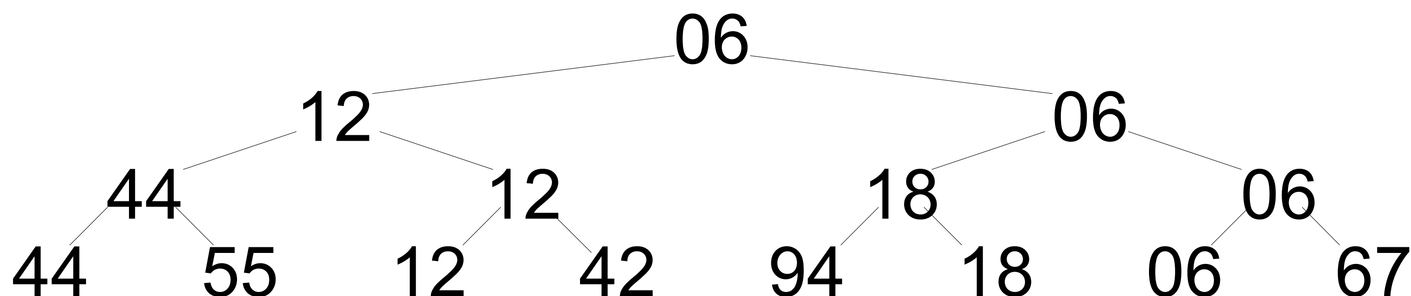
	44	**55**	**12**	**42**	**94**	**18**	**06**	**67**
4	**44**	**18**	**06**	**42**	**94**	**55**	**12**	**67**
2	**06**	**18**	**12**	**42**	**44**	**55**	**94**	**67**
1	**06**	**12**	**18**	**42**	**44**	**55**	**67**	**94**

Insertion sort by diminishing increment (Shellsort)

- Nie jest stabilny.
- Powstaje pytanie na ile części dzielić? D. Knuth proponuje następujący ciąg:
  - $h_1 = 1$
  - $h_k = 3*h_{k-1}+1$ aż do $h_k \geq n$
  - h_k i h_{k-1} (dwa ostatnie elementy) odrzucamy, kolejne elementy ciągu wyznaczają ilość partycji np. Dla n=1000 mamy: **1, 4, 13, 40, 121**, 364, 1093>n
- Inny ciąg Knutha: $h_k = 2*h_{k-1}+1$ (1,3,7,15,...) daje złożoność $O(n^{1.2})$

Sortowanie przez kopcowanie *(tree sort, heapsort)*

- Sortowanie przez wybór można ulepszyć ułatwiając znalezienie minimalnego elementu. Pomóc w tym może struktura drzewiasta.
- Wystarczy n/2 porównań, by znaleźć mniejszy element dla każdej dwójki, n/4 by znaleźć najmniejszy z dwóch dwójek, itd. W sumie n-1 operacji aby uzyskać poniższe odwzorowanie:

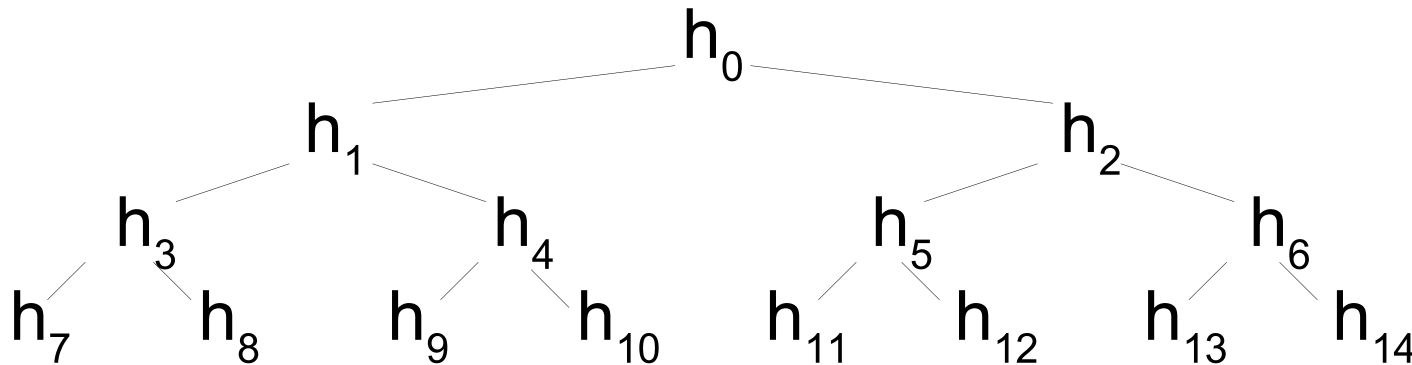


Sortowanie przez kopcowanie *(tree sort, heapsort)*

- Posortowany ciąg uzyskujemy schodząc po najniższym elemencie w dół, usuwamy go i zastępujemy dziury najmniejszymi elementami z pary. Potrzebujemy log n przejść, a więc w sumie n*log n operacji.
- Metoda J.Williamsa zwana *Heapsort* rozwija tę procedurę w ten sposób, iż pozwala na sortowanie in situ, bez użycia dodatkowej pamięci, dzięki strukturze zwanej **kopcem**.

Kopiec

- Kopiec to sekwencja kluczy $h_L, h_{L+1}, ..., h_R$ taka, że: $h_i < h_{2i+1}$ i $h_i < h_{2i+2}$ dla i=L...R/2-1
- Kopiec można przedstawić jako drzewo, w którym węzły na danym poziomie są większe od swoich potomków:



Tworzenie kopca

- Elementy h_k dla k=n/2...n-1 są już w poprawnej części kopca, gdyż nie mają potomków.
- Kolejne elementy dla k=n/2...0 dodajemy do kopca. Dodawanie polega na ustawieniu elementu na szczycie kopca i „przesianiu” go w dół na odpowiednie miejsce.

```
L := n DIV 2;
WHILE L > 0 DO
   DEC(L);
   sift(L, n-1)
END
```

Tworzenie kopca

```
44 55 12 42 94 18 06 67
44 55 12 42|94 18 06 67
44 55 12|42 94 18 06 67
44 55|06 42 94 18 12 67
44|42 06 55 94 18 12 67
06 42 12 55 94 18 44 67
```

Sortowanie z użyciem kopca

- Zamień element znajdujący sie w korzeniu z ostatnim elementem w kopcu. Napraw kopiec „przesiewając” element z korzenia w dół. Elementy przenoszone na koniec kopca tworzą część posortowaną (nie wchodzącą już w skład kopca).

```
R := n-1;
WHILE R > 0 DO
   x := a[0]; a[0] := a[R]; a[R] := x;
   DEC(R); sift(1, R)
END
```

Sortowanie z użyciem kopca

```
06 42 12 55 94 18 44 67
12 42 18 55 94 67 44|06
18 42 44 55 94 67|12 06
42 55 44 67 94|18 12 06
44 55 94 67|42 18 12 06
55 67 94|44 42 18 12 06
67 94|55 44 42 18 12 06
94|67 55 44 42 18 12 06
```

```
PROCEDURE sift(L, R: INTEGER);
   VAR i, j: INTEGER; x: Item;
BEGIN i := L; j := 2*i+1; x := a[i];
   IF (j < R) & (a[j] < a[j+1]) THEN j := j+1 END;
   WHILE (j <= R) & (x < a[j]) DO
      a[i] := a[j]; i := j; j := 2*j+1;
      IF (j < R) & (a[j] < a[j+1]) THEN j := j+1
      END
   END ;
   a[i] := x
END sift;

PROCEDURE HeapSort;
   VAR L, R: INTEGER; x: Item;
BEGIN L := n DIV 2; R := n-1;
   WHILE L > 0 DO DEC(L); sift(L, R) END ;
   WHILE R > 0 DO
      x := a[0]; a[0] := a[R]; a[R] := x;
      DEC(R); sift(L, R)
   END
END HeapSort
```

Sortowanie przez kopcowanie *(tree sort, heapsort)*

- Aby zbudować kopiec potrzeba n/2 przesiań przez max. log n/2 poziomów. Aby posortować, n-1 przesiań przez max. log n-1 poziomów. A więc złożoność wynosi O(nlog n).
- Pomimo tego, że elementy są przesuwane najpierw w jednym kierunku, a potem w drugim, algorytm jest szybszy niż Shellsort dla dużych n.
- Generalnie heapsort najlepiej działa na tablicach posortowanych odwrotnie, a więc zachowuje się w sposób nienaturalny.
- Nie jest stabilny.

Sortowanie przez podział *(partition sort, quicksort)*

- Sortowanie przez zamianę próbujemy ulepszyć poprzez zamianę nie elementów najbliższych, ale najdalszych (będzie działać idealnie dla ciągu posortowanego odwrotnie).
- C.A.R. Hoare zaproponował w latach 60-tych procedurę opartą o strategię „dziel i zwyciężaj” (*divide and conquer*):

Sortowanie przez podział *(partition sort, quicksort)*

- **Dziel:** najpierw sortowany zbiór dzielimy na dwie części w taki sposób, aby wszystkie elementy leżące w pierwszej części (zwanej lewą partycją) były mniejsze lub równe od wszystkich elementów drugiej części zbioru (zwanej prawą partycją).
- **Zwyciężaj:** każdą z partycji sortujemy rekurencyjnie tym samym algorytmem.
- **Połącz:** połączenie tych dwóch partycji w jeden zbiór daje w wyniku zbiór posortowany.

Sortowanie przez podział *(partition sort, quicksort)*

- Aby podzielić zbiór, musimy wybrać wartość względem której będziemy dzielić. Najlepiej żeby to była mediana, ale ponieważ jej znalezienie kosztuje możemy wybrać element środkowy (będzie to korzystne gdy ciąg jest posortowany w jakimś kierunku).
- Poruszamy się dwoma indeksami (nazwijmy je i,j) po tablicy. Najłatwiej startować z dwóch stron tablicy, ale można też z jednej. Jeśli element pod indeksem i jest większy od elementu pod indeksem j to je zamieniamy.

Sortowanie przez podział *(partition sort, quicksort)*

- 44 55 12 42 94 06 18 67 – wybieramy 42
- 18 55 12 42 94 06 44 67 – zamiana 18 i 44
- 18 06 12 42 94 55 44 67 – zamiana 6 i 55
- Procedurę powtarzamy dla partycji [18 06 12] i [94 55 44 67]

```
PROCEDURE sort(L, R: INTEGER);
   VAR i, j: INTEGER; w, x: Item;
BEGIN i := L; j := R;
   x := a[(L+R) DIV 2];
   REPEAT
      WHILE a[i] < x DO INC(i) END ;
      WHILE x < a[j] DO DEC(j) END ;
      IF i <= j THEN
         w := a[i]; a[i] := a[j]; a[j] := w;
         i := i+1; j := j-1;
      END
   UNTIL i > j;
   IF L < j THEN sort(L, j) END ;
   IF i < R THEN sort(i, R) END ;
END sort;

PROCEDURE QuickSort;
BEGIN
   sort(0, n-1)
END QuickSort
```

Sortowanie przez podział *(partition sort, quicksort)*

- Nie jest stabilny. W najlepszym przypadku złożoność O(nlog n), w najgorszym n^2 (jeśli zawsze wybieramy element największy lub najmniejszy).
- W najgorszym przypadku potrzeba n wywołań funkcji, co grozi przepełnieniem stosu. Można rozwiązać ten problem poprzez wywoływanie procedury zawsze dla mniejszej z partycji.
- Zadanie domowe: skonstruuj najgorsze dla algorytmu quicksort ułożenie powyższego ciągu. Sprawdź ile zostanie wykonanych operacji podziału i porównań.

Zadanie na ćwiczenia

- Implementacja i porównanie dwóch algorytmów sortowania. Porównujemy dla tablic: posortowanej, posortowanej odwrotnie, losowo nieuporządkowanej. Sprawozdanie powinno zawierać wykresy
- Przykładowe funkcje mierzące czas:
  - Time() - dokładność do sekundy
  - Now() - biblioteka VCL, zwraca typ TDateTime (1.0 = 1 doba)
  - GetSystemTime() - sytemowa, zwraca strukturę, skomplikowany sposób obliczania różnicy czasów

Sortowanie przez scalanie (*mergesort*)

- Strategia „Dziel i rządź”
- Dzielimy tablicę na dwie podtablice, te dzielimy znowu rekurencyjnie aż dojdziemy do tablic jednoelementowych (już posortowanych).
- Przy powrocie z funkcji łączymy posortowane podtablice korzystając z dodatkowej tablicy. Wykorzystujemy dwa indeksy dla tablic źródłowych i indeks dla tablicy wynikowej:

```
IF tab1[a]<tab2[b] THEN
   tab_wynik[c++] := tab1[a++]; END;
ELSE
   tab_wynik[c++] := tab2[b++]; END;
```

Scalanie zbiorów

1. Przygotuj pusty zbiór tymczasowy
2. Dopóki żaden ze scalanych zbiorów nie jest pusty, porównuj ze sobą pierwsze elementy każdego z nich i w zbiorze tymczasowym umieszczaj mniejszy z elementów usuwając go jednocześnie ze scalanego zbioru.
3. W zbiorze tymczasowym umieść zawartość tego scalanego zbioru, który zawiera jeszcze elementy.
4. Zawartość zbioru tymczasowego przepisz do zbioru wynikowego i zakończ algorytm.

Sortowanie przez scalanie (*mergesort*)

```
44 55 12 42 94 18 06 67
44 55 12 42|94 18 06 67
44 55|12 42|94 18|06 67
44|55|12|42|94|18|06|67
44 55|12 42|18 94|06 67
12 42 44 55|06 18 67 94
06 12 18 42 44 55 67 94
```

Sortowanie przez scalanie (*mergesort*)

- Stabilny.
- Nie sortujemy w miejscu.
- Złożoność O(nlog n).

Sortowanie rozrzutowe

- Zamiast porównywać elementy, rozrzucamy je do poszczególnych kategorii. Musimy znać ilość elementów w kategorii.
- Przykład dla 24 kart (Knuth): kolejne karty układamy na 6 stosów wg figur. Karty pobieramy z kupek w tej samej kolejności, w której były na nie wstawiane. W drugim kroku otrzymaną talie rozkładamy na 4 stosy wg kolorów. Teraz wystarczy połączyć ze sobą otrzymane stosy w jedną talię 24 kart.

♥D ♦K ♣K ♥9 ♠D ♣W
♠A ♥K ♣9 ♦T ♦A ♠K
♥T ♠W ♥A ♣D ♦D ♥W
♠9 ♣A ♦9 ♦W ♠T ♣T

A	**K**	**D**	**W**	**T**	**9**
♠A	♦K	♥D	♣W	♦T	♥9
♦A	♣K	♠D	♠W	♥T	♣9
♥A	♥K	♣D	♥W	♠T	♠9
♣A	♠K	♦D	♦W	♣T	♦9

♠	♥	♦	♣
♠A	♥A	♦A	♣A
♠K	♥K	♦K	♣K
♠D	♥D	♦D	♣D
♠W	♥W	♦W	♣W
♠T	♥T	♦T	♣T
♠9	♥9	♦9	♣9

Sortowanie rozrzutowe

- Pierwsza operacja - rozłożenie n elementów na m kupek ma klasę złożoności O(n).
- Druga operacja - złączenie m kupek w n elementów ma klasę złożoności O(m).
- W sumie złożoność O(m+n).
- Potrzebna dodatkowa tablica wielkości n.

Sortowanie kubełkowe (*bucket sort*)

- 1956 E. J. Issac i R. C. Singleton
- Kubełki najlepiej reprezentować za pomocą uporządkowanej listy (czyli stosujemy sortowanie przez wstawianie).
- Złożoność O(n) dla równomiernie rozłożonych elementów. Dla nierównomiernie $O(n^2)$.

1. Podziel zadany przedział liczb na n podprzedziałów (kubełków) o równej długości.
2. Przypisz liczby z sortowanej tablicy do odpowiednich kubełków.
3. Sortuj liczby w niepustych kubełkach.
4. Wypisz po kolei zawartość niepustych kubełków.

Sortowanie kubełkowe (*bucket sort*)

44 55 12 42 94 18 06 67
(94-6)/4 = 22 (wielkość kubełka)

```
<6;28)      06 12 18
<28;50)     42 44
<50;72)     55 67
<72;94+1)   94
```

Sortowanie przez zliczanie (*counting sort*)

- Stosowane gdy sortujemy liczby całkowite z ograniczonego zakresu. Dla każdej wartości w zbiorze przygotowujemy licznik i ustawiamy go na 0.
- Obieg zliczający: przeglądamy kolejne elementy zbioru i zliczamy ich wystąpienia w odpowiednich licznikach. Po wykonaniu tego obiegu w poszczególnych licznikach mamy ilość wystąpień każdej wartości.
- Jeśli nie zależy nam na stabilności sortowania, to w tym momencie możemy przepisać elementy to tablicy wynikowej.

Sortowanie przez zliczanie (*counting sort*)

636149018264937592732418708583625 3

Liczniki:
[0:2][1:3][2:4][3:5][4:3][5:3][6:4][7:3][8:4][9:3]

Wypisujemy liczby:
0011122223333344455566667778888999

Sortowanie przez zliczanie (*counting sort*)

- Jeśli chcemy zachować stabilność sortowania (wartości całkowite są kluczami dla większych struktur danych) wykonujemy obieg dystrybucyjny: obliczamy dystrybuantę, czyli ilość elementów mniejszych lub równych od danej wartości.
- Przeglądamy jeszcze raz zbiór wejściowy idąc od ostatniego elementu do pierwszego. Każdy element umieszczamy w zbiorze wynikowym na pozycji równej zawartości licznika dla tego elementu. Po wykonaniu tej operacji licznik zmniejszamy o 1.

Sortowanie przez zliczanie (*counting sort*)

6361490182649375927324187085836253

Liczniki:
[0:2][1:5][2:9][3:14][4:17][5:20][6:24][7:27][8:31][9:34]

Elementy zachowują kolejność:
0011122222333334445556666777888999

Sortowanie przez zliczanie (*counting sort*)

- Wykonujemy m zerowań liczników.
- Następnie n inkrementacji licznika (dla każdego elementu).
- Następnie m sumowań liczników przy obliczaniu dystrybuanty.
- Następnie n dekrementacji licznika.
- Złożoność O(m+n).
- Potrzeba m+n dodatkowej pamięci.

Sortowanie pozycyjne (*radix sort*)

- Polega na sortowaniu elementów (znaków w ciągu lub cyfr w liczbie) według kolejnych pozycji. Cyfry liczby sortujemy od prawej do lewej, ciągi znaków odwrotnie.
- Algorytm sortujący musi być stabilny, doskonale nadaje się więc sortowanie przez zliczanie (mamy tylko 10 cyfr, a więc 10 liczników).

Sortowanie pozycyjne (*radix sort*)

We.	↓	↓	↓
726	30**1**	3**0**1	**1**23
237	94**1**	7**0**7	**2**37
725	12**3**	1**2**3	**3**01
123	49**4**	7**2**5	**4**94
301	72**5**	7**2**6	**7**07
494	72**6**	2**3**7	**7**25
707	23**7**	9**4**1	**7**26
941	70**7**	4**9**4	**9**41

Sortowanie pozycyjne (*radix sort*)

- Sortowanie musimy wykonać tyle razy, z ilu cyfr zbudowane są liczby reprezentujące sortowane elementy.
- Złożoność dla sortowania liczb wynosi O(d*(n+10)) gdzie d to długość najdłuższej liczby.
- Potrzeba m+n dodatkowej pamięci.