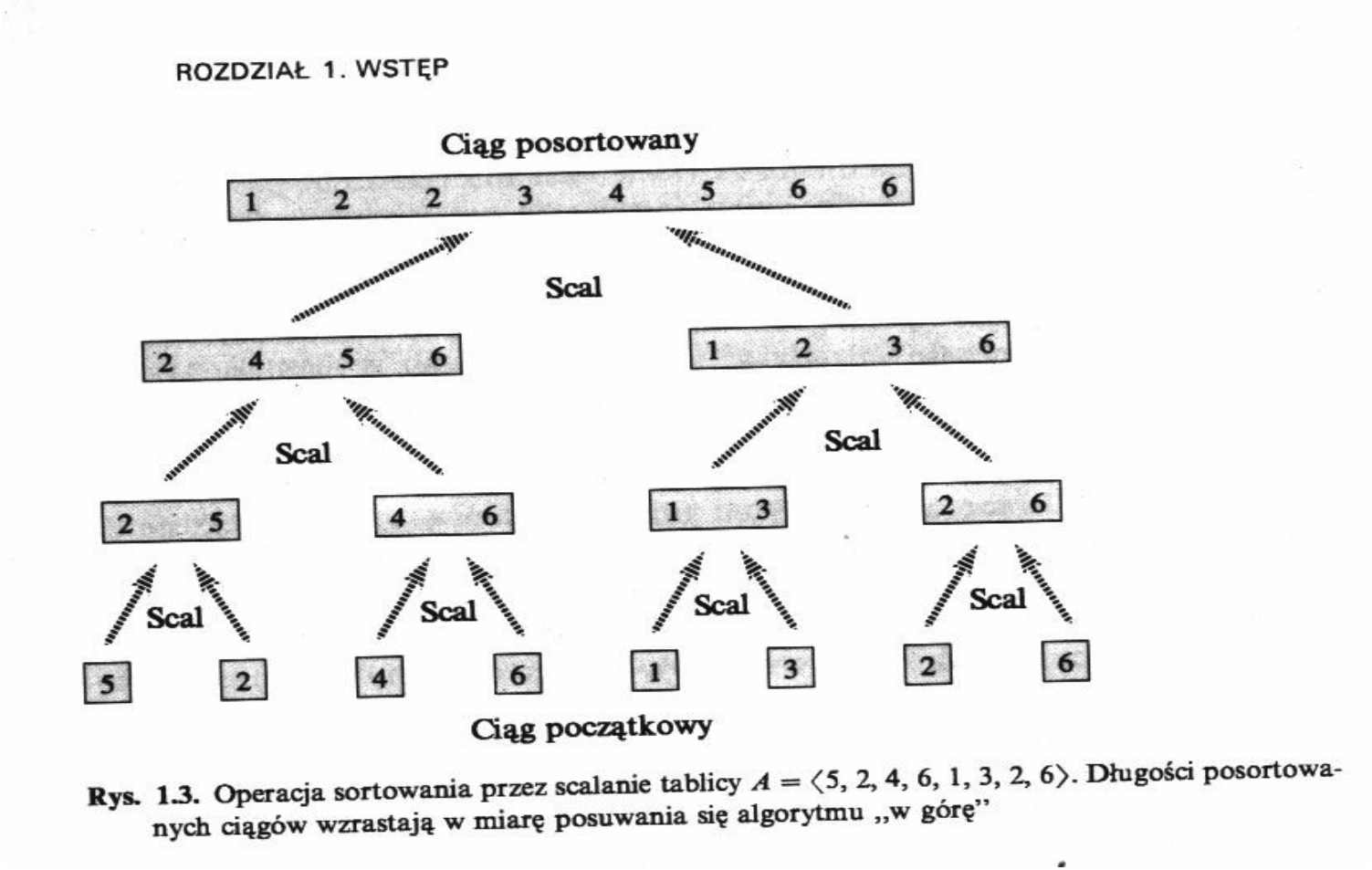
Algorytmy sortowania w czasie O(n logn)

MergeSort

patrz zadanie 5: https://mhanckow.students.wmi.amu.edu.pl/asd_old/asd22002.htm

+ pomocnicza tablica n-elem (wynik scalania)
+ zasada „dziel i zwyciężaj” (redukujemy problem do mniejszych kopii)



Rys. 1.3. Operacja sortowania przez scalanie tablicy $A = \langle 5, 2, 4, 6, 1, 3, 2, 6 \rangle$. Długości posortowanych ciągów wzrastają w miarę posuwania się algorytmu „w górę”

Algorytmy sortowania w czasie O(n logn)

MergeSort

+ czas działania ?

$T(n) = \Theta(1)$, jeśli $n=1$

$T(n) = 2\,T(n/2) + \Theta(n)$, jeśli $n>1$

zgadujemy rozwiązanie: $T(n):=\Theta(n \log n)$;

$2T(n/2)+\Theta(n) = 2\, n/2 \log(n/2) + n = n(\log n - 1) + n = n \log n$

Algorytmy sortowania w czasie O(n^2) lub EO(n logn)

QuickSort

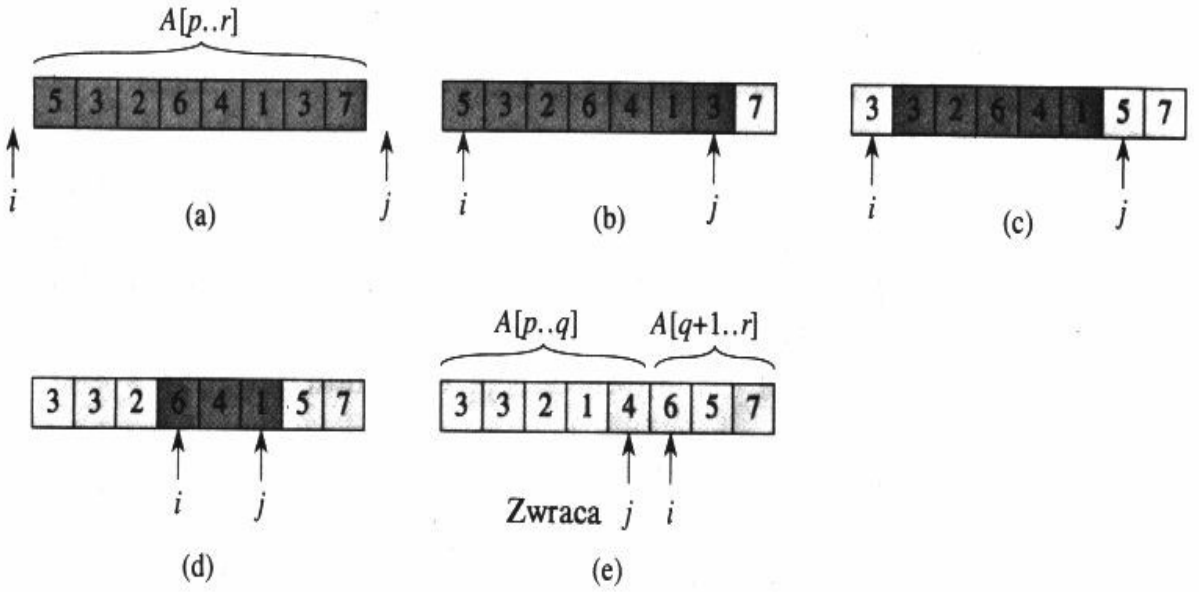
patrz zadanie 6: https://mhanckow.students.wmi.amu.edu.pl/asd_old/asd22003.htm

+ sortowanie „w miejscu” (brak pomocniczej tabl)
+ procedura „podział” jest trikowa... elem <=x i elem >=x
+ „dziel i zwyciężaj”, ale części NIE są równe (jak w MergeSort) !!

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q)
4          QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
 1  x ← A[p]
 2  i ← p − 1
 3  j ← r + 1
 4  while TRUE
 5     do repeat j ← j − 1
 6           until A[j] ⩽ x
 7        repeat i ← i + 1
 8           until A[i] ⩾ x
 9        if i < j
10           then zamień A[i] ↔ A[j]
11           else return j
```

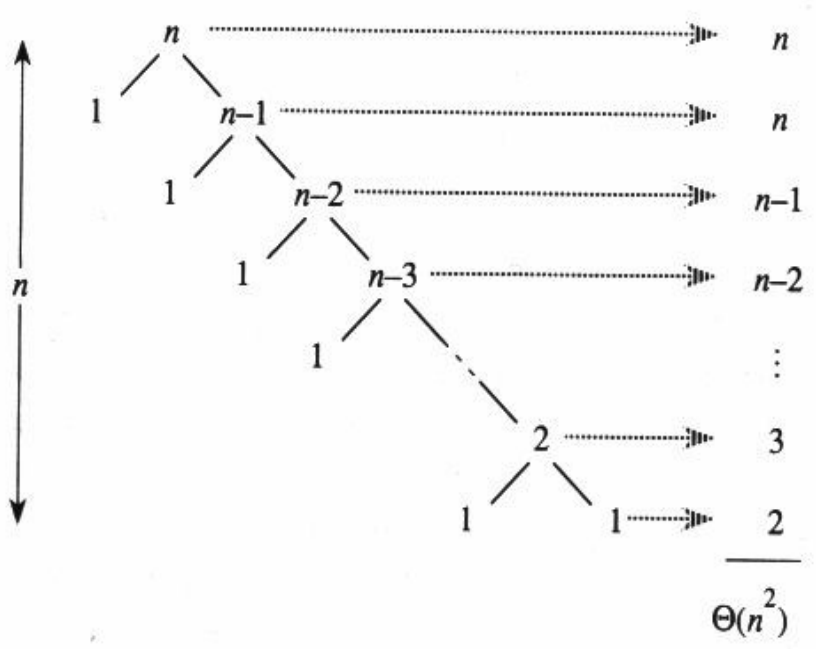
Objaśnienie proc „podział”:

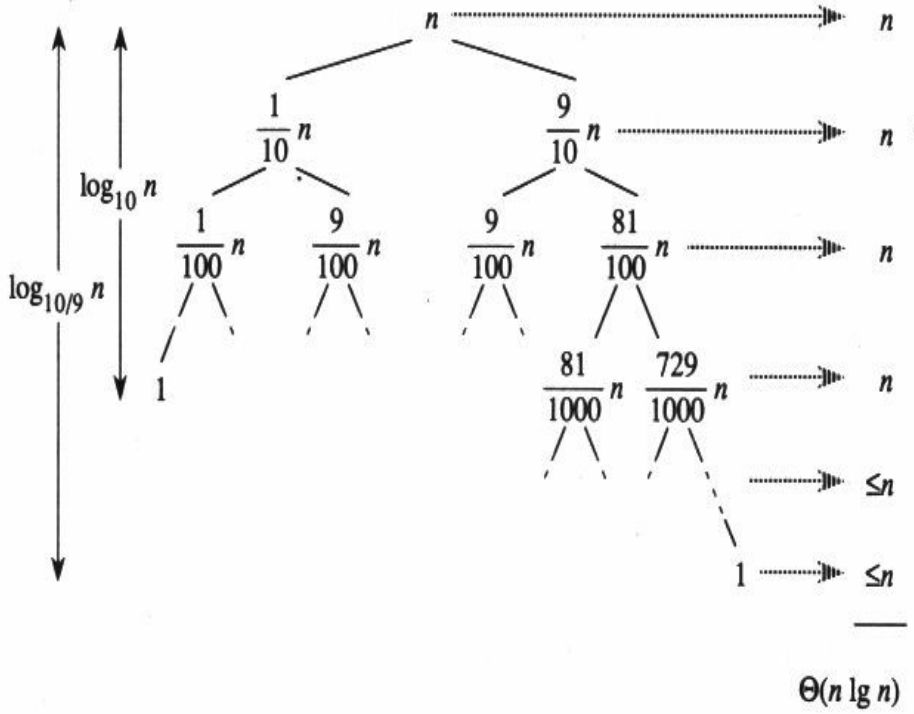


Algorytmy sortowania w czasie O(n^2) lub EO(n logn)

QuickSort

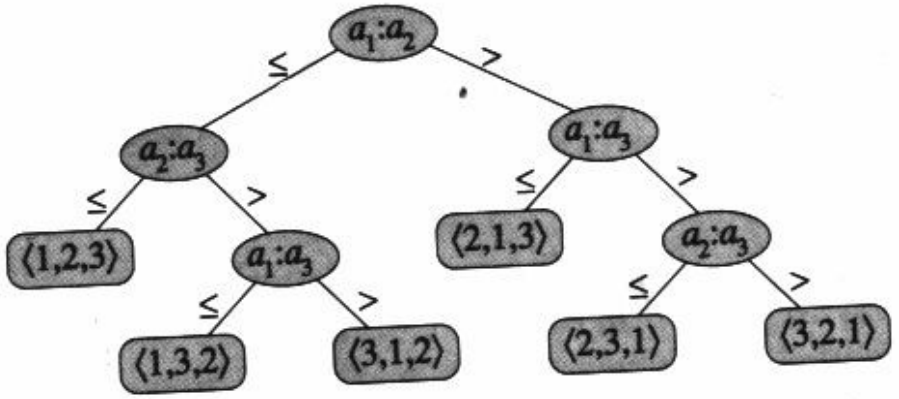
+ czas działania ??
  T(n) = T(1) + T(n-1) + Θ(n), rozwiązanie: T(n):=O(n^2)
  T(n) = 2 T(n/2) + Θ(n), rozwiązanie: T(n):=O(n logn)
  T(n) = T(n*1/10) + T(n*9/10) + Θ(n), rozwiązanie: T(n):=O(n logn)
+ zbalansowane podziały? „Losowe wybieranie x” lub „losowe dane we”
  oczekiwany czas działania = O(n logn)





Sortowanie w czasie O(n)

Dolne ograniczenie na czas sortowania n-elem tablicy,
*jeśli jedynie **porównujemy** elementy tablicy*
+ każdy alg sortujący oblicza pewną permutację...
+ dla każdego alg sortującego można zbudować drzewo decyzyjne
+ pesymistyczna liczba porównań elem = długość ścieżki korzeń-liść



Rys. 9.1. Drzewo decyzyjne odpowiadające algorytmowi sortowania przez wstawianie, działającemu na ciągu 3-elementowym. Istnieje 3! = 6 możliwych permutacji ciągu wejściowego, więc drzewo decyzyjne musi mieć co najmniej 6 liści

Tw 9.1 [Cormen] każde drzewo decyzyjne odp alg sortującemu ma wysokość Ω(n logn)

dowód:
drzewo musi mieć n! liści; drzewo o wysokości h ma 2^h liści
czyli musi być: n! <= 2^h; stosujemy wzór Stirlinga n!>(n/e)^n
h >= log(n!) >= log((n/e)^n) = n log(n/e) = Ω(n logn)

Sortowanie w czasie O(n)

Sortowanie przez zliczanie:

+ zakładamy, że elem tablicy są liczbami z {1,2, …, k}, k=O(n)
+ A-we, B-wy, C-pomoc;

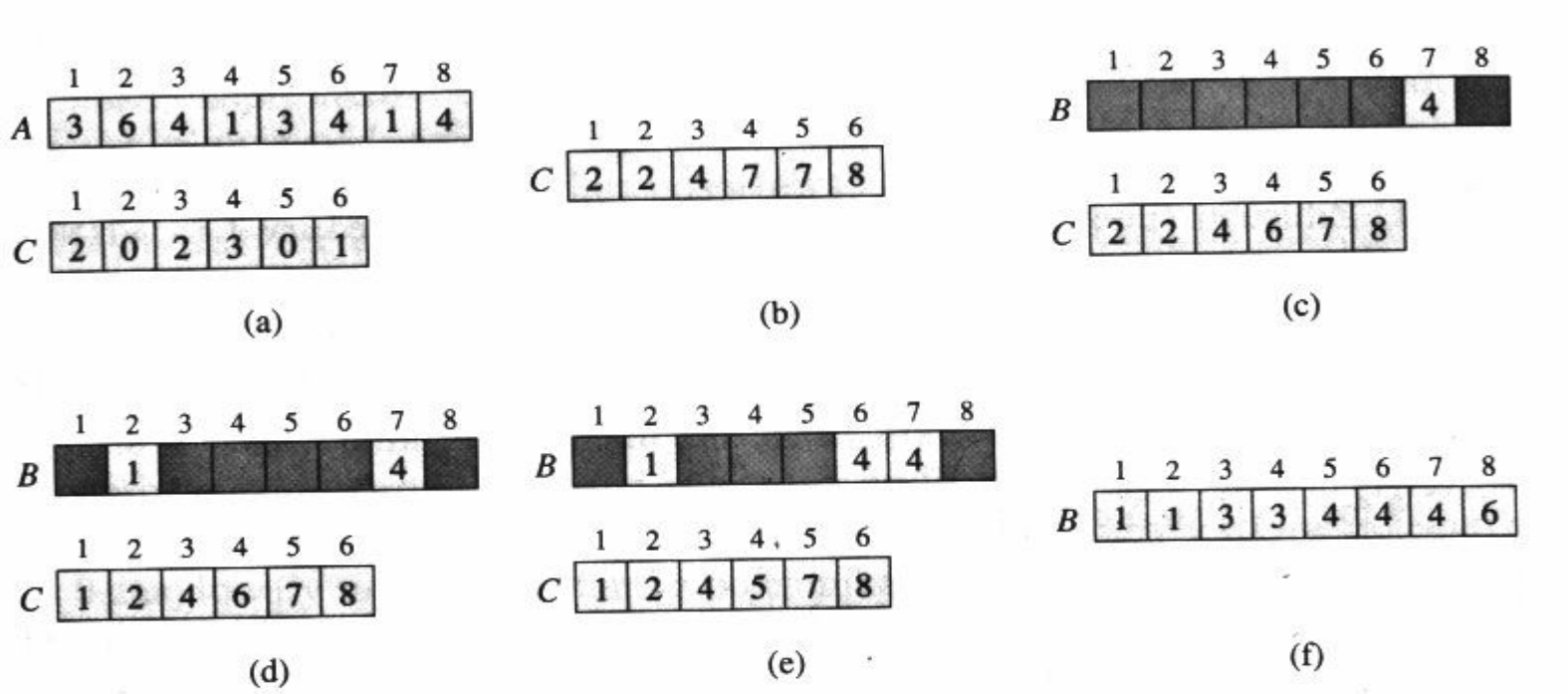
dla „x” obliczamy ile jest elem <= x w A,

to nam mówi gdzie powinien trafić „x” w B...

```
COUNTING-SORT(A, B, k)
 1  for i ← 1 to k
 2      do C[i] ← 0
 3  for j ← 1 to length[A]
 4      do C[A[i]] ← C[A[i]] + 1
 5  ▷ C[i] zawiera teraz liczbę elementów równych i.
 6  for i ← 2 to k
 7      do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] zawiera teraz liczbę elementów mniejszych lub równych i.
 9  for j ← length[A] downto 1
10      do B[C[A[i]]] ← A[i]
11         C[A[i]] ← C[A[i]] − 1
```

Sortowanie w czasie O(n)

Sortowanie przez zliczanie – objaśnienie działania:



Rys. 9.2. Działanie procedury COUNTING-SORT dla tablicy wejściowej $A[1..8]$, w której każdy element jest dodatnią liczbą całkowitą nie większą od $k = 6$. **(a)** Tablica A oraz pomocnicza tablica C po wykonaniu wiersza 4. **(b)** Tablica C po wykonaniu wiersza 7. **(c)-(e)** Tablica B oraz pomocnicza tablica C po wykonaniu odpowiednio jednej, dwóch oraz trzech iteracji pętli w wierszach 9-11. Tylko jasnoszare elementy tablicy B zostały wypełnione. **(f)** Ostateczna zawartość tablicy B

Sortowanie w czasie O(n)

Sortowanie pozycyjne:

+ używane przez starodawne maszyny sortujące karty perforowane
+ sortowanie „stabline” (elem o wartości k **nie** zmieniają kolejności)
+ zał: sortujemy liczby dziesiętne, 10 kubełków,
  sortujemy wg najmłodszej cyfry 0-9, potem wg starszej cyfry, itd.
  ważne aby to było sortowanie stabilne !!
+ czas? cyfry: {0..k-1}, liczby d-cyfrowe, n-liczb
  czas działania Θ(d(n+k)), jeśli d-stała, k=O(n), to czas=O(n)