Wyrażenie nawiasowe

Wyrażeniem nawiasowym nazywamy dowolny skończony ciąg nawiasów. Każdemu nawiasowi otwierającemu odpowiada dokładnie jeden nawias zamykający. Poprawne wyrażenie nawiasowe definiujemy następująco:

– Wyrażenie puste jest poprawnym wyrażeniem nawiasowym.

– Wyrażenie postaci AB, gdzie A i B są poprawnymi niepustymi wyrażeniami nawiasowymi, jest poprawnym wyrażeniem nawiasowym.

– Wyrażenie postaci kAk’, gdzie k jest nawiasem otwierającym, A jest poprawnym wyrażeniem nawiasowym, a k’ – nawiasem zamykającym odpowiadającym k, jest poprawnym wyrażeniem nawiasowym.

– Żadne inne wyrażenie nie jest poprawnym wyrażeniem nawiasowym.

Przykład:

poprawne wyrażenie nawiasowe: `[[]][]`
niepoprawne wyrażenie nawiasowe: `[]][[[]]`

Nawiasy mogą być różnego typu. Możemy je np. reprezentować przez zbiór liczb, gdzie liczba dodatnia oznacza nawias otwierający, a odpowiadająca jej liczba ujemna – nawias zamykający.

Przykład:

poprawne wyrażenie nawiasowe: `1 -1 2 1 -1 -2`
niepoprawne wyrażenie nawiasowe: `1 2 -1 1 -2 -1`

Automat stosowy

Automat skończony

Abstrakcyjna maszyna o skończonej liczbie stanów (z wyróżnionym stanem początkowym), czytająca słowa znajdujące się na wejściu i zmieniająca swój stan, w zależności od bieżącego stanu oraz aktualnie przeczytanego znaku.

Automat stosowy

Automat skończony operujący na nieskończenie dużej pamięci, działającej na zasadzie stosu (czyli na wierzch stosu można zawsze włożyć nowy element; jeśli stos nie jest pusty, to można zawsze zobaczyć jaki element jest na wierzchu i ew. zdjąć ten element; nie można oglądać ani wyjmować elementów z wnętrza stosu). Dodatkowo, wykonanemu przejściu nie musi towarzyszyć wczytanie znaku z wejścia (tzw. ε-przejście).

Formalna definicja

Automat stosowy to zbiór $M=<Q, \Sigma, \Gamma, \delta, s, \perp>$ taki, że:

– Q to skończony zbiór stanów,
– Σ to skończony alfabet wejściowy,
– Γ to skończony alfabet stosowy,
– δ to relacja przejścia pomiędzy stanami; określa ona do jakiego stanu należy przejść i co należy włożyć / zdjąć ze stosu, w zależności od aktualnego stanu, znaku na wierzchołku stosu i znaku na wejściu,
– $s \in Q$ to stan początkowy,
– $\perp \in \Gamma$ to symbol początkowy na stosie („pinezka”).

Konfiguracja automatu

Stan automatu (ze skończonego zbioru stanów), zawartość stosu i pozostałe do wczytania wejście.

Czynności automatu dla jednego kroku obliczeń

1. Podgląda znak czekający na wczytanie na wejściu.
2. Podgląda element na wierzchołku stosu.
3. Na podstawie tych informacji wybiera jedno z przejść do wykonania.
4. Jeśli to nie jest ε-przejście, to wczytywany jest jeden znak z wejścia.
5. Zdejmowany jest element z wierzchołku stosu.
6. Pewna liczba określonych elementów może być włożona na stos.
7. Zgodnie z przejściem zmieniany jest stan.

Uwaga! w każdym kroku zdejmujemy element z wierzchołka stosu (punkt 5). Jeżeli chcemy tylko zdjąć element z wierzchołku stosu, to wystarczy, że nie będziemy nic wkładać (w punkcie 6). Jeżeli chcemy podmienić element na wierzchołku, to wystarczy, że włożymy jeden element, który zastąpi element zdjęty z wierzchołka. Jeżeli chcemy włożyć elementy na stos, to wkładamy na stos właśnie zdjęty element, wraz z elementami, które mają być umieszczone na stosie. Jeśli nie chcemy zmieniać zawartości stosu, to wkładamy dokładnie ten sam element, który właśnie został zdjęty. Możemy włożyć na stos kilka elementów w jednym kroku, ale nie możemy w jednym kroku zdjąć więcej niż jeden element.

Rola stosu

Przedstawiony schemat postępowania wymaga, żeby stos nie był pusty – w przypadku stosu pustego nie ma czego podglądać ani z niego zdejmować. Stąd wymóg, żeby w momencie uruchomienia automatu stos nie był pusty a zawierał jeden wyróżniony element $\perp$ – „pinezkę”.

Jak tylko stos zostaje opróżniony, automat zatrzymuje się i dalsze jego działanie nie jest możliwe. Jeżeli automat zatrzymuje się z pustym stosem po wczytaniu całego słowa z wejścia, to przyjmujemy, że słowo to zostało zaakceptowane.

Innymi słowy: automat stosowy akceptuje takie słowa, dla których istnieją obliczenia prowadzące od konfiguracji początkowej do konfiguracji, w której całe słowo zostało wczytane, a stos opróżniony.

Przykład 1 – automat akceptujący słowa postaci $a^n b^n$ dla $n \geq 0$.

Wejście: znaki a oraz b.

Dwa stany, s i q, przy czym s niech będzie stanem początkowym.

Na stosie będziemy przechowywać dwa rodzaje elementów: $\perp$ i A.

Będąc w stanie s automat ma dwa przejścia do wyboru: może wczytać z wejścia znak a i włożyć na stos A, lub też nic nie wczytywać ani nie zmieniać zawartości stosu, tylko przejść do stanu q.

W stanie q automat ma również dwa przejścia do wyboru: jeżeli na wejściu czeka na wczytanie znak b i na wierzchołku stosu jest A, to może wczytać z wejścia b i zdjąć ze stosu A. Jeżeli natomiast na wierzchołku stosu jest $\perp$, to automat może zdjąć ze stosu $\perp$ i tym samym zakończyć działanie (stos pozostanie pusty).

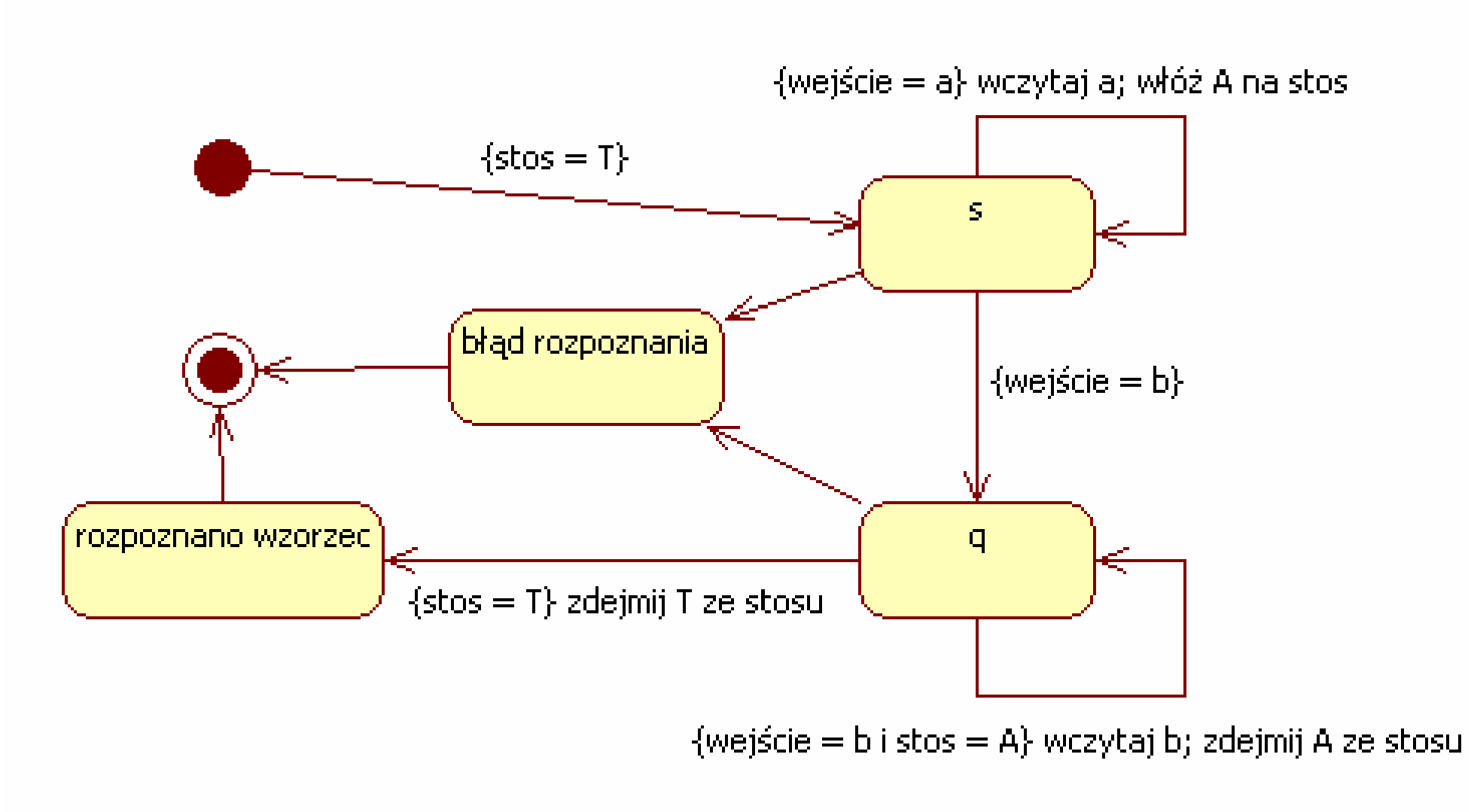
Działanie automatu

Przedstawiony automat wczytuje sekwencję a^n i umieszcza na stosie A^n. Gdy napotka inny znak niż a, przechodzi do stanu `q` i wczytuje znaki b zdejmując ze stosu `A`. Jednych i drugich musi być tyle samo, czyli wczytana zostaje sekwencja b^n. Po wczytaniu tej sekwencji na wierzchołku stosu odsłania się $\perp$ i jest z niego zdejmowana. Jeżeli automat ma zaakceptować słowo, to musi ono być w tym momencie całe wczytane, czyli było to słowo $a^n b^n$.

(patrz plik `w8-ex1.html`)

Podaną powyżej procedurę można uzupełnić o wyświetlanie błędu, gdy np. w stanie `s` na wejściu znajdzie się inny znak niż a lub b, czy też w przypadku, gdy w stanie `q` na wejściu znajdzie się inny znak niż b.

Diagram maszyny stanowej dla automatu (z obsługą błędów)



Przykład 2 – automat akceptujący proste wyrażenia nawiasowe

Wejście: znaki `[` oraz `]`.

Stosu użyjemy jako licznika nawiasów. Licznik będzie mógł przyjmować tylko wartości dodatnie.

Automat będzie posiadał tylko jeden stan `s`.

Działanie:
Na stosie trzymamy pinezkę i tyle nawiasów zamykających (`]`) ile wynosi wartość licznika. Z wczytaniem każdego nawiasu otwierającego licznik jest zwiększany o 1, a z wczytaniem każdego nawiasu zamykającego zmniejszany o 1. Jeżeli na stosie pozostanie już tylko znak ⊥, zdejmujemy go a liczba nawiasów zamykających się zbilansowała z otwierającymi.

(patrz plik `w8-ex2.html`)

Sortowanie liniowe (w ramach remanentu)

Sortowanie za pomocą porównań

Porządek wyjściowy jest wyznaczany jedynie na podstawie wyników porównań między elementami.

– sortowanie przez wstawianie
– sortowanie przez scalanie
– sortowanie przez kopcowanie
– sortowanie szybkie (quicksort)

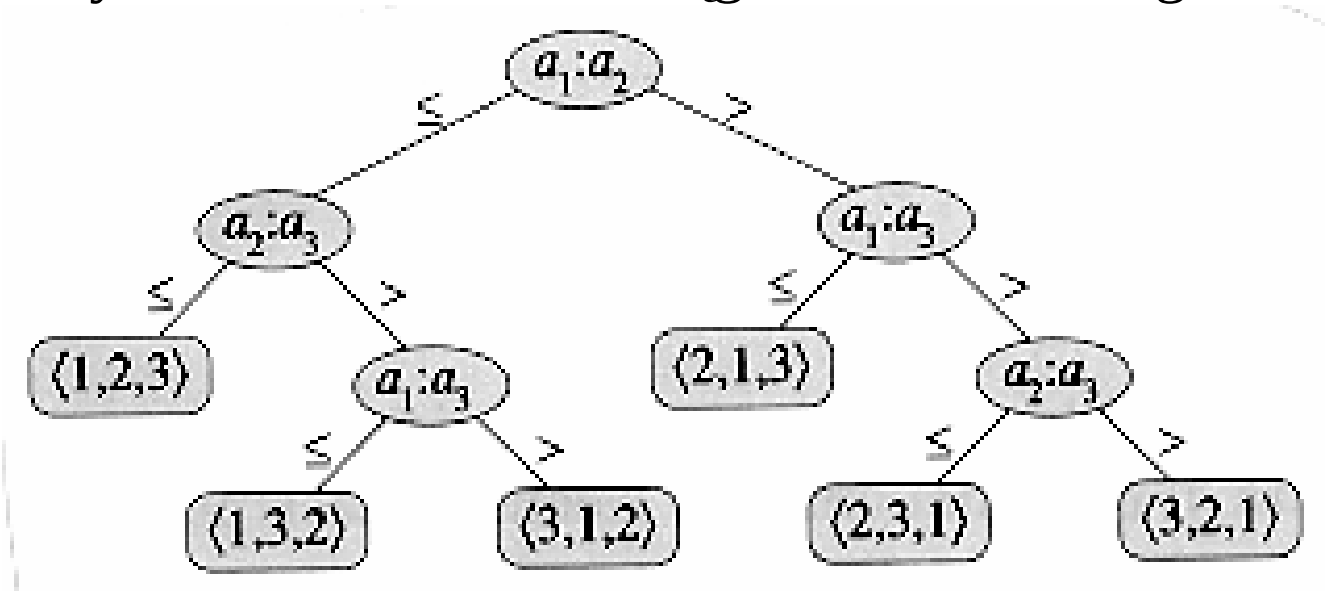
Dolne ograniczenie sortowania za pomocą porównań

Mamy dane elementy ciągu: $\{a_1, a_2, \ldots, a_n\}$. Dla danych elementów a_i i a_j możemy wykonać jedno z porównań, aby wyznaczyć ich wzajemne położenie: $a_i<a_j$, $a_i \leq a_j$, $a_i=a_j$, $a_i \geq a_j$, $a_i>a_j$. Ogólnie można przyjąć, że wystarczy wykonywać porównania typu $a_i \leq a_j$.

Drzewa decyzyjne

Przedstawia graficznie porównania wykonywane przez algorytm sortujący dla danych ustalonego rozmiaru.

Przykład dla sortowania ciągu 3-elementowego:



Wykonanie algorytmu sortującego odpowiada przejściu od korzenia do jednego z liści (których jest $n!$). Porównania elementów następują w węzłach.

Długość najdłuższej ścieżki od korzenia do dowolnego liścia odpowiada pesymistycznej liczbie porównań wykonanych przez algorytm sortujący. Zatem dolne ograniczenie na wysokość drzew decyzyjnych stanowi dolne ograniczenie na czas działania dowolnego algorytmu sortującego za pomocą porównań.

Twierdzenie

Każde drzewo decyzyjne, odpowiadające (poprawnemu) algorytmowi sortującemu n elementów, ma wysokość $\Omega(n \log n)$.

Dowód

Rozważmy drzewo decyzyjne o wysokości h, odpowiadające algorytmowi sortowania n elementów. Liczba permutacji n elementów wynosi $n!$ a każda permutacja odpowiada jednoznacznie pewnemu uporządkowaniu elementów. Drzewo musi mieć więc co najmniej $n!$ liści. Drzewo o wysokości h nie może mieć jednak więcej niż 2^h liści. Zatem:

$n! \leq 2^h$

$h \geq \log(n!)$ (funkcja logarytmiczna jest rosnąca)

stosujemy oszacowanie: $n! \geq (n/e)^n$ (gdzie $e=2.71828...$)

$h \geq \log(n/e)^n = n \log n - n \log e = \Omega(n \log n)$.

Wniosek:
Sortowanie przez kopcowanie, scalanie oraz quicksort są asymptotycznie optymalnymi algorytmami sortującymi za pomocą porównań.

Sortowanie przez zliczanie

– omówione na ćwiczeniach (w nieco innej wersji; podobnej do sortowania kubełkowego)

Założenie: każdy z n sortowanych elementów jest liczbą całkowitą z przedziału od 1 do k dla pewnego ustalonego k.

Idea: dla każdego elementu wejściowego x należy wyznaczyć ile elementów jest mniejszych od x. Znając te liczbę, znamy jednocześnie dokładną pozycję liczby x w ciągu posortowanym.

Przykład: jeżeli od x jest mniejszych 17 elementów, to x powinien się znaleźć na miejscu 18 w ciągu posortowanym (przy założeniu, że elementy nie mogą się powtarzać).

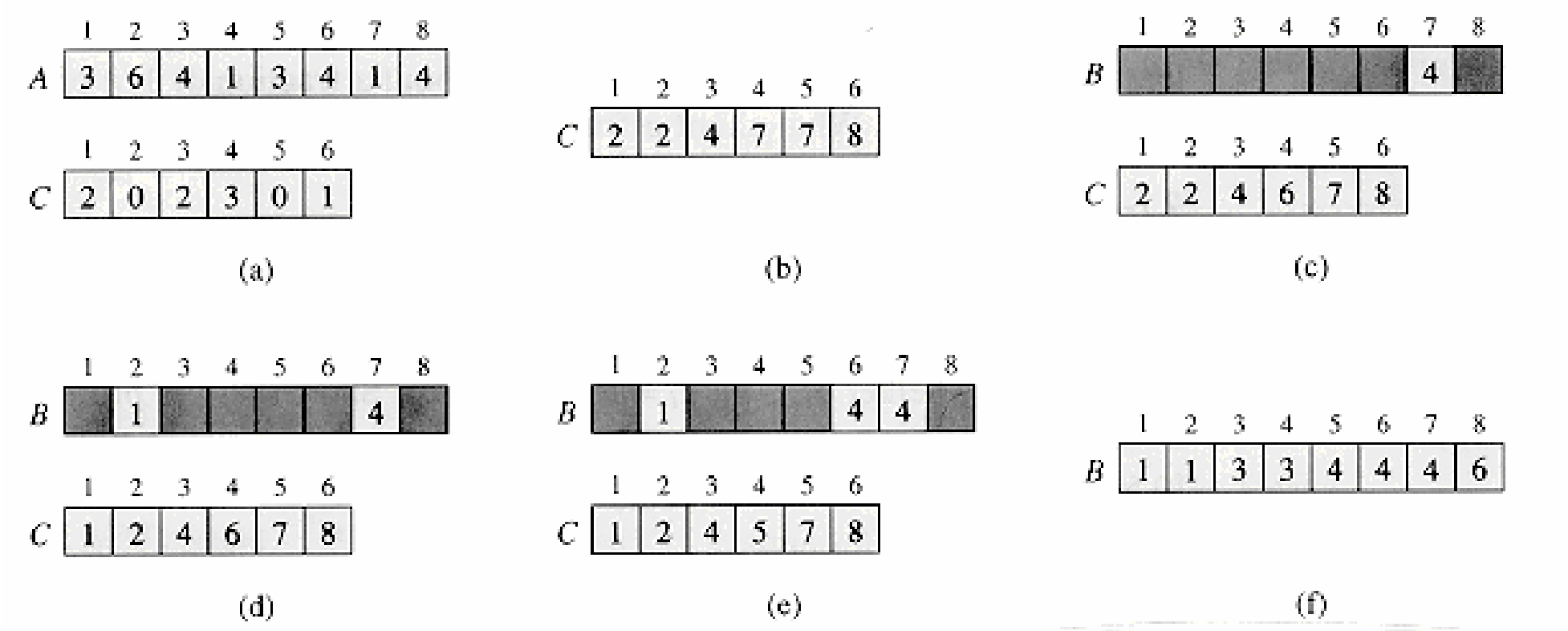
Implementacja

Tablica A – elementy wejściowe; B – elementy wyjściowe (posortowane); C – dane pomocnicze

```
Counting-Sort(A, B, k):
   pętla od i=1 do k
      C[i]=0
   pętla od j=1 do n
      C[A[j]] = C[A[j]]+1
   // C[i] zawiera teraz liczbę elementów równych i
   pętla od i=2 do k
      C[i] = C[i] + C[i-1]
   // C[i] zawiera liczbę elementów mniejszych lub równych i
   pętla od j=n do 1
      B[C[A[i]]] = A[i]
      C[A[i]] = C[A[i]]-1
```

Pętla 1: inicjalizacja. **Pętla 2:** zliczenie elementów równych indeksowi tablicy. **Pętla 3:** zliczenie elementów mniejszych lub równych indeksowi tablicy. **Pętla 4:** zapisanie wyników do wyjściowej tablicy; zmniejszenie zawartości tablicy C w celu uniknięcia konfliktu przy powtarzających się liczbach.

Ilustracja działania procedury



a) stan po wykonaniu drugiej pętli,
b) stan po wykonaniu 3 pętli,
c) d) e) stan po wykonaniu odpowiednio 1, 2, 3 iteracji w czwartej pętli,
f) ostateczne zawartość tablicy wyjściowej.

Czas działania

W algorytmie nie występują porównania elementów, zatem nie ma tu zastosowania wniosek dotyczący dolnego ograniczenia dla metod sortowania przez porównanie.

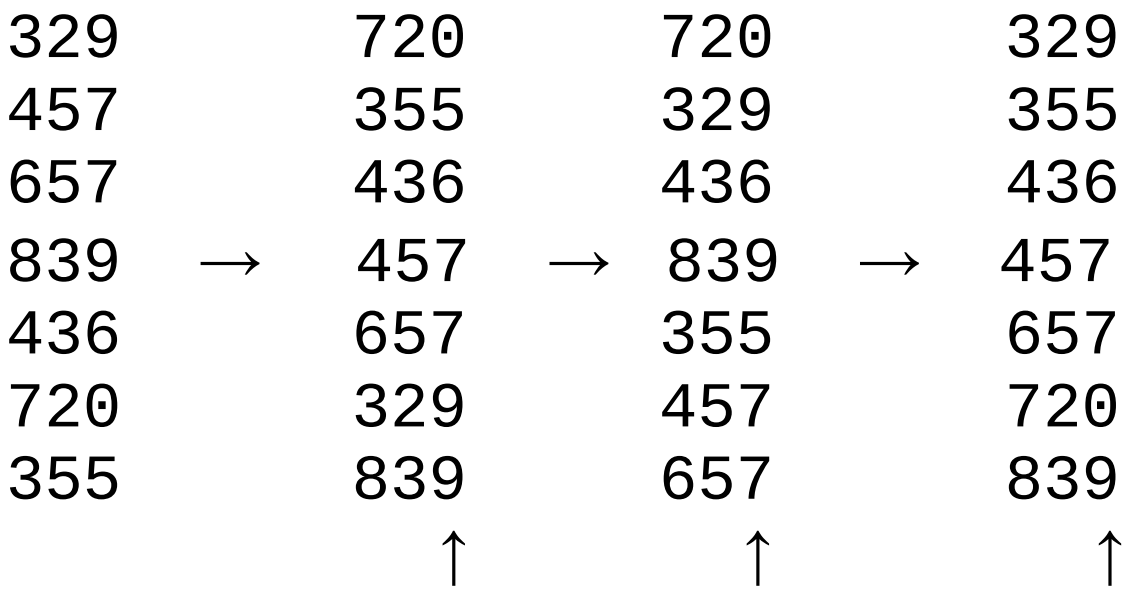
Pierwsza pętla: k wykonań,
Druga pętla: n wykonań,
Trzecia pętla: k wykonań,
Czwarta pętla: n wykonań,

Razem liczba operacji: $O(n+k)$. W praktyce najczęściej $k=O(n)$, zatem czas działania procedury wynosi w takim przypadku $O(n)$ – mniej niż czas $\Omega(n \log n)$.

Algorytm jest stabilny (nie zmienia kolejności takich samych liczb w tablicy wynikowej).

Sortowanie pozycyjne

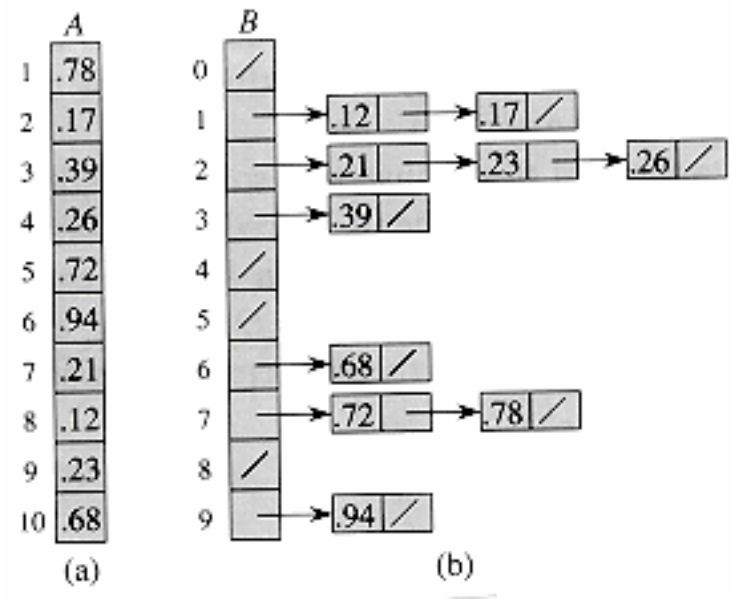
– szczegółowo omówione na ćwiczeniach
– polega na sortowaniu liczby według najmniej znaczącej cyfry, proces jest powtarzany dla wszystkich cyfr



– dla pewnych długości elementów do posortowania oraz ich liczby, algorytm działa w czasie liniowym (dużo też zależy od wyboru algorytmu sortującego wg kolejnej cyfry – najczęściej stosuje się zliczanie)

Sortowanie kubełkowe

– szczególny przypadek (dla liczb całkowitych) omówiony na ćwiczeniach
– ogólnie polega na utworzeniu „kubełków” – pojemników, w których są przechowywane liczby przeznaczone do posortowania
– kubełki tworzymy poprzez podzielenie przedziału do jakiego należą sortowane liczby na szereg podprzedziałów
– liczby wrzucamy do odpowiedniego kubełka, sortujemy w każdym z nich i wypisujemy od kubełka pierwszego do ostatniego



a) tablica do posortowania
b) 10 kubełków i liczby, które do nich należą; liczby (po posortowaniu np. przez wstawianie) są wyświetlane od kubełka o najniższym numerze (`B[0]`) do tego o najwyższym. Operacja ta wykonywana jest w czasie liniowym $O(n)$.