Sortowanie przez porównywanie

Działanie poznanych dotychczas algorytmów sortowania oparte było na porównywaniu elementów. W omawianych przykładach sortowanymi elementami były liczby, ale w ogólności mogły to być dowolne elementy, które daje się porównać: napisy tekstowe sortowane alfabetycznie, dane o miastach sortowane liczbą ludności, dane o statkach sortowane wypornością, itp. Najlepsze poznane algorytmy sortowania przez porównywanie działają w czasie $\Theta(n \log n)$, i można udowodnić, że dowolny algorytm sortujący przez porównywanie ma czas działania w najgorszym przypadku $\Omega(n \log n)$.

Jednak w pewnych przypadkach można uzyskać lepsze czasy sortowania. Dzieje się tak w sytuacjach kiedy nie ma potrzeby porównywania ze sobą wszystkich elementów. W tych sytuacjach można osiągnąć sortowanie w czasie liniowym $\Theta(n)$.

Typowa sytuacją tego typu zachodzi na przykład, gdy sortowaniu podlega zbiór liczb z określonego, znanego przedziału. Ich posortowanie nie wymaga porównywania każdej wartości z każdą, ponieważ sprawdzenie pojedynczego elementu dostarcza już dość konkretnej informacji o jej położeniu w docelowej sekwencji.

Sortowanie przez zliczanie

Algorytm **sortowania przez zliczanie** (*counting sort*) zakłada, że elementy do posortowania są liczbami całkowitymi w przedziale $[0..k]$. Algorytm działa w czasie $\Theta(n+k)$, a więc gdy k jest ograniczoną liczbą, nieprzekraczającą $O(n)$ to algorytm działa w czasie $\Theta(n)$.

Algorytm najpierw oblicza, dla każdego elementu x ciągu wejściowego, ile elementów nie przekracza wartości x. Na tej podstawie wyznaczana jest pozycja każdego elementu w docelowej sekwencji.

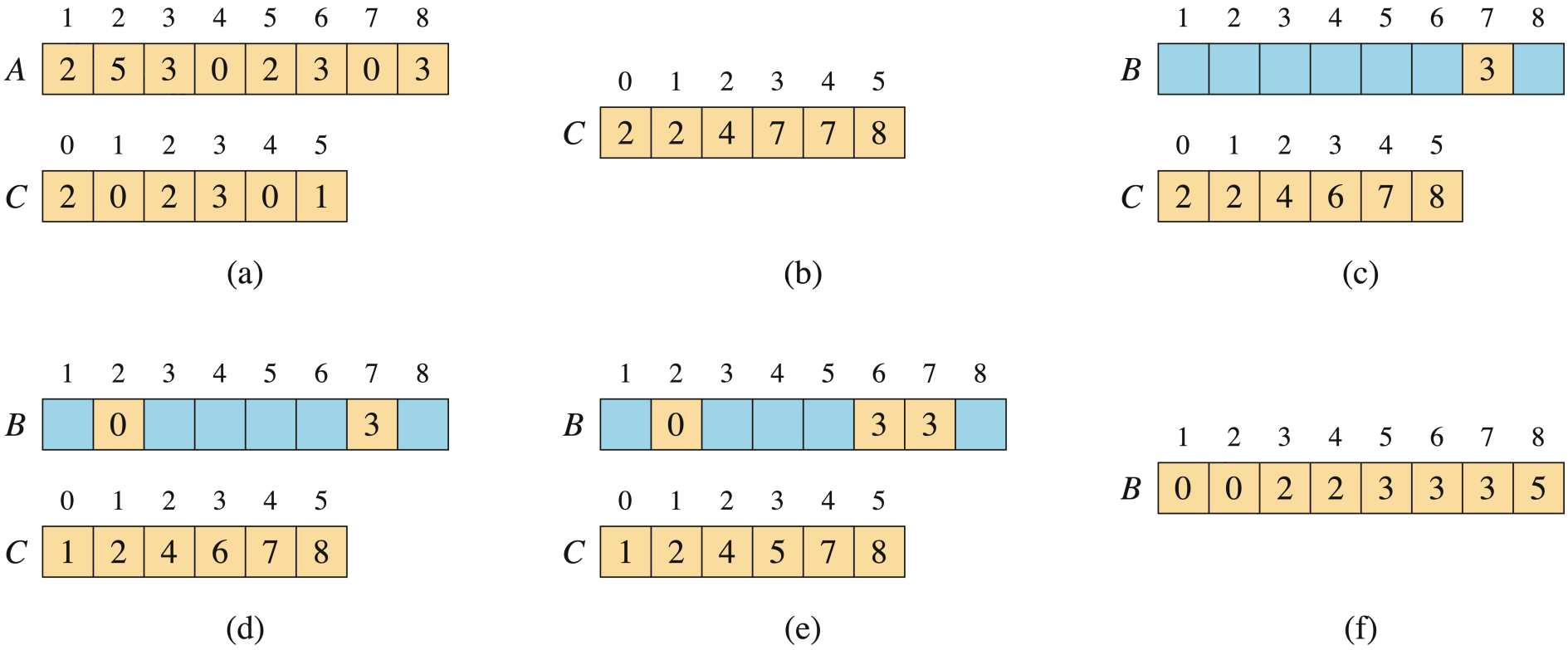
Na przykład, jeśli wiemy, że 17 elementów jest mniejsze lub równe wartości x, to x powinno znaleźć się na pozycji 17-tej.

Ta idea działania algorytmu wymaga pewnego uzupełnienia dla uwzględnienia przypadku, że wiele elementów może mieć tę samą wartość. Te szczegóły ilustruje pseudokod algorytmu.

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3        C[i] = 0
 4  for j = 1 to n
 5        C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8        C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12        B[C[A[j]]] = A[j]
13        C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

W kroku 12 algorytm umieszcza wartość $A[j]$ na swoim właściwym miejscu w tablicy B zgodnie z ideą z poprzedniej strony. Następnie w kroku 13 dekrementuje ten numer pozycji, aby następna instancja wartości $A[j]$, jeśli taka istnieje w tablicy A, znalazła się na wcześniejszej pozycji w B.

Przykład wykonania algorytmu COUNTING-SORT: (a) tablica A i zawartość tablicy C po wykonaniu kroków 4-5, (b) zawartość tablicy C po wykonaniu kroków 7-8, (c),(d),(e) zawartości tablic B i C po jedno-, dwu-, i trzykrotnym wykonaniu pętli w krokach 11–13, (f) zawartość tablicy B po zakończeniu pracy algorytmu.



Ćwiczenie: przedstaw zawartość tablic B i C po dalszych wykonaniach pętli w krokach 11-13.

Własności sortowania przez zliczanie

Jak powiedzieliśmy na początku, algorytm COUNTING-SORT działa w czasie $\Theta(n+k)$, co widać w zapisie algorytmu, który wykonuje dwie pętle n razy i dwie pętle k razy. W praktycznych przypadkach wartość k jest często ograniczona, a więc przyjmując $k = O(n)$ czas działania wynosi $\Theta(n)$.

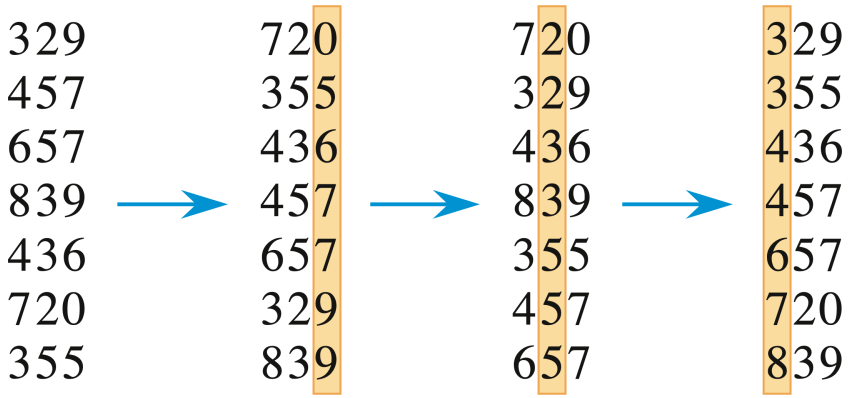
Ten liniowy czas działania wynika z braku porównań. Algorytm nie musi wykonywać wielu operacji dla każdego elementu sekwencji wejściowej, a więc może działać w czasie liniowym.

COUNTING-SORT jest również algorytmem **stabilnym**, to znaczy, równe elementy ciągu wejściowego znajdą się w ciągu wyjściowym w tej samej kolejności. Ta własność może mieć znaczenie dla niektórych zastosowań, i zobaczymy przykład wykorzystania tej własności algorytmu jako procedury użytej w kolejnym algorytmie sortowania (sortowaniu pozycyjnym).

Sortowanie pozycyjne

Rozważmy zadanie sortowania zbioru trzycyfrowych liczb jak na rysunku po lewej. Moglibyśmy wykorzystać ideę sortowania zbioru liczb o małym zakresie wartości aby sortować te liczby kolejno po cyfrach w kolejnych pozycjach. (Zakres wartości cyfr na kolejnych pozycjach jest $0..9$.)

Jednak po posortowaniu pierwszej, najbardziej znaczącej, cyfry, otrzymamy dziesięć podzbiorów do dalszego posortowania, a po posortowaniu po drugiej cyfrze, już sto oddzielnych podzbiorów do sortowania, i tak dalej.



Zamiast tego, algorytm **sortowania pozycyjnego** sortuje cyfry w kolejności od najmniej do najbardziej znaczącej pozycji, a więc jakby trochę nielogicznie.

Gdy jednak do tego sortowania wykorzystamy stabilny algorytm sortowania, to po przesortowaniu zbioru po najmniej znaczącej cyfrze, sortowanie po drugiej z kolei przemiesza tylko wartości różniące się drugą cyfrą, ale te o równej drugiej cyfrze pozostaną w już ustalonej kolejności.

Zapis algorytmu realizującego tę ideę jest dość trywialny:

```
RADIX-SORT(A, n, d)
1  for i = 1 to d
2        use a stable sort to sort array A[1 : n] on digit i
```

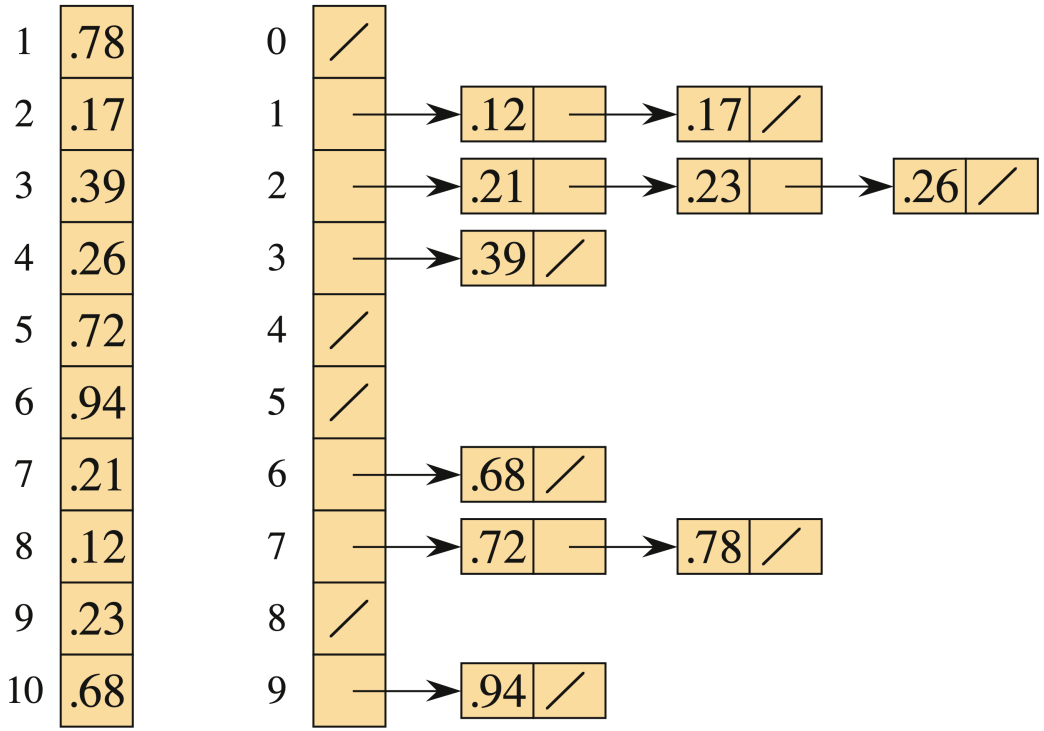
Jeśli w roli wewnętrznej procedury sortowania stabilnego zostanie użyte sortowanie przez zliczanie, to można dokonać dodatkowej optymalizacji współpracy tych algorytmów. W czasie pracy COUNTING-SORT alokuje dwie tablice robocze: $B[1 : n]$ i $C[0 : k]$. Tablica C może być prealokowana na wszystkie kolejne wywołania sortowania stabilnego, aczkolwiek ta tablica jest normalnie mniejsza, oraz i tak wymaga wyzerowania przy każdym wywołaniu, więc ta prealokacja zaoszczędzi niewiele.

Jednak tablica B jest z założenia większa, i jest tworzona dla przechowania kolejno otrzymywanych wyników. Zamiast więc pozwolić procedurze COUNTING-SORT każdorazowo tworzyć tablicę wynikową B i w każdym kolejnym wywołaniu przekopiowywać wyniki B z poprzedniego wywołania do tablicy A w nowym wywołaniu, można prealokować obie tablice A i B w procedurze RADIX-SORT, i w kolejnych wywołaniach COUNTING-SORT zamieniać tablice wejściową i wyjściową.

Sortowanie kubełkowe

Sortowanie kubełkowe zakłada, że wartości do sortowania są liczbami losowo wybranymi z pewnego przedziału w rozkładzie równomiernym. Dla uproszczenia można założyć, że będzie to przedział $[0..1)$ (wartość 1 jest wyłączona).

Algorytm rozdziela liczby ze zbioru wejściowego do n równych przedziałów, zwanych **kubełkami** (*buckets*). Gdyby liczby były rzeczywiście równomiernie rozłożone, to do każdego kubełka trafiłaby dokładnie jedna liczba. Jednak przy losowym wyborze sytuacja może być taka jak na przykładowym rysunku:



Po podziale, wszystkie kubełki są indywidualnie sortowane dowolnym algorytmem porównującym (np. INSERT-SORT), i ich zawartości są kolejno wysyłane na wyjście.

Pseudokod algorytmu:

```
BUCKET-SORT(A, n)
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3        make B[i] an empty list
4  for i = 1 to n
5        insert [A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7        sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```

Jest niemal oczywiste, że w najgorszym wypadku algorytm działa w czasie $\Theta(n^2)$. Może się bowiem zdarzyć, że wszystkie liczby trafią do jednego kubełka, i potrzeba będzie $\Omega(n^2)$ czasu by je posortować.

Jednak interesujący jest przypadek średni. Można udowodnić, że przy losowym, równomiernym rozkładzie liczb, wartość oczekiwana czasu pracy algorytmu jest $\Theta(n)$.

Ponieważ sortowanie przez wstawianie jest stabilne, sortowanie kubełkowe jest również.

Krótkie podsumowanie — pytania sprawdzające

1. Wzorując się na rysunkach z przykładu na stronie 5, przedstaw kolejne kroki wykonania algorytmu COUNTING-SORT na tablicy $A = \{6, 0, 2, 0, 1, 3, 4, 6, 3, 2\}$.

2. Wzorując się na rysunkach z przykładu na stronie 7, przedstaw kolejne etapy wykonania algorytmu RADIX-SORT na ciągu słów: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

3. Wzorując się na rysunkach z przykładu na stronie 9, przedstaw działanie algorytmu BUCKET-SORT na tablicy
$A = \{0.79, 0.13, 0.16, 0.64, 0.39, 0.20, 0.89, 0.53, 0.71, 0.42\}$.

Literatura i materiały pomocnicze

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, Clifford Stein: Wprowadzenie do algorytmów, PWN, 2024, rozdział 8.