Skrypt do Algorytmów i Struktur Danych

K. Kleczkowski M. Pietrek

14 marca 2018

Spis treści

Część I

Algorytmy

Rozdział 1

Algorytmy sortowania

> Jeśli coś jest głupie i działa, to nie jest głupie.
>
> autor nieznany

Uważa się, że jednym z najbardziej fundamentalnych problemów leżących u podstaw informatyki jest problem sortowania. W tym rozdziale zostaną przedstawione jedne z najbardziej popularnych algorytmów sortowania. Zanim jednak podejmiemy się prób opisu tych algorytmów, przyda się teoretyczne wprowadzenie.

1.1. Wprowadzenie

Wprowadźmy ważne definicje.

Definicja 1.1.1 (Tablica)**.** Niech D będzie niepustym zbiorem. Tablicą n-elementową ($n \in \mathbb{N}$) o elementach ze zbioru D nazywamy skończony ciąg $A = (a_1, a_2, a_3, \ldots, a_{n-1}, a_n) \in D^n$.

Uwaga. W przypadku podciągu $(a_i, a_{i+1}, a_{i+2}, \ldots, a_{j-1}, a_j)$ stosujemy oznaczenie $A\,[a \,..\, b]$, przy czym $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n$ oraz $i \leqslant j$. W szczególności $A\,[1 \,..\, n]$ oznacza całą tablicę n-elementową. Stosować będziemy również oznaczenie na i-ty element tej tablicy jako $A\,[i]$.

Mając zdefiniowaną tablicę możemy sformułować następujący problem.

Problem 1.1.2 (Problem sortowania)**.** Niech D będzie niepustym zbiorem i $A\,[1 \,..\, n]$ będzie tablicą n-elementową ($n \in \mathbb{N}$) o elementach ze zbioru D. Niech porządek $\leqslant$ będzie porządkiem liniowym na elementach ciągu A. Należy znaleźć taką permutację $\sigma : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$, że $a_{\sigma(1)} \leqslant a_{\sigma(2)} \leqslant a_{\sigma(3)} \leqslant \ldots \leqslant a_{\sigma(n-1)} \leqslant a_{\sigma(n)}$.

Uwaga. W szczególności będziemy mówić krótko, że n-elementowa tablica A jest posortowana wtedy, gdy $a_1 \leqslant a_2 \leqslant a_3 \leqslant \ldots \leqslant a_{n-1} \leqslant a_n$.

Szeroką klasą algorytmów sortowania rozwiązujący problem sortowania są algorytmy, które wykorzystują pewien porządek liniowy $\leqslant$, by uzyskać żądaną permutację. Jednym z nich, który stosunkowo jest prosty w implementacji jest sortowanie przez wstawianie.

1.2. Sortowanie przez wstawianie

Sortowanie przez wstawianie intuicyjnie odbywa się w taki sposób, w jaki ludzie układają karty. W każdym kroku układania tych kart zostaje wstawiona karta do podtablicy już posortowanych kart. Jak przekonamy się później, ten algorytm łatwo zaimplementować.

1.2.1. Algorytm wstawiania

Omówmy sobie algorytm, który odpowiada za działanie InsertionSorta. Jest to algorytm wstawiania do posegregowanej podtablicy $A[1 \,..\, i]$ i-tego elementu tablicy $A[1 \,..\, n]$, który będziemy określać mianem Insert. Udowodnijmy ważny niezmiennik.

Algorytm 1.2.1 Wstawianie do posortowanej podtablicy

Require: The array $A[1 \,..\, i-1]$ is already sorted.
Ensure: The array $A[1 \,..\, i]$ is already sorted.

```
1: procedure INSERT(A[1 .. i])
2:     let key ← A[i]
3:     let j ← i − 1
4:     while j ⩾ 1 ∧ A[j] > key do
5:         A[j + 1] ← A[j]
6:         j ← j − 1
7:     end while
8:     A[j + 1] ← key
9: end procedure
```

Niezmiennik 1.2.1. *Dla każdej iteracji pętli (w linii 4)* $0 \leqslant j_{min} \leqslant j \leqslant i-1$ *podtablica* $A[j+1 \,..\, i]$ *zawiera elementy większe od klucza, przy czym* $j_{min} = \min\{k : A[k] > key\}$.

Dowód. Indukcja względem iteracji. Załóżmy, że $j_{\min} \leqslant i-1$. W wypadku, gdy $j_{\min} > i-1$ klucz jest wstawiony trywialnie, bowiem jest on elementem maksymalnym w rozpatrywanej podtablicy. Sprawdźmy tezę dla $j = i-1$. Ponieważ $j_{\min} \leqslant j$ oraz tablica $A[1 \,..\, i-1]$ jest posortowana, to $A[j_{\min}] >$ key i następnie $A[j_{\min}] < A[i-1]$, czyli $A[i-1] >$ key. Zatem po wykonaniu tego kroku mamy, że $A[i] \leftarrow A[i-1]$, czyli otrzymujemy, że $A[i] >$ key, innymi słowy, tablica $A[i \,..\, i]$ spełnia tezę.

Załóżmy teraz, że dla pewnego k takiego, że $j_{\min} \leqslant k \leqslant i-1$, podtablica $A[k+1 \,..\, i]$ jest posortowana i zawiera elementy większe od klucza. Niech $k' = k-1$ takie, że $j_{\min} \leqslant k'$. Ponieważ $j_{\min} \leqslant k'$ oraz tablica $A[1 \,..\, i]$ jest posortowana, to $A[j_{\min}] >$ key i następnie $A[j_{\min}] < A[k']$, czyli $A[k'] >$ key. Wobec tego tablica zostanie przesunięta o jeden element, to znaczy, $A = (a_1, a_2, \ldots, a_{j_{\min}}, a_{j_{\min}+1}, \ldots, a_{k'-1}, a_{k'}, a_{k'}, a_{k'+1}, \ldots, a_i, a_{i+1}, \ldots, a_{n-1}, a_n)$. Z założenia indukcyjnego podtablica $A[k+1 \,..\, i]$ ma elementy większe od klucza. Ponieważ $A[k] >$ key, to $A[k \,..\, i] = A[k'+1 \,..\, i]$ ma elementy większe od klucza, co należało pokazać. □

1.2.2. Algorytm sortowania przez wstawianie

Mając udowodniony algorytm wstawiania, możemy w prosty sposób ułożyć algorytm sortowania przez wstawianie.

Algorytm 1.2.2 Sortowanie przez wstawianie

Require: $n \in \{k \in \mathbb{N} : k \geqslant 1\}$
Ensure: The array $A[1 \,..\, n]$ is already sorted.
1: **procedure** InsertionSort($A[1 \,..\, n]$)
2: **for** $i \leftarrow 2 \,..\, n$ **do** Insert($A[1 \,..\, i]$)
3: **end for**
4: **end procedure**

Niezmiennik 1.2.2. *Dla każdego kroku $2 \leqslant i \leqslant n$, tablica $A[1 \,..\, i]$ jest posortowana.*

Dowód. Dzięki niezmiennikowi 1.2.1, po wykonaniu algorytmu Insert otrzymujemy, iż $j = j_{\min} - 1$ oraz podtablica $A[j_{\min} + 1 \,..\, i]$ zawiera elementy większe od klucza. Ponieważ tablica $A[1 \,..\, i-1]$ jest posortowana, to $A[j_{\min} - 1] \leqslant A[j_{\min}] \leqslant \text{key}$ i stąd wstawiamy $A[j+1] \leftarrow \text{key}$. Mamy wobec tego, że tablica $A[1 \,..\, i]$ jest posortowana, ponieważ $A[j_{\min} - 1] \leqslant \text{key} \leqslant A[j_{\min} + 1]$. □

Dzięki temu algorytm sortowania przez wstawianie sprawia, że tablica $A[1 \,..\, n]$ jest posortowana.

1.2.3. Analiza złożoności sortowania przez wstawianie

Wykonajmy analizę złożoności pesymistycznej.

Fakt 1.2.3. *Czas działania algorytmu 1.2.1 jest rzędu $\mathcal{O}(i)$.*

Dowód. Niech tablica $A[1 \,..\, i]$ będzie posortowana odwrotnie, to znaczy $a_1 > a_2 > \ldots > a_{i-1} > a_i$. Stąd pętla (w linii 4) wykona się $(i-1) - j_{\min} + 1$ razy, przy czym $j_{\min} = 1$, czyli $i - 1$ razy. Stąd widzimy, że $i - 1 = \mathcal{O}(i)$. □

Fakt 1.2.4. *Czas działania algorytmu 1.2.2 jest rzędu $\mathcal{O}(n^2)$.*

Dowód. Niech tablica $A[1 \,..\, n]$ będzie posortowana odwrotnie, to znaczy $a_1 > a_2 > \ldots > a_{n-1} > a_n$. Stąd procedura 1.2.1 wykona się $n-1$. Stąd widzimy, że $\sum_{i=2}^{n} \mathcal{O}(i) = \mathcal{O}\left(\frac{n(n+1)}{2}\right) = \mathcal{O}(n^2)$. □

Jak można zauważyć, algorytm jest niewydajny dla dużych danych dzięki złożoności kwadratowej. Również pokażemy, że dla przypadku średniego nadal nie otrzymujemy lepszej złożoności. Zanim przejdziemy do analizy, przypomnijmy sobie z algebry następujące definicje.

Definicja 1.2.1. Permutacją π skończonego zbioru A jest bijekcja $\pi : A \to A$.

Definicja 1.2.2. Niech $(A, \leqslant)$ będzie porządkiem liniowym. Inwersją dla permutacji $\pi : A \to A$ nazywamy parę $(i, j) \in A \times A$ taką, że $i < j$ oraz $\pi(i) > \pi(j)$.

Czytelnik może się zastanawiać, czemu potrzebne są nam inwersje w analizie złożoności średniej. Należy zauważyć, że

Obserwacja 1.2.5. *Po wykonaniu algorytmu 1.2.1 liczba inwersji zmniejsza się o jeden. Algorytm 1.2.2 stąd kończy pracę, gdy liczba inwersji jest równa zero.*

Dowód tej obserwacji pozostawiamy jako ćwiczenie. Stąd można spojrzeć na inwersje istniejące w tablicy, która zostanie posortowana.

Fakt 1.2.6. *Średni czas działania algorytmu 1.2.2 wynosi* $\Theta(n^2)$.

Dowód. Niech $A = (a_1, a_2, \ldots, a_n)$ będzie tablicą oraz, π będzie permutacją taką, że $a_{\pi(1)} \leqslant a_{\pi(2)} \leqslant \ldots \leqslant a_{\pi(n)}$. Niech $I_{i,j}$ będzie zmienną losową, która jest zadana wzorem.

$$I_{i,j} = \begin{cases} 1 & i < j \wedge \pi(i) > \pi(j) \\ 0 & \text{w p.w.} \end{cases}$$

Niech $I = \sum_{i<j} I_{i,j}$. Z liniowości wartości oczekiwanej mamy $\mathbb{E}[I] = \mathbb{E}\left[\sum_{i<j} I_{i,j}\right] = \sum_{i<j} \mathbb{E}[I_{i,j}]$.

Zauważmy, że jeśli permutacja π ma inwersję $i < j$, to można przyporządkować

$$(\pi(1), \ldots, \pi(i), \ldots \pi(j), \ldots, \pi(n)) \mapsto (\pi(1), \ldots, \pi(j), \ldots \pi(i), \ldots, \pi(n))$$

przy czym to przyporządkowanie jest bijekcją. Stąd otrzymujemy, że permutacji z inwersją $i < j$ jest tyle samo, co bez inwersji, czyli możemy natrafić na nią z jednakowym prawdopodobieństwem $\mathbb{P}[I_{i,j} = 1] = \frac{1}{2}$ dla dowolnych $i < j$. Oczywiście $\mathbb{E}[I_{i,j}] = \mathbb{P}[I_{i,j} = 1] = \frac{1}{2}$. Uporządkowanych par (i, j) możemy ustalić na $\binom{n}{2}$ sposobów, ponieważ wystarczy wybrać podzbiór dwuelementowy i z tego, że mamy liniowy porządek, możemy utworzyć ciąg rosnący. Wobec tego mamy, że $\mathbb{E}[I] = \binom{n}{2} \cdot \frac{1}{2} = \frac{n(n-1)}{4} = \Theta(n^2)$. □

To, że algorytm ma złożoność kwadratową, nie oznacza, że jest nieprzydatny. Biegły Czytelnik zawuaży, że złożoność pamięciowa tego algorytmu jest rzędu $\mathcal{O}(1)$, czyli jest tani w kwestii użycia pamięci. Zatem algorytm nadaje się do małych próbek danych oraz w technikach hybrydowych.

1.3. Sortowanie przez scalanie

Sortowanie przez scalanie zostało opracowane przez Johna von Neumanna w roku 1945 [1]. Jest jednym z reprezentatywnych przykładów algorytmów opartych o metodę *dziel i zwyciężaj*.

1.3.1. Algorytm scalania

Opiszemy tutaj algorytm scalania, który odpowiada za frazę *zwyciężaj* w naszej metodzie. Jest to algorytm, który nie jest algorytmem w miejscu, to znaczy, potrzebuje osobnej pamięci, by wykonać scalanie.

Niezmiennik 1.3.1. *Dla każdego kroku pętli (w linii 4)* $1 \leqslant k \leqslant \min\{n, m\}$ *tablica* $C[1 \,..\, k]$ *jest posortowana i zawiera elementy tablic* $A[1 \,..\, i]$ *lub* $A[1 \,..\, j]$.

Dowód. Indukcja względem iteracji pętli. Sprawdźmy dla $k = 1$. Mamy wobec tego po wykonaniu ciała $C[1] = \min\{A[1], B[1]\}$, stąd trywialnie mamy spełnioną tezę.

Załóżmy teraz, że dla pewnego l takiego, że $1 \leqslant l \leqslant \min\{n, m\}$, tablica $C[1 \,..\, l]$ jest posortowana i zawiera elementy tablic $A[1 \,..\, i]$ lub $A[1 \,..\, j]$, gdzie $1 \leqslant i \leqslant n$ oraz $1 \leqslant j \leqslant m$ są ustalone.

Algorytm 1.3.1 Złączanie dwóch tablic

Require: Arrays $A[1\,..\,n]$ and $B[1\,..\,m]$ are already sorted.
Ensure: Returned array is already sorted and consists of A and B.

```
1:  procedure MERGE(A[1 .. n], B[1 .. m])
2:      let i ← 1, j ← 1, k ← 1
3:      let C[1 .. n + m]
4:      while i ⩽ n ∧ j ⩽ m do
5:          if A[i] ⩽ B[j] then
6:              C[k] ← A[i]
7:              i ← i + 1
8:              k ← k + 1
9:          else
10:             C[k] ← A[j]
11:             j ← j + 1
12:             k ← k + 1
13:         end if
14:     end while
15:     while i ⩽ n do
16:         C[k] ← A[i]
17:         i ← i + 1
18:         k ← k + 1
19:     end while
20:     while j ⩽ m do
21:         C[k] ← A[j]
22:         j ← j + 1
23:         k ← k + 1
24:     end while
25:     return C
26: end procedure
```

W kolejnym kroku $l' = l + 1 \leqslant \min\{n, m\}$ zostaje dodany element $C[l'] = \min\{A[i], B[j]\}$. Pokażemy teraz, że $C[l] \leqslant C[l']$. Ponieważ tablica C z założenia indukcyjnego jest posortowana, to $C[l]$ jest maksymalnym elementem tablicy $C[1 \,..\, l]$. Ponieważ kresem górnym tablicy $A[1 \,..\, i-1]$ jest $A[i]$ oraz podobnie dla $B[1 \,..\, j-1]$ jest $B[j]$, to otrzymujemy, że $C[l] \leqslant A[i]$ lub $C[l] \leqslant B[j]$, czyli $C[l] \leqslant C[l']$, co należało dowieść. □

Fakt 1.3.2. *Po wykonaniu algorytmu 1.3.1 spełniony jest warunek końcowy.*

Dowód. Ponieważ udowodniliśmy prawdziwość niezmiennika 1.3.1, to tablica $C[1 \,..\, \min\{n, m\}]$ zawiera elementy tablic $A[1 \,..\, \min\{n, m\}]$ lub $B[1 \,..\, \min\{n, m\}]$. Jeśli $n = \min\{n, m\}$, to należy przekopiować tablicę $B[\min\{n, m\} + 1, m]$. Istotnie, tablica B jest posortowana i mamy, że $C[\min\{n, m\}] \leqslant B[\min\{n, m\} + 1]$. Analogiczna sytuacja następuje, gdy $m = \min\{n, m\}$. □

1.3.2. Algorytm sortowania przez scalanie

Mając udowodnioną poprawność algorytmu scalania możemy sformułować sam algorytm sortowania.

Algorytm 1.3.2 Sortowanie przez scalanie

Require: $n \in \{k \in \mathbb{N} : k \geqslant 1\}$
Ensure: The array $A[1 \,..\, n]$ is already sorted.

```
1: procedure MergeSort(A[1 .. n])
2:     if n = 1 then
3:         return
4:     end if
5:     let mid ← ⌊n/2⌋
6:     MergeSort(A[1 .. mid])
7:     MergeSort(A[mid + 1 .. n])
8:     let A' ← Merge(A[1 .. mid], A[mid + 1 .. n])
9:     for i ← 1 .. n do
10:        A[i] ← A'[i]
11:    end for
12: end procedure
```

Fakt 1.3.3. *Po wykonaniu algorytmu 1.3.2 spełniony jest warunek końcowy dla każdego $n \geqslant 1$.*

Dowód. Indukcja zupełna względem wielkości tablicy. Dla $n = 1$ mamy trywialnie posortowaną tablicę. Załóżmy, że dla pewnego $k' \geqslant 1$ mamy, że dla każdego k, l takiego, że $k' \geqslant k \geqslant l \geqslant 1$ i $k \neq l$, tablica $A[l \,..\, k]$ jest posortowana przez ów algorytm. Wobec tego, w szczególności posortowane są $A[1 \,..\, \lfloor \frac{n}{2} \rfloor]$ oraz $A[\lfloor \frac{n}{2} \rfloor + 1 \,..\, k']$. Z warunku końcowego algorytmu 1.3.1 mamy, że tablica $A[1 \,..\, k']$ jest posortowana, co należało dowieść. □

1.3.3. Analiza złożoności

Przeanalizujmy czas algorytmu.

Fakt 1.3.4. *Czas wykonywania algorytmu 1.3.1 ze względu na częstość zapisu do tablicy C jest rzędu $\Theta(n + m)$.*

Dowód. Należy zauważyć, że pętla w linii 4 wykona się dokładnie $\min\{n, m\}$ razy. Dodatkowo wykonywane są operacje kopiowania $\max\{n, m\} - \min\{n, m\}$ razy. Łączny koszt wynosi $\max\{n, m\} - \min\{n, m\} + \min\{n, m\} = \max\{n, m\} = \Theta(n + m)$ □

Fakt 1.3.5. *Czas wykonywania algorytmu 1.3.2 jest rzędu* $\Theta(n \log n)$.

Dowód. Mamy następującą funkcję czasu

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases} \tag{1.3.1}$$

Z twierdzenia o rekurencji uniwersalnej mamy, że $T(n) = \Theta(n \log n)$. □

Należy zauważyć, że czas średni, zarówno jak i optymistyczny jest taki sam, wynika to ze złożoności algorytmu scalania.

Bibliografia

[1] Wikipedia. *Merge sort*. URL: https://en.wikipedia.org/wiki/Merge_sort.