

## QUICKSORT

## 1 Quicksort

O algorytmie *Quicksort* wspomnieliśmy omawiając strategię dziel i zwyciężaj. Podany tam schemat algorytmu można zapisać w następujący sposób:

```

procedure quicksort( $A[1..n], p, r$ )
  if  $r - p$  jest małe then insert – sort( $A[p..r]$ )
  else choosepivot( $A, p, r$ )
         $q \leftarrow$  partition( $A, p, r$ )
        quicksort( $A, p, q$ )
        quicksort( $A, q + 1, r$ )

```

Kluczowe znaczenie dla efektywności algorytmu mają wybór *pivota*, tj. elementu dzielącego, dokonywany w procedurze *choosepivot*, oraz implementacja procedury *partition* dokonującej przestawienia elementów tablicy  $A$ .

### 1.1 Implementacja procedury partition

Zakładamy, że w momencie wywołania *partition*( $A, p, r$ ) pivot znajduje się w  $A[p]$ . Procedura przestawia elementy podtablicy  $A[p..r]$  dokonując jej podziału na dwie części: w pierwszej –  $A[p..q]$  – znajdują się elementy nie większe od pivota, w drugiej –  $A[q + 1, r]$  – elementy nie mniejsze od pivota. Granica tego podziału, wartość  $q$ , jest przekazywana jako wynik procedury.

```

procedure partition( $A[1..n], p, r$ )
   $x \leftarrow A[p]$ 
   $i \leftarrow p - 1$ 
   $j \leftarrow r + 1$ 
  while  $i < j$  do
    repeat  $j \leftarrow j - 1$  until  $A[j] \leq x$ 
    repeat  $i \leftarrow i + 1$  until  $A[i] \geq x$ 
    if  $i < j$  then zamień  $A[i]$  i  $A[j]$  miejscami
    else return  $j$ 

```

**Fakt 1** Koszt procedury *partition*( $A[1..n], p, r$ ) wynosi  $\Theta(r - p)$ .

### 1.2 Wybór pivota

Istnieje wiele metod wyboru pivota implementowanych w procedurze *quicksort*. Decydując się na którąś z nich musimy dokonać kompromisu między jakością pivota a czasem działania algorytmu. Nierozważne wybory pivotów mogą w skrajnym przypadku prowadzić do takich podziałów tablicy  $A$ , w których jedna z podtablic jest jednoelementowa, a to implikuje liniową głębokość rekursji i, w konsekwencji, kwadratowy czas działania procedury *quicksort*.

Wydawać się może, że idealnym pivotem jest mediana<sup>1</sup>, ponieważ daje zrównoważone podziały tablicy  $A$ , co ogranicza głębokość rekursji do  $\log n$ . Ponadto istnieją algorytmy wyznaczające medianę w czasie liniowym (poznamy je później), więc czas działania procedury *quicksort* wyraża się równaniem  $t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \Theta(n)$ , co daje optymalnie asymptotyczny czas  $\Theta(n \log n)$ . Problem w tym, że stała ukryta pod  $\Theta$  jest zbyt duża, by taki algorytm był praktyczny.

### 1.2.1 Prosta metoda deterministyczna

Najprostszą, dość często stosowaną, metodą jest wybór pierwszego elementu tablicy  $A[p..r]$  jako elementu dzielącego. W naszym algorytmie sprowadza się ona do pominięcia wywołania *choosepivot*( $A, p, r$ ).

Metoda ta oczywiście może prowadzić do nierównomiernych podziałów. W szczególności, czas kwadratowy jest osiągalny, gdy dane wejściowe są uporządkowane. Z drugiej strony, na losowych danych algorytm działa bardzo szybko.

### 1.2.2 Prosty wybór zrandomizowany

Jako pivot obieramy losowy element spośród elementów  $A[p..r]$ .

```

procedure choosepivot( $A[1..n], p, r$ )
   $i \leftarrow \text{random}(p, q)$ ;
  zamień  $A[p]$  i  $A[i]$  miejscami

```

Przy takim wyborze pivota również może się zdarzyć, że algorytm będzie działał w czasie kwadratowym, jednak prawdopodobieństwo takiego zdarzenia jest zanedbywalnie małe.

Zasadnicza różnica w stosunku do metody deterministycznej polega na tym, że teraz przebieg algorytmu zależy nie tylko od danych wejściowych, ale także od generatora liczb losowych (pseudolosowych). W szczególności teraz nie istnieją dane wejściowe lepsze i gorsze. Na każdym algorytm może działać jednakowo szybko i na każdym może się zdarzyć, że będzie działał w czasie kwadratowym.

### 1.2.3 Mediana z małej próbki

Często stosowaną metodą jest wybieranie jako pivota mediany z trzech losowo wybranych elementów tablicy. To prowadzi do istotnego zmniejszenia prawdopodobieństwa nierównomiernych podziałów. Ceną jest konieczność wykonania dwóch dodatkowych porównań i przede wszystkim dwóch dodatkowych wywołań generatora liczb losowych.

"Medianę z trzech" stosuje się także w wersji deterministycznej. Najczęściej wybiera się ją wówczas spośród pierwszego, środkowego i ostatniego elementu tablicy.

Eksperymentalnie stwierdzono, że zastosowanie "mediany z trzech" zamiast prostego wyboru pivota prowadzi do przyspieszenia *quicksortu* o kilka do kilkunastu procent (zależnie od zastosowanej wersji wyboru elementów i sprawności implementacyjnej przeprowadzającego eksperymenty).

Metodę tę można rozszerzać na liczniejsze próbki, jednak uzyskane zyski czasowe są znikome.

## 1.3 Oczekiwany koszt algorytmu

Założmy, że jako pivot wybierany jest z jednakowym prawdopodobieństwem dowolny element tablicy. Pokażemy, że przy tym założeniu oczekiwany koszt algorytmu *quicksort* wynosi  $\Theta(n \log n)$ . Dla uproszczenia analizy założymy ponadto, że wszystkie elementy sortowanej tablicy są różne.

Niech  $n = r - p + 1$  oznacza liczbę elementów w  $A[p..r]$  i niech

$$\text{rank}(x, A[p..r]) \stackrel{\text{df}}{=} |\{j : p \leq j \leq r \text{ i } A[j] \leq x\}|.$$

<sup>1</sup>Medianą zbioru  $S$  nazywamy taki jego element, który jest większy od dokładnie  $\lfloor |S|/2 \rfloor$  elementów zbioru  $S$ . Definicja w naturalny sposób uogólnia się na wielozbiory.

Ponieważ w momencie wywoływania procedury *partition* w  $A[p]$  znajduje się losowy element z  $A[p..r]$ , więc wówczas

$$\forall_{i=1,\dots,n} \Pr[\text{rank}(A[p], A[p..r]) = i] = \frac{1}{n}.$$

Wynik procedury *partition* w oczywisty sposób zależy od wartości  $\text{rank}(A[p], A[p..r])$ . Gdy jest ona równa  $i$  (dla  $i = 2, \dots, n$ ), wynikiem *partition* jest  $p + i - 2$ . Ponadto, gdy  $\text{rank}(A[p], A[p..r]) = 1$ , wynikiem jest  $p$ . Tak więc zmienna  $q$  z procedury *quicksort* przyjmuje wartość  $p$  z prawdopodobieństwem  $2/n$ , a każdą z pozostałych wartości (tj.  $p + 1, p + 2, \dots, r - 1$ ) z prawdopodobieństwem  $1/n$ . Stąd oczekiwany czas działania procedury *quicksort* wyraża się równaniem

$$\begin{cases} T(1) = 1 \\ T(n) = \frac{1}{n} \left[ (T(1) + T(n-1)) + \sum_{d=1}^{n-1} (T(d) + T(n-d)) \right] + \Theta(n) \end{cases}$$

Zmienna  $d = q - p + 1$  oznacza długość pierwszej z podtablic.

Ponieważ  $T(1) = \Theta(1)$  a  $T(n-1)$  w najgorszym przypadku jest równe  $\Theta(n^2)$ , więc

$$\frac{1}{n}(T(1) + T(n-1)) = O(n).$$

To pozwala nam pominąć ten składnik, ponieważ będzie on uwzględniony w ostatnim członie sumy. Tak więc:

$$T(n) = \frac{1}{n} \sum_{d=1}^{n-1} (T(d) + T(n-d)) + \Theta(n).$$

W tej sumie każdy element  $T(k)$  jest dodawany dwukrotnie (np.  $T(1)$  raz dla  $q = 1$  i raz dla  $q = n - 1$ ), więc możemy napisać:

$$T(n) = \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n) \quad (1)$$

Ponieważ mamy silne przesłanki, by przypuszczać, że rozwiązanie tego równania jest rzędu  $\Theta(n \log n)$ , ograniczymy się do sprawdzenia tego faktu. Niech

$$T(n) = \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n) \leq an \log n + b$$

dla pewnych stałych  $a, b > 0$ . Naszym zadaniem jest pokazanie, że takie stałe  $a$  i  $b$  istnieją.

Bierzemy  $b$  wystarczająco duże by  $T(1) \leq b$ . Dla  $n > 1$  mamy:

$$T(n) = \frac{2}{n} \sum_{k=1}^{n-1} (ak \log k + b) + \Theta(n) \leq \frac{2a}{n} \sum_{k=1}^{n-1} k \log k + \frac{2b}{n}(n-1) + \Theta(n)$$

Proste oszacowanie  $\sum_{k=1}^{n-1} k \log k$  przez  $\frac{1}{2}n^2 \log n$  nie prowadzi do celu, ponieważ musimy pozbyć się składnika  $\Theta(n)$ . Oszacujmy więc  $\sum_{k=1}^{n-1} k \log k$  nieco staranniej:

**Fakt 2**  $\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2$

DOWÓD. Rozbijamy sumę na dwie części:

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$$

Szacując  $\log k$  przez  $\log \frac{n}{2}$  dla  $k < \lceil \frac{n}{2} \rceil$  oraz przez  $\log n$  dla  $k \geq \lceil \frac{n}{2} \rceil$ , otrzymujemy:

$$\begin{aligned} \sum_{k=1}^{n-1} k \log k &\leq ((\log n) - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k = \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \leq \\ &\frac{1}{2}n(n-1) \log n - \frac{1}{2}\left(\frac{n}{2} - 1\right) \frac{n}{2} \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2 \end{aligned}$$

□

Teraz możemy napisać

$$\begin{aligned} \frac{2a}{n} \left( \frac{1}{2}n^2 \log n - \frac{1}{8}n^2 \right) + \frac{2b}{n}(n-1) + \Theta(n) &\leq an \log n - \frac{a}{4}n + 2b + \Theta(n) = \\ &an \log n + b + \left( \Theta(n) + b - \frac{a}{4}n \right) \end{aligned}$$

Składową  $(\Theta(n) + b - \frac{a}{4}n)$  możemy pominąć, dobierając  $a$  tak, by  $\frac{a}{4}n \geq \Theta(n) + b$ . Zauważmy, że taki dobór zależy jedynie od stałej  $b$  oraz od stałej ukrytej pod  $\Theta$ , a więc za  $a$  można przyjąć odpowiednio dużą stałą.

To kończy sprawdzenie, że  $T(n) \leq an \log n + b$  dla pewnych stałych  $a, b > 0$ .

## 1.4 Inny sposób oszacowania oczekiwanego kosztu

Rozwiązywanie równań rekurencyjnych, określających oczekiwany czas działania algorytmów zrandomizowanych, niekoniecznie należy do przyjemnych zadań. W poprzednim paragrafie udało nam się tego uniknąć, ponieważ mieliśmy silne przesłanki co do wartości rozwiązania i wystarczyło tylko naszą hipotezę zweryfikować. W tym paragrafie pokażemy, że zanim "wdepniemy" w uciążliwe obliczenia, warto nieco głębiej przeanalizować algorytm.

Czynimy trzy upraszczające (ale nie wypaczające problemu) założenia:

- wszystkie elementy tablicy  $A$  są różne,
- z rekursją w algorytmie *quicksort* schodzimy aż do momentu gdy podtablice są jednoelementowe,
- procedura *partition* umieszcza pivot na dobrej pozycji (tj. na prawo od wszystkich elementów mniejszych od niego i na lewo od elementów większych) i nie bierze on już udziału w kolejnych wywołaniach rekurencyjnych.

Niech  $y_1, \dots, y_n$  będzie ciągiem wartości tablicy  $A$  uporządkowanym rosnąco.

**Fakt 3** Przy powyższych założeniach  $\forall 1 \leq i < j \leq n$  *quicksort* porównuje elementy  $y_i$  i  $y_j$  co najwyżej jeden raz.

Określmy następujące zmienne losowe:

- $X$  = liczba porównań wykonanych przez *quicksort*,
- $\forall 1 \leq i < j \leq n$   $X_{ij} = \begin{cases} 1 & \text{jeśli } \textit{quicksort} \text{ porównał elementy } y_i \text{ i } y_j, \\ 0 & \text{w przeciwnym przypadku.} \end{cases}$

Chcemy obliczyć  $E[X]$ . Mamy

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}]$$

Wystarczy więc określić prawdopodobieństwo tego, że *quicksort* porówna elementy  $y_i$  i  $y_j$ .

**Fakt 4**

$$\forall_{1 \leq i < j \leq n} \Pr[X_{ij} = 1] = \frac{2}{j - i + 1}.$$

UZASADNIENIE. Dopóki jako pivot wybierany jest element spoza zbioru  $y_i, y_{i+1}, \dots, y_j$ , obydwa elementy,  $y_i$  i  $y_j$ , nie będą ze sobą porównywane i w kolejnych wywołaniach rekurencyjnych będą znajdować się w tej samej podtablicy. Tak więc o tym, czy dojdzie do porównania  $y_i$  i  $y_j$  decyduje to, który spośród elementów  $y_i, y_{i+1}, \dots, y_j$  jako pierwszy zostanie pivotem. Jeśli tym elementem będzie  $y_i$  lub  $y_j$ , to dojdzie do tego porównania. Jeśli będzie nim dowolny  $y_k$  (dla  $i < k < j$ ), to *partition* przedstawiająca elementy względem  $y_k$ , nie porówna tych elementów, a po tym przedstawieniu  $y_i$  i  $y_j$  znajdą się w różnych podtablicach.  $\square$

Tak więc  $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1}$ , co jak łatwo pokazać jest równe  $2n \ln n + \Theta(n)$ .

**1.5 Jeszcze inny sposób oszacowania oczekiwanego kosztu**

Jeszcze inny, bardzo elegancki i prosty, sposób szacowania oczekiwanego kosztu *quicksortu* został podany w pracy [1]. Najpierw rozważana jest zmodyfikowana wersja algorytmu, nazwana *insistent q-sort*, w której do wywołania rekurencyjnego dochodzi dopiero po wylosowaniu *dobrego pivota*, a za taki uważa się pivot, który dzieli tablicę w stosunku nie gorszym niż 1:3. Jeśli wylosowany pivot dzieli tablicę w sposób mniej zbalansowany, zapominamy o nim i ponawiamy losowanie. To zapewnia nam, że drzewo rekursji ma wysokość ograniczoną przez  $O(\log n)$ . Ponieważ prawdopodobieństwo wylosowania dobrego pivota jest równe  $1/2$ , więc oczekiwana liczba prób potrzebnych do jego wylosowania jest równa 2. Stąd oczekiwana praca wykonana między kolejnymi wywołaniami rekurencyjnymi jest liniowa względem rozmiaru podtablicy, a stąd oczekiwana praca algorytmu *insistent q-sort* jest ograniczona przez  $O(n \log n)$ .

Ta analiza jest podstawą analizy oryginalnego *quicksorta*. Wierzchołki w drzewie rekursji dzielimy na dwie grupy: wierzchołki niebieskie i wierzchołki czerwone. Wierzchołek  $v$  jest niebieski, jeśli algorytm operuje w nim na podtablicy rozmiaru większego niż  $0.75$  rozmiaru podtablicy, na której algorytm operuje w ojcu  $v$ . Pozostałe wierzchołki (w tym korzeń) są czerwone. Ponieważ żaden wierzchołek nie może mieć dwóch niebieskich synów, niebieskie wierzchołki tworzą w drzewie proste ścieżki. Co więcej, oczekiwana długość takich niebieskich ścieżek jest ograniczona przez 1. Jeśli pracę wykonaną w niebieskich ścieżkach przypiszemy czerwonym wierzchołkom, od których te ścieżki odchodzą, a następnie niebieskie ścieżki zastąpimy pojedynczymi krawędziami, otrzymamy, podobnie jak w *insistent q-sort*, drzewo o wysokości  $O(\log n)$  złożone z wierzchołków, z których każdy "wykonuje" pracę liniową względem wielkości podtablicy, na której operuje. Po detale tej analizy odsyłam do pracy [1].

**1.6 Usprawnienia algorytmu**

*Quicksort* jest dość powszechnie uważany za najszybszą (a przynajmniej jedną z najszybszych) metodę sortowania. Jego znaczenie spowodowało, że wiele wysiłku włożono w opracowanie modyfikacji, mających na celu uzyskanie jak największej efektywności. Poniżej wymieniamy kilka z nich:

- Trójpodział. W przypadku, gdy spodziewamy się, że sortowane klucze mogą się wielokrotnie powtarzać (np. gdy przestrzeń kluczy jest mała), opłacalne może być zmodyfikowanie procedury *partition* tak, by dawała podział na trzy części: elementy mniejsze od pivota, równe pivotowi i większe od pivota. Oczywiście *quicksort* jest rekurencyjnie wywoływany jedynie do pierwszej i trzeciej części. W przypadku, gdy liczba elementów równych pivotowi jest znaczna, może to przynieść istotne przyspieszenie.
- Eliminacja rekursji.
  - Tak jak w przypadku wszystkich algorytmów opartych na strategii dziel i zwyciężaj, spory zysk można otrzymać, starannie dobierając próg na rozmiar danych, poniżej którego opłaca się zastosować prosty algorytm nierekurencyjny w miejsce rekurencyjnych wywołań procedury *quicksort*.

- W wielu implementacjach *quicksortu* przeznaczonych do powszechnego użytku (np. w bibliotekach procedur) w ogóle wyeliminowano rekursję.
- Optymalizacja pętli wewnętrznej, aż do zapisania jej w języku wewnętrznym procesora.
- W zastosowaniach, w których krytycznym zasobem jest pamięć (np. w układach realizujących sortowanie hardware’owo), stosowana bywa nierekurencyjna wersja (rekursja wymaga pamięci na stos wywołań) działająca ”w miejscu”, a więc wykorzystująca co najwyżej  $O(1)$  komórek pamięci poza tymi, które zajmuje sortowany ciąg.

## Literatura

- [1] Michael L. Fredman, An intuitive and simple bounding argument for Quicksort. *Inform. Process. Lett.*, 114(2014), 137-139.