Quicksort

```
procedure quicksort(A[1..n], p, r)
    if r − p jest małe then insert − sort(A[p..r])
    else choosepivot(A, p, r)
         q ← partition(A, p, r)
         quicksort(A, p, q)
         quicksort(A, q + 1, r)
```

1. Wybierz piwot p
2. Podziel tablicę T względem piwotu **P** umieszczając mniejsze wartości od **P** po lewej a większe – po prawej. Nazywamy tą funkcję partition.

<	**P**	>

3. Rekurencyjnie wywołaj quicksort na lewej części tablicy
1. Rekurencyjnie wywołaj quicksort na prawej części tablicy

```
procedure partition(A[1..n], p, r)
    x ← A[p]
    i ← p − 1
    j ← r + 1
    while i < j do
        repeat j ← j − 1 until A[j] ≤ x
        repeat i ← i + 1 until A[i] ≥ 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* $\Theta(r - p)$.

Wybór piwota – metoda losowa
O(n^2) różnica od metody deterministycznej taka, że dodatkowym aspektem branym pod uwagę w analizie jest generator liczb pseudolosowych.

Wybór piwota – metoda deterministyczna
wybór pierwszego elementu w tablicy. O(n^2) prosta implementacyjnie i dla losowych danych bardzo szybka

Wydawać się może, że idealnym pivotem jest mediana[1], 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 Θ jest zbyt duża, by taki algorytm był praktyczny.

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

Załóżmy, 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

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

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

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

Wynik procedury *partition* w oczywisty sposób zależy od wartości $rank(A[p], A[p..r])$. Gdy jest ona równa i (dla $i = 2, \ldots, n$), wynikiem *partition* jest $p+i-2$. Ponadto, gdy $rank(A[p], A[p..r]) = 1$, wynikiem jest p. Tak więc zmienna q z procedury *quicksort* przyjmuje wartość p z prawdopodobieństwem $2/n$, a każdą z pozostałych wartości (tj. $p+1, p+2, \ldots, r-1$) 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:

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

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:

$$\sum_{k=1}^{n-1} k \log k \;\leq\; ((\log n) - 1)\sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k = \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \leq$$
$$\frac{1}{2}n(n-1)\log n - \frac{1}{2}(\frac{n}{2} - 1)\frac{n}{2} \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2$$

$\square$

Teraz możemy napisać

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

Składową $\left(\Theta(n) + b - \frac{a}{4}n\right)$ 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 Θ, 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 $y_1, \ldots, y_n$ 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

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

1.5 Jeszcze inny sposób oszacowania oczekiwanego kosztu

Jeszcze inny, bardzo elegancki i prosty, sposób szacowania oczekiwanego kosztu *quicksortu* został podany w pracy [?]. 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 [?].

1.6 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.

Napisz procedurę partition (nie musi być to wersja z wykładu, ale musi być efektywna).

Pivot jest w A[p], funkcja zwaraca granicę podziału

```
Part(A[1..n], p, r)
  x <- A[p]
  i <- p-1

  j <- r+1

    while(i<j) do
      repeat(j--) until A[j] <= x
      repeat(i++) until A[i] >= x

      if(i<j) swap(A[i], A[i])
      else return j
```

Treść

Jaką złozoność ma algorytm Quicksort, w którym pivot wybierany jest algorytmem magicznych piątek?

▼ Rozwiązanie

Dzięki algorytmowi magicznych piątek uda nam się znaleźć mediane w czasie $T(n) = T(\frac{n}{5}) + T(\frac{7}{10}n) + O(n)$.

Stąd

$$T(n) = 2T(\frac{n}{2}) + T(\frac{n}{5}) + T(\frac{7}{10}n) + O(n)$$

$$T(n) >= 3T(\frac{n}{2}) + O(n) = O(n^{log_2 3})$$

bardzo pesymistyczny scenariusz daje nam złożoność większą niż nlogn

Treść

W analizie złożoności algorytmu QuickSort zakładaliśmy, że każde dwa elementy ciągu są porównywane ze sobą nie więcej niż jeden raz.

Zapisz w pseudokodzie procedury QuickSort i Partition realizujące tę własność.

▼ Rozwiązanie

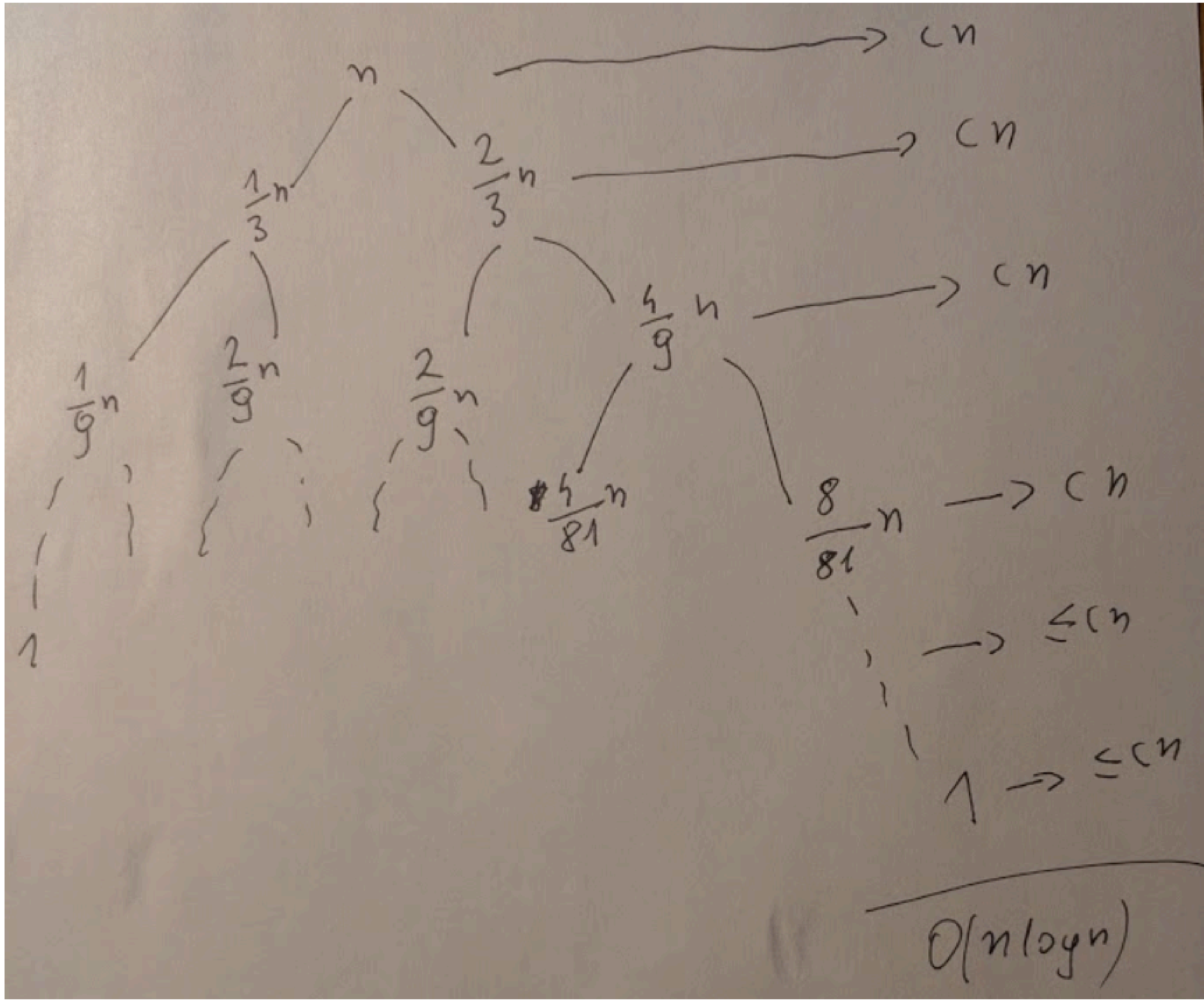
Trzeba napisać zwykłego Quicksorta i zwykłe partition.

Treść

Zauważyłeś, że Quicksort (z deterministycznym pivotem) zachowuje się zadziwiająco regularnie na ciągach z pewnej rodziny A.

Otóż okazało się, że w trakcie wszystkich wywołań rekurencyjnych proderura Partition dokonuje podziału ciągu wejściowego na podciągi o długościach nie mniejszych niż 1/3 i nie większych niż 2/3 długości ciągu wejściowego.

W jakim czasie działa Quicksort na ciągach z rodziny A?



1. (2pkt) Podaj nierekurencyjną wersję procedury *Quicksort*, która

   - działa *w miejscu*, tj. poza tablicą z danymi (**int** $A[n]$) używa tylko stałej (niezależnej od n) liczby komórek typu **int** (zakładamy, że $\max(n, \max\{A[i] \mid i = 1,..,n\})$ jest największą liczbą jaką może pomieścić taka komórka),
   - czas jej działania jest co najwyżej o stały czynnik gorszy od czasu działania wersji rekurencyjnej.

2. (Z 2pkt) Ułóż algorytm sortujący stabilnie i w miejscu ciągi rekordów o kluczach ze zbioru $\{1, 2, 3\}$.

3. (1pkt) Podaj algorytm sprawdzający izomorfizm drzew nieukorzenionych.

4. (1.5 pkt) Oszacuj oczekiwany czas działania Algorytmu Hoare'a (znajdowania mediany w ciągu). Mile widziane będzie zastosowanie metody Fredmana (z artykułu załączonego na stronie wykładu).

11. (2pkt) *Serią* w ciągu nazwiemy dowolny niemalejący podciąg kolejnych jego elementów. Seria jest *maksymalna*, jeśli nie można jej rozszerzyć o kolejne elementy. Załóżmy, że algorytm *InsertSort* uruchamiany będzie jedynie na permutacjach zbioru $\{1, 2, \ldots, n\}$, które można rozbić na co najwyżej dwie serie maksymalne. Zbadaj średnią złożoność algorytmu przy założeniu, że dla każdego n, wszystkie takie permutacje n-elementowe są jednakowo prawdopodobne.

12. (2pkt) Rozważmy permutacje liczb $\{1, 2, \ldots, n\}$, których wszystkie 2-podciągi i 3-podciągi są uporządkowane.

    (a) Ile jest takich permutacji?

    (b) Jaka jest maksymalna liczba inwersji w takiej permutacji?

    (c) Jaka jest łączna liczba inwersji w takich permutacjach?

Zadania dodatkowe - do samodzielnego rozwiązywania

1. (0pkt) Pokaż, że *Quicksort* działa w czasie $\Theta(n \log n)$, gdy wszystkie elementy tablicy A mają tę samą wartość.

2. (0pkt) Pokaż, że *Quicksort* działa w czasie $\Theta(n^2)$, gdy tablica A jest uporządkowana niemalejąco.

3. (0pkt) Załóżmy, że na każdym poziomie rekursji procedury *Quicksort* procedura *partition* dzieli daną tablicę na dwie podtablice w proporcji $1 - \alpha$ do α, gdzie $0 < \alpha eq\frac{1}{2}$ jest stałą. Pokaż, że minimalna głębokość liścia w drzewie rekursji wynosi około $-\frac{\log n}{\log \alpha}$ a maksymalna głębokość liścia wynosi około $-\frac{\log n}{\log(1-\alpha)}$.

4. (1pkt) Opracuj wersję algorytmu *Quicksort*, która będzie efektywnie działać na ciągach zawierających wielokrotne powtórzenia kluczy.

5. (2pkt) Opracuj wersję algorytmu *Mergesort*, która działa w miejscu.

6. (2pkt) Pokaż w jaki sposób można zaimplementować kolejkę priorytetową tak, by operacje na niej wykonywane były w czasie $O(\log \log m)$, gdzie m jest mocą uniwersum, z którego pochodzą klucze.

7. (2pkt) Niech $A = a_1, a_2, \ldots, a_n$ będzie ciągiem elementów oraz niech p i q będą dodatnimi liczbami naturalnymi. Rozważmy p-podciągi ciągu A, tj. podciągi utworzone przez wybranie co p-tego elementu. Posortujmy osobno każdy z tych podciągów. Powtórzmy to postępowanie dla wszystkich q−podciągów. Udowodnij, że po tym wszystkie p−podciągi pozostaną posortowane.

8. (2pkt) n-elementowym ciągiem o jednym zaburzeniu nazywamy dowolny ciąg, który może być otrzymany z ciągu $\{1, 2, \ldots, n\}$ poprzez wykonanie jednej transpozycji. Załóżmy, że algorytm *InsertSort* będzie uruchamiany jedynie na ciągach o jednym zaburzeniu. Zbadaj średnią złożoność algorytmu przy założeniu, że dla każdego n, wszystkie takie ciągi n-elementowe są jednakowo prawdopodobne.

9. (1pkt) (Poprawność procedury *Partition*). Rozważ następującą procedurę:

```
Partition(A, p, r)
x ← A[p]
i ← p − 1
j ← r + 1
 while true do
     repeat j − −
         until A[j] ⩽ x
     repeat i + +
         until A[i] ⩾ x
     if i < j
         then zamień A[i] ↔ A[j]
         else return j
```

   Udowodnij co następuje

   (a) Indeksy i oraz j nigdy nie wskazują na element A poza przedziałem $[p..r]$.

   (b) Po zakończeniu *Partition* indeks j nie jest równy r (tak więc podział jest nietrywialny).

   (c) Po zakończeniu *Partition* każdy element $A[p..j]$ jest mniejszy lub równy od dowolnego elementu $A[j + 1, r]$.

10. (1pkt) Ułóż algorytm sortujący ciąg n liczb całkowitych w czasie $O(n)$ i pamięci $O(n)$. Przyjmij, że liczby są z zakresu **long long**.