Algorytmy i Struktury Danych - rozwiązania zadań z kolokwiów

2023-11-21

1 Klasówka 2007 (1), zadanie 1

Opracuj strukturę danych, która pozwala wykonywać następujące operacje:

- Ini(k):: inicjacja struktury danych i ustalenie długości krotek liczb całkowitych na k
- Insert($< a_1, a_2, \ldots, a_k >$):: dodaje do struktury krotkę $< a_1, a_2, \ldots, a_k >$
- Min:: podaje najmniejszą leksykograficznie krotkę w strukturze
- ExtractMin:: usuwa najmniejszą leksykograficznie krotkę ze struktury

W Twoim rozwiązaniu operacje Insert i ExtractMin powinny by wykonywane w czasie $O(\log n + k)$

Rozwiązanie:

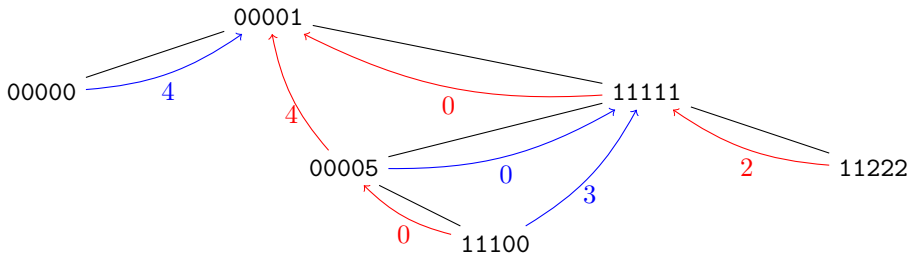
Jako bazową strukturę będziemy używać drzew Czerwono-czarnych. Dzięki temu wysokość struktury nie będzie przekraczać $O(\log n)$, oraz co później okaże się istotne liczba rotacji czy wstawianiu/usuwaniu kluczy jest rzędu $O(1)$.

W każdym węźle utrzymujemy następujące informacje:

- *key*: krotkę $< a_1, \ldots, a_k >$,
- p_{left} wskaźnik do najbliższego przodka będącego po lewej stronie,
- p_{right} wskaźnik do najbliższego przodka będącego po prawej stronie,
- lcp_{left} najdłuższy wspólny prefiks pomiędzy kluczem z węzła i kluczem z p_{left},
- lcp_{right} najdłuższy wspólny prefiks pomiędzy kluczem z węzła i kluczem z p_{right},

W jaki sposób aktualizować takie dodatkowe atrybuty?

- dodawanie nowego liścia – musimy wyznaczyć atrybuty p_{left} i p_{right} (w czasie $O(\log n)$), następnie obliczamy wartości lcp_{left} i lcp_{right} (w czasie $O(k)$),



Rysunek 1: Przykładowe drzewo dla krotek $k = 5$, wskaźniki p_{left} zaznaczone są na czerwono, wskaźniki p_{right} na niebiesko. Wartości lcp_{left} i lcp_{right} zaznaczone są jako etykiety krawędzi.

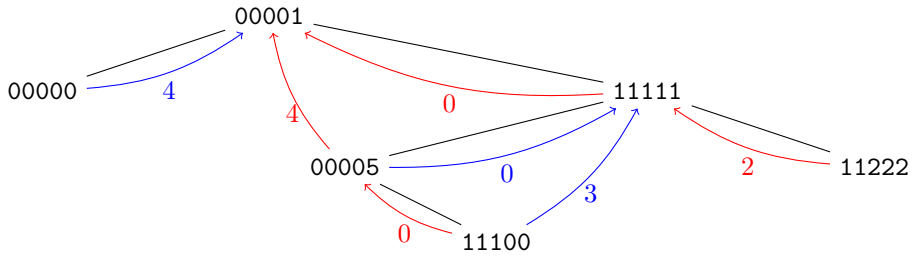
- usuwanie skrajnie prawego węzła (najmniejszy klucz w drzewie) – jeśli węzeł jest liściem to nie musimy nic zrobić, jeśli ma prawego syna (z własności drzew czerwono czarnych prawy syn musi być już liściem) to poprawiamy w nim wartości p_{left} i p_{lcp} (zmieniając je na NULL)
- rotacje – jeśli wykonujemy rotację węzłów x i $y = parent(x)$ to zauważamy, że musimy jedynie zaktualizować część atrybutów z x i y, wartości atrybutów w pozostałych węzłach zostają bez zmian. Koszt aktualizacji pojedynczego węzła wynosi $O(k)$.

Jak zrealizować operację $Insert$ w czasie $O(\log n + k)$?

Musimy pokazać, że dzięki dodatkowym atrybutom lcp uda nam się znacząco przyspieszyć porównywanie krotki z węzłami podczas przechodzenia po drzewie od korzenia do miejsce w którym nowa krotka powinna być wstawiona. Dzieje się, tak dlatego, że dzięki wartościom lcp możemy często w czasie $O(1)$ porównać wstawianą krotkę z wartością w węźle.

Algorytm 1: Insert(r, $A =< a_1, \ldots, a_k >$)

$v := r$; $lcp_l =$ `NULL`; $lcp_r =$ `NULL`
while $v \neq$ `NullNode` **do**
  niech $p_l = p_{left}(v)$, $p_r = p_{right}(v)$
  niezmiennik: $lcp_l =$ `NULL` lub $LCP(A, key(p_l)) = lcp_l \leq lcp_{left}(v)$
  niezmiennik: $lcp_r =$ `NULL` lub $LCP(A, key(p_r)) = lcp_r \leq lcp_{right}(v)$
  if $lcp_l! =$ `NULL` **and** $lcp_l < lcp_{left}(v)$ **then**
    $v = right(v)$
  else if $lcp_r! =$ `NULL` **and** $lcp_r < lcp_{right}(v)$ **then**
    $v = left(v)$
  else
    $l = max(coalesce(lcp_l, 0), coalesce(lcp_r, 0))$
    while $l < k$ **and** $a_{l+1} = key(v)[l+1]$ **do**
      $l = l + 1$
    if $l = k$ **then**
      krotka już istnieje w drzewie
    else if $a_{l+1} < key(v)[l+1]$ **then**
      $v = left(v)$; $lcp_r = l$
    else if $a_{l+1} > key(v)[l+1]$ **then**
      $v = right(v)$; $lcp_l = l$



2 Klasówka 2007 (1), zadanie 2

Udowodnij, że jeśli algorytm sortujący tablicę $A[1..n]$ porównuje i zamienia wyłącznie elementy odległe co najwyżej o 2007 (tzn. jeśli porównuje $A[i]$ z $A[j]$, to $|i - j| \leq 2007$), to jego pesymistyczny czas działania jest co najmniej kwadratowy.

Rozwiązanie: Pojedyncza zamiana usuwa $O(1)$ inwersji więc dla ciągu odwrotnie uporządkowanego (który ma $\Theta(n^2)$ inwersji) algorytm wymaga czasu $\Omega(n^2)$.

3 Klasówka 2008 (1), zadanie 2

Zaproponuj wzbogacenie kopca zupełnego w taki sposób, żeby efektywnie w czasie zamortyzowanym wykonywane były operacje: Min, DeleteMin, Insert, CountMin. Ostatnia operacja polega na podaniu aktualnej liczby elementów

w kopcu o wartości równej Min. Przeprowadź analizę kosztu zamortyzowanego wykonania poszczególnych operacji.

Rozwiązanie: Wzbogacamy węzły kopca o atrybut `countEq` oznaczającą liczbę węzłów **w poddrzewie** zawierających identyczną wartość co ten zapisany w kluczu. Uwaga! atrybut ten nie ma znaczenia globalnego (bo trudno byłoby aktualizować jego wartości) tylko lokalne i dotyczy tylko poddrzewa danego węzła.

Dzięki takiemu atrybutowi CountMin jest operacją trywialną. Możemy też aktualizować wartość tego atrybutu przy wszystkich operacjach kopcowych.

4 Klasówka 2009 (1), zadanie 1

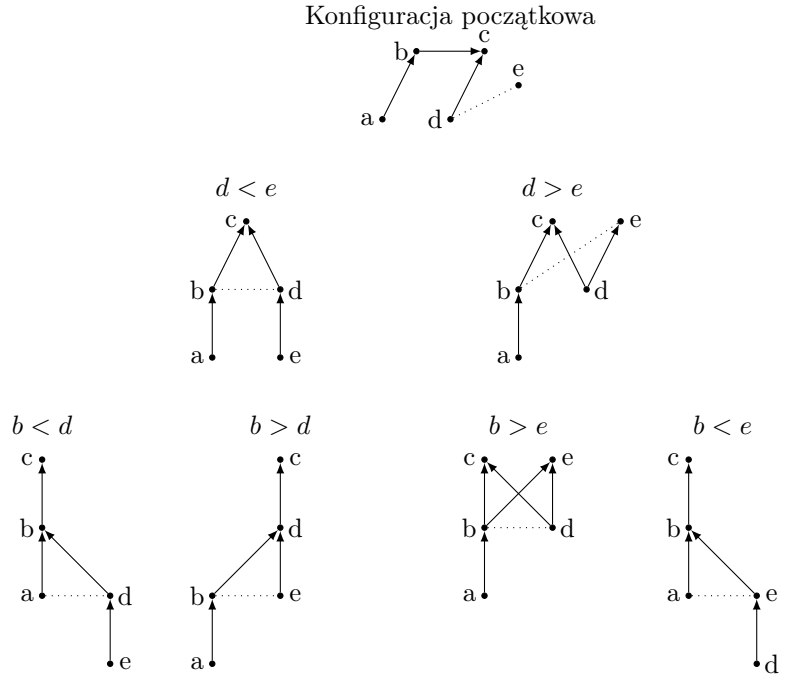
Dana jest tablica $n \times n$, $n > 1$, w której w każde pole wpisano liczbę całkowitą. Chcemy przejść z dolnego lewego rogu (z $(1, 1)$) do górnego prawego rogu (do (n, n)) i wrócić, idąc w drodze z (1,1) zawsze w prawo lub w górę, a z powrotem - w lewo lub w dół. Z danego pola można przejść tylko na pola sąsiednie (współrzędne różnią się o 1 na dokładnie jednej pozycji). Żadne pole nie może się pojawić na całej trasie (czyli tam i z powrotem) więcej niż raz, poza polem (1,1), które pojawia się na początku i na końcu trasy. Zaprojektuj algorytm znajdowania najtańszej trasy, czyli takiej, na której suma wartości pól jest najmniejsza. **Rozwiązanie:** dynamik po przekątnych

5 Klasówka 2010 (1), zadanie 2

Wykaż, że każdy algorytm znajdujący medianę w zbiorze 5-elementowym wykona w pesymistycznym przypadku co najmniej 5 porównań. Zaproponuj algorytm dokonujący tego za pomocą co najwyżej 6 porównań.

Rozwiązanie: Dolna granica: dzielimy wszystkie permutacje $\{1, \ldots, 5\}$ na klasy abstrakcji: (pozycja mediany, zbiór pozycji elementów mniejszych od mediany). Na przykład permutacja $(5, 1, 4, 3, 2)$ należy do klasy abstrakcji $(4, \{2, 5\})$. Takich klas abstrakcji jest $5 \cdot \binom{4}{2} = 30$. Dowolne drzewo porównań które rozróżnia wszystkie klasy abstrakcji musi mieć wysokość $h \geq \log_2 30 > 4$. Zauważmy, że jeśli algorytm utożsamia jakieś dwie klasy abstrakcji to możemy skonstruować dane dla których udzieli nieprawidłowej odpowiedzi.

Algorytm wykonujący 6 porównań. Porównaj a i b, porównaj c i d, porównaj $\max(a, b)$ i $\max(c, d)$. Bez utraty ogólności $a > b > c$ i $d > c$. Następnie:



6 Klasówka 2011 (1), zadanie 1

Danych jest n słów o takiej samej długości k, zbudowanych ze znaków n-elementowego, uporządkowanego alfabetu. Rozmiarem zadania w tym przypadku jest $R = nk$.

- Zaproponuj algorytm, który dla danego i, $1 \leq i \leq k$, obliczy w czasie $O(R)$ liczbę wszystkich par słów, które różnią się tylko na i-tej pozycji.
- Zaproponuj algorytm, który obliczy w czasie $O(R)$ liczbę wszystkich par słów, które różnią się tylko na dokładnie jednej pozycji.

Rozwiązanie: Zakładamy że wszystkie słowa na wejściu są różne (możemy to łatwo sprawdzić).

Dla dowolnego i, j przez $pref(i, j)$ oznaczamy kod prefiksu słowa w_i długości j, chcemy żeby kody były liczbami z zakresu 1..n takimi, że, $w_i[1..j] = w_q[1..j]$ wtw $pref(i, j) = pref(q, j)$ (czyli mogą służyć do porównywania prefiksów ustalonej długości)

Analogicznie definiujemy dla sufiksów: $suf(i, j)$.

Rozwiązujemy w czasie $O(n)$ każdy problem z osobna dla $j \in 1..k$ (w tym kroku będziemy liczyć pary słów które różnią się dokładnie na j-tej pozycji)

```
P = ∅
for i:=1..n do
    for j:=1..k do
        P+= (pref(i, j − 1), suf(i, k − j), i) (czyli zapisujemy kod słowa
          bez j-tego znaku)
posortuj leksykograficznie trójki z P
ile:=0
foreach grupy G trójek o tych samych wartościach pierwszych dwóch
  elementów do
    // dowolna para słów z G różni się jedynie na j-tej pozycji
    ile+ = |G| ∗ (|G| − 1)/2
```

Warto zauważyć, że jeśli jakieś dwa słowa różnią się na dokładnie jednej pozycji to istnieje tylko jedna wartość j w której zostaną zliczone

Ponieważ każda faza zajmuje czas $O(n)$ i faz jest k więc cały algorytm zajmuje $O(nk)$.

Pozostaje jeszcze powiedzieć jak obliczyć pref/suf - robimy to podobnie jak w izomorfizmie drzew, trzeba po prostu kompresować kody:

```
for i:=1 to n do
    pref(i, 1) = w_i[1]
for j:=2 to k do
    P = ∅
    for i:=1 to n do
        P += (pref(i, j − 1), w_i[j], i)
    posortuj leksykograficznie trójki z P
    zgrupuj trójki o tych samych wartościach pierwszych dwóch
      elementów w G_1, G_2, ..G_p
    for t:=1 to p do
        foreach (p, q, i) ∈ G do
            pref(i, j) = t
```

7 Klasówka 2011 (1), zadanie 2

W tym zadaniu rozważamy n-elementowe ciągi k-uporządkowane (i-ty element ciągu jest nie większy od elementu $i + k$), $1 \leq k \leq n$.

(5 pkt) Udowodnij, że każdy algorytm sortujący przez porównania wymaga w pesymistycznym przypadku $\Omega(n \log k)$ porównań do posortowania n-elementowego ciągu k-uporządkowanego.

(5 pkt) Zaproponuj algorytm sortujący takie ciągi w czasie $O(n \log k)$.

Rozwiązanie: TODO

8 Klasówka 2012 (1), zadanie 2

Powiemy, że dwa napisy są podobne wtedy i tylko wtedy, gdy zawierają jedna-

kowe liczby wystąpień tych samych znaków. Danych jest n napisów nad alfabetem m-znakowym $\{1, 2, \ldots, m\}$. Zaproponuj algorytm, który stwierdza, ile jest wśród nich różnych klas napisów podobnych. Twój algorytm powinien działać w czasie $O(R + m)$, gdzie R jest sumą długości wszystkich napisów.

Rozwiązanie: Podstawowa idea:

- dla każdego słowa w_i oblicz jego kod $code(w_i) = sorted(w_i)$, gdzie $sorted(w)$ oznacza słowo w z uporządkowanymi niemalejącymi znakami (np. $sorted(adbacab) = aaabbcd$)
- posortuj słowa $code(w_1), \ldots, code(w_n)$ używając algorytmu z ćwiczeń (sortowanie leksykograficzne słów różnej długości)
- usuń duplikaty z posortowanej listy.

Kroki drugi i trzeci w uczywisty sposób zajmą czas $O(R + m)$ Niestety jeśli pierwszy krok tego algorytmu zaimplementujemy naiwnie, to może się okazać, że obliczenie $code(w_i)$ zajmie nam czas $O(|w_i| + m)$, co w sumie może dać $O(R + nm)$.

Na szczęście możemy wygenerować kody słów w efektywniejszy sposób. Każdy znak z $w_1, \ldots, w_n$ zastępujemy przez trójkę (c, i, j) oznaczającą że $w_i[j] = c$. Sortujemy wszystkie trójki w jednym kroku. Teraz dzięki tej posortowanej liście mamy uporządkowane wszystkie litery z całego zbioru słów i możemy je kolejno dopisywać do kodów słów:

Input: lista słów $w_1, \ldots, w_n$
T:=pusta lista
foreach $w_i \in w_1, \ldots, w_n$ **do**
  foreach $j \in 1, \ldots, |w_i|$ **do**
    dodaj $(w_i[j], i, j)$ do T
posortuj leksykograficznie trójki z T
foreach $i \in 1, \ldots, n$ **do**
  $code[w_i] = \epsilon$ (pusty napis)
foreach $(c, i, j) \in T$ **do**
  $code[w_i] += c$

Dzięki "zbiorczemu" sortowaniu listy T udało się obliczyć kody wszystkich słów w w czasie $O(R + m)$.
Przykład:

```
w_1 = aba
w_2 = ba
w_3 = caa
w_4 = ab

T = [
  (a, 1, 1), (b, 1, 2), (a, 1, 3),
  (b, 2, 1), (a, 2, 2),
  (c, 3, 1), (a, 3, 2), (a, 3, 3),
  (a, 4, 1), (b, 4, 2)
]
```

```
posortowane T = [
  (a, 1, 1), (a, 1, 3), (a, 2, 2), (a, 3, 2), (a, 3, 3), (a, 4, 1),
  (b, 1, 2), (b, 2, 1), (b, 4, 2),
  (c, 3, 1)
]

code(w_1) = aab
code(w_2) = ab
code(w_3) = aac
code(w_4) = ab
```

9 Klasówka 2012 (1), zadanie 3

Dana jest $2n$-elementowa tablica zawierająca n zer i n jedynek. Chcemy ją uporządkować tak, żeby zera i jedynki były ułożone na przemian, począwszy od zera, tj. 010101... Zaproponuj efektywny algorytm, który wykona to w miejscu i stabilnie (tj. kolejność zer i kolejność jedynek z wejścia muszą być zachowane).

Rozwiązanie: Posortuj stabilnie (ala MergeSort) a następnie rekurencyjnie poprzeplataj.

Algorytm 2: Sort(A)

if $|A| \geq 2$ **then**
  (Z_l, O_l)=Sort($A[1..n/2]$)
  (Z_r, O_r)=Sort($A[n/2+1..n]$)
  Exchange(O_l, Z_r)
  return $(Z_l + Z_r, O_l + O_r)$
else
  return $(A, \emptyset)$ (if A=[0]) or $(\emptyset, A)$ otherwise

Algorytm 3: Unpack(A)

if $|A| > 2$ **then**
  $l = \lfloor |A|/4 \rfloor$; $r = \lceil |A|/4 \rceil$;
  // zamień ciąg $0^{|A|/2}1^{|A|/2}$ na $0^l1^l0^r1^r$
  Exchange($A[(l+1)..2l], A[(2l+1)..(2l+r)]$)
  Unpack($A[1..2l]$)
  Unpack($A[(2l+1)..n]$)

10 Klasówka 2012 (2), zadanie 1

Zaprojektuj strukturę danych, która umożliwia efektywne wykonywanie ciągu operacji Łącz(u,v) i Głębokość(u) na lesie drzew ukorzenionych o zbiorze wierzchołków $1, 2, \ldots, n$. Początkowo każde drzewo jest jednowierzchołkowe. Operacja Łącz(u,v) polega na połączeniu dwóch różnych drzew o korzeniach u i v w jedno drzewo o korzeniu v, poprzez uczynienie u synem v (podwiązanie u do v). Operacja Głębokość(u) polega na wyznaczeniu głębokości wierzchołka

u w aktualnie zawierającym go drzewie w lesie. Podaj sposób i koszt inicjacji swojej struktury danych, a następnie koszt wykonania każdej z operacji Łącz i Głębokość w zaprojektowanym przez siebie rozwiązaniu.

Rozwiązanie:

Do każdego węzła Find-Union dodajemy atrybut Δ (początkowo równy 0), który będzie zawierał względną odległość do ojca.

Algorytm 4: Głębokość(v)

```
v' = Find(v)
if v ≠ v' then
    return Δ(v) + Δ(v')
else
    return Δ(v)
```

Algorytm 5: Find(v)

```
if v = p(v) then
    return v
else
    p' = p(v)
    v' = Find(p')
    p(v) := v' - kompresja ścieżki
    if v' ≠ p' then
        Δ(v) := Delta(v) + Delta(p')
    return v'
```

Algorytm 6: Łącz(u, v)

```
u' := Find(u)
v' := Find(v)
łączymy według rozmiarów drzew (mniejsze do większego) if
 size(u') < size(v') then
    p(u') := v'
    Δ(u') := Δ(u') + 1
else
    p(v') := u'
    Δ(u') := Δ(u') + 1
    Δ(v') := Δ(v') − Δ(u')
```

11 Klasówka 2013 (1), zadanie 1

Zaprojektuj optymalny algorytm pod względem pesymistycznej liczby porównań, który znajduje dwa środkowe elementy w zbiorze czterech elementów. Dowiedź poprawności swojego rozwiązania.

Rozwiązanie: Algorytm używający 4 porównań:

- porównaj (a, b) i (c, d)
- porównaj $\min(a, b)$ i $\min(c, d)$
- porównaj $\max(a, b)$ i $\max(c, d)$

- w ten sposób wyznaczamy $\min(a, b, c, d)$ i $\max(a, b, c, d)$, pozostałe dwa elementy to poszukiwane środkowe wartości.

12 Klasówka 2013 (1), zadanie 2

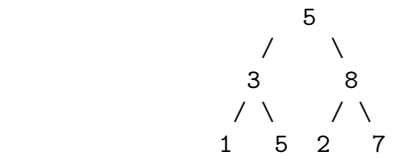
Drzewem klasówkowym nazywamy pełne drzewo binarne, w którym klucze są rozmieszczone zgodnie z następującą regułą: dla każdego węzła x najmniejszy klucz w poddrzewie o korzeniu x znajduje się w jego lewym podrzewie.

Zaproponuj implementację drzewa klasówkowego w sposób, który umożliwia wydajne wykonywanie operacji kolejki priorytetowej:

- Ini:: mając dane $n = 2^k - 1$ kluczy zbuduj n-węzłowe drzewo klasówkowe
- Min:: podaj wartość najmniejszego klucza w drzewie
- ChangeKey(x,k):: zmień wartość klucza we wskazanym węźle x na k

Uzasadnij poprawność swoich rozwiązań oraz dokonaj analizy ich złożności obliczeniowej.

Rozwiązanie: Przykład drzewa klasówkowego:



Wzbogacamy węzły v o dodatkowy atrybut min, który zawiera najmniejszy element z poddrzewa (łącznie z wartością $v.x$). Formuła na aktualizację tego atrybutu:

$$v.min := \min(v.left.min, v.right.min, v.x)$$

Tak jak w kopcu jesteśmy w stanie zdefiniować operację $DownHeap(v)$:

- jeśli v jest liściem to nic nie rób,
- jeśli $v.left.min > v.right.min$ -> $swap(v.left, v.right)$
- jeśli $v.x < v.left.min, v.right.min$ -> zamień $v.x$ i $v.left.x$, zaktualizuj $v.min$ i wykonaj $DownHeap(v.left)$

Analogicznie $UpHeap(v)$:

- jeśli v jest korzeniem to nic nie rób,
- niech $p = parent(v)$,
- zaktualizuj $p.min$,
- jeśli $p.left.min > p.right.min$ -> $swap(p.left, p.right)$
- wykonaj $UpHeap(p)$

Implementacja operacji:

- Ini:: utwórz drzewo a następnie wykonaj $DownHeap$ dla wszystkich węzłów idąc od warstwy k to 1 (tak jak w liniowym algorytmie tworzenia kopca)
- Min:: zwróć $root.min$
- ChangeKey:: zmień klucz i wykonaj $DownHeap(v)$ i $UpHeap(v)$

13 Klasówka 2013 (1), zadanie 3

Danych jest k uporządkowanych list o długościach będących parami różnymi potęgami dwójki. Zaproponuj wydajny algorytm scalenia tych list w jedną listę uporządkowaną. Uzasadnij poprawność swojego algorytmu i dokonaj analizy jego złożoności obliczeniowej ze względu na liczbę porównań wykonywanych podczas scalania.

Rozwiązanie: Uporządkuj listy rosnąco według długości i scalaj od najkrótszej do najdłuższej. Złożoność czasowa: $O(\sum |L_i|)$. Złożoność pamięciowa: $O(k)$ (na potrzeby uporządkowania list, nie potrzebujemy dodatkowej pamięci na scalanie bo operujemy na listach, które można scalać w pamięci $O(1)$).

14 Klasówka 2014 (1), zadanie 3

Dane są liczby całkowite dodatnie n, k, przy czym $k \leq \sqrt{n}$. W tablicy $a[1..n]$ zapisano n liczb całkowitych o co najmniej k różnych wartościach. Należy zaprojektować algorytm, który stabilnie i w miejscu przemieści k parami różnych liczb na początek tablicy a i uporządkuje je rosnąco. Stabilność w tym przypadku oznacza, że kolejność występowania w tablicy liczb o tych samych wartościach zostaje zachowana. Twój algorytm powinien działać w czasie $O(n \log n)$.

Rozwiązanie: Możemy użyć następującego algorytmu:

Algorytm 7: Solution1(A, k)

```
Niech B oznacza blok A w którym będziemy gromadzić posortowane
  rosnąco różne elementy z A
Początkowo B jest pusty blokiem na samym początku A
foreach i ∈ 1, ..., n do
    if binarySearch(A[i], B) then
        // element A[i] jest już znany więc go ignorujemy
    else
        // element A[i] jest nowy i chcemy go dodać do B
        niech X oznacza blok zaczynający się za B i kończący na
          A[i − 1]
        Exchange(B, X)
        dodaj A[i] do B
        if |B| ≥ k then break
przenieś blok B na początek A
```

Analiza: Koszt $O(n \log n)$ ze względu na wykonywane $O(n)$ razy wyszukiwanie binarne. Pozostałe operacje zajmują $O(n)$ czasu:

- koszt dodawania nowych elementów to $O(k^2)$ czyli $O(\sqrt{n}^2) = O(n)$,
- koszt wszystkich operacji Exchange to $O(n)$ ponieważ $\sum_{j=1}^{p} |X_j| \leq n$ (zauważmy, że wszystkie zbiory X_j są rozłączne), oraz $\sum_{j=1}^{p} |B_j| \leq k^2 \leq n$.

15 Klasówka 2015 (1), zadanie 3

Rozważamy dynamicznie zmieniający się ciąg $A =< a_1, a_2, \ldots, a_n >$, parami różnych n liczb całkowitych. Na ciągu A dozwolona jest jedyna operacja $NaPoczatek(i)$, $1 \leq i \leq n$, która przesuwa element a_i na początek A.

Przykład: Dla $A =< 4, 1, 3, 5, 2 >$, po wykonaniu $NaPoczatek(3)$ dostajemy $A =< 3, 4, 1, 5, 2 >$.

Interesuje nas ciąg operacji NaPoczątek o minimalnej długości, których wykonanie posortuje A. Nazwijmy go minimalnym ciągiem sortującym.

a) [3 punkty] Zaprojektuj algorytm, który w czasie $O(n)$ wyznacza pierwszy element minimalnego ciągu sortującego.

b) [2 punkty] Zaprojektuj efektywny algorytm wyznaczający cały minimalny ciąg sortujący.

c) [4 punkty] Udowodnij poprawność swoich rozwiązań.

Dokonaj analizy złożoności czasowej zaproponowanych algorytmów.

Rozwiązanie: Zauważmy, że jeśli algorytm sortujący wykona przeniesienie elementu $a_i = x$ to w kolejnych ruchach musi również przenieść wszystkie elementy o wartościach $1, \ldots, x-1$. Dodatkowo optymalny algorytm sortujący nie przenosi żadnego elementu więcej niż 1 raz.
Podpunkt (a):
Należy wyznaczyć, minimalne $k \leq n$, takie, że $k+1, k+2, \ldots, n$ jest podciągiem A. Można to zrobić w czasie $O(n)$ analizując ciąg A od prawej strony i szukając kolejno n, $n-1$, itd. Pierwsza operacja to $NaPoczatek(A^{-1}(k))$ (przeniesienie elementu o wartości k).
Podpunkt (b):
Kolejne operacje to $NaPoczatek(A^{-1}(k-1))$, $NaPoczatek(A^{-1}(k-2))$, $\ldots$, $NaPoczatek(A^{-1}(1))$.

Niestety żeby efektywnie wykonywać operację A^1 konieczna jest struktura danych, która pozwala na:

- znalezienie indeksu elementu o wartości x
- usunięcia elementu o wartości x z ciągu
- dodanie elementu o wartości x na początek ciągu

Przy pomocy drzew zrównoważonych, każdą z tych operacji można wykonać w czasie $O(\log n)$. Co daje algorytm o złożoności $(n \log n)$.

Przykład:

```
A   = 4 1 5 6 3 7 2
k=3, ponieważ [4,5,6,7] jest podciągiem A
NaPoczatek(A^{-1}(3)=5)
A_1 = 3 4 1 5 6 7 2
NaPoczatek(A^{-1}(2)=7)
A_2 = 2 3 4 1 5 6 7
NaPoczatek(A^{-1}(1)=4)
A_3 = 1 2 3 4 5 6 7
```

16 Klasówka 2015 (2), zadanie 2

Niech A będzie skończonym, dynamicznie zmieniającym się ciągiem, którego elementami są liczby ze zbioru $\{-1, 0, 1\}$. Podciąg kolejnych elementów A nazwiemy dobrym, gdy jego suma jest równa zero. Podciąg jest super-dobry, gdy jest dobry i suma elementów w każdym jego prefiksie jest nieujemna.

Przykład: W ciągu A = [1,1,0,-1,0,0,1,-1,1,1,-1], podciąg -1,1,1,-1 jest dobry, ale nie super-dobry. Super-dobrym podciągiem jest na przykład 1,0,-1.

1. Zaproponuj algorytm, który w czasie liniowym obliczy długość najdłuższego super-dobrego podciągu danego ciągu A.

2. Zaproponuj strukturę danych, która pozwoli na wydajne wykonywanie następujących operacji na A:

   - Ini(A):: A := []; //wykonywana tylko raz, na początku
   - Wstaw(A,e,i):: wstaw nowy element e jako i-ty w A, $1 \leq i \leq |A| + 1$;
   - Usuń(A,i):: usuń i-ty element z A, $1 \leq i \leq |A|$
   - SuperDobry(A,i,j):: sprawdź, czy podciąg A[i..j] jest super-dobry, $1 \leq i \leq j \leq |A|$;

W każdym zadaniu uzasadnij poprawność swoich rozwiązań i dokonaj analizy złożoności obliczeniowej zaproponowanych algorytmów.

17 Klasówka 2016 (1), zadanie 2

Niech n będzie dodatnią liczbą całkowitą. Dla dodatniej liczby całkowitej k powiemy, że ciąg liczb $a[1], \ldots, a[n]$ jest k-dobry, jeśli każda inwersja (i, j), $1 \leq i < j \leq n$, spełnia $j \leq i + k$.

a) [8 punktów] zaproponuj asymptotycznie optymalny ze względu na porównania algorytm sortujący ciągi k-dobre. Uzasadnij asymptotyczną optymalność swojego algorytmu. Uwaga: w tym zadaniu argumentami funkcji złożoności są k i n.

b) [5 punktów] zaproponuj efektywny czasowo i pamięciowo algorytm, który sprawdza czy ciąg liczb $a[1], \ldots, a[n]$ dla zadanej liczby całkowitej k, jest k-dobry. Uzasadnij poprawność algorytmu i dokonaj analizy czasowej i pamięciowej

Rozwiązanie: (a) Dowolny algorytm sortujący oparty o porównania musi wykonać $\Omega(n \log k)$ porównań: istnieje co najmniej $(k!)^{n/k}$ różnych permutacji n-elementowych, które są k-dobre. Stąd algorytm sortujący musi wykonać co najmniej $\log((k!)^{n/k})$ porównań, czyli $\Omega(n/k \cdot k \log k) = \Omega(n \log k)$. Konstrukcja tej rodziny permutacji: podziel liczby na bloki $B_1 = 1..k$, $B_2 = k+1..2k$, itd. permutacja otrzymana z dowolnego przemieszania bloków $p(B_1) + p(B_2) + p(B_3) \ldots$ jest k-dobra.

Algorytm: sortuj kolejno bloki długości $2k$ zaczynające się na pozycjach $1, k+1, 2k+1, \ldots$.

(b) Niech $pmax[i]$ oznacza $\max a[1..i)$ (maksymalne wartości dla wszystkich prefiksów tablicy, uwaga! $pmax[i]$ nie zawiera $a[i]$). Tablica a jest k-dobra, jeśli

$$\forall_{k \leq j \leq n} pmax[j-k] \leq a[j]$$

18 Klasówka 2016 (2), zadanie 1

Zaprojektuj strukturę danych, która implementuje standardowe operacje słownika reprezentującego zbiór S liczb całkowitych (Insert, Delete, Find) oraz dodatkowo operację:

- $MaxSubset(d, S)$:: podaj rozmiar maksymalnego podzbioru zbioru S, w którym każde dwie liczby różnią się co najmniej o d, gdzie d jest zadaną z góry stałą całkowitą.

Podaj rozwiązanie dla:

- (4 punkty) d=2,
- (7 punktów) d=10.

Uzasadnij poprawność swoich rozwiązań i przeanalizuj złożoność czasową poszczególnych operacji na strukturze danych względem n (aktualnego rozmiaru struktury).

19 Klasówka 2017 (1), zadanie 1

W liczbowym, różnowartościowym ciągu $< a_1, a_2, \ldots, a_n >$, $n > 2$, element a_i, $1 < i < n$, nazywamy lokalnym ekstremum gdy jest mniejszy lub większy od obu sąsiadów, tzn.

albo $a_{i-1} > a_i < a_{i+1}$, albo $a_{i-1} < a_i > a_{i+1}$

a) Udowodnij, że każdy algorytm sortujący przez porównania 4-elementowe ciągi z co najwyżej jednym lokalnym ekstremum wymaga wykonania w pesymistycznym przypadku co najmniej 4 porównań.

b) Zaproponuj algorytm sortowania 4-elementowych ciągów z co najwyżej 1 lokalnym ekstremum za pomocą co najwyżej 4 porównań

c) Zaproponuj algorytm, asymptotycznie optymalny ze względu na liczbę porównań, sortujący ciągi o co najwyżej k lokalnych ekstremach dla zadanego k, $0 < k < n$. Dowiedź optymalności swojego rozwiązania.

20 Klasówka 2017 (2), zadanie 1

Niech S będzie skończonym podzbiorem różnych, dodatnich liczb całkowitych, a d dodatnią liczbą całkowitą.

1. (5 punktów) Przyjmij, że elementy zbioru zapisano w uporządkowanej malejąco tablicy a[1..n], gdzie $n = |S|$. Zaprojektuj wydajny algorytm, który policzy liczbę (nieuporządkowanych) par różnych elementów z S, których suma jest nie większa od d. Przykład Dla $S = \{6, 5, 2, 1\}$ i $d = 7$, liczba takich par wynosi 4.

2. (5 punktów) Załóżmy, że d jest ustalone (ale może być bardzo duże), natomiast S jest zbiorem dynamicznym. Zaprojektuj efektywną strukturę danych, która umożliwi wydajne wykonywanie na zbiorze S następujących operacji:

   - Ini(S):: $S := \emptyset$; // operacja wykonywana raz, na samym początku obliczeń;
   - Insert(x, S):: $S := S \cup \{x\}$;
   - Delete(x, S):: $S := S - \{x\}$;
   - Pairs(S):: return liczba par różnych elementów z S, których suma jest nie większa od d.

Zaproponuj rozwiązanie, w którym operacje Insert i Delete są wykonywane w czasie $O(\log |S|)$, a operacja Pairs w czasie stałym, niezależnie od wielkości d. Możesz przyjąć, że operacje arytmetyczne i porównania na liczbach są wykonywane w stałym czasie.

21 Klasówka 2019 (1), zadanie 1

a) (2 punkty) Ile jest permutacji liczb od 1 do n o co najwyżej 2 inwersjach?

b) (3 punkty) Zaproponuj optymalny (ze względu na liczbę porównań) algorytm sortujący przez porównania ciągi różnowartościowe o co najwyżej 2 inwersjach.

22 Klasówka 2019 (1), zadanie 2

Rozważamy klasyczne sortowanie przez wstawianie tablicy a[1..n], w porządku niemalejącym, w której zapisano pewną permutację liczb od 1 do n. Zaprojektuj wydajny algorytm, który dla danej zawartości tablicy a wyznaczy najmniejsze takie i, że podczas sortowania przez wstawianie a[i] jest porównywane najczęściej.

Rozwiązanie:

Element $a[i]$ jest porównany:

- niech $k = |\{j : a[j] > a[i] \text{ oraz } j < i\}|$ z k elementami podczas i–tej fazy sortowania, które zakończyły się zamianą elementów

- niech $m = |\{j : a[j] < a[i] \text{ oraz } j > i\}|$ z m elementami podczas kolejnych faz sortowania, które zakończyły się zamianą elementów
- niech p to liczba elementów z którymi porównywany jest $a[i]$ ale porównanie nie zakończyło się zamianą.

Jak policzyć wartości p:

```
foreach x ∈ A do
    p_x = 0
S := ∅
for i = 1..n do
    x = A[i]
    y = max{y ∈ S : y < x}
    if y jest zdefiniowane then
        p_x := p_x + 1
        p_y := p_y + 1
    S = S + {A[i]}
```

23 Klasówka 2019 (1), zadanie 3

Dana jest tablica $a[1..4n]$, w której znajduje się po n rekordów etykietowanych każdą z liczb ze zbioru $\{0, 1, 2, 3\}$.

a) (4 punkty) Zaprojektuj liniowy algorytm sortujący tablicę a względem etykiet rekordów w miejscu.

b) (4 punkty) Podaj algorytm działający w czasie $O(n \log n)$, który sortuje tablicę a względem etykiet rekordów w miejscu i stabilnie (tzn. kolejność rekordów o tej samej etykiecie przed i po sortowaniu jest taka sama).

Rozwiązanie:

Algorytm liniowy (3 krotne rozwiązanie problemu flagi polskiej):

```
foreach x ∈ {0, 1, 2} do
    j:=1
    for i:=1 to 4n do
        if a[i] <= x then
            if j < i then swap(i, j)
            j:=j+1;
```

Algorytm $O(n \log n)$ (3 krotne rozwiązanie problemu stabilnego sortowania ciągów 0/1)

Algorytm 8: Sort

Data: $a[i..j]$, x
Result: $a[i..j]$ uporządkowana w ten sposób, że wszystkie wartości $\leq x$ występują przed wartościami $> x$
if $i < j$ **then**
  $m := \lfloor (i+j)/2 \rfloor$
  $Sort(a[i..m], x)$
  $Sort(a[m+1..j], x)$
  niech L blok $a[i..m]$ zawierający wartości $> x$
  niech R blok $a[m+1..j]$ zawierający wartości $\leq x$
  if $|L| > 0$ *and* $|R| > 0$ **then**
    zamień ze sobą bloki L i R

foreach $x \in \{0, 1, 2\}$ **do**
  $Sort(a, x)$

Powyższy algorytm używa pamięci dodatkowej rzędu $O(\log n)$ ale łatwo zamienić rekurencję na iterację i osiągnąć pamięć $O(1)$.

24 Klasówka 2020 (1), zadanie 1

Na uporządkowanym rosnąco n-elementowym ciągu $x_1, x_2, \ldots, x_n$ dokonano dokładnie k zmian wartości elementów w tym ciągu, $0 \leq k \leq n$. Ciąg otrzymany w ten sposób nazywamy *k-zaburzonym*.

Przykład

Uporządkowany ciąg $\langle 2, 6, 8, 12, 21 \rangle$. Po zmianach wartości elementów - pierwszego na 7 i czwartego na 1 - dostajemy ciąg 2-zaburzony $\langle 7, 6, 8, 1, 21 \rangle$.

Dana jest liczba całkowita k, $0 \leq k \leq n$, oraz *k*-zaburzony ciąg $x = \langle x_1, x_2, \ldots, x_n \rangle$. Interesuje nas wydajne sortowanie ciągu x ze względu na porównania.

a) Zaproponuj optymalne sortowanie ze względu na porównania dla $k = 1$ oraz
   - $n = 4$
   - $n = 5$

b) Zaproponuj asymptotycznie optymalny algorytm sortowania k-zaburzonych ciągów n-elementowych. Pamiętaj o uwzględnieniu obu parametrów - n i k.

Rozwiązanie: Punkt b)

Algorytm 9: Rozwiązanie($(x_1, \ldots, x_n), k$)

$A = Y = \emptyset$
for $i = 1, \ldots, n$ **do**
  if $A \neq \emptyset$ *and* $Top(A) > x_i$ **then**
    ▷ co najmniej jeden elemenet z $\{Top(A), x_i\}$ jest zaburzony, dla pewmości wrzucamy oba
    $Push(Y, x_i)$
    $Push(Y, Pop(A))$
  else
    $Push(A, x_i)$
▷ $|Y| \leq 2k$, A zawiera ciąg uporządkowany rosnąco
$Y' = sort(Y)$
return *Merge*(A, Y')

25 Klasówka 2021 (1), zadanie 1

Powiemy, że zbiór n liczb całkowitych jest prawie gęsty, jeśli zawiera podzbiór o rozmiarze większym niż $n/3$, w którym różnica pomiędzy największym i najmniejszym elementem jest mniejsza od n. Taki podzbiór nazywamy świadectwem. Dana jest dodatnia liczba całkowita n oraz n-elementowy zbiór liczb całkowitych S. Zaproponuj algorytm, który w czasie liniowym sprawdzi, czy S jest prawie gęsty.
Uwaga: 3 punkty uzyskasz za algorytm, który w czasie liniowym sprawdza, czy wskazany, dowolny element z S należy do jakiegoś świadectwa.

Rozwiązanie:

Algorytm 10: CzyNależy(S, n, x)

niech $S' = \{e \in S : |e - x| < n\}$
ponieważ wszystkie elementy z S' należą do przedziału $[e - n, \ldots, e + n]$ długości $2n + 1$ stąd możemy posortować S' w czasie liniowym
$S'' = sorted(S')$
for $i = 1, \ldots, m - \lceil \frac{n}{3} \rceil + 1$ **do**
  $j := i + \lceil \frac{n}{3} \rceil - 1$
  if $S''[j] - S''[i] < n$ **then**
    return *TAK* (ponieważ $s_i, \ldots, s_j$ + być ewentualnie x są świadectwem)
return *NIE*

Algorytm 11: Rozwiązanie(S, n)

foreach $k \in \{\lceil n/4 \rceil, \lceil n/2 \rceil, \lceil 3n/4 \rceil\}$ **do**
  $x :=$ DeterministicSelectKthElement(S, k)
  if *CzyNależy*(S, n, x) **then**
    return *TAK*
return *NIE*

26 Klasówka 2021 (1), zadanie 2

Zaproponuj optymalny ze względu na porównania algorytm sortowania siedmioelementowego ciągu $x_1, \ldots, x_7$, o którym wiadomo, że $x_1 < x_2$, $x_1 < x_3$, $x_4 < x_5$, $x_4 < x_6$. Udowodnij optymalność swojego algorytmu.

Rozwiązanie: Wszystkich permutacji 7 elementowych spełniających warunki zadania jest:

$$\binom{7}{3} \cdot 2 \cdot \binom{4}{3} \cdot 2 = 35 \cdot 2 \cdot 4 \cdot 2 = 560$$

Ponieważ:

- $\{x_1, x_2, x_3\}$ możemy wybrać z 7 elementów na $\binom{7}{3}$ sposobów
- mamy dwie możliwości na porównanie x_2 i x_3 ($x_2 < x_3$ lub $x_2 > x_3$)
- $\{x_4, x_5, x_6\}$ możemy wybrać z pozostałych 4 elementów na $\binom{4}{3}$ sposobów,
- mamy dwie możliwości na porównanie x_5 i x_6 ($x_5 < x_6$ lub $x_5 > x_6$)

Algorytm 12: Rozwiązanie

cmp(x_2, x_3) (bez straty ogólności $x_2 < x_3$)
cmp(x_5, x_6) (bez straty ogólności $x_5 < x_6$)
wstaw x_7 pomiędzy $x_1 < x_2 < x_3$ używając 2 porównań
scal dwa uporządkowane ciągu (4 i 3 elementowe) przy użyciu 6 porównań

27 Klasówka 2021 (1), zadanie 3

Dla dodatniej liczby całkowitej n kratownicą M_n nazywamy skierowany graf (V, E) bez pętli, w którym $V = \{(x, y) : x = 0, 1, ..., n \text{ oraz } y = 0, 1, ..., n\}$ i

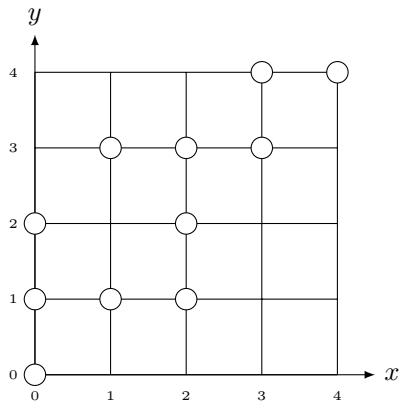
$$E = \{(x, y) \to (x', y') : 0 \leq x' - x \leq 1 \text{ oraz } 0 \leq y' - y \leq 1\}.$$

Wierzchołki grafu M_n pomalowano na biało lub czarno. Białą ścieżką nazwiemy każdą ścieżkę, na której wszystkie wierzchołki są białe. Dane są liczba całkowita $n > 0$, nieujemna liczba całkowita $m \leq (n+1)^2$ oraz m różnych, białych wierzchołków $(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$. Pozostałe wierzchołki są czarne. Zaproponuj wydajny (czasowo i pamięciowo) algorytm, który obliczy liczbę wszystkich białych ścieżek z wierzchołka $(0, 0)$ do wierzchołka (n, n).

Rozwiązanie: Aby uprościć rozumowanie załóżmy, że wszystkie operacje arytmetyczne możemy wykonywać w czasie stałym. Jest to o tyle istotne, że wynikowa wartość może być większa niż $\binom{2n}{n}$ (to jest liczba ścieżek dla pełnego białego grafu M_n w których poruszamy się tylko w prawo lub w górę).
Przykład:
Dla $n = 4$ i $m = 11$ białych wierzchołków:

$$B = \{(0,0), (0,1), (0,2), (1,1), (1,3), (2,1), (2,2), (2,3), (3,3), (3,4), (4,4)\}$$

mamy następujący graf:



Niech $Ile(i,j)$ oznacza liczbę białych ścieżek z $(0,0)$ do (i,j) spełniających warunki zadania.

Jeśli $m = \Theta(n^2)$ to możemy w czasie $O(n^2)$ policzyć $Ile(n,n)$ korzystając z następujacych wzorów:

$$Ile(0,0) = \begin{cases} 1 & \text{jeśli wierzchołek } (0,0) \text{ jest biały} \\ 0 & wpp \end{cases}$$

$$Ile(\cdot,-1) = Ile(-1,\cdot) = 0$$

Jeśli (i,j) jest biały to:

$$Ile(i,j) = Ile(i-1,j) + Ile(i-1,j-1) + Ile(i,j-1)$$

Jeśli (i,j) jest czarny to:

$$Ile(i,j) = 0$$

Jeśli $m < n$ to odpowiedzią jest 0 (nie ma wystarczająco dużo białych węzłów).

Jeśli $m = \Omega(n)$ to możemy policzyć $Ile(n,n)$ w czasie $O(m)$. Załóżmy, że wierzchołki $(0,0)$ i (n,n) są białe (jeśli byłoby inaczej to oczywiście odpowiedzią jest 0).
Zdefiniujmy sobie 3 porządki ($<_{x,y}$, $<_{y,x}$, $<_d$) na parach liczb:

- $(x_1,y_1) <_{x,y} (x_2,y_2)$ wtw $(x_1 < x_2)$ lub $(x_1 = x_2) \wedge (y_1 < y_2)$
- $(x_1,y_1) <_{y,x} (x_2,y_2)$ wtw $(y_1 < y_2)$ lub $(y_1 = y_2) \wedge (x_1 < x_2)$
- $(x_1,y_1) <_d (x_2,y_2)$ wtw $(x_1 - y_1 < x_2 - y_2)$ lub $(x_1 - y_1 = x_2 - y_2) \wedge (x_1 < x_2)$

W pierszym kroku uporządkujmy leksykograficznie (czyli wg. $<_{x,y}$) ciąg B. Od teraz zakładamy, że:

$$(x_1,y_1) <_{x,y} (x_2,y_2) <_{x,y} \ldots <_{x,y} (x_m,y_m)$$

Przy takim uporządkowaniu definiujemy ILE$[i]$ jako liczbę białych ścieżek z $(0,0)$ do (x_i,y_i).

Dodatkowo definiujemy pomocnicze wartości:

$$\textsc{GdzieX}[i] = \begin{cases} j & \text{jeśli } (x_j = x_i) \wedge (y_j + 1 = y_i) \\ -1 & \text{jeśli takie } j \text{ nie istnieje} \end{cases}$$

$$\textsc{GdzieY}[i] = \begin{cases} j & \text{jeśli } (y_j = y_i) \wedge (x_j + 1 = x_i) \\ -1 & \text{jeśli takie } j \text{ nie istnieje} \end{cases}$$

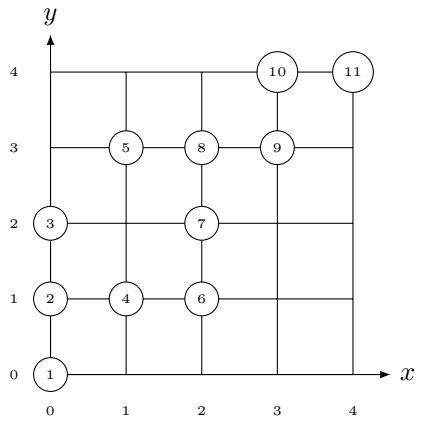
$$\textsc{GdzieD}[i] = \begin{cases} j & \text{jeśli } (x_j + 1 = x_i) \wedge (y_j + 1 = x_i) \\ -1 & \text{jeśli takie } j \text{ nie istnieje} \end{cases}$$

Wartości $\textsc{Gdzie}(X/Y/D)$ można obliczyć w czasie $O(m)$ sortując wierzchołki wg porządków $<_{x,y}$ / $<_{y,x}$ / $<_d$ (co możemy zrobić w czasie $O(m)$). Jedynym kandydatem na wartość Gdzie jest poprzedni wierzchołek w danym porządku (co możemy zweryfikować w czasie $O(1)$).
Gdy już mamy obliczone wartości Gdzie, wartości *Ile* obliczamy korzystając ze wzoru:

$$\textsc{Ile}[i] = \textsc{Ile}[GdzieX[i]] + \textsc{Ile}[GdzieY[i]] + \textsc{Ile}[GdzieD[i]]$$

(przy czym sztucznie definiujemy $\textsc{Ile}[-1] = -0$)

Przykład:



i	1	2	3	4	5	6	7	8	9	10	11
(x_i, y_i)	(0,0)	(0,1)	(0,2)	(1,1)	(1,3)	(2,1)	(2,2)	(2,3)	(3,3)	(3,4)	(4,4)
$\textsc{GdzieX}[i]$	-1	1	2	-1	-1	-1	6	7	-1	9	-1
$\textsc{GdzieY}[i]$	-1	-1	-1	2	-1	4	-1	5	8	-1	10
$\textsc{GdzieD}[i]$	-1	-1	-1	1	3	-1	4	-1	7	8	9
$\textsc{Ile}[i]$	1	1	1	2	1	1	3	4	7	11	18

28 Klasówka 2022 (1), zadanie 1

Dla dodatniej liczby całkowitej n, w tablicy a[1..n] zapisano n różnych elementów z pewnego liniowo uporządkowanego uniwersum. Rangą elementu a[i], oznaczanego przez R(i), nazywamy pozycję tego elementu w tablicy a po jej uporządkowaniu rosnąco.

Przykład

Dla $a = [2, 5, 3, 4]$, rangami elementów $a[1], a[2], a[3]ia[4]$ są odpowiednio 1, 4, 2, 3.

Dla nieujemnej liczby całkowitej k powiemy, że tablica jest k-zaburzona wtedy i tylko wtedy, gdy dla każdego $i = 1, 2, \ldots, n$, $|R(i) - i| \leq k$. Dana jest liczba całkowita k, $0 \leq k < n$, oraz k-zaburzona tablica a[1..n].

a) Zaproponuj wydajny algorytm, który zbuduje w tablicy a kopiec zupełny typu MIN.

b) Zaproponuj algorytm, optymalny ze względu na liczbę porównań, sortowania 1-zaburzonej tablicy a.

c) Zaproponuj asymptotycznie optymalny algorytm sortowania tablicy a.

Rozwiązanie: Obserwacja: w tablicy k-zaburzonej jeśli $i + 2k < j$ to mamy $A[i] < A[j]$.

Punkt a) utwórz kopiec na pierwszych $4k$ elementach (pozostałe elementy będą już tworzyć dobry porządek kopcowy.

Punkt b) w tablicy 1-zaburzonej tylko sąsiednie elementy mogą być zamienione. Więc wystarczy $n - 1$ porównań.

Punkt c) złożoność czasowa $O(n \log k)$:

Algorytm 13: Rozwiązanie(A, n, k)

```
i = 1
while i < n do
    HeapSort(A[i, ..., i + 4k])
    i+ = 2k
```

29 Klasówka 2022 (1), zadanie 2

W tym zadaniu rozważamy skończone słowa nad alfabetem $\{d, i, k, s\}$. Niech s będzie słowem i niech $s[j]$ będzie j-tym znakiem w tym słowie. Blokiem znaku $s[j]$ nazywamy maksymalne podsłowo s zawierające znak $s[j]$, w którym wszystkie znaki są takie same, równe $s[j]$. Taki blok oznaczamy przez $B(s, j)$.

Przykład

W słowie $s = abbaac$ mamy $B(s, 3) = bb$, $B(s, 4) = aa$.

O dwóch słowach s_1 i s_2 powiemy, ze są podobne wtedy i tylko wtedy, gdy $|s_1| = |s_2|$ oraz dla każdego $j = 1, \ldots, |s_1|$, bloki $B(s_1, j)$ i $B(s_2, j)$ są tej samej długości.

Zaprojektuj wydajny algorytm, który dla danego zbioru Q złożonego z n skończonych słów nad alfabetem $\{d, i, k, s\}$, wyznaczy wszystkie jego maksymalne (nie dające się rozszerzyć) podzbiory słów podobnych.

Rozwiązanie: Dla słów z Q definiujemy funkcję kodową $C(x_1, ldots, x_k) = (i_1, \ldots, i_p)$ takie, że $1 \leq i_1 \leq i_2 \leq i_p = k$ oraz $x[i_j] \neq x[i_{j+1}]$.

Następnie sotujemy wszystkie kody słów i liczymy ile różnych kodów otrzymaliśmy.