Ezoteryczne Kartki
Pierwiastki

Jakub Bachurski

wersja 1.2.1.2

Wstęp

Pierwiastki nie są jedną konkretną techniką. Są raczej zbiorem metod, które mają jedną część wspólną: pierwiastek w złożoności. Czasem nazywa się je *square-root decomposition* (rozkład pierwiastkowy), bo w rozwiązaniu sprytnie dzielimy coś na części wielkości pierwiastka z wielkości dzielonego obiektu. Jest to jedyny podział, w którym liczba części i ich wielkość jest taka sama.

$$\begin{aligned} k &= \frac{n}{k} \\ k^2 &= n \\ k &= \sqrt{n} \end{aligned}$$

1 Rozkład ciągu

Najprostszym rodzajem pierwiastków jest rozkład ciągu długości n na bloki (pierwiastki) wielkości $\sqrt{n}$. W każdym bloku budujemy pewną strukturę, utrzymującą informacje o $\sqrt{n}$ elementach. Chcąc łączyć informacje z wielu struktur, będziemy musieli to zrobić co najwyżej $O(\sqrt{n})$ razy – bo jest ich $\sqrt{n}$. Dzięki temu, że w ten sposób powstaje nam swego rodzaju jednopoziomowe drzewo, możemy pozwolić sobie na elastyczny wybór struktur.

1.1 Sumy

Treść Rozważmy najpierw klasyczny, prosty przykład: rozwiązywanie dynamicznego problemu zapytań o sumę i dodawania na przedziale. Formalnie, chcemy obsługiwać operacje na ciągu $a = (a_1, ..., a_n)$:

1. Dla podanego $[l, r]$ podaj sumę $\sum_{i=l}^{r} a_i$.
2. Dla podanego x oraz $[l, r]$, dla każdego $i \in [l, r]$ ustaw $a_i := a_i + x$.

Rozwiązanie Umiemy rozwiązać to drzewem przedziałowym, dzieląc ciąg na połowy, i konstruując rekurencyjnie poddrzewa na tych połowach. Alternatywnie możemy wykorzystać dwa drzewa Fenwicka, które przechowują sumy na sprytnie wybranych podciągach.

Wykorzystajmy jednak bardziej prymitywny sposób: podzielmy ciąg a na rozłączne bloki równej wielkości (lub prawie równej, jeżeli np. ostatni będzie za mały, to nic się nie stanie) – oznaczmy ich wielkość przez k (załóżmy, że k dzieli n).

Dla przykładu, weźmy ciąg a wielkości $n = 12$ i blok wielkości $k = 3$. Wpiszmy do niego trochę liczb całkowitych.

$\boldsymbol{a}$	4	-2	0	5	3	-7	10	4	4	-3	1	2

Spróbujmy dobrać strukturę bloku do operacji. Skupmy się najpierw na pierwszej z nich. Nasuwa się prosty pomysł: odpowiadając na zapytanie o sumę, jeżeli pewien blok zawiera się w przedziale z zapytania, dodajmy zapisaną wcześniej sumę elementów z tego bloku. Zapiszmy zatem w każdym bloku sumę elementów Σ.

$\boldsymbol{a}$	4	-2	0	5	3	-7	-3	4	4	-3	1	2
$\boldsymbol{\Sigma}$	2			1			5			0		

Pozostaje jeszcze kwestia, jak wykorzystać bloki przy operacji drugiej. Jeżeli pewien blok zawiera się w całości w przedziale modyfikowanym, możemy do niego wpisać wartość Δ sygnalizującą, że wszystkie elementy w bloku powiększono o Δ. Czyli powiększamy Δ o x, Σ musimy zmodyfikować o $k \cdot x$, a odczytując pojedyncze elementy musimy zwrócić $a_i + \Delta$. Elementy leżące poza blokami modyfikowane są zwyczajnie.

Wykonajmy modyfikację na przedziale $[2, 7]$ z $x = 2$.

$\boldsymbol{a}$	4	-2	2	5	3	-7	-1	6	4	-3	1	2
$\boldsymbol{\Sigma}$	4			7			9			0		
$\boldsymbol{\Delta}$	0			2			0			0		

Przeanalizujmy złożoność tego podejścia. Zauważmy, że bloków jest $\frac{n}{k}$. Realizując operacje, w przedziale będzie się zawierać co najwyżej $\frac{n}{k}$ bloków. Natomiast na krańcach przedziału znajdzie się co najwyżej $2(k-1)$ elementów, które należy aktualizować pojedynczo. Aktualizacja bloku lub pojedynczego elementu przebiega w $O(1)$.

Zatem sumarycznie zapytanie realizujemy w $O(k + \frac{n}{k})$. Można pokazać, że ta złożoność jest minimalna dla $k = \sqrt{n}$, tym samym otrzymujemy sumaryczną złożoność $O(\sqrt{n})$.

Podsumowując: w prosty sposób podzieliliśmy ciąg na równej wielkości bloki, a następnie tak dobraliśmy strukturę tych bloków, by łatwo realizować zapytania.

1.2 [UVa Online Judge] „Dynamiczne" inwersje

Treść W zadaniu mamy daną permutację a liczb od 1 do n ($1 \leqslant n \leqslant 200\,000$) oraz m zapytań ($1 \leqslant m \leqslant 100\,000$). W każdym zapytaniu należy wypisać liczbę inwersji w permutacji, a następnie usunąć z niej podaną liczbę x.

Rozwiązanie Zacznijmy od policzenia inwersji którymś klasycznym sposobem w $O(n \log n)$. Zastanówmy się teraz, jak zmienia się liczba inwersji ciągu po usunięciu elementu a_i: liczba inwersji spadnie o liczbę elementów większych na wcześniejszych indeksach, oraz liczbę elementów mniejszych na późniejszych.

Dochodzimy do wniosku, że chcemy obsługiwać strukturę działającą na ciągu, która umie zliczać na prefiksie elementy większe od pewnego x, na sufiksie elementy mniejsze od x, oraz usuwać element z danego indeksu.[1]

Najłatwiej operacje zliczania wykonywałoby się na ciągu posortowanym – możemy wykorzystać wyszukiwanie binarne i zrobić to w $O(\log n)$. Nie możemy jednak posortować całego ciągu. Podzielmy zatem ciąg na bloki wielkości k, i w każdym bloku zapiszmy oryginalną i posortowaną wersję podciągu.

Weźmy dla przykładu $a = (5, 3, 8, 4, 2, 9, 1, 6, 7)$ i podzielmy je na bloki wielkości $k = 3$.

5	3	8	4	2	9	1	6	7
$5, 3, 8$			$4, 2, 9$			$1, 6, 7$		
$3, 5, 8$			$2, 4, 9$			$1, 6, 7$		

Po usunięciu $3, 8, 9, 1, 6, 7$ chcemy, żeby struktura wyglądała tak:

5	?	?	4	2	?	?	?	?
5			$4, 2$					
5			$2, 4$					

Co nam to daje? Posiadając taką strukturę, operacje zliczania można wykonywać w $O(k + \frac{n}{k} \log n)$ (na każdym z bloków uruchamiamy wyszukiwanie binarne, elementy leżące w krańcowym bloku zliczamy iteracją), a usunięcie z niej to jedynie $O(k)$ (jeżeli trzymamy je na zwykłych dynamicznych tablicach – `vector`-ach). Początkowa konstrukcja wypada bardzo prosto: możemy ją wykonać w czasie $O(\frac{n}{k} \cdot k \log k) = O(n \log k)$. Podsumowując czas działania:

$$O(n \log k + m(k + \frac{n \log n}{k})) \underset{k=\sqrt{n \log n}}{\overset{min}{=}} O((n+m)\sqrt{n \log n})$$

Warto zwracać uwagę na przypadki, gdzie rozwiązaniem nie jest dokładnie pierwiastkiem – w tym wypadku czynnik $\sqrt{\log n}$ zapewnił nam kilkakrotne przyspieszenie.

[1] Istnieje alternatywne rozwiązanie tego zadania, wykorzystujące drzewo falkowe i działające w $O((n+m)\log^2 n)$. Czy umiesz je znaleźć, czytelniku?

2 Podział algorytmu

Tym razem zamiast na zwykłym ciągu, skupimy się na różnych obiektach. Chcąc rozwiązać dla nich pewien problem, może być to prostsze lub trudniejsze, w zależności od ich właściwości. Możemy jednak skorzystać z tego, że liczba bardziej skomplikowanych (tudzież trudniejszych do rozwiązania) obiektów jest ograniczona. Innym razem, różne przypadki po prostu lepiej rozwiąże się różnymi sposobami. O wiele lepiej widać to na konkretnych przykładach zadań.

2.1 [*Guide to Competitive Programming*] Odległość liter

Treść Weźmy planszę $n \times n$ wypełnioną wartościami (przyjmijmy, że są to litery z bardzo dużego alfabetu). Dla każdej z występujących w niej liter, chcemy znaleźć parę liter położonych najbliżej w odległości manhattańskiej[2].

A	F	B	A
C	E	G	E
B	D	A	F
A	C	B	D

Rozwiązanie Powiedzmy, że rozpatrujemy każdą z różnych liter oddzielnie. Oznaczmy obecnie rozpatrywaną literę x, a liczbę jej wystąpień na planszy jako c_x. Zastanówmy się nad możliwymi podejściami do tego problemu.

1. Możemy uruchomić równoległego BFS z każdego pola, gdzie występuje x. Między polami istnieje krawędź, jeżeli mają wspólny bok – odległość w takim grafie jest równa odległości manhattańskiej na planszy.
2. Dla każdej pary pól z x możemy sprawdzić odległość ze wzoru.

Podejścia te różnią się złożonością: odpowiednio $O(n^2)$ i $O(c_x^2)$. Zastanówmy się przez chwilę. Pierwszy algorytm opłaca się dopiero wtedy, gdy $c_x > n$. Spróbujmy to dokładniej przeanalizować.

Ustalmy pewną stałą k i stwórzmy dwie grupy liter: w grupie V będą litery, dla których $c_x < k$, a w grupie W takie, dla których $c_x \geqslant k$. Zauważmy, że musi zachodzić $|W| \cdot k \leqslant n^2$, zatem $|W| \leqslant \frac{n^2}{k}$. Ponieważ jest to grupa częściej występujących liter, a takich liter jest mało, wykorzystajmy w niej BFS, a w grupie V sprawdzajmy wszystkie pary. Przeanalizujmy otrzymaną złożoność[3]:

$$O(\sum_{x \in V} c_x^2 + |W| \cdot n^2) \leqslant O(|V| \cdot k^2 + |W| \cdot n^2) \leqslant O(n^2 \cdot k + n^2 \cdot \frac{n^2}{k}) \overset{k=n}{\underset{min}{=}} O(n^3)$$

Otrzymana złożoność $O(n^3)$ jest zauważalnie lepsza od $O(n^4)$, które otrzymalibyśmy wykorzystując tylko jeden z algorytmów.

[2] [Odległość manhattańska] między punktami (x_1, y_1) i (x_2, y_2) wynosi $|x_1 - x_2| + |y_1 - y_2|$.

[3] Pominąłem tutaj analizę tego, że najgorszy przypadek grupy V jest wtedy, gdy składa się z $\frac{n^2}{k}$ liter występujących k razy.

2.2 Rosyjscy oligarchowie

Treść Mamy n ($1 \leqslant n \leqslant 10^6$) rosyjskich oligarchów oraz m ($1 \leqslant m \leqslant 10^6$) firm. Każda z firm ma swoją wartość v_i. Jako majątek oligarchy definiujemy sumę wartości firm, których jest właścicielem. Jedna firma może mieć wielu właścicieli, a do majątku każdego z nich wlicza się pełna kwota v_i. Na początku każdy oligarcha nie jest właścicielem żadnej z firm. Należy obsłużyć q ($1 \leqslant q \leqslant 10^5$) operacji jednego z trzech rodzajów:

1. Oligarcha j staje się właścicielem firmy i.
2. Wartość firmy i zmienia się na v'_i.
3. Zapytanie: podaj majątek oligarchy j.

Rozwiązanie Są dwa narzucające się sposoby obsługiwania zapytania:

1. Oligarcha przechowuje firmy, których jest właścicielem, i zwyczajnie sumujemy ich wartości.
2. Firmy przechowują oligarchów, którzy są ich właścicielami, i wraz z modyfikacjami aktualizują ich majątek.

Ponieważ liczba firm, których dany oligarcha jest właścicielem, może tylko rosnąć, możemy skupić się na oligarchach którzy mają dużo firm – oznaczmy ich zbiór przez W. Zdefiniujmy także zbiór firm, których oligarcha j jest właścicielem jako o_j.

Ustalmy, że j znajduje się w W jeżeli $|o_j| \geqslant k$ dla ustalonego k. Wyznaczmy ograniczenie na wielkość zbioru W: w każdym zapytaniu o_j może wzrosnąć co najwyżej o 1, a musi wzrosnąć k razy, aby w W pojawił się nowy element. Zatem:

$$|W| \leqslant \frac{q}{k}$$

Pozostałych oligarchów wpiszmy do zbioru V i do ich procesowania wykorzystajmy algorytm 1., wykorzystujący, że oligarchowie są właścicielami małej liczby firm ($|o_j| < k$). Oligarchów w zbiorze W będziemy obsługiwać algorytmem 2. Porównajmy czasy obsługiwania operacji dla obu zbiorów:

	1	2	3
V	$O(1)$	0	$O(k)$
W	$O(1)$	$O(\frac{q}{k})$	$O(1)$

Przyjmując, że każde z zapytań występuje mniej więcej tak samo często, możemy ustalić $k = \sqrt{q}$, otrzymując co najwyżej $O(\sqrt{q})$ na zapytanie.

Małe limity pozwalają nam na trochę wolności: możemy zauważyć, że liczy się tylko q zarówno oligarchów jak i firm (bo więcej nie może wystąpić w zapytaniach). Możemy wcześniej przydzielić oligarchów do zbiorów V i W (jeżeli wcześniej znamy zapytania), lub dynamicznie ich przepisywać w momencie, gdy o_j staje się dostatecznie duże.

3 Pierwiastkowa liczność

Do tej kategorii trafią wszystkie algorytmy, które korzystają z faktu, że czegoś jest pierwiastkowo wiele. Często w ich dowodzie znajduje się stwierdzenie „jak dużo może być mniejszych od pierwiastka, a ile większych od pierwiastka?".

3.1 Lemat harmoniczny

Twierdzenie. *W zbiorze* $\{\lfloor \frac{n}{1} \rfloor, \lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{3} \rfloor ..., \lfloor \frac{n}{n} \rfloor\}$ *znajduje się* $O(\sqrt{n})$ *unikatowych wartości.*

Dowód. Spójrzmy najpierw na i, dla których $\lfloor \frac{n}{i} \rfloor \geqslant \sqrt{n}$. Przekształcając:

$$\frac{n}{i} \geqslant \sqrt{n}$$
$$i \leqslant \sqrt{n}$$

Takich i jest co najwyżej $\sqrt{n}$. Jednakże pozostały nam jedynie $\lfloor \frac{n}{i} \rfloor < \sqrt{n}$, a ich jest co najwyżej $\sqrt{n} - 1$. Zatem sumarycznie występuje $O(\sqrt{n})$ unikatowych wartości.

□

Ten fakt możemy wykorzystać często w zadaniach z teorii liczb. Jeżeli mamy do czynienia z sumą, wystarczy tak ją przekształcić, by była ona postaci (dla pewnego wyrażenia $f(i)$, które umiemy policzyć w $O(1)$):

$$\sum_{i=1}^{n} f(i) \left\lfloor \frac{n}{i} \right\rfloor$$

Ponieważ $\lfloor \frac{n}{i} \rfloor$ jest malejące i zmienia się jedynie $O(\sqrt{n})$ razy, to możemy przyspieszyć liczenie tej sumy z $O(n)$ do $O(\sqrt{n})$, o ile umiemy liczyć sumę $f(i) + f(i+1) + ... + f(j-1) + f(j)$ w $O(1)$.

Aby to zrobić, szukamy każdego przedziału $[a, b]$, dla którego $\lfloor \frac{n}{i} \rfloor = x$, i do wyniku dodajemy $x \cdot (f(a) + f(a+1) + ... + f(b))$.

Przyda się przy tym trik, który pozwala łatwo znaleźć największą wartość j, dla którego $\lfloor \frac{n}{j} \rfloor = \lfloor \frac{n}{i} \rfloor$. (Trik ten pochodzi z [tego bloga na CF])

```
for (int i = 1, j; i <= n; i = j + 1)
{
        j = n / (n / i);
        // Dla każdego k w [i, j], n / i = n / k.
}
```

3.2 Podział całkowity

Twierdzenie. *Dana jest nieujemna liczba całkowita* n *oraz ciąg nieujemnych liczb całkowitych* a *taki, że* $\sum a_i = n$. *W ciągu* a *występuje* $O(\sqrt{n})$ *unikatowych wartości.*

Dowód. Istnieje klasyczny dowód tego twierdzenia korzystający z tego, że suma pierwszych k liczb naturalnych rośnie kwadratowo. Spróbujmy jednak wykorzystać powyższy sposób.

Skupmy się na $a_i \geqslant \sqrt{n}$. Niech indeksów spełniających tą nierówność będzie l. Jasnym jest, że $l\sqrt{n} \leqslant n$, zatem $l \leqslant \sqrt{n}$. Pozostają $a_i < \sqrt{n}$ – ich jest oczywiście co najwyżej $\sqrt{n}$. Sumarycznie mamy $O(\sqrt{n})$. □

To twierdzenie przydaje się przede wszystkim w dwóch rodzajach zadań: wykorzystujących problem plecakowy oraz takich, gdzie mamy pewną liczbę ciągów, których sumaryczna długość jest ograniczona.

3.2.1 Problem plecakowy

Problem plecakowy, w którym występuje $O(\sqrt{n})$ unikatowych wartości możemy rozwiązać w $O(W\sqrt{n})$, gdzie W stanowi największą rozpatrywaną wagę. Jest to opisane w tym blogu: `https://codeforces.com/blog/entry/59606`

W skrócie, w plecaku rozważamy oddzielnie każdą z unikatowych wag w_i, i w stanie dynamika utrzymujemy, ile co najmniej obecnie rozpatrywanej wagi potrzeba, by sumarycznie uzbierać daną wagę.

3.2.2 Sumaryczna długość ciągów

Drugi rodzaj często spotkamy w tekstówkach z wykorzystaniem hashów. Rozważmy dla przykładu taki problem: mamy dane m ciągów znaków (wzorców), których sumaryczna długość Σ wynosi co najwyżej 10^5, oraz tekst długości n ($n \leqslant 10^5$). Dla każdego z ciągów znaków chcemy znaleźć liczbę jego wystąpień w tekście. Można to rozwiązać na wiele sposobów w $O(nm + \Sigma)$, jednak chcielibyśmy poszukać rozwiązania, które naraz liczy wiele odpowiedzi.

Pomysł jest taki: możemy równocześnie rozpatrywać wszystkie wzorce tej samej długości: oznaczmy tę długość jako k. Liczymy ich hashe i utrzymujemy je w zbiorze. Przechodzimy przez wszystkie podsłowa długości k w tekście, sprawdzając czy hash każdego z nich znajduje się w zbiorze. Wykonamy co najwyżej $O(\sqrt{\Sigma})$ przejść, bo tyle jest k. Na podstawie tego pomysłu otrzymamy rozwiązanie w $O(n\sqrt{\Sigma} + \Sigma)$. Całość można optymalnie rozwiązać Aho-Corasick w $O(n + \Sigma)$.

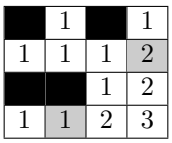
4 Batching

Ta technika pojawia się rzadziej, ale warto mieć ją na uwadze. Pojawia się, gdy chcemy jednocześnie obsługiwać modyfikacje i zapytania. Utrzymujemy przy tym pewną strukturę, którą jest trudno aktualizować. Chcemy zatem robić to jak najrzadziej: robimy to co pewną liczbę operacji. Czasem będziemy specjalnie zbierać modyfikacje nieuwzględnione w strukturze, a odpowiadając na zapytania, będziemy rozpatrywać je oddzielnie. Innym razem będziemy aplikować modyfikacje *ad-hoc* ale mniej efektywnym sposobem, który „zaśmieca" strukturę.

4.1 [*Guide to Competitive Programming*] Czarne pola

Treść Mamy daną planszę $n \times n$. Na początku wszystkie pola są białe, lecz z czasem będą przemalowywane na czarno. W podanej kolejności odwiedzamy wszystkie n^2 pól. Gdy odwiedzamy pole, należy wypisać jego odległość manhattańską do najbliższego czarnego pola, a następnie zamalować je na czarno.

Rozwiązanie Policzenie tej odległości dla każdego białego pola można wykonać w $O(n^2)$ równoległym BFS-em z pól czarnych, podobnie jak w zadaniu z literami. Alternatywnie, możemy przejrzeć wszystkie czarne pola w $O(m)$, gdzie m to ich liczba. Spróbujmy wykonać mieszane podejście: będziemy co k przemalowań wykonywać BFS-a, a nieuwzględnione w nim czarne pola będziemy rozważać oddzielnie (będzie ich co najwyżej k).



Na obrazku liczby to odległości po ostatnim BFS, czarne pola są w nim uwzględnione, a szare jeszcze nie.

Sumarycznie wykonamy $O(\frac{n^2}{k})$ przeszukiwań wszerz i $O(n^2k)$ razy spojrzymy na pole indywidualnie. Sumarycznie:

$$O(\frac{n^2}{k}n^2 + n^2k) \underset{k=n}{\overset{min}{=}} O(n^3)$$

5 Algorytm Mo

Na koniec bardzo prosty i przydatny algorytm, który dla danego ciągu umie odpowiadać na wiele zapytań na jego przedziałach *offline*. Wymaga on jedynie struktury działającej na pewnym przedziale $[l, r)$ tego ciągu, na której można wykonywać operacje:

- `get`: Podaj wynik dla obecnego przedziału
- `push`: Dodaj element na początek lub koniec (z indeksu $l-1$ lub r)
- `pop`: Usuń element z początku lub końca (z indeksu l lub $r-1$)

Mając taką strukturę i znając odpowiedź na pewnym przedziale $[l_1, r_1)$, poznanie odpowiedzi na $[l_2, r_2)$ to kwestia odpowiednio zwężenia i poszerzenia przedziału za pomocą `push` i `pop`. Jednak pesymistycznie, dla złośliwie dobranej kolejności odwiedzania zapytań, możemy potrzebować kwadratowo wiele wywołań. Oznaczmy długość ciągu jako n, a liczbę zapytań jako q.

Algorytm Mo tak uporządkuje zapytania, że wykonane zostanie co najwyżej $O(n\sqrt{q})$ operacji `push` i `pop`. Algorytm Mo nie robi nic poza zwróceniem nam porządku, w którym powinniśmy odwiedzać zapytania tak, żeby odpowiedzieć na nie wykonując jak najmniej „przesunięć" początku i końcu przedziału.

W ogólności, mając strukturę z odpowiednimi operacjami oraz komparator wykorzystujący algorytm Mo do sortowania, kod będzie wyglądał następująco:

```
sort(queries.begin(), queries.end(), mo_comparator);
size_t L = 0, R = 0; // [L, R)
for(auto q : queries)
{
    while(R < q.R)  push(R++);
    while(L > q.L)  push(--L);
    while(R > q.R)  pop(--R);
    while(L < q.L)  pop(L++);

    answer[q.index] = get();
}
```

Przechodząc do sedna działania algorytmu: tym razem zamiast dzielić ciąg, będziemy dzielić zapytania na podstawie zakresów, które obejmują. Oznaczmy te zakresy przez $[l_i, r_i)$ Dokładniej, ponownie weźmiemy parametr k i podzielimy zapytania w taki sposób, że jeżeli dwa zapytania o numerach i i j spełniają poniższy warunek, to są w tym samym bloku:

$$\left\lfloor \frac{l_i}{k} \right\rfloor = \left\lfloor \frac{l_j}{k} \right\rfloor$$

Nietrudno zauważyć, że bloków jest $O(\frac{n}{k})$. Po podzieleniu zapytań na bloki Mo posortuje je po r_i.

Spójrzmy, jak takie uporządkowanie wygląda (kolor niebieski oznacza że element był w przedziale również w poprzednim zapytaniu, czerwony oznacza, że został usunięty (`pop`), a zielony, że został dodany (`push`)):

$\boldsymbol{a_0}$	$\boldsymbol{a_1}$	$\boldsymbol{a_2}$	$\boldsymbol{a_3}$	$\boldsymbol{a_4}$	$\boldsymbol{a_5}$	$\boldsymbol{a_6}$	$\boldsymbol{a_7}$	$\boldsymbol{a_8}$	$\boldsymbol{a_9}$	$\boldsymbol{a_{10}}$	$\boldsymbol{a_{11}}$
a_0	a_1	$\boldsymbol{a_2}$	$\boldsymbol{a_3}$	$\boldsymbol{a_4}$	$\boldsymbol{a_5}$	$\boldsymbol{a_6}$	a_7	a_8	a_9	a_{10}	a_{11}
$\boldsymbol{a_0}$	$\boldsymbol{a_1}$	$\boldsymbol{a_2}$	$\boldsymbol{a_3}$	$\boldsymbol{a_4}$	$\boldsymbol{a_5}$	$\boldsymbol{a_6}$	$\boldsymbol{a_7}$	$\boldsymbol{a_8}$	$\boldsymbol{a_9}$	$\boldsymbol{a_{10}}$	$\boldsymbol{a_{11}}$
a_0	a_1	a_2	a_3	$\boldsymbol{a_4}$	$\boldsymbol{a_5}$	$\boldsymbol{a_6}$	$\boldsymbol{a_7}$	$\boldsymbol{a_8}$	a_9	a_{10}	a_{11}
a_0	a_1	a_2	a_3	a_4	$\boldsymbol{a_5}$	$\boldsymbol{a_6}$	$\boldsymbol{a_7}$	$\boldsymbol{a_8}$	a_9	a_{10}	a_{11}
a_0	a_1	a_2	a_3	a_4	$\boldsymbol{a_5}$	$\boldsymbol{a_6}$	$\boldsymbol{a_7}$	$\boldsymbol{a_8}$	$\boldsymbol{a_9}$	$\boldsymbol{a_{10}}$	a_{11}
a_0	a_1	a_2	a_3	a_4	a_5	a_6	$\boldsymbol{a_7}$	$\boldsymbol{a_8}$	$\boldsymbol{a_9}$	a_{10}	a_{11}

Pozostaje zatem rozważyć złożoność. Po pierwsze, aby przejść z przedziału $[l_i, r_i)$ do $[l_j, r_j)$ należy wykonać $|l_i - l_j| + |r_i - r_j|$ operacji (przesuwamy początek i koniec operacjami `push` i `pop`). Skupmy się na jednym z bloków i oznaczmy jego wielkość jako m. W tym bloku pomiędzy każdym z zapytań lewy koniec przesuniemy co najwyżej k razy – razem $O(mk)$. Poza tym, prawy koniec będziemy przesuwać ciągle w prawo (bo zapytania w tym samym bloku są posortowane względem prawego końca) – $O(n)$. Razem na każdy blok wykonujemy $O(mk + n)$ operacji. Sumując po wszystkich blokach:

$$O(qk + \frac{n^2}{k}) = O(q(k + \frac{n^2/q}{k})) \underset{k=\frac{n}{\sqrt{q}}}{\overset{min}{=}} O(n\sqrt{q})$$

Napisanie komparatora jest łatwe i przyjemne. Możemy różne bloki dalej przechowywać w tej samej tablicy zapytań i sortować wszystko razem – nie ma sensu sobie tego komplikować:

```
bool mo_comparator(query lhs, query rhs)
{
    if(lhs.L / k == rhs.L / k)
        return lhs.R < rhs.R;
    else
        return lhs.L < rhs.L;
}
```

5.1 Przykładowe zadanie: [SPOJ] D-query

Treść Mamy ciąg $a = (a_1, a_2, ..., a_n)$ i chcemy na nim obsługiwać zapytanie: dla przedziału $[l, r)$ podaj, ile unikatowych elementów się w nim znajduje.

Rozwiązanie Wystarczy zauważyć, że opracowanie odpowiednich `get`, `push`, `pop` jest bardzo proste. Trzeba przechowywać tablicę liczności elementów w obecnym przedziale i modyfikować wynik, gdy liczność staje się zerowa lub niezerowa.

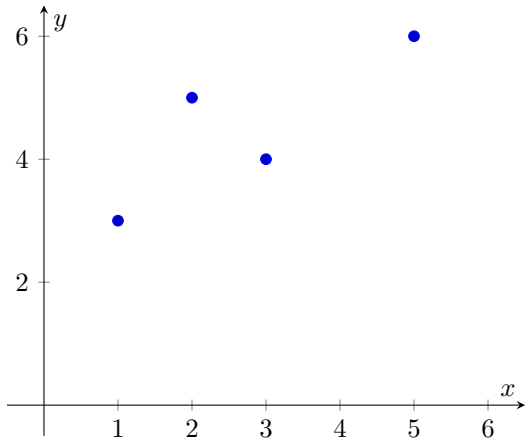
5.2 Ulepszenie komparatora

W komparatorze można dodać pewne kreatywne ulepszenie: ponieważ po rozpatrzeniu zapytań z bloku prawy koniec przedziału będzie daleko z prawej, możemy prawy koniec przedziałów zapytań z następnego bloku sortować w kolejności odwrotnej. Najprościej to osiągnąć, jeżeli co drugi blok posortujemy rosnąco, a pozostałe malejąco – prawie zawsze polepszy nam to czas działania.

```
bool better_mo_comparator(query lhs, query rhs)
{
    if(lhs.L / k == rhs.L / k)
        return (lhs.L/k) & 1 ? (lhs.R > rhs.R) : (lhs.R < rhs.R);
    else
        return lhs.L < rhs.L;
}
```

5.3 Mo jako rozwiązywanie TSP

Zauważmy, że wcześniej stwierdziliśmy, że przejście z zapytania o przedziale $[l_i, r_i)$ do $[l_j, r_j)$ kosztuje nas $|l_i - l_j| + |r_i - r_j|$ operacji. Taka odległość między dwoma zapytaniami jest analogiczna do odległości między punktami w odległości manhattańskiej (metryce miejskiej). Zamiast przedziałów możemy sobie zatem wyobrazić zapytania jako punkty.



Zapytania $[1, 3)$, $[2, 5)$, $[3, 4)$, $[5, 6)$

A zatem Mo tak na prawdę wyznacza kolejność, w której należy odwiedzić te punkty, aby przebyta odległość była jak najmniejsza – rozwiązuje przypadek problemu komiwojażera (Travelling Salesman Problem). Możemy zatem wykorzystać algorytmy aproksymacyjne do TSP (bo dokładne rozwiązanie jest NP), aby polepszyć działanie naszego Mo. Tutaj można przeczytać o przykładzie wykorzystania [krzywej Hilberta] do optymalizacji Mo: `https://codeforces.com/blog/entry/61203`

5.4 Algorytm Mo z aktualizacjami

Istnieje pewna modyfikacja Mo która pozwala na pojawianie się nie tylko zapytań, ale również aktualizacji. Dalej musimy wykonywać je *offline*.

Próbując jednocześnie obsługiwać zapytania i aktualizacje zwykłym Mo napotkamy problem: Mo może tak przetasować zapytania, że pojawią się one w kolejności niezgodnej z występowaniem aktualizacji, i musielibyśmy je wielokrotnie wykonywać i cofać. Musimy to jakoś rozwiązać.

Zauważmy, że każdego zapytania dotyczy pewien prefiks t aktualizacji – tych, które wystąpiły, zanim pojawiło się zapytanie. Powiedzmy, że nasza struktura obsługuje `update` – dodanie aktualizacji – oraz `rollback` – wycofanie ostatniej aktualizacji[4]. Zatem możemy przesuwać nasz prefiks do $t+1$ lub wycofać go do $t-1$. Nasuwa się pewien pomysł: wróćmy do metafory zapytań jako punktów i dodajmy trzeci wymiar – czas t. Teraz `update` i `rollback` są analogiami `push` i `pop` – przesuwają nasz punkt-stan w czasie.

Wyznaczmy wielkość bloku k i ponownie dzielmy zapytania na bloki względem l ($\lfloor \frac{l}{k} \rfloor$), lecz tym razem pozostają nam jeszcze dwa wymiary. Podzielmy je zatem ponownie względem r ($\lfloor \frac{r}{k} \rfloor$) – pozostał nam tylko czas t, który możemy posortować rosnąco. Kod algorytmu wygląda analogicznie.

```
sort(queries.begin(), queries.end(), umo_comparator);
size_t L = 0, R = 0, T = 0;
for(auto q : queries)
{
    while(T < q.T)  update(T++);
    while(T > q.T)  rollback(--T);
    while(R < q.R)  push(R++);
    while(L > q.L)  push(--L);
    while(R > q.R)  pop(--R);
    while(L < q.L)  pop(L++);
    answer[q.index] = get();
}
```

Pozostaje przeanalizować złożoność i dobrać parametr k. Zacznijmy od tego, że bloków jest $\left(\frac{n}{k}\right)^2$ ($\frac{n}{k}$ l-bloków, $\frac{n}{k}$ r-bloków). Musimy pamiętać, żeby rozróżnić liczbę aktualizacji (czas) od liczby zapytań, jeżeli tych pierwszych jest mało. Żeby sobie ułatwić, przyjmijmy że jest ich mniej więcej tyle samo – q – i że $n \approx q$, zatem $k < q$. Po pierwsze, $(\frac{n}{k})^2$ razy będziemy przesuwać się do następnego bloku w $O(k+q) = O(q)$. Natomiast wewnątrz bloku wykonamy $O(q)$ przesunięć czasu i między każdą parą zapytań wykonamy $O(k)$ przesunięć przedziału.

Podsumowując:

$$O(\frac{n^2 q}{k^2} + qk) \underset{k=n^{\frac{2}{3}}}{\overset{min}{=}} O(qn^{\frac{2}{3}})$$

[4] W szczególności `rollback` może być zaimplementowane jako szczególny przypadek `update`.

6 *Zadanka do kminienia*

- Lista zadań na cp-algorithms: `https://cp-algorithms.com/data_structures/sqrt_decomposition.html`
- Lista parunastu zadań z Codeforces: `https://codeforces.com/blog/entry/23005`

Trochę pogrupowanych technikami:

6.1 Rozkład ciągu

- [SPOJ] Give Away
- [Codeforces] Holes

6.2 Podział algorytmu

- [XXIV OI] Kontenery
- [XXII OI] Odwiedziny

6.3 Pierwiastkowa liczność

- [PREOI 2018] (Kółko Olimpijskie 2018/19) Pawian
- [ONTAK 2014] Bajka o dzielnikach
- [XVII ILOCAMP] (Treść) Totolotek
- [XVIII ILOCAMP] (Treść) Obiad 2

6.4 Batching

- [Codeforces] Serega and Fun – także rozkład ciągu
- [ONTAK 2014] Taśmy produkcyjne

6.5 Algorytm Mo

- [ONTAK 2011] Potężna tablica
- [Codeforces] XOR and Favorite Number
- [Codeforces] Points on Plane