Uniwersytet Jagielloński

Informatyka Analityczna

Problem Range Mode Query

Leopold Kozioł

Opiekun pracy
dr Lech Duraj

wrzesień 2021

Streszczenie

W problemie Range Mode Query, dana jest na wejściu n-elementowa tablica liczb naturalnych A oraz q przedziałów tablicy A. Dla każdego danego na wejściu przedziału chcemy policzyć jego dominantę, czyli element który najczęściej występuje na danym fragmencie. Praca ta ma celu implementację i porównanie różnych algorytmów i struktur danych rozwiązujących problem Range Mode Query.

Spis treści

1 Wprowadzenie

1.1 Wstęp

W problemie Range Mode Query, albo w skrócie RMQ, dana jest na wejściu n-elementowa tablica liczb naturalnych A oraz q przedziałów tablicy A. Dla każdego danego na wejściu przedziału chcemy policzyć jego dominantę, czyli element który najczęściej występuje na danym fragmencie.

Łatwo zaproponować rozwiązanie naiwne działające w czasie liniowym od długości przedziału. Pierwsze usprawnienie względem algorytmu naiwnego zostało zaproponowane przez Krizanc i innych w 2005 roku w pracy [4]. Autorzy pokazali deterministyczny algorytm rozwiązujący problem RMQ w czasie $\mathcal{O}(n^e \log n)$ oraz czasie inicjalizacji $\mathcal{O}(n^{2-2e})$, gdzie $e \in (0, 1/2]$.

W 2014 roku Timothy M. Chan i inni w pracy [1] pokazali warunkowe ograniczenie dolne dla problemu RMQ za pomocą redukcji z mnożenia macierzy boolowskich. Każdy algorytm rozwiązujący problem RMQ ma czas inicjalizacji $\Omega(n^{\omega/2})$ lub czas zapytania $\Omega(n^{\omega/2-1})$, gdzie ω to wykładnik algorytmu mnożenia macierzy. Obecnie nie jest znany algorytm kombinatoryczny mnożenia macierzy boolowskich, który działałby w czasie $\mathcal{O}(n^{3-\epsilon})$, co oznacza, że przy obecnym stanie wiedzy nie potrafimy rozwiązać problemu RMQ w czasie inicjalizacji $\mathcal{O}(n^{3/2-\epsilon})$ oraz czasie zapytania $\mathcal{O}(n^{1/2-\epsilon})$ przy użyciu czysto kombinatorycznych technik.

W tej samej pracy autorzy zaproponowali algorytm o czasie zapytania $\mathcal{O}(\sqrt{n})$ oraz czasie inicjalizacji $\mathcal{O}(n\sqrt{n})$. Później pokazali jego usprawnienie w standardowym modelu RAM, o czasie zapytania $\mathcal{O}(\sqrt{n/w})$ oraz czasie inicjalizacji $\mathcal{O}(n\sqrt{nw})$, gdzie w oznacza długość słowa maszynowego. Oba algorytmy zużywają liniową pamięć.

Celem tej pracy jest opis, implementacja i analiza porównawcza wyżej wymienionych algorytmów i struktur danych dla problemu RMQ. W szczególności zajmujemy się tylko algorytmami, które zużywają liniową pamięć. W rozdziale 2 prezentujemy algorytm naiwny w dwóch wariantach, a następnie w rozdziale 3 pokazujemy prosty algorytm offline bazujący na algorytmie naiwnym. Następnie przechodzimy do prezentacji algorytmów z prac wspomnianych powyżej, zaczynając od algorytmu [4] w rozdziale 4. Następne 4 rozdziały poświęcone są pracy [1], w rozdziale 5 pokazujemy strukturę danych odpowiadającą na zapytanie RMQ w czasie $\mathcal{O}(\sqrt{n})$, a w rozdziale 6 przedstawiamy jej usprawnienie działające w modelu RAM. Następnie pokazujemy w rozdziale 7 prostą strukturę danych, której czas zapytania uzależniony jest od liczby unikalnych elementów tablicy A. W rozdziale 8 pokazujemy strukturę danych będącą połączeniem struktur opisanych w rozdziałach 6 oraz 7. Na końcu w rozdziale 9 prezentujemy wyniki ewaluacji naszych implementacji powyższych algorytmów i struktur danych. Wszystkie implementacje są dostępne pod linkiem `https://github.com/v3ctor/ubiquitous-journey/tree/master/code`.

1.2 Terminologia i notacja

Tablice Przez $A[1:n]$ będziemy oznaczali tablicę n-elementową A indeksowaną od 1, a przez $A[k]$ k-ty jej element. Podobnie będziemy oznaczać spójny fragment tablicy A od i-tego do j-tego elementu włącznie przez $A[i:j]$ dla $i \leq j$. Gdy $i > j$, $A[i:j]$ oznacza pusty fragment. Dla przedziału $A[i:j]$, indeks i będziemy nazywać *lewym końcem przedziału*, a j *prawym końcem przedziału*.

Częstotliwość Dla tablicy A, przez $F_x^A(i,j)$ oznaczamy *częstotliwość* liczby x na przedziale $A[i:j]$, to znaczy liczbę elementów $A[k]$ równych x dla $k \in \{i, i+1, \ldots, j\}$. Formalnie $F_x^A(i,j) = |\{k \mid A[k] = x \wedge k \in \{i, i+1, \ldots, j\}\}|$. Ponadto definiujemy przez $F^A(i,j) = \max_{k \in \{A[i], A[i+1], \ldots, A[j]\}} \{F_k^A(i,j)\}$. Zamiast *częstotliwość* będziemy też wymiennie używać nazwy *liczba wystąpień*.

Dominanta Element $A[k]$ tablicy $A[1:n]$ nazwiemy *dominantą*, jeżeli $F_{A[k]}^A(1,n) = F^A(1,n)$, innymi słowy $A[k]$ jest dominantą, gdy jest elementem maksymalnym pod względem częstotliwości w tablicy A. Analogicznie definiujemy dominanty na przedziałach tablic.

Nazwy struktur danych W przypadku, gdy struktura danych nie jest nazwana w pracy odnosimy się do niej poprzez inicjały autorów pracy. Ponadto część struktur jest parametryzowana liczbą od której to zależy jej złożoność pamięciowa lub czasowa. Dla przykładu struktura $CDLMW(s)$ jest parametryzowana liczbą s.

Od tego momentu będziemy przez cały czas używać nazwy A na określenie tablicę wejściowej dla algorytmów i struktur danych. Ponadto oznaczamy przez Δ liczbę unikalnych elementów w tablicy A.

1.3 Redukcja Rank Space

Niech $D(x)$ oznacza zbiór elementów tablicy $A[1 : n]$ w przedziale $A[1 : x]$. Tablicę $B[1 : n]$ nazwiemy *rank-space-redukcją*, albo w skrócie *redukcją* tablicy A, gdy istnieje pewna bijekcja $f : D(n) \to \{1, 2, \ldots, |D(n)|\}$ taka, że $B[i] = f(A[i])$ dla każdego $i \in \{1, 2, \ldots, n\}$. Pracując na tablicy zredukowanej można efektywniej wykonać na niej pewne operacje w porównaniu do tablicy niezredukowanej. Dlatego przy omawianiu większości algorytmów w tej pracy będziemy zakładać, że tablica na której pracujemy jest w tej postaci. Gdy nie będziemy wymagać, aby tablica była zredukowana napiszemy o tym explicite. W praktyce nie możemy założyć, że podana tablica na wejściu jest zredukowana dlatego podajemy liniowy, randomizowany algorytm konwersji tablicy $A[1 : n]$ do jej zredukowanej postaci.

Algorithm 1 Randomizowany algorytm rank-space-redukcji

```
1: function RankSpaceReduction(A[1 : n])
2:     Niech B[1 : n] to tablica n-elementowa
3:     uniq ← HashMap()
4:     for k ← 1, . . . , n do
5:         if uniq.contains(A[i]) then
6:             B ← uniq.get(A[i])
7:         else
8:             B ← uniq.size()+1
9:             uniq.insert(A[i], uniq.size()+1)
       return B, uniq
```

Utrzymujemy dwa niezmienniki na końcu pętli:

1. *uniq* reprezentuje bijekcję ze zbioru $D(k)$ do $\{1, 2, \ldots, |D(k)|\}$.
2. $B[1 : k]$ jest redukcją $A[1 : k]$ powstałą przez aplikacje bijekcji *uniq* do każdego elementu.

Na końcu zwracamy zredukowaną tablicę B oraz hash-mapę *uniq* która po odwróceniu kluczy z wartościami pozwoli na odtworzenie tablicy wejściowej A.

2 Algorytm naiwny

Pierwszym i zarazem najprostszym algorytmem jaki opiszemy jest algorytm naiwny w dwóch wariantach. Pierwszy wariant nie będzie wymagał aby tablica wejściowa była rank-space-zredukowana zredukowana, w przeciwieństwie do drugiego.

2.1 Wariant pierwszy

2.1.1 Konstrukcja

Algorytm naiwny nie wymaga żadnego przetwarzania wstępnego.

2.1.2 Algorytm zapytania

Algorithm 2 Algorytm NAIWNY

```
1:  function NaiveQuery(i,j)
2:      count ← HashMap()
3:      mfreq ← 0
4:      mvalue ← 0
5:      for k ← i, ..., j do
6:          freq ← 1
7:          if count.contains(A[k]) then
8:              freq ←count.get(A[k])+1
9:          count[A[k]] ← freq
10:         if freq > mfreq then
11:             mfreq ← freq
12:             mvalue ← A[k]
        return mfreq, mvalue
```

Dla danego zapytania o dominantę na przedziale $A[i : j]$ będziemy używać hash-mapy do zliczania częstotliwości jego elementów. Przeglądamy każdy element $A[k]$ dla $k \in \{i, i+1, \ldots, j\}$. Gdy elementu $A[k]$ nie ma w hash-mapie dodajemy go do niej z wartością 1, a w przeciwnym przypadku zwiększamy wartość elementu w hash-mapie o 1. Podczas iteracji przechowujemy maksymalną wartość zawartą w hash-mapie oraz odpowiadający jej klucz. Po przejrzeniu wszystkich elementów hash-mapa będzie zawierała częstotliwości wszystkich wartości na przedziale $A[i : j]$. Zatem zapisane maksimum to jest częstotliwość dominanty, a odpowiadający klucz to dominanta na na tym przedziale.

2.1.3 Analiza złożoności pamięciowej

Dla zapytania o dominantę na przedziale $A[i : j]$ elementów w hash-mapie jest maksymalnie $j - i + 1$. Z drugiej strony elementów w hash-mapie nigdy nie będzie więcej niż Δ. Zatem łącznie algorytm zapytania zużywa $\mathcal{O}(\min(j - i, \Delta))$ pamięci.

2.1.4 Analiza złożoności czasowej

Podczas pojedynczej iteracji pętli używamy $\mathcal{O}(1)$ operacji na hash-mapie, oraz $\mathcal{O}(1)$ operacji arytmetycznych. Iteracji pętli jest $j - i + 1$. Zakładając, że wszystkie operacje na hash-mapie trwają zamortyzowany czas $\mathcal{O}(1)$. Cały algorytm zajmuje $\mathcal{O}(j - i)$ czasu.

2.1.5 Implementacja

Używamy `unordered_map` z standardowej biblioteki C++ jako hash-mapy w tym wariancie algorytmu.

2.2 Wariant drugi

Zauważamy, że w praktyce operacje na hash-mapie, mimo że teoretycznie w amortyzowanym czasie stałym, są wolniejsze niż operacje na tablicy. Zakładając że tablica wejściowa jest rank-space-zredukowana, możemy zastąpić hash-mapę tablicą o rozmiarze Δ.

2.2.1 Algorytm zapytania

Algorytm zapytania wygląda prawie identycznie jak w wariancie pierwszym. Zastępujemy hash-mapę *count* przez tablicę o rozmiarze Δ. Ponadto na samym końcu algorytmu przed zwróceniem dominanty zerujemy elementy tablicy $count[A[k]]$ dla $k \in \{i, i+1, \ldots, j\}$. Złożoność czasowa algorytmu jest taka sama jak w wariancie pierwszym.

2.2.2 Analiza złożoności pamięciowej

W tym wariancie count zawsze zajmuje $\mathcal{O}(\Delta)$ pamięci, zatem cała struktura danych łącznie zużywa $O(\Delta)$ pamięci.

3 Algorytm Offline

Najpierw wprowadzimy strukturę danych MultiMode, która umożliwi nam dla tablicy A obliczenie dominanty $A[k:l]$ z dominanty $A[i:j]$ w czasie $\mathcal{O}((|k-i|+|l-j|)\log n)$. Następnie pokażemy algorytm[1] używający tej struktury danych, aby obsłużyć zbiór zapytań o dominanty $Q = \{Q_1, Q_2, \ldots, Q_q\}$ w czasie $\mathcal{O}(n\sqrt{q}\log n)$, gdzie $q = |Q|$.

3.1 Struktura MultiMode

Pokażemy strukturę danych MultiMode, która będzie utrzymywać pewien multizbiór M i wspierać na nim operacje:

- insert(x) – wstaw x do multizbioru M
- remove(x) – usuń x z multizbioru M
- query() – zwróć dominantę i jej częstotliwość

Wszystkie operacje działają w czasie $\mathcal{O}(\log n)$, ponadto możemy zbudować strukturę MultiMode w $\mathcal{O}(1)$ czasu. Struktura danych będzie reprezentować multizbiór M przez hash-mapę *count* oraz binarne drzewo wyszukiwań $freqval$. Hash-mapa *count* przyporządkowuje każdemu unikalnemu elementowi M jego liczbę wystąpień. Binarne drzewo $freqval$ zawiera pary $(count[v], v)$ uporządkowane po $count[v]$ dla każdego klucza v hash-mapy *count*. Dzięki $freqval$ będziemy w stanie zapytać o klucz hash-mapy *count* o maksymalnej wartości, co pozwoli nam na zwrócenie dominanty w czasie $\mathcal{O}(\log n)$.

Algorithm 3 Operacja insert

```
1: function INSERT(x)
2:     freq ← 0
3:     if count.contains(A[x]) then
4:         freq ← count.get(A[x])
5:         freqval.remove((freq, A[x]))
6:     freq ← freq+1
7:     freqval.insert((freq, A[x]))
8:     count[A[x]] = freq
```

Algorithm 4 Operacja remove

```
1: function REMOVE(x)
2:     freq ← count.get(A[x])
3:     freqval.remove((freq, A[x]))
4:     freq ← freq −1
5:     if freq ≠ 0 then
6:         freqval.insert((freq, A[x]))
7:     count[A[x]] = freq
```

Algorithm 5 Operacja query

```
1: function QUERY(x)
2:     if freqval.size() = 0 then
3:         return (0, 0)
       return freqval.max()
```

Operacja insert Gdy dodajemy element x do multizbioru M liczba jego wystąpień zwiększa się o 1, zatem musimy zwiększyć wartość x w hash-mapie *count*, robimy to w linijce 8. Po zaktualizowaniu *count* musimy poprawić BST $freqval$ tak, aby odpowiadało zmienionej hash-mapie *count*. Jeżeli element x należał wcześniej do multizbioru M to musimy usunąć starą parę w $freqval$ i wstawić nową, robimy to odpowiednio w linijkach 5 oraz 7. W drugim przypadku, gdy x jest nowym unikalnym elementem M wystarczy tylko wstawić nową parę do $freqval$, robimy to w linijce 7.

Operacja remove Zakładamy. że chcemy usunąć już istniejącą wartość x z multizbioru M dlatego nie rozważamy przypadku, gdy x nie należy do M. Operacja remove jest analogiczna w implementacji do operacji insert.

[1] Podany algorytm jest czasem nazywany "rozbiciem pierwiastkowym", albo "algorytmem Mo"

Operacja QUERY Jeżeli M jest pusty to zwracamy parę (0, 0), gdzie pierwszy element reprezentuje liczbę wstąpień, a drugi dominantę. W przeciwnym wypadku zwracamy największy element znajdujący się w BST $freqval$.

Algorithm 6 TRANSFORM

```
1:  function TRANSFORM(mm,i,j,k,l)
2:      while j < l do
3:          mm.insert(j + 1)
4:          j ← j + 1
5:      while i < k do
6:          mm.remove(i)
7:          i ← i+1
8:      while i > k do
9:          mm.insert(i − 1)
10:         i ← i − 1
11:     while j > l do
12:         mm.remove(j)
13:         j ← j − 1
```

Transformacja przedziałów Załóżmy, że używamy struktury MULTIMODE, aby reprezentować spójny przedział tablicy $A[i : j]$. Możemy zamienić reprezentowany przedział $A[i : j]$ na dowolny inny $A[k : l]$ odpowiednio wywołując operacje INSERT oraz REMOVE. Robimy to w taki sposób, aby w każdym kroku struktura MULTIMODE reprezentowała spójny przedział tablicy A. Dla przykładu gdy $j < l$ wywołujemy po kolei INSERT(x) dla $x = i + 1, i + 2, \ldots, l$, Robimy to w linijkach 2–4. Pozostałe 3 przypadki obsługujemy analogicznie.

Alternatywna implementacja MULTIMODE Alternatywnie multizbiór M można utrzymywać poprzez tablicę $G[1 : n]$ list, gdzie $G[i]$ przechowuje w liście wszystkie unikalne elementy o częstotliwości i. Operację INSERT(x) można zrealizować w czasie stałym, poprzez przepięcie elementu do następnej listy, analogicznie można obsłużyć operację REMOVE. Przy realizacji obu operacji utrzymujemy największy indeks niepustej listy, co pozwala nam na obsługę QUERY w czasie stałym. Ponadto, aby znaleźć odpowiadającą listę elementu x spamiętujemy dla każdego unikalnego elementu w której jest liście w tablicy lub hash-mapie. Nazywamy tą wersję MULTIMODELIST, a oryginalną MULTIMODEBST.

3.2 Konstrukcja

Algorytm offline nie wymaga żadnego przetwarzania wstępnego.

3.3 Algorytm zapytania

Algorithm 7 ALGORYTM ZAPYTANIA OFFLINE

```
1: function OfflineQuery(Q)
2:     Niech modes to tablica q-elementowa.
3:     Niech U to ciąg zapytań z zbioru Q w porządku zdefiniowanym poniżej.
4:     mm ← MultiMode()
5:     i ← 0
6:     j ← −1
7:     for all Q_i ∈ U do
8:         transform(mm, i, j, l_i, r_i)
9:         i ← l_i
10:        j ← r_i
11:        modes[i] ← mm.query()
12:    return modes
```

Załóżmy, że chcemy odpowiedzieć na zbiór zapytań $Q = \{Q_1, Q_2, \ldots, Q_q\}$ o dominanty tablicy A. Oznaczmy przez $A[l_i : r_i]$ przedział o jaki pyta zapytanie Q_i. Niestety nie możemy wywołać TRANSFORM po kolei dla zapytań $Q_1, Q_2, \ldots, Q_q$. Ponieważ pesymistycznie złożoność czasowa będzie $\mathcal{O}(qn \log n)$. Aby temu zaradzić posortujemy zapytania w odpowiedni sposób. Dzielimy zbiór zapytań Q na $s = \lceil \sqrt{q} \rceil$ rozłącznych części. Niech P_i oznazna i-tą część, $i \in \{1, 2, \ldots, s\}$. Definiujemy P_i jako $\{Q_k \mid \lceil l_k s/n \rceil = i\}$. Innymi słowy do P_i należą zapytania Q_k których lewy koniec l_k leży w przedziale $((i-1)n/s, in/s]$. Oznaczamy przez p_k numer części do której należy zapytanie Q_k ($Q_k \in P_{p_k}$). Niech U to ciąg zapytań z zbioru Q uporządkowany leksykograficznie po parze $(p_i,\ r_i)$, równoważnie możemy zdefiniować porządek $Q_i < Q_j \iff p_i < p_j \vee (p_i = p_j \wedge r_i < r_j)$.

3.4 Analiza złożoności czasowej

Inicjalizacje zmiennych Ciąg U otrzymujemy przez uporządkowanie zbioru Q. Tworzymy ten ciąg za pomocą algorytmu sortowania przez scalanie, co zajmie $\mathcal{O}(q \log q)$ czasu.

Pętla w linijce 7 Zauważmy, że wszystkie zapytania należące do części P_k dla pewnego k tworzą spójny fragment ciągu U. Dlatego podzielimy analizę na dwa przypadki. Pierwszy, ile zajmuje czasu przetworzenie zapytań w jednej części P_k, oraz drugi ile czasu zajmuje przejście z jednej części do drugiej.

Pierwszy przypadek Gdy zapytania Q_i oraz Q_{i+1} należą do tej samej części P_k wiemy, że ich lewe końce są oddalone od siebie o maksymalnie n/s z definicji części. Zatem funkcja TRANSFORM w linijce 8 użyje co najwyżej n/s operacji INSERT lub REMOVE do przesunięcia lewego końca. Łącznie dla ustalonej części P_k przesuwanie lewych końców zajmie $\mathcal{O}(|P_k| n/s \log n)$ czasu. Z drugiej strony wiemy, że prawe końce wszystkich zapytań w P_k będą obsługiwane w niemalejące kolejności. Zatem łącznie funkcja TRANSFORM wywoła co najwyżej $\mathcal{O}(n)$ operacji INSERT dla ustalonego P_k. Zatem czas spędzony w przypadku pierwszym możemy oszacować przez $\mathcal{O}(\sum_{k=1}^{s} [|P_k| n/s \log n + n \log n]) = \mathcal{O}(qn/s \log n + ns \log n) = \mathcal{O}((q/s + s) n \log n)$

Drugi przypadek W tym przypadku szacujemy, że funkcja TRANSFORM nie przesunie końców przedziału $A[i : j]$, o więcej niż $2n$. Ponieważ zapytania z dowolnej ustalonej części P_k są spójnym fragmentem w ciągu U, zapytania Q_i oraz Q_{i+1} będą w różnych częściach co najwyżej $s - 1$ razy. Łącznie w przypadku drugim możemy oszacować czas przez $\mathcal{O}(sn \log n)$.

Podsumowanie Łącząc wszystkie powyższe czasy dostajemy ograniczenie $\mathcal{O}(q \log q + (q/s + s + 1) n \log n)$. Podstawiając $s = \lceil \sqrt{q} \rceil$ otrzymujemy $\mathcal{O}(q \log q + n\sqrt{q} \log n) \subset \mathcal{O}(n\sqrt{q} \log n)$. Zamortyzowany czas dla jednego zapytania zajmuje $\mathcal{O}(n/\sqrt{q} \log n)$ czasu. Używając alternatywnej implementacji MULTIMODELIST z czasu złożoności znika nam czynnik $\log n$ przez co łączny czas dla tej wersji wynosi $\mathcal{O}(q \log q + n\sqrt{q})$ oraz amortyzowany czas dla pojedynczego zapytania wynosi $\mathcal{O}(\log q + n/\sqrt{q})$.

3.5 Analiza złożoności pamięciowej

Struktura MultiModeBST zajmuje co najwyżej $\mathcal{O}(\Delta)$ pamięci. Ciąg zapytań U zajmuje $\mathcal{O}(q)$ pamięci, oraz jego obliczenie wymaga użycia algorytmu sortowania, który też używa $\mathcal{O}(q)$ pamięci. Potrzebujemy tablicy *modes*, aby zapisać dominanty co zajmuje $\mathcal{O}(q)$ pamięci. Funkcja TRASFORM używa łącznie $\mathcal{O}(1)$ pamięci wliczając to wywoływane operacje INSERT oraz REMOVE. Łącznie zużywamy $\mathcal{O}(q)$ pamięci. Alternatywna implementacja MultiModeLIST utrzymuje tablicę o rozmiarze n, używając tej wesji łącznie zużywamy $\mathcal{O}(n+q)$ pamięci.

3.6 Implementacja

Jako hash-mapy użyliśmy `unordered_map` z standardowej bilbioteki c++ oraz użyliśmy `set` jako binarnego drzewa wyszukiwań, również z standardowej biblioteki c++. Ponadto liczbę s zaokrąglamy do najbliższej potęgi dwójki. Pozwala to nam na szybkie obliczenie do której części należy zapytanie, co znacząco przyśpiesza posortowanie zbioru zapytań Q.

4 Struktura danych KMS(s)

Pokażemy strukturę $KMS(s)$ z pracy [4], parametryzowaną po s, która dla statycznej tablicy $A[1:n]$ będzie wspierała poniższe operacje:

1. COUNT(x,i,j) – zwraca $F_x^A(i,j)$ w czasie $\mathcal{O}(\log n)$
2. QUERY(i,j) – zwraca dominantę i jej częstotliwość na przedziale $A[i:j]$ w czasie $\mathcal{O}(n/s \log n)$

4.1 Konstrukcja struktury KMS(s)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	1	2	3	1	3	4	3	3	2	3	2	1	2	2	4	1
	B_0				B_1				B_2				B_3			

S

i \ j	0	1	2	3
0	2	4	5	5
1	0	3	4	4
2	0	0	2	4
3	0	0	0	2

S'

i \ j	0	1	2	3
0	1	3	3	3
1	0	3	3	3
2	0	0	2	2
3	0	0	0	2

Q_1	Q_2	Q_3	Q_4
1	2	3	6
4	9	5	15
12	11	7	
16	13	8	
	14	10	

Rysunek 1: Przykładowa tablica $A[1:16]$ z liczbą unikalnych elementów $\Delta = 4$ i podziałem na $s = 4$ bloki, każdy o rozmiarze $t = 4$. Ponadto zaznaczamy zapytanie o dominantę $A[8:14]$ z rozbiciem na prefiks $A[8:8]$, środek $A[9:12]$ oraz sufiks $A[13:14]$, które są zaznaczone odpowiednio kolorami zielonym, żółtym i zielonym.

Tworzymy Δ tablic $Q_x[0 : F_x^A(1,n)-1]$ dla każdego unikalnego elementu $x \in \{1,2,\ldots,\Delta\}$. Tablica Q_x na i-tej pozycji przechowuje indeks $(i+1)$-tego wystąpienia wartości x w tablicy A, innymi słowy $Q_x[i] = j \iff A[j] = x \wedge F_x^A(1,j) = i+1$. Ponadto dzielimy tablicę A na s bloków, każdy, prócz ostatniego, o rozmiarze $t = \lceil n/s \rceil$. i-ty blok B_i dla $i = 0,1,\ldots,s-1$ reprezentuje przedział $A[it+1, \min(n,(i+1)t)]$. Dla każdej pary bloków B_i, B_j przechowujemy w tablicy dwuwymiarowej $S[i,j]$ dominantę multizbioru $B_{i,j} = B_i \cup B_{i+1} \cup \cdots \cup B_j$. Ponadto przechowujemy tablicę dwuwymiarową $S'[i,j]$, która odpowiada częstotliwości dominanty $S[i,j]$ w multizbiorze $B_{i,j}$.

4.2 Operacja COUNT

Algorithm 8 Operacja COUNT

```
1: function COUNT(x,i,j)
2:     m ← binary_search(Q_x, i)
3:     l ← binary_search(Q_x, j + 1)
4:     return l − m
```

Najpierw pokażemy jak można dla dowolnej wartości $x \in \{1,2,\ldots,\Delta\}$ obliczyć jej częstotliwość na przedziale $A[i:j]$ w czasie $\mathcal{O}(\log n)$. Oznaczmy przez $x_0 < x_1 < \cdots < x_{k-1}$ indeksy wszystkich wystąpień wartości x w przedziale $A[i:j]$. Ponieważ tablica Q_x jest rosnąca, wystąpienia $x_0,\ldots,x_{k-1}$ odpowiadają spójnemu fragmentowi tablicy $Q_x[l : l+k-1]$, gdzie $Q_x[l+o] = x_o$ dla $o \in \{0,1,\ldots,k-1\}$. Możemy łatwo znaleźć l używając wyszukiwania binarnego, jest to pierwsza wartość w tablicy Q_x większa bądź równa od i. Analogicznie szukamy indeksu m następnika x_{k-1} w tablicy Q_x, jest to pierwsza wartość większa niż j. W przypadku kiedy $k = 0$ przyjmujemy $l = m = 1$, oraz w przypadku kiedy x_{k-1} jest ostatnim wystąpieniem wartości x w tablicy A przyjmujemy $m = F_x^A(1,n)$. Teraz łatwo zauważyć, że $F_x^A(i,j) = m-l$. Dla przykładu rozważmy wartość 2 na przedziale $A[8:14]$ w rysunku 1. dwa występuje 4 razy na tym przedziale, na indeksach $x_0 = 9, x_1 = 11, x_2 = 13, x_3 = 14$, które odpowiadają przedziałowi $Q_2[1:4]$. Używając binarnego wyszukiwania możemy szybko znaleźć indeksy $l = 1$ oraz $m = 5$ odpowiadające x_0 oraz następnikowi x_3 w tablicy Q_2. Z indeksów m,l można obliczyć częstotliwość elementu 2 na przedziale $A[8:14]$: $F_2^A(8,14) = l - m = 5 - 1 = 4$.

Algorithm 9 Operacja QUERY

```
1:  function QUERY(i,j)
2:      mfreq ← 0
3:      mmode ← 0
4:      if j − i ≤ t then
5:          for k ← i, i + 1, . . . j do
6:              freq ← count(A[k], i, j)
7:              if freq > mfreq then
8:                  mfreq ← freq
9:                  mmode ← A[k]
10:         return mfreq, mmode
11:     b_i ← ⌊(i − 1)/t⌋
12:     b_j ← ⌊(j − 1)/t⌋
13:     if b_i + 1 ≤ b_j − 1 then
14:         mfreq ← S'[b_i + 1, b_j − 1]
15:         mmode ← S[b_i + 1, b_j − 1]
16:     for k ∈ prefiks ∪ sufiks do
17:         freq ← count(A[k], i, j)
18:         if freq > mfreq then
19:             mfreq ← freq
20:             mmode ← A[k]
21:     return freq, mode
```

4.3 Operacja QUERY

Lemat 1 *([4]) Niech A, B, C będą multizbiorami. Jeżeli dominanta $A \cup B \cup C$ nie jest w A, ani w C to jest nią dominanta multizbioru B.*

Zapytanie o dominantę na przedziale $A[i : j]$ podzielimy na 2 przypadki w zależności od rozmiaru przedziału. W pierwszym przypadku gdy $j - i \leq t$, iterujemy się liniowo po elementach przedziału $A[i : j]$ (linijki 4-10) i liczymy ich częstotliwość używając operacji COUNT. Zwracamy element o maksymalnej częstotliwości.
W drugim przypadku, gdy $j - i > t$, elementy $A[i]$ oraz $A[j]$ znajdują się w różnych blokach. Oznaczmy przez $b_i = \lfloor (i-1)/t \rfloor, b_j = \lfloor (j-1)/t \rfloor$ odpowiednio numery bloków do jakich należą elementy $A[i]$ oraz $A[j]$. Dzielimy $A[i : j]$ na trzy części $A[i : (b_i + 1)t]$, $A[(b_i + 1)t + 1 : b_j t]$ oraz $A[b_j t + 1 : j]$. Nazwijmy pierwszą i ostatnią część prefiksem i sufiksem. Wtedy z lematu 1 wiemy, że dominantą $A[i : j]$ jest dominanta środkowej części lub któryś z elementów prefiksu lub sufiksu. Dominantę środkowej części przechowujemy w $S[b_i + 1, b_j - 1]$ (linijki 11-15). Prefiks i sufiks mają ograniczony rozmiar przez t, więc obsługujemy te dwa fragmenty tak jak w przypadku pierwszym (linijki 16-21).

4.4 Analiza złożoności czasowej

Konstrukcja Tablice Q_i można stworzyć podczas jednego skanowania tablicy A co kosztuje $\mathcal{O}(n)$ czasu. i-te wiersze tablic $S[i, *], S'[i, *]$ możemy stworzyć liniowo skanując tablicę A. Wierszy w tablicach S oraz S' jest s. Łącznie na konstrukcje tablic S, S' potrzebujemy $O(sn)$ czasu. W takim razie, konstrukcja struktury KMS(S) zajmuje $\mathcal{O}(n + sn) = \mathcal{O}(sn)$ czasu.

Operacja COUNT Jak wspomniano wcześniej, operacja COUNT trwa $\mathcal{O}(\log n)$ czasu, ponieważ używa dwóch binarnych wyszukań na danych o wielkości $\mathcal{O}(n)$.

Operacja QUERY W pierwszym przypadku, gdy $j - i \leq t$ operacja QUERY wykonuje $\mathcal{O}(t)$ operacji COUNT, każda trwa po $\mathcal{O}(\log n)$ czasu, zatem przypadek pierwszy zabiera co najwyżej $\mathcal{O}(t \log n)$ czasu. W drugim

przypadku wykonujemy operacje COUNT iterując się po prefiksie i sufiksie. Prefiks jak i sufiks mają ograniczony rozmiar przez $\mathcal{O}(t)$. Łącznie drugi przypadek trwa $\mathcal{O}(t \log n)$ czasu. Oba przypadki łącznie zajmują $\mathcal{O}(t \log n + t \log n) = \mathcal{O}(t \log n)$ czasu.

4.5 Analiza złożoności pamięciowej

Konstrukcja Podczas konstrukcji tworzymy Δ tablic Q_i, które łącznie zajmują $\mathcal{O}(n)$ pamięci. Ponadto tworzymy dwie dwuwymiarowe tablice S oraz S' każda o rozmiarze $\mathcal{O}(s^2)$. Łącznie struktura danych zajmuje $\mathcal{O}(n + s^2)$ pamięci.

Operacje COUNT i QUERY Obie wspierane operacje używają stałej pamięci.

4.6 Dobór parametru s

Podstawiając $s = n^{1-e}$ dla $e \in (0, 1/2]$ Otrzymujemy strukturę danych o czasie konstrukcji $\mathcal{O}(n^{2-e})$, która zajmuje $\mathcal{O}(n^{2-2e})$ pamięci. Pozwala ona na obliczenie dominanty w czasie $\mathcal{O}(n^e \log n)$. W szczególności dobierając $e = 1/2$ dostajemy liniową pamięć i $\mathcal{O}(\sqrt{n} \log n)$ czas zapytania.

5 Struktura danych CDLMW(s)

Struktura CDLMW(s) (algorytm 1 z pracy [1]) jest modyfikacją struktury KMS(s). Główną zmianą w porównaniu do struktury KMS(s) jest sposób w jaki obsługujemy prefiks i sufiks podczas operacji QUERY. Dzięki tej zmianie pozbywamy się logarytmu w czasie operacji QUERY. Analogicznie do struktury KMS(s) będziemy dzielić tablicę A na s bloków, każdy, prócz ostatniego, o rozmiarze $t = \lceil n/s \rceil$. i-ty blok B_i dla $i = 0, 1, \ldots, s-1$ reprezentuje przedział $A[it+1, \min(n, (i+1)t)]$. Struktura CDLMW(s) wspiera następujące operacje dla statycznej tablicy $A[1:n]$:

1. COUNT(h, i, j) – Oblicza $F^A_{A[i]}(i, j)$ w czasie $\mathcal{O}(F^A_{A[i]}(i, j) - h)$. Zakładamy, że $1 \leq h \leq F^A_{A[i]}(i, j)$.
2. QUERY(i, j) – Oblicza dominantę i jej częstotliwość w czasie $\mathcal{O}(n/s)$.

Parametr h operacji COUNT(h, i, j) jest podpowiedzią ile razy co najmniej wartość $A[i]$ występuje w przedziale $A[i:j]$.

5.1 Konstrukcja struktury CDLMW(s)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	1	2	3	1	3	4	3	3	2	3	2	1	2	2	4	1
R	0	0	0	1	1	0	2	3	1	4	2	2	3	4	1	3
	B_0				B_1				B_2				B_3			

S

i \ j	0	1	2	3
0	2	4	5	5
1	0	3	4	4
2	0	0	2	4
3	0	0	0	2

S'

i \ j	0	1	2	3
0	1	3	3	3
1	0	3	3	3
2	0	0	2	2
3	0	0	0	2

Q_1	Q_2	Q_3	Q_4
1	2	3	6
4	9	5	15
12	11	7	
16	13	8	
	14	10	

Rysunek 2: Przykładowa tablica $A[1:16]$ z liczbą unikalnych elementów $\Delta = 4$ i podziałem na $s = 4$ bloki, każdy o rozmiarze $t = 4$. Ponadto zaznaczamy zapytanie o dominantę $A[8:14]$ z rozbiciem na prefiks $A[8:8]$, środek $A[9:12]$ oraz sufiks $A[13:14]$, które są zaznaczone odpowiednio kolorami zielonym, żółtym i zielonym.

Konstruujemy tablice S, S' oraz Δ tablic Q_x, które są opisane przy konstrukcji struktury danych KMS. Ponadto konstruujemy jedną dodatkową tablicę $R[1:n]$, która na i-tej pozycji zawiera indeks wartości $A[i]$ w tablicy $Q_{A[i]}$. Równoważnie możemy zdefiniować R, aby spełniała taką własność: $Q_{A[i]}[R[i]] = i$.

5.2 Operacja COUNT

Algorithm 10 Operacja COUNT

```
1: function COUNT(h,i,j)
2:     l ← R[i]
3:     m ← R[i] + h
4:     while m < size(Q_{A[i]}) ∧ Q_{A[i]}[m] ≤ j do
5:         m ← m + 1
6:     return m − l
```

Załóżmy, że chcemy policzyć $F^A_{A[i]}(i, j)$, z dodatkową wiedzą, że element $A[i]$ występuje w przedziale $A[i:j]$ co najmniej h razy. Wiemy, że wszystkie indeksy wystąpień $A[i]$ na przedziale $A[i:j]$ tworzą spójny fragment w tablicy $Q_{A[i]}$. Tablica R umożliwia nam znalezienie początku tego fragmentu w czasie stałym. Moglibyśmy przeskanować liniowo ten fragment tablicy $Q_{A[i]}$ i znaleźć wszystkie wystąpienia elementu $A[i]$ na przedziale $A[i:j]$. Jednakże dzięki podpowiedzi h możemy to przyśpieszyć i ominąć pierwsze h indeksów zaczynając od indeksu $R[i]$ w tablicy $Q_{A[i]}$.

Algorithm 11 Operacja QUERY

```
1: function QUERY(i,j)
2:     b_i ← ⌊(i − 1)/t⌋
3:     b_j ← ⌊(j − 1)/t⌋
4:     mfreq ← S'[b_i + 1, b_j − 1]
5:     mmode ← S[b_i + 1, b_j − 1]
6:     for k ∈ prefiks ∪ sufiks do
7:         if R[k] = 0 ∨ Q_{A[i]}[R[k] − 1] < i then
8:             if k + mfreq < size(Q_{A[k]}) ∧ Q_{A[k]}[R[k] + mfreq] ≤ j then
9:                 mfreq ← COUNT(mfreq + 1, k, j)
10:                mmode ← A[k]
11:    return mfreq, mmode
```

5.3 Operacja QUERY

Załóżmy, że chcemy odpowiedzieć na zapytanie o dominantę na przedziale $A[i : j]$. Analogicznie do operacji QUERY struktury KMS oznaczamy $b_i = \lfloor (i-1)/t \rfloor$, $b_j = \lfloor (j-1)/t \rfloor$ odpowiednio numery bloków do których należą elementy $A[i]$, $A[j]$. Dzielimy przedział $A[i : j]$ na trzy części $A[i : \min(j, (b_i + 1)t)]$, $A[(b_i + 1)t + 1 : b_j t]$ oraz $A[\max(b_j t + 1, (b_i + 1)t + 1) : j]$. Nazywamy je odpowiednio prefiks, środek oraz sufiks. Minima i maksima przy definicji prefiksu oraz sufiksu gwarantują, że przedziały te są rozłączne w przypadku kiedy $b_i = b_j$. Gdy $b_i = b_j$ środek oraz sufiks są puste oraz gdy, $b_i = b_j - 1$ środek jest pusty. Jeżeli $b_i + 1 > b_j - 1$ przyjmujemy $S'[b_i + 1, b_j - 1] = 0$
Aby efektywnie używać operacji COUNT będziemy przechowywać element m o największej częstotliwości środka oraz przejrzanych już elementów prefiksu i sufiksu w pętli w linijce 6. W zmiennych $mfreq$ oraz $mmode$ przechowujemy odpowiednio częstotliwość i wartość tego elementu. Element prefiksu lub sufiksu $A[k]$ nazwiemy *kandydatem* jeżeli:

1. Występuje jako pierwszy w przedziale $A[i : j]$
2. Występuje co najmniej $mfreq + 1$ razy na przedziale $A[i : j]$

Jeżeli element m nie jest dominantą to musi istnieć element m', który występuje od niego częściej na przedziale $A[i : j]$. W szczególności pierwsze wystąpienie m' na przedziale $A[i : j]$ zostanie zaklasyfikowane jako kandydat. W takiej sytuacji, kiedy napotkamy pierwsze wystąpienie m' w pętli w linijce 6, spełni warunki kandydata i przejdzie warunki w linijkach 7 oraz 8, w których to odpowiednio sprawdzamy warunek (1) oraz (2) bycia kandydatem. Wówczas podmieniamy element m na nowo znaleziony, liczniejszy, element m', którego częstotliwość obliczamy za pomocą operacji COUNT.

5.4 Analiza złożoności obliczeniowej

Konstrukcja Oprócz wszystkich tablic struktury KMS tworzymy jedną dodatkową tablicę R, którą łatwo skonstruować przeglądając liniowo tablice Q_x o łącznym rozmiarze n. Zatem możemy skonstruować strukturę CDLMW(S) w czasie $\mathcal{O}(ns)$.

Operacja COUNT Liniowo skanujemy tablicę $Q_{A[i]}$ od pozycji $R[i]+h$, a wiemy, że $Q_{A[i]}[R[i]+F^A_{A[i]}(i,j)] > j$ zatem pętla w linijce 4 nie wykona się więcej niż $(R[i] + F^A_{A[i]}(i,j)) - (R[i] + h)) = F^A_{A[i]}(i,j) - h$ razy. Każda iteracja trwa $\mathcal{O}(1)$ czasu, więc operacja COUNT potrzebuje $\mathcal{O}(F^A_{A[i]}(i,j) - h)$ czasu.

Operacja QUERY Przeanalizujemy ile czasu potrzebujemy łącznie na wszystkie operacje COUNT w linijce 9. Zauważamy, że jeżeli pętla operacji $\text{COUNT}(h, i, j)$ wykona k kroków to zwróci ona $h + k$. Wywołujemy tą operacje z $h = mfreq + 1$, zatem za każą iteracją pętli w operacji COUNT zwiększymy mfreq o 1. Niech m będzie dominantą przedziału $A[i : j]$. Oznaczmy przez a, b, c odpowiednio liczbę wstąpień m w prefiksie, środku oraz sufiksie. Wartość zmiennej $mfreq$ jest ograniczona przez $a + b + c$, a jej wartość początkowa to $S'[b_i + 1, b_j - 1]$. Dzięki temu wiemy, że operacja COUNT wykona maksymalnie $a + b + c - S'[b_i + 1, b_j - 1]$ iteracji. Ponadto wiemy, że $b \leq S'[b_i + 1, b_j - 1]$ oraz możemy ograniczyć a,b przez rozmiar bloku t. Łącząc

wszystkie te informacje dostajemy ograniczene na wszystkie operacje COUNT w pojedynczym zapytaniu: $\mathcal{O}(a+b+c-S'[b_i+1, b_j-1]) \subset \mathcal{O}(a+c) \subset \mathcal{O}(t) = \mathcal{O}(n/s)$. Sprawdzanie czy elementy są kandatami możemy ograniczyć przez rozmiar bloku. Łącząc wszystko operacja QUERY zajmuje $\mathcal{O}(n/s)$ czasu.

5.5 Analiza złożoności pamięciowej

Konstrukcja Tworzymy jedną dodatkową tablicę R o rozmiarze n w porównaniu do struktury KMS, zatem potrzebujemy $\mathcal{O}(n+s^2)$ pamięci.

Operacje COUNT i QUERY Używamy stałej pamięci podczas tych operacji.

6 Struktura danych CDLMW BP(s)

Struktura ta działa przy założeniu standardowego modelu obliczeń RAM z słowem maszynowym $w \in \Omega(\log n)$ oraz wymaga, aby częstotliwość dominanty była nie większa niż s. Struktura CDLMW BIT-PACKING, w skrócie CDLMW BP (algorytm 2 z pracy [1]) jest modyfikacją struktury CDLMW, która zmienia reprezentację tablicy S, aby była bardziej zwięzła. Ponadto nie będziemy przechowywać tablicy S'. Dzięki tym modyfikacjom uzyskuje mniejsze zużycie pamięci względem rozmiaru bloku, co pozwoli nam na dobranie większej liczby bloków $s = \sqrt{nw}$. Analogicznie do struktury KMS(s) będziemy dzielić tablicę A na s bloków, każdy, prócz ostatniego, o rozmiarze $t = \lceil n/s \rceil$. i-ty blok B_i dla $i = 0, 1, \ldots, s-1$ reprezentuje przedział $A[it+1, \min(n, (i+1)t)]$. Struktura CDLMW BP wspiera następujące operacje:

1. BCOUNT(b_i, b_j) – Oblicza dominantę wraz z częstotliwością przedziału $A[b_i t + 1 : \min(n, (b_j + 1)t)]$ w czasie $\mathcal{O}(n/s)$, przy założeniu $b_i \leq b_j < s$.
2. COUNT(h, i, j) – Oblicza $F^A_{A[i]}(i, j)$ w czasie $\mathcal{O}(F^A_{A[i]}(i, j) - h)$. Zakładamy, że $1 \leq h \leq F^A_{A[i]}(i, j)$.
3. QUERY(i, j) – Oblicza dominantę i jej częstotliwość w czasie $\mathcal{O}(n/s)$.

Operacja BCOUNT(b_i, b_j) oblicza dokładnie to samo co było przechowywane w wartościach tablic $S[b_i, b_j]$ oraz $S'[b_i, b_j]$ w strukturze KMS(s).

6.1 Konstrukcja

Konstruujemy tablicę R która jest opisana przy strukturze danych CDLMW oraz konstruujemy Δ tablic Q_x które są opisane przy strukturze danych KMS. Tworzymy, dla każdego $i \in \{0, 1, \ldots, s-1\}$ ciąg binarny $T[i]$, który odpowiada wierszowi $S[i, *]$ z tablicy w algorytmie KMS. Dla każdego $j \in \{i, i+1, \ldots, s-1\}$ dopisujemy na koniec ciągu $T[i]$ najpierw $S[i, j] - S[i, j-1]$ zer, a potem jedną jedynkę. Przyjmujemy $S[i, i-1] = 0$. Dla przykładu wiersz $2, 3, 5, 5$ zamieniamy na ciąg binarny 001010011.

6.2 Operacje RANK i SELECT

Dla ciągu binarnego $B = (b_0, b_1, ..., b_{k-1})$, oraz liczby $b \in \{0, 1\}$ definiujemy operację RANK$_b(B, i) = |\{j \mid b_j = b \wedge j \in \{0, 1, \ldots, i-1\}\}|$, innymi słowy RANK$_b(B, i)$ to liczba elementów ciągu B równa b na przedziale $[0, i)$. Definiujemy również operację SELECT$_b(B, i) = \min\{k \in \{1, 2, \ldots n\} \mid rank_b(k) = i\}$ – jest to indeks i-tego wystąpienia liczby b w ciągu B, gdzie $i > 0$. Operacje te pozwolą nam efektywnie wyciągać informacje z ciągów binarnych zawartych w T. Dla przykładu, znając pozycję p_j j-tej jedynki w binarnym ciągu $T[i]$, możemy obliczyć częstotliwość dominanty $B_i \cup B_{i+1} \cdots \cup B_{i+j-1}$ poprzez zapytanie RANK$_0(T[i], p_j)$. Oczywiście p_j możemy obliczyć za pomocą zapytania SELECT$_1(T[i], j)$. Dla działania algorytmu kluczowym jest, aby operacje te zużywały maksymalnie liniową liczbę bitów od długości ciągu na którym operują. RANK oraz SELECT są podstawą zwięzłych struktur danych (ang. succinct data structures), w związku z czym są dobrze zbadane ([3], [2], [6]). Dla obu operacji istnieją struktury zużywające podliniową pamięć i pozwalające na zapytania w stałym czasie, przy liniowym czasie konstrukcji. My jednak zdecydowaliśmy się zaimplementować bardziej praktyczne ([7], [5]) wersje tych operacji, które zużywają większą, liniową pamięć. Opiszemy implementację tych operacji w późniejszej sekcji.

6.3 Operacja BCOUNT

Załóżmy, że chcemy obliczyć dominantę $B_{b_i, b_j} = B_{b_i} \cup B_{b_i+1} \cup \cdots \cup B_{b_j}$ oraz jej częstotliwość. Oznaczmy przez $l = b_i t + 1$, $r = (b_j + 1)t$ odpowiednio lewy oraz prawy koniec B_{b_i, b_j}. Z konstrukcji $T[b_i]$ łatwo zauważyć, że częstotliwość dominanty B_{b_i, b_j} to liczba zer między początkiem ciągu $T[b_i]$, a jego $(b_j - b_i + 1)$-tą jedynką. Robimy to w linijkach 2-3. Następnie zauważmy, że ostatnie zero występujące przed $(b_j - b_i + 1)$-tą jedynką zostało wygenerowane przez pewną dominantę B_{b_i, b_j}. Możemy znaleźć pozycję tego zera jednym zapytaniem SELECT$_0$, robimy to w 4-tej linijce. Znając pozycję ostatniego zera wygenerowanego przez dominantę B_{b_i, b_j} łatwo znaleźć, w którym bloku ta dominanta się znajduje za pomocą operacji SELECT$_1$. Znajdujemy dominantę B_{b_i, b_j} skanując ostatni blok w którym się ona zawiera, dla każdego elementu bloku sprawdzamy czy występuje on co najmniej $F^A(l, r)$ razy w B_{b_i, b_j}.

Algorithm 12 Operacja BCOUNT

```
1: function BCOUNT(b_i,b_j)
2:     pos_{b_j} ← select_1(T[b_i], b_j − b_i + 1)
3:     freq ← rank_0(T[b_i], pos_{b_j})
4:     pos_last ← select_0(T[b_i], freq)
5:     b_last ← rank_1(T[b_i], pos_last) + b_i
6:     first ← tb_last + 1
7:     last ← min(n, first + t)
8:     for k ← first, ..., last do
9:         if R[k] − freq + 1 ≥ 0 ∧ Q_{A[k]}[R[k] − freq + 1] ≥ first then
10:            return freq, A[k]
```

6.4 Operacje COUNT oraz QUERY

Operacja COUNT pozostaje niezmieniona względem struktury CDLMW(S), a jedyną zmianą w operacji QUERY jest sposób obliczania dominanty B_{b_i+1,b_j-1}. Używamy do tego operacji BCOUNT zamiast dostępu do tablic S, S'.

6.5 Analiza złożoności czasowej

Konstrukcja Jedyna zmiana analizy względem struktury CDLMW(S) jest konstrukcja binarnych ciągów $T[i]$. Ciągi te konstruujemy skanując tablicę A s razy. Inicjalizacja operacji RANK oraz SELECT jest liniowa od długości ciągu binarnego. Ponieważ założyliśmy, że częstotliwość dominanty jest ograniczona przez s każdy ciąg binarny jest maksymalnie rozmiaru $2s$. Konstrukcja T wraz z RANK i SELECT zajmuje $\mathcal{O}(ns)$ czasu. Łącznie na konstrukcje potrzebujemy $\mathcal{O}(ns)$ czasu.

Operajca BCOUNT Używamy po 2 operacje RANK i SELECT, które zajmują stały czas. Ponadto liniowo skanujemy jeden blok o rozmiarze t. Łącznie potrzebujemy $O(t)$ czasu.

Operacje COUNT Nic nie zostało zmienione względem poprzedniej struktury danych. $\mathcal{O}(F^A_{A[i]}(i,j) - h)$ czasu.

Operacja QUERY Względem poprzedniej struktury danych obliczenie zmiennych $mfreq$ oraz $mmode$ wzrosło do $\mathcal{O}(t)$ ponieważ używamy operacji BCOUNT. Jednakże nie zmienia to łącznej złożoności czasowej $\mathcal{O}(t)$.

6.6 Analiza złożoności pamięciowej

Konstrukcja Ponieważ założyliśmy model RAM możemy ciągi binarne T spakować w słowa maszynowe dzięki czemu mamy s ciągów, każdy zajmujący $\mathcal{O}(s/w)$ słów maszynowych. Zatem łącznie na ciągi T potrzebujemy $\mathcal{O}(s^2/w)$ słów maszynowych, wliczając w to wsparcie dla operacji RANK oraz SELECT. Doliczając do tego tablicę R oraz Δ tablic Q_x łącznie struktura danych zużywa $\mathcal{O}(n + s^2/w)$ pamięci.

Operajca BCOUNT Używamy stałej pamięci podczas tej operacji.

Operacje COUNT i QUERY Koszt pamięci tych operacji pozostaje w dalszym ciągu stały.

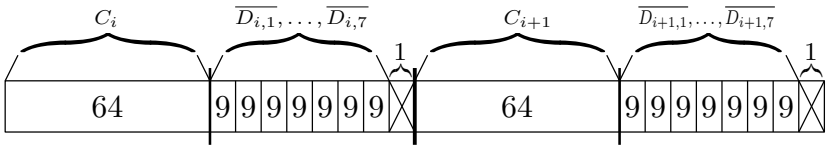
6.7 Dobór parametru s

Dzięki zmniejszonemu zużyciu pamięci możemy dobrać większą liczbę bloków $s = \sqrt{nw}$ co da nam strukturę o liniowej pamięci, która pozwala obliczać dominanty na dowolnym przedziale w czasie $\mathcal{O}(\sqrt{n/w})$.

6.8 Implementacja

W sekcji tej będziemy operować na binarnym ciągu $B = (b_0, b_1, \ldots, b_{k-1})$. Dla którego będziemy chcieli szybko odpowiadać na zapytania RANK oraz SELECT.

6.8.1 RANK



Rysunek 3: Fragment układu pamięci danych $\overline{C_i}, \overline{C_{i+1}}, \overline{D_{i,*}}, \overline{D_{i+1,*}}$. Liczby reprezentują rozmiar danych w bitach.

Zaimplementowaliśmy strukturę RANK9 z pracy [7] autorstwa Sebastiano Vigna, która pozwala odpowiadać na zapytania RANK_1. Struktura ta jest specjalnie zoptymalizowana pod procesory o 64-bitowym słowie maszynowym.

Konstrukcja Dla uproszczenia załóżmy, że k jest wielokrotnością 512. Dzielimy ciąg B na $k/512$ bloków $C_0, C_2, \ldots, C_{k/512-1}$, gdzie i-ty blok reprezentuje bity $B[512i : 512(i+1)-1]$. Ponadto, każdy blok C_i dzielimy na 8 podbloków $D_{i,0}, \ldots, D_{i,7}$, gdzie j-ty podblok reprezentuje bity $B[512i + 64j : 512i + 64(j + 1) - 1]$. Dla każdego bloku C_i przechowujemy w liczbie 64-bitowej $\overline{C_i} = \text{RANK}_1(B, 512i)$. Ponadto dla każdego podbloku $D_{i,j}$ zapisujemy używając 9-bitów $\overline{D_{i,j}} = \text{RANK}_1(C_i, 64j)$ – możemy to zrobić ponieważ, każdy podblok przechowuje RANK_1 ciągu o rozmiarze co najwyżej 512 bitów zatem wynik zmieści się w 9 bitach.

Zapytanie Oznaczmy przez $C_i, D_{i,j}$ blok i podblok do którego należy $B[l-1]$. Obliczamy zapytanie $\text{RANK}_1(B, l)$ następująco: $\overline{C_i} + \overline{D_{i,j}} + \text{RANK}_1(B[512i + 64j : i], l - 512i - 64j)$. Ostatni RANK_1 jest nad ciągiem o maksymalnej długości 64, który możemy obsłużyć za pomocą instrukcji `_mm_countbits_64`[2] w czasie stałym. W praktyce jednak, używamy funkcji wbudowanej `__builtin_popcountll`[3], która jest wspierana przez większość dużych kompilatorów języka c++ (GCC, Clang, MSVC). Generuje ona odpowiedni kod maszynowy w zależności od docelowej platformy.

Optymalizacje $\overline{D_{i,0}}$ zawsze wynosi 0, zatem nie musimy przechowywać wartości tej w pamięci. Dla każdego bloku C_i przechowujemy $\overline{C_i}$ oraz $\overline{D_{i,j}}$, gdzie $j = 1 \ldots 7$. $\overline{C_i}$ zajmuje 64 bity, a $\overline{D_{i,j}}$ łącznie zużywają $7 * 9 = 63$ bity, więc możemy wszystkie $\overline{D_{i,j}}$ spakować do jednego słowa maszynowego. Ponadto zauważamy, że zaraz po odczycie $\overline{C_i}$ robimy odczyt $\overline{D_{i,j}}$ zatem pakujemy te 2 słowa maszynowe obok siebie w pamięci, w celu zminimalizowania liczby nietrafień w pamięć podręczną procesora. Można zobaczyć jak wygląda układ pamięci tych danych na rysunku 3. Dla każdych 512-bitów używamy 2 64-bitowe słowa maszynowe. Podsumowując narzut pamięciowy operacji RANK_1 wynosi $128 \lceil k/512 \rceil \approx 0.25k$ bitów.

RANK_0 $\text{RANK}_0(B, i)$ możemy obsługiwać poprzez prostą obserwacje: $\text{RANK}_0(B, i) = i - \text{RANK}_1(B, i)$.

[2] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=_mm_countbits_64`
[3] `https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html`

6.8.2 SELECT

S_B	1				0				1				1				0				1			
D_B			1	0						1					1	0					1			
B	0	0	1	1	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	0
	C_0				C_1				C_2				C_3				C_4				C_5			

Rysunek 4: Przykładowy ciąg B oraz jego ciąg streszczony S_B oraz ciąg przerywający D_B z zaznaczonymi blokami $C_0, \ldots, C_5$. Długość słowa maszynowego na tym przykładzie wynosi $w = 4$.

Zaimplementowaliśmy lekko zmodyfikowany algorytm 2 z pracy [5] autorstwa Chae i innych. Nasza modyfikacja polega na wykorzystaniu dwóch instrukcji `_pdep_u64`[4], `_mm_tzcnt_64`[5] dostępnych na procesorach Intel oraz AMD. Instrukcje te pozwolą nam na wykonanie operacji SELECT_1 wewnątrz słowa maszynowego w stałym czasie i stałej pamięci. Oznaczamy przez m liczbę jedynek w ciągu B. Dla uproszczenia załóżmy, że k jest wielokrotnością w. Dzielimy ciąg B na k/w bloków $C_0, C_1, \ldots, C_{k/w-1}$, gdzie i-ty blok reprezentuje bity $B[wi : w(i+1) - 1]$.

Dla ciągu binarnego B definiujemy 2 pochodne ciągi binarne:

1. *Streszczony ciąg* $S_B[0 : k/w - 1]$ ciągu B ma na i-tej pozycji 1 gdy, dowolny bit bloku C_i jest nie zerowy. W przeciwnym wypadku $S_B[i] = 0$.
2. *Przerywający ciąg* $D_B[0 : m - 1]$ ciągu B ma na i-tej pozycji 1 gdy $(i+1)$-wsza jedynka ciągu B jest pierwszą jedynką w bloku ją zawierającym. W przeciwnym przypadku $D_B[i] = 0$.

Redukcja do SELECT_1 w pojedynczym bloku Naszym celem będzie zredukowanie $\text{SELECT}_1(B, i)$ do operacji SELECT_1 wewnątrz pojedynczego bloku. Aby to osiągnąć najpierw pokażemy jak obliczyć $\text{SELECT}_1(S_B, i)$. Ciąg S_B jest krótki, jego rozmiar wynosi n/w. Pozwala to nam na zapamiętanie pozycji wszystkich jedynek. Potrzebujemy $\log(n/w)$ bitów, aby zapisać jeden indeks. Indeksów do zapisania jest co najwyżej n/w. Korzystając z założenia, że $w \in \Omega(\log n)$ możemy oszacować potrzebną pamięć przez $\mathcal{O}(n/\log(n)\log(n/\log(n))) \subset \mathcal{O}(n/\log(n)\log(n)) = \mathcal{O}(n)$. Dzięki operacji SELECT_1 dla ciągu S_B jesteśmy wstanie znaleźć indeks k-tego niepustego bloku w czasie stałym. Ponadto możemy znaleźć numer niepustego bloku w którym znajduje się i-ta jedynka za pomocą operacji $\text{RANK}_1(D_B, i)$. Przy użyciu tych dwóch operacji potrafimy znaleźć indeks s_i bloku w którym jest i-ta jedynka oraz $p = \text{RANK}_1(B, s_i w)$ liczbę jedynek poprzedzającą blok C_{s_i}. Łącząc to wszystko dostajemy wzór: $\text{SELECT}_1(B, i) = s_i w + \text{SELECT}_1(C_{s_i}, i - p)$. Zredukowaliśmy $\text{SELECT}_1(B, i)$ do SELECT_1 w bloku C_{s_i}. Dla przykładu znajdziemy 5-tą jedynkę w ciągu B na rysunku 4. Najpierw znajdujemy numer niepustego bloku w którym znajduje się 5-ta jedynka $\text{RANK}_1(D_B, 5) = 3$. Następnie obliczamy indeks bloku s_i w którym znajduje się 5-ta jedynka $s_i = \text{SELECT}_1(S_B, 3) = 3$. Następnie liczymy p liczbę jedynek poprzedzającą blok s_i: $p = \text{RANK}_1(B, s_i w) = \text{RANK}_1(B, 12) = 3$. Korzystamy ze wzoru $\text{SELECT}_1(B, 5) = s_i w + \text{SELECT}_1(C_3, 2) = 12 + 3 = 15$. Jak obliczyć SELECT_1 w bloku C_3 pokazujemy w następnym paragrafie.

SELECT_1 w pojedynczym bloku W tym przypadku używamy dwóch instrukcji procesora `_pdep_u64` oraz `_mm_tzcnt_64`. Niech $a[0 : 63], msk[0 : 63]$ to 64-bitowe liczby. Oznaczmy przez l liczbę niezerowych bitów msk oraz $x_0, \ldots, x_{l-1}$ indeksy tych bitów. Wówczas `_pdep_u64`(a, msk) zwróci 64-bitową liczbę $K[0 : 63]$ zdefiniowaną jako $K[x_i] = a[i]$ oraz $K[j] = 0$ dla $j \neq x_i$. W szczególności `_pdep_u64`$(2^{c-1}, x)$ zwróci liczbę, która ma jeden nie zerowy bit na pozycji c-tej jedynki liczby x. Instrukcja `_mm_tzcnt_64` umożliwia nam znalezienie tej pozycji poprzez zliczenie końcowych zer. Dla przykładu znajdziemy 5-tą jedynkę liczby $a = 010011001101_2$. `_pdep_u64`$(2^{5-1} = 00001_2, 010011001101_2) = 000000000100_2$. Teraz liczmy końcowe zera – `_mm_tzcnt_64`$(000000000100_2) = 9$. 5-ta jedynka liczby a znajduje się na 9-tej pozycji.

Narzut pamięciowy Potrzebujemy wykonywać operacje RANK dla ciągów B oraz D_B, zakładając użycie struktury RANK9 potrzebujemy $\approx 2 * 0.25k = 0.5k$ bitów. Sam ciąg D_B zajmuje pesymistycznie k bitów. Ponadto, użyliśmy SELECT_1 na ciągu S_B, dla którego zapisaliśmy wszystkie pozycje jedynek. Zakładając $w \geq \log k$ zużywamy co najwyżej k bitów. Łącznie potrzebujemy co najwyżej $2.5k$ bitów.

[4] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=_pdep_u64`
[5] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=_mm_tzcnt_64`

7 Struktura danych CDLMW SF

Przedstawimy strukturę danych (algorytm 3 z pracy [1]), którego czas działania jest wrażliwy na liczbę unikalnych elementów Δ. Wspiera ona jedną operacje – QUERY(i, j) – zwracającą dominantę $A[i : j]$, w czasie $\mathcal{O}(\Delta)$.

7.1 Konstrukcja

				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			A	1	2	3	1	3	4	3	3	2	3	2	1	2	2	4	1
0	0	0	0	2	1	1	0	2	1	4	1	3	3	5	1	4	5	5	2
C_0				$C_1 = C_{b_{i-1}}$				C_2				$C_3 = C_{b_j}$				C_4			

Rysunek 5: Przykładowa tablica $A[1 : 16]$ z liczbą unikalnych elementów $\Delta = 4$ i podziałem na $s = 4$ bloki, każdy o rozmiarze $t = 4$. Ponadto zaznaczamy zapytanie o dominantę $A[2 : 10]$, prefiks $A[2 : 4]$ oraz sufiks $A[11 : 12]$, które są zaznaczone kolorem żółtym, zielonym oraz zielonym. Dodatkowo $C_1 = C_{b_{i-1}}$ oraz $C_3 = C_{b_j}$

Dzielimy tablicę A na bloki o rozmiarze Δ, przy czym dopuszczamy by ostatni blok miał mniejszy rozmiar. Dla każdego $i \in \{0, ..., \lceil \frac{n}{\Delta} \rceil\}$ tworzymy tablicę $C_i[1 : \Delta]$, która na j-tej pozycji zawiera liczbę wystąpień wartości j w pierwszych i blokach. Innymi słowy, $C_i[j]$ to liczba wystąpień wartości j w przedziale $A[1 : \min\{n, i\Delta\}]$. Rysunek 5 przedstawia przykładową tablicę i podział na bloki o rozmiarze $\Delta = 4$.

7.2 Operacja QUERY

Algorithm 13 Operacja QUERY

```
1:  function SFQuery(i, j)
2:      Niech C[1 : Δ] będzie tymczasową tablicą.
3:      b_{i-1} ← ⌈(i − 1)/Δ⌉
4:      b_j ← ⌈j/Δ⌉
5:      for k ← 1, . . . , Δ do
6:          C[k] ← C_{b_j}[k] − C_{b_{i-1}}[k]
7:      prefiks ← min(n, b_{i-1}Δ)
8:      for k ← i, . . . ,prefiks do
9:          C[A[k]] ← C[A[k]] + 1
10:     sufiks ← min(n, b_jΔ)
11:     for k ← j + 1, . . . ,sufiks do
12:         C[A[k]] ← C[A[k]] − 1
13:     mode ← argmax C
14:     return C[mode], mode
```

Rozważmy zapytanie o dominantę na przedziale $A[i : j]$. Niech $b_j = \lceil j/\Delta \rceil$ oraz $b_{i-1} = \lceil (i-1)/\Delta \rceil$ to indeksy bloków, do których należą odpowiednio wartości $A[j]$ oraz $A[i-1]$. Zauważmy że częstotliwość wszystkich elementów na przedziale $A[b_{i-1}\Delta + 1 : b_j\Delta]$ możemy obliczyć przez odjęcie $C_{b_{i-1}}$ od C_{b_j}. Oznaczmy różnicę tych dwóch tablic przez C. Aby uzyskać częstotliwość wszystkich elementów na przedziale $A[i : j]$ trzeba uwzględnić brakujące wartości $A[b_{i-1}\Delta : i]$ oraz nadmiarowe wartości $A[j + 1 : b_j\Delta]$ ($A[j+1 : n]$, gdy b_j jest ostatnim blokiem). Nazwijmy brakujące elementy prefiksem, a nadmiarowe elementy sufiksem. Elementów prefiksu i sufiksu nie jest zbyt wiele, dlatego przeglądamy je po kolei aktualizując ich częstotliwość w tablicy C.

7.3 Analiza złożoności pamięciowej

Przechowujemy $1 + \lceil n/\Delta \rceil$ tablic C_i, każda o rozmiarze Δ. W trakcie obliczania dominanty posługujemy się roboczą tablicą o rozmiarze Δ. Łącznie zużywamy $(1 + \lceil n/\Delta \rceil)\,\Delta + \Delta \in \mathcal{O}(n)$ pamięci.

7.4 Analiza złożoności czasowej

Obliczenie różnicy $C_{b_j} - C_{b_{i-1}}$ zajmuje czas $\mathcal{O}(\Delta)$. Rozmiar prefiksu i sufiksu możemy ograniczyć od góry przez $\Delta - 1$, zatem dodanie elementów prefiksu i odjęcie elementów sufiksu łącznie zajmie $\mathcal{O}(\Delta)$ czasu. W takim razie operacje zapytania, łącznie zajmują czas $\mathcal{O}(\Delta)$.

7.5 Implementacja

Dzięki temu że tablice $C_{b_{i-1}}$ oraz C_{b_j} zajmują ciągły fragment pamięci, możemy łatwo zwektoryzować obliczanie ich różnicy. W naszej implementacji używamy instrukcji wektorowej `_mm512_sub_epi64`[6], która pozwala na odjęcie 4 wartości 64bitowych na raz. Instrukcja ta jest dostępna na większości nowych procesorów Intel oraz AMD.

W zależności od rozmiaru sufiksu możemy uzyskać lepszy praktyczny czas działania, obsługując sufiks inaczej. Zauważmy, że zamiast uwzględniać cały blok b_j i potem odejmować od niego sufiks, równoważnie możemy go zignorować i dodać elementy $A[(b_j - 1)\Delta + 1 : j]$. W przypadku gdy sufiks $A[j + 1 : b_j\Delta]$ jest mniejszy niż $\lceil \Delta/2 \rceil$ używamy oryginalnego przetwarzania, a w przeciwnym przypadku używamy nowego. Dzięki temu pesymistycznie będziemy musieli przeskanować $\lceil \Delta/2 \rceil$ zamiast $\Delta - 1$ elementów. Analogiczną optymalizacje możemy zastosować do skanowania prefiksu. Łącznie pesymistycznie będziemy musieli przeskanować $2\lceil \Delta/2 \rceil$ elementów zamiast $2\Delta - 2$.

[6] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=_mm512_sub_epi64`

8 Struktura danych CDLMW BP + SF

Struktura ta (końcowy algorytm z pracy [1]) to połączenie struktury CDLMW BP oraz CDLMW SF. Struktura CDLMW BP zakłada, że dominanta nie może być duża, a CDLMW SF działa szybko kiedy liczba unikalnych elementów jest ograniczona. Metody te idealnie się dopełniają pozwalając ominąć ograniczenia obu struktur. Łączona struktura wspiera jedną operacje – QUERY(i, j) – zwracającą dominantę $A[i : j]$, w czasie $\mathcal{O}(\sqrt{n/w})$.

8.1 Konstrukcja

Dzielimy tablicę $A[1 : n]$ na dwie, mniejsze, tablice $A_1[1 : n']$ oraz $A_2[1 : n - n']$. Do tablicy A_1 należą elementy tablicy A o częstotliwości co najwyżej $\sqrt{nw}$, a do tablicy A_2 pozostałe elementy. Relatywny porządek w tablicach A_1, A_2 jest taki sam jak w tablicy A. Tworzymy A_1, A_2 obliczając częstotliwości wszystkich elementów używając tymczasowej tablicy o rozmiarze Δ i dzieląc tablicę A za niej pomocą. Konstruujemy strukturę CDLMW BP($\sqrt{nw}$) dla tablicy A_1, oraz strukturę CDLMW SF dla tablicy A_2. Dodatkowo skonstruujemy 4 tablice $I_a[1 : n], J_a[1 : n]$, gdzie $a \in \{1, 2\}$. Pozwolą one na konwersję zapytania o dominantę na przedziale $A[i : j]$ na dwa podzapytania o dominanty tablic $A_a[I_a[i], J_a[j]]$. Formalnie $I_a[i]$ to indeks w tablicy A_a pierwszego elementu A leżącego na prawo od $A[i]$, który jest w tablicy A_a. Analogicznie definiujemy $J_a[j]$ jako indeks tablicy A_a pierwszego elementu tablicy A leżącego na lewo od $A[j]$, który jest w tablicy A_a.

8.2 Operacja QUERY

Dla zapytania o dominantę $A[i : j]$ pytamy podstruktury CDLMW BP, CDLMW SF odpowiednio o dominanty na przedziałach $A_1[I_1[i] : J_1[i]]$, $A_2[I_2[i] : J_2[i]]$ i zwracamy element o większej częstotliwości.

8.3 Analiza złożoności czasowej

Konstrukcja Tablice I_a, J_a można łatwo skonstruować przechodząc tablicę A jeden raz. Podstruktury potrzebują $\mathcal{O}(n\sqrt{nw})$ oraz $\mathcal{O}(n)$ czasu na konstrukcję. Łącznie potrzebujemy $\mathcal{O}(n\sqrt{nw})$ czasu na konstrukcję.

Operacja QUERY Zauważamy, że tablica A_2 ma maksymalnie $n/\sqrt{nw} = \sqrt{n/w}$ elementów. Zatem obie podstruktury potrzebują $\mathcal{O}(\sqrt{n/w})$ czasu na policzenie dominant, więc operacja QUERY działa w czasie $\mathcal{O}(\sqrt{n/w})$.

8.4 Analiza złożoności pamięciowej

Konstrukcja Obie struktury zużywają liniową pamięci, tak samo jak tablice I_*, J_*. Zatem potrzebujemy łącznie liniową pamięć.

Operacja QUERY W operacji tej wykonujemy operacje na strukturach, gdzie każda zużywa stałą dodatkową pamięć. Zatem zużywamy $\mathcal{O}(1)$ pamięci.

Oprogramowanie	
System operacyjny	Ubuntu 20.04 LTS
Wersja jądra	5.4
Sprzęt	
Procesor	Intel Xeon Gold 6154 CPU @ 3.00GHz
Pamięć podręczna L1/L2/L3	32KiB/1MiB/24MiB
Drożność pamięci podręcznej	8
Użyte rozszerzenia procesora	AVX512VL, MBI2

Tablica 1: Sprzęt oraz jego oprogramowanie na którym zostały przeprowadzone eksperymenty

9 Eksperymety

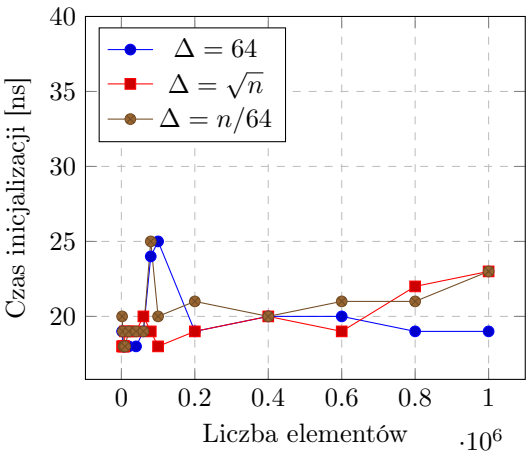
9.1 Testowe dane oraz zapytania

Testowe dane Przeprowadziliśmy testy wydajności czasu konstrukcji jak i czasu zapytań wszystkich zaimplementowanych algorytmów i struktur danych. W celu przeprowadzenia testów wygenerowaliśmy przykładowe tablice o rozmiarze w przedziale od 1000 do 4000000 elementów. Tablice te podzieliliśmy na trzy kategorie w zależności od liczby unikalnych elementów: $\Delta = 64, \Delta = \sqrt{n}$ oraz $\Delta = n/64$.

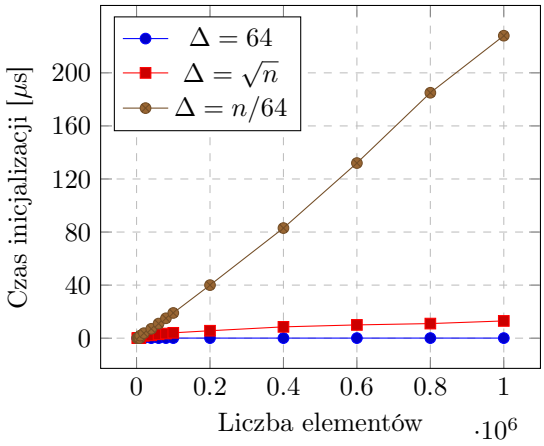
Testowe zapytania Dla każdej tablicy o rozmiarze n wygenerowaliśmy n zapytań. Ponadto, każde zapytanie reprezentowaliśmy jako wylosowanie jednostajnie dwie liczby $i, j \in \{1, 2, \ldots, n\}$, które odpowiadają zapytaniu o przedział $A[\min(i,j) : \max(i,j)]$. Przez takie dobranie zapytań do danych, każde zapytanie ma oczekiwaną długość $1/3n$. Długość przedziału ma szczególne znaczenie dla algorytmów naiwnych, których to czas zapytania jest bezpośrednio od niej uzależniony. Ponadto, dla algorytmów offline wygenerowaliśmy dodatkowe testy ze zmienną liczbą zapytań, aby zbadać jaki faktyczny ma to wpływ na czas zapytania.

9.2 Algorytm naiwny

9.2.1 Czas inicjalizacji



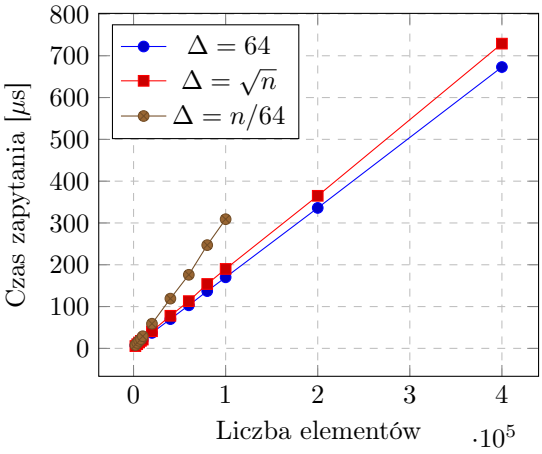
Rysunek 6: Czas inicjalizacji algorytmu naiwnego



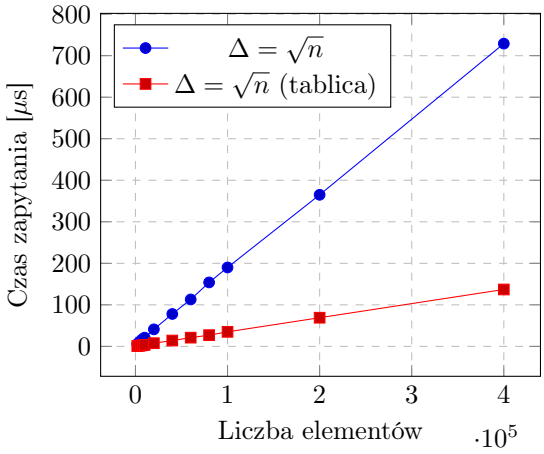
Rysunek 7: Czas inicjalizacji algorytmu naiwnego wariantu drugiego

Czas inicjalizacji wariantu pierwszego jest praktycznie zerowy, w naszej implementacji zapamiętujemy wskaźnik do tablicy na której będziemy wykonywać zapytania. Z drugiej strony podczas inicjalizacji wariantu drugiego tworzymy tablicę zastępującą hash-mapie w wariancie pierwszym. Moglibyśmy ją tworzyć za każdym razem podczas zapytania, aczkolwiek czas zapytania wzrósłby z $\mathcal{O}(j-i)$ do $\mathcal{O}(max(\Delta, j-i))$. Jak widać na rysunku 7, czas inicjalizacji rośnie wraz z Δ, aczkolwiek pomijalnie mało w porównaniu do czasu pojedynczego zapytania.

9.2.2 Czas zapytania



Rysunek 8: Czas zapytania algorytmu naiwnego



Rysunek 9: Porównanie dwóch wariantów

W naszych danych oczekiwana długość zapytania zależy liniowo od liczby elementów, dlatego przewidujemy, że czas zapytania algorytmu naiwnego również będzie liniowy od rozmiaru. Dokładnie takie zachowanie obserwujemy na rysunku 8. Ponadto możemy zauważyć, że algorytm naiwny dla liniowej liczby unikalnych elementów działa około półtora raza wolniej w porównaniu do innych kategorii danych. Podejrzewamy, że jest to spowodowane większą liczbą elementów w hash-mapie. Znalezienie odpowiedniej wartości może generować więcej nietrafień w pamięć podręczną procesora.

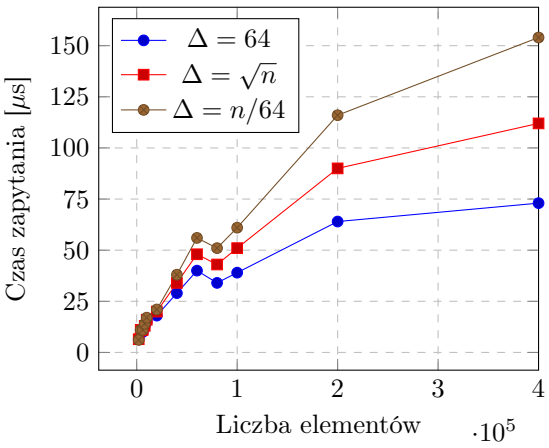
Tak jak podejrzewaliśmy, wariant drugi jest zdecydowanie szybszy (około 6 razy szybszy) w porównaniu do wariantu pierwszego. Operacje na hash-mapie, chociaż w teoretycznie czasie stałym, w praktyce są dużo wolniejsze od operacjach na tablicy.

9.3 Algorytm offline

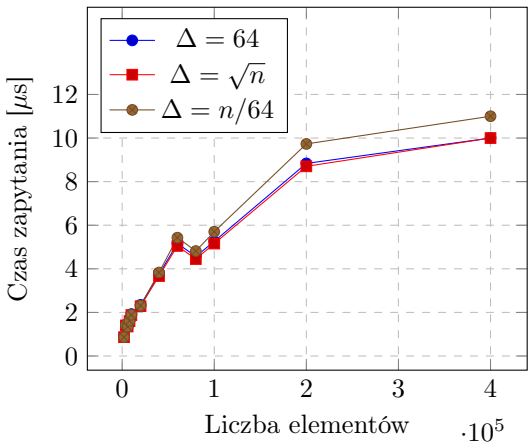
9.3.1 Czas inicjalizacji

Czas inicjalizacji algorytmu offline w obu wariantach jest analogiczny do drugiego wariantu algorytmu naiwnego. W pierwszej implementacji przy użyciu binarnych drzew tworzymy tablicę zastępującą hash-mapę do zliczania elementów, a w przypadku drugim dodatkowo tworzymy tablicę list. W obu implementacjach czas inicjalizacji jest pomijalnie mały w porównaniu do czasu zapytania.

9.3.2 Czas zapytania



Rysunek 10: Czas zapytania algorytmu offline BST

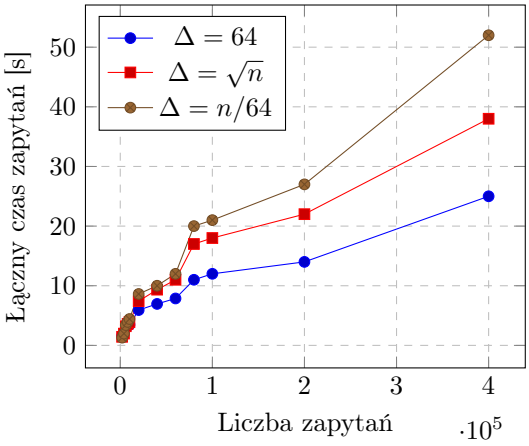


Rysunek 11: Czas zapytania algorytmu offline LIST

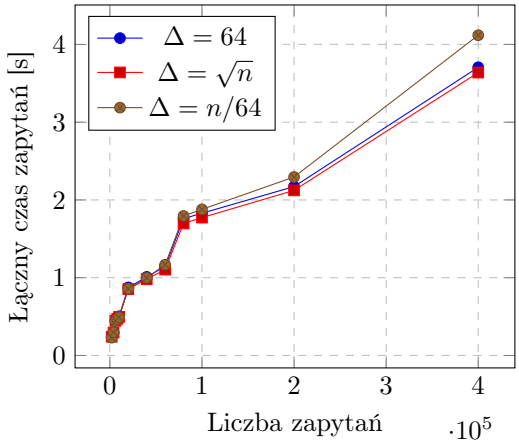
Dla danego rozmiaru tablicy wygenerowaliśmy liniową liczbę zapytań, dlatego przewidujemy czas działania

$\mathcal{O}(\sqrt{n}\log n)$ w wersji BST oraz $\mathcal{O}(\sqrt{n})$ w wersji LIST dla pojedynczego zapytania. Zauważamy, że implementacja za pomocą tablicy list jest zdecydowanie lepsza, działa, aż 7–15 razy szybciej.

9.3.3 Czas zapytania vs liczba zapytań



Rysunek 12: Łączny czas zapytań algorytmu offline BST

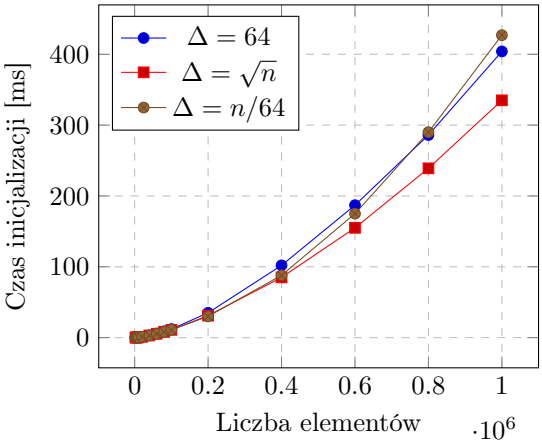


Rysunek 13: Łączny czas zapytań algorytmu offline LIST

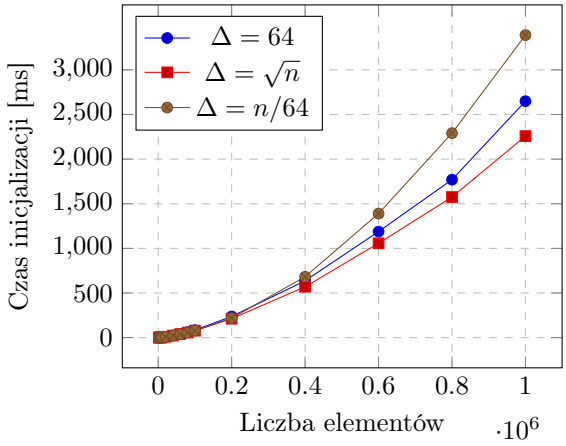
Algorytm offline jest jedynym opisanym przez nas algorytmem, którego czas działania zależy od liczby zapytań. Dlatego w trosce o kompletność przeprowadziliśmy testy sprawdzające jak zachowuje się algorytm offline w zależności od liczby zapytań. Zrobiliśmy testy dla $n = 400000$.

9.4 Struktury danych CDLMW oraz CDLMW BP

9.4.1 Czas inicjalizacji



Rysunek 14: Czas inicjalizacji struktury danych CDLMW



Rysunek 15: Czas inicjalizacji struktury danych CDLMW BP

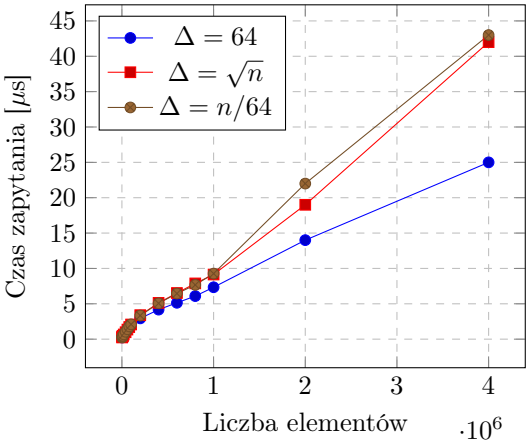
Przypominamy, że czas inicjalizacji struktur danych CDLMW, CDLMW BP wynosi odpowiednio $\mathcal{O}(n\sqrt{n})$, $\mathcal{O}(n\sqrt{nw})$. W naszej implementacji słowo maszynowe $w = 64$, więc bloki w wersji BP są $\sqrt{w} = 8$ razy mniejsze. Z tego powodu spodziewamy się co najmniej 8 razy wolniejszego czasu tworzenia struktury w wersji BP. W praktyce rozkłady $\Delta = 16$ oraz $\Delta = \sqrt{n}$ radzą sobie lepiej, struktura BP działa około 6.5 razy wolniej. W przypadku dużej ilości unikalnych elementów struktura BP jest 8 razy wolniejsza.

Podejrzewaliśmy, że dodatkowa inicjalizacja RANK oraz SELECT w wersji BP struktury danych może mieć znaczący wpływ na czas konstrukcji. Po użyciu profilera[7] dowiedzieliśmy się, że ponad 90% czasu podczas

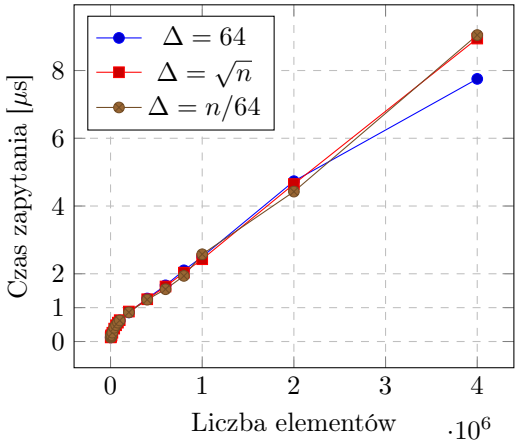
[7] `https://perf.wiki.kernel.org/index.php/Tutorial`

inicjalizacji zajmuje obliczanie dominanty dla każdej pary bloków. Aby to potwierdzić postanowiliśmy zwiększyć liczbę bloków w algorytmie CDLMW do $\sqrt{nw}$. Otrzymaliśmy bardzo zbliżone czasy inicjalizacji do wersji BP.

9.4.2 Czas zapytania



Rysunek 16: Czas zapytania algorytmu CDLMW

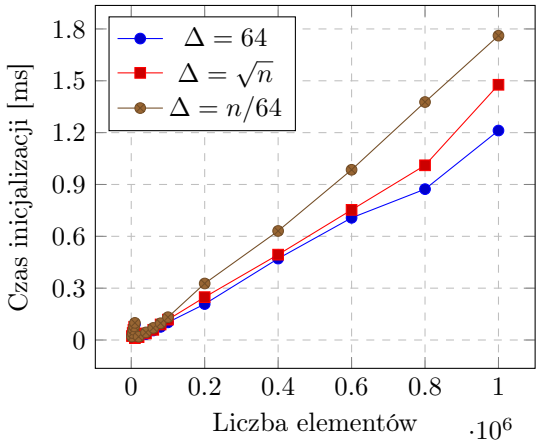


Rysunek 17: Czas zapytania algorytmu CDLMW BP

Podczas zapytania w wersji BP skanujemy o jeden blok więcej, zatem oczekujemy, że zapytanie będzie $8 * (2/3) = 5.33$ razy szybsze od wersji podstawowej algorytmu. W praktyce jednak tak nie jest, gdy liczba elementów jest mała doświadczamy około trzykrotnego przyśpieszenia. Wraz ze wzrostem liczby unikalnych elementów dostajemy około 4.75-krotne przyśpieszenie dla danych $\Delta = \sqrt{n}$ oraz $\Delta = n/64$.

9.5 Struktura danych CDLMW SF

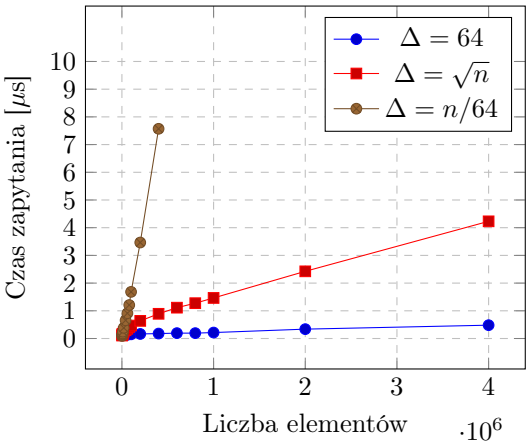
9.5.1 Czas inicjalizacji



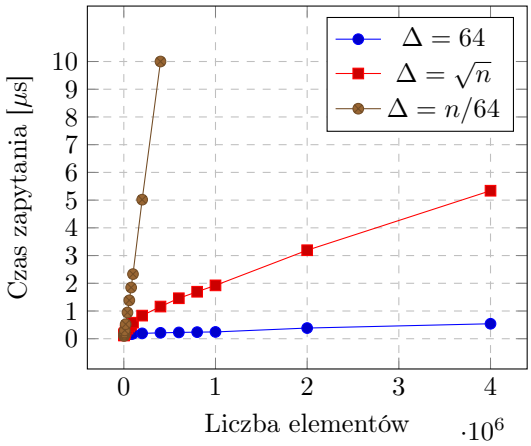
Rysunek 18: Czas inicjalizacji struktury danych CLDMW SF

Czas inicjalizacji struktury danych CDLMW SF zależy liniowo od rozmiaru tablicy. Na wykresie 18 możemy zaobserwować, że w praktyce im więcej unikalnych elementów tym większy czas inicjalizacji.

9.5.2 Czas zapytania



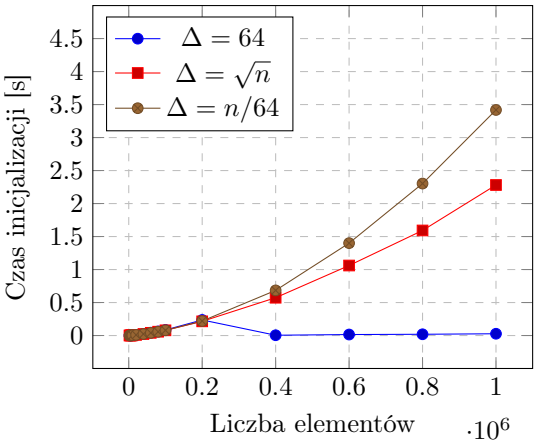
Rysunek 19: Czas zapytania algorytmu CLDMW SF z optymalizacjami



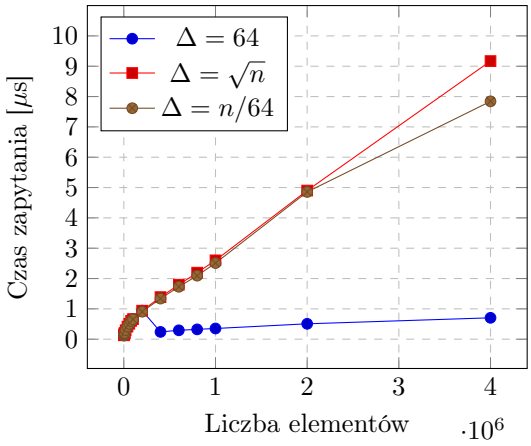
Rysunek 20: Czas zapytania algorytmu CDLMW SF bez optymalizacji

Testy czasu zapytań przeprowadziliśmy dla dwóch wersji struktury CDLMW SF: z optymalizacjami opisanych w sekcji 7.5, oraz bez optymalizacji. Wyniki testów można zobaczyć odpowiednio na rysunkach 19 oraz 20. Dla danych z stałą liczbą unikalnych elementów czas zapytania w obu wersjach jest praktycznie taki sam. Dla kategorii $\Delta = \sqrt{n}$ oraz $\Delta = n/64$ zoptymalizowana wersja jest o około 1.25 razy szybsza od podstawowej.

9.6 Struktura danych CDLMW BP + SF



Rysunek 21: Czas inicjalizacji struktury danych CLDMW BP + SF



Rysunek 22: Czas zapytania struktury danych CLDMW BP + SF

Czas inicjalizacji Przypominamy, że czas inicjalizacji struktury CDLMW BP + SF wynosi $\mathcal{O}(n\sqrt{nw})$. Dla kategorii danych $\Delta = 64$ oraz $\Delta = \sqrt{n}$ wykres 21 to odzwierciedla, aczkolwiek dla typu $\Delta = 64$ przy $n = 4 * 10^4$ elementów następuje nagły spadek czasu inicjalizacji. Wynika to z faktu, że dla kategorii $\Delta = 64$ oraz $n \geq 512^2 \approx 2.6 * 10^5$ oczekiwana częstotliwość każdego elementu jest większa niż $\sqrt{nw}$, przez co wszystkie elementy są obsługiwane przez podstrukturę CDLMW SF, która to ma znacznie szybszy, liniowy, czas inicjalizacji.

9.7 Podsumowanie

W sekcji tej postaramy się odpowiedzieć na pytanie jakiego algorytmu/struktury danych należy użyć, aby rozwiązać problem RMQ jak najszybciej.

Algorytm naiwny Polecamy użyć algorytmu naiwnego w przypadku, gdy suma długości przedziałów zapytań nie przekracza n. Wszystkie opisane struktury danych potrzebują co najmniej linowego czasu na inicjalizację, czas ten równie dobrze możemy zagospodarować na obliczanie dominanty.

Algorytm offline Na wstępie zauważamy, że nie ma sensu używać wariantu BST, jest on od 7 do 15 razy wolniejszy od wersji LIST. Gdy algorytm α rozwiązujący problem RMQ działa w czasie $f(n) \geq \log n$ to używamy algorytmu offline[8], gdy liczba zapytań $|Q| \geq C(n/f(x))^2$ dla odpowiednio dobranej stałej C[9]. Dla takiego doboru $|Q|$ algorytm offline działa nie wolniej niż algorytm α. W szczególności dla algorytmów CDLMW SF oraz CDLMW BP + SF dobieramy liczbę zapytań odpowiednio $128(n/\Delta)^2$ oraz $2(n/\sqrt{n/w})^2$ dla których algorytm offline LIST osiąga podobne czasy zapytań.

Struktura danych KMS Nie zalecamy używać tej struktury, struktura CDLMW posiada praktycznie identyczny czas inicjalizacji i jednocześnie szybszą operacje QUERY.

Struktura danych CDLMW SF Gdy unikalnych elementów nie jest więcej niż $45\sqrt{n/w}$ proponujemy użyć struktury CDLMW SF, posiada ona liniowy czas inicjalizacji oraz czas zapytania jest najszybszy ze wszystkich struktury danych przy tej dystrybucji danych.

Struktury danych CDLMW BP + SF Gdy unikalnych elementów jest więcej niż $45\sqrt{n/w}$ sugerujemy użycie struktury CDLMW BP + SF. Niestety czas inicjalizacji jest znacząco gorszy w porównaniu do struktury CDLMW SF, aczkolwiek czas zapytania jest co najmniej dwukrotnie szybszy.

[8] O ile wszystkie zapytania są dostępne a priori

[9] Wszystkie stałe w tym podrozdziale zostały dobrane przez dodatkowe eksperymenty

Bibliografia

[1] Timothy M. Chan i in. „Linear-Space Data Structures for Range Mode Query in Arrays". W: *Theory Comput. Syst.* 55.4 (2014), s. 719–741. DOI: 10.1007/s00224-013-9455-2. URL: https://doi.org/10.1007/s00224-013-9455-2.

[2] David Clark. „Compact pat trees". W: (1997).

[3] Guy Jacobson. „Space-efficient Static Trees and Graphs". W: *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*. IEEE Computer Society, 1989, s. 549–554. DOI: 10.1109/SFCS.1989.63533. URL: https://doi.org/10.1109/SFCS.1989.63533.

[4] Danny Krizanc, Pat Morin i Michiel H. M. Smid. „Range Mode and Range Median Queries on Lists and Trees". W: *Nord. J. Comput.* 12.1 (2005), s. 1–17.

[5] Joong Chae Na i in. „Fast Computation of Rank and Select Functions for Succinct Representation". W: *IEICE Trans. Inf. Syst.* 92-D.10 (2009), s. 2025–2033. DOI: 10.1587/transinf.E92.D.2025. URL: https://doi.org/10.1587/transinf.E92.D.2025.

[6] Rajeev Raman, Venkatesh Raman i Srinivasa Rao Satti. „Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets". W: *ACM Trans. Algorithms* 3.4 (2007), s. 43. DOI: 10.1145/1290672.1290680. URL: https://doi.org/10.1145/1290672.1290680.

[7] Sebastiano Vigna. „Broadword Implementation of Rank/Select Queries". W: *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*. Red. Catherine C. McGeoch. T. 5038. Lecture Notes in Computer Science. Springer, 2008, s. 154–168. DOI: 10.1007/978-3-540-68552-4_12. URL: https://doi.org/10.1007/978-3-540-68552-4%5C_12.