Математичка гимназија, Београд

Матурски рад
из Рачунарства и информатике

Сегментно и Фенвиково стабло

Ментор:
Снежана Јелић

Ученик:
Михајло Марковић 4е

У Београду, мај 2023.

Садржај

1. Увод

1.1. Упити над статичким низовима

Циљ овог рада је упознавање са структурама које омогућавају ефикасно израчунавање различитих упита над интервалима. Кроз наредни пример видећемо неке технике за израчунавање резултата упита над статичким низовима, тј. над низовима у којима не долази до промена у току решавања упита. Ове технике су интуитивније, једноставније су и за разумевање и за имплементацију од оних које се користе код сегментног и Фенвиковог стабла, али је и њихова примена веома ограничена, па ћемо, након уочавања њихових недостатака, схватити потребу за увођење комплекснијих структура.

Пример: (а) Нека је дат низ $a_1, a_2, \dots, a_n$ и Q упита у којима треба израчунати суму елемената из датог низа који се налазе на позицијама из интервала $[l, r]$.

(б) Нека је дата матрица $\begin{bmatrix} a_{1,1} & \dots & a_{1,m} \\ \dots & \dots & \dots \\ a_{n,1} & \dots & a_{n,m} \end{bmatrix}$ и Q упита у којима треба израчунати суму елемената који се налазе унутар правоугаоника чије горње лево теме има координате (x_1, y_1), а доње десно (x_2, y_2).

(в) Нека је дат низ $a_1, a_2, \dots, a_n$ и Q упита у којима треба израчунати минимум елемената из датог низа који се налазе на позицијама из интервала $[l, r]$.

Решење: (а) Најочигледније решење је пролазак кроз елементе низа на позицијама из интервала $[l, r]$. У зависности од величине низа и броја упита, ово решење може бити веома неефикасно (временска сложеност је $O(NQ)$). Задатак можемо решити коришћењем **префиксних сума**. Уведимо низ p, у ком ће се на позицији i чувати сума свих елемената низа a чија позиција није већа од i ($p_i = \sum_{j=1}^{j \le i} a_j$). Вредности низа p можемо израчунати у сложености $O(N)$ помоћу формуле $p_i = p_{i-1} + a_i$. Користећи низ p резултат сваког упита налазимо у сложености $O(1)$ помоћу формуле $res = p_r - p_{l-1}$, па је укупна сложеност овог решења $O(N + Q)$.

(б) Техника префиксних сума може се применити и на **вишедимензионалне низове**. Ако низ p дефинишемо преко формуле $p_{x,y} = \sum_{i=1, j=1}^{i \le x, j \le y} a_{i,j}$, онда вредности низа p можемо израчунати у сложености $O(NM)$ помоћу формуле $p_{x,y} = p_{x-1,y} + p_{x,y-1} - p_{x-1,y-1} + a_{x,y}$. Резултат сваког упита налазимо у сложености $O(1)$ помоћу формуле $res = p_{x2,y2} - p_{x2,y1-1} - p_{x1-1,y2} + p_{x1-1,y1-1}$, па је укупна сложеност $O(NM + Q)$.

(в) Идеју са префиксним сумама овде не можемо применити у њеном основном облику због следећег: ако је $\min(1, l) = \min(1, r)$ (при чему је $\min(x, y)$ вредност

најмањег елемента низа са позиција из интервала $[x, y]$), помоћу тако креиране структуре не можемо добити тачан резултат за упит $\min(l, r)$. Закључујемо да приликом израчунавања упита $\min(l, r)$ не смемо користити међурезултате за интервале чији се делови налазе ван интервала $[l, r]$. Дакле, потребно је креирати структуру која ће чувати довољно међурезултата тако да одговарање на упите $\min(l, r)$ буде ефикасно, а да при том и меморијска сложеност буде задовољавајућа. Прво, приметимо да је за сваку позицију потребно чувати по неки резултат интервала чија је лева граница управо та позиција, као и по неки резултат интервала чија је десна граница та позиција, како бисмо омогућили да се при израчунавању не изађе из задатог интервала $[l, r]$. Критеријум за избор интервала чије ћемо резултате чувати би требало да за сваку позицију буде приближно исти, како би и у најгорем случају решење било ефикасно.

Посматрајмо интервале дужине $d = 2^k$. За сваку позицију p чувамо резултате свих интервала облика $[p, p + 2^k)$. Означимо те резултате са $st[p][k]$. Како је код операције минимум дозвољено преклапање интервала, $\min(l, r)$ једноставно можемо наћи претрагом укупно 2 елемента низа st. Нека је $s[g] = 2^g$ највећи степен двојке који није већи од дужине задатог интервала, тј. $g = \log_2(r - l + 1)$. Тражени резултат је једнак минимуму следеће две вредности: $st[l][g]$ и $st[r - s[g] + 1][g]$. Меморијска сложеност решења је $O(N \log_2 N)$ и временска сложеност прекалкулација, тј. рачунања вредности елемената матрице st је $O(N \log_2 N)$ (само је потребно поћи од интервала дужине 1, ићи редом ка већим интервалима и матрицу попуњавати помоћу формуле $st[p][k] = \min(st[p][k - 1], st[p + s[k - 1]][k - 1])$. Временска сложеност израчунавања сваког упита је $O(1)$, па је укупна временска сложеност решења $O(N \log_2 N + Q)$.

Ова структура се у литератури назива ***sparse table*** и применљива је за упите над статичким низовима са операцијама код којих „преклапање интервала“ не доводи до промене резултата (минимум, максимум, AND, OR, највећи заједнички делилац, најмањи заједнички садржалац...). Структура подржава и решавање упита са другим операцијама попут сабирања и множења, али је временска сложеност обраде сваког упита $O(\log_2 N)$.

Можемо још и приметити да смо уместо критеријума да дужине интервала буду степени двојке, могли одабрати и неки други критеријум. Ако посматрамо степене неког броја $x > 2$, умањићемо меморијску сложеност неколико пута, што некад може бити значајно с обзиром на то да је меморијска сложеност за *sparse table* прилично велика, а за обраду сваког упита потребно је пронаћи тачно x вредности из матрице, па према томе можемо мењати x у зависности од тога шта нам је приоритет у конкретном проблему. С друге стране, можемо уместо степена неког броја одабрати и неки скроз другачији критеријум и добити структуре које су применљиве у различитим класама проблема. На пример, ако низ дужине N

поделимо на интервале дужина $\sqrt{N}$, омогућавамо израчунавање различитих упита над интервалима у сложености $O(\sqrt{N})$ помоћу Моовог алгоритма.

Као што видимо, постоје разни алгоритми и структуре за решавање упита над интервалима. Заједничко за претходно описане структуре јесте чињеница да не подржавају измене основних низова над којима су креиране, тј. бесмислено их је користити у таквим ситуацијама јер је практично за сваку измену неопходно поново проћи кроз велики део те структуре. У овом раду бавићемо се структурама које подржавају и упите измене садржаја низова и упите израчунавања резултата након тих измена. Операције које подржавају те структуре су углавном оне које су разматране и кроз уводни пример. Прво ћемо се бавити проблемима са једнодимензионалним низовима, а затим ћемо проширити класу проблема на дводимензионалне низове, након чега ће бити речи и о општем случају, n-димензионалним низовима.

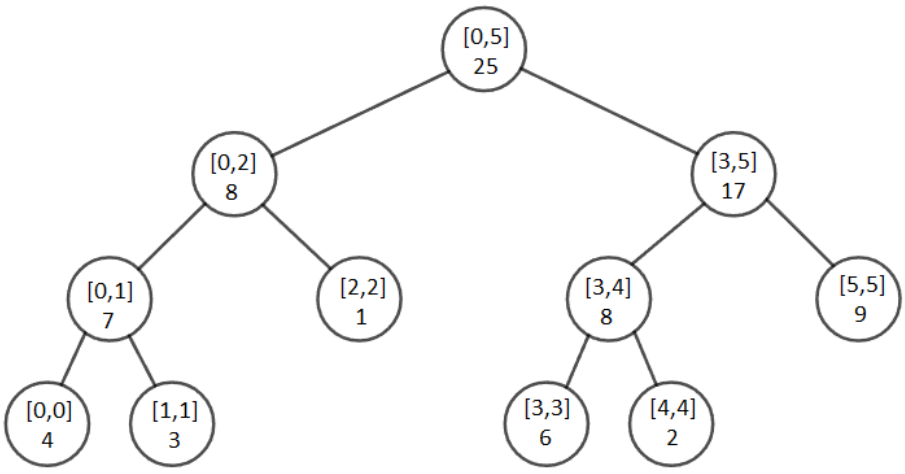
Како у савременом свету веома често постоји потреба за што ефикаснијим ажурирањем великог броја података, значај ових структура је велики јер су управо оне у основи многих система за управљање базама података и омогућавају брзо и прецизно налажење тражених резултата, као и њихову измену. Осим тога, проналазак ових структура и њихово даље истраживање су свакако послужили као инспирација и идеја за креирање и других напреднијих структура, па је због тога разумевање ових структура корисно за сваког ко се бави проучавањем алгоритама и структура података. Разумевање општег n-димензионалног случаја омогућава решавање и оптимизацију многих проблема у областима као што су географски информациони системи, обрада слика и видео-записа...

2. Сегментно стабло

2.1. Структура и имплементација

Сегментно стабло је структура која чува информације о појединим интервалима помоћу којих се ефикасније извршавају различите врсте упита над произвољним интервалима, укључујући и упите у којима се мења и сам садржај стабла. То је бинарно стабло у ком сваки чвор садржи резултат операције реализоване над своја два детета. Операције које подржава сегментно стабло морају бити бинарне, асоцијативне и затворене у скупу над којим се извршавају. Пожељно је да постоји и одговарајући неутрални елемент за операцију која се извршава на том скупу јер то у неким случајевима олакшава имплементацију неких функција. Неке од операција које подржава основна имплементација сегментног стабла су: сабирање, множење, минимум, максимум, највећи заједнички делилац, најмањи заједнички садржалац...

Листови сегментног стабла су вредности елемената полазног низа над којим се формира стабло. Родитељ два листа садржи резултат операције коју подржава стабло, извршене над тим листовима. Корен стабла чува резултат те операције извршене над свим елементима низа. Нека полазни низ има n елемената, а тражена операција је сабирање. Корен стабла садржи суму свих елемената низа чије су позиције из интервала $[0, n-1]$. Једно његово дете (у наставку ћемо га звати лево) садржи суму елемената полазног низа чије су позиције из интервала $[0, (n-1)/2]$, а друго дете (десно) суму елемената чије су позиције из интервала $[(n-1)/2+1, n-1]$. Све док чвор садржи резултат за више од једног елемента, проширићемо га тако што ћемо му додати двоје деце при чему ће његово лево дете чувати информацију о левој половини интервала за који је тај чвор задужен, а десно дете за десну половину тог интервала. На слици је приказана структура и садржај једног таквог стабла.



Слика 1: Интервали за које су задужени чворови сегментног стабла

Иако на први поглед можда не делује тако, за имплементацију овако постављеног сегментног стабла довољан је један низ, упркос томе што не можемо директно, само на основу индекса чвора, одредити колико деце тај чвор има (0 или 2) и за који интервал је задужен. Нека корен стабла има индекс 1, а деца чвора са индексом i имају индексе $2i$ и $2i + 1$ (конкретно, за овај начин имплементације, не прави никакву разлику да ли ће индекс корена бити 1, а индекси деце чвора i бити $2i$ и $2i + 1$, или индекс корена 0, а индекси деце $2i + 1$ и $2i + 2$, али је узет индекс 1 да би се индекси поклапали са другим начином имплементације, који ће бити објашњен касније). Сада је могуће креирати стабло помоћу следеће рекурзивне функције коју позивамо са почетним вредностима $create(1, 0,\ n - 1)$.

```
void create(int ind, int l, int r) {
	if (l == r) {
		segtree[ind] = a[l];
		return;
	}
	create(ind * 2, l, (l + r) / 2);
	create(ind * 2 + 1, (l + r) / 2 + 1, r);
	segtree[ind] = segtree[ind * 2] + segtree[ind * 2 + 1];
}
```

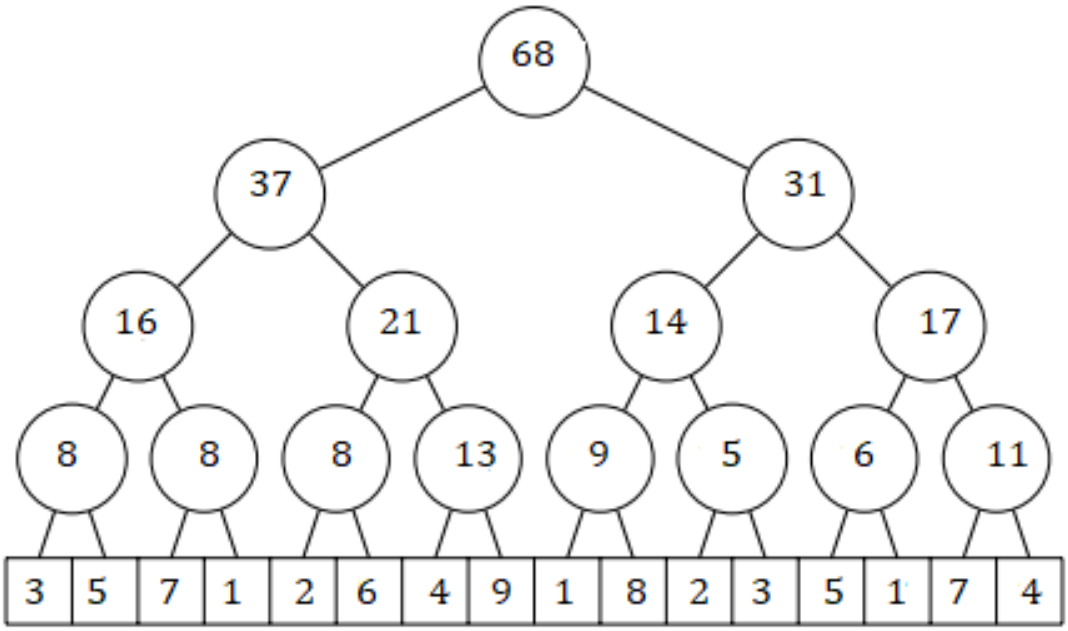
Временска сложеност креирања сегментног стабла на овај начин је $O(n)$, док ово стабло у себи садржи највише $2n - 1$ чворова. Међутим, максимални индекс неког чвора у овом стаблу може бити већи од $2n - 1$ при оваквом начину имплементације (док ће неки индекси бити неискоришћени).

Сличним рекурзивним функцијама можемо извршити израчунавање резултата на неком интервалу (у нашем случају израчунавања збира) и ажурирања појединачних елемената у низу (у нашем случају додавања вредности x том елементу). Не заборавимо да није довољно само пронаћи и изменити лист стабла који одговара елементу низа чију вредност мењамо, већ је потребно и изменити вредност свих његових предака. Управо таква измена садржаја стабла нам омогућава ефикасно одговарање на упите за израчунавање резултата на неком интервалу. У наставку је приказана имплементација поменуте две функције.

```
long long getsum(int ind, int l, int r, int ql, int qr) {
	if (ql <= l && qr >= r)
		return segtree[ind];
	return getsum(ind * 2, l, (l + r) / 2, ql, qr)
		+ getsum(ind * 2 + 1, (l + r) / 2 + 1, r, ql, qr);
}

void update(int ind, int l, int r, int pos, long long val) {
	if (pos<l || pos>r) return;
	segtree[ind] += val;
	if (l != r)
		update(ind * 2, l, (l + r) / 2, pos, val),
		update(ind * 2 + 1, (l + r) / 2 + 1, r, pos, val);
}
```

Овакво приказивање сегментних стабала, иако у потпуности може да одговори свим захтевима, није баш најпогоднije приликом саме имплементације. Једини начин да дођемо до неког конкретног листа или другог интервала који нас занима је да пођемо од корена и делимо интервале док не дођемо до траженог. Међутим, анализирајмо стабло које је креирано над низом који има $n = 2^p$ елемената. Једно такво стабло приказано је и на следећој слици.



Слика 2: Сегментно стабло формирано над низом који има $n = 2^p$ елемената

Приметимо следеће: ако је корен стабла индексиран од 1, свака генерација чворова ће бити индексирана тако да чвор који садржи резултат за интервал са најмањим границама има индекс који је једнак неком степену двојке (прецизније, степену двојке који одговара редном броју генерације ком припада тај чвор). Како у оваквом стаблу има тачно $n - 1$ чворова који нису листови, а не постоје индекси који се не користе и прескачу (јер сваки чвор који није лист има тачно двоје деце), знамо да ће елементу са индексом i из низа над којим се формира стабло, у стаблу одговарати лист који има индекс $n + i$. Ово у значајној мери олакшава и креирање стабла, које се више не мора обављати рекурзивно, већ итеративно (али обавезно од већих индекса стабла ка мањим).

```
vector<long long> create(vector<long long> a) {
	int n = a.size();
	int N = pow(2, (int)log2(n - 1) + 1);
	vector<long long> segtree(2 * N);
	for (int i = 0; i < n; i++)
		segtree[N + i] = a[i];
	for (int i = n; i < N; i++)
		segtree[N + i] = 0;
	for (int i = N - 1; i > 0; i--)
		segtree[i] = segtree[i * 2] + segtree[i * 2 + 1];
	return segtree;
}
```

Ипак, највећа предност оваквог приказивања стабла је могућност израчунавања резултата и измене вредности полазећи од листова ка корену, као и унапред познате позиције чворова који чувају резултат интервала који нас занима. Наравно, ова стабла се у пракси много чешће користе јер су једноставнија и за разумевање и за имплементацију и пружају више могућности, а када број елемената низа није једнак неком степену двојке, једноставно се остатак низа допуни до првог наредног степена двојке одговарајућим неутралом за операцију коју то стабло подржава. Јасно је, а и са слике се једноставно може закључити, да је временска сложеност упита измене елемената и израчунавања резултата на интервалу $O(\log_2 N)$.

```
long long getsum(int l, int r) {
	long long sum = 0;
	l += segtree.size() / 2, r += segtree.size() / 2;
	while (l <= r) {
		if (l % 2 == 1) sum += segtree[l++];
		if (r % 2 == 0) sum += segtree[r--];
		l /= 2, r /= 2;
	}
	return sum;
}
void update(int pos, int val) {
	pos += segtree.size() / 2;
	while (pos > 0) {
		segtree[pos] += val;
		pos /= 2;
	}
}
```

Претходне две имплементације приказују сегментно стабло на имплицитан начин. Највећи број проблема се може решити применом неког од тих начина, који су готово еквивалентни, па избор зависи искључиво од тога који начин је некоме једноставнији за разумевање. Остало је да споменемо још један начин имплементације, који се ређе користи, али и те како има своју примену у одређеном скупу проблема. Реч је о експлицитном сегментном стаблу где је родитељски чвор повезан са децом директно. Прецизније, сваки чвор чува показивач на своје лево дете и показивач на своје десно дете. Уколико неко дете не постоји у стаблу, на месту предвиђеном за њега ће у меморији стајати вредност NULL. Специјално, сваки чвор може чувати и показивач на свог родитеља, уколико у задатом проблему та информација може бити корисна. Примећујемо да оваквим приступом сегментно стабло можемо значајно изменити и прилагодити конкретном захтеву, па тако можемо креирати доста другачије структуре помоћу којих се могу решавати проблеми које није било могуће решити стандардном имплицитном имплементацијом сегментног стабла. Једна од најзначајнијих примена овог начина имплементације биће приказана нешто касније кроз пример.

2.2. Примери

Пример 1: Дат је бесконачан низ природних бројева 1,2,3... Над низом је могуће извршити операцију $swap(a, b)$, приликом које елемент на позицији a добија вредност елемента са позиције b и елемент са позиције b добија вредност елемента са позиције a. Наћи број инверзија у низу добијеном након N операција $swap$. Под инверзијом се подразумева пар (i, j) за који важи $i < j$ и $x[i] > x[j]$.

Решење 1: Пошто бројеви a и b могу бити велики, извршићемо **технику компресије низа** која је често заступљена и врло корисна у многим проблемима у којима се користи сегментно стабло и сличне структуре. Све елементе низа (без понављања) над којима треба применити операцију $swap(a, b)$ треба убацити у низ C. Након сортирања низа C, позиција неког елемента у том низу ће даље у задатку једнозначно одређивати тај елемент, тј. уместо вредности $C[i]$ можемо користити i, а позицију неке вредности x у низу C једноставно налазимо помоћу бинарне претраге.

Сада се над копијом низа C, који садржи индексе, једноставно извршавају операције $swap(a, b)$, редом којим су задате, и добијамо резултујући низ индекса R. Пребројавањем инверзија у крајњем низу и поређењем индекса из R са њиховим правим вредностима из C добијамо резултат. Нека је $R[i] = j$. Дакле, j је почетна позиција елемента $C[j]$, а i његова тренутна позиција у низу R. Испред елемента са вредношћу $C[j]$ се у крајњем низу налазе елементи од 1 до $C[i] - 1$, осим елемената $C[1], C[2] \dots C[i-1]$, уместо којих су ту елементи скупа $S = \{C[R[1]], \dots C[R[i-1]]\}$. Потребно је израчунати колико од тих елемената су већи од $C[j]$. Ако је $j < i$, број таквих елемената ван скупа S је $C[i] - C[j] - (i - j)$. У супротном, таквих елемената нема, али у том случају треба пребројати елементе који су мањи од $C[j]$, а већи од $C[i]$ и не налазе се у C. Њих има $C[j] - C[i] - (j - i)$.

За пребројавање елемената скупа S који су већи од $C[j]$ користићемо сегментно стабло. Како је низ C сортиран, проблем се своди на пребројавање елемената из $I = \{R[1], R[2] \dots R[i-1]\}$ који су већи од j. При убацивању сваког новог елемента x у скуп I ажурирамо сегментно стабло додавањем вредности 1 на позицију x. У сваком кораку број елемената мањих од j налазимо као решење упита за суму над интервалом $(0, j-1)$.

Пример 2: Нека је дато N прстенова при чему i-ти прстен има унутрашњи пречник a_i, спољашњи пречник b_i и висину h_i. Користећи дате прстенове изградити кулу максималне висине при чему важи да прстен j може стајати на прстену i само ако је испуњено $a_i < b_j \leq b_i$.

Решење 2: Кључно питање које треба поставити у овом проблему је колика је максимална висина неке куле у чијем се подножју налази прстен i. На прстену i,

могу стајати само прстенови чији спољашњи пречник испуњава неједнакост из услова задатка, па максималну висину постижемо ако од прстенова који испуњавају тај услов одаберемо онај чија је кула максималне висине ако се он налази у подножју. Дакле, поћи ћемо од прстенова најмањег спољашњег пречника. Након што искористимо неки прстен, тј. ставимо га у подножје куле, његов унутрашњи пречник више нема никакву улогу. Пре тога, потребно нам је да интервал $[a_i + 1, b_i]$ буде што већи, па ћемо прстенове једнаког спољашњег пречника сортирати у неопадајућем поретку по њиховом унутрашњем пречнику.

Уочимо и да ће се, од прстенова који имају једнак спољашњи пречник, у кули максималне висине налазити или сви или ниједан. Из тог разлога у сегментном стаблу можемо чувати тренутне резултате за максималну висину куле чији прстен у подножју има спољашњи пречник x. При додавању сваког следећег прстена долази до операције $update(r[i].b, r[i].h + getmax(r[i].a + 1, r[i].b))$ при чему је неопходно извршити компресију низова a и b пре креирања сегментног стабла.

У приказаним примерима смо видели да је сама имплементација сегментног стабла увек слична, а највећи проблем у задацима представља препознавање где се оно може искористити, као и „паковање“ осталих датих вредности, тј. трансформације које треба извршити да бисмо добили сегментно стабло са смисленим вредностима где ће израчунавање резултата за неки интервал имати примену.

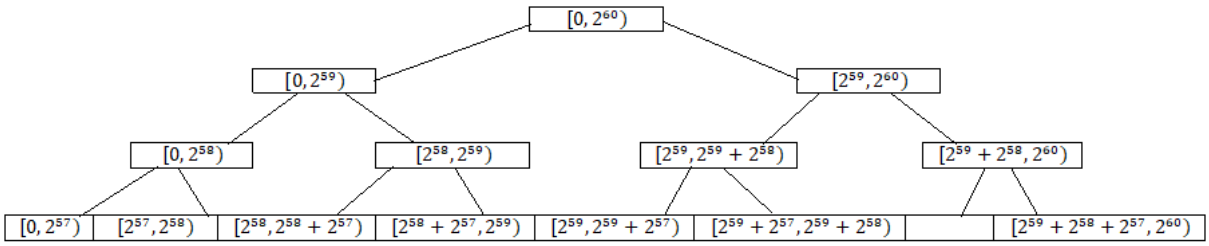
2.3. Техника лење пропагације

Уопштење претходних проблема би било омогућити измену свих елемената из неког интервала и израчунавање резултата на неком интервалу. Ако бисмо за измену сваког елемента из интервала појединачно позивали функцију $update$, јасно је да временска сложеност не би била задовољавајућа. Овај проблем је могуће решити помоћу технике лење пропагације (*lazy propagation*). Као што из назива можемо закључити, циљ нам је да минимизујемо број чворова у стаблу кроз које треба проћи, а да ипак успешно извршимо измену интервала, без губитка података. То можемо постићи на следећи начин: обилазећи стабло од његовог корена (чвора у ком се налази резултат целог стабла), кад год наиђемо на чвор који чува резултат за интервал који је подскуп интервала из упита, у посебно стабло *lazy* (које има идентичну структуру као и главно стабло) уписаћемо нову вредност и сматраћемо да тај чвор покрива и све своје потомке, па њих нећемо обилазити. За то време, над сваким чвором главног стабла које обилазимо, а који не представља подскуп интервала који мењамо, морамо одговарајући број пута извршити тражену операцију (ако је потребно свим елементима из интервала $[2, 6]$ додати вредност 3, онда чвору који је задужен за интервал $[0, 7]$ треба додати вредност $(6 - 2 + 1) \cdot 3$.

На овај начин смо припремили стабла за израчунавање резултата обиласком од корена ка листовима. Кад наиђемо на чвор који чува и резултат за неке позиције које су ван траженог интервала, вредност са тог чвора из стабла *lazy* ћемо пренети на његову децу у стаблу *lazy* и на тај чвор у главном стаблу, а у тај чвор у стаблу *lazy* убацити неутрал за операцију коју извршавамо (нпр. за сабирање 0, множење 1, минимум $+\infty$...). Ово у неким случајевима није неопходно урадити – код рачунања збира све измене можемо додавати у стабло *lazy* и одатле их користити, али у том случају морамо водити рачуна о дужини пресека интервала који нам је потребан и интервала чији се резултат налази у чвору; код минимума можемо имати помоћну променљиву у којој ћемо памтити минимум посећених чворова стабла *lazy*...

Пример 3: Дат је празан скуп S и Q упита, при чему је сваки упит неког од облика: $'1\ l\ r'$ (у скуп S убацити све целе бројеве из интервала $[l,r]$ који већ нису у S), $'2\ l\ r'$ (из скупа S избацити све целе бројеве из интервала $[l,r]$ који су у скупу S), $'3\ l\ r'$ (исписати колико има елемената из интервала $[l,r]$ који се налазе у S), $'4\ p'$ (исписати p-ти најмањи елемент из скупа S; ако не постоји, исписати -1). Пре учитавања сваког наредног упита, неопходно је исписати одговор на претходни упит (уколико су то упити типа 3 или 4). $1 \leq l,r,p \leq 10^{18}$, $Q \leq 10^5$.

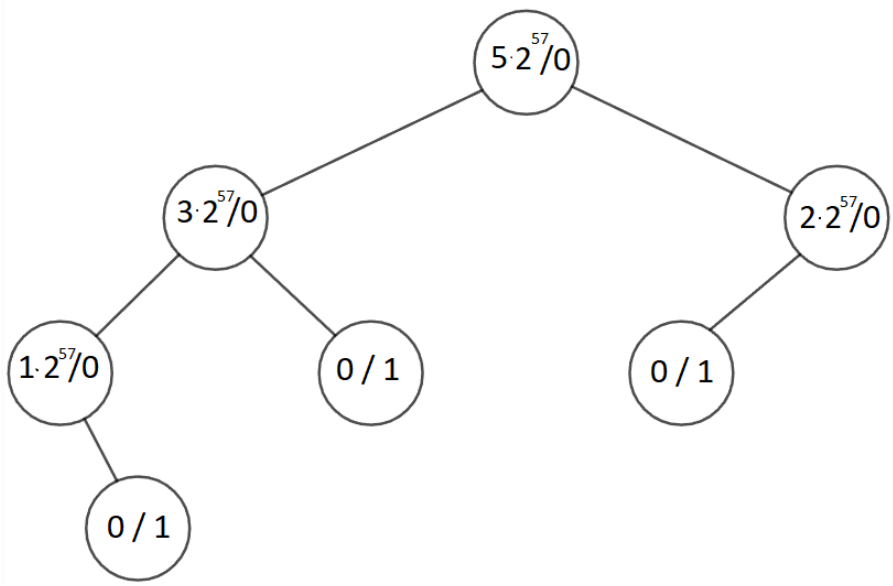
Решење 3: У овом задатку не знамо унапред све упите, па самим тим не знамо унапред које се све границе интервала могу појавити. Из тог разлога не можемо користити компресију низа. Упит у оваквим проблемима се у литератури често назива *online query*. Јасно је да нам је сегментно стабло у овом задатку неопходно, а било каква подела интервала $[1, 10^{18}]$ на мање интервале не би била довољно добра да меморијска и временска сложеност решења буде задовољавајућа. Понашајмо се, ипак, као да у меморији имамо на располагању цело сегментно стабло чији листови треба да имају вредност 1 ако се елемент на тој позицији налази у скупу S. На слици су приказани интервали за које чворови из првих неколико нивоа стабла чувају суме.



Слика 3: Интервали чворова сегментног стабла

Ако стабло обилазимо од врха ка дну користећи технику лење пропагације, сваки упит можемо решити у сложености $O(\log_2 MAX)$. Дакле, једини проблем је креирање стабла чија је меморијска сложеност мања од $O(MAX)$. Идеја је следећа: на почетку је дат празан скуп, што заправо значи да имамо информацију о томе

да су све вредности у стаблу једнаке нули, па можемо само додати корен стабла и у њега уписати вредност 0. Елементе из интервала $[l, r]$ убацујемо у скуп S техником лење пропагације, а све чворове који још нису креирани јер никад до тог тренутка нису посећени, креирамо управо тада. Дакле, у сваком упиту типа 1 или 2 ћемо креирати максимално $2 \cdot \log_2 MAX$ нових чворова, па ће укупна меморијска сложеност бити $O(Q \log_2 MAX)$. Примећујемо да је ово један од ретких случајева где нам, од три описана начина имплементације сегментног стабла, највише одговара експлицитна имплементација стабла, тј. да сваки чвор чува показиваче на своје потомке. Описана структура се у литератури често назива ***sparse segment tree*** или **динамичко сегментно стабло**. На следећој слици је приказана структура овог стабла након додавања вредности 1 свим елементима из сегмента $[2^{57}, 2^{59} + 2^{58})$.



Слика 4: Sparse segment tree и лења пропагација

На крају, неколико кратких упутстава за решавање задатка: код упита првог типа, пре додавања интервала, морамо проверити колико елемената из тог интервала се већ налази у скупу S помоћу функције за суму коју користимо и за упите трећег типа. Међутим, водимо рачуна о томе да нам није значајан само укупан број елемената који се налазе на том интервалу, него за сваки чвор који посетимо треба да сачувамо број елемената из $[l, r]$ који се већ налазе у S на интервалу за који је тај чвор задужен. Код додавања елемената из $[l, r]$ посећујемо исте чворове као и код рачунања суме за интервал $[l, r]$, па се испоставља да нам је ова информација сасвим довољна.

Упити уклањања се раде потпуно аналогно. Једино што треба нагласити јесте чињеница да би они у *lazy* стаблу требало да узимају вредност -1, а вредност 0 треба оставити када нема никаквих промена. Због чега је ово битно? Приликом обиласка стабла, прва вредност *lazy* (она која је најближа корену)

аутоматски поништава $lazy$ вредности свих својих потомака, јер знамо да је то последња извршена промена над скупом S.

Сваки елемент се може у скупу појавити највише једном, па сваки лист у себи носи вредност 0 или 1. Ако са x означимо p-ти најмањи елемент из скупа S, како нема негативних вредности у стаблу, закључујемо да је сума елемената из интервала $[1, x]$ управо p. Вредност x налазимо методом **бинарне претраге у сегментном стаблу**. Ако наиђемо на чвор који нема потомке, нема потребе да тај чвор проширујемо јер сви листови за које је задужен тај чвор се налазе у стаблу ако је вредност тог чвора у $lazy$ стаблу 1, па на леву границу интервала за који је задужен тај чвор додајемо део броја p који је преостао и налазимо вредност x.

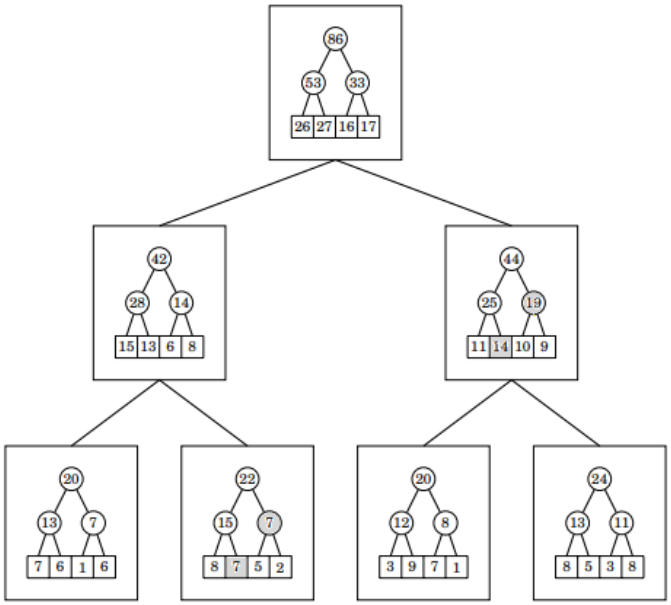
2.4. Перзистентно сегментно стабло

Приказаћемо сада још један тип сегментног стабла за који је погодно користити експлицитан начин имплементације. Са досад описаним техникама можемо решити једну велику класу проблема у којима је потребно извршити неке трансформације низова и затим такве новоформиране низове даље користити. Међутим, може се некад десити да нам је потребно да сачувамо и претходна стања тих низова и да ефикасно обрадимо неки упит над интервалом низа у неком од тих претходних стања. Таквом проблему можемо приступити на више начина од којих ће временска сложеност неких бити задовољавајућа, али ће та решења заузимати превише меморије. Наш циљ је да приликом упита измене садржаја низа додамо минималан број нових чворова у сегментно стабло тако да се сваки упит може још увек решити у временској сложености $O(\log_2 N)$, невезано за то да ли је тај упит над тренутним или неким претходним стањем низа.

Обележимо свако ново стање низа неким временским тренутком $t[i]$. Када дође до промене вредности неког елемента или интервала у низу, мењају се само вредности њихових директних предака у стаблу. Дакле, мења се највише $\log_2 N$ чворова (или $2 \cdot \log_2 N$ уколико мењамо интервал). Уместо измена стабла, старе чворове ћемо задржати непромењене, а за измењене ћемо увести нове чворове. Пошто се увек мења вредност корена стабла, од њега ћемо и почети – увешћемо нови корен ком ћемо доделити тренутно време и нови резултат операције извршене над целим низом. Осим тога, нови чвор ће садржати показивач на чвор задужен за лево подстабло, уколико у том подстаблу нема промена, или ће садржати показивач на нови чвор задужен за лево подстабло са изменама. Интервал у ком нема измена даље не посећујемо и на њега ће се надовезивати стари чворови који ће бити заједнички делови старог и новог стабла. Аналогно је и за десно подстабло. Примећујемо да се уз ову структуру природно уклапа и техника лење пропагације – интервал који треба изменити ће се делити и сужавати и неће бити потребе за посећивање свих листова.

2.5. Дводимензионално сегментно стабло

Проблем из увода са могућношћу измене појединачних елемената матрице можемо ефикасно решити помоћу дводимензионалних сегментних стабала. Као што из самог назива структуре можемо закључити, сваки чвор главног стабла неће бити обична вредност, већ ново сегментно стабло које има конкретно значење. Наиме, можемо направити по једно класично сегментно стабло за сваки ред матрице. Дакле, за стабло које представља i-ти ред матрице, листови су вредности елемената матрице из сваке колоне i-тог реда. Након тога, ова стабла представљају листове главног стабла које представља целу матрицу. На следећој слици је са леве стране приказано дводимензионално сегментно стабло за израчунавање сума над интервалима матрице из горњег десног угла.



	1	2	3	4	5	6	7
1	86	53	33	26	27	16	17
2	42	28	14	15	13	6	8
3	44	25	19	11	14	10	9
4	20	13	7	7	6	1	6
5	22	15	7	8	7	5	2
6	20	12	8	3	9	7	1
7	24	13	11	8	5	3	8

7	6	1	6
8	7	5	2
3	9	7	1
8	5	3	8

Слика 5: Дводимензионално сегментно стабло

У сваком мањем стаблу чвор на одређеној позицији представља исти интервал колона чији резултат обједињује. На тај начин се добија резултат чвора родитељског стабла једноставним поређењем чворова потомачких стабала који се налазе на истим позицијама. Анализирајмо пример са слике. Потребно је израчунати суму елемената из правоугаоника ограниченог тачкама (1,1) и (3,3). Суму елемената из првог реда налазимо из стабла задуженог за први ред. Други и трећи ред имају своје обједињено родитељско стабло, па њихове суме рачунамо налажењем чворова из родитељског стабла који одговарају траженим колонама.

На први поглед, имплементација дводимензионалног стабла делује веома сложено. Међутим, ако се сетимо да укупан број врста и колона допунимо до првог већег степена двојке, свако мање стабло можемо представити обичним једнодимензионалним низом, а цело стабло ће бити дводимензионална матрица. Дакле, меморијска сложеност решења је $O(NM)$.

Креирање стабла од почетне матрице можемо извршити у сложености $O(NM)$ (или позивањем функције *update* за сваки елемент појединачно). Приказом стабла као матрице и имплементација функција измене и суме постаје доста једноставнија. Њихова сложеност је $O(\log_2 N \cdot \log_2 M)$.

```
vector<vector<long long>> create(vector<vector<long long>> a,int n,int m) {
	N = pow(2, (int)log2(n - 1) + 1);
	M = pow(2, (int)log2(m - 1) + 1);
	vector<vector<long long>> tree(2 * N, vector<long long>(2 * M, 0));
	for (int i = 0; i < N; i++)
		for (int j = 0; j < M; j++)
			tree[N + i][M + j] = a[i][j];
	for (int i = 2 * N - 1; i >= N; i--)
		for (int j = M - 1; j > 0; j--)
			tree[i][j] = tree[i][j * 2] + tree[i][j * 2 + 1];
	for (int i = N - 1; i > 0; i--)
		for (int j = 2 * M - 1; j > 0; j--)
			tree[i][j] = tree[i * 2][j] + tree[i * 2 + 1][j];
	return tree;
}
```

3. Фенвиково стабло

3.1. Структура и имплементација

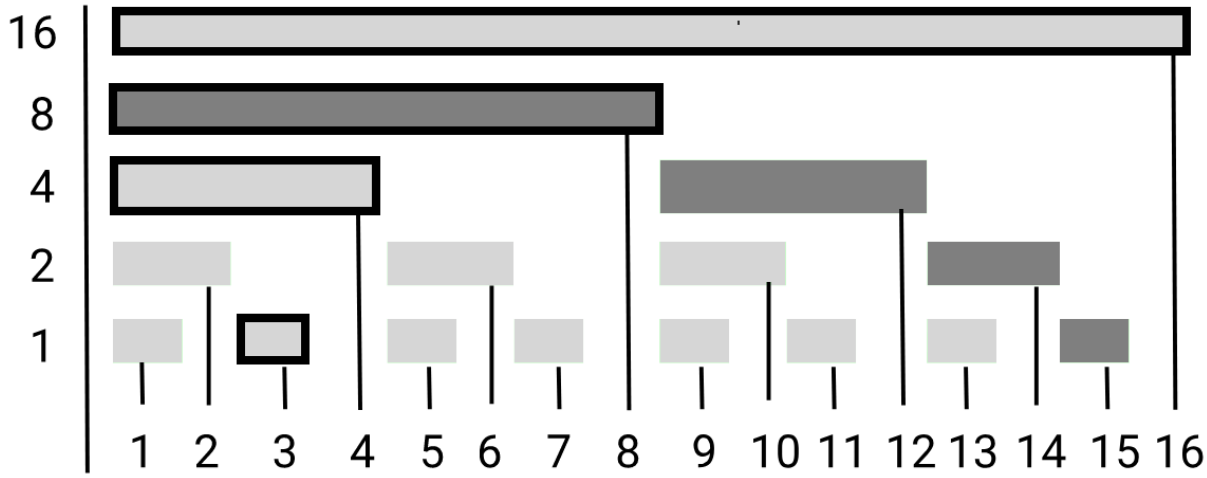
Фенвиково стабло (бинарно индексирано стабло) је структура која на мало другачији начин од сегментног стабла чува резултате операције коју подржава за одређене интервале помоћу којих се ефикасно решавају упити над интервалима. Због специфичне структуре овог стабла, најчешће се користи за израчунавање збира, али постоји и могућност решавања упита са свим осталим операцијама које подржава и сегментно стабло, али у нешто лошијој временској сложености, о чему ће бити више речи касније. Међутим, одговарајућим трансформацијама и комбинацијом више од једног Фенвиковог стабла се оно може користити равноправно са сегментним, а на крају ће бити приказани и случајеви где комбинација неколико Фенвикових стабала пружа ефикасније решење од решења до којих се може доћи помоћу сегментних стабала и техника које су за њих карактеристичне.

За разумевање Фенвиковог стабла корисна је идеја са префиксним сумама примењена у уводном примеру. Помоћу њих је израчунавање суме елемената неког интервала било могуће извршити у сложености $O(1)$, али би се евентуалне измене вредности елемената извршавале у сложености $O(N)$ јер је неопходно изменити префиксну суму свих елемената од те позиције до краја низа. Циљ нам је да прилагодимо ову идеју тако да се обе врсте упита могу извршавати у сложености $O(\log_2 N)$, слично као код сегментног стабла. Дакле, желимо да пронађемо структуру у којој је вредност сваког елемента низа садржана у највише $\log_2 N$ чворова, при чему желимо и да укупан број чворова буде N.

Посматрајмо бинарни запис позиције чију префиксну суму желимо да израчунамо. На основу тога, ту позицију можемо представити као збир неких степена двојке. На пример, $27_{10} = 11011_2 = 16 + 8 + 2 + 1$. Дакле, да бисмо задовољили наведена својства о укупном броју чворова у којима се садржи одређени елемент и постигли ефикасно одговарање на упите, у структури коју формирамо можемо на позицији 16 чувати суму првих 16 елемената низа, на позицији 24 суму елемената низа са позиција $17 - 24$, на позицији 26 суму 25. и 26. елемента, а на позицији 27 вредност елемента низа са позиције 27. Из саме идеје посматрања бинарног записа закључујемо да ћемо приликом израчунавања префиксне суме на овај начин проћи кроз највише $\log_2 N$ чворова.

Како сваки бит представља различит степен двојке, јасно је да за сваку позицију нове структуре можемо јединствено одредити интервал за који она чува суму. Да би број чворова чије вредности треба изменити био минималан и да не би долазило до преклапања интервала приликом измена и израчунавања сума, јединственост се постиже на следећи начин: сваки чвор стабла чува суму

елемената низа на дужини интервала која је једнака највећем степену двојке којим је позиција тог чвора дељива, а десни крај интервала низа једнак је позицији чвора стабла. На следећој слици су приказане позиције основног низа за које је сваки чвор Фенвиковог стабла задужен.



Слика 6: Интервали за које су задужени чворови Фенвиковог стабла

За упите израчунавања префиксне суме на интервалу поступак је прво био приказан кроз пример. Почевши од позиције чију префиксну суму рачунамо, у резултат додајемо вредност из чвора стабла са истом позицијом, а затим од те позиције одузимамо максимални степен двојке којим је она дељива и додајемо вредности у резултат док позицију не доведемо до почетка низа. На слици су затамњени интервали посећених чворова приликом израчунавања префиксне суме првих 15 елемената низа.

Код упита измене треба пронаћи чворове који треба да садрже вредност са позиције низа коју мењамо. Поступак је врло сличан израчунавању префиксне суме, само се одвија у супротном смеру. Полазимо од позиције низа коју мењамо тако што мењамо вредност чвора стабла са истом том позицијом, а затим увећавамо позицију за највећи степен двојке којим је она дељива, све док не дођемо до краја низа. И овде је потпуно јасно да укупан број чворова које треба изменити није већи од $\log_2 N$. На слици су уоквирени интервали посећених чворова приликом измене трећег елемента низа.

Пре имплементације, остало је прокоментарисати налажење највећег степена двојке којим је неки број дељив. Иако ово можемо унапред ефикасно израчунати за сваку позицију и чувати у неком помоћном низу, показаћемо поступак налажења ових степена који је карактеристичан код имплементације Фенвиковог стабла. Налажење највећег степена двојке којим је неки број дељив еквивалентно је налажењу позиције прве јединице (посматрано здесна налево) у бинарном запису тог броја. До те позиције можемо доћи одговарајућим битовним операцијама.

Нека је 2^p највећи степен двојке којим је дељив број n. То другачије можемо записати на следећи начин $n = (2k+1) \cdot 2^p$, где су n, k, p ненегативни цели бројеви. Из претходног добијамо $n = k \cdot 2^{p+1} + 2^p$. Ако уведемо ознаку $m = 2^p$, број n у бинарном запису можемо представити на следећи начин $n_2 = \overline{k_2 m_2}$. Приметимо следеће: ако инвертујемо све битове броја n_2 (све нуле постају јединице и све јединице постају нуле), последњих p позиција броја n_2 постају јединице, док је на позицији $p+1$ цифра нула. Дакле, број који формира последњих $p+1$ цифара једнак је броју $m-1$. Ако инвертовани број повећамо за 1, постижемо да је последњих $p+1$ цифара новог броја једнако тим цифрама полазног броја (тј. оне саме формирају број $m = 2^p$), док ће остале цифре остати инвертоване у односу на n_2. Како нам је циљ да добијени број има само јединицу на позицији $p+1$, погодно нам је да над полазним и новодобијеним бројем искористимо битовну операцију *and* (у програмском језику C++ то је операција &) јер инвертовањем постижемо да број k_2 постане једнак нули, а на осталим позицијама неће бити промена.

Да закључимо, потребно је битове полазног броја прво инвертовати, затим добијени број увећати за 1 и над тако добијеним бројем и полазним бројем применити операцију *and*. Инвертовање битова броја се може постићи на разне начине, којима се овде нећемо бавити јер ћемо коначан израз мало трансформисати. Наиме, негативни бројеви се у рачунару представљају на следећи начин: битови позитивног броја се инвертују и тако добијеном броју се дода 1, а управо то су кораци које смо применили да бисмо добили највећи степен двојке броја n, па је коначан израз којим се то постиже $n \; and \; (-n)$.

Након налажења израза којим се издваја највећи степен двојке којим је број дељив, имплементације функција измене вредности и израчунавања суме на интервалу су врло једноставне, а већ је приказано да обе имају временску сложеност $O(\log_2 N)$, док је меморијска сложеност Фенвиковог стабла $O(N)$.

```
long long getprefixsum(int p) {
	long long sum = 0;
	while (p > 0) {
		sum += fenwick[p];
		p -= p & (-p);
	}
	return sum;
}
```

```
void update(int p, long long val) {
	while (p <= N)
		val += fenwick[p],
		p += p & (-p);
}
long long getsum(int l, int r) {
	return getprefixsum(r) - getprefixsum(l - 1);
}
```

3.2. Фенвиково стабло са другим операцијама

Кроз наредни пример ћемо показати како се Фенвиково стабло може користити за решавање упита са неким другим операцијама које нису сабирање, али ћемо увидети и неке предности сегментног стабла у односу на Фенвиково. Приликом решавања, пођи ћемо од идеја које су интуитивније и једноставније за разумевање, да бисмо на крају дошли до главног решења којим се показује да је сваки проблем, који је могуће решити сегментним стаблом, решив и помоћу Фенвикових стабала у истој временској сложености. Међутим, главна примена Фенвикових стабала остаје у израчунавању префиксних сума, док се за остале операције чешће користе сегментна јер су решења једноставнија за разумевање.

Пример 4: Нека је дат низ природних бројева дужине N и Q упита од којих је сваки облика '1 p v' (на позицију p уписати вредност v) или облика '2 l r' (исписати резултат операције op извршене над елементима низа на позицијама од l до r). Размотрити операције: (а) множење по модулу mod, (б) AND, OR, (в) XOR, (г) минимум, максимум.

Решење 4: Решења ових примера помоћу сегментног стабла су врло слична као и раније описани поступак израчунавања суме. С друге стране, ако бисмо у примеру (а) користили Фенвиково стабло, наишли бисмо на проблем дељења производа првих r елемената производом првих l елемената након већ извршеног рачунања остатка при дељењу, па бисмо добили погрешан резултат. Како бисмо избегли тај проблем, могли бисмо да направимо по једно Фенвиково стабло за сваки прост број мањи од максималног елемента и бројимо колико се пута он садржи у сваком од елемената низа. Међутим, ово решење је чак и лошије сложености од $O(NQ)$, тј. лошије је од сложености решења где у сваком упиту пролазимо кроз све елементе из интервала $[l, r]$.

У примеру (б) такође имамо проблем. Ако је резултат операције AND за првих l елемената једнак 0, то ће бити и резултат за првих r елемената, па не можемо проверити да ли је резултат за интервал $[l, r]$ различит од нуле. Проблем је могуће решити помоћу Фенвиковог стабла ако бисмо посматрали бинарни запис бројева из низа и сваки бит гледали одвојено. Можемо направити по једно Фенвиково стабло за сваки степен броја 2 (укупно $\log_2 MAX$ стабала, при чему је MAX највећи елемент низа) и у њима рачунати колико пута се појављује 1 на тој позицији бинарног записа бројева из неког интервала. Ако је резултат једнак дужини интервала резултат операције AND за ту позицију је 1, а ако је резултат на том интервалу већи од 0, резултат операције OR ће бити 1. Дакле, решење помоћу Фенвиковог стабла овде постоји, али има већу временску ($O((N + Q) \log_2 N \log_2 MAX)$) и меморијску сложеност ($O(N \cdot \log_2 MAX)$) у односу на решење са сегментним стаблом.

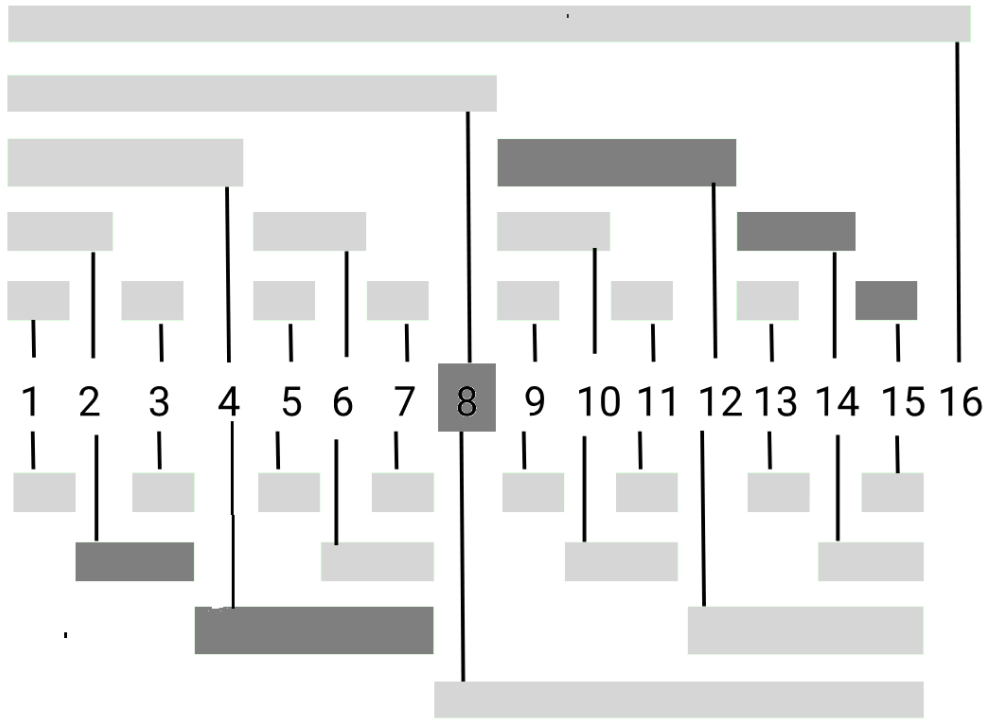
Приметимо да се пример (в) исто може решити техником одвајања на битове и затим провером да ли је резултат на интервалу паран број. Међутим, приликом замене елемента чија је вредност x вредношћу y, можемо једноставно проћи кроз све чворове Фенвиковог стабла у којима је посматрана позиција урачуната и тренутни резултат t на том чвору заменити са $t\ XOR\ x\ XOR\ y$. Дакле, испоставља се да је ово решење потпуно аналогно рачунању суме помоћу Фенвиковог стабла.

У примеру (г) наизалимо опет на проблем као у примерима (а) и (б) јер нам резултати за интервале $[1, l-1]$ и $[1, r]$ не могу дати информацију о резултату за интервал $[l, r]$. Намеће се питање да ли је некако могуће добити резултат за интервал $[l, r]$, а да у том процесу не излазимо из тог интервала. Приметимо следеће: резултат операције $x\&(-x)$ је, као што је раније показано, највећи степен броја 2 којим је број x дељив, али је то уједно и број позиција лево од x за које елемент $fenwick[x]$ чува резултат операције. Дакле, можемо се од елемента r постепено враћати кроз Фенвиково стабло и, када дођемо до корака x за који важи $x - x\&(-x) < l$, уместо да у наш резултат урачунамо вредност $fenwick[x]$, користићемо вредност из почетног низа за позицију x, а затим умањити x за 1 и наставити исти поступак. Колика је сложеност описаног алгоритма? Желимо да израчунамо резултат неке операције на интервалу $[2, 31]$. Део $[17, 31]$ биће израчунат кроз 4 корака помоћу интервала чије су дужине $1, 2, 4$ и 8. Међутим, наилазимо на степен броја 2 и након тог корака десна граница постаје $15_{10} = 1111_2$. До следећег степена двојке долазимо у 3 корака, а након његовог проласка десна граница постаје $7_{10} = 111_2$. Кроз овај пример закључујемо да је временска сложеност приказаног поступка $O((\log_2 n)^2)$. На сличан начин реализујемо и упите у којима треба променити вредност неког елемента (претходно је показано да имамо највише $\log_2 n$ чворова кроз које треба проћи у којима је елемент на тој позицији урачунат и за сваки од њих потребно је максимално $\log_2 n$ корака да би се израчунао нови резултат).

Главно решење:

Кључни недостатак претходне идеје је то што се често заустављамо на чворовима који чувају резултат за веће интервале и након њиховог проласка опет пролазимо кроз чворове са интервалима чије су дужине $1, 2, 4 \ldots$ Ово се може избећи формирањем другог Фенвиковог стабла које ће бити коришћено паралелно са првим. Идеја је да тражени интервал помоћу ових стабала обилазимо са два краја. Прво стабло стандардно чува резултате на начин на који смо га и досад користили и њега користимо за пролазак кроз интервал низа здесна налево. Како друго стабло користимо за обилазак слева надесно, логичан закључак је да ћемо га формирати на „обрнут“ начин у односу на прво. Прецизније, чвор p првог стабла чува резултат за интервал $(p - p\&(-p), p]$, док у

другом стаблу чвор p чува резултат интервала $[p, p + p\&(-p))$. На следећој слици приказани су интервали које чувају одговарајући чворови оба стабла.



Слика 7: Комбинација Фенвикових стабала за ефикасно израчунавање упита са другим операцијама

Приликом упита израчунавања резултата на интервалу $[l, r]$, прво стабло обилазимо почевши од чвора $p = r$ и ово стабло користимо све док важи $p - p\&(-p) + 1 \geq l$. Аналогно, друго стабло обилазимо од чвора $p = l$ и заустављамо се када је испуњен услов $p + p\&(-p) - 1 > r$. На слици су приказани посећени чворови приликом израчунавања резултата за интервал $[2,15]$. Код рачунања минимума, максимума и операција AND, OR не морамо водити рачуна о томе да приликом ова два обиласка може доћи до преклапања интервала (проверити нпр. интервал $[10,15]$). С друге стране, код примера (а), (в), али и у многим другим случајевима, преклапање интервала доводи до погрешног резултата. То можемо избећи тиме што након обиласка првог стабла, поставимо нову десну границу интервала на позицију на којој смо се зауставили. Код упита измене појединачних елемената нема никаквих разлика у односу на стандардан поступак са једним Фенвиковим стаблом.

Користећи описану методу стабло одвајамо на максимално три сегмента: леви, чији резултат израчунавамо помоћу другог (обрнутог) стабла; десни, који израчунавамо помоћу првог стабла; и евентуално један елемент који се неће наћи ни у једном од прва два сегмента, а његову вредност узимамо из полазног низа. Докажимо да може постојати максимално један такав елемент. Уколико такав чвор постоји, он неће бити посећен ни у једном стаблу, а за било који други чвор

p неопходно је да важи следеће: $p - p\&(-p) + 1 \geq l$ или $p + p\&(-p) - 1 \leq r$. Претпоставимо супротно, нека постоје бар два различита чвора p и q који не испуњавају тај услов. Без умањења општости, нека су p и q два најмања броја из $[l, r]$ који не испуњавају тај услов, нека је $p < q$ и $p\&(-p) < q\&(-q)$. Тада је $q - p = p\&(-p)$, па важи $p + p\&(-p) = q \leq r$, што је супротно полазној претпоставци. Ако је $p\&(-p) > q\&(-q)$, онда важи $q - q\&(-q) + 1 = p \geq l$. Доказ је потпуно аналоган у осталим случајевима, а када се за p и q узму било која два узастопна броја који не испуњавају услове, не може бити $p\&(-p) = q\&(-q)$. Временска сложеност описаног алгоритма је $O(\log_2 N)$, а у пракси се показује да ово решење може бити до неколико пута брже од решења са сегментним стаблом.

3.3. Дводимензионално Фенвиково стабло

Решићемо сада пример (б) из увода, али са могућношћу измене елемената. Видели смо да је проблем могуће решити помоћу сегментног стабла, при чему је временска сложеност сваког упита $O(\log_2 n)^2$. Формирајмо сада Фенвиково стабло на следећи начин: нека чвор из реда x и колоне y $fenwick[x][y]$ чува резултат свих елемената $a[i][j]$ који припадају правоугаонику чије горње лево теме има координате $(x - x\&(-x) + 1, y - y\&(-y) + 1)$, а доње десно теме има координате (x, y). Нека су почетне вредности свих елемената матрица a и $fenwick$ постављене на 0. Упите измене вршимо на следећи начин:

```
void update(int x, int y, long long val) {
	for (int i = x; i <= n; i += i & (-i))
		for (int j = y; j <= m; j += j & (-j))
			fenwick[i][j] += val;
}
```

Инверзним поступком добијамо и суму елемената из правоугаоника чије је горње лево теме (1,1), а доње десно (x, y), па суму из неког произвољног интервала добијамо формулом из уводног примера.

```
long long getsum(int x, int y) {
	long long sum = 0;
	for (int i = x; i > 0; i -= i & (-i))
		for (int j = y; j > 0; j -= j & (-j))
			sum += fenwick[i][j];
	return sum;
}
```

Да бисмо се уверили да не долази до преклапања интервала код рачунања суме, на следећој слици је приказано који интервали су у сваком кораку обухваћени код рачунања префиксне суме чвора са координатама (3,7).

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)
(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)
(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)

Временске сложености операција $update$ и $getsum$ су $O(\log_2 N \cdot \log_2 M)$, па је укупна временска сложеност приказаног алгоритма $O((\mathrm{NM} + \mathrm{Q}) \cdot \log_2 N \cdot \log_2 M)$.

3.4. Генерализација проблема *range-update, range-query* на више димензије

Упити измене интервала и израчунавања тренутне вредности једног елемента низа

Потребно је направити помоћни низ d у ком ће се чувати разлика вредности елемената из главног низа са тренутне и претходне позиције. Дакле, низ формирамо у сложености $O(N)$ помоћу формуле: $d_i = a_i - a_{i-1}$. Елементе низа a преко елемената низа d налазимо по следећој формули: $a_i = \sum_{j=0}^{j \le i} d_j$. Додавање вредности x свим елементима на позицијама из интервала $[l, r]$ можемо извршити додавањем вредности x елементу d_l и вредности $-x$ елементу d_{r+1}. Пошто нам је неопходно да после сваке измене интервала ефикасно налазимо тренутне вредности елемената низа a, морамо омогућити ефикасно рачунање префиксних сума низа d, па ћемо над низом d формирати сегментно или Фенвиково стабло, а сви упити измене интервала се, коришћењем низа d своде на измене појединачних елемената које изводимо у сложености $O(\log_2 N)$.

Упити измене интервала и израчунавања суме елемената из интервала (помоћу Фенвиковог стабла)

Идеја је нешто другачија у односу на решење са сегментним стабла. Нека су све вредности на почетку постављене на 0. Након додавања вредности x елементима из интервала $[l, r]$ префиксна сума за елемент i се рачуна по следећој формули:

$$prefix[i] = \begin{cases} 0, & i < l \\ x(i - (l-1)), & l \le i \le r \\ x(r - (l-1)), & i > r \end{cases}$$

Као што видимо, ову формулу можемо раздвојити на два дела: један који је константан и један који зависи од i, па проблем можемо решити коришћењем два Фенвикова стабла. За део који зависи од i идеја је слична као код примера где је било потребно изменити интервал и вратити појединачну вредност: у стабло $biti$ које је задужено за зависан део додајемо вредност x на позицију l и вредност $-x$ на позицију $r+1$. Друго стабло $bitc$ чува константан део на такав начин да укупна префиксна сума може да се израчуна по формули:

$$prefix[i] = i \cdot getsum(biti, i) + getsum(bitc, i).$$

Сређивањем једначина постаје јасан садржај стабла $bitc$: елементу на позицији l треба додати вредност $-x \cdot (l-1)$ (јер је префиксна сума стабла $biti$ на позицији i једнака x, а за укупну префиксну суму се добија $x \cdot i$ уместо $x(i - (l-1))$), а елементу на позицији $r+1$ треба додати вредност $x \cdot r$ (јер на

позицијама које су веће од r префиксна сума из $biti$ је једнака нули, префиксна сума из $bitc$ је једнака $-x \cdot (l-1)$, а резултат треба да буде $x(r-(l-1))$.
Као што видимо, решење се свело на креирање два Фенвикова стабла и укупно четири измене појединачних елемената при сваком упиту измене интервала и рачунања четири префиксне суме при израчунавању суме неког интервала.

```
void updaterange(int l, int r, long long x) {
	update(biti, l, x);
	update(biti, r + 1, -x);
	update(bitc, l, -x * (l - 1));
	update(bitc, r + 1, x * (r - l + 1));
}
```

```
long long getsumrange(int l, int r) {
	return r * getsum(biti, r)
		+ getsum(bitc, r)
		- (l - 1) * getsum(biti, l - 1)
		- getsum(bitc, l - 1);
}
```

Упити измене интервала и израчунавања тренутне вредности једног елемента дводимензионалног низа

Потребно је додати вредност x свим елементима који припадају правоугаонику чије горње лево и доње десно теме имају координате (x_1, y_1) и (x_2, y_2). Као и у једнодимензионалном проблему, имамо помоћну матрицу d у којој чувамо одговарајуће разлике на следећи начин: $d_{i,j} = a_{i,j} - a_{i-1,j} - a_{i,j-1} + a_{i-1,j-1}$. Из претходне формуле изводимо да важи $a_{x,y} = \sum_{i=0}^{i \le x} \sum_{j=0}^{j \le y} d_{i,j}$. Проблем се сада своди на измену појединачних вредности и рачунање префиксних сума у матрици. Дакле, само је потребно формирати дводимензионално Фенвиково стабло над вредностима матрице d и рачунати одговарајуће префиксне суме. Упити измене се своде на додавање вредности x елементима са координатама (x_1, y_1) и (x_2+1, y_2+1) и вредности $-x$ елементима са координатама (x_2+1, y_1) и (x_1, y_2+1).

Упити измене интервала и израчунавања суме елемената из интервала дводимензионалног низа

За почетак, анализирајмо вредности префиксних сума након додавања вредности v елементима подматрице ограничене тачкама (x_1, y_1) и (x_2, y_2).

$$prefix[i,j] = \begin{cases} 0, & i < x_1 \text{ или } j < y_1 \\ v(i-x_1+1)(j-y_1+1), & x_1 \le i \le x_2 \text{ и } y_1 \le j \le y_2 \\ v(i-x_1+1)(y_2-y_1+1), & x_1 \le i \le x_2 \text{ и } j > y_2 \\ v(x_2-x_1+1)(j-y_1+1), & i > x_2 \text{ и } y_1 \le j \le y_2 \\ v(x_2-x_1+1)(y_2-y_1+1), & i > x_2 \text{ и } j > y_2 \end{cases}$$

Из претходне формуле закључујемо да у префиксним сумама имамо делове који зависе само од i, само од j, од производа ij, као и константан део. У складу са тим, формираћемо четири Фенвикова стабла, а коначан резултат $prefix[i,j]$ једнак је $getsum(bitij, i, j) \cdot i \cdot j + getsum(biti, i, j) \cdot i + getsum(bitj, i, j) \cdot j + getsum(bitc, i, j)$.

За измене вредности у сваком од стабала, значајне су четири тачке $A(x_1, y_1)$, $B(x_2 + 1, y_1)$, $C(x_1, y_2 + 1)$ и $D(x_2 + 1, y_2 + 1)$. У стаблу $bitij$ додајемо вредност v тачкама A и D и вредност $-v$ тачкама B и C. У стаблу $biti$ треба додати вредност $v(-y_1 + 1)$ тачкама A и D и вредност $-v(-y_1 + 1)$ тачкама B и C да бисмо задовољили услов $x_1 \leq i \leq x_2$ и $y_1 \leq j \leq y_2$. Међутим, члан i се самостално појављује и у формули која важи при услову $x_1 \leq i \leq x_2$ и $j > y_2$, па у стабло $biti$ треба додати $v(y_2 - y_1 + 1)$ тачки C и вредност $-v(y_2 - y_1 + 1)$ тачки D. Анализа стабла $bitj$ је потпуно аналогна. У стаблу $bitc$ потребно је додати:
тачки A вредност: $v(-x_1 + 1)(-y_1 + 1)$,
тачки C вредност: $-v(-x_1 + 1)(-y_1 + 1) + v(-x_1 + 1)(y_2 - y_1 + 1)$,
тачки B вредност: $-v(-x_1 + 1)(-y_1 + 1) + v(-y_1 + 1)(x_2 - x_1 + 1)$,
тачки D вредност: $v(-x_1 + 1)(-y_1 + 1) - v(-x_1 + 1)(y_2 - y_1 + 1) - v(-y_1 + 1)(x_2 - x_1 + 1) + v(x_2 - x_1 + 1)(y_2 - y_1 + 1)$.

Генерализација на више димензије

Након детаљно описаног поступка извођења свих корака ажурирања интервала код дводимензионалног стабла, постаје јасно да је проблем *range update, range query,* помоћу Фенвиковог стабла, могуће једноставно проширити и на веће димензије. Анализирајмо сложеност општег случаја, d-димензионалног Фенвиковог стабла. Видели смо да су код једнодимензионалног проблема била неопходна два једнодимензионална Фенвикова стабла, а код дводимензионалног проблема четири дводимензионална Фенвикова стабла. Приметимо следеће: број потребних Фенвикових стабала зависи искључиво од тога колико имамо различитих линеарних комбинација које могу формирати координате из d димензија, док димензионалност самих стабала се мора подударати са димензијом проблема који решавамо. Одатле се лако изводи закључак да је неопходно 2^d d-димензионалних Фенвикових стабала. Ово тврђење се другачије може доказати принципом математичке индукције. За измену вредности сваког d-димензионалног стабла неопходно је $O((\log_2 n)^d)$ времена, па је укупна временска сложеност неког упита измене или израчунавања вредности на интервалу d-димензионалног низа једнака $O((2 \cdot \log_2 n)^d)$.

Решење овог проблема је веома значајно с обзиром на то да немамо друге ефикасније структуре за решавање оваквих сложенијих упита над вишедимензионалним низовима. Може се показати и да је технику лење пропагације код вишедимензионалних сегментних стабала могуће искористити само над једном димензијом тог стабла, па се показује да је за проблеме већих димензија Фенвиково стабло далеко погодније од сегментног.

4. Закључак

Теме обухваћене овим радом су највише оријентисане ка такмичарском програмирању и верујем да би рад могао да буде од користи ученицима који треба да се упознају са овим структурама које су веома заступљене на такмичењима. Ако је некоме овај рад управо и послужио у ту сврху, важно је нагласити да се никако на томе не треба зауставити – постоји велики број извора помоћу којих се може усавршавати знање такмичарског програмирања и сматрам да свако може себи пронаћи онај који му највише одговара. Циљ ми је био да обухватим значајније идеје које се провлаче у задацима из ове области, док је у другом делу рада акценат био на вишедимензионалним проблемима који нису толико заступљени на такмичењима, већ они више могу наћи примену у неким другим областима, као што је и наведено у уводу. Осим тога, један од циљева је био и приказати што више особина Фенвиковог стабла која у комбинацији обично дају решење једнаке или чак и боље сложености од сегментних. Приказано је и да се Фенвиково стабло може користити и за операције попут минимума, а сматрам да је тај пример веома користан јер је показано да свака структура може дати више примена од онога за шта је првобитно намењена. Иако можда нема практичну примену јер у ту сврху увек можемо користити сегментно стабло, мислим да управо анализа таквих примера највише доприноси откривању нових ефикаснијих алгоритама и структура.

На крају бих желео да се захвалим професорки Снежани Јелић због пружене подршке при изради рада, али пре свега због њене посвећености приликом припреме часова и великог утицаја на то да стекнем добре основе програмирања за две године и да за то време заволим програмирање и наставим даље самостално да истражујем и напредујем у такмичарском програмирању. Захвалио бих се и осталим професорима информатичких и математичких предмета који су ми предавали и омогућили да стекнем различита знања и заинтересујем се за разне области математике и информатике.

5. Литература

[1] Laaksonen, Antti. "Competitive programmer's handbook." *Preprint* 5 (2017).

[2] Mishra, Pushkar. "A new algorithm for updating and querying sub-arrays of multidimensional arrays." *arXiv preprint arXiv:1311.6093* (2013).

[3] Dima, Mircea, and Rodica Ceterchi. "Efficient range minimum queries using binary indexed trees." *Olympiad in Informatics* 9.1 (2015): 39-44.

[4] https://codeforces.com

[5] https://www.geeksforgeeks.org

[6] https://cp-algorithms.com