Modifikacija steka i reda za nalaženje minimuma za O(1)

Razmotrićemo tri zadatka: modifikaciju steka sa novom operacijom izvlačenja minimuma za `O(1)`, analognu modifikaciju za red i primjenu u zadatku nalaženja minimuma za `O(n)` u svim podsegmentima fiksirane dužine datog niza.

Modifikacija steka

Dodati u stek operaciju izvlačenja minimuma za `O(1)`, a pritom sačuvati složenost opercija dodavanja elementa (`push`) i uklanjanja elementa (`pop`).

U steku čuvamo ne samo elemente, već parove (element, minimumu na steku počevši od tog elementa naniže). Drugim riječima, ako stek predstavljamo kao niz parova (konkretno preko strukture `pair` iz STL-a), tada je

```
stack[i].second = min { stack[j].first }
                  j = 0..i
```

Jasno, nalaženje minimuma u steku se svodi na prosto čitanje vrijednosti `stack.top().second`.

Takođe, pri dodavanju novog elementa u stek, vrijednost `second` biće jednaka vrijednosti `min (stack.top().second, new_element)`. Uklanjanje elementa sa steka ne razlikuje se od uklanjanja elementa iz običnog steka, jer uklonjeni element nikako ne može uticati na preostale elemente.

Implementacija:
`stack< pair<int,int> > st;`
Dodavanje elementa:
```
int minima = st.empty() ? new_element : min (new_element, st.top().second);
st.push (make_pair (new_element, minima));
```
Izvlačenje elementa:
```
int result = st.top().first;
st.pop();
```
Nalaženje minimuma:
```
minima = st.top().second;
```

Modifikacija reda – prvi način

Razmotrićemo prosti način modifikacije reda koji ima manu što red ne može čuvati sve elemente (tj. pri izvlačenju elementa iz reda moramo znati vrijednost elementa koji želimo izvući iz reda). Ovo je veliki nedostatak i relativno rijetko se pojavljuje u zadacima, ali je izuzetno prost. Npr. zadatak nalaženja minimuma na podsegmentima niza se može riješiti primjenom ovakvog reda.

Ključna ideja je da čuvamo samo one elemente koji su nam potrebni za nalaženje minimuma. a ne sve elemente. Naime, prazan red predstavlja nerastući niz brojeva (tj. u glavi reda se čuva najmanja vrijednost) i to ne proizvoljan niz već takav da je glava minimum. Prije dodavanja novog elementa, moramo podrezati dati red: dok se na kraju reda nalazi element veći od novog elementa, uklanjamo taj element iz reda; zatim dodamo novi element na kraj

reda. Na taj način ne narušavamo poredak, i ne gubimo tekući element ako on bude eventualno minimum u nekom od sljedećih koraka. Međutim, pri izvlačenju elementa iz glave reda, moguće je da njega uopšte ne bude u redu, jer ga je naš modificrani red ranije izbacio. Zbog toga pri uklanjanju elementa moramo znati njegovu vrijednost: ako se element sa tom vrijednošću nalazi u glavi reda, izvlačimo ga iz reda; inače, ništa ne radimo.

Razmotrimo realizaciju opisanih operacija:
`deque<int> q;`
Nalaženje minimuma:
```
current_minimum = q.front();
```
Dodavanje elementa:
```
while (!q.empty() && q.back() > added_element)
        q.pop_back();
q.push_back (added_element);
```
Izvlačenje elementa:
```
if (!q.empty() && q.front() == removed_element)
      q.pop_front();
```
Jasno je da je prosječno vrijeme izvršavanja svih operacija O (1).

Modifikacija reda – drugi način

Drugi način je složeniji, ali nema nedostatke prethodnog metoda: svi elementi se stvarno čuvaju u redu i pri uklanjanju elementa nije potrebno znati njegovu vrijednost. Ideja je da svedemo problem na stek, koji smo već riješili. Modeliramo red pomoću dva steka `s1` i `s2`, koji su modificirani za traženje minimuma za `O(1)`. Nove elemente dodajemo samo u stek `s1` a elemente uklanjamo samo iz steka `s2`. Ako pri pokušaju uklanjanja elementa iz `s2` on bude prazan, tada sve elemente iz `s1` prebacimo u `s2` (tj. u `s2` biće elementi iz `s1` u obrnutom redosljedu, što nam odgovara; `s1` biće prazan). Na kraju, nalaženje minumuma se svodi na traženje manjeg od minimuma iz `s1` i minimuma `s2`. Složenost ove operacije je `O(1)`, jer se u najgorem slučaju svaki element jednom dodaje u `s1`, jednom prenosi u `s2` i jednom uklanja iz `s2`.

Implementacija:
`stack< pair<int,int> > s1, s2;`
Nalaženje minimuma:
```
if (s1.empty() || s2.empty())
       current_minimum = s1.empty ? s2.top().second : s1.top().second;
else
      current_minimum = min (s1.top().second, s2.top().second);
```
Dodavanje elementa:
```
int minima = s1.empty() ? new_element : min (new_element, s1.top().second);
s1.push (make_pair (new_element, minima));
```
Izvlačenje elementa:
```
if (s2.empty())
       while (!s1.empty()) {
              int element = s1.top().first;
              s1.pop();
              int minima = s2.empty() ? element : min (element, s2.top().second);
              s2.push (make_pair (element, minima));
       }
result = s2.top().first;
s2.pop();
```

Nalaženje minumuma u svim podsegmentima fiksirane dužine

Dat je niz `A` dužine `N`, i prirodan broj `M ≤ N`. Odrediti minimum u svakom segmentu niza dužine `M`, tj. naći:

```
min A[i],    min A[i],    min A[i],    ...,    min A[i]
0≤i≤M-1      1≤i≤M        2≤i≤M+1              N-M≤i≤N-1
```

Pokažimo rješenje za linearno vrijeme tj. `O(N)`. Dovoljno je uvesti red modificiran za nalaženje minumuma za `O(1)`, pri ćemu možemo koristiti bilo koji od navedenih načina. Dodajmo u red prvih `M` elemenata niza, nađemo minimum i uklonimo ga iz reda, dodamo u red sljedeći element, uklonimo prvi element iz reda, itd. Kako se sve operacije izvršavaju za konstantno vrijeme, to je složenost algoritma `O(N)`.

Prvi način modifikacije je prostiji, ali će vjerovatno biti potrebno čuvati cio niz (jer u `i`-tom koraku nam trebaju `i`-ti i `(i-M)`-ti element niza). Ako koristimo drugi način za red, tada niz `A` ne treba eksplicitno čuvati – dovoljno je znati sljedeći `i`-ti element niza.

Sqrt-dekompozicija

Sqrt-dekompozicija je metod ili struktura podataka koja nam dozvoljava izvršavanje tipičnih operacija kao što su sumiranje elemenata podniza, nalaženje minimuma-maksimuma, itd. za vrijeme O($\sqrt{n}$) , što je značajno brže od O(n) za trivijalni algoritam.

Prvo ćemo opisati strukturu za jednostavne primjene te ideje, a zatim kako uopštiti tu ideju na složenije zadatke. Na kraju, pokazaćemo jednu neuobičajenu primjenu: razbijanje upita na sqrt-blokove.

Struktura podataka sqrt-dekompozicije

Dat je niz a[0..n-1]. Uvesti strukturu koja će za O($\sqrt{n}$) operacija nalaziti sumu a[l..r] za proizvoljne l i r.

Algoritam

Osnovna ideja je da uradimo prethodna izračunavanja: podijelimo niz a na blokove veličine približno $\sqrt{n}$ i u svakom bloku i nađemo sumu b[i] elemenata u takvom bloku. Broj elemenata u bloku je:

$$s = \lceil \sqrt{n} \rceil ,$$

Niz a razbijamo na sljedeće blokove:

$$\underbrace{a[0]\ a[1]\ \ldots\ a[s-1]}_{b[0]}\quad \underbrace{a[s]\ a[s+1]\ \ldots\ a[2\cdot s-1]}_{b[1]}\quad \ldots\quad \underbrace{a[(s-1)\cdot s]\ \ldots\ a[n]}_{b[s-1]}.$$

Iako posljednji blok može imati manje od s elemenata, to ne utiče na rasuđivanje.

Za svaki blok k odredimo zbir elemenata b[k] u bloku:

$$b[k] = \sum_{i=k\cdot s}^{\min(n-1,(k+1)\cdot s-1)} a[i].$$

Dakle, za O(n) smo dobili elemente niza b. Kako iskoristiti niz b za izračunavanje zbira elemenata na segmentu [l,r]? Ako je segment [l,r] dugačak, tada će u njemu biti nekoliko cijelih blokova i za njih već znamo zbir. Ostaju eventualno dva bloka koja djelimično upadaju u segment [l,r] i za te elemente računamo zbir trivijalnim algoritmom.

Na donjoj slici je data ilustracija, gdje k označava broj bloka u kojem leži l a p broj bloka u kojem leži r:

$$\ldots\ \overbrace{a[l]\ \ldots\ a[(k+1)\cdot s-1]\ \underbrace{a[(k+1)\cdot s]\ \ldots\ a[(k+2)\cdot s-1]}_{b[k+1]}\ \ldots\ \underbrace{a[(p-1)\cdot s]\ \ldots\ a[p\cdot s-1]}_{b[p]}}^{sum=?}\ a[p\cdot s]\ \ldots a_r\ \ldots$$

Da bi našli sumu u segmentu [l,r], treba sabrati elemente u „repovima” [l, (k+1)·s-1] i [p·s, r] i sabrati vrijednosti b[i] za sve blokove od k+1 do p-1:

$$\sum_{i=l}^{r} a[i] = \sum_{i=l}^{(k+1)\cdot s-1} a[i] + \sum_{i=k+1}^{p-1} b[i] + \sum_{i=p\cdot s}^{r} a[i]$$

(Napomena: ova formula nije tačna kada je k=p; tada treba samo sabrati elemente od l do r)

Na taj način smo uštedjeli na broju operacija, jer veličina „repa" ne prelazi veličinu bloka s, a broj blokova također ne prelazi s. Kako smo izabrali s da bude približno $\sqrt{n}$, broj operacija je zaista O($\sqrt{n}$).

Implementacija

Prvo prikažimo najprostiju implementaciju:

```
// ulazni podaci
int n;
vector<int> a (n);

// racunanje suma po blokovima
int len = (int) sqrt (n + .0) + 1; // velicina bloka i broj blokova
vector<int> b (len);
for (int i=0; i<n; ++i)
	b[i / len] += a[i];

// odgovor na upit
for (;;) {
	int l, r; // ucitavano granice l i r
	int sum = 0;
	for (int i=l; i<=r; )
		if (i % len == 0 && i + len - 1 <= r) {
		// ako i pokazuje na pocetak bloka ko je cio u [l,r]
			sum += b[i / len];
			i += len;
		}
		else {
			sum += a[i];
			++i;
		}
}
```

Nedostatak ove implementacije je što ima dosta operacija dijeljenja, za koje je poznato da se izvršavaju znatno sporije od ostalih aritmetičkih operacija. Umjesto toga, moguće je izračunati brojeve blokova c_l i c_r u kojima leže granice l i r respektivno i zatim napraviti petlju od blokovima od c_l+1 do c_r-1 i odvojeno napraviti petlje po „repovima" u blokovima c_l i c_r. Treba obratiti pažnju na specijalni slučaj c_l = c_r:

```
int sum = 0;
int c_l = l / len,   c_r = r / len;
if (c_l == c_r)
	for (int i=l; i<=r; ++i)
```

```
            sum += a[i];
else {
      for (int i=l, end=(c_l+1)*len-1; i<=end; ++i)
            sum += a[i];
      for (int i=c_l+1; i<=c_r-1; ++i)
            sum += b[i];
      for (int i=c_r*len; i<=r; ++i)
            sum += a[i];
}
```

Drugi zadaci

Razmatrali smo zadatak zbira elemenata u nekom segmentu niza. Možemo proširiti taj zadatak: dozvoljeno je mijenjati neke elemente niza. Ako se mijenja npr. element a[i] tada je potrebno promijeniti b[k] za onaj blok u kojem leži a[i]. Tada je `k=i/len` i:

$$b[k] += a[i] - old\_a[i].$$

Sa druge strane, umjesto zbira elemenata analogno se može rješavati zadatak o minimalnom ili maksimalnom elementu u nizu. Ako u tim zadacima dopuštamo i promjene nekih elemenata niza, tada je neophodno pregledati čitav blok b[k] u kojem leži promijenjeni element, što traje O(len) = O($\sqrt{n}$) operacija.

Primjenom ovog metoda mogu se rješavati i drugi slični zadaci: broj nula elemenata, prvi element koji je jednak nuli, broj elemenata jednakih datom broju, itd. Postoji i cijela klasa zadataka u kojima je potrebno mijenjati elemente na cijelom segmentu: dodavanje ili dodjeljivanje elemenata u nekom segmentu niza. Na primjer, data su sljedeća dva tipa upita: svim elementima u segmentu [l,r] dodati neku veličinu d i odrediti trenutnu vrijednost elementa `a[i]`. Tada za b[k] uzimamo tu veličinu koju je potrebno dodati svim elementima k-tog bloka (na primjer, na početku je b[k]=0); tada pri upitu tipa dodavanje, treba svim elementima a[i] koji su repovi dodati datu vrijednost d, a zatim uraditi dodavanje svim blokovima koji u cjelosti leže u segmentu [l,r]. Odgovor na upit drugog tipa je prosto a[i]+b[k], gdje je k=i/len. Na taj način se upit dodavanje obavlja za O($\sqrt{n}$) a upit vrijednost određenog elementa za O(1).

Na kraju, moguće je kombinovati oba tipa zadataka: izmjena elemenata na segmentu i upiti na segmentu. Oba tipa operacija imaju složenost O($\sqrt{n}$). Potrebno je kreirati dva niza blokova b i c: jedan za izmjene na segmentu a drugi za odgovore na upite.

Primjenom ove dekompozicije moguće je rješavati i zadatak o skupu brojeva sa operacijama dodavanja i uklanjanja brojeva, provjeru da li broj pripada skupu i traženje k-tog po veličini broja. Potrebno je čuvati brojeve u rastućem poretku, podijeljene u nekoliko blokova sa po $\sqrt{n}$ elemenata u bloku. Pri dodavanju ili brisanju elemenata, potrebno je prebacivati brojeve sa početaka ili krajeva blokova na početke ili krajeve susjednih blokova.

Sqrt-dekompozicija upita

Razmotrimo sada potpuno drugi načim primjene ideje o sqrt-dekompozicji. Pretpostavimo da imamo neke ulazne podatke, a zatim je dato k komandi (upita) koje moramo obraditi i dati odgovor na njih. Upiti mogu biti takvi da ne mijenjaju stanje ulaznih podataka ili mogu mijenjati stanje podataka. Konkretan zadatak može biti veoma složen i njegovo rješenje može biti

izuzetno tehnički složeno i često nedostupno. Sa druge strane, rješenje off-line verzije zadatka (tj. kada nema operacija koje mijenjaju stanje) često je znatno jednostavnije. Pretpostavimo da umijemo riješiti off-line verziju našeg zadataka tj. za vrijeme B(n) napraviti neku strukturu koja umije odgovarati na upite ali ne umije obraditi upite tipa modifikacije podataka. Razbijemo upite na blokove veličine s. Konkretnu vrijednost za s odredićemo kasnije. Na početku obrade svakog bloka, kreirajmo za vrijeme B(n) strukturu za off-line verziju zadataka sa stanjem podataka u momentu početka tog bloka. Sad redom obrađujemo upite u tekućem bloku. Ako je tekući upit tipa modifikacija, propuštamo ga. Ako tekući upit nije modifikacija, tada gledamo našu strukturu podataka i tražimo odgovor, ali **uzimamo u obzir sve upite tipa modifikacije koji su mu prethodili** u tekućem bloku. Taj proces nije moguće uvijek uraditi, a ako je moguće on se mora uraditi dovoljno brzo za vrijeme koje nije mnogo gore od O(S) – označimo to vrijeme sa Q(s). Na taj način, ako imamo m upita, znamo da će nam za njihovu obradu trebati $B(m) \cdot \frac{m}{s} + m \cdot Q(s)$ jedinica vremena. Veličinu s biramo na osnovu konkretnog oblika funkcija B i Q. Na primjer, ako je B(m)=O(m) i Q(s)=O(s), tada je optimalna vrijednost za s približno jednaka $\sqrt{n}$, pa je ukupna složenost O($m\sqrt{m}$) . Kako su ova razmatranja prilično apstraktna, navešćeme nekoliko primjera gdje se ona mogu primjeniti.

Primjer zadatka: dodavanje na segmentu

Dat je niz brojeva a[1..n] i upiti dva tipa: odrediti vrijednost i-tog elementa niza i dodati neki broj x svim elementima niza u segmentu a[l..r].

Iako je ovaj zadatak moguće rješavati i bez sqrt-dekompozicije upita, mi ćemo ga riješiti upravo tom metodom. Razbijemo ulazne upite na blokove veličine sqrt(m), gdje je m broj upita. Na početku prvog bloka nije nam potrebna nikakva struktura, samo čuvamo niz a. Sada idemo po upitima prvog bloka, Ako je tekući upit tipa modifikacije, preskačemo ga, Ako je upit tipa „odrediti vrijednost elementa i“, to prvo uzmemo kao odgovor vrijednost a[i]. Zatim idemo po svi propuštenim upitima modifikacije u tom bloku i za svaki koji uključuje element i primjenimo modifikaciju, tj, dodamo odgovarajući broj x odgovoru. Na taj način smo naučili odgovarati na upite tipa „naći vrijednost elementa i“ za vrijeme O(sqrt(m)). Još samo na kraju bloka moramo primjeniti sve upite tipa modifikacije na niz a, što se može lako uraditi za O(n); za svaki upit dodavanja (l,r,x) u pomoćnom nizu označiti u tački l broj x a u tački r+1 broj –x (minus x) i zatim proći po tom nizu, dodajući tekuću sumu nizu a. Ukupna složenost je $O(\sqrt{m}(n+m))$.

Primjer zadatka: disjoint-set-union sa razdvajanjem

Dat je neusmjereni graf sa n čvorova i m grana. Imamo upite tri vrste: dodati granu (xi, yi), ukloniti granu (xi,yi) i provjeriti da li su povezani čvorovi xi i yi.

Ako ne bi postojali upiti oblika „ukloniti granu“, tada se zadatak može riješiti primjenom strukture disjunktnih skupova (DSU). Pri postojanju upita „ukloniti granu“, ovaj zadatak se značajno usložnjava. Primjenimo sqrt-dekompoziciju upita. Na početku svakog bloka upita, pogledajmo koje sve grane uklanajamo i odmah ih sve uklonimo iz grafa. Sada kreiramo DSU za dobijeni graf. Naša struktura zna sve o svim granama osim o onim koji se dodaju ili uklanjaju iz grafa u tekućem bloku. Dobra vijest je što ne moramo ništa uklanjati iz grafa, to smo već uradili na početku bloka. Dakle, ostala su samo dodavanja grana kojih može biti najviše sqrt(m). Pri odgovoru na tekući upit možemo pošto pokrenuti BFS po komponentama povezanosti DSU koji završi posao za O(sqrt(m)) jer imamo upravo toliko grana.

Off-line zadaci sa upitima na segmentima niza i univerzalna sqrt-heuristika za njih

Razmotrimo još jednu zanimljivu varijaciju ideje sqrt-dekompozicije. Dat je niz brojeva i dobijamo upite oblika (l,r) – odrediti neku informaciju od segmentu [l,r] niza a. Smatraćemo da upiti nisu tipa modifikacije niza i da su nam poznati unaprijed, tj. zadatak je off-line. I posljednje ograničenje: smatramo da umijemo brzo naći odgovor ako se lijeva ili desna granica pomjere za jedinicu (tj. ako znamo odgovor na upit (l,r) možemo brzo naći odgovor na upit (l+1, r) ili (l-1,r) ili (l, r+1) ili (l, r-1)).

Opišimo univerzalnu heuristiku za sve takve zadatke. Sortiramo upite po paru $(l \text{ div } \sqrt{n}, r)$ tj. sortiramo ih po broju sqrt-bloka u kojem leži lijevi kraj segmenta, a ako su jednaki, onda po desnom kraju segmenta. Posmatrajmo sada grupu upita sa istom vrijednošću $l \text{ div } \sqrt{n}$. Svi upiti su sortirani po desnoj granici, pa možemo krenuti od praznog segmenta [l,l-1], i povećavati desnu granicu za 1 – na kraju ćemo odgovoriti na sve upite.

Dobar primjer za primjenu ove heuristike je sljedeći zadatak: odrediti broj različitih brojeva u segmentu [l,r] datog niza a. Ovaj zadatak se teško rješava tradicionalnim metodama.

Nešto složenija varijanta ovog zadatka bila je na jednom Codeforces takmičenju:

Zadatak – Snaga niza (5 sec, 256 MB)
Dat je niz prirodnih brojeva $a_1, a_2, ..., a_n$. Posmatrajmo jedan njegov podniz $a_l, a_{l+1}, ..., a_r$, gdje je $1 \le l \le r \le n$, i za svaki prirodan broj s označimo sa K_s broj pojavljivanja broja s u tom podnizu. Nazovimo **snagom podniza** zbir svih proizvoda $K_s \cdot K_s \cdot s$ po svim mogućim različitim vrijednostima s. Kako je broj različitih brojeva u nizu a konačan, to navedena suma ima konačan broj sabiraka. Potrebno je odrediti snagu svakog od navedenih t podnizova niza a.
Ulaz: Prvi red sadrži dva cijela boja n i t ($1 \le n, t \le 200000$) – dužina niza i broj upita. Drugi red sadrži n prirodnih brojeva a_i ($1 \le a_i \le 10^6$) – elementi niza. Sljedećih t redova sadrže po dva cijela broja l i r ($1 \le l \le r \le n$) – indeksi lijeve i desne granice segmenta niza.
Izlaz: Štampati t redova, gdje i-ti red sadrži samo jedan broj – snagu podniza iz i-tog upita.
Napomena: ne koristite specifikator %lld za učitavanje 64-bitnih cijelih brojeva. Koristiti cout ili specifikator %I64d.

Test primjeri:

Ulaz 3 2 1 2 1 1 2 1 3 Izlaz 3 6	Ulaz 8 3 1 1 2 2 1 3 1 1 2 7 1 6 2 7 Izlaz 20 20 20

Pojašnjenje primjera 2: posmatrajmo podniz [2,7] (elementi podniza su obojani):

Tada je $K_1 = 3$, $K_2 = 2$, $K_3 = 1$, i snaga je $3^2 \cdot 1 + 2^2 \cdot 2 + 1^2 \cdot 3 = 20$.

Fenvikovo drvo

Fenvikovo drvo je struktura koja ima sljedeće osobine:

1. dozvoljava izračunavanje vrijednosti neke reverzibilne operacije G na svakom segmentu [L, R] za vrijeme **O (log N)**;
2. dozvoljava izmjenu bilo kog elementa za **O (log N)**;
3. količina memorije je **O (N),** tačno onoliko kolika je veličina ulaznog niza;
4. lako se uopštava na višedimnezionalne nizove.

Najčešća upotreba Fenvikovog drveta je za izračunavanje zbira na segmentu tj. funkcija je G(X1, ..., Xk) = X1 + ... + Xk.

Ova struktura je prvi put opisana u radu "A new data structure for cumulative frequency tables" (Peter M. Fenwick, 1994).

Algoritam

Pretpostavićemo da je operacija G sabiranje, što pojednostavljuje opis. Dat je niz A[0..N-1]. Fenvikovo drvo je niz **T**[0..N-1], u čijem se svakom elementu čuva zbir nekoliko elemenata niza A:

T_i = zbir A_j za sve **F(i) <= j <= i**,

gdje je F(i) – neka funkcija koju ćemo odrediti kasnije.

Sada možemo napisati pseudokod za funkciju nalaženja zbira na segmentu [0, R] i za izmjenu elementa:

```
int sum (int r)
{
	int result = 0;
	while (r >= 0) {
		result += t[r];
		r = f(r) - 1;
	}
	return result;
}

void inc (int i, int delta)
{
	Za svako j za koje je F(j) <= i <= j
	{
		t[j] += delta;
	}
}
```

Funkcija sum radi na sljedeći način: umjesto pregleda svih elemenata niza A, krećemo se po nizu T, praveći skokove po segmentima tamo gdje je to moguće. Prvo se odgovoru dodaje zbir na segmentu [F(R); R], zatim na segmentu [F(F(R)-1); F(R)-1], i tako dalje dok se ne dođe do nule.

Funkcija inc kreće se u obratnu stranu tj, uvećavanja indeksa, mijenjajući vrijednosti zbira T_j samo na onim pozicijama na kojim je to potrebno, tj. za sve j za koje je F(j) <= i <= j.

Očigledno od izbora funkcije F zavisi brzina izvršavanja ovih operacija. Definišimo F tako da dobijemo logaritamsku složenost.

Posmatrajmo binarni zapis broja X u njegov najmlađi bit. Ako je taj bit jednak 0, tada je F(X) = X. U suprotnom, binarni zapis se završava sa grupom od jedne ili više jedinica. Zamijenimo sve te jedinice sa nulama i dobijeni broj biće vrijednost funkcije F(X). Ovako složeni opis odgovara veoma prostoj formuli:

`F(X) = X & (X+1)`

Ovdje je & operacija bit po bit konjunkcije („bitwise and“) . Ostalo je još da nađeno način da brzo nalazimo brojeve j takve da je F(j) <= i <= j. Lako se vidi da se takvi brojevi j dobijaju iz i uzastopnim zamjenama krajnje desne (najmlađe) nule u binarnom zapisu broja i. Na primjer, za i=10, dobijamo j=11,15, 31, 63... I ovoj operaciji odgovara jednostavna formula:

`H(X) = X | (X+1).`

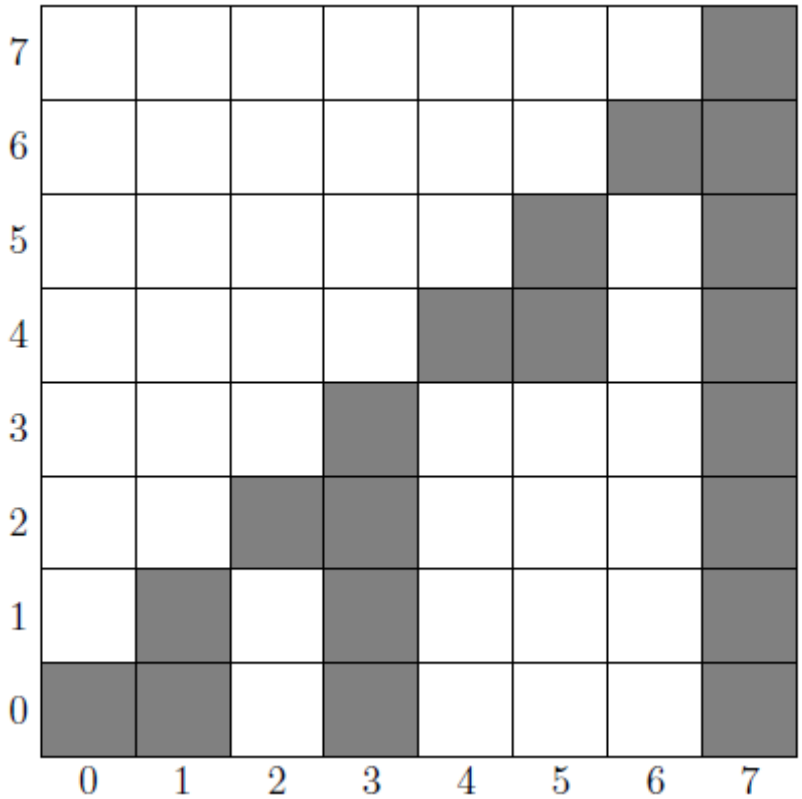
Ovdje je | operacija bit po bit disjunkcije („bitwise or“).

Ako tražimo zbir na segmentu [L,R] tada se on dobija po formuli:

`rsq(L,R) = sum(R) - sum (L-1).`

Ovdje je rsq skraćenica za „range sum query“.

Na donjoj slici vide se elementi niza T za slučaja n=8:



Implementacija za 1D niz i zbir

```
vector<int> t;
int n;

void init (int nn)
{
	n = nn;
	t.assign (n, 0);
}

int sum (int r)
{
	int result = 0;
	for (; r >= 0; r = (r & (r+1)) - 1)
		result += t[r];
	return result;
}

void inc (int i, int delta)
{
	for (; i < n; i = (i | (i+1)))
		t[i] += delta;
}

int sum (int l, int r)
{
	return sum (r) - sum (l-1);
}

void init (vector<int> a)
{
	init ((int) a.size());
	for (unsigned i = 0; i < a.size(); i++)
		inc (i, a[i]);
}
```

Implementacija za 1D niz i minimum

Fenvikovo drvo dozvoljava da nađemo vrijednost funkcije na proizvoljnom segmentu [0;R], pa nikako ne možemo naći minimum na segmentu [L;R] gdje je L > 0. Dalje, sve promjene mogu ići samo stranu umanjenja (opet, ne možemo dobiti inverznu funkciju od min). Ovo su sve značajna ograničenja.

```
vector<int> t;
int n;

const int INF = 1000*1000*1000;

void init (int nn)
{
```

```
	n = nn;
	t.assign (n, INF);
}

int getmin (int r)
{
	int result = INF;
	for (; r >= 0; r = (r & (r+1)) - 1)
		result = min (result, t[r]);
	return result;
}

void update (int i, int new_val)
{
	for (; i < n; i = (i | (i+1)))
		t[i] = min (t[i], new_val);
}

void init (vector<int> a)
{
	init ((int) a.size());
	for (unsigned i = 0; i < a.size(); i++)
		update (i, a[i]);
}
```

Implementacija za 2D niz i zbir

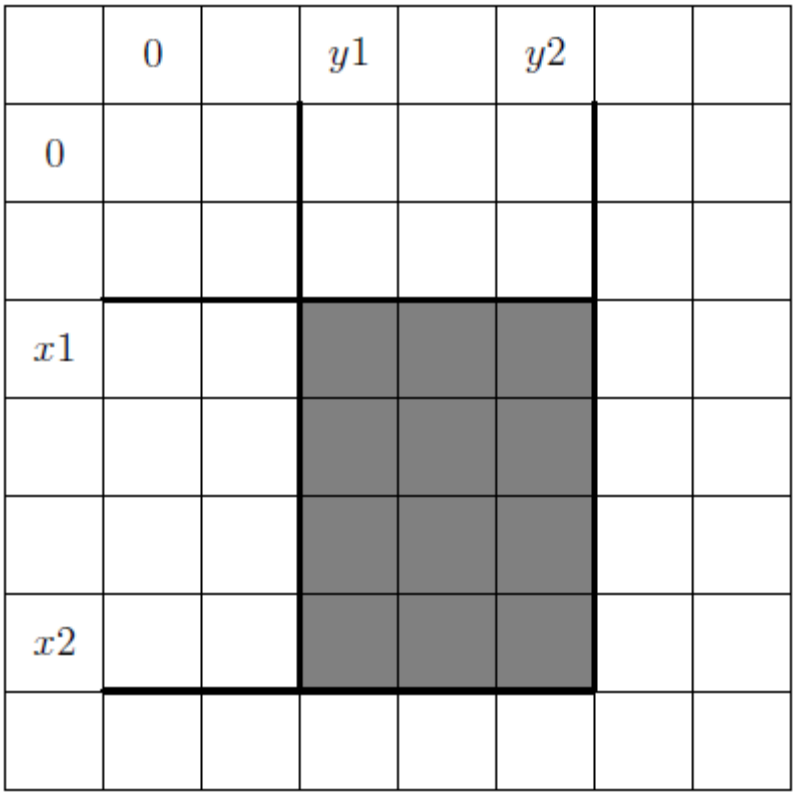
Fenvikovo drvo se lako uopštava na višedimenzionalni slučaj:

```
vector <vector <int> > t;
int n, m;

int sum (int x, int y)
{
	int result = 0;
	for (int i = x; i >= 0; i = (i & (i+1)) - 1)
		for (int j = y; j >= 0; j = (j & (j+1)) - 1)
			result += t[i][j];
	return result;
}

void inc (int x, int y, int delta)
{
	for (int i = x; i < n; i = (i | (i+1)))
		for (int j = y; j < m; j = (j | (j+1)))
			t[i][j] += delta;
}
```

Lk



Ako sa `rsq(x1,y1,x2,y2)` označimo sumu u pravougaoniku sa slike, tada je:

`rsq(x1,y1,x2,y2)=sum(x2,y2)-sum(x1-1,y2)-sum(x2,y1-1)+sum(x1-1,y1-1).`

Online judges zadaci

Spisak zadataka sa primjenom Fenvikovog drveta:

acm.timus.ru – 1028 Stars [složenost: srednja]
UVa 11525 - Permutation
UVa 11610 - Reverse Prime
The 2006 ACM Asia Programming Contest - Potentiometers