HASHING

1. KRATKI UVOD I MALO TEORIJE

ŠTO JE TO HASHING?

Prilikom rješavanja zadataka često koristimo ispitivanje jednakosti nekakvih objekata. Ako se radi o intovima naš je problem vrlo jednostavan. Što ako imamo hrpu nizova intova i zanima nas koji od njih su međusobno jednaki? Možemo ih uspoređivati element po element, no to se ne čini previše efikasno. Pogledajmo o čemu se radi na malo apstraktnijoj razini.

Imamo neki veliki skup **A** i njegov podskup **B** čije elemente želimo brzo uspoređivati. Označimo veličinu jednog elementa s **n** i pretpostavimo da je složenost usporedbe koju radimo na elementima O(**n**). Za nešto veće **n** ta složenost počinje utjecati na ukupnu složenost algoritma u kojem koristimo predviđenu usporedbu. Kako bismo tome doskočili uvodimo treći skup **C** znatno manji od **A** te posebnu funkciju $\boldsymbol{h} : \mathbf{A} \rightarrow \mathbf{C}$. Za funkciju ***h*** kažemo da je ravnomjerna ako mapira otprilike jednak broj elemenata **A** u elemente **C**. Pritom omjer **|A|/|C|** nazivamo *load factor* i predstavlja očekivani broj elemenata **A** koje ***h*** mapira u neki element **C**. Funkciju ***h*** nazivamo *hash* funkcija, skup **A** skupom svih mogućih *ključeva*, a skup **C** skupom *hash* vrijednosti.

Hashing možemo definirati kao mapiranje nekih većih objekata u manje kako bismo ih lakše uspoređivali. Pritom postoji problem koji se naziva *kolizija* i posljedica je činjenice da ovim postupkom gubimo informacije i da se dvije vrijednosti mogu mapirati u istu *hash* vrijednost.

JEDAN JEDNOSTAVAN PRIMJER

Recimo da želimo implementirati trodimenzionalni niz intova čija je svaka dimenzija veličine milijun. Takav niz ne možemo realno ugurati u memoriju pa je potrebno iskoristiti neku drugu foru. Jedan od očitih pristupa bilo bi korištenje STL *map* strukture, ali ovdje nam je cilj pokazati što možemo izvesti korištenjem *hash* funkcija. Skup **A** u ovom primjeru sadrži sve moguće trojke intova do milijun, dakle ima veličinu 10^{18}. Te ćemo trojke prvo morati smanjiti na neku prihvatljivu veličinu. Funkcija koju ćemo za trojku (x, y, z) koristiti je sljedeća:

$$h(x, y, z) = (xa^2 + ya + z) \bmod p$$

Za p ćemo u našem slučaju uzeti broj 1000003, a za a neki nasumični broj iz intervala [0, p-1]. Nula i jedan se ovdje možda i ne čine kao najbolji izbor. Vrijednosti koje ćemo dobiti takvim mapiranjem trojki biti će puno manje od početnih pa ćemo za naš fiktivni niz N[x][y][z] zapravo koristiti nešto manji niz N[h(x,y,z)].

Kako će tu i tamo sigurno doći do preklapanja odnosno *kolizije*, morat ćemo doskočiti i tom problemu.

CHAINING METODA RJEŠAVANJA KOLIZIJE

Ova metoda rješavanja preklapanja vrlo je jednostavna i vjerojatno prva pada na pamet kao nadogradnja naše dosadašnje ideje. Što ako N[h(x, y, z)] nije samo niz intova nego lista četvorki (x, y, z, v)? Prva tri elementa (x, y, z) predstavljaju trojku a v vrijednost koja je toj trojki pridružena u našem nizu, odnosno N[x][y][z].

Ako želimo elementu na poziciji (x, y, z) dodijeliti neku vrijednost v prvo ćemo ga potražiti u listi koja mu je pridružena već opisanom funkcijom. Ako ga ne nađemo na kraj liste dodat ćemo četvorku (x, y, z, v). Ako ga pak pronađemo jednostavno ćemo modificirati vrijednost v u pripadajućoj četvorki.

Upravo smo vrlo jednostavno implementirali jako velik niz. No jedna stvar još nije jasna. Kako znamo da će naša funkcija ravnomjerno raspoređivati trojke?

MALA ANALIZA HASH FUNKCIJE

Funkcija koju smo definirali polinom je u a. Pogledajmo što dobijemo izjednačavanjem funkcije za dvije trojke (x_0, y_0, z_0) i (x_1, y_1, z_1):

$$h(x_0,\ y_0, z_0) = h(x_1, y_1, z_1)$$

$$(x_0 - x_1)a^2 + (y_0 - y_1)a + (z_0 - z_1)\ \equiv 0\ (mod\ p)$$

Jednakost dakle dobivamo u slučaju kada je a nultočka nekog trećeg polinoma modulo p. Polinom je maksimalno drugog stupnja pa tako ima maksimalno dvije različite nultočke (ovo usput nije odmah očito i vrijedi samo za prost p). Vjerojatnost da je nasumični a nultočka ovog polinoma tako je $2/p$, oko $2 \cdot 10^{-6}$ za naš p, što smatramo jako malim brojem i možemo očekivati da će funkcija biti prilično ravnomjerna.

Sada ako netko ne poznaje naš izbor a i p vrlo mu je teško namjestiti primjer u kojem se puno trojki mapira u istu vrijednost. Osim ovakve analize moguće je funkciju implementirati i isprobati u praksi. Tu treba imati na umu i „pravilne“ inpute koji iz perspektive *hash* funkcije ne bi smjeli biti drukčiji od nasumičnih.

Dobro je primijetiti kako se ovaj primjer jednostavno poopći i na razne druge tipove indeksa i time dobivamo strukturu *hash* tablice, vrlo korisnu za brzo mapiranje.

2. HASHING STRINGOVA

Od svega što se može susresti na natjecanjima stringovi su uvjerljivo najčešće žrtve hashiranja. Usporedbe stringova znak po znak nisu uvijek dovoljno efikasne, pa ćemo zato sada proučiti jednu vrlo popularnu metodu *hashiranja* stringova koja dozvoljava brzo izdvajanje pojedinih podstringova (tj. dobivanje njihovih *hash* vrijednosti). Također ćemo vidjeti vrlo efikasan način leksikografskog uspoređivanja stringova.

HASH FUNKCIJA ZA STRINGOVE

Hash funkcija koju obično koristimo za stringove također ima oblik polinoma, gdje su koeficijenti znakovi samog stringa. Vrijednosti ćemo pohranjivati u intove, ovaj puta bez direktnog računanja nekog ostatka (za to će se pobrinuti sama implementacija inta).

Nazovimo string s kojim radimo **S**, njegovu duljinu **n** i neki prosti broj *p*. Pomoćne nizove **P** i **H** definiramo rekurzivno na sljedeći način:

$$P_0 = 1, P_i = pP_{i-1}$$

$$H_0 = S_0, H_i = pH_{i-1} + S_i$$

Niz **P** očito je jednostavno niz potencija broja *p*. No što nam predstavlja niz **H**? Izgleda kao da ima veze s polinomima, preciznije jako je sličan Hornerovom algoritmu za evaluiranje polinoma u nekoj točki. Isti taj niz zato možemo zapisati u nešto preglednijem obliku:

$$H_i = \sum_{j=0}^{i} S_j P_{i-j}$$

Kao *hash* vrijednost zadanog stringa podrazumijevamo $\mathbf{H_{n-1}}$, a ostale članove niza nazivamo *prefiks hashevima*. Ovdje ću još samo napomenuti kako za samu *hash* vrijednost $\mathbf{S_j}$ ovdje uzimamo vrijednost pripadajućeg chara. Također nećemo brinuti o *overflow* problemima koji bi eventualno nastali i jednostavno ćemo ih ignorirati. Možete se uvjeriti kako se sve operacije u tom slučaju slične onima modulo 2^{32}.

Sljedeći korak je izdvajanje *hash* vrijednosti podstringova stringa **S**. Označimo sa **S[i,j]** podstring stringa **S** koji se provlači od znaka **i** do znaka **j**. Po našoj definiciji *hash* vrijednosti stringa dobivamo sljedeću sumu:

$$h(S[i,j]) = \sum_{k=i}^{j} S_k P_{j-k}$$

Začudo tu vrijednost možemo jednostavno dobiti iz već izračunatog pomoćnog niza **H**.

Sljedeća formula daje nam tu poveznicu:

$$h(S[i,j]) = H_j - H_{i-1}P_{j-i+1}$$

Ideja kojom dolazimo do te formule je ta da prvo uzmemo prefiks **S[0,j]** i tada od njega jednostavno odrežemo prefiks **S[0,i-1]**. Kako je znak **i-1** u odnosu na znak **j** na potenciji višoj za **j-i**+1 (što odgovara duljini podstringa), taj drugi prefiks potrebno je pomnožiti s P_{j-i+1}. Pri konkretnoj implementaciji ove formule naravno moramo paziti na to da ne gledamo negativne indekse.

Sad već posjedujemo puno alata i bacamo se na neke jednostavnije primjene.

TRAŽENJE MALOG STRINGA U VEĆEM

Sad bi već trebalo biti očito kako ćemo pristupiti ovom problemu. Pretpostavimo da imamo dva stringa **S** i **Z** s duljinama redom **n** i **m**, i neka je **n** manje od **m** tj. tražimo **S** u **Z**. Za početak izračunamo hash vrijednost stringa **S** – h_S i pomoćne nizove **H** i **P** za string **Z**. Tada se ovo traženje svodi na ispitivanje je li *h(***Z[i,i+n-1]**) jednak h_S za sve odgovarajuće indekse **i** (ima ih očito **m**-**n**+1).

Ova ideja čini osnovu Rabin-Karp algoritma za traženje patterna. Primijetite i da je ovo moguće izvesti i bez pomoćnih nizova za string **Z** ako pri pomicanju na sljedeći indeks održavamo vrijednost h_Z kao *hash* vrijednost trenutno promatranog podstringa. Tada pomoću niza **P** izbacujemo prvi znak iz trenutnog *hasha* i dodajemo novi znak na kraj.

Kako bismo bili potpuno sigurni da nemamo slučaj *kolizije* potrebno je i ručno provjeriti svaki podstring koji se pokaže jednak po *hash* vrijednosti, no to će se uz zadanu *hash* funkciju događati vrlo rijetko. Zbog toga se čak možemo praviti da *hash* vrijednost predstavlja sam string.

Sličan princip možemo primijeniti na višedimenzionalne nizove. U 2d slučaju možemo na primjer prvo izračunati *hash* vrijednosti stupaca (i njihovih podstringova), i zatim te podstringove promatrati kao jednodimenzionalne nizove. Također možemo lako prebrojavati podstringove određene fiksne duljine i još mnogo toga.

PALINDROMI I PERIODIČNOST

Palindromi su još jedan primjer gdje *hashing* može dobro doći. Ako umijesto samo jednog pomoćnog niza **H** konstruiramo i niz $\mathbf{H^r}$ kao niz *prefiks hasheva* obrnutog stringa. Označimo i obrnuti string **S** sa $\mathbf{S^r}$. Tada ispitivanje je li podstring **S[i,j]** palindrom svodi na ispitivanje jednakosti *h*(**S[i,j]**) i *h*($\mathbf{S^r}$**[n-j**-1,**n-i**-1**]**).

Periodičnost stringa također je jednostavna za provjeriti, uzmemo svaki prefiks stringa, pomičemo se za njegovu duljinu i pritom testiramo jednakost. Budući da duljina mora biti djeljiva duljinom perioda broj početnih prefiksa je prilično ograničen.

NAJDULJI ZAJEDNIČKI PREFIKS

Najdulji zajednički prefiks (engl. *longest common prefix*, skraćeno LCP) dvaju stringova definira se kao najdulji prefiks prvog stringa koji je ujedno i prefiks drugog stringa. Rubni slučaj je naravno onaj u kojem je jedan od stringova prefiks drugoga. Traženje LCP-a može se vrlo efikasno izvesti korištenjem već opisane *hash* funkcije i binarnog pretraživanja. Uistinu ako je string **A** prefiks stringa **B** tada je svaki prefiks stringa **A** također prefiks stringa **B**. Pretpostavimo da su nam zadana dva podstringa stringa **S**, redom $\mathbf{S[x_0,y_0]}$ i $\mathbf{S[x_1,y_1]}$. Tada bi C kod za računanje duljine njihovog LCP-a išao otprilike ovako:

```
int LCP( int x0, int y0, int x1, int y1 ) {
  if( S[x0] != S[x1] ) return 0;
  int lo = 1, hi = min( y0-x0+1, y1-x1+1 ), mid;

  while( lo < hi ) {
    mid = ( lo + hi + 1 ) / 2;
    if( hash( x0, x0+mid-1 ) == hash( x1, x1+mid-1 ) ) lo = mid;
    else hi = mid-1;
  }

  return lo;
}
```

Jasno je što kod radi pa ga nećemo specijalno pojašnjavati nego idemo pogledati kako se primjenjuje.

LEKSIKOGRAFSKA USPOREDBA PODSTRINGOVA

Usporedba dva podstringa vrlo je jednostavno i efikasno izvediva koristeći LCP. Evo odmah C koda:

```
bool cmp( int x0, int y0, int x1, int y1 ) {
  int L = LCP( x0, y0, x1, y1 );
  if( L == y0 - x0 + 1 ) return 1;
  if( L == y1 - x1 + 1 ) return 0;
  return ( S[x0+L] < S[x1+L] );
}
```

Funkcija *cmp* dakle vraća vrijednost *true* u koliko je prvi podstring leksikografski manji od drugog. Složenost jedne komparacije je O(lg N). Primjer zadatka u kojem se ovakva leksikografska komparacija može iskoristiti je pronalaženje leksikografski minimalne rotacije nekog stringa. Ako tražimo takvu rotaciju u stringu **S** onda je jedno elegantno rješenje promatrati string **Z** = **S** + **S**. Kao početne pozicije promatramo prvih pola i uspoređujemo podstringove leksikografski s onima već viđenim (detaljnije u zadacima).

Još jedna primjena ovog komparatora je izgradnja jedne od najvažnijih struktura za obradu stringova tzv. *suffix arraya*. Ideja je sortirati sve sufikse leksikografski (koristeći njihove početne indekse). U takvom nizu sufiksa možemo vrlo efikasno tražiti podstringove, kao i rješavati još nebrojene teške probleme. Nažalost ta struktura daleko nadilazi ovaj kratki uvod, no pozivam sve zainteresirane da strukturu prouče na webu.

3. PAR (DOSLOVNO PAR) RIJEŠENH ZADATAKA

NAJDULJI PALINDROMSKI PODSTRING

Zadatak možete pronaći na adresi: http://www.spoj.pl/problems/LPS/.

Ukratko, zadan je string **S** duljine do 10^5 i moramo pronaći duljinu najduljeg podstringa koji je ujedno i palindrom. Ključ rješenja jest primijetiti kako svaki palindrom ima nekakvu (ovisno o parnosti duljine) sredinu. U koliko je neparan tada je u sredini neki znak, a inače dva znaka dijele tu sredinu. Sljedeća važna činjenica je ta da brisanjem krajnja dva znaka nekog palindroma opet dobivamo palindrom. To nas navodi na zaključak kako je traženu duljinu za fiksirani centar moguće pronaći binarnim pretraživanjem. Evo koda:

```
#include <cstdio>
#include <cstring>

#include <vector>
#include <algorithm>

using namespace std;
const int p = 10007;

#define MAX 100005

int N;
char S[ MAX ];

int P[ MAX ];
int H[ MAX ];
int Hr[ MAX ];

int _hash( int *H, int i, int j ) {
  int ret = H[j];
  if( i ) ret -= H[i-1] * P[j-i+1];
  return ret;
}

int h( int i, int j ) { return _hash( H, i, j ); }
int hr( int i, int j ) { return _hash( Hr, N-j-1, N-i-1 ); }

int neparni( int i ) { // najdulji neparni palindrom s centrom u i
  int lo = 0, hi = min( i, N-i-1 ), mid;

  while( lo < hi ) {
    mid = ( lo + hi + 1 ) / 2;
    if( h( i - mid, i + mid ) == hr( i - mid, i + mid ) ) lo = mid;
    else hi = mid - 1;
  }

  return 2 * lo + 1;
}

int parni( int i ) { // najdulji parni palindrom s centrom izmedju i i i+1
```

```
  int lo = 0, hi = min( i+1, N-i ), mid;

  while( lo < hi ) {
    mid = ( lo + hi + 1 ) / 2;
    if( h( i - mid + 1, i + mid ) == hr( i - mid + 1, i + mid ) ) lo = mid;
    else hi = mid - 1;
  }

  return 2 * lo;
}

int main( void )
{
  scanf( "%d", &N );
  scanf( "%s", S );

  P[0] = 1;
  H[0] = S[0];
  Hr[0] = S[N-1];

  for( int i = 1; i < N; ++i ) {
    P[i] = P[i-1] * p;
    H[i] = H[i-1] * p + S[i];
    Hr[i] = Hr[i-1] * p + S[N-i-1];
  }

  int Sol = 0;

  for( int i = 0; i < N; ++i )
    Sol = max( Sol, neparni( i ) );

  for( int i = 0; i < N-1; ++i )
    Sol = max( Sol, parni( i ) );

  printf( "%d\n", Sol );

  return 0;
}
```

Ovo je dakle čitav kod koji rješava problem u složenosti O(N lg N). Okvirni algoritam je fiksiranje svih mogućih centara i za svaki binarno pretraživanje najveće duljine pomoću prije izračunatih *prefix hasheva*.

Funkcije koje računaju neparni odnosno parni palindrom napisane su zasebno no sam njihov sadržaj je gotovo identičan. Tražimo najveći *lo* takav da se palindrom sastoji od dvije polovice te duljine (i u neparnom slučaju jednog znaka između njih). Funkcija *h* vraća normalni *hash* određenog podstringa dok funkcija *hr* vraća *hash* istog tog podstringa ali u obrnutom poretku znakova.

LEKSIKOGRAFSKI MINIMALNA ROTACIJA

Zadatak možete pronaći na adresi: http://www.spoj.pl/problems/MINMOVE/.

Ovaj primjer navodim čisto kao ilustraciju korištenja LCP funkcije, kao i komparatora koji se pomoću nje izvodi. Ono što je zgodno iskoristiti jest činjenica da se sve rotacije nekog stringa mogu pronaći kao podstringovi stringa dobivenog njegovom konkatenacijom sa samim sobom. Evo i koda:

```
#include <cstdio>
#include <cstring>

const int p = 10007;

#define MAX 200005

int N;
char S[ MAX ];

int P[ MAX ];
int H[ MAX ];

int h( int i, int j ) {
  int ret = H[j];
  if( i ) ret -= H[i-1] * P[j-i+1];
  return ret;
}

int LCP( int x0, int y0, int x1, int y1 ) {
  if( S[x0] != S[x1] ) return 0;
  int lo = 1, hi = N, mid;

  while( lo < hi ) {
    mid = ( lo + hi + 1 ) / 2;
    if( h( x0, x0+mid-1 ) == h( x1, x1+mid-1 ) ) lo = mid;
    else hi = mid - 1;
  }

  return lo;
}

bool cmp( int x0, int y0, int x1, int y1 ) {
  int L = LCP( x0, y0, x1, y1 );
  if( L == x0 - y0 + 1 ) return 0;
  return ( S[x0+L] < S[x1+L] );
}

int main( void )
{
  scanf( "%s", S ); N = strlen( S );

  for( int i = 0; i < N; ++i ) // konkatenacija sa samim sobom
    S[i+N] = S[i];

  P[0] = 1;
  H[0] = S[0];

  for( int i = 1; i < 2 * N; ++i ) {
    P[i] = P[i-1] * p;
    H[i] = H[i-1] * p + S[i];
  }

  int best = 0;
  for( int i = 1; i < N; ++i )
    if( cmp( i, i+N-1, best, best+N-1 ) ) best = i;
```

```
  printf( "%d\n", best );
  return 0;
}
```

Kao što sam već rekao, radi se o najobičnijem korištenju *cmp* funkcije za brzo leksikografsko uspoređivanje podstringova. Još ću samo napomenuti kako se ta funkcija u ovom kodu malo razlikuje od već viđene budući da su u ovom slučaju svi stringovi koje razmatramo jednake duljine. Uz to ako vidimo string koji je jednak prethodnome njega ne uzimamo u razmatranje jer nas to ne dovodi do minimalnog broja rotacija koji se u zadatku traži.

Za kraj ću predložiti par težih (dosta težih) zadataka, samo za hrabre:

1. http://www.spoj.pl/problems/TREEISO/
2. http://www.spoj.pl/problems/HSEQ/
3. http://www.spoj.pl/problems/LSQF/
4. http://www.spoj.pl/problems/DISUBSTR/

Opet napominjem ovo nisu zadaci koje je potrebno riješiti, tu su ako se netko u njima želi okušati.

Evo i nekih hintova redom:

1. Postoji čvor (ili dva) u stablu koji se nazivaju centar i jedinstveni su svakom stablu. Obilazak stabla iz nekog od ta dva čvora može se zapisati u niz. (Taj niz opet treba biti pažljivo izgrađen budući da redoslijed čvorova u listi koju koristimo za obilazak nije nužno isti u oba stabla). Zbog slabijih primjera moguće je i da neke druge heuristike također prolaze.
2. Osnovni hint je *suffix array* odnosno niz sortiranih sufiksa stringa. Potrebno je promatrati LCP susjednih sufiksa u tom nizu. Uz malo promatranja lako je prepoznati problem najvećeg pravokutnika u histogramu (možete pogledati i zadatak http://www.spoj.pl/problems/HISTOGRA/).
3. Postoji način provjeravanja postoji li takav string duljine K za bilokoji K u $O(N/K)$. Taj algoritam promatra prvih K, drugih K itd. uzastopnih znakova i koristeći LCP i LCS (*longest common suffix* koji se računa gotovo isto kao LCP) traži postoji li opisani podstring upravo te duljine. Rješenje se dobije prolaskom po svim mogućim K i složenosti je $O(N \lg^2 N)$.
4. Za kraj još jedan *suffix array* zadatak. Opet je potrebno sortirati sufikse i promatrati uzastopne LCP-ove. Ideja je od ukupnog broja substringova oduzeti ponavljanja. Lako je vidjeti kako LCP dva susjedna sufiksa duljine L predstavlja točno L ponovljenih substringova.

Još jednom napominjem kako su ovo zadaci za jako nabrijane (ako takvih ima) i ne trebate se njima zamarati ako vas ne zanimaju. Normalni zadaci će doći u zadaći.