# BIT MASKE

autor: Matija Korpar

Od davnih dana u naravi ljudi je olakšavati si život na sve moguće načine. Koristili smo se raznim metodama poput kotača, vatre, priprema za demose iz digitalne i slično. Bilo kako bilo, iskustvo na dragom nam FER-u naučilo nas je da se ispod teških riječi poput Hammingov koder, binarno stablo, troškovi zbog nekakvoće, zapravo krije sasvim jednostavna procedura razrješavanja određenih problema. Tako je i s bit maskama! Sad kad smo oboružani optimizmom bacamo se na posao shvaćanja tih famoznih bit maski.

## Uvod u bit maske

Zamislimo sljedeću situaciju:

U nekoj matematičkoj gimnaziji zamolili su vas da uskočite na mjesto profesora informatike, koje ste vi naravno zdušno prihvatili. I kako ne poznajete djecu, te ste uz to i veoma slabi sa imenima, postaje vam prava muka na početku sata prozivati učenika po učenika i upisivati tko nedostaje, jer naravno redari nisu obavili svoj posao. Kako bi se riješili te muke bacate se na razmišljanje. Cilj vam je naći veoma brzu i preciznu metodu kojom bi saznali tko je u razredu, a tko nije. Pošto imate pred sobom razred vrsnih matematičara, znate da možete računati i na njihovu pomoć.

Prvo krenete od najjednostavnijeg rješenja. Svatko se potpiše na papirić te vi nakon sata uzmete papiriće i zapišete u imenik prisustvo. Ubrzo uviđate da su papirići svaki puta u drugom poretku, dok je imenik konzistentan, svakom đaku je pridružen broj koji se ne mijenja. Sada rješavate taj problem tako da tražite da se ti listići sortiraju. Kako je to mukotrpan posao odbacujemo rješenje s listićima. Naiđete na sasvim jednostavno rješenje. Imate popis s brojevima đaka i svatko stavi plus kraj svoga imena. Uvelike ste si olakšali posao, no s vremenom vam je i to postalo previše. Pošto ipak podučavate informatiku odlučili ste napraviti program koji će pamtiti prisustva. Iskrsnuo je naravno novi problem. Popis brojevima s plusevima je naravno za vas previše za obrađivati te bi bili najsretniji kada bi to bio jedan jedini broj na temelju kojeg će program obaviti sve ostalo. Nakon kraćega razmišljanja sine vam genijalna ideja, pluseve zamijenimo s jedinicama, prazna mjesta s nulama i kada ih poredamo imamo binaran broj koji sadrži ukupnu informaciju!  To je upravo binarno kodiran popis prisustva ili narodu poznatija bit maska popisa prisustva!

# Implementacija

## *Binarni operatori*

Postoje li u programskom jeziku c binarni brojevi? Ne :(. Što ćemo onda iskoristiti da implementiramo ovu problematiku? Prisjetimo se kako su pohranjeni cijeli brojevi u c-u. To je upravo ono što mi tražimo, kao binarni brojevi. A kako raditi s njima? Potrebna nam je informacija da li je na nekom određenom mjestu jedinica ili nula. Za cijele brojeve imamo +,-,*,/ i slične operatore. Samo to? Naravno da ne! Spektakularnim pojavljivanjem na scenu stupaju binarni operatori! Pa da ih predstavimo:

1. << ***shift left*** - binarni oblik cijelog broja pomakni za jedno mjesto ulijevo i popunimo nulama, tj. pomnožimo ga sa brojem 2
   - 7 (111) << 1 = 14 (1110)
2. >> ***shift right*** – binarni oblik cijelog broja pomakni za jedno mjesto desno i dopunimo ga nulama, tj. dijelimo broj sa 2
   - 7(111) >> 1, broj 7 pomaknemo za jedno mjesto u desno i dopunimo nulama = 3 ( 011 )
3. ***&*** - logičko i, nad svakim parom bitova operanada radi logičko i te rezultat sprema kao cijeli broj
   - 3(11) & 2(10) = 2(10)
   - 5(101) & 7(111) =5(101)
4. | - logičko ili, , nad svakim parom bitova operanada radi logičko ili te rezultat sprema kao cijeli broj
   - 3(11) | 2(10) = 3(11)
   - 5(101) & 2(10) =7(111)

## *Ispitivanje bitova*

Kako bismo mogli ispravno raditi s našom bit maskom moramo znati provjeriti da li nam na određenom mjestu stoji jedinica ili nula. Naša bit maska pohranjena je kao cijeli broj. Uzmimo da je naša maska broj 21.

21 = 10101, ili ako raspišemo po potencijama od broja 2, 21 = 2^4 + 2^2 + 2^0

Uočimo da usporedbom s odgovarajućom potencijom broja 2, vrlo lako možemo dobiti da li je na mjestu koje odgovara toj potenciji jedinica ili nije. Isto tako primijetimo da ćemo tu potenciju veoma lagano dobiti operatorom << nad brojem 1:

1 << 2 , odgovora množenju sa brojem 2 dva puta

Primijetimo općenito da vrijedi

1 << n = 2^n

Ovim zaključkom veoma lagano dolazimo do odgovarajuće potencije broja 2, ostaje nam još samo usporediti tu potenciju s bit maskom i zaključiti rezultate.

10101 & (1<<2) = 100
10101 & (1<<3) = 0

Uviđamo ukoliko je na mjestu kojeg provjeravamo jedinica rezultat usporedbe biti će različit od nule.

## *Gornja granica bit maske prikazane kao cijeli broj*

Ukoliko imamo n podataka, u ovom našem slučaju učenika, tada će najveći bit maska biti zapisana kao suma od 0 do n-1 od 2^n što je jednako 2^n – 1. To je bit maska u kojoj je na svakom mjestu jedinica.

## *Pseudokod*

Dakle imamo određeni niz podataka u kojem su pohranjena imena naših učenika i želimo ispisati sve kombinacije prisustva moguće.

```
podaci[n] // podaci koje kombiniramo

za i idi od 0 do (2^n) - 1
        za j idi od 0 do n - 1 // provjerimo svaki bit svake kombinacije
                ako na mjestu j broj i ima 1 ispisi podaci[i]
        novi red
gotovo
```

Ovaj pseudokod će upravo ispisati sve moguće različite kombinacije naših učenika!

## *C-kod*

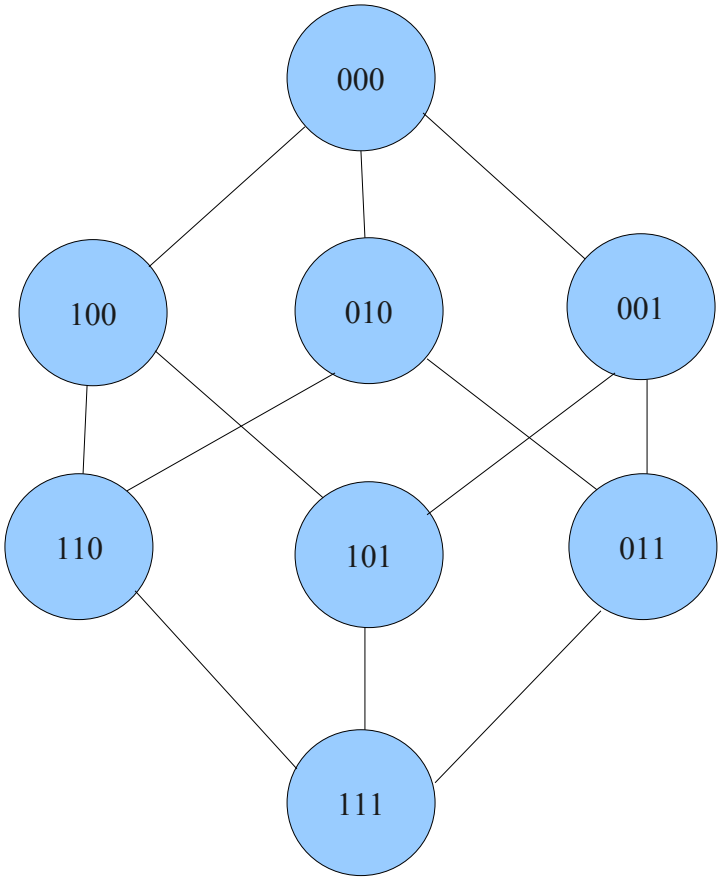
```
char podaci[n][200]; // podaci su naprimjer spremljeni kao niz nizova charova

for ( int i = 0;  i < Math.pow(2,n); i++ )
        for ( int j = 0;  j < n; j++)
                if ( (i & ( 1 << j )) != 0 ) printf("%s",podaci[i]);
        printf("\n");
```

To je to! Kod ispisuje sve moguće kombinacije naših učenika :)!

# Primjena na prvi laboratorij iz UTR-a

Generiranje različitih kombinacija podataka pomoću bit maski upravo nam razrješava jedan od kritičnih dijelova laboratorija. Gornjim ćemo kodom veoma lagano generirate sva stanja učitavanja podresursa. Ukoliko primijetimo još jednu činjenicu tada će nam i veze biti veoma jednostavno generirati.




Slika 1: primjer bit maskom kodiranih stanja za 3 podresursa

Gornja slika prikazuje što trebamo dobiti. Primijetimo da su povezana upravo stanja koja se razlikuju u jednoj jedinici! Sad znamo kako dobiti stanja i znamo između koji stanja dolaze prijelazi. Pošto nam je za definiciju automata dovoljno zapravo sam popis prijelaza primijetimo kako možemo iskoristiti gornji pseudokod:

```
za i idi od 0 do (2^n) - 1
        za j idi od 0 do n - 1
                ako na mjestu j broj i ima 0
                        ispisi prijelaz
                                iz (generiraj ime(podaci,i) )
                                za prijelaz podaci[2^j]
                                u generiraj ime (podaci,i + 2^j )
gotovo
```

i + 2^j – na j-to mjesto postavlja jedinicu

Funkcija ispisi prijelaz, po vašem nahođenju ispisuje prijelaz u datoteku.

Primijetimo li još da funkcija generiraj ime ima pseudokod s dva posebna slučaja:

```
generiraj ime ( podaci , bit maska )
        ako je bit maska = 0
                vrati Ucitavanje // ako je bit maska na svim mjestima nula to označava stanje učitavanja
        ako je bit maska = 2^podaci.duzina -1
                vrati Adresa // ako je bit maska na svim mjestima jedan to označava učitanu adresu
        za i idi od 0 do bit maska.duzina
                ako je na i-tom mjestu bit maske 1
                        ime += podaci[i]
        vrati ime
```

I ovim smo dobili veoma efikasno i kratko rješenje kritičnog dijela prvog laboratorija. :)