BHOI2016 Zadaci i rješenja

Adnan Mehanović, Elvir Crnčević, Rijad Muminović,
Jan Ahmetspahić, Ermin Hodžić

Juni 2016

1 Pijun

BHOI Game Studio — kompanija za razvoj video igara i igračkih konzola trenutno je jedan od najpoželjnijih poslodavaca u svijetu, a razlozi za to su mnogobrojni. Ljeto je veoma uzbudljiv period za sve one koji žele da im se pridruže, obzirom da BHOI Game Studio već tradicionalno samo tada zapošljava. Ove godine svi kandidati za posao moraju da učestvuju u razvijanju programskog modula za logičku igru baziranu na šahu. Obzirom da je riječ o velikom naslovu koji treba da ekspresno osvoji tržište, detalji o igri se oprezno čuvaju i plasiraju u javnost. S druge strane, sam modul koji kandidati trebaju razviti objašnjen je detaljno:

Data je tabla veličine $\boldsymbol{N}$, na kojoj se nalazi pijun. Početne koordinate polja na kojem se nalazi pijun označene su kao $\boldsymbol{X}$ i $\boldsymbol{Y}$. Polja table su numerirana brojevima od 1 do $\boldsymbol{N}$, a početno polje — polje s koordinatama $(1, 1)$ se nalazi u donjem lijevom uglu. Pijun može da se kreće gore, dolje, desno i lijevo pomjerajući se za tačno jedno polje. Osim toga, poznat je i broj kretnji koje će pijun napraviti, $\boldsymbol{P}$. Svaka od tih kretnji kodirana je slovom. Slovo u kodira kretnju prema gore, d prema dolje, r udesno, a l ulijevo. Tabla je cirkularna, što znači da će pijun, ukoliko se nalazi na desnom rubu table i ako napravi kretnju ka desno, završiti na lijevom rubu table. Također, ukoliko se pijun nalazi na gornjem rubu table i ako napravi kretnju ka gore, završit će na donjem rubu table. Vrijede i obrnuti slučajevi.

Vaš zadatak je da napišete funkciju/proceduru s prototipom:

```
void PremjestiPijuna( int N, int X, int Y, int P, char potezi[], int rjesenje[] ); [C++]
```

```
Procedure PremjestiPijuna( N : LongInt; X : LongInt; Y : LontInt; P :
LongInt; potezi : Array of Char; rjesenje : Array of LongInt ); [Pascal]
```

Parametri $\boldsymbol{N}$, $\boldsymbol{X}$, $\boldsymbol{Y}$ i $\boldsymbol{P}$ su prethodno opisani. Prametar ***potezi*** je niz karaktera veličine $\boldsymbol{P}$ kojim je kodirano kretanje pijuna i sadrži samo znakove u, d, r i l. Potrebno je odrediti konačnu poziciju pijuna — koordinate x i y po završetku svih kretanja. Koordinatu x smjestite u varijablu $rjesenje[0]$, a koordinatu y u varijablu $rjesenje[1]$. Ne zaboravite da tačnim rješavanjem ovog zadatka možete ostvariti svoj san i postati kandidat za zaposlenje u BHOI Game Studio.

Primjeri

```
potezi[] = {'d', 'r', 'd', 'r', 'd', 'r', 'd', 'r', 'u', 'l'};
```

Poziv funkcije	Stanje nakon poziva
PremjestiPijuna(3, 1, 1, 10, potezi, rjesenje)	rjesenje[0] = 1 rjesenje[1] = 1

Ograničenja na resurse

$1 \leq \boldsymbol{N} \leq 10000$
$1 \leq \boldsymbol{X}, \boldsymbol{Y} \leq \boldsymbol{N}$
$1 \leq \boldsymbol{P} \leq 10000$

Vremenska i memorijska ograničenja su dostupna na sistemu za ocjenjivanje.

Rješenje

Tip rješenja: ad hoc

Rješenje zadatka se zasniva na jednostavnoj modularnoj aritmetici. Neka tabla ima dimenzije $n \times n$. Prije svega primjetimo da koordinata x zavisi samo od poteza lijevo (-1) i desno $(+1)$, i da koordinata y zavisi samo od poteza gore $(+1)$ i dolje (-1). Dakle, nakon serije poteza koja se sastoji od ukupno br_l poteza lijevo i br_r poteza desno, pijun će se pomaknuti br_d puta desno i br_l puta lijevo. Tj. pomaknut će se $br_r - br_l$

mjesta desno (ako je ovaj broj negativan, onda se pomjera lijevo). Isto tako, ako je broj poteza gore br_u a broj poteza dolje br_d, onda će se pomaknuti $br_u - br_d$ mjesta gore.

Svaki put kada pijun pređe rub tabele, vraća se na mjesto na kojem je ta koordinata 1 – mjesto $n+1$ postaje mjesto 1. Ukoliko je pijun na osnovu formule u prethodnom paragrafu završio na mjestu $br_r - br_l$ koje je veće od n onda je rub prešao ukupno $\lfloor \frac{br_r - br_l}{n} \rfloor$ puta i napravio još $(br_r - br_l) - n \cdot \lfloor \frac{br_r - br_l}{n} \rfloor$ koraka desno – što je ostatak pri dijeljenju $br_r - br_l$ sa n.

Vremenska složenost algoritma je $O(P)$ pošto moramo pročitati sve poteze. Zbog toga, rješenje se može uraditi i simulacijom poteza (promjenom koordinata nakon svakog poteza) u istoj složenosti. U oba slučaja, potrebno je samo $O(1)$ radne memorije.

2 KDobar

Takmičarska komisija je odlučila da za ovaj zadatak neće izmišljati neku anegdotu koja uključuje članove takmičarske komisije (mada bi najvjerovatniji izbor za glavnog junaka spomenute anegdote bio Jan), te je odlučila da pređe na formulaciju zadatka već u drugom paragrafu.

String se zove KDobar ako i samo ako sadrži ne više od (tj. manje ili jednako) $\boldsymbol{k}$ različitih znakova (mala slova engleskog alfabeta). Podstring nekog stringa je niz uzastopnih karaktera tog stringa. Vaš zadatak je da za neki dati string $\boldsymbol{s}$ pronađete dužinu najdužeg KDobrog podstringa. Dakle, potrebno je implementirati funkciju sa sljedećim prototipom:

```
int kdobar( string s, int k ); [C++]
```

```
Function kdobar ( s : Array of Char; k : LongInt ) : LongInt; [Pascal]
```

Funkcija *kdobar* vraća dužinu najdužeg KDobrog podstringa stringa $\boldsymbol{s}$.

Primjeri

Poziv funkcije	Stanje nakon poziva
kdobar(”abcdef”, 3)	3
kdobar(”aaaaa”, 5)	5
kdobar(”aaaaaaaabc”, 3)	10

Napomena: U trećem primjeru, moguće je uzeti kompletan string obzirom na to da sadrži samo 3 različita karaktera.

Ograničenja na resurse i opis podzadataka

Zadatak će biti testiran na tri podzadatka, od kojih svaki nosi određeni broj bodova i ima sljedeća ograničenja:

Podzadatak 1 (25 bodova): $N \leq 250, 1 \leq \boldsymbol{k} \leq 26$
Podzadatak 2 (25 bodova): $N \leq 5050, 1 \leq \boldsymbol{k} \leq 26$
Podzadatak 3 (50 bodova): $N \leq 300000, 1 \leq \boldsymbol{k} \leq 26$

N predstavlja dužinu stringa $\boldsymbol{s}$.

Vremenska i memorijska ograničenja su dostupna na sistemu za ocjenjivanje.

Rješenje

Tip rješenja: greedy

Da bi neki podstring $s[i..j]$ mogao biti rješenje, potrebno je prije svega da elementi $s_i, s_{i+1}, ..., s_j$ ne sadrže više od k različitih znakova, tj. da je KDobar. Trivijalno je provjeriti svih $O(|s|^2)$ podstringova i izabrati najduži od njih koji je KDobar. Zvanično rješenje koristi sljedeći *greedy* algoritam.

Posmatrajmo sve podstringove koji završavaju na indeksu j ($s[1..j], s[2..j], ..., s[j-1..j], s[j..j]$). Sa obzirom da je svaki posmatrani uzastopni podstring sufiks prethodnog, vrijedi sljedeće: podstring $s[i..j]$ je KDobar ako i samo ako je svaki podstring $s[m..j]$ KDobar ($i \leq m \leq j$). Pretpostavimo da ovo nije tačno i da neki podstring $s[m..j]$ nije KDobar ($i \leq m$). To znači da sadrži više od k različitih znakova. Pomjerajući se lijevo od m prema i, nailazimo na dodatne znakove koji mogu povećati broj različitih znakova, ali nikako ga smanjiti. Dakle, kada dođemo do mjesta i, i dalje ćemo imati više od k različitih znakova pa $s[i..j]$ ne može biti KDobar.

Gore napisano sugeriše sljedeći *greedy* algoritam:

1. Krećući od početka stringa s, postavimo lijevu granicu na indeks 1 i "šetamo" desnu granicu od 1 do $|s|$.
2. Kada pomjerimo desnu granicu, ubrojimo znak na novoj poziciji. Ukoliko taj znak povećava broj različitih znakova viđenih od lijeve do desne granice iznad k, onda pomjeramo lijevu granicu prema desnoj sve dok god je broj viđenih znakova i dalje iznad k.
3. Kada petlja u prethodnom koraku stane, lijeva i desna granica sada ponovo definišu KDobar podstring. Sada možemo uporediti dužinu ovog KDobrog podstringa sa ranije najdužom i nastaviti dalje sa pomjeranjem desne granice.

Na osnovu ranije dokazanog, znamo da u koraku 3 algoritma nijedan podstring koji počinje lijevo od pozicije lijeve granice a završava tamo gdje je desna granica ne može biti KDobar. Time ne propuštamo nijedan validan KDobar string.

Ostaje još samo da pokažemo kako efikasno vršiti brojanje različitih znakova u podstringu. Za tu svrhu možemo koristiti cjelobrojni niz gdje za svaki različit znak alfabeta držimo koliko puta je viđen u trenutnim granicama. Kada pomjeramo desnu granicu, povećamo element niza koji odgovara novom znaku za jedan – ukoliko je taj element prethodno bio nula, znači da smo dodali novi znak koji ranije nije bio viđen pa povećamo brojač jedinstvenih znakova za jedan. Kada pomjeramo lijevu granicu, smanjujemo element niza koji odgovara tom znaku za jedan – ukoliko je nakon smanjenja taj element nula, znači da smo potpuno eliminisali taj znak u trenutnim granicama, pa smanjimo brojač jedinstvenih znakova za jedan.

Za držanje broja pojavljivanja svakog od znakova nam treba $O(|alfabet|)$ cjelobrojnih varijabli, što je u ovom slučaju 26. U svakom koraku algoritma povećamo desnu granicu za 1, i možda pomjeramo lijevu granicu desno. Desna granica ukupno učini $|s|$ koraka (gdje je $|s|$ dužina stringa s), a lijevu nikada ne možemo pomjeriti više od desne, pa i lijeva granica na kraju ukupno napravi najviše $|s|$ koraka. Na kraju, ukupna vremenska složenost je $O(|s|)$, a prostorna $O(1)$.

3 Manje, veće, nula

Kao da nije bilo dosta pogađanja brojeva, opet ste dobili taj zadatak. Međutim ovaj put je autor bio dosta strožiji i nemate ograničenja samo na broj pokušaja nego i na način na koji možete pogriješiti.

Tačnije, *grader* će zamisliti neki broj u intervalu 1 do $\boldsymbol{N}$, a vi ćete imati priliku da preko funkcije pogodi probate pogoditi broj koji je grader zamislio. Funkcija pogodi vraća 0 ukoliko ste pogodili, -1 ukoliko je zamišljeni broj manji od vašeg (tj. vaš broj je prevelik), i 1 ako je zamišljeni broj veći od vašeg.

Upravo na ove rezultate imate ograničenja, jer uspješnim pogađanjem se smatra samo pogađanje koje pogriješi maksimalno $\boldsymbol{L}$ puta brojem većim od zamišljenog (greške tipa L), tj. kada funkcija vrati -1, maksimalno $\boldsymbol{H}$ puta brojem manjim od zamišljenog (greške tipa H), tj. kada funkcija vrati 1 i tačno jednom da pogodi broj, tj. kada funkcija vrati 0.

Npr. Ako je $\boldsymbol{L} = 5$, a $\boldsymbol{H} = 0$, onda pet puta možete pogađati tako da vam je broj veći od zamišljenog, dok niti jednom ne smijete pogriješiti tako da vaš broj bude manji od zamišljenog. Za $\boldsymbol{N} = 5$, a zamišljeni broj $\boldsymbol{X} = 3$, smatrat će se pogrešnim ako npr. napravite poziv pogodi(4), jer radite grešku tipa H, a nemate te rezerve, dok će se sekvenca pogodi(1), pogodi(2), pogodi(3) završiti uspješno, jer ste pogodili broj 3 i iskoristili samo 2 greške tipa L.

Poštujući ova ograničenja, jasno je da za određeno $\boldsymbol{H}$ i $\boldsymbol{L}$, ne možemo garantovati da ćemo sigurno pogoditi bilo koji broj manji ili jednak $\boldsymbol{N}$, ako $\boldsymbol{N}$ postane dovoljno veliko za te brojeve $\boldsymbol{L}$ i $\boldsymbol{H}$. Zbog toga je u svakom testu garantovano da se, poštujući optimalan način pogađanja, može pogoditi svaki broj između 1 i $\boldsymbol{N}$ (uključujući 1 i $\boldsymbol{N}$).

Vaš zadatak je da izvedete pravilno pogađanje broja u datim ograničenjima. Broj poziva funkcije se ne broji i ne utiče na bodove, međutim itekako se broji broj specifičnih pogrešaka.

Prototip funkcije pogodi, koju smijete pozivati, je:

```
int pogodi( int x ); [C++]
```

```
Function pogodi( x : LongInt ) : LongInt; [Pascal]
```

Gdje je parametar $\boldsymbol{x}$ broj za koji želite da dobijete odgovor.

Na početku će biti pozvana funkcija/procedura *init*:

```
void init( int N, int L, int H ); [C++]
```

```
Procedure init( N : LongInt; L : LongInt; H : LongInt ); [Pascal]
```

Gdje su parametri $\boldsymbol{N}$, $\boldsymbol{L}$ i $\boldsymbol{H}$ prethodno opisani.

Primjeri

Poziv funkcije	Stanje nakon poziva
init(5,1,1)	-
pogodi(3)	1
pogodi(5)	-1
pogodi(4)	0

Poziv funkcije	Stanje nakon poziva
init(17,2,2)	-
pogodi(8)	-1
pogodi(5)	-1
pogodi(1)	0

Samo primjer pogađanja, ne mora značiti da bi ista procedura uspjela za svaki broj manji ili jednak od $\boldsymbol{N}$.

Ograničenja na resurse i opis podzadataka

Zadatak će biti testiran na četiri podzadatka, od kojih svaki nosi određeni broj bodova i ima sljedeća ograničenja:

Podzadatak 1 (5 bodova): $N \leq 1000, \quad H = L = \lfloor \frac{N+1}{2} \rfloor$
Podzadatak 2 (21 bodova): $N \leq 2^{30} - 1, \quad H = L = \lfloor \log_2 N + 2 \rfloor$
Podzadatak 3 (33 bodova): $N \leq 10000, \quad H, L \leq 20$
Podzadatak 4 (41 bodova): $N \leq 2^{30} - 1, \quad H, L \leq 20$

U svim testovima je garantovano da postoji optimalan način pogađanja bilo kojeg broja. Vremenska i memorijska ograničenja su dostupna na sistemu za ocjenjivanje.

Rješenje

Tip rješenja: dinamičko programiranje, binarna pretraga

Općenito rješenje problema je da naš prostor pretrage (interval u kojem je moguće da se nalazi zamišljeni broj), koji je na početku interval svih brojeva od 1 do N, postepeno svedemo na tačno jedan broj koji će predstavljati zamišljeni broj X i primjetimo da to optimalnim algoritmom moramo postići za bilo koji validan ulaz. U svakom koraku razmatramo interval brojeva u kojem smo sigurni da se nalazi zamišljeni broj – donja granica tog intervala je za jedan veća od najvećeg broja za koji smo dobili informaciju da je manji od X, a gornja je za jedan manji broj od najmanjeg broja za koji smo saznali da je veći od X.

Nazovimo granice ovog intervala redom a i b, tj. posmatramo $[a, b]$ i ovaj interval je na početku $[1, N]$. Jasno je da van ovog intervala sigurno ne može biti traženo X, te sva pitanja postavljamo (baš unutar tog intervala) u cilju da smanjimo njegovu veličinu na jedan, tj. da nam ostane samo jedan broj koji bi mogao predstavljati X, što će on upravo i biti. Naravno, moguće je i ranije završiti ali na to ne smijemo računati, te treba uvijek posmatrati najgori slučaj i za njega moramo dizajnirati algoritam.

Ukoliko pogledamo ograničenja podzadataka, jasno je da prvi podzadatak možemo riješiti običnom linearnom pretragom, dok drugi možemo uraditi preko binarne pretrage jer su L i H takvi da nam to omogućavaju. Ako problem svedemo na interval $[a, b]$, onda ćemo u binarnoj pretrazi uvijek pitati za broj $\lfloor \frac{a+b}{2} \rfloor$, te na osnovu toga uvijek prepoloviti interval. Pošto binarna pretraga uradi $\lfloor logN \rfloor$ pitanja prije nego što svede interval na jedan broj, to su nam granice u drugom podzadatku dovoljne, međutim to nije slučaj za sljedeći podzadatak.

Za sljedeći podzadatak, razne heuristike koristeći informacije o H i L su mogle uraditi većinu testova, a neke bolje uspijevaju na čitavom podzadatku. Jedna takva je paziti da li je preostalo više od 0 mogućih promašaja a interval dijelimo umjesto na sredinu ($1 : 1$ intevrale), na $L : H$ intervale, slično kao i binarnom pretragom.

Međutim za sve bodove se morao koristiti skroz drugačiji pristup. Naime, obratimo pažnju na paragraf koji kaže da će u testovima uvijek biti dato N, L i H takvi da postoji optimalno rješenje koje pronađe bilo koji zamišljeni broj $X (1 \leq X \leq N)$. To zapravo znači da uvijek možemo svesti čitav interval na samo jedan broj, ili i prije zaključiti, i pri tome biti sigurni da ćemo ispoštovati ograničenja L, H. Pokušajmo za par L i H da pronađemo najveće N tako da je ovo svojstvo zadovoljeno i taj broj ćemo označiti sa $N_{max}[L][H]$.

Za $L = 0$ imamo $N_{max}[0][H] = H + 1$, a za $H = 0$ je $N_{max}[L][0] = L + 1$, jer u tim situacijama možemo samo linearnom pretragom biti sigurni da neće doći do greške. Ovo smatrajmo početnim uslovima. Tako je za $L = 0, H = 3$ zapravo $N_{max}[0][3] = 4$. Drugi interesantan slučaj je $L = 1, H = 1$, probavajući par primjera zaključujemo da je $N_{max}[1][1] = 5$ i optimalna strategija je prvo pitati za srednji član, te nastaviti linearnim pretragama po dijelu u kojem znamo da se zamišljeni broj nalazi.

Sada ako imamo interval $[a, b]$, dužine $b - a + 1 \leq N_{max}[L][H]$ i ograničenja L i H, i na pitanje za neki broj U dobijemo odgovor npr. da je zamišljeni broj X manji od U, onda nikako ne smijemo dopustiti da veličina intervala brojeva manjih od U, tj. $U - a$ bude veća od maksimalnog N za koje možemo garantovati pronalazak rješenja, tj. moramo zadovoljiti $U - a \leq N_{max}[L-1][H]$. Slično i za gornji interval (ako dobijemo odgovor da je X veći od U), dužina intervala brojeva većih od U mora biti manja ili jednaka maksimalno dozvoljenoj, tj. $b - U \leq N_{max}[L][H-1]$.

Ovom analizom smo zapravo dobili rekurzivnu formulu kojom možemo generisati svako $N_{max}[L][H]$, a ona je $N_{max}[L][H] = N_{max}[L-1][H] + 1 + N_{max}[L][H-1]$, s tim da za $L = 0$ ili $H = 0$ uzimamo početne uslove gore definisane. Sada $N_{max}[L][H]$ za $L, H <= 20$ možemo izračunati u tabelu, pomoću dinamičkog programiranja, tj. memoizacije unutar rekurzije koja realizuje ovu formulu. Naravno postoji i "bottom-up" rješenje (korištenjem samo petlji bez rekurzije) koje nije bitno komplikovanije od rekurzivnog, jer se formula primjenjuje isto.

Nakon izračunatih N_{max} za sve kombinacije L i H, sada tražimo rješenje koristeći zaključak iz gornjeg paragrafa, tj. tako da nezavisno od odgovora uvijek rezultiramo intervalom sa zadovoljavajućom veličinom za L i H koje se može riješiti optimalno, te možemo za interval od $[a, b]$ uvijek odabrati upit $min(b, a + N_{max}[L-1][H])$, jer tako smo osigurali da u slučaju odgovora L opet imamo optimalnu strategiju za rješavanje intervala $[a, a + N_{max}[L-1][H] - 1]$, jer je on upravo maksimalne veličine za ograničenja, a u slučaju odgovora H dobijamo interval $[a + N_{max}[L-1][H] + 1, b]$, koji je veličine koja je sigurno rješiva unutar ograničenja tj.:

$b - (a + N_{max}[L-1][H] + 1) + 1 = b - a + 1 - N_{max}[L-1][H] - 1 \leq N_{max}[L][H] - N_{max}[L-1][H] - 1 = N_{max}[L-1][H] + N_{max}[L][H-1] + 1 - N_{max}[L-1][H] - 1 = N_{max}[L][H-1]$

Sama implementacija se može izvršiti rekurzivno tako što pratimo trenutne granice niza i preostale L i H i primjenjujemo logiku kretanja koja je opisana iznad. Naravno isti efekat se postiže while petljom, u kojoj se dešavaju iteracije smanjivanja intervala i ona dosta liči na klasičnu binarnu pretragu, ali nam je izbor broja za upit određen gore opisanim ograničenjima.

Jedina dodatna stvar koja se mora paziti je veličina ograničenja u prvim podzadacima, a to možemo riješiti svodeći $H = L = 20$ ili posmatrajući te velike brojeve kao poseban slučaj, jednostavno ne provjeravajući granice ili uvijek pogađanjem u sredinu intervala dok ograničenja dopuštaju, te linearnom pretragom ako moramo. Ovo napominjemo jer je $N_{max}[20][20] = 538257874439$, te sam broj ne bi stao u 32-bitni int, u čijem opsegu je sigurno naš N.

4 Paketi

Ćatil, koji sada radi kao poštar, dobio je naređenje od šefa da pokupi K specijalnih paketa koje prevoze kamioni cestama regiona. Svaki tip paketa prevozi tačno jedan kamion, dakle ukupno K kamiona prevoze specijalne pakete. Region se sastoji od gradova i dvosmjernih ulica između gradova. Za sve dvosmjerne ulice imate informaciju koliko vremena treba da se ona pređe u bilo kom od dva smjera (bezdimenzionalno vrijeme). Garantirano je da postoji najviše jedna dvosmjerna ulica između bilo koja dva grada regiona, ali gradovi mogu biti nepovezani.

Ćatil ima auto (koje nije ništa brže ni sporije od kamiona) i u početnom trenutku ($t = 0$) se nalazi u nekom gradu regiona. On sada treba da se kreće kroz region cestama i presretne kamione i pokupi njihove pakete. Ćatil se može proizvoljno dugo zadržavati u gradovima (čak i u početnom gradu) i može da se kreće između njih na potpuno proizvoljan način (da ponavlja iste gradove, da se vrti u krugove, čak da stoji sve vrijeme u početnom gradu i sl.). Da bi ispunio svoju obavezu on treba da pokupi svaki od K paketa. Pri tom za ispunjavanje obaveze nije bitno u kojem gradu je završio nakon što je pokupio svih K paketa (jer će šefu lično uručiti pakete u gradu u kojem završi putovanje). Red kojim Ćatil kupi pakete također nije bitan.

Svaki kamion se kreće određenom putanjom koja je data kao niz gradova, takav da prvi grad predstavlja grad u kom se kamion nalazi u početnom trenutku ($t = 0$), a svaki sljedeći predstavlja sljedeći grad na putanji. Iz grada u ovom nizu do njegovog sljedbenika u nizu kamion se obavezno kreće cestom između njih, za koju je garantovano da postoji. Nakon što kamion završi svoje putovanje, njegov paket se više nigdje ne može pokupiti. Svaki od gradova regiona na ruti jednog od kamiona se može pojaviti najviše jednom.

Ćatil uspjeva presretnuti kamion ako i samo ako se nađe u isto vrijeme u istom gradu kao i taj kamion. Njemu treba nula jedinica vremena da pokupi paket iz tog kamiona. Ukoliko se više kamiona i Ćatil nađu u istom trenutku u nekom gradu, on može pokupiti paket iz svakog od tih kamiona (ponovo u nula jedinica vremena).

Također, šefu je veoma bitno da ovo bude gotovo što prije, pa je Ćatilu naređeno da uradi posao u najmanjem mogućem vremenu. Nažalost, on nije dobar programer, pa je posao optimizacije ostavljen vama. Moguće je da se svi paketi uopće ne mogu pokupiti. U tom slučaju je Ćatilu veoma bitno da o tome obavijesti šefa prijevremeno.

Vaš zadatak je da implementirate funkciju sa prototipom:

```
int NajmanjeVrijeme( int N, int M, int* u, int* v, int* w, int K, int* d, int** p, int s );
    [C++]
```

```
Function NajmanjeVrijeme( N : LongInt; M : LongInt; u : Array of LongInt; v : Array of
    LongInt; w : Array of LongInt; K : LongInt; d : Array of LongInt; p : Array of Array of
    LongInt; s : LongInt ) : LongInt; [Pascal]
```

N je broj gradova u regionu. Gradovi su numerisani $[0..N-1]$.

M je broj cesta u regionu. Ceste su numerisane $[0..M-1]$.

u, v i w su nizovi brojeva dužine M degenerirani u pokazivače i imaju sljedeće značenje: cesta broj i je između gradova u_i i v_i i vrijeme potrebno da se ta cesta pređe u bilo kom pravcu je w_i. Garantovano je $u_i < v_i$.

K je broj paketa / kamiona. Oba su numerisani $[0..K-1]$.

d je niz dužine K degeneriran u pokazivač koji označava broj gradova na ruti svakog od K kamiona. Rute su neprazne.

$\boldsymbol{p}$ je dinamički (fragmentalno) alocirana grbava matrica (matrica sa nejednakom dužinom redova) koja ima $\boldsymbol{K}$ redova. $\boldsymbol{i}$-ti red ima $\boldsymbol{d_i}$ elemenata koji predstavljaju, redom, gradove na ruti $\boldsymbol{i}$-tog kamiona.

$\boldsymbol{S}$ je broj iz $[0..\boldsymbol{N}-1]$ koji predstavlja indeks grada u kome Ćatil počinje putovanje.

Funkcija treba da vrati opisano najmanje vrijeme opisano u zadatku. Ukoliko je ono što se traži neizvodivo neovisno od količine vremena, funkcija treba da vrati -1.

Primjeri

Poziv funkcije	Vraćena vrijednost
4 4 0 1 2 0 2 1 2 3 3 1 2 4 1 4 3 2 1 0 0	3

Format prikaza je sljedeći: U prvom redu $\boldsymbol{N}$ i $\boldsymbol{M}$, pa $\boldsymbol{M}$ redova koji sadrže $\boldsymbol{u_i}$, $\boldsymbol{v_i}$ i $\boldsymbol{w_i}$, pa u novom redu $\boldsymbol{K}$, pa $\boldsymbol{K}$ redova koji sadrže $\boldsymbol{d_i}$, popraćen sa $\boldsymbol{d_i}$ razmakom odvojenih brojeva $\boldsymbol{p_{i,j}}$ i u zadnjem redu se nalazi $\boldsymbol{s}$.

Objašnjenje: Ćatilu se najviše isplati premjestiti iz grada 0 u grad 2 i sačekati kamion u tom gradu ili raditi bilo šta slično što će ga u trenutku t = 3 dovesti u grad 2.

Ograničenja na resurse i opis podzadataka

Zadatak će biti testiran na tri podzadatka, od kojih svaki nosi određeni broj bodova i ima sljedeća ograničenja:

$1 \leq N \leq 100$
$1 \leq w_i \leq 50000$
$1 \leq K \leq 10$
$d_i > 0$

Podzadatak 1 (30 bodova): $1 \leq N < 10, 1 \leq w_i \leq 11$
Podzadatak 2 (20 bodova): $K = 1$
Podzadatak 3 (50 bodova): Nema dodatnih ograničenja

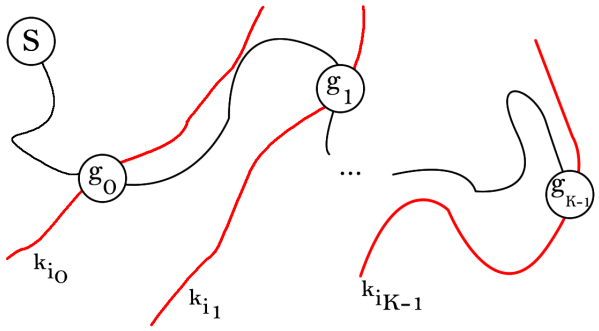
Vremenska i memorijska ograničenja su dostupna na sistemu za ocjenjivanje.

Rješenje

Tip rješenja: najkraći putevi u grafu, greedy

Prije svega, primjetimo da ako želimo pokupiti paket nekog kamiona K_i $(0 \leq i \leq K-1)$ u gradu p_{ij}, onda moramo u taj grad stići najkasnije u vremenu $\sum_{l=1}^{j} w(p_{i,l-1}, p_{i,l})$, gdje je $w(p_{i,l-1}, p_{i,l})$ vrijeme potrebno da se pređe put od jednog grada do drugog. Pa prije svega ostalog, izračunajmo to najkasnije vrijeme za svaki grad.

Posmatrajmo neko rješenje koje opisuje putanju Ćatila. On počinje u gradu S, i onda se pomjera nekim određenim putem do nekog grada u kojem pokupi paket kamiona k_{i_0}, sa mogućim čekanjem na kamion u tom gradu. Tada kreće dalje nekom putanjom do sljedećeg nekog grada u kojem onda čeka na kamion k_{i_1} itd., sve dok ne pokupi paket zadnjeg kamiona $k_{i_{K-1}}$ u nekom gradu. Rješenje se sastoji od niza bitnih



Slika 1: **Skica rješenja. Crvenom bojom su skicirane putanje svakog kamiona.** Ćatil krene iz S i putuje prema nekom gradu g_0 gdje pokupi paket iz nekog kamiona k_{i_0}. Dalje nastavlja prema nekom gradu g_1 gdje pokupi paket nekog kamiona k_{i_1} itd. sve dok ne stigne do nekog grada u kojem pokupi paket posljednjeg kamiona $k_{i_{K-1}}$

gradova u kojim se kupe paketi kamiona k_{i_j} (gdje su i_j indeksi iz niza $\{0, 1, 2, ..., K-1\}$ u nekom određenom redoslijedu) u određenom optimalnom redoslijedu, i najkraćih puteva između njih.

Iz ovoga se može odmah napraviti jednostavan algoritam složenosti $O(N^3 + K! \cdot K \cdot N^2)$ u kojem isprobamo svaki mogući redoslijed kupljenja paketa iz kamiona (za K različitih kamiona), i za svaki redoslijed onda dinamičkim programiranjem odlučimo u kojem gradu pokupimo paket svakog kamiona (svaki kamion može imati $O(N)$ gradova na svojoj putanji). Prije dinamičkog programiranja, izračunamo najkraće puteve između svaka dva grada u vremenu $O(N^3)$ metodom Floyd–Warshall. Tada je optimalno vrijeme u kojem možemo pokupiti paket kamiona k_{i_j} u bilo kojem gradu na njegovoj putanji dato kao minimum od optimalnih vremena potrebnih da pokupimo paket kamiona $k_{i_{j-1}}$ za svaki grad na putanji tog kamiona, plus vrijeme potrebno da se pređe najkraći put od tog grada do trenutnog, i na kraju još tom broju dodamo vrijeme čekanja na kamion u trenutnom gradu (ne možemo pokupiti paket prije nego kamion stigne). Ukoliko je neko od ovih vremena iznad dozvoljene granice, taj grad više ne razmatramo kao dio daljeg rješenja.

Ovaj metod se može ubrzati ako primjetimo da nam dinamičko programiranje ne treba jer radi *greedy* pristup: prvi grad na putanji kamiona k_{i_j} u koji možemo stići (poštujući trenutni redoslijed) najkasnije kada i kamion k_{i_j}, nazovimo ga g_p, je dio nekog optimalnog rješenja za trenutni redoslijed. Ovo vrijedi zato što kamion ide kroz gradove na svojoj putanji u određenom redoslijedu koji je dat kao ulaz. Da bi stigao u neki kasniji grad g_k, mora proći kroz sve gradove između g_p i g_k. Pa iako možda možemo u g_k doći ranije nego u g_p, to nam je uzalud jer moramo čekati da kamion stigne da bismo preuzeli paket. Sa obzirom da kamion ne može stići u g_k prije nego u g_p, to znači da g_k ne može biti grad u kojem u najkraćem roku pokupimo paket kamiona k_{i_j}. Da bismo se uvjerili da optimalan put kojim pokupimo paket sljedećeg kamiona može proći kroz g_p, dovoljno je primjetiti da ako bismo pokušali zaobići taj grad opet ne bi mogli ni iz jednog kasnijeg grada na putanji trenutnog kamiona krenuti dalje ka sljedećem paketu prije nego tu kamion stigne i pokupimo trenutni paket. A sa obzirom da su svi takvi gradovi na putanji iza g_p, onda možemo proći kroz g_p i ići dalje ka kasnijim gradovima ukoliko je to potrebno. Dakle, postoji optimalno rješenje kojim pokupimo sve pakete u najkraćem vremenu a koje se sastoji od prvih gradova na putanjama svakog od kamiona u koje možemo stići prije nego kamion ode, i najkraćih puteva između njih.

Naivno implementiran, ovaj *greedy* metod ima vrijeme izvršavanja $O(N^3 + K! \cdot K \cdot N)$, gdje za nalaženje prvog grada nekog kamiona u koji je moguće stići prije nego taj kamion ode iterativno prođemo kroz gradove, a svaki kamion može imati $O(N)$ gradova na svojoj putanji, i prvi grad koji je rješenje može biti uvijek jedan od zadnjih gradova na putanji svakog kamiona. Sa obzirom da je traženi grad prvi od gradova za koje vrijedi da u njih možemo stići prije nego kamion ode (jednostavno šetamo redom od prvog validnog grada do svakog iza, i uvijek ćemo biti ispred kamiona), možemo koristiti binarnu pretragu. Pitanje koje postavljamo u svakom koraku binarne pretrage je: "da li je moguće stići u trenutni grad iz najboljeg grada za prethodni kamion prije nego trenutni kamion ode iz ovog grada?" Ukoliko nije, odbacujemo prvu polovinu gradova koje razmatramo na putanji trenutnog kamiona; a ukoliko jeste, onda odbacujemo drugu polov-

inu. Time dobijamo algoritam složenosti $O(N^3 + K! \cdot K \cdot logN)$, pošto najkraće puteve između gradova i najkasnije dozvoljeno vrijeme dolaska u svaki grad izračunamo prije nego počnemo isprobavati sve redoslijede.

Iako je prethodno opisan metod dovoljno brz za dobijanje svih bodova na zadatku, nije potrebno prolaziti kroz sve moguće rasporede kupljenja paketa. Posmatrajmo drugačiji način razmišljanja o rješenju ovog zadatka. Intuitivno, optimalno rješenje završava u nekom gradu gdje pokupimo neki zadnji paket, a da bismo završili tu moramo prvo pokupiti sve ostale pakete nekim određenim putem i onda se pomjeriti u zadnji grad preko nekog susjeda. Postoji više različitih načina da završimo u trenutnom gradu. Prije svega moguće je da pokupimo bilo koji od K paketa kao zadnji. Dalje, moguće je da pokupimo ostale pakete i dođemo u trenutni grad iz bilo kojeg susjeda. Isto tako je moguće da pokupimo nekoliko prethodnih paketa u istom gradu čekajući da kamioni naiđu, tj. da prelaz bude iz istog grada sa manje pokupljenih paketa u isti grad sa više pokupljenih paketa. Ovakav način razmišljanja o načinima dolaska u posljednji grad se može dalje rekurzivno primjeniti na sve moguće opisane veze prema trenutnom gradu i onda možemo izabrati onaj put koji vodi ka najmanjem potrošenom vremenu. Nećemo koristiti rekurziju ili dinamičko programiranje, ali ovo nas vodi ka tačnom i brzom postupku rješavanja.

Pređimo sada na formalni opis algoritma. Svakom čvoru možemo dodijeliti bitmasku koja opisuje "koji su paketi pokupljeni na putu koji završava u ovom čvoru". Npr. neki čvor u sa bitmaskom "0010101101" opisuje neki put na kojem smo pokupili treći, peti, sedmi, osmi i deseti paket, a koji završava u čvoru u. Svaki čvor može imati dodijeljenih 2^K različitih bitmaski koje opisuju sva moguća stanja puta kojim idemo a koji je trenutno stao u tom čvoru. Kao što je ranije opisano, moguće je u čvor sa trenutnom bitmaskom doći iz istog čvora sa bitmaskom koja je identična trenutnoj osim na mjestu jedne jedinice gdje se nalazi nula. Pa tako naš graf mogućih stanja ima 2^K različitih čvorova za svaki čvor u prvobitnom grafu, i između tih čvorova se nalazi direktna grana ako je bitmaska jednog čvora nadogradnja sa bitmaske prethodnog tako što dodamo jednu jedinicu. Te grane opisuju čekanja u istom gradu na dolazak nekog kamiona čime preuzimamo jedan paket više. Druga vrsta grana koja se javljaju u ovakvom grafu su grane koje vode iz jednog čvora sa određenom bitmaskom u neki od njegovih susjeda sa identičnom bitmaskom. Ovakve grane opisuju pomjeranje iz jednog grada u drugi pri čemu se ne mijenja broj paketa. Osim ove dvije vrste grana, ne trebaju nam nikakve druge – kupljenje paketa u jednom gradu, pomjeranje u susjeda, i kupljenje dodatnog paketa u susjedu, odgovara kompoziciji "prva grana, druga grana, prva grana". Grane koje vode iz neke bitmaske u bitmasku sa manje jedinica se ne koriste jer nakon što pokupimo paket nema smisla "da ga odbacimo". Ono što na kraju imamo je graf stanja sa $N \cdot 2^K$ čvorova, u kojem krećemo iz čvora S sa bitmaskom "$\{0\}^K$" (nijedan pokupljen paket) a završavamo u nekom od čvorova u grafu sa bitmaskom "$\{1\}^K$" (svi pokupljeni paketi), pri tome prateći grane koje vode iz jedne bitmaske istog čvora u striktno veću ili grane koje vode iz jedne bitmaske jednog čvora u istu bitmasku drugog čvora. Težine grana prve vrste odgovaraju vremenu čekanja na kamion koji odgovara dodanom paketu (ove težine ne možemo znati prije nego saznamo put koji vodi do trenutnog čvora), a težine grana druge vrste su jednake težinama grana između čvorova u prvobitnom grafu. Ukoliko su neke grane nemoguće (npr. vode ka manjoj bitmaski ili je kamion sa datim paketom već davno prošao ili nikako ne dolazi u trenutni grad) ne uzimaju se u razmatranje. Tako se ovaj zadatak svodi na nalaženje najkraćeg puta od S sa praznom bitmaskom (sve nule) do svih čvorova sa punim bitmaskama (sve jedinice) i biranja onog čvora među njima prema kojem je put najkraći.

Za rješavanje ovog problema možemo koristiti Dijkstrin algoritam sa *min-heapom*, u kojem stavimo sve čvorove u heap i pri svakoj iteraciji biramo sljedeći grad koji je najbliži trenutno "otkrivenom" dijelu grafa (grad koji je na vrhu heapa). Nakon što izaberemo taj grad, provjerimo da li grane koje vode iz njega smanjuju vrijeme otkrivanja nekog od njegovih susjeda (ujedno izračunamo težine onih grana koje zavise od trenutnih pozicija kamiona) i po potrebi radimo update heapa. Vrijeme izvršavanja je $O(|E| \log |V|)$, gdje je $|V|$ broj čvorova a $|E|$ broj grana u grafu. Znamo da je $|V| = N \cdot 2^K$. Grane između susjeda se jednostavno kopiraju između čvorova za svaku bitmasku pa je ukupan broj takvih grana $O(N^2 \cdot 2^K)$. Broj grana između bitmaski jednog čvora je jednak broju grana višedimenzionalne kocke dimenzije K (koja ima 2^K čvorova sa vezama između dva čvora samo ako im se bitmaska razlikuje za jedan bit) a taj broj je $K \cdot 2^{K-1}$. Tako je ukupan broj grana $|E| = O(N^2 \cdot 2^K)$. Dijkstrin algoritam primjenjen nad ovakvim grafom ima vremensku složenost $O(N^2 \cdot 2^K \cdot \log(N \cdot 2^K))$. Prostorna složenost je $O(N \cdot 2^K)$.

5 BHOI Grader

Svima je poznato da BHOI robot za ocjenjivanje takmičarskih rješenja upoređuje izvorne kodove takmičara sa zvaničnim rješenjem i ocjenu daje na osnovu sličnosti datih kodova, kao najpravedniji način ocjenjivanja. Ono što je manje poznato je to da je jedan član BHOI komisije (koji ostaje anoniman zbog njegove sigurnosti) jutros slučajno polio robota vrelim čajem i time uzrokovao pad sistema. Obzirom da je BHOI komisija zauzeta štampanjem diploma i apsolutno niko nema vremena da napiše novi grader, nadamo se da de neko od vas uspjeti uraditi ovaj zadatak (i time implementirati novi grader), tako da rezultati mogu biti objavljeni danas.

Potrebno je implementirati algoritam koji radi nad skupom (moguće različitih) stringova iste dužine, a koji za svaki upit u vidu stringa $\boldsymbol{S}$ vraća broj stringova iz početnog skupa koji se od $\boldsymbol{S}$ razlikuju za najviše $\boldsymbol{K}$ znakova. Za dva stringa A i B kažemo da se razlikuju u najviše K znakova ukoliko je $A_i \neq B_i$ za najviše K različitih indeksa i (poznato kao Hammingova udaljenost). Da bismo bili uvjereni u tačnost vaše implementacije gradera, algoritam će morati odgovoriti na mnogo upita.

Vaš zadatak je da implementirate funkciju/proceduru sljedećeg prototipa:

```
void Slicni( char* A[], int N, int M, char* S[], int Q, int R[], int K ); [C++]
```

```
Procedure Slicni( A : Array of Array of Char; N : LongInt; M : LongInt; S : Array of Array
    of Char; Q : LongInt; R : Array of LongInt; K : LongInt ); [Pascal]
```

Svi stringovi (i početni i oni koji dolaze kao upiti) imaju tačno $\boldsymbol{M}$ znakova. Broj $\boldsymbol{N}$ označava broj početnih stringova, a $\boldsymbol{A}$ je niz tih stringova. $\boldsymbol{Q}$ predstavlja broj upita, a $\boldsymbol{S}$ je niz koji sadrži stringove koje je potrebno naći u $\boldsymbol{A}$ sa najviše $\boldsymbol{K}$ razlika. $\boldsymbol{R}$ je niz dužine $\boldsymbol{Q}$ u koji treba da zapišete rezultate svakog upita ($\boldsymbol{R}_i$ treba da sadrži izlaz za upit $\boldsymbol{Q}_i$). Svi znakovi u stringovima će biti standardni alfanumerički znakovi (velika i mala slova alfabeta i brojevi), i elementi niza $\boldsymbol{A}$ se neće razlikovati za više od 50 znakova.

Primjeri

Poziv funkcije	Stanje nakon poziva
Slicni({"23", "153", "224"}, 3, 3, {"456", "453", "124"}, 3, R, 0)	R={0, 0, 0}
Slicni({"23", "153", "224"}, 3, 3, {"456", "453", "124"}, 3, R, 1)	R={0, 1, 2}
Slicni({"23", "153", "224"}, 3, 3, {"456", "453", "124"}, 3, R, 2)	R={1, 2, 3}
Slicni({"23", "153", "224"}, 3, 3, {"456", "453", "124"}, 3, R, 3)	R={3, 3, 3}

Ograničenja na resurse

$\boldsymbol{N} \leq 5000$
$\boldsymbol{Q} \leq 5000$
$\boldsymbol{M} \leq 50$
$\boldsymbol{K} \leq 50$

Vremenska i memorijska ograničenja su dostupna na sistemu za ocjenjivanje.

Rješenje

Tip rješenja: brute-force, divide & conquer, bitmaske

Sa obzirom da je potrebno vršiti uporedbe stringova sa proizvoljnim mjestima K grešaka, nije dostupan pametniji način od poređenja S_i sa svakim A_j $(1 < j < N)$. $Brute - force$ način bi bio da svaki put uporedimo kompletne stringove i brojimo broj grešaka, što daje algoritam složenosti $O(NQM)$. Korištenjem bitmaski, ovo se može uraditi brže.

GTCCATCGAC	l_1
GTCGATCGGC	l_2
ATTCATCGAA	l_3
GTCCATCGAC	l_4
GTCCATCTAC	l_5
GTGGATCGAC	l_6

Ideja je sljedeća: Znajući da su stringovi skupa A slični, iskoristit ćemo tu sličnost tako da nakon jednog potpunog poređenja ubrzamo poređenja sa ostalim stringovima izbjegavajući poređenja nepotrebnih dijelova. Radi ilustracije, posmatrajmo tabelu ispod na kojoj su prikazana poređenja stringa l_1 sa stringovima $l_2, ..., l_6$ — ukoliko se na nekoj poziciji dva stringa razlikuju, stavimo na tu poziciju ”1”, u suprotnom stavimo ”0”. Kao rezultat dobijemo:

GTCCATCGAC	l_1
0001000010	b_2
1010000001	b_3
0000000000	b_4
0000000100	b_5
0011000000	b_6

Sada, uporedimo string S = ”GTSCATFGAC” sa ovom listom. Poređenjem sa l_1 dobijemo bitmasku b_S = "0010001000". Uporedimo ovu bitmasku sa npr. bitmaskom b_6. b_S ima bit ”1” na pozicijama 3 i 7 (što nam govori da se S razlikuje od l_1 na pozicijama 3 i 7), dok b_6 ima bit ”1” na pozicijama 3 i 4 (što nam govori da se l_6 i l_1 razlikuju na pozicijama 3 i 4).

Ono što je važno primjetiti je sljedeće:

1. Ukoliko se na nekoj poziciji l_1 i l_6 ne razlikuju, a S i l_1 se razlikuju, ta razlika između S i l_1 se automatski prenosi na l_6.
2. Ukoliko se na nekoj poziciji l_1 i l_6 razlikuju, a S i l_1 se ne razlikuju, ta sličnost između S i l_1 automatski postaje razlika na l_6.
3. Ukoliko se na nekoj poziciji l_1 i l_6 ne razlikuju, a ni S i l_1 se ne razlikuju, onda se ta sličnost između S i l_1 prenosi i na l_6.
4. Ukoliko se na nekoj poziciji l_1 i l_6 razlikuju, a i S i l_1 se razlikuju, onda ne znamo da li se S i l_6 na toj poziciji razlikuju bez da to provjerimo upoređujući znakove na tom mjestu.

Iz opisana četiri slučaja vidimo da nalaženjem mjesta između b_S i l_j na kojim je u jednoj bitmaski 0 a u drugoj 1 (XOR operacija), dobijamo sva mjesta sa sigurnim greškama između S i l_j. Na mjestima gdje se u obje bitmaske nalazi 1 (AND operacija), treba provjeriti da li se greška slučajno ”ispravila” ili je i dalje pogrešan znak na tom mjestu.

Da bismo odredili broj jedinica u rezultatu XOR-a (koji nam govori koliko ima sigurnih grešaka) i AND-a (koji nam govori broj grešaka koje je potrebno provjeriti), koristimo kombinaciju *divide & conquer* i *brute-force* pristupa. To radimo tako što izračunamo u linearnom vremenu rezultate za sve brojeve sa 16 bita (veoma jednostavnim dinamičkim programiranjem) a onda velike bitmaske isječemo na dijelove veličine 16 i saberemo rezultate.

Sa obzirom da je u rezultatu AND operacije potrebno provjeriti znakove na mjestima gdje se nalaze biti 1, treba nam način da efikasno odredimo ta mjesta. Primjetite da se negacijom bita nekog broja i sabiranjem sa 1 dobija novi broj koji ima sve bite različite osim na mjestu najmanjeg bita 1. Ako uradimo AND tog broja sa prvobitnim brojem, dobijemo broj koji ima sve bite 0 osim na mjestu gdje je prvobitni broj imao najmanji bit 1. Nalaženje indeksa tog bita je analogno nalaženju logaritma po bazi dva — što se također može riješiti kombinacijom *brute-force* i *divide & conquer* pristupa. Nakon što završimo provjeru na tom mjestu, oduzmemo taj broj od početnog i ponovimo postupak da dobijemo sljedeću poziciju greške koju treba provjeriti.

Opisane operacije manipulacija bitima i izračunavanja broja jedinica i logaritama se sve vrše u vremenu $O(1)$ nakon što se izvrši preprocesiranje podataka. Slučaj 4), u kojem je potrebno vršiti provjere na lokacijama grešaka, se obavlja u konstantnom vremenu po lokaciji, što ukupno može biti $O(R)$ vremena, gdje sa R označavamo maksimalnu razliku bilo koja dva stringa u A. Preprocesiranje niza A da se dobiju bitmaske razlika se izvršava u vremenu $O(NM)$ jer se svaki string poredi samo sa prvim. Na kraju, pošto stringovi mogu imati najviše 50 elemenata, bitmaske mogu u potpunosti stati u 64-bitne cjelobrojne varijable. To daje ukupno vrijeme preprocesiranja $O(NM)$ i vrijeme izvršavanja svakog upita $O(NR)$, što je ukupno $O(NQR)$ vremena. Ovo je brže od $O(NQM)$ ako je $R < M$, a ako je $R = M$ onda je brže dok god se svaka dva stringa u A ne razlikuju u svim mjestima.