1. Caesari šiffer

Minimaalse programmeerimisvaevaga viis selle ülesande lahendamiseks on kirjutada programm, mis loeb kasutajalt eeldatava nihke ja dešifreerib ühe faili selle nihke abil. Selline programm on toodud failis `cslah0.pas`. Vajalike väljundfailide saamiseks oleks selle programmi kasutaja pidanud püüdma sisendfaile erinevate nihetega dešifreerida ja kontrollima, millise nihke korral on tulemuseks lubatud sõnadest koosnev tekst. Praktikas oleks küll piisanud vaid pealiskaudsest ülevaatusest, et tulemus enam-vähem inglise keele moodi välja näeb.

Sisuliselt sama funktsionaalsus on ka failis `cstestgen.c` toodud testigeneraatoris. Kuna see on mõeldud dešifreerimise asemel šifreerima, tuleb selle programmi kasutamisel failis `cslah0.pas` toodud lahendusega sama tulemuse saamiseks nihke N asemel anda ette nihe $26 - N$. Kuna aga nihke suurust lahenduse osana ei küsitud, pole sellel tegelikult tähtsust.

Kasutajale mugavam on failis `cslah1.pas` toodud lahendus, mis proovib ise kõik võimalikud nihked läbi ja kontrollib, milline neist annab ainult lubatud sõnadest koosneva tulemuse. Kuna Pascalis on tingimuse "näeb enam-vähem inglise keele moodi välja" programmeerimine keerulisem kui sõnastikust otsimine, kontrollib see lahendus tulemuse lubatavust sõnastiku alusel.

Testid

10 testi, à 3 punkti, kokku 30 punkti.

2. Sõnede loendamine

Selle ülesande lahendamiseks tuleb tähele panna, et N-tähelises tähestikus K-tähelise sõne genereerimisel võib sõne igat tähte valida teistest sõltumatult N erineval viisil. See tähendab, et N-tähelise tähestiku korral on võimalik moodustada N^K täpselt K-tähelist sõnet. Kuni M-täheliste sõnede koguarv N-tähelises tähestikus on seega

$$N + N^2 + N^3 + \cdots + N^M.$$

Kuna Pascalis astendamisfunktsiooni ei ole, arvutab failis `sllah1.pas` toodud lahendus N astmed välja järjestikuste korrutamistega ja liidab nad jooksvalt kokku.

Keeltes, kus astendamine on standardselt olemas (või kus seda on võimalik logaritmide kaudu avaldada, nagu on tehtud failis `sllah2.pas`), võib muidugi korduse asemel kasutada ka geomeetrilise jada summa valemit

$$N + N^2 + N^3 + \cdots + N^M = \frac{N(N^M - 1)}{N - 1}$$

aga siis peab olema ettevaatlik, et mitte jagada nulliga, kui $N = 1$.

Testid

1. $N = 1$, $M = 1$. Vastus 1. Minimaalne test. 3 punkti.
2. $N = 4$, $M = 4$. Vastus 340. Väike lihtne test. 3 punkti.
3. $N = 1$, $M = 100$. Vastus 100. Maksimaalse pikkusega sõne. 3 punkti.
4. $N = 7$, $M = 5$. Vastus 19 607. Keskmise suurusega juhuslik test. 3 punkti.
5. $N = 17$, $M = 4$. Vastus 88 740. Keskmise suurusega juhuslik test. 3 punkti.
6. $N = 10$, $M = 8$. Vastus 111 111 110. Variantide läbivaatus jääb ajahätta. 3 punkti.
7. $N = 19$, $M = 7$. Vastus 943 531 279. Juhuslik suur test. 3 punkti.
8. $N = 100$, $M = 4$. Vastus 101 010 100. Maksimaalne tähestik. 3 punkti.
9. $N = 35$, $M = 6$. Vastus 1 892 332 260. Suur tähestik, maksimumilähedane vastus. 3 punkti.
10. $N = 2$, $M = 30$. Vastus 2 147 483 646. Pikad sõned, maksimaalne vastus. 3 punkti.

Kokku 30 punkti.

3. IPv6 aadressid

Tegemist on praktilise kallakuga ülesandega, mille lahendamiseks pole vaja erilist teooriat ega ideed. Töötab täiesti sirgjooneline algoritm:

1. Tuvastame aadressis olevate koolonite arvu järgi, kas ja kui palju nulle sellest välja on jäetud.
2. Taastame eemaldatud nullid, juhindudes topeltkooloni asukohast.
3. Muudame kõik lühemad komponendid algusnullidega lisamisega neljakohalisteks.
4. Asendame kõik kuueteistkümnendnumbrid ülesande tekstis toodud tabeli põhjal neljakohaliste kahendarvudega.

Failis `iplah1.pas` toodud lahendus kaldub programmeerimismugavuse huvides eeltoodud skeemist veidi kõrvale, kuid erinevused on vaid detailides.

Testid

1. Sisend: `1234:5678:90AB:CDEF:FEDC:BA09:8765:4321`. Täisaadress, kõik komponendid 4-kohaliselt välja kirjutatud. 5 punkti.
2. Sisend: `1234:567:89:0:A:B:CD:EF`. Täisaadress, mõnes komponendis tuleb algusnullid juurde mõelda. 5 punkti.
3. Sisend: `1:3:5::B:D:F`. Aadressi keskelt komponente puudu. 5 punkti.

4.a. Sisend: `::2:4:6`. Aadressi algusest komponente puudu. 5 punkti.

4.b. Sisend: `:2:4:6`. Aadressi algusest komponente puudu. 5 punkti.

5.a. Sisend: `ABCD:EF::`. Aadressi lõpust komponente puudu. 5 punkti.

5.b. Sisend: `ABCD:EF:`. Aadressi lõpust komponente puudu. 5 punkti.

6.a. Sisend: `::`. Tühiaadress. 5 punkti.

6.b. Sisend: (tühi sõne). Tühiaadress. 5 punkti.

6.c. Sisend: `:`. Tühiaadress. 5 punkti.

Kokku 30 punkti.

Märkus. Kuna ülesande teksti esialgne sõnastus oli aadressi algusest ja lõpust nullide eemaldamise osas mitmetimõistetav, said testides 4 kuni 6 punkte kõik lahendused, mis andsid õiged vastused vähemalt ühe sisendivariandi korral, kuid ühe testi kõigi variantide peale kokku siiski mitte üle 5 punkti.

4. Suluavaldised

Selle ülesande lahendamiseks on kõigepealt vaja märgata, et liiasuse definitsioon tähendab sisuliselt seda, et avaldise koostamisel väiksematest osadest ei tohi me suluavaldise definitsioonis toodud reegleid 2 ja 3 kumbagi kasutada kahte korda järjest — reegli 2 alusel loodud avaldisi võib omavahel kokku kleepida ainult reegli 3 alusel ja vastupidi, reegli 3 abil kokkukleebitud avaldist tohib kasvatada ainult reegli 2 alusel.

Pärast selle tähelepaneku tegemist pole enam kuigi raske mõelda välja lihtsat algoritmi: alustame tühjast sõnest ja rakendame neid reegleid kordamööda, lisades reegli 3 rakendamisel olemasolevale avaldisele alati juurde aatomi (). Selline konstruktsioon tagab, et loome ainult korrektseid ja liiasuseta avaldisi. Kuna igal sammul kasvab avaldise pikkus täpselt kahe märgi võrra, on võimalik sobival hetkel lõpetades saada ükskõik millise paarisarvulise pikkusega avaldis.

Failis `salah0.c` toodud lahenduse ainus puudus on, et avaldiste kasvatamise käigus kopeeritakse neid pidevalt mälus ühest kohast teise. Kuna N märgist koosneva avaldise saamiseks tuleb seda avaldist pikendada $N/2$ korda ja keskmiselt kopeeritakse $N/2$-märgilisi avaldisi, tähendab see kokku umbes $N^2/4$ omistamist, mida on maksimumilähedaste N väärtuste korral liiga palju. Nii jääbki see lahendus paaris suuremas testis ajahätta.

Osaliste avaldiste mälus edasi-tagasi kopeerimise vältimiseks on muidugi kõige parem lõpptulemus kohe vasakult paremale valmis konstrueerida. Siis pole vaja teda isegi mälus hoida, vaid võib ta kohe konstreerimise käigus märkhaaval faili väljastada. Selle idee realiseerimisel on ainus raskus asjaolu, et reegli 2 rakendamisel loome vasakpoolse sulu väljastamisega endale tulevikuks kohustuse õigesse kohta sellele paariliseks parempoolne sulg väljastada. Seda kohustust on kõige lihtsam meeles pidada ja õigel ajal täita, kui kasutame alamavaldise loomiseks rekursiivselt sama algoritmi ja väljastame parempoolse sulu just siis, kui alamavaldise konstrueerimine ja väljastamine lõpeb. Reeglite vaheldumisi rakendamise meelespidamiseks võib siis vaadata parasjagu konstrueeritava alamavaldise pikkuse jääki neljaga jagamisel, nagu on tehtud failis `salah1.c` toodud lahenduses, või kasutada selleks eraldi abimuutujat, nagu on tehtud failis `salah1.pas` toodud lahenduses.

Seni vaadeldud lahendused ei ole sõltunud sellest, kas reegli 3 rakendamisel kleebitakse alamavaldis () oma paarilise külge vasakule või paremale või kasvõi juhuslikult kord ühele, kord teisele poole. Kui aga panna lisaks paika reegel, et tühi sulupaar lisatakse alati paremale, on võimalik vähese vaevaga välja arvutada, milline sulg (kas vasak- või parempoolne) peab valmis avaldise igal positsioonil olema, ja vajalikud sulud selle arvutuse põhjal ka ilma igasuguse rekursioonita järjest väljastada, nagu ongi tehtud failis `salah2.c` toodud lahenduses.

Loomulikult on lihtne kirjutada viimane lahendus ümber juhule, kui tühje sulupaare soovitakse lisada vasakule. Huvitavamad lisaharjutused oleks

- kirjutada ilma rekursioonita programm, mis lisab tühje sulupaare kordamööda vasakule ja paremale;
- kirjutada (algul rekursiooniga, siis rekursioonita) programm, mis püüab reegli 3 rakendamisel kleepida kokku kaks võimalikult võrdse pikkusega alamavaldist;
- kirjutada programm, mis jagab reegli 3 rakendamisel pikkuse kahe alamavaldise vahel juhuslikult (selle kirjutamine ilma rekursioonita pole otseselt võimatu, küll aga mõttetult tülikas).

Testid

1. $N = 8$. 3 punkti.
2. $N = 18$. 3 punkti.
3. $N = 98$. 3 punkti.
4. $N = 500$. 3 punkti.
5. $N = 1\,002$. 3 punkti.
6. $N = 4\,400$. 3 punkti.
7. $N = 7\,778$. 3 punkti.
8. $N = 14\,052$. 3 punkti.
9. $N = 50\,052$. 3 punkti.
10. $N = 100\,000$. 3 punkti.

Kokku 30 punkti.

5. Rott ja juust

Üks võimalus selle ülesande lahendamiseks on vaadata iga ruudu jaoks läbi kõik sellest ruudust algavad võimalikud liikumisteed — lähteruudust selle alumisse ja paremasse naabrisse, neist kummastki omakorda alla ja paremale jne — ja kontrollida otseselt, kas mõni neist viib sihile, ehk juustuni. Selline lahendus on põhimõtteliselt õige, aga kaunikesti ebaefektiivne. Nii lahendabki failis `rjlah0.pas` toodud programm ajalimiidi piires ainult 7 testi.

Naiivse kõigi võimalike variantide läbivaatusega lahenduse ebaefektiivsuse põhjuseks on, et väga paljusid variante uuritakse korduvalt. Kui mingi ruudu R alumine, parempoolne ja diagonaalis alla paremale jääv naaber on kõik läbitavad, siis vaadatakse ruudu R analüüsimisel seda diagonaalset naabrit läbi kaks korda — esimest korda teel ruudust R alla ja siis paremale, teist korda teel ruudust R paremale ja siis alla. Lisaks tehakse sama topelttööd uuesti iga kord, kui analüüsitakse mõnd ruutu, mis on ruudust R üleval või vasakul ja millest on tee ruutu R. Lihtne võimalus ajatut tööd vähendada on mistahes ruudu esmakordsel analüüsimisel tulemus meelde jätta ja järgmine kord seda ilma uuesti arvutamata mälust kasutada. Sellel ideel põhinev lahendus on toodud failis `rjlah1.pas` ja lahendab ajalimiiti ületamata kõik 10 testi.

Aga on veel parem võimalus. Nimelt sõltub labürindi mistahes ruutu väljastatav tulemus ainult sellest, kas tema alumisest ja parempoolsest naabrist on olemas mingi tee juustuni. See tähendab, et kui me analüüsime labürindi ruute sellises järjekorras, et iga ruudu R alumine ja parempoolne naaber saavad uuritud enne ruutu R ennast, siis polegi mingit variantide läbivaatust vaja. Kõige lihtsam seda tingimust rahuldav järjekord on falist lugemisele vastupidine: ridade kaupa alt üles ja igas reas paremale vasakule. Sellel ideel põhinevad lahendused on toodud failides `rjlah2.pas` ja `rjlah2.cpp`.

Testid

1. $N = 1$, $M = 1$. Minimaalne test. 4 punkti.
2. $N = 1$, $M = 10$. Üherealine test. 4 punkti.
3. $N = 20$, $M = 1$. Üheveeruline test. 4 punkti.
4. $N = 3$, $M = 3$. Ainult kaks takistust, aga juust täielikult blokeeritud, kuskilt ei saa ligi. 4 punkti.
5. $N = 40$, $M = 40$. Juustuni viib kitsas diagonaalne tunnel. 4 punkti.
6. $N = 24$, $M = 22$. Labürindi keskosa blokeerib suur L-kujuline takistus. 4 punkti.
7. $N = 17$, $M = 33$. Liikumine vertikaalsete takistusribade vahel. 4 punkti.
8. $N = 50$, $M = 40$. Juhuslik väheste takistustega test. 4 punkti.
9. $N = 34$, $M = 90$. Juhuslik rohkete takistustega test. 4 punkti.
10. $N = 100$, $M = 100$. Maksimaalne juhuslik test, peaaegu läbipääsmatu. 4 punkti.

Kokku 40 punkti.

6. Ekspressbussid

Ilmselt on tegemist graafiülesandega. Kui modelleerida liinivõrk loomulikul viisil — peatused on graafi tipud ja liinid nende vahel servad —, siis on meil tegemist lühimate teede leidmise ja nende loendamisega.

Üks võimalus seda ülesannet lahendada on kasutada kõigi võimalike lähtepeatusest algavate marsruutide ammendavat läbivaatust. Kuna me otsime minimaalse pikkusega teid lähtepeatusest sihtpeatusse, võime oma läbivaatuse kohe kitsendada aiult sellistele marsruutidele, mis ei läbi ühtki peatust korduvalt. Edasi on kasulik eraldi muutujates meeles pidada seni leitud marsruutide pikkuste miinimumi ja selle minimaalse pikkusega marsruutide arvu. Sel juhul võime iga kord sihtpeatusse jõudes võrrelda äsjaleitud marsruudi pikkust senise parima pikkusega. Edasi on kolm võimalust:

- kui uus marsruut on pikem, pole ta kindlasti minimaalne võimalik ja seega me teda ei loe;
- kui uus marsruut on senise miinumumiga ühepikkune, suurendame minimaalsete loendurit ühe võrra;
- kui uus marsruut on lühem, tähendab see, et senine miinimum ei olnud tegelikult optimaalne; siis määrame uue marsruudi pikkuse uueks miinimumiks ja alustame loendamist jälle ühest.

Selline lahendus on toodud failis `eblah0a.pas`. Paraku on see üsna ebaefektiive, sest võimalike marsruutide arv kasvab peatuste ja liinide lisandudes väga kiiresti.

Lihtne viis seda programmi parandada on märgata, et kui parajasti vaadeldav tee on juba parima seni leitu pikkune, aga pole veel sihtpeatusse välja jõudnud, pole enam mõtet seda edasi uurida. See lihtne parandus tõstab programmi efektiivsust märgatavalt ja failis `eblah0b.pas` toodud lahendus saab ajalimiidi piires hakkama 7 testiga eelmise variandi 4 asemel. Sellist variantide arvu vähendamist nimetatakse otsingupuu pügamiseks.

Optimaalse lahenduse leidmiseks tuleks sügavuti läbimise asemel kasutada graafi läbimist laiuti. Lühidalt kokku võttes tähendab laiuti läbimine, et me teeme kõigepealt nimekirja neist peatustest, kuhu saab lähtepeatusest otseliiniga; edasi uue nimekirja neist peatustest, kuhu me veel pole jõudnud, kuid kuhu saab otseliinidega esimese nimekirja peatustest; seejärel kolmanda nimekirja neist, kuhu saab teise nimekirja omadest jne. Peaks olema üsna selge, et niimoodi satuvad igasse nimekirja peatused, millesse jõudmiseks on vaja vähemalt nii palju talonge, kui suur on selle nimekirja järjekorranumber. Samas on algoritm ka üsna efektiivne, sest igast peatusest väljuvaid liine vaadatakse läbi ainult üks kord — siis, kui selle peatuse nimekirja põhjal koostatakse järgmist.

Minimaalse piletite arvu oleme me sellega leidnud, vaja on veel leida erinevate marsruutide arv. Selle efektiivseks leidmiseks paneme tähele, et peatusse B saame K piletiga sõita ainult nii, et vahetult eelmisena peame läbima mingi peatuse C, mille kaugus lähtepeatusest on $K-1$. Seejuures annab iga $K-1$ piletiga marsruut üskõik millisesse peatusse C, millest läheb otseliin peatusse B, ühe unikaalse K piletiga marsruudi peatusse B. Järelikult saame marsruutide arvud loendada paralleelselt nende pikkuste leidmisega — iga kord ühest peatuste nimekirjast teise nimekirja tegemisel liidame iga uude nimekirja paigutatava peatuse D marsruutide arvu saamiseks kokku eelmise nimekirja kõigi nende peatuste marsruutide arvud, millest on otseliin peatusse D.

Sellel ideel põhinevad lahendused on toodud failides `eblah1a.pas` ja `eblah1b.cpp`. Kumbki neist ei jaga tegelikult peatusi eraldi nimekirjadesse, vaid peab iga peatuse jaoks lihtsalt meeles, mitmendasse nimekirja see kuuluma peaks. Oluline on ainult, et nad mõlemad vaatavad peatu-

si läbi nende nimekrijanumbrite järjekorras. Mõlemad lahendused teenivad maksimumpunktid, kuigi C++ versioon on Pascali omast õige veidi efektiivsem tänu spetsiaalse järjekorratüübi kasutamisele. Pascalis sellist andmestruktuuri standardvarustuses ei ole, selle kasutamise soovi korral peaks selle ise programmeerima (see poleks ka kuigi keeruline, aga on lahenduse lihtsuse huvides tegemata jäetud).

Testid

1. $N = 2$, $M = 1$. Vastus: 1 pilet, 1 marsruut. Minimaalne test. 4 punkti.
2. $N = 5$, $M = 5$. Vastus: 2 piletit, 1 marsruut. Väike tsüklikujuline test. 4 punkti.
3. $N = 9$, $M = 12$. Vastus: 4 piletit, 6 marsruuti. Liinivõrk moodustab korrapärase 3×3 ruudustiku. 4 punkti.
4. $N = 20$, $M = 30$. Vastus: 4 piletit, 2 marsruuti. Juhuslik test. 4 punkti.
5. $N = 40$, $M = 100$. Vastus: 3 piletit, 8 marsruuti. Juhuslik test. 4 punkti.
6. $N = 60$, $M = 300$. Vastus: 3 piletit, 19 marsruuti. Juhuslik test. 4 punkti.
7. $N = 100$, $M = 180$. Vastus: 18 piletit, 48 620 marsruuti. Liinivõrk moodustab korrapärase 10×10 ruudustiku. 4 punkti.
8. $N = 100$, $M = 132$. Vastus: 60 piletit, 1 073 741 824 marsruuti. Suur "rombidega" graaf. Mõnel peatusel üle 2^{31} marsruudi. 4 punkti.
9. $N = 100$, $M = 150$. Vastus: 25 piletit, 1 062 882 marsruuti. Suur rõngakujuline "3-haruliste rombidega" graaf. 4 punkti.
10. $N = 100$, $M = 1000$. Vastus: 3 piletit, 69 marsruuti. Maksimaalne juhuslik test. 4 punkti.

Kokku 40 punkti.