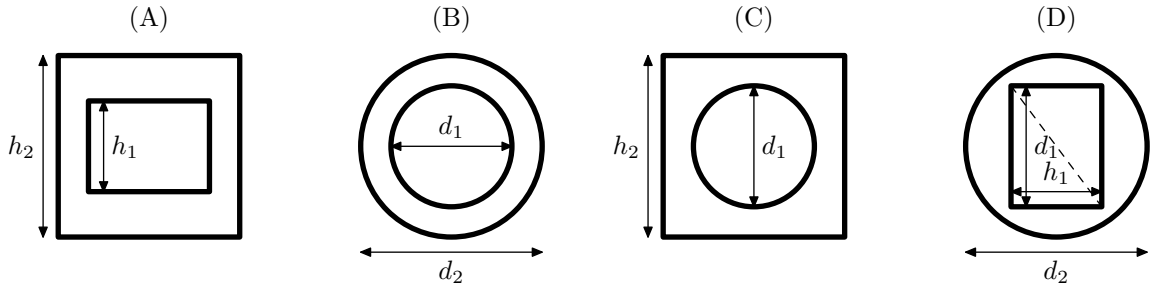
1. Vaadid

Vaatleme eraldi kaht ülesande tingimustes lubatud vaatide üksteise sisse paigutamise viisi.

Kui mõlemad vaadid on püsti, näeme joonisel (A) kujutatud külgvaatest, et sisemise vaadi kõrgus peab olema väiksem välimise omast, ja joonisel (B) kujutatud pealtvaatest, et ka läbimõõt peab sisemisel vaadil olema väiksem kui välimisel. Kokku saame sel juhul ühe vaadi teise sisse mahtumise tingimuseks $d_1 < d_2$ ja $h_1 < h_2$.



Kui sisemine vaat on külili, näeme joonisel (C) kujutatud külgvaatest (millel sisemine vaat on meie poole otsaga), et sisemise vaadi läbimõõt peab olema väiksem välimise vaadi kõrgusest, ja joonisel (D) kujutatud pealtvaatest (millel sisemine vaat on meie poole küljega), et sisemise vaadi läbilõike diagonaali pikkus peab olema väiksem välimise vaadi läbimõõdust.

Pythagorase teoreemi kohaselt rahuldab ristküliku diagonaali pikkus l tingimust $l^2 = d_1^2 + h_1^2$, millest võiksime avaldada $l = \sqrt{d_1^2 + h_1^2}$. Siiski on kasulikum seda vältida, sest juurimise järel peaksime arvutama reaalarvudega, mis on arvutis ligikaudsed suurused ja võivad põhjustada ümardamisvigu. Selle asemel võime kasutada ära asjaolu, et ruutfunktsioon on mittenegatiivsetel arvudel rangelt kasvav, mis tähendab, et arvude endi asemel võime võrrelda nende ruutusid.

Kokku saame sel juhul ühe vaadi teise sisse mahtumise tingimuseks $d_1 < h_2$ ja $d_1^2 + h_1^2 < d_2^2$. Programmi kirjutamisel peaks veel tähele panema, et kuigi vaatide mõõdud jäävad 16-bitiste täisarvude määramispiirkonda, võivad nende ruudud olla suuremad, seega tuleks ruutude jaoks kindlasti kasutada vähemalt 32-bitiseid andmetüüpe.

Testid

1. v_1 `->` v_2 `->` v_3, kõik vaadid võivad olla püsti. 3 punkti.
2. v_1 `<-` v_2 `<-` v_3, kõik vaadid võivad olla püsti. 3 punkti.
3. v_1 `->` v_2 `->` v_3, aga v_1 peab alati olema külili. 3 punkti.
4. v_1 `<-` v_2 `<-` v_3, aga v_3 peab alati olema külili. 3 punkti.
5. v_1 `--` v_2 `--` v_3 `--` v_1. 3 punkti.
6. v_1 `->` v_2 `<-` v_3 `--` v_1, üks mahtumine püsti, teine külili. 3 punkti.
7. v_1 `<-` v_2 `->` v_3 `--` v_1, üks mahtumine püsti, teine külili. 3 punkti.
8. v_1 `--` v_2 `--` v_3 `--` v_1, mitterange võrratus püstijuhul annab vale vastuse. 3 punkti.
9. v_1 `--` v_2 `--` v_3 `--` v_1, mitterange võrratus külilijuhul annab vale vastuse. 3 punkti.
10. v_1 `->` v_2 `<-` v_3 `--` v_1, suured arvud. 3 punkti.

Kokku 30 punkti.

2. Maleseis

Selle ülesande lahendamiseks on kasulik tähele panna, et sisend kirjeldab malelaua väljasid samas järjekorras kui väljund — sisuliselt on tegu väga primitiivse pakkimisega, kus tühjade väljade ükshaaval esitamise asemel näidatakse nende arv ja muud andmed säilitatakse enam-vähem esialgsel kujul.

Lisaks on sisendis andmeid väikese liiaga — kuigi malendite ja tühjade ruutude loendamisega on niigi võimalik aru saada, kus malelaua üks rida lõpeb ja teine algab, on reavahed ka eraldi märgitud. Sellest tulenevalt ongi kaks põhilist võimalust selle ülesande "käsitsi" lahendamiseks

- loendada juba väljastatud märke ja lisada iga kaheksa märgi järele reavahetus, nagu on tehtud failides `malelah1.c`, `malelah1.cpp` ja `malelah1.pas` toodud lahendustes;
- teha reavahetus iga sisendist loetud kaldkriipsu kohal, nagu on tehtud failis `malelah2.pas` toodud lahenduses.

Tasub tähele panna, et terve seisu mällu valmis konstrueerimine pole selle ülesande lahendamiseks üldse vajalik, kuigi seda võib teha, nagu on näha failides `malelah3.pas` ja `malelah4.pas` toodud näidetes.

Ja lõpuks, keeltes, mille standardvarustusse kuulub sõneoperatsioon "otsi ja asenda", võib ülesande lahendada ka selle abil, nagu on tehtud failides `malelah5.java`, `malelah5.py` ning `malelah5a.pl` ja `malelah5b.pl` toodud lahendustes.

Testid

1. Igal real ainult üks malend, kas rea alguses või lõpus. 3 punkti.
2. Igal real ainult üks malend, erinevates positsioonides. Üks ettur on lipustunud. 3 punkti.
3. Igal real maksimaalselt kaks malendit. Ükse ettur on lipustunud. 3 punkti.
4. Palju malendeid, tühjade väljadega vaheldumisi. Maksimaalse pikkusega sisend. 3 punkti.
5. Palju malendeid, mitu malendit kõrvuti. 3 punkti.
6. Juhuslik seis avamängust (malendid koondunud laua ülemisse ja alumisse serva). 3 punkti.
7. Juhuslik seis keskmängust (malendid ühtlaselt üle laua). 3 punkti.
8. Juhuslik seis keskmängust (malendid ühtlaselt üle laua). 3 punkti.
9. Juhuslik seis lõppmängust (malendeid laual üsna hõredalt). 3 punkti.
10. Juhuslik seis lõppmängust (malendeid laual üsna hõredalt). 3 punkti.

Kokku 30 punkti.

3. Sõnastik

Kõige sirgjoonelisem viis selle ülesande lahendamiseks on võrrelda päringusõna iga sõnaga sõnastikust. Nagu näha failis `sonadlah0.pas`, on selline lahendus küll lihtne, aga kulutab M-sõnalise sõnastiku ja N päringusõna korral umbes $N \cdot M$ võrdlust ja jääb suuremates testides ajahätta.

Eelmises lõigus kirjeldatud lineaarse otsingu ebaefektiivsuse põhjus on see, et iga võrdlusega heidame kõrvale ainult ühe sobimatu kandidaadi — selle sõna, mida me just päringusõnaga võrdlesime. Teades, et sõnastik on sorteeritud, võime aga iga võrdluse põhjal heita kõrvale ligi pooled veel alles jäänud kandidaatidest. Selleks tuleb otsingu igal sammul võrrelda päringusõna veel allesjäänud sõnadest keskmisega. Vaatleme näiteks alloleval joonisel kujutatud seisu, kus meil on vaja leida sõna s sorteeritud jadas $a_v \leqslant a_{v+1} \leqslant \ldots \leqslant a_{p-1} \leqslant a_p$. Kui me selles olukorras valime võrdlemiseks jada keskmise elemendi (a_k, kus $k = (v+p)/2$), saame tulemusest $s < a_k$ järeldada, et $s < a_i$ ka kõigi $i > k$ korral ja kahandada otsimisvahemiku üheainsa võrdluse põhjal jadaosale $a_{v\ldots k-1}$, ehk poole väiksemaks. Analoogiliselt saame $s > a_k$ korral heita kõrvale terve vasaku jadapoole ja kahandada otsimisvahemiku jadaosale $a_{k+1\ldots p}$. M-elemendilist jada saame niimoodi poolitada maksimaalselt $\log_2 M$ korda — siis oleme kas otsitava elemendi leidnud või otsimispiirkonna tühjaks hulgaks kahandanud. Sellel ideel põhinebki failis `sonadlah1.pas` toodud lahendus, mis realiseerib kahendotsingu ilmutatud kujul, ja failis `sonadlah1.cpp` toodud lahendus, mis kasutab C++ standardteegis realiseeritud kahendotsingut.

$$\overbrace{a_v \leqslant \ldots \leqslant a_{k-1}}^{\leqslant a_k} \leqslant a_k \leqslant \overbrace{a_{k+1} \leqslant \ldots \leqslant a_p}^{\geqslant a_k}$$

Keeltes, kus on võimalik moodustada suvalist tüüpi elementidest hulki, on ka need tavaliselt piisavalt efektiivselt realiseeritud ja nii oleks ka failides `sonadlah2.cpp` ja `sonadlah2.py` toodud lahenduste eest saanud maksimumpunktid.

Keeltes, kus otseselt hulgatüüpi ei ole, on mõnikord võimalik "kuritarvitada" mõnd teist piisavalt paindlikku andmetüüpi (tavaliselt sõnastikutüüpi ehk assotsiatiivset massiivi), nagu on tehtud failides `sonadlah3.cpp` ja `sonadlah3.pl`, millest oleks samuti täispunktideks piisanud.

Testid

1. 3 sõna, 7 päringut. Väike väga lihtne test. 4/2 punkti.
2. 4 sõna, 8 päringut. Üks päring otsib sõnastiku esimest sõna. 4/2 punkti.
3. 4 sõna, 8 päringut. Üks päring otsib sõnastiku viimast sõna. 4/2 punkti.
4. 0 sõna, 12 päringut. Tühi sõnastik, järelikult vastus alati `EI`. 4/2 punkti.
5. 30 sõna, 100 päringut. Kõik read maksimaalse pikkusega. 4/2 punkti.
6. 1000 sõna, 10 päringut. 4/2 punkti.
7. 30 sõna, 1000 päringut. 4/2 punkti.
8. 10 000 sõna, 1000 päringut. 4/2 punkti.
9. 10 000 sõna, 10 000 päringut. Lineaarne otsing jääb kindlalt ajahätta. 8/4 punkti.

Kokku algajatel 40, edasijõudnutel 20 punkti.

4. Sidekaablite remont

Tegu on graafiteooria standardülesandega, mida nimetatakse sildade leidmiseks. Silllaks nimetatakse serva, mille eemaldamisel suureneb graafi sidususkomponentide arv. Järelikult muutub algselt sidus graaf mistahes silla eemaldamisel mittesidusaks.

Lihtsaim lahendus ongi proovida iga serva ükshaaval graafist eemaldada ja vaadata, kas graaf endiselt sidusaks jääb, st kas igast külast saab endiselt igasse teise. Sealt veidi edasi mõeldes selgub, et piisab ka sellest, kui leidub tee ühest eemaldatud serva tipust teise, sest siis ei saa graaf selle serva eemaldamisel veel kaheks eraldiseisvaks osaks jaguneda. Sellel ideel põhinev lahendus kulutab N tipu ja M servaga graafi töötlemiseks sõltuvalt tee otsimise algoritmist kas $O(M \cdot N^2)$ või $O(M^2)$ sammu, mis on selles ülesandes maksimumpunktide teenimiseks piisavalt efektiivne, nagu näitavad failides `sidelah1.pas`, `sidelah2.c` ja `sidelah3.c` toodud lahendused.

Aga leidub ka kiirem lahendus, mis töötab graafi servade arvu suhtes lineaarse ajaga. See toimib graafi sügavuti läbimisega, kus tuleb teha mõned suhteliselt kavalad tähelepanekud. On selge, et läbimine kasutab kõiki sildu, sest vastasel juhul ei jõuaks ta lihtsalt kõikide tippudeni. Kui nüüd läbimine kasutab tipust a edasi liikumisel serva $a-b$, on viimane sild parajasti juhul, kui ühestki b kaudu külastatud tipust ei lähe ühtegi serva tagasi a-sse ega ühtegi enne temani jõudmist külastatud tippu. Et sügavuti läbimine külastab tippe mingis järjekorras, on võimalik iga tipu juures hoida ka tema külastamise järjekorranumbrit. Kui nüüd mõnest tipust viib serv mõnesse varem külastatusse, siis läbimine seda serva ei kasuta, kuid me võime selle fakti sellegipoolest üles märkida. Kui nüüd otsingupuus tagasi minnes see väärtus samuti edasi anda, on iga tipu jaoks võimalik leida vähima järjekorranumbriga tipu indeks, kuhu oli võimalik pääseda temast või tema alampuu tippudest (võttes miinimumi tema enda järjekorranumbrist, servadest, mis temast lähtuvad, kuid mida läbimine temast edasi ei kasutanud ja sellest samast väärtusest kõikidel nendel tippudel, kuhu otsingus temast edasi mindi). Kui nüüd mõne tipu jaoks juhtub, et see miinimum on ta enda järjekorranumber, on teada, et tema alampuust temast eespool läbi käidud tippudesse kuidagi minna ei saa, st serv, mille kaudu sellesse konkreetsesse tippu jõuti, peabki olema sild. Sel viisil tulevad tõepoolest välja kõik sillad. Sellel ideel põhinebki failis `sidelah4.c` toodud lahendus, mis töötab kõikides testides silmapilkselt.

Testid

1. 1 tipp, silmusega. 0 silda. 4 punkti.
2. 3 tippu, ahelas, ühendatud vastavalt 1, 2, 3 servaga. 1 sild. 4 punkti.
3. 5-tipuline puu, lisaks kõik tipud silmustega. 4 silda. 4 punkti.
4. 25-tipuline tsükkel, 0 silda. 4 punkti.
5. 30 tippu, 50 serva, juhuslik graaf. 6 silda. 4 punkti.
6. 40 tippu, 100 serva, juhuslik graaf. 1 sild. 4 punkti.
7. 100 tippu, 337 serva, juhuslik graaf. 4 silda. 4 punkti.
8. 500-tipuline puu. 499 silda. 4 punkti.
9. 100-tipuline täisgraaf. 0 silda. 4 punkti.
10. 500 tippu, 5000 serva, maksimaalse suurusega juhuslik graaf. Üks “pettesild” — serv, mis oleks sild, kui teda topelt poleks. 7 silda. 4 punkti.

Kokku 40 punkti.

5. Sudoku

Arvatavasti pole raske näha, et selles ülesandes võib vaja minna variantide läbivaatust. Olukorras, kus piiratud aja jooksul on vaja kirjutada variantide läbivaatus ja lahendada selle abil võimalikult palju teste, on väga kasulik osata oma lahendust kõiki teste paralleelselt lahendama panna. Erinevaid teste ükshaaval käivitades on väga raske hinnata optimaalset aega, mida iga testi peale kulutada. Kui programmi valmimise hetkel on võistluse lõpuni jäänud 2 tundi ja meil on vaja lahendada 10 testi, võib tunduda, et parim oleks anda igale testile ajapiiriks 12 minutit. See pole aga optimaalne strateegia näiteks juhul, kui 1. testi lahendamiseks oleks vaja 15 minutit ja 2. testis saame vastuse juba 3 minutiga: siis oleks me 2. testi arvelt võinud 1. testile rohkem aega anda, aga kuna me seda õigel ajal ei teadnud, peaks me nüüd 1. testi lahendamist uuesti algusest alustama.

Konkreetselt sudoku-ülesande poole pöördudes on esimene võimalus selle lahendamiseks lihtne kõigi varientide täielik läbivaatus. Loomulikult tuleb kohe vahele jätta variandid, kus mõni täht esineb korduvalt samas reas, veerus või 3×5 ristkülikus. Sellel ideel põhinev programm on üsna lihtne, aga ka aga üsna ebaefektiivne. Näiteks, kui panna failis `sudokulah1.pas` toodud lahendus 2 GHz Celeron protsessoriga arvutil paralleelselt kõiki teste lahendama, saame 2 tunni jooksul 6 vastust, neist viimase umbes 90 minutit pärast otsingu käivitamist.

Üks võimalus variantide arvu vähendada (ja sellega nende läbivaatamist kiirendada), on pidada veidi täpsemat arvet igas ruudus veel võimalike tähtede hulga üle. Ilmselt on see hulk etteantud väärtustega ruutudel üheelemendiline ja teistel üldjuhul 15-elemendiline, miinus juba leitud kitsendused. Kõige lihtsam kitsendus on muidugi see, et kuna ükski täht ei tohi samas reas, veerus või 3×5 ristkülikus (edasises nimetame neid kokkuvõtlikult blokkideks) esineda rohkem kui ühes eksemplaris, ei tohi ühtki sisendis ette antud tähte oma bloki piires mujal kasutada. Aga seda ideed võib üldistada: kui meil on mingi n-elemendiline ühte blokki kuuluvate ruutude hulk ja nende ruutude lubatud tähtede hulkade ühendis on täpselt n tähte, siis ei või ükski neist tähtedest selle bloki piires mujal esineda. Tõepoolest, kui me mõnda neist tähtedest mujal kasutaks, siis peaks me ta kõigist neist n hulgast eemaldama ja kokkuvõttes jääks meil n ruudu täitmiseks ainult $n-1$ tähte, millest selgelt ei jätku. Kuna igas 15 ruudust koosnevas blokis on mittetühje ruutude alamhulki $2^{15} - 1 = 32\,767$ ja blokke endid on meil $3 \cdot 15$, kulub kõigi alamhulkade ammendavaks läbivaatuseks vaid mõni sekund ja seega võime endale lubada kõigi kitsendamisvõimaluste korduvat läbivaatust, kuni selle reegli alusel enam ühtki hulka kitsendada ei õnnestu. Pärast seda alustame variantide läbivaatust, proovides igasse ruutu ainult neid tähti, mida eelnev töö veel välistanud pole. Sellel ideel põhinebki failis `sudokulah2.pas` toodud lahendus, mis on eelmisest nii palju efektiivsem, et leiab esimesed 6 vastust juba 4 minutiga. Kui esimese lahenduse täiendamine uueks võtab aega vähem kui 90 minutit, siis saame teisel juhul programmeerimise ja programmi täitmise kokkuvõttes samapalju teste (ja punkte) lühema ajaga.

Muidugi on võimalik variantide arvu kitsendamise heuristikaid veelgi parandada. Näiteks failis `sudokulah3.c` toodud lahendus kasutab üsna paljusid vilunud sudoku-lahendaja reegleid ja leiab ükskõik millises antud testidest vastuse maksimaalselt paari sekundiga, kulutades seega kogu komplektile kokku tunduvalt alla 1 minuti.

Testid

Kõigis kasutatud testides on õige vastus ühene.

10 testi, igaüks 4 punkti, kokku 40 punkti.