Janez Demšar

Algoritmi

Algoritem je "recept", kako rešiti nek splošen problem. Algoritem mora *točno določati postopek* – računalnik ne more ničesar delati "po občutku". Algoritem lahko uporablja naključne operacije ("izberi tri naključne elemente zaporedja"), ne more pa vsebovati operacij, za katere ni jasno, kako naj bi jih izvedli ("izberi tisti element, ki te bo najhitreje pripeljal do rešitve" – pri čemer ni jasno povedano, kateri je ta element).

Ko opazujemo algoritme, nas najprej zanima, ali so pravilni, torej, ali dajo vedno pravi rezultat. Poleg točnih algoritmov poznamo poznamo tudi hevristične: to so algoritmi, ki morda ne dajo pravega odgovora temveč samo približek. Če bi radi izvedeli, recimo, najmanjše število, ki ustreza določenim pogojem, vendar bi njegovo iskanje trajalo predolgo, zahtevalo preveč pomnilnika ali kaj podobnega, bomo navadno zadovoljni že z "približno najmanjšim" številom.

Drugo, kar nas zanima ob algoritmih, je, kako hitri so. Ker so računalniki različno hitri – in postajajo vedno hitrejši – hitrosti ne merimo absolutno, temveč relativno, glede na velikost problema. Če v tuji kuhinji iščemo predal s pokrovkami, bomo v kuhinji z dvakrat več predali potrebovali dvakrat toliko časa; algoritmom, ki se vedejo tako, pravimo, da imajo linearno časovno zahtevnost. Včasih je zahtevnost manjša od linearne in za dvakrat večji problem potrebujemo, recimo, en sam korak algoritma več. Včasih pa za dvakrat večji problem potrebujemo štirikrat toliko (in za petkrat večji petindvajsetkrat toliko) časa. So pa še hujši problemi, pri katerih že povečanje problema (na primer števila predalov) za en sam element podvoji čas, potreben za iskanje rešitve.

Algoritmi so, skupaj s podatkovnimi strukturami, eden najpomembnejših (in za mnoge najtežjih) predmetov v študiju računalništva. Kot bi vedeli povedati stari profesorji, so "o tem napisane debele knjige". V tem kratkem pregledu očitno ne bomo mogli spoznati vseh. Izbor bomo krojili po tem, kateri algoritmi se pogosto pojavljajo na tekmovanju, kateri algoritmi so najbolj poučni in kateri so najbolj zanimivi.

Predvsem algoritmi na grafih so zelo popularna tema nalog na tekmovanju Bober, gre pa le za nekaj različnih algoritmov. Če je naš namen le pripravljati otroke na tekmovanje, se tej temi gotovo splača posvetiti. Obenem so algoritmi na grafih zanimivi in privlačni, zato se jih splača poučevati tudi, če bi otrokom radi pokazali čare algoritmičnega razmišljanja. Končno, grafe v resnici srečamo na vsakem vogalu, tudi tam, kjer jih res ne bi pričakovali. Številne vsakdanje probleme – ali vsaj vsakdanje uganke – lahko prevedemo na katerega od "standardnih" problemov na grafih.

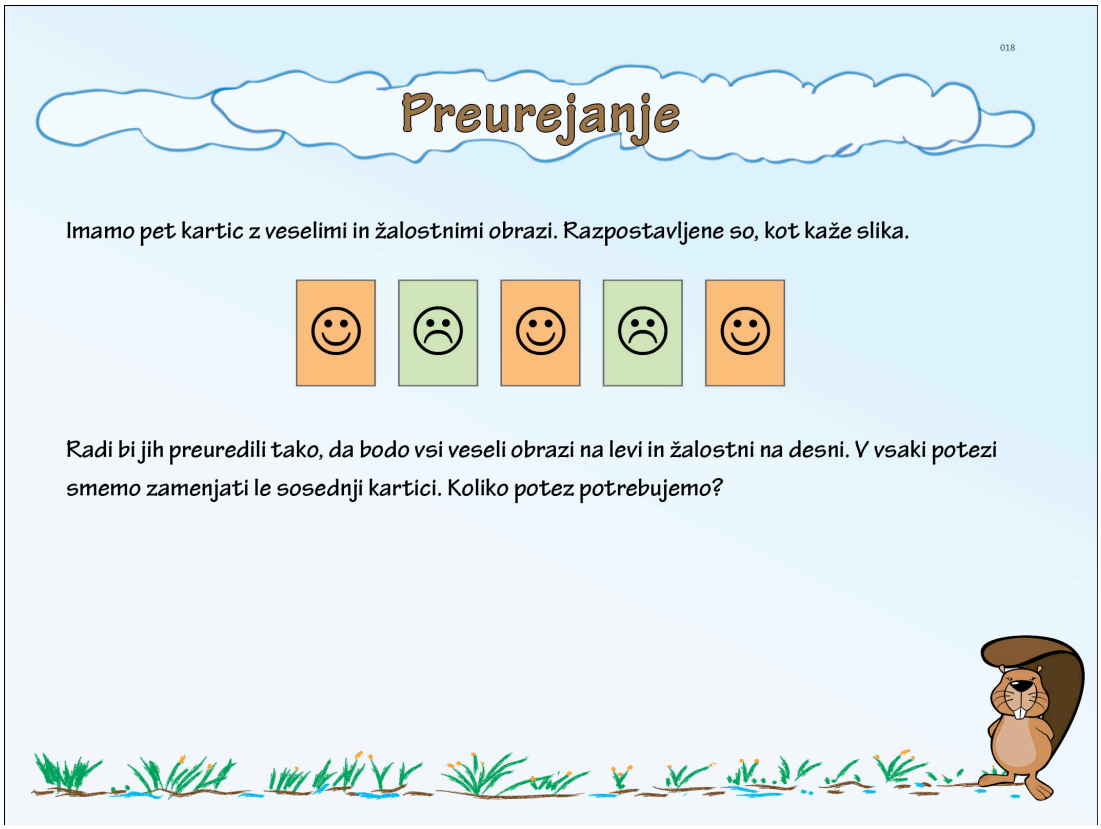
Urejanje

Algoritmi urejanja so priljubljen poligon za študij algoritmov. Po eni strani so zelo uporabni: računalniki pogosto urejajo reči po velikosti, abecedi ali kako drugače. Urejanje ni potrebno le v uporabniških vmesnikih – da pokažemo elektronsko pošto urejeno po prejemnikih, datumih ali naslovih – temveč tudi znotraj programov. Z urejanjem namreč dosežejo, da hitreje najdejo, kar iščejo; če programi ne bi interno zlagali reči na pametne načine, bi bili veliko počasnejši.

Po drugi strani so algoritmi urejanja dober primer za analiziranje "časovne zahtevnosti" algoritmov: problem je relativno preprost, predpostavke in cilji so jasne, operaciji sta le dve ("primerjaj" in "zamenjaj"), zato preživijo študenti ob analizi teh algoritmov veliko časa. Obenem je lahko analiza hitrosti delovanja teh algoritmov povsem razumljiva tudi otrokom.

S perspektive poučevanja računalniškega razmišljanja za otroke so algoritmi urejanja uporabni, ker so dovolj preprosti, da jih lahko otroci odkrijejo sami. Algoritmom urejanja sta posvečeni dve obsežni aktivnosti v okviru Vidre. Na Bobru se pojavljajo le delčki problema. Tu bomo videli nekaj nalog, nato pa spoznali nekaj algoritmov urejanja in videli, kako te naloge sodijo v širšo zgodbo.

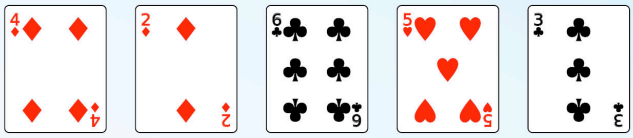
Urejanje z mehurčki



085

Urejanje kart

Spodnje karte bi rad uredil po velikosti. Pri urejanju boš vedno zamenjaval le sosednje pare kart; na začetku, na primer, lahko zamenjaš 2 in 6, ne pa, recimo, 2 in 5. Koliko zamenjav boš potreboval?



089

Postopek urejanja

Bober Donald ima nenavaden način urejanja števil po velikosti. Recimo, da mora urediti zaporedje 5, 4, 7, 2, 0, 3, 6, 1. Urejanje poteka po korakih. V prvih štirih korakih se vrstni red spreminja takole:

1. 5, 4, 7, 2, 0, 3, 6, 1
2. 4, 5, 2, 0, 3, 6, 1, 7
3. 4, 2, 0, 3, 5, 1, 6, 7
4. 2, 0, 3, 4, 1, 5, 6, 7

Kako je videti po naslednjem koraku?

- × 0, 2, 3, 1, 4, 5, 6, 7
- × 0, 1, 2, 3, 4, 5, 6, 7
- × 0, 2, 3, 4, 1, 5, 6, 7
- × 0, 2, 1, 3, 4, 5, 6, 7

Vse tri naloge so povezane z enim najpreprostejših postopkov urejanja, urejanjem z mehurčki. Kako deluje, si oglejmo kar z opisom aktivnosti z Vidre, saj ga lahko v tej obliki preprosto predstavimo tudi učencem.

1. *Izberi osem učencev, na katerih boš pokazal postopek. Postavi jih v vrsto in jim okrog vratu obesi številke v pomešanem vrstnem redu.*
2. *Razloži, da bo urejanje z mehurčki nekoliko drugačno od postopkov, ki smo jih videli doslej. Zahteva namreč več prehodov prek vrste. Učenci si bodo podajali palico: v vsakem trenutku bo palico držal en par učencev. Učenca v paru bosta primerjala svoji številki in če stojita v napačnem vrstnem redu, se zamenjata. Nato palico drži naslednji par.*
   Na primer, da so učenci razporejeni, kot kaže slika. Palico drži prvi par.

   50––20 60 30 10 80 40 70

   Ker sta obrnjena narobe, se zamenjata.

   20––50 60 30 10 80 40 70

   Nato dobi palico naslednji par.

   20 50––60 30 10 80 40 70

   Par je obrnjen pravilno, zato se ne zamenja, temveč le poda palico naslednjemu paru.

   20 50 60––30 10 80 40 70

   Ker je 60 večje od 30, se morata učenca zamenjati.

   20 50 30––60 10 80 40 70

   Nato podata palico naslednjemu paru.

   20 50 30 60––10 80 40 70

   Tako nadaljujemo. Ko pride palica do konca, je razpored takšen

   20 50 30 10 60 40 70––80

   Zadnji učenec (v gornjem primeru ta, ki ima številko 80) stopi korak vstran.

   20 50 30 10 60 40 70 80

   S tem smo končali prvi krog.

3. *Palico vrnemo prvemu paru in ponovimo vse skupaj, vendar brez zadnjega učenca – onega, ki je stopil vstran. Pride palica do konca (torej do predzadnjega učenca), stopi še ta vstran. Po drugem krogu je stanje takšno:*

20 30 10 50 40 60 70 80

4. *Palico spet dobi prvi par. Izvedemo tretji krog, ki nas pripelje do*

20 10 30 40 50 60 70 80

5. *Po četrtem krogu dobimo tole.*

10 20 30 40 50 60 70 80

Opomba: Računalnik v resnici še ne bi vedel, da je vrsta že urejena, zato bi izvedel še peti krog, v katerem pa bi opazil, da ni potrebna nobena zamenjava več in iz tega sklepal, da lahko konča z delom. Učencev s tem ni potrebno obremenjevati.

Bo rezultat algoritma vedno urejen seznam? Bo. Po prvem krogu, ko prida palica do konca, bo učenec z največjo številko prav gotovo stal na koncu vrste. Ko namreč enkrat dobi v roko palico, je ne bo več izpustil, temveč bo potoval z njo do konca. V drugem krogu bo učenec z drugo največjo številko prinesel palico do predzadnjega mesta... in tako naprej.

V tretji od gornjih nalog je uporabljen natančno takšen postopek urejanja. Naloga morda ni najbolj posrečena, saj od učencev zahteva, da postopek že poznajo in ga prepoznajo – najlažje ga prepoznamo po tem, da največja številka "roma" na desno. Če postopka še niso videli, jim ne bo lahko uganiti, kako deluje...

Prvi dve nalogi sprašujeta po nečem pomembnejšem: koliko zamenjav je potrebnih? Računalnikarje navadno zanima čas izvajanja algoritmov (ne le urejevalnih, temveč tudi drugih) v najslabšem primeru. (Zakaj v najslabšem? Morda zato, ker nas zanima, kaj nas bo čakalo v najslabšem primeru, morda pa zato, ker je iskanje najslabšega scenarija neprimerno preprosteje od izračuna poprečnega scenarija, ki si ga privoščimo le redko.)

Kaj je najslabše, kar se nam lahko zgodi pri urejanju z mehurčki? Zgodi se lahko, da bomo potrebovali le en krog manj, kot je številk. Torej, recimo, sedem krogov za osem številk. Tudi, kdaj nas to doleti, je preprosto videti: kadar je seznam obrnjen ravno narobe, 80 70 60 50 40 30 20 10. V prvem koraku bo šla 80 na desno, vse ostale številke ostanejo v takšnem vrstnem redu, kot so bile. V drugem krogu gre na desno 70, vse ostale številke spet obdržijo svoj vrstni red... in tako naprej do predzadnjega kroga, ko dobimo 20 10 30 40 50 60 70 80, da potem v zadnjem zamenjamo še 20 in 10.

Koliko zamenjav potrebujemo za to? Ko je 80 potovala na desno, je bilo za to potrebnih 7 zamenjav. Pri 70 jih je bilo potrebnih 6. Pri 60 5 ... in pri 20 1. Skupaj je to 7 + 6 + 5 + 4 + 3 + 2 + 1 = 28 zamenjav.

Za urejanje n števil po tej metodi bi potrebovali $n \times (n - 1) / 2$ zamenjav. Na Vidri je predstavljen način, kako to formulo izpeljati tudi za mlajše otroke; namesto "Gaussove" lahko uporabimo tudi drugo, jasnejšo metodo.

Za računalnikarje je $n \times (n - 1) / 2$ isto kot n^2. Razmišljamo namreč takole. $n \times (n - 1) / 2 = (n^2 - n) / 2$. Čim so številke dovolj velike, je n^2 veliko veliko večji od n, torej je $n^2 - n$ komaj kaj manj kot n^2. Ergo, $n \times (n - 1) / 2 = n^2 / 2$. Nato pa odmislimo še polovico, takole: nekateri računalniki so hitrejši, drugi počasnejši. Kar mojemu računalniku

vzame osem sekund, vzame tvojemu morda le štiri. Drugo leto pa si bom kupil štirikrat hitrejši računalnik in isti postopek bo trajal dve sekundi. Zato me bolj zanima, kaj se zgodi, če moram urejati dvakrat, trikrat ali petkrat daljše vrste števil: kolikokrat več časa bo potreboval algoritem? Zato polovico preprosto ignoriram: najsi pišem n^2 ali $n^2 / 2$, v obeh primerih bo postopek za dvakrat več števil potreboval štirikrat toliko menjav, za trikrat več devetkrat toliko in za petkrat več petindvajsetkrat toliko.

Računalnikarji rečemo, da ima algoritem kvadratno časovno zahtevnost, kar zapišemo kot $O(n^2)$. Tisti O pomeni, da smo zanemarili vse nepomembne člene (v gornjem primeru –n) in koeficiente (polovica). Za vsem skupaj je seveda še nekaj matematike (limite, konvergiranje, asimptote in podobne grozote), s katerimi pa tule raje ne strašimo.

Pri analizi algoritmov urejanja navadno ne štejemo zamenjav temveč primerjave. Pri urejanju z mehurčki razlika ni preveč pomembna, že ob naslednjem pa bomo lažje razmišljali o primerjavah.

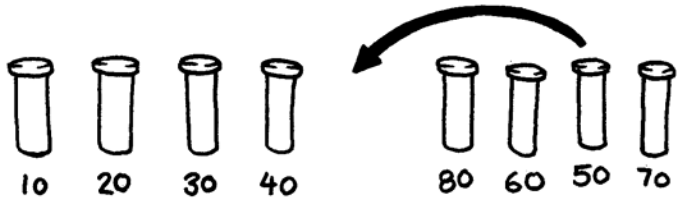
Urejanje z izbiranjem

Urejanje z izbiranjem se (kakor je videti) na Bobru še ni pojavilo, vseeno pa je prav, da vemo zanj. Gre namreč za podobno preprost postopek, ki je, poleg tega, ravno postopek, ki ga otroci tipično odkrijejo, če so prepuščeni sami sebi.

Spet poglejmo opis z Vidre, kjer otroci aktivnost izvajajo s škatlicami (na primer različno polnimi plastenkami jogurta) in preprosto tehtnico narejeno iz deščice.

Najprej poiščemo najtežjo *škatlico, tako da na tehtnico postavimo par škatlic. Odstranimo lažjo, težjo pa primerjamo z naslednjo škatlico. Spet odstranimo lažjo in vzamemo naslednjo. To ponavljamo, dokler ne preverimo vseh škatlic in ta, ki ostane, je najtežja. Postavimo jo na desno stran vrste (to je, vrste v nastajanju ;).*

Celoten postopek ponovimo na preostalih škatlicah. Ko najdemo najtežjo med njimi (torej: drugo najtežjo), jo postavimo levo od prve škatlico. Postopek spet ponovimo z ostalimi, dokler ne uredimo vseh škatlic.



Spet razmislimo ali deluje in koliko časa potrebuje.

Da deluje, je spet očitno. V vsakem koraku gotovo poiščemo najtežjo škatlico. Najtežja škatlica bo tako gotovo končala čisto na desni, druga najtežja gotovo čisto na desni od vseh ostalih in tako naprej... Rezultat mora biti urejen, drugače ne more biti.

Koliko primerjav potrebujemo? Ravno toliko kot pri mehurčkih. Ko iščemo najtežjo škatlico od osmih, bomo potrebovali sedem primerjav. Ko iščemo najtežjo od sedmih, jih potrebujemo šest ... in tako naprej. Tudi pri urejanju z izbiranjem število primerjav narašča s kvadratom števila škatlic (otrok, številk ali česarkoli že).

Hitro urejanje

Naloga spet ni posebej posrečena, saj kaže le košček algoritma, ne pa njegovega bistva. Od učencev zahteva bolj, da znajo slediti navodilom (in to ritensko), ničesar pa ne pove o algoritmu urejanja, ki se skriva za njimi.

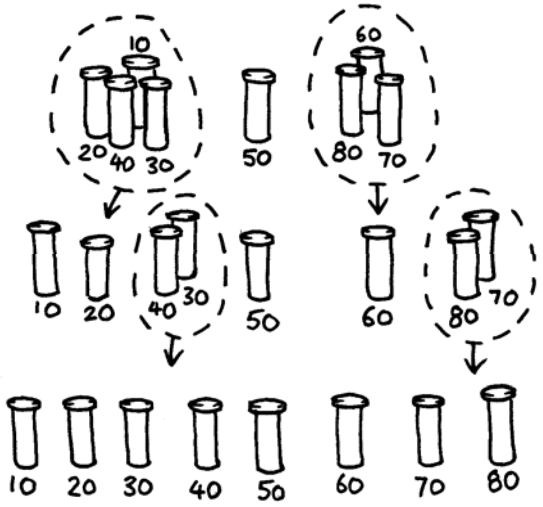
Skupaj z gornjima algoritmoma, urejanjem z mehurčki in urejanjem z izbiranjem, se pogosto predstavlja še algoritem urejanja z vstavljanjem. Tu ga bomo preskočili; dovolj je bilo. Vsem trem je skupno, da imajo kvadratno časovno zahtevnost, kar nam ni preveč všeč. A kakorkoli bi si izmišljali svoje algoritme, skoraj gotovo bi pridelali nekaj, kar ima kvadratno zahtevnost (in kaj verjetno bi se domislili le enega od običajnih treh, a v kaki preobleki).

Algoritem, o katerem (zelo prikrito) govori naloga, se imenuje hitro urejanje. To pa zato, ker je hitro. Spet si ga oglejmo z Vidro.

1. *Določi otroka, ki bo pod tvojim vodstvom urejal škatle in otroka, ki bo beležil število tehtanj.*
2. *Naključno izberi eno škatlico in jo postavi na levo stran tehtnice.*

3. *Vse škatlice primerjaj z izbrano, tako da jih eno za drugo postavljaš na desno stran tehtnice. Škatlice odlagaj na dva kupa: na levega dajaj tiste, ki so lažje in na desnega tiste, ki so težje od izbrane škatlice.*
4. *Ko si primerjal vse škatlice z izbrano, jo postavi na sredo med kupa.*

Na spodnji sliki smo si izbrali škatlico s težo 50. Lažje škatlice (10, 20, 40, 30) so na levi, težje (80, 60, 70) na desni, izbrana pa je v sredini.



Če imamo smolo, se bo pripetilo, da bo na eni strani veliko več škatlic kot na drugi; lahko se zgodi celo, da bodo vse na isti strani, ker si si izbral ravno najtežjo ali najlažjo. Nič ne de, bomo preživeli. Če škatlice niso enake, pa si zapomni, katera je približno na sredi po teži in poskrbi, da si bo otrok izbral to škatlo (lahko mu jo podaš, češ, "izberi si eno škatlo, recimo tole").

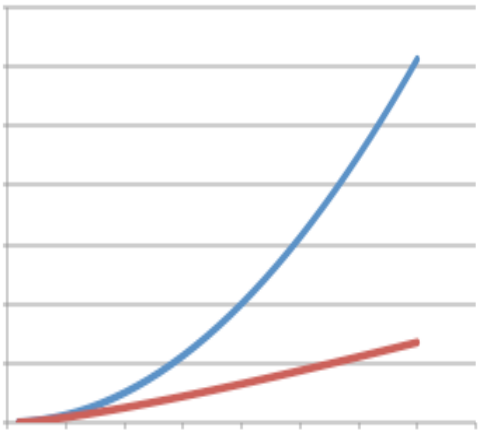
5. *Razloži otrokom, da so škatle nekako napol urejene: tista na sredi je že tam, kjer mora biti, zdaj pa moramo urediti še oba kupa. Lotili se bomo vsakega posebej. Torej:*
    a. *Med škatlicami na levi naključno izberi eno škatlico in razdeli ostale na tiste, ki so lažje in tiste, ki so težje, tako kot si to storil prej. Spet boš dobil škatlo na sredini in dva kupa, ki ju bo potrebno urediti. Lotiš se, spet, vsakega posebej...*
    b. *V desni skupini stori isto.*
6. *Kupe drobiš, dokler ne dobiš skupin z eno samo škatlico, kjer ni več kaj urejati. In, glej, škatlice so urejene!*

Gre za primer rekurzivnega algoritma: algoritem deluje tako, da razdeli problem na dva podproblema, na katerih je potrebno ponovno uporabiti prav taisti algoritem (ki bo vsakega od njiju razdelil na dva podproblema, na katerih je potrebno ponovno uporabiti prav taisti algoritem (ki bo vsakega ... in tako naprej)). Rekurzija je strah in trepet brucov, ki se učijo programiranja; kakor videvamo pri poučevanju v drugi triadi OŠ (in že pri mlajših!), pa je rekurzija v resnici tako naravna, da se otroci petega koraka zgornjih navodil, domislijo kar sami, ko jih vprašamo, kaj bomo naredili z levo in desno skupino škatlic.

Ali algoritem deluje pravilno? Da, in to tako očitno, da spet ne bomo ničesar dokazovali.

Koliko primerjav potrebuje? Tu pa postanejo stvari zanimive. Najhujše, kar se nam lahko zgodi, je, da najprej izberemo ravno najtežji (ali najlažji) element. Primerjali ga bomo z vsemi ostalimi in dobili en prazen kup in drugega, na katerem bodo vse škatle razen (ponesrečeno) izbrane. Nato se nam smola ponovi, tudi med ostalimi izberemo najtežjega (ali najlažjega). In spet in spet. Na koncu bomo potrebovali natančno toliko primerjav kot pri urejanju z izbiranjem.

Zakaj pa se potem algoritem ponaša z imenom "hitro urejanje"? Zato, ker je v poprečju najhitrejše, kar moremo narediti v okviru pravil (ki jih še ne poznamo, a jih bomo spoznali vsak čas). V poprečju potrebuje algoritem število primerjanj, ki je sorazmerno $n \ln n$. Slika kaže, kako s številom elementov naraščata funkciji n^2 in $n \ln n$.



Goljufamo? Primerjamo *poprečni* čas, ki ga potrebuje hitro urejanje, z *najslabšim* časom, ki ga potrebujejo metode, ki smo jih spoznali prej? Ne, izkaže se, da one, počasne metode, tudi v poprečju potrebujejo čas sorazmeren kvadratu števila elementov.

Pravila igre

Gre še hitreje? Si je mogoče izmisliti postopek, ki bo vedno, tudi v najslabšem primeru, potreboval število primerjav, ki bo sorazmerno $n \ln n$? Da. Takšno je, na primer, urejanje z zlivanjem.

Pa še hitreje? Postopek, ki vedno potrebuje manj kot $n \ln n$ primerjav?

To pa je odvisno od pravil igre. Običajno so takšna: števila so napisana v tabeli (ali, fizično, otroci, ki nosijo različne številke ali različno težke škatle, so postavljene na poljih). Edini operaciji, ki sta dovoljeni, sta primerjava dveh števil in zamenjava dveh elementov. Prepovedano je primerjati več števil hkrati in prepovedano je odlagati števila (elemente, škatle, otroke) v novo tabelo (ali na nova, dodatna polja). Algoritem ne sme uporabljati nobenega dodatnega prostora. Če se dogovorimo za takšna pravila, je kar preprosto dokazati, da je nemogoče razviti postopek, ki bi vedno potreboval manj kot $n \ln n$ primerjav.

Čemu ta omejujoča pravila? Prvo, da smemo primerjati le po dve števili naenkrat, izvira iz načina, na katerega so narejeni (praktično vsi) današnji računalniki. Ena od osnovnih operacij, ki jih zna izvesti "glava" računalnika, procesor, je primerjava dveh števil. Operacije, kakršno je "iskanje največjega števila med poljubno mnogimi", niso možne že zato, ker računalniški pomnilniki ne delujejo na način, ki bi to omogočal.

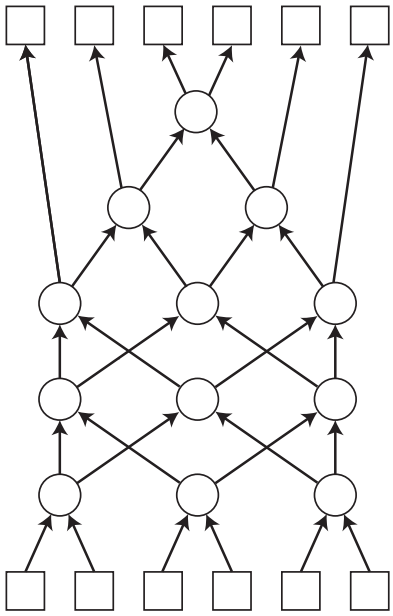
Druga omejitev izvira zgolj iz škrtosti: nočemo, da bi algoritem zapravil preveč pomnilnika. Če mu pustimo dodatni pomnilnik, lahko naredimo postopke, ki delujejo v linearnem času: dvakrat več elementov bi pomenilo le dvakrat daljši čas urejanja. Hitreje pa ne gre, saj moramo vsak element vsaj enkrat pogledati in že to nam prinese število operacij, ki je sorazmerno številu elementov.

Vzporedno urejanje

Naloga kaže postopek za vzporedno urejanje s pomočjo mreže. Kako deluje algoritem, je očitno, saj ga opisuje že naloga. Mreža iz naloge zna urediti štiri elemente. Za to sicer potrebuje pet primerjav, vendar se nekatere izvajajo vzporedno. Hitrost algoritma je takšna, kot da bi imeli le tri primerjave, vendar za to potrebujemo dve "glavi", računalnik z dvema procesorjema oz. jedroma.

Tovrstni postopki – in predvsem, kako jih predstaviti otrokom v telovadnici ali na parkirišču – je predmet posebne aktivnosti na Vidri.

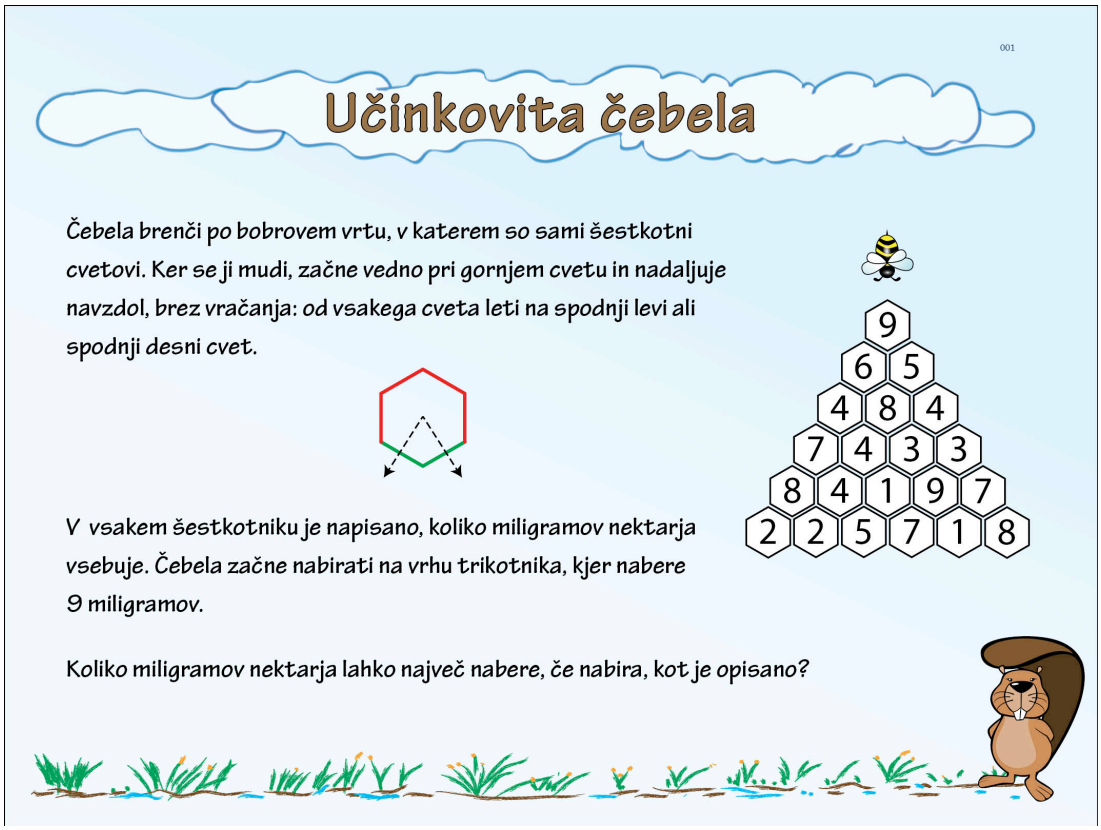
Podobne mreže lahko sestavimo tudi za večje število elementov. Tule je takšna za šest:



Hitra je tako, kot da bi imeli le pet primerjav, zahteva pa tri "glave". Kako sestavljati takšne mreže za res velike tabele, je aktivno področje raziskovanja. V praksi je dolžina mreže navadno sorazmerna številu elementov, njena širina ("število glav") pa je polovica števila elementov.

Pa imamo računalnike z "več glavami"? Tipični računalniki, ki jih kupujemo danes, imajo štiri jedra, kar pomeni, da lahko mislijo štiri misli naenkrat (to je, recimo, primerjajo štiri pare števil). Za takšne postopke urejanja to ni dovolj. Pač pa imajo dandanašnji računalniki precej zmogljive grafične kartice, ki imajo lahko tudi nekaj tisoč procesorjev. Ti so sicer povezani tako, da vsi hkrati izvajajo isto operacijo – a to je točno to, kar potrebujemo za tovrstne algoritme. V času pisanja tega besedila se grafične kartice, poleg očitnega namena, zagotavljanja čimbolj realistične grafike v igricah, uporabljajo za vzporedno procesiranje predvsem v raziskovalne namene. V času branja tega besedila pa utegne biti že drugače.

Dinamično programiranje



Dinamično programiranje ni algoritem, temveč način snovanja algoritmov. Preden ga opišemo, povejmo za dva druga.

Pristop *deli in vladaj* rešuje probleme tako, da jih razdeli na podprobleme in se loti vsakega posebej. Primer takšnega postopka je bilo hitro urejanje: namesto, da bi uredili celotno zbirko elementov, jo razdelimo na dva dela in uredimo vsako posebej. Algoritmi sestavljeni po tem vzorcu, imajo tipično tri korake: delitev, izvajanje algoritma na vsakem poddelu (ki spet obsega delitev in tako naprej) in nato združevanje. V primeru hitrega urejanja smo elemente, ki jih je bilo potrebno urediti, razdelili na manjše in večje od nekega izbranega elementa. Algoritem smo ponovili na obeh podmnožicah. Kakega posebnega združevanja pa ni bilo. Kadar delitev in združevanje zahteva čas, sorazmeren številu elementov n, bomo običajno dobili algoritme z zahtevnostjo sorazmerno $n \log n$.

Požrešni algoritmi delujejo tako, da se v vsakem koraku odločijo za trenutno najboljšo izbiro. Recimo, da smo na poti in imamo čarobni kompas, ki vedno kaže proti kraju, v katerega želimo priti. Potujemo lahko tako, da se na vsakem razpotju odločimo za pot, ki gre v najbolj pravo smer. Ali nas bo to res pripeljalo po najkrajši poti do cilja, je odvisno od tega, kako so speljane ceste.

Tudi ko s podobnim pristopom rešujemo računalniške probleme, nas pri nekaterih vrstah problemov to pripelje do najboljše rešitve, v nekaterih da slabšo rešitev od najboljše možne, v nekaterih pa nas sploh ne pripelje do rešitve. Recimo, da imamo veliko število evrskih kovancev. Plačati je potrebno določen znesek, pri čemer želimo uporabiti čim manjše število kovancev. Očitna – in pravilna, optimalna – rešitev je, da najprej odštevamo dvo evrske kovance, ko znesek pade pod dva evra, po potrebi

dodamo kovanec za evro, nato po potrebi za 50 centov, en ali dva kovanca za dvajset centov, po potrebi kovanec za deset centov, za pet centov, enega ali dva za dva centa in po potrebi kovanec za cent.

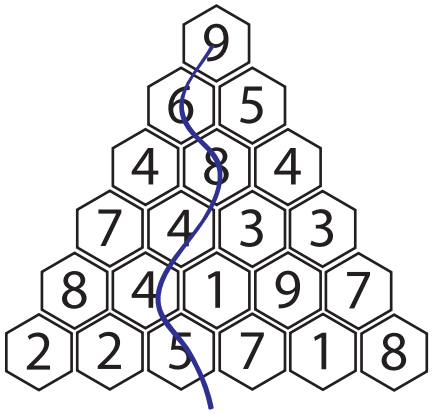
Pri kakem drugačnem sistemu kovancev postopek ne deluje. Če imamo, recimo, kovance za 10, 20, 40 in 50 centov, plačati pa je potrebno 80 centov, bi s požrešno metodo plačali s tremi kovanci (50 + 20 + 10), optimalna rešitev pa zahteva dva (40 + 40).

Še nerodneje nas lahko požrešna metoda zapelje, če imamo kovance za 25, 10 in 4 cente, plačati pa je potrebno 41 centov. Požrešna metoda izbere kovanca za 25 in 10 centov, razlike, 6 centov, pa s kovanci po 4 cente ne more plačati. Tako sploh ne najde rešitve problema – optimalne ali neoptimalne – čeprav ta obstaja (25 + 4 + 4 + 4 + 4).

Požrešne algoritme pogosto uporabljamo kot hevristične algoritme: v primerih, ko bi bilo iskanje optimalne rešitve prepočasno ali kako drugače prezahtevno, smo zadovoljni tudi s slabšo rešitvijo, ki jo najde hiter požrešen algoritem.

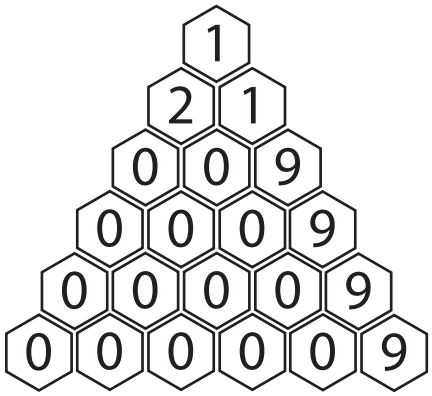
Dinamično programiranje je zanimivejša zadeva. Deluje tako, da odgovor izračuna iz rešitev preprostejših problemov. Učinkovita čebela je odličen primer tega pristopa, zato jo bomo tule temeljito secirali.

Prva – kar takoj povejmo, da ne preveč dobra – ideja, je požrešna: želva v vsakem koraku zavije k cvetu z več medu.



Rezultat je 36 mg medu. Pesimist pomisli, da je to najbrž tudi največ, kar lahko dobimo. Optimist se strinja rekoč, da je požrešna metoda gotovo dobra metoda.

Najprej razočarajmo optimista: požrešna metoda sicer lahko včasih slučajno da najboljšo rešitev, v splošnem pa ne. O tem nas hitro prepriča spodnja hudobna postavitev cvetov.



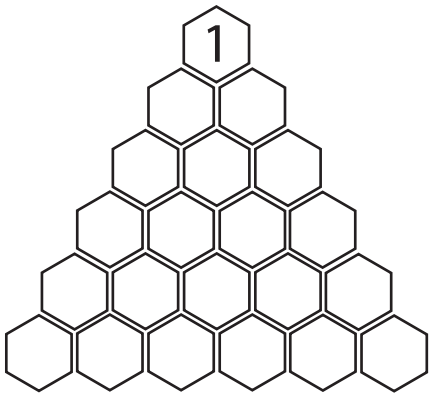
Ko čebela v prvem koraku podleže požrešnosti in odleti na cvet z dvema miligramoma medu, je njena usoda zapečatena: nabrala ne bo ničesar več. Pohlevnost v prvem koraku bi jo vodila prek bogato založenih cvetov na desni.

Tudi otroci, ki rešujejo nalogo s tekmovanja, opazijo, da bi bilo namesto zaključka 4 + 4 + 5 bolj donosno iti po poti 3 + 9 + 7 – ko bi iz cveta z 8 mg zavili na 3 namesto na 4. Tedaj se pojavijo dvomi: je 9 + 6 + 8 + 3 + 8 + 7 največ, kar lahko dobimo, ali pa bi šlo še boljše?

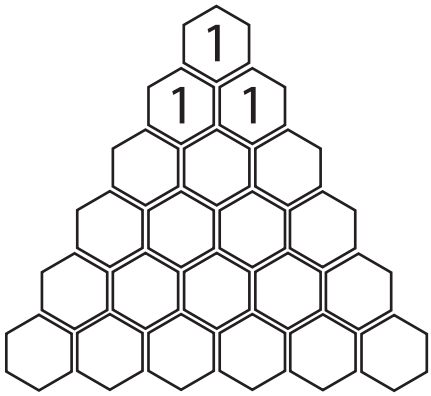
Na prvi pogled je rešitev le ena: preveriti vse možne poti. Nekateri otroci so se res lotili tega podviga; rezultat je odvisen od njihove vztrajnosti in sistematičnosti. Kdor poseduje oboje: ni težav. Nesistematični se bodo izgubili. Nevztrajni pa se bodo sredi dela vprašali: koliko tega me še čaka?

Odgovorimo jim. Na koliko različnih načinov lahko čebela preleti vrt, od gornjega polja do kateregakoli od cvetov v spodnji vrstici? Odgovor na to vprašanje nam sicer ne bo pomagalo rešiti problema (ali pač, speljal nas bo na prave misli), je pa zanimiv, zato ga le poiščimo.

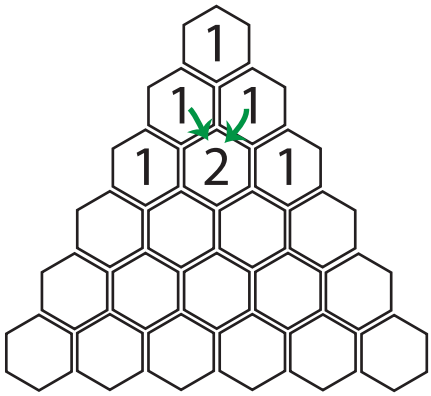
Najprej se vprašajmo nekaj preprostega: na koliko načinov lahko pridemo na prvo polje? Očitno na enega samega – tam pač začnemo.



Na koliko načinov pa pridemo na polji v drugi vrsti? Na vsakega od njiju pridemo na en sam način, namreč s prvega polja.

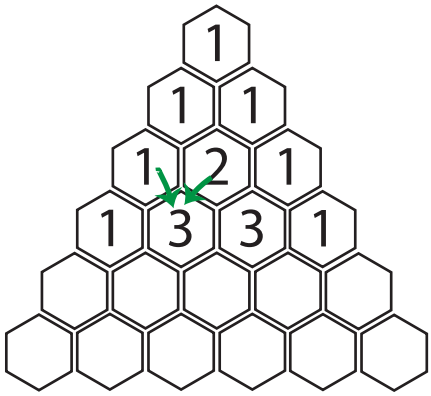


Na koliko načinov pa pridemo do polj v tretji vrsti? Na skrajno levo in skrajno desno polje pridemo na en sam način, iz skrajno levega ali skrajno desnega polja prejšnje vrste. Na srednje pa pač na dva, bodisi z leve bodisi z desne.

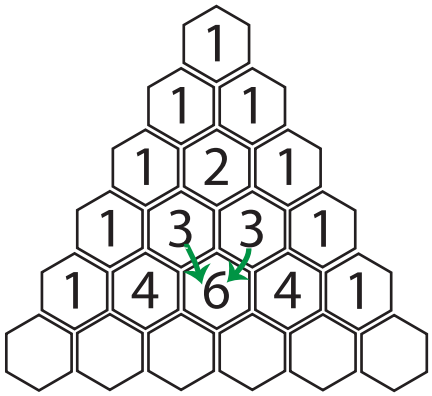


Zdaj pa postane reč zanimivejša, čeprav bo vprašanje enako: na koliko načinov pridemo z začetnega polja do posameznih polj v četrti vrsti? Na skrajni polji še vedno pridemo le iz skrajnih polj. Na drugo polje pa lahko pridemo na tri načine. Takole. Obstaja natančno en način, na katerega pridemo od začetka pa do levega polje tretje vrste; torej obstaja en način, kako priti od začetka prek levega polja tretje vrste do drugega polja četrte. Od začetnega polja do srednjega polja tretje vrste pa pridemo, kot vemo, na dva načina. Torej obstajata dva načina, da pridemo od začetnega polja prek srednjega polja tretje vrste do drugega polja četrte. Skupaj so torej 1 + 2 = 3 načini, da pridemo od začetnega polja do drugega polja četrte vrste.

Tretje polje je podobna zgodba.



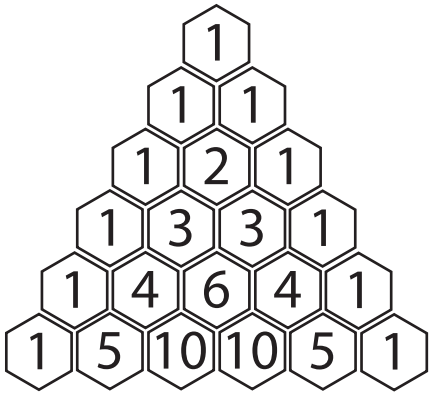
Pa četrta vrsta? Še enkrat ponovimo razmišljanje od prej, a tokrat se osredotočimo na najzanimivejše, srednje polje, tisto, v katerem piše 6.



Vanj lahko pridemo iz drugega ali tretjega polja četrte vrste. Kot že vemo, vodijo natančno tri različne poti od začetnega polja do drugega polja četrte vrste. To pomeni, da obstajajo tri poti od začetnega polja prek drugega polja četrte do srednjega polja pete vrste. Prav tako obstajajo tri poti prek tretjega polja četrte vrste. Skupaj pridemo do srednjega polja pete vrste na 3 + 3 = 6 načinov – tri poti pripeljejo z leve, tri z desne.

Podobno smo naračunali štirico: do drugega polja pete vrste pridemo na en način z leve in na tri načine z desne (ker pač obstajajo tri poti do polja na desni (ali levi, s čebeline perspektive)).

Enako naračunamo še zadnjo vrsto.



Otroci tega ne vedo, mi, ki smo odrasle, zrele osebe, pa vemo, kako se reče tej reči: Pascalov trikotnik.

Zdaj vemo, ena od možnih poti se konča na skrajnem levem polju, pet na drugem, deset na tretjem in tako naprej. Vseh možnih poti je 1 + 5 + 10 + 10 + 5 + 1 = 32.

32? Nekam čudno okrogla številka – za računalnikarja. Hm, če bi imeli eno vrsto manj, bi bilo možnih poti 1 + 4 + 6 + 4 + 1 = 16. Če bi imeli le štiri vrste, bi jih bilo 1 + 3 + 3 + 1 = 8, s tremi jih je 1 + 2 + 1 = 4, z dvema 1 + 1 = 2 in z eno 1 = 1. Nenavadno naključje?

Niti ne. Če vemo, kaj računa Pascalov trikotnik, je jasno, da mora biti tako. Pascalov trikotnik računa binomske koeficiente in z njimi velja:

$$(a+b)^n = \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \binom{n}{2} a^{n-2} b^2 + \cdots \binom{n}{n} a^0 b^n$$

če zvito rečemo $a = b = 1$, izvemo, da je $2^n = \binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n}$.

V našem primeru je *n* enak 5 (prva vrstica ima številko 0), torej imamo $2^5 = 32$ možnih poti.

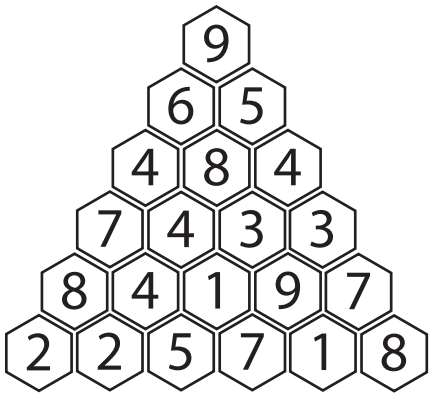
Ah, pa je bil ves ta trud res potreben? Saj gre preprosteje! Čebela se mora petkrat odločiti, kam gre. Vsakič ima dve možnosti. Z nekaj kombinatorike – dovolj preproste, da jo razložimo desetletniku, če debato začnemo z vprašanje, na koliko načinov se lahko obleče, če ima dvoje čevljev, dvoje hlač in dvoje srajc –ugotovimo, da je možnih kombinacij odločitev 2×2×2×2×2=32.

Čemu smo se mučili z vsem tem preštevanjem? Zaradi dveh stvari. Najprej, želimo se prepričati, kako narašča število poti: vsaka dodatna vrstica podvoji število poti. Algoritem, ki pravi "preveri vse možne poti" je neuporabno počasen: čas računanja ne narašča linearno s številom vrstic (dvakrat več vrstic = dvakrat več računanja), niti ne kvadratno s številom vrstic (dvakrat več vrstic = štirikrat več računanja). (Mimogrede se spomnimo, da pri algoritmih urejanja nismo marali tistih s kvadratno časovno zahtevnostjo!) Še več, ne narašča niti s kubom ali četrto potenco števila vrstic: *ena vrstica več = dvakrat več računanja*. Čas izvajanja je sorazmeren 2^n, kjer je *n* število vrstic. Računalnikarji to sorazmerje označimo z $O(2^n)$.

Takšne časovne zahtevnosti so nesprejemljive. Takšnih algoritmov ne maramo, saj delujejo le pri res res res majhnih problemih. Potrebno si bo izmisliti boljšega.

In zdaj pride drugi razlog, zakaj se nam je zdelo koristno preštevati poti: ker bomo na podoben način razmišljali, ko bomo sestavljali algoritem.

Na tem mestu bomo postali nekoliko matemetični, formalni in zateženi. Bralec, ki ga to plaši, lahko ta del mirno preskoči. S seboj povabim druge vas junake: stvar ni tako težka, zato vsaj poskusite. Z ostalimi se ponovno vidimo, ko bomo **začeli razmišljati naprej**.



Najprej se zmenimo za oznake. Polja bomo označevali tako, da bomo zapisali številko vrstice in nato številko polja. Tretje polje v šesti vrstici zapišemo s "koordinatami" (6, 3). Zanimalo nas bo, koliko medu lahko čebela največ nabere na poti do nekega polja (a, b); to število bomo označili kot $m(a, b)$. Če rečemo, recimo, m(3, 1) = 23, bo to pomenilo,

da lahko čebela na poti do (vključno) polja (3, 1) (srednje polje v tretji vrsti) nabere največ 23 mg medu.

Da bomo lažje napisali kakšno splošno formulo, recimo še, da s $q(a, b)$ označimo količino medu v polju (a, b). Tako je $q(6, 3) = 5$.

Za začetek problem rešujmo narobe, ritensko. Izberimo si neko polje, recimo tretje polje v zadnji, šesti vrsti in se vprašajmo, koliko, največ, medu lahko nabere čebela, če bo pot končala v njem, torej, koliko je m(6, 3). Odgovora sicer ne vemo, vemo pa, da čebela pride v (6, 3) bodisi s polja (5, 2) bodisi s (5, 3). Ker jo v (6, 3) čaka 5 mg medu, bo $m(6, 3)$ enak bodisi $m(5, 2) + 5$ bodisi $m(5, 3) + 5$ – toliko, kolikor nabere do enega od teh dveh polj in še 5 zraven.

Zdaj pa pazimo, $m(6, 3)$ nista dve številki, temveč ena sama: $m(6, 3)$ pove, koliko največ medu lahko čebela nabere do (vključno) polja (6, 3). Odgovor na to bi bil lahko preprost: če je m(5, 2) večje od m(5, 3), bo šla čebela, ki hoče na vsak način končati pot v polju (6, 3), raje prek (5, 2), saj bo tako nabrala več. Če je m(5, 3) večje od m(5, 2), pa bo šla do (6, 3) raje prek (5, 3). Nabrala bo torej toliko, kolikor dobi v enem ali drugem od teh dveh polj in še 5 zraven, torej $m(6, 3) = \max(m(5, 2), m(5, 3)) + 5$.

Žal pa smo mogli napisati le "bi bil lahko preprost", to pa zato, ker se nam niti ne sanja, koliko bi utegnila biti m(5, 2) in m(5, 3). A tudi na to vprašanje vemo poiskati odgovor: do (5, 2) pridemo bodisi iz (4, 1) bodisi iz (4, 2) – pač od ondod, od koder se bolj splača. V (5, 2) so 4 mg medu, zato $m(5, 2) = \max(m(4, 1), m(4, 2)) + 4$. A tu spet naletimo na isti problem: koliko medu lahko naberemo do (4, 1) in koliko do (4, 2)? Tudi ta problem rešimo na enak način.

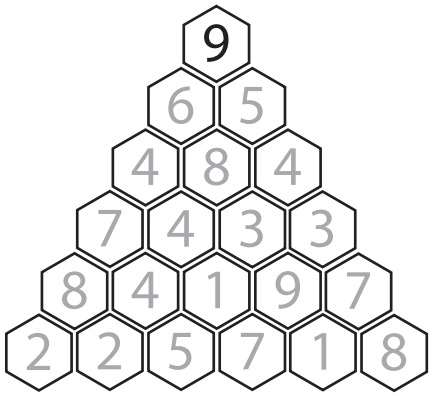
V splošnem velja: $m(a, b) = \max(m(a - 1, b - 1), m(a - 1, b)) + q(a, b)$, oziroma, na robovih, $m(a, 0) = m(a - 1, 0) + q(a, 0)$ ter $m(a, a) = m(a - 1, a - 1) + q(a, a)$.

Problem ritenskega razmišljanja je tule: ko bomo računali, koliko medu lahko naberemo do (5, 2), bomo morali izračunati, koliko ga je mogoče nabrati do (4, 2). Nekoliko kasneje se bomo vprašali, koliko medu je mogoče nabrati do (5, 3), zapisali bomo $m(5, 3) = \max(m(4, 2), m(4, 3)) + 1$ ... in ponovno računali, koliko medu je potrebno nabrati do (4, 2)!

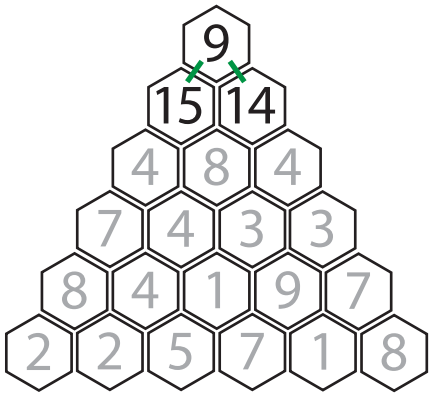
Z malo nespretnosti bomo prišli do algoritma, za katerega z malo spretnosti izračunamo, da bo potrebovali nebodijihtreba 2^n računanj. Temu se izognemo tako, da si vse, kar smo že enkrat izračunali, zapomnimo. Pri računanju nazaj je to nerodno.

Zato raje razmišljajmo naprej. (Mastni tisk je uporabljen, da pritegne vse matematikofobne bralce, za katere je besedilo od tega mesta naprej spet varno.) Razmišljanje naprej bo nenavadno podobno načinu, na katerega smo ravno prejle zabredli v računanje Pascalovega trikotnika.

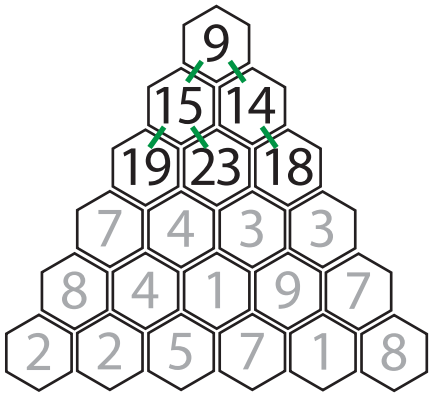
Na poti do (vključno) prvega polja čebela vedno nabere 9 mg medu.



Tudi polji v drugi vrstici ne zahtevata posebnega razmišljanja: do levega nabere 9 + 6 = 15 mg in do desnega 9 + 5 = 14 mg. Številki zapišimo kar v polje, namesto (ali prek, če rešujemo na papir) številk s količino medu.

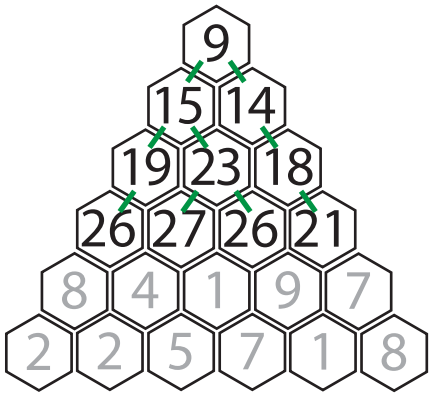


Tretja vrstica je malenkost zanimivejša:

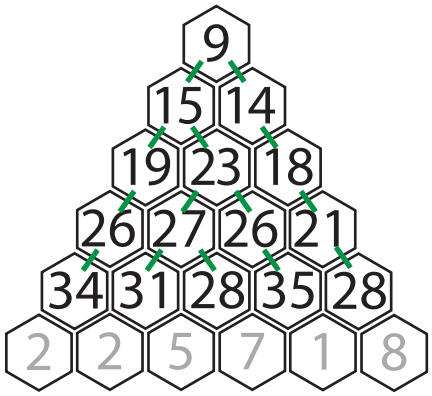


Stranski polji sta trivialni, do srednjega pa se bolj splača priti z leve, s 15. Na ta način čebela nabere 15 + 8 = 23 mg medu; če bi prišla z druge strani, bi ga le 14 + 8 = 22 mg. V polje napišemo 23 in zabeležimo, s katere strani je potrebno prileteti vanj.

V četrti vrstici skrajni polji, kot vedno, nimata dilem. V srednji dve se najbolj splača prileteti s srednjega polja tretje vrstice, saj čebela tako nabere 23 + 4 = 27 in 23 + 3 = 26 mg medu. Številki napišemo v polji in narišemo črtici, s katero označimo, odkod je potrebno priti nanju.

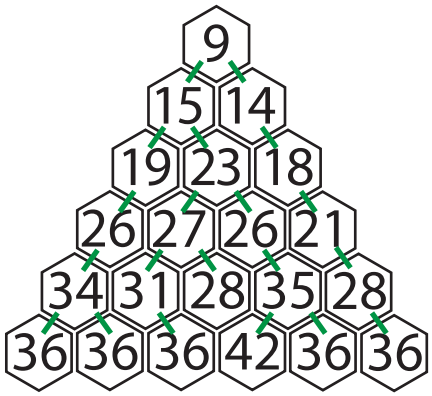


Zdaj preračunamo peto vrstico.

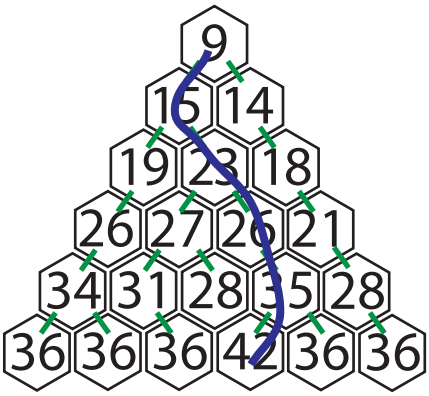


Prvo polje je jasno. V drugega pridemo z desne, s polja s številko 27 (ker je pač 27 več kot 26). Tako čebela namere 27 + 4 = 31 mg, kar napišemo v polje in dodamo črtico. Tudi v tretje polje pride z istega polja iz četrte vrstice; nabrala bo 27 + 1 = 28 mg medu, kar vpišemo in dodamo črtico. V četrto polje prileti z leve (ker je 26 več kot 21), pri čemer dobi 26 + 9 = 35 mg medu; vpišemo in narišemo črtico. Skrajno desno polje je spet trivialno.

Na koncu na enak način izračunamo še zadnjo vrstico.



Če se čebela odloči končati na četrtem cvetu zadnje vrste, lahko nabere (največ) 42 mg medu; če kjerkoli drugje, le 36. Kje pa mora iti, da nabere toliko? Zdaj sledimo poti nazaj: v ta cvet mora priti z desne (35), tja z leve (26), vanj z leve (23) in vanje spet z leve (15), vanj pa, jasno, s prvega cveta.



Koliko računov je potrebno opraviti za n vrstic? Toliko, kolikor je v n vrsticah polj. To pa je 1 + 2 + 3 + 4 + ... + n, kar je, kot že vemo, sorazmerno n^2. Ob urejanju smo nad časovnimi zahtevnostmi, ki so bile sorazmerne kvadratu števila elementov, godrnjali. Tu je to najboljše, na kar moremo upati: vsako polje moramo pač nujno pogledati vsaj enkrat, ne?

Splošno o dinamičnem programiranju

Dinamično programiranje je učinkovit in popularen pristop k snovanju algoritmov. Tule smo si izmislili algoritem za računanje optimalne poti čebele: algoritem je splošen in bi dal pravilen rezultat tudi, če bi bila količina medu v cvetovih drugačna.

Dinamično programiranje je praktično vedno, kadar lahko rešitev naračunamo iz rešitev istega problema, ki so v nekem smislu enostavnejše. Tule "enostavnost" pomeni krajšo pot: kakšna je optimalna pot do nekega cveta izvemo, če poznamo eno ali dve krajši optimalni poti. Tudi tidve naračunamo iz še krajših poti ... dokler ne pridemo do ene same poti, kjer ni več kaj razmišljati (v našem primeru do začetnega polja).

Ali, če razmišljamo naprej: zanima nas rešitev določenega problema (najboljše poti do določenega polja). Namesto, da bi rešili ta problem, rešujemo preprostejše probleme (vedno daljše poti od začetnega polja) in tako "širimo fronto" svojega znanja, dokler ne pridemo do rešitve, ki jo dejansko iščemo (količine nabranega medu v vseh končnih poljih).

Pri reševanju te naloge smo računali polja od spodaj navzgor. Fronto bi lahko peljali tudi drugače, na primer z leve proti desni – najprej bi izračunali maksimalno količino medu do vseh skrajnih desnih polj, nato do vseh drugih polj v vrsticah, pa do tretjih, četrtih... Lahko bi bili celo povsem nesistematičnih, paziti bi morali le, da za vsako polje, ki se ga lotimo računati, že poznamo količino medu v poljih nad njim.

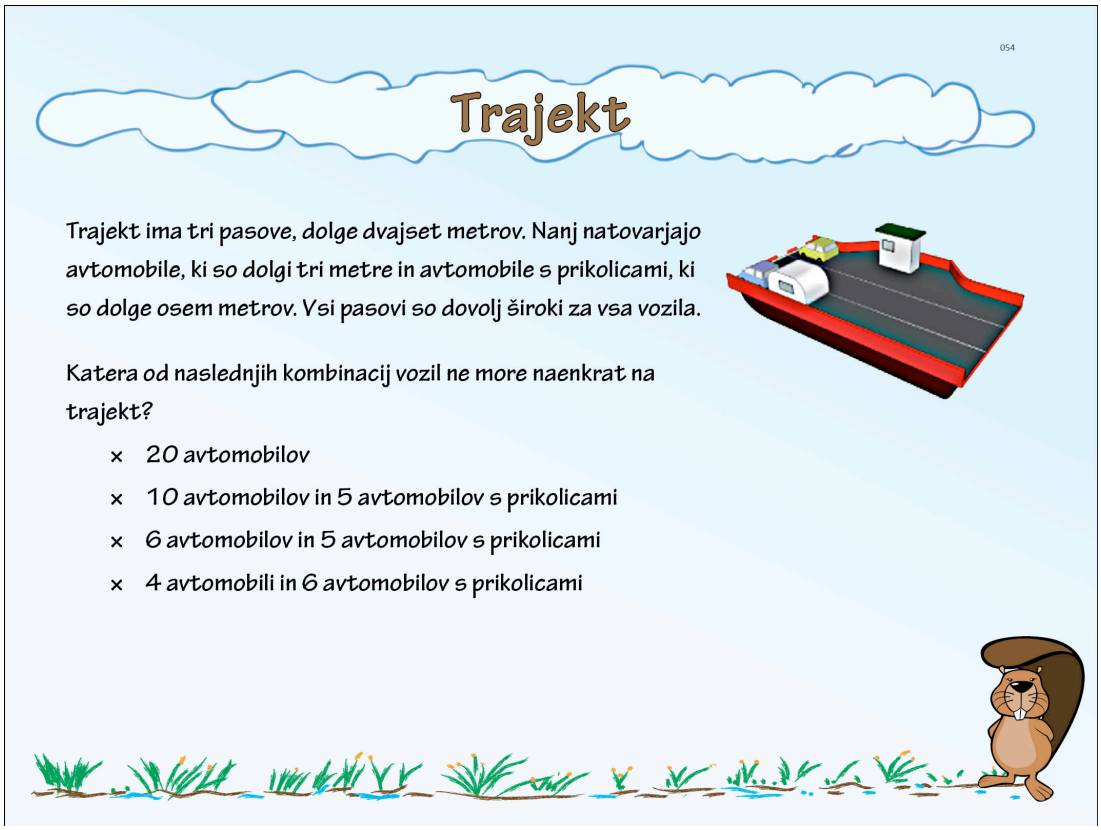
Še več, v to smo včasih prisiljeni. V nalogi s čebelo so bili cvetovi lepo zloženi v trikotnik. Dinamično programiranje pogosto uporabljamo tudi v problemih, ki nimajo tako lepe, jasne, prostorske ponazoritve. Tam niti ne moremo govoriti o "od zgoraj navzdol" ali "z leve na desno", temveč pazimo le, da gremo po vrsti v tem smislu, da vedno poznamo vse, kar je potrebno poznati za izračun vrednosti funkcije pri določenih vrednostih.

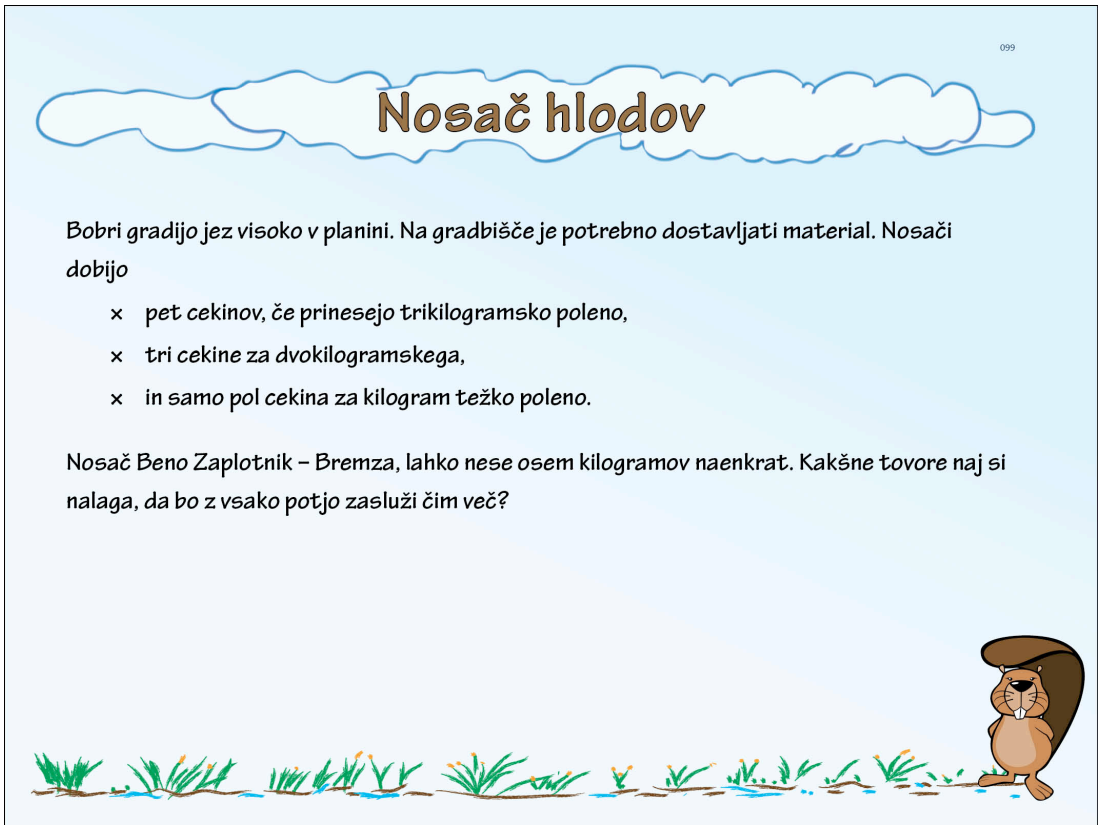
Dinamično programiranje se obnese predvsem v problemih, v katerih bi bili sicer prisiljeni večkrat računati eno in isto vrednost funkcije. Ob razmišljanju o čebeli smo

tako opazili, da bi z malo nespretnosti dvakrat računali vrednost v polju (4, 2). Dinamično programiranje nas tega reši.

V kontekstu bobra še svarilo: algoritme, sestavljene s pristopom dinamičnega programiranja (ob grafih bomo namreč spoznali še enega podobnega), je težko izvajati ročno. Naloge na tekmovanjih je zato preprosteje in varneje reševati z drugo metodo, metodo ostrega pogleda, ki pravi, da *bulji, dokler ne vidiš*. Nič pa ni narobe, če otrokom, predvsem starejšim ali bistrejšim, telovadimo možgane tudi s sistematičnim reševanjem, kakršnega smo opisali tule.

Polnjenje nahrbnika





Nalogi opisujeta enega klasičnih problemov s področja algoritmov: problem polnjenja nahrbtnika (knapsack problem). Običajna formulacija problema je takšna, kot v drugi

nalogi: imamo kup reči, za vsako je znana njena vrednost in teža. Poiskati želimo nabor z maksimalno vrednostjo, pri čemer ni dovoljeno preseči podane teže.

Obstajajo več različic problema. Prvi pravimo *0-1 nahrbtnik*: vsako reč lahko pustimo ali vzamemo, nobene reči pa ne moremo vzeti dva kosa ali več. Druga je *omejeni nahrbtnik*: vsake reči je na voljo določeno število komadov. Tretja je *neomejeni nahrbtnik*: vsake reči lahko vzamemo, kolikor je moremo nositi.

Precej lažja različica naloge je takšna, pri kateri smemo stvari rezati. Tu je stvar trivialna: pogledamo, kaj ima najboljše razmerje med ceno in težo ter natrpamo nahrbtnik z njo.

Še kar lahka različica je takšna, v kateri imajo vse reči enako vrednost glede na težo. Dva kilograma sta vedno dvakrat vrednejša od enega kilograma, pa četudi govorimo o dveh kilogramih slame in kilogramu zlata. Naloga je torej le čimbolj do roba napolniti nahrbtnik. V tej obliki nam ni neznana: vsakič, ko se odpravljamo na morje in poskušamo na različne načine stlačiti v prtljažnik vse cunje, otroške igrače, čoln in vse ostalo, rešujemo problem polnjenja nahrbtnika v teh obliki – le da je tam omejitev predvsem v obliki, ne v teži.

Pozabimo prtljažnik; recimo, da je omejitev teža. Prvi preblisk, ki ga dobimo, je požrešna metoda. Recimo, da smemo v nahrbtnik (ali, recimo raje v samokolnico) naložiti do 41 kilogramov, na voljo pa so paketi s 25, 10 in 4 kilogrami. Hm, nismo prav teh številk že nekoč videli? Jasno, požrešna metoda izbere kovanca, hočem reči paketa, po 25 in 10 kilogramov in enega za 4; tako naložimo 39 kg, v preostanek pa nimamo česa dati. Optimalna rešitev bi bila, kot se spomnimo od kovancev 25 + 4 + 4 + 4 + 4.

Požrešna metoda očitno ne bo prava. Katera pa je? Prav to je tisto: ni je. Pri metodah urejanja smo tarnali nad kvadratno časovno zahtevnostjo in se zadovoljili z n ln n. Požrešna čebela je razmislila svoje v času sorazmernem kvadratu števila vrstic (to je, v času sorazmernem številu polj). Včasih se moramo zadovoljiti s kubično časovno zahtevnostjo. Tule pa ne gre: računalnikarji verjamemo (čeprav tega (še) ne znamo dokazati), da čas, ki je potreben za reševanje problema nahrbtnika, narašča eksponentno s številom reči.

Optimalno rešitev 0-1 nahrbtnika lahko, očitno, poiščemo tako, da preskusimo vse kombinacije reči. Za tiste, ki so dovoljene (to je, niso pretežke), izračunamo njihovo skupno vrednost in si zapomnimo najboljšo. Vseh kombinacij n reči pa je ravno 2^n in vsaka dodatna reč podvoji število kombinacij.

Z omejenim nahrbtnikom je podobno: spet lahko preskusimo vse kombinacije, ki so pod dovoljeno težo, le da se zdaj ne odločamo, ali stvar dati v nahrbtnik ali ne, temveč ali jo bomo dali ničkrat, enkrat, dvakrat, trikrat in tako naprej do tolikokrat, kolikor smemo.

Kako pa ta problem rešujemo ... v resnici? Saj gre za praktičen problem, ne? Dve možnosti imamo. Nekaj se da postoriti z dinamičnim programiranjem, a to tule pustimo pri miru. Druga možnost so hevristični postopki – postopki, ki ne dajo nujno optimalnega rezultata, dajo pa "kar dobrega". Kako se hevristično lotiti nahrbtnika? Natančno tako, kot bi se ga v resničnem življenju: napolnimo, kaj odvzamemo, kaj dodamo...

Na Bobru se naloge, podobne problemu nahrbtnika, sicer kar pogosto pojavijo, vendar sestavljalci vedno poskrbijo, da je prava rešitev očitna ali pa sestavijo podatke tako, da tudi požrešna metoda da optimalno rešitev.

Algoritmi na grafih

Iskanje najkrajše poti

Med dvema točkama na grafu navadno obstaja več možnih poti. Če imajo različne povezave različno ceno, nas pogosto zanima tista, pri kateri je vsota povezav na njej najmanjša. Takšni poti rečemo *najkrajša pot*.

Algoritem, ki ga bomo spoznali, zahteva, da nobena cena ni negativna. Če ni tako, potrebujemo popolnoma drugačen algoritem, ki zahteva tudi veliko več časa (navadno celo preveč, da bi bil praktičen, zato namesto njega uporabljamo približne algoritme). Predpostavimo tudi, da iskana pot obstaja; če ciljno vozlišče sploh ni dosegljivo iz začetnega, bomo pot iskali zaman.

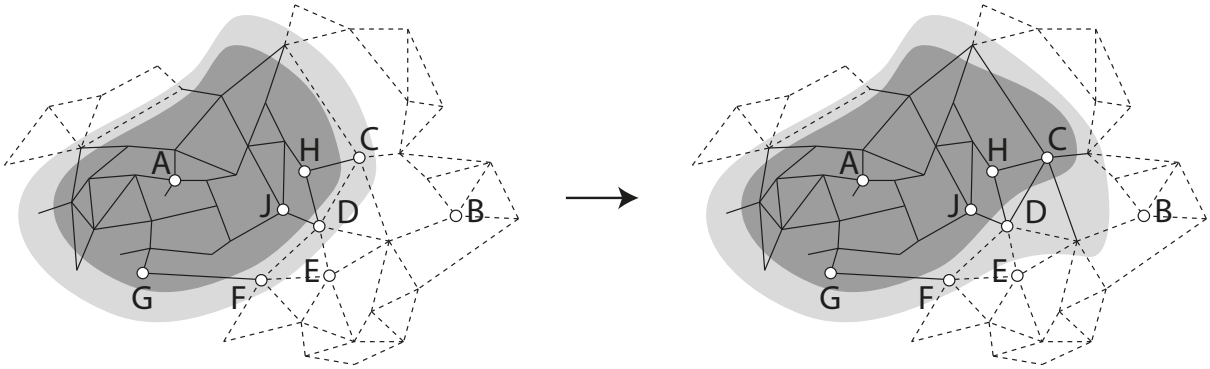
Iskanje najkrajših poti je ena najpopularnejših tipov nalog na tekmovanju Bober. Naloge na to temo: od očitnih, kjer je potrebno poiskati najkrajšo pot na zemljevidu, do prikritih, kot je iskanje najcenejšega ali najpreprostejšega zaporedja del, ki nas pripelje do končnega izdelka.

V nalogah, s kakršnimi se spopadajo osnovnošolci na tekmovanjih, je najkrajšo pot navadno mogoče poiskati kar z metodo ostrega pogleda. Sestavljene so namreč tako, da je število poti (razen očitno predolgih, na primer takšnih, v katerih se točke ponavljajo) dovolj majhno, da jih lahko sistematično pregledamo, ali pa se različne poti na enem ali več mestih združijo in lahko nalogo rešujemo tako, da iščemo najkrajše poti po kosih, med posameznimi "ozkimi grli".

Računalnik nima ostrega pogleda, pa tudi ljudje pri dovolj velikih grafih ne moremo več biti prepričani, da smo res preverili vse možnosti. Tedaj uporabimo algoritem, ki je dobil ime po slavnem nizozemskem računalnikarju Edsgerju W. Dijkstri: Dijsktrin algoritem.

Algoritem ne poišče le najkrajše poti od začetne do ciljne točke, temveč tudi najkrajše poti do množice drugih točk. To počne tako, da začne pri začetni točki in nato postopno širi množico točk, do katerih pozna najkrajšo možno pot.

Točkam, do katerih je že odkrita najkrajša pot, bomo – iz razlogov, ki bodo postali jasni vsak čas – rekli *obiskane točke*. Točke, ki so neposredno povezane z obiskanimi, rečemo *mejne točke*. Algoritem si bomo najprej ogledali na skici, nato še na konkretnem primeru.



Poiskati želimo najkrajšo pot od A do B na gornji sliki. Začnimo na levi strani. Temnejši del kaže obiskane točke. Zanje že vemo, kakšna je najkrajša pot od A do njih – prek katerih točk vodi in koliko je dolga.

Mejne točke so v svetlejšem delu; primeri takšnih točk so C, D in F. Za mejne točke še ne poznamo najkrajše poti. Seveda lahko izračunamo, kako dolga bi bila, recimo, najkrajša pot do F, ki bi vodila prek G: k (že znani) dolžini najkrajše poti do G prištejemo dolžino povezave med G in F. Prav tako lahko za D izračunamo dolžino poti prek H (kot vsoto znane najkrajše poti do H in dolžine povezave med H in D) ter prek J; najkrajša pot do D, ki vodi naposredno iz obiskanega dela, je dolžina krajše od teh dveh možnih poti.

Nihče pa nam ne zagotavlja, da so tako izračunane poti res najkrajše poti do mejnih točk. Najkrajša pot do F ne vodi nujno po neposredni povezavi iz točke G; morda se do F bolj splača iti iz druge mejne točke D, morda pa celo prek mejne točke D in še točke E, o razdalji do katere še ne vemo prav ničesar.

Če razmislimo, pa bomo odkrili, da vsaj za eno mejno točko že poznamo tudi najkrajšo razdaljo do nje. Med mejnimi točkami C, D, F in ostalimi, ki na skici niso označene, poiščemo tisto, do katere vodi najkrajša pot iz ene od obiskanih točk. Recimo, da je to točka C: recimo, da je pot od A do C, ki vodi iz H, krajša od katerekoli druge poti do katerekoli druge mejne točke – krajša, torej, od poti prek H ali J do D, krajša od poti prek G do F, krajša od vseh drugih poti do mejnih točk.

Če je tako je pot do C, ki vodi iz H, tudi najkrajša možna pot do C. Res, ne more biti drugače. Če obstaja kakšna še krajša pot, bi morala voditi prek kake druge mejne točke, recimo D. To pa ne more biti, saj že je pot do D daljša kot pot do C-ja – rekli smo, da je C

"najbližja" mejna točka. Do točke C je seveda mogoče priti tudi iz točk, ki niso mejne, a takšne poti bi bile še daljše, saj moramo tudi do teh točk nekako priti, pridemo pa lahko le prek mejnih točk, ki pa so, spet, lahko le daljše.

Zdaj, ko poznamo najkrajšo pot do C, jo lahko *obiščemo*. Rezultat obiska kaže desni del slike. C smo dodali med obiskane točke in si zapomnili, iz katere obiskane točke smo prišli vanj. Vse točke, s katerimi je C povezana, so postale mejne: zanje zdaj poznamo najkrajšo pot do njih, ki vodi prek C (seveda pa to, kot zdaj vemo, ni nujno tudi najkrajša pot do njih). Za obstoječe mejne točke pa se je najkrajša znana razdalja do njih morda skrajšala: najkrajša znana pot iz obiskanih točk do D morda po novem ne vodi več prek H ali J, temveč prek C.

Korak ponavljamo: spet poiščemo najbližjo mejno točko in jo dodamo med obiskane. Postopek lahko končamo, ko obiščemo B, saj nam je s tem znana najkrajša razdalja do nje. Morebitne preostale mejne točke in točke onstran nas ne zanimajo več.

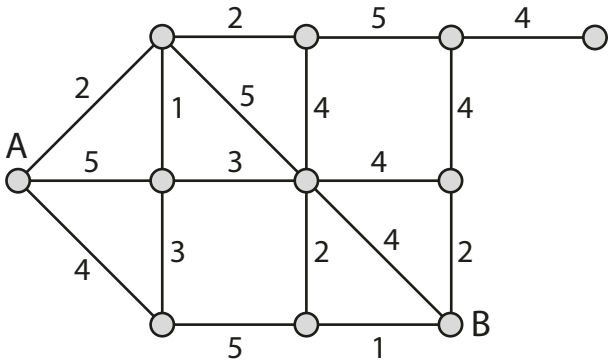
Še enkrat povejmo, ker je pomembno: postopek se ustavi, ko obiščemo ciljno točko in ne že takrat, ko postane mejna. Za mejne točke namreč še ne vemo, ali poznamo najkrajšo pot do njih ali ne. Za obiskane točke pa je najkrajša pot znana.

Če bi ravno hoteli, pa bi lahko postopek gnali še naprej, dokler ne obiščemo vseh točk v grafu. S tem bi dobili *drevo* najkrajših poti iz točke A do vseh drugih točk v grafu (zakaj drevo, bo jasno iz konkretnega primera spodaj). O drevesu smo maloprej povedali nekaj lepega: v drevesu je do vsake točke mogoče priti le na en način. Koren drevesa najkrajših poti je začetna točka, A, torej nam bo drevo povedalo, kako iz A najhitreje priti do vseh drugih točk.

Le še, kako se postopek začne, smo pozabili povedati. A četudi se ne bi spomnili, je menda očitno: začnemo tako, da imamo le eno obiskano točko, namreč začetno točko, A. Pot do nje je dolga 0.

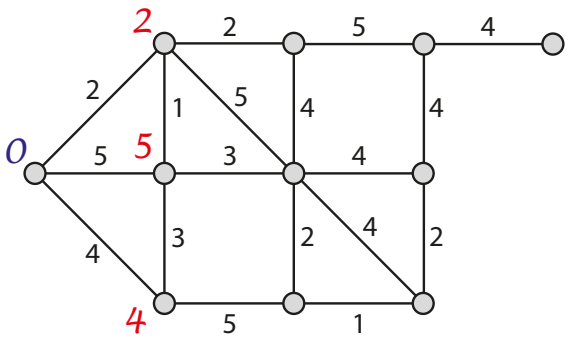
Kaj pa, če ciljno vozlišče iz začetnega sploh ni dosegljivo? Algoritem bo to pravzaprav opazil: zgodilo se mu bo, da bo mejnih točk zmanjkalo, ciljna pa še vedno ne bo obiskana. V tem primeru pač iščemo najkrajšo pot, ki je ni in neuspeh je neizbežen.

In zdaj konkretni primer: poiskati želimo najkrajšo pot od A do B na grafu, ki ga kaže slika.

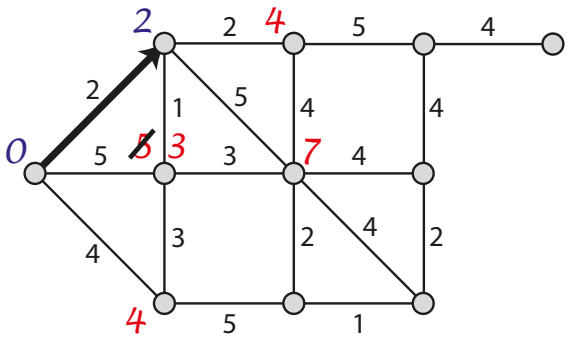


Potem algoritma je ilustriran spodaj.

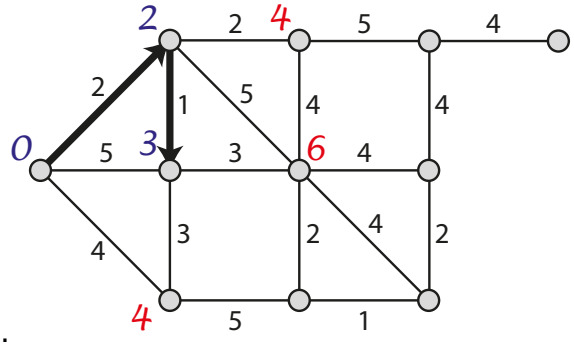
1.



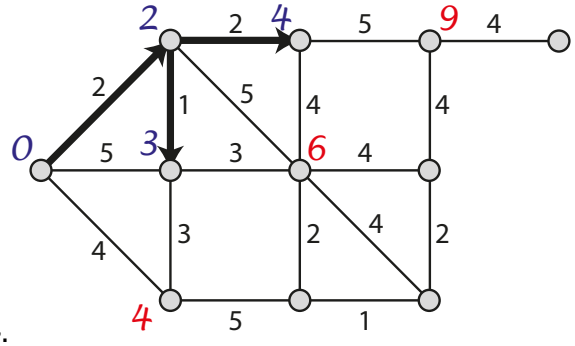
2.



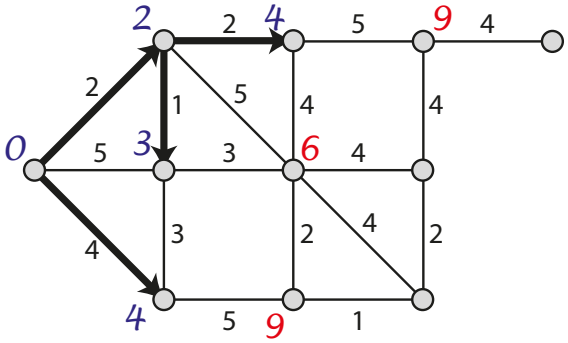
3.



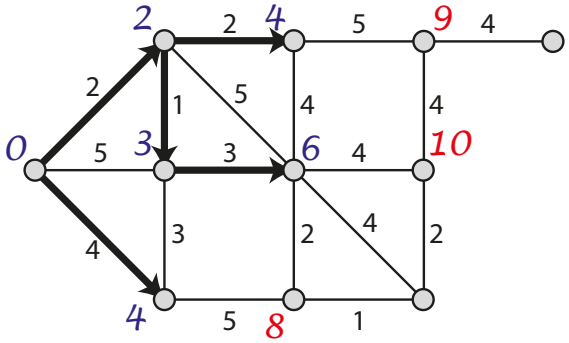
4.



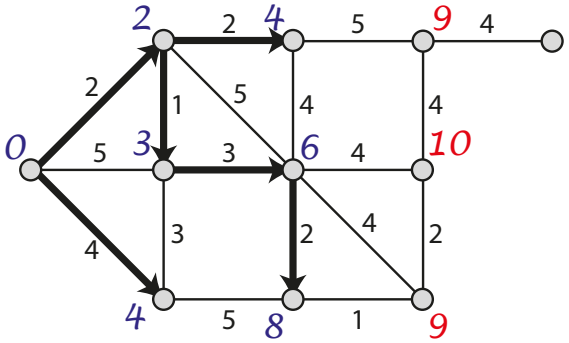
5.



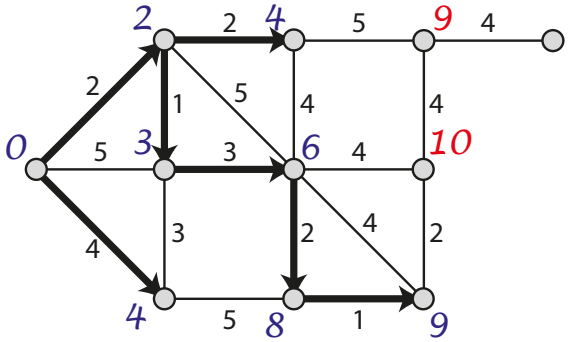
6.



7.



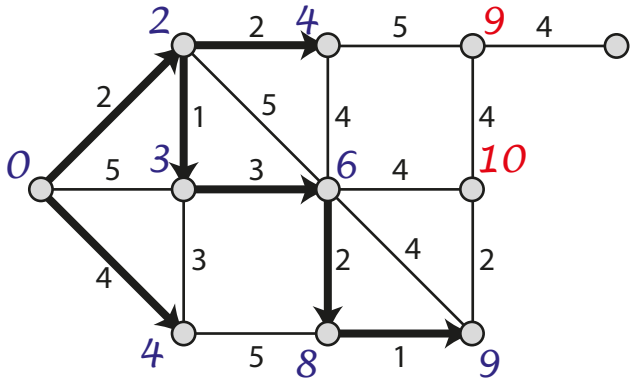
8.



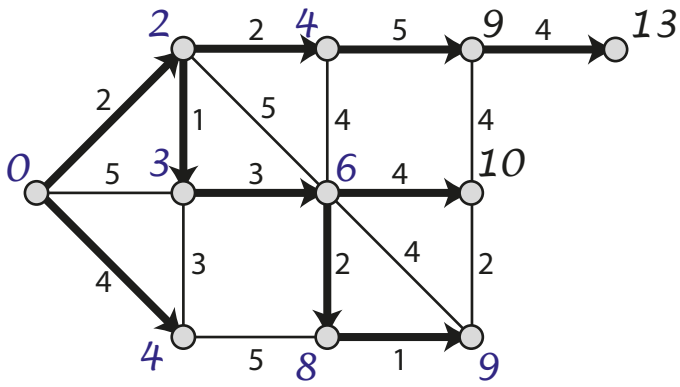
1. V začetku je obiskana točka A, razdalja do nje je 0. Razdalje do treh mejnih točk so 2, 5 in 4.
2. V naslednjem koraku razglasimo najbližjo mejno točko za obiskano. S tem dobimo dve novi mejni točki; razdalji do njiju sta 4 in 7. Do mejne točke, ki je bila prej oddaljena 5, zdaj poznamo krajšo pot dolžine 3.
3. Med obiskane prestavimo mejno točko, oddaljeno 3. To ne prinese novih mejnih točk, le pri eni od obstoječih popravimo razdaljo s 7 na 6.

4. Zdaj imamo dve mejni točki na razdalji 4. Odločimo se za katerokoli od njiju; izbira lahko vpliva na to, kakšna bo najkrajša pot, ki jo bomo našli, ne pa tudi na to, kako dolga bo. Če, recimo, izberemo zgornjo točko, to prinese novo mejno točko na razdalji 9.
5. Med tremi mejnimi točkamim (razdalje do njih so 4, 6 in 9) izberemo najbližjo in jo prestavimo med obiskane. Dobil smo novo mejno točko, razdalja do nje je 9.
6. Zdaj prestavimo med obiskane mejno točko, ki je na razdalji 6. To nam da eno novo mejno točko (na razdalji 10) in zmanjša razdaljo do ene od obstoječih mejnih točk z 9 na 8.
7. Zdaj obiščemo mejno točko, ki je na razdalji 8. To prestavi ciljno točko med mejne točke ... nismo pa še prepričani, da že poznamo najkrajšo razdaljo do nje.
8. Pravzaprav jo: mejne točke so oddaljene 9, 10 in 9. Ciljna točka je najbližja mejna točka (no, ena od njih), torej jo obiščemo. Delo je končano, najkrajša pot in njena dolžina sta znani.

Mimogrede smo izračunali še najkrajše poti do vseh drugih obiskanih vozlišč. Najkrajših poti do ostalih mejnih in do morebitnih vozlišč nismo izračunali in nas tudi ne zanimajo. Lahko pa bi nadaljevali, dokler ne obiščemo vseh vozlišč; tako bi izvedeli najkrajše poti od A do vseh drugih vozlišč.



Če izpustimo povezave, ki nas ne zanimajo, saj ne nastopajo v najkrajših poteh, dobimo drevo.



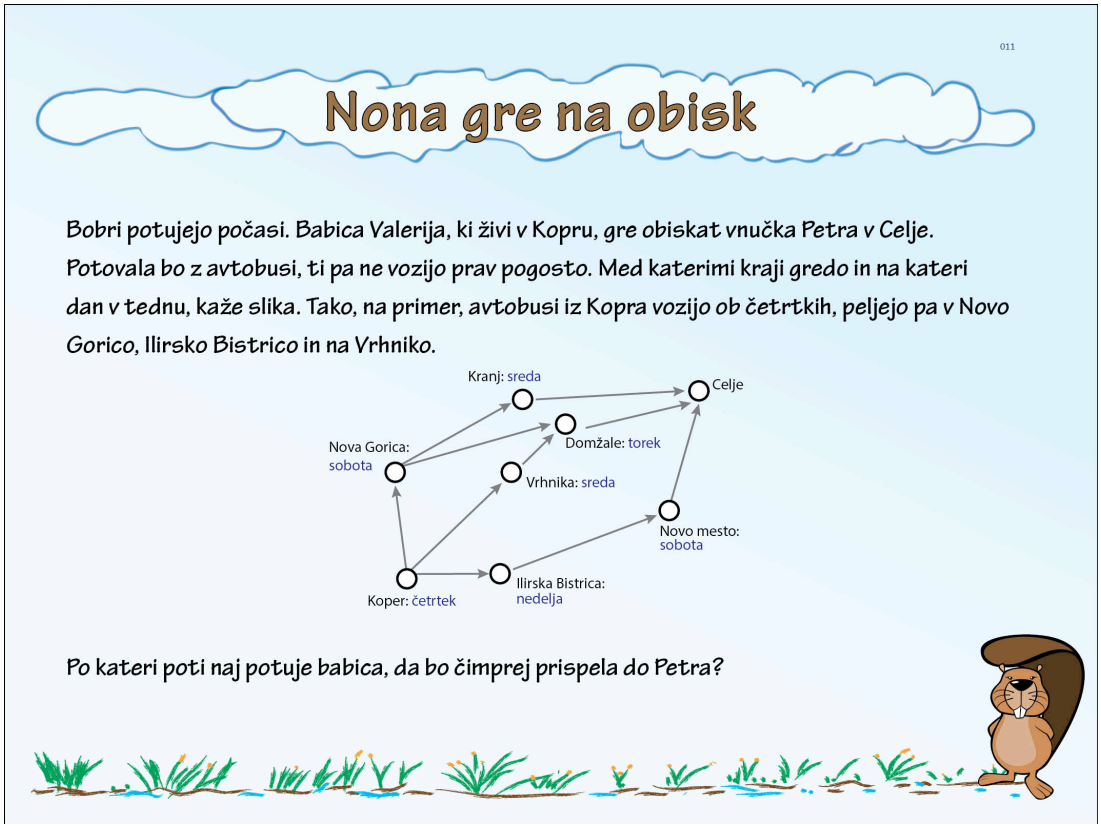
Da mora biti rezultat drevo, je očitno: povezave vodijo do vseh vozlišč (vsaj do vseh vozlišč v tistem delu grafa, ki je dosegljiv iz začetne točke) in do vsakega vozlišča vodi le ena povezava, samo smo vsako vozlišče obiskali (to je, premaknili iz množice mejnih v množico obiskanih točk) le enkrat.

Ne spreglejmo, da drevo govori le o najkrajših poteh iz A v B, ne pa tudi o najkrajših potem med drugimi pari vozlišč. No, med nekaterimi že: najkrajša pot iz vozlišča označenega z 2 do vozlišča z oznako 13 gre gotovo točno tam, kjer jo kaže drevo; če bi obstajala kaka krajša, bi jo uporabili tudi tu. Tudi pot od 2 do 10 vodi tako, kot kaže tole drevo. Pot od 8 do 13 pa vodi bogvekje. Če bi jo hoteli poiskati, bi morali pognati Dijkstrin algoritem iz točke 8.

Dijsktrin algoritem je hiter: čas izvajanja je sorazmeren $M \log N$, kjer je M število povezav in N število točk v grafu. Če predpostavimo, da imajo vse točke bolj ali manj podobno stopnjo, na primer k, bo čas izvajanja sorazmeren $k\,N \log N$: za dvakrat večji graf bo algoritem potreboval nekaj več kot dvakrat daljši čas. O tem se sicer ni težko prepričati, vendar zahteva, da vemo, recimo, kaj je kopica, tega pa ne vedo ne otroci ne mnogi izmed bralcev tega besedila (razen v agronomskem pomenu besede, seveda).

Mimogrede se spomnimo čebele in dinamičnega programiranja. Dijkstrin algoritem je lep primer algoritma, sestavljenega po načelu dinamičnega programiranja. Tako kot smo pri čebeli počasi širili fronto od začetnega cveta proti spodnji vrstici in za vsak cvet opazovali, odkod se nam najbolj splača priti vanj, tudi tu počasi širimo fronto od začetnega vozlišča. Razlika je pravzaprav le v tem, da pri čebeli iščemo maksimum, tu pa minimum, in da imamo pri čebeli dobičke na vozliščih grafa, tu pa ceno na njegovih povezavah. Sicer pa gre za eno in isto reč.

Tule je še lep primer naloge, v kateri je potrebno razmišljati malenkost drugače.



Minimalno vpeto drevo

Včasih nas ne zanima le najkrajša pot iz ene točke v neko drugo ali v vse druge, temveč ... najkrajše poti prek grafa nasploh. Točneje, radi bi zmanjšali število povezav v grafu; graf mora ostati še vedno povezan, skupna dolžina povezav pa čim krajša.

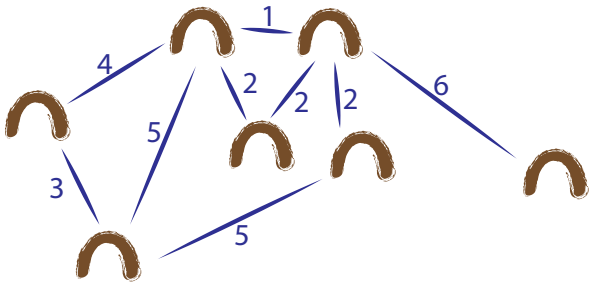
Dijkstrin algoritem nam tu ne pomaga. Potrebujemo nekaj, čemur po svetu pravijo Primov algoritem, Čehi pa vedo povedati, da je to Jarnikov algoritem, ki mu le tujci pomotoma in po krivici pravijo Primov. Pripravljeni pa so, Čehi namreč, na kompromis; pravijo, da bi se temu lahko reklo algoritem DJP, pri čemer je J Jarnik, P pa Prim. Pa D? No, D je Dijkstra, ki se je iste reči spomnil še nekoliko kasneje od prvih dveh. K zmedi pomaga še, da algoritem zamenjujemo z nekim drugim algoritmom za isti namen, s Kruskalovim algoritmom. Da bi bilo vse še bolj zapleteno, je nesrečni Kruskal v istem članku objavil dva algoritma. Le enega od njiju imenujemo Kruskalov, a bi bilo skoraj vseeno, če bi oba, saj sta oba Kruskalova in spet delata isto, le na drugačen način.

Za tiste, ki so ob branju odstavka neuspešno poskušali šteti algoritme, povejmo, da so trije. In jih potolažimo, da smo z imeni končali saj niso pomembna. Sami algoritmi pa so zelo preprosti.

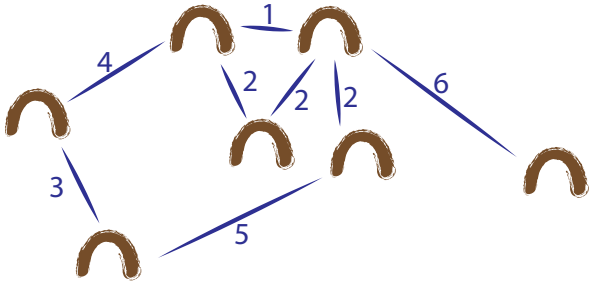
Gredo v eno ali drugo smer. Najprej pokažimo, kako reč teče nazaj. V vsakem koraku pogledamo najdražjo povezavo in se je znebimo, če se je smemo. Povezave se lahko znebimo, če omrežje ostane povezano; če bi ga prekinili, jo pustimo in se lotimo naslednje. Kadar je več enako dragih povezav, se jih lotimo v poljubnem vrstnem redu. Ko ne moremo odstraniti nobene več, nehamo. Izkazalo se bo, da nam vedno ostane ena povezava manj, kot je točk. Kar dobimo, bo vedno drevo, pravimo pa mu *minimalno*

vpeto drevo. *Minimalno* zato, ker ima minimalno ceno, *vpeto* pa zato, ker je vpeto v nek graf.

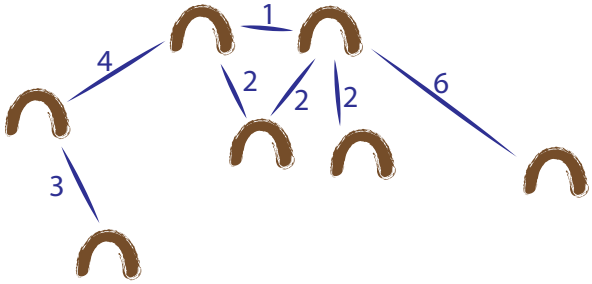
Začnemo torej z grafom takšnim, kot je.



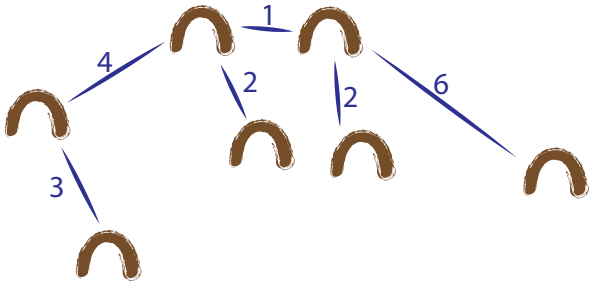
Najdražja povezava ima ceno 6 in z veseljem bi jo pobrisali ... a ne smemo, saj bi s tem odrezali brlog na desni. Nato pogledamo naslednjo najcenejšo povezavo. Dve imamo. Lahko odrežemo levo – bo omrežje razpadlo? Ne bo. Proč z njo!



Pa druga povezava s ceno 5? Tudi brez nje bodo vsi brlogi še vedno povezani, torej se je lahko znebimo.



Lahko odstranimo povezavo s ceno 4? Bognedaj, leva brloga bi ostala brez povezave z ostalimi. Prav, naj ostane. Pa povezava s ceno 3? Tudi ta je nujno potrebna. Zdaj pa povezave s ceno 2. Tri so. Desno potrebujemo, eno od ostalih dveh pa lahko pobrišemo.



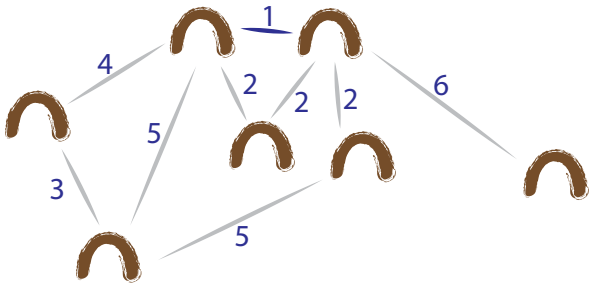
Druge povezave s ceno dve ne smemo pobrisati. Naslednja najdražja povezava je povezava s ceno 1 in ta mora ostati.

Tako smo prišli do konca. Potrebovali bomo 18 metrov žice. Če parafraziramo Tita: z manj ne bo šlo, več ne potrebujemo.

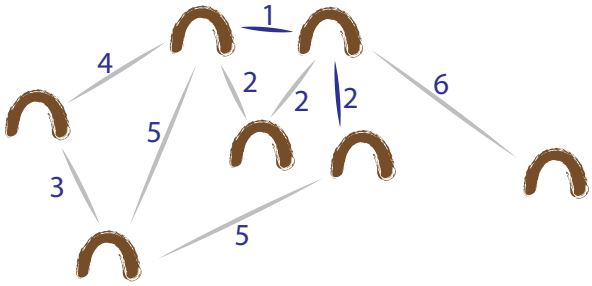
Drugi algoritem deluje v nasprotno smer: namesto da bi začel z vsemi povezavami in jih brisal, jih postopoma dodaja. V vsakem koraku dodamo najcenejšo neuporabljano povezavo, vendar le, če je potrebna. Če bi povezala dve točki, ki sta že povezani (posredno, prek drugih povezav), gremo na naslednjo najcenejšo.

Začnemo z grafom brez povezav; povezave, ki bi lahko bile, a jih še nismo dodali, bomo označili s sivo.

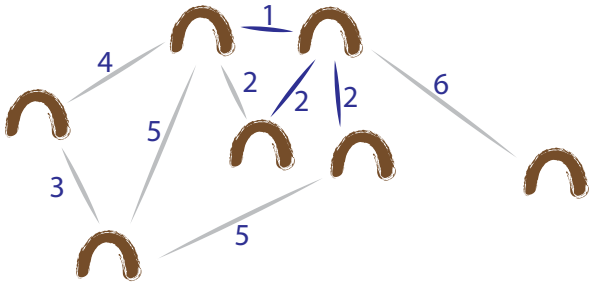
Poiščemo najcenejšo povezavo in jo dodamo. Tule je to za povezava s ceno 1.



Nato dodamo eno od povezav s ceno 2. Katero, je vseeno.

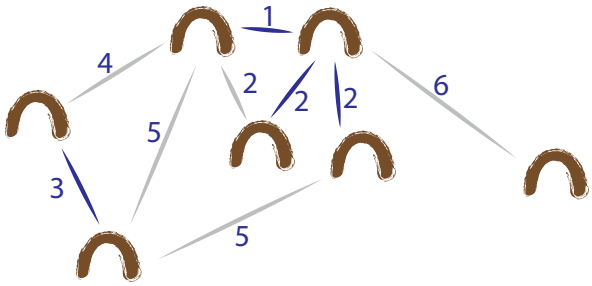


Nato dodamo naslednjo povezavo s ceno 2. Spet je vseeno, katero.

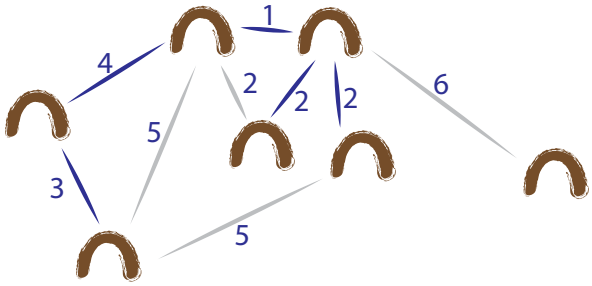


Še eno povezavo s ceno 2 imamo. To pa pustimo, saj bi povezala dva brloga, ki sta povezana tudi brez nje.

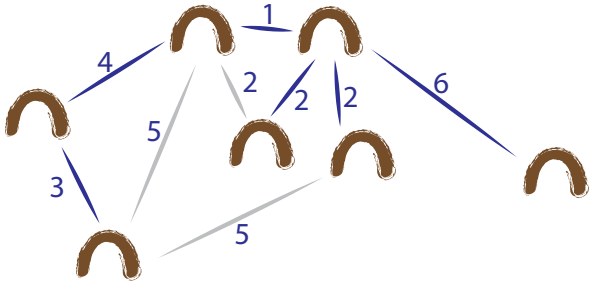
Naslednja najcenejša povezava ima ceno 3. Dodajmo jo.



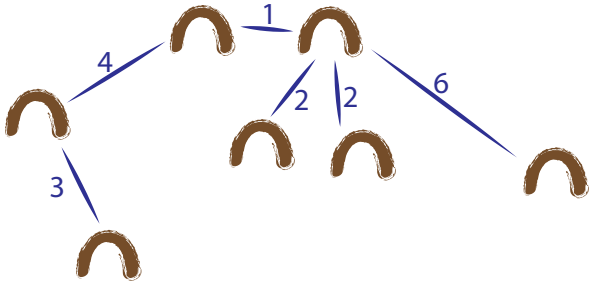
Zdaj je na vrsti povezava s ceno 4. Tudi ta je koristna, saj poveže dva brloga z ostalimi tremi, ki so že povezani.



Naslednja najcenejša povezava ima ceno 5. Pravzaprav sta dve takšni, a nobena nas ne zanima, saj nam ne prineseta ničesar novega: vse, kar bi povezali, je že povezano. Naslednja najcenejša (in edina, pravzaprav že kar najdražja povezava) ima ceno 6. To pa potrebujemo in jo dodamo, cena gor ali dol.



Pobrišimo neuporabljene povezave, da bomo boljše videli, kaj smo pridelali.

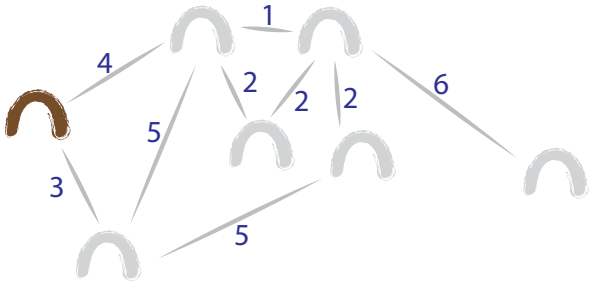


Rezultat je podoben kot prej. Algoritem bo vedno sestavil najcenejše drevo, vendar ne nujno vedno istega. Do razlike pride, ker lahko včasih izbiramo med več enako dragimi povezavami. Končna dolžina povezav pa je enaka, 18.

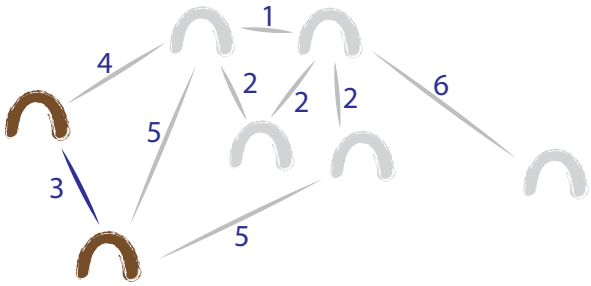
Omenili smo tri algoritme. Tole sta dva. Kje je tretji?

Tudi tretji dodaja, vendar ne povezav, temveč točke (skupaj s povezavami).

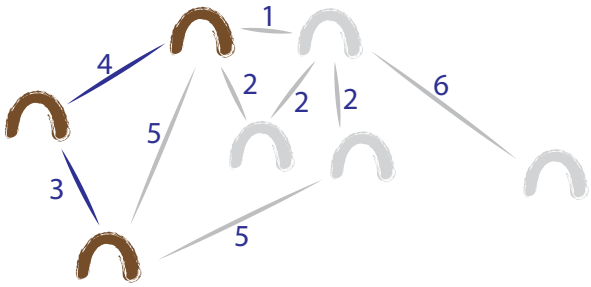
Najprej si izberemo katerikoli brlog. Recimo onega na levi.



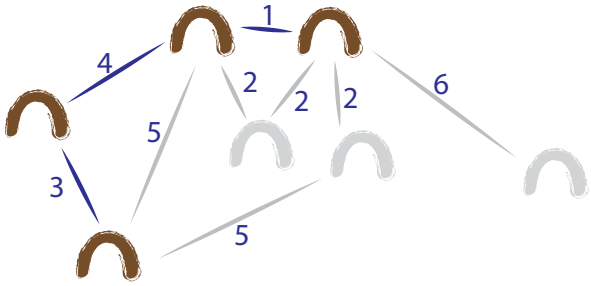
Nato poglejmo, s katerim brlogom ga lahko čim ceneje povežemo. To je brlog pod njim, cena povezave je 3.



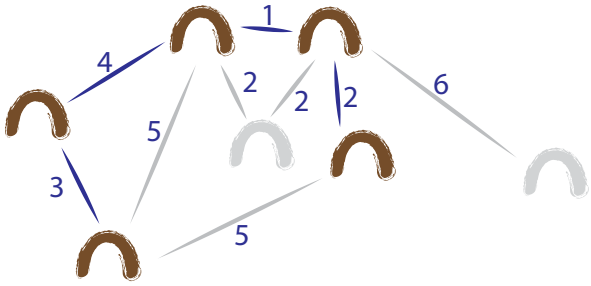
S katerim brlogom lahko najceneje povežemo tadva brloga? Brloga iz izbranih dveh brlogov vodijo tri povezave, njihove cene so 4, 5 in 5. Izberemo tistega s ceno 4.



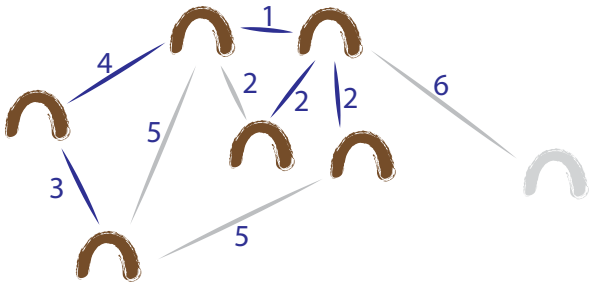
S katerim brlogom bi povezali te tri? Iz njih vodijo tri povezave do brlogov, ki še niso izbrani, njihove cene so 1, 2, 5 in 5. Dodali bomo brlog, do katerega vodi povezava s ceno 1.



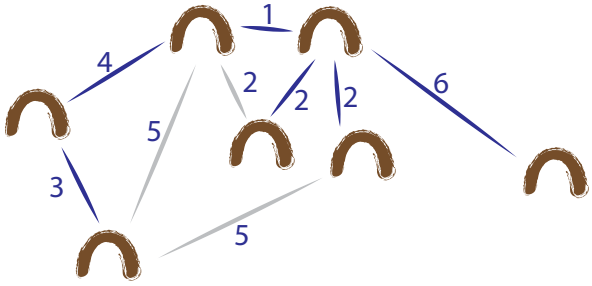
Pa zdaj? Izbrane imamo štiri brloge, do neizbranih treh vodijo povezave s cenami 2, 2, 5 in 6. Izbrali bomo enega od dveh brlogov, do katerega vodi povezava s ceno 2.



Le še dva brloga sta ostala. Do enega pridemo po povezavah z dolžino 2, do drugega po povezavah z dolžino 6. Bolj nam je všeč prvi. (Tule se za hip ustavimo. Zakaj prvi? Navsezadnje bomo morali prej ko slej dodati tudi drugega? Res je, vendar algoritem v tem trenutku še ne ve – ker računalnik pač ne "vidi", kakor vidimo ljudje – ali bo do zadnjega brloga res potrebno uporabiti povezavo s ceno 6, ali pa mora obstaja še kaka cenejša povezava med brlogom, ki ga pravkar dodajamo in zadnjim brlogom.)



Pobožne želje niso bile uslišane in zadnji brlog bomo morali dodati prek povezave s ceno 6.



Končni rezultat je spet tak kot prej. Zadnja dva algoritma sta si v resnici zelo podobna, razlika je le v tem, da pri zadnjem vedno dodajamo v že povezani del, v prejšnjem pa

imamo lahko nekaj časa nepovezane dele, ki se šele v naslednjih korakih povežejo med seboj.

Problem iskanja najmanjših vpetih dreves je zanimiv, ker obstajajo zanj trije različni algoritmi in vsi so preprosti, vsi dajo vedno optimalen rezultat (čeprav ne nujno enakega). Za otroke je pri večini nalog verjetno najpreprostejši postopek z brisanjem. Postopek z dodajanjem pa je prikladen, kadar graf ni podan vnaprej: včasih naletimo na nalogo, ko so podane le koordinate točk, dovoljene pa so vse povezave, med poljubnim parom točk. Tedaj bomo najprej dodali najbližji par točk, nato točk, ki jima je najbližja in tako naprej – tako, kot smo videli v zadnjem algoritmu.

Iskanju minimalnih vpetih dreves je posvečena tudi ena od aktivnosti na Vidri: http://vidra.fri.uni-lj.si/pomagajmo-cestarjem.

Obhodi

Poleg najkrajših poti po grafu nas pogosto zanimajo tudi kake druge, recimo najdaljše. Vsi smo že kot otroci reševali naloge vrste "Prehodi vse poti v parku", tako da boš na vsako stopil le enkrat, ali pa obišči vsa drevesa v parku, a mimo vsakega smeš le enkrat. Zdaj vemo, da lahko te parke narišemo kot grafe in po vsem, kar smo se tule naučili doslej, bi rekli, da morajo najbrž obstajati tudi nekakšni postopki, s katerimi te naloge sistematično rešujemo?

Pot, ki gre prek vsake stezice natančno enkrat, imenujemo Eulerjeva pot ali, če se konča tam, kjer se je začela, Eulerjev obhod. Algoritem zanj je preprost in ga otroci odkrijejo sami. Za začetek recimo, da v vsako križišče vodi sodo število poti. V tem primeru

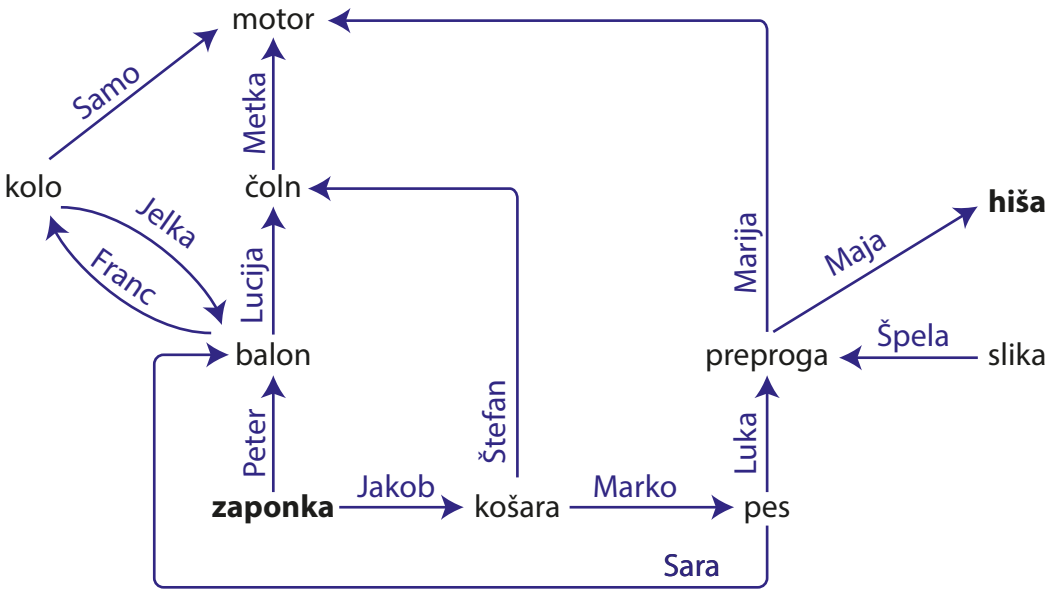
nalogo rešimo tako, da začnemo v kateremkoli in hodimo naokrog, dokler moremo in kakor hočemo. Če se bo kje ustavilo in ne bomo mogli naprej, se bo gotovo nekoč, ko se bomo vrnili v začetno vozlišče. Drugače ne more biti: vsa vozlišča imajo sodo število poti in če smo prišli v neko vozlišče, smo gotovo uporabili liho število poti, ki vodijo vanj oz. iz njega, torej mora gotovo obstajati vsaj še en izhod. Izjema je začetno vozlišče: ko pridemo vanj, je uporabljeno sodo število poti okrog njega. Če se torej zataknemo v njem, obdržimo pot, ki smo jo naredili, a jo popravimo tako, da vanjo "vstavimo" kako od poti, ki je še neizkoriščena.

Če imamo v grafu (ali parku) tudi križišča z lihim številom poti, potem smo lahko prepričani, da takšno križišče ni le eno, temveč vsaj dve. (V resnici jih je vedno sodo število, a to tule ni pomembno.) Pot moramo, kot vedo že otroci, začeti v enem od vozlišč z lihim številom poti, končali pa ga bomo v drugem. Če je križišč z lihim številom poti več, recimo štiri, je naloga nerešljiva.

Drugi problem, obiskati vsa križišča, se imenuje po Hamiltonu – Hamiltonova pot oziroma Hamiltonov obhod. Tu nismo posebej pametni: no, smo, matematiki so si domislili kup izrekov in računalnikarji kup algoritmov, v resnici pa ne poznamo algoritma, ki bi učinkovito poiskal Hamiltonovo pot ali obhod. Čim je graf nekoliko večji, bo algoritem potreboval ogromno časa, da bo poiskal pot. Ne le, da takšnega algoritma ne poznamo, temveč imamo kar dobre razloge, da verjamemo, da ga ni.

Namesto, da bi tule razpravljali o algoritmih, raje poglejmo dva lepa primera nalog, ki ju je lahko rešiti. Na Bobru se občasno pojavljajo naloge, v katerih iščemo Eulerjevo ali Hamiltonovo pot, vendar so precej dolgočasne: navadno je graf že narisan (le da mu v nalogi ne rečemo graf, temveč zemljevid ali kaj podobnega) in je potrebno le poiskati želeni tip poti. Nalogi, ki ju bomo videli tule, sprašujeta po drugačni poti; lepi sta, ker graf ni očiten na prvi pogled.

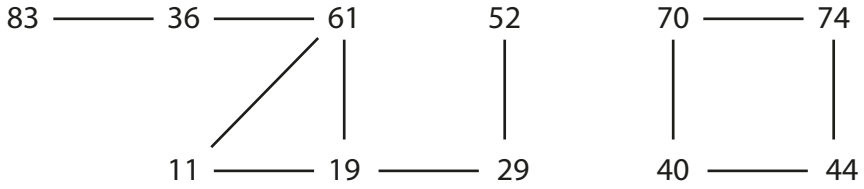
Prva je naloga s trgovanjem. Nalogo je težko reševati zato, ker se iz tabele ne znajdemo. Prerišemo jo lahko v spodnji graf.



Čim pridemo do sem, je reč trivialna: zaponko je potrebno zamenjati za košaro, to za psa, psa za preprogo in preprogo za hišo.

Lepota naloge je v tem, da mora učenec poiskati primerno predstavitev problema in potem je vse enostavno. V kontekstu Bobra je sicer nekoliko nerodna, ker je morda vseeno hitrejše poiskati rešitev s poskušanjem po tabeli kot izgubljati čas s prerisovanjem. A nič hudega: v takšnih primerih lahko izven tekmovanja vidijo, kako se takšno reč reši lepo, sistematično.

Lepota naloge je v tem, da graf ni podan eksplicitno (kot recimo pri Telefonski mreži), niti niso ekplicitno podane relacije (kot pri prejšnji nalogi). Na prvi pogled tole niti od daleč ne diši po grafih. Po drugi strani: kako naj se človek loti te reči? Ne gre drugače, kot da si narišemo hiše in povežemo tiste, med katerimi sme iti poštar, se pravi te, ki imajo kako skupno številko.



V jeziku tega predavanja je to spet, očitno, graf in to, kar iščemo, je najdaljša Hamiltonova pot po grafu. Graf ni povezan (če poštar začne, recimo, pri 29, ta dan ne more iti do 40), zato prave Hamiltonove poti ne bomo našli. Najboljše, kar lahko sestavimo, je 83 – 36 – 61 – 11 – 19 – 29 – 52 ali obratno.

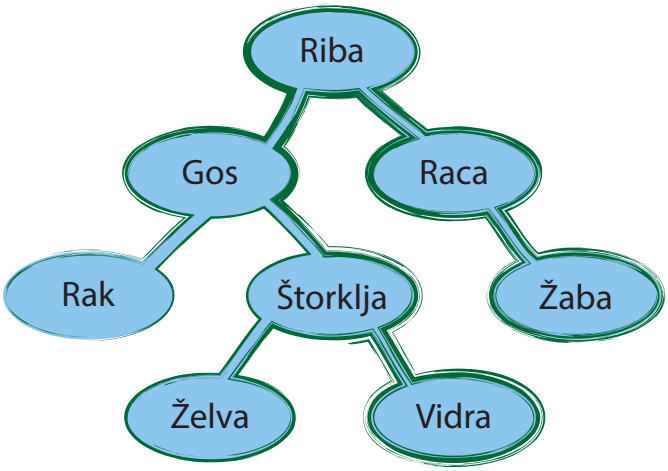
Tako kot prejšnja postane tudi ta naloga trivialna, če se spomnimo na grafe in jo primerno narišemo.

Obhodi dreves

Nekoliko drugačna vrsta obhodov gre prek nekoliko drugačne vrste grafov – dreves, ki smo jih spoznali v prejšnjem sklopu. Čeprav ne sodi čisto k algoritmom, jo omenimo tu.

Ker je tema pomembna, se naloge te vrste pogosto pojavljajo na Bobru. Po drugi strani jo je težko osmisliti izven precej globjega konteksta, za katerega tu ni časa, pa tudi pri Bobru ne pride do izraza – naloge te vrste so pogosto puste, saj nimajo prave zgodbe, temveč govorijo le o bobrih, ki po kakšnih čudnih in z ničemer utemeljenih vrstnih redih obiskujejo svoje prijatelje, raziskujejo jame in podobno.

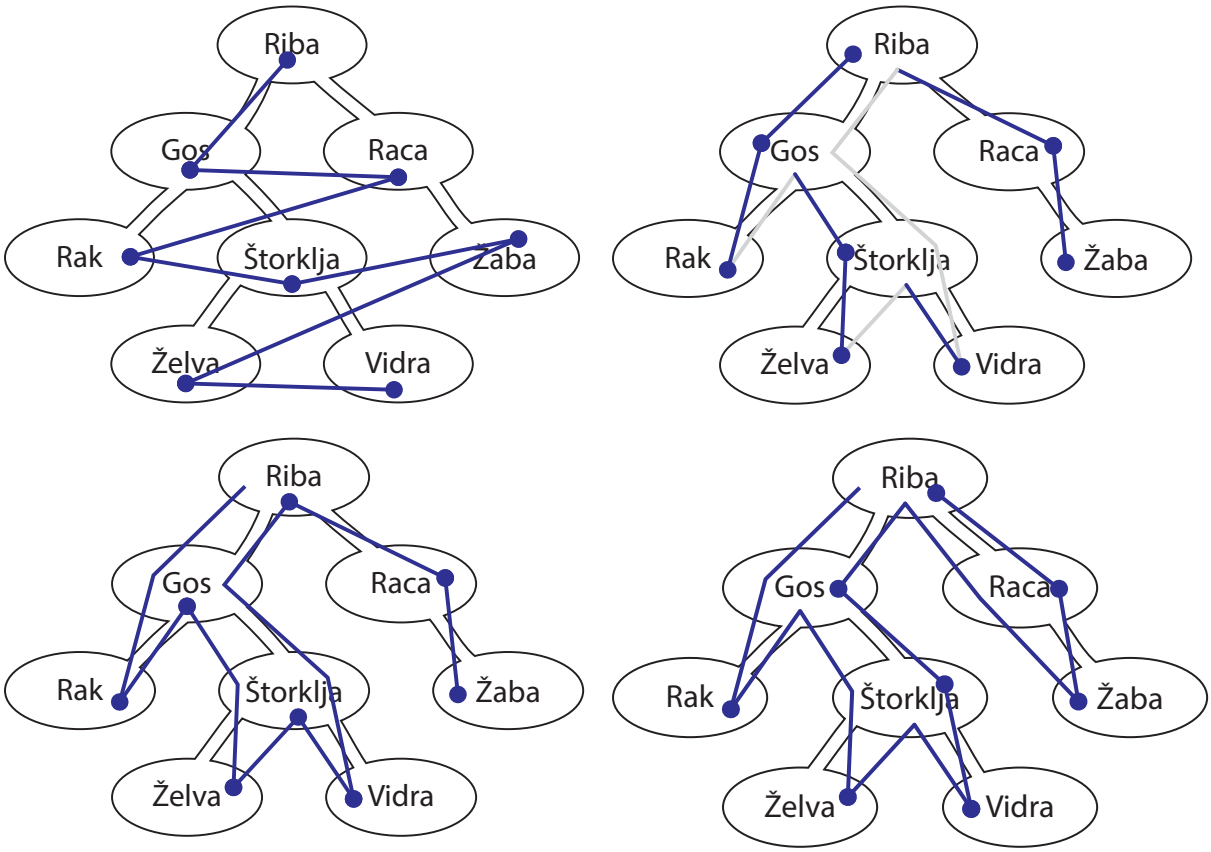
Sicer pa reč ni zapletena. V osnovi gre za tole: recimo, da v mlakah, povezanih s kanali, živijo različne živali, tako kot kaže spodnja slika.



Kako bi našteli prebivalce mlak? Sistematično, po vrsti – po kakršnikoli vrsti, ki se vam zdi smiselna glede na drevo (torej, kriterij za vrstni red naj narekuje drevo, ne abeceda ali kaj podobnega)?

Večina jih najbrž našteje takole: riba, gos, raca, rak, štorklja, žaba, želva, vidra. Nekateri bi šli po drugem sistemu: riba, gos, rak, štorklja, želva, vidra, raca, žaba. Lahko pa tudi tako: rak, gos, želva, štorklja, vidra, riba, raca, žaba. Ali: rak, želva, vidra, štorklja, gos, žaba, raca, riba.

Aha, pravite, da je prvi vrstni red je očiten in edini normalen, naslednji pa vedno bolj čudni ali celo čisto nerazumljivi? No, poglejmo. Za začetek jih narišimo – pri zadnjih dveh je brez slike res težko videti, za kakšen vrstni red (če sploh kakšen?) gre.



Prvi vrstni red imenujemo preiskovanje v širino: najprej "preiščemo" prvi nivo, nato drugega, tretjega in tako naprej, dokler ne pridemo do konca. Že, da govorimo o "preiskovanju" ne naštevanju ali čem podobnem, nakazuje, da prihaja reč iz nekega drugega vica, vendar zanj nimamo časa.

Pri ostalih treh gremo najprej v globino. Po drevesu (ali mlakah) se v vseh treh primerih sprehajamo v enakem vrstnem redu: bober se vedno najprej zapelje v levo, nato v desno mlako, potem pa vrne nivo višje. Razlika je v tem, kdaj "imenujemo" žival v posamezni mlaki.

Predstavljamo si bobra-akviziterja, ki se s čolnom vozi po kanalih med mlakami (dasiravno so bobri odlični plavalci, na tekmovanju Bober skoraj vedno uporabljajo čolne) in jim poskuša prodati kakije ali pa polivinilaste vrečke, kakor je navada pri akviziterjih. (V originalni nalogi je zgodba nekoliko drugačna; tole je različica za odrasle, ki črpa iz sloga socialnega realizma.)

Prvi vrstni red je popolnoma neuporaben, saj mu je čoln le v napoto – namesto da bi se z njim vozil med kanali, ga bo moral skupaj s kakiji in vrečkami vlačiti po kopnem med mlakami. Tudi za programiranje (recimo, da bi imeli takšno drevo shranjeno v pomnilniku in bi morali izpisati živali) je ta vrstni red najbolj zoprn.

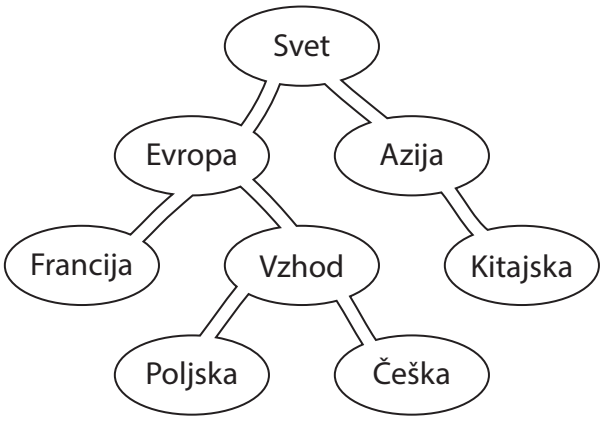
V drugem vrstnem redu (prvem izmed teh, ki gredo v globino, ne širino) se bober vozi med mlakami. Ko pride do določene mlake, najprej nadleguje žival, ki živi v tej mlaki, nato vse živali na levi strani in nato one na desni. Ko se vrača, živali, ki jo je že nadlegoval, ne nadleguje ponovno – tega tudi resnični akviziterji pretežno ne počnejo.

V tretjem vrstnem redu najprej nadleguje vse živali na levi, nato žival, ki živi v mlaki in nato vse na desni. Če gremo od začetka: najprej se bo lotil živali, ki so levo od ribe, nato ribe in potem živali desno od ribe. Na vsaki strani se zgodba ponovi: med živalmi levo od ribe bo najprej nadlegoval tiste, ki so levo od gosi, nato gos in potem vse, ki so desno od gosi. Levo od gosi je le rak, desno od gosi pa spet ponovi isto: najprej živali levo od štorklje (torej želvo), nato štorkljo in nato živali desno od štorklje (vidro).

V zadnjem vrstnem redu najprej nadleguje živali levo od posamezne mlake, nato živali desno in končno še žival v mlaki. Tako bo, recimo, najprej nadlegoval tiste levo od ribe, nato tiste desno in na koncu še ribo samo. Najlepše se vrstni red vidi spodaj, kjer se loti najprej želve, nato vidre in šele na koncu štorklje.

Tudi ti trije vrstni redi imajo imena: prvi je *premi*, drugi *vmesni* in tretji *obratni*. Lažje razumljive tujke jim pravijo *prefiksni*, *infiksni* in *postfiksni*: zapomnimo si jih po tem, da povedo, kje se pojavi "koren", žival, ki jo srečamo prvo – pred ostalimi (*pre*-), med njimi (*in*-) ali za njimi (*post*-). Nobene škode pa ni, če si imen – ne latinskih ne kranjskih – sploh ne zapomnimo, saj gre vendar samo za imena.

Namesto tega vse čudne vrstne rede – vse razen prvega, ki menda ni čuden – upravičimo, ovsakdanjimo. Tule je razdelitev držav po svetu.

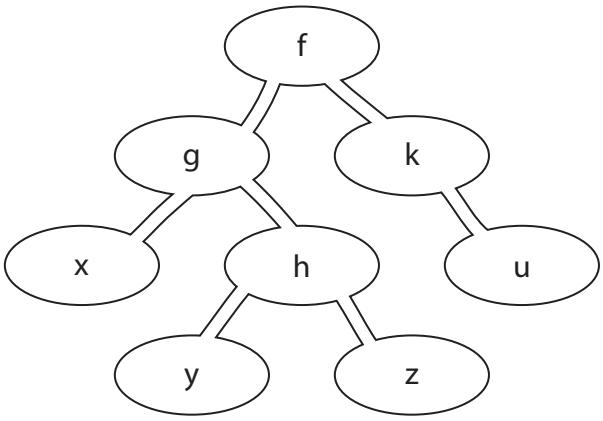


Če bi jo hoteli prepisati v (gnezdene) alineje, bi rekli

- Svet
    - Evropa
        - Francija
        - Vzhod
            - Poljska
            - Češka
    - Azija
        - Kitajska

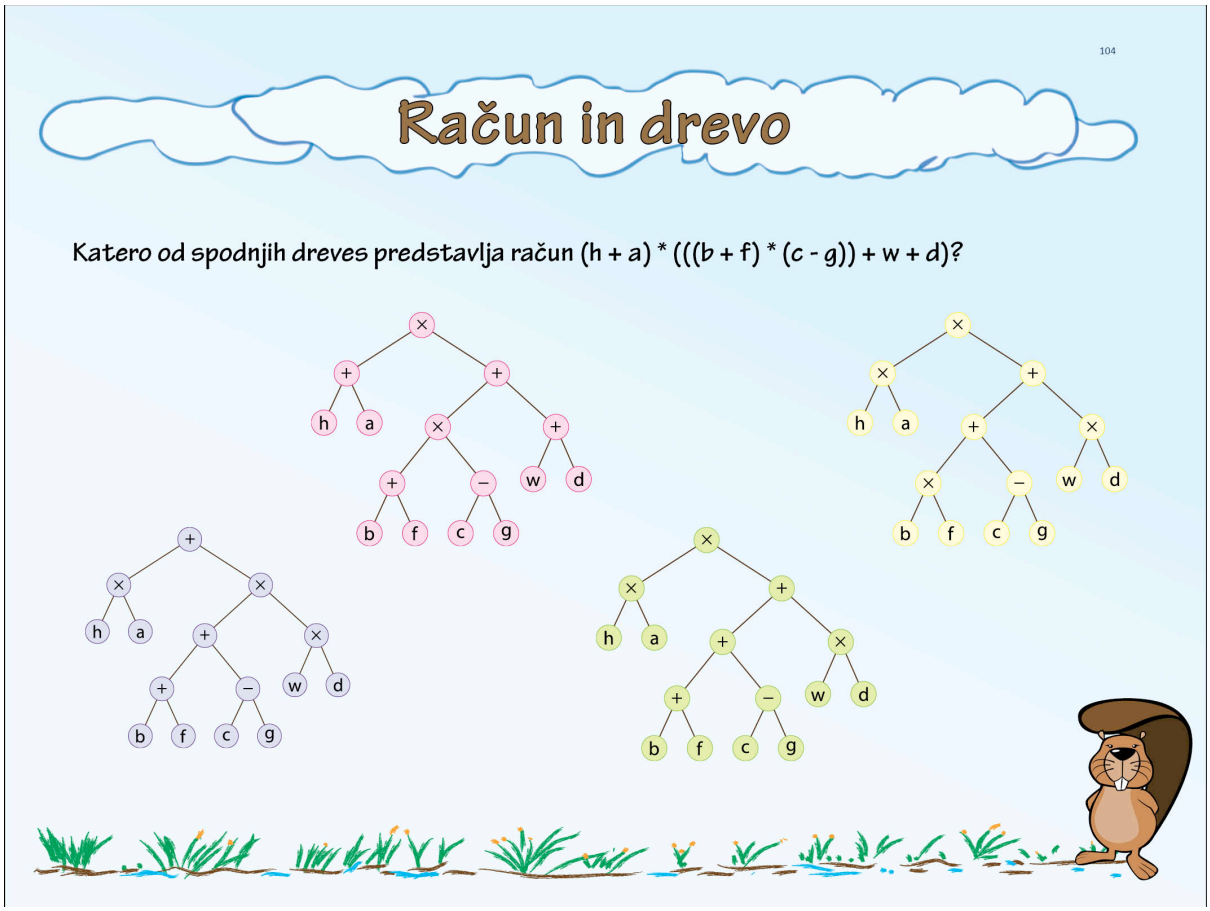
Vrstni red, v katerem smo izpisali drevo, je natančno takšen kot v drugem razporedu.

Tu je še malo drugačen primer, v spomin na nalogo z aritmetičnimi izrazi. Recimo, da imamo funkcije *f*, *g*, *h* in *k* ter spremenljivke *x*, *y*, *z* in *u*. Tedaj bi drevo



ustrezalo izrazu $f(g(x, (h(y, z)), k(u))$. Vrstni red, v katerem se pojavljajo simboli, f, g, x, h, y, z, k, u je natančno drugi vrstni red.

Za utemeljitev tretjega načina se spomnimo naloge



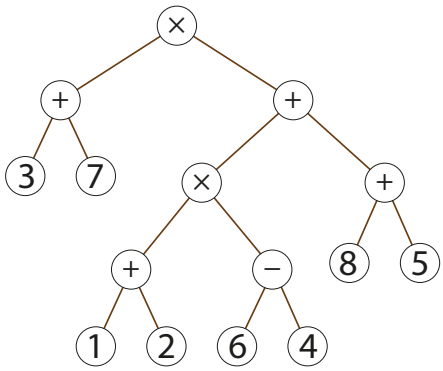
(Da se ne mučimo ponovno: pravilno je rožnato drevo.) V izrazu (h + a in tako naprej) se najprej pojavi levo poddrevo, nato simbol, nato desno poddrevo. Predstavitev izrazov z drevesi pravzaprav niti ni tako slaba ideja: izraz (h + a) * (((b + f) * (c – g) + w + d) je jako nepregleden. Kar dobro ga moramo pogledati, da odkrijemo, katera je najbolj zunanja operacija. Iz drevesa pa je povsem očitno, da bomo nekaj zmnožili – namreč tisto na levi s tistim na desni. Na levi je vsota h in a, na desni pa vsota ... uf, tistega, kar je na levi in tistega, kar je na desni te vsote.

Za zadnji vrstni red najprej recimo, da akviziter ve, da bo vsaka žival kupila toliko kakijev, kot oba njena soseda skupaj. Da bi lahko določeni živali (recimo gosi) povedal, koliko sta kupila njena soseda (rak in štorklja), bo moral najprej k raku in štorklji, šele nato k želvi.

Zgodba zveni privlečena za lase, pa ni: kako bi v resnici izračunali vrednost izraza $f(g(x, (h(y, z)), k(u))$ (ker ga je težko brati, morda raje poglejte drevo, ki ga predstavlja)? Lahko najprej izračunamo vrednost funkcije f, nato g in h? Ne. Prav tako ne moremo najprej izračunati g, nato f, nato h. Ne, najprej je potrebno izračunati to, kar je levo (g) in kar je desno (h); nato iz rezultata tega dvojega izračunamo f.

Zadnji vrstni red je od vseh treh zato najbolj naraven – čeprav nas je v začetku morda najbolj zmedel. Se opravičujem, a tako je. To je vrstni red, v katerem računamo.

Razlike med zapisi še enkrat povejmo na primeru izrazov, le da črke zamenjajmo s številkami, da bo nazorneje.

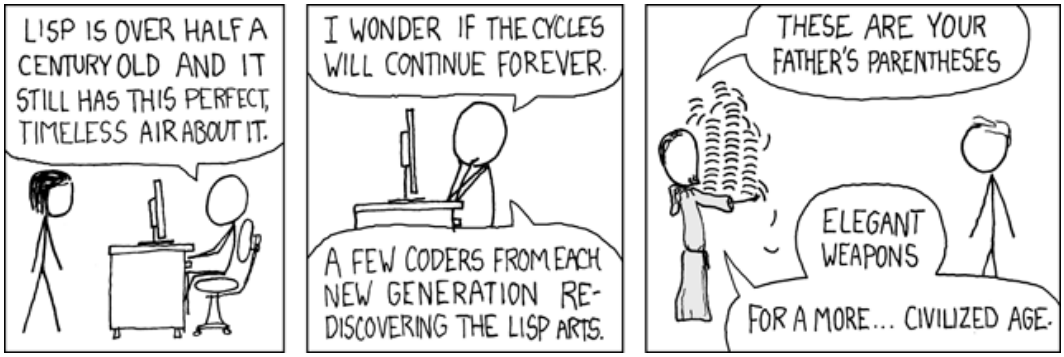


(3 + 7) * (((1 + 2) * (6 – 4) + 8 + 5).

Tega, infiksnega zapisa smo tako vajeni, da se nam zdi, da drugačni zapisi nimajo smisla. Obstajajo pa računalniški jeziki, v katerih bi morali takšen izraz zapisati kot

(* (+ 3 7) (+ (* (+ 1 2) (- 6 4) (+ 8 5)))),

torej najprej operator in nato operandi. V tej obliki je izraz zapisan prefiksno. To se po svoje lepo bere: zmnoži vsoto 3 in 7 ter vsoto zmnožka vsote 1 in 2 in razlike 6 in 4 ter vsote 8 in 5. Edini problem tega jezika in takšnega opisa je, da potrebujemo oklepaje (in to, očitno, veliko oklepajev, predvsempa enako zaklepajev, ki se naberejo na koncu!), da jasno določimo vrstni red. Ni pa povsem nenaravno. (Povejmo: gre za jezik Lisp in njegove naslednike. Lisp je eden najstarejših, a tudi najvplivnejših in, za razliko od njegovega opešanega vrstnika Fortrana, še vedno atraktivnih jezikov. Prednost takšnega zapisa je, da vodi v preproste jezike (po neki definiciji preprostosti, ki je jasna predvsem ljubiteljem takšnih jezikov;

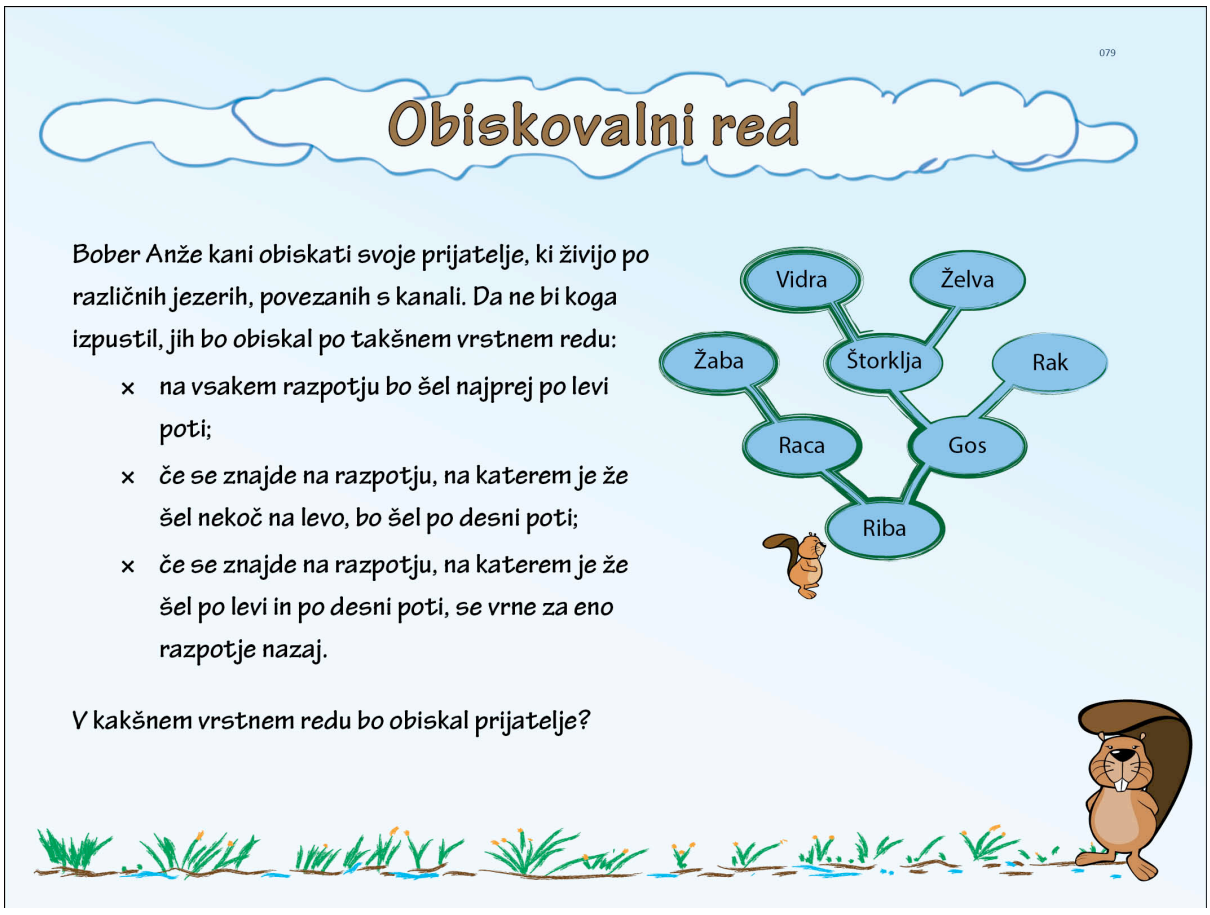


(Vir: http://xkcd.com/297/; strip je pod licenco CC-BY-NC, dovoljena uporaba v nekomercialne namene)

).

Obstaja pa še tretji način zapisa izrazov: 3 7 + 1 2 + 6 4 - * 8 5 + + *. Ta zapis ustreza zadnjemu postfiksnemu. V tem vrstnem redu računamo v resnici: vzamemo 3 in 7 ter ju seštejemo ter si zapomnimo vsoto. Vzamemo 1 in 2 ter ju seštejemo. Vzamemo 6 in 4 ter ju odštejemo. Zmnožimo zadnji dve številki... Lepota tega zapisa je v tem, da zanj nikoli ne potrebujemo oklepajev.

Ste med kupovanjem tiskalnika ali nameščanjem gonilnikov zanj kdaj naleteli na besedo PostScript? PostScript je jezik, v katerem računalnik pove tiskalniku, kaj in kako naj izpiše. Ime je dobil po postfiksnem zapisu vsega – ne le aritmetičnih izrazov, ves jezik je narejen tako kot nemščina, z glagoli na koncu. Tudi datoteka PDF ni nič drugega kot program, ki v jeziku PostScript pove bralniku (na primer Acrobat Readerju), kaj naj nariše in kakšno besedilo pokaže.



V tej nalogi, na kateri je temeljil razdelek, je drevo obrnjeno drugače. Tako je praktično, saj sta leva in desna s tem natančno določeni: če rišemo drevo od zgoraj navzdol, kot je običaj med računalnikarji, bober potuje proti gledalcu in njegova leva je naša desna, kar zmede opis.

Kot smo omenili na začetku razdelka, je zoprna lastnost teh nalog, da nimajo posebne zgodbe: čudaški bober se je odločil za tak in tak vrstni red obiskovanja mlak. Zakaj, naloga ne pove.

Druga nerodna reč ob teh nalogah je, da od otrok zahtevajo bolj ali manj samo, da razumejo opis postopka. Če ga razumejo, je rešitev trivialna; če ne ... pa je otrokovih težav morda kriv samo nejasen opis. In nekatere vrste obhodov je v resnici težko lepo

opisati. Pisec tega besedila čuti do Bloomove taksonomije in podobnih reči zmeren odpor, tule pa jo morda omenimo – če ne zaradi drugega, vsaj zato, da učitelji ne bodo imeli občutka, da so se o tem piflali zaman (dasiravno bi bil ta občutek morda na mestu): tovrstne naloge zahtevajo predvsem razumevanje in, v najboljšem primeru, čisto preprosto uporabo. Od tekmovalnih nalog bi si človek upal pričakovati več.

Žal pa so precej pogoste, zato ne bo nič narobe, če jih otrokom pokažemo. Da jim bomo imeli ob tem povedati še kaj zanimivega, pa smo v tem razdelku kar obilno poskrbeli.

Barvanje grafov

Naloga je navidez podobna nalogam o Pavlinih ploščicah in o socialnih omrežjih, ki smo jih reševali, ko smo spoznavali grafe. Saj tudi je. Ob grafih jih omenjamo zaradi algoritma, povezanega s tovrstnimi grafi.

Francis Guthrie je sredi 19. stoletja barval zemljevid angleških grofij in odkril, da ga že s štirimi barvami lahko pobarva tako, da sta sosednji grofiji vedno različnih barv (pri tem grofiji, ki se stikata le v eni točki nista sosednji; poleg tega nobena grofija ni sestavljena iz več kosov, kot recimo Hrvaška, ki jo bosanski Neum preseka v dva dela; nadalje moramo predpostaviti, da je zemljevid planaren, torej ni na krogli ali čem še bolj čudnem, recimo torusu). Opazil je, da to ni kaka posebnost angleških grofij, temveč to velja za vsak zemljevid – vsaj tako se mu je zdelo, vendar tega ni mogel dokazati. Guthriejev brat je to pokazal znanemu matematiku de Morganu, ta pa je to omenil v pismu, ki ga je leta 1852 napisal Hamiltonu, ki smo ga že srečali na naši poti v prejšnjem razdelku.

Čemu ta zgodba? Ker je zanimiva: problem je matematike frustriral dobrih 120 let, preden so leta 1976 končno dokazali, da je s štirimi barvami res mogoče pobarvati vsak (ravninski) zemljevid. Ker so si pri dokazovanju morali pomagati z računalnikom, so matematiki nad dokazom še dolgo negodovali.

Na Bobru so pogoste naloge, ki zahtevajo barvanje zemljevidov. Da uvidimo zvezo med zemljevidi in grafi, smo zdaj menda že dovolj pametni. Kako pridemo iz zemljevida do grafa, je povedala naloga: vsaka pokrajina je točka in dve točki sta povezani, če imata skupno mejo. S tem seveda še nismo rešili problema, le prevedli smo ga na grafe. Zdaj moramo vsaki točki prirediti barvo, a tako, da sta povezani točki vedno različnih barv.

Algoritem? Za ročno reševanje (in tudi reševanje z računalnikom) se splača začeti pri točki, ki je najbolj zapletena, ki ima največ povezav. Dodelimo ji neko barvo, nato pa – po občutku – nadaljujemo z drugo najbolj zapleteno točko ali pa s kako točko, ki je povezana s to in ima tudi sama veliko povezav.

Drugo, na kar se splača paziti so polni podgrafi. Če v grafu zalotimo štiri točke, ki so povezane med sabo, vemo, da bodo morale biti štirih različnih barv. Barvanje lahko začnemo s takšnimi štirimi točkami in potem nadaljujemo po poti, ki nas najbolj "omejuje", se pravi tako, da vedno pobarvamo tisto točko, pri kateri imamo najmanj izbire.

Tale opis, kot prvo, ni ravno opis algoritma, saj računalnik ne more delati "po občutku". Poleg tega ta algoritem ne da nujno optimalne rešitve. Lahko se zgodi, recimo, da bomo porabili pet barv. A nič hudega: če najdemo rešitev s petimi barvami, vemo (hvala, matematiki!), da se nismo dovolj potrudili. Rešitev popravljamo, dokler ne najdemo takšne s štirimi barvami.

Lepo, vendar: kakšno zvezo ima to z računalništvom? In kakšen pomen ima, pravzaprav, to na splošno? Je vredno izgubljati čas z barvanjem zemljevidov?

Barvanje zemljevidov je le začetek zgodbe. Na zemljevide lahko pozabimo: predstavljajmo si, da imamo kar nek graf, sestavljen kakorsižebodi in česarsižebodi, naša naloga pa ga je pobarvati tako, da so vse povezane točke različnih barv in pri tem uporabiti čim manj različnih barv. V splošnem bomo namreč potrebovali več kot štiri, to je jasno: le predstavljajte si poln graf z osmimi točkami – ker je vsaka povezana z vsako, bomo potrebovali nič manj kot osem barv. Ker v tem primeru *ne vemo* vnaprej, da bomo potrebovali le štiri (ali celo le tri ali dve) barvi, je naša naloga bistveno težja. Imamo ogromen graf in pobarvamo ga s sedmimi barvami; kako lahko vemo, da ga ni mogoče tudi s šestimi?

Če je kdo pričakoval, da mu bomo, tako kot pri najkrajših poteh in najmanjših vpetih drevesih spet postregli z algoritmom, se moti: tudi za ta problem ne poznamo učinkovitega algoritma in verjamemo, da ga tudi nikoli ne bomo, ker ga ni.

Tečnež iz zadnje vrste pa se spet oglasi: kakšno zvezo ima to z računalništvom ali čemerkoli drugim? Čemu izgubljati čas z barvanjem grafov?

Barvanje grafov je morda še pomembnejši algoritem od iskanja poti, iskanja najkrajših poti, vpetih dreves... Na problem barvanja grafov lahko prevedemo kup zelo različnih problemov. Da ne bomo predolgi, vas tule le usmerim na gradivo: na strani http://vidra.fri.uni-lj.si/ubogi-geograf, ki sicer opisuje aktivnosti, ki jih lahko na to temo izvajamo z otroki, boste mimogrede izvedeli tudi, kako lahko z barvanjem grafov sestavljamo urnike in kako tudi reševanje sudokuja ni nič drugega kot eno samo duhamorno barvanje grafov.

Pokritje

Zadnji problem s področja grafov bomo le še omenili – toliko, da vemo zanj in da ga ne zamenjujemo s kakim minimalnimi vpetimi drevesi.

V grafih pogosto iščemo različne oblike pokritij. Med vsemi vozlišči želimo, recimo, poiskati takšno podmnožico vozlišč, da se vsaka povezava dotika enega izmed označenih vozlišč. Recimo, da želimo razpostaviti policiste po križiščih tako, da bo stal policist vsaj na enem koncu vsake ulice (lahko pa tudi na obeh). Ob tem pa želimo, da je policistov čim manj; takšnemu pokritju rečemo minimalno pokritje.

V obrnjeni različici naloge postavljamo policiste na ulice in želimo, da v vsako križišče vodi vsaj ena ulica, ki ima policista. Spet v tretji različici postavljamo policiste na križišča in želimo, da za vsako križišče velja, da ima bodisi svojega policista bodisi je policist v enem sosednjih križišč.

Za naloge iskanja minimalnih pokritij – zdaj bi lahko že uganili, da je tako, ne? – nimamo dobrih algoritmov. Tudi za te probleme velja, da s številom točk v grafu čas reševanja zelo hitro narašča in da verjamemo, da hitrejši algoritmi za ta problem ne obstajajo.

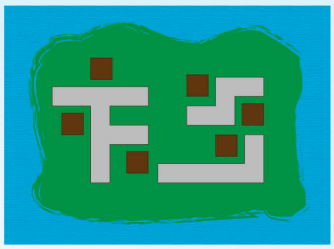
Naloge na Bobru zato lahko rešujemo le po zdravi pameti, z opazovanjem. Naloge s pokritji so – tako kot gornja – žal pogosto sestavljene tako, da je že na prvi pogled jasno, da gre za graf, tako da učenec pri reševanju ne potrebuje posebnega prebliska, temveč le bistre oči in koncentracijo.

Eden od problemov pokritij je predstavljen tudi na Vidri, v aktivnosti povezani s Piranskimi sladoledarji.

Problemov pokritij si lahko izmislimo, kolikor hočemo. Tole je različica, ki ima poleg vsega še cene. Rešujemo jih lahko, vsaj na Bobru, le po občutku.

Cestne svetilke

Bobri bi radi razsvetlili Bobrograd. Na zemljevidu na desni so s temno barvo označene hiše, ulice so sive; ostale povezave so podzemne. Bobri bi radi razsvetlili (nadzemne) ulice. Na voljo so svetilke, ki svetijo eno, dve ali tri smeri. Njihove cene so različne:




5 bevrov


6 bevrov


7 bevrov

Koliko najmanj bo stala osvetlitev?