Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

ŠTÁTNY PEDAGOGICKÝ ÚSTAV
NATIONAL INSTITUTE FOR EDUCATION

Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

Kapitoly z informatiky

Predmet: Kapitoly z informatiky

Línia: Vlastný odborový kontext informatiky a informatickej výchovy

Operačný program VZDELÁVANIE

EURÓPSKA ÚNIA
Európsky sociálny fond

Európska únia
Európsky sociálny fond

Moderné vzdelávanie pre vedomostnú spoločnosť/Projekt je spolufinancovaný zo zdrojov ES

Kapitoly z informatiky

Identifikácia modulu

Aktivita projektu:	1.3 Ďalšie vzdelávanie kvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ
Línia aktivity:	Vlastný odborový kontext informatiky a informatickej výchovy
Predmet:	Kapitoly z informatiky

Garant predmetu:

Doc. RNDr. Gabriela Andrejková, CSc., ÚINF PF UPJŠ, Košice
Gabriela.Andrejkova@upjs.sk

Autori:

Doc. RNDr. Gabriela Andrejková, CSc., ÚINF PF UPJŠ, Košice
RNDr. Michal Forišek, PhD., KI FMFI UK, Bratislava
Mgr. Juliana Šišková, KZVI FMFI UK, Bratislava
RNDr. Michal Winczer, PhD. KZVI FMFI UK, Bratislava

Zaradenie modulu

Tento modul patrí medzi voliteľné moduly a je považovaný za modul, ktorý prinesie frekventantom niektoré najnovšie metódy a poznatky z oblasti informatiky. Je zaradený po všetkých programátorských moduloch a module o počítačových systémoch.

Abstrakt modulu

Informatika je dnes považovaná za samostatnú vedeckú disciplínu. Mnohým ľuďom však táto skutočnosť uniká a za informatikov považujú ľudí, ktorí vedia pracovať s aplikáciami na počítačoch a vedia pracovať s rôznymi softvérovými systémami. Modul pozostáva z dvoch voliteľných oblastí informatiky, ktoré poukážu na vedecký výskum, ktorý prebieha v informatike. Výber oblastí bol urobený tak, aby na jednej strane uviedol frekventantov do problematiky a umožnil im robiť vlastné skúmanie malých problémov (vlastné overenie známych výsledkov), a na druhej strane ukázal na nové trendy, ktorými sa výskum v informatike zaoberá.

Prvá oblasť je venovaná skúmaniu efektívnosti algoritmov, ktorá vyústi k štúdiu metód používaných pri tvorbe efektívnych algoritmov. Táto časť poskytne metódy, ktoré sú použiteľné vo výučbe študentov stredných škôl, ktorí sa chcú zapájať do rôznych programátorských súťaží. Prvá časť je zdrojom vzorových riešení pri hľadaní čo najlepších algoritmov.

Druhá oblasť sleduje najnovšie trendy, je zameraná na štúdium aproximačných a pravdepodobnostných algoritmov, na spracovanie informácií uložených v DNA a nový model výpočtov urobených pomocou DNA molekúl. Témy vybrané do tejto časti, považujeme za veľmi zaujímavé a dôležité, pretože prinášajú v ich výskume množstvo prekvapení a očakávaní.

Obsah

Úvod

Informatika je dnes považovaná za samostatnú vedeckú disciplínu. Mnohým ľuďom však táto skutočnosť uniká a za informatikov považujú ľudí, ktorí vedia pracovať s aplikáciami na počítačoch a vedia pracovať s rôznymi softvérovými systémami. V module budú uvedené oblasti informatiky, ktoré poukážu na vedecký výskum, ktorý prebieha v informatike. Výber oblastí bol urobený tak, aby na jednej strane uviedol frekventantov do problematiky a umožnil im robiť vlastné skúmanie malých problémov (vlastné overenie známych výsledkov), a na druhej strane ukázal na nové trendy, ktorými sa výskum v informatike zaoberá.

Prvá oblasť je venovaná skúmaniu efektívnosti algoritmov, ktorá vyústi k štúdiu metód používaných pri tvorbe efektívnych algoritmov. Táto časť poskytne metódy, ktoré sú použiteľné vo výučbe študentov stredných škôl, ktorí sa chcú zapájať do rôznych programátorských súťaží.

Druhá oblasť sleduje najnovšie trendy, je zameraná na štúdium aproximačných a pravdepodobnostných algoritmov a na nový model výpočtov urobených pomocou DNA molekúl. Témy vybrané do tejto časti, menšej svojím rozsahom, považujeme za veľmi zaujímavé a dôležité, pretože prinášajú v ich výskume množstvo prekvapení a očakávaní. Prvá časť je zdrojom vzorových **riešení pri hľadaní čo najlepších algoritmov**.

Softvérové a hardvérové požiadavky a odporúčania

Niektoré algoritmy budú naprogramované v programovacích jazykoch, s ktorými sa účastníci už zoznámili, predovšetkým vo Free Pascale v prostredí Lazarus alebo Delphi.

Cieľ modulu

Priblížiť účastníkom niektoré ďalšie problémy a oblasti informatiky a aspoň čiastočne ukázať, čím sa informatika ako veda zaoberá. Modul je rozdelený do dvoch voliteľných častí. V prvej časti sledujeme metódy pre vytváranie efektívnych algoritmov a ich zložitosť. V druhej časti ukážeme nové trendy pri vytváraní algoritmov a riešení algoritmických problémov. Prvá časť bude urobená tak, aby frekventant mal pocit zapojenia sa do výskumu efektívnych algoritmov pre algoritmické problémy.

Vstupné vedomosti

Požadované prerekvizity

Sú potrebné programátorské moduly 3Prog1 - 3Prog4.

Predpokladané vstupné vedomosti, skúsenosti a zručnosti

Vstupné predpoklady: kvalifikovaní učitelia informatiky, ktorí vedia programovať, majú základné vedomosti o výpočtovej zložitosti algoritmov a problémov. Tiež sú potrebné základné vedomosti o práci klasických počítačov.

Preverenie vstupných vedomostí

Frekventant vie odpovedať na nasledujúce otázky: Po absolvovaní prvej časti, aký je počet vykonaných aritmetických operácií pri výpočte algoritmu pre daný vstup, aký je počet vykonaných overených podmienok pri výpočte algoritmu pre daný vstup, koľkokrát sa vykoná telo cyklu. Po absolvovaní druhej časti bude vedieť vysvetliť princíp pravdepodobnostných algoritmov a princíp DNA počítačov.

Kapitola 1: Efektívne algoritmy

Cieľom tohto materiálu je poskytnúť prehľad základných techník používaných pri návrhu a analýze efektívnych algoritmov. Skôr než sa im začneme venovať, však považujeme za potrebné zodpovedať jednu oveľa základnejšiu otázku: *Sú vôbec efektívne algoritmy potrebné?*

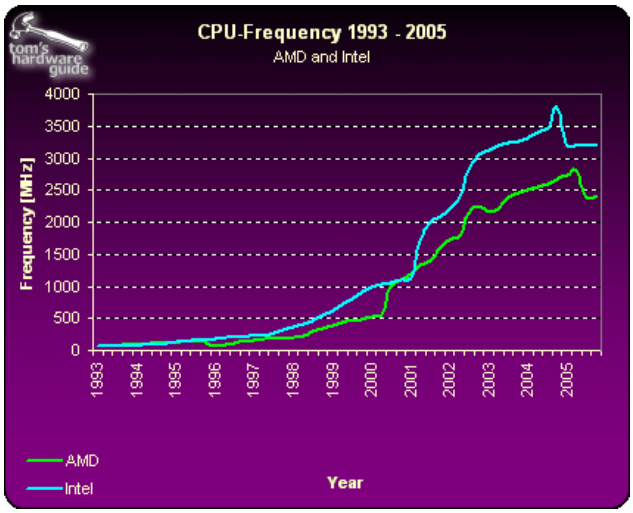
Potreba efektívnych algoritmov

Gordon E. Moore (1929–). (Zdroj: Wikipédia.)

Laický pohľad skutočne môže naznačovať, že efektívne algoritmy vôbec nepotrebujeme. Veď predsa každý rok sa výrobcovia počítačov predbiehajú v tom, kto vyrobí ešte výkonnejší procesor, ešte rýchlejšiu pamäť, ...

Tento trend je dokonca natoľko výrazný, že má aj svoje meno: Moorov zákon. Pomenovaný je podľa jedného zo zakladateľov Intelu, Gordona E. Moora, ktorý v roku 1965 predpovedal, že každé dva roky sa zdvojnásobí počet tranzistorov, ktoré sa podarí umiestniť na integrovaný obvod danej veľkosti. V súčasnosti sa názov „Moorov zákon“ používa všeobecnejšie a asi najznámejšia jeho formulácia znie: *„približne každých 18 mesiacov sa zdvojnásobí výpočtová sila počítačov bežne dostupných v obchodoch“*.

Uvedieme jeden príklad: približne pred 15 rokmi, 27. marca 1995, prišiel Intel na trh s najnovším modelom procesoru Pentium s taktovacou rýchlosťou 120 MHz. V súčasnosti sú bežne dostupné procesory, ktoré majú taktovaciu rýchlosť vyše 3 GHz (vyše 25-násobný nárast), štyri nezávislé jadrá, dve úrovne vyrovnávacej pamäte a iné vylepšenia, ktoré dokopy skutočne spôsobujú, že dnešné počítače sú približne tisíckrát výkonnejšie ako tie spred 15 rokov.



Obrázok 1: Vývoj taktovacej frekvencie procesorov. (Zdroj: Tom‘s Hardware.)

Keď teda máme k dispozícii takto výkonné počítače (a nádej, že v budúcnosti budú ešte lepšie), potrebujeme teda na niečo efektívne algoritmy?

Áno, potrebujeme, ba dokonca viac ako kedykoľvek predtým.

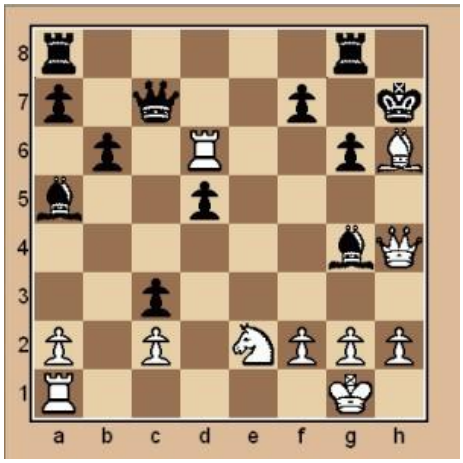
Spolu s rastom výpočtovej sily bežných počítačov rastie totiž aj objem dát, ktoré naša spoločnosť potrebuje denne spracovať.

Keď sa opäť pozrieme 15 rokov do minulosti, do roku 1995, ocitneme sa v situácii, keď sa kapacita bežne dostupných pevných diskov blížila k 1 GB a dáta sme bežne prenášali na 3.5″ disketách. V súčasnosti sú už dostupné pevné disky s kapacitou výrazne presahujúcou 1 TB. (Opäť ide o vyše tisícnásobný nárast.)

Podľa The Official Google Blog prvý index webstránok, ktorý algoritmy vyhľadávača Google zostrojili v roku 1998, obsahoval 26 miliónov webstránok. Okolo roku 2000 tento počet dosiahol miliardu a v lete 2008 už Google poznal dokonca bilión (10^{12}) rôznych webstránok. Množstvo webstránok teda len za 10 rokov narástlo takmer 50000-krát – omnoho výraznejšie ako výpočtová sila počítačov! A to nehovoríme o tom, že dnešné webstránky sú často plné multimediálneho obsahu, a teda sú neporovnateľne väčšie ako tie z roku 1998.

A nielen samotné množstvo spracúvaných dát nás núti hľadať čo najefektívnejšie algoritmy. Druhým dôvodom je obtiažnosť problémov, ktoré sa snažíme pomocou počítačov riešiť. Častokrát je počet všetkých možných riešení problému natoľko obrovský, že keď hľadáme to najlepšie z nich, nemôžeme si ani zďaleka dovoliť vyskúšať ich všetky.

Dobre si to môžeme ilustrovať na príklade programov, ktoré hrajú šach. Súčasné šachové programy sú neuveriteľne komplikované (obsahujú napríklad samostatné knižnice rôznych otvorení a koncoviek), ale všetky majú spoločné jadro: Keď sa takýto program rozhoduje, aký ťah má zahrať, spraví to tak, že začne prezerať možné priebehy nasledujúcich ťahov oboch hráčov. Keď mu vyprší čas, vyberie si ten ťah, ktorý na základe možných vývojov partie vyhodnotí ako najlepší.



Obrázok 2: Mat 3. Ťahom
(Zdroj: Online Chess Strategy.)

Koľko toho stihne takýto program prezrieť? Nech trebárs v každej pozícii existuje v priemere 10 rozumných možností, ako potiahnuť. Ak sa budeme pozerať na nasledujúce dva polťahy, je $10 \times 10 = 100$ možností, ako budú vyzerať – na každý z ťahov počítača môže jeho protihráč zareagovať 10 spôsobmi. Ak by sme chceli prezrieť všetky možné priebehy nasledujúcich troch polťahov, už ich počet narastie na 10^3 a tak ďalej.

Polťah je správny šachový pojem pre ťah jedného hráča.

Počet priebehov partie teda rastie exponenciálne. Už napríklad pri deviatich polťahoch dostávame odhadom miliardu možných priebehov. Tento počet je tak na hranici toho, čo dnešný počítač zvládne prezrieť za jednu minútu. Zjednodušene teda môžeme povedať: Ak by sme nášmu šachovému programu dovolili minútu počítať, stihol by vyhodnotiť, čo všetko sa môže stať v nasledujúcich deviatich polťahoch.

Ako veľmi nám pri hraní šachu pomôže, ak si kúpime nový počítač? Odpoveď je jednoduchá: na to, aby sa nový počítač stihol za minútu pozrieť čo i len o jediný polťah ďalej, musí byť 10-krát rýchlejší od toho, ktorý máme teraz. Ak sa aj v budúcnosti bude výpočtová sila počítačov správať podľa Moorovho zákona, dočkáme sa takého počítača až približne o 5 rokov.

S podobnou situáciou sa často stretneme pri spracúvaní dát v praxi: Ak nepoznáme efektívny algoritmus, ktorý by náš problém riešil, samotný nárast výpočtovej sily počítačov nás nemá ako zachrániť. A práve tu prichádza k slovu návrh efektívnych algoritmov: ak v praxi narazíme na ťažký problém, potrebujeme ho vedieť analyzovať a odhaliť čo najlepší spôsob, ako ho algoritmicky riešiť.

Časová zložitosť

Skôr než sa dostaneme k samotnému návrhu algoritmov, potrebujeme spraviť jednu dôležitú odbočku. Keď totiž vymyslíme k nejakému problému viacero rôznych algoritmov, budeme potrebovať vyhodnotiť, ktorý z nich je lepší. Ako ale algoritmy porovnávať?

Ako prvý asi každému napadne priamočiary prístup: Keď máme vymyslené dva algoritmy, skúsime oba naprogramovať a spustiť. Ktorý skôr skončí, ten je lepší.

Tento prístup je však veľmi nepraktický, a to hneď z viacerých dôvodov:

- Vyžaduje presne to, čomu sa chceme vyhnúť. My nechceme programovať všetky riešenia, ktoré nám napadnú, práve naopak. Chceme vybrať to najlepšie z nich a naprogramovať len to.
- Je nepresný. Rýchlosť behu programu závisí od mnohých faktorov, ako napríklad momentálna záťaž procesora inými úlohami, množstvo voľnej pamäte, architektúra procesora a podobne.
- Nemusí byť použiteľný. Môže sa stať, že dáta sú natoľko veľké, že takéto praktické testy by trvali neúnosne dlho. (Obzvlášť ak ani jeden z našich algoritmov nie je dostatočne dobrý.)
- Je nedostatočný. Tým, že programy spustíme pre nejaké konkrétne vstupné dáta, sa dozvieme len to, ako sa správajú pre tento konkrétny vstup. Čo nám ale zaručí, že aj hocijaké iné dáta zvládne ten program spracovať rovnako rýchlo?

Pri analýze algoritmov sa preto bežne používa iný prístup: Nebudeme používať žiaden konkrétny počítač, budeme sa na vykonávanie algoritmu dívať ako na postupnosť logických krokov. Čím menej krokov potrebujeme spraviť pri vykonávaní daného algoritmu, tým lepší bude.

Počítanie počtu krokov

Spočítať, koľko presne krokov spraví daný algoritmus (pre dané vstupné dáta), môže byť už aj pre veľmi jednoduché algoritmy veľmi zložitá úloha. Bude preto vhodné začať jednoduchou návodnou úlohou.

Zadanie 1	Na obchodnom stretnutí sa stretlo 20 podnikateľov. Každý z nich si podal ruku so všetkými ostatnými. Koľko podaní rúk sa dokopy uskutočnilo?

Riešenie tejto úlohy je jednoduché: dvojíc podnikateľov je $\binom{20}{2} = \frac{20 \times 19}{2} = 190$ a každá dvojica si raz podala ruku, preto podaní rúk bolo práve 190.

Zadanie 2	Koľko hviezdičiek vypíše nasledujúca časť programu, ak premenná `N` obsahuje hodnotu `20`?
	
	`for i := 1 to N do`
	`  for j := i + 1 to N do`
	`  write('*');`

Zadanie 3	Keby sa hodnota premennej `N` načítavala z klávesnice, ako by počet hviezdičiek závisel od hodnoty, ktorú používateľ zadá?

Toto je vlastne tá istá úloha, len teraz je zamaskovaná v inom šate. Aby sme lepšie videli, čo sa pri vykonávaní programu deje, upravme si ho nasledovne:

```
for i := 1 to N do
  for j := i+1 to N do
    writeln(i,' ',j);
```

Skúste pre hodnotu $N = 5$ ručne odsimulovať tento upravený program. Aký výstup dostanete? Koľko má riadkov?

Keby sme si podnikateľov z predchádzajúcej úlohy očíslovali od 1 po 20, tento program by práve raz vypísal každú dvojicu ich čísel. Tento upravený program teda vypíše presne 190 riadkov, a teda pôvodný program vypíše 190 hviezdičiek.

Mimochodom, úlohy typu „koľko hviezdičiek tento program vypíše“ sú výborným nástrojom pri výučbe analýzy algoritmov — a to od úloh, kde na vyriešenie stačí jednoduchá simulácia, až po úlohy veľmi komplikované. Ľahko sa týmto spôsobom overuje, či žiaci rozumejú rôznym konceptom. Vhodné vloženie riadku vypisujúceho hviezdičky do programu umožňuje zvoliť si časť programu, na ktorú sa pri riešení treba sústrediť.

Ukážeme si ešte jednu úlohu tohto typu, tentokrát trochu zložitejšiu.

Zadanie 4

Koľko hviezdičiek vypíše nasledujúca časť programu pre `S = 'abeceda zjedla deda'` a `T ='eden'`?

```
for i := 1 to 16 do
  for j := 1 to 4 do begin
    write('*');
    if S[i+j-1] <> T[j] then break;
    if j = 4 then halt;
  end;
```

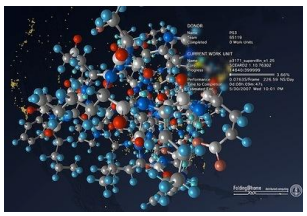
O niečo náročnejšia úloha: Viete reťazce `S` a `T` zmeniť na iné reťazce rovnakej dĺžky tak, aby tento program vypísal viac ako 50 hviezdičiek?

(Správna odpoveď je 21. Uvedený program hľadá výskyt reťazca `T` v reťazci `S` tak, že postupne vyskúša a skontroluje všetky pozície, kde tento výskyt môže začínať. Každý výpis hviezdičky zodpovedá jednému porovnaniu znakov.)

Praktický pohľad

V predchádzajúcej časti sme pre veľmi jednoduché programy dokázali presne spočítať, koľko krokov urobia. Ale potrebuje programátor z praxe niekedy takéto presné výsledky?

Programátor väčšinou nevie presne, aké dáta bude jeho program spracúvať. V lepšom prípade vie približne odhadnúť ich veľkosť, v horšom ani to nie.



Napríklad programátor Andrej píše modul pre prihlasovanie na webový portál, ktorý má dnes $100\,000$ užívateľov. Počas najrušnejšej hodiny dňa sa ich prihlási okolo $5\,000$. Andrej teda približne vie, aké veľké dáta bude jeho program spracúvať, a vie, že jeho program musí byť dosť rýchly na to, aby stihol spracovať každého užívateľa za niekoľko desatín sekundy.

Bibiana píše program, ktorý bude simulovať skladanie proteínov. Čím bude jej program rýchlejší, tým viac a väčších simulácií stihne v rozumnom čase spraviť, a tým skôr sa vedci dostanú k dostatočným dátam.

Ak máte počítač, ktorý je v noci nevyužitý, môžete na ňom nechať bežať simuláciu skladania proteínov, a prispieť tak k významným pokrokom v zdravotníctve. Viac na stránke http://folding.stanford.edu .

Obaja v skutočnosti nepotrebujú vedieť presný počet krokov, ktorý ich program v danej situácii spraví. Stačil by im spôsob, ako približne zistiť, ako dlho ich program v danej situácii pobeží, resp. či je dostatočne rýchly.

Ako definovať časovú zložitosť?

V zadaní 2 sme si mohli všimnúť, že počet krokov, ktoré vykonáme pri simulácii dotyčného algoritmu, závisí od hodnoty premennej N. V tomto konkrétnom prípade by sme napríklad mohli povedať, že pre vstupnú hodnotu N daný program spraví $1.5N^2 - 0.5N + 1$ krokov.

Podobnú úvahu vieme spraviť pre ľubovoľný algoritmus A. Vždy sa dá definovať nasledujúca funkcia k_A: Nech v je nejaký vstup, na ktorom môžeme vykonať

algoritmus A. Potom $k_A(v)$ je počet krokov, ktoré pri tom spravíme.

Ako sme však uviedli v predchádzajúcej časti, takýto pohľad je zbytočne presný. Všimnime si napríklad zadanie 4. Ako počet vypísaných hviezdičiek, tak aj celkový počet krokov algoritmu závisí nie len od dĺžky reťazcov `S` a `T`, ale aj od toho, ako presne tieto reťazce vyzerajú. Ľahko nahliadneme, že už pre takýto jednoduchý algoritmus neexistuje žiaden pekný jednoduchý matematický zápis funkcie k_A.

Našťastie ho ani nepotrebujeme poznať. To, čo nám o algoritme stačí vo väčšine prípadov vedieť, je jeho **najhorší možný prípad**. V prípade zadania 3 by nás teda mohla zaujímať odpoveď na otázku: Ak má reťazec S dĺžku x a reťazec T dĺžku y znakov, koľko **najviac** krokov spraví uvedený program?

Ak by sme poznali odpoveď na túto otázku, môžeme sa na ňu dívať ako na **záruku**: nech konkrétne reťazce vyzerajú, ako len chcú, my vieme zaručiť, že náš program sa po takom-a-takom počte krokov skončí.

Pod slovami „veľkosť vstupu“ si môžeme predstaviť jednoducho veľkosť (v bajtoch) súboru, v ktorom by bol daný vstup uložený.

A presne takto sa v oblasti návrhu a analýzy efektívnych algoritmov **časová zložitosť** definuje: Časovou zložitosťou algoritmu A je funkcia t_A taká, že hodnota $t_A(N)$ je najmenší počet krokov, ktoré A stačia na spracovanie ľubovoľného vstupu veľkosti N. (Alebo inými slovami, keby sme A vykonali pre všetky možné vstupy veľkosti N a zakaždým si zapísali počet vykonaných krokov, tak by $t_A(N)$ bolo maximum z týchto počtov.)

Časovú zložitosť stačí odhadnúť

Súčasné procesory zvládajú vykonať približne miliardu inštrukcií za sekundu. Pomocou tohto približného údaja môžeme jednoducho z časovej zložitosti programu odhadnúť, ako dlho by bežal na súčasnom počítači. Napr. program s časovou zložitosťou $5N^2 + 4N$ by pre $N = 10\,000$ bežal približne pol sekundy. (Samozrejme, konkrétna hodnota závisí od konkrétneho počítača.)

Skúsme si teraz položiť otázku opačne: ak vieme, akú má náš program časovú zložitosť a vieme, ako dlho ho sme ochotní nechať bežať, aký najväčší vstup ešte stihne spracovať?

V nasledujúcej tabuľke uvádzame názorný prehľad odpovedí na túto otázku pre niekoľko zaujímavých časových zložitostí.

Zoberme si napríklad Zadanie 2 s vypisovaním hviezdičiek. Toto riešenie patrí pod zložitosť N^2. Chceme, aby nám program zbehol do hodiny. Z tabuľky môžeme vyčítať, že najväčší vstup, aký môžeme zadať, aby sme sa do hodiny dočkali výsledku, je číslo $2\,000\,000$. Ak by sme potrebovali výsledok do sekundy, vstup by musel byť najviac $30\,000$.

Na tabuľku sa môžeme ešte pozerať ináč. Máme rôzne efektívne riešenia tej istej úlohy, častokrát to býva dvojica N^2 a 2^N. Ak chceme výsledok do sekundy, v prvom prípade môžeme zadať vstup veľkosti $30\,000$ a v druhom najviac 30.

čas/zložitosť	N	N^2	N^3	2^N	$N!$
milisekunda	1 000 000	1 000	100	20	10
sekunda	1 000 000 000	30 000	1 000	30	12
minúta	∞	250 000	4 000	35	14
hodina	∞	2 000 000	15 000	41	15
deň	∞	9 000 000	44 000	46	16
mesiac	∞	51 000 000	130 000	51	17
rok	∞	170 000 000	310 000	54	18
tisícročie	∞	∞	3 100 000	64	21

Tabuľka 1: Najväčšie N, pre ktoré program s danou časovou zložitosťou skončí v danom čase.

(Veľké hodnoty sú zaokrúhlené, väčšie ako miliarda sú nahradené symbolom ∞.)

Ak sa na algoritmy dívame z tejto perspektívy, ľahko si všimneme, že príliš nezáleží na tom, či má náš algoritmus časovú zložitosť N^2 alebo $N^2 + 100N$. Totiž akonáhle zoberieme dostatočne veľké N, hodnota N^2 bude rádovo väčšia ako $100N$, a teda tých $100N$ krokov navyše môžeme zanedbať. Keby sme do našej tabuľky pridali nový stĺpec pre zložitosť $N^2 + 100N$, vyzeral by skoro identicky ako stĺpec pre N^2.

A takisto nie je žiaden výrazný rozdiel medzi algoritmami s časovou zložitosťou N^3 a $7N^3$. Presný čas behu samozrejme závisí od presnej rýchlosti nášho počítača, ale ešte stále nám naša tabuľka povie, čo rádovo môžeme očakávať. Ak chceme spracovať vstupné dáta veľkosti $N = 40\,000$ a náš program má časovú zložitosť $7N^3$, bude mu to trvať niekoľko dní. Ak by bolo $N = 7\,000\,000$, rovno vieme povedať, že sa odpovede nedožijeme.

Zadanie 5

Programátor Cyril napísal program, ktorý načíta súradnice stredov a polomery N rôznych kružníc v rovine a následne zistí, koľko majú dokopy priesečníkov. Svoj program vám popísal nasledovne: *„Pre každú dvojicu kružníc vypočítam vzdialenosť ich stredov a porovnám ju so súčtom polomerov. Podľa toho, či je väčšia, rovnaká alebo menšia, som našiel 0, 1 alebo 2 priesečníky.“*

Odhadnite, ako rýchlo tento program spracuje $1\,000$ kružníc. A čo $100\,000$ kružníc?

Cyril pri návrhu svojho programu urobil drobnú chybu. Viete nájsť vstup, pre ktorý jeho program nedá správnu odpoveď?

Všetkých dvojíc kružníc je $\binom{N}{2} = \frac{N(N-1)}{2}$. Cyrilov program pre každú dvojicu kružníc spraví konštantný počet krokov: niekoľko matematických operácií a jedno vyhodnotenie podmienky. Časová zložitosť tohto programu je teda zrejme nejakou kvadratickou funkciou premennej N.

Tisíc kružníc by teda tento problém mal bez problémov zvládnuť spracovať za výrazne menej ako sekundu. Pri stotisíc kružniciach bude čas behu rádovo minúta.

Zadanie 6

Danica vlani dostala od šéfa za úlohu zistiť, či nemajú v databáze zákazníkov nejakého zákazníka omylom viackrát. Túto úlohu vyriešila tak, že napísala program, ktorý porovnal každú dvojicu zákazníkov. Keď ho spustila, program necelé dve minúty bežal a na záver vypísal, že žiadne duplikáty nenašiel.

Dnes má firma trikrát toľko zákazníkov ako pred rokom. Ak by dnes Danica spustila svoj program (na tom istom počítači ako vlani), ako dlho by čakala, kým program dobehne?

O Danicinom programe opäť môžeme rozumne predpokladať len jediné: že jeho časová zložitosť je kvadratickou funkciou od počtu zákazníkov. Teraz už len stačí, keď si uvedomíme, že nič viac ani vedieť nepotrebujeme.

Ak si počet zákazníkov označíme N, tak pre dostatočne veľké N už bude časová zložitosť Danicinho programu takmer presne priamo úmerná N^2.

Trochu formálnejšie: ak pre N zákazníkov program spraví približne kN^2 krokov, pre $3N$ zákazníkov to bude približne $k(3N)^2 = 9kN^2$, čiže 9-krát viac.

My síce nepoznáme koeficient tejto priamej úmernosti, ani ho však poznať nemusíme. Stačí si uvedomiť, že počet krokov, ktoré program vykoná pre $3N$ zákazníkov, musí byť s rovnakým koeficientom priamo úmerný číslu $(3N)^2$. A keďže $(3N)^2 = 9N^2$, bude to programu trvať 9-krát dlhšie ako v prvom prípade – čiže približne štvrť hodiny.

Formálne značenie

Emil a Filoména sú vedci. Obaja nedávno vymysleli nové algoritmy na triedenie čísel a zistili ich časové zložitosti. Časovú zložitosť Emilovho programu označíme e a

Filoméninho f. Teda vieme, že Emilov algoritmus potrebuje na utriedenie N čísel spraviť v najhoršom prípade $e(N)$ krokov, zatiaľ čo Filoménin algoritmus ich potrebuje spraviť $f(N)$. Ako povedať, ktorý z nich je lepší?

Ak máme na mysli nejaké konkrétne použitie, pri ktorom približne vieme, ako veľa čísel budeme triediť, stačilo by porovnať zodpovedajúce hodnoty $e(N)$ a $f(N)$. Čo ak to ale nevieme? Dá sa niekedy povedať, že jeden z týchto algoritmov je *„vo všeobecnosti lepší“* ako druhý?

Pri formálnejšom prístupe sa na tomto mieste používa matematický pojem limity: Funkcia e rastie rádovo rýchlejšie ako f vtedy, ak je limita podielu $\frac{e(N)}{f(N)}$ pre N rastúce do nekonečna rovná nekonečnu.

Ukazuje sa, že áno. Pozerajme sa na podiel $\frac{e(N)}{f(N)}$. Táto hodnota nám hovorí, koľkokrát je pre dotyčné N Emilov algoritmus pomalší ako Filoménin. Položme si teraz otázku, čo sa stane, ak budeme N postupne zväčšovať. Ak hodnota podielu $\frac{e(N)}{f(N)}$ porastie cez všetky medze, znamená to, že Emilov algoritmus je čím ďalej, tým výraznejšie horší ako ten Filoménin. A teda až na konečne veľa malých prípadov sa vždy viac oplatí použiť Filoménin algoritmus.

Ak teda navrhujeme algoritmus v situácii, keď nevieme, kto ho kedy použije a na akých veľkých dátach, snažíme sa o to, aby bol vo všeobecnosti čo najlepší — teda aby jeho časová zložitosť rástla čo najpomalšie.

Príklad: Ak má Emilov algoritmus časovú zložitosť $e(N) = N^3 + N^2 + 1$ a Filoménin $f(N) = 100N^2 + 100N$, tak platí $\frac{e(N)}{f(N)} = \frac{N}{100} + \frac{1}{(100N^2+100N)}$. Pre pár malých hodnôt N je Emilov algoritmus rýchlejší, ale už napr. pre $N = 200$ je približne dvakrát pomalší od Filoméninho a pre $N = 10\,000$ bude už Emilov algoritmus bežať až stokrát dlhšie.

Všimnite si, že v predchádzajúcom príklade by sa nič podstatné nezmenilo, ak by e bola iná kubická funkcia a f iná kvadratická funkcia. Ich podiel by opäť bol približne rovný nejakej lineárnej funkcii, a teda čím väčšie vstupné dáta by sme uvažovali, tým horší by bol Emilov algoritmus v porovnaní s Filoméninym.

Keď sa v odbornej literatúre hovorí o časovej zložitosti algoritmov, používa sa pri tom na zjednodušenie zápisu notácia prevzatá z matematickej analýzy. Najjednoduchšie používané značenie je veľké písmeno O, ktorého význam je definovaný nasledovne:

Pozor! Zápis $f(n) = O(g(n))$ je príkladom preťaženia operátora $=$ v matematike. Neznamená rovnosť, ale „patrí do“, teda $\in$. $O(g(n))$ je množina funkcií.
Takže musíme dávať pozor, lebo v tomto zápise nie je $=$ komutatívne.

Nech f a g sú rastúce funkcie na prirodzených číslach. Potom píšeme $f(n) = O(g(n))$, ak existuje kladná konštanta c taká, že pre všetky dostatočne veľké n platí $f(n) \leq c \cdot g(n)$. Tento zápis čítame *„funkcia f je veľké O od funkcie g“*. Preložené z matematickej reči, tento zápis hovorí, že funkcia f rastie nanajvýš rádovo tak rýchlo ako funkcia g.

Toto značenie používame vtedy, keď chceme zhora ohraničiť, ako rýchlo rastie nejaká funkcia (ktorej presné vyjadrenie často nepoznáme).

Príklady použitia:

- *„Časová zložitosť Cyrilovho programu zo zadania 2 je $O(N^2)$.“*
- *„Každá kubická funkcia je $O(N^3)$.“*
- *„Ak $f(n) = 2^n$ a $g(n) = n!$, tak $f(n) = O(g(n))$, ale neplatí $g(n) = O(f(n))$.“*
  (Slovne: Funkcia 2^n rastie nanajvýš tak rýchlo ako $n!$, ale naopak to neplatí.)
- *„$n^2 + 10n = O(n^{47})$.“*
  (Všimnite si, že veľké O predstavuje len horný odhad, ten môže byť niekedy veľmi voľný.)

Zhrnutie

Nájsť horný odhad časovej zložitosti je často **výrazne** jednoduchšie ako ju určovať

presne. Všimnime si napríklad náš starý známy program:

```
for i := 1 to N do
  for j := i + 1 to N do
    write('*');
```

Vidíme, že obsahuje dva vnorené for-cykly, pričom rozsah každého z nich je nanajvýš N. Dokopy teda tento algoritmus vykoná nanajvýš N^2 iterácií, a teda je jeho časová zložitosť nanajvýš kvadratická od N. Toto môžeme matematicky zapísať: „*tento algoritmus má časovú zložitosť* $O(N^2)$“.

Takto v praxi bežne vyzerá analýza zložitosti algoritmu. Celý postup si môžeme zhrnúť nasledovne:

- Zistíme, čo a ako ten algoritmus robí.
- Čo najtesnejšie zhora odhadneme počet krokov, ktoré spraví.
- Získaný odhad zapíšeme pomocou matematickej notácie (alebo slovne).

Čo sme sa naučili

Ak chceme, aby naše programy bežali rýchlo, či aspoň, aby sme sa dožili ich úspešného ukončenia, mali by sme dbať na ich efektivitu. Tá sa dá merať časovou zložitosťou, čo je časové ohraničenie vzhľadom na veľkosť vstupu.

Úlohy na precvičenie

Zadanie 7

Premenná `N` obsahuje hodnotu `20` a premenná `A` je dostatočne veľké pole celých čísel.

Koľko najviac hviezdičiek môže vypísať nasledujúca časť programu? A koľko najmenej?

```
for i := 1 to N do
  read( A[i] );
for j := 1 to N do
  for k := j + 1 to N do
    if A[j] <> A[k] then
      write('*');
```

Zadanie 8

Je pravdivé tvrdenie: „*Časová zložitosť programu zo zadania 7 je kubickou funkciou premennej* N“?

Ak áno, prečo? Ak nie, viete ho opraviť?

Zadanie 9

Akých 20 hodnôt treba zadať programu zo zadania 7, aby vypísal presne 99 hviezdičiek?

(Riešení je veľa, stačí nájsť jedno ľubovoľné.)

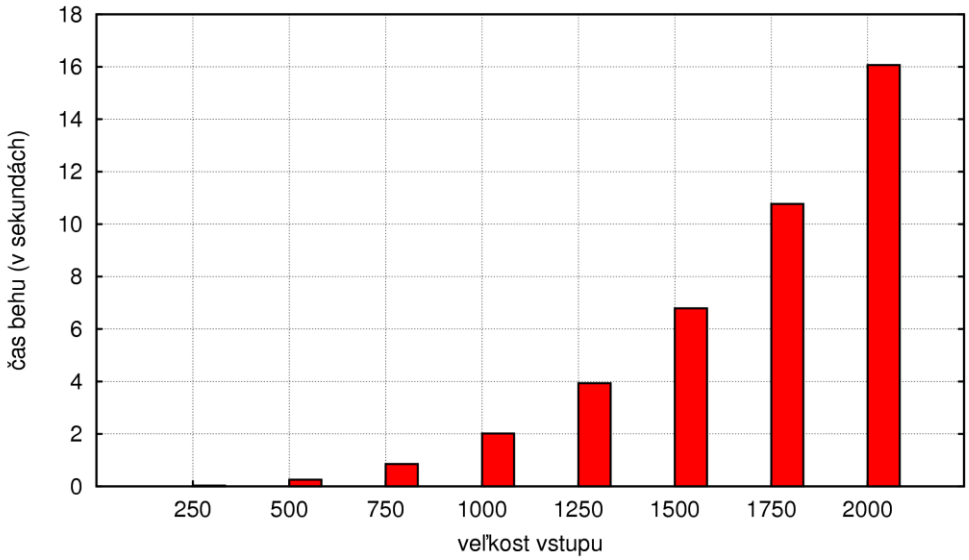
Zadanie 10

Odhadnite, ako dlho by program zo zadania 7 bežal pre $N = 10\,000$.

(Predpokladajte, že načítanie čísel zo vstupu ho nijak nezdrží.)

Tiež odhadnite, pre rádovo aké najväčšie N by sa tento program skončil do hodiny.

Zadanie 11	Našli sme na disku neznámy program. Vyskúšali sme ho spustiť na vstupoch rôznej veľkosti. V grafe na obrázku 3 je pre každú veľkosť vstupu, ktorú sme skúšali, zaznačený nameraný čas behu. Čo viete na základe tohto grafu usúdiť o časovej zložitosti nášho neznámeho programu?



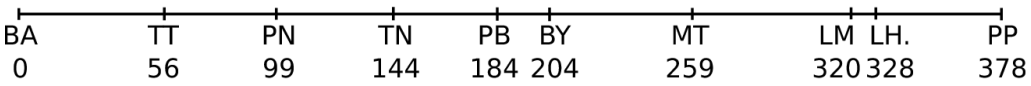
Obrázok 3: Závislosť času behu programu (Zadanie 11) od veľkosti vstupu

Pažravé algoritmy

Niektoré typy algoritmických príkladov môžeme riešiť pažravou metódou. Túto metódu postupne objavíme riešením príkladov tohto typu.

Úloha s benzínovými čerpadlami

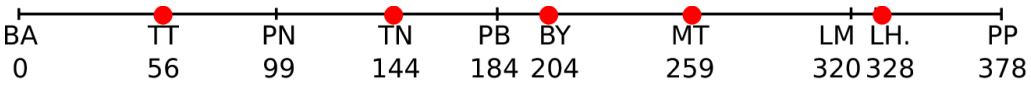
Zadanie 12	Cestujeme z Bratislavy do Popradu. Na obrázku je nakreslená trasa, benzínové čerpadlá na nej a ich vzdialenosť na trase. Auto prejde najviac 100 km na plnú nádrž. Zistite, na ktorých benzínových čerpadlách máme tankovať, aby sme čo najmenší počet krát zastavovali.



Zoberme si papier a pero a skúšajme si vyberať, na ktorých čerpadlách zastavíme.

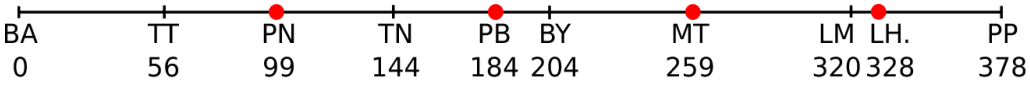
Na úvod môžeme skúsiť vyberať zastávky len tak „od oka“, nech vidíme, či sa nám vôbec podarí do Popradu dostať.

V Bratislave máme plnú nádrž, ktorá nám vydrží 100 km. Prvýkrát natankujeme v **Trnave** na 56. kilometri. Keďže opäť máme plnú nádrž, vieme prejsť nanajvýš o 100 kilometrov ďalej. Teda môžeme prejsť cez Piešťany a zastať v **Trenčíne** na 144. kilometri. Následne ešte zastavíme napr. v **Bytči** na 204. kilometri, v **Martine** na 259. kilometri a v **Liptovskom Hrádku** na 328. kilometri. Po natankovaní v Liptovskom Hrádku sa už úspešne dostaneme do Popradu.



Práve sme ukázali, že naša úloha má riešenie a našli sme jedno konkrétne s piatimi zastávkami. Je však toto riešenie optimálne, alebo to ide aj lepšie?

Skúsme na to ísť nejako systematickejšie. Napríklad si môžeme povedať, že zakaždým skúsime ísť čo najďalej, vždy až po poslednú možnosť natankovať. Pri tomto riešení prvýkrát natankujeme v **Piešťanoch** na 99. kilometri, potom v **Považskej Bystrici** na 184. kilometri, v **Martine** na 259. kilometri a v **Liptovskom Hrádku** na 328. kilometri. Takto sa vieme do Popradu dostať len so štyrmi zastávkami.

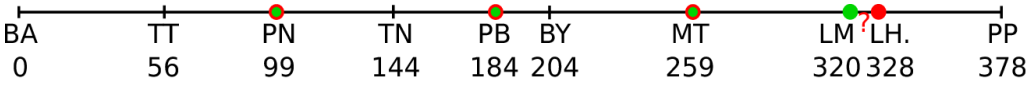


Toto riešenie pôsobí celkom presvedčivo – v každom kroku sme predsa vybrali najlepšiu možnosť, tak asi už žiadne iné, lepšie riešenie existovať nebude. Pokúsme sa teda túto intuíciu nejak presnejšie sformulovať a zdôvodniť, že práve nájdené riešenie je najlepšie možné.

Nezabudnite si vyskúšať ešte pár možností. Skúšanie pomáha nachádzať protipríklady a princípy, o ktoré sa v dôkazoch opierame.

Zoberme si ľubovoľné optimálne riešenie. Dokážeme, že ho vieme upraviť tak, aby sme (bez zmeny počtu zastávok) z neho vyrobili to naše riešenie. Pozrime sa na prvú zastávku, v ktorej sa tieto dve riešenia líšia. V tom našom riešení sme išli najďalej ako sa len dalo, v tom optimálnom sme teda museli zastať niekde skôr. Upravme teraz optimálne riešenie tak, že túto zastávku posunieme až do toho istého mesta ako v našom riešení. Zjavne sme tým nič nepokazili – úsek končiaci v posunutom meste má nanajvýš 100 km, lebo sme ho vedeli v našom riešení prejsť, a úsek začínajúci v posunutom meste sme skrátili. Postupným opakovaním tohto postupu vieme ľubovoľné optimálne riešenie prerobiť na to naše.

Konkrétny príklad: Uvažujme optimálnu postupnosť tankovaní **Piešťany**, **Považská Bystrica**, **Martin** a **Liptovský Mikuláš**. Rozdiel oproti nášmu riešeniu je len v poslednej zastávke. Ak tú zmeníme na **Liptovský Hrádok**, nič nepokazíme. (Dôjsť z Martina do Liptovského Hrádku vieme. A keďže sme vedeli dôjsť z Liptovského Mikuláša do Popradu, tým skôr to pôjde aj z Liptovského Hrádku.)



Uvedomme si dôležitosť tohto argumentu. Ten hovorí, že nebola náhoda, že náš postup fungoval. Práve naopak, náš postup musí fungovať v každej podobnej situácii. Vedeli by sme ním vyriešiť napríklad aj nasledujúcu úlohu:

Na zamyslenie: Predstavte si, že by sme cestu zostrojovali od konca. Spomedzi miest, odkiaľ vieme s plnou nádržou prísť do cieľa, vyberieme ako zastávku to, ktoré je od cieľa najďalej. Následne celý postup opakujeme s práve vybratým mestom ako cieľom atď.

Nájde tento postup vždy optimálne riešenie? Bude to vždy to isté riešenie ako dostaneme pôvodným prístupom?

Zadanie 13

Cestujeme z Bratislavy do Lisabonu. Vieme, kde sa na nami zamýšľanej trase nachádzajú benzínové čerpadlá, aj to, koľko najviac kilometrov prejde naše auto na plnú nádrž.

Zistite, na ktorých benzínových čerpadlách máme tankovať, aby sme čo najmenší počet krát zastavovali.

Časová zložitosť algoritmu, ktorý podľa našej stratégie zostrojí optimálne riešenie, je lineárna závislá od počtu benzínových čerpadiel na ceste.

Úlohy s mincami

Zadanie 14

V obchode chceme zaplatiť nákup v hodnote 5.69 EUR. Používame bežnú sadu euro mincí, t. j. mince v hodnotách 1, 2, 5, 10, 20, 50 centov a 1, 2 eurá. Z každej hodnoty máme dostatočne veľa mincí. Aké mince použiť na zaplatenie vyššie spomenutej sumy, aby sme dokopy použili čo najmenej kusov?

Sumu sme zaplatili nasledovne: 2 × 2 euro, 1 × 1 euro, 1 × 50 centov, 1 × 10 centov, 1 × 5 centov a 2 × 2 centy. Postupovali sme teda pažravo: vždy sme použili najväčšiu mincu, ktorú sme mohli použiť a pokračovali s menšou sumou. Tento pažravý algoritmus platenia ľahko naprogramujeme:

Riešenie zadania 14

```
const Mince : array [1..8] of longint =
      (200, 100, 50, 20, 10, 5, 2, 1);

procedure Platba ( SumaVCentoch : longint );
var AktualnaSuma, m : longint;
begin
  AktualnaSuma := SumaVCentoch;
  m := 1;
  while AktualnaSuma > 0 do begin
    if Mince[ m ] <= AktualnaSuma then begin
      writeln ( Mince[ m ] );
      AktualnaSuma := AktualnaSuma -
                      Mince[ m ];
    end else
      inc(m);
  end;
end;
```

Rovnako ako v predchádzajúcej úlohe, aj teraz stojíme pred otázkou: je tento pažravý algoritmus skutočne optimálny? Zaplatí ľubovoľnú sumu najmenším možným počtom mincí, či nie?

Až na dve výnimky sú euro mince navrhnuté tak, že každá nasledujúca nominálna hodnota je násobkom predchádzajúcej. A vtedy je zjavné, že sa menšie mince používať oplatí, až keď naozaj musíme. Keby sme napríklad platili 1.50 EUR, neoplatí sa nám to robiť pomocou 50-centových mincí. Totiž namiesto dvoch z nich môžeme použiť jednu mincu s hodnotou 1 euro.

Spomínanými výnimkami sú mince s hodnotami: 2 a 5 centov a 20 a 50 centov: 50 nie je násobkom 20. Ale ľahko nahliadneme, že ani tieto mince nám nič nepokazia – stačí overiť, že pri sumách 60, 70, 80 a 90 centov sa aj tak vždy 50-centovú mincu použiť oplatí.

So sadou euro mincí sa ešte trochu pohráme, aby sme lepšie porozumeli pažravým algoritmom.

Zadanie 15

Nájdite **najmenšiu** sumu, na ktorej zaplatenie potrebujeme použiť minimálne 8 mincí.

Riešenie neuvádzame, nech sa zvedavý čitateľ potrápi ☺.

Vráťme sa k zadaniu úlohy 14 a položme si otázku: je to náhoda, že pažravý algoritmus funguje, či by to tak bolo pre ľubovoľné nominálne hodnoty mincí?

Náhoda to samozrejme nie je, euro mince sú navrhované tak, aby sa nimi ľahko platilo. Ale tiež nejde o žiadnu zákonitosť. Existujú totiž iné sady hodnôt, ktoré túto príjemnú vlastnosť nemajú.

Napríklad v histórii Veľkej Británie bolo obdobie (okolo roku 1980), kedy platné mince tvorili takúto „škaredú“ sadu: obsahovala okrem iných aj 1-pencu, 6-pencu a 10-pencu. Poznáte nejakú krajinu, kde v súčasnosti používajú sadu mincí, pre ktoré pažravý algoritmus nefunguje?

Zadanie 16

Navrhnite nejakú sadu nominálnych hodnôt mincí, pre ktorú pažravý algoritmus nefunguje – teda existuje suma, ktorá sa dá zaplatiť menej mincami ako povie pažravý algoritmus.

Takouto sadou mincí môže byť napríklad sada obsahujúca hodnoty 6 korún, 4 koruny a 1 koruna. Ak by sme chceli zaplatiť sumu 8 korún, pažravý algoritmus by nám

poradil zaplatiť troma mincami: 6, 1, 1. Lepšie by však bolo zaplatiť dvoma mincami, 4 a 4.

Úloha o dláždení

Teraz si ukážeme aktivitu, v ktorej je tvorba efektívnych algoritmov schovaná. Ak začneme skúšať možnosti, postupne si vymyslíme nejaký algoritmus. Ten bude pravdepodobne pažravý, pretože ľudia majú tendenciu nachádzať práve pažravé riešenia.

Aktivita

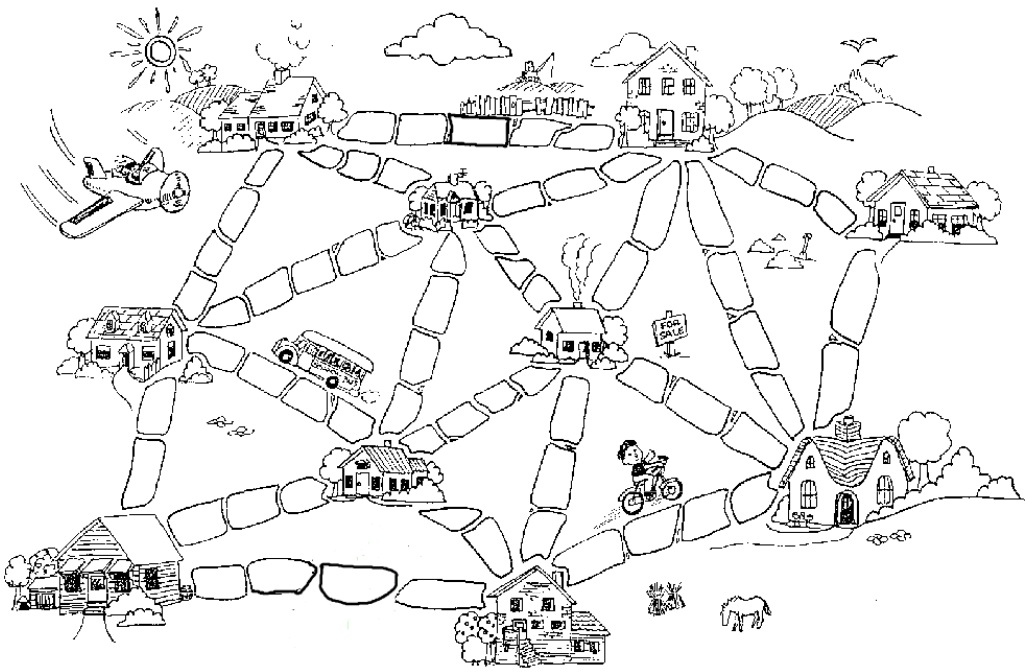
Bolo raz jedno mesto a to malo cesty nevydláždené. Akonáhle sa strhla búrka, všetky cesty boli zablatené. Magistrát mesta sa rozhodol s touto situáciou niečo spraviť – vydláždiť niektoré cesty. Peňazí však nikdy nie je nazvyš, a tak chcú použiť čo najmenej dlaždíc.

Na papieri dostanete obrázok mesta a cesty medzi domčekmi. Pre každú cestu z obrázka vidíte, koľko dlaždíc treba na jej vydláždenie.

Vyfarbite cesty, ktoré chcete vydláždiť. Tieto cesty vyberte tak, aby sa po nich dalo dostať z každého domčeka do každého. Pokúste sa položiť čo najmenej dlaždíc.

Ak je tento počet menší ako dosiaľ nájdený, povedzte ho nahlas.

Bližšie informácie o aktivite nájdete na webstránke http://csunplugged.org/. Na tejto stránke nájdete rôzne ďalšie aktivity a didaktické poznámky. Obrázok mesta je použitý z tejto stránky, je však trochu pozmenený kvôli pôvodnej nejednoznačnosti.



Obrázok 4: Vyfarbite cesty medzi domčekmi tak, aby ste použili čo najmenej dlaždíc a mohli ste prejsť medzi ľubovoľnými dvoma domčekmi (Zdroj: http://csunplugged.org/)

V tejto aktivite je dôležité, aby mohli riešitelia vykrikovať stále menšie a menšie čísla, a tým sa predbiehať. Toto súťaživé prostredie ich motivuje vylepšovať svoju stratégiu, a tak postupne mnohí z nich objavia zákonitosti vedúce k optimálnemu postupu.

Ľudia pri tejto aktivite najčastejšie postupujú dvoma spôsobmi:

- Začneme z prázdnej mapy. Postupne dláždime cesty tak, aby každá nová cesta spojila nejaké domčeky, medzi ktorými sa dovtedy nedalo prejsť po dlaždiciach.

Akú stratégiu ste použili vy?

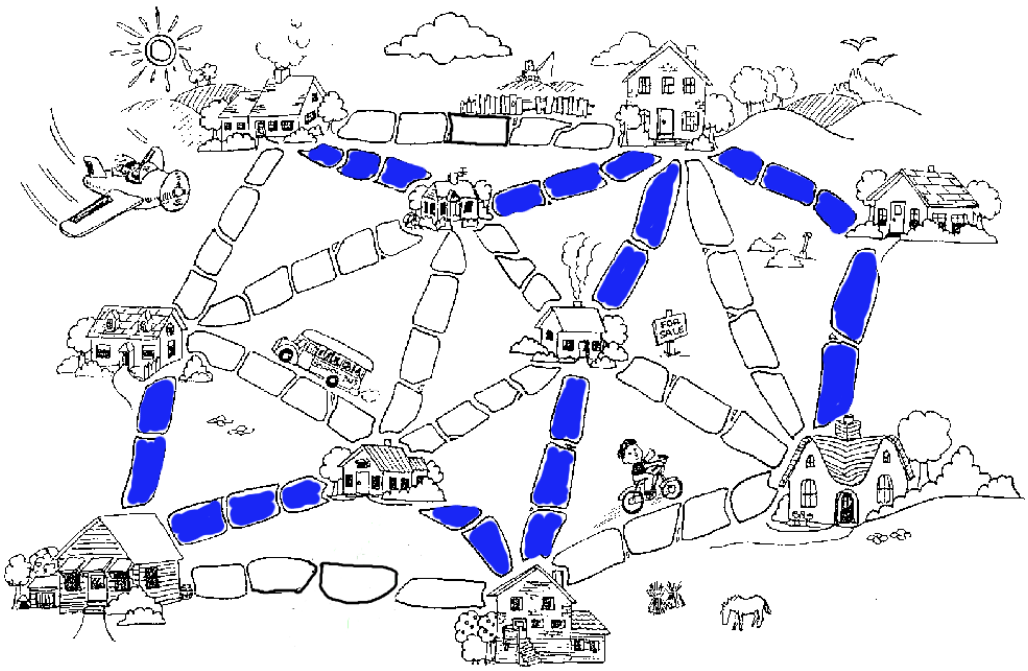
- Začneme z mapy, kde sú všetky cesty vydláždené. Postupne mažeme („oddlažďujeme“) cesty, ktoré sú v danej chvíli nadbytočné.

Ako však dosiahnuť najmenší počet dlaždíc? Najjednoduchšie pozorovanie, ktoré odhalia takmer všetci riešitelia: nikdy sa neoplatí postaviť cesty „do kruhu“. Samo o sebe však toto pozorovanie nestačí.

Ukazuje sa, že k optimálnemu riešeniu vedie hneď niekoľko rôznych pažravých prístupov:

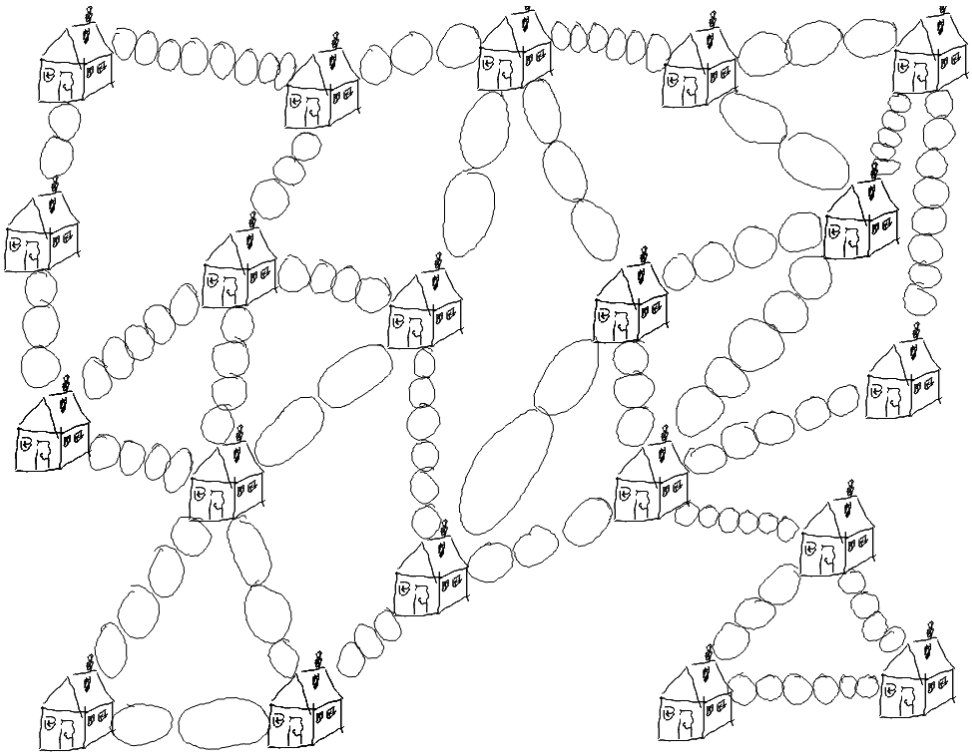
- Budeme sa na cesty pozerať v poradí od cesty s najmenším počtom dlaždíc k ceste s najväčším počtom. Ak už medzi danými domčekmi prejsť vieme, cestu necháme nevydláždenú, ak ešte nie, tak ju vydláždime.
- Na začiatku prehlásime všetky cesty za vydláždené. Teraz začneme cesty spracúvať v opačnom poradí, začínajúc tou, na ktorú treba dlaždíc najviac. Zakaždým overíme, či nám odstránenie cesty nerozpojí cestnú sieť, a ak nie, tak dotyčnú cestu odstránime.
- Začneme z ľubovoľného domčeka. K nemu pripojíme jeho najbližšieho suseda (t. j. toho, ku ktorému vedie cesta s najmenej dlaždicami). Teraz nájdeme spomedzi zvyšných domčekov ten, ktorý vieme najlacnejšie pripojiť k jednému z prvých dvoch, a pripojíme ho k nim. Postup opakujeme, až kým k vznikajúcej cestnej sieti postupne nepripojíme všetky domčeky.

Jeden z možných výsledkov si ukážeme na obrázku.



Obrázok 5: Jedno z riešení pre aktivitu dláždenia použitím prvého algoritmu

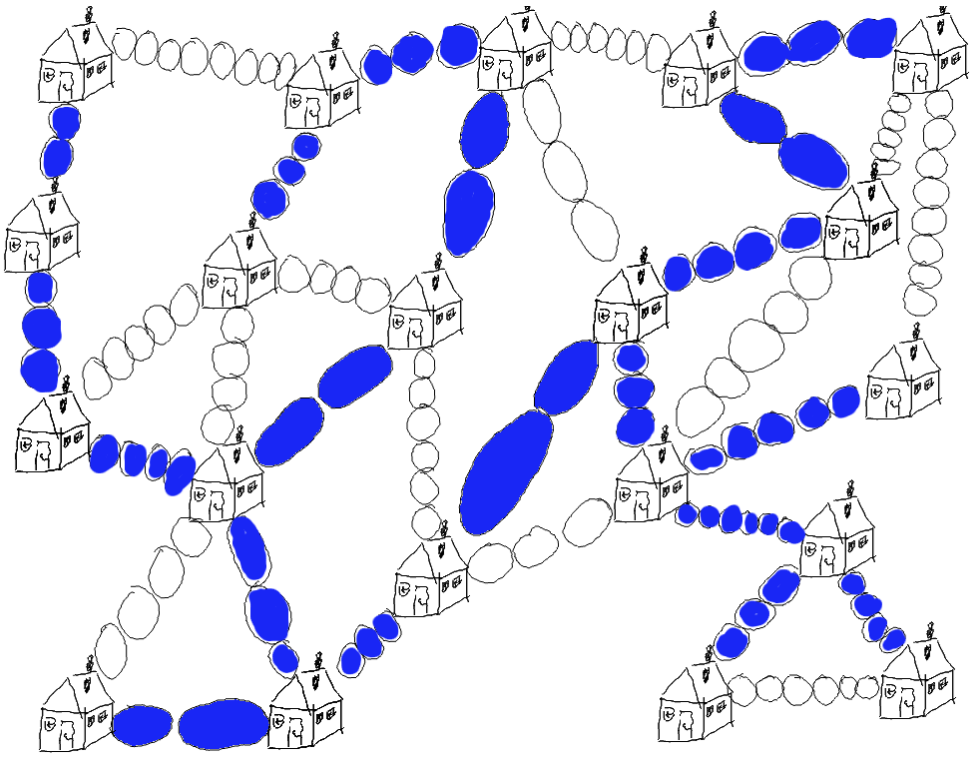
Skúsme si ešte niektorý z postupov na väčšom meste. To nám pomôže nájsť v postupe nevyriešené prípady a väčšie mesto zvýrazní potrebu použiť naozajstný algoritmus, nie iba intuitívne vyfarbovanie.



Obrázok 6: Aktivita na väčšom meste

Väčšinou si nepovieme, v akom poradí prechádzame po cestách skladajúcich sa z dvoch dlaždíc. Keď kreslíme, vyberáme si podľa rôznych preferencií. Ak už však algoritmus programujeme, počítač si už nevyberá a musí mať naprogramované presné poradie.

Rôzne algoritmy budú produkovať rôzne riešenia, ktoré ale budú mať rovnaký (optimálny) počet použitých dlaždíc. Dokonca ak použijeme ten istý „algoritmus“, sa naše riešenia môžu líšiť. Bude to preto, že tento „algoritmus“ nie je popísaný do detailov a neriešime, v akom poradí vyberať cesty s rovnakým počtom dlaždíc. Jedno z riešení, použitím prvého algoritmu, môže vyzerať nasledovne:



Obrázok 7: Jedno z riešení pre aktivitu na väčšom meste

Tak ako aj v predchádzajúcich úlohách, aj tu je dôležité vedieť argumentovať, prečo pažravé riešenie použije najmenej dlaždíc, a teda je optimálne. Vyberte si jeden z týchto postupov (alebo svoj vlastný, o ktorom ste presvedčení, že vždy funguje). Pokúste sa zamyslieť, ako by ste zdôvodnili, že vami zvolený postup skutočne pre ľubovoľnú mapu nájde najlacnejšie riešenie.

Čo to teda sú tie pažravé algoritmy?

> Ak riešime úlohu pažravo, znamená to, že v každej situácii sa snažíme nájsť najlepšie možné riešenie a dúfame, že výsledné riešenie bude tiež najlepšie možné.

V úlohe o tankovaní to znamenalo, že v každej situácii sme tankovali na poslednú chvíľu, v úlohe o euro-minciach sme vždy použili najväčšiu možnú mincu a v úlohe o dláždení mesta sme dláždili tie cesty, na ktoré stačilo použiť najmenej dlaždíc.

Takýto postup je prirodzený, lebo väčšina ľudí sa správa pažravo „od prírody“, či už pre uspokojenie svojich potrieb alebo potrieb rodiny. Ak však chceme nachádzať optimálne riešenia, musíme si dávať pozor na to, či pažravá stratégia dáva naozaj optimálne riešenie pre konkrétnu úlohu.

Úlohy na precvičenie

Príklad:
Strúhanka
2 EUR/kg 7,5 kg
Múka
1 EUR/kg 10,45 kg
Mak
3 EUR/kg 6,25 kg
Cukor
1,20 EUR/kg 4 kg

Príklad:
Začiatok a koniec diery
1,2 1,3
5,4 6,5
1,95 2,1

Zadanie 17	Gertrúda vyhrala v televíznej súťaži. Dostala ruksak s nosnosťou 20 kilogramov, teraz si má vybrať z výkladu odmenu. Ku každému tovaru vie jeho jednotkovú cenu za kilogram a jeho množstvo, ktoré je priamo vo výklade. Z každého tovaru si môže zobrať ľubovoľnú časť. Ako má brať tak, aby získala tovar v čo najvyššej celkovej cene a pritom sa jej ešte zmestil do ruksaku?
Zadanie 18	Na istý ostrov príde Jeho Veličenstvo. Rozhodli sme sa rozprestrieť červený koberec dĺžky 30 metrov. Na koberci sú však diery a treba ich zaplátať. Máme záplaty rovnakej šírky ako koberec a dĺžky 1 meter. Zistite, koľko najmenej záplat treba použiť na koberec, ak viete, kde sa presne nachádzajú diery (daný je začiatok a koniec diery na koberci, šírku nepotrebujete, preto nie je v zadaní.). [1]
Zadanie 19	Na mape zaostalej krajiny je N miest očíslovaných $1, 2, \ldots, N$ so súradnicami $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$. Vládca sa práve rozhodol svoju krajinu elektrifikovať. V meste č. 1 nechal postaviť elektráreň a teraz ešte potrebuje vybudovať elektrickú sieť. Pozval si preto inžinierov z celej krajiny a pod trestom smrti im nariadil postaviť túto sieť za minimálne prostriedky. Inžinieri sa ocitli v nezávideniahodnej situácii a potrebujú vašu pomoc, lebo je im jasné iba to, že čím je vedenie kratšie, tým je aj lacnejšie. Vymyslite algoritmus, ktorý pre danú mapu zistí, ako je najlepšie spojiť mestá elektrickým vedením, aby malo každé mesto elektrinu a dĺžka vedenia bola čo najkratšia. Elektrinu môžete ťahať iba z mesta do mesta. [1]

Dynamické programovanie

V tejto časti textu si predstavíme druhú z techník návrhu efektívnych algoritmov. Presnejšie, pôjde o dve techniky – na prvý pohľad úplne rozdielne, no povedú v podstate k tomu istému algoritmu. Ale skôr, než si tieto techniky ukážeme, skúste si vyriešiť nasledujúcu úlohu.

Zadanie 20	Zlodej Hugo unesie najviac 50 kilogramov vecí. Vlámal sa do domu a o každej veci, ktorú našiel, si zistil jej cenu aj jej hmotnosť. Rozhodol sa, že použije pažravý algoritmus. Dokola bude opakovať nasledujúci postup: nájde najdrahšiu vec, ktorú ešte unesie, a tú zoberie. Akonáhle už nič ďalšie neunesie, odíde aj so svojím lupom. Nájdite nejakú množinu vecí, pre ktorú Hugovo riešenie nebude optimálne (t. j. keby si vyberal šikovnejšie, mohol by odniesť veci s väčšou celkovou cenou, ako majú tie, ktoré vyberie pri svojom pažravom prístupe).

Existuje veľa možných riešení tejto úlohy, uvedieme jedno z nich. V celom dome boli len tri predmety: historická lampa (30 kg, cena 1000 eur), vyrezávané hojdacie kreslo (26 kg, cena 800 eur) a televízor (24 kg, cena 600 eur).

Optimálnym riešením by bolo zobrať kreslo a televízor (dokopy 50 kg a cena 1400 eur), náš zlodej však ako prvú zoberie lampu a následne už neunesie žiadny zo zvyšných dvoch predmetov.

Táto úloha nám ilustruje dôležité pozorovanie: pri niektorých úlohách nemusí pažravé riešenie viesť k najlepšiemu možnému výsledku.

Čo v takejto situácii robiť? Prvý pokus, ako zlodejovu úlohu vyriešiť, môžeme založiť na skúšaní všetkých možností. Nech je v dome N vecí. Potom máme N možností, ktorú vec zlodej skúsi vziať ako prvú. Keď ju vezme, máme $N-1$ možností, ktorú skúsi vziať ako druhú, a tak ďalej. Ak tieto možnosti postupne preskúšame všetky a vyberieme najlepšiu z nich, určite dostaneme správny výsledok.

Aké efektívne je ale toto riešenie? V najhoršom možnom prípade by toto riešenie preskúšalo všetkých $N!$ poradí, v akých mohol náš zlodej brať veci. A to nie je žiadna výhra. Už napríklad pre $N = 20$ by sme sa odpovede nemuseli dožiť.

Skúsme teda vymyslieť niečo lepšie. Jedno miesto, kde sa predchádzajúce riešenie dá zlepšiť, je odstránenie zbytočného skúšania rôznych poradí. Všimnime si totiž, že na poradí, v ktorom zlodej berie veci, vôbec nezáleží. Či vezme najskôr kreslo a potom televízor, alebo naopak najskôr televízor a potom kreslo, tak či tak odíde s kreslom a s televízorom. Nepotrebujeme teda skúšať všetkých $N!$ poradí vecí. Stačí sa nám pozrieť na všetkých 2^N možných množín vecí, vybrať spomedzi nich tie, ktoré náš zlodej celé unesie, a spomedzi tých nájsť najdrahšiu možnú.

Jednoduchý a elegantný spôsob, ako takéto riešenie naprogramovať: veci v dome si označíme číslami od 1 do N. Potom sa náš zlodej bude postupne na veci v tomto poradí pozerať. Pre každú vec má na výber dve možnosti: buď ju vezme, alebo ju nechá v dome. Zjavne každá z 2^N možných postupností výberov, čo vziať a čo nie, zodpovedá jednej z 2^N podmnožín vecí v dome.

Príliš sme si však nepomohli. Takéto riešenie má ešte stále časovú zložitosť exponenciálnu od počtu vecí, a teda síce je ešte použiteľné povedzme pre $N = 30$, ale $N = 50$ už je ďaleko za hranicou našich možností.

Memoizácia

Asi najdôležitejším pravidlom pri návrhu efektívnych algoritmov je zásada: **Nikdy zbytočne nepočítať tú istú vec dvakrát.**

Dopúšťame sa tejto chyby v našom riešení zlodejovej úlohy? Niekedy veru áno.

Predstavme si napríklad, že sa náš zlodej vlámal do domu, v ktorom našiel okrem iných vecí dva drahé kamene. Oba vážia po 0,1 kg.

Očíslujme si veci tak, aby drahé kamene mali čísla 1 a 2.

Pozerajme sa teraz, ako prebieha skúšanie všetkých 2^N možných výberov predmetov. Celé toto skúšanie môžeme v myšlienkach rozdeliť na štyri fázy:

- Zoberie prvý aj druhý kameň, vyskúša všetky možnosti, čo zo zvyšných vecí zoberie.
- Zoberie prvý kameň, nezoberie druhý, vyskúša všetky možnosti, čo zo zvyšných vecí zoberie.
- Nezoberie prvý kameň, zoberie druhý, vyskúša všetky možnosti, čo zo zvyšných vecí zoberie.
- Nezoberie prvý ani druhý kameň, vyskúša všetky možnosti, čo zo zvyšných vecí zoberie.

Čo tu počítame zbytočne dvakrát?

Všimnime si situáciu, kedy sa náš zlodej rozhodol, že prvý drahý kameň zoberie, ale druhý nie. V tomto okamihu ešte zvláda odniesť 49,9 kg a na výber má ľubovoľnú podmnožinu predmetov s číslami 3 až N.

No ale v presne rovnakej situácii sa náš zlodej neskôr ocitne, ak sa rozhodne, že nevezme prvý kameň ale vezme druhý. Opäť bude potrebovať zodpovedať tú istú otázku: *„Koľko najviac si viem zarobiť, ak spomedzi vecí s číslami 3 až N vyberiem nejakú podmnožinu s hmotnosťou nanajvýš 49,9 kg?“*

Ak by sme si ešte pamätali odpoveď na ňu, mohli by sme ju rovno použiť a ušetrili by sme si kopu práce. Všimnime si poriadnejšie, čo sa deje pri našom skúšaní možností. Na začiatku stojí zlodej pred úlohou: *„Zvládam ešte uniesť 50 kg vecí a mám na výber veci s číslami 1 až N, koľko najviac viem získať?“*

Zoberie do ruky prvú vec (s hmotnosťou h_1 a cenou c_1) a rozhodne sa, či ju vezme alebo nie.

- Ak prvú vec nevezme, stojí pred novou úlohou:
  „Zvládam ešte uniesť 50 kg vecí a mám na výber veci s číslami $\mathbf{2}$ až N, koľko najviac viem získať?“
- Ak prvú vec vezme, opäť stojí pred novou úlohou:
  „Zvládam ešte uniesť $\mathbf{50 - h_1}$ kg vecí a mám na výber veci s číslami $\mathbf{2}$ až N, koľko najviac viem získať?“

Deje sa tu teda niečo podobné ako u pažravých algoritmov: opäť potrebujeme vyriešiť ten istý problém, len pre nové, menšie vstupné dáta. Lenže zatiaľ čo pri pažravých algoritmoch sme vedeli nájsť kritérium, podľa ktorého rovno povieme, ktorá možnosť je optimálna, tu také kritérium neexistuje. Potrebujeme postupne vyskúšať obe možnosti a vybrať si lepšiu z nich.

Môžeme si všimnúť, že každú situáciu, v ktorej sa počas skúšania možností ocitneme, vieme celú popísať dvoma číslami: stačí nám vedieť, akú nosnosť H ešte máme, a najmenšie číslo K veci, pre ktorú ešte máme na výber, či ju vziať, alebo nie.

Ak si označíme $Z(H,K)$ najväčší zisk dosiahnuteľný pre nosnosť H a veci s číslami K až N, môžeme riešenie matematicky popísať nasledovne:

$$Z(H,K) = \begin{cases} 0 & \leftarrow \text{ak } K > N \\ Z(H, K+1) & \leftarrow \text{ak } h_K > H \\ \max\big(\ Z(H,K+1), & \\ \qquad c_K + Z(H-h_K, K+1)\ \big) & \leftarrow \text{inak} \end{cases}$$

Vysvetlíme slovne jednotlivé možnosti:

- Ak $K > N$, nezostali nám už na výber žiadne predmety, a teda zisk je určite nulový.
- V opačnom prípade sa pozrime na predmet s číslom K. Ten má hmotnosť h_K a cenu c_K. Ak platí $h_K > H$, je tento predmet pre nás už priťažký – váži viac ako ešte zvládame uniesť. A teda nemáme na výber – musíme tento predmet nechať na mieste.

  Zostáva nám nosnosť H a predmety s číslami $K+1$ až N. V tomto prípade teda platí $Z(H,K) = Z(H,K+1)$.
- Zostáva nám jediný prípad, kedy sa potrebujeme rozhodnúť: vezmeme predmet K alebo nie?

  Ak ho nevezmeme, tak rovnako ako v predchádzajúcej možnosti dostávame $Z(H,K) = Z(H,K+1)$.

  Ak ho vezmeme, budeme mať o jeho hmotnosť nižšiu nosnosť. Spomedzi zvyšných vecí už zvládneme uniesť len $H - h_K$ kilogramov. Najlepšie riešenie pre zvyšné veci je teda rovné $Z(H-h_K, K+1)$. K tomu ešte musíme pripočítať cenu c_K veci, ktorú sme práve vybrali.

  No a keďže chceme čo najväčší zisk, z týchto dvoch možností si vyberieme tú lepšiu. $Z(H,K)$ je preto rovné maximu z uvedených dvoch hodnôt.

Matematickú definíciu si ľahko môžeme prepísať do rekurzívnej funkcie. (Namiesto h_K a c_K v programe používame `hmotnost[K]` a `cena[K]`.)

Riešenie 1 zadania 20

```
function Z ( H, K : longint ) : longint;

var vezme, nevezme : longint;
begin
  if K > N then
     Z := 0
  else begin
    nevezme := Z( H, K+1 );
    if hmotnost[K] > H then
      Z := nevezme
    else begin
      vezme := cena[K] +
               Z( H-hmotnost[K], K+1 );
      if vezme > nevezme then
        Z := vezme
      else
        Z := nevezme;
    end;
  end;
end;
```

Aby sme pochopili, čo táto funkcia robí, skúsme vykonať program na príklade. V celom dome boli len tri predmety: historická lampa (30 kg, cena 1000 eur), vyrezávané hojdacie kreslo (26 kg, cena 800 eur) a televízor (24 kg, cena 600 eur).

Pozrime sa, ktoré hodnoty funkcie počítame.

- Vieme, že $Z(50,1) = \max(\, Z(50,2), 1000 + Z(20,2)\,)$.
  Slovne: najlepší zisk, ak unesieme 50 kg a máme na výber veci 1, 2, 3, dostaneme tak, že vyskúšame dve možnosti. Ak vec 1 (lampu) nevezmeme, hľadáme najlepšie riešenie pre 50 kg a veci 2, 3. Ak lampu vezmeme, máme zisk 1000 a na veci 2, 3 nám už ostalo len 20 kg.
- Podobne $Z(50,2) = max(\, Z(50,3), 800 + Z(24,3)\,)$.
- A taktiež $Z(50,3) = max(\, Z(50,4), 600 + Z(26,4)\,)$.

- $Z(50,4) = 0$, lebo nám už nezostali žiadne veci.
- Tiež $Z(26,4) = 0$.
- V tomto okamihu už vieme vypočítať, že $Z(50,3) = \max(0,600 + 0) = 600$.
- Vraciame sa späť k $Z(50,2)$. Po dosadení práve spočítanej hodnoty $Z(50,3)$ máme $Z(50,2) = \max(600, 800 + Z(24,3))$.
- Vypočítame $Z(24,3) = \max(Z(24,4), 600 + Z(0,4)) = 600$.
- Dosadíme a spočítame $Z(50,2) = \max(600, 800 + 600) = 1400$.
- A sme späť na začiatku. Vieme už, že $Z(50,1) = \max(1400, 1000 + Z(20,2))$.
- $Z(20,2) = Z(20,3)$, lebo druhý predmet je ťažší ako 20 kg.
- $Z(20,3) = Z(20,4)$, lebo aj tretí predmet je ťažší ako 20 kg.
- $Z(20,4) = 0$, a teda aj $Z(20,3) = 0$ a taktiež $Z(20,2) = 0$.
- A teda dostávame $Z(50,1) = max(1400, 1000) = 1400$. Najlepší zisk je 1400 eur.

Táto rekurzívna funkcia sama o sebe nepredstavuje lepšie riešenie problému – postupne skúša všetky možné výbery vecí, v najhoršom prípade teda postupne prezrie všetkých 2^N možností.

Zlepšenie dostaneme až v okamihu, keď sa začneme riadiť zásadou „*nič nepočítať zbytočne dvakrát*“. Akonáhle konkrétnu hodnotu $Z(H,K)$ vypočítame, zapamätáme si ju. A ak v budúcnosti budeme opäť potrebovať túto hodnotu vedieť, namiesto rekurzívnych volaní rovno vrátime zapamätanú hodnotu. Upravený program by mohol vyzerať napríklad takto:

Riešenie 2 zadania 20

```
function Z ( H, K : longint ) : longint;
var vezme, nevezme : longint;
begin
  if spocital[H,K] then begin
    Z := vysledok[H,K];
    exit;
  end;
  if K > N then
     Z := 0
  else begin
    nevezme := Z( H, K+1 );
    if hmotnost[K] > H then
      Z := nevezme
    else begin
      vezme := cena[K] +
               Z( H-hmotnost[K], K+1 );
      if vezme > nevezme then
        Z := vezme
      else
        Z := nevezme;
    end;
  end;
  spocital[H,K] := true;
  vysledok[H,K] := Z;
end;
```

Práve popísaný spôsob „vylepšenia“ algoritmu sa v odbornej literatúre zvykne označovať názvom **memoizácia** (v preklade: zapamätávanie si)

Čo sme tým získali? Riešenie, ktoré bude mať v niektorých prípadoch výrazne lepšiu časovú zložitosť. Predpokladajme, že všetky hmotnosti predmetov sú celé čísla. (Ak by sme mali na vstupe váhy predmetov v gramoch, je to úplne prirodzený predpoklad.) Celkovú nosnosť nášho zlodeja označme $H_{\max}$.

Samotným riešením našej úlohy bude hodnota $Z(H_{max}, 1)$: vieme uniesť H_{max} gramov vecí a máme na výber veci s číslami od 1 po N.

Pri počítaní tejto hodnoty zjavne budeme potrebovať len hodnoty $Z(H, K)$ pre $0 \leq H \leq H_{\max}$ a $1 \leq K \leq N$ – totiž v priebehu skúšania aj naša nosnosť, aj počet vecí na výber len klesajú. Týchto hodnôt je rádovo $N \cdot H_{\max}$. No a ľubovoľnú konkrétnu z nich vieme vypočítať z iných hodnôt v konštantnom čase. Dokopy je teda časová zložitosť tohto algoritmu priamo úmerná hodnote $N \cdot H_{\max}$.

Všimnite si, že sme dosiahli skutočne výrazné zlepšenie: napríklad pre pôvodnú hodnotu $H_{\max} = 50\,000$ gramov by toto riešenie bolo prakticky použiteľné ešte aj pre tisíce predmetov!

Dynamické programovanie

Na vyššie uvedené riešenie sa môžeme pozerať aj „z opačného konca“. Pripomeňme si matematický popis (tzv. rekurentný vzťah) hodnôt $Z(H, K)$:

$$Z(H,K) = \begin{cases} 0 & \leftarrow \text{ak } K > N \\ Z(H, K+1) & \leftarrow \text{ak } h_K > H \\ \max\big(\ Z(H,K+1), & \\ \qquad c_K + Z(H - h_K, K+1)\ \big) & \leftarrow \text{inak} \end{cases}$$

Všimnime si, že každá hodnota $Z(H, K)$ závisí len na nejakých hodnotách $Z(nieco, K+1)$ – inými slovami, na riešeniach situácií s menším počtom predmetov.

Na začiatku vieme, že pre ľubovoľné H platí $Z(H, N+1) = 0$.

Pomocou tejto informácie vieme teraz postupne vypočítať hodnoty $Z(H, N)$ pre všetky H. Totiž na vypočítanie konkrétnej hodnoty $Z(H, N)$ potrebujeme poznať len hodnotu $Z(H, N+1)$, a možno ešte hodnotu $Z(H - h_K, N+1)$ a obe tieto hodnoty už poznáme.

Následne, keď už poznáme hodnoty $Z(H, N)$ pre všetky H, vieme pomocou nich vypočítať hodnoty $Z(H, N-1)$ pre všetky H. A tak ďalej, až kým sa nedopracujeme k želanej hodnote $Z(H_{\max}, 1)$.

Aby sme lepšie porozumeli hľadaniu riešenia, ukážme si ho na príklade. V celom dome boli len tri predmety: historická lampa (30 kg, cena 1000 eur), vyrezávané hojdacie kreslo (26 kg, cena 800 eur) a televízor (24 kg, cena 600 eur). Tabuľka pre výpočet riešenia by vyzerala nasledovne:

H/K	1	2	3
0	**0**	**0**	**0**
…	0	0	0
23	0	0	0
24	**600**	**600**	**600**
…	600	600	600
26	**800**	**800**	**600**
…	800	800	600
30	**1000**	**800**	**600**
…	1000	800	600

49	1000	800	600
50	**1400**	**1400**	**600**

Výsledkom je hodnota `Z[H_MAX,1]`, čiže 1400 eur. A naozaj. Zlodej vie získať veci v hodnote 1400 eur tak, že zoberie hojdacie kreslo a televízor.

V programe by toto riešenie mohlo vyzerať napríklad nasledovne:

Riešenie 3 zadania 20

```
var
  Z : array [0..H_MAX, 1..N+1] of longint;
  H, K, vezme, nevezme : longint;
begin
  for H := 0 to H_MAX do Z[H,N+1] := 0;
  for K := N downto 1 do
    for H := 0 to H_MAX do begin
      nevezme := Z[H,K+1];
      if hmotnost[K] > H then
        Z[H,K] := nevezme
      else begin
        vezme := cena[K] +
                 Z[ H - hmotnost[K], K+1 ];
        if vezme > nevezme then
          Z[H,K] := vezme
        else
          Z[H,K] := nevezme;
      end;
    end;
  writeln('Riesenie je ', Z[H_MAX,1] );
end;
```

Pri tomto programe je ešte zjavnejšie, že má časovú zložitosť priamo úmernú hodnote $N \cdot H_{\max}$. A dokonca počíta presne to isté - hodnoty, ktoré skončia v poli na pozícii `Z[H,K]` sú totožné aj s našimi hodnotami $Z(H,K)$, aj s hodnotami, ktoré budú na konci behu riešenia s memoizáciou uložené v poli `vysledok[H,K]`.

Pre túto metódu implementácie riešenia sa (z obskúrnych historických dôvodov) zaužíval názov **dynamické programovanie**.

Vidíme teda, že dynamické programovanie a memoizácia sú dve strany jednej mince. Oba prístupy slúžia na to, aby sme všetko potrebné počítali len raz.

Samozrejme, rovnako ako niektoré problémy nemajú pažravé riešenie, sú aj problémy, kde nám dynamické programovanie nijak nepomôže. Dynamické programovanie (resp. memoizáciu) má zmysel použiť len vtedy, ak riešenie úlohy „hrubou silou“ zahŕňa riešenie veľa podobných podúloh, pričom mnohé z nich sa opakujú.

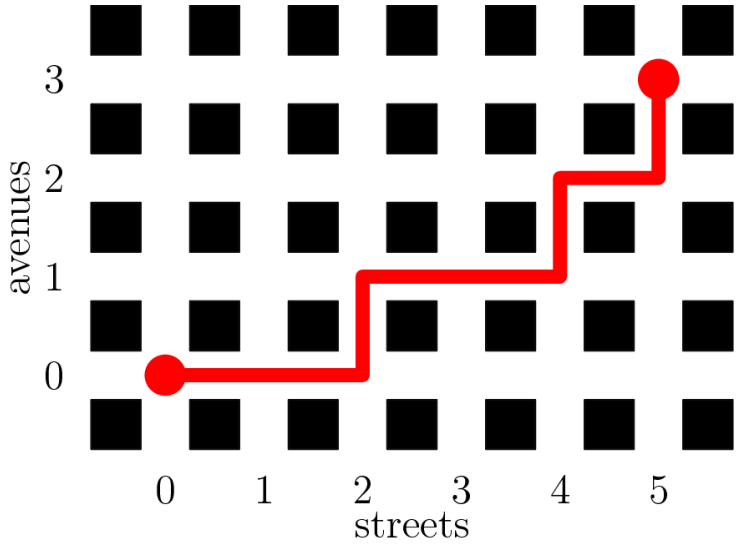
Cesty v štvorcovej sieti

Koncept dynamického programovania si ešte priblížime na jednom inom type úloh.

Zadanie 21

Pôdorys Manhattanu má tvar štvorcovej siete. Niektoré ulice (nazývané avenues) sú rovnobežné s osou ostrova, ostatné (nazývané streets) sú na ne kolmé. Aj jedny, aj druhé ulice sú očíslované. Každá križovatka má teda dve súradnice (A,S): číslo avenue a číslo street, ktoré sa tam pretínajú.

Irena sa nachádza na križovatke $(0,0)$ a chce sa čo najrýchlejšie (teda najkratšou možnou cestou) dostať na križovatku (A,S). Spomedzi koľkých ciest si môže vybrať?



Obrázok 8: Ilustračný plán Manhattanu a jedna cesta z $(0,0)$ na $(3,5)$.

Túto úlohu vieme riešiť priamo, matematickou úvahou. Zjavne každá najkratšia cesta má dĺžku $A+S$. (Dĺžku počítame ako počet blokov domov, okolo ktorých prejdeme. Inými slovami, za jednotkovú vzdialenosť považujeme vzdialenosť dvoch susedných ulíc.) Irena teda cestou spraví presne $A+S$ presunov „o križovatku ďalej“. Spomedzi týchto presunov bude A v jednom smere (na obrázku je to dohora) a S v druhom smere (na obrázku doprava). No a každá cesta je jednoznačne určená tým, že vyberieme, ktorých A spomedzi všetkých $A+S$ krokov povedie dohora. Preto je všetkých možných ciest $\binom{A+S}{A}$.

My si však ukážeme aj iné, programátorské riešenie tejto úlohy. To názorne ukáže súvis medzi kombinačnými číslami a Pascalovým trojuholníkom, a navyše ho neskôr budeme ľahko vedieť upraviť na riešenie všeobecnejšej úlohy, kde už jednoduché kombinačné čísla stačiť nebudú.

Podobne ako v úlohe so zlodejom začneme tým, že nájdeme rekurentné vyjadrenie odpovede, ktorú hľadáme – inými slovami, prevedieme riešenie zadaného problému na niekoľko menších.

Počet najkratších ciest z križovatky $(0,0)$ na križovatku (a,s) označme $P(a,s)$. Pozrime sa teraz na všetky tieto cesty. A presnejšie, všimnime si ich posledný krok. Máme len dve možnosti, odkiaľ sme na križovatku (a,s) mohli prísť: buď zdola, z križovatky $(a-1,s)$, alebo zľava, z križovatky $(a,s-1)$. Ak sa teda chceme dostať z $(0,0)$ na (a,s), máme na výber nasledujúce možnosti:

- Jednou z $P(a-1,s)$ ciest prídeme z $(0,0)$ na $(a-1,s)$ a odtiaľ prejdeme na (a,s).
- Jednou z $P(a,s-1)$ ciest prídeme z $(0,0)$ na $(a,s-1)$ a odtiaľ prejdeme na (a,s)

Celkový počet ciest z $(0,0)$ na (a,s) dostávame sčítaním počtov pre prvú a druhú možnosť. Dostávame tak veľmi jednoduchý vzťah:

$$P(a,s) = P(a-1,s) + P(a,s-1)$$

(Nesmieme ešte zabúdať na okrajové podmienky: $P(0,0)=1$ a $P(a,s)=0$ ak je a alebo s záporné.)

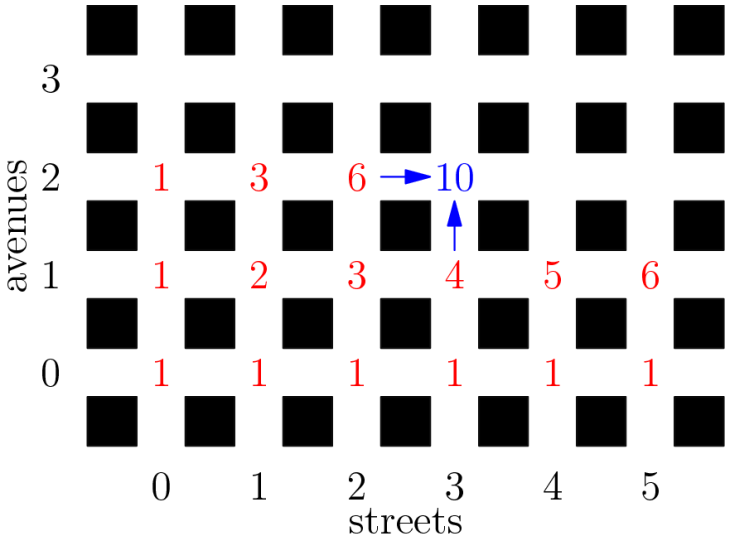
Pomocou práve odvodeného vzťahu vieme metódou dynamického programovania zostrojiť algoritmus, ktorý hodnotu $P(A,S)$ vypočíta v čase priamo úmernom $A \cdot S$:

Riešenie zadania 21

```
var
  P : array [ 0..A, 0..S ] of longint;
  x, y, z : longint;
begin
  for x := 0 to A do
    for y := 0 to S do begin
      if ( x = 0 ) and ( y = 0 ) then
        P[ x, y ] := 1
      else begin
        z := 0;
        if x > 0 then
          z := z + P[ x-1, y ];
        if y > 0 then
          z := z + P[ x, y-1 ];
        P[x,y] := z;
      end;
    end;
end;
```

Tento algoritmus si vieme veľmi ľahko graficky ilustrovať. Predstavme si, že na každú križovatku na mape postupne napíšeme počet ciest, ktorými sa na ňu vieme dostať. Pre konkrétnu križovatku vieme jej hodnotu vypočítať vo chvíli, kedy poznáme hodnotu pre križovatku pod ňou a vľavo od nej. Náš algoritmus tieto hodnoty vypĺňa systematicky, po riadkoch, čím si zabezpečí, že vždy všetky potrebné hodnoty pozná. Konkrétnu situáciu počas behu nášho algoritmu si môžeme pozrieť na nasledujúcom obrázku

Na križovatke so súradnicami $(2,2)$ je napísaná hodnota 6. Viete nakresliť všetkých 6 spôsobov, ako sa sem z križovatky $(0,0)$ najkratšou cestou dostať?



Obrázok 9: Hodnotu aktuálneho políčka (10) vypočítame ako súčet políčok vľavo a dole (6 + 4).

Aby sme videli, v čom je tento programátorský pohľad lepší ako pôvodný kombinatorický, pozrime sa na zložitejšiu verziu našej úlohy.

Zadanie 22

Juraj sa tiež nachádza v Manhattane na križovatke $(0,0)$. Aj on by sa chcel nejakou cestou dĺžky $A+S$ dostať na križovatku (A,S). Dnes však tím New York Rangers hrá dôležitý zápas, v dôsledku čoho sú niektoré križovatky zablokované a nedá sa tadiaľ prejsť. Juraj presne vie, ktoré križovatky sú zablokované a ktoré nie. Vysvetlite, ako by ste zistili:

- či sa Juraj vôbec vie na križovatku (A,S) teoreticky najkratšou cestou dostať,
- ak áno, koľko možností má na výber.

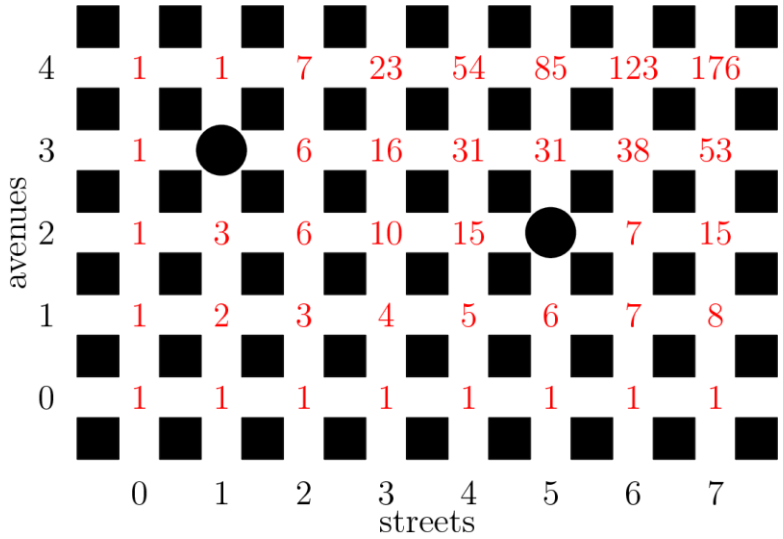
Zatiaľ čo kombinatorickou úvahou sa už táto úloha riešiť nedá, programátorský pohľad zostáva takmer nezmenený. Rovnako ako predtým si označíme $P(a,s)$ počet najkratších ciest z $(0,0)$, ktoré končia na križovatke (a,s). S tým rozdielom, že tentokrát počítame len cesty, ktoré prechádzajú len cez nezablokované križovatky.

Rovnakou úvahou ako v predchádzajúcom riešení dostávame nasledujúci rekurentný vzťah:

$$P(a,s) = \begin{cases} 1 & \leftarrow \text{ak } (a,s) = (0,0) \\ 0 & \leftarrow \text{ak } a < 0 \text{ alebo } s < 0 \text{ (zlý smer)} \\ 0 & \leftarrow \text{ak je križovatka } (a,s) \text{ zablokovaná} \\ P(a-1,s) + P(a,s-1) & \leftarrow \text{inak} \end{cases}$$

Predpokladajme, že v poli `blok[x,y]` máme o každej križovatke informáciu, či je zablokovaná alebo nie. Pokúste sa upraviť vyššie uvedený program tak, aby využíval túto informáciu.

Na nasledujúcom obrázku je príklad hodnôt spočítaných takto upraveným algoritmom pre jednu konkrétnu sadu zablokovaných križovatiek.



Obrázok 10: Zistenie počtu najkratších ciest pre dve zablokované križovatky.

Čo sme sa naučili

Niektoré úlohy môžeme riešiť metódou dynamického programovania. Pri tejto metóde si k výpočtu pomáhame riešením menších problémov.

Môžeme sa na ňu pozerať z dvoch strán. Pri memoizácii postupujeme od väčšieho problému k menším a pamätáme všetky výsledky, ktoré sme kedy vypočítali, aby sme ich nemuseli počítať viackrát. Pri skutočnom dynamickom programovaní postupujeme od menších problémov k väčším a hodnoty, ktoré by sme ešte mohli potrebovať, si pamätáme.

Princíp vyváženosti

Do pestrej a bohatej oblasti návrhu efektívnych algoritmov sme zatiaľ načreli len skraja. Zoznámili sme sa zatiaľ s dvomi metódami: s pažravým prístupom k riešeniu úloh a s dynamickým programovaním. Mnohé iné prístupy nestihneme ani len spomenúť.

Aby sme pomohli čitateľovi spraviť si lepšiu predstavu, o ako rozsiahlu oblasť ide, uvedieme, že asi najznámejšia učebnica **základov** tvorby efektívnych algoritmov [2] má približne tisíc tristo strán.

Pripadá nám zbytočné snažiť sa na tomto mieste vymenúvať ďalšie prístupy k návrhu efektívnych algoritmov. Bez potrebných detailov by takýto zoznam čitateľovi aj tak nebol nijako užitočný. Namiesto toho sme si zvolili jeden všeobecný princíp, využívaný pri rôznych efektívnych algoritmoch. Jeho užitočnosť si ukážeme na niekoľkých príkladoch.

Spomínaný princíp sme nazvali „princíp vyváženosti“ a sformulovať ho môžeme napríklad nasledovne:

> Keď pri spracúvaní rovnocenných dát nastane situácia, kedy ich potrebujeme rozdeliť, oplatí sa deliť ich približne na polovicu.

Aké číslo si myslím?

Zadanie 23	Karol si myslí číslo od 1 po 30. Lucia sa ho môže pýtať ľubovoľné otázky, na ktoré je odpoveď len áno alebo nie, a Karol jej na ne bude pravdivo odpovedať. Poraďte Lucii stratégiu, pomocou ktorej Karolovo číslo určite uhádne, a to pomocou nanajvýš 5 otázok.

Na začiatku hry Lucia o Karolovom čísle nevie nič, môže to byť hociktoré z čísel od 1 po 30. Lucia má teda 30 možných riešení a musí zistiť, ktoré z nich je to správne. Každou otázkou sa jej môže podariť niektoré možnosti vylúčiť.

Najlepšia stratégia pre Luciu je očividná: treba sa vždy pýtať tak, aby zostávajúce možnosti rozdelila na polovicu.

Ako prvú teda môže položiť napríklad otázku: „*Je tvoje číslo väčšie ako 15?*“

Ak Karol odpovie áno, Lucia sa dozvie, že hľadá číslo od 16 do 30, v opačnom prípade zase vie, že hľadá číslo od 1 do 15. Aj v jednom, aj v druhom prípade už Lucii zostalo len 15 možností. A v rovnakom duchu môže Lucia pokračovať aj ďalej. Druhú otázku položí tak, aby spomedzi 15 možností, medzi ktorými sa rozhoduje, bola pre 7 odpoveď áno a pre 8 nie. Napr. ak háda číslo od 16 do 30, môže položiť otázku: „Je tvoje číslo menšie ako 23?“

Po druhej Karolovej odpovedi teda Lucii zostane najviac osem možností. Ak Lucia opäť správne položí otázku, po tretej odpovedi zostanú už len štyri možnosti, po štvrtej odpovedi to budú nanajvýš dve možnosti, a ak ešte Lucia nevie odpoveď, tak piatou otázkou ju zaručene zistí.

Nasleduje jeden možný priebeh hry. Lucia pri ňom používa vyššie popísanú stratégiu.

Príklad k zadaniu 23

L: Leží tvoje číslo v rozsahu od 1 do 15?

K: Nie.

L: Leží tvoje číslo v rozsahu od 16 do 22?

K: Áno.

L: Leží tvoje číslo v rozsahu od 16 do 19?

K: Áno.

L: Leží tvoje číslo v rozsahu od 16 do 17?

K: Nie.

L: Je tvoje číslo 18?

K: Nie.

L: Tak potom už viem, že si myslíš číslo 19!

Lucia teda postupne položila 5 otázok, dostala na ne odpovede a po piatej odpovedi už mala istotu, že vie, aké číslo si Karol myslí.

Je táto Luciina stratégia najlepšia možná, alebo existuje postup, ktorým Lucia vždy uhádne myslené číslo už po štvrtej otázke?

Zdôvodnenie prenecháme na čitateľa, ale napovieme nasledujúcou úlohou.

Iná stratégia, ktorá Lucii tiež zaručí, že číslo uhádne po piatich otázkach: stačí sa Karola postupne pýtať na jednotlivé cifry jeho čísla, keď ho zapíšeme v dvojkovej sústave.

Zadanie 24

Pri tejto hre môže Karol ľahko „podvádzať“. Číslo, ktoré „si myslí“, si totiž nemusí zvoliť hneď. Namiesto toho bude iba počúvať Luciine otázky a zakaždým odpovie tak, aby Lucii zostalo čo najviac možností.

Ak by napríklad Lucia položila prvú otázku „Myslíš si jednociferné číslo?“, Karol jej odpovie, že nie. Takto totiž Lucii zostane ešte 21 možností, medzi ktorými musí nájsť tú správnu.

Zdôvodnite, že takto Karol donúti Luciu položiť aspoň 5 otázok.

Binárne vyhľadávanie

Presne rovnakú myšlienku ako Luciina stratégia v predchádzajúcej úlohe má aj algoritmus binárneho vyhľadávania prvku v usporiadanom poli.

Zadanie 25

Pred hodinou telesnej výchovy sa žiaci zoradili do radu podľa veľkosti. Ich učiteľa Mariána zaujala otázka: „*Meria niektorý z nich presne 155 centimetrov?*“

Mohol by samozrejme začať merať zaradom všetkých žiakov, to by mu ale zabralo celú hodinu. Viete mu poradiť lepšiu stratégiu?

V reálnom živote by samozrejme stačilo odhadnúť, ktoré deti majú približne 155 centimetrov, a odmerať ich výšky. My ale ukážeme postup, ktorý by telocvikár Marián dokázal použiť aj vtedy, ak by nemal vôbec žiadny odhad výšok.

Marián má pred sebou nastúpených žiakov. Ich výšky môžu vyzerať napr. nasledovne:

135 137 142 145 145 155 158 160 163 163 167 170 171 172 188

V prvom kroku Marián odmeria výšku žiaka stojaceho uprostred. V našom prípade to

je žiak vysoký 160 cm.

135 137 142 145 145 155 158 **160** 163 163 167 170 171 172 188

Keďže 160 cm je už priveľa, vieme, že hľadaný žiak (ak takého vôbec má) musí stáť niekde naľavo. Marián preto pošle práve odmeraného žiaka aj všetkých napravo od neho sadnúť na lavičku. Zostalo nám sedem žiakov:

135 137 142 145 145 155 158 160 163 163 167 170 171 172 188

Celý postup zopakujeme: Odmeriame prostredného žiaka, čím zistíme, že meria 145 cm. A keďže to je primálo, jeho aj všetkých menších od neho pošleme preč. Zostanú nám už len traja žiaci:

135 137 142 145 145 155 158 160 163 163 167 170 171 172 188

A pri treťom opakovaní postupu hľadaného žiaka nájdeme. Spomedzi pätnástich nastúpených žiakov nám teda stačilo odmerať len troch.

135 137 142 145 145 155 158 160 163 163 167 170 171 172 188

Dôvodom, vďaka ktorému bol náš postup efektívny, je práve uplatnenie princípu vyváženosti. Tým, že sme zvolili žiaka stojaceho uprostred, sme vlastne žiakov rozdelili na dve polovice: nižšiu a vyššiu. A následne sa tým, že zvoleného žiaka odmeriame, dozvieme, v ktorej polovici sa nachádza ten, ktorého hľadáme.

Odtiaľ aj pochádza názov tohto postupu: **binárne vyhľadávanie** (teda vyhľadávanie delením na polovicu).

Teda pomocou jedného jediného merania sme dokázali vylúčiť celú jednu polovicu všetkých žiakov. Ich presné výšky nás už nebudú zaujímať. Inými slovami, každým meraním sme zmenšili počet možností, medzi ktorými sa rozhodujeme, približne na polovicu.

Skutočnú predstavu, ako efektívny je tento postup, získame vtedy, ak si predstavíme jeho použitie na väčšom príklade.

Zadanie 26

V poli s milión položkami máme záznamy o všetkých používateľoch nášho webového portálu. Tieto záznamy sú usporiadané podľa prihlasovacieho mena.

Pri spracúvaní registrácie nového používateľa je potrebné overiť, či ním zvolené prihlasovacie meno už nepoužíva niekto iný. Ako to efektívne urobiť?

K dispozícii máme usporiadaný zoznam všetkých už použitých prihlasovacích mien. Môžeme teda použiť binárne vyhľadávanie. Zakaždým, keď porovnáme nové prihlasovacie meno s vhodne zvoleným použitým, dokážeme tým vylúčiť polovicu možností. A keďže $1\,000\,000 < 2^{20}$, po 20 porovnaniach už budeme vedieť, či sa nové prihlasovacie meno nachádza medzi použitými alebo nie.

Formálne by sme povedali, že časová zložitosť binárneho vyhľadávania je logaritmická od počtu záznamov. Na nájdenie prvku v usporiadanom poli s N prvkami potrebujeme rádovo $\log_2 N$ porovnaní.

Tu už je efektivita binárneho vyhľadávania zjavná: namiesto milióna porovnaní nám ich stačilo len dvadsať.

Spájanie utriedených postupností

Zadanie 27

Nina tiež učí telesnú výchovu. Jej žiaci však boli naučení postaviť sa na začiatku hodiny do dvoch radov. V každom rade sú žiaci zoradení podľa výšky.

Nina by chcela žiakov preusporiadať tak, aby všetci stáli zoradení podľa veľkosti v jednom dlhom rade. Ako má postupovať?

Rozostavenie žiakov môže vyzerať napr. takto:

prvý rad: 135 137 145 145 163 170 172 188

druhý rad: 142 155 158 160 163 167 171

Nina môže napríklad začať žiakov stavať do nového radu. Na jeho konci musí stáť najvyšší zo všetkých žiakov. Kde ho nájde? Sú len dve možnosti: buď stojí na konci prvého, alebo na konci druhého radu. Stačí jej teda porovnať žiakov, ktorí práve stoja na koncoch radov, a vyššieho z nich postaviť na koniec nového radu.

V našom príklade by Nina porovnala žiakov vysokých 188 a 171 cm. Vyššieho z nich by postavila do nového radu, čím by vznikla nasledujúca situácia:

prvý rad: 135 137 145 145 163 170 172

druhý rad: 142 155 158 160 163 167 171

nový rad: 188

Teraz môžeme zopakovať predchádzajúcu úvahu. Najvyšší zo žiakov, ktorí ešte stoja v pôvodných radoch, musí byť na konci jedného z nich. Nina teda porovná týchto dvoch žiakov a vyššieho z nich pošle do nového radu.

V našom príklade by Nina teraz porovnala žiakov vysokých 172 a 171 cm. Po tom, ako vyššieho z nich presunie do nového radu, by situácia vyzerala takto:

prvý rad: 135 137 145 145 163 170

druhý rad: 142 155 158 160 163 167 171

nový rad: 172 188

Po tom, ako Nina ešte trikrát zopakuje vyššie popísaný postup, by situácia vyzerala nasledovne:

prvý rad: 135 137 145 145 163

druhý rad: 142 155 158 160 163

nový rad: 167 170 171 172 188

A ak takto postupne porovná aj zvyšných žiakov, na konci dostane jeden nový rad, v ktorom budú všetci žiaci usporiadaní podľa výšky.

Časová zložitosť tohto algoritmu je lineárna vzhľadom na počet žiakov, keďže každý krok vieme vyhodnotiť v konštantnom čase a v každom kroku jedného zo žiakov pridáme do výsledného radu.

Úmyselne sme vynechali niekoľko detailov. Môžete sa pokúsiť ich domyslieť: Čo presne má Nina spraviť, ak sú obaja porovnávaní žiaci rovnako vysokí? A čo sa stane, keď sa v jednom rade už minú žiaci?

Niektorí by mohli namietať, že správne sa povie usporadúvanie, nie triedenie. A majú čiastočne pravdu. Pojem triedenie je však zaužívaný a sú dôvody, prečo úlohu usporadúvania prvkov ľudia takto pomenovali.

Triedenie spájaním

Postup z predchádzajúcej časti je základom jedného z najefektívnejších triediacich algoritmov: triedenia spájaním, po anglicky nazývaného MergeSort.

Slovne si tento algoritmus môžeme popísať nasledovne:

- Ak má postupnosť len jeden člen, je už utriedená.
- Ak má postupnosť viac ako jeden člen, postupne vykonáme nasledujúce kroky:

  1 Postupnosť rozdelíme na dve rovnako veľké časti.

  (Resp. ak je nepárnej dĺžky, jedna časť bude mať o prvok viac.)

  2 Použitím tohto istého algoritmu utriedime prvú časť postupnosti.

  3 Použitím tohto istého algoritmu utriedime druhú časť postupnosti.

  4 Použitím algoritmu, ktorý použila v predchádzajúcej úlohe telocvikárka Nina, obe utriedené postupnosti spojíme do jednej.

Vykonávanie tohto algoritmu si môžeme ilustrovať na nasledujúcom príklade.

Zadanie 28	Generál si všimol na nádvorí skupinku 16 vojakov. Rozhodol sa, že ich nechá nastúpiť do radu zoradených v abecednom poradí. Vojaci sú však nevycvičení a sami sa zoradiť nevedia. Ako má generál postupovať?

Náš generál sa rozhodol použiť triedenie spájaním. Vojakov postavil do pozoru a rozdelil ich na dve skupinky po 8 vojakov.

Potom si zavolal dvoch plukovníkov. Každému z nich pridelil jednu skupinku a rozkázal mu, aby ich utriedil.

No a keď onedlho obaja plukovníci rozkaz splnili, zobral generál oba utriedené rady vojakov a rovnakým postupom ako Nina v predchádzajúcej úlohe ich spojil do jedného.

A ako postupoval každý z plukovníkov pri triedení svojich 8 vojakov? Presne rovnako ako generál: vojakov rozdelil na dve štvorice, zavolal dvoch poručíkov a prikázal im, nech každý utriedi svoju štvoricu. Utriedené štvorice si potom plukovník spojil do jednej osmice a s tou prišiel za generálom.

Každému je už asi jasné, ako to vyzeralo ďalej. Každý poručík rozdelil vojakov na dve dvojice a nechal ich utriediť čatárom. No a čatár zobral dvojicu a rozdelil ju na vojaka a vojaka. Jedného vojaka už triediť netreba, preto čatár už nikoho nevolal a iba svojich dvoch vojakov porovnal.

Armádnou hierarchiou sme si práve pomohli pri ilustrácii toho, čo sa deje pri vykonávaní **rekurzívneho** algoritmu. Technike používanej v práve prezentovanom algoritme sa niekedy hovorí **metóda rozdeľuj a panuj**. Jej princíp spočíva v tom, že problém najskôr rozdelíme na niekoľko menších podproblémov, samostatne každý z nich vyriešime a na záver pomocou získaných výsledkov zostrojíme riešenie pôvodného problému.

Triedenie spájaním bez rekurzie

Na to, čo sa deje pri triedení spájaním, sa môžeme pozerať aj z iného uhlu, bez nutnosti používať rekurziu. Zatiaľ čo výhoda rekurzívneho postupu spočíva v jednoduchej implementácii, pohľad, ktorý si ukážeme teraz, je konceptuálne jednoduchší.

Algoritmus triedenia spájaním si totiž môžeme preformulovať aj nasledovne:

- Na začiatku máme N prvkov, ktoré chceme utriediť. Môžeme sa na ne dívať tak, že každý prvok je samostatná utriedená postupnosť dĺžky 1.
- Kým máme viac ako jednu postupnosť, zoberieme dve najkratšie postupnosti a známym postupom ich spojíme do jednej novej utriedenej postupnosti.

Zostaňme pri našom príklade s vojakmi. Ak by sme mali utriediť 8 vojakov abecedne, môže na začiatku behu algoritmu vyzerať situácia napr. nasledovne:

Mirko	Janko	Albert	Milan	Ferko	Zdenko	Boris	Gusto

Zoberieme prvé dve postupnosti (prvú tvorí len Mirko, druhú len Janko) a spojíme ich do jednej postupnosti (Janko, Mirko):

Albert	Milan	Ferko	Zdenko	Boris	Gusto	Janko	Mirko

To isté zopakujeme s ďalšou dvojicou: spojením postupností (Albert) a (Milan) vznikne postupnosť (Albert, Milan):

Ferko	Zdenko	Boris	Gusto	Janko	Mirko	Albert	Milan

Rovnako spracujeme zvyšné dve dvojice vojakov:

Janko	Mirko	Albert	Milan	Ferko	Zdenko	Boris	Gusto

Tým sme sa už dostali do situácie, kedy máme štyri utriedené postupnosti, každú dĺžky 2. A pokračujeme ďalej: zoberieme postupnosti (Janko, Mirko) a (Albert, Milan) a spojíme ich do jednej.

Ferko	Zdenko	Boris	Gusto	Albert	Janko	Milan	Mirko

To isté spravíme so zvyšnými dvomi postupnosťami dĺžky 2:

Albert	Janko	Milan	Mirko	Boris	Ferko	Gusto	Zdenko

No a už nás čaká len posledný krok. Zoberieme obe postupnosti dĺžky 4, spojíme ich do jednej utriedenej postupnosti dĺžky 8, a tým sme skončili.

Albert	Boris	Ferko	Gusto	Janko	Milan	Mirko	Zdenko

Pri tomto pohľade na triedenie spájaním vieme relatívne ľahko odhadnúť aj jeho časovú zložitosť. Stačí si všimnúť ľubovoľného konkrétneho vojaka. Vždy, keď sme ho spracovali, dostal sa tým do dvakrát väčšej skupiny ako predtým. Ak teda použijeme tento postup na triedenie N vojakov, každého konkrétneho vojaka spracujeme len približne $\log_2 N$ krát.

Z tohto pozorovania teda vyplýva, že časová zložitosť triedenia spájaním je priamo úmerná $N \cdot \log_2 N$.

Princíp vyváženosti pri triedení spájaním

Pri triedení spájaním nesmieme zabudnúť na hlavný princíp, ktorý si v tejto kapitole predstavujeme: princíp vyváženia.

Všimnime si ešte raz popis nášho triediaceho algoritmu.

1. **Postupnosť rozdelíme na dve rovnako veľké časti.**

   (Resp. ak je nepárnej dĺžky, jedna časť bude mať o prvok viac.)
2. Použitím tohto istého algoritmu utriedime prvú časť postupnosti.
3. Použitím tohto istého algoritmu utriedime druhú časť postupnosti.
4. Použitím algoritmu, ktorý použila v predchádzajúcej úlohe telocvikárka Nina, obe utriedené postupnosti spojíme do jednej.

Funkčnosť algoritmu vôbec nezávisí na tom, na aké veľké časti v prvom kroku

postupnosť rozdelíme. Prečo sa teda oplatí deliť ju práve na polovice?

Toto delenie síce nijako neovplyvní správnosť algoritmu, výrazne ale ovplyvní jeho časovú zložitosť. Aby sme videli, aká výrazná môže byť táto zmena, pozrime sa na opačný extrém.

Zadanie 29	Čo by sa stalo, keby sme zmenili prvý krok triedenia spájaním nasledovne: 1'. Postupnosť rozdelíme na dve časti – prvú budú tvoriť všetky prvky okrem posledného, druhú bude tvoriť posledný prvok.

Takto „vylepšené“ triedenie by malo časovú zložitosť v najhoršom prípade až kvadratickú od počtu triedených prvkov.

Skúste si napríklad odsimulovať takéto triedenie na postupnosti $(5, 4, 3, 2, 1)$. Presne rovnako to bude prebiehať pre ľubovoľnú postupnosť $(N, N-1, \ldots, 3, 2, 1)$. Takúto postupnosť náš upravený algoritmus síce utriedi, ale pritom postupne porovná úplne každú dvojicu prvkov.

Správne triedenie, ktoré sa drží princípu vyváženosti, je natoľko efektívne, že za jedinú sekundu zvládne utriediť stotisíce prvkov. Naproti tomu táto upravená verzia by mala problém za sekundu spracovať čo i len desaťtisíc prvkov.

Efektívne údajové štruktúry

Všetky algoritmy, ktoré sme si doteraz predstavili, boli „jednorazové“ – načítame dáta, spracujeme dáta, hotovo. V praxi sa však často stretneme aj s omnoho náročnejšími problémami. Údaje sa často v priebehu času menia a my s tým potrebujeme počítať.

Viackrát sme už napríklad spomínali príklad s prihlasovaním používateľov na webový portál. Videli sme napríklad, že ak by sme mali všetkých používateľov uložených v utriedenom poli, vedeli by sme medzi nimi efektívne vyhľadávať pomocou algoritmu binárneho vyhľadávania.

Lenže denno-denne sa na portáli registrujú noví používatelia. A ak už máme povedzme stotisíc používateľov, nemôžeme si dovoliť všetkých znova a znova triediť, zakaždým keď sa nejaký nový zaregistruje.

A tu prichádzajú k slovu **údajové štruktúry**. Údajová štruktúra je vlastne spôsob organizácie údajov v pamäti počítača. Zvonka sa programátor môže dívať na údajovú štruktúru ako na čiernu krabičku, ktorej môže dávať príkazy. Napríklad:

- vložiť do nej nový údaj,
- odstrániť z nej konkrétny údaj,
- opýtať sa, koľko údajov obsahuje,
- opýtať sa, či už obsahuje konkrétny údaj,
- …

Od vnútra dotyčnej čiernej krabičky (čiže od jej **implementácie**) závisí, aké presne operácie umožňuje programátorovi vykonať, a tiež to, ako efektívne tieto operácie budú vykonané.

Moderné programovacie jazyky (C++, Java, a tiež mnohé skriptovacie jazyky ako Python a Ruby) priamo poskytujú programátorom viacero užitočných údajových štruktúr. Programátor, ktorý ich pozná, si z nich v prípade potreby vhodnú môže vybrať a použiť.

Toto je v praxi veľmi dôležité. Používanie správnych štandardných údajových štruktúr vedie k efektívnemu programu, ktorý má navyše omnoho stručnejší a zrozumiteľnejší zdrojový kód, ako keby si programátor dotyčnú údajovú štruktúru implementoval sám. Nehovoriac o tom, že všetky takto dostupné údajové štruktúry sú kvalitne odladené a pravdepodobnosť toho, že obsahujú nejakú chybu, je zanedbateľne nízka.

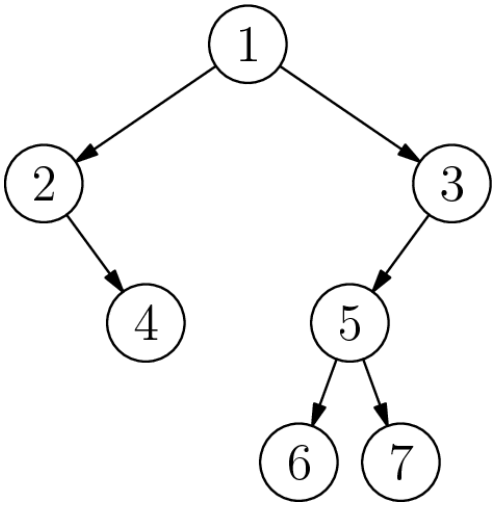
Na záver nášho materiálu o návrhu efektívnych algoritmov si predstavíme jednu netriviálnu údajovú štruktúru: **binárny vyhľadávací strom**. V našej prezentácii sa sústredíme na objasnenie základov, nebudeme teda príliš zachádzať do implementačných detailov.

Binárne stromy

Binárny strom je matematická štruktúra (presnejšie graf). Tvorí ho nejaký kladný počet objektov nazývaných **vrcholy**. Tie sú medzi sebou usporiadané do hierarchie podobnej rodokmeňu.

Na vrchu hierarchie je jediný vrchol nazývaný **koreň**. Koreň má nanajvýš dvoch **synov**: môže mať jedného **ľavého syna** a tiež jedného **pravého syna**. A tak isto ďalej: každý ďalší vrchol v strome má opäť nanajvýš jedného ľavého a nanajvýš jedného pravého syna.

Na nasledujúcom obrázku je jeden možný binárny strom so siedmimi vrcholmi.



Obrázok 11: Binárny strom so siedmimi vrcholmi

Koreň je vrchol s číslom 1. Jeho ľavý syn má číslo 2, pravý má číslo 3. Vrchol číslo 2 ľavého syna nemá, jeho pravý syn má číslo 4. A tak ďalej.

Pokiaľ si takýto strom potrebujeme uložiť v programe, je vhodné použiť smerníky (pointre): Pre každý vrchol budeme mať niekde v pamäti uloženú hodnotu, ktorú obsahuje, a hneď za ňou smerníky na miesta v pamäti, kde sa nachádzajú jeho synovia. Navyše si už len zapamätáme, kde v pamäti máme uložený koreň.

Smerník je po anglicky pointer.

Vyhľadávacie stromy

Binárne stromy môžeme používať ako údajovú štruktúru. Do každého vrcholu si môžeme uložiť jeden údaj – napríklad meno alebo číselné ID jedného z našich zákazníkov.

Prečo ale ukladať údaje do vrcholov stromu, keď ich môžeme mať omnoho pohodlnejšie uložené v obyčajnom poli?

Preto, že ak ich tam uložíme šikovne, veľmi nám to pomôže pri ich efektívnom spracúvaní.

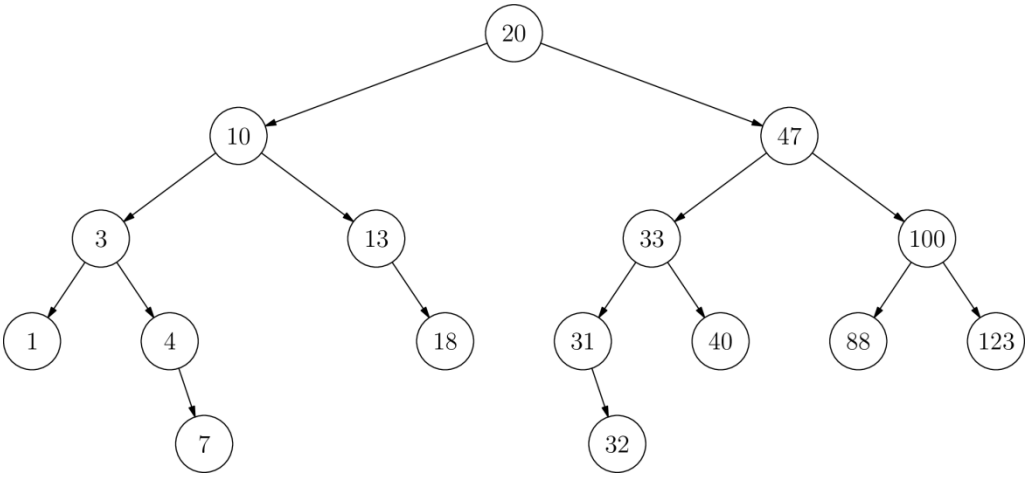
V ďalšom texte budeme pre jednoduchosť predpokladať, že do vrcholov stromu ukladáme prirodzené čísla. Rovnako dobre by sme tam ale mohli ukladať reťazce alebo rovno celé záznamy. Dôležité bude len to, aby sme jednotlivé objekty medzi sebou vedeli porovnávať.

Akonáhle totiž vieme hodnoty vo vrcholoch porovnávať, môžeme ich ukladať tak, aby boli vhodným spôsobom usporiadané. No a v usporiadaných hodnotách, ako už vieme, sa hneď lepšie hľadá.

Presnejšie môžeme tento náš cieľ sformulovať nasledovne: Binárny strom, ktorý má v každom vrchole jeden údaj, voláme **vyhľadávací**, ak sú pre každý vrchol splnené obe nasledujúce podmienky:

- Hodnoty uložené v jeho ľavom synovi (ak ho má) aj vo všetkých jeho potomkoch musia všetky byť **menšie** ako hodnota v samotnom vrchole.
- Hodnoty uložené v jeho pravom synovi (ak ho má) aj vo všetkých jeho potomkoch musia všetky byť **väčšie** ako hodnota v samotnom vrchole.

Na nasledujúcom obrázku je binárny vyhľadávací strom obsahujúci 16 hodnôt.



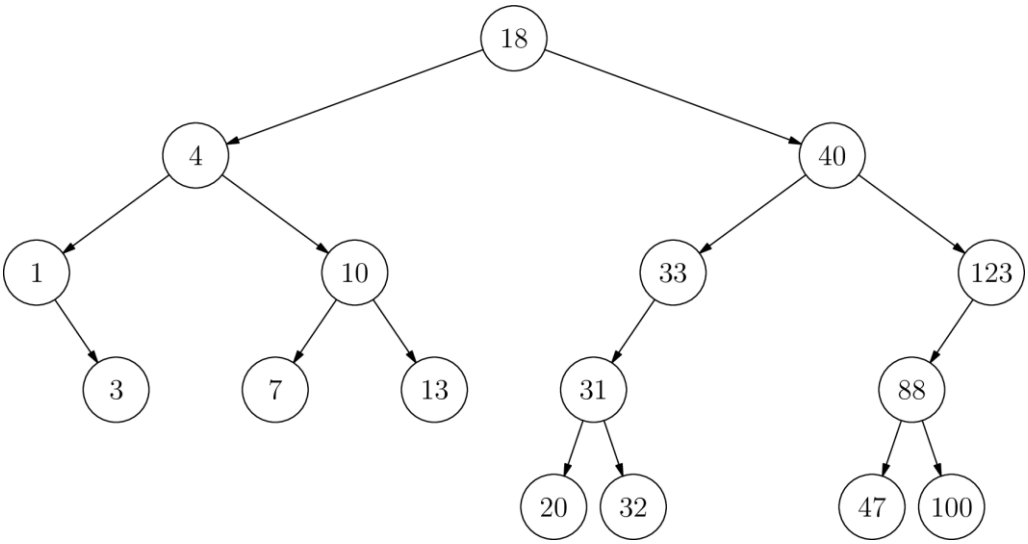
Obrázok 12: Binárny vyhľadávací strom

Overme si, že tento strom naozaj spĺňa požadovanú podmienku.

Ak si napríklad všimneme vrchol s číslom 10:

- Jeho ľavý syn má číslo 3 a jeho potomkovia majú čísla 1, 4 a 7. Všetky tieto hodnoty sú menšie ako 10.
- Jeho pravý syn má číslo 13 a jeho potomok má číslo 18. Obe tieto hodnoty sú väčšie ako 10.

Samozrejme, existuje veľa stromov rozličných tvarov. Tých istých 16 hodnôt môžeme napríklad uložiť aj nasledovne:



Obrázok 13: Binárny vyhľadávací strom s iným rozmiestnením prvkov

Zadanie 30	Navrhnite algoritmus, ktorý dokáže efektívne zistiť, či sa v danom vyhľadávacom strome nachádza daná hodnota.

Postup, ktorý použijeme, bude veľmi podobný práve binárnemu vyhľadávaniu. Označme si danú hodnotu ako X. Začneme v koreni a porovnáme jeho hodnotu s číslom X. Sú tri možnosti, čo sa stane:

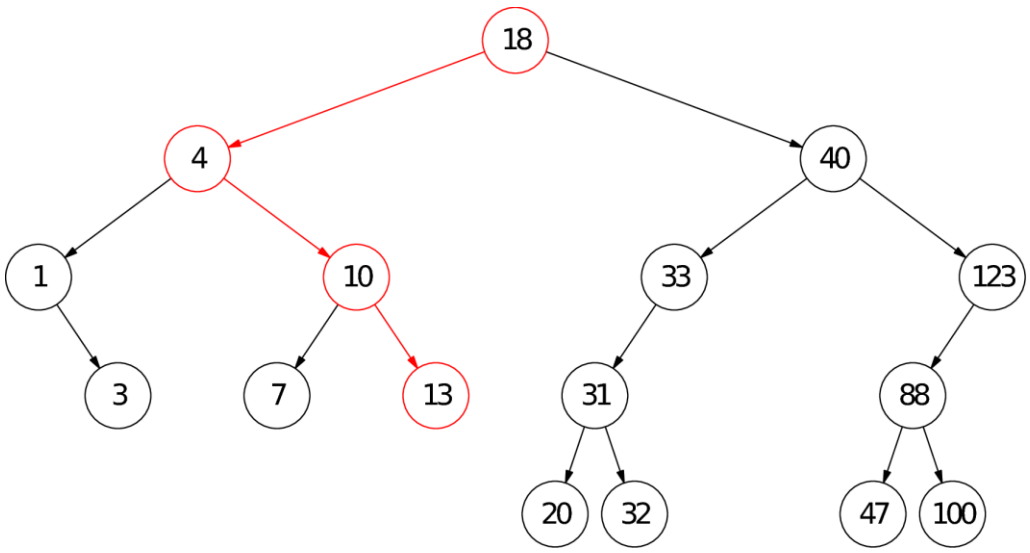
- Ak nastane rovnosť, môžeme skončiť.
- Ak je X **menšie** ako hodnota v koreni, vieme, že musí byť v **ľavom** synovi alebo jednom z jeho potomkov.
- A naopak, ak je X od hodnoty v koreni **väčšie**, musí byť niekde **napravo** od koreňa.

Presunieme sa teda o vrchol nadol na správnu stranu a tam celú úvahu zopakujeme. Toto opakujeme, až kým buď X nenájdeme, alebo nezistíme, že sa v našom strome nenachádza.

Ako by napríklad vyzeral beh tohto algoritmu pre vyššie nakreslený strom (ten s číslom 18 v koreni) a hodnotu $X = 15$?

Začneme v koreni. Keďže $15 < 18$, presunieme sa do jeho ľavého syna. Tam je hodnota 4.

Keďže $15 > 4$, v hľadaní pokračujeme v pravom synovi. Dostali sme sa do vrcholu s číslom 10. Odtiaľ opäť pokračujeme do pravého syna, ten má číslo 13. Aj tu opäť platí, že $15 > 13$, chceli by sme teda pokračovať do pravého syna. Vrchol obsahujúci číslo 13 však už pravého syna nemá – a v tomto okamihu už môžeme s istotou prehlásiť, že číslo 15 sa v našom strome nenachádza.



Zadanie 31

Koľko vrcholov navštívite, ak budete vyššie uvedeným algoritmom v tom istom strome hľadať číslo 88? A ak budete hľadať číslo 32?

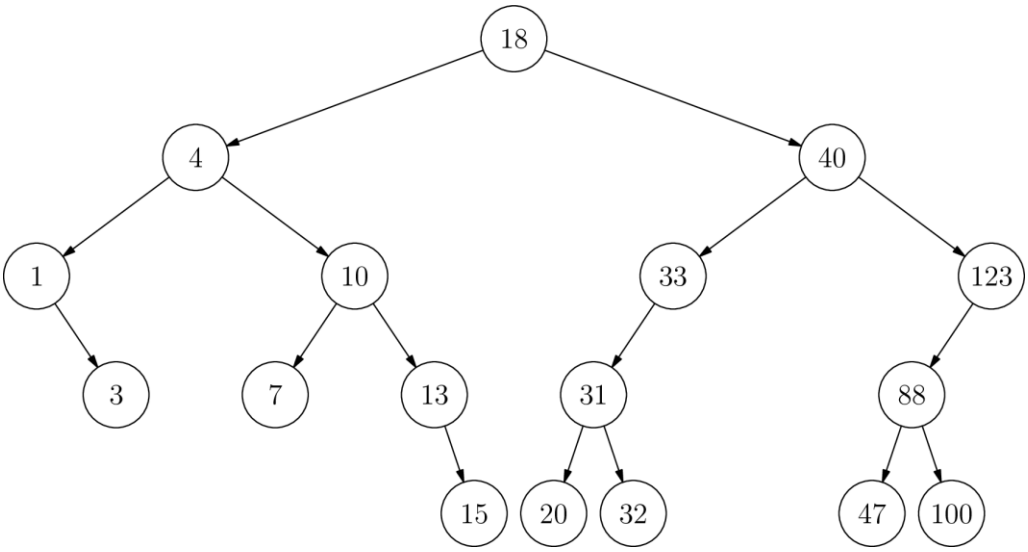
Vkladanie nových údajov

Veľkou výhodou vyhľadávacieho stromu oproti poľu je, že doň ľahko vieme pridávať nové údaje. Pridávanie novej hodnoty X vyzerá skoro rovnako ako jej vyhľadanie:

Postupne prechádzame zhora dole po strome a zisťujeme, či sa v ňom X nachádza. Ak ho nájdeme, sme hotoví. Ak sa X ešte v strome nenachádza, skončí náš algoritmus v okamihu, kedy by chcel z nejakého vrcholu prejsť do jeho syna, ale tam už nič nie je.

Zostaňme pri našom príklade. Keď sme vyhľadávali $X = 15$, skončili sme vo vrchole s číslom 13. Odtiaľ sme chceli ísť do pravého syna, žiaden tam však už nebol. To ale znamená, že keď tomuto vrcholu nového pravého syna pridáme a umiestnime doň číslo 15, bude „na správnom mieste“ – teda aj nový strom, ktorý takto dostaneme, bude opäť vyhľadávací.

Takto bude vyzerať náš predchádzajúci strom po vložení čísla 15:



Obrázok 14: Pridanie prvku do binárneho vyhľadávacieho stromu

Ak vás však zaujíma, môžete nad ním porozmýšľať. Nezabúdajte na to, že binárny vyhľadávací strom musí spĺňať vyššie spomenuté podmienky.

Z vyhľadávacieho stromu vieme podobne efektívne údaje aj vyberať. Algoritmus pre výber údaju a odstránenie príslušného vrcholu je však komplikovanejší, preto sa ním nebudeme zaoberať.

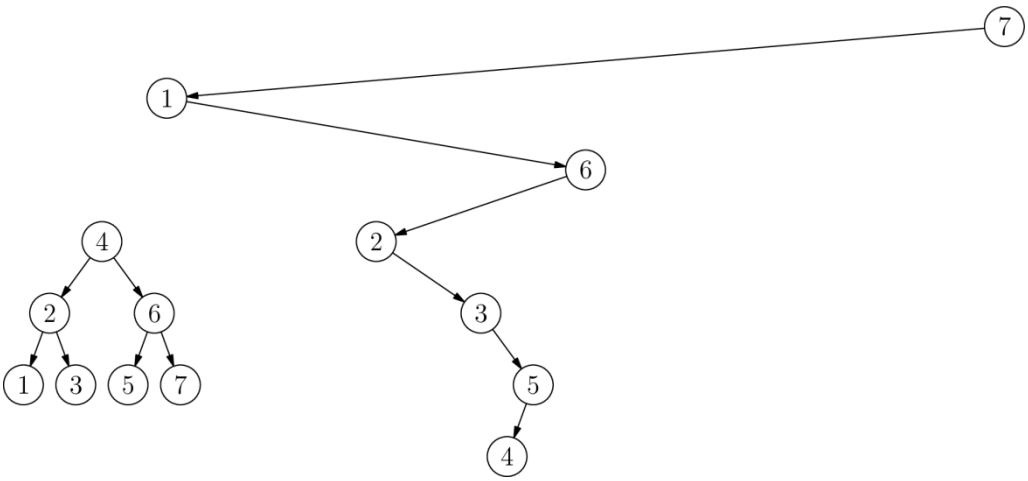
Časová zložitosť operácií

Všimnime si, čo presne sa deje, keď vo vyhľadávacom strome hľadáme konkrétnu hodnotu X: v každom kroku sa presunieme o úroveň hlbšie, do jedného zo synov aktuálneho vrcholu. Počet krokov, ktoré spravíme, vieme teda zhora odhadnúť **hĺbkou** nášho stromu.

Teda platí: čím menšiu hĺbku má vyhľadávací strom, tým lepšie – tým rýchlejšie s ním budeme vedieť robiť všetky operácie.

A tu opäť prichádza k slovu princíp vyváženosti: malá hĺbka stromu súvisí práve s tým, aby bol strom „vyvážený“. Presnejšie, v dobrom strome chceme, aby v každom vrchole platilo, že počet vrcholov, ktoré sú potomkami jeho ľavého syna, by mal byť porovnateľne veľký ako počet vrcholov, ktoré sú potomkami pravého syna.

Ako príklad si pozrime dva rôzne stromy, obsahujúce tú istú množinu údajov:



Obrázok 15: Vyvážený strom (vľavo) a nevyvážený strom (vpravo)

Zatiaľ čo v ľavom, vyváženom strome pri vyhľadávaní ľubovoľného prvku navštívime nanajvýš tri vrcholy, v pravom, ktorý vyvážený nie je, pri hľadaní čísla 4 dokonca navštívime všetkých sedem vrcholov.

Tento rozdiel je opäť omnoho výraznejší pri veľkých množstvách údajov. Milión čísel vieme bez problémov uložiť vo vhodnom vyváženom vyhľadávacom strome tak, aby nám na nájdenie ľubovoľného z nich stačilo niekoľko desiatok krokov.

V praxi používané stromové údajové štruktúry sú komplikovanejšie od tej, ktorú sme si popísali. Počas vkladania a vyberania prvkov totiž so stromom robia navyše ďalšie úpravy, pomocou ktorých zabezpečujú, že je neustále vyvážený. Takéto údajové štruktúry sú teda zaručene veľmi efektívne.

Čo sme sa naučili

Ak chceme vytvárať efektívne algoritmy, je dobré dodržovať istú vyváženosť.

To znamená, že ak problém rozdeľujeme, mali by sme ho deliť na približne rovnaké časti (napríklad polovice). Pomôže nám aj používanie údajových štruktúr, ktoré sú vyvážené.

Literatúra a použité zdroje

[1] Winczer, M. a kol. (2001) *Zbierka úloh Korešpondenčného seminára z programovania (1983-2001)*. Bratislava: Fakulta matematiky, fyziky a informatiky Univerzity Komenského.

[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) *Introduction to Algorithms (3rd ed.)*. MIT Press. ISBN 0-262-03384-4

Kapitola 2: Pravdepodobnostné algoritmy

V tejto časti sa budeme zaoberať úlohou náhody pri riešení úloh. Ukazuje sa, že šikovné využitie náhody nám môže pomôcť vyriešiť úlohy, ktoré „klasickými“ technikami nevieme prakticky riešiť. Cena, ktorú za to zaplatíme, je zvyčajne v strate istoty, že vypočítane riešenie je optimálne, resp. správne. Prekvapujúce je, že túto neistotu vieme často ľubovoľne zmenšiť až na požadovanú mieru, ktorú sme ochotní akceptovať.

Pravdepodobnostný = randomizovaný

Na presnú analýzu týchto algoritmov potrebujeme často vedomosti z teórie pravdepodobnosti, ktoré u našich poslucháčov nepredpokladáme a nemáme ani priestor na ich podrobnejšie vysvetlenie. V tejto kapitole je aj tak matematiky viac ako vo zvyšných dvoch. Je to tak preto, že sme sa usilovali uviesť potrebné argumenty, ktoré zdôvodňujú uvádzané postupy. Inak by hrozilo, že uvedieme len akúsi zbierku, možno zaujímavých, postupov, ktorým môžeme, ale nemusíme veriť. Väčšina materiálu je tak prezentovaná viac v rovine myšlienok než technických detailov a matematických argumentov. Technické detaily uvedieme v prípade, že nebudú vyžadovať viac matematických vedomostí než sa predpokladá na strednej škole.

Tradičné algoritmy, s ktorými ste sa väčšinou stretávali v doterajšom vzdelávaní, sa označujú aj ako *deterministické*. V každom kroku výpočtu algoritmu je jednoznačne určené, aký bude nasledujúci krok a hodnoty premenných. Úsilie tvorcu algoritmu je dokázať, že algoritmus daný problém (vždy) vyrieši správne a rýchlo (to väčšinou znamená, že počet operácií ktoré algoritmus vykoná na nájdenie riešenia závisí polynomiálne od veľkosti vstupných údajov).

Pozor!
Pravdepodobnostná analýza algoritmov je niečo iné ako pravdepodobnostný algoritmus! Pri nej sa študuje pravdepodobnosť výskytu jednotlivých prípadov vstupných údajov, a aká je potrebná práca algoritmu na vyriešenie konkrétneho vstupu. Usilujeme sa ukázať, že algoritmus pre väčšinu

Pravdepodobnostné algoritmy okrem vstupných údajov používajú ešte aj „skrytý“ vstup – zdroj náhodných čísiel, na základe ktorých robia počas behu rozhodnutia. Beh pravdepodobnostného algoritmu na rovnakých vstupných údajoch môže byť vždy iný. Pri návrhu pravdepodobnostných algoritmov sa usilujeme ukázať, že sa budú pravdepodobne správať dobre pre ľubovoľné vstupné údaje (pravdepodobnosť závisí len od náhodných čísiel, ktoré používa algoritmus, ale nie od vstupných údajov).

Výhodou mnohých pravdepodobnostných algoritmov je:

- ich jednoduchosť
- efektívnosť

Pre mnoho problémov je pravdepodobnostný algoritmus najjednoduchší alebo najrýchlejší, alebo oboje súčasne.

Príklady použitie pravdepodobnostných algoritmov sú v algoritmoch z teórie čísiel, v údajových štruktúrach, testovaní zhody, matematickom programovaní, grafových algoritmoch, v zisťovaní počtu určitých štruktúr, paralelných a distribuovaných výpočtoch, pravdepodobnostných dôkazoch existencie a ďalších.

Zaujímavou a ťažkou oblasťou je tzv. *derandomizácia* – úprava pravdepodobnostných algoritmov na rovnako efektívne deterministické.

V nasledujúcom texte si uvedieme príklady viacerých pravdepodobnostných algoritmov a ukážeme si, že sú dva základné typy pravdepodobnostných algoritmov: Monte Carlo a Las Vegas.

Randomizovaný Quicksort

V informatike sa často používa pojem triedenie v zmysle usporiadanie.

Algoritmus triedenia quicksort patrí medzi štandardne používané a je súčasťou každej knižnice programov. Majme množinu čísiel M, ktoré chceme utriediť podľa veľkosti. Keď vieme z M vybrať prvok y tak, že polovica prvkov v M je od neho väčšia a polovica menšia, môžeme postupovať takto: M rozdelíme na dve množiny M_1 a M_2, ktoré rekurzívne utriedime a výsledok dostaneme spojením utriedenej množiny M_1, za ktorú dáme prvok y a utriedenú množinu M_2. Ak prvok y vieme

vybrať na cn operácií, prvky M vieme rozdeliť na M_1 a M_2 na $n-1$ operácií. Teda celkový počet operácii môžeme vyjadriť rekurenciou

(1) $$T(n) \leq 2T\left(\frac{n}{2}\right) + c(n-1),$$

kde $T(n)$ je počet operácií potrebný na utriedenie n prvkov. Riešením uvedenej rekurencie je $T(n) \leq c'n\log_2 n$, pre nejakú konštantu c'. Z praxe vieme, že problematický je výber uvedeného prvku y, ktorý by rozdelil množinu M na dve približne rovnako veľké množiny. Čo v tomto prípade znamená približne rovnako veľké množiny? Skúmaním (1) zistíme, že aj v prípade, že M rozdelíme „nerovnomernejšie“ a jedna časť bude mať iba $n/4$ prvkov a druhá až $3n/4$, potrebný počet operácií bude stále len konštantný násobok $n\log_2 n$, a dokonca to tak bude aj v prípade rozdelenia na časti s $n/8$ prvkami a $7n/8$ prvkami, či ľubovoľného iného rozdelenia na časti s n/a a $(a-1)n/a$ prvkami, kde a nezávisí od n. Takže máme nádej, že s výberom prvku y to nebude až také vážne, ako to na prvý pohľad vyzeralo. Hoci čím nerovnomernejšie budeme prvky deliť, tým väčšia bude konštanta, ktorou budeme násobiť v celkovom počte operácií výraz $n\log_2 n$.

Jedno riešenie sa núka ihneď: vyberme prvok y náhodne spomedzi všetkých prvkov. Môžeme očakávať, že takýmto spôsobom dostaneme pomerne často rozdelenie na zhruba rovnaké časti. Takáto verzia algoritmu quicksort sa nazýva randomizovaný quicksort a schematicky ho môžeme zapísať takto:

Vstup: Množina čísiel $\boldsymbol{M}$.

Výstup: Prvky $\boldsymbol{M}$ utriedené od najmenšieho po najväčší.

1. Vyberme z $\boldsymbol{M}$ náhodne prvok $\boldsymbol{y}$ – pivot; každý prvok z $\boldsymbol{M}$ môže byť vybraný s rovnakou pravdepodobnosťou
2. Porovnávaním $\boldsymbol{y}$ s každým prvkom z $\boldsymbol{M}$ určíme množinu $\boldsymbol{M_1}$ prvkov menších než $\boldsymbol{y}$ a množinu $\boldsymbol{M_2}$ prvkov väčších než $\boldsymbol{y}$.
3. Rekurzívne utriedime $\boldsymbol{M_1}$ a $\boldsymbol{M_2}$. Výstup budú utriedené prvky $\boldsymbol{M_1}$, prvok $\boldsymbol{y}$ a utriedené prvky $\boldsymbol{M_2}$.

Je to príklad algoritmu, ktorý robí náhodné rozhodnutia počas svojej práce. Vieme niečo povedať o počte operácií ktoré celkovo vykoná? Budeme si všímať počet porovnaní. Naším cieľom je určiť aký bude očakávaný počet porovnávaní, odborne nazývaný stredná hodnota. V nasledujúcom uvažujeme nejaký jeden výpočet V uvedeného algoritmu. Označme výsledok algoritmu – utriedené prvky $m_1, m_2, \cdots, m_n$, teda m_i je i-ty najmenší prvok z M. Definujme si pre $1 \leq i < j \leq n$ hodnotu X_{ij} takto

Pozor! Je dôležité, že $m_1, m_2, \cdots, m_n$ sú usporiadané podľa veľkosti.

$$X_{ij} = \begin{cases} 1, \text{ ak } m_i \text{ je porovnávaný s } m_j \\ 0, \text{ ak } m_i \text{ nie je porovnávaný s } m_j \end{cases}.$$

Je zrejmé, že hodnota

$$T = \sum_{i=1}^{n}\sum_{i<j} X_{ij}$$

je celkový počet porovnaní vykonaných počas výpočtu V. Zaujíma nás aká je jej stredná hodnota $\mathrm{E}[T]$, t.j.

$$\mathrm{E}[T] = \mathrm{E}\left[\sum_{i=1}^{n}\sum_{i<j} X_{ij}\right] = \sum_{i=1}^{n}\sum_{i<j} \mathrm{E}\left[X_{ij}\right].$$

Platí, že $\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y]$. Táto vlastnosť sa nazýva lineárnosť strednej hodnoty.

Na to aby sme vedeli spočítať hodnotu $\mathrm{E}[T]$ musíme odhadnúť hodnoty $\mathrm{E}\left[X_{ij}\right]$,

pre $1 \leq i < j \leq n$. Keď si označíme p_{ij} pravdepodobnosť, že prvky m_i a m_j budú počas výpočtu V porovnávané, dostaneme vzhľadom na našu definíciu X_{ij}, že

Stredná hodnota výskytu udalosti, ktorá buď nastane, alebo nenastane, sa rovná pravdepodobnosti, že udalosť nastane.

$$\mathrm{E}[X_{ij}] = p_{ij} \cdot 1 + (1 - p_{ij}) \cdot 0 = p_{ij}.$$

Úlohu sme takto zredukovali na určenie p_{ij}. Na základe uvedeného algoritmu si všimnime pozornejšie, kedy sa vôbec prvky m_i a m_j počas výpočtu V porovnávajú: iba vtedy, keď bol buď jeden, alebo druhý vybraný ako pivot, a to skôr než ľubovoľný z prvkov m_{i+1}, m_{i+2} až m_{j-1}.

Úloha 1.

Prečo keby bol vybraný ako pivot niektorý z prvkov m_{i+1}, m_{i+2} až m_{j-1} skôr než prvky m_i alebo m_j, by sa tieto vôbec neporovnávali? Pomôcka: nakreslite si obrázok.

Každý z prvkov m_i až m_j má rovnakú šancu byť pivotom, to je $j - i + 1$ prvkov, ale iba dva z nich sú také, pri ktorých sa m_i a m_j počas výpočtu V porovnávajú. Sú to práve prvky m_i a m_j. Takže

$$p_{ij} = \frac{2}{j - i + 1}.$$

Po dosadení do vzťahu pre $\mathrm{E}[T]$ máme $\mathrm{E}[T] = \sum_{i=1}^{n} \sum_{i<j} p_{ij} = 2 . \sum_{i=1}^{n} \sum_{i<j} \frac{1}{j-i+1}$. Po ďalších úpravách dostaneme výsledok $2nH_n$, kde H_n je n-té harmonické číslo, t.j. $\sum_{i=1}^{n} \frac{1}{i}$, platí, že $H_n \approx n \ln n$. Takže keď dáme všetko dokopy, máme $\mathrm{E}[T] \approx n \log_2 n$. To znamená, že očakávaný počet operácií randomizovaného quicksortu je konštantný násobok $n \log_2 n$.

Uvedený výsledok platí *pre všetky vstupné údaje*. Nepredpokladáme nič o tom, ako sú rozdelené vstupné údaje (napr. či sú utriedené, alebo nie, a pod.) Náš odhad závisí iba od náhodných rozhodnutí, ktoré robí algoritmus počas svojho výpočtu. Počet operácií sa môže samozrejme líšiť pre jednotlivé výpočty algoritmu, aj keď na vstupe budú tie isté vstupné údaje. Pri uvedenej analýze sme vyšetrovali práve tento počet operácií.

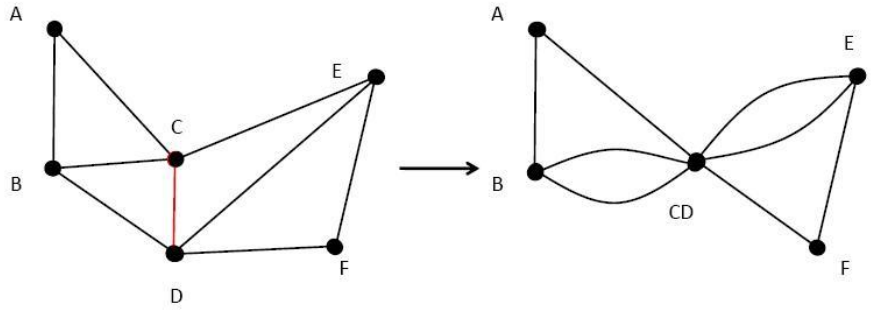
Extrémne nepriaznivý prípad by nastal, keby sa nám podarilo vyberať pivotov tak nešikovne, že by vždy jedna z množín M_1 alebo M_2 bola prázdna. Vtedy by bol počet operácií úmerný až n^2. Táto možnosť je ale vysoko nepravdepodobná.

Uvedenú analýzu by sme mohli ešte ďalej spresniť, že s vysokou pravdepodobnosťou sa skutočný počet operácií nebude líšiť od vypočítaného očakávaného počtu.

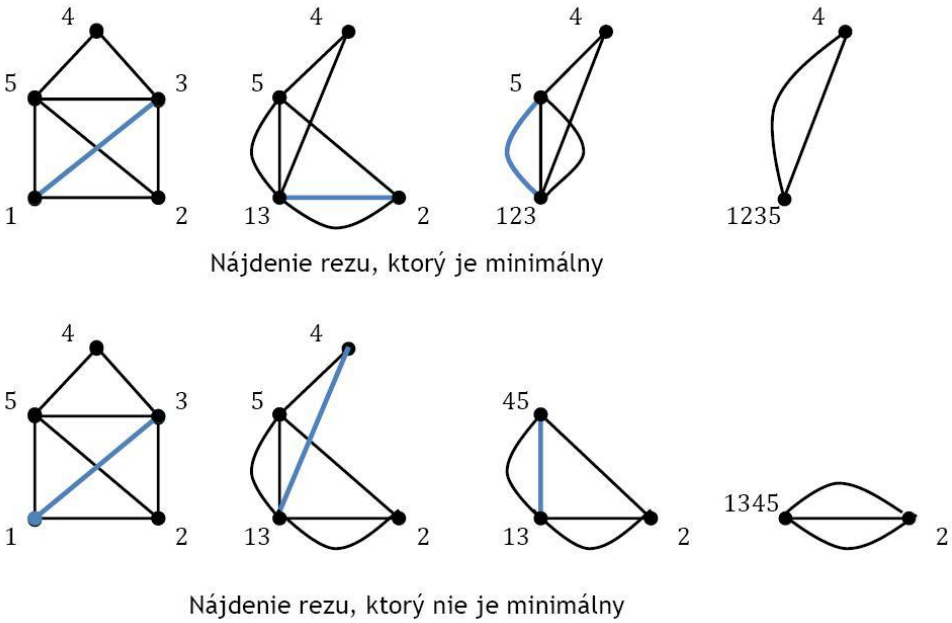
Minimálny rez

V tejto úlohe je cieľom nájsť pre zadaný graf čo najmenšiu množinu hrán, ktoré keď z neho odstránime, spôsobia, že sa graf rozpadne na dve alebo viac súvislých častí. Táto úloha sa vyskytuje v praxi napríklad pri štúdiu spoľahlivosti sietí (nielen počítačových) alebo štruktúry webových stránok. Graf môže obsahovať medzi dvoma vrcholmi aj viac než jednu hranu – násobné hrany.

Budeme opakovať nasledujúci postup: náhodne si vyberieme jednu hranu grafu a oba jej konce – vrcholy zlúčime do jedného vrcholu. Všetky hrany, ktoré boli medzi zlúčenými vrcholmi, z grafu odstránime. Ak je novovzniknutý vrchol spojený s viacerými vrcholmi, ponecháme v grafe všetky hrany (Obr. 1).



Obr. 1: Príklad odstránenia hrany CD a zlúčenia vrcholov



Obr. 2: Príklad úspešného (našiel minimálny rez) a neúspešného behu algoritmu (našiel rez, ale nie minimálny). Modrou je označená hrana, ktorej konce sa zlúčia.

Je zrejmé, že zlúčenie dvoch vrcholov grafu spôsobí pokles ich počtu o 1, takže po $n-2$ opakovaniach tohto kroku bude mať graf práve 2 vrcholy. Výstupom algoritmu budú hrany, ktoré ich spájajú. Uvedomme si, že rez grafu v ľubovoľnom okamihu je aj rezom pôvodného grafu. Ale pôvodný graf mohol mať minimálny rez, ktorý nie je rezom grafu po zlúčení niektorých vrcholov, to je dôvod, prečo rez redukovaného grafu nemusí byť minimálnym rezom pôvodného grafu (Obr. 2). Zaujíma nás, aká je pravdepodobnosť, že takýto algoritmus vypočíta správny výsledok. Ukážeme si, že platí nasledujúce tvrdenie:

Pravdepodobnosť, že algoritmus nájde minimálny rez grafu s n vrcholmi, je aspoň $2/n(n-1)$.

Nech má minimálny rez k hrán. Minimálnych rezov môže mať graf viacero. Zoberme si nejaký jeden z nich a označme si ho R. Je zrejmé, že graf G má aspoň $kn/2$ hrán. Keby mal menej, musel by existovať vrchol, ktorý by bol spojený s ostatnými menej než k hranami, ktoré by tvorili rez menšej veľkosti než k. Odhadneme pravdepodobnosť, že počas procesu odoberania hrán/zlučovania vrcholov si nikdy nevyberieme hranu z R, teda hrany ktoré „prežijú“ do konca, budú práve hrany minimálneho rezu R. Výber hrany a zlúčenie zodpovedajúcich vrcholov budeme nazývať krok.

Pravdepodobnosť, že v prvom kroku algoritmu bude náhodne vybraná hrana z R, je najviac $k/(kn/2) = 2/n$. (R ma k hrán a celý graf ich má $x \geq kn/2$, preto je k/x

Keď $x \leq y$, potom $\frac{1}{x} \geq \frac{1}{y}$.

najviac $k/(kn/2)$.) Teda pravdepodobnosť, že v prvom kroku si nevyberieme hranu z R, je aspoň $1 - 2/n$.

Predpokladajme, že v prvom kroku sme si nevybrali hranu z R. Po prvom kroku máme graf s $n-1$ hranami, ktorý má minimálny rez veľkosti k. Stupeň každého vrchola je teda aspoň k, takže graf má najmenej $k(n-1)/2$ hrán. Analogicky ako v prvom kroku, aj teraz je pravdepodobnosť toho, že si vyberieme hranu, ktorá je z R, najviac $k/(k(n-1)/2) = 2/(n-1)$ a pravdepodobnosť, že si nevyberieme hranu z R, je aspoň $1 - 2/(n-1)$.

Keď predpokladáme, že žiadna z prvých $i-1$ vybraných hrán nebola z minimálneho rezu R, je pravdepodobnosť, že i-tu hranu náhodne nevyberieme z R, je $1 - 2/(n-i+1)$.

Aká je pravdepodobnosť p, že všetkých $n-2$ hrán sme nevybrali z R? Musíme vynásobiť pravdepodobnosti, že sme v žiadnom kroku nevybrali hranu z R, a dostaneme:

Dali sme na spoločného menovateľa.

$$p \geq \prod_{i=1}^{n-2}\left(1 - \frac{2}{(n-i+1)}\right) = \prod_{i=1}^{n-2}\left(\frac{n-i-1}{n-i+1}\right) =$$

Všimnite si, že čitatele a menovatele sa navzájom vykrátia.

$$= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) =$$

$$= \frac{2}{n(n-1)},$$

čo sme chceli ukázať.

Napríklad keď zvolíme $k = n^2/2$, dostaneme

$$\left(1 - \frac{2}{n^2}\right)^{n^2/2} < \frac{1}{\mathrm{e}},$$

pretože $\left(1 + \frac{x}{n}\right)^n \leq \mathrm{e}^x$, pre $|x| < n$, $n \in \mathbb{N}$

Takže pravdepodobnosť, že algoritmus nájde nejaký minimálny rez, je väčšia než $2/n^2$. Algoritmus teda môže dať aj nesprávny výsledok (Obr. 2). Situáciu môžeme ale zachrániť tak, že algoritmus spustíme opakovane k-krát. Pravdepodobnosť, že beh algoritmu dá zlý výsledok, je najviac $1 - 2/n^2$. Pravdepodobnosť, že by sa algoritmus „pomýlil“ vo všetkých k opakovaných behoch, je $(1 - 2/n^2)^k$, čo sa zo zvyšujúcou hodnotou k zmenšuje (k hodnote 0). Samozrejme za znižovanie pravdepodobnosti zlej odpovede „platíme“ zvyšovaním času výpočtu.

Je typické, že uvedený pravdepodobnostný algoritmus je veľmi jednoduchý. Deterministický algoritmus, ktorý rieši tú istú úlohu, je oveľa komplikovanejší (je založený na riešení úlohy o maximálnom toku v sieti).

Las Vegas a Monte Carlo

Predchádzajúce príklady randomizovaného quicksortu a určovania minimálneho rezu v grafe predstavujú dva rôzne typy pravdepodobnostných algoritmov. Quicksort dá vždy správny výsledok – utriedi prvky. Počas svojej práce sa snaží využiť náhodné rozhodnutia na to, aby sa čo najrýchlejšie dostal k správnemu riešeniu. Jednotlivé behy algoritmu s tými istými vstupnými údajmi sa môžu od seba líšiť iba časom potrebným na výpočet. Takéto algoritmy sa nazývajú *Las Vegas algoritmy.*

V niektorých prípadoch sa hodí viac Las Vegas algoritmus, inokedy zasa Monte Carlo. Nedá sa jednoznačne vo všeobecnosti povedať, ktorý je lepší.

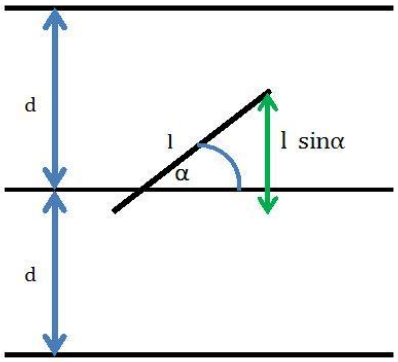
Oproti tomu sa algoritmus pre minimálny rez môže pomýliť, dá nesprávny výsledok. Takýto algoritmus voláme *Monte Carlo*. Jeho užitočná vlastnosť je, že pravdepodobnosť chybnej odpovede sa, za cenu času potrebného na výpočet, dá ľubovoľne zmenšiť jeho opakovaným vykonávaním. Pre rozhodovacie problémy (ich výsledok je odpoveď ÁNO alebo NIE) sú dva druhy Monte Carlo algoritmov, podľa toho, kedy môžu odpovedať nesprávne. Monte Carlo s obojstrannou chybou sú také, keď je nenulová pravdepodobnosť chybnej odpovede aj v prípade ANO, aj NIE. Jednostranná chyba je, keď je nulová pravdepodobnosť chybnej odpovede aspoň pri jednej z odpovedí ANO alebo NIE.

Hovoríme, že algoritmus Monte Carlo, resp. Las Vegas je efektívny, keď je očakávaný čas jeho behu ohraničený polynomickou funkciou vzhľadom na veľkosť vstupu.

Niekedy sa pod typ Las Vegas zahŕňajú aj algoritmy využívajúce náhodu, ktoré môžu dať aj odpoveď „neviem“.

Buffonova ihla

Táto známa úloha patrí medzi jeden z prvých pravdepodobnostných algoritmov a dá sa dobre simulovať na počítači a dokonca aj mechanicky.



Obr. 3: Znázornenie náhodnej polohy ihly

Georges Louis Leclerc, Comte de Buffon, francúzsky prírodovedec (1707–1788), publikoval tento problém v r. 1777.

Na podložku s nakreslenými paralelnými čiarami, ktoré sú od seba vzdialené o rovnakú vzdialenosť d, hádžeme náhodne ihlu dĺžky $l \leq d$ a zanedbateľnej šírky a zisťujeme, či padne tak, že pretne čiaru. Úlohou je odhadnúť koľkokrát ihla pretne čiaru bez toho, aby sme skutočne ihlu hádzali. Prekvapujúco týmto spôsobom môžeme vypočítať číslo π s ľubovoľnou presnosťou (ale pomaly).

Úloha 2.

Napíšte program na simulovanie hádzania Buffonovej ihly. Zistite výsledky simulácie pre rôzne vzdialenosti čiar, dĺžky ihly a počty hodov. Skúste $1000, 10000, 100000, \cdots$.

Dá sa ukázať, že ak chceme π spočítať na p desatinných miest, musíme ihlu hodiť približne 10^{2p} krát. To je dosť „pomaly“. Na výpočet π existujú oveľa efektívnejšie spôsoby.

Ak ihla padne vzhľadom na čiaru pod uhlom α, jej priemet na kolmicu k čiare bude mať dĺžku $l \sin \alpha$ (Obr. 3). Pravdepodobnosť, že ihla ležiaca na priamke, ktorá zviera s čiarami uhol α, pretne čiaru je $(l \sin \alpha)/d$. Aby sme zistili s akou pravdepodobnosťou ihla pretne čiaru v ľubovoľnom prípade, musíme urobiť priemer cez všetky hodnoty uhla α z intervalu $\langle 0, \pi)$. Dostaneme

$$\frac{1}{\pi}\int_0^{\pi} \frac{l \sin \alpha}{d}\,\mathrm{d}\alpha = \frac{l}{\pi d}\int_0^{\pi} \sin \alpha\,\mathrm{d}\alpha = \frac{l}{\pi d}[-\cos \alpha]_0^{\pi} = \frac{l}{\pi d}[-(-1)-(-1)] = \frac{2l}{\pi d}.$$

Ak teda použijeme ihlu dĺžky $d/2$ bude pravdepodobnosť, že ihla pretne čiaru $1/\pi$. Teraz môžeme postup „obrátiť“. Keď ihlu hodíme n-krát a z toho k-krát ihla pretne čiaru, hodnota zlomku n/k by sa mala s rastúcim n približovať k π.

Pre úplnosť ešte dodáme, že keby sme chceli uvedenú simuláciu hádzania ihly naprogramovať v programe musíme generovať náhodný uhol α, z ktorého potom vypočítate $\sin \alpha$. Aby boli výpočty v programe čo najjednoduchšie, je vhodné zvoliť si $l = d/2$.

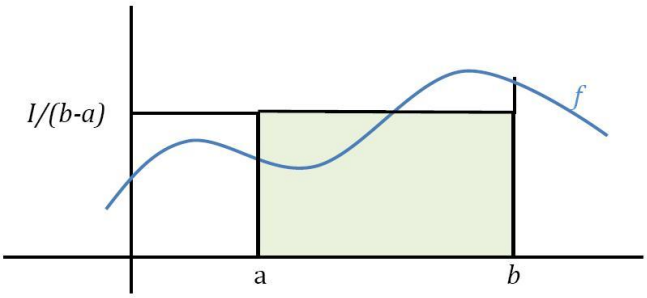
A prečo nemôžeme generovať rovno náhodnú hodnotu $\sin \alpha$?

Výpočet plochy (objemu)

Spolu s predchádzajúcim problémom Buffonovej ihly je výpočet plochy (určitého integrálu) ďalším príkladom numerického pravdepodobnostného algoritmu. Tieto metódy sa nazývajú niekedy aj „Monte Carlo“ metódy. (Len pripomíname, že v súvislosti s pravdepodobnostnými algoritmami sa ale v súčasnosti termín Monte Carlo algoritmus používa v inom význame!)

Názov Monte Carlo pochádza z článku, ktorý napísali Metropolis a Ulam v r. 1949.

Chceme vypočítať $I = \int_a^b f(x)\,\mathrm{d}x$, čo je plocha pod funkciou $f(x)$ na intervale $\langle a, b\rangle$. Hlavná myšlienka výpočtu plochy je znázornená na Obr. 4. Je zrejmé, že obdĺžnik s výškou $I/(b-a)$ má plochu rovnú I. Takže na intervale $\langle a, b\rangle$ musí byť priemerná výška krivky a obdĺžnika rovnaká a rovnajúca sa $I/(b-a)$. Pravdepodobnostný algoritmus výpočtu integrálu je založený na odhade priemernej výšky krivky na intervale $\langle a, b\rangle$, ktorú vynásobíme $b-a$. Priemernú výšku spočítame tak, že náhodne vyberieme n bodov, v ktorých vypočítame hodnotu f, tie spočítame a nakoniec vydelíme s n.



Obr. 4: Zelenou je vyfarbená plocha, ktorá sa rovná veľkosťou ploche pod krivkou na intervale $\langle a, b\rangle$.

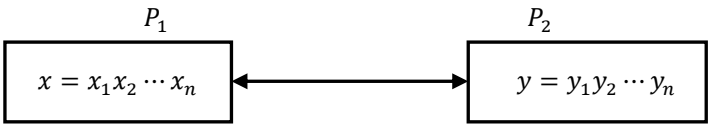
```
type
  TFunc = function(x : Real) : Real;

function IntegrovanieMonteCarlo(f : TFunc;
                                n : Integer; a, b : Real);
var
  Sucet : Real;
  i : Integer;
begin
  Sucet := 0;
  for i := 1 to n do
    Sucet := Sucet + f(Random * (b - a));
  Result := (b - a)*(Sucet / n);
end;
```

Prirodzená otázka je aké musí byť n, aby sme vypočítali I s presnosťou na p desatinných miest? Bohužiaľ rovnako ako v prípade Buffonovej ihly, každá presná cifra výsledku vyžaduje 100-krát väčšiu prácu počítača. Deterministické metódy integrovania sú predsa oveľa efektívnejšie! Keby sme body, v ktorých počítame hodnotu funkcie f, nevyberali náhodne, ale systematicky, napríklad $a + i * \Delta$, kde $i = 0,1,\ldots,n-1$ a $\Delta = (b-a)/n$, dostali by sme presnú hodnotu I s oveľa menším n. Nevýhodou každej deterministickej metódy ale je, že pre ňu existuje nejaký príklad „nevhodnej“ funkcie, ktorá ju prekabáti a vypočítaná hodnota nebude správna. Pri pravdepodobnostnom algoritme takáto nevhodná funkcia neexistuje, ale na druhej strane so zanedbateľne malou pravdepodobnosťou môže vypočítať zlý výsledok aj pre obyčajnú funkciu.

Výhoda pravdepodobnostného algoritmu sa ukáže ak potrebujeme vypočítať viacnásobný integrál. V prípade deterministického algoritmu používame systematický spôsob vyberania bodov, v ktorých počítame hodnotu funkcie. Keď máme napríklad 100 bodov na určenie jednoduchého integrálu, aby sme dosiahli rovnakú presnosť, na určenie dvojitého integrálu budeme potrebovať takmer isto 100×100 bodov. Pri metóde integrovania Monte Carlo nezáleží veľmi na tom, koľkorozmerný je integrál. Samozrejme, aj tu rastie počet potrebných operácií, ale dimenzia integrálu nemá taký veľký vplyv ako pri deterministickej metóde. Od rozmeru 4 sa v praxi používajú na integrovanie rôzne varianty metódy Monte Carlo, lebo nepoznáme žiadnu porovnateľne efektívnu deterministickú metódu.

Určovanie zhody

Uvažujme nasledujúcu situáciu: majme dva od seba vzdialené počítače P_1 a P_2 (Obr. 5), napríklad z bezpečnostných dôvodov. V oboch sú tie isté údaje, napríklad nejaká veľká databáza. Všetky operácie sa súčasne vykonávajú v oboch databázach. Po nejakom čase potrebujeme zistiť, či sú obe databázy naozaj identické.



Obr. 5 Dva počítače s dvomi databázami x a y.

V nasledujúcom texte ukážeme, ako nám náhoda pomôže neuveriteľným spôsobom zmenšiť množstvo údajov, ktoré musíme vzájomne porovnať a súčasne môžeme dosiahnuť ľubovoľne vysokú mieru istoty, že údaje sú zhodné, a v prípade, že nie sú zhodné, dokonca sto percentnú istotu.

Musíme navrhnúť algoritmus, na základe ktorého určíme, či P_1a P_2 obsahujú rovnaké x a y. Takýto postup sa nazýva *komunikačný protokol* a mierou jeho efektívnosti bude, koľko informácií si musia vzájomne P_1 a P_2 vymeniť. Množstvo informácii budeme merať počtom bitov. Predpokladajme, že počet bitov n je veľký, napr. 10^{16}. Klasickým deterministickým spôsobom musíme vždy medzi P_1 a P_2 vymeniť všetkých n bitov (dá sa to aj matematicky dokázať). Nehľadiac na to, že pri takom množstve chýb môže nastať prenosová chyba, trvá to nezanedbateľne dlhú dobu, a to aj vtedy, keď použijeme 1Gb prenosovú rýchlosť, a už ani nehovoriac o počte prípadných DVD nosičov, keby sme údaje chceli preniesť takýmto spôsobom.

Gb je gigabit.

Označme $\text{Číslo}(x)$ prirodzené číslo, ktorého zápis v dvojkovej sústave je reťazec x, t.j. $\text{Číslo}(x) = \sum_{i=1}^{n} 2^{n-i}\, x_i$. Ukážeme si, že riešením bude nasledujúci pravdepodobnostný protokol na určenie zhody:

Počítač P_1 má postupnosť bitov x dĺžky n, $x = x_1x_2 \cdots x_n$ a počítač P_2 má postupnosť bitov y dĺžky n, $y = y_1y_2 \cdots y_n$.

1. P_1 si zvolí z intervalu $[2, n^2]$ náhodne prvočíslo p. Všetky prvočísla majú rovnakú šancu, aby boli vybrané.

2. P_1 vypočíta $s = \text{Číslo}(x) \bmod p$ a pošle P_2 binárnu reprezentáciu čísiel s a p.

3. P_2 prečíta s a p a vypočíta $q = \text{Číslo}(y) \bmod p$.
   Ak platí, že $q \neq s$, P_2 dá výsledok „x **sa nerovná** y“.
   Ak platí, že $q = s$, P_2 dá výsledok „x **sa rovná** y“.

Všimnime si koľko bitov informácií si musia P_1 a P_2 vymeniť. Keď si uvedomíme, že $s < p < n^2$ je zrejmé, že celková dĺžka správ v bode 2 je najviac

$$2 \cdot \lceil \log_2 n^2 \rceil = 4 \cdot \lceil \log_2 n \rceil.$$

Pomôcka: na zápis čísla n potrebujeme $\lfloor \log_2 n \rfloor + 1$ bitov, teda $\lceil \log_2 n \rceil$ bitov bude na zápis istotne stačiť.

Teda pre našu databázu údajov veľkosti $n = 10^{16}$ dostaneme $4 \cdot 16 \cdot \lceil \log_2 10 \rceil = 256$ (8 bajtov). To je neporovnateľne menej než 10^{16}!

Je takýto algoritmus vôbec správny?

Úloha 2.

Zoberme si päťbitové postupnosti napr. $x = 01011$ a $y = 11110$ ($\text{Číslo}(x) = 11$, $\text{Číslo}(y) = 28$). Vykonajte uvedený algoritmus na určenie zhody. Pomôcka: $n = 5$.

Úloha 3.	Predpokladajte, že si algoritmus zvolil prvočíslo $p = 5$ alebo $p = 17$. Vykonajte algoritmus v oboch prípadoch. Čo ste zistili?

Takže uvedený algoritmus sa môže pomýliť. Mýli sa často?

Videli sme, že v úlohách 2 a 3 pri niektorých voľbách prvočísiel dal algoritmus správne výsledky a pri iných odpovedal nesprávne. V ďalšom sa budeme usilovať určiť pravdepodobnosť, že algoritmus dá chybný výsledok. Pre každý vstup x a y rozdelíme množinu prvočísiel na dve podmnožiny, ktoré označíme *prvočísla dobré pre* (x, y), kde algoritmus dá správny výsledok, a *prvočísla zlé pre* (x, y), kde dá chybný výsledok. Vzhľadom na to, že je rovnako pravdepodobné, aby si algoritmus zvolil ľubovoľné spomedzi prvočísiel, je pravdepodobnosť chybnej odpovede

$$\frac{\text{počet zlých prvočísiel pre } (x,y)}{\text{počet všetkých prvočísiel } \leq n^2}.$$

Skúsme odhadnúť čitateľ aj menovateľ tohto zlomku. Menovateľ je ľahší, lebo známa veta o prvočíslach hovorí, že

$$\text{počet prvočísiel menších alebo rovných než } m \text{ je väčší než } \frac{m}{\ln m}, \text{ pre } m > 67.$$

V našom prípade $m = n^2$, takže počet prvočísiel $\leq n^2$ je väčší než $n^2/(2 \ln n)$ pre $n \geq 9$. A čo čitateľ? Chceme ukázať, že pre každý vstup (x, y) je počet zlých prvočísiel najviac $n - 1$. V takom prípade vieme odhadnúť zlomok vyjadrujúci pravdepodobnosť pomýlenia sa:

$$\frac{\text{počet zlých prvočísiel pre } (x,y)}{\text{počet všetkých prvočísiel } \leq n^2} \leq \frac{n-1}{n^2/2\ln n} \leq \frac{2 \ln n}{n}.$$

Takže pravdepodobnosť chybného výsledku pre vstup (x, y) je najviac

$$\frac{2 \ln n}{n}.$$

Pre $n = 10^{16}$ to je menej než 10^{-14}. V prípade, že sa nám takáto pravdepodobnosť zdá ešte stále privysoká, môžeme ju ešte znížiť opakovaným spustením rovnakého algoritmu. Každý beh algoritmu je nezávislý od predchádzajúceho. Všimnime si, že keď dostaneme výsledok: „nie sú rovnaké“, je to určite tak a $x \neq y$. Chyba môže nastať iba v prípade odpovede: „sú rovnaké“. Pravdepodobnosť, že by sa algoritmus pomýlil povedzme k-krát za sebou, je preto

$$\left(\frac{2 \ln n}{n}\right)^k,$$

čo je pre $k = 10$ menej než 10^{-140}, a to je už prakticky zanedbateľné. Je oveľa pravdepodobnejšie, že sa vyskytne chyba výpočtu v dôsledku chyby hardvéru.

Ešte nám ostalo ukázať, že počet zlých prvočísiel pre každý vstup (x, y) je naozaj najviac $n - 1$.

Keď je $x = y$ je každé prvočíslo dobré – pre ľubovoľné prvočíslo p platí, že $\text{Číslo}(x) \bmod p = \text{Číslo}(y) \bmod p$.

V prípade, že je $x \neq y$, dostaneme zlú odpoveď „x sa rovná y“ len keď $\text{Číslo}(x) \bmod p = \text{Číslo}(y) \bmod p$, čo znamená, že $p \left| \left(\text{Číslo}(x) - \text{Číslo}(y) \right) \right.$. Takže prvočíslo p je zlé pre (x, y) práve vtedy, keď delí hodnotu $z = \text{Číslo}(x) - \text{Číslo}(y)$. Odhadneme veľkosť z. Pripomeňme si, že x aj y sú postupnosti bitov, ktorých dĺžka je n, takže aj čísla, ktoré reprezentujú sú $< 2^n$, teda platí aj $z < 2^n$. Číslo z môžeme napísať rozložené na prvočísla:

Keď $s = X \bmod p = Y \bmod p$, potom $X = k_1 p + s$ a $Y = k_2 p + s$ a $X - Y = (k_1 - k_2)p$

$$z = p_1^{e_1} p_2^{e_2} p_3^{e_3} \cdots p_k^{e_k},$$

kde $p_1 < p_2 < \cdots < p_k$ a $e_1, e_2, \cdots e_k$ sú nezáporné celé čísla. Najmenšie prvočíslo je 2, takže $2^k < z$, ale už vieme, že platí aj $z < 2^n$, teda $2^k < 2^n$, to platí, keď $k \leq n-1$, čo sme chceli ukázať.

Testovanie prvočíselnosti

Zisťovanie prvočíselnosti v dnešnej dobe nie je len matematická zábavka, ale prvočísla hrajú dôležitú úlohu napríklad v kryptografii.

Millerov test prvočíselnosti

Pierre Fermat (1601-1665) bol francúzsky právnik a matematik - samouk.

Pripomeňme si *malú Fermatovu vetu* :

> Nech p je prvočíslo a a ľubovolné číslo, potom $a^P \equiv a\ (mod\ p)$.

$b \equiv a \pmod n$ znamená, že $n\,|(b-a)$. Alebo ešte inak: že b aj a majú rovnaký zvyšok po delení s n

Keď chceme overiť, či je p prvočíslo, nemôžeme využiť predchádzajúcu vetu, lebo nanešťastie neplatí opačná implikácia. Netreba sa však vzdávať. Môžeme využiť, že ak nejaké n je prvočíslo, musí byť $a^{n-1} \equiv 1 \pmod n$. Teda ak $a^{n-1} \not\equiv 1 \pmod n$, s istotou vieme, že n nie je prvočíslo.

Keď začneme s nepárnym prvočíslom n, podľa Fermatovej vety musí platiť, že $a^{n-1} \equiv 1 \pmod n$. Vieme, že $n-1$ je párne a môžeme vypočítať $x = a^{\frac{n-1}{2}}$. Pretože $x^2 \equiv 1 \pmod n$, musí byť $x \equiv \pm 1 \pmod n$. V prípade, že $x = 1$ a $\frac{n-1}{2}$ je párne, môžeme pokračovať rovnakým spôsobom a vyrátať $y = a^{\frac{n-1}{4}}$. Z rovnakých dôvodov ako predtým, pretože $y^2 \equiv 1 \pmod n$, musí byť aj $y \equiv \pm 1 \pmod n$. Takto môžeme pokračovať, pokým nedosiahneme hodnotu -1 alebo nepárny exponent (môžu nastať aj oba prípady súčasne). Intuitívne je idea zrejmá, zložené číslo sa len ťažko zamaskuje za prvočíslo. Dostali sme test, ktorý navrhol Gary Lee Miller v roku 1976.

Millerov test prvočíselnosti vzhľadom na bázu a

Nech n je nepárne prvočíslo a nech $\mathrm{nsd}(a,n) = 1$ a $1 < a < n$.

```
begin
  k := n - 1;
  r := a^k mod n;
  while (r = 1) and not Odd(k) do begin
    k := k/2;
    r := a^k mod n
  end;
  if (r = -1) or (r = n - 1) then
    Prešiel
  else
    Neprešiel
end
```

Zložené číslo, ktoré prejde testom rovnako, ako by ním prešlo aj prvočíslo, budeme nazývať *silné pseudoprvočíslo vzhľadom na bázu* a. Je zrejmé, že keď je n prvočíslo, nemôže sa stať, že neprejde Millerovým testom (uvedomme si, že $x \equiv \pm 1 \pmod n$ znamená, že $x = 1$ alebo $x = n-1$), čo môžeme sformulovať do nasledujúceho tvrdenia:

> Ak prirodzené číslo neprejde Millerovým testom, je zložené.

Pravdepodobnostné testovanie prvočíselnosti – Rabinov test

Keď sa pokúsime nájsť najmenšie n, ktoré je súčasne silným pseudoprvočíslom vzhľadom na bázy 3, 5 a 7, získame podozrenie, že výskyt čísiel, ktoré sú silnými pseudoprvočíslami súčasne vzhľadom na väčší počet báz, je veľmi zriedkavý. Ozaj je pravda, že čím väčším počtom Millerových testov číslo prejde, tým je väčšia šanca, že je prvočíslom. Nasledujúcu vetu uvedieme bez dôkazu.

Nech n je nepárne zložené číslo. Potom n spĺňa Millerov test najviac pre $(n-1)/4$ báz b, $1 \leq b \leq n-1$.

Inak povedané, n nemôže byť silným pseudoprvočíslom vzhľadom na všetky bázy. Na to, aby sme sa presvedčili, či je číslo n zložené, by nebolo praktické skúšať všetkých $(n-1)/4$ báz. Keď n prejde Millerovým testom vzhľadom na bázu b, je pravdepodobnosť toho, že sme si zvolili bázu, pri ktorej sa to stane, rovná $1/4$. Keď n prejde k Millerovými testami vzhľadom na rôzne bázy, pričom predpokladáme, že boli vybrané náhodne, tak pravdepodobnosť, že je n zložené je $1/4^k$. Istota, že n je prvočíslo, rastie s rastúcim k. Táto metóda sa nazýva *Rabinov pravdepodobnostný test*, lebo ho vymyslel Michael Oser Rabin. Miller ukázal, že za istých špeciálnych predpokladov (že platí rozšírená Riemannova hypotéza) zložené n neprejde testom pre bázu menšiu než $2(\log n)^2$, čo je malé číslo aj pre veľmi veľké n.

Úloha 4.	Napíšte funkciu, ktoré pre zadané a, k a n výpočíta hodnotu $a^k \bmod n$. Pokúste sa aby procedúra potrebovala na výpočet len približne $\log_2 n$ operácií násobenia. Ako rýchlo viete vynásobiť b-bitové číslo c-bitovým?
Úloha 5.	Ak ste absolvovali modul programovanie v Jave alebo v Pythone, môžete skúsiť naprogramovať Rabinov test. Oba jazyky podporujú aritmetiku s veľkými číslami.

Splniteľnosť logických formúl

Majme logické premenné $x_1, x_2, \cdots, x_n$. Výraz

$$\Phi = F_1 \wedge F_2 \wedge \cdots \wedge F_m,$$

$m \geq 1$, a F_i je logický výraz (klauzula) obsahujúci len logický súčet niekoľkých (môže byť aj len jedna) z uvedených premenných x sa nazýva *konjunktívna normálna forma* (KNF). Úlohou je nájsť také hodnoty premenných x, aby bolo splnených čo najviac výrazov F_i. Táto úloha je príkladom optimalizačnej úlohy. Nájsť optimálne riešenie je ťažké (NP-ťažké). Pokúsime sa nájsť také priradenie hodnôt premenným, pri ktorom je s vysokou pravdepodobnosťou splnených dostatočné množstvo výrazov.

Prekvapujúco úspešná je takáto jednoduchá stratégia (náhodný výber):

1. Zvoľme priradenie $(\alpha_1, \alpha_2, \cdots, \alpha_n)$ premenným $x_1, x_2, \cdots, x_n$, pričom pre $i = 1,2, \cdots, n$ je s pravdepodobnosťou $1/2$ α_i buď 0 alebo 1.
2. Výsledok je $(\alpha_1, \alpha_2, \cdots, \alpha_n)$.

Je zrejmé, že každé priradenie hodnôt premenným je riešenie. Tento algoritmus nikdy nevráti zlé riešenie. Vzhľadom na to, že každý algoritmus musí nejako vypísať výsledok, potrebuje na svoju prácu aspoň čas úmerný počtu prvkov na vstupe. Navrhovaný algoritmus tento limit dosahuje, je teda z pohľadu času na výpočet je optimálny.

Odhadnime ako dobrý je uvedený algoritmus – určíme podiel splnených klauzúl ku všetkým klauzulám v Φ. Budeme postupovať podobne ako pri randomizovanom quicksorte. Pre každý výraz Φ a priradenie hodnôt α si definujeme hodnotu

$$Z_i(\alpha) = \begin{cases} 1, \text{keď je } F_i \text{ splnené priradením } \alpha \\ 0, \text{keď nie je } F_i \text{ splnené priradením } \alpha \end{cases}$$

Hodnoty $Z = \sum_{i=1}^{m} Z_i$ je vlastne počet splnených klauzúl, zaujíma nás jej stredná hodnota

$$E[Z] = E\left[\sum_{i=1}^{m} Z_i\right] = \sum_{i=1}^{m} E[Z_i].$$

Hodnotu $E[Z_i]$ zistíme ľahko, lebo v našom prípade sa rovná pravdepodobnosti, že $F_i$je splnená. Keď majú klauzule tvar

$$F_i = l_{i1} \vee l_{i2} \vee \cdots \vee l_{ik},$$

kde l_{ij} je niektorá z n premenných x. Klauzula F_i nie je splnená, keď nie sú splnené súčasne všetky $l_{i1}, l_{i2}, \cdots, l_{ik}$. Pravdepodobnosť, že niektoré l_{ij} nie je splnené je $1/2$. V bode 1. sme zvolili jednotlivé hodnoty α_i nezávisle na sebe, preto je pravdepodobnosť, že F_i nie je splnená presne

$$\left(\frac{1}{2}\right)^k = \frac{1}{2^k}.$$

Takže pravdepodobnosť, že F_i je splnená je

$$E[Z_i] = 1 - \frac{1}{2^k}.$$

Pretože každá klauzula F_i obsahuje aspoň jednu premennú, je $E[Z_i] \geq 1/2$. Keď všetko spojíme, dostávame

$$E[Z] = \sum_{i=1}^{m} E[Z_i] \geq \sum_{i=1}^{m} \frac{1}{2} = \frac{m}{2},$$

čo znamená, že pri náhodnom priradení hodnôt 0 a 1 premenným môžeme očakávať, že aspoň polovica klauzúl bude splnená. Ak vieme, že každá klauzula má aspoň tri premenné, potom dostaneme $E[Z_i] \geq 7/8$, pre každé $i = 1, \ldots, m$, takže môžeme očakávať, že aspoň $7/8$ všetkých klauzúl bude splnených.

Skip List

Je randomizovaná údajová štruktúra, ktorou sa dá realizovať slovník – t.j. operácie `Nájdi`, `Pridaj` a `Zmaz`, pričom každá z nich má očakávanú zložitosť lineárne úmernú $\log_2 n$, kde n je počet prvkov v slovníku. Skip list kombinuje utriedený spájaný zoznam a myšlienku efektívneho binárneho vyhľadávania, pri ktorom sa v každom kroku eliminuje polovica prvkov medzi ktorými hľadáme hľadaný prvok. V zozname dlho trvá vyhľadanie prvku, ale umožňuje veľmi rýchlo vložiť alebo zmazať prvok z konkrétnej pozíciu. Skip list umožňuje navyše aj rýchle vyhľadanie v zozname.

S trochou predstavivosti sa dá povedať, že skip list spája jednoduchosť zoznamu a efektívnosť binárneho vyváženého stromu.

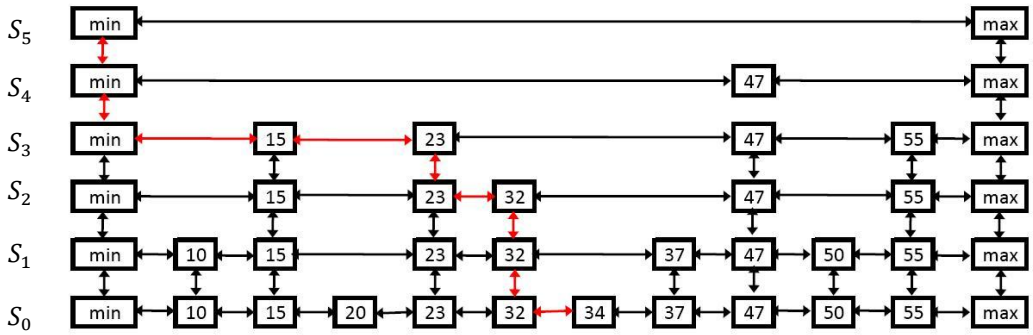
Táto údajová štruktúra využíva pri realizácií operácií náhodu, a tak sa „bráni“ proti „nevhodným“ vstupom, pre ktoré by tieto operácie trvali „pridlho“. Tradičné údajové štruktúry s rovnakou zložitosťou uvedených slovníkových operácií sú vyvážené binárne stromy, ktorých implementácia je neporovnateľne zložitejšia a zložitosť jednotlivých operácií sa počíta ako priemer ich trvania cez všetky možné vstupné údaje dĺžky n. Pri skip listoch odhadnuté zložitosti jednotlivých operácií vôbec nezáležia od vstupných údajov (napr. ako sú usporiadané).

$-\infty$ a ∞ sú zarážky

Skip list S, v ktorom sú uložené všetky prvky z nejakej množiny P je séria postupností $S_0, S_1, \cdots, S_{h-1}$, kde postupnosť S_i je utriedená podmnožina prvkov P a dva špeciálne prvky $-\infty$ a ∞, kde $-\infty$ je menší od všetkých prvkov a ∞ je väčší od všetkých prvkov, ktoré môžu byť vložené do P. Ďalej musí séria S spĺňať

- S_0 obsahuje všetky prvky P (a navyše prvky $-\infty$ a ∞)
- S_i obsahuje náhodne vybrané prvky z S_{i-1} ($S_i \subseteq S_{i-1}$), pre $i = 1, \ldots, h-1$.
- S_h obsahuje len prvky $-\infty$ a ∞.

Príklad skip listu je na *Obr. 6 Príklad skip listu.*. Postupnosť S_0 sa zvyčajne zobrazuje na najspodnejšej úrovni a postupnosti $S_1, \cdots, S_{h-1}$ na úrovniach na ňou. h sa nazýva výška skip listu S.



Obr. 6 Príklad skip listu. Červenou je znázornená cesta prvkami skip listu pri hľadaní prvku 34, t.j. vykonávaní operácie `Najdi`*. (min predstavuje* $-\infty$ *a max* ∞*)*

Predpokladáme, že minca je pravá a je rovnaká pravdepodobnosť, rovnajúca sa $1/2$, že padne rub alebo líc.

Vidíme, že postupnosť S_{i+1} má najviac toľko prvkov ako S_i. V skutočnosti prvky do S_{i+1} vyberieme tak, že pre každý prvok z S_i si hodíme mincou a keď padne líc, pridáme ho do S_{i+1} a keď padne rub tak ho nepridáme. Takže očakávame, že keď má S_0 n prvkov, bude mať S_1 $n/2$ prvkov, S_2 $n/4$ prvkov, a vo všeobecnosti S_i $n/2^i$ prvkov. Takže očakávaná výška skip listu bude $log_2 n$.

Skip list si môžeme predstaviť (Obr. 6) aj ako vertikálne úrovne prvkov a horizontálne veže obsahujúce rovnaké prvky v jednotlivých úrovniach. Jednotlivé prvky v skip liste môžeme prechádzať s využitím nasledujúcich operácií:

`za(p)`: pozícia prvku nasledujúceho za prvkom p na tej istej úrovni.

`pred(p)`: pozícia prvku predchádzajúceho pred prvkom p na tej istej úrovni.

`pod(p)`: pozícia prvku pod prvkom p na o jedno nižšej úrovni v tej istej veži.

`nad(p)`: pozícia prvku nad prvkom p na o jedno vyššej úrovni v tej istej veži.

Úloha 6.

Navrhnite programátorské riešenie na realizáciu týchto operácií prechádzania skip listom, ktoré vyžaduje len konštantný počet operácií na každú operáciu. Vyberte si jednu z operácií a naprogramujte ju.

Vyhľadávanie

Operácia vyhľadávania je základnou operáciou v skip liste. Využívajú ju všetky ostatné operácie. Pre danú hodnotu h, zistí, či sa v skip liste nachádza prvok s hodnotou h. Presnejšie povedané, zistí pozíciu najväčšieho prvku v úrovni S_0, ktorý je menší alebo rovný h.

Hľadanie začneme „vľavo hore“ (na najvyššej úrovni najľavejšej veže) na hodnote $-\infty$. Postupne sa budeme posúvať doprava v tej úrovni, kde sa nachádzame až

po najväčší prvok, ktorý je menší alebo rovný ako hľadaná hodnota h. Tu ak sa dá zídeme na nižšiu úroveň pokračujeme takým istým spôsobom na tejto úrovni. Ak nižšia úroveň už neexistuje (sme v úrovni S_0), našli sme hľadanú pozíciu. Podrobnejšie je procedúra `Hladaj` zapísaná nižšie:

```
// THodnota je typ hodnoty
// TPozicia je ukazovateľ pozicie
function Hladaj(h : THodonota) : TPozicia;
begin
  Result := PoziciaNajvyssejNajlavejsej;
  while pod(Result) <> nil do begin
    Result := pod(Result);   // ideme o úroveň nižšie
    while Hodnota(po(Result)) <= h do
      Result := po(Result)   // ideme doprava
  end;
end
```

Vidíme, že hľadanie je ozaj až neuveriteľne jednoduché.

Pridávanie

Pridávanie prvku s hodnotou h do skip listu môžeme rozdeliť na dve etapy: v prvej sa nájde pomocou funkcie `Hladaj(h)` miesto, kde má daný prvok byť v najnižšej úrovni skip listu S_0 a v druhej etape sa vytvorí z pridávaného prvku v skip liste veža náhodnej výšky (počtu úrovní).

Po nájdení pozície prvku p, za ktorý máme pridať prvok s hodnotou h, prichádza na rad náhoda. Prvok pridáme do úrovne S_0 a hodíme si mincou. Keď padne rub pridávanie je ukončené. Keď padne líc pridáme prvok aj do úrovne S_1 na miesto presne nad pridaný prvok v S_0 a pokračujem v hádzaní mincou a eventuálnym pridávaním do vyšších a vyšších úrovní až pokým nám v niektorej nepadne rub.

Takže pravdepodobnosť, že ukončíme zvyšovanie veže alebo ju zvýšime o ďalšie poschodie je v každom kroku rovná $1/2$.

Na pridávanie do vyššej úrovne použijeme funkciu `PridajNadAZa(p, q, h)`, ktorá pridá prvok s hodnotou h nad prvok na pozícii q a za prvok na pozícii p. Táto funkcia nastaví v pridanom prvku všetky ukazovatele (šípky na Obr. 6), tak aby po pridaní pracovali správne funkcie `pred`, `po`, `nad` a `pod`. Funkcia `PridajNadAZa` vráti pozíciu pridaného prvku.

```
procedure Pridaj(h : THodonota);
var
  p, q : TPozicia;
begin
  p := Najdi(h);
  q := PridajNadAZa(nil, p, h);  // pridáme do S0
  while random < 1/2 do begin   // kým padá líc
    while nad(p) = nil do       // hľadáme kde sa dá ísť hore
      p := pred(p);
    p := nad(p);                // ideme do vyššej úrovne
    q := PridajNadAZa(q, p, h)
  end
end
```

Mazanie

Operácia `Zmaz(h)` sa skladá tiež z dvoch etáp: v prvej nájdeme pozíciu prvku s hodnotou h. Ak taký prvok v skip liste neexistuje, situáciu spracujeme (nerobíme nič, vyhlásime chybu,...). Keď v skip liste je taký prvok, musíme ho odstrániť - zo všetkých úrovní, t.j. celú vežu. Musíme pri tom aktualizovať na jednotlivých úrovniach ukazovatele pre a za zmazaným prvkom.

Koľko to stojí? alebo Jednoduchá analýza

Aká je očakávaná výška najvyššej veže, alebo inak povedané, koľko úrovní má skip list. Pravdepodobnosť, že nejaký prvok sa nachádza v úrovni u, je rovnaká ako, že

za sebou padne na minci u-krát líc, t.j. $1/2^u$. Takže pravdepodobnosť P_u, že úroveň u má aspoň jeden prvok je najviac

n je počet prvkov v slovníku.

$$P_u \leq \frac{n}{2^u}.$$

Keď za u dosadíme napr. $3\log_2 n$, dostaneme, že $P_{3\log_2 n} \leq \frac{n}{2^{3\log_2 n}} = \frac{n}{n^3} = \frac{1}{n^2}$. Teda s rastúcim n, klesá pravdepodobnosť, že výška skip listu bude viac než konštantný násobok $\log_2 n$.

Ale pri hľadaní a pridávaní nejdeme len nadol cez všetky úrovne, ale aj dopredu (alebo dozadu) v rámci jednotlivých úrovní. Vieme odhadnúť počet prejdených prvkov aj v rámci úrovní? Kľúčové je nasledujúce pozorovanie: keď nejaký prvok navštívime pri pohybe vpred (vzad) v rámci úrovne i, nemôžeme ho navštíviť pri tejto činnosti aj úrovni $i+1$. Takže pravdepodobnosť, že nejaký prvok prejdeme v úrovni i je $1/2$ a očakávaný počet prvkov, ktoré prejdeme v úrovni i sa rovná počtu hodení mincou aby sme dostali rub (teda, že klesneme na nižšiu úroveň), a ten je 2. Teda spolu dole a dopredu (dozadu) prejdeme konštantný násobok $\log_2 n$.

Prečo ho nemôžeme navštíviť aj v úrovni $i+1$?

Koľko pamäti zaberie skip list, keď má slovník n prvkov? Ako sme už spomínali, očakávaný počet prvkov na úrovni u je $n/2^u$,takže očakávaný celkový počet prvkov v skip liste je

$$\sum_{u=0}^{h} \frac{n}{2^u} = n \sum_{u=0}^{h} \frac{1}{2^u} \leq 2n.$$

To znamená, že slovník realizovaný skip listom v pamäti zaberie len lineárny násobok počtu prvkov slovníka.

Čo sme sa naučili

Zoznámili sme sa s pravdepodobnostnými algoritmami typu Monte Carlo a Las Vegas. Ukázali sme si, ako nám náhoda môže pomáhať „proti“ zlým prípadom vstupov a dokážeme pomocou nej vytvoriť jednoduché a efektívne algoritmy a štruktúry údajov.

Literatúra a použité zdroje

[1] Motwani, R., Raghavan, P. (1995) Randomized Algorithms. Cambridge University Press. ISBN 0-521-47465-5

[2] Hromkovič, J. (2005) Design and Analysis of Randomized Algorithms. Springer, ISBN 3-540-23949-9

[3] Hromkovič, J. (2009) Algorithimc Adventures, from Knowledge to Magic, Springer, ISBN 978-3-540-85985-7

[4] Brassard, G., Bratley, P. (1996) Fundamentals of Algorithms, Prentice Hall. ISBN 0-13-335068-1

[5] Tijms, H. (2007) Understanding Probability, Chance Rules in Everyday Life. Cambridge University Press, ISBN 978-0-521-70172-3

[6] Giblin, P.: Primes and Programming, Introduction to Number theory with Computing, Cambridge University Press, 1993;

[7] Goodrich, M., Tamassia, R. Using Randomization in the Teaching of Data Structures and Algorithms, ACM SIGCSE ’99, New Orleans

Kapitola 3: Výpočty na DNA a DNA výpočty

Ciele tematického celku

V úvodnej časti uvedieme základné pojmy o biologických nosičoch informácie, prejdeme od biologickej informácie k sekvenciám (reťazcom) vytvoreným nad abecedou charakterizujúcou DNA a RNA molekuly. Sme už presvedčení o tom, že DNA sekvencie sú nositeľmi biologických informácií a že všetky procesy prebiehajúce v živých organizmoch sú riadené inštrukciami (informáciami) uloženými v DNA sekvenciách. Je pravda, že málo vieme o tom, akým spôsobom toto riadenie prebieha, ale napriek tomu boli uskutočnené prvé výpočty pomocou DNA sekvencií, ktoré vieme biochemickými operáciami ovládať. Pretože DNA je možné reprezentovať pomocou sekvencií (reťazcov, postupnosťami symbolov), sú študované aj niektoré vlastnosti týchto sekvencií a v informatike v tomto smere nájdeme zaujímavé výsledky. V úvodnej časti uvedieme tieto niekoľko zaujímavých problémov práve z tejto oblasti spracovania sekvencií, ktoré súvisia aj s DNA sekvenciami.

V ďalšej časti vysvetlíme ideu nových výpočtov vykonávaných na molekulách, bude vysvetlené, prečo je to možné robiť. Na tomto základe bude uvedený zoznam realizovateľných biochemických operácií, ktoré postačujú na to, aby sa pomocou nich dal urobiť ľubovoľný výpočet. Popíšeme známy Adlemanov experiment, pomocou ktorého vyriešil konkrétny prípad optimalizačnej úlohy obchodného cestujúceho. Budeme sa tiež zaoberať silnými a slabými stránkami technológie DNA počítačov.

Forma výučby

Klasická prednáška a cvičenie. Je tu možnosť využiť znalosti z programovania.

Prehľad základných pojmov

Deoxyribonukleová kyselina (DNA) je makromolekula, pozostávajúca z dvoch okolo seba obtočených reťazcov (známa dvojitá závitnica). Každý reťazec sa skladá zo štyroch typov pomerne jednoduchých organických molekúl, pospájaných kovalentnými väzbami. Týmito molekulami sú *deoxyadenosinfosfát* (A), *deoxycytidinfosfát (C), deoxyguanosinfosfát (G) a thymidinfosfát (T)*. Spoločne ich nazývame ***nukleotidy*.** Poradie nukleotidov, ktorých je v každom reťazci DNA mnoho tisíc až miliónov, tvorí dedičnú informáciu. Môžeme to dobre prirovnať k informácii zapísanej v poradí písmen textu.

Ribonukleová kyselina (**RNA** alebo **RNK**) je nukleová kyselina tvorená jedným vláknom kovalentne naviazaných *ribonukleotidov*. Je biochemicky rozlíšiteľná od DNA vďaka prítomnosti dodatočnej hydroxylovej skupiny pripojenej ku každej pentózovej molekule reťazca a prítomnosti *uracilu* namiesto *tymínu*. Jednou z hlavných funkcií RNA je okopírovať genetickú informáciu z DNA a fyzicky ju preniesť na miesto, kde dôjde k jeho preloženiu na výsledný proteín. Priamo túto funkciu plní iba jedna trieda RNA, mediátorová RNA (mRNA).

DNA vytvára dlhú dvojitú špirálu, zloženú z dvoch komplementárnych vlákien. RNA vytvára relatívne kratšie jednovláknové reťazce. V niektorých prípadoch má komplementárnosť dvoch molekúl RNA aj fyziologickú funkciu (RNAi).

Formálnejšie vyjadrenie: V abstraktnom zmysle slova možno molekulu DNA a podobne aj molekulu RNA stotožniť so symbolickým reťazcom $\boldsymbol{x} = x_1, x_2, \ldots, x_n$ v abecede $\Sigma = \{\mathrm{A}, \mathrm{G}, \mathrm{T}, \mathrm{C}\}$.

Na obrázku 3.1.a. je uvedený príklad štruktúry RNA. DNA a ani RNA nie sú lineárne reťazce a to komplikuje ich analýzu. Avšak prvotným problémom je nájsť čo najlepšie algoritmy pre lineárne reťazce, a až potom analyzovať všetky odbočky a slučky. V nasledujúcom odseku ukážeme, ako je možné zapísať reťazec a pomocou sekundárnej štruktúry zaevidovať jeho členenie.

Genetika a genomika

Genetika je veda, ktorá sa zaoberá štúdiom jednotlivých génov a ich významom pre dedičnosť. Gény nesú základné inštrukcie pre tvorbu proteínov, ktoré vykonávajú najrôznejšie funkcie v rámci bunky aj celého organizmu. Gény určujú napríklad farbu a typ vlasov, farbu očí, ale tiež ovplyvňujú ľudské zdravie a naznačujú aké choroby sa v priebehu života vyvinú. Medzi choroby spôsobené poškodením jednotlivých génov patrí napr.: cystická fibróza alebo fenylketonúria.

Podľa [4]

Genomika je relatívne nový, biologický odbor, ktorého úlohou je štúdium genómu (genetická informácia zakódovaná v DNA) jednotlivých organizmov. Na rozdiel od genetických odborov má za úlohu porozumieť vlastnostiam študovaného genómu v celku. Genomika sa zaoberá predovšetkým štúdiom komplexných ochorení, u ktorých sa predpokladá účasť väčšieho počtu génov a faktorov životného prostredia, napr.: astma, kardiovaskulárne či nádorové choroby. Medzi hlavné nástroje používané genomikou patrí bioinformatika, genetická analýza, sledovanie génovej expresie a štúdium funkcie génov.

Podľa [4]

Rozprávanie Američana Jamesa D. Watsona o jeho ceste k objavu štruktúry DNA, ktorý jemu a jeho spolupracovníkom priniesol Nobelovu cenu, charakterizuje britský spisovateľ, známy svojou snahou preklenúť priepasť medzi prírodnými a humanitnými vedami, C. P. Snow slovami: „Knižka ako doposiaľ žiadna iná ukazuje, ako sa robí tvorivá veda. Pre laických čitateľov otvára nový svet." V dobe publikovania tejto knihy mal autor necelých 25 rokov. Objasnenie štruktúry molekuly DNA súčasne ukázalo, ako sa gény množia, ako fungujú a z čoho sú zložené. Mnoho vedcov považuje tento objav za najzávažnejší vo vede o živote od čias C. Darwina a G. Mendela.

Ku knihe J. D. Watson: Tajemství DNA. Academia, Praha, 1995

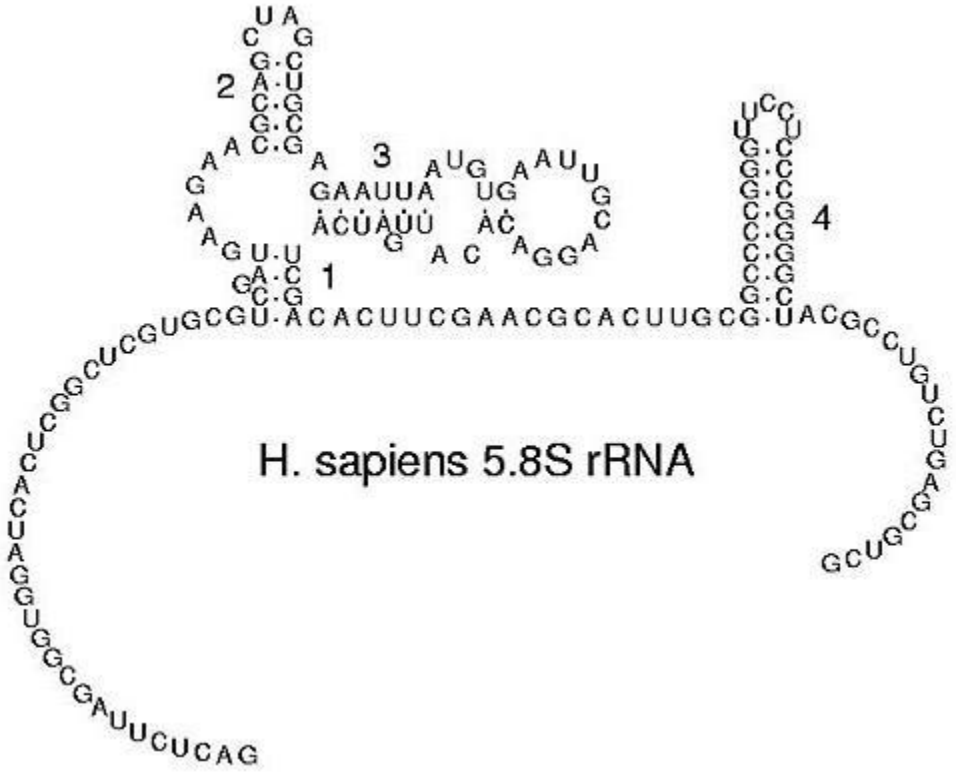
Projekt ľudského genómu: Za míľnik v dejinách vedy sa považuje správa zverejnená 26. 6. 2000, ktorú prinieslo Medzinárodné konzorcium Projektu ľudského genómu: prvý pracovný koncept ľudského genómu je na svete.

SME, 27. 6. 2000

V roku 2002 bola vytvorená finálna verzia ľudského genómu.

Kvôli predstave: dedičná informácia nejakej „jednoduchej" baktérie je uložená v jednej jedinej molekule DNA zloženej z niekoľko miliónov nukleotidov.

Postupnosť a kombináciu týchto „písmen" vie organizmus interpretovať ako návod pre svoj život. Informácia zapísaná v poradí nukleotidov je rozdelená na úseky, podobne ako je písaný text rozdelený do viet. **Gén** je úsek DNA kódujúci vznik jedného typu proteínu. A proteíny sú tie molekuly, ktoré riadia (katalyzujú) chemické reakcie v organizme. Napríklad DNA jednoduchej baktérie obsahuje niekoľko tisíc génov; k životu baktérie teda postačuje niekoľko tisíc rôznych proteínov. Celá dedičná informácia bunky, teda súbor všetkých génov ale i DNA, ktorá nie je súčasťou génov, sa nazýva **genóm**.



Obr. 3.1.a. Príklad štruktúry RNA.

Na rozdiel od baktérií majú bunky vyšších organizmov zapísanú svoju dedičnú informáciu vo viacerých molekulách DNA uložených v jadre buniek v útvaroch nazvaných **chromozómy**.

Jedným z najväčších pokrokov biológie je vypracovanie metód na stanovenie sledu (postupnosti) nukleotidov v molekulách DNA. Táto „postupnosť" nukleotidov v DNA nám doslova umožňuje „prečítať si" dedičnú informáciu organizmov.

Dnes už poznáme úplnú dedičnú informáciu mnohých organizmov [4]. Väčšina týchto organizmov sú baktérie. To nie je nijak nepochopiteľné, pretože bakteriálne genómy sú pomerne malé, zložené z miliónov nukleotidov, a tiež preto, že sú často príčinou chorôb (sú patogénne).

Baktérie sú jednobunkové organizmy. Vyššie organizmy sa skladajú z mnohých buniek rôzneho vzhľadu a rôznej funkcie. Bunky však voľným okom nevidíme – na to potrebujeme dobrý mikroskop. Bunky ľudského tela tvoria rôzne tkanivá, napríklad pečeň, koža, svaly. Prakticky všetky bunky, bez ohľadu na to, z akej časti organizmu pochádzajú, nesú úplnú dedičnú informáciu pre vývoj celého organizmu: všetky majú rovnakú DNA. Kvôli lepšej predstave o zložitosti tohto systému si uveďme, že napríklad molekuly DNA z jednej ľudskej bunky, pokiaľ by sa nám ich podarilo

nadviazať a natiahnuť, by merali takmer dva metre. Keby sme ich zväčšili na hrúbku nite, bola by dlhá asi 300 kilometrov.

Na stránku formátu A4 sa zmestí okolo troch tisíc husto skomprimovaných písmen. Na zápis ľudskej genetickej informácie by sme teda potrebovali okolo dvoch tisíc kníh, z ktorých by každá obsahovala úctyhodných 500 husto popísaných stránok. To je už slušná knižnica. A to by bola len základná informácia, t. j. súvislý, nám nezrozumiteľný text, zložený zo štyroch rôzne za sebou zaradených písmen reprezentujúcich adenin , guanín, cytozín a tymin.

Ľudský genóm sa skladá z molekúl DNA o celkovom obsahu asi tri miliardy nukleotidov. Tieto molekuly DNA sú uložené v jadre ľudskej bunky v 23 chromozómoch. Vlastne v pároch chromozómov, pretože každá telová bunka obsahuje jeden chromozóm pôvodne pochádzajúci od otca a k nemu do páru chromozóm od matky.

Koncom roku 1999 bola publikovaná nukleotidová sekvencia ľudského chromozómu 22. Unikátna časť tohto chromozómu (ktorý je druhým najmenším ľudským chromozómom) sa skladá z 34,4 miliónov nukleotidov. Tieto tvoria 545 génov. 97% DNA nenesie žiadnu zmysluplnú správu! A to platí všeobecne: len približne 3% ľudskej DNA a DNA mnohých ďalších vyšších organizmov kóduje vznik funkčných molekúl, hlavne proteínov.

Genomika je odbor, ktorého cieľom je určiť úplnú dedičnú informáciu organizmov a interpretovať ju v termínoch životných pochodov. Niekedy sa genomika rozdeľuje na tzv. **štruktúrnu genomiku**, spočívajúcu v stanovení sledu nukleotidov genómu organizmu, na **bioinformatiku**, ktorá počítačovými metódami a prácou v databázach interpretuje prečítanú dedičnú informáciu, a na **funkčnú genomiku**, kde sa pomocou experimentu, napríklad vyradením nejakého génu z činnosti, snažíme priradiť funkciu neznámym génom, prípadne funkciu génov študovať.

Z článku: Bioinformatika a genomika - Móda anebo nový obor v biologii

Jaroslav Kypr, Vesmír 76, 195, 1997/4

Medzi hlavné problémy bioinformatiky a výpočtovej biológie patrí rozvoj a implementácia nástrojov, ktoré umožnia efektívny prístup a využitie rôznych typov informácií. Ďalším riešeným problémom je vývoj nových algoritmov, matematických formúl a štatistík, vďaka ktorým by sa dala ohodnotiť príbuznosť medzi jednotlivými členmi rozsiahlych množín dát. Tu patria metódy na hľadanie génov v sekvenciách, predpovedanie proteínovej štruktúry a funkcie a zoskupovanie proteínových sekvencií do rodín obsahujúcich príbuzné sekvencie.

Z konkrétnych problémov, ktoré rieši bioinformatika, môžeme spomenúť tieto [5]:

Proteíny (bielkoviny): Katalyzujú väčšinu biochemických reakcií v bunke (enzýmy), prenášajú signály v rámci bunky aj medzi bunkami, Sú dôležité pre stavbu bunky a pohyb. Sú to reťazce aminokyselín (20 rôznych).

1. **Zarovnávanie sekvencií:** Ide o spôsob usporiadania dvoch, alebo viacerých reťazcov DNA, RNA alebo proteínov na identifikáciu podobných častí, ktoré môžu byť dôsledkom funkčnej, štrukturálnej, alebo evolučnej príbuznosti medzi týmito reťazcami. Tento problém a niektoré metódy jeho riešenia je popísaný v nasledujúcej podkapitole.
2. **Vyhľadávanie génov:** Týka sa predovšetkým vyhľadávania úsekov sekvencií genotypovej DNA. Medzi tieto úseky patria gény kódujúce rôzne proteíny a RNA gény.
3. **Spájanie genómov**: Ide o spájanie množstva krátkych DNA sekvencií na vytvorenie reprezentácie pôvodného chromozómu, z ktorého DNA vznikla. Tieto krátke DNA sekvencie vznikajú na základe procesu „Shotgun sequencing“, ktorý rozdelí DNA na milióny krátkych častí. Tie sú potom načítané pomocou zarovnávacieho stroja. Algoritmus na spájanie genómov tieto sekvencie zarovná. Na miestach, kde sa dve sekvencie prelínajú, sa môžu spojiť dokopy. Tento problém je zložitý, pretože genómy obsahujú podreťazce, ktoré sa opakujú.
4. **Zarovnávanie proteínovej štruktúry:** Štrukturálne zarovnávanie je forma zarovnávania sekvencií založená na porovnávaní tvaru a trojdimenzionálnej štruktúry.
5. **Predpovedanie proteínovej štruktúry:** Ide o predpovedanie trojdimenzionálnej štruktúry proteínov na základe sekvencie aminokyselín. Táto metóda je dôležitá napríklad v medicíne pri navrhovaní liekov, alebo v biotechnológii pri navrhovaní nových enzýmov.

Z knihy J. D. Watson: Tajemství DNA. Academia, Praha, 1995

Pred mojím príchodom do Cambridge premýšľal Francis (Crick) o kyseline deoxyribonukleovej (DNA) a jej úlohe v dedičnosti len príležitostne. Nie preto, že by ju považoval za nezaujímavú. Práve naopak. Hlavnou príčinou, prečo opustil fyziku a začal sa zaujímať o biológiu, bolo, že si v roku 1946 prečítal knihu *What is Life?* slávneho teoretického fyzika Erwina Schroedingera. Táto kniha ponúkala úvahy o presvedčení, že gény sú kľúčovými zložkami živých buniek, a aby sme pochopili, čo je život, musíme vedieť, ako gény fungujú. V roku 1944, keď Schroedinger svoju knihu písal, sa vo všeobecnosti usudzovalo, že gény sú špeciálne bielkovinové molekuly. Ale takmer v tom istom čase pokusy bakteriológa O. T. Averyho v Rockefellerovom ústave v New Yorku ukázali, že dedičné znaky môžu byť prenášané z jednej bakteriálnej bunky na druhu len prostredníctvom samotných molekúl DNA.

6. **Určovanie funkcie proteínu** – na riešenie tohto problému zatiaľ neexistujú vhodné dáta. Momentálnym cieľom je zostrojiť databázu obsahujúcu reakcie proteínov. Najprv však treba vytvoriť vhodný spôsob ich reprezentácie.
7. **Modelovanie evolučnej histórie** – ide o zoskupovanie sekvencií na základe ich podobnosti do stromu. Takýto fylogenetický strom potom reprezentuje zmeny v sekvenciách počas evolúcie.

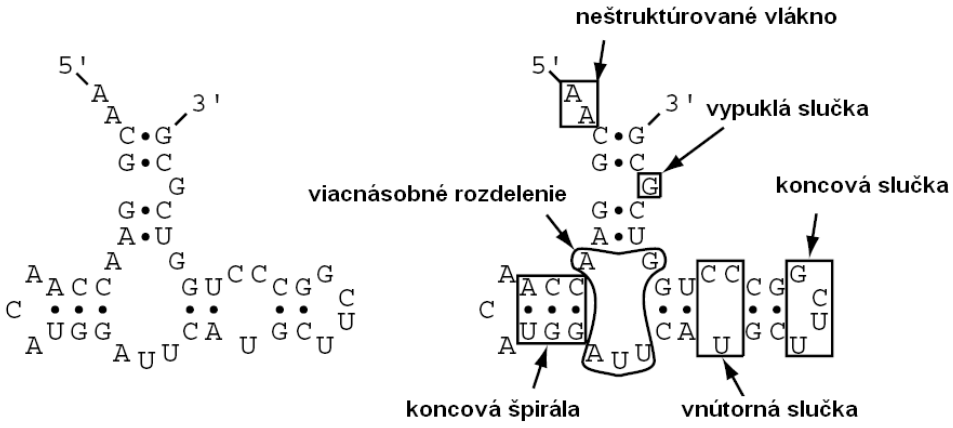
Reprezentácia sekundárnej štruktúry RNA

Jednoduchý spôsob reprezentácie sekundárnej štruktúry je pomocou zátvoriek. Napríklad (((((. . . .))))) reprezentuje špirálu zloženú z piatich párov nukleotidov a slučku zo štyroch nukleotidov. V takomto zápise je však ťažké interpretovať zložitejšie štruktúry.

Jednoduchšie je to pomocou WUSS notácie (Washington University Secondary Structure notation). Tá používa nasledujúce znaky:

- „< >“ na reprezentáciu jednoduchých koncových špirál,
- „( )“ na reprezentáciu vnútorných špirál ukončujúcich rozdelenia obsahujúce len koncové špirály,
- „[ ]“ na reprezentáciu vnútorných špirál ukončujúcich rozdelenia obsahujúce aspoň jednu slučku označenú „( )“,
- „{ }“ na reprezentáciu všetkých vnútorných špirál ukončujúcich hlbšie rozdelenia,
- „ _ “ na reprezentáciu koncových slučiek.
- „ - “ na reprezentáciu vypuklých a vnútorných slučiek,
- „ , “ na reprezentáciu zvyškov v rozdeľovacích slučkách,
- „ : “ na reprezentáciu samostatných vlákien mimo štruktúry,
- „ . “ na reprezentáciu vložení nepatriacich do známej štruktúry,
- na reprezentáciu jednoduchých pseudouzlov sa používa označenie „Aa“, napríklad <<<_AA___>>>aa; ďalšie pseudouzly môžu byť označené „Bb“, „Cc“, atď.

Na obrázku 3.1.b je uvedený príklad použitia WUSS notácie pre vytvorenie sekundárnej štruktúry reťazca RNA




```
::(((((,<<<____>>>,,,<<-<<_____>>--->>,)))-)))
AACGGAACCAACAUGGAUUCAUGCUUCGGCCCUGGUCGCG
```

Obrázok 3.1.b. Príklad WUSS notácie sekundárnej štruktúry

Vyhľadávanie génov v sekvenciách

Úloha nájsť v sekvencii znakov všetky výskyty iných zadaných kratších sekvencií (slov, alebo ich častí) patrí v praxi medzi najrozšírenejšie.

Kde všade sa stretneme s vyhľadávaním:

- textové editory,
- utility typu grep v Unixe,
- rešeršné systémy,
- ale i taký internet (google, zoznam, ...).

Pričom pokiaľ to má byť vyhľadávanie v nie veľmi dlhých sekvenciách, tak nie je to problém, stačí nám jednoduchý „naivný algoritmus“, ale v prípade veľkých rešeršných systémov by to bolo veľmi pomalé.

Formulácia problému - všeobecne :

Nech je dané

- **abeceda S, konečná množina prvkov,**
- **v abecede vytvárame konečné postupnosti jej prvkov - slová,**
- **slovo z =z[1]z[2]...z[n], patriace do S* (prehľadávaná sekvencia),**
- **a množina *K* = {y[1], y[2],...,y[k]} neprázdnych slov v abecede S, y[p]=y[p,1]...y[p, l(p)], l(p)<=n, pre p =1..k (hľadané vzorky sekvencií).**
- **označme ešte l= l(1)+..+ l(k) súčet dĺžok vzoriek.**

Máme za úlohu nájsť všetky výskyty vzoriek z *K* v dlhej sekvencii z, t. j. všetky dvojice <i, y[p]> také, že y[p] je príponou v slove z[1]z[2]...z[i].

Spôsob hlásenia nás nebude zaujímať - budeme predpokladať, že hlásenie zariadi bližšie nešpecifikovaná procedúra Report(i, y[p]).

Naivné riešenie napadne každého; tu si ho napíšeme preto, aby sme sa pozreli na jeho časovú zložitosť:

Idea: Pre každé slovo y K skontrolujem postupne, kde sa nachádza v dlhom slove z prechodom cez z po jednom znaku.

```
    begin
{1}   for p := 1 to k do begin {cyklus pre slová v K}
{2}     for i := 1 to n-l(p)+1 do begin
{3}       j := 0;
{4}       while (j < l(p)) and (z[i+j]=y[p,1+j]) do
            j := j+1;
{5}       if j = l(p) then Report(i,y[p]);
{6}     end
{7}   end
    end.
```

Označme T(n,l) počet krokov tohto naivného algoritmu pri rozmeroch vstupov n, l.

Celý cyklus **for** na riadkoch {2}..{6} bude pre každé p = 1, 2, ...k vykonaný (n-l(p)+1)-krát a každé vykonanie vyžaduje 2l(p)+1 podmienok (riadky {4} a {5}), l(p) priraďovacích príkazov (riadok 4) a 1 príkaz v riadku {3}.

Zložitosť je

$$T(n,l) \le \sum_{p=1}^{k} \left((n - l(y_p) + 1)(3l(y_p) + 1) \right) = 3l.n - 3\sum_{p=1}^{k} l(y_p)^2 + 3l + kn + k \le 3(l.n - l^2) = O(l.n - l^2)$$

pričom n - dĺžka textu v ktorom hľadáme, l - dĺžka všetkých slov, ktoré hľadáme. Ale existujú aj šikovnejšie algoritmy, ktoré umožňujú rýchlejší postup v prehľadávanej sekvencii, poskytujú možnosť preskočiť niektoré prvky abecedy. Toto je užitočné pri veľmi dlhých prehľadávaných sekvenciách.

Maurice Hugh Frederick Wilkins (* 15. december 1916 Pongaroa, Nord-Wairarapa – † 5. október 2004, Londýn) bol fyzik, narodený na Novom Zélande. Laureát Nobelovej ceny.

Fyziku študoval na St John's Colege v Cambridge, v roku 1940 získal na univerzite v Birminghame doktorát. Počas druhej svetovej vojny spolupracoval na projekte Manhattan na univerzite Berkeley. Podieľal sa na výskume fosforescencie, radaru, izotopovej separácie a difrakcie röntgenového žiarenia. Jeho najznámejšou prácou na King's College v Londýne bol objav štruktúry DNA, molekuly, ktorá nesie genetickú informáciu, za ktorý dostal on, Francis Crick a James Watson (obidvaja z Cavendish Laboratory v Cambridgi) v roku 1962 Nobelovu cenu za fyziológiu alebo medicínu.

Podľa: „http://sk.wikipedia.org/wiki/Maurice_Wilkins“

Z knihy J. D. Watson: Tajemství DNA. Academia, Praha, 1995

Bol to Wilkins, kto vo mne vzbudil záujem o röntgenovú analýzu DNA. Stalo sa to v Neapoli pri malom vedeckom stretnutí, ktoré bolo venované štruktúram veľkých molekúl vyskytujúcich sa v živých bunkách. Vtedy na jar v roku 1951 som ešte o Francisovi Crickovi nevedel ani to, že existuje. V Európe som bol na postgraduálnom štipendiu, aby som sa naučil biochémiu DNA. Môj záujem o DNA sa prebudil už v dobe štúdia na univerzite a pramenil z túžby zistiť, čo je to gén.

Viacnásobné zarovnávanie reťazcov (sekvencií)

Aby sme si uvedomili, že práca s písmenkovými reťazcami nie je veľmi komplikovaná, ale pri dlhých reťazcoch je časovo náročná, v nasledujúcej časti sa budeme podrobnejšie venovať viacnásobnému zarovnaniu sekvencií, ktoré je v bioinformatike jednou z najčastejšie riešených úloh. Je prvou fázou riešenia mnohých zložitejších problémov, ako napríklad určovanie sekundárnej štruktúry RNA, konštrukcia fylogenetického stromu, či identifikácia častí DNA alebo proteínov, ktoré súvisia s určitou funkciou.

Zarovnania sa využívajú v úlohách nasledujúcich typov:

- **Orientácia v obrovských databázach.** Genbank má vyše 100 GB sekvencií.
- **Určovanie funkcie (napríklad proteínu).** Podobné sekvencie majú často podobné vlastnosti.
- **Štúdium evolúcie.** Hľadáme homológy, sekvencie, ktoré sa vyvinuli z toho istého spoločného predka. V ideálnom prípade medzery zodpovedajú vkladaniam a vymazaniam, zarovnané bázy zachovaným bázam a substitúciám.
- **Hľadanie génov a iných funkčných prvkov.** Menia sa pomalšie ako iné sekvencie.

Zarovnanie dvoch reťazcov spočíva v ich doplnení o medzery a zapísaní jednotlivých znakov pod seba. Pre nás je dôležité umiestniť medzery na správne miesta tak, aby boli podobné časti zarovnané pod sebou (Obrázok 3.1.c).

```
q a c _ d b d
q a v x _ b _
```

Obrázok 3.1.c Jedno z možných zarovnaní reťazcov qacdbd *a* qavxb

Aby sme mohli nájsť najvhodnejšie zarovnanie, potrebujeme ho najprv nejakým spôsobom definovať. Využíva sa na to hodnotiaca funkcia ***edit distance*** (vzdialenosť editovania), ktorá zarovnanie ohodnotí podľa počtu a typu zmien potrebných na to, aby sme prvý reťazec zmenili na druhý. Povolené zmeny sú *vloženie znaku do prvého reťazca*, *vymazanie znaku z prvého reťazca a nahradenie znaku v prvom reťazci znakom z druhého reťazca*. Každá z týchto zmien má svoje ohodnotenie (obvykle 1, ak s daným znakom nemusíme vykonať žiadnu operáciu, pretože sa zhoduje so znakom v druhom reťazci, hodnotíme to obvykle 0). Ohodnotenie zarovnania je rovné súčtu všetkých potrebných zmien vynásobených ich ohodnotením.

Za **optimálne zarovnanie** sa považuje zarovnanie s minimálnou vzdialenosťou editovania. Optimálne zarovnanie sa hľadá pomocou dynamického programovania.

V praxi sú riešené dva nasledujúce problémy zarovnania, ktoré vyjadríme formálnejšie:

Problém 1. Globálne zarovnanie (global alignment)
Vstup: 2 reťazce $P = p_1 p_2 \dots p_n$, $Q = q_1 q_2 \dots q_m$.
Výstup: Zarovnanie P a Q s najmenšou vzdialenosťou editovania.

Problém 2. Lokálne zarovnanie (local alignment)
Vstup: 2 reťazce $P = p_1 p_2 \dots p_n$, $Q = q_1 q_2 \dots q_m$.

Výstup: Zarovnania podreťazcov $p_i \dots p_j$ a $q_k \dots q_l$ s najmenšou vzdialenosťou editovania.

Pri riešení problému 1 pre 2 reťazce P a Q sa vypočíta tabuľka vzdialeností D, v ktorej $D(i,j)$ je najmenšie ohodnotenie operácií potrebných na zmenu prvých i znakov reťazca P na prvých j znakov Q. Na jej výpočet sa používajú tieto základné vzťahy:

$$D(0,0) = i$$
$$D(i,0) = i$$
$$D(0,j) = j$$
$$D(i,j) = \min\{D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1) + t(i,j)\}$$

pričom $t(i,j) = 0$, ak $p_i = q_j$, inak $t(i,j) = 1$.

Použitie tohto algoritmu je ilustrované v príklade 3.1. Podfarbené bunky v tabuľke ilustrujú cestu zarovnávania. Vzdialenosť editovania v príklade je rovná 4.

Pre riešenie problému 2 je použiteľný algoritmus pre riešenie problému 1, ak vytvoríme všetky rôzne súvislé podreťazce daných reťazcov. Takéto riešenie je funkčné, avšak jeho časová zložitosť bude veľká aj vzhľadom na to, že niektoré výpočty sa budú opakovať. Preto je potrebné hľadať efektívnejší algoritmus, ktorý výpočty neopakuje. Riešenie je možné nájsť v [13].

Príklad 3.1.: Pre reťazce uvedené na Obrázku 3.1.c dostávame nasledujúcu tabuľku vzdialeností editovania *Tabuľka 3.1.*:

D		**q**	**a**	**c**	**d**	**b**	**d**
	0	**1**	**2**	**3**	**4**	**5**	**6**
q	**1**	**0**	**1**	**2**	**3**	**4**	**5**
a	**2**	**1**	**0**	**1**	**2**	**3**	**4**
v	**3**	**2**	**2**	**1**	**2**	**3**	**4**
x	**4**	**3**	**2**	**2**	**2**	**3**	**4**
b	**5**	**4**	**4**	**3**	**3**	**2**	**4**

q a c - d b d

q a v x - b -

Tabuľka 3.1. Výpočet vzdialeností editovania a zarovnanie

Príklad 3.2.: Zarovnávanie troch reťazcov sleduje rovnaké prvky vo všetkých troch reťazcoch na rovnakých pozíciách. Pri nerovnosti je možné niekde vsunúť symbol „podškrtovník“.

A _ B C A	_ A B C A
A A _ C A	A A _ C A
A A B C _	A A B C _

Z knihy J. D. Watson: Tajemvství DNA. Academia, Praha, 1995

Naproti tomu Mauriceov roentgenovo difrakčný obraz DNA bol dôkazom mieriacim k jadru veci. Bol premietnutý na plátno na konci jeho prednášky. Mauriceova suchá angličtina nevyvolala žiadne nadšenie, keď konštatoval, že obraz ukazuje ďaleko viac detailov než predchádzajúce obrazy a je možné uvažovať o tom, že vznikol z kryštalickej substancie. A keď bude známa štruktúra DNA, skôr porozumieme, ako gény fungujú.

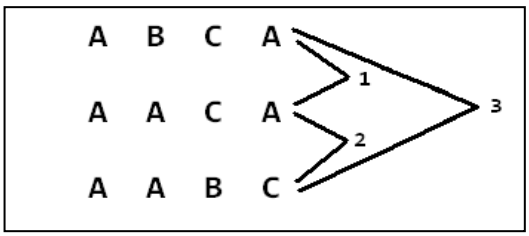
Náhle ma chémia začala vzrušovať. Pred Mauriceovou prednáškou som sa trápil možnosťou, že gén by mohol byť fantasticky nepravidelný. Teraz som videl, že gény môžu kryštalizovať . Musia mať teda pravidelnú štruktúru, ktorá môže byť rozlúštená priamo.

Viacnásobné zarovnanie (Obrázok 3.2) sa týka zarovnávania viacerých reťazcov. V bioinformatike predovšetkým DNA, alebo RNA reťazcov či proteínových reťazcov, ktoré nazývame sekvenciami.

```
G U U G G U G G U _ U A U U G U G U C G G
G U C G G U G G U _ G U U A G C G G U G G
U A C G G C G G U C A A U A G C G G C A G
U A C G G C G G U C C A U A G C G G C A G
U A C G G C G G C _ C A U A G C G G C A G
U A C G G C G G C _ C A C A G C G G C A G
U A C G G C G G U _ U A U A G C G G U G G
U A C G G C G G C _ C A U A G C G A C A G
U G C G G U G G U _ G A U A G U G G U G G
U G C G G U G G U _ G A U A G U G G U G G
_ G C C U U G G U _ C A C A G C C C C U G
```

Obrázok 3.2. Príklad viacnásobného zarovnania sekvencií

Pri zarovnávaní dvoch sekvencií sa ohodnotenie počíta jednoducho podľa počtu zhôd, zmien, vložení a vymazaní. Ohodnotenie viacnásobného zarovnania je však o niečo zložitejší problém. Jedným z možných riešení je hľadať minimálne **sum-of-pairs (SP) ohodnotenie**, ktoré je súčtom ohodnotení jednotlivých dvojíc zarovnaných sekvencií. Napríklad, zarovnanie 3 reťazcov uvedených na obrázku 3. po dvojiciach dáva veľkosť zarovnania 6 zmien. Na obrázku je tiež uvedené jedno možné zarovnanie.




```
A - B C A
A A - C A
A A B C -
```

Úloha 3.1.	Nájdite tabuľku vzdialeností pre reťazce ***AGCA a AAGC***.
Úloha 3.2.	Druhý najmenší ľudský chromozómom sa skladá z 34,4 miliónov nukleotidov. Chceli by sme zarovnať dva také chromozómy. Ako dlho by trval výpočet podľa tabuľky vzdialeností, keby výpočet hodnoty v jednom políčku trval 10^{-3} ms. Čo sa stane, ak tento čas ešte zmenšíme?
Úloha 3.3.	Máme 4 nukleotidy v úlohe 3.2. Ako by sa zmenila situácia z predchádzajúcej úlohy, keby sa počet nukleotidov znížil na 2?
***Úloha 3.4.**	Vytvorte program, ktorý by počítal tabuľku D (uvedená v príklade 3.1) pre dva reťazce dĺžky najviac 200 prvkov.
Úloha 3.5.	Nájdite veľkosť SP zarovnania pre reťazce **AGCA, AGAC** a **AAGC.** Navrhnite nejaké vhodné zarovnanie.

*Úloha 3.6.	Navrhnite algoritmus pre výpočet veľkosti SP zarovnania pre 3 reťazce. Určte jeho časovú zložitosť vzhľadom na dĺžku reťazcov.

Princíp práce DNA počítača

V podkapitole popíšeme technológiu biopočítačov, s ktorými už v praxi prebehli prvé experimenty, a to, kedy a či sa budú v budúcnosti využívať, závisí od viacerých faktorov, hlavne od biochemických metód používaných na výskum DNA sekvencií. Kapitola je spracovaná na základe knihy Juraja Hromkoviča: Algorithmic Adventures [2], kapitoly 8 - Computing with DNA Molecules, or Biological Computer Technology on the Horizon.

Fyzikálne hranice výkonu počítačov boli vypočítané už dávnejšie a už v roku 1959 sa známy fyzik R. Feynmann pýtal „Ako to pôjde ďalej?“. Ďalšia miniaturizácia prvkov počítačov bude možná už len tak, že sa prejde na výpočty pomocou chemických častíc a molekúl. Je pravda, že už chemici vedia kontrolovať chemické reakcie, vedia ich riadiť, ale akým spôsobom by sme to mohli robiť tak, aby výsledkom boli hodnoty, ktoré potrebujeme vypočítať?

Hlavná myšlienka DNA počítača je pomerne jednoduchá. Vieme, že typická molekula DNA sa vyskytuje v tvare dvojitej závitnice (vyjadriteľné tiež ako dvojitý reťazec písmen A, C, G, T), pričom väzby vznikajú medzi A a T a medzi G a C.

Princípy, na ktorých je založený postup DNA počítača:

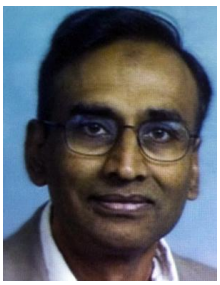
- Chemické väzby medzi A—T a G—C sú podstatne slabšie ako ostatné väzby v reťazci.
- Do určitej miery vieme regulovať molekulárnu stabilitu DNA.
- Naše údaje si môžeme predstaviť ako zakódované texty pomocou symbolov A, C, G a T.
- Ak sa nám podarí fyzicky vytvoriť presne takú DNA sekvenciu, tak by sme mali dané pripravené vstupy.
- S údajmi pripravenými týmto spôsobom je možné robiť v laboratóriách biochemické reakcie v skúmavkách, čím je možné DNA sekvencie meniť.
- Vzniknuté výsledné DNA sekvencie prečítame, odkódujeme a interpretujeme ako vypočítané výsledky

Dá sa dokázať, že takéto DNA počítače dokážu urobiť presne to isté, čo klasické elektronické počítače. To znamená, že naše chápanie algoritmov zostane v platnosti aj tu. Čo dokážeme vyriešiť algoritmicky (automaticky na počítači), to vieme urobiť aj pomocou DNA výpočtov, a platí to tiež opačne.

V čom je výhoda použitia DNA počítačov? Predstavme si kvapku vody, ktorá obsahuje 10^{19} molekúl. Keď máme v skúmavke 100 kvapiek, tak to je 10^{21} DNA sekvencií. Všetky reakcie (operácie) sa vykonávajú súčasne (paralelne) na všetkých molekulách. Toto je paralelizmus, ktorý súčasné počítače nevedia urobiť dostatočne rýchlo. Teda, keby sme vedeli pracovať s DNA sekvenciami, vieme robiť veľmi mohutný paralelizmus. Samozrejme, že nemáme doma také laboratóriá, aby sme toto všetko mohli realizovať, a preto je to zatiaľ ťažké. Je potrebné hľadať jednoduché prístupy, aby sa toto celé dalo urobiť.

"**Nobelova cena za chémiu v roku 2009** oceňuje výskum jedného z pre život najdôležitejších procesov: **ako ribozóm prekladá informáciu DNA do života,**" stojí v zdôvodnení udelenia ceny. Ramakrishan, Steitz a Yonathová ukázali, ako ribozóm vyzerá i ako funguje na atomárnej úrovni. Využili na to metódu röntgenovej štruktúrnej analýzy (röntgenová kryštalografia), pomocou ktorej zmapovali veľké množstvo (až státisíce) atómov, z ktorých sú ribozómy vytvorené.

Podľa:
veda.sme.sk



Venkatraman Ramakrishnan sa narodil v roku 1952 v indickom Chidambarame. Doktorát vo fyzike získal v roku 1976 na americkej univerzite v Ohio. Dnes je občanom Spojených štátov, no pracuje v Británii.

Thomas A. Steitz sa narodil v roku 1940 v americkom Milwaukee. Doktorát v molekulárnej biológií a biochémii získal v roku 1966 na Harvardskej univerzite. Dnes pôsobí ako profesor na univerzite v Yale.

Ada E. Yonathová sa narodila v roku 1939 v izraelskom Jeruzaleme. Doktorát získala v roku 1968 za prácu v röntgenovej kryštalografii. Dnes je riaditeľkou jedného z centier na Weizmannovom vedeckom inštitúte v izraelskom meste Rehovot.

Podľa:
veda.sme.sk

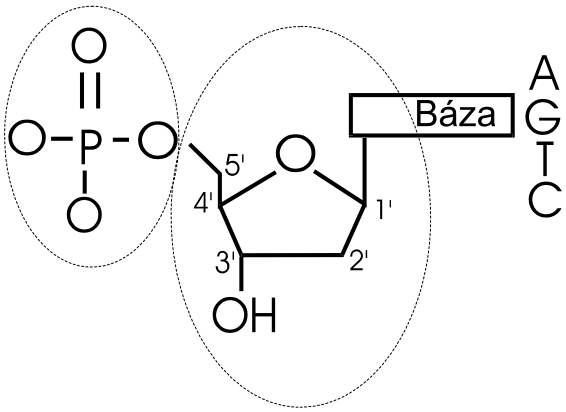
Teória výpočtov na báze molekúl DNA sa spája s menom amerického matematika Leonarda M. Adlemana z Kalifornskej univerzity. L. Adleman si všimol podobnosť biologických procesov so základnými procesmi súčasných výpočtových systémov. Ukázal, že DNA možno použiť ako výpočtové médium. To, čo na to potrebujeme, je aparát realizujúci základné editovacie operácie ako vymaž, presuň, či skopíruj sekvenciu DNA, ale na úrovni chemicko-biologickej.

Predpokladajme, že na to, aby bolo možné vykonávať operácie s DNA molekulami, máme k dispozícii potrebné chemikálie a potrebné predmety. Nebudeme podrobne analyzovať, prečo to prebieha práve takto, len pripomenieme niektoré dôležité poznatky z biológie a biochémie, aby sme mohli pochopiť a uveriť funkčnosti celého procesu.

Fakt, že sa navzájom môžu spájať len bázy A s T a G s C, nazývame Watsonova-Crickova komplementárnosť. DNA molekula má zložitú trojrozmernú štruktúru, ktorá je podstatná z hľadiska jej funkčnosti.

Štruktúra DNA

Podľa obrázku 2 na pozícii 1' sa na deoxyribózu viažu dusíkaté *bázy*, ktoré „vyčnievajú“ z reťazca DNA. Reťazec DNA je tvorený molekulami cukru vzájomne pospájanými na pozícii 5' a 3' *zvyškom kyseliny fosforečnej*. Teda základnou stavebnou jednotkou molekuly DNA je *nukleotid* (nucleotide). Nukleotidy sa odlišujú len bázou. Preto je možné abstrahovať od skutočných tvarov a zaoberať sa vlastne reťazcami v abecede $\{A, G, T, C\}$.



Obrázok 3.2. Štruktúra nukleotidu - základnej stavebnej jednotky DNA. V menšom kruhu sa nachádza fosfátová skupina, väčší kruh označuje cukor, na ktorý sa viaže jedna zo štyroch bázových molekúl - adenín (A), guanín (G), tymín (T) a cytozín (C).

Nukleotidy sa reťazia do molekuly DNA v procese nazývanom *polymerizácia*. Polymerizácia spočíva v reakcii medzi fosfátom na pozícii 5’ jedného nukleotidu a hydroxylovou skupinou na pozícii 3’ susedného nukleotidu.

Pre nás je dôležité vedieť, že rozpadnutie sa slabších väzieb medzi bázami môžeme dosiahnuť zvýšením energie (napríklad ohriatím). Týmto spôsobom vzniknú dva samostatné reťazce. Pri správnych podmienkach sa naopak z dvoch samostatných sekvencií môže vytvoriť jedna dvojreťazcová molekula, avšak len v prípade, že postupnosti sú komplementárne.

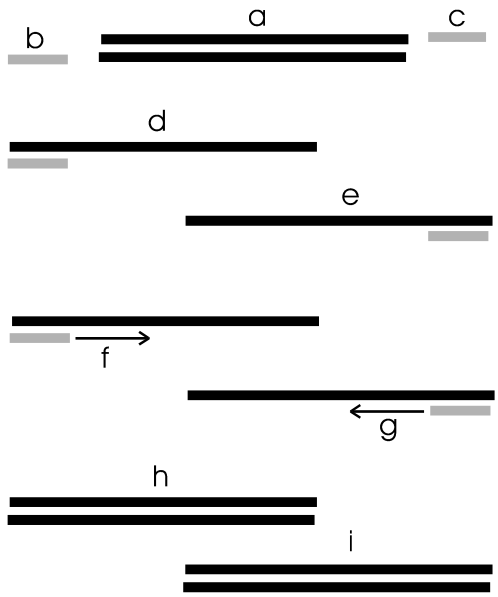
Ďalšou dôležitou vlastnosťou molekuly DNA je jej záporný náboj, ktorého veľkosť je priamoúmerná dĺžke molekuly.

Na základe toho vieme opísať niektoré chemické operácie s obsahmi skúmaviek nasledujúcim spôsobom: Zaveďme označenie skúmaviek R_1, R_2, R_3, T, atď. Operácie:

- ***Union*(R_i, R_j, R_k)** - Obsahy skúmaviek R_i a R_j daj do skúmavky R_k.

- ***Amplify*** (R_i) – Zdvojnásob počet DNA sekvencií v skúmavke R_i.

Tieto operácie sa nazývajú polymérová reťazová reakcia a sú založené na Watsonovej-Crickovej komplementárnosti. Dvojité reťazce sa najprv zahriatím rozložia na jednoduché, čo nazývame **denaturizácia**. Pridaním nukleotidov a ochladením dôjde k naviazaniu na komplementárne bázy jednotlivých reťazcov a znovu sa vytvoria identické DNA molekuly, ale ich počet je dvojnásobný. Na obrázku 3.3 je symbolicky znázornený priebeh vytvárania kópie molekuly DNA.



*Obrázok 3.3. Znázornenie techniky polymerizačnej reťazovej reakcie. Dvojitú závitnicu vzorovej DNA **a** zahriatím rozdelíme na dve komplementárne jednovláknové molekuly DNA **d-e**. Po ochladení sa iniciátory **b-c** viažu na vlákna **d** a **e**. Polymerizácia syntetizuje príslušné komplementárne vlákna **f-g**, takže z pôvodnej dvojitej závitnice DNA **a** dostávame dve identické kópie **h-i**.*

- ***Empty?*** (R_i) – Otestuj, či v skúmavke R_i sa nachádza aspoň jedna DNA molekula, alebo nie.
- ***Length-Separate*** (R_i, l) pre prirodzené číslo l – Operácia odstráni zo skúmavky R_i všetky DNA molekuly, ktoré nemajú dĺžku presne l báz.

Pri týchto operáciách sa využíva technika gélovej elektroforézy. Pretože DNA molekuly majú záporný náboj, keď ich umiestnime do elektrického poľa, budú sa pohybovať ku kladnej elektróde. Rýchlosť pohybu je ovplyvňovaná dvomi faktormi, a to veľkosťou a elektrickým nábojom DNA molekuly. Veľkosť molekuly ju brzdí, náboj molekuly ju zrýchľuje. Zdá sa, že všetky molekuly sa pohybujú rovnako rýchlo. Z tohto dôvodu pridáme do poľa gél, ktorý spomalí pohyb veľkých molekúl (dlhých DNA sekvencií). Pretože DNA molekuly sú bezfarebné, aby sme mohli celý proces pohybu molekúl sledovať, môžeme ich označiť fluorescenčnými farbami alebo rádioaktívne. Dá sa vypočítať rýchlosť jednotlivých molekúl v závislosti od ich dĺžky. Keď dosiahne najkratšia molekula kladnú elektródu, elektrické pole sa vypne a podľa dĺžok prejdených dráh je možné vypočítať dĺžky DNA sekvencií.

- ***Concatenate*** (R_i) – DNA sekvencie sa môžu spájať do dlhších DNA sekvencií pripojením za seba.
- ***Separate*** (R_i, w) – Operácia odstráni zo skúmavky R_i všetky DNA molekuly, ktoré v sebe neobsahujú sekvenciu w.

Napríklad $w = \text{ATTC}$ je časťou $x = \text{AATTCGATC}$, pretože celé w vyskytuje súvislé v x. Táto operácia si z chemického hľadiska vyžaduje oveľa viac námahy. Najprv sa všetky dvojité reťazce zahriatím rozpoja na jednoduché. Potom je potrebné pridať veľa kópií DNA sekvencií komplementárnych k w a všetko mierne ochladiť. Sekvencie

komplementárne k w sa naviažu na odpovedajúce časti w. Jednoduché reťazce, ktoré neobsahujú w, sa nedoplnia a ostanú jednoduché. Potom je potrebné všetko prefiltrovať, aby sa jednoduché reťazce odstránili. Zostanú dvojité reťazce, ktoré nie sú kompletné, ale tie vieme skompletizovať pridaním nukleotidov.

- ***Separate-Prefix(R_i, w)*** – Operácia odstráni zo skúmavky všetky DNA molekuly, ktoré nezačínajú sekvenciou w.
- ***Separate-Suffix(R_i, w)*** – Operácia odstráni zo skúmavky všetky DNA molekuly, ktoré sa nekončia sekvenciou w.

Úloha 3.7.	Nech je daná abeceda $\Sigma = \{A, G, T, C\}$. Koľko rôznych reťazcov dĺžky 1000 je možné vytvoriť z prvkov danej abecedy.
Úloha 3.8.	Nech skúmavka obsahuje nasledujúce sekvencie ATTGCATGC, ATATCAGCT, TTGCACGG, AACT, AGATGGT. Ktoré DNA molekuly zostanú po vykonaní nasledujúcich operácií? (a) *Lengt-Separate(R, 7),* (b) *Separate(R, TTGC),* (c) *Separate-Prefix(R,TTGC),* (d) *Separate-Suffix(R, GCT).*
Úloha 3.9.	Chcete vykonať operáciu Separate(R, AACT). Ktoré DNA sekvencie musíte po zahriatí pridať k R, aby ste mohli nasledujúcou filtráciou oddeliť nevhodné jednoduché reťazce?

V ďalšom opíšeme, akým spôsobom uvedené operácie je možné prakticky zrealizovať pomocou chemických reakcií a uvedieme tiež Adlemanov experiment aplikovaný na problém obchodného cestujúceho.

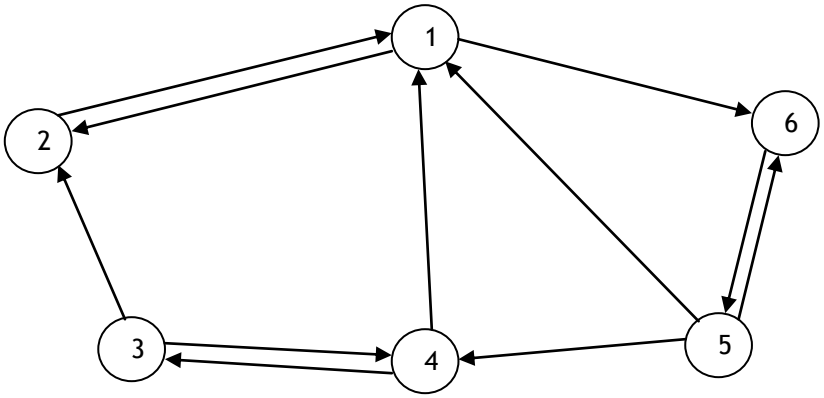
Adlemanov experiment

Uvedieme príklad DNA počítača, ktorý vyrieši algoritmický problém. Adleman vytvoril DNA počítač pre problém Hamiltonovej cesty (HPP – Hamilton Path Problem) v orientovanom grafe (v cestnej sieti). Je pravda, že riešenie urobil len pre jednu inštanciu problému, ale vytvoril tým základ ďalšieho výskumu v tejto oblasti.

Pre jednoduchosť, predstavme si cestnú sieť, mestá sú pospájané cestami. Takúto cestnú sieť vieme nakresliť tak, že mestá alebo križovatky budú znázornené krúžkami a cesty budú znázornené čiarami so šípkou, kam smerujú. Keď sa cesty pretínajú niekde inde ako na križovatke alebo v meste, to je to isté, akoby sa nepretínali. Cesty sú jednosmerné. Cestu z mesta m_1 do mesta m_2 budeme označovať $m_1 \rightarrow m_2$.

Problém Hamiltonovej cesty je rozhodovacím problémom a dáva odpoveď na otázku, či je možné začať v meste m_i, prejsť práve jedenkrát cez každé mesto v cestnej sieti a skončiť v meste m_j. Postupnosť ciest medzi mestami, ktorá spĺňa túto požiadavku sa, sa nazýva Hamiltonova cesta.

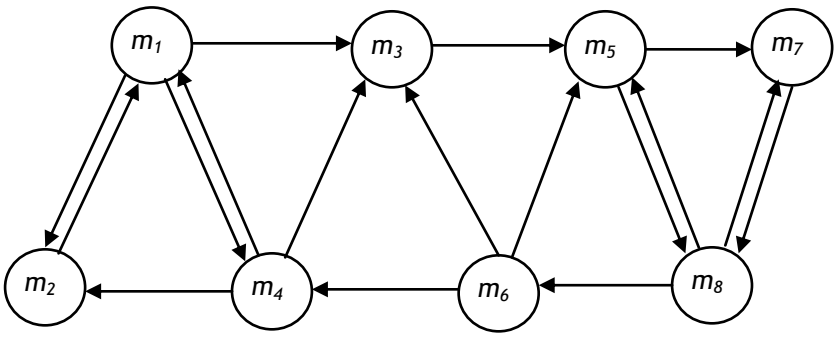
Príklad 3.1: Na obrázku 3.4. je znázornená cestná sieť so 6 mestami. Cesta zaznamenaná ako postupnosť vrcholov 3-2-1-6-5-4 je Hamiltonova cesta medzi mestami 3 a 4. Priama cesta z mesta 3 do mesta 4 nie je Hamiltonova cesta.



Obrázok 3.4. Cestná sieť so 6 mestami a 11 cestami.

Úloha 3.10.	Uvažujme cestnú sieť na obrázku 3.4. Nájdite dvojicu miest, medzi ktorými neexistuje Hamiltonova cesta.
Úloha 3.11.	Uvažujme cestnú sieť na obrázku 3.5. Existuje dvojica miest, medzi ktorými neexistuje Hamiltonova cesta?
Úloha 3.12.	Navrhnite cestnú sieť bez izolovaných miest, v ktorej pre žiadnu dvojicu neexistuje Hamiltonova cesta.
Úloha 3.13.	Navrhnite algoritmus na hľadanie Hamiltonovej cesty. Odhadnite jeho časovú zložitosť.

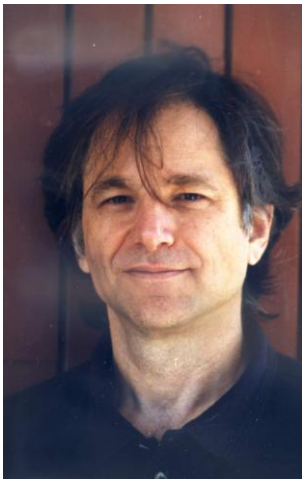
Príklad 3.2: Na obrázku 3.5. je cestná sieť tvorená 8 mestami. Postupnosť $m_1 \to m_3, m_3 \to m_5, m_5 \to m_7, m_7 \to m_8, m_8 \to m_6, m_6 \to m_4, m_4 \to m_2$ je Hamiltonova cesta.



Obrázok 3.5. Cestná sieť s 8 mestami a 17 cestami

Adleman uvažoval o inštancii uvedenej na obrázku 3.6. so štartom m_0 a cieľom m_6. Jeho stratégia bola nasledujúca:

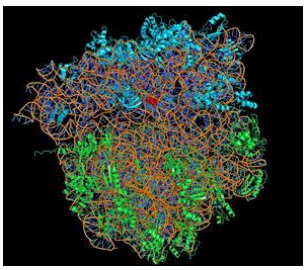
> *Mená miest v sieti je potrebné zakódovať pomocou DNA sekvencií. Potom je potrebné umožniť to, aby sa zakódované mená vzájomne prepojených miest mohli spojiť za sebou do DNA sekvencie. Použiť toľko DNA sekvencií pre každé mesto v sieti, aby pri náhodne zvolených vzájomných prepojeniach mohli vzniknúť všetky možné cesty v sieti. Aplikáciou rôznych operácií **Separate** je možné odstrániť zo skúmavky všetky cesty, ktoré nepredstavujú Hamiltonovu cestu z m_0 do m_6.*



Leonard Max Adleman (*31. december 1945, San Francisco, Kalifornia, USA) je americký informatik a profesor informatiky a molekulárnej biológie. Spoločne s Ronaldom Rivestom a Adim Shamirom je autorom šifrovacieho algoritmu RSA, za čo dostali všetci traja Turingovu cenu za rok 2002.

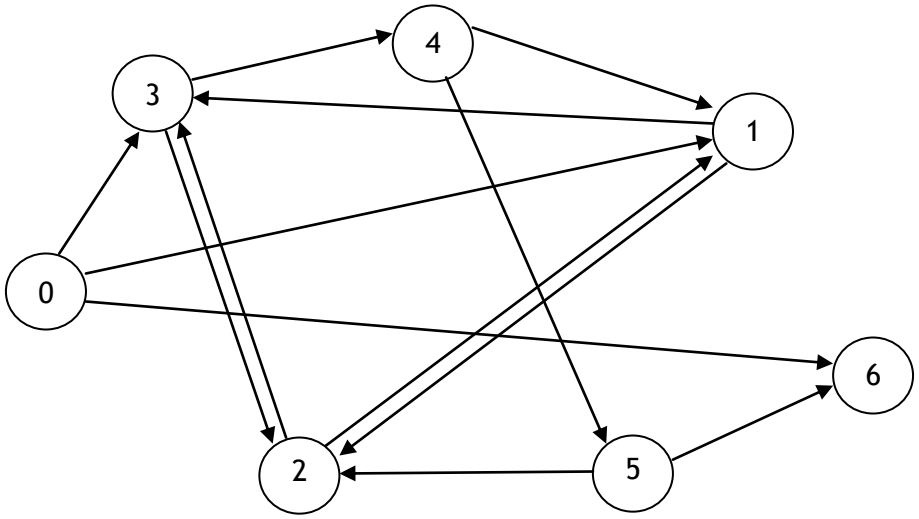
Podľa Wikipedie

Viac na: http://www.usc.edu/dept/molecular-science/fm-adleman-vitae.htm



Štruktúra ribozómu na molekulárnej úrovni bola objasnená metódou röntgenovej kryštalografie.

Podľa: http://korzar.sme.sk/c/5080073/nobelova-cena-za-chemiu.html#ixzz0mOalNJKV



Obrázok 3.6. Cestná sieť z Adlemanovho pokusu.

Predtým, než biochemicky uskutočnil tento postup, vybral pre mestá pomenovania pomocou DNA sekvencií, t. j. mená tvorené písmenami A, C, G, T. Napríklad pre mestá m_2, m_3, m_4 zvolil sekvencie báz dĺžky 20 takto:

m_2 = TATCGGATCGGTATATCCGA

m_3 = GCTATTCGAGCTTAAAGCTA

m_4 = GGCTAGGTACCAGCATGCTT

Teraz je potrebné reprezentovať cesty medzi dvomi mestami m_i a m_j ako tak jednoduché DNA reťazce, že operáciou *Concatenate* budú môcť vyniknúť len také dlhšie DNA reťazce, ktoré zodpovedajú existujúcim cestám v sieti.

Využijeme pritom vlastnosti väzieb báz v DNA, a síce A s T, C s G. Pre cestu z mesta m_i do mesta m_j urobíme nasledujúce:

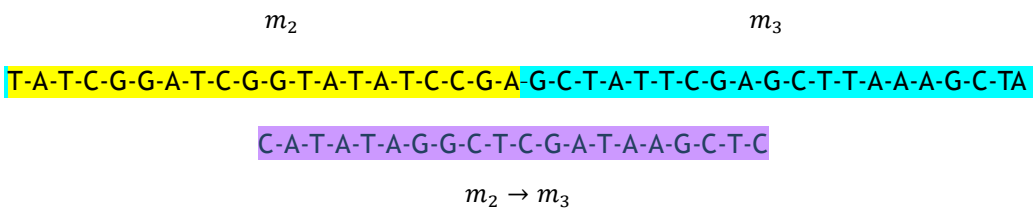
- Rozdelíme ich kódujúce mená (DNA reprezentácie miest m_i a m_j) vždy v strede na dve polovice.
- Vytvoríme doplnky k druhej časti m_i a prvej časti m_j – nahradíme A pomocou T, C pomocou G a opačne.
- Pre cestu z mesta m_i do mesta m_j vytvoríme reťazec (DNA reprezentáciu) spojením týchto dvoch doplnkov.

Dôležité je tiež to, že týmto kódovaním je určený aj smer cesty. V tomto príklade to znamená toto:

$m_2 \rightarrow m_3$ = CATATAGGCT CGATAAGCTC (ilustrácia na obrázku 3.6.)

$m_3 \rightarrow m_2$ = GAATTTCGAT ATAGCCTAGC

$m_3 \rightarrow m_4$ = GAATTTCGAT CCGATCCATG



Obrázok 3.7. Princíp vytvárania kódov pre cesty medzi susednými mestami

Úloha 3.14.	Uvažujme cestnú sieť na obrázku 3.6. Napíšte kód cesty $m_3 \rightarrow m_2$ a $m_3 \rightarrow m_4$.
Úloha 3.15.	Predstavme si, že do cestnej siete na obrázku 3.6. pridáme cestu z mesta m_2 do mesta m_4. Aký reťazec vyjadruje túto cestu?
Úloha 3.16.	Opíšte akým spôsobom je možné vytvoriť v skúmavke ku každej DNA molekule jej kópiu. Na konci by sme mali mať dvojnásobok molekúl (molekula a jej kópia).

Keď teraz zoberieme jednoduché reťazce ako DNA sekvencie odpovedajúce tomuto kódovaniu, dáme ich do skúmavky a pripravíme vhodné podmienky, reťazce sa môžu spájať tak, ako je to uvedené na obrázku 3.6.

Každý takýto dvojitý reťazec opisuje nejakú cestu v cestnej sieti. Aby toto prebehlo bezchybne, je potrebné zabezpečiť, aby sa kódy ciest predlžovali len vhodným spôsobom (tak, ako je to uvedené v príklade). Predovšetkým musia byť všetky polovice kódov miest navzájom rôzne.

Keď je kódovanie takto zabezpečené, je možné zrealizovať Adlemanovu stratégiu hľadania Hamiltonovej cesty podľa nasledujúceho postupu:

1. Do skúmavky T dáme DNA kódy všetkých miest a ciest (ako jednoduché reťazce). Pridáme DNA sekvenciu dĺžky 10, ktorá je doplnkom k prvej polovici prvého mesta m_0 a DNA sekvenciu dĺžky 10, ktorá je doplnkom k druhej polovici cieľového mesta m_n.

2. Zopakujeme $(2n \log_2 n)$-krát operáciu *Amplify(T)*, aby vzniklo aspoň n^{2n} kópií z každého z týchto DNA reťazcov.

3. Pomocou operácie *Concatenate(T)* vytvoríme veľké množstvo dvojreťazcových DNA sekvencií, ktoré predstavujú rôzne dlhé cesty v cestnej sieti. Tento proces prebieha náhodne a veľký počet názvov miest nám zaručí, že s veľkou pravdepodobnosťou vznikne každá zo všetkých možných ciest, ktorých dĺžka je až n.

4. Aplikáciou operácie *Length-Separate(T, l)*, kde l je n násobok dĺžky kódu jedného mesta. V T ostanú len kódy zodpovedajúce cestám, ktoré prechádzajú presne cez n miest.

5. Použitím operácie *Separate-Prefix(T, m_0)* dosiahneme to, aby v T zostali len také DNA sekvencie, ktoré odpovedajú kódom ciest začínajúcim v m_0, čo je počiatočné mesto.

6. Použitím operácie *Separate-Suffix(T, m_n)* dosiahneme to, aby v T zostali len také DNA sekvencie, ktoré odpovedajú kódom ciest končiacim v m_n, čo je koncové mesto.

7. Použijeme $(n-2)$-krát operáciu *Separate*(T,x) pre každý z $n-2$ kódov zvyšných miest. Takto v T zostanú len tie cesty, ktoré každé mesto obsahujú aspoň raz. (Pretože 4. krok nám zaručí, že v T sú len cesty prechádzajúce presne n mestami, obsahujú tieto cesty každé mesto práve jedenkrát.)

8. Preskúmame obsah T pomocou *Empty?*(T) a dostaneme odpoveď ÁNO, ak v T zostala aspoň jedna DNA sekvencia. V opačnom prípade dostaneme odpoveď NIE.

Použitím tohto postupu na sieť na obrázku 3.5. so štartom v meste m_0 a cieľom v meste m_6 dostaneme, že v skúmavke zostane dvojitý reťazec DNA, ktorý predstavuje Hamiltonovu cestu $m_0 \to m_1$, $m_1 \to m_2$, $m_2 \to m_3$, $m_3 \to m_4$, $m_4 \to m_5$, $m_5 \to m_6$. Teda odpoveď bude ÁNO.

Úloha 3.17.	Nájdite aspoň tri rôzne cesty v sieti na obrázku 3.6., ktoré zostanú v skúmavke po vykonaní piateho kroku v Adlemanovom postupe. Čo zostane po vykonaní 6. kroku?
Úloha 3.18.	Uvažujme cestnú sieť na obrázku 3.5. Navrhnite (čo najkratšie) kódovanie (prostredníctvom DNA) miest také, že budú splnené nasledujúce podmienky: ▪ Kód každého mesta sa odlišuje od kódu ľubovoľného iného mesta najmenej na 4 pozíciách. ▪ Prvá aj druhá polovica kódu každého mesta sa odlišuje od prvej aj druhej polovice kódu ľubovoľného iného mesta. Ako budú vyzerať kódy ciest medzi mestami?
Úloha 3.19.	Opíšte detailne priebeh Adlemanovho postupu pre vstup z úlohy 3.18., pričom uveďte postupnosť operácií s konkrétnymi parametrami (DNA sekvenciami a dĺžkami).
Úloha 3.20.	Sú požiadavky na kódovanie miest uvedené v úlohe 3.18. postačujúce na to, aby každý vytvorený dvojitý DNA reťazec zodpovedal ceste medzi mestami? Zdôvodnite svoju odpoveď.

Práce venované výpočtom na báze DNA sledujú dva hlavné trendy [1]:

1. skúmanie vhodných techník kódovania dát sekvenciami DNA, spracovania dát manipuláciou s DNA a interpretácie výsledkov výpočtov;
2. určenie výpočtovej sily rôznych modifikácií výpočtov na báze DNA.

Zo zaujímavých aplikácií výpočtov na báze DNA spomenieme pokus D. Boneha, C. Dunwortha a R. Liptona navrhnúť systém na dešifrovanie štandardu šifrovania dát (Data Encryption Standard - DES) [9]. Tento problém študoval aj „otec“ počítačov na báze DNA L. Adleman [10]. Výpočty na báze DNA sa realizujú zväčša *in vitro*, teda v skúmavkách mimo živých organizmov. Alternatívne je možné skonštruovať počítače na báze DNA *in vivo*. Ako vhodné médium boli použité baktérie. V baktériách *Escherichia coli* možno znásobiť želané úseky DNA technikou molekulárneho klonovania využívajúcou plazmidy ako nosiče cudzorodej DNA.

Čo sme sa naučili

Naučili sme sa základné idey a pracovné postupy prudko sa rozvíjajúcej vednej disciplíny - výpočtov na báze DNA. Z informatického hľadiska môžu počítače na báze DNA podstatne posunúť hranice praktickej riešiteľnosti mnohých kombinatorických problémov, v ktorých priestor potenciálnych riešení rastie exponenciálne s rozsahom problému.

Literatúra a použité zdroje

[1] Kvasnička, V., Pospíchal, J., Tiňo, P.: Evolučné algoritmy. STU Publishing, Bratislava 2000, ISBN 80-227-1377-5.

[2] Hromkovič, J. (2009) Algorithimc Adventures, From Knowledge to Magic, Springer, ISBN 978-3-540-85985-7

[3] Pačes, J.: Co se o sobě dovídáme z naší genetické informace, Ústav molekulární genetiky AV ČR, Praha, *http://archiv.otevrena-veda.cz/users/Image/ default/ C1Kurzy/Biolog/7paces.pdf*

[4] Pačes, V.: Genomika - věda pro 21. století. Ústav molekulární genetiky AV ČR a VŠChT, Praha. http://www.img.cas.cz/paces/Genomika_2000.htm

[5] Palušáková, M.: Vizualizácia a analýza algoritmov pre prácu s DNA a RNA štruktúrami. Bakalárska záverečná práca. PF UPJŠ, Košice, 2008.

[6] Adleman, L.: Molecular computation of solutions to combinatorial problems. *Science* **266** (1994) 1021-1024.

[7] Lipton, R.J.: DNA Solution of Hard Combinatorial Problems. *Science* **268** (1995) 542-545.

[8] Gusfield, D.: Algorithms on Strings, Trees and Sequences. Cambridge Univ. Press, Cambridge, 1997.

[9] Adleman, L. M., Rothemund, P. W. K.. Roweis, S. and Winfee, E.: On Applying Molecular Computation to the Data Encryption Standard. *Journal of Computational Biology* **6**(1) (1999) 53-64.

[10] Bertone, P. N.: *Combinatorial Solution of Search Problems via Biomolecular Computation in vivo* (manuscript) (1996). Dostupný na internetovskej adrese *http://www.umassd.edu/Public/People/PBertone/research/Biocomp_Rev_227.pdf*.

[11] Vinař, T.: Systém MENDEL - genetika na počítači (The MENDEL system - genetics on computer), Informatika v škole 9/1993, s. 20-22, UIP Bratislava

[12] Watson, J.D.: Tajemvství DNA . Akademie, Praha 1995.

[13] Brejová, B., Vinař, T.: Letná škola bioinformatiky. Bratislava, 2007. Dostupné na: *compgen.bscb.cornell.edu/~tvinar/lsb2007/notes.pdf*

Zadanie 3.1.	Vytvoriť algoritmus a program pre vyhľadávanie krátkej sekvencie (do 10 prvkov) vo veľmi dlhej sekvencii prvkov (aspoň 1000 prvkov). Vypíšte dĺžku prehľadávanej sekvencie a počet spotrebovaných porovnaní.

Čo sme sa naučili v tomto module

Zhrnutie

Modul pozostáva z dvoch častí, ktoré poskytnú čitateľovi obraz o

- princípoch tvorby efektívnych algoritmov
- nových trendoch v informatike zameraných na pravdepodobnostné algoritmy a Preverenie výstupných vedomostí

Preverenie výstupných vedomostí

Preverenie výstupných vedomostí je možné urobiť pomocou analýzy jednoduchých algoritmov na záver prezenčnej časti modulu alebo odovzdaním navrhnutého zadania. Lektor hodnotí aktivitu frekventantov pri riešení problému.

Tento študijný materiál vznikol ako súčasť národného projektu Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika v rámci Aktivity „Ďalšie vzdelávanie kvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ“.

Autori © Doc. RNDr. Gabriela Andrejková, CSc.
RNDr. Michal Forišek, PhD.
Mgr. Juliana Šišková
RNDr. Michal Winczer, PhD.

Názov Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

Podnázov Kapitoly z informatiky

Študijný materiál prešiel recenzným pokračovaním.

Recenzenti doc. RNDr. Stanislav Krajči, PhD.
Mgr. Ján Skalka, PhD.

Počet strán 72

Náklad 400 ks

Prvé vydanie, Bratislava 2010

Vydal Štátny pedagogický ústav, Pluhová 8, 830 00 Bratislava, v súčinnosti s Univerzitou Pavla Jozefa Šafárika v Košiciach, Univerzitou Komenského v Bratislave, Univerzitou Konštantína Filozofa v Nitre, Univerzitou Mateja Bela v Banskej Bystrici a Žilinskou univerzitou v Žiline

Vytlačil BRATIA SABOVCI, s r.o., Zvolen

ISBN 978-80-8118-052-1