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 1

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 1

Identifikácia modulu

Aktivita projektu:	1.2 Vzdelávanie nekvalifikovaných 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:

RNDr. Michal Forišek, PhD., KI FMFI UK, Bratislava
Mgr. Juliana Šišková, KZVI FMFI UK, Bratislava

Zaradenie modulu

Modul *Kapitoly z informatiky 1* je zo série modulov *Kapitoly z informatiky*, ktoré majú za úlohu priniesť frekventantom základy informatiky ako vedy. Modul má ako prerekvizitu všetky programátorské moduly *Programovanie 1-9* a tiež moduly *Matematika 1-3*, *Algoritmy a údajové štruktúry 1, 2* a *Počítačové systémy 1-5*.

Modul má e-learningovú podporu

Abstrakt modulu

Predmet *Kapitoly z informatiky* má za úlohu frekventantom predstaviť rôzne oblasti informatiky ako vedy. V tomto module sme vybrali tému efektívnych algoritmov, a to preto, že je to jednak oblasť, ktorá pomáha pochopiť veľa rôznych disciplín teoretickej a aj aplikovanej informatiky, a tiež preto, že učitelia potrebujú mať isté kompetencie v tvorbe efektívnych algoritmov, aby mohli nasmerovať talentovaných študentov a rozumieť, čomu sa študenti v informatike venujú v rámci neformálneho vzdelávania.

Budeme sa snažiť frekventantov motivovať k tomu, aby k problému pristupovali rôznymi spôsobmi a vybrali si tie z ich riešení, ktoré sú aj reálne uskutočniteľné. Naučíme ich používať niektoré techniky, ktoré im pomôžu tvoriť *efektívne* algoritmy. Týmito technikami budú pažravé algoritmy a princíp dynamického programovania.

Obsah

Kapitoly z informatiky 1 1
Identifikácia modulu 1
Zaradenie modulu 1
Abstrakt modulu 1
Obsah 2
Úvod 3
Cieľ modulu 3
Vstupné vedomosti 3
Požadované prerekvizity 3
Predpokladané vstupné vedomosti a zručnosti 3
Preverenie vstupných vedomostí 3
1. Efektivita algoritmov 4
1.1 Potreba efektívnych algoritmov 4
1.2 Časová zložitosť 6
1.3 Čo sme sa naučili 12
1.4 Úlohy na precvičenie 12
2. Pažravé algoritmy 14
2.1 Úloha o zlate 14
2.2 Úloha o zastávkach autobusu 14
2.3 Úloha o dláždení 16
2.4 Čo to teda sú tie pažravé algoritmy? 19
2.5 Úlohy na precvičenie 19
3. Programovanie pomocou rekurzie 20
3.1 Čo je to rekurzia 20
3.2 Fraktály 20
3.3 Rekurzia v matematike 21
3.4 Programovanie pomocou rekurzie 22
3.5 Kedy je lepšie rekurziu použiť 24
3.6 Čo sme sa naučili 28
4. Dynamické programovanie 29
4.1 Úloha o loďke 29
4.2 Memoizácia 30
4.3 Dynamické programovanie 32
4.4 Čo sme sa naučili 36
Čo sme sa naučili v tomto module 37
Preverenie výstupných vedomostí 37
Literatúra a použité zdroje 37

Úvod

Predmet *Kapitoly z informatiky* je venovaný rôznym témam informatiky. Jeho prvá časť sa zaoberá efektívnymi algoritmami. Postupne sa naučíme posudzovať efektívnosť algoritmov pomocou odhadovania časovej a pamäťovej zložitosti a budeme vytvárať efektívne algoritmy na riešenie rôznorodých problémov využitím niektorých techník.

Efektivita algoritmov je jedna zo stavebných tém teoretickej, ale aj aplikovanej informatiky. Ak človek nájde riešenie problému, častokrát to nepostačuje. Musí tiež zhodnotiť, či jeho program prebehne v reálnych podmienkach v dostupnom čase. V teoretickej informatike sa zase zložitosti používajú na analýzu sily programovacieho modelu (napríklad aj programovacieho jazyka).

V module sa poslucháč naučí riešiť časť algoritmických problémov z rôznych oblastí. To by mu malo zmeniť pohľad na svet, keď zistí, že sme vlastne obklopení algoritmickými úlohami. Náš mozog si denno-denne vytvára algoritmy na riešenie drobných úloh. My sa v module naučíme tieto algoritmy vytvárať tak, aby boli dostatočne rýchle.

Všetky uvedené témy sú dôležité nielen na získanie nadhľadu nad informatikou, ale tiež na odbornú prípravu nadaných študentov.

Niektoré časti tohto modulu sú spoločné s modulom [1], ktorý tematicky pokrýva väčšinu tohto modulu. Pre tento modul však bola vyvinutá nová sada úloh a príkladov, na ktorých sú tieto spoločné témy prezentované. Pre čitateľa jedného textu teda môže byť poučné siahnuť aj po druhom z nich.

Cieľ modulu

Cieľom modulu je motivovať poslucháčov k hľadaniu efektívnych algoritmov, naučiť ich analyzovať časovú a pamäťovú zložitosť algoritmu a zhodnotiť, či je dostatočne efektívny pre dnešné počítače.

Po absolvovaní modulu by mali byť schopní naprogramovať riešenie niektorých typov problémov a nemali by sa zľaknúť riešiť neznáme úlohy, pretože už budú poznať klasické techniky v informatike.

Vstupné vedomosti

Požadované prerekvizity

Účastník vzdelávania má mať absolvované všetky moduly *Programovanie* a moduly predmetu *Algoritmy a údajové štruktúry*.

Predpokladané vstupné vedomosti a zručnosti

Účastník vzdelávania dokáže programovať ľahšie algoritmy, podobné tým, s ktorými sa stretol v predchádzajúcich moduloch. Nezľakne sa neznámeho typu problému. Dokáže riešenie popísať algoritmom. Vie riešiť ľahšie úlohy z kombinatoriky.

Preverenie vstupných vedomostí

Frekventant odpovie na otázky z kombinatoriky. Popíše vlastnými slovami zadaný algoritmus.

1. Efektivita algoritmov

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?*

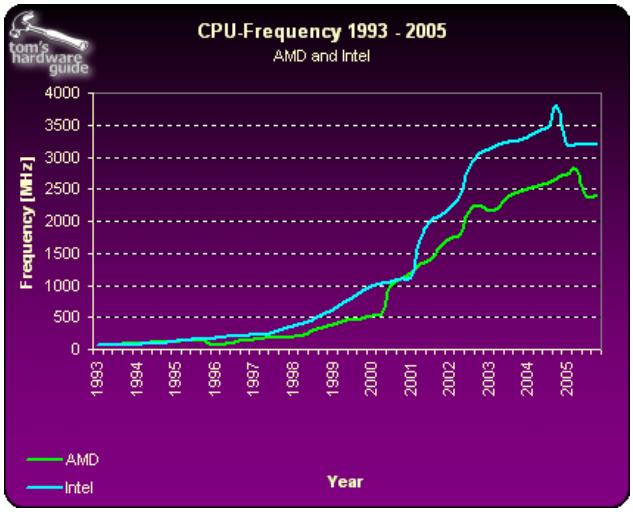
1.1 Potreba efektívnych algoritmov

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 jeho asi najznámejšia 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.

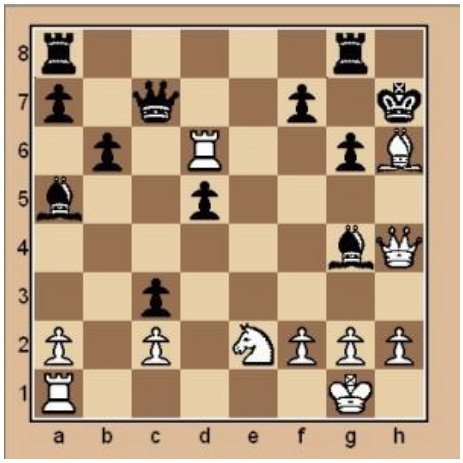
Spolu s rastom výpočtovej sily bežných počítačov rastie totiž aj objem dát, ktoré naša spoločnosť potrebuje denne spracovať.

Keď sa opäť pozrieme približne 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 obtažnosť 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.

1.2 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 však 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

Vypočítať, 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.

Úloha 1

Študent Adam má dva týždne na to, aby sa naučil na skúšku. Učí sa tak, že rieši úlohy. Prvý deň vyriešil len jednu úlohu. Ako sa blíži termín skúšky, tak sa aj Adam viac sústreďuje na učenie: na druhý deň vyrieši tri úlohy, na tretí deň ich je päť, a tak ďalej – každý deň o dve úlohy viac ako v predchádzajúci deň. Koľko úloh dokopy vyrieši?

Riešenie je jednoduché: vypočítame súčet aritmetickej postupnosti, ktorá má 14 členov a začína $1, 3, 5, 7, \ldots$ Priamym výpočtom dostávame, že Adam vyrieši 196 úloh.

Úloha 2

Koľko hviezdičiek vypíše nasledujúci program, ak premenná `N` obsahuje hodnotu `14`?

```
for i := 1 to N do
  for j := 1 to 2*N - 1 do
    write('*');
```

V programe sme použili príkaz write. Ten sa v Delphi/Lazarus používa pri písaní do súboru alebo v konzolovej aplikácii. Príkaz write môžeme nahradiť vypisovaním do komponentu TMemo, avšak samotný kód by tým vyzeral zložitejšie.

Úloha 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 tento krát oblečená do programátorského kabáta.

Aby sme lepšie videli, čo sa pri vykonávaní programu deje, upravme si ho nasledovne:

```
for i := 1 to N do
begin
   writeln('zacal den cislo ', i);
   for j := 1 to 2*N - 1 do
     writeln(' Adam vyriesil ulohu');
end;
```

Skúste pre hodnotu $N = 5$ ručne odsimulovať tento upravený program. Aký výstup dostanete? Koľko riadkov 'Adam vyriesil ulohu' sa vypíše?

Premenná `i` teda zodpovedá dňu v našom prvom príklade, a každá vypísaná hviezdička zodpovedá jednej Adamom vyriešenej úlohe. Teda ak premenná `N` obsahuje hodnotu `14`, vypíše tento program presne `196` hviezdičiek.

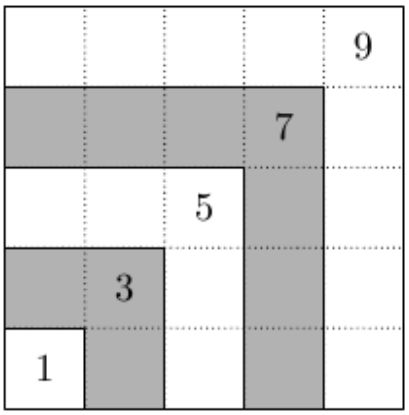
Ako je to vo všeobecnosti? Pozrime sa na vzorec pre súčet aritmetickej postupnosti. Aritmetická postupnosť, ktorá má n členov, prvý člen a a diferenciu d, má súčet $an + \frac{n(n-1)\ d}{2}$. V našom prípade je $a = 1$ a $d = 2$, teda prvých n členov Adamovej postupnosti má súčet $n + n(n-1) = n^2$. Oplatí sa spomenúť aj to, prečo vyššie uvedený vzorec platí: Členy postupnosti, ktorú sa snažíme sčítať, sú a, $a + d$, …, $a + (n-1)d$. Vyplňme si tabuľku s dvomi riadkami a n stĺpcami. Do prvého riadku vpíšeme našu postupnosť, do druhého tú istú postupnosť, ale sprava doľava.

1	3	5	7	9	11	13	15	17
17	15	13	11	9	7	5	3	1

Tabuľka 1: Súčet aritmetickej postupnosti. Príklad pre $a = 1$, $d = 2$, $n = 9$.

Takto dostaneme tabuľku s n stĺpcami, pričom v každom stĺpci je súčet $2a + (n-1)d$. Súčet celej tabuľky je teda $2an + n(n-1)d$. No a súčet celej tabuľky je vlastne dvojnásobkom súčtu našej aritmetickej postupnosti, čím dostávame vyššie uvedený vzorec.

V našom konkrétnom prípade však na dôkaz toho, že Adam za n dní vypracuje presne n^2 úloh, stačí jednoduchý obrázok:



Obrázok 3: Súčet zadanej postupnosti

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.

Úloha 4

Koľko hviezdičiek vypíše nasledujúci program pre `N=10` a pole `A[1..N]` obsahujúce hodnoty `(1, 2, 3, 10, 10, 11, 12, 16, 33, 38)`?

```
for i := 1 to N do
begin
  j := i + 1;
  while (j <= N) and (A[j] - A[i] <= 5) do
  begin
    write('*');
    int(j);
  end;
end;
```

Úloha 5

Slovne popíšte, ako súvisí počet vypísaných hviezdičiek s obsahom poľa `A` (za predpokladu, že je toto pole usporiadané podľa veľkosti).

Ručnou simuláciou môžeme zistiť, že pre konkrétny vstup zo zadania vypíše tento program 12 hviezdičiek.

Ak je pole `A` usporiadané podľa veľkosti, ľahko nahliadneme, že tento program vypíše hviezdičku za každú dvojicu prvkov, ktorá sa líši nanajvýš o `5`.

Úloha 6

Nájdite k programu z predchádzajúcej úlohy pre `N=10` nejaké pole `A`, pre ktoré tento program vypíše najväčší možný počet hviezdičiek.

Ako tento najväčší možný počet hviezdičiek závisí od čísla `N`?

Jeden konkrétny prípad, kedy program vypíše najviac hviezdičiek, je prípad, keď sú všetky prvky v poli `A` rovnaké. Vtedy program vypíše hviezdičku pre každú dvojicu prvkov – ak teda máme `N` prvkov, dostaneme na výstupe $\frac{N\,(N-1)}{2}$ hviezdičiek.

Praktický pohľad

V predchádzajúcej časti sme pre veľmi jednoduché programy dokázali presne vypočítať, 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 Alojz píše softvér pre kódovanie hlasu počas telefónneho rozhovoru. S frekvenciou 8000 Hz mu hardvér zaznamenáva hlas človeka, hovoriaceho do mikrofónu, a Alojzov program musí tieto dáta stíhať v reálnom čase spracúvať a odosielať.

Betkin program slúži na kompresiu videa z DVD nosiča. Vie, že čím bude jej program rýchlejší, tým skôr sa používateľ dočká výsledku, a tým viac DVD stihne za rovnaký čas spracovať.

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.

Človek by si mohol myslieť, že program zo zadania spraví N^2 krokov, ale v skutočnosti spraví viac, lebo je tam ešte aj pár schovaných krokov (priradenie do premennej vo for-cykle). Neskôr zistíme, že to presné číslo aj tak nie je dôležité.

Ako definovať časovú zložitosť?

V úlohe 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 $2N^2 + N$ 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 úlohy o prvkoch, ktoré sa líšia nanajvýš o 5. Ako počet vypísaných hviezdičiek, tak aj celkový počet krokov algoritmu závisí nie len od veľkosti vstupného poľa, ale aj od toho, aké hodnoty toto pole obsahuje. Ľahko nahliadneme, že už pre takýto jednoduchý algoritmus neexistuje žiaden dostatočne 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 uvedeného zadania by nás teda mohla zaujímať odpoveď na otázku: Ak má pole `P` dĺžku `N` , koľko **najviac** krokov spraví uvedený program?

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

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.)

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

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.

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 len čo 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.

Úloha 7

Programátor Cyrano si chce v lese vyznačiť štvorcovú parcelu, a to tak, že natiahne špagát okolo vhodných štyroch stromov. Nepodarilo sa mu však také stromy nájsť, preto sa rozhodol, že si na to napíše program. Jeho program zo vstupu načíta súradnice všetkých stromov v lese a potom postupne vyskúša všetky štvorice stromov a pre každú zistí, či dotyčné štyri stromy určujú štvorec.

Odhadnite, ako rýchlo tento program spracuje 100 stromov. A čo $1\,000$ stromov? Alebo $10\,000$?

Ak máme N stromov, tak štvoríc stromov je $\binom{N}{4} = \frac{N\,(N-1)(N-2)(N-3)}{24}$, čo je približne rovné $\frac{N^4}{24}$. Pre každú štvoricu nám stačí konštantný počet matematických operácií na to, aby sme overili, či ide o štvorec alebo nie. Čas behu programu bude teda rásť približne so štvrtou mocninou počtu spracúvaných stromov.

Dosadením konkrétnych hodnôt za N dostávame, že pre $N = 100$ pobeží program rádovo stotiny sekundy, pre $N = 1\,000$ to už bude niekoľko minút a pre $N = 10\,000$ už budeme na odpoveď čakať niekoľko mesiacov.

Úloha 8

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 Filoménineho 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?

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.

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.

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 Filoménineho 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 Filoménineym.

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:

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 $f(n) \leq c \cdot g(n)$. Preložené z matematickej reči, tento zápis hovorí, že funkcia f rastie nanajvýš rádovo tak rýchlo ako funkcia g.

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.

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

Príklady použitia:

- „Časová zložitosť Cyranovho programu z úlohy 7 je $O(N^4)$.“
- „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).

1.3 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.

1.4 Úlohy na precvičenie

Úloha 9

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

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

```
for i := 1 to N do
  read( A[i] );
for i := 1 to N do
  for j := i + 1 to N do
    if 2*A[i] = A[j] then
      write('*');
```

Úloha 10

Je pravdivé tvrdenie: „*Časová zložitosť programu z úlohy 9 je kubickou funkciou premennej* N“?

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

A je pravdivé tvrdenie: „Časová zložitosť programu z úlohy 9 je $O(N^3)$? V čom je rozdiel oproti predchádzajúcej otázke?

Úloha 11

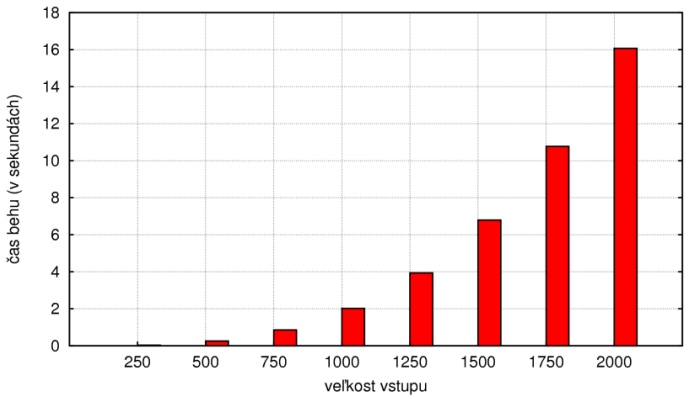
Akých 20 hodnôt treba zadať programu z úlohy 9, aby vypísal presne 99 hviezdičiek? (Riešení je veľa, stačí nájsť jedno ľubovoľné.)

Úloha 12

Odhadnite, ako dlho by program z úlohy 9 bežal pre `N = 10000`. (Predpokladajte, že načítanie čísel zo vstupu ho nijak nezdrží.)

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

Úloha 13



Našli sme na disku neznámy program. Vyskúšali sme ho spustiť na vstupoch rôznej veľkosti (a zavírili sme si počítač). V grafe 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?

2. Pažravé algoritmy

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

2.1 Úloha o zlate

Úloha 14

Veštica Teodora, ktorá sa nikdy nemýli, nám predpovedala, ako sa bude vyvíjať cena zlata v nasledujúcich dňoch. Ako s ním obchodovať, aby sme si čo najviac zarobili?

Skúste si napríklad ručne vyriešiť nasledovný prípad:

Máme v hotovosti **300** eur. Dnešná cena zlata je **34** eur za gram. V nasledujúcich dňoch sa táto cena bude meniť nasledovne: zajtra **32** eur/g, pozajtra **30** eur/g, v ďalších dňoch to bude postupne **35**, **33**, **32**, **38**, **40** a **37** eur/g.

Nebojte sa vyskúšať možnosti. Skúšanie pomáha nachádzať protipríklady a princípy, o ktoré sa v dôkazoch opierame.

Viete mať na konci posledného dňa **400** eur? A dá sa dosiahnuť ešte viac?

Riešenie úlohy 14

Optimálne riešenie vyzerá nasledovne: Počkáme, kým cena klesne na 30 eur za gram. Vtedy nakúpime za všetky peniaze 10 gramov zlata. Na druhý deň ho zase všetko predáme, takže budeme mať 350 eur. Znova počkáme, kým cena klesne na 32 eur/g. Vtedy nakúpime zlato, budeme ho mať zhruba 10,9 gramu. Počkáme si na cenu 40 eur/g a všetko zlato predáme, čím dostaneme výslednú sumu 437,5 eura.

Koľko budeme mať na konci obchodovania peňazí, ak nakúpime pri najlacnejšom zlate a predáme pri najdrahšom?

Všimnime si, že v našej stratégii na riešenie úlohy sa striedajú dva kroky: počkáme na lacné zlato a nakúpime; počkáme na drahé zlato a predáme. Ako ale exaktne definovať, čo je „lacné zlato , a teda kedy nakupovať a kedy predávať?

Vo všeobecnosti sa algoritmus na nájdenie optimálneho zárobku dá sformulovať až prekvapujúco jednoducho. Všimnime si, že každú noc sa nejak zmení cena zlata. Ak narastie, chceme tú noc vlastniť zlato, aby sme rastom ceny zarobili. Ak klesne, chceme tú noc vlastniť peniaze, aby sme neprerobili.

Takto sa teda každý večer môžeme pažravo rozhodnúť, či chceme zlato alebo peniaze, a podľa toho nakúpiť alebo predať.

V tejto jednoduchej úlohe sme sa teda stretli so situáciou, kedy sme **globálne optimálne** riešenie zostrojili tak, že sme postupne urobili niekoľko **lokálne optimálnych rozhodnutí**. (K najvyššiemu záverečnému kapitálu sme sa dopracovali tak, že sme každý deň zodpovedali otázku: „Čo spraviť, aby som mal zajtra čo najviac peňazí?“)

2.2 Úloha o zastávkach autobusu

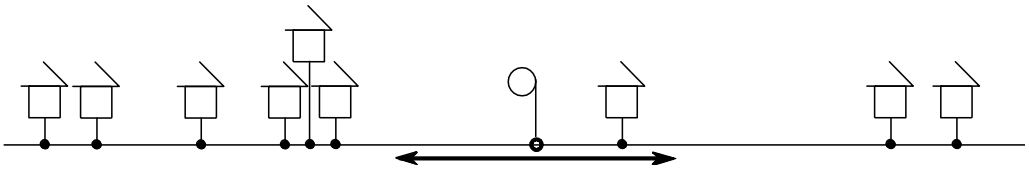
Úloha 15

V obci stojí niekoľko domov. Všetky stoja pri jedinej rovnej ceste, ktorá cez obec prechádza. O každom dome vieme, na koľkom metri cesty od začiatku obce má bránu.

Našou úlohou je rozmiestniť na ceste čo najmenej autobusových zastávok tak, aby sme nimi pokryli celú obec. Presnejšie, zastávky musia byť umiestnené tak, aby to nik nemal z domu na zastávku ďalej ako 500 metrov.

Mnohé naše obce sú postavené pozdĺž ciest, častokrát sa okolo nich tiahnu celé kilometre. Napríklad Kolárovice v okrese Bytča majú dĺžku okolo 10 km. V tejto úlohe budeme do takejto obce rozmiestňovať autobusové zastávky.

Príklad: V Kocúrkove stoja domy na nasledujúcich súradniciach: 100, 250, 600, 1000, 1100, 1200, 2300, 3300 a 3450 metrov od začiatku obce. Nájdite ručne čo najlepšie rozmiestnenie zastávok. Situáciu si môžeme znázorniť nasledovne:



Začiatok obce je vľavo. Na obrázku je znázornené aj jedno možné umiestnenie zastávky, je tam však len kvôli znázorneniu jej dosahu — teda vy ju na danom mieste použiť nemusíte.

Optimálne riešenie je postaviť štyri zastávky. Existuje veľa spôsobov, ako to spraviť: napríklad budeme mať zastávky na súradniciach 300 (sem budú chodiť z prvých troch domov), 1100 (štvrtý až šiesty dom), 2800 (siedmy a ôsmy dom) a 3350 (posledný dom).

Iné, rovnako dobré riešenie, je postaviť zastávky na súradniciach 600, 1200, 2300 a 3333. Ako ale dokázať, že nám na túto obec nestačia tri zastávky? Jeden možný argument môže vyzerať nasledovne:

Všimnime si prvý a šiesty dom. Ich vzdialenosť je 1100 metrov, čo je viac ako dĺžka úseku obsluhovaného jednou zastávkou. Preto nech by sme akokoľvek umiestnili zastávku, ktorá obsluhuje prvý dom, nikdy nebude stačiť na obslúženie šiesteho. (Presnejšie, vieme ňou obslúžiť nanajvýš prvých 5 domov, a to tak, že ju postavíme na súradnici 600.)

Analogicky sa môžeme pozrieť na šiesty a siedmy dom, opäť vidíme, že ich vzdialenosť je priveľká na pokrytie tou istou zastávkou. A pre siedmy a deviaty dom je situácia rovnaká. Na to, aby sme obslúžili domy s číslami 1, 6, 7 a 9, teda určite musíme použiť štyri rôzne zastávky.

Ako teda sformulovať všeobecný algoritmus, ktorý bude túto úlohu riešiť a vždy nájde optimálne riešenie? Náznak správnej úvahy sme už spravili vo vyššie uvedenom dôkaze.

Všimnime si prvý dom v obci. Ten musí mať určite vo svojom okolí nejakú zastávku. Kam ju umiestniť? Globálne najlepšie riešenie nestratíme, ak ju dáme najďalej ako to ide — teda 500 metrov za prvý dom. Totiž ak by sme ju posunuli z tejto polohy kamkoľvek bližšie k začiatku obce, žiadne nové domy nám do intervalu, ktorý zastávka obsluhuje, nepribudnú — pred prvým domom žiadne iné nie sú. Ale naopak, pre nejaké iné domy by už posunutá zastávka mohla byť priďaleko. Jej posunutím teda nemáme čo získať, môžeme len stratiť.

Výborne, umiestnili sme prvú zastávku, čo teraz ďalej? V tejto chvíli môžeme zabudnúť na všetky domy, ktoré táto zastávka obslúžila – akoby v obci ani neboli. A dostávame opäť presne tú istú úlohu ako na začiatku: zvyšné domy treba pokryť čo najmenej zastávkami.

Optimálne riešenie teda vieme zostrojiť tak, že stále opakujeme nasledujúce kroky:

1. Nájdeme prvý dom v obci, ktorý ešte pri sebe nemá zastávku.
2. Postavíme zastávku 500 metrov zaň.

Podľa tohto algoritmu sa dá napísať počítačový program, ktorý optimálne rozmiestnenie zastávok nájde v čase lineárne závislom od počtu domov (za predpokladu, že na vstupe ich už dostaneme usporiadané). Stačí totiž spracúvať domy v poradí, v akom by sme ich stretli pri prechode obcou a počas toho si pamätať, kde sme postavili doteraz poslednú zastávku.

Poznámka: V praxi by sa oplatilo ešte na koniec každú zastávku posunúť tak, aby bola čo najviac uprostred domov, ktoré obsluhuje. Napríklad v obci, kde sú každé dva domy viac ako kilometer od seba, určite obyvatelia viac uvítajú riešenie, kde je zastávka pri každom dome, ako riešenie, kde je zastávka 500 metrov za každým domom…

2.3 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

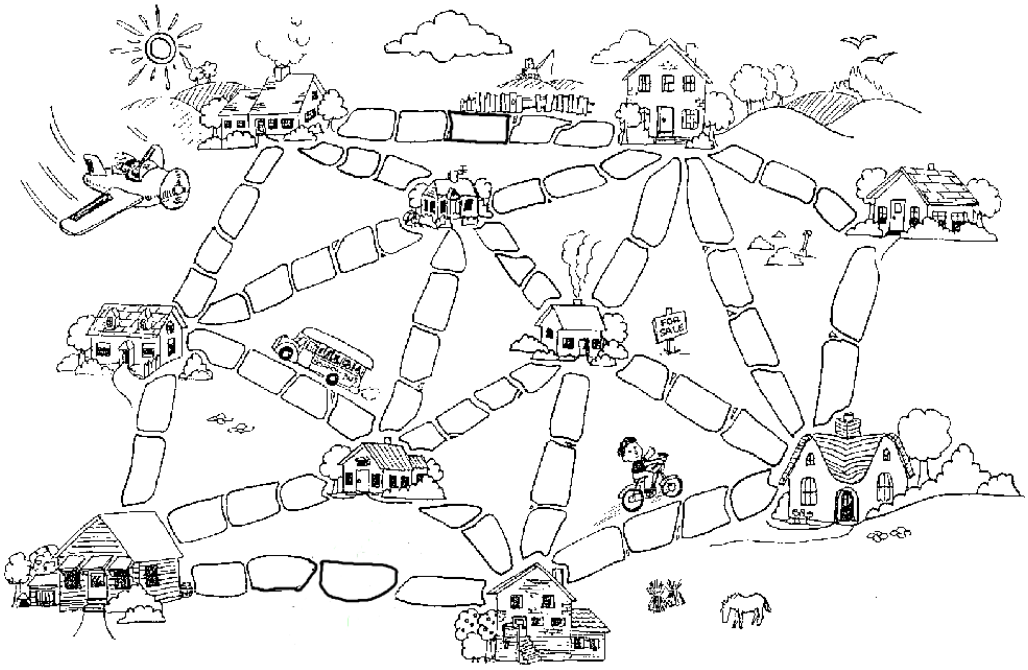
Bolo raz jedno mesto a to malo cesty nevydláždené. Len čo sa strhla búrka, všetky cesty boli zablatené. Magistrát mesta sa rozhodol s touto situáciou niečo spraviť – vydláždiť niektoré cesty. Peňazí však nikdy nie je nazvyš, a tak chcú použiť čo najmenej dlaždíc.

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.

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.



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/)

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

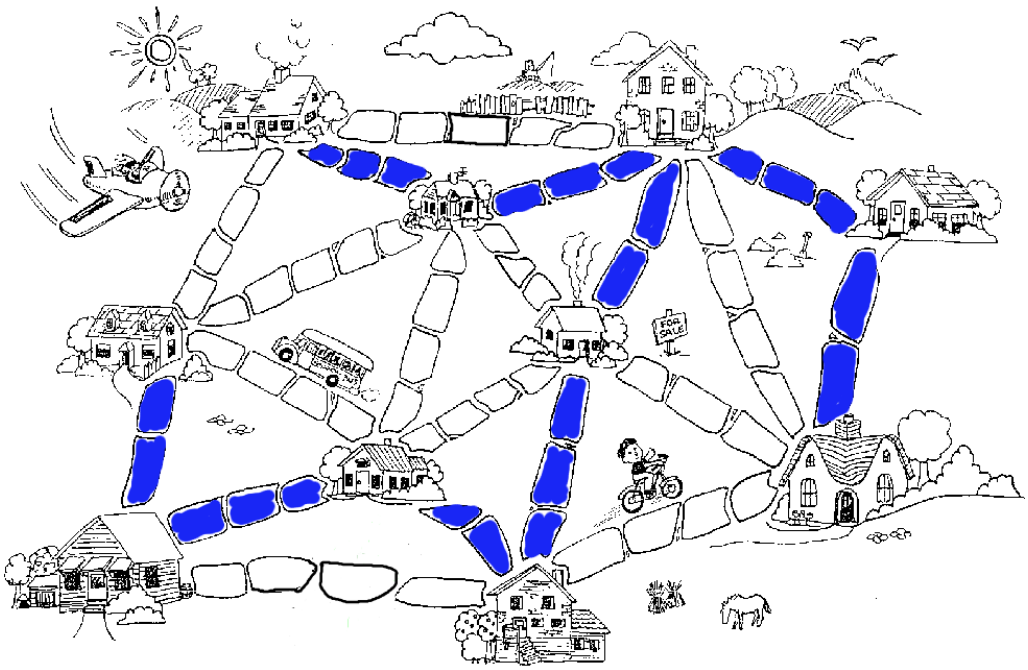
Ako však dosiahnuť najmenší počet dlaždíc? Najjednoduchšie pozorovanie, ktoré odhalia takmer všetci riešitelia: nikdy sa neoplatí postaviť cesty „do kruhu“. Samo osebe však toto pozorovanie nestačí.

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.

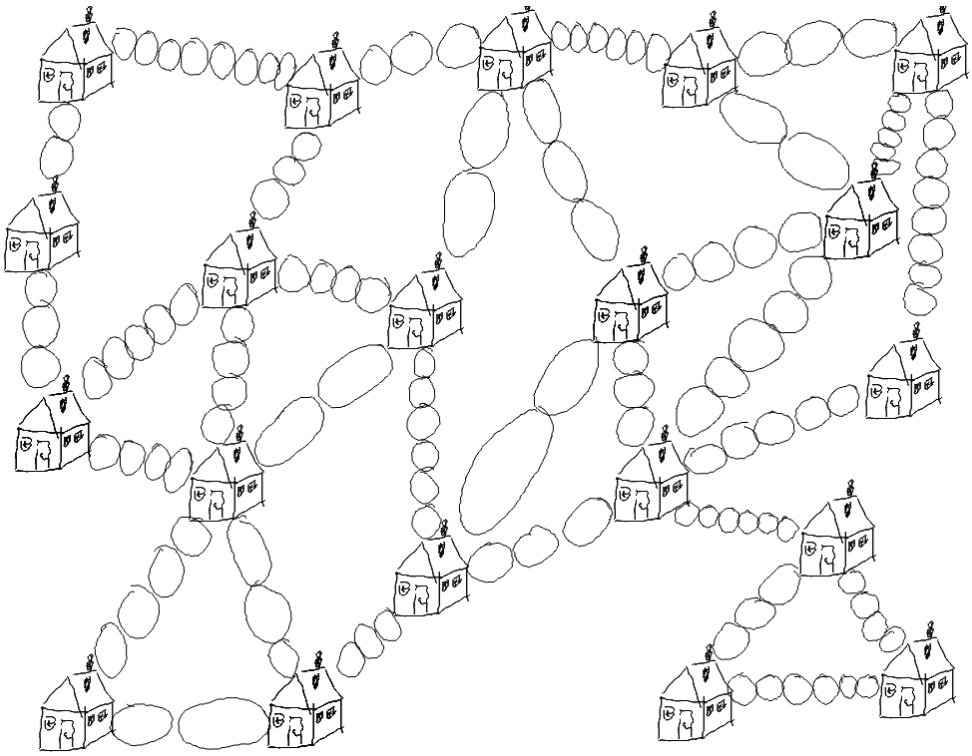
Jeden z možných výsledkov si ukážeme na obrázku.



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

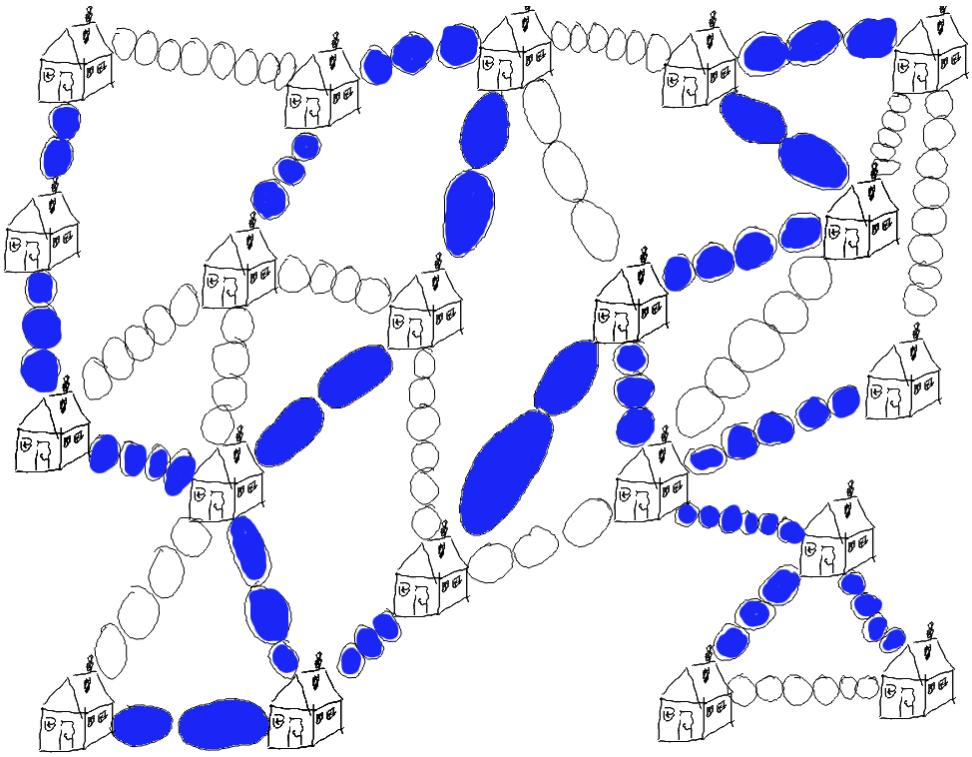
Skúsme si ešte niektorý z postupov na väčšom meste. To nám pomôže nájsť v postupe nevyriešené prípady a väčšie mesto zvýrazní potrebu použiť naozajstný algoritmus, nie iba intuitívne vyfarbovanie.

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.



Obrázok 6: Aktivita na väčšom meste

Rôzne algoritmy budú produkovať rôzne riešenia, ktoré však 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.

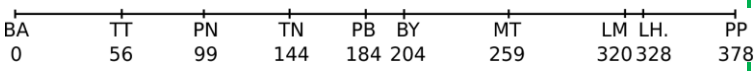
2.4 Čo to teda sú tie pažravé algoritmy?

> 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 zlate to znamenalo, že v každej situácii sme nakupovali na poslednú chvíľu, kým ešte cena klesala a predávali na poslednú chvíľu, kým cena stúpala. V úlohe o zastávkach sme sa snažili zastávky postaviť čo najneskôr, ako sme mohli. 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.

2.5 Úlohy na precvičenie

Úloha 16	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 ráz zastavovali.  Na základe postupu, ktorý ste použili, sa pokúste sformulovať všeobecný algoritmus a dokázať o ňom, že vždy nájde optimálne riešenie.
Úloha 17	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?
Úloha 18	Pri platení eurami môžeme používať pažravý algoritmus „pri platení vždy použi najväčšiu mincu, akú môžeš“. Použije tento algoritmus vždy optimálny (t. j. najmenší možný) počet mincí?
Úloha 19	Nájdite najmenšiu sumu v centoch, na ktorej zaplatenie potrebujeme použiť aspoň osem mincí.

3. Programovanie pomocou rekurzie

Rekurzia je spôsob programovania, ktorý postupne predstavíme. Nie je to metóda, ktorá zaručene vedie k efektívnym algoritmom. Je však veľa algoritmických úloh, pri ktorých rekurzia pomôže ľahko nájsť efektívne riešenie. Ba čo viac, zápis v programovacom jazyku bude pri veľkom množstve úloh oveľa prehľadnejší, ako keby sme rekurziu nepoužili.

3.1 Čo je to rekurzia

V niektorých slovníkoch je pojem rekurzia definovaný vtipne. Je v nich uvedené:

rekurzia — viď rekurzia

Môžeme to prečítať aj takto: „*Na to, aby sme zistili, čo znamená slovo rekurzia, musíme vedieť, čo znamená slovo rekurzia.*“ A presne toto rekurziu vystihuje.

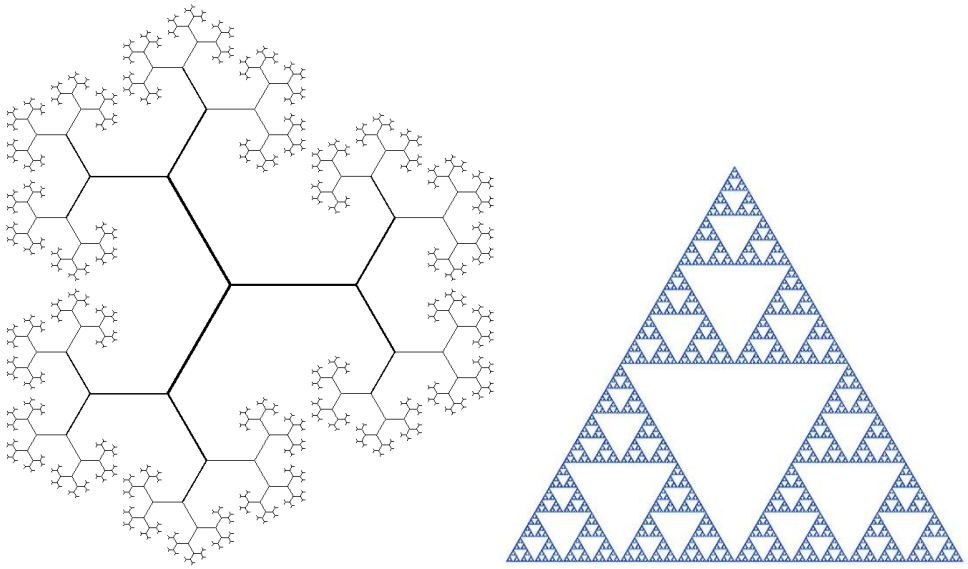
Slovo rekurzia je utvorené od latinského slova *recurrere*: bežať späť. V matematike sa často stretneme s pojmom „rekurentná definícia“. Ide o definíciu pojmu, pri ktorej už vhodným spôsobom využijeme samotný definovaný pojem.

Príkladom takejto definície je definícia toho, kto sú predkovia človeka: Medzi predkov človeka patria jeho rodičia a všetci predkovia každého z nich.

Pri programovaní používame slovo „rekurzia“ vo veľmi príbuznom význame: označujeme ním situácie, kedy funkcia alebo procedúra volá samu seba (zväčša však s inými parametrami).

3.2 Fraktály

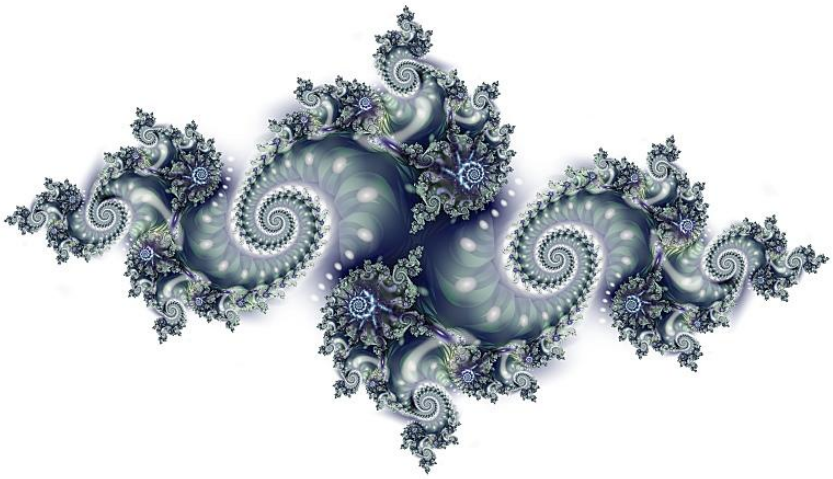
Krásnym príkladom rekurzie v matematike sú geometrické útvary nazývané fraktály. Ide o útvary, ktoré sa skladajú z viacerých menších kópií seba samých.



Obrázok 8: Fraktály – útvary skladajúce sa z kópií samého seba. Útvar vpravo sa nazýva Sierpinského trojuholník.

(Zdroje: http://mumble.net/~jar/visuals/fractal.png a

http://commons.wikimedia.org/wiki/File:Sierpinski_triangle_(blue).jpg)



Obrázok 9: Juliova množina, takisto má v sebe kópie saméj seba

(Zdroj: http://math.youngzones.org/Fractal%20webpages/Julia_set.jpg)

Mnohé objekty v prírode majú podobný charakter. Typickým príkladom je členité morské pobrežie. Častokrát nevieme presne odmerať jeho dĺžku, pretože čím presnejšie sa naň pozeráme, tým väčšiu úroveň detailu vidíme a tým dlhšie sa nám zdá.

3.3 Rekurzia v matematike

Okrem toho, že nám rekurzia pomáha popisovať pekné obrázky, ktoré sme videli v predchádzajúcej časti, je častokrát aj užitočná.

Jedným z historicky najstarších príkladov využitia rekurzie je Euklidov algoritmus na hľadanie najväčšieho spoločného deliteľa dvoch čísel.

Euklides si všimol, že najväčší spoločný deliteľ má nasledovnú vlastnosť: Prirodzené čísla A a B majú vždy rovnakého najväčšieho spoločného deliteľa ako čísla B a $A-B$. (Dôkaz je ľahký: ak nejaké D delí aj A, aj B, musí deliť aj ich rozdiel. A naopak, ak nejaké D deli aj B aj $A-B$, musí deliť aj ich súčet, čo je práve číslo A.)

Samo o sebe toto pozorovanie vyzerá natoľko triviálne, že je ťažké uveriť jeho užitočnosti. A v tom práve spočívala Euklidova genialita: uvedomil si totiž, že tento proces vie opakovať, a tým obe čísla zmenšovať až dovtedy, kým jedno z nich neklesne na nulu. V tej chvíli už je všetko jasné: najväčší spoločný deliteľ x a nuly je totiž zjavne práve číslo x.

Matematicky si tento Euklidov algoritmus teda môžeme zapísať nasledovne:

$$\mathrm{nsd}(a,b)=\begin{cases} a, & b=0 \\ \mathrm{nsd}(b,a), & a<b \\ \mathrm{nsd}(b,a-b), & \text{inak} \end{cases}$$

Úloha 20	Vypočítajte najväčší spoločný deliteľ dvoch čísel 768 a 524. Použite Euklidov algoritmus.
Riešenie	$\mathbf{nsd(768,524)} = \mathrm{nsd}(524, 768-524) = \mathrm{nsd}(524,244) =$ $\mathrm{nsd}(524-244,244) = \mathrm{nsd}(280,244) =$ $\mathrm{nsd}(244,280-244) = \mathrm{nsd}(244,36) = \cdots =$ $\mathrm{nsd}(36,28) = \mathrm{nsd}(28,8) = \cdots = \mathrm{nsd}(8,4) = \mathrm{nsd}(4,0) = \mathbf{4}$

Nezabudnite, že najväčší spoločný deliteľ je dôležitý pri úprave zlomku na základný tvar.

Existuje aj jeho krajší variant, keď si uvedomíme, že výsledkom niekoľkonásobného odčítavania toho istého čísla je vlastne zvyšok po delení. To znamená, že namiesto $\mathrm{nsd}(28,8) = \mathrm{nsd}(20,8) = \mathrm{nsd}(12,8) = \mathrm{nsd}(8,4)$ stačí zistiť zvyšok čísla 28 po delení 8. T. j. $\mathrm{nsd}(28,8) = \mathrm{nsd}(8, 28 \bmod 8) = \mathrm{nsd}(8,4)$.

Navyše, ak použijeme variant so zvyškom nemusíme sa trápiť s prehadzovaním vstupných parametrov, pretože zvyšok po delení je vždy menší, ako samotný deliteľ.

$$\text{nsd}(a,b) = \begin{cases} a, & b = 0 \\ \text{nsd}(b, a \text{ modulo } b), & inak \end{cases}$$

Úloha 21	Ručne vypočítajte najväčší spoločný deliteľ čísel 372 a 444. Použite oba uvedené varianty Euklidovho algoritmu.

3.4 Programovanie pomocou rekurzie

Programátorský zápis sa od matematického veľmi nelíši. Skúsime naprogramovať funkciu, ktorá najväčší spoločný deliteľ dvoch čísel vypočíta.

Úloha 22	Naprogramujte funkciu, ktorá vypočíta najväčší spoločný deliteľ dvoch čísel. Pomôcka: využite rekurziu.

Skúste si tiež nechať v programe vypísať, s akými parametrami sa funkcia volá. Čiže na začiatku funkcie sa vypíše `a` a `b`.

Riešenie

```
function nsd ( a, b : longint ) : longint;
begin
  if b = 0 then
    result := a
  else
    result := nsd( b, a mod b);
end;
```

Všimnime si volanie zvýraznené červenou farbou. Toto volanie je rekurzívne. Spĺňa vyššie uvedený význam pojmu rekurzia. T. j. funkcia `nsd` volá sama seba. V tomto prípade s inými vstupnými parametrami, namiesto parametrov `a` a `b` sa vypočíta hodnota pre parametre `b` a `a mod b`.

Na zamyslenie	Zdôvodnite, prečo sa funkcia `nsd` vždy zastaví, prečo jej výsledok nebude počítač počítať do nekonečna pre nejaké hodnoty `a` a `b`.

Skúste si ju naprogramovať bez rekurzie.

Samozrejme, funkciu `nsd` by sme mohli naprogramovať aj bez použitia rekurzie. Program by bol však o niečo dlhší, a hlavne menej prehľadný. V našom prípade sme len rekurzívnu definíciu prepisovali do rekurzívneho zápisu v programovacom jazyku.

Toto je jeden z dôvodov, prečo je rekurzia taká dôležitá a prečo sa používa. Je prehľadná napríklad vďaka tomu, že veľa vecí vo svete funguje rekurzívne. Má však pár problémov. Naučiť sa programovať pomocou rekurzie chce veľa tréningu a zmenu myslenia. Nie je to priamočiary program, na ktorý sme boli zvyknutí. Taktiež netreba zabúdať na podmienky, ktoré rátanie programu zastavia. V prípade funkcie `nsd` to bola podmienka, či je `b = 0`.

Kombinačné čísla

Úloha 23	Naprogramujte funkciu, ktorá pre dané `n` a `k` vypočíta kombinačné číslo $\binom{n}{k}$. Využite pritom rekurzívny matematický vzorec: $$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Samotný vzorec nám teraz presne hovorí, ako by mal vyzerať program. Skúsme si to napísať:

```
function kombinacie ( n, k : longint ) : longint;
begin
  result := kombinacie ( n-1, k-1 ) + kombinacie ( n-1, k );
end;
```

Celkom jednoduché, nie? Skoro. Tento program by sa nezastavil. Do nekonečna by sa rekurzívne volala funkcia `kombinacie`. To preto, že ani samotný matematický vzorec nám nepovedal všetky pravidlá. Chýbala v ňom informácia, kedy sa treba zastaviť.

Matematický vzorec mierne rozšírme:

$$\binom{n}{k} = \begin{cases} 1, & k = 0 \\ 1, & k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & \text{inak} \end{cases}$$

Pridali sme známu rovnosť: $\binom{n}{0} = 1$ a $\binom{n}{n} = 1$. Tieto dve podmienky zariadia zastavenie výpočtu. Pridajme tieto podmienky aj do programu.

Riešenie

```
function kombinacie(n, k: longint): longint;
begin
  if k = 0 then
    result := 1
  else
    if k = n then
      result := 1
    else
      result := kombinacie( n-1, k-1 ) +
                kombinacie( n-1, k );
end;
```

Úloha 24

Znázornite na papieri výpočet kombinačného čísla $\binom{5}{3}$ v predchádzajúcom programe.

Na tomto príklade sme si uvedomili, aké dôležité sú podmienky pre zastavenie behu rekurzívnej funkcie. A nielen to.

Porozmýšľajme, ako by sme takýto výpočet naprogramovali bez použitia rekurzie. Pravdepodobne vymyslíte spôsob, ktorý sa naučíme v kapitole o dynamickom programovaní. Asi však všetci uznáme, že vyššie uvedené riešenie bolo veľmi prehľadné a nie je ťažké ho naprogramovať.

Fibonacciho čísla

V prírode sa často vyskytujú čísla, ktoré sa dajú ľahko rekurzívne popísať. Veď predsa príroda sa mení rozmnožovaním, takže z niekoľkých generácií vzniká nová generácia. Najznámejšie takéto čísla sú Fibonacciho čísla. Ich rekurzívny vzorec vyzerá nasledovne:

$$F_0 = 0, F_1 = 1$$

$$F_{n+2} = F_{n+1} + F_n$$

Číslo F_n vyjadruje hodnotu n. Fibonacciho čísla. Začnime počítať jednotlivé hodnoty. Z definície vieme rovno prvé dve hodnoty, takže $F_0 = 0$ a $F_1 = 1$. $F_2 = F_1 + F_0 = 1 + 0 = 1$. $F_3 = F_2 + F_1 = 1 + 1 = 2$. Ak budeme ďalej počítať, zistíme, že postupnosť vyzerá takto: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$. Keďže sme programátori, uľahčíme si prácu a necháme Fibonacciho čísla rátať počítač.

Fibonacciho čísla sú pomenované podľa Leonarda z Pisy, ktorý mal prezývku Fibonacci. Pomocou Fibonacciho postupnosti popísal rast populácie zajacov v ideálnom svete, kde zajace neumierajú a každá dospelá zajačica má s dospelým zajacom dve deti - jedného samca a jednu samicu. Samotná postupnosť však bola objavená skôr, prečítajte si viac napríklad na wikipedii. http://en.wikipedia.org/wiki/Fibonacci_number

Úloha 25

Naprogramujte funkciu, ktorá pre zadané číslo n vypočíta n. Fibonacciho číslo F_n.

Riešenie

```
function fibonacci(n: longint): longint;
begin
  if n = 0 then
    result := 0
  else
    if n = 1 then
      result := 1
    else
      result := fibonacci( n-1 ) +
                fibonacci( n-2 );

end;
```

Teraz, keď sme sa už naučili programovať rekurzívne funkcie, táto úloha išla priamočiaro. Matematický vzorec sme iba prepísali do programovacieho jazyka. Aby sme sa toľko netešili, pozrime sa na časovú zložitosť tohto algoritmu.

Ak budeme dôslední v počítaní časovej zložitosti, zistíme, že je to presne $O(F_n)$. Zaujímavé, nie? To je rádovo $O((\frac{1+\sqrt{5}}{2})^n)$

Spomeňte si na tabuľku 1, pre aké najväčšie n zbehne program do hodiny.

Všimnime si, že funkcia `fibonacci(n)` volá `fibonacci(n-1)` a `fibonacci(n-2)`. Funkcia `fibonacci(n-1)` následne volá znova funkciu `fibonacci(n-2)` a ešte `fibonacci(n-3)`. Funkcia `fibonacci(n-2)` sa teda bude počítať dvakrát, aj keď by nemusela. Funkcia `fibonacci(n-3)` sa bude počítať trikrát a `fibonacci(n-4)` dokonca päťkrát. Počet volaní bude veľmi rýchlo narastať. Časová zložitosť je dokonca až exponenciálna funkcia.

V tomto príklade použitie rekurzie pri programovaní nám pomohlo vytvoriť prehľadný a priamočiaro naprogramovateľný program. Bolo to však za cenu veľmi zlej časovej zložitosti. To vôbec nevadí, pretože v kapitole o dynamickom programovaní sa naučíme z podobných časovo zložitých riešení vytvárať efektívne algoritmy.

3.5 Kedy je lepšie rekurziu použiť

V predchádzajúcich príkladoch nebolo nevyhnutné programovať pomocou rekurzie. Namiesto rekurzie stačilo vhodne použiť cyklus. Ukážeme typy príkladov, kedy je rekurzia potrebná.

Vyhodnocovanie matematických výrazov

Úloha 26

Ako vyhodnotiť matematický výraz skladajúci sa iba z čísel, zátvoriek a operácií plus a krát? Skúste vyhodnotiť výraz:

$$\big(5 + 2 \cdot (7 + 9)\big) \cdot 2 \cdot (4 + 13) + (4 \cdot 2 + 11 \cdot (1 + 1))$$

Kde sa v tejto úlohe nachádza rekurzia?

Ak začneme výraz vyhodnocovať, zistíme, že zátvorkami sa vnárame do ďalšej vrstvy. Až keď vnútro zátvorky vyčíslime, môžeme výsledok použiť. A takto vlastne funguje aj rekurzia, pretože vo vnútri zátvorky sa znova nachádza matematický výraz spĺňajúci zadanie, ba čo viac, je kratší, takže po niekoľko vnoreniach sa dostaneme k výrazu bez zátvoriek, ktorý bude ľahké vyhodnotiť.

Spracovanie vstupu, keď sa v ňom snažíme rozpoznať prvky (napr. čísla) sa tiež nazýva *parsovanie vstupu*.

Riešenie úlohy môžeme naprogramovať. Nie je to však ľahké, pretože keď to skúsime, pravdepodobne narazíme na problém so spracovaním vstupu. Treba pracovať so zátvorkami, číslami, operátormi a tieto operácie aj vykonávať.

```
function vyhodnot (zaciatok, dlzka : longint) : longint;
var vysledok, aktualnySucin, cislo, kde, kam : longint;
    operacia : char;
begin
   vysledok := 0;
   aktualnySucin := 0;
   operacia := '+';
   kde := zaciatok;
```

```
   while kde < zaciatok+dlzka do begin
      { zistíme nasledujúcu hodnotu }
      if vyraz[kde]='(' then begin
         { našli sme uzátvorkovanú časť výrazu,
           rekurzívnym volaním zistíme jej hodnotu }
         kam := najdiKoniec(kde);
         cislo := vyhodnot(kde+1,kam-kde-1);
         kde := kam+1;
      end else begin
         { našli sme číslo, načítame ho }
         cislo := nacitajCislo(kde);
      end;

      { spracujeme ju podľa operátora pred ňou }
      if operacia='+' then begin
         vysledok := vysledok + aktualnySucin;
         aktualnySucin := cislo;
      end else begin
         aktualnySucin := aktualnySucin * cislo;
      end;

      { zistíme nasledujúci operátor }
      if kde < zaciatok+dlzka then begin
         operacia := vyraz[kde];
         inc(kde);
      end;
   end;
   result := vysledok + aktualnySucin;
end;
```

Výsledok funkcie `najdiKoniec(kde)` je pozícia pravej zátvorky prislúchajúcej k ľavej zátvorke nachádzajúcej sa na pozícii `kde`.

Výsledok funkcie `nacitajCislo(kde)` je číslo začínajúce na pozícii `kde`. Treba si uvedomiť, že toto číslo môže mať aj viac cifier, takže jeho načítanie uprostred zátvoriek a operátorov nie je také jednoduché.

V reťazci `vyraz` je uložený matematický výraz, ktorý sa snažíme vyhodnotiť. Funkcia `vyhodnot` má ako vstupné parametre začiatok a dĺžku reťazca v reťazci `vyraz`, pre ktoré má výraz vyhodnotiť. Prednastavená operácia je `+` a počiatočná hodnota `0`.

Ak funkcia narazí na ľavú zátvorku, vyhľadá k nej prislúchajúcu pravú zátvorku a zavolá sa rekurzívne na podreťazec medzi zátvorkami. Keď sa volaná funkcia vyhodnotí, výsledok sa použije tak, akoby zátvorka neexistovala a namiesto nej by bolo na vstupe obyčajné číslo.

Ak funkcia narazí na operátor (`+` alebo `*`), zapamätané čísla spracuje.

Takto funkcia postupne spracuje všetky znaky na vstupe, teda v danej časti reťazca.

Ak by sme podobnú funkciu chceli programovať bez rekurzie, bolo by to omnoho nepríjemnejšie a menej prehľadné. Výhodou rekurzívneho programu je, že jeho výpočet priamo zodpovedá tomu, ako by sme tento výraz vyhodnocovali my ručne.

Rekurzia pri skúšaní všetkých možností

Pri riešení ťažkých problémov je mnohokrát jedinou cestou, ako nájsť najlepšie možné riešenie, hrubá sila: treba prezrieť všetky možné riešenia a vybrať to najlepšie z nich.

Príkladom takejto ťažkej úlohy je tzv. **Problém obchodného cestujúceho**.

Obchodný cestujúci, nazvime ho pán *Hamilton*, plánuje navštíviť krajinu, kde je niekoľko miest. Rád by spravil okružnú cestu, pri ktorej každé z miest práve raz navštívi a predá tam časť svojho tovaru. Pre každú dvojicu miest vie, koľko by ho stála priama cesta medzi nimi.

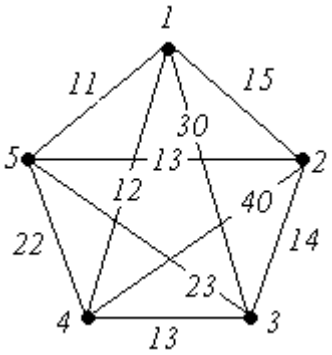
Obrázok 10: Hamiltonova kružnica spájajúca 15 najväčších miest v Nemecku

(Zdroj: http://commons.wikimedia.org/wiki/File:TSP_Deutschland_3.png)

Na obrázku je načrtnutá najkratšia okružná cesta, ak chceme navštíviť 15 najväčších nemeckých miest. Za zjednodušujúceho predpokladu, že cena cesty je priamo úmerná vzdialenosti, by toto bola optimálna trasa pre pána Hamiltona. Myslíte, že je ju ľahké nájsť medzi približne 43 miliardami iných možností?

Úloha 27

Nájdite najlacnejšiu okružnú cestu pre päť miest z obrázka.



(Zdroj: http://wps.prenhall.com/wps/media/objects/922/944272/nna1.gif)

Ako teda úlohu pána Hamiltona algoritmicky riešiť? Tak, že vyskúšame všetky možné okružné cesty. Očíslujme si mestá, ktoré má navštíviť, číslami 1 až N. Keďže jeho cesta je okružná, môžeme bez ujmy na všeobecnosti predpokladať, že ju bude začínať aj končiť v meste N. Aby sme teda zostrojili plán cesty, musíme povedať, v akom poradí má navštíviť mestá 1 až $N-1$. Inými slovami, každý plán cesty zodpovedá nejakej permutácii čísel 1 až $N-1$.

Tým sme našu úlohu previedli na jednoduchú kombinatorickú úlohu: pre dané číslo X potrebujeme vygenerovať všetky permutácie množiny $\{1, 2, \ldots, X\}$.

Ako v takejto úlohe využiť rekurziu? Jednoducho: permutácie `X` prvkov budeme vyrábať z permutácií `X-1` prvkov. Veď predsa každú permutáciu prvkov `1` až `X` vieme zostrojiť tak, že do vhodnej permutácie prvkov `1` až `X-1` na vhodné miesto vložíme prvok `X`.

Na základe tohto pozorovania môžeme teraz navrhnúť rekurzívny algoritmus:

`zoznam_permutácii(x)`:

- ak `x=1`:
  - výsledok je zoznam obsahujúci jedinú permutáciu `(1)`
- inak:
  - nech `Z` je `zoznam_permutácii(x-1)`
  - nech `V` je prázdny zoznam
  - pre každú permutáciu `z` v zozname `Z`:
    - postupne vyrob `x` nových permutácií tak, že do permutácie `z` na všetky možné miesta vložíš prvok `x`; každú novú permutáciu pridaj do zoznamu `V`
  - výsledok je `V`

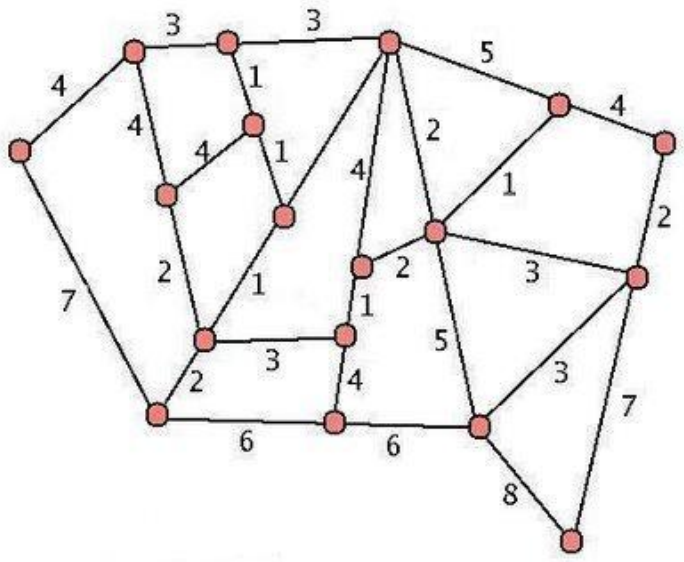
Pre `x=2` dostávame zoznam obsahujúci permutácie `(2,1)` a `(1,2)`.

Pre `x=3` prebehne tento algoritmus nasledovne:

- rekurzívnym volaním zistíme, že `Z` obsahuje permutácie `(2,1)` a `(1,2)`
- z permutácie `z=(2,1)` vyrobíme nové permutácie `(3,2,1)`, `(2,3,1)` a `(2,1,3)`
- z permutácie `z=(1,2)` vyrobíme nové permutácie `(3,1,2)`, `(1,3,2)` a `(1,2,3)`

Úloha 28	Ručne odsimulujte priebeh tohto algoritmu pre `x=4`.

V praxi však vieme častokrát úlohu pána Hamiltona riešiť aj lepšie. Prečo? Jednoducho preto, že niektoré priame spojenia vôbec neexistujú. Všimnime si napríklad cestnú sieť z nasledovného obrázka:



Obrázok 11: Príklad cestnej siete

Keby sme v nej chceli nájsť najlacnejšiu okružnú cestu, bolo by riešenie generujúce všetky permutácie prakticky nepoužiteľné: máme 18 miest, teda by bolo potrebné vyskúšať $17!$ permutácií. A to je približne tristo biliónov ($3{,}55 \cdot 10^{14}$) možností. Lenže ľahko nahliadneme, že generovať $17!$ permutácií je neefektívne: drvivá väčšina z nich bude zodpovedať okružným cestám, ktoré vôbec nevieme spraviť, lebo také cesty v našej cestnej sieti neexistujú.

Omnoho lepšie bude generovať len tie permutácie, ktoré zodpovedajú naozaj existujúcim cestám. A pri snahe vymyslieť algoritmus, ktorý bude mať túto vlastnosť, nám opäť príde na pomoc rekurzia. Technika, ktorú použijeme, sa nazýva prehľadávanie s návratom (po anglicky backtracking). Myšlienka je jednoduchá: cestu generujeme postupne, krok za krokom. Ak máme niekedy viac možností na výber, postupne vyskúšame každú z nich. A ak niekedy nemáme na výber žiadnu

možnosť, okamžite sa vrátime späť k predchádzajúcemu rozhodnutiu a tam sa vyskúšame rozhodnúť ináč.

Algoritmus riešiaci úlohu pána Hamiltona pomocou prehľadávania s návratom by mohol schématicky vyzerať nasledovne:

```
vyskúšaj_všetky_cesty(C):
```
{ `C` je cesta, ktorou som sem prišiel — teda zoznam všetkých už navštívených miest }

- ak už `C` obsahuje všetky mestá:
  - vypíš cestu `C`
- inak:
  - nech `m` je posledné mesto v `C`
  - pre každé mesto `x` susediace s mestom `m`:
    - ak `x` ešte nie je v ceste `C`:
      - `vyskúšaj_všetky_cesty(C+x);`
        { `C+x` je cesta, ktorá vznikne z `C` pridaním `x` na koniec }

Toto riešenie je ešte stále veľmi neefektívne. Pre husté cestné siete bude jeho časová zložitosť rásť exponenciálne závisle od počtu miest. Avšak oproti skúšaniu úplne všetkých permutácií sme si výrazne polepšili.

Na podobnom prístupe (prehľadávaní s návratom) je založené riešenie mnohých praktických problémov. Navyše do veľmi podobnej kategórie môžeme zaradiť aj programy hrajúce logické hry, ako je napríklad šach. Tie taktiež rekurzívne prehľadávajú a vyhodnocujú všetky možné priebehy nasledujúcich ťahov partie.

3.6 Čo sme sa naučili

Zistili sme, že rekurzia je výborným nástrojom ako v matematike, tak aj v programovaní. V matematike nám rekurzia umožňuje ľahko definovať niektoré typy objektov so zložitou štruktúrou, napríklad aritmetické výrazy. V programovaní rekurzia získava aj nový rozmer: rekurzívne volanie funkcie priamo zodpovedá opakovanej aplikácií toho istého algoritmu na nový (zväčša jednoduchší) podproblém. Príkladom takejto situácie je práve vyhodnocovanie spomínaných aritmetických výrazov: výraz rozdelíme na jednotlivé časti, pomocou rekurzívnych volaní každú z častí vyhodnotíme a z takto získaných čiastočných výsledkov následne vypočítame výsledok.

Ukázali sme si tiež, ako pomocou rekurzie implementovať techniku prehľadávania s návratom, ktorá slúži na šikovné skúšanie všetkých potenciálnych riešení daného problému.

4. 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, skúsme si vyriešiť nasledujúcu úlohu.

4.1 Úloha o loďke

Na brehu rieky stojí skupinka ľudí, ktorá sa chce dostať na druhú stranu. Cez rieku však nevedie žiaden most. Je tu len starý prievozník so svojou ešte staršou loďou. Všetci sa do nej naraz rozhodne nezmestia. A ani na dvakrát to nemusí byť také jednoduché.

Úloha 29	Dané sú hmotnosti všetkých ľudí $m_1, \ldots, m_n$ a nosnosť loďky L (všetko sú celé čísla v kilogramoch). Zistite, či sa dá na dve jazdy previezť všetkých ľudí.

Ľahko nájdeme nutnú podmienku toho, aby sa dalo všetkých ľudí previezť: musí nám stačiť kapacita. Ak je súčet všetkých m_i väčší ako $2 \cdot L$, určite máme smolu.

Táto podmienka ale nie je postačujúca. Ak napríklad máme 7 ľudí, z ktorých každý váži 100 kg, tak nám veru loďka s nosnosťou 350 kg stačiť nebude.

Po skúsenostiach s návrhom pažravých algoritmov nás môže napadnúť, že aj tu by sa mohla dať použiť takáto stratégia. Nebude to však bohužiaľ tak — žiaden funkčný pažravý algoritmus, ktorý by túto úlohu riešil, nie je známy a veľmi pravdepodobne ani neexistuje.

Úloha 30	Prievozník navrhol nasledujúcu pažravú stratégiu: Vždy, keď nakladá loďku, musia ľudia nastupovať po jednom, pričom vždy nastúpi najťažší z tých, ktorí sa na loď ešte zmestia. Nájdite konkrétny prípad (t. j. nosnosť loďky a hmotnosti ľudí), ktorý má riešenie, ale prievozníkov postup ho nenájde.

Príkladom takejto situácie je loďka s nosnosťou 180 kg a ľudia s hmotnosťami 100, 60, 60, 60, 40 a 40 kg. Prievozník by najskôr naložil človeka vážiaceho 100 kg, potom človeka vážiaceho 60 kg, a už by sa mu do loďky nik nezmestil. A na brehu mu zostali ľudia vážiaci dokopy 200 kg, v druhej jazde už teda nemá šancu ich zobrať.

Existuje však riešenie: prvýkrát musí zobrať ľudí vážiacich $(100 + 40 + 40)$ kg, druhýkrát zvyšných troch.

Keď teda pažravé riešenie nefunguje, musíme sa zamyslieť nad iným prístupom. Avšak skôr, než budeme čítať ďalej, skúsme si ručne vyriešiť nasledujúce zadanie: Nosnosť loďky je 507 kg, váhy ľudí v kg sú 23, 47, 59, 88, 91, 100, 111, 133, 157 a 205.

Súčet hmotností všetkých desiatich ľudí zo zadania je presne 1014 kg, nemáme teda žiadnu voľnosť pri ich ukladaní do loďky. Musíme nájsť rozdelenie na dve skupiny, z ktorých každá bude dokopy vážiť presne 507 kg.

Už aj pre desať ľudí sa ukazuje, že vôbec nejde o ľahkú úlohu. Pri jej riešení je ťažké nájsť nejaký efektívny postup, ostávame viac-menej odkázaní na postupné skúšanie možností. Ak vytrváme, zistíme, že sa ľudí previezť skutočne dá. Dokonca existujú dve rôzne riešenia. Pri jednom idú naraz v loďke ľudia vážiaci 205, 100, 91, 88 a 23, pri druhom sú to 205, 111, 100 a 91.

Prvý všeobecný algoritmus, ktorý bude riešiť túto úlohu, môžeme sformulovať nasledovne: Budeme postupne skúšať všetky možné zloženia posádky loďky počas

prvej jazdy a zakaždým overíme, či majú aj ostatní ľudia dokopy dostatočne malú hmotnosť.

Do loďky môžeme skúsiť posadiť ľubovoľnú z 2^n podmnožín množiny všetkých ľudí. Časová zložitosť tohto algoritmu bude teda až exponenciálna od počtu ľudí.

Príliš sme si takýmto algoritmom nepomohli. Takéto riešenie je síce ešte použiteľné povedzme pre $N = 30$, ale $N = 50$ už je pre vhodne malé hmotnosti ľudí ďaleko za hranicou našich možností.

4.2 Memoizácia

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

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

Predstavme si napríklad, že máme dvoch ľudí, ktorí vážia rovnako. Nazvime si ich A a B. Pozerajme sa teraz, ako prebieha skúšanie všetkých $2 \cdot N$ možných výberov, koho poslať prvou loďkou. Všetky možné rozdelenia si môžeme rozdeliť do štyroch tried:

1. Aj A, aj B idú prvou loďkou.
2. A ide prvou loďkou, B ostane čakať na brehu.
3. B ide prvou loďkou, A ostane čakať na brehu.
4. Obaja čakajú na druhú loďku.

Čo tu počítame zbytočne dvakrát? No predsa možnosti 2 a 3. Z nášho pohľadu je úplne jedno, kto z nich ostane a kto pôjde, v oboch prípadoch dostaneme rovnakú situáciu.

Pred tým, ako vyššie uvedené pozorovanie zovšeobecníme, skúsme si poriadnejšie sformulovať úlohu, ktorú budeme riešiť. Naším cieľom teraz bude nájsť spôsob, ako vyplniť loďku tak, aby v nej ostalo čo najmenej nevyužitej nosnosti — inými slovami, aby posádka dokopy vážila čo najviac. Keď tento spôsob nájdeme, už vieme vyriešiť pôvodnú úlohu: stačí sa pozrieť, či sa aj všetci ostatní ľudia spolu do loďky zmestia.

Všimnime si poriadnejšie, čo sa deje pri našom skúšaní možností. Na začiatku stojíme pred otázkou: „Ako najlepšie viem vyplniť L kg voľného miesta v loďke pomocou ľudí s číslami 1 až N?“

Uvažujme človeka číslo 1 s hmotnosťou m_1. Máme dve možnosti: buď mu povieme, že bude čakať, alebo ho posadíme do loďky.

- Ak bude prvý človek čakať na brehu, stojíme pred novou úlohou: „Ako najlepšie viem vyplniť L kg voľného miesta v loďke pomocou ľudí s číslami 2 až N?“
- Ak ho pošleme do loďky, nová otázka bude znieť: „Ako najlepšie viem vyplniť $(L - m_1)$ kg voľného miesta v loďke pomocou ľudí s číslami 2 až N?“

Deje sa tu teda niečo podobné ako pri pažravých algoritmoch: 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 loďky H ešte máme nevyužitú, a najmenšie číslo K človeka, pre ktorého ešte máme na výber, či skončí v loďke, alebo nie.

Označenie $Z(M, K)$ bude predstavovať najmenšie množstvo voľného miesta, ktoré môže byť na konci v lodi nosnosti M, ak do nej umiestňujeme niektorých ľudí spomedzi ľudí s číslami K až N. Platí:

$$Z(M,K) = \begin{cases} M, & K > N \\ Z(M, K+1), & m_K > M \\ \min\left(Z(M,K+1), Z(M-m_K, K+1)\right), & \text{inak} \end{cases}$$

Vysvetlíme slovne jednotlivé možnosti:

- Ak $K > N$, nezostali nám už na výber žiadni ľudia, všetko voľné miesto, ktoré ešte máme, teda zostalo nevyužité.
- V opačnom prípade sa pozrime na človeka s číslom K. Ten má hmotnosť m_K. Ak platí $m_K > M$, tento človek sa už do lode nezmestí, a teda nemáme na výber: musíme ho nechať čakať.
  Zostáva nám teda nosnosť M, ktorú už môžu vyplniť len ľudia s číslami $K+1$ až N . V tomto prípade teda platí $Z(M,K) = Z(M,K+1)$.
- Zostáva nám jediný prípad, kedy sa potrebujeme rozhodnúť: vezmeme K. človeka, alebo nie?
  - Ak ho nevezmeme, tak rovnako ako v predchádzajúcej možnosti dostávame $Z(M,K) = Z(M,K+1)$.
  - Ak ho vezmeme, budeme mať o jeho hmotnosť nižšiu nosnosť. Spomedzi zvyšných ľudí už zvládneme uniesť len $(M - m_K)$ kilogramov. Najlepšie riešenie pre zvyšných ľudí je teda rovné $Z(M - m_K, K+1)$.

  No a keďže chceme čo najmenej nevyužitého miesta, z týchto dvoch možností si vyberieme tú menšiu. $Z(M,K)$ je preto rovné minimu z uvedených dvoch hodnôt.

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

```
function Z ( M, K : longint ) : longint;
var vezme, nevezme : longint;
begin
  if K > N then
    result := M
  else begin
    nevezme := Z(M,K+1);
    if hmotnost[K] > M then
      result := nevezme
    else begin
      vezme := Z( M-hmotnost[K], K+1 );
      if vezme < nevezme then
        result := vezme
      else
        result := nevezme;
    end;
  end;
end;
```

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

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

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

```
      result := nevezme
    else begin
      vezme := Z( M-hmotnost[K], K+1 );
      if vezme < nevezme then
        result := vezme
      else
        result := nevezme;
    end;
  end;
  spocital[M,K] := true;
  vysledok[M,K] := result;
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. Hodnotou, ktorá nás zaujíma, je výstup funkcie $Z(L,1)$: potrebujeme vyplniť čo najviac z nosnosti prázdnej loďky L a k dispozícii máme všetkých ľudí.

Pri počítaní tejto hodnoty zjavne budeme potrebovať len hodnoty $Z(M,K)$ pre $0 \leq M \leq L$ a $1 \leq K \leq N$ — totiž v priebehu skúšania aj naša nosnosť, aj počet ľudí, ktorých sme ešte nerozdelili, len klesajú. Týchto hodnôt je rádovo $L \cdot N$. No a ľubovoľnú konkrétnu z nich vieme vypočítať z iných hodnôt v konštantnom čase. Dokopy je teda časová zložitosť tohto algoritmu priamo úmerná hodnote $L \cdot N$.

Všimnime si, že sme dosiahli skutočne výrazné zlepšenie. Na rozdiel od pôvodného riešenia, ktoré bolo nepoužiteľné už pre 50 ľudí, môžeme toto riešenie pokojne použiť napr. pre 1000 ľudí a loď s nosnosťou niekoľko ton!

4.3 Dynamické programovanie

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

$$Z(M,K) = \begin{cases} M, & K > N \\ Z(M,K+1), & m_K > M \\ \min\left(Z(M,K+1), Z(M-m_K,K+1)\right), & \text{inak} \end{cases}$$

Všimnime si, že každá hodnota $Z(M,K)$ závisí len na nejakých hodnotách $Z(\text{niečo}, K+1)$ - inými slovami, na riešeniach situácií s menším počtom predmetov.

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

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

Následne, keď už poznáme hodnoty $Z(M,N)$ pre všetky M, vieme pomocou nich vypočítať hodnoty $Z(M,N-1)$ pre všetky M. A tak ďalej, až kým sa nedopracujeme k želanej hodnote $Z(L,1)$.

V programe by toto riešenie mohlo vyzerať napríklad nasledovne:

```
var Z : array [0..L, 1..N+1] of longint;
    M, K, vezme, nevezme : longint;
begin
  for M := 0 to L do
    Z[M,N+1] := 0;
  for K := N downto 1 do
    for M := 0 to L do begin
      nevezme := Z[M,K+1];
      if hmotnost[K] > M then
        Z[M,K] := nevezme
      else begin
```

```
        vezme := Z[ M-hmotnost[K], K+1 ];
        if vezme < nevezme then
          Z[M,K] := vezme
        else
          Z[M,K] := nevezme;
      end;
  end;
  writeln('Riesenie je ', Z[L,1] );
end;
```

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

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

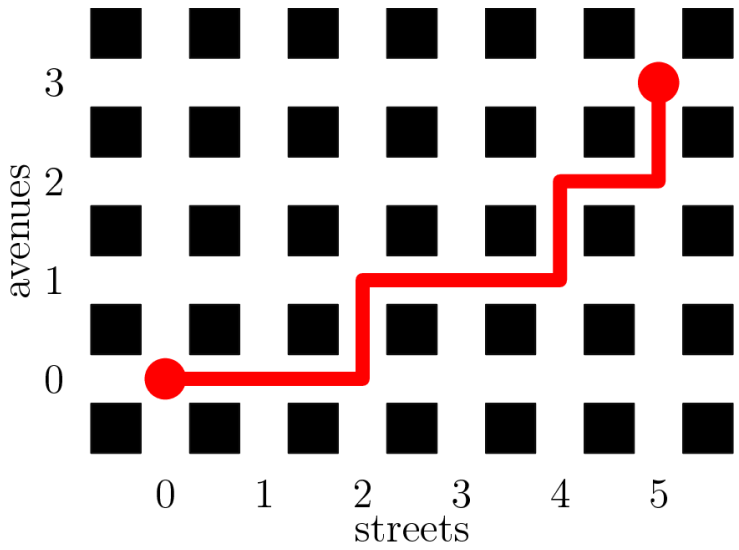
Vidíme teda, že dynamické programovanie a memoizácia sú dve strany jednej mince. Oba prístupy slúžia na to, aby sme všetko potrebné počítali len raz.

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.

Úloha 31	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 12: 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 o loďke 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:

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

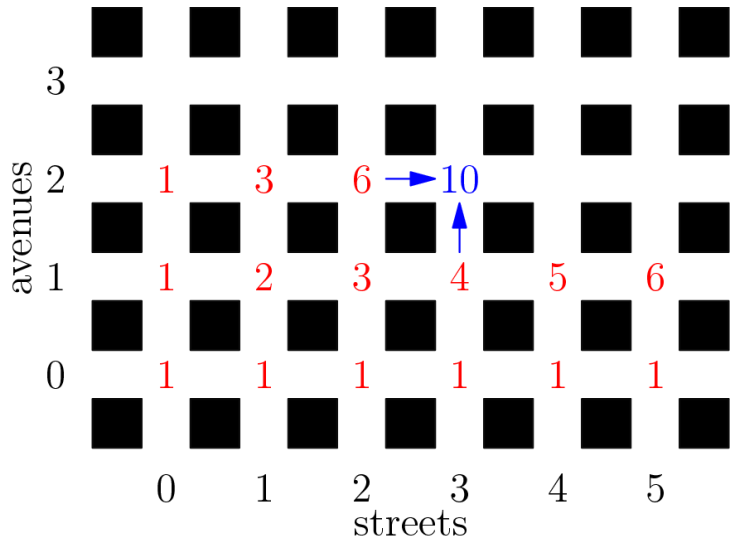
(Nesmieme ešte zabúdať na okrajové podmienky: $P(0,0)=1$ a $P(a,s)=0$, ak je a alebo s záporné.)

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

Riešenie úlohy 31

```
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.



Obrázok 13: Hodnotu aktuálneho políčka (10) vypočítame ako súčet políčok vľavo a dole (6 + 4).

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?

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

Úloha 32

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.

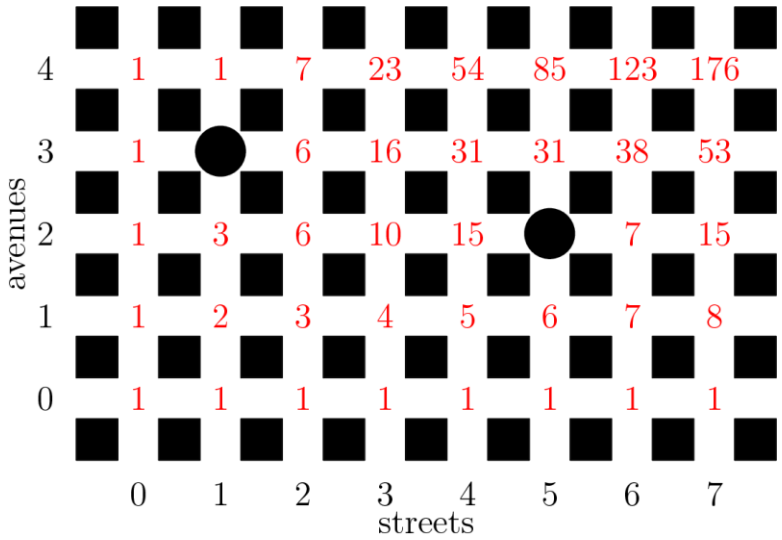
Zatiaľ čo kombinatorickou úvahou sa už táto úloha riešiť nedá, programátorský pohľad zostáva takmer nezmenený. Rovnako ako predtým si označíme $P(a,s)$ počet najkratších ciest z $(0,0)$, ktoré končia na križovatke (a,s). S tým rozdielom, že tentokrát počítame len cesty, ktoré prechádzajú len cez nezablokované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



Obrázok 14: Zistenie počtu najkratších ciest pre dve zablokované križovatky.

4.4 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.

Čo sme sa naučili v tomto module

Zhrnutie

Študijný materiál obsahuje štyri časti. V časti *efektivita algoritmov* sme sa naučili posudzovať vhodnosť výberu algoritmu na základe jeho efektivity, zhodnotiť čas, potrebný na beh algoritmu v závislosti od veľkosti vstupu a tieto údaje zapísať do zaužívanej notácie.

V kapitole *pažravé algoritmy* sme zistili, že niektoré problémy sa dajú riešiť touto prirodzenou metódou a dokázali sme argumentovať, prečo pažravá metóda vyprodukuje optimálne riešenie úlohy.

Časť *Rekurzia* nám dala programátorský nástroj na riešenie rôznych problémov, vďaka nej sme zistili, že programovať sa dá aj ináč, ako sme boli doteraz zvyknutí.

V poslednej kapitole *Dynamické programovanie* sme objavili techniku programovania, ktorá pomáha vytvárať efektívne riešenia problémov vďaka zásade „*Nikdy nepočítať nič viackrát*“.

Preverenie výstupných vedomostí

Preverenie vedomostí vykonajú lektori priamo počas výučby na základe aktivity pri riešení problémov.

Hodnotenie účastníkov bude realizované slovne — absolvoval/neabsolvoval.

Literatúra a použité zdroje

Niektoré časti tohto modulu sú spoločné s modulom [1], ktorý tematicky pokrýva väčšinu tohto modulu. Pre tento modul však bola vyvinutá nová sada úloh a príkladov, na ktorých sú tieto spoločné témy prezentované.

[1] Andrejková, G., Forišek, M., Šišková, J., Winczer, M., (2010), *Kapitoly z informatiky : Ďalšie vzdelávanie kvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ*, Štátny pedagogický ústav Bratislava, 2010

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

Autori © RNDr. Michal Forišek, PhD.
Mgr. Juliana Šišková

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

Podnázov Kapitoly z informatiky 1

Študijný materiál prešiel recenzným pokračovaním.

Recenzenti doc. RNDr. Stanislav Krajči, PhD.
RNDr. František Galčík, PhD.

Počet strán 40

Náklad 300 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 978-80-8118-071-2