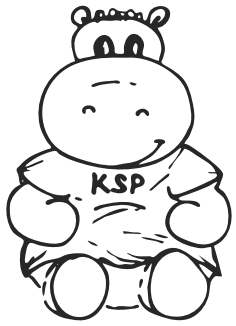
Milí řešitelé a řešitelky!



Držíte v ruce již druhý leták 24. ročníku KSP. Každá série letos obsahuje 8 úloh a z nich se 5 nejlépe vyřešených započítává do celkového bodového hodnocení.

V tomto ročníku jsou nově v každé sérii dvě lehké úlohy pro začátečníky za menší počet bodů.

Nově je také možno být přijat na MFF UK za úspěšné řešení KSP. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150.

Upozorňujeme letošní maturanty, že termín odevzdání páté série bude příliš pozdě na to, aby pátou sérií doháněli chybějící body. Diplom úspěšného řešitele ale můžeme v případě potřeby zaslat i dříve, budete-li mít dost bodů.

Termín odevzdání druhé série je stanoven na **pondělí 19. prosince** v 8:00 SELČ, což znamená, že papírové řešení byste měli podat na poštu do středy 14. prosince, aby nám stihlo přijít.

Řešení přijímáme elektronicky na stránce *https://ksp.mff.cuni.cz/submit/*. Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – zde je jeho SHA1 hash: `7F:53:E7:00:60:F2:24:93:8F:52:51:EC:1E:A8:34:54:86:69:32:7D`.

Také nám řešení můžete poslat klasickou poštou na adresu

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
118 00 Praha 1

Druhá série čtyřiadvacátého ročníku KSP

Bylo nebylo, povídali si takhle dva členové prvobytně pospolné společnosti, kolik má který ovcí. Jeden měl donedávna deset ovcí, leč před pár dny se jedné z nich narodilo jehně, a nyní má tedy ovcí jedenáct.

Onen přírůstek jedenácté ovce byl sám o sobě radostnou událostí, nicméně jejího šťastného majitele trápil drobný problém. Už nemohl jednoduše ukázat na prstech, kolik ovcí má, musel ukázat celých 10 a poznamenat k tomu, že má ještě jednu navíc.

Když si postěžoval kamarádovi, oba se zamysleli, co s tím. Po chvíli vzal jeden z nich poměrně ostrý primitivní nástroj, jenž by se dal označit jako nůž, sebral ze země delší klacík a udělal na něm 11 zářezů.

Možná tak, možná nějak jinak vznikla tato primitivní metoda počítání ovcí. Jistě se však hodila jednomu z jejich potomků, který tuhle na louce pásl stádo, když tu najednou zjistil, že v sousedním lese hoří. Navíc jediná rozumná cesta z té louky vedla právě tím lesem.

24-2-1 Požární poplach — 11 bodů

V lese hoří. Představme si les jako čtvercovou síť, na každém políčku je buďto skála (neprůchozí políčko), požár, nebo les. Požár se za jednotku času rozšíří na všechna sousední políčka, na kterých je ještě les.

Lesem je však potřeba bez úhony projít a nás zajímá, jak dlouho to ještě bude možné. Váš program dostane na vstupu nejprve rozměry lesa (R a S) a potom R řádků délky S složených ze znaků `@` (oheň), `#` (skála) a `.` (les).

```
6 6
@#...@    @#@@@@
.#.###    @#@###
......  → @@@@@@
####.#    ####@#
.....#    ....o#
####..    ####..
```

Na výstupu vypíšete, kolik jednotek času bude les ještě průchozí zleva doprava. To znamená, že z alespoň jednoho políčka na levém okraji bude existovat cesta pouze nehořícím lesem do nějakého políčka na pravém okraji. Pohyb je povolen pouze svisle a vodorovně, nikoli šikmo.

Pro zobrazený vstup je správným řešením číslo 7 – oheň se rozhoří jako na druhém obrázku. O chvíli později by již hořelo i políčko označené `o` a les by byl neprůchozí.

Jste jistě zvědavi, k čemu pasáčkovi byla ona slibovaná metoda. Jednoduše si po průchodu lesem spočítal, kolik ovcí mu zbylo a kolik ovcí uhořelo. Co jiného byste čekali?

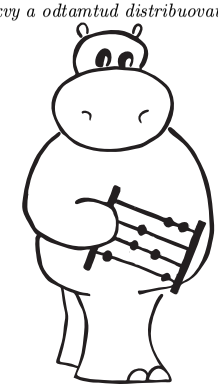
* * *

Uplynulo mnoho vody v řece, přes kterou potom pasáček převedl ovce, aby neuhořely v rozsáhlém lesním požáru, než někoho napadlo počítat třeba přesouváním kuliček na počítadle. I to je však nástroj značně dávného původu.

Každý z nás snad někdy počítal na počítadle v první třídě, občas někdo slyšel o ruském sčotu.

Dovolme si malou odbočku. V Rusku bylo ještě před nějakou dobou naprosto běžné počítat na sčotech. Byl o ně tudíž velký zájem, a tak byly nedostatkovým zbožím.

Problém byl v chybně umístěném centrálním skladu. Velení armády totiž rozhodlo (sčoty byly strategickým zbožím), že všechny vyrobené sčoty se budou svážet do centrálního skladu do Moskvy a odtamtud distribuovat po celém Rusku.



24-2-2 Centrální sklad — 9 bodů

Pro zjednodušení si představme celé Rusko jako přímku. Na přímce leží N bodů – výrobců sčotů. Nalezněte ideální místo pro centrální sklad – takové, že průměrná vzdálenost mezi centrálním skladem a výrobci bude minimální.

Na vstupu je na prvním řádku číslo N a na druhém N čísel oddělených mezerou – souřadnic výrobců. Vypište jediné číslo – souřadnici centrálního skladu. Je-li více možných řešení, vypište libovolné z nich.

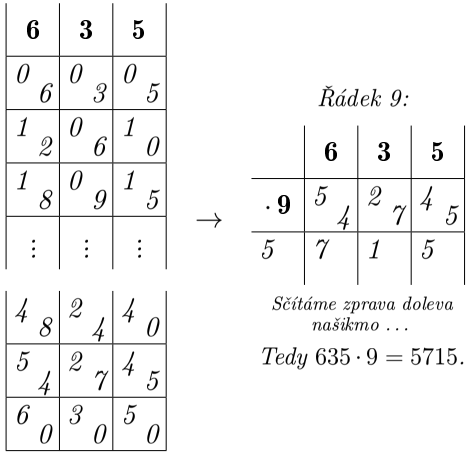
Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[1] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

Generální štáb vzápětí zjistil, že chyba byla nejen ve špatně umístěném centrálním skladu, ale i v centralizaci celého zásobování, takže sčoty byly nedostatkovým zbožím i nadále.

* * *

Zajímavým stupněm vývoje byly takzvané Napierovy kostky.[2] John Napier na přelomu 16. a 17. století vynalezl zajímavou dřevěnou pomůcku, která usnadňovala zvláště násobení dlouhého čísla jednociferným.

Pomůcka obsahovala podlouhlé hranoly, pro každé číslo od 0 do 9 jeden, na kterých byly vhodně napsané násobky těchto čísel – postupně 0-násobek až 9-násobek. Když se pak poskládaly hranoly správně k sobě, stačilo už akorát přepočítat přenosy.



Řádek 9:

	6	3	5
·9	5 4	2 7	4 5
5	7	1	5

Sčítáme zprava doleva našikmo . . .

Tedy $635 \cdot 9 = 5715$.

24-2-3 Odčítání — 7 bodů

Následující program simuluje něco jako mechanické počítadlo… tedy aspoň jej simulovat má. Jestli to je pravda, ověřte vy.

Předložená funkce má odčítat dvě čísla a vracet jejich rozdíl. Její vstup má být zadán tak, že na pozici `[0]` je číslice nejvyššího řádu.

Varianta v C bere jako první dva parametry vstup, ve třetím vrací výstup a čtvrtý určuje, kolik cifer čísla mají. Program v Pythonu bere vstup ve svých parametrech a pole vrací přímo.

Zadání je v desítkové soustavě (cifry 0–9) a čísla musí mít stejně cifer, byť by jedno z nich mělo začínat nulami.

Když tedy chcete odčítat $635 - 21$ v C, voláte

```
int p[] = {6, 3, 5};
int d[] = {0, 2, 1};
int v[3];
funkce(p, d, v, 3);
```

a v Pythonu jednoduše `funkce([6,3,5], [0,2,1])`.

V kódu nehledejte ani syntaktické chyby, ani podivný styl, ani jiné formální problémy. Vaším úkolem je dokázat nebo vyvrátit jeho správnost a v každém případě určit jeho složitost (časovou i paměťovou).

```
void funkce(int prvni[], int druhe[],
            int vysledek[], int delka) {
  int index = 0, i;
  for (i = 0; i < delka; ++ i)
    vysledek[i] = prvni[i] + 9 - druhe[i];
  vysledek[delka - 1] ++;
  while (index < delka) {
    if (vysledek[index] >= 10) {
      vysledek[index] -= 10;
      index --;
      if (index >= 0)
        vysledek[index] ++;
      else
        index ++;
    } else
      index ++;
  }
}
```

```
def funkce(prvni, druhe):
  vysledek = prvni[:]     # Kopie prvního pole
  for i in range(0, len(druhe)):
    vysledek[i] += 9 - druhe[i]
  vysledek[-1] += 1       # -1 = poslední prvek
  index = 0
  while index < len(vysledek):
    if vysledek[index] >= 10:
      vysledek[index] -= 10
      index -= 1
      if index >= 0:
        vysledek[index] += 1
      else:
        index += 1
    else:
      index += 1
  return vysledek
```

Napierovy kostky byly mimochodem v 19. století překonány ještě šílenějším vynálezem – Genaillovými-Lucasovými pravítky.[3] Tato pravítka počítala automagicky i přenosy.

Autor příběhu si pravděpodobně jednu takovou sadu pořídí a bude s ní machrovat na zkoušce z Analýzy III.

* * *

[1] http://ksp.mff.cuni.cz/zaciname/codex.html
[2] http://en.wikipedia.org/wiki/Napier%27s_bones
[3] http://en.wikipedia.org/wiki/Genaille%E2%80%93Lucas_rulers

nejvyšší řádek, ve kterém se tato hodnota také vyskytuje, v našem případě je to řádek 12. Druhé písmeno tedy budeme určovat z $D[10,7]$, třetí z $D[9,6]$, atd.

Jednou z hledaných podposloupností je tedy:

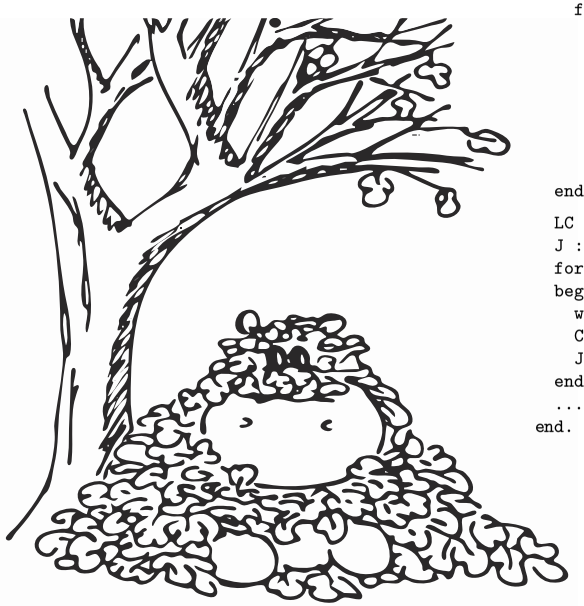
poslupnost:	2	3	1	2	2	3	1	2
indexy v A:	1	2	4	5	7	9	10	12
indexy v B:	2	5	6	7	8	9	11	12

Již zbývá jen odhadnout složitost algoritmu. Časově nejnáročnější byl vlastní výpočet hodnot v poli, který se skládá ze dvou hlavních cyklů o délce $|A|$ a $|B|$, což jsou délky posloupností A a B.

Vnořený cyklus while proběhne celkem maximálně $|A|$-krát a časovou složitost nám nezhorší. Můžeme tedy říct, že časová složitost je $\mathcal{O}(|A| \cdot |B|)$.

Posloupnosti jsme si prohodili tak, aby první byla ta kratší, protože pak je maximální délka společné podposloupnosti i počet kroků algoritmu roven délce kratší posloupnosti, a tedy i velikost pole s daty je kvadrát této délky.

Paměťovou složitost odhadneme $\mathcal{O}(N^2+M)$, kde N je délka kratší posloupnosti a M té delší.



```
program Podposloupnost;
var
  A, B, C: array[0..MaxN - 1] of Integer;
  LA, LB, LC: Integer;  { Délky posloupností }
  D: array[0..MaxN, 1..MaxN] of Integer;
  I, J, L, MaxL, T: Integer;
begin
  ...
  if LA > LB then begin { A bude kratší z obou }
    C := A;
    A := B;
    B := C;
    T := LA;
    LA := LB;
    LB := T;
  end;

  for I := 1 to LA do
    D[0, I] := LB;

  L := 0;
  MaxL := 0;
  for I := 1 to LA do begin
    for J := 1 to LA do
      D[I, J] := D[I-1, J];

    L := 0;
    for J := 0 to LB-1 do
      if B[J] = A[I-1] then begin
        while (L = 0) or (D[I-1, L] < J) do
          L:=L+1;
        if D[I, L] >= J then
          D[I, L] := J;
      end;
      if L > MaxL then MaxL := L;
  end;

  LC := MaxL;
  J := LA;
  for I := LC downto 1 do
  begin
    while D[J-1, I] = D[J, I] do J:=J-1;
    C[I-1] := A[J-1];
    J:=J-1;
  end;
  ...
end.
```

Dnešní menu servírovali
Martin Mareš a Petr Škoda

Tou dobou také začínaly vznikat první mechanické počítací stroje, obvykle na objednávku bankovních ústavů nebo zámožných obchodníků, kteří potřebovali (jak jinak) počítat peníze.

Autory těchto strojů byli například Blaise Pascal nebo Gottfried Wilhelm Leibniz. Roku 1820 pak šéf pojišťovacích společností Charles Thomas sestrojil už poměrně sofistikovaný stroj, který uměl sčítat, odčítat, násobit i dělit ve velmi rychlém čase.

Onomu stroji se říkalo Arithmometer a zvládnul vynásobit dvě osmimístná čísla za 18 sekund a na dělení potřeboval necelou půlminutu.

Byl to na svou dobu dokonalý výrobek, takže Charles Thomas založil první továrnu na výrobu počítacích strojů. Jeden její obchodní cestující prý prodal Arithmometer i na tehdejších Královských Vinohradech (součástí Prahy se staly až v roce 1922). Jak by to vypadalo dnes?

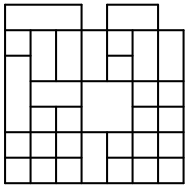
24-2-4 Odbočení vlevo — 7 bodů

Pražské Vinohrady se vyznačují tím, že na žádné křižovatce není povoleno odbočit vlevo. Vždy se smí jet jen doprava a rovně. Alespoň to tvrdí zlí jazykové.

Obchodní cestující se potřebuje dostat z jednoho místa na Vinohradech na druhé. Vaším úkolem bude najít mu nejkratší cestu, přičemž je potřeba respektovat globální zákaz odbočení vlevo.

Na vstupu dostanete mapu Vinohrad – čtvrti s pravoúhlou soustavou ulic (což je podmnožina jednotkové čtvercové mřížky); dále pak start a cíl cesty (nějaké dva úseky ulic). Výstupem vašeho algoritmu bude itinerář sestávající z příkazů typu „jeď rovně“ a „odboč vpravo“.

Jednu možnou mapu Vinohrad jsme vám zde připravili jako příklad. Nějaký start a cíl si jistě vymyslíte sami.



Počítací stroje se dále vyvíjely, v polovině 20. století bylo například možno občas potkat příruční kalkulačku Curta, která se vešla do dlaně. Dlouho ji prý používali například v rallye, a to i v době elektronických kalkulaček, které nevydržely otřesy při jízdě.

⋆⋆⋆

V první polovině 20. století začínali konstruktéři počítacích strojů pomalu opouštět plně mechanická zařízení. Vznikaly například reléové počítací stroje, nebo později elektronkové počítače.

Tehdejší počítače však zpočátku nebyly dvojkové – neměly logické obvody, ale složitější členy. Nepočítalo se v nich pouze v nulách a jedničkách, ale spojitě v napětí mezi nulou a nějakým maximem.

Pak však kohosi napadlo, že by se dalo počítat jinak než analogově – číslicově, ale ne v desítkové soustavě, jak bývalo zvykem, ale ve dvojkové. Vznikaly tedy stroje počítající ve dvojkové soustavě, průkopníkem byl například Zuse Z3.

Jiné stroje, například ENIAC, počítaly v desítkové soustavě, ale každá cifra byla kódována do 4 bitů, se kterými se počítalo dvojkově (BCD – Binary Coded Decimal).

24-2-5 Logická formule — 11 bodů

V průkopnické době digitálních přístrojů řešili vývojáři a konstruktéři různé zajímavé úlohy. Například tuto.

Mějme zadaný neuzávorkovaný logický výraz, který obsahuje pouze nuly, jedničky, AND a OR. Například

```
0 AND 1 AND 0 OR 1.
```

Nalezněte, kolika různými způsoby je možno zadaný výraz úplně uzávorkovat, aby jeho hodnota byla 1, a kolika způsoby naopak dostaneme nulu. V našem příkladu by třikrát vyšla 0 a dvakrát 1:

```
(0 AND 1) AND (0 OR 1) = 0
0 AND (1 AND (0 OR 1)) = 0
0 AND ((1 AND 0) OR 1) = 0
((0 AND 1) AND 0) OR 1 = 1
(0 AND (1 AND 0)) OR 1 = 1
```

Pak už nastoupila éra tranzistorů a šlo to ráz na ráz. Výhodou tranzistorů byla značná úspora místa. Najednou mohly počítače zabírat ne jednu velkou místnost, ale jen jednu velkou plechovou bednu. A nebo se do té velké místnosti vešlo víc výpočetního výkonu.

Tou dobou bylo prodáno okolo 10 000 kusů IBM 1401. Z toho počítače už byla největší součástí čtečka děrných štítků. . .

Navíc byly tranzistory na výrobu výrazně levnější než elektronky.

Takové stroje už byly dostatečně výkonné na to, aby dokázaly počítat všemožné zajímavé úlohy. Nebyly však ještě dostatečně výkonné na to, aby počítaly neefektivně.

24-2-6 Závorky — 13 bodů

Mějme na začátku N levých závorek. Nyní nám začnou chodit příkazy „otoč závorku na pozici i“. Po každém takovém příkazu vypíšete, jestli je teď řetězec dobře uzávorkovaný.

Dobře uzávorkovaný řetězec levých a pravých závorek splňuje podmínku, že levé a pravé závorky je možno přirozeným způsobem spárovat.

Program si na začátku může něco předpočítat – na vstupu dostanete N, což bude počet závorek v řetězci. Potom bude postupně dostávat výše definované příkazy a hned je bude zpracovávat.

Nemůžete si tedy počkat, až dostanete třeba celou posloupnost příkazů, nebo je brát po několika. Vždy přijde příkaz a vy jej zpracujete. Pak teprve přijde další příkaz atd.

Nějakou dobu vývojáři zmenšovali tranzistory, až někoho napadlo dát jich do jednoho pouzdra víc – v jednu dobu se sešly rovnou dva vynálezy integrovaných obvodů – Jack St. Clair Kinby a Robert Norton Noyce nezávisle na sobě vynalezli mikročip na konci 50. let 20. století.

Trvalo už jen pár let, než byl vynalezen mikroprocesor. V roce 1971 byl vyroben mikroprocesor Intel 4004.

V roce 1981, o celých 10 let později, pak miniaturizace dosáhla takového stavu, že bylo možno vydat IBM PC.

To byl první počítač, který se vešel na stůl a byl rozumně levný, takže jeho rozšíření nabylo značných rozměrů a firma IBM měla značné zisky.

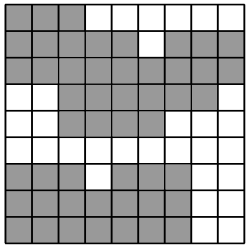
Na výsluní slávy se pomalu začal dostávat Microsoft se „svým“ MS-DOSem (slovy zlých jazyků, Messy, Slow and Dirty Operating System). . .

24-2-7 Štětcování — 10 bodů

V době vydání IBM PC bylo potřeba vyrobit propagační plakát.

Grafici dostali podivné zadání (ale není se čemu divit vzhledem k tomu, jak vypadá například instrukční sada procesoru Intel 8088...) – plakát je potřeba nakreslit co největším štětcem.

Jak se kreslí štětcem velikosti K? Vybereme si na čtvercové síti libovolný čtverec velikosti $K \times K$. Ten vybarvíme; postup opakujeme.



Obrázek je černobílý (tedy políčko je buďto vybarvené, nebo nevybarvené). Políčka je možno obarvit opakovaně.

Na vstupu dostanete obrázek jako tabulku `1` a `0`; na výstupu vypište jedno celé číslo – K – maximální možnou velikost štětce.

Například pro uvedený obrázek platí $K = 2$.

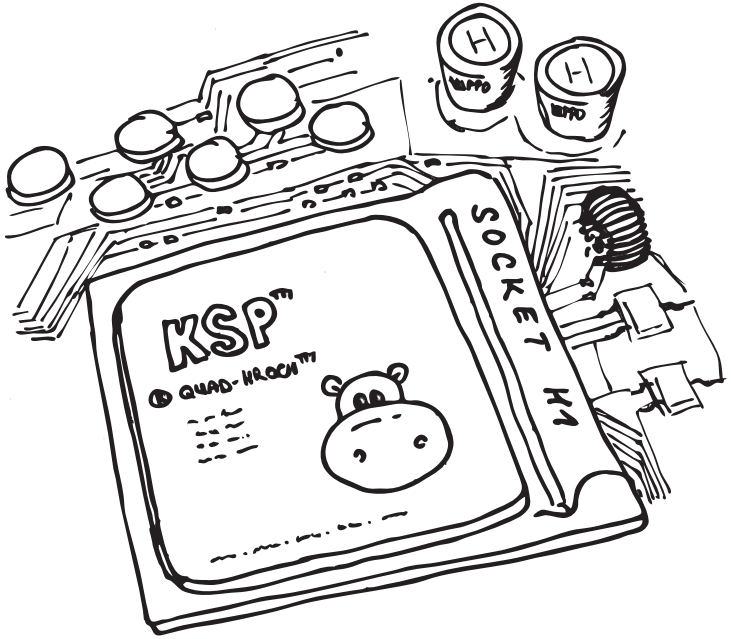
IBM PC v podstatě vytlačil z trhu veškerou konkurenci. Jeho jednoduchá a modulární architektura spolu s procesory firmy Intel (8088 apod.) způsobila, že náklady na výrobu i opravy byly (na svou dobu) velmi nízké.

Navíc v podstatě všechny další procesory, které Intel vyvíjel, byly s 8088 zpětně kompatibilní, co se týče instrukční sady. Nebyl proto problém spustit starý program na novějších strojích, což byl další důvod masivního rozšíření této platformy.

I současné počítače v sobě v drtivé většině mají procesory, na kterých při troše vůle i prastaré programy půjdou spustit. Nebude to asi ani pohodlné, ani rychlé, nicméně s velkou pravděpodobností poběží.

Z notebooku s procesorem Intel Core 2 Duo se s vámi loučí autor příběhu

Jan „Moskyto“ Matějka



Poznámky

- Popis algoritmu vysloveně svádí k „rejpnutí“: Jak víme, že spojením dvou cest, které provádíme, vznikne zase cesta (tj. že se na ní nemohou nějaké vrcholy opakovat)?

  To samozřejmě nevíme, ale všimněme si, že kdykoliv by to cesta nebyla, tak si ji nevybereme, protože původní cesta bez vrcholu k bude vždy kratší nebo alespoň stejně dlouhá... tedy alespoň pokud se v naší zemi nevyskytuje cyklus záporné délky. To bychom měli přidat do předpokladů našeho algoritmu, kdybychom byli pedanti.
- Pozor na pořadí cyklů – program vysloveně svádí k tomu, abychom psali cyklus pro `k` jako vnitřní... jenže pak samozřejmě nebude fungovat.

Cvičení

- Jak by algoritmus fungoval, kdyby silnice byly jednosměrné?
- Na první pohled nejpřirozenější hodnota, kterou bychom mohli použít pro ∞, je `maxint`. To ovšem nebude fungovat, protože $\infty + \infty$ přeteče. Stačí `maxint div 2`?
- Hodnoty v poli si přepisujeme pod rukama, takže by se nám mohly poplést hodnoty z předchozí fáze s těmi z fáze současné. Ale zachrání nás to, že čísla, o která jde, vyjdou v obou fázích stejně. Proč?

Nejdelší společná podposloupnost

Poslední příklad dynamického programování, který si předvedeme, se bude týkat posloupností. Mějme dvě posloupnosti čísel A a B.

Chceme najít jejich nejdelší společnou podposloupnost, tedy takovou posloupnost, kterou můžeme získat z A i B odstraněním některých prvků. Například pro posloupnosti

$$A = 2\,3\,3\,1\,2\,3\,2\,2\,3\,1\,1\,2$$
$$B = 3\,2\,2\,1\,3\,1\,2\,2\,3\,3\,1\,2\,2\,3$$

je jednou z nejdelších společných podposloupností tato posloupnost:

$$C = 2\,3\,1\,2\,2\,3\,1\,2.$$

Jakým způsobem můžeme takovou podposloupnost najít? Nejdříve nás asi napadne vygenerovat všechny podposloupnosti a ty pak porovnat.

Jakmile si ale spočítáme, že všech podposloupností posloupnosti o délce n je 2^n (každý prvek nezávisle na ostatních buď použijeme, nebo ne), najdeme raději nějaké rychlejší řešení.

Zkusme využít následující myšlenku: vyřešíme tento problém pouze pro první prvek posloupnosti A. Pak najdeme řešení pro první dva prvky A, přičemž využijeme předchozích výsledků. Takto pokračujeme pro první tři, čtyři, ... až n prvků.

Nejprve si rozmyslíme, co všechno si musíme v každém kroku pamatovat, abychom z toho dokázali spočíst krok následující. Určitě nám nebude stačit pamatovat si pouze nejdelší podposloupnost, jenže množina všech společných podposloupností je už zase moc velká.

Podívejme se tedy detailněji, jak se změní tato množina při přidání dalšího prvku k A: Všechny podposloupnosti, které v množině byly, tam zůstanou a navíc přibude několik nových, končících právě přidaným prvkem.

Ovšem my si podposloupnosti pamatujeme proto, abychom je časem rozšířili na nejdelší společnou podposloupnost.

Takže pokud známe nějaké dvě stejně dlouhé podposloupnosti P a Q končící nově přidaným prvkem v A a víme, že P končí v B dříve než Q, stačí si z nich pamatovat pouze P.

V libovolném rozšíření Q-čka totiž můžeme Q vyměnit za P a získat tím stejně dlouhou společnou podposloupnost.

Proto si stačí pro již zpracovaných a prvků posloupnosti A pamatovat pro každou délku l tu ze společných podposloupností $A[1 \ldots a]$ a B délky l, která v B končí na nejlevějším možném místě.

Dokonce nám bude stačit si místo celé podposloupnosti uložit jen pozici jejího konce v B. K tomu použijeme dvojrozměrné pole $D[a, l]$.

Ještě si dovolíme jedno malé pozorování: Koncové pozice uložené v poli D se zvětšují s rostoucí délkou podposloupnosti, čili $D[a, l] < D[a, l+1]$, protože posloupnosti délky $l+1$ nejsou ničím jiným než rozšířeními posloupností délky l o 1 prvek.

Teď již výpočet samotný:

Pokud už známe celý a-tý řádek pole D, můžeme z něj získat $(a+1)$-ní řádek. Projdeme postupně posloupnost B. Když najdeme v B prvek $A[a+1]$ (ten právě přidávaný do A), můžeme rozšířit všechny podposloupnosti končící před aktuální pozicí v B.

Nás bude zajímat pouze ta nejdelší z nich, protože rozšířením všech kratších získáme posloupnost, jejíž koncová pozice je větší než koncová pozice některé posloupnosti, kterou již známe. Rozšíříme tedy tu nejdelší podposloupnost a uložíme ji místo původní podposloupnosti.

Toto provedeme pro každý výskyt nového prvku v posloupnosti B. Všimněme si, že nemusíme procházet pole s podposloupnostmi stále od začátku, ale můžeme se v něm posouvat od nejmenší délky k největší.

Ukážeme si, jak vypadá zaplněné pole hodnotami při řešení problému s posloupnostmi z našeho příkladu. Řádky jsou pozice v A, sloupce délky podposloupností.

D	1	2	3	4	5	6	7	8	9	10	11	12
1	2	–	–	–	–	–	–	–	–	–	–	–
2	1	5	–	–	–	–	–	–	–	–	–	–
3	1	5	9	–	–	–	–	–	–	–	–	–
4	1	4	6	11	–	–	–	–	–	–	–	–
5	1	2	5	7	12	–	–	–	–	–	–	–
6	1	2	3	7	9	14	–	–	–	–	–	–
7	1	2	3	7	8	12	–	–	–	–	–	–
8	1	2	3	7	8	12	13	–	–	–	–	–
9	1	2	3	5	8	9	13	14	–	–	–	–
10	1	2	3	4	6	9	11	14	–	–	–	–
11	1	2	3	4	6	9	11	14	–	–	–	–
12	1	2	3	4	6	7	11	12	–	–	–	–

Zbývá popsat, jak z těchto dat zvládneme rekonstruovat hledanou nejdelší společnou podposloupnost (NSP).

Ukážeme si to na našem příkladu – jelikož poslední nenulové číslo na posledním řádku je v 8. sloupci, má hledaná NSP délku 8.

$D[12, 8] = 12$ říká, že poslední písmeno NSP je na pozici 12 v posloupnosti B. Jeho pozici v posloupnosti A určuje

Časová složitost algoritmu je $\mathcal{O}(NM)$, neboť se skládá z N kroků, z nichž každý vyžaduje čas $\mathcal{O}(M)$. Paměťová složitost činí $\mathcal{O}(N+M)$, což představuje paměť potřebnou pro uložení pomocného pole A a hmotností daných předmětů.

Cvičení a poznámky

- Proč pole A procházíme pozadu a ne popředu?
- Složitost algoritmu vypadá jako polynomiální, ale to je trochu podvod. Závisí totiž na hodnotě M. Pokud tuto hodnotu na vstupu zapíšeme obvyklým způsobem, tedy v desítkové nebo dvojkové soustavě, použijeme řádově $\log M$ cifer. Naše M proto bude vzhledem k délce vstupu až exponenciálně velké. To je typický příklad takzvaného *pseudopolynomiálního* algoritmu – tedy takového, jenž je vzhledem k hodnotám na vstupu polynomiální, ale k délce vstupu exponenciální. Podrobnosti si můžete přečíst v kuchařce o těžkých úlohách.[4]

```
var N: integer; {   počet předmětů   }
    M: integer; { hmotnostní omezení }
    hmotnost: array[1..N] of integer;
                { hmotnosti daných předmětů }
    A: array[0..M] of integer;
    i, k: integer;
begin
  A[0]:=-1;
  for i:=1 to M do A[i]:=0;
  for k:=1 to N do
    for i:=M downto hmotnost[k] do
      if (A[i-hmotnost[k]]<>0) and (A[i]=0) then
        A[i]:=k;
  i:=M; while A[i]=0 do i:=i-1;
  writeln('Maximální hmotnost: ',i);
  write('Předměty v množině:');
  while A[i]<>-1 do begin
    write(' ',A[i]);
    i:=i-hmotnost[A[i]];
  end;
  writeln;
end.
```

Nejkratší cesty a Floydův-Warshallův algoritmus

Náš další příklad bude z oblasti grafových algoritmů, ale zkusíme si jej nejdříve říci bez grafů:

Bylo-nebylo-je N měst. Mezi některými dvojicemi měst vedou (obousměrné) *silnice,* jejichž (nezáporné) délky jsou dány na vstupu. Předpokládáme, že silnice se jinde než ve městech nepotkávají (pokud se kříží, tak mimoúrovňově).

Úkolem je spočítat nejkratší vzdálenosti mezi všemi dvojicemi měst, tj. délky nejkratších cest mezi všemi dvojicemi měst.

Cestou rozumíme posloupnost měst takovou, že každá dvě po sobě následující města jsou spojené silnicí, a délka cesty je součet délek silnic, které tato města spojují.

V grafové terminologii tedy máme daný ohodnocený neorientovaný graf a chceme zjistit délky nejkratších cest mezi všemi dvojicemi jeho vrcholů.

Půjdeme na to následovně – vzdálenosti mezi městy jsou na začátku algoritmu uloženy ve dvourozměrném poli D, tj. $D[i][j]$ je vzdálenost z města i do města j. Pokud mezi městy i a j nevede žádná silnice, bude $D[i][j] = \infty$ (v programu bude tato hodnota rovna nějakému dostatečně velkému číslu).

V průběhu výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty mezi městy i a j.

Algoritmus se skládá z N fází. Na konci k-té fáze bude v $D[i][j]$ uložena délka nejkratší cesty mezi městy i a j, která může procházet skrz libovolná z měst $1,\ldots,k$.

V průběhu k-té fáze tedy stačí vyzkoušet, zda je mezi městy i a j kratší stávající cesta přes města $1,\ldots,k-1$, jejíž délka je uložena v $D[i][j]$, nebo nová cesta přes město k.

Pokud nejkratší cesta prochází přes město k, můžeme si ji rozdělit na nejkratší cestu z i do k a nejkratší cestu z k do j. Délka takové cesty je tedy rovna $D[i][k]+D[k][j]$.

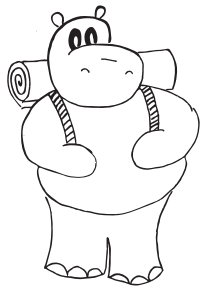
Takže pokud je součet $D[i][k]+D[k][j]$ menší než stávající hodnota $D[i][j]$, nahradíme hodnotu na pozici $D[i][j]$ tímto součtem, jinak ji ponecháme.

Z popisu algoritmu přímo plyne, že po N-té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města i do města j.

Protože v každé z N fází algoritmu musíme vyzkoušet všechny dvojice i a j, vyžaduje každá fáze čas $\mathcal{O}(N^2)$. Celková časová složitost našeho algoritmu tedy je $\mathcal{O}(N^3)$. Co se paměti týče, vystačíme si s polem D a to má velikost $\mathcal{O}(N^2)$.

Program bude vypadat následovně:

```
var N: integer; { počet měst }
    D: array[1..N] of array[1..N] of longint;
      { délky silnic mezi městy, D[i][i]=0,
        místo neexistujících je "nekonečno" }
    i, j, k: integer;
begin
  for k:=1 to N do
    for i:=1 to N do
      for j:=1 to N do
        if D[i][k]+D[k][j] < D[i][j] then
          D[i][j]:=D[i][k] + D[k][j];
end.
```



Popišme si ještě, jak bychom postupovali, kdybychom kromě vzdáleností mezi městy chtěli nalézt i nejkratší cesty mezi nimi.

To lze jednoduše vyřešit například tak, že si navíc budeme udržovat pomocné pole $E[i][j]$ a do něj při změně hodnoty $D[i][j]$ uložíme nejvyšší číslo města na cestě z i do j délky $D[i][j]$ (při změně v k-té fázi je to číslo k).

Máme-li pak vypsat nejkratší cestu z i do j, vypíšeme nejprve cestu z i do $E[i][j]$ a pak cestu z $E[i][j]$ do j. Tyto cesty nalezneme stejným (rekurzivním) postupem.

[4] http://ksp.mff.cuni.cz/tasks/23/cook5.html

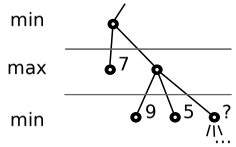
24-2-8 Alfa-beta ořezávání a piškvorky 14 bodů

V prvním dílu jsme si představili minimaxový algoritmus. Zjistili jsme také, že je možno zrychlovat jeho běh tím, že neuvažujeme některé možné tahy vedoucí z aktuální pozice. Za to však platíme možným přehlédnutím (nepravděpodobně) optimálního tahu.

Pro jednoduchost si označme hráče táhnoucího v maximalizačních úrovních jako Max (to je ten, pro něhož hledáme nejlepší tah) a jeho soupeře Min (ten táhne v úrovních, kde se vybírá minimum hodnot ze synů).

Alfa-beta ořezávání je úprava minimaxu založená na jednoduchém pozorování, které znatelně urychluje průměrnou dobu běhu prohledávání, aniž by ovlivnilo výsledek.

Uvažme následující situaci:



Potřebuje stroj znát hodnotu pozice s otazníkem, aby mohl vrátit výsledek? Nikoliv, protože hodnota pozice v nadřazeném vrcholu už větší nebude, ať připočítáme cokoliv.

Proč tomu tak je? Když bude na pozici s otazníkem více než 5, tak minimem v této úrovni bude 5; v opačném případě bude minimum menší než 5.

Hodnota pozice v nadřazeném vrcholu tudíž bude maximálně 5, ale to už je méně než 7 – hodnota vedlejší pozice – takže pozice s otazníkem už rozhodování nikterak neovlivní.

Uvědomme si, jak je v dané pozici takové pozorování cenné – skutečně můžeme přestat počítat (říká se *zaříznout*) celou *větev.* Zkoumat ji by byla spousta práce.

Kdy přesně lze větev zaříznout? Pokud se nacházíme v minimalizační fázi a v libovolném vrcholu v maximalizační fázi na cestě ke kořeni prohledávacího stromu existuje už dopočítaný syn, který má vyšší hodnotu než aktuální minimum v této úrovni.

To vše obdobně pro vrcholy ve fázi maximalizační.

Nyní je vhodná chvíle přestat na chvíli číst a rozmyslet si, jestli opravdu všemu rozumíte. Zkuste si nakreslit složitější situaci a chvíli ji analyzujte.

Takové chování se příjemně implementuje za použití dvou proměnných, kterým budeme nečekaně říkat *alfa* a *beta*. *Alfa* bude maximum z dopočítaných synů v maximalizačních fázích a *beta* minimum z fází minimalizačních. S těmi pak bude stačit nově dopočítané hodnoty vrcholů porovnávat.

Na začátku zvolíme *alfu* jako $-\infty$ (Max může prohrát) a *betu* jako $+\infty$ (i Min může prohrát). V maximalizačních úrovních pak upravujeme dle hodnoty v synech *alfu*, v minimalizačních *betu*.

Jelikož *alfa* udává, jakou nejlepší hodnotu v prozkoumané části stromu může získat Max, a *beta* nejlepší nalezenou hodnotu pro hráče Min, platí v každém okamžiku, že *alfa* < *beta*. Když se tedy v průběhu výpočtu dostaneme s *alfou* nad *betu* (či s *betou* pod *alfu*), můžeme skončit prohledávání synů zkoumaného uzlu.

Znovu se zde ujistěte, jestli tušíte, proč si algoritmus může takové ořezávání dovolit.

K lepšímu pochopení může posloužit pseudokód.

```
def alfabeta(pozice, hloubka,
             alfa, beta, natahu):
  if hloubka == 0 or konec_hry(pozice)
    return hodnota(pozice)

  if natahu = Max:
    for p in mozne_tahy(pozice, Max):
      alfa = max(alfa, alfabeta(p, hloubka-1,
                                alfa, beta, Min) )
      if beta <= alfa:
        break
    return alfa

  if natahu = Min:
    for p in mozne_tahy(pozice, Min):
      beta = min(beta, alfabeta(p, hloubka-1,
                                alfa, beta, Max) )
      if beta <= alfa:
        break
    return beta
```

Alfa-beta ořezávání je nejúčinnější, jsou-li první prozkoumávané tahy nejlepší možné pro hráče, v jehož úrovni se nacházíme. Pak je možné větve pod ostatními tahy rychle zaříznout a minimax tak výrazně zrychlit. Máme-li však smůlu v seřazení tahů od nejhoršího, alfa-beta nám vůbec nepomůže.

Zkoumat na začátku dobré tahy ale samozřejmě není tak jednoduché – víme-li, které tahy jsou nejlepší, žádný minimax už nepotřebujeme. Rozhodně se nám ale vyplatí začít vymýšlet heuristiky pro generátory tahů, které se budou snažit nějak chytře předřazovat.

Často používaná je metoda *killer move*. Pamatujeme si, které tahy v jiných větvích výpočtu vedly k dobrým výsledkům, a ty pak upřednostňujeme při prohledávání.

Trochu jednodušší je *iterativní prohlubování,* při kterém prostě minimax pouštíme pro stále větší hloubku propočítávání, tj. nejdříve procházíme strom do hloubky 1, pak do hloubky 2, 3 atd. Kromě nejnižšího patra stromu používáme pro řazení tahů výsledky z předchozího výpočtu.

Neplýtváme časem, když některé části stromu prohledáváme vícekrát? Ne, protože exponenciální časová složitost minimaxu vede k tomu, že všechna hledání dohromady zpravidla zaberou méně času než to poslední, nejhlubší.

Piškvorková teorie

Poodstupme opět od vulgárního světa strojového prohledávání pozic. Matematika nám v minulém díle pomohla při analýze Nimu. Piškvorky jsou podobně jednoduše definovatelná hra – dokážeme vyřešit tu?

Záleží trochu, co přesně si pod piškvorkami představíme. Nejčastěji asi hru na neomezeném čtverečkovaném papíře, kde se střídáme v pokládání značek, přičemž výhry dosáhne hráč, který jich první vytvoří pět v řadě.

O takových piškvorkách tušíme, že v nich má výhodu začínající hráč. Jak ale velkou? Má vyhrávající strategii? Může alespoň vždy zabránit prohře? Jednoduchá, ale důležitá skutečnost zní: druhé tvrzení platí, začínající hráč v piškvorkách má *neprohrávající* strategii.

Představíme-li si pro spor, že druhý hráč má vyhrávající strategii (což je negace tvrzení „začínající hráč má strategii neprohrávající“), můžeme aplikovat přeslavný *argument o kradení strategie.*

První hráč by mohl svůj první tah zahrát libovolně na desku, zapomenout na něj (tj. nadále přemýšlet o desce, jako by tam nebyl) a potom hrát podle postulované vyhrávající strategie druhého hráče.

Onen libovolný tah by mu v žádných možných pokynech takové strategie nepřekážel – dostal-li by za úkol hrát na toto místo, učinil by prostě libovolný jiný tah a zapomněl by na ten.

Z toho však plyne, že vyhrávající strategii má jako hráč začínající (tj. zloděj), tak hráč druhý! To je spor, ke kterému jsme chtěli dospět. Vyhrávající strategie druhého hráče neexistuje, první hráč má strategii neprohrávající.

Argument je to slavný, protože je široce aplikovatelný – můžeme jej použít na všechny *poziční hry,* což je terminus technicus pro hry, ve kterých hráči trvale zabírají části herní plochy a vítězí hráč, který zabere některou z vítězných podmnožin těchto částí.

Nabízí se ohromně zajímavá otázka, má-li první hráč vyhrávající strategii, nebo je-li to remíza. Někde daleko před odpovědí na tuto otázku však možnosti současné matematiky končí. Pro modifikovanou variantu piškvorek, kde vyhrává řada osmi značek, je dokázáno, že bezchybná hra již remízou končí.

Úkol 1 [5b]: Určete a dokažte výsledek piškvorek, kde vyhrává řada čtyř značek.

Úkol 2 [9b]: Dokažte, že v piškvorkách, kde vyhrává řada devíti značek, má druhý hráč neprohrávající strategii. *Nápověda: Rozdělení do párů.*

Dobře, neomezená herní plocha může být háčkem. Pojďme hrát piškvorky na omezené desce n^d, což je d-rozměrná krychle o hraně n. Za vítěze budeme pokládat hráče, který první udělá sérii n značek ve stejném směru.

Pro $n = 3$ a $d = 2$ dostáváme známé tic-tac-toe, pro $n = 4$, $d = 3$ oblíbené trojrozměrné piškvorky…

Ani zde není situace příliš růžová a za to, že víme, že má v uvedených trojrozměrných piškvorkách začínající hráč vítěznou strategii, vděčíme dosti komplikovanému důkazu plného rozborů případů. Situace pro $n = 5$, $d = 3$ je zatím otevřená.

Lukáš Lánský a Pavel Veselý

Recepty z programátorské kuchařky

Rekurzivní funkce a dynamické programování

Rekurzivní funkce je taková funkce, která při svém běhu volá sama sebe, často i více než jednou, což v důsledku může vést na exponenciální algoritmus.

Dynamické programování je technika, kterou jde z pomaléhо rekurzivního algoritmu vyrobit pěkný polynomiální (až na výjimečné případy). Ale nepředbíhejme, nejdříve se podíváme na jednoduchý příklad rekurze:

Fibonacciho čísla

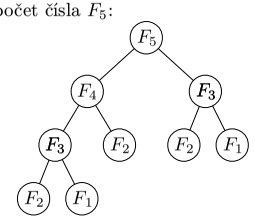
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejímiž prvními dvěma členy jsou jedničky ($F_1 = 1$, $F_2 = 1$) a každý další člen je součtem dvou předchozích ($F_n = F_{n-1} + F_{n-2}$ pro $n > 2$). Začíná takto:

$$1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad \ldots$$

Pro nalezení n-tého členu (ten budeme značit F_n) si napíšeme rekurzivní funkci `Fibonacci(n)`, která bude postupovat přesně podle definice – zeptá se sama sebe rekurzivně, jaká jsou dvě předchozí čísla, a pak je sečte. Možná více řekne program:

```
function Fibonacci(n: integer): integer;
begin
  if n <= 2 then
    Fibonacci := 1
  else
    Fibonacci := Fibonacci(n-1) + Fibonacci(n-2)
end;
```

To, jak funkce volá sama sebe, si můžeme snadno nakreslit třeba pro výpočet čísla F_5:



Vidíme, že se program rozvětvuje, což tvoří strom volání. V každém vrcholu tohoto stromu trávíme konstantní čas, takže časová složitost celého algoritmu je až na konstantu rovna počtu vrcholů tohoto stromu. Kolik to je, spočítáme jednoduchou úvahou.

Každý vrchol stromu vrací hodnotu, která je součtem hodnot v jeho synech. Proto je hodnota v kořeni rovna součtu hodnot v listech. V listech jsou ovšem jedničky (F_1 a F_2), takže listů musí být právě F_n a všech vrcholů dohromady aspoň F_n.

Proto na spočítání n-tého Fibonacciho čísla spotřebujeme čas alespoň takový, kolik je ono číslo samo. Ale jak velké takové F_n vlastně je? Můžeme třeba využít toho, že

$$F_n = F_{n-1} + F_{n-2} \geq 2 \cdot F_{n-2},$$

z čehož indukcí dokážeme

$$F_n \geq 2^{n/2} \quad \text{pro } n \geq 6.$$

Funkce `Fibonacci` má tedy alespoň exponenciální časovou složitost což není nic vítaného.

Jak najít efektivnější algoritmus? Všimneme si, že některé podstromy jsou shodné. Zřejmě to budou ty části, které reprezentují výpočet stejného Fibonacciho čísla – v našem příkladě třeba třetího. Tyto výpočty opakujeme stále dokola.

Nenabízí se proto nic snazšího, než si jejich výsledky uložit a pak je kdykoliv vytáhnout jako pověstného králíka z klobouku s minimem námahy.

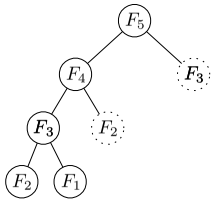
Právě zde je zmínka o králících příhodná.
Legenda o Fibonacciho číslech vypráví, že k jejich objevu došlo při výzkumu rozmnožování králíků.

Leonardo Pisánský (známý též jako Fibonacci) totiž pěstoval králíky. První dva měsíce měl 1 pár, další měsíc měl 2 páry, pak 3, pak 5, …

Bude nám k tomu stačit jednoduché pole P o n prvcích, na počátku inicializované nulami. Kdykoliv budeme chtít spočítat některý člen, nejdříve se podíváme do pole, zda jsme ho již jednou nespočetli. A naopak jakmile hodnotu spočítáme, hned si ji do pole poznamenáme:


```
var P: array[1..MaxN] of integer;
function Fibonacci(n: integer): integer;
begin
  if P[n] = 0 then
  begin
    if n <= 2 then
      P[n] := 1
    else
      P[n] := Fibonacci(n-1) + Fibonacci(n-2)
  end;
  Fibonacci := P[n]
end;
```

Podívejme se, jak vypadá strom volání nyní:



Na každý člen posloupnosti se tentokrát ptáme maximálně dvakrát – k výpočtu ho potřebují dva následující členy. To ale znamená, že funkci `Fibonacci` zavoláme maximálně $2n$-krát, čili jsme touto jednoduchou úpravou zlepšili exponenciální složitost na lineární.

Zdálo by se, že abychom získali čas, museli jsme obětovat paměť, ale to není tak úplně pravda. V prvním příkladu sice nepoužíváme žádné pole, ale při volání funkce si musíme zapamatovat některé údaje, jako je třeba návratová adresa, parametry funkce a její lokální proměnné, a na to samotné potřebujeme určitě paměť lineární s hloubkou vnoření, v našem případě tedy lineární s n.

Určitě vás už také napadlo, že n-té Fibonacciho číslo se dá snadno spočítat i bez rekurze. Stačí prvky našeho pole P plnit od začátku – kdykoli známe $P[1 \ldots k] = F_{1 \ldots k}$ (všechny prvky pole na pozicích od 1 do k), dokážeme snadno spočítat i $P[k+1] = F_{k+1}$:

```
function Fibonacci(n: integer): integer;
var
  P: array[1..MaxN] of integer;
  i: integer;
begin
  P[1] := 1;
  P[2] := 1;
  for i := 3 to n do
    P[i] := P[i-1] + P[i-2];
  Fibonacci := P[n]
end;
```

Zopakujme si, co jsme postupně udělali – nejprve jsme vymysleli pomalou rekurzivní funkci, kterou jsme zrychlili zapamatováváním si mezivýsledků.

Nakonec jsme ale celou rekurzi „obrátili naruby“ a mezivýsledky počítali od nejmenšího k největšímu, aniž bychom se starali o to, jak se na ně původní rekurze ptala.

V případě Fibonacciho čísel je samozřejmě snadné přijít rovnou na nerekurzivní řešení a dokonce si všimnout, že si stačí pamatovat jen poslední dvě hodnoty, a paměťovou složitost tak zredukovat na konstantní.

Zmíněný obecný postup zrychlování rekurze nebo rovnou řešení úlohy od nejmenších podproblémů k těm největším funguje i pro řadu složitějších úloh. Obvykle se mu říká *dynamické programování.*

Problém batohu

Je dáno N předmětů o hmotnostech $m_1, \ldots, m_N$ (celočíselných) a také číslo M (nosnost batohu). Úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, ale přitom nepřekročil M. Předvedeme si algoritmus, který tento problém řeší v čase $\mathcal{O}(MN)$.

Náš algoritmus bude používat pomocné pole $A[0 \ldots M]$ a jeho činnost bude rozdělena do N kroků. Na konci k-tého kroku bude prvek $A[i]$ nenulový právě tehdy, jestliže z prvních k předmětů lze vybrat předměty, jejichž součet hmotností je přesně i.

Před prvním krokem (po nultém kroku) jsou všechny hodnoty $A[i]$ pro $i > 0$ nulové a $A[0]$ má nějakou nenulovou hodnotu, řekněme -1.

Všimněme si, jak kroky algoritmu odpovídají podúlohám, které řešíme – v prvním kroku vyřešíme podúlohu tvořenou jen prvním předmětem, ve druhém kroku prvními dvěma předměty, pak prvními třemi předměty atd.

Popišme si nyní k-tý krok algoritmu. Pole A budeme procházet od konce, tj. od $i = M$. Pokud je hodnota $A[i]$ stále nulová, ale hodnota $A[i - m_k]$ je nenulová, změníme hodnotu uloženou v $A[i]$ na k (později si vysvětlíme, proč zrovna na k).

Nyní si rozmyslíme, že po provedení k-tého kroku odpovídají nenulové hodnoty v poli A hmotnostem podmnožin z prvních k předmětů (*podmnožina* je v podstatě jen výběr nějaké části předmětů).

Pokud je hodnota $A[i]$ nenulová, pak buď byla nenulová před k-tým krokem (a v tom případě odpovídá hmotnosti nějaké podmnožiny prvních $k-1$ předmětů) anebo se stala nenulovou v k-tém kroku.

Potom ale hodnota $A[i - m_k]$ byla před k-tým krokem nenulová, a tedy existuje podmnožina prvních $k-1$ předmětů, jejíž hmotnost je $i - m_k$. Přidáním k-tého předmětu k této podmnožině vytvoříme podmnožinu předmětů hmotnosti přesně i.

Naopak, pokud lze vytvořit podmnožinu X hmotnosti i z prvních k předmětů, pak takovou podmnožinu X lze buď vytvořit jen z prvních $k-1$ předmětů, a tedy hodnota $A[i]$ je nenulová již před k-tým krokem, anebo k-tý předmět je obsažen v takové množině X.

Potom ale hodnota $A[i - m_k]$ je nenulová před k-tým krokem (hmotnost podmnožiny X bez k-tého prvku je $i - m_k$) a hodnota $A[i]$ se stane nenulovou v k-tém kroku.

Po provedení všech N kroků odpovídají nenulové hodnoty $A[i]$ přesně hmotnostem podmnožin ze všech předmětů, co máme k dispozici. Speciálně největší index i_0 takový, že hodnota $A[i_0]$ je nenulová, odpovídá hmotnosti nejtěžší podmnožiny předmětů, která nepřekročí hmotnost M.

Nalézt jednu množinu této hmotnosti také není obtížné: V k-tém kroku jsme měnili nulové hodnoty v poli A na hodnotu k, takže v $A[i_0]$ je uloženo číslo jednoho z předmětů nějaké takové množiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu atd. Zdrojový kód tohoto algoritmu lze nalézt na další straně.