KSI

Korespondenční Seminář z Informatiky

Milí řešitelé,

věříme, že se vám první sada našeho semináře líbila a že jste nám poslali správná řešení příkladů, za které budeme moct rozdat spoustu bodů. Aby jste však nevyšli z formy a mohli dál namáhat své mozky, nachystali jsme pro vás další sadu zajímavých příkladů.

V této sadě se budeme zabývat tříděním. V úvodníku si vysvětlíme, co to vlastně je a k čemu je to dobré. Ukážeme si také několik algoritmů pro třídění a popíšeme jejich silné a slabé stránky. Ukážeme si také některé vlastnosti algoritmů, které je dobré zohlednit při výběru vhodného třídícího algoritmu pro váš problém. Navíc si můžete vše vyzkoušet na příkladech této sady!

Třídění

I když se to na první pohled nemusí zdát, třídění je velmi důležitou součástí informatiky. Jak totiž napovídá už samotný název, informatika je věda o práci s informacemi. A informací, se kterými naše programy pracují, bývá zpravidla hodně. Je proto potřeba v těchto informacích mít systém, který nám pomůže se v nich vyznat, čímž se dostáváme ke třídění. S hezky roztříděnými informacemi se totiž pracuje mnohem lépe, než s hromadou náhodně naskládaných dat — zkoušeli jste někdy vytrhat všechny stránky z telefonního seznamu, smíchat je a pak zavolat paní Mrázkové? Pak jistě víte, o čem se bavíme ...

Pod tříděním v informatice rozumíme uspořádání nějaké sady dat, ve které jsou prvky rozmístěny náhodně, do takové datové struktury, ve které jsou uloženy podle nějakého pravidla, aby se v nich dobře hledalo. Podle jakého pravidla konkrétně budeme třídit záleží na situaci — k řazení závodníků použijeme jejich umístění v závodu. Telefonní seznam budeme raději řadit podle abecedy, aby se nám v něm dobře hledala paní Mrázková. V některých případech můžete dokonce použít nějakou ošklivou matematickou formuli (ale ne v řešení KSI!), která zrovna náhodou popisuje důležité vlastnosti prvků, které třídíte. V následujícím textu budeme pro jednoduchost třídit pouze čísla, a to podle velikosti.

Bubble sort — třídění probubláváním

První algoritmus, který si představíme, bude postupně vyměňovat prvky, které jsou setřízeny špatně. K tomu budeme používat pouze porovnání dvou sousedních prvků posloupnosti: Pokud bude větší prvek vlevo, tak tyto dva prvky vyměníme.

Tímto porovnáním zkontrolujeme postupně každou z dvojic sousedících prvků ([0, 1], [1, 2], [2, 3] ... [n-2, n-1]). Pokud jsme při průchodu vyměnili nějaké dva prvky, pak víme že posloupnost nebyla správně setřízena a je nutné udělat další průchod celou posloupností. Pokud jsme však při průchodu žádnou dvojici vyměnit nemuseli, tak to znamená, že je správně setřízená.

Na třídění probubláváním si nyní popíšeme některé důležité vlastnosti, které může třídící algoritmus mít:

Časová složitost: Jeden průchod posloupností zabere $n-1$ kroků, kde n je velikost zahrádky. V nejhorším případě, kdy úplně vpravo bude nejmenší číslo, je potřeba $n-1$ průchodů celou posloupností pro přesun na správné místo a jeden průchod na kontrolu. Dohromady tedy dostáváme složitost $\mathcal{O}((n-1)\cdot((n-1)+1)) = \mathcal{O}((n-1)\cdot n) = \mathcal{O}(n^2)$.

Prostorová složitost: Algoritmus má prostorovou složitost $\mathcal{O}(n)$. Pro jednu výměnu totiž stačí pouze jedno odkládací místo, kam si můžeme odložit číslo z pozice i. Na pozici i pak přesuneme číslo z $i+1$ a na $i+1$ dáme číslo z odkládacího místa, čímž ho uvolníme pro další výměny. O takovém algoritmu, kterému stačí jen jedno místo pro dočasné odkládání dat, říkáme že pracuje *na místě*.

Algorithm 1 BubbleSort

Vstup: pole čísel *vstupniPole* o délce n
  boolean *checked* ← **false**; // proběhla kontrola pole úspěšně, nebo je třeba další průchod?
  while not *checked* **do**
    checked ← **true**
    for $i = 0 \rightarrow n - 2$ **do**
      if $vstupniPole[i] > vstupniPole[i + 1]$ **then**
        vyměň prvky $(i, i + 1)$
        checked ← **false**
      end if
    end for
  end while
  return *vstupniPole*

Online zpracování: Některé algoritmy dokáží pracovat se seznamem, který ještě není ani celý. Pokud lze přidávat prvky na konec seznamu, který už je algoritmem částečně zpracovaný, říkáme že *pracuje online*. Bubble sort tuto vlastnost nemá.

Algoritmus třídění probubláváním je díky své jednoduchosti vhodný k pochopení problematiky třídění. Protože má ale kvadratickou složitost, není příliš vhodný k třídění větších polí. Bubble sortu nepomůže ani to, že dokáže pracovat na místě, protože existují algoritmy, které to zvládají i s výrazně nižší časovou složitostí.

Insert sort — třídění vkládáním

Dalším třídícím algoritmem, který si uvedeme, bude Insert sort. Tento algoritmus si udržuje částečně setřízený seznam, ve kterém je prvních i prvků vstupního seznamu již uspořádaných tak, jak mají být. V každé iteraci do tohoto seznamu přidá jeden prvek na místo, kam patří. Po přidání posledního prvku tak máme celý seznam setřízený ve výstupním poli. Pseudokód algoritmu by tedy vypadal asi takto:

Algorithm 2 InsertSort

Vstup: pole čísel *vstupniPole* o délce n
  int *vystupniPole*[n]
  for $i = 0 \rightarrow n - 1$ **do**
    //chceme zařadit i-tý prvek seznamu
    prvek ← *vstupniPole*[i]
    $j \leftarrow 0$
    while vystupniPole[j] < prvek **do**
      $j \leftarrow j + 1$
    end while
    //nyní j značí index první položky, která je větší než zařazovaný prvek
    vlož *prvek* do pole *vystupniPole* na index j
    prvky s indexem j a vyšším posuň o jedno místo dál
  end for
  return *vystupniPole*

Časová složitost: Ve vnitřním cyklu provedeme nejvýše i operací, kde i je index zařazovaného prvku. Vnější cyklus provedeme n krát a i se bude postupně zvyšovat až do n. Dostáváme tedy složitost $\mathcal{O}(1 + 2 + 3 + .. + n) = \mathcal{O}(n \cdot (n + 1)/2) = \mathcal{O}(n^2)$.

Prostorová složitost: Algoritmus potřebuje zvlášť pole na ukládání vytvářeného seznamu, jinak nic. Prostorová složitost je tedy $\mathcal{O}(2n) = \mathcal{O}(n)$. Algoritmus nepracuje na místě (ale lze upravit tak, aby pracoval).

Online zpracování: Insert sort vezme vždy pouze první nezpracovaný prvek vstupního seznamu a na zbytek nehledí. Lze proto na konec seznamu přidávat i po spuštění algoritmu, tedy pracuje online.

Třídění vkládáním má sice kvadratickou složitost, ale je velmi zajímavý pro aplikace, které produkují hodnoty průběžně. Příkladem takové aplikace může být třeba registrace závodníků, kdy potřebujeme abecedně setřízený seznam jmen, ale zároveň potřebujeme být kdykoliv schopni přidat další záznam, aniž bychom třídili celou posloupnost od začátku. Většina rychlejších algoritmů totiž nedokáže využít toho, že část posloupnosti je již nějakým způsobem setříděná a bezhlavě ji třídí celou znovu.

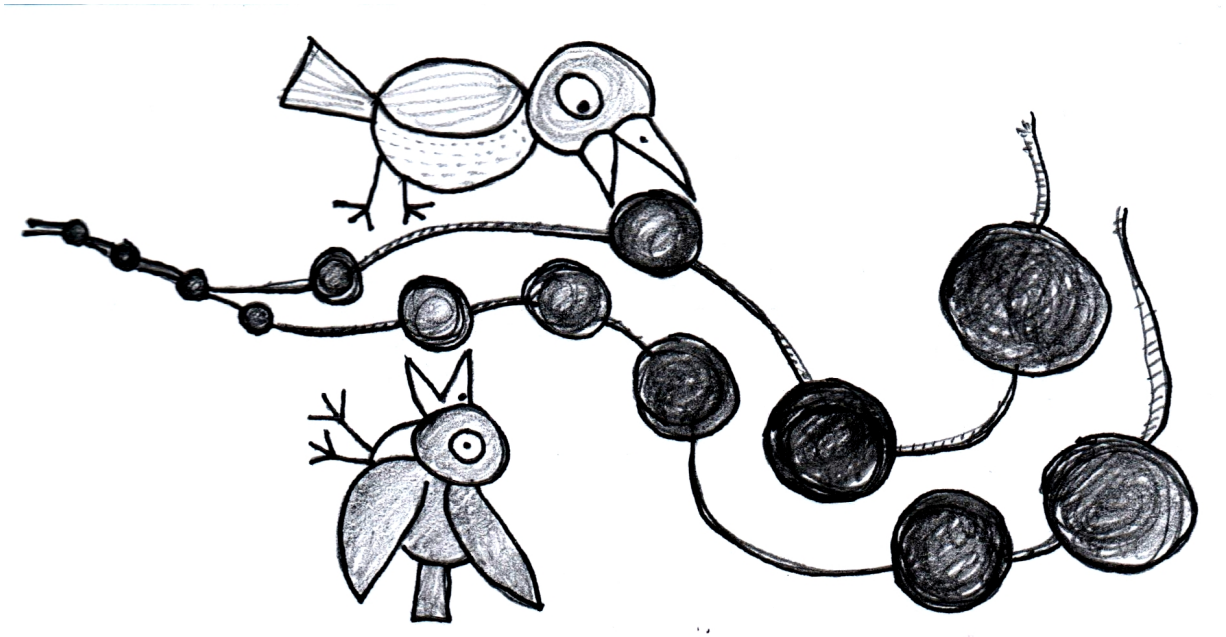
Merge sort — třídění spojováním

Bubble sort i Insert sort pracují s časovou složitostí $\mathcal{O}(n^2)$. Nyní si popíšeme algoritmus Merge sort, který má složitost $\mathcal{O}(n \cdot log\ n)$, což je zároveň nejnižší složitost, jaké může obecný třídící algoritmus dosáhnout.

Klíčovou součástí Merge sortu je rekurze. Pro lepší představu si ji můžete zatím představit jako kouzelnou krabičku, která setřídí pole, které jí zadáme. Podmínkou však je, že toto pole musí být kratší než to, které třídíme.

Merge sort funguje tak, že vstupní pole rozdělíme na poloviny (`a` a `b`), které setřídíme pomocí rekurze (kouzelnou krabičkou). Setřízená pole `a, b` použijeme jako fronty, kde vepředu je vždy nejmenší prvek z celé fronty. Porovnáním čelních prvků obou front získáme nejmenší prvek celého vstupního pole. Tento prvek zařadíme do výstupního pole a popojdeme na další prvek příslušné fronty. Postup opakujeme, dokud jsou v alespoň jedné frontě nezpracované prvky.

Jelikož víme, že jednoprvkové pole je setříděné, můžeme v tomto případě vrátit přímo toto pole (první část kódu). Díky tomu se vyhneme tomu, že by jsme se zacyklili dělením jednoprvkových polí na poloviny, čtvrtiny, osminy atd. Toto je také základ naší 'kouzelné krabičky': protože víme, jak setřídit pole délky 1, dokážeme také pomocí popsané metody setřídit pole délky 2. Potom ale naše funkce zvládne přece setřídit i pole délky 4, protože ji rozdělíme na dvě pole velikosti 2, ty setřídíme zvlášť a pak je spojíme do jednoho. A takto se postupně dostaneme k libovolné délce vstupního pole.



Algorithm 3 MergeSort

Vstup: pole čísel $vstupniPole$ o délce n
  if $n = 1$ **then**
    return $vstupniPole$ // omezení rekurze
  end if

  int $a[] \leftarrow vstupniPole[0 \cdots n/2 - 1]$
  int $b[] \leftarrow vstupniPole[n/2 \cdots n - 1]$
  // rekurzivně setřídíme pole a a b
  $a \leftarrow \text{mergeSort}(a)$
  $b \leftarrow \text{mergeSort}(b)$

  // slijeme setříděná pole a, b do jednoho setříděného pole
  int $ia \leftarrow 0, ib \leftarrow 0$ // ukazatele na pozici hlavy fronty v polích a, b
  int $i \leftarrow 0$
  while $ia < delka(a)$ **or** $ib < delka(b)$ **do**
    if $a[ia] < b[ib]$ **then**
      $vystupniPole[i] \leftarrow a[ia]$
      $ia \leftarrow ia + 1$
    else
      $vystupniPole[i] \leftarrow b[ib]$
      $ib \leftarrow ib + 1$
    end if
    $i \leftarrow i + 1$
  end while
  return $vystupniPole$

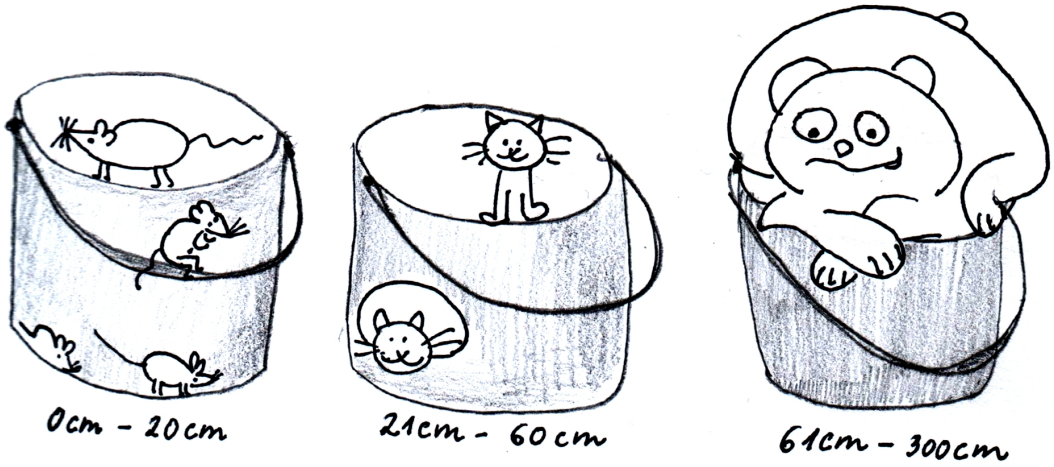
Časová složitost: Pro jednoduchost budeme nyní uvažovat pouze pole délky $n_0 = 2^k$, kde k je libovolné přirozené číslo. Po rozdělení vstupního pole na poloviny získáme dvě pole délky $n_1 = 2^{k-1}$. Jejich rozpůlením získáme $4 = 2^2$ polí délky $n_2 = 2^{k-2}$. Rozpůlením těchto polí získáme 2^3 polí délky $n_3 = 2^{k-3}$. Takto budeme pokračovat, dokud nemáme 2^k jednoprvkových polí. Získáme tak k úrovní.
Spojení dvou setřízených polí o délkách na, nb do jednoho setřízeného pole má časovou složitost $\mathcal{O}(na + nb)$. Zároveň víme, že na každé z úrovní je 2^i polí o délce 2^{k-i}. Tyto pole budeme spojovat po dvou, čímž získáme časovou složitost jedné úrovně $\mathcal{O}(2^i \cdot 2^{k-i}) = \mathcal{O}(2^{i+k-i}) = \mathcal{O}(2^k)$. Když tedy sečteme časovou složitost všech úrovní, kterých je k, získáme celkovou složitost našeho algoritmu jako $\mathcal{O}(k \cdot 2^k)$, z čehož dosazením získáme $\mathcal{O}(n \cdot log\ n)$.

Prostorová složitost: Jak jsme si odvodili u časové složitosti, potřebujeme uložit $n \cdot log\ n$ prvků, prostorová složitost je tedy $\mathcal{O}(n \cdot log\ n)$.

Online zpracování: Jelikož hned v prvním kroku rozdělíme pole na dvě poloviny (které musí být přibližně stejně velké), nemůžeme přidávat prvky v průběhu zpracování.

Třídění spojováním patří k nejjednodušším třídícím algoritmům využívajícím rekurzi. Zároveň dosahuje nejnižší časové složitosti, které může třídící algoritmus dosáhnout, čímž se řadí ke špičce třídících algoritmů.

Merge sort býval velice oblíbeným algoritmem pro třídění na starých počítačích s malou primární pamětí, protože má dobře předvídatelný způsob přístupu k datům (čte je postupně a po jednom) a nebrzdila ho tak pomalá trvalá úložiště. V dnešní době lze tohoto fenoménu využívat ve specifických případech při optimalizaci přístupu ke složité hierarchii pamětí současných počítačů.



Bucket sort — kyblíková metoda

V následujícím příkladu budeme třídit čísla od 100 do 999.

Nejprve budeme potřebovat 9 kyblíků. Kyblíky si označíme 1xx, 2xx .. 9xx. Čísla do kyblíků rozdělíme tak, že v kyblíku 1xx budou čísla 100-199, v kyblíku 2xx čísla 200-299 atd. Následně vezmeme druhou sadu kyblíků, kterých bude 10, a označíme je x0x, x1x, x2x .. x9x. Následně vezmeme kyblík 1xx a čísla z něho rozdělíme tak, aby druhá cifra těchto čísel odpovídala značce na kyblíku. V kyblíku x0x tedy budou čísla 100-109, v kyblíku x1x čísla 110-119 atd. Nyní vezmeme poslední sadu kyblíků, označených xx0 .. xx9 a čísla z kyblíku x0x do nich opět rozdělíme.

V kyblíku xx0 jsou tedy pouze záznamy s číslem 100, v kyblíku xx1 záznamy s číslem 101 atd. Vezmeme tedy kyblík xx0, čísla z něj zapíšeme do výstupního pole a kyblík vyprázdníme. Následně kyblík xx1, pak xx2 a tak dále, dokud nevyprázdníme celou sadu xxX. Pak vezmeme kyblík x1x a opět jej rozdělíme do třetí sady kyblíků. Tu opět zapíšeme do výstupního pole a pokračujeme čísly z kyblíku x2x. Takto postupně vyprázdníme celou sadu xXx. Potom vezmeme kyblík 2xx, rozdělíme je do sady xXx a postup opakujeme. Na konci máme ve výstupním poli setřízenou posloupnost.

Časová složitost: Každé číslo přiřadíme do každé ze sad Xxx, xXx a xxX právě jednou. Pokud kyblík reprezentujeme jako frontu, má zápis i odebrání čísla z kyblíku konstantní složitost. Máme n čísel, takže časová složitost algoritmu je $\mathcal{O}(3n) = \mathcal{O}(n)$. Poctivý čtenář si jistě všiml, že jsme si u Merge říkali, že obecný třídící algoritmus musí mít složitost alespoň $\mathcal{O}(n \cdot log\ n)$, jinak nemůže fungovat. Jak je to tedy možné?

Háček Bucket sortu je v tom, že máme omezenou *doménu* vstupních dat — možných hodnot, které lze zpracovávat, musí být konečný počet. Můžeme tedy například zpracovávat celá čísla od 1 do 100000, ale nedokážeme například setřídit reálná čísla mezi 0 a 1. Nejedná se tak o *obecný* třídící algoritmus.

Prostorová složitost: Pro odvození prostorové složitosti hodně záleží na datových strukturách, které použijeme pro ukládání v jednotlivých kyblících. Při vhodném použití zřetězených seznamů lze dosáhnout dokonce toho, že algoritmus pracuje na místě.

Online zpracování: První krok algoritmu je rozdělení celé vstupní posloupnosti do kyblíků sady Xxx. Potom už se vstupní posloupností nepracuje, a není tedy schopen online zpracování.

Třídění kyblíkovou metodou má velkou konstantní složku složitosti, která ale není zohledněna při odhadech složitosti pomocí $\mathcal{O}$ (v našem případě vždy proběhne 900× vnitřní cyklus, i

Algorithm 4 BucketSort

Vstup: pole čísel *vstupniPole*
  kyblik 1xx, 2xx, 3xx ... 9xx;
  kyblik x0x, x1x ... x9x;
  kyblik xx0, xx1 ... xx9;

  rozděl čísla ze *vstupniPole* do kyblíků Xxx
  for $i \leftarrow 1 \rightarrow 9$ **do**
    rozděl čísla z kyblíku ixx do kyblíků xXx
    for $j \leftarrow 0 \rightarrow 9$ **do**
      rozděl čísla z kyblíku ijx do kyblíků xxX
      for $k \leftarrow 0 \rightarrow 9$ **do**
        připiš čísla z kyblíku ijk na konec *vystupniPole*
        vyprázdni kyblík ijk
      end for
      vyprázdni kyblík ijx
    end for
    vyprázdni kyblík ixx
  end for
  return *vystupniPole*

když třídíme pole délky 5). V některých případech, zejména pro vstupní pole která mají hodně prvků se stejnými klíči, však může takovýto algoritmus pracovat rychleji než obecné třídící algoritmy. Budeme-li například třídit seznam o deseti tisících prvků s hodnotami 100-999, odhad složitosti našeho algoritmu bude 30 000 operací pro třídění a 900 operací pro režii cyklů. Pro srovnání, nejrychlejší obecné algoritmy potřebují asi $10\,000 \times log_2 10\,000 \times 4 = 40\,000 \times 16 = 640\,000$ operací pro samotné třídění, tedy jsou $20\times$ pomalejší.

Další vlastnosti algoritmů

Na konec úvodníku přidáváme několik dalších vlastností, které může třídící algoritmus mít. Nepřiřazujeme je ale jednotlivým algoritmům — to necháme na rozmyšlení pro vás.

Zachování pořadí: Pokud třídíme posloupnosti, kde se jedno číslo může vyskytovat vícekrát, je určitě zajímavé zajistit, aby se tyto prvky nepředbíhaly. Příkladem může být třeba výsledková listina závodu, kde vstupem je seznam závodníků setřízený podle abecedy. Výstupem třídícího algoritmu je seznam setřízený podle dosažených výsledků. Pokud ale dosáhnou dva závodníci shodný počet bodů, musí být v listině zapsáni podle abecedy. Pokud náš algoritmus zachovává pořadí, pak máme hotovo. V opačném případě by jsme ale museli algoritmus rozšířit tak, aby zohledňoval i abecedu. U většiny třídících algoritmů lze zachování pořadí zajistit pouhou opatrností při psaní kódu. Některé algoritmy (například Heap sort) však pořadí zachovávat nedokážou a je pak nutné vymyslet složitější postupy.

Počet zápisů vs počet porovnání: V některých případech nemusí být porovnání dvou prvků posloupnosti stejně rychlé jako přesun celých záznamů v paměti. Příkladem může být řazení velkých souborů podle abecedy. V takovém případě může být přesouvání prvků velmi pomalé a počet porovnání názvů souborů se ztratí v času potřebném pro zápis do paměti. V řešeních úloh této sady ale uvažujte pouze situaci, kdy časová náročnost zápisu je srovnatelná s náročností porovnání.

Nejlepší/nejhorší/průměrná složitost: Výkonnost některých algoritmů je závislá na tom, jestli je již vstupní posloupnost nějak uspořádaná. Příkladem může být Bubble sort, který již setřízenou posloupnost pouze jednou zkontroluje a vrátí. Složitost v takovém případě

je $\mathcal{O}(n)$. Naopak v nejhorším případě, kdy je posloupnost setřízená pozpátku, potřebuje n průchodů polem a ještě jeden průchod na kontrolu, tedy $\mathcal{O}(n^2)$. Pokud takovýto algoritmus používáme častěji, může nás zajímat taky průměrná složitost jednoho běhu. Její výpočet je však poměrně komplikovaný a zahrnuje spoustu pokročilé matematiky.

Jak psát řešení?

V úvodníku jsme si uvedli spoustu různých vlastností, které třídící algoritmy mohou mít. Nejdůležitější vlastností je však vždy časová složitost. Ve svých řešeních ji proto nezapomínejte uvádět! Pokud navrhnete algoritmus, který bude mít některou z popsaných vlastností splňovat a zvládnete to řádně zdůvodnit, můžete se nám samozřejmě také v řešení pochlubit. Pamatujte ale, že nejdůležitější je napsat řešení správně!

Zadání 2. sady úloh KSI (termín odevzdání: 25. 11. 2012)

Řešení zasílejte pomocí internetového systému na adrese `http://ksi.fi.muni.cz`.

Příklad 1: Lovci (10 bodů)

Pravekí ľudia sa raz denne vyberú loviť šabľozubých tigrov. Každý loví sám, a keďže je to náročná úloha, uloví maximálne jedného tigra. Na konci dňa lovci do jedného riadku na stene zaznamenajú, kto bol ako úspešný (napíše buď O – tiger alebo X – nič). Korisť skladujú v podzemí, aby ju na konci týždňa mohli prerozdeliť. Celý úlovok z najúspešnejšieho dňa dostane šaman. Problémom je zistiť najúspešnejší deň lovu, keďže nikto v jaskyni nevie počítať. Pomôžte lovcom nájsť spôsob, ako zistiť ich najúspešnejší deň s tým, že pravekí muži sú schopní upravovať pôvodnú "tabuľku"na stene jaskyne iba dvoma spôsobmi - gumovať jednotlivé záznamy a zapisovať na vygumované miesto.

Takto by vyzeral záznam úlovkov za týždeň, keby bolo lovcov 5.

O X O O X
X X O X X
X O X O X
O X O X O
O O O O X
O O X X O
O O X O X

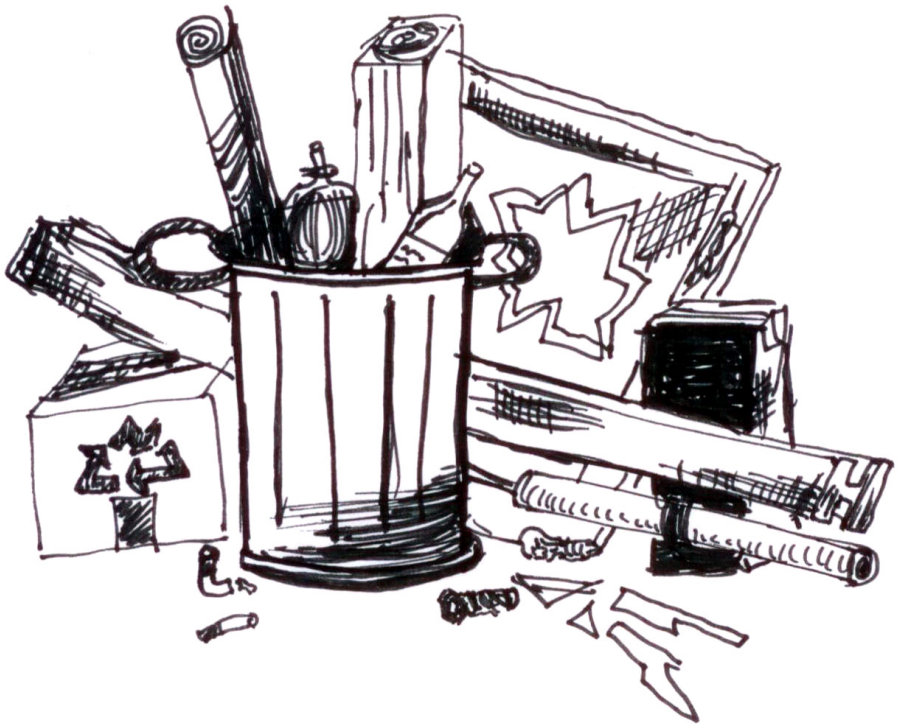
Najúspešnejší bol v tomto prípade 6. deň.

Ako riešenie nám posielajte algoritmus, ktorý budú môcť použiť aj pravekí ľudia (preto napríklad jednoduché spočítanie symbolov O v jednotlivých riadkoch nie je povolené) vo forme pseudokódu s rozumným komentárom.

Příklad 2: Informatický pětiboj (10 bodů)

Organizátoři KSI se rozhodli uspořádat velkolepý závod. Disciplíny budou velmi různorodé: jízda na kole, programování, hod oštěpem, šifrování a hraní počítačových her. Pravidla závodu jsou jednoduchá: V pěti disciplínách soutěží n závodníků (a protože informatiků je spousta, očekáváme, že n bude dost vysoké číslo), přičemž každý z nich může za každou disciplínu získat 1-10 bodů, celkem tedy 50 bodů. Vyhraje ten, který získá nejvíce bodů.

Vaším úkolem je vytvořit co nejefektivnější algoritmus, který seřadí závodníky podle počtu bodů a pro každé umístění vypíše číslo závodníka a počet bodů, který získal. V případě, že má více závodníků stejný počet bodů, umístí se na stejném místě.



Příklad 3: Triedenie odpadu (10 bodů)

Brnenskí bezdomovci Pepa, Karel a Jožo raz šli okolo Fakulty informatiky. Ako ju videli, takú rozbúranú, pomysleli si, že na takom mieste určite zostala kopa cenných informatických vecí. I nelenili a hneď sa vydali privyrobiť si trošku. Ich plán bol nasledovný: v noci sa vkradnú na stavenisko a ukradnú toho čo najviac, koľko dokážu uniesť. Následne si lup rozdelia a predajú ho do výkupu kovu, skla a papiera.

V noci prebehlo všetko hladko. Podarilo sa im urkadnúť 12 vecí, pričom o každej rovno vedia, koľko kovu, skla a papiera obsahuje:

	kov	sklo	papier
1. Diplomovku	0,2 kg	0 kg	0,3 kg
2. Plazmovú guľu	0,5 kg	2 kg	0 kg
3. Dioptrie veľkosť 15.	0,1 kg	3 kg	0 kg
4. Kotúč 8-bitovej diernej pásky	0 kg	1,5 kg	9 kg
5. Von Neumanovu schému	0.5 kg	0,5 kg	5 kg
6. Pizzu v krabici	0,05 kg	0 kg	0,5 kg
7. Zásuvku s nápisom FIMU	0,2 kg	0,1 kg	0 kg
8. Počítačový bug	0,5 kg	1 kg	0 kg
9. Lineárnu zložitosť	3 kg	0 kg	9 kg
10. Petriho sieť	2 kg	2 kg	2 kg
11. ASCII tabuľku znakov v rámiku	0,5 kg	0,5 kg	0,3 kg
12. Prototyp robota Karla	1,5 kg	1,3 kg	1 kg

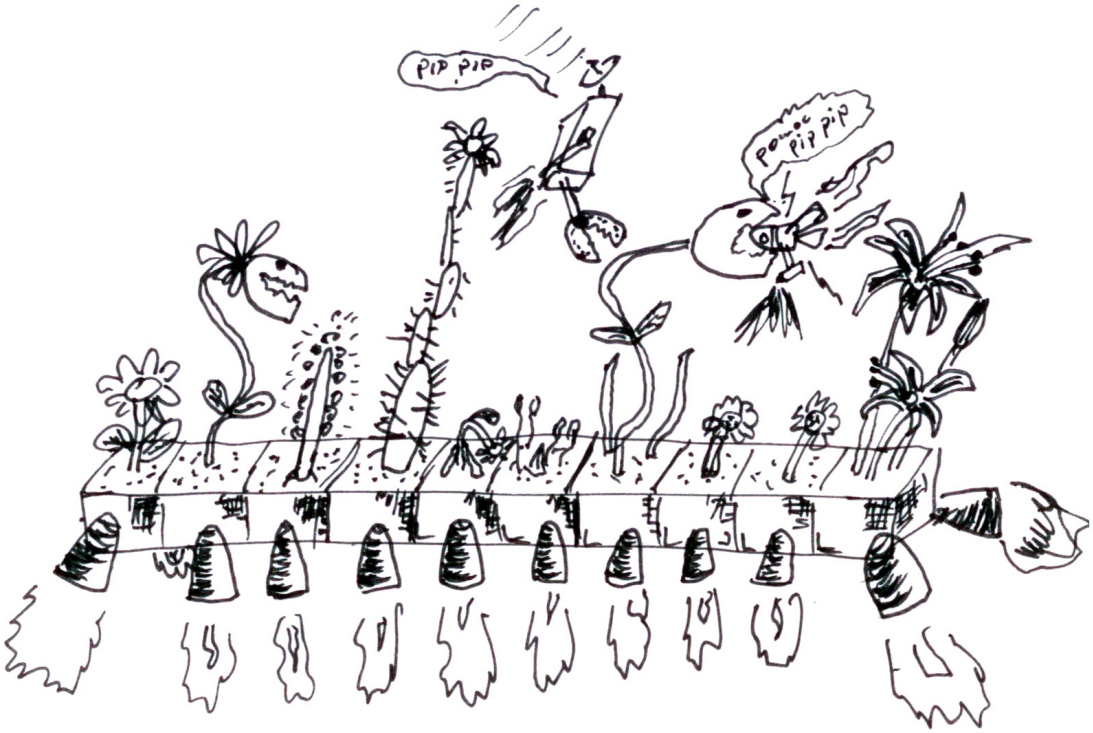
Následne sa dohodli, že si veci rozdelia na tri rovnako veľké kopy po štyri predmety, a každý pôjde so svojou kopou do iného výkupu. Lenže aby zarobili čo najviac, potrebujú svoj lup rozdeliť čo najefektívnejšie. A s tým si nevedia rady. Pomôžeš im?

Tvojou úlohou bude rozdeliť ich lup na 3 kopy po 4 prvky. Veci z prvej kopy sa predajú vo výkupe kovu, z druhej vo výkupe skla a z tretej vo výkupe papiera. Cieľom je samozrejme, aby sa predajom týchto vecí dokopy zarobilo najviac, pričom za veci z prvej kopy dostanú peniaze len za ich kovové časti, a to konkrétne 110kč/kg, za veci v druhej kope dostanú 60kč/kg za ich sklenené časti, a za veci v tretej kope dostanú 10kč/kg za ich papierové časti. Ako riešenie nám pošli rozdelenie predmetov na tieto 3 kopy.

Příklad 4: Farmár Gusta (10 bodů)

Vo štvrtom príklade budeme pomáhať istému mimoriadne podnikavému farmárovi Gustovi, ktorý sa rozhodol rozšíriť svoj biznis pestovania rastlín. Gusto je pestovateľom s najväčším počtom rôznych rastlín a do svojej zbierky plánuje pridať ďalšie. Aby sa novým rastlinám darilo, rád by ich zasadil na najúrodnejšie pole, aké má. Na každom poli ale rastú iné rastliny a porovnávať úrodnosť polí na rôznych druhoch rastlín nevedie k žiadnemu výsledku. Rozhodol sa preto, že nájde také druhy rastlín, ktoré má vysadené na všetkých poliach, aby na nich porovnal ich úrodnosť. Pre každé pole vedie zoznam, na ktorý pripisuje rastliny vždy, keď ich zasadí. Rôznych druhov na každom poli sú však stovky a keďže Gusto podniká vo veľkom, počet polí tiež nie je zanedbateľný.

Aby ste pomohli Gustovi, navrhnite algoritmus, ktorý dokáže nájsť spoločné prvky v týchto zoznamoch. Algoritmus by mal fungovať pre ľubovoľný počet zoznamov a jeho úlohou je nájsť všetky prvky, ktoré sa nachádzajú v každom zozname. Poradie prvkov v zoznamoch môže byť ľubovoľné. Pri návrhu myslite na efektivitu - Gusto podniká skutočne vo veľkom!



Příklad 5: Abstraktní kód (10 bodů)

Hezky napsaný kód je jako krásná báseň a naopak slátaný (a navíc neokomentovaný) kód bez hlavy a paty připomíná abstraktní poezii, které nerozumí ani samotný autor. Program se pak těžko ladí, těžko udržuje, těžko optimalizuje a co je nejhorší: těžko se opravuje. Bohužel i takové nacházíme neustále mezi došlými řešeními a my organizátoři pak musíme aktivovat veškeré krypotaanalytické schopnosti a strávit spoustu času jejich luštěním.

Abyste si příště rozmysleli, jak si na svém kódu dáte záležet, nabídneme Vám malou ochutnávku příšerného kódu. A Vaším úkolem bude to, co obvykle děláme my - zjistit, co program dělá a vysvětlit, proč je uvedené řešení neefektivní.

```
procedure Zatazeno (S, U, N)
begin
  for i := N*(-1) to -1 do
  begin
    picasso := N+i+1
    braque := (-1)*i
    U[picasso] := S[braque]
  end
```

```
end

procedure Oblacno (S, U, N)
begin
  S[N] :=
U~for i := N downto 2 do
  begin
    if (S[i] < S[i-1]) then
    begin
      op := S[i]; art := S[i-1]; S[i] := art; art := op; S[i-1] := art
    end
  end
end

procedure ProvedNeco (X, Y, Z, N, M)
begin
  epizeuxis := N-M > M-N
  repeat
    Zatazeno(X, Z, N)
  until (epizeuxis || !epizeuxis)
  for i := 1 to M do
  begin
    sarkasmus := N+i+1
    Oblacno(Z, Y[sarkasmus-N-1], sarkasmus-1)
  end
  for t := 1 to N do
  begin
    for k~:= 1 to N+M-1 do
    begin
      if (Z[k] > Z[k+1]) then
      begin
        kupka := Z[k]; Z[k] := Z[k+1]; Z[k+1] := kupka
      end
    end
  end
end
```

Vaše řešení by tedy mělo obsahovat:

- Co se stane, když zavoláme funkci ProvedNeco(X, Y, Z, N, M), kde X, Y a Z jsou pole čísel (indexovaná od 1) a N, M jsou vysoká celá čísla (co označují, zjistíte pohledem do zdrojového kódu).
- Proč je daný způsob řešení neefektivní a jak by to šlo udělat lépe.
- Implementace Vašeho řešení, které bude efektivní z hlediska průměrné časové složitosti. Pokud možno v hezkém pseudokódu a okomentované, ať s tím nemám při opravování moc práce :-)

A to je z druhé sady KSI vše. Přejeme ti hodně úspěchů při řešení, a když budeš mít jakékoliv otázky, neváhej se na nás obrátit e-mailem na adresu `ksi@fi.muni.cz` nebo v diskuzním fóru na webových stránkách.

Termín odevzdání 2. sady úloh KSI: 25. 11. 2012

http://ksi.fi.muni.cz