

IV003 2014, SADA 2, PRÍKLAD 1

JAKUB SENKO (373902), ŠTEFAN UHERČÍK (374375)

Zaveďme všeobecnú reprezentáciu budov. Každá uvažovaná budova sa dá reprezentovať ako množina dvojíc (x_k, h_k) , určujúcich výšku budovy h na súradnici x . Zápis sa dá zjednodušiť usporiadaním bodov vzostupne podľa x . Stačí uvažovať len tie dvojice, ktoré označujú miesto, v ktorom nastáva zmena výšky budovy. Tento zápis je ekvivalentný so zápisom použitým v zadaní

$$(1, 5, 5) \sim ((1, 5), (5, 0)) \quad (0.1)$$

ide len o vnútornú reprezentáciu za účelom zjednodušenia algoritmu.

MERGE

Uvažujme algoritmus MERGE, ktorý z reprezentácie dvoch budov vypočíta reprezentáciu ich siluety.

Algoritmus využíva object BUILDING_ITERATOR pomocou ktorého je možné postupne prechádzať reprezentáciou danej budovy. Obsahuje tri metódy.

NEXT_COORDINATE_EXISTS a NEXT_COORDINATE_POSITION sú triviálne a neposúvajú pozíciu iterátora. Tretia metóda, GET_HEIGHT(x) vráti výšku budovy na zadanej súradnici. Táto metóda spôsobí dostatočný posun iterátora v prípade, že daná pozícia je väčšia alebo rovná ako NEXT_COORDINATE_POSITION. Keďže iterátor je jednorázový, túto metódu je nie je možné zavolať s argumentom menším ako v predchádzajúcom volaní. Iterátor si jednoducho pamätá poslednú výšku.

Samotný MERGE pracuje s dvoma iterátormi, pre každú budovu jeden a výstup postupne ukladá do samostatného zoznamu. Základom je *while* smyčka, ktorá sa vykoná ak aspoň pre jeden z iterátorov platí NEXT_COORDINATE_EXISTS. Algoritmus potom vybere menšie x z NEXT_COORDINATE_POSITION a zavolá metódu GET_HEIGHT na oboch iterátoroch. Následne vybere väčšiu z výšok, h a zavolá funkciu TRY_ADD, ktorá jednoducho vloží novú súradnicu (x, h) do výsledného zoznamu v prípade, že sa výška siluety zmenila (čo nemusí nastať).

Tento algoritmus funguje pre ľubovoľné reprezentácie s dĺžkou n_1, n_2 v čase $\mathcal{O}(n_1 + n_2)$, čo je $\mathcal{O}(n)$ pre budovy s rovnako veľkou reprezentáciou. Zdôvodnenie je jednoduché - využíva jednosmerný iterátor na jedno použitie pre každú reprezentáciu - a teda každú súradnicu spracuje práve raz. Algoritmus je konečný pretože pri každom priechode cyklom metóda GET_HEIGHT posunie aspoň jeden z iterátorov.

ROZDEĽ A PANUJ

Výslednú siluetu dosiahneme aplikovaním funkcie MERGE na vhodné podproblémy. Toto delenie funguje rovnako ako pri algoritme *merge sort*. Funkcia COMPUTE_SILHOUETTE zoberie ako argument množinu reprezentácií budov. Ak táto množina obsahuje jednu budovu, tak ju vráti. Ak dve budovy, zavolá na nich MERGE a vráti výsledok. Ak viac, rozdelí množinu na dve rovnaké (s rozdielom jednej budovy v prípade nepárneho počtu) množiny, rekurzívne sa na oboch zavolá a výsledok znovu spojí pomocou MERGE a vráti. Týmto spôsobom funkcia COMPUTE_SILHOUETTE vždy vráti merge všetkých spojích argumentov (merge nezávisí na poradí).

Zložitosť závisí na počte MERGE operácii a veľkosti ich vstupu. Na každej úrovni rekurzie je suma veľkosti všetkých reprezentácií rovnaká (n dĺžky 2 na začiatku vs dve dlhé n na konci, kde n je počet budov) a počet úrovní je $\log_2 n$. Výsledná zložitosť je teda $\mathcal{O}(n \log n)$

IV003 2014, SADA 2, PRÍKLAD 2

JAKUB SENKO (373902), ŠTEFAN UHERČÍK (374375)

Slovník pojmov:

slovo - reťazec, pre ktorý vráti funkcia dict true

veta - postupnosť zreťazených slov, medzi jednotlivými slovami sa nenachádzajú žiadne iné znaky

Algoritmus overí všetky možné rozdelenia vstupného reťazca na 2 časti, prefix a suffix. Ak je daný reťazec veta, existuje aspoň jedno rozdelenie také, že suffix je slovo a prefix je znovu veta. Tento test sa vykoná v tele cyklu, kde sa rekurzívne použije IS_SENTENCE na danom prefixe vstupného slova. Premenná *result* je true práve vtedy ak je aspoň jeden z týchto testov je true.

```
1: function IS_SENTENCE( $w[1 \dots n]$ )
2:   result = DICT( $w[1 \dots n]$ );
3:   for  $i = 1 \dots n$  do
4:     item = IS_SENTENCE( $w[1 \dots i]$ )  $\wedge$  DICT( $w[i + 1 \dots n]$ );
5:     result = result  $\vee$  item;
6:   end for
7:   return result;
8: end function
```

Pri každom z rekurzívnych volaní je argument funkcie IS_SENTENCE *vždy prefixom* argumentu volajúcej funkcie. Pri tomto rekurzívnom algoritme môže existovať v strome rekurzie viacero ciest k volaniu IS_SENTENCE s rovnakým argumentom, avšak každý reťazec má konečné množstvo prefixov, konkrétne $n - 1$. Preto stačí vykonať iba lineárne množstvo volaní. Tieto volania sa dajú *usporiadať* tak, že sa funkcia postupne volá pre rastúci prefix. Táto myšlienka je základom nasledujúcej dynamickej varianty funkcie IS_SENTENCE.

Predstavme si nasledovný prípad:

Na vstup dostane algoritmus reťazec o dĺžke n .

Pri overovaní jednotlivých rozdelení nájde slovo o dĺžke a (a zároveň sa rekurzívne zanorí na prefixe o dĺžke $(n-a)$) a ďalej pokračuje v overovaní ďalších rozdelení (s prefixami dĺžky $(n-a+1), (n-a+2), \dots$).

Algoritmus sa rekurzívne zavolá na reťazci o dĺžke $n-a$. Pri týchto volaniach však overuje prefixy s dĺžkami $1 \dots n-a$. Tieto prefixy však overoval aj predošlý priechod algoritmu.

Môžeme povedať, že volanie funkcie IS_SENTENCE aplikované na reťazci dĺžky n je závislé na všetkých možných volaniach funkcie IS_SENTENCE aplikovaných na prefixoch

tohto reťazca.

Z tohoto dôvodu je výhodné, ak vypočítame `IS_SENTENCE` na prefixoch pôvodného reťazca a výsledky týchto volaní si uložíme do rovnomenného asociatívneho poľa. Kľúč tohto poľa bude tvorený prefixom pôvodného reťazca, hodnota bude typu boolean.

```
1: function VERIFY_SENTENCE( $w[1 \dots n]$ )
2:   for  $i = 1 \dots n$  do
3:     item = [] of boolean
4:     result = DICT( $w[1 \dots i]$ );
5:     for  $j = 1 \dots i - 1$  do
6:       item = IS_SENTENCE( $w[1 \dots j]$ )  $\wedge$  DICT( $w[j + 1 \dots i - 1]$ );
7:       result = result  $\vee$  item;
8:     end for
9:     IS_SENTENCE( $w[1 \dots i]$ ) = result;
10:  end for
11:  return IS_SENTENCE( $w[1 \dots n]$ );
12: end function
```

KOREKTNOSŤ:

KONVERGENCIA:

Prvý cyklus sa vykoná pre každý prefix slova w . V cykle sa hodnota premennej i reprezentujúcej dĺžku prefixu pri každom priechode zvýši o 1 a iterovanie skončí, keď premenná i dosiahne hodnotu n .

Vnorený cyklus sa vykoná pre každý prefix tohto prefixu.

Jediné miesta, u ktorých hrozí, že algoritmus nezastaví, sú volania cyklov.

V cykle s iterujúcou premennou j sa zaručene pri každom priechode zvýši hodnota premennej j a iterovanie skončí, keď premenná j dosiahne hodnotu $i-1$.

PARCIÁLNA KOREKTNOSŤ:

Dôkaz, že algoritmus vráti true pri validnej postupnosti slov pomocou matematickej indukcie:

Bázový krok:

$S_0 = w_0$ je veta ktorá sa skladá z jedného slova w_0 (w_0 sa nachádza v slovníku). Pre takúto vetu vráti algoritmus true. Dôvod: položky subresults pre prvok `IS_SENTENCE(w_0)`

budú obsahovať položku $dict(w_0)$, ktorá sa vyhodnotí na true. Na položky je aplikovaný operátor logiký súčet a pretože obsahujú minimálne jednu položku z hodnotou true, algoritmus vráti true.

Indukčný krok:

Predpokladáme, že pre vetu s_1 zloženú z k slov: $s_1 = w_1.w_2...w_k$ vráti korektnú odpoveď true.

Predpokladáme, že pre vetu s_2 zloženú $s_2 = s_1.w_{k+1}$ algoritmus taktiež vráti true.

Jeden z prefixov vety s_2 musí byť reťazec zložený zo slov $w_1...w_k$. Keďže algoritmus prechádza všetky prefixy, nastane situácia, jedna z položiek v poli subresults pre prvok s_2 bude $IS_SENTENCE(s_1) \text{ logical_and } dict(w_{k+1})$.

ZLOŽITOSŤ

Cyklus s iterujúcou premennou i bude vykonaný n -krát.

Vnorený cyklus s iterujúcou premennou j bude vykonaný $(i-1)$ krát, a plaží, že $(i-1) < n$. Všetky operácie použité v ňom majú konštantnú zložitosť. Celková zložitosť bude teda: $\mathcal{O}(n^2)$.

IV003 2014, SADA 2, PRÍKLAD 3

JAKUB SENKO (373902), ŠTEFAN UHERČÍK (374375)

Vstupom algoritmu bude: pole pravdepodobností: $C[p_1, \dots, p_n]$

k - počet padnutých orlov

PVD - pravdepodobnostná funkcia

Problém je možné definovať nasledovne: Vypočítať pravdepodobnosť, že v poli padne k orlov z n mincí

Táto pravdepodobnosť je ekvivalentná súčtu pravdepodobností nasledovných prípadov:

- 1.) pravdepodobnosť prípadu, že posledná minca bude orol
táto pravdepodobnosť je ekvivalentná súčinu čísla p_n a pravdepodobnosti, že medzi prvými $n-1$ mincami bude $k-1$ orlov
- 2.) pravdepodobnosť prípadu, že posledná minca nebude orol
táto pravdepodobnosť je ekvivalentná súčinu čísla $(1 - p_n)$ a pravdepodobnosti, že medzi prvými $n-1$ mincami bude k orlov

Tento poznatok nám umožňuje definovať jednoduchý rekurzívny algoritmus (v ktorom zároveň ošetrujeme krajné prípady $n=k$ a $n=0$)

```
1: function PVD( $C[p_1, \dots, p_n], k$ )
2:   if  $k=0$  then
3:     return  $PVD(C[p_1, \dots, p_{n-1}], 0) * (1 - p_n)$ ;
4:   end if
5:   if  $k=n$  then
6:     return  $PVD(C[p_1, \dots, p_{n-1}], k - 1) * p(n)$ ;
7:   end if
8:   return  $PVD(C[p_1, \dots, p_{n-1}], k - 1) * p_n + PVD(C[p_1, \dots, p_{n-1}], k) * (1 - p_n)$ ;
9: end function
```

Ak rozpíšeme vetvenie algoritmu vykonávanie bude vyzeráť približne nasledovne:

$$PVD(C[p_1, \dots, p_n], k) = PVD(C[p_1, \dots, p_{n-1}], k-1) * p_n + PVD(C[p_1, \dots, p_{n-1}], k) * (1 - p_n)$$

$$\begin{aligned} PVD(C[p_1, \dots, p_{n-1}], k) = \\ PVD(C[p_1, \dots, p_{n-2}], k-1) * p_{n-1} + PVD(C[p_1, \dots, p_{n-2}], k) * (1 - p_{n-1}) \end{aligned}$$

$$\begin{aligned} PVD(C[p_1, \dots, p_{n-1}], k-1) = \\ PVD(C[p_1, \dots, p_{n-2}], k-2) * p_{n-1} + PVD(C[p_1, \dots, p_{n-2}], k-1) * (1 - p_{n-1}) \dots \end{aligned}$$

Z predchádzajúceho zápisu volaní funkcií je možné vidieť, že $PVD(C[p_1, \dots, p_{n-1}], k-1)$ sa zavolá 2 krát na druhej úrovni rekurzívneho stromu. Využijeme techniku dynamického programovania, aby sme sa vyhli opakovanému volaniu funkcie PVD na rovnakých parametroch. Z algoritmu je zreteľné, že volanie funkcie PVD, ktorá berie ako parameter pole o dĺžke a , je závislá výlučne na volaniach funkcií PVD, ktoré berú ako parameter pole o dĺžke $a-1$.

Z tohoto dôvodu je výhodné, ak vypočítame najprv. funkcie s parametrami $PVD(C[p_1]), 0$, $PVD(C[p_1], 1)$, $PVD(C[p_1, p_2]), 0$,...

Pre tento účel vytvoríme asociatívne pole s názvom PVD, v ktorom kľúče budú mať tvar: $(C[p(1), \dots, p(n)], k)$ a hodnoty budú obsahovať napočítanú pravdepodobnosť. Na naplnenie tohto poľa vytvoríme jednoduchú nerekurzívnu funkciu:

Ich výsledky si budem ukladať do asociatívneho poľa a postupným volaním sa dopracujem k hodnote $PVD(C[p(1), \dots, p(n)], k)$, ktorá je výsledkom celého problému.

```

1: function COUNT_PVD( $C[p_1, \dots, p_n], k$ )
2:    $PVD([p_1], 0) = p_1$ ;
3:    $PVD([p_1], 1) = (1 - p_1)$ ;
4:   for  $i = 1..n$  do
5:      $bottom = \max(0, k - (n - i))$ ;
6:      $up = \min(i, k)$ ;
7:     for  $j = bottom .. up$  do
8:       if  $j == 0$  then
9:          $PVD(C[p_1, \dots, p_i], j) = PVD(C[p_1, \dots, p_{i-1}], 0) * (1 - p_i)$ ;
10:      else
11:        if  $k == n$  then
12:           $PVD(C[p_1, \dots, p_i], j) = PVD(C[p_1, \dots, p_{i-1}], j - 1) * p_i$ ;
13:        else
14:           $PVD(C[p_1, \dots, p_i], j) =$ 
15:             $PVD(C[p_1, \dots, p_{i-1}], j - 1) * p_i + PVD(C[p_1, \dots, p_{i-1}], j) * (1 - p_i)$ ;
16:        end if
17:      end if
18:    end for
19:  end for
20:  return  $PVD(C[p_1, \dots, p_n], k)$ ;
21: end function

```

Zložitosť: Cyklus s iterujúcou premennou i sa vykoná n krát. V ňom sa vnorený cyklus iterujúcou premennou j vykoná vždy $(up - bottom)$ krát. V každom cykle bude hodnota premennej $bottom$ minimálne 0 a hodnota premennej up maximálne k , z čoho vyplýva,

že počet týchto cyklov bude maximálne k . Je zaručené, že $k < n$ a teda zložitosť celého algoritmu bude patriť do triedy $\mathcal{O}(n^2)$

IV003 2014, SADA 2, PRÍKLAD 4

JAKUB SENKO (373902), ŠTEFAN UHERČÍK (374375)

Cenu optimálneho rozdelenia áut do autosalónov je možné vypočítať pomocou jednoduchého rekurzívneho algoritmu. Maticu C chápeme ako n trojíc, pričom každá je v samostatnom riadku. Číslo auta je zároveň jeho index riadku v matici cien. Parameter n v algoritme `bestPrice` udáva počet áut ktoré ešte neboli priradené do autosalónu a keďže sa autá priradujú postupne, tak udáva zároveň aj číslo nasledujúceho nepriradeného auta. Parameter `freePlaces` udáva počet neobsadených miest v jednotlivých autosalónoch (číslo na indexe i v tomto poli udáva počet voľných miest v autosalóne i). V každom rekurzívnom volaní priradíme auto do každého autosalónu az týchto priradení vrátime maximálnu cenu. Rekurzia sa zastaví v prípade že sme priradili už všetky autá. V nasledujúcich algoritmoch predpokladáme, že počet áut je deliteľný počtom autosalónov.

```
1: function BEST_PRICE( $n$ ,freePlaces[])
2:   if  $n == 0$  then
3:     return 0
4:   end if
5:   values = [] of number
6:   for  $i = 1 .. \text{freePlaces.size}$  do
7:     if freePlaces[i] != 0 then
8:       freePlacesCopy = copy of freePlaces;
9:       freePlacesCopy[i] = freePlacesCopy[i] - 1;
10:      values[i] = best_price( $n-1$ ,freePlacesCopy) + C[n][i];
11:    end if
12:  end for
13:  return max(values);
14: end function
```

TECHNIKA DYNAMICKÉHO PROGRAMOVANIA

Každé volanie funkcie `bestPrice` na matici s počtom riadkov k potrebuje výsledky volaní funkcie `bestPrice` na matici s počtom riadkov $k-1$. Toto nám jasne definuje závislosť a teda možné usporiadanie volaní.

Vytvoríme štruktúru `bestPriceData`, ktorá bude mať typ asociatívneho poľa, v ktorom:
// kľúče budú zodpovedať možným parametrom funkcie `best_price` - teda dvojica `(n,freePlaces[])` hodnoty budú zodpovedať vypočítanej najvyššej možnej cene

Aby sme zaznamenali všetky možné kombinácie parametrov n a `freePlaces`, vo funkcii `bestPrice` sme vytvorili funkciu `findSpecialPermutations`. Táto funkcia vráti všetky možnosti ako môže vyzeráť pole `freePlaces` pre sum áut v prípade že do jedného autosalónu môžeme dať maximálne `threshold` áut. Aby sme ju popísali viac formálne, pre vstupné parametre `sum` a `threshold` hľadáme permutácie trojice čísel, pričom tieto čísla sú priro-

dzené alebo 0. Tieto čísla musia dávať súčet rovný parametru sum, ale zároveň žiadne z nich nemôže byť väčšie, ako hodnota threshold.

```

function FINDSPECIALPERMUTATIONS(sum,threshold)
  if sum = 0 then
    return [0,0,0]
  end if
  permutations = [] of [];
  bottom1 = max(0,sum - 2*threshold);
  upper1 = min(sum,threshold);
  for i = bottom1 .. upper1 do
    permutation = [] of length 3
    permutation[1] = i;
    bottom2 = max(0,sum - i - threshold);
    upper2 = min(sum - i,threshold);
    for j=bottom2 .. upper2 do
      permutation[2] = j;
      permutation[3] = sum - (i+j);
    end for
    permutations.add(permutation);
  end for
  return permutations;
end function

```

Nasledovná nerekurzívna funkcia postupne priradí do štruktúry bestPriceData hodnoty najlepších možných cien pre danú konfiguráciu. Vonkajší for cyklus s iterujúcou premennou i v každom priechode spočíta maximalne ceny pre všetky distribúcie pr i aut. Využívame hodnoty v bestPriceData vypočítané v predchádzajúcich priechodoch cyklom.

```

1: function COUNTBESTPRICE(C)
2:   bestPriceData[ (0, [0,0,0]) ] = 0;
3:   for i = 1 .. C.rows do
4:     permutations = findSpecialPermutations(i,C.rows/3);
5:     for permutation in permutations do
6:       items = [] of number;
7:       for j = 1 .. permutation.size do
8:         modifiedPermutation = copy of permutation
9:         if modifiedPermutation[j]>0 then
10:          modifiedPermutation[j] = modifiedPermutation[j] - 1;
11:          items.add(bestPriceData[(i-1,modifiedPermutation)] + C[i][j]);
12:        end if
13:      end for
14:      bestPriceData[(i,permutation)] = max(items);

```

```

15:         end for
16:     end for
17:     return bestPriceData[C.rows,[C.rows/3,C.rows/3,C.rows/3]];
18: end function

```

KOREKTNOST'

KONVERGENCIA

Algoritmus môže nekonvergovať len pri vyhodnotení podmienky jedného z troch cyklov. Cyklus s iterujúcou premennou j sa skončí pretože `permutation.size` je kladná konštanta. Cyklus s iterujúcou premennou `permutation` skončí pretože funkcia `findSpecialPermutations` vráti konečný počet položiek. Vonkajší cyklus pretože `C.rows` je kladná konštanta.

PARCIÁLNA KOREKTNOST'

Dokážeme matematickou indukciou podľa počtu áut pre ktorý sme už vypočítali maximálnu cenu (vonkajší cyklus).

Bázový krok:

Pre $i = 0$ Táto situácia je riešená pred cyklom naplnením pola `bestPriceData` hodnotou $(0, [0,0,0])$.

Indukčný krok:

Pre $i = k$ Predpokladáme že `bestPriceData` obsahuje správne vypočítané maximálne ceny pre všetky možné konfigurácie `freePlaces` pre počet áut k , označme ich `currentFreePlaces`. Vnútny for cyklus s iterujúcou premennou `permutation` vypočíta maximálne ceny pre všetky možné konfigurácie `freePlaces`. Každá maximálna cena sa vypočíta ako maximum z už vypočítaných konfigurácií. Pretože for cyklus prebehne pre všetky možné permutácie pre k áut, budú dostupné pre $k+1$ iteráciu cyklu.

1 ZLOŽITOSŤ

funkcia `findSpecialPermutations`

vo funkcii sa nachádzajú dva cykly: počet vykonaní prvého cyklu je $(upper1 - bottom1)$, rozdiel týchto premenných bude vždy menší ako počet áut. Ekvivalentné tvrdenie môžeme spraviť pre druhý cyklus s počtom vykonaní $(upper1 - bottom1)$. Každá operácia použitá v cykloch má konštantnú zložitosť, preto je jej celková zložitosť $\mathcal{O}(n^2)$.

funkcia `countBestPrice`

vonkajší cyklus sa vykoná pre každé auto práve raz, teda `C.row` krát.

Zo zložistosti funkcie `findSpecialPermutations` môžeme povedať, že počet prvkov v poli `permutations` je maximálne kvadraticky závislý na premennej i .

Zložitosť operácií v cykle s iterujúcou premennou j je konštatný, vzhľadom na to, že hodnota `permutation.size` zodpovedá počtu autosalónov a ten je konštantný počas celého vykonávania programu. Zložitosť cyklu s iterujúcou premennou `permutation` má preto zložitosť $\mathcal{O}(n^2)$ (kvôli počtu prvkov v poli `permutations`).

Predošlé tvrdenia môžeme zredukovať na tvrdenie, že v cykle s iterujúcou premennou i sa vždy vykonajú dve operácie so zložitosťou $\mathcal{O}(n^2)$. Jeho zložitosť je teda $\mathcal{O}(n^3)$.

ROZŠÍRENIE ALGORITMU

Pre krátkosť tento algoritmus počíta len výšku najlepšej ceny za akú je možné autá predať, ale je triviálne modifikovateľný aby si spolu s cenou pamätal aj konkrétne priradenie áut do autosalónov. Namiesto ceny je stačí uložiť do `bestPriceData` dvojicu (cena, priradenie). Priradenie je pole o dĺžke `C.rows` a hodnota na indexe i v tomto poli vyjadruje číslo autosalónu do ktorého bude auto priradené, prípadne 0 ak ešte nie je priradené. Nasledujúci algoritmus toto implementuje:

```

1: function COUNTBESTPRICE(C) bestPriceData[(0,[0,0,0])] = 0;
2:   for i = 1 .. C.rows do
3:     permutations = findSpecialPermutations(i,C.rows/3);
4:     for permutation in permutations do
5:       items = [] of item (price -> number, distribution -> [] of number);
6:       for j = 1 .. permutation.size do
7:         modifiedPermutation = copy of permutation
8:         if modifiedPermutation[j]>0 then
9:           modifiedPermutation[j] = modifiedPermutation[j] - 1;
10:          item it;
11:          aPrice = bestPriceData[(i-1,modifiedPermutation)];
12:          it.price = aPrice.price + C[i][j];
13:          it.distribution = aPrice.distribution;
14:          it.distribution[i] = j;
15:          items.add(it);
16:        end if
17:      end for
18:      highestPriceIndex = index into items for item with highest price;
19:      bestPriceData[(i,permutation)] = items[highestPriceIndex];
20:    end for
21:  end for
22:  return bestPriceData[C.rows,[C.rows/3,C.rows/3,C.rows/3]];
23: end function

```

Algoritmus je rozšíriteľný pre väčší počet autosalónov, je však potrebné zmeniť implementáciu funkcie `findSpecialPermutations`.

IV003 2014, SADA 2, PRÍKLAD 5

JAKUB SENKO (373902), ŠTEFAN UHERČÍK (374375)

Hladový algoritmus nemusí nájsť optimálne riešenie pre druhý a tretí problém.

Protipríklad:

Dokument s dĺžkou riadku: 12

Zoznam slov obsahuje slová s dĺžkami 5,4,4,12

Algoritmus umiestni slová nasledovne:

5,4

4

12

Druhý slovný problém:

Celková penalizácia bude $3^2 + 8^2 + 0 = 9 + 64 = 73$

Optimálne riešenie je však:

5

4,4

12

pri ktorom bude celková penalizácia $7^2 + 4^2 = 49 + 16 = 64$

Tretí slovný problém:

Celková penalizácia bude: $\max(3,8,0) = 8$

Optimálne riešenie je však:

5

4,4

12

pri ktorom bude celková penalizácia $\max(7,4,0) = 7$