Homework 2

Jakub Senko, Štefan Uherčík

13. apríla 2014

Príklad 1

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

$$(1, \mathbf{5}, 5) \sim ((1, 5), (5, 0)) \tag{0.1}$$

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 prechádzat 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 zadaná pozícia je väčšia alebo rovná ako NEXT_COORDINATE_POSITION. Kedže 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 s 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 ľubovolné 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 reprezentácii 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 spojich 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 $\mathcal{O}(n \log n)$

Príklad 2

Pre riešenie tohto problému existuje jednoduchý rekurzívny algoritmus, jeho zložitosť je však exponenciálna z dôvodu opakovaného volania funkcie na rovnakých podproblémoch. Aby sme tomuto zabránili, využili sme závislosti medzi jednotlivými volaniami funkcie IS_SENTENCE. Volania funkcie IS_SENTENCE s konkrétnymi parametrami budeme ukladať do dátovej štruktúry typu asociatívne pole (hash tabulka). Vyhľadávanie v hash tabulke má konštantnú zložitosť, preto aj každé volanie funkcie IS_SENTENCE s rovnakým parametrom bude mať konštantnú zložitosť.

```
1: function IS_SENTENCE(w[1..n])
2:     items = [] of boolean
3:     items.add(DICT(w[1..n]));
4:     for i = 1 to n do
5:         item = IS_SENTENCE(w[1 .. i]) logical_and DICT(w[i + 1 .. n]);
6:         items.add(item);
7:     end for
8:     return apply logical_or on items;
9: end function
```

asociatívne pole IS_SENTENCE (Map<String,Boolean>) V nasledujúcom algoritme využívame techniku dynamického programovania. Štruktúra IS_SENTENCE plní rovnakú funkciu ako metóda IS_SENTENCE v predchádzajúcom príklade, rozdiel je však v tom, že každý prístup k nej bude mať konštantnú zložitosť

```
1: function IS_SENTENCE(w[1..n])
2:     for i = 1 .. n do
3:         subresults = [] of boolean;
4:         subresults.add(DICT(w[1..i]));
5:         for j = 1 .. i - 1 do
6:             subresult = IS_SENTENCE(w[1 .. j]) logical_and DICT(w[j + 1 .. i - i]);
7:             subresults.add(subresult);
8:         end for
9:         return IS_SENTENCE(w[1..i]) = apply logical_or on subresults;
10:    end for
11: end function
```

Zložitosť: $\mathcal{O}(n^2)$

Príklad 3

pole pravdepodobností: $C[p(1), .., p(n)]$
k - počet hláv
PVD - pravdepodobnostná funkcia

jednoduchý rekurzívny algoritmus:

```
1: function PVD(C[p1,..,pn],k)
2:     if k=0 then
3:         return PVD(C[p(1),..,p(n-1)],0)*(1-p(n));
4:     end if
5:     if k=n then
6:         return PVD(C[p(1),..,p(n-1)],k-1)*p(n);
7:     end if
8:     return PVD(C[p(1),..,p(n-1)],k-1)*p(n) + PVD(C[p(1),..,p(n-1)],k)*(1-p(n));
9: end function
```

technika dynamického programovania vytvorím asociatívne pole typu Map<Pair<Float[],Integer>,Integer> s názvom PVD

```
1: PVD([p(1)],0) = p1
2: PVD([p(1)],1) = (1-p1);
3: for i = 1 .. n do
4:     bottom = max(0,k-(n-i));
5:     up = min(i,k);
6:     for j = bottom .. up do
7:         if k=0 then
8:             PVD(C[p(1),..,p(i-1)],0)*(1-p(i));
9:         else
10:            if k=n then
11:                PVD(C[p(1),..,p(i-1)],j-1)*p(i);
12:            else
13:                PVD(C[p(1),..,p(i-1)],j-1)*p(i) + PVD(C[p(1),..,p(i-1)],j)*(1-p(i));
14:            end if
15:        end if
16:    end for
17: end for
```

Príklad 4

rekurzívny algoritmus

```
1: function bestPrice(lastIndexOfC,freePlaces[])
2:     if hasLastToFill(freePlaces) then
3:         indexOf1 = index of number 1 in array
4:         chosenItems is array of 0;
5:         fill chosenItems with values of 0;
6:         chosenItems[1] = indexOf1;
7:         return (C[1][indexOf1],chosenItems);
8:     end if
       ▷ values is array of pair(number , chosenItems); ▷ number: price ▷ chosenItems
   is array for example [0,0,1,2]
9:     for i = 1 .. freePlaces.size do
10:        if freePlaces[i] != 0 then
11:            freePlacesCopy = freePlaces;
12:            freePlacesCopy[i] = freePlacesCopy[i] - 1;
13:            pref = bestPrice(lastIndexOfC - 1,freePlacesCopy[]);
14:            values[i].number = pref.number + C[lastIndexOfC][i];
15:            values[i].chosenItems[lastIndexOfC] = i;
16:        end if
17:    end for
18:    highestValueIndex = index of highest value in values[1].number .. values[values.size].number;
19:    return values[highestValueIndex];
20: end function
21: function hasLastToFill(freePlaces[])
22:    return highest value in freePlaces is 1 and (freePlaces.size - 1) items in freePlaces
   is equal to 0
23: end function
```

Technika dynamického programovania

Vytvorím štruktúru bestPrice, ktorá bude typu asociatívne pole kľúče budú typu Pair<Integer,Integer []> (lastIndexOfC,freePlaces []) prvá hodnota oznaèuje index riadku v poli C druhá hodnota bude typu pole, jeho dĺžka bude zodpovedať počtu autosalónov hodnota na indexe i bude zodpovedať počtu áut, ktoré je možné ešte predať do autosalónu s číslom i
a = počet autosalónov
hodnoty budú typu Pair<Integer,Integer []> (price,distribution []) prvá hodnota značí vypočítanú najlepšiu celkovú cenu druhá hodnota bude typu pole, jeho dåžka bude zodpovedať počtu riadkov v poli C, hodnoty v čom budú slúžiť na určenie toho, do ktorého salónu bude predané auto s indexom na ktorom je prvok umiestnený 0 bude značiť, že auto zatiaľ nie je priradené do žiadneho salónu, 1 až a bude značiť konkrétny autosalón

```
1: function countBestPrice(C)
2:     for i = 1 .. C.rows do
```

```
3:          combinations = nájdeme všetky kombinácie práve a čísel z N0, ktoré sú menšie
    ako C.rows a, ktoré dávajú súčet i;                ▷ každý prvok bude zoznam o dlžke a
4:          permutations = []
5:          for combination in combinations do
6:              perm = všetky permutácie zoznamu combination
7:              permutations.addAll(perm);
8:          end for
9:          for permutation in permutations do
10:             items = [] of (price,distribution [])
11:             if hasLastToFill(freePlaces) then
12:                 indexOf1 = index of number 1 in array
13:                 chosenItems is array of 0;
14:                 fill chosenItems with values of 0;
15:                 chosenItems[1] = indexOf1;
16:                 return (C[1][indexOf1],chosenItems);
17:             end if
18:             for  j = 1 .. permutation.size do
19:                 modifiedPermutation = copy of permutation
20:                 if modifiedPermutation[j]>0 then
21:                     modifiedPermutation[j] = modifiedPermutation[j] - 1;
22:                     item it;
23:                     aPrice = bestPrice(i-1,modifiedPermutation);
24:                     it.price = aPrice.price + C[i][j];
25:                     it.distribution = aPrice.distribution;
26:                     it.distribution[i] = j;
27:                     items.add(it)
28:                 end if
29:             end for
30:             highestValueIndex = index najvacsej hodnoty spomedzi items.price;
31:             bestPrice[(i,permutation)] = items[highestValueIndex];
32:         end for
33:     end for
34:     return bestPrice with key(C.rows);
35: end function
```

Príklad 5

Hladový algoritmus nenájde správne 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 $\max(7,4,0) = 7$