Homework 2

Jakub Senko, Štefan Uherčík

14. 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

Algoritmus má nasledovný princíp: Pre každý vstupný reťazec urobí všetky možné rozdelenia tohto reťazca na 2 časti. Na prvú časť reťazca bude znova aplikovaná funkcia IS_SENTENCE, kým na druhú časť bude aplikovaná funkcia DICT. Na výsledky volaní týchto funkcií aplikujeme operátor logický súčin a uložíme do poľa items. V prípade, že aspoň jeden prvok z poľa items obsahuje hodnotu true, pôvodný reťazec je možné rozdeliť na slová zo slovníka. Pre zefektívnenie algoritmu využijeme pri funkcii logical_and tzv. Short-circuit evaluation.

```
function IS_SENTENCE(w[1..n])
    items = [] of boolean
    items.add(DICT(w[1..n]));
    for i = 1 to n do
        item = DICT(w[i + 1 .. n]) logical_and IS_SENTENCE(w[1 .. i]);
        items.add(item);
    end for
    return apply logical_or on items;
end function
```

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),...).
Algoritmus sa rekurzívne zavolá na reťazci o dĺžke n-a. Pri týchto volaniach však overuje prefixy s dĺžkami 1 .. 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.

```
function VERIFY_SENTENCE(w[1..n])
    for i = 1 .. n do
        subresults = [] of boolean;
        subresults.add(DICT(w[1..i]));
        for j = 1 .. i - 1 do
            subresult = IS_SENTENCE(w[1 .. j]) logical_and DICT(w[j + 1 .. i - i]);
            subresults.add(subresult);
        end for
        return IS_SENTENCE(w[1..i]) = apply logical_or on subresults;
    end for
    return IS_SENTENCE(w[1..n]);
```

end function

Korektnosť:
Konvergencia: Jediné miesta, u ktorých hrozí, že algoritmus nezastaví, sú volania cyklov. V cykle s iterujúcou premennou i sa zaručene pri každom priechode zvýši hodnota premennej i a iterovanie skončí, keď premenná i dosiahne hodnotu n. 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 pomocou matematickej indukcie: 1.) Algoritmus správne spočíta výsledok na reťazci o dĺžke jedného znaku a: cyklus s iterujúcou premennou i sa vykoná práve raz do poľa subresults vložíme práve jednu hodnotu - dict(a); ak aplikujeme logický súčet na pole s jednou položkou, výsledok bude práve táto položka, teda hodnota dict(a)
2.) Predpokladáme, že algoritmus je korektný na reťazci o dĺžke k: Pokúsime sa dokázať jeho korektnosť na reťazci o dĺžke 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. Volanie apply logical_or on subresults bude mať rovnakú zložitosť ako vnorený cyklus (vzhľadom k tomu, že počet jeho položiek zodpovedá počtu iterácií).
$\mathcal{O}(n^2)$

Príklad 3

Vstupom algoritmu bude: pole pravdepodobností: $C[p(1),..,p(n)]$
k - počet padnutých orlov
PVD - pravdepodobnostná funkcia

Problém je možné definovať nasledovne: vypočíť pravdepodobnosť, že v poli padne k orlov z n mincí
táto pravdepodobnosť je ekvivaletná 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 ktorm zároveň ošetrujeme krajné prípady n=k a n=0)

```
function PVD(C[p1,..,pn],k)
    if k=0 then
        return PVD(C[p(1),..,p(n-1)],0)*(1-p(n));
    end if
    if k=n then
        return PVD(C[p(1),..,p(n-1)],k-1)*p(n);
    end if
    return PVD(C[p(1),..,p(n-1)],k-1)*p(n) + PVD(C[p(1),..,p(n-1)],k)*(1-p(n));
end function
```

Ak rozpíšeme vetvenie algoritmu vykonávanie bude vyzerať približne nasledovne PVD(C[p(1),..,p(n)],k) = PVD(C[p(1),..,p(n-1)],k-1)*p(n) + PVD(C[p(1),..,p(n-1)],k)*(1-p(n))
PVD(C[p(1),..,p(n-1)],k) = PVD(C[p(1),..,p(n-1)],k-1)*p(n-1) + PVD(C[p(1),..,p(n-1)],k)*(1-p(n-1))
PVD(C[p(1),..,p(n-1)],k-1) = PVD(C[p(1),..,p(n-1)],k-2)*p(n-1) + PVD(C[p(1),..,p(n-1)],k-1)*(1-p(n-1))
... Z predchádzajúceho zápisu volaní funkcií je možné vidieť, že PVD(C[p(1),..,p(n-1)],k-1) sa zavolá 2 krát na jednej 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),... ,ich výsledky si budem ukladať do asociatívneho poľa a postupným volaním sa dopracujem k hodnote PVD(C[p(1),..,p(n)],k), ktorá je výsledkom celého problému. Pre tento účel vytvoríme

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

```
function countPVD(C[p1,..,pn],k)
    PVD([p(1)],0) = p1
    PVD([p(1)],1) = (1-p1);
    for i = 1 .. n do
        bottom = max(0,k-(n-i));
        up = min(i,k);
        for j = bottom .. up do
            if k=0 then
                PVD(C[p(1),..,p(i-1)],0)*(1-p(i));
            else
                if k=n then
                    PVD(C[p(1),..,p(i-1)],j-1)*p(i);
                else
                    PVD(C[p(1),..,p(i-1)],j-1)*p(i) + PVD(C[p(1),..,p(i-1)],j)*(1-p(i));
                end if
            end if
        end for
    end for
    return PVD(C[p(1),..,p(n-1)],k);
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 $\mathcal{O}(n^2)$

Príklad 4

rekurzívny algoritmus

```
function bestPrice(lastIndexOfC,freePlaces[])
    if hasLastToFill(freePlaces) then
        indexOf1 = index of number 1 in array
        chosenItems is array of 0;
        fill chosenItems with values of 0;
        chosenItems[1] = indexOf1;
        return (C[1][indexOf1],chosenItems);
    end if
    ▷ values is array of pair(number , chosenItems); ▷ number: price ▷ chosenItems
is array for example [0,0,1,2]
    for i = 1 .. freePlaces.size do
        if freePlaces[i] != 0 then
            freePlacesCopy = freePlaces;
            freePlacesCopy[i] = freePlacesCopy[i] - 1;
            pref = bestPrice(lastIndexOfC - 1,freePlacesCopy[]);
            values[i].number = pref.number + C[lastIndexOfC][i];
            values[i].chosenItems[lastIndexOfC] = i;
        end if
    end for
    highestValueIndex = index of highest value in values[1].number .. values[values.size].number;
    return values[highestValueIndex];
end function
function hasLastToFill(freePlaces[])
    return highest value in freePlaces is 1 and (freePlaces.size - 1) items in freePlaces
is equal to 0
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

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

```
        combinations = nájdeme všetky kombinácie práve a císel z N0, ktoré sú menšie ako C.rows a, ktoré dávajú súčet i;    ▷ každý prvok bude zoznam o dlžke a
        permutations = []
        for combination in combinations do
            perm = všetky permutácie zoznamu combination
            permutations.addAll(perm);
        end for
        for permutation in permutations do
            items = [] of (price,distribution [])
            if hasLastToFill(freePlaces) then
                indexOf1 = index of number 1 in array
                chosenItems is array of 0;
                fill chosenItems with values of 0;
                chosenItems[1] = indexOf1;
                return (C[1][indexOf1],chosenItems);
            end if
            for  j = 1 .. permutation.size do
                modifiedPermutation = copy of permutation
                if modifiedPermutation[j]>0 then
                    modifiedPermutation[j] = modifiedPermutation[j] - 1;
                    item it;
                    aPrice = bestPrice(i-1,modifiedPermutation);
                    it.price = aPrice.price + C[i][j];
                    it.distribution = aPrice.distribution;
                    it.distribution[i] = j;
                    items.add(it)
                end if
            end for
            highestValueIndex = index najvacsej hodnoty spomedzi items.price;
            bestPrice[(i,permutation)] = items[highestValueIndex];
        end for
    end for
    return bestPrice with key(C.rows);
end function
```

Konvergencia

Parciálna korektnosť

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) $= 7$