IV003 - SADA 2

Řešitelé: Tomáš Skopal (374549)
Vojtěch Bělovský (374032)

Příklad 1

Základem našeho řešení je algoritmus MergeSort. Označme vstupní seznam budov (trojic) B_L, jeho velikost je *n*. Nejdříve potřebujeme seznam uspořádat. Konkrétně seznam uspořádáme podle souřadnice levého okraje budovy (*L*). Pokud budou souřadnice L_i a L_j budov B_i a B_j shodné pak porovnáme ještě podle výšky budovy H_i a H_j, tak, že nižší budovu dáme před vyšší.

Takto uspořádaný seznam nazvěme B_S.

MergeSort – pracuje tak, že nejdříve pole rozdělí na jednotlivé prvky a ty poté slučuje a při slučování je řadí (rozděl a panuj).

Jakmile je seznam uspořádaný využijeme druhou část MergeSortu a to právě slučování. Ze seznamu B_s budeme brát budovy po dvojicích a vytvářet z každé dvojice dílčí siluetu. Při spojování budov B_i a B_{i+1} mohou nastat 4 situace:

a) 1 budova „obsahuje“ druhou:
   Například $B_i = (1,1,2)$ a $B_{i+1} = (1,3,4)$. V takovém případě bude silueta trojicí takovou, že $(L(B_i), MAX(H(B_i), H(B_{i+1})), MAX(R(B_i), R(B_{i+1})))$
b) Budovy se částečně překrývají:
   V takovém případě bude silueta pěticí takovou, že
   $(L(B_i),* H(B_i), PRUSECIK, *H(B_{i+1}), MAX(R(B_i), R(B_{i+1})))$
   PRUSECIK – označuje souřadnici, kde mají obě budovy stejnou x-ovou souřadnici.
   Například pro $B_i = (1,1,3)$ a $B_{i+1} = (2,5,4)$ je to 2.
   *-V jediném případě budou souřadnice prohozeny a to tehdy pokud je $L(B_i) = L(B_{i+1})$ a zároveň $H(B_i) < H(B_{i+1})$.
c) Budovy nemají žádnou společnou část, $R(B_i) < L(B_{i+1})$:
   V takovém případě je silueta šesticí takovou, že $(L(B_i), H(B_i), R(B_i), L(B_{i+1}), H(B_{i+1}), R(B_{i+1}))$
d) Budovy se překrývají úplně $R(B_i) > R(B_{i+1})$ a zároveň $L(B_i) < R(B_{i+1})$:
   V takovém případě je silueta sedmicí takovou, že $(L(B_i), H(B_i), L(B_{i+1}), H(B_{i+1}), R(B_{i+1}), H(B_i), R(B_i))$

Dále algoritmus slučuje dílčí siluety. Je vždy potřeba prozkoumat hraniční oblast trojice hodnot, a rozhodnout, jak bude výsledná silueta vypadat. Může nastat opět několik situací:

a) Siluety nemají žádnou společnou část (viz c) výše), pak stačí jen souřadnice zřetězit.
b) Siluety se dotýkají jen hranou, pak se souřadnice zřetězí, s tím, že hraniční souřadnice se napíše jen jednou.

c) Siluety se překrývají, pak je potřeba rozhodnout v místě průsečíku (viz b) výše), které souřadnice budov se uplatní.

Tím, že je pole seřazené stačí prozkoumat hraniční oblast pouze v hraniční trojici hodnot, zbytek posloupnosti se zkopíruje. Takto se dílčí posloupnosti slučují, až je sestavena celá silueta.

Korektnost:
V prvním kroku, kdy se slučují jednotlivé trojice (budovy) jsou v algoritmu popsány všechny stavy. Výsledkem je vždy správná silueta.
V dalších krocích musíme řešit pouze krajní trojici hodnot, protože díky seřazení vstupní posloupnosti budov máme zaručeno, že dílčí siluety se nebudou překrývat o více než jednu budovu. To ukážeme na protipříkladu:
Mějme dvě siluety, které chceme sloučit A = (1, 3, 2, 1, 4, 4, 5), B = (3, 2, 6, 1, 7). Ty se překrývají o více než jednu trojici souřadnic a algoritmus by takovou siluetu vyhodnotil špatně. Ale díky tomu, že jsme na začátku pole seřadili, tak taková situace nemůže nastat. Protože budova (3, 2, x) ze siluety B by se při řazení dostala před budovu (4, 4, 6) ze siluety A. To je spor, proto stačí porovnat hraniční trojici souřadnic (siluety se mohou překrývat nanejvýš právě o jednu budovu). Postupným slučováním dílčích siluet, pak dostaneme korektní výslednou siluetu.

Složitost:
Nejdříve budovy utřídíme, což jsme schopni zvládnout v časové složitosti O(n log n).
Slučování siluet je pak částí algoritmu MergeSort, který má časovou složitost O(n log n).
Porovnání krajních hodnot při slučování je v konstantním čase, protože se jedná o nejvýše 3 hodnoty.

IV003 - SADA 2

Řešitelé: Tomáš Skopal (374549)
Vojtěch Bělovský (374032)

Příklad 2

```
isSeparable(String str)
{
    int size = str.length();
    if (size == 0)   return true;

    boolean[] breaks = new boolean[size+1];
    breaks <- false; //initialize array to false values

    for (int i=1; i<=size; i++)
    {
        if (breaks[i] == false && dict(str.substring(0, i)))
            breaks[i] = true;
        if (breaks[i] == true)
        {
            if (i == size)
                return true;
            for (int j = i+1; j <= size; j++)
            {
                if (breaks[j] == false && dict(str.substring(i, j)))
                    breaks[j] = true;
                if (j == size && breaks[j] == true)
                    return true;
            }
        }
    }
}
```

Hodnota breaks[i], kde 0 < i <= size, udává, jestli podřetězec délky 0..i-1, vytvořený ze vstupního řetězce (str), je možno rozdělit na slova podle slovníku. Pokud obsahuje true, pak ano.

Pokud řetězec délky 0..i je platným slovem, pak prohledám všechny řetězce, začínající na indexu i+1. Princip je stejný, opět si ukládám do pole breaks hodnoty, jestli řetězec i+1..j je platným slovem. O prohledání sufixů a uložení hodnot se stará vnitřní for cyklus.

Složitost:
Jak je vidět z kódu, vnější cyklus běží n-krát, kde n je délka vstupního řetězce a vnitřní cyklus běží vždy od aktuálního indexu ‚i' až do n (maximálně). Tedy algoritmus je ve složitostní třídě $O(n^2)$.

Korektnost:

c[i] označuje délku posloupnosti písmen od indexu i až do konce vstupního řetězce (str).

$$\text{c[i]} \begin{cases} 0 & i = str.lenght \\ > 0 & i < str.lenght \end{cases}$$

Pokud

IV003 - SADA 2

Řešitelé: Tomáš Skopal (374549)
Vojtěch Bělovský (374032)

Příklad 3

p_i…… pravděpodobnost že na i-té minci padne orel.
A[1..n]…… pole čísel p_i od p_1 do p_n pro n zadaných mincí.

Následující rekurentní vztah popisuje pravděpodobnost toho, že na n mincích padne k orlů. Kde i je index od 1 do n a hodnoty p_i se pro výpočet vybírají podle indexu i z pole A.

$P(n,k,i) = p_i * P(n-1,k-1,i+1) + (1-p_i) * P(n,k,i+1)$

Na základě výše popsané rovnice nám vznikne následující algoritmus:
vstup:
n……. počet minci
k……. počet kolikrát padne orel
A[1..n]…… obsahující hodnoty p_i

Algoritmus:

```
//pole B[1] nám slouží jako pomocné pole k "nulování" některých pravděpodobností
B[1..n+2][1..n+1] = 0; //všude přiřadíme nuly
B[2][n+1] = 1;
for(i = 1; i <= n+2; i++){
	for(j = n; j >= 1; j--){
		B[i + 1][j] = A[j] * B[i][j + 1] + (1 - A[j]) * B[i + 1][j + 1];
	}
}
return B[k + 2][1];
```

Pole B obsahuje pole, jež obsahují pole, které mají na první pozici pravděpodobnosti toho, že padne k-krát orel. Tyto pravděpodobnosti se počítají z již dříve napočítaných pravděpodobností.

Složitost:
První forcyklus proběhne n+2 krát. V něm probíhá druhý for cyklus který pokaždé proběhne n krát. To je dohromady (n+2)*n proběhnutí vnitřního forcyklu. Toto číslo se ještě musí vynásobit dvěma, jelikož v jednom průběhu vnitřního forcyklu násobíme dvakrát. Což nám ve výsledku dává (n+2)*n*2 násobení. A to můžeme upravit na n^2*4n násobení. Což je ale asymptoticky stále ve složitostní třídě $O(n^2)$.

Korektnost:

Korektnost plyne z rekurentní rovnice a algoritmus počítá přesně to co zadává rekurentní rovnice. Nejde o nic jiného než o to že si každou situaci rozdělíme na dva případy (padne nebo nepadne orel) a pro výpočet již používáme předem vypočtené hodnoty (memorizace), čímž ušetříme složitost (už nenásobíme zbytečně některé prvky pořád dokola, jelikož pravděpodobnost předešlé části si pamatujeme v našem poli, které reprezentuje vlastně tabulku i = řádek, j = sloupec).

IV003 - SADA 2

Řešitelé: Tomáš Skopal (374549)
Vojtěch Bělovský (374032)

Příklad 4

IV003 - SADA 2

Řešitelé: Tomáš Skopal (374549)
Vojtěch Bělovský (374032)

Příklad 5

1. Slovní problém

Pro první slovní problém najde hladový algoritmus vždy optimální řešení. Je to tím že celková penalizace je dána součtem penalizací jednotlivých řádků, kdy penalizace každého řádku je dána pouze rozdílem L - K. Když bychom umístili nějaké i-té slovo S_i místo na konec řádku, kde by se ještě vlezlo, na začátek nového, tak nám tím můžou vzniknout dvě situace:

1) Slovo které přesuneme na nový řádek způsobí že se některá slova, která by se tam jinak vlezla, na nový řádek nevlezou.

2) Slovo které přesuneme na nový řádek způsobí to že se slova na daném řádku posunou, ale celková délka slov nepřesáhne optimální délku řádku.

V prvním případě platí, že délka slov (případně součet jejich délek, pokud jich je více), které se už na nový řádek kvůli přesunuti slova S_i nevlezou musí být alespoň tak velká jako je délka slova S_i. Z toho plyne že penalizace buď zůstane stejná (pokud je slovo S_i stejně dlouhé jako součet délek slov, které musíme umístit na následující řádek), nebo se může zhoršit (pokud je součet délek slov, které musíme umístit na následující řádek, větší než délka slova S_i).

V druhém případě by se nám podařilo snížit penalizaci následujícího řádku maximálně o délku daného slova. Ovšem o tuto délku jsme snížili délku předchozího řádku, takže se jeho penalizace zvýšila o hodnotu d_i. Tím jsme výslednou penalizaci nezměnili.

Pokud se tento posun vypropaguje až na poslední řádek, může opět nastat jedna z výše popisovaných situací. Pokud nastane první situace, pak penalizace může být nejlépe nulová, což nám ale nepřilepší. A pokud nastane druhý případ tak si stále celkově nepřilepšíme, jelikož to co se na poslední řádek vypropaguje, tak muselo někde ubýt (o to se tedy musela minimálně zvednout penalizace některých předchozích řádků), takže se celková penalizace nemohla snížit.

2. a 3. Slovní problém

Pro druhý a třetí slovní problém hladový algoritmus nemusí najít optimální řešení. Jako protipříklad můžeme uvést například případ kdy L=5 (délka řádku je 5 znaků) a seznam slov o délce [3,2,1]. Hladový algoritmus nám v takovém případě umístí první a druhé slovo na první řádek a třetí slovo (délky jedna) na druhý řádek. Optimální řešení pro druhý a třetí slovní problém je ale rozdělení slov tak, že by první slovo bylo na prvním řádku a druhé a třetí slovo by byly na druhém řádku. Můžeme si snadno spočítat, že hladový algoritmus by sestavil slova

tak, že by penalizace pro druhý slovní problém byla rovna 16 (4^2), kdežto optimální rozdělení má penalizaci pouze 8 ($2^2 + 2^2$). Pro třetí slovní problém vidíme penalizaci rovnou, jelikož v případě, který nám sestaví hladový algoritmus je penalizace 4 a optimální řešení má penalizaci pouze 2.