**Algoritmy a dátové štruktúry II (IV003)** **SADA PROBLÉMOV 2**

**Autor**: Michal Lukáč, 430614

**Spoluriešiteľ**: -

**Úloha 1**

V architektonickóm štúdiu sa rieší problem siluet. Navrhnite algoritmus, ktorý vypočíta siluetu zadaných budov s časovou složitosťou O(nlogn), kde n je počet budov. Použite techniku rozdel a panuj. Budova Bi je reprezentovaná trojicou(Li, Hi, Ri). Kde Li a Ri sú súradnice okrajov a Hi je výška budovy. Využite techniku divide and conquer.

Základom spôsobu divide and conquer v tomto prípade je, že pokiaľ máme n budov potom týchto n budov rekurzívne rozdelujeme na dve časti a na každej vytvárame opäť siluetu. Najprv si reprezentujeme dátové štruktúry, ktoré budeme využívať:

Štruktúru B s premennými L, H, R reprezentujúcu súradnice a výšku. Štruktúru Silueta obsahujúcu pole štruktúr Line pozostávajúce z L, H teda z ľavého rohu a výšky pričom reprezentuje pás. Pole týchto Line nám udáva siluetu. Štruktúra siluety obsahuje operáciu RemoveFirst, ktorá vráti a odstráni Line na prvom mieste. Interne si v štruktúre udržiavame počet prvkov Line.

Základom sú dve funkcie a to pre rozdelenie na podproblémy, respektíve menšie časti pomocou DivideSiluety a následné spájanie pomocou funkcie MergeSiluety. Algoritmus je veľmi podobný MergeSortu.

1. *Silueta* ***DivideSiluety****(B bpole[], int l, int h)*
2. *if l == h: // spojenie*
3. *Silueta si(2) // silueta o velkosti 2*
4. *si.Append(Line l(B[l].L, B[l].H))*
5. *si.Append(Line l(B[l].R, 0))*
6. *return si // vrátenie novo vytvorenej siluety*
7. *else // rozdelenie na ešte menšie podproblémy*
8. *Silueta s1 = DivideSiluety (B, l, (l+h)/2)*
9. *Silueta s2 = DivideSiluety (B, (l+h)/2 + 1, h)*
10. *return MergeSilueta(s1, s2)*

V DivideSiluety postupne rozďelujeme pole budov a tvoríme z nich siluety až kým prídeme k jednej budove a silueta jednej budovy je samotná budova(2-6). Na riadkoch 8-9 voláme rekurzívne rozdelenie na pole budov s následným spojením siluet(10).

1. *Silueta* ***MergeSiluety****(Silueta S1, Silueta S2)*
2. *Silueta s(S1.Count + S2.Count)*
3. *int h1= 0; int h2 = 0;*
4. *while S1.Count > 0 && S2.Count > 0*
5. *if S1.Head.L < S2.Head.L*
6. *int CX = S1.Head.L*
7. *h1 = S1.Head.H*
8. *maxH = max(h1,h2)*
9. *s.Append(Line l(CX, maxH))*
10. *S1.RemoveFirst()*
11. *else*
12. *… // obdobne pre S2.Head*
13. *...*
14. *while S1.Count > 0*
15. *Line l = S1.RemoveFirst()*
16. *s.Append(l)*
17. *while S2.Count > 0*
18. *Line l = S2.RemoveFirst()*
19. *s.Append(l)*
20. *return s*

Spojenie dvoch siluet prebieha následne. Inicializujeme si štruktúru so siluetou, ktorá bude mať veľkosť oboch spájaných siluét. Pokiaľ v obidvoch siluet máme nejaké Line. Vyberieme z oboch štruktúr prvú line a porovnáme, ktorá začína skôr. Následne vyberieme maximum z veľkosti výšky budov a novo vytvorenú Line pridáme do novej siluety a odstránime linu zo spracovávanej siluety. Pokiaľ už neplatí podmienka tohto cyklu, doplníme novo vytváranú Siluetu o zvyšné Line. V premenných h1 a h2 si udržiavame, že výšky z jednotlivých siluet.

Algoritmus je korektný, prejde všetky budovy a siluety, ktoré následne spája. Pri spájaní as vždy uvažuje ktorá budová ma bližšiu L súradnicu a jej veľkosť. Pri každom spájaní sa vytvorí teda úplne nová silueta, ktorá je už zotriedená. Algoritmus má časovú zložitost O(n.logn). Spájanie dvoch siluet o velkosti m zabera O(2\*m) pretože prechádzame cykly *while S1.Count > 0 && S2.Count > 0 a while S1.Count > 0 a while S2.Count > 0* v ktorých postupne odstraňujeme prvky buď Siluety S1 alebo S2 a pridávame ich do novo vytvorenej siluety. My vieme, že siluety majú veľkosť n/2. preto trvá spojenie O(n). Rekurentná rovnica má vzťah T(n) = 2T(n/2)+O(n). Teda dvakrát voláme DivideSiluety na n/2 prvkov a jedenkrát spájam n/2 + n/2 prvkov teda n. Na základe master theorem určíme z rekurentnej rovnice, že algoritmus je O(nlogn).

**Algoritmy a dátové štruktúry II (IV003)** **SADA PROBLÉMOV 2**

**Autor**: Michal Lukáč, 430614

**Spoluriešiteľ**: -

**Úloha 2**

Je daný reťazec n znakov T[1..n], z ktorého boli vypustené interpunkčné znamienka. Úlohou je zrekonštruovať text do čitatelnej podoby za pomoci slovníka. Slovník je k dispozícií v tvaru booleovskej funkcie dict() ktorá pre libovolný reťazec w vráti true ak je platné slovo alebo false.

Algoritmus pre rekonštrukciu textu je následný:

1. ***Reconstruct****(T):*
2. *for d = 0 to n -1*
3. *for i = 0 to n – d*
4. *j = i + d*
5. *if dict(T[i…j]) then*
6. *x(i,j) = true*
7. *else*
8. *for k = i to j – 1*
9. *if x(i,k) == True && x(k+1,j) == true*
10. *x(i,j) = True*
11. *y(i,j) = k // medzera na mieste k*

Algoritmus sa skladá z troch zanorených for cyklov. Prvým for cyklom označujeme rozsah prehľadávania. Pokiaľ je d = 0, skúmame vždy jeden znak. Pokiaľ je d = 1 skúmame slovo o dvoch znakoch. Druhým for cyklom prehľadávame text. Postupne prechádzame od i=0 a j = i+d a prechádzame vždy rozsah T[i..j]. Pokiaľ na danom reťazci T[i…j] nájdeme slovo poznačíme si tento nález do 2d matice x ako x(i,j) = True. Posledným for cyklom si označíme súvislosť slov po sebe a kde sa takéto slova delia. Môžeme si to ukázať na príklade s reťazcom „ahojpeto“. Pokiaľ je d = 7, i = 0, j = 7. Pričom pri prehľadávaní k narazíme na to kedy x[i=0…k=3] je true a x[k+1=4…j=7] je true potom medzeru medzi slovami si zapamätáme pomocou matice y(i,j) = k.

Algoritmus prehľada každú veľkosť reťazca a vždy označí slovo, ktoré sa nachádza v slovníku. Vždy rozdelí slová, ktoré sú platen a sú vedľa seba to je zaistené v poslednom cykle. Časová zložitosť tohoto algoritmu je O(n^3), keďže algoritmus pozostáva z troch zanorení for cyklov a je teda polynomiálny. Pričom v každom for cyklu sa prejde max veľkosť reťazca. Preto O(n^3).

**Algoritmy a dátové štruktúry II (IV003)** **SADA PROBLÉMOV 2**

**Autor**: Michal Lukáč, 430614

**Spoluriešiteľ**: -

**Úloha 3**

Vstupom sú čísla n, k a n čísel p1, p2, ..., pn. Úlohou je vypočítať aká je pravdepodobnosť toho, že pri hádzaní n mincami padne k krát orol. Predpokladajme, že jednotlivé hody sú nezávislé a pravdepodobnosť, že pri hodu mincou i padne orol je pi. Cena násobenía a sčítania dvoch čísel je O(1). Využite princíp dynamického programovania.

Navrhnutý program:

double **ProbabilityCoin**(int n, int k, double []p)

1. 2dMat<double> A = new 2dMat<double>(); A[0][0] = 1.0 // 2d matica pre zapamätovanie
2. for (int i = 0; i < k; i++)
3. if (i != 0)
4. A[i][i] = A[i-1][i-1]\*p[i] // pravdepodobnost, že pri i hodoch i krát padne orol
5. for (int j = i + 1, j < n-k+i; j++) // pravdepodobnosť, že pri j hodoch i krát padne or.
6. if (i != 0):
7. A[i][j] = p[j]\*A[i-1][j-1] + (1-p[j]) \* A[i][j-1]
8. else
9. A[i][j] = (1.0 – p[j]) \* A[i][j-1] // pravdepodobnosť, že 0 krát padne orol
10. return A[k][n] // pravdepodobnosť, že pri n hodoch padne k krát orol

Na začiatku si inicializujeme maticu typu double. Prvý prvok matice nastavíme na 1.0 keďže pravdepodobnosť pri nehádzaní je 1.0. Potom prechádzame for cyklom od 0 po k. A zisťujeme pravdepodobosť Časová složitosť je O(n^2), keďže prechádzame dva for cykli prvý od nula po k a druhý for cyklus od 1 po (n-k+i) teda až po n. Keďže, algoritmus môže dostať na vstupe k=n tak preto O(n^2).

**Algoritmy a dátové štruktúry II (IV003)** **SADA PROBLÉMOV 2**

**Autor**: Michal Lukáč, 430614

**Spoluriešiteľ**: -

**Úloha 4**

**Algoritmy a dátové štruktúry II (IV003)** **SADA PROBLÉMOV 2**

**Autor**: Michal Lukáč, 430614

**Spoluriešiteľ**: -

**Úloha 5**

Vstupom je usporiadaný zoznam obsahujúci n slov, pričom i-té slovo má di znakov. Slová sa majú rozdeliť do riadkov pri zachovaní daného poradia slov. Pre jednoduchosť nezapočítáváme medzi znaky mezery mezi slovy. Optimálna dĺžka riadku je L znakov. Riadok nemôže obsahovať viac znakov.

**Hladový algoritmus**

For i = 1 to n

Pokiaľ není prekčený povolený počet znakov

Umiestni slovo na aktuálny riadok

Inak umiestni i-té slovo na nový riadok

**Prvý slovný problém**

Celková penalizácia je rovná súčtu penalizácií jednotlivých riadkov(L-K). Hladový algoritmus nájde optimále riešenie. Posunutím slova na ďalší rádok sa rovnomerne zmení cena v stávajúcom a ďalšom riadku. Preto je suma vždy rovnaká.

**Druhý slovný problém**

Penalizácia riadku je daná ako (L-K)^2. Celková penalizácia rozdelenia slov do riadkov je rovná súčtu penalizácií jednotlivých riadkov. Pre tento problém algoritmus **nemusí nájsť optimálne riešenie.** Uvažujme daný proti príklad na vstupu sú slová v poradí 12345, 12, 1, 123, 12345678. Optimálna dĺžka riadku L = 10.

Hladový algoritmus zarovná sekvenciu následne:

12345 12 1 | penal = 4

123 | penal = 49

12345678 | penal = 4

Suma = 57

Existuje však napríklad nasledujúce lepšie riešenie, ktoré by vzniklo posunutím slova 1 na ďalší riadok. Potom by problém bol následný:

12345 12 | penal = 9

1 123 | penal = 36

12345678 | penal = 4

Suma = 49

Suma nového riešenia je menšia ako predchádzajúca preto hladový algoritmus pre tento problém není optimálny.

**Tretí slovný problém**

Sa líši od prvého v definícií celkovej penalizácií. Celková penalizácia je rovná maximu penalizácií jednotlivých riadkov. Úlohou je nájsť rozdelenie s najnižšiou možnou penalizáciou.

Pre tento problém opäť **nie je hladový algoritmus optimálny**. Uvažujme daný proti príklad so sekvenciou 12345, 12, 1, 123, 12345678.

Hladový algoritmus zarovná sekvenciu následne:

12345 12 1 | penal = 2

123 | penal = 7

12345678 | penal = 2

Výsledok = 7

Existuje však napríklad nasledujúce lepšie riešenie, ktoré by vzniklo posunutím slova 1 na ďalší riadok. Potom by problém bol následný:

12345 12 | penal = 3

1 123 | penal = 6

12345678 | penal = 2

Výsledok = 6

Vidíme že 6 < 7 a preto nie je hladový algoritmus optimálny.