


23/12/2016



Probleme asociate procesarii sirurilor de caractere



Maftai Stefan - Radu, 324CD

Cuprins

1. Introducere.....	2
2. Prezentarea problemelor si algoritmilor	3
2.1. Cea mai lunga subsecventa palindromica (LPS).....	3
2.1.1 Solutia Brute Force	3
2.1.2 Solutia Dynamic Programming	4
2.1.3 Solutia Clever Approach.....	5
2.1.4 Solutia Manacher	5
2.2 Cel mai lung subsir comun (LCS).....	7
2.2.1 Solutia Recursive.....	8
2.2.2 Solutia Dynamic Programming.....	9
3. Evaluare si discutii	10
3.1 LPS.....	10
3.1 LCS	11
4. Concluzii.....	12
5. Bibliografie	12

1. Introducere

Una din ariile importante ale proiectarii algoritmilor este studiul algoritmilor pentru siruri de caractere. Cea mai importanta problema in acest domeniu este cautarea eficienta a unei subsecvente sau mai general a unui pattern intr-un text de dimensiune mare (exemple cunoscute care fac acest lucru ar fi editoarele de text sau functii ca "grep" atunci cand are loc o cautare).

In multe situatii nu dorim sa cautam o bucata exacta din text, ci o bucata "similara". Acest lucru se intampla in studiul geneticii. Codurile genetice sunt stocate ca niste molecule lungi de ADN. Lanturile de ADN pot fi rupte in siruri lungi, fiecare fiind una din cele 4 tipuri de baza: C, G, T, A.

Potrivirile exacte au loc rar in biologie, din cauza micilor schimbari in replicatia ADN. Cautarea subsecventelor exacte va gasi mereu potriviri exacte. Din acest motiv, este util sa fie calculate similaritati intre siruri de caractere care nu se potrivesc exact. Aceasta metoda a similaritatii sirurilor de caractere ar trebui sa nu tina cont de inserari sau stergeri aleatoare de caractere din sirul original.

Astfel exista un set de masuri pentru similaritatile din sirurile de caractere. Dintre acestea noi vom studia "Cea mai lunga subsecventa palindromica" si "Cel mai lung subsir comun".

2. Prezentarea problemelor si algoritmilor

2.1. Cea mai lunga subsecventa palindromica (LPS)

Pentru inceput sa definim notiunea de "palindrom". Un palindrom este un sir de caractere care se citește identic din ambele directii (de la stanga la dreapta si de la dreapta la stanga). De exemplu: "aba" este un palindrom, dar "abc" nu este.

Enuntul problemei: Fie sirul de caractere S . Inversul lui S este S' . Gasiti cea mai lunga subsecventa comuna intre S si S' , care trebuie de asemenea sa fie cea mai lunga subsecventa palindromica.

Recunoasterea palindromica este importanta in biologia computationala. Structurile palindromice pot fi gasite frecvent in proteine si identificarea lor ofera cercetatorilor idei despre acizii nucleici.

2.1.1 Solutia Brute Force

Solutia Brute Force imediata este de a alege toate pozitiile de inceput si de sfarsit pentru o subsecventa, verificandu-se daca subsecventa este palindrom. Sunt in total C_n^2 astfel de subsecvente (excluzand posibilitatea solutiei triviale unde un singur caracter este palindrom).

Pentru ca fiecare subsecventa are o complexitate de $O(n)$ (timp), complexitatea la rulare va fi $O(n^3)$ (timp).

2.1.2 Solutia Dynamic Programming

Pentru a imbunatati o solutie Brute Force printr-o abordare a programarii dinamice, trebuie in primul rand sa ne gandim cum sa evitam recalculările care nu sunt necesare in validarea palindromurilor. Sa consideram cazul "ababa". Daca deja stiam ca "bab" este palindrom, este evident ca "ababa" trebuie sa fie palindrom, din cauza ca am adaugat la stanga si la dreapta aceeasi litera.

Scris mai formal avem:

Definim $P[i, j] \leftarrow true$, daca subsecventa $S_i \dots S_j$ este palindrom, altfel *false*

Astfel $P[i, j] \leftarrow (P[i + 1, j - 1] \text{ si } S_i = S_j)$

Cazurile de baza sunt:
$$\begin{cases} P[i, j] \leftarrow true \\ P[i, i + 1] \leftarrow (S_i = S_{i+1}) \end{cases}$$

Astfel, folosind programarea dinamica, avem solutia in care initializam palindromurile cu una si doua litere, obtinand toate palindromurile de trei litere si asa mai departe, pana la cel mai lung palindrom ce se poate obtine.

Acest algoritm are o complexitate la rulare de $O(n^2)$, folosind spatiu $O(n^2)$ pentru a stoca tabelul $(P[i, j])$.

Spre deosebire de restul algoritmilor, in cazul in care sunt mai multe subsecvente palindromice de aceeasi lungime maxima, acest algoritm va afisa ultima subsecventa palindromica de lungime maxima. Restul algoritmilor afiseaza prima subsecventa palindromica de lungime maxima.

2.1.3 Solutia Clever Approach

Folosind principiul de mai sus putem aborda problema astfel incat sa avem spatiu $O(1)$, timpul ramanand acelasi $O(n^2)$.

Observam ca un palindrom se oglindeste in jurul centrului sau. Tinand cont de asta, un palindrom poate fi extins din centrul sau, un sir de caractere avand doar $2 \cdot N - 1$ astfel de centre. O intrebare care se pune imediat este "De ce $2 \cdot N - 1$, dar nu N centre? Motivul este ca centrul palindromului poate fi intre doua litere. Astfel de palindromuri au un numar par de litere (de exemplu "abba", centrul sau fiind intre doua 'b'-uri).

Pentru ca extinderea unui palindrom in jurul centrului sau poate avea un timp $O(n)$, complexitatea algoritmului va fi $O(n^2)$.

2.1.4 Solutia Manacher

Manacher (1975) a gasit un algoritm cu timp linear pentru gasirea celei mai lungi subsecvente palindromice dintr-un sir de caractere.

In prima faza transformam sirul de caractere de intrare S in alt sir de caractere T prin insertia unui caracter special "|" intre litere. Motivul pentru acest lucru va fi clarificat imediat. (ex: $S = \text{"abaaba"}$, $T = \text{"|a|b|a|a|b|a|"}).$

Pentru a gasi cea mai lunga subsecventa palindromica, va trebui sa extindem fiecare T_i astfel incat $T_{i-d} \dots T_{i+d}$ sa formeze un palindrom. Acest d reprezinta lungimea palindromului centrat in T_i .

Retinem rezultatul intr-un vector P , unde $P[i]$ este egal cu lungimea palindromului centrat in T_i . Cea mai lunga subsecventa palindromica va fi elementul maxim din P .

ex:

T	=		a		b		a		a		b		a	
P	=	0	1	0	3	0	1	6	1	0	3	0	1	0

Uitandu-ne la P observam imediat ca palindromul este "abaaba", indicat de $P_6 = 6$. Observam ca numerele din P sunt simetrice.

Algoritmul scris formal arata in felul urmator:

Daca $P[i'] \leq R - i$,

Atunci $P[i] \leftarrow P[i']$

Altfel $P[i] \geq P[i']$. (Pe care noi trebuie sa il extindem dincolo de marginea din dreapta (R) pentru a gasi $P[i]$).

Partea finala este sa determinam cand ar trebui sa mutam pozitia centrului (C) impreuna cu pozitia marginii din dreapta (R) la dreapta. Vom avea conditia:

"Daca palindromul centrat in i se extinde dupa R, atunci actualizam C la i , (centrul acestui nou palindrom), si extindem R la noua margine din dreapta a noului palindrom."

In orice pas sunt doua posibilitati. Daca $P[i] \leq R - i$, setam $P[i]$ la $P[i']$ care are exact un pas. Altfel trebuie sa schimbam centrul palindromului la i prin extinderea lui la marginea din dreapta, R. Extinderea lui R dureaza cel mult N pasi, iar pozitionarea si testarea fiecarui centru dureaza in total N pasi, de asemenea.

Astfel, acest algoritm se terminat garantat in cel mult $2 \cdot N$ pasi, deci solutia este in timp linear $O(n)$.

2.2 Cel mai lung subsir comun (LCS)

Un sir de caractere s este un subsir al sirului de caractere s' daca s poate fi obtinut din s' prin eliminarea unor caractere in s' .

Enuntul problemei: Dandu-se doua siruri de caractere, sa se gaseasca subsirul lor comun care are lungime maxima.

Aceasta problema a fost studiata intens in ultimii 30 de ani, ea avand multe aplicatii. Cel mai lung subsir comun poate fi o masura a similaritatii a

doua siruri de caractere si astfel problema e utila in biologie moleculara, recunoasterea pattern-urilor si compresia textelor.

2.2.1 Solutia Recursive

Folosind recursivitatea, comparam cele 2 siruri de caractere in ordine inversa un caracter pe rand. Avem 2 cazuri:

1. Ambele caractere sunt identice.

In acest caz se adauga la sfarsit caracterul si se elimina ultimul caracter din ambele siruri de caractere si reluam recursivitatea cu sirurile de caractere modificate.

2. Caractere diferite.

In acest caz se elimina ultimul caracter din primul sir de caractere si se face un apel recursiv, si se elimina ultimul caracter din al doilea sir de caractere si se face un apel recursiv, comparandu-se lungimea celor doua siruri rezultate in urma apelului; apoi se apeleaza apelul care a oferit un sir mai mare.

Pentru un sir de caractere de lungime n se pot face 2^n subsiruri, deci daca rezolvam problema prin recursivitate complexitatea va fi $O(2^n)$ (timp), tinand cont ca noi rezolvam subprobleme repetat.

2.2.2 Solutia Dynamic Programming

Vom folosi tehnica programarii dinamice, retinand solutia subproblemelor intr-un vector de solutii si o sa-l folosim oricand avem nevoie de el. Aceasta tehnica se numeste tehnica "Memoization".

		A	B	C	D	A
		0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
B	0	1	2	2	2	2
D	0	1	2	2	3	3
E	0	1	2	2	3	3
A	0	1	2	2	3	4

LCS - "ACDA"

Imagine 1. Ex. cu rezultatul "ACDA" pentru sirurile "ABCD A" si "ACBDEA"

Pentru obtinerea sirului vom porni din coltul din dreapta jos si vom urmari calea, marcand celulele a caror valoare vin din mersul pe diagonala (adica ultimul caracter al ambelor siruri de caractere au fost identice, reducand lungimea cu 1, deci am mers pe diagonala). Celulele marcate vor oferi raspunsul.

Complexitatea acestei solutii este $O(n^2)$, intrecand cu mult solutia precedenta din punct de vedere al timpului de executie.

3. Evaluare si discutii

Pentru testarea algoritmilor am creat un generator de siruri de caractere random format din litere mari. Functia primeste ca parametru numarul de caractere din sir si intoarce un sir de caractere random cu lungimea specificata.

3.1 LPS

Am creat 10 teste cu siruri de caractere de lungimi diferite pe care am aplicat cei 4 algoritmi, obtinand timpii urmatiori in nanosecunde:

<i>Nr Test</i>	<i>Lungime sir</i>	<i>Timp Solutie1 - $O(n^3)$</i>	<i>Timp Solutie2 - $O(n^2)$</i>	<i>Timp Solutie3 - $O(n^2)$ clever</i>	<i>Timp Solutie4 - $O(n)$</i>
0	10	1930963	71053948	1114836	971787
1	50	1138087	41564585	64235	52018
2	100	1858848	84580545	186003	91425
3	500	55552644	39348705	159206	131621
4	750	155106039	75460485	225411	224229
5	1000	283548622	65291795	382646	397227
6	2500	3502552616	38649618	976516	952871
7	5000	22189841712	84470993	1848996	1060454
8	7500	45838801967	78188662	1166066	1451376
9	10000	68146367050	139598430	1628315	737707

Dupa cum era de asteptat timpii scad considerabil de la $O(n^3)$ la $O(n)$. Pe seturi mici diferenta nu este foarte mare, dar pe seturi mari diferenta este chiar de 10^5 nanosecunde.

3.1 LCS

Am creat 6 teste cu siruri de caractere de lungimi diferite pe care am aplicat cei 2 algoritmi. Cel de-al doilea sir de caractere l-am pastrat constant la valoarea de 8 caractere. Am obtinut timpii urmatoari in nanosecunde:

<i>Nr Test</i>	<i>Lungime sir de caractere</i>	<i>Timp Solutie1 - $O(2^n)$</i>	<i>Timp Solutie2 - $O(n^2)$</i>
0	3	2498036	605298
1	5	872481	40589
2	8	126104	49653
3	10	9614623	43742
4	15	272658381	67781
5	20	5725438634	47289

Am ales ca cel de-al doilea sir de caractere sa fie constant pentru a evidientia ce rezultate genereaza lungimile mici, egale sau mai mari ca 8.

Primul algoritm avand complexitate asimptotica exponentiala la $O(2^n)$ are timpi de executie mari odata cu cresterea lungimii setului de intrare. Am ales sa ma opresc la lungimea de 20 de caractere, deoarece timpul de executie ar fi fost foarte mare si oricum prin datele obtinute se observa diferenta considerabila intre cei doi algoritmi.

Solutia folosind programare dinamica, avand complexitatea $O(n^2)$ are timpi foarte mici in comparatie cu solutia recursiva.

4. Concluzii

Cele doua probleme analizate au o importanta mare in obtinerea de similiaritati intre siruri de caractere prin procesarea acestora. Algoritmii analizati au complexitati diferite, unii putand fi folositi doar pe seturi mici de date, astfel incat timpul sa fie relativ mic.

Algoritmii cu solutii eficiente s-au descurcat si au demonstrat rapiditatea pe seturi mari de date. Acesti algoritmi au provenit din observarea unor proprietati ale seturilor de date si a unor analize pentru a pastra generalitatea.

Astfel o problema poate fi redusa in timp de executie mai mic prin analiza proprietatilor si comportamentul seturilor de date.

5. Bibliografie

Pentru realizarea acestui raport am citit si analizat urmatoarele surse:

1. <http://articles.leetcode.com/longest-palindromic-substring-part-i/>
2. <http://articles.leetcode.com/longest-palindromic-substring-part-ii>
3. <https://www.quora.com/What-is-the-practical-use-of-finding-palindrome-in-any-programming-language>
4. <https://www.careercup.com/question?id=4128790>

5. https://en.wikipedia.org/wiki/Longest_palindromic_substring
6. <http://algorithms.tutorialhorizon.com/dynamic-programming-longest-common-subsequence/>
7. <http://www.cs.umd.edu/~meesh/351/mount/lectures/lect25-longest-common-subseq.pdf>
8. <http://m-hikari.com/asb/asb2012/asb5-8-2012/popovASB5-8-2012-1.pdf>