**Probleme specifice programării dinamice**

Determinarea subșirului maximal strict crescător

Se dă un șir cu n numere. Se cere determinarea celui mai lung subșir care are TOATE elementele în ordine strict crescătoare.

**Subșir**: se formează cu indici **"pe sărite"** din șirul inițial, indici care trebuie să fie în ordine crescătoare

(a nu se confunda cu **Secvență**: o "bucată" din șir, cu indici consecutivi <=> TOATE elementele dintr-o secvență sunt vecine atât în secvență cât și în șirul dat)

Ex:

fie șirul

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| a[i] | 10 | 7 | 2 | 6 | 3 | 9 | 4 | 5 | 20 | 15 | 3 | 17 | 25 | 30 | 11 | 5 |

Exemple de subșiruri: 7 3 4 20 25 (elementele fiind luate de pe indicii 2,5,7,9,13)

10 20 25 30 (elementele fiind luate de pe indicii 1,9,13,14)

(Btw: un șir cu **n** elemente admite **2n-1** subșiruri pentru că indicii formează de fapt **submulțimi** ale indicilor de la 1 la n, or, știm deja de la backtracking că numărul de submulțimi ale unei mulțimi cu n elemente este 2n (doar că pe noi NU ne interesează aia vidă)

DEJA din observația de mai sus, putem găsi un algoritm naiv, evident și FOARTE ineficient de a determina cel mai lung subșir crescător, prin backtracking: generăm submulțimile indicilor și le verificăm pe fiecare dacă dau un subșir crescător.

Exemple de secvențe: 20 15 3 17 (indicii 9,10,11,12) , 10 7 2 6 3 9 4 (indicii 1,2,3,4,5,6,7), 17 25 (indicii 12,13)

Rezolvarea problemei prin programare dinamică presupune găsirea unei relații de recurență, care să ne ajute la rezolvarea unei probleme de o anumită lungime în funcție de rezolvarea problemei pentru o lungime (niște lungimi) mai mici.

În cazul nostru, ne gândim cam ce anume ar fi nevoie să structurăm în plus față de șirul dat, astfel încât știind date despre problema de ordin k-1 (gen date despre subșiruri maximale crescătoare până la indicele k-1) să trecem la problema de ordin k.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **a[i]** | 10 | 7 | 2 | 6 | 3 | 9 | 4 | 8 |
| **t[i]** | 0 | 0 | 0 | 3 | 3 | 4 | 5 |  |
| **l[i]** | 1 | 1 | 1 | 2 | 2 | 3 | 3 |  |

Astfel, ajungem la următoarea concluzie:

în momentul în care luăm în calcul un element nou, pe baza celor de dinaintea lor, este important să putem determina care este cea mai convenabilă adăugare a elementului nou la vreunul dintre subșirurile determinate la pașii anteriori.

Pe exemplul de mai sus se poate vedea că ultimul element, de valoare 8, poate fi adăugat ca o "continuare" a unuia dintre subșirurile care se termină cu valori mai mici, adică în exemplul nostru 7, 2, 6, 3, 4. Evident, dintre acestea îl vom alege pe acela pentru care subșirul ce se obține va fi de lungime cât mai mare.

Așadar, la fiecare element mai luăm două informații, deci doi alți vectori în care le stocăm:

**t[i] =** indicele elementului anterior lui a[i] în subșirul cel mai convenabil (maximal crescător)

**l[i] =** lungimea subșirului maximal care se termină cu **a[i]**.

Dacă elementul curent **a[i]** NU poate fi adăugat la niciunul dintre subșirurile anterioare, atunci **t[i]=0** iar **l[i]=1** (cu el începe un noi potențial subșir)

Algoritmul care face trecerea (extinderea) de la pasul k-1 la pasul k este următorul:

**lmax=0;imax=0;**

**for(i=1;i<k;i++)**

**if(a[k]>a[i]&&l[i]>lmax)**

**{lmax=l[i];imax=i;}**

**l[i]=lmax+1;**

**t[i]=imax;**

După ce aplicăm această extindere pentru toate valorile de la 1 la n, avem toate datele pentru a determina subșirul cerut. Pentru asta mai trebuie să facem o parcurgere a vectorului l, să verificăm care este lungimea maximă (!NU e musai a ultimului element) și apoi mergând pe legătura tată să obținem subșirul crescător (el se obține în ordine inversă, dar asta nu e o problemă)

Ordinul de complexitate: O(n2).

Cantitativ, dacă n=100 se fac aprox. 10.000 de pași față de 2100=1.267.650.600.228.229.401.496.703.205.376 câți s-ar fi făcut prin backtracking.

Vezi **subs\_max\_cresc** - pe exemplul luat obținem:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **a[i]** | 10 | 7 | 2 | 6 | 3 | 9 | 4 | 5 | 20 | 15 | 3 | 17 | 25 | 30 | 11 | 5 |
| **t[i]** | 0 | 0 | 0 | 3 | 3 | 4 | 5 | 7 | 8 | 8 | 3 | 10 | 12 | 13 | 8 | 7 |
| **l[i]** | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 2 | 6 | 7 | 8 | 5 | 4 |

lungimea maximă a vreunui subșir=maximul din vectorul l=8. Mergând înapoi de la el pe legătura tată obținem și elementele subșirului.

**Determinarea subșirului maximal comun**

Se dau două șiruri de numere, unul cu n elemente și altul cu m elemente. Se cere să determinăm un subșir de lungime maximă, care să fie comun ambelor șiruri date.

Ex:

n=6 și a=(10,2,8,5,3,9)

m=8 și b=(2,5,8,10,4,5,9,3)

Vom defini o funcție (pe care apoi o calculăm într-o matrice)

**lmax(i,j)** cu semnficația:

lungimea subșirului maximal comun dintre secvențele **a[1..i]** și **b[1..j]**.

De exemplu, pentru șirurile noastre:

**lmax(1,1) = lmax(a[1..1],b[1..1])=0**

**lmax(1,2) = lmax(a[1..1],b[1..2])=0**

**lmax(2,1) = lmax(a[1..2],b[1..1])=1**

**lmax(3,3) = lmax(a[1..3],b[1..3])=2**

Relația de recurență este următoarea - exprimăm **lmax(i,j)** în funcție de valori anterioare.

**lmax(i,j)=**

În implementare folosim o matrice cu n+1 linii și m+1 coloane, indici de la 0..n, 0..m

Iată cum funcționează pentru exemplul luat.

Remarcăm că, dacă **a[i]==b[j]** atunci valoarea lui lmax se obține adăugând valoarea de pe diagonală stânga-sus cu 1, iar dacă sunt diferite, luând maximul dintre valoarea de deasupra și cea din stânga.

iată matricea lmax:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| b[j] și j  a[i] și i | 0 | 2  1 | 5  2 | 8  3 | 10  4 | 4  5 | 5  6 | 9  7 | 3  8 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 2 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 8 3 | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| 5 4 | 0 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 |
| 3 5 | 0 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 4 |
| 9 6 | 0 | 1 | 2 | 2 | 2 | 2 | 3 | 4 | 4 |

În concluzie subșirul maximal comun are lungimea 4.

Ca să îl recuperăm, e suficient să refacem traseul înapoi, plecând de la lmax[i=n,j=m] astfel:

- dacă a[i]==b[j] reținem acea valoare ca făcând parte din subșir, și apoi mergem pe diagonală stânga-sus : i--, j--

- dacă NU, mergem fie în stânga, fie în sus, și anume pe acea valoare EGALĂ cu lmax[i,j] (mergem pe "firul" care ne-a dat maximul). La egalitate, nu contează.

Iată cum arată drumul recuperat pe exemplul de mai sus:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| b[j] și j  a[i] și i | 0 | **2**  1 | 5  2 | **8**  3 | 10  4 | 4  5 | **5**  6 | **9**  7 | 3  8 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| **2** 2 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **8** 3 | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| **5** 4 | 0 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 |
| 3 5 | 0 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 4 |
| **9** 6 | 0 | 1 | 2 | 2 | 2 | 2 | 3 | 4 | 4 |

Valorile pentru care a[i]==b[j], în cazul nostru 2,8,5,9 ne dau subșirul maximal comun.