Programmazione dinamica (VII parte)

Progettazione di Algoritmi a.a. 2023-24

Matricole congrue a 1

Docente: Annalisa De Bonis

Sottosequenza crescente piu` lunga

elementi non necessariamente contigui

Questo algoritmo stampa la sottosequenza crescente piu` lunga.
Indichiamo con max l'indice in cui si trova l'elemento massimo di M, cioe`
M[max]=max{OPT(i): 0<=i<=n-1}

```
PrintLIS(A,i):
if i >= 0
    PrintLis(A,P[i])
    print(A[i])
```

prima chiamata con i=max

M e P costruiti in precedenza

Sottosequenza crescente piu` lunga

elementi non necessariamente contigui

Questa e` la versione iterativa dell'algoritmo LIS

```
1.  ITLIS(A)
2.    n = A.length
3.    for i=0 to n-1
4.      M[i]=1
5.      P[i]=-1
6.    for i=0 to n-1 //ogni iterazione computa OPT(i), i=0,...,n-1
7.      for j = 0 to i-1 //ogni iterazione considera OPT(j) se A[j]<A[i]
8.        if (A[j] < A[i])
9.          if M[j] + 1 >M[i]
10.             M[i]=M[j]+1
11.             P[i]=j
12.   max= M[0]
13.   indexmax=0
14.   for i=0 to n-1
15.     if M[i]>max
16.       max=A[i]
17.       indexmax=i
18.   return max
```

M[i]=lunghezza sottosequenza crescente piu` lunga che termina in A[i]
P[i]= indice predecessore di A[i] nella sottosequenza crescente piu` lunga che termina in A[i]

Sottosequenza crescente piu` lunga

elementi non necessariamente contigui

- Esempio: A=<3 12 9 4 12 5 8 11 6 13 10>

	3	12	9	4	12	5	8	11	6	13	10
M	1	2	2	2	3	3	4	5	4	6	5
P	-1	0	0	0	3	3	5	6	5	7	8

- Per aggiornare M[1] nel for esterno (linea 6) consideriamo solo A[0] nel for interno (linea 7):
  - max{M[1],M[0]+1}={1,2}=2
- Per aggiornare M[2] nel for esterno (linea 6) consideriamo A[0] e A[1] nel for interno (linea 7) ma A[0] è l'unico < A[2]:
  - max{M[2],M[0]+1}={1,2}=2
- Per aggiornare M[3] nel for esterno (linea 6) consideriamo A[0] , A[1] e A[2] nel for interno (linea 7) ma A[0] è l'unico < A[3]:
  - max{M[3],M[2]+1}={1,2}=2
- Per aggiornare M[4] nel for esterno (linea 6) consideriamo A[0] , A[1], A[2] e A[3] nel for interno (linea 7). Tra questi solo A[0], A[2] e A[3] sono < A[4]:
  - max{M[4],M[0]+1,M[2]+1, M[3]+1}={1,2,3}=3
- Per aggiornare M[5] nel for esterno (linea 6) consideriamo A[0], …, A[4] nel for interno (linea 7). Tra questi solo A[0] e A[3] sono < A[5]:
  - max{M[5],M[0]+1,M[3]+1}={1,2,3}=3
- ecc.

Sottosequenza crescente piu` lunga elementi contigui

- Esercizio:
- Scrivere un algoritmo per calcolare la lunghezza della sottosequenza crescente piu` lunga nel caso in cui la sottosequenza deve consistere di elementi consecutivi possiamo usare divide et impera.
- Nel caso in cui la sottosequenza deve consistere di elementi consecutivi possiamo usare divide et impera.

114

Sottosequenza crescente piu` lunga elementi contigui

- Vogliamo calcolare la lunghezza della sottosequenza crescente piu` lunga.
- Nel caso in cui la sottosequenza deve consistere di elementi consecutivi possiamo usare divide et impera.

```
1.  LISCons(A,p,q):
2.  if p>q return 0
3.  if p==q return 1
4.  c=(p+q)/2
5.  n1= LISCons(A,p,c)
6.  n2= LISCons(A,c+1,q)
7.  if A[c]<A[c+1]{
8.       nc=1;
9.       j=c
10.      while(j>p && A[j]>A[j-1]) {  j--, nc++}
11.      j=c+1
12.      while(j<q && A[j]<A[j+1])  {j++, nc++}
13. }
14. return max(n1,n2,nc)
```

O(nlog n) ma si puo` fare di meglio…

115

Sottosequenza crescente piu` lunga elementi contigui

- Vogliamo calcolare la lunghezza della sottosequenza crescente piu` lunga.
- Nel caso in cui la sottosequenza deve consistere di elementi consecutivi possiamo usare divide et impera.

```
1.  LISCons(A,s,d):
2.  if s>d return 0
3.  if s==d return 1
4.  c=(s+d)/2
5.  if A[c]<A[c+1]{
6.        i=c
7.        nc=1
8.        while(i>s && A[i]>A[i-1]) {  i--, nc++}
9.        j=c
10.       while(j<d && A[j]<A[j+1]) {  j++, nc++}
11.   }
12. if(i>nc)  n1= LISCons(A,s,i-1) else n1=0
13. if(d-j>nc)  n2= LISCons(Aj+1,d) else n2=0
14. return max(n1,n2,nc)
```

O(n) vediamo perche'

Sottosequenza crescente piu` lunga elementi contigui

1. T(n)<=c' se n<=1
2. T(n)<= c''n se n>1 e l'algoritmo effettua il lavoro di decomposizione ma non effettua la ricorsione
3. T(n)<=c'''(j-i)+T(i)+T(n-j) altrimenti, dove c',c'' e c''' sono costanti maggiori di zero

- NB: se viene fatta solo una delle due chiamate ricorsive allora T(n)<=c'''(j-i)+max{T(i),T(n-j)} <=c'''(j-i)+T(i)+T(n-j) per cui vale comunque la 3.
- Dimostriamo per induzione che T(n)=O(n), cioe` che T(n)<=cn per un certo c>0 per tutti gli n maggiori o uguali di un certo n_0 >=0
- **Caso base** : n=1 T(n)<=c'=c'*1. Perche' sia T(n)<=cn basta prendere c>=c'
- **Passo induttivo:** Assumiamo che T(m)<=cm per ogni intero positivo m<n. Dimostriamo che vale T(n)<cn.
- Se si verifica il caso 2 allora basta porre c>= c'' perche' valga la disequazione T(n)<=cn
- Se si verifica il caso 3 allora si usa la ricorrenza T(n)<=c'''(j-i)+T(i)+T(n-j). Applicando l'ipotesi induttiva a T(i) e T(n-j) abbiamo che T(i)<ci e T(n-j)<c(n-j) da cui si ha che T(n)<=c'''(j-i)+ci+c(n-j)= c'''(j-i)+c(n-(j-i)) Perche' valga T(n)<=cn basta prendere c>=c'''
- Poniamo quindi n_0 =1 e c=max(c',c'',c'''}

Sottosequenza crescente piu` lunga elementi contigui

- possiamo usare idea simile a quella usata per trovare sottosequenza di somma max
- L_j = lunghezza sottosequenza crescente piu` lunga che termina in A[j]
- $L_{j+1} = L_j+1$ se $A[j]<A[j+1]$; $L_{j+1} = 1$ altrimenti

```
1.  LISCons(A,n):
2.   L[0]=1
3.   primo[0]=0
4.   massimo=1
5.   for i=1 to n-1
6.      if A[i]>A[i-1]
7.         L[i]=L[i-1]+1
8.         primo[i]=primo[i-1]
9.      else
10.         L[i]=1
11.         primo[i]=i
12.     if massimo < L[i]
13.        massimo=L[i]
14. return massimo
```

primo[i] contiene l'indice dell'elemento iniziale della sequenza crescente piu` lunga che termina in A[i]

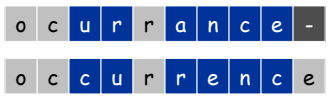
la sequenza crescente piu` lunga parte da A[primo[massimo] e finiesce in A[massimo]

O(n)

118

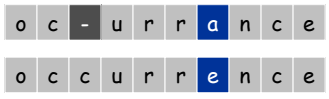
Allineamento di sequenze

- Quanto sono simili le due stringhe seguenti ?
  - **ocurrance**
  - **occurrence**



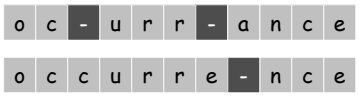
6 mismatch, 1 gap

  - Allineiamo i caratteri
  - Ci sono diversi modi per fare questo allineamento



1 mismatch, 1 gap

  - Qual e` migliore?



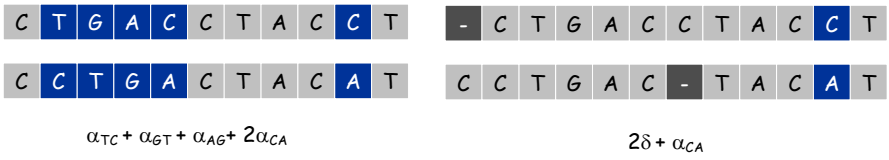
0 mismatch, 3 gap

119

Edit Distance

- Applicazioni.
  - Base per il comando Unix diff.
  - Riconoscimento del linguaggio.
  - Biologia computazionale.

- Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]
  - Gap penalty δ;
  - Mismatch penalty α_{pq}. Si assume $\alpha_{pp}=0$



$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$

$2\delta + \alpha_{CA}$

120

Applicazione del problema dell'allineamento di sequenze

- I problemi su stringhe sorgono naturalmente in biologia: il genoma di un organismo è suddiviso in molecole di DNA chiamate cromosomi, ciascuno dei quali serve come dispositivo di immagazzinamento chimico.

- Di fatto, si può pensare ad esso come ad un enorme nastro contenente una stringa sull'alfabeto {*A*,*C*,*G*,*T*}. La stringa di simboli codifica le istruzioni per costruire molecole di proteine: usando un meccanismo chimico per leggere porzioni di cromosomi, una cellula può costruire proteine che controllano il suo metabolismo.

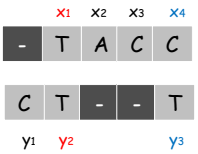
121

Applicazione del problema dell'allineamento di sequenze

- Perche` le somiglianze tra stringhe sono rilevanti in questo scenario?
- Le sequenze di simboli nel genoma di un organismo determinano le proprieta` dell'organismo.
- Esempio. Supponiamo di avere due ceppi di batteri X e Y che sono strettamente connessi dal punto di vista evolutivo.
- Supponiamo di aver determinato che una certa sottostringa nel DNA di X sia la codifica di una certa tossina.
- Se scopriamo una sottostringa molto simile nel DNA di Y, possiamo ipotizzare che questa porzione del DNA di Y codifichi un tipo di tossina molto simile a quella codificata nel DNA di X.
- Esperimenti possono quindi essere effettuati per convalidare questa ipotesi.
- Questo e` un tipico esempio di come la computazione venga usata in biologia computazionale per prendere decisioni circa gli esperimenti biologici.

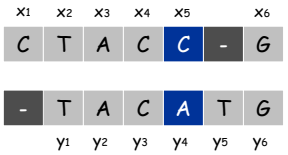
Allineamento di sequenze

- Abbiamo bisogno di un modo per allineare i caratteri di due stringhe
  - Formiamo un insieme di coppie di caratteri, dove ciascuna coppia e` formata da un carattere della prima stringa e uno della seconda stringa.
- Def. insieme di coppie è un matching se ogni elemento appartiene ad al più una coppia
- Def. Le coppie (x_i,y_j) e $(x_{i'},y_{j'})$ si incrociano se $i < i'$ ma $j > j'$.
  - (x_1, y_2) e (x_4,y_3) non si incrociano
  - (x_2, y_3) e (x_4,y_2)



Def. Un allineamento M è un insieme di coppie (x_i,y_j) tali che
- M e` un matching
- M non contiene coppie che si incrociano

Esempio: CTACCG **VS.** TACATG
Allineamento:
M = (x_2,y_1), (x_3,y_2), (x_4,y_3), (x_5,y_4), (x_6,y_6).



Allineamento di sequenze

- Obiettivo: Date due stringhe X = $x_1 x_2 \ldots x_m$ e Y = $y_1 y_2 \ldots y_n$ trova l'allineamento di minimo costo.

$$\text{cost}(M) = \underbrace{\sum_{(x_i,y_j)\in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i:x_i \text{ unmatched}} \delta + \sum_{j:y_j \text{ unmatched}} \delta}_{\text{gap}}$$

- Affermazione.
- Dato un allineamento M di due stringhe X = $x_1 x_2 \ldots x_m$ e Y = $y_1 y_2 \ldots y_n$, se in M non c'è la coppia (x_m,y_n) allora o x_m non e` accoppiato in M o $y_n$ non e` accoppiato in M.
- Dim. Supponiamo che x_m e y_n sono entrambi accoppiati ma non tra di loro. Supponiamo che x_m sia accoppiato con y_j e y_n sia accoppiato con x_i. In altre parole M contiene le coppie (x_m,y_j) e (x_i,y_n). Siccome i<m ma n>j allora si ha un incrocio e ciò contraddice il fatto che M è allineamento.

Allineamento di sequenze: struttura del problema

- Def. OPT(i, j) = costo dell'allineamento ottimo per le due stringhe $x_1 x_2 \ldots x_i$ e $y_1 y_2 \ldots y_j$.
  - Caso 1: x_i e y_j sono accoppiati nella soluzione ottima per $x_1 x_2 \ldots x_i$ e $y_1 y_2 \ldots y_j$
    - OPT(i, j) = Costo dell'eventuale mismatch tra x_i e y_j + costo dell'allineamento ottimo di $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$
  - Caso 2a: x_i non e` accoppiato nella soluzione ottima per $x_1 x_2 \ldots x_i$ e $y_1 y_2 \ldots y_j$
    - OPT(i, j) = Costo del gap x_i + costo dell'allineamento ottimo di $x_1 x_2 \ldots x_{i-1}$ e $y_1 y_2 \ldots y_j$
  - Case 2b: y_j non e` accoppiato.nella soluzione ottima per $x_1 x_2 \ldots x_i$ e $y_1 y_2 \ldots y_j$
    - OPT(i, j) = Costo del gap y_j + costo dell'allineamento ottimo di $x_1 x_2 \ldots x_i$ e $y_1 y_2 \ldots y_{j-1}$

$$OPT(i,j) = \begin{cases} j\delta & \text{se } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{altrimenti} \\ i\delta & \text{se } j = 0 \end{cases}$$

Allineamento di sequenze: algoritmo

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[i, 0] = iδ
   for j = 0 to n
      M[0, j] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

- Analisi. Tempo e spazio $\Theta(mn)$.
- Parole inglesi: m, n $\leq$ 10.
- Applicazioni di biologia computazionale: m = n = 100,000.
- Quindi mxn=10 miliardi . OK per il tempo ma non per lo spazio (10GB)