Subset sums

- Input
  - n job 1,2,…,n
    - il job i richiede tempo $w_i > 0$.
  - Un limite W al tempo per il quale il processore puo` essere utilizzato
- Obiettivo: selezionare un sottoinsieme S degli n job tale che $\sum_{i \in S} w_i$ sia quanto piu` grande e` possibile, con il vincolo $\sum_{i \in S} w_i \leq W$

Greedy 1: ad ogni passo inserisce in S il job con peso piu` alto in modo che la durata complessiva dei job in S non superi W

Esempio: Input una volta ordinato [W/2+1,W/2,W/2]. L'algoritmo greedy seleziona solo il primo mentre la soluzione ottima e` formata dagli ultimi due.

Greedy 2: ad ogni passo inserisce in S il job con peso piu` basso in modo che la duranta complessiva dei job in S non superi W

Esempio: Input [1,W/2,W/2] una volta ordinato . L'algoritmo greedy seleziona i primi due per un peso complessivo di 1+W/2. mentre la soluzione ottima e` formata dagli ultimi due di peso complessivo W.

Programmazione dinamica: falsa partenza

- Def. OPT(i) = valore della soluzione ottima per {1, …, i}.
  - Caso 1: OPT non seleziona i.
    - OPT seleziona la soluzione ottima per { 1, 2, …, i-1 }
  - Caso 2: OPT seleziona i.
    - Prendere i non implica immediatamente l'esclusione di altri elementi.
    - Se non conosciamo i job selezionati prima di i, non sappiamo neanche se c'e` tempo sufficiente per eseguire i
- Conclusione. Approccio sbagliato!

Programmazione dinamica: approccio corretto

- Per esprimere il valore della soluzione ottima per un certo i in termini dei valori delle soluzioni ottime per input piu` piccoli di i, dobbiamo introdurre un limite al tempo totale da dedicare all'esecuzione dei job selezionati prima di i.

- Per ciascun j, consideriamo il valore della soluzione ottima per i job 1, …, j con il vincolo che il tempo necessario per eseguire i job selezionati non superari un certo w.

-

- Def. OPT(i, w) = valore della soluzione ottima per i job 1, …, i con limite w sul tempo di utilizzo del processore.

Programmazione dinamica: approccio corretto

- Def. OPT(i, w) = valore della soluzione ottima per i job 1, …, i con limite w sul tempo di utilizzo del processore.

- Caso 1: OPT non seleziona il job i.
  - OPT produce la soluzione ottima per { 1, 2, …, i-1 } in modo che il tempo di esecuzione totale dei job non superi w

- Case 2: OPT seleziona il job i.
  - OPT produce la soluzione ottima per { 1, 2, …, i–1 } in modo che il tempo di esecuzione totale dei job non superi w– w_i

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ \ w_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Subset sums: algoritmo

- Versione iterativa in cui si computa la soluzione in modo bottom-up
- Si riempie un array bidimensionale nxW a partire dalle locazioni di indice di riga i piu` piccolo

```
Input: n, w1,…,wN,W

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}

return M[n, W]
```

Subset sums: esempio di esecuzione dell'algoritmo

Limite $W = 6$, durate job $w_1 = 2, w_2 = 2, w_3 = 3$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
(1)	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 1$

3							
(2)	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 2$

(3)	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 3$

Subset sums: correttezza algoritmo

- Induzione sui i. Dimostriamo che all'i-esima iterazione ogni riga di M con indice r compreso tra 0 e i contiene i valori OPT(r,0),OPT(r,1),….,OPT(r, W)

- Base induzione. i=0: La riga 0 ha correttamente tutte le entrate uguali a 0
- Passo induttivo: Supponiamo che alla iterazione i-1 ≥0, le righe di indice r compreso tra 0 e i-1 contengano i valori OPT(r,0),OPT(r,1),….,OPT(r, W).
- Vediamo cosa succede all'i-esima iterazione.
- Per ipotesi induttiva M[i-1, w]=OPT(i-1,w) ed M[i-1, w-w_i]=OPT(i-1,w-w_i)
- All'i-esima iterazione, l'algoritmo pone

M[i, w] = M[i-1, w] = OPT(i-1,w) **se** w_i **>** **w**

M[i, w] = max {M[i-1, w], w_i + M[i-1, w-w_i]}
= max {OPT[i-1, w], w_i + OPT[i-1, w-w_i]} altrimenti

- Per cui M[i,w] = OPT(i,w)

Subset sums: tempo di esecuzione algoritmo

- Tempo di esecuzione. $\Theta(n\ W)$.

  - Non e` polinomiale nella dimensione dell'input!
  - "Pseudo-polinomiale": L'algoritmo e` efficiente quando W ha un valore ragionevolmente piccolo.

  - Se volessimo produrre la soluzione ottima, potremmo scrivere un algoritmo simile a quelli visti prima in cui la soluzione ottima si ricostruisce andando a ritroso nella matrice M. Tempo O(n).

  - Esercizio: Scrivere lo pseudocodice dell'algoritmo che produce la soluzione ottima per un'istanza di Subset Sum.

Problema dello zaino

- Input
  - n oggetti ed uno zaino
  - L'oggetto i pesa $w_i > 0$ chili e ha valore $v_i > 0$.
  - Lo zaino puo` trasportare fino a W chili.
- Obiettivo: riempire lo zaino in modo da massimizzare il valore totale degli oggetti inseriti.
- Esempio: { 3, 4 } ha valore 40.

W = 11

Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Greedy: seleziona ad ogni passo l'oggetto con il rapporto v_i/w_i piu` grande in modo che il peso totale dei pesi selezionati non superi w
- Esempio: soluzione greedy { 5, 2, 1 } ha valore = 35 $\Rightarrow$ greedy non e` ottimo

Problema dello zaino

- Input
  - n oggetti ed uno zaino
  - L'oggetto i pesa $w_i > 0$ chili e ha valore $v_i > 0$.
  - Lo zaino puo` trasportare fino a W chili.
- Obiettivo: riempire lo zaino in modo da massimizzare il valore totale degli oggetti inseriti.
- Corrisponde al problema subset sums quanto $v_i = w_i$ per ogni i.

Problema dello zaino: estensione approccio usato per Subset Sums

- Def. OPT(i, w) = valore della soluzione ottima per gli oggetti 1, …, i con limite di peso totale w.

  - Caso 1: OPT non seleziona l'elemento i.
    - OPT produce la soluzione ottima per { 1, 2, …, i-1 } in modo che il peso totale dgli elementi non superi w

  - Case 2: OPT seleziona l' elemento i.
    - OPT produce la soluzione ottima per { 1, 2, …, i–1 } in modo che il peso totale degli elementi non superi w– w_i

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\quad v_i + OPT(i-1,w-w_i)\,\} & \text{otherwise} \end{cases}$$

Problema dello zaino: algoritmo

```
Input: n, W,w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

Algoritmo per il problema della zaino: esempio

W →

n ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

$OPT(5,11) = OPT(4,11) = v_4 + OPT(3,11-w_4) = v_4 + OPT(3,5)$
$= v_4 + v_3 + OPT(2,0) = v_4 + v_3 + OPT(1,0)$
$= v_4 + v_3 + OPT(0,0) = 22 + 18 + 0 = 40$

Soluzione ottima : { 4, 3 }
Valore soluzione ottima = 22 + 18 = 40

W = 11

Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Esercizio 27 cap. 6

- I proprietari di una pompa di carburante devono confrontarsi con la seguente situazione:
- Hanno un grande serbatioio che immagazzina gas; il serbatoio puo` immagazzinare fino ad L galloni alla volta.
- Ordinare carburante e` molto costoso e per questo essi vogliono farlo raramente. Per ciascun ordine pagano un prezzo fisso P in aggiunta al costo della carburante.
- Immagazzinare un gallone di carburante per un giorno in piu` costa c dollari per cui ordinare carburante troppo in anticipo aumenta i costi di immagazzinamento.
- Essi stanno progettando di chiudere per una settimana durante l'inverno e vogliono che per allora il serbatoio sia vuoto.
- In base all'esperienza degli anni precedenti, essi sanno esattamente di quanto carburante hanno bisogno. Assumendo che chiuderanno dopo n giorni e che hanno bisogno di g_i galloni per ciascun giorno i=1,…n e che al giorno 0 il serbatoio e` vuoto, dare un algoritmo per decidere in quali giorni devono effettuare gli ordini e la quantita` di carburante che devono ordinare in modo da minimizzare il costo.

Esercizio 27 cap. 6: Soluzione

- Supponiamo che il giorno 1 i proprietari ordinino carburante per i primi i-1 giorni: $g_1+g_2+\ldots+g_{i-1}$
- Il costo di questa operazione si traduce in un costo fisso di P piu` un costo di $c(g_2+2g_3+3g_4\ldots+(i-2)g_{i-1})$

- Sia OPT(d) il costo della soluzione ottima per il periodo che va dal giorno d al giorno n partendo con il serbatoio vuoto
- La soluzione ottima da d ad n include sicuramente un ordine al giorno d per un certo quantitativo di carburante. Sia f il giorno in cui verra` fatto il prossimo ordine.
  - La quantita ordinata il giorno d deve essere $g_d+g_{d+1}+\ldots+g_{f-1}$. ($g_d+g_{d+1}+\ldots+g_{f-} \leq L$)
  - Il costo connesso a questo ordine e` $P+c(g_{d+1}+2g_{d+2}+3g_{d+3}\ldots+(f-1-d)g_{f-1})$
  - Il costo della soluzione ottima da d ad n se il secondo ordine in questo intervallo avviene al tempo f e` $P + \sum_{i=d}^{f-1} c(i-d)g_i + OPT(f)$

$$OPT(d) = P + \min_{f>d:\sum_{i=d}^{f-1} g_i \leq L} \sum_{i=d}^{f-1} c(i-d)g_i + OPT(f)$$

Esercizio 27 cap. 6: Soluzione

Possiamo scrivere un algoritmo iterativo che computa le soluzioni a partire da d = n fino a d=1 e memorizza le soluzioni in un array

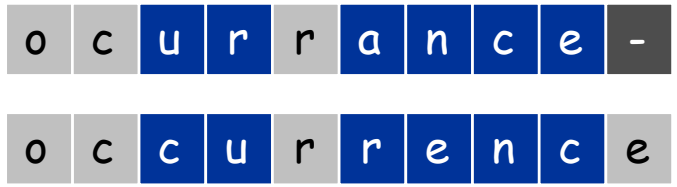
```
Input: n, L, g1,…,gn
A[d,f]  //contiene la somma dei gi per i=d,…,f
S[d,f]   //contiene la somma dei gi(d-i)per i=d,…f
M[d] //contiene soluzione ottima da d ad n


For d = 1 to n
  A[d,d]=gd
  S[d,d]=0

For d = n to 1{                                          O(n2)
   min= large_value
   For  x=d to n  {      //x=f-1
      If A[d,x-1]+gx<=L{
         A[d,x]=A[d,x-1]+gx
         S[d,x] =S[d,x-1]+ (x-d)gx
         If P+S[d,x]+ M[x+1]<min
           min= P+S[d,x]+ M[x+1]
         M[d]=min}
   Else  break }}//ha computato l'ottimo per giorni da d ad n
return M[1]
```

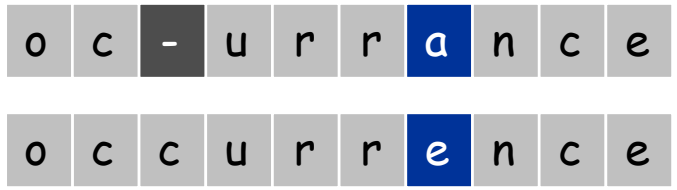
Allineamento di sequenze

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



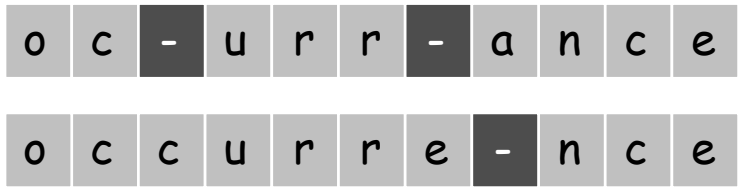
6 mismatch, 1 gap

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



1 mismatch, 1 gap

- Qual e` migliore?

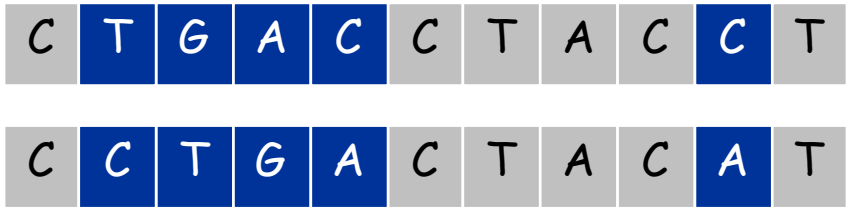


0 mismatch, 3 gap

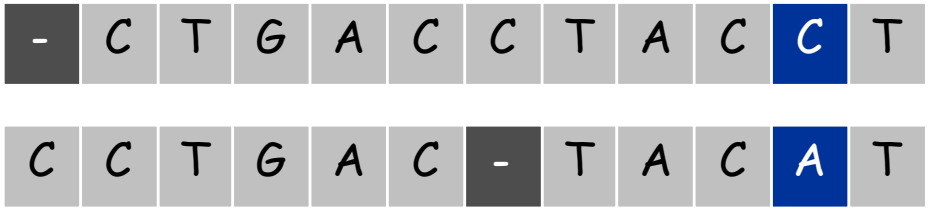
Edit Distance

- 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 $\alpha_{pp}=0$
  - Costo = somma delle due penalita`



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



$2\delta + \alpha_{CA}$

Applicazione del problema dell'allineamento di sequenze

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

- Di fatto, si puo` 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 puo` costruire proteine che controllano il suo metabolismo.

Continua nella prossima slide

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.

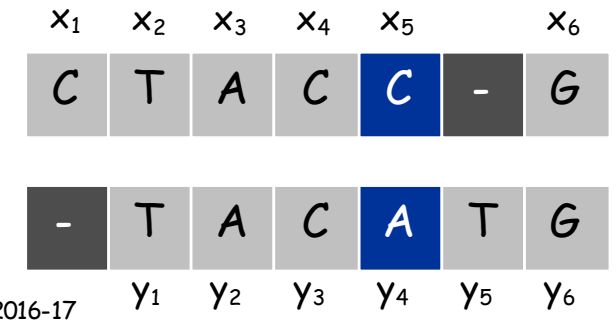
- Def. insieme di coppie e` un matching se ogni elemento appartiene ad una sola coppia

- Def. Le coppie (x_i, y_j) e $(x_{i'}, y_{j'})$ si incrociano se $i < i'$ ma $j > j'$.

Def. Un allineamento M e` 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.

Soluzione: $M = (x_2, y_1), (x_3, y_2), (x_4, y_3), (x_5, y_4), (x_6, y_6)$.



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'e` 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 cio` contraddice il fatto che M e` allineamento.

Allineamento di sequenze: struttura del problema

- Def. OPT(i, j) = costo minimo dell'allineamento delle due stringhe $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.
  - Caso 1: OPT accoppia x_i e y_j.
    - OPT(i, j) = Costo dell'eventuale mismatch tra x_i e y_j + costo minimo dell'allineamento di $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$
  - Caso 2a: OPT lascia x_i non accoppiato.
    - OPT(i, j) = Costo del gap x_i + costo minimo dell'allineamento di $x_1 x_2 \ldots x_{i-1}$ e $y_1 y_2 \ldots y_j$
  - Case 2b: OPT lascia y_j non accoppiato.
    - OPT(i, j) = Costo del gap y_j + costo minimo dell'allineamento di $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$

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

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)