Algoritmi e Strutture Dati
Esercizi Svolti

Giuseppe Persiano
Dipartimento di Informatica ed Appl. Renato M. Capocelli
Università di Salerno

Indice

Sommario

Questo documento contiene esercizi svolti per il corso di Algoritmi e Strutture Dati che l'autore tiene presso la Facoltà di Scienze dell'Università di Salerno.

La versione aggiornata di questo documento si trova
http://libeccio.dia.unisa.it/ASD2005/Esercizi2005/Esercizi.pdf.

Esercizio 1.2-3

Qual è il più piccolo valore di n per cui un algoritmo con tempo di esecuzione $100n^2$ viene eseguito più velocemente di un algoritmo con tempo di esecuzione 2^n sulla stessa macchina.

Soluzione. Dobbiamo trovare il più piccolo $n > 0$ per cui $100n^2 < 2^n$. Abbiamo che, per $n = 14$, $100n^2 = 19600 > 16384 = 2^{14}$ ma per $n = 15$ abbiamo $100n^2 = 22500 < 32768 = 2^{15}$.

Esercizio 2.3-4

L'algoritmo Insertion-Sort può essere espresso come una procedura ricorsiva nel modo seguente: per ordinare $A[1 \cdots n]$, si ordina in modo ricorsivo $A[1 \cdots n-1]$ e poi si inserisce $A[n]$ nell'array ordinato $A[1 \cdots n-1]$.

Scrivete una ricorrenza per il tempo di esecuzione di questa versione ricorsiva di insertion sort.

Soluzione. Denotiamo con $T(n)$ il tempo impiegato nel caso peggiore dalla versione ricorsiva dell'algoritmo Insertion-Sort. Abbiamo che $T(n)$ è uguale al tempo per ricorsivamente ordinare un array di $n-1$ elementi (che è uguale a $T(n-1)$) più il tempo per inserire $A[n]$ (che nel caso peggiore è $\Theta(n)$). Pertanto abbiamo

$$T(n) = T(n-1) + \Theta(n).$$

Esercizio 6.3-3

Si consideri un heap T con n nodi. Definiamo la profondità $d(v)$ di un nodo $v \in T$ come la lunghezza del più lungo cammino da v ad una foglia discendente di v. Denotiamo inoltre con n_h il numero di nodi di profondità h.

LEMMA 1. *Per ogni heap T con n nodi per ogni $h \geq 0$ abbiamo che*

$$n_h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil .$$

DIMOSTRAZIONE. Procediamo per induzione.

Base: $h = 0$. Dimostriamo che $n_0 = \lceil \frac{n}{2} \rceil$. Notiamo che n_0 è il numero di foglie.

Denotiamo con x il numero di nodi che si trovano a distanza massima dalla radice e distinguiamo due casi.

(1) n è dispari. In questo caso abbiamo che x è pari e quindi tutti i nodi (che chiameremo interni) che hanno dei figli hanno in realtà esattamente due figli. In questo caso abbiamo che il numero dei nodi interni è uguale al numero delle foglie meno 1. Pertanto abbiamo che

$$n = 2 \cdot \#\text{ foglie } - 1$$

da cui

$$n_0 = \#\text{ foglie } = \frac{n+1}{2}$$

e siccome n è dispari abbiamo

$$n_0 = \left\lceil \frac{n}{2} \right\rceil .$$

(2) n è pari. In questo caso x è dispari e pertanto abbiamo una foglia v (la foglia più a destra all'ultimo livello) che non ha fratello. Consideriamo quindi l'albero T' che si ottiene da T aggiungendo ad esso il nodo fratello della foglia v. Ovviamente denotando con n' il numero di nodi di T' abbiamo che $n' = n + 1$ e $n_0 = n'0 - 1$. Inoltre n' è dispari e quindi, per il ragionamento al punto precedente, abbiamo che

$$n_0 = n'_0 - 1 = \left\lceil \frac{n'}{2} \right\rceil - 1 = \left\lceil \frac{n+1}{2} \right\rceil - 1$$

e siccome n è pari abbiamo

$$\left\lceil \frac{n+1}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil + 1$$

da cui deriva

$$n_0 = \left\lceil \frac{n}{2} \right\rceil .$$

Passo induttivo: Assumiamo che il lemma valga per tutti gli alberi e per tutte le profondità $< h$. Proviamo il lemma per h.

Consideriamo l'albero T' che si ottiene da T rimuovendo tutte le foglie. Notiamo che $n' = n - n_0 = \lfloor \frac{n}{2} \rfloor$. Inoltre $n_h = n'_{h-1}$ (n'_{h-1} denota il numero di nodi di T' a profondità $h-1$). Applicando l'ipotesi induttiva a T' abbiamo

$$n'_{h-1} \leq \left\lceil \frac{n'}{2^h} \right\rceil = \left\lceil \frac{\lfloor \frac{n}{2} \rfloor}{2^h} \right\rceil \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil .$$

□

Il lemma appena dimostrato è utilizzato per provare che la procedura BuildHeap impiega tempo $O(n)$ quando è eseguita con input un albero di n elementi. La procedura BuildHeap spende tempo $O(h)$ per ogni vertice dell'albero di profondità h. Pertanto il numero di passi esseguito è

$$O\left(\sum_{i=0}^{\lceil \log n \rceil} n_h \cdot h\right) = O\left(n \sum_{i=0}^{\lceil \log n \rceil} \frac{h}{2^h}\right) = O\left(n \sum_{i=0}^{\infty} \frac{h}{2^h}\right).$$

Siccome, per ogni x con $|x| < 1$,

$$\sum_{i=0}^{\infty} i x^i = \frac{x}{(1-x)^2}$$

possiamo concludere che BuildHeap impiega tempo lineare.

Esercizio 6.5-8

Descrivete un algoritmo con tempo $O(n \log k)$ per fondere k liste ordinate in un'unica lista ordinata, dove n è il numero totale di elementi di tutte le liste date in input.

Soluzione. L'algoritmo MERGE per fondere due liste ordinate mantiene per ogni lista il puntatore al più piccolo elemento che non è stato ancora dato in output e ad ogni passo dà in output il minimo tra i due elementi. Un'ovvia generalizzazione dell'algoritmo richiede di poter calcolare il minimo tra k elementi. Ciò può essere ovviamente fatto in tempo $O(k)$ dando quindi un algoritmo di fusione di k liste che prende tempo $O(kn)$. Possiamo migliorare il tempo per la fusione utilizzando un min-heap nel modo seguente. Il min-heap contiene ad ogni passo il più piccolo elemento di ciascuna delle k liste che non è stato dato ancora in output. Ad ogni passo estraiamo il minimo dall'heap (questa operazione prende tempo $O(\log k)$ in quanto l'heap contiene al più k elementi) ed inseriamo il prossimo elemento più piccolo (se esiste) che proviene dalla stessa lista cui appartiene il minimo appena estratto (anche questa operazione prende tempo $O(\log k)$). In totale spendiamo tempo $O(n \log k)$.

Versione: 1.3 del 19 novembre 2005.

Esercizio 7.1-2

Quale valore di q restituisce l'algoritmo PARTITION *quando tutti gli elementi di* $A[p \dots r]$ *hanno lo stesso valore? Argomentare la risposta.*

Soluzione. Riportiamo la procedura PARTITION.

```
PARTITION(A, p, r)
1  x ← A[r];
2  i ← p − 1;
3  for j ← p to r − 1
4      do if A[j] ≤ x
5             bf then i ← i + 1;
6                  scambia A[i] ↔ A[j];
7  scambia A[i + 1] ↔ A[r];
8  return i + 1;
```

Se tutti gli elementi di $A[p \dots r]$ sono uguali ad x allora la condizione di riga 4 è sempre soddisfatta e quindi i è incrementato da ogni iterazione. Poiché i assume inizialmente il valore $p - 1$ (riga 2), al termine del ciclo **for** di linea 3, i ha il valore $r - 1$. Quindi PARTITION restituisce il valore r.

Di seguito diamo una seconda possibile soluzione. Sappiamo che durante l'esecuzione di PARTITION gli elementi che si trovano nell'array $A[p \dots i]$ sono $\leq x$ e che gli elementi che si trovano nell'array $A[i + 1, j - 1]$ sono $> x$. Se tutti gli elementi sono uguali allora non esiste nessun elemento dell'array maggiore di x. Quindi l'array $A[i + 1, j - 1]$ deve essere vuoto e cioè $i = j$ durante tutto il ciclo **for** di linea 3. Poichè al termine di PARTITION $j = r - 1$, abbiamo che $i = r - 1$ e PARTITION restituisce il valore $i + 1 = r$.

Esercizio 8.1-3

Dimostrate che non esiste un ordinamento per confronti il cui tempo di esecuzione è lineare per almeno metà degli $n!$ input di lunghezza n. Che cosa accade per una frazione $1/n$ degli input di lunghezza n? E per una frazione $1/2^n$?

Soluzione. Supponiamo che esista una costante c ed un algoritmo di ordinamento per confronti che effettua al più cn confronti per almeno metà degli $n!$ input di lunghezza n. Allora il corrispondente albero dei confronti deve avere almeno $n!/2$ foglie ad altezza al più cn. Però un albero di altezza cn ha al più 2^{cn} foglie e, per tutte le costanti c e per n sufficientemente grande, $2^{cn} < n!/2$. Se invece ci accontentiamo di una frazione $1/n$ degli input, il numero di foglie di profondità al più n deve essere almeno $(n-1)!$ e, per n sufficientemente grande, $2^{cn} < (n-1)!$. Il terzo caso è risolto osservando che, per n sufficientemente grande, $2^{cn} < n!/2^n$.

Esercizio 8.1-4

Sia data una sequenza di n elementi da ordinare. La sequenza di input è formata da n/k sottosequenze, ciascuna di k elementi. Gli elementi di una sottosequenza sono più piccoli degli elementi della sottosequenza successiva e più grande degli elementi della sottosequenza precedente. Dimostrare che $\Omega(n \log k)$ è un limite inferiore sul numero di confronti necessari a risolvere questa variante del problema dell'ordinamento.

Soluzione. Notiamo che non è corretto combinare semplicemente i limiti inferiori delle singole sequenze. Questo argomento considera solo gli algoritmi che ordinano ciascuna sottosequenza indipendentemente dalle altre. L'esercizio ci richiede invece un limite inferiore a *tutti* gli algoritmi (anche quelli che combinano in qualche modo le sottosequenze prima di ordinarle).

La prova che proponiamo invece segue lo schema adottato per provare il limite sull'ordinamento: contiamo il numero di possibili permutazioni che possiamo avere in output e il logaritmo di questa quantità dà un lower bound al limite di confronti.

Per ciascuna sequenza abbiamo $k!$ possibili permutazioni e poiché abbiamo n/k sottosequenze il numero di possibili permutazioni è $k!^{n/k}$. Quindi il numero di confronti è almeno

$$\log(k!^{\frac{n}{k}}) = \frac{n}{k} \cdot \log(k!) = \Omega(n \log k).$$

Esercizio 9.3-7

Descrivete un algoritmo con tempo $O(n)$ che, dato un insieme $S = \{s_1, \cdots, s_n\}$ di n interi distinti e un intero positivo $k \leq n$, trova i k numeri di S che sono più vicini alla mediana.

Soluzione. Usiamo il fatto che una qualsiasi statistica di ordine di un insieme di n elementi può essere determinata in tempo $O(n)$. L'algoritmo effettua i seguenti passi, ciascuno dei quali può essere eseguito in tempo $O(n)$.

(1) Determiniamo la mediana m di S.
(2) Costruiamo l'insieme D di record contenti due campi; l'i-esimo record di D contiene i campi s_i e $|s_i - m|$.
(3) Calcoliamo il k-esimo elemento x più piccolo di D rispetto al secondo campo.
(4) Costruiamo l'insieme M degli elementi D che sono minori o uguali a x (sempre rispetto al secondo campo).
(5) Diamo in output il primo campo di tutti i record di M.

Esercizio 9.3-8

Siano $X[1 \dots n]$ e $Y[1 \dots n]$ due array, ciascuno contenente n interi già ordinati. Descrivete un algoritmo che in tempo $O(\log n)$ trova la mediana dei $2n$ elementi degli array X e Y (cioè dell'insieme $X \cup Y$).

Soluzione. Assumiamo per semplicità che tutti gli elementi siano distinti e che vogliamo calcolare la mediana inferiore di $X \cup Y$.

Consideriamo due casi.

- n **dispari:** Siano m_X e m_Y le mediane degli array X e Y. Possiamo calcolare m_X e m_Y in tempo costante in quanto si trovano nella posizione $\lceil n/2 \rceil$ di X e Y, rispettivamente. Supponiamo che $m_X < m_Y$ (l'altro caso è simile). Allora nessuno degli elementi di $x \in X$ tali che $x < m_X$ può essere mediana di $X \cup Y$. Infatti contiamo il numero di elementi di $X \cup Y$ che sono maggiori di x. In X ne esistono almeno $\lceil n/2 \rceil$ (m_X e tutti gli elementi maggiori di m_X). Inoltre, poichè $m_X < m_Y$ tutti gli $\lceil n/2 \rceil$ elementi di Y che sono $\geq m_Y$ sono maggiori di m_X e quindi anche di x. In totale abbiamo quindi almeno $2\lceil n/2 \rceil = n + 1$ elementi maggiori di x. Un simile ragionamento prova che nessuno degli elementi di Y che sono maggiori di m_Y può essere una mediana di $X \cup Y$. Quindi la mediana deve essere ricorsivamente ricercata in $X[\lceil n/2 \rceil, \cdots, n]$ e $Y[1 \cdots \lceil n/2 \rceil]$.
- n **pari:** In questo caso ognuno dei due array ha due mediane. Consideriamo la mediana inferiore $m_X = X[n/2]$ di X e la mediana superiore $m_y = Y[n/2+1]$ di Y e supponiamo che $m_X < m_Y$ (gli altri casi sono simili).

  Sia $x \in X$ tale che $x < m_X$. Contando gli elementi di $X \cup Y$ che sono maggiori di x otteniamo $n/2 + 1$ elementi da X (la mediana m_X e gli $n/2$ elementi di X maggiori di m_X) e $n/2$ elementi da Y (la mediana m_Y e gli $n/2 - 1$ elementi di Y maggiori di m_y) per un totale di $n + 1$ elementi e quindi x non può essere mediana. Un simile ragionamento prova che nessuno degli elementi di Y maggiori m_Y può essere mediana. Quindi la mediana deve essere ricorsivamente ricercata in $X[n/2, \cdots, n]$ e $Y[1 \cdots n/2]$.

In entrambi i casi abbiamo dimezzato la taglia degli array in tempo costante e quindi l'algoritmo prende tempo $O(\log n)$.

Esercizio 13.1-5

Dimostrate che, in un albero rosso-nero, il percorso semplice più lungo che va da un nodo x ad una foglia discendente di x ha un'altezza al massimo doppia di quello del percorso semplice più breve dal nodo x a una foglia discendente.

Soluzione. Per le proprietà degli alberi rosso-neri tutti i cammini semplici da x ad una foglia discendente contengono lo stesso numero di nodi neri b. Poiché lungo un cammino semplice non possiamo avere due nodi rossi consecutivi, denotando con r il numero di nodi rossi in un cammino semplice da x ad una foglia discendente, abbiamo che $0 \leq r \leq b + 1$. Pertanto, per la lunghezza $\ell = r + b$ di un cammino da un nodo x ad una foglia discendente, abbiamo

$$b \leq \ell \leq 2b + 1.$$

Possiamo quindi concludere che il rapporto tra il più lungo e il più breve cammino semplice da x ad una foglia discendente è al più $2 + 1/b$.

Esercizio 13.2-4

Dimostrate che ogni albero di ricerca di n nodi può essere trasformato in un qualsiasi altro albero binario di ricerca con n nodi effettuando $O(n)$ rotazioni.

Soluzione. Dimostriamo prima di tutto che bastano al più $n-1$ rotazioni destre R per trasformare un qualsiasi albero T di n nodi in una catena destra. Definiamo il *dorso destro* di un albero come l'insieme dei nodi che si trova sul cammino dalla radice alla foglia più a destra ed osserviamo che il dorso contiene sempre almeno un nodo (la radice). Notiamo che se effettuiamo una rotazione a destra su un nodo del dorso destro che ha un figlio sinistro, il numero di nodi nel dorso destro aumenta di uno. Quindi dopo $n-1$ rotazioni il dorso contiene n nodi (e quindi l'albero è una catena destra).

Supponiamo di voler trasformare l'albero T_1 nell'albero T_2. Sappiamo che esiste una sequenza R_1 di rotazioni destre che trasforma T_1 nella catena destra e una sequenza R_2 di rotazione destre che trasforma T_2 nella catena destra. Sia L_2 la sequenza che si ottiene da T_2 sostiutuendo ogni rotazione destra con la corrispondente rotazione sinistra ed invertendo l'ordine delle rotazioni. Osserviamo che se applichiamo la sequenza L_2 alla catena destra otteniamo T_2. Quindi applicando R_1 a T_1 otteniamo la catena destra ed applicando L_2 alla catena destra otteniamo T_2. Siccome sia R_1 che L_2 contengono $O(n)$ rotazioni abbiamo mostrato una sequenza di $O(n)$ rotazioni che trasforma T_1 in T_2.

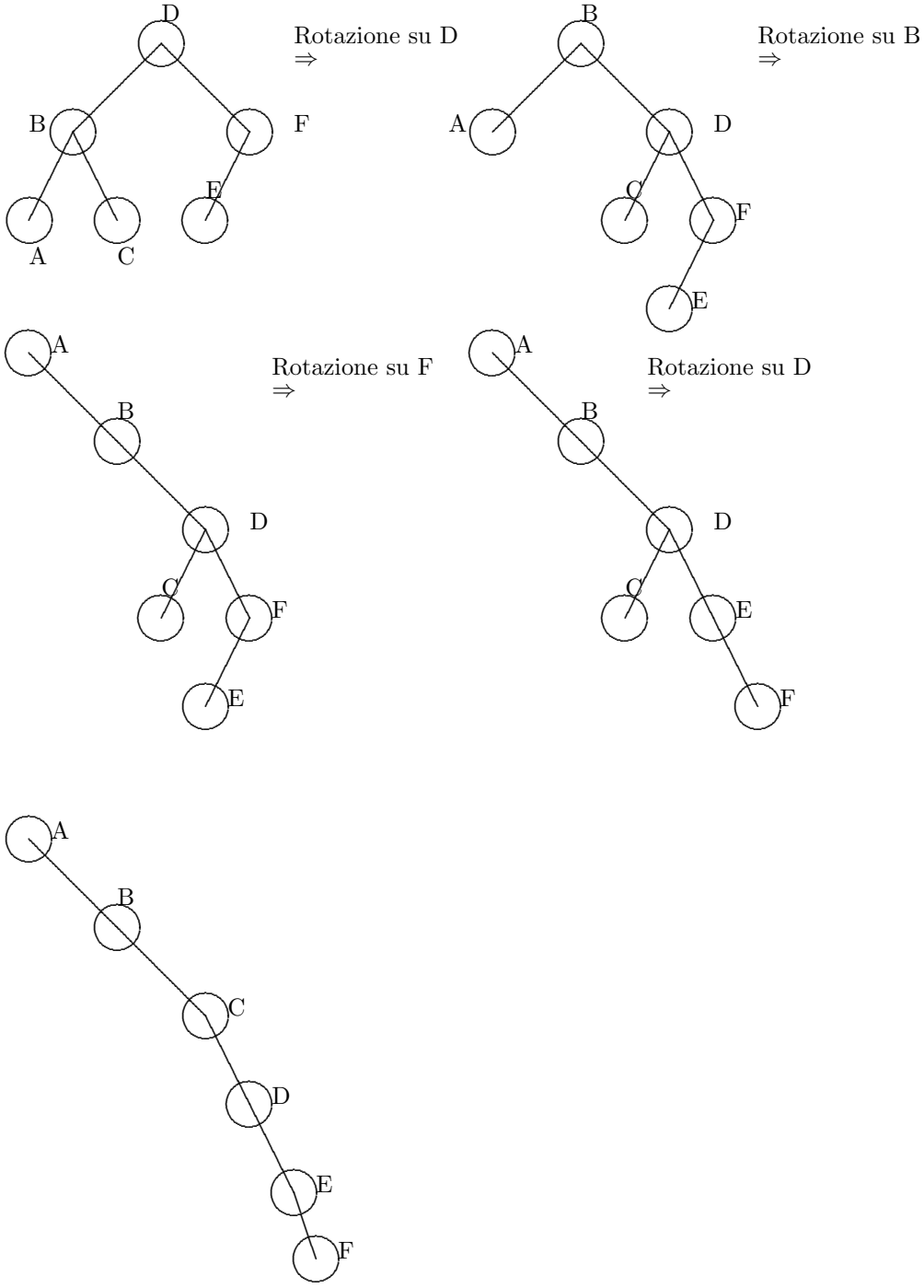

FIGURA 1. Un esempio di trasformazione di un albero in una catena destra mediante rotazioni a destra.

Esercizio 15.1-4

Le tabelle con i valori di $f_i[j]$ e $l_i[j]$ contengono, in totale, $4n-2$ elementi. Spiegate come ridurre lo spazio in memoria a un totale di $2n+2$ elementi, continuando a calcolare f^* e a essere in grado di elencare tutte le stazione del percorso più rapido che attraversa lo stabilimento.

Soluzione. Notiamo che ad ogni iterazione del ciclo *for* su j dell'algoritmo FASTEST-WAY, il calcolo di $f_1[j], f_2[j], l_1[j]$ e $l_2[j]$ dipende solo dai valori $f_1[j-1], f_2[j-1]$. Quindi possiamo evitare di memorizzare l'intera tabella $f_i[j]$ (che occupa spazio $2n$) e invece memorizzare solo i due valori calcolati all'iterazione precedente e i due valori calcolati all'iterazione corrente (occupando quindi spazio 4). La memoria richiesta è quindi $2n-2$ (per la tabella $l_i[j]$) + 4.

Esercizio 15-6

Mosse sulla scacchiera. *Supponete di avere una scacchiera di $n \times n$ caselle e una pedina che dovete muovere dall'estremità inferiore della scacchiera, rispettando le seguenti regole. In ogni mossa la pedina può andare in una delle seguenti caselle:*

(1) *la casella immediatamente in alto;*
(2) *la casella immediatamente in alto a sinistra (a meno che la pedina non si trovi già nell'ultima colonna a sinistra);*
(3) *la casella immediatamente in alto a destra (a meno che la pedina non si trovi già nell'ultima colonna a destra).*

Ogni volta che spostate la pedina dalla casella x alla casella y, riceverete $p(x, y)$ Euro. Riceverete tale importo per tutte le coppie (x, y) per le quali è valida la mossa da x a y. Non bisogna supporre che $p(x, y)$ sia positivo.

Descrivete un algoritmo che calcola l'insieme delle mosse che dovrà fare la pedina partendo da una casella dell'estremità inferiore della scacchiera fino a raggiungere una casella dell'estremità superiore, guadagnando il maggiore numero possibile di Euro. L'algortimo è libero di scegliere qualsiasi casella nell'estremità inferiore come punto di partenza e qualsiasi casella nell'estremità superiore come punto di arrivo. Qual è il tempo di esecuzione del vostro algortimo?

Soluzione.

Denotiamo con $M(r, c)$ il massimo pagamento ottenibile da un cammino che parte dalla casella (r, c) e che arriva ad una casella sull'estremità superiore della scacchiera. Ovviamente abbiamo che $M(n, c) = 0$, per ogni $1 \leq c \leq n$. Se invece $1 \leq r < n$ abbiamo

$$M(r,c) = \begin{cases} \max\{M(r+1,c) + p((r,c),(r+1,c)), \\ \qquad M(r+1,c+1) + p((r,c),(r+1,c+1))\}, & \text{se } c = 1; \\ \max\{M(r+1,c) + p((r,c),(r+1,c)), \\ \qquad M(r+1,c-1) + p((r,c),(r+1,c-1))\}, & \text{se } c = n; \\ \max\{M(r+1,c) + p((r,c),(r+1,c)), \\ \qquad M(r+1,c-1) + p((r,c),(r+1,c-1)), \\ \qquad M(r+1,c+1) + p((r,c),(r+1,c+1))\}, & \text{altrimenti.} \end{cases}$$

Progettiamo quindi un algoritmo che calcola la matrice $M(r, c)$ per valori decrescenti di r. Una volta calcolati i valore della prima riga, il valore richiesto è il massimo di $M(1, c)$ per $1 \leq c \leq n$. Per poter dare in output non solo il valore massimo ma anche il cammino che ottiene tale valore massimo, per ogni casella (r, c) conserviamo anche il primo passo del cammino che, tra quelli che partono da (r, c), ha il valore massimo. Più precisamente, per ogni casella (r, c) l'algoritmo calcola $S(r, c)$ che è uguale a $-1, 0$, oppure $+1$ a secondo se il cammino di valore massimo che parte dalla casella (r, c) si sposta nella colonna a sinistra (-1), resta nella stessa colonna (0) o si sposta nella colonna a destra $(+1)$ quando passa dalla riga r alla riga $r + 1$.

L'algoritmo discusso è descritto dal seguente pseudocodice.

Algoritmo PEDINA$(n, p(\cdot,\cdot))$

```
01. for c = 1 to n
02.    M(n, c) = 0;
03. end;
04. for r = n − 1 downto 1
05.    for c = 1 to n
06.       M(r, c) = M(r, c + 1) + p((r, c), (r + 1, c));
07.       S(r, c) = 0;
08.          if (c ≠ 1) and M(r + 1, c − 1) + p((r, c), (r + 1, c − 1)) > M(r, c) then
09.             M(r, c) = M(r + 1, c − 1) + p((r, c), (r + 1, c − 1));
10.             S(r, c) = −1;
11.          if (c ≠ n) and M(r + 1, c + 1) + p((r, c), (r + 1, c + 1)) > M(r, c) then
12.             M(r, c) = M(r + 1, c + 1) + p((r, c), (r + 1, c + 1));
13.             S(r, c) = +1;
14.    end;
15. end;
16. M = M(1, 1);
17. S = 1;
18. for c = 2 to n
19.    if (M < M(1, c)) then
20.       M = M(1, c);
21.       S = c;
22. end;
23. Print La sequenza di massimo profitto vale M;
24. Print e consiste delle seguenti mosse;
25. for r = 1 to n − 1
26.    Print((r, S), (r + 1, S + S(r, c));
27.    S = S + S(r, c);
```

Esercizio 16.1-2

Anziché selezionare sempre l'attività che finisce per prima, supponete di selezionare quella che inizia per ultima tra quelle compatibili con tutte le attività precedentemente selezionate. Dimostrate che questo algoritmo fornisce una soluzione ottima.

Soluzione. Consideriamo le n attività $A = (a_1, \cdots, a_n)$ ordinate in ordine decrescente per tempo di inizio. Proviamo che esiste sempre una soluzione ottima che consiste dell'attività a_1 e dalla soluzione ottima per l'istanza A' che si ottiene da A eliminando a_1 e tutte le attività in conflitto con a_1.

Supponiamo che esista una soluzione S per A che non contiene a_1 e sia a_j l'attività che inizia per ultima tra quelle di S. Allora la soluzione $S^\star = S \setminus \{a_j\} \cup \{a_1\}$ è ammissibile, contiene a_1 ed è ottima.

Sia ora S una soluzione ottima che contiene a_1 e supponiamo che $S - \{a_1\}$ non sia ottima per A'. Allora, se S' è una soluzione ottima per A', abbiamo che $S' \cup \{a_1\}$ è una soluzione per A di cardinalità maggiore di S.

Una soluzione alternativa al problema si ottiene facendo la seguente osservazione. Denotiamo con N il tempo massimo di fine di un'attività. Per un'istanza A costruiamo l'istanza $\bar{A}$ che, per ogni $a_i \in A$, contiene l'attività $\bar{a}_i$ definita come segue: se $a_i = [s_i, f_i)$ allora $\bar{a}_i = [N - f_i, N - s_i)$. È facile osservare che se S è una soluzione ottima per A allora $\bar{S}$ è una soluzione ottima per $\bar{A}$. Infine osserviamo che scegliere le attività in A in ordine crescente di tempo di fine corrisponde a scegliere le attività in ordine decrescente di tempo di inizio in $\bar{A}$. Poiché sappiamo che il primo algoritmo (ordine crescente per tempo di fine) restituisce una soluzione ottima per A, allora il secondo algoritmo (ordine decrescente per tempo di inizio) fornisce una soluzione ottima per $\bar{A}$.

Versione: 1.1 del 20 gennaio 2006.

Esercizio 16.1-4

Non sempre un approccio goloso (greedy) al problema della selezione delle attività genera un insieme massimo di attività mutuamente compatibili. Descrivete una istanza per cui l'algoritmo che sceglie l'attività di durata minima fra quelle che sono compatibili con le attività precedentemente selezionate non restituisce una soluzione ottima.

Fate la stessa cosa per l'algoritmo che sceglie sempre l'attività restante compatibile che si sovrappone al minor numero di attività restanti e per l'algoritmo che sceglie l'attività restante compatibile che il minimo tempo di inizio.

Soluzione.

Durata minima. Consideriamo l'istanza comprendenti gli intervalli $[1, 10)$, $[10, 20)$ e $[9, 11)$. La soluzione ottima comprende gli intervalli $[1, 10)$, $[10, 20)$ mentre l'algoritmo restituisce la soluzione comprendente solo l'intervallo $[9, 11)$.

Minima sovrapposizione. Consideriamo l'istanza comprendente gli intervalli

$$[0, 4), [4, 6), [6, 10), [0, 1), [1, 5), [5, 9), [9, 10), [0, 3), [0, 2), [7, 10), [8, 10).$$

L'intervallo con minima sovrapposizione è $[4, 6)$ che impedisce la scelta della soluzione ottima

$$[0, 1), [1, 5), [5, 9), [9, 10)$$

in quanto si interseca con due intervalli.

Minimo tempo di inizio. Consideriamo l'istanza comprendenti gli intervalli $[1, 10)$, $[3, 4)$ e $[4, 5)$. La soluzione ottima comprende gli intervalli $[3, 4)$, $[4, 5)$ mentre l'algoritmo restituisce la soluzione comprendente solo l'intervallo $[1, 10)$.

Esercizio 28.2-3

Supponiamo che sappiamo moltiplicare due matrici 3×3 effettuando l moltiplicazioni. Che condizione deve soddisfare l affinché sia possibile utilizzare questo risultato per moltiplicare delle matrici $n \times n$ in tempo n^k per $k < \log_2 7$?

Soluzione. Se sappiamo moltiplicare due matrici 3×3 usando l moltiplicazioni possiamo ricorsivamente moltiplicare due matrici $n \times n$ usando tempo $T(n) = lT(n/3) + \Theta(n^2)$. Usando il master theorem otteniamo che $T(n) = n^{\log_3 l}$. Pertanto l deve essere tale che $\log_3 l < \log_2 7$ e quindi $l < 3^{\log_2 7} = 3^{2.81} = 21.9134$.