Algoritmi e Strutture Dati

AA 2018/2019

Paolo Baldan
Dipartimento di Matematica
Università di Padova

22 giugno 2019

Segue una raccolta di esercizi svolti, suddivisi approssimativamente per aree tematiche. Gli esercizi sono spesso accompagnati da una bozza di soluzione, che contiene una indicazione su come procedere nella soluzione dell'esercizio. Spesso la soluzione è solo abbozzata ed è quindi da intendersi come una guida alla soluzione. In particolare, quando la domanda richieda definizioni o illustrazioni di procedimenti senza contributi originali, la soluzione non le include. Inoltre il documento è di recente redazione e può contenere sviste e piccoli errori. Segnalazioni in questo senso sono benvenute.

Si ringraziano per le segnalazioni (in ordine temporale): Bogdan Stanciu, Bianca Andreea Ciuche, Alberto Schiabel, Alberto Miola, Elisabetta Piombin, Jordan Gottardo, Sara Righetto, Ilaria Peron.

1 Limiti asintotici e ricorrenze

Domanda 1 Risolvere la ricorrenza $T(n) = 4\,T(n/2) + n \log n$ utilizzando il master theorem.

Soluzione: Rispetto allo schema generale si ha $a = 4$, $b = 2$, $f(n) = n \log n$. Si osserva che $\log_b a = 2$ quindi $f(n) = O(n^{\log_b a - \epsilon})$ (per $0 < \epsilon < 1$) e quindi $T(n) = \Theta(n^2)$.

Domanda 2 Mostrare che la ricorrenza $T(n) = T(n/2) + T(n/4) + n$ ammette soluzione $T(n) = \Theta(n)$ utilizzando il metodo di sostituzione.

Soluzione: Si prova separatamente che asintoticamente

1. $T(n) \leq cn$, per un'opportuna costante c
2. $T(n) \geq dn$, per un'opportuna costante d

Ad esempio, per la prima,

$$\begin{array}{rcl} T(n) & = & T(n/2) + T(n/4) + n \\ & \leq & n/2\,c + n/4\,c + n \\ & = & (1 + 3/4\,c)n \\ & \leq & cn \end{array}$$

quando $1 + 3/4\,c \leq c$, ovvero $c \geq 4$.

Domanda 3 Risolvere la ricorrenza $T(n) = 4T(n/2) + n^2\sqrt{n}$ utilizzando il master theorem.

Soluzione: Rispetto allo schema generale si ha $a = 4$, $b = 2$, $f(n) = n^2\sqrt{n} = n^{\frac{5}{2}}$. Si osserva che $\log_b a = 2$ quindi $f(n) = \Omega(n^{\log_b a+\epsilon})$ (per $0 < \epsilon \leq \frac{1}{2}$). In aggiunta vale la condizione di regolarità, ovvero $af(\frac{n}{b}) \leq cf(n)$ per qualche $c < 1$. Infatti $af(\frac{n}{b}) = 4f(\frac{n}{2}) = 4\frac{n^2}{4}\sqrt{\frac{n}{2}} = n^2\sqrt{\frac{n}{2}} \leq cn^2\sqrt{n}$ per $c \geq \frac{1}{\sqrt{2}}$ (che è < 1). Quindi $T(n) = \Theta(n^2\sqrt{n})$.

Domanda 4 Risolvere la ricorrenza $T(n) = T(n-2) + 2n$ utilizzando il metodo di sostituzione per dimostrare un limite asintotico stretto.

Soluzione: Mostriamo che $T(n) = \Theta(n^2)$ Si prova separatamente che asintoticamente

1. $T(n) \leq cn^2$, per un'opportuna costante c
2. $T(n) \geq dn^2$, per un'opportuna costante d

Ad esempio, per la prima,

$$\begin{array}{rcl} T(n) & = & T(n-2) + 2n \\ & \leq & c(n-2)^2 + 2n \\ & = & c(n^2 - 4n + 4) + 2n \\ & = & cn^2 - ((4c-2)n - 4c) \\ & \leq & cn^2 \end{array}$$

quando $(4c-2)n - 4c \geq 0$, che vale per $c > \frac{1}{2}$ e $n \geq \frac{4c}{4c-2}$, ad es. $c = 1$ e $n \geq 2$.

Per la seconda, risulterà, dualmente $(4d-2)n - 4d \leq 0$, che vale, ad esempio, per qualunque n e $d = \frac{1}{2}$.

Domanda 5 Risolvere la ricorrenza

$$T(n) = \begin{cases} 3 & \text{se } n = 0 \\ T(n-1) + 2 & \text{se } n > 0 \end{cases}$$

utilizzando il metodo di sostituzione per determinare una soluzione esatta (non asintotica).

Soluzione: Mostriamo per induzione che $T(n) = an + b$, determinando a e b. Per $n = 0$ si ottiene $T(n) = 3 = b$, quindi $b = 3$. Nel caso induttivo, si ottiene

$$T(n+1) = T(n) + 2 = an + b + 2$$

Affinché $an + b + 2 = a(n+1) + b$ occorre che $a = 2$. Quindi, $T(n) = 2n + 3$.

Domanda 6 Sia data la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 3T(n/5) + T(n/6) + n & \text{se } n > 1 \end{cases}$$

Si fornisca un limite asintotico stretto per la soluzione.

Soluzione: Vale $T(n) = \Theta(n)$. Dimostriamo ad esempio che

$$T(n) = \Omega(n)$$

ovvero che asintoticamente, per un'opportuna costante $c > 0$

$$T(n) \geq cn$$

Infatti

$$\begin{array}{ll} T(n) = & 3T(n/5) + T(n/6) + n \leq \quad \text{per ipotesi induttiva} \\ & 3cn/5 + cn/6 + n = \\ & c(23/30)n + n \end{array}$$

vorremmo fosse $\geq cn$, che vale se $(1 - 23/30)c \leq 1$, ovvero $c \leq 30/7$.

Domanda 7 Sia data la seguente equazione di ricorrenza:

$$T(n) = 5T(\lfloor n/3 \rfloor) + 2n^2$$

Si fornisca un limite asintotico stretto per la soluzione.

Soluzione: Vale $T(n) = \Theta(n^2)$, tramite il master theorem. Infatti, secondo lo schema generale $a = 5$, $b = 3$, quindi occorre confrontare $n^{\log_3 5}$ e $f(n) = 2n^2$. Facile vedere che, essendo $\log_3 5 < 2$, vale $f(n) = \Omega(n^{\log_3 5+\epsilon})$ per un opportuno $\epsilon > 0$. Inoltre vale la condizione di regolarità $af(n/b) \leq kf(n)$ per un opportuno $k < 1$, infatti

$$af(n/b) = 5f(n/3) = 5 * 2(n/3)^2 = 10/9n^2 \leq kf(n) = k2 * n^2$$

si riduce a $5/9 \leq k$, che vale, ad es. per $k = 5/9$.

Domanda 8 Sia data la seguente equazione di ricorrenza:

$$T(n) = T(n-1) + \log n$$

Si dimostri che $T(n) = O(n \log n)$.

Soluzione: Si procede una prova induttiva del fatto che $T(n) \leq cn \log n$.

$$\begin{array}{rcl} T(n) & = & T(n-1) + \log n \\ & \leq & c(n-1)\log(n-1) + \log n \\ & \leq & c(n-1)\log n + \log n \\ & = & (cn - c + 1)\log n \\ & \leq & cn \log n \end{array}$$

per $c \geq 1$.

Domanda 9 Sia data la seguente equazione di ricorrenza:

$$T(n) = T(n/2) + T(\sqrt{n}/2) + 2n$$

Si dimostri che $T(n) = \Theta(n)$.

Soluzione: Si procede una prova induttiva del fatto che $T(n) \leq cn$ e $T(n) \geq dn$, per opportune costanti $c, d > 0$, asintoticamente. Dimostriamo prima che $T(n) \leq cn$:

$$
\begin{aligned}
T(n) &= T(\frac{n}{2}) + T(\frac{\sqrt{n}}{2}) + 2n \\
&\leq c(\frac{n}{2}) + c(\frac{\sqrt{n}}{2}) + 2n && \text{per ipotesi induttiva} \\
&= c\frac{n+\sqrt{n}}{2} + 2n \\
&\leq c(n + \frac{n}{2})/2 + 2n && \text{dato che } \sqrt{n} \leq \frac{n}{2} \text{ per } n \geq 4 \\
&\leq c(3n/4) + 2n \\
&= n(3/4c + 2)
\end{aligned}
$$

per c tale che $3/4c + 2 \leq c$, ovvero $c \geq 8$.

Per la parte $T(n) \geq dn$, osserviamo che

$$
\begin{aligned}
T(n) &= T(\frac{n}{2}) + T(\frac{\sqrt{n}}{2}) + 2n \\
&\geq d(\frac{n}{2}) + d(\frac{\sqrt{n}}{2}) + 2n && \text{per ipotesi induttiva} \\
&= d\frac{n+\sqrt{n}}{2} + 2n \\
&\geq d\frac{n}{2} + 2n && \text{dato che } \sqrt{n} \geq 0 \\
&\geq dn
\end{aligned}
$$

per $d \leq 4$.

Domanda 10 Si consideri la ricorrenza

$$T(n) = T\left(\frac{4n}{5}\right) + \frac{n}{2} + \log n$$

Fornire limite asintotico stretto per la soluzione.

Soluzione: Si può utilizzare il master theorem dato che la ricorrenza è nella forma

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

con $a = 1$, $b = 5/4$ e $f(n) = n/2 + \log n$.

Vale che $n^{\log_b a} = n^{\log_{5/4} 1} = n^0 = 1$, quindi

$$f(n) = \Omega(n^{\log_b a + \epsilon}) = \Omega(n^\epsilon)$$

per $0 < \epsilon < 1$.

Inoltre vale la condizione di regolarità $a\,f(n/b) \leq kf(n)$ per qualche $0 < k < 1$. Infatti

$$
\begin{aligned}
a\,f(n/b) &= f(4n/5) \\
&= 2n/5 + \log 4n/5 \\
&= 2n/5 + \log n - \log(5/4) \\
&< k(n/2 + \log n)
\end{aligned}
$$

che risulta soddisfatta per $k > 4/5$ (quindi $k/2 > 2/5$) e n abbastanza grande. Quindi $T(n) = \Theta(n + \log n) = \Theta(n)$.

Domanda 11 Si dimostri che la ricorrenza che segue ha soluzione $T(n) = \Theta(n)$

$$T(n) = \frac{1}{2}\,T(n-1) + n$$

Soluzione: Si prova separatamente che asintoticamente

1. $T(n) \leq cn$, per un'opportuna costante c
2. $T(n) \leq dn$, per un'opportuna costante d

Ad esempio, per la prima,

$$\begin{array}{rcl} T(n) & = & \frac{1}{2}T(n-1) + n \\ & \leq & \frac{1}{2}c(n-1) + n \\ & = & (\frac{1}{2}c + 1)n - \frac{1}{2}c \\ & \leq & (\frac{1}{2}c + 1)n \\ & \leq & cn \end{array}$$

dove l'ultima disuguaglianza vale xquando $c \geq 2$.

Per la seconda, risulterà, dualmente

$$\begin{array}{rcl} T(n) & \geq & (\frac{1}{2}d + 1)n - \frac{1}{2}d \\ & \geq & dn \end{array}$$

vale, ad esempio, per qualunque $n \geq 1$ e $d \geq 1$.

Domanda 12 Dare la definizione di $\Omega(f(n))$. Dimostrare che la ricorrenza che segue ha soluzione $T(n) = \Omega(2^n)$

$$T(n) = \sum_{k=1}^{n-1} T(k)\,T(n-k)$$

Soluzione: Si deve provare che esiste una costante $c > 0$ e un n_0 tali che per ogni $n \geq n_0$

$$T(n) \geq c2^n$$

Si procede induttivamente:

$$\begin{array}{rcl} T(n) & = & \sum_{k=1}^{n-1} T(k)\,T(n-k) \\ & \geq & \sum_{k=1}^{n-1} c2^k c2^{n-k} \\ & = & \sum_{k=1}^{n-1} c^2 2^n \\ & = & c^2(n-1)2^n \\ & \geq & c2^n \end{array}$$

per $n \geq 2$, e qualunque $c \geq 1$.

Domanda 13 Dare la definizione di $O(f(n))$. Dimostrare che la ricorrenza che segue ha soluzione $T(n) = O(n)$

$$T(n) = \frac{2}{3}T(n-1) + 2n$$

Soluzione:

Si deve provare che esiste una costante $c > 0$ e un n_0 tali che per ogni $n \geq n_0$

$$T(n) \leq cn$$

Si procede induttivamente:

$$\begin{aligned} T(n) &= \tfrac{2}{3}T(n-1) + 2n \\ &\leq \tfrac{2}{3}c(n-1) + 2n \\ &\leq cn \end{aligned}$$

Affinché l'ultima disuguaglianza valga, occorre che

$$c(\frac{1}{3}n + \frac{2}{3}) \geq 2n$$

Ora

$$c(\frac{1}{3}n + \frac{2}{3}) \geq c\frac{2}{3}n \geq 2n$$

dove l'ultima disuguaglianza è certamente verificata per per ogni n, se $c \geq 3$.

Domanda 14 Dare una soluzione asintotica per la ricorrenza $T(n) = 3T(n/2) + n(n+1)$.

Soluzione: Si usa il master theorem con $a = 3$, $b = 2$, $f(n) = n(n+1)$. Si deve confrontare $f(n)$ con $n^{\log_b a}) = n^{\log_2 3}$ ed essendo che $\log_2 3 < 2$ per qualunque $0 < \epsilon < 2 - \log_2 3$ si vede facilmente che $f(n) = \Omega(n^{\log_2 3+\epsilon})$.

Per concludere che $T(n) = \Theta(f(n)) = \Theta(n^2)$ usando il caso 3 del master theorem occorre dimostrare la regolarità di $f(n)$, ovvero che $af(n/b) < kf(n)$ per $0 < k < 1$, asintoticamente. Si imposta

$$af(n/b) = 3n/2(n/2+1) < kn(n+1)$$

e si vede che $n/2 + 1 < 5n/8$ per $n > n_0 = 8$, quindi sotto questa condizione

$$af(n/b) = 3n/2(n/2+1) < 15/16n(n+1)$$

ovvero $k = 15/16$ va bene.

Domanda 15 Data la ricorrenza $T(n) = T(n/2) + T(n/3) + \sqrt{n} + 2$ dimostrare che ha soluzione $T(n) = O(n)$. Il limite è stretto, ovvero vale anche $T(n) = \Omega(n)$? [Questa seconda parte della domanda, per un errore nella scelta dei coefficienti risultava più complessa di quanto preventivato, quindi non è stata considerata nella valutazione. Allego comunque la soluzione per completezza.]

Soluzione: Per dimostrare che $T(n) = O(n)$ occorre trovare $c > 0$ e $n_0 \in \mathbb{N}$ tale che per ogni $n \geq n_0$, $T(n) \leq cn$. Procediamo con un ragionamento induttivo

$$\begin{aligned} T(n) &= T(n/2) + T(n/3) + \sqrt{n} + 2 \\ &\leq cn/2 + cn/3 + \sqrt{n} + 2 \\ &= cn/2 + cn/3 + \sqrt{n} + 2 \\ &= 5/6cn + \sqrt{n} + 2 \\ &= 5/6cn + n + 2 \\ &\leq cn \end{aligned}$$

vale se $1/6cn \geq n + 2$m, quindi, ad esempio, per $c \geq 7$ e $n \geq 12$.

Il limite non è stretto. Un tentativo di dimostrazione induttiva che $T(n) \geq dn$ per qualche $d > 0$ fallisce. Tuttavia questo non consente di concludere.

Si può invece dimostrare che $T(n) = O(n^\alpha)$ con $\alpha = 5/6$. Infatti, ancora in modo induttivo:

$$\begin{aligned} T(n) &= T(n/2) + T(n/3) + \sqrt{n} + 2 \\ &\leq cn/2 + cn/3 + \sqrt{n} + 2 \\ &= c(n/2)^\alpha + c(n/3)^\alpha + \sqrt{n} + 2 \quad [\text{dato che } \sqrt{n} + 2 \leq n^\alpha \text{ per } n \geq 5] \\ &\leq cn^\alpha((1/2)^\alpha + (1/3)^\alpha) + n^\alpha \\ &\leq cn^\alpha \end{aligned}$$

vale se $c(1-(1/2)^\alpha-(1/3)^\alpha)n^\alpha \geq n^\alpha$, quindi, osservando che la quantità $1-(1/2)^\alpha-(1/3)^\alpha > 0$, è sufficiente scegliere $c \geq 1/(1-(1/2)^\alpha-(1/3)^\alpha)$ e $n \geq 5$.

Questo prova che $T(n) = O(n^\alpha)$ e quindi non può essere $T(n) = \Omega(n)$.

Domanda 16 Data la ricorrenza $T(n) = 5T(n/3) + (n-2)^2$, trovare la soluzione asintotica.

Soluzione: Si usa il master theorem con $a = 5$, $b = 3$, $f(n) = (n-2)^2$. Si deve confrontare $f(n)$ con $n^{\log_b a}) = n^{\log_3 5}$ ed essendo che $\log_3 5 < 2$ per qualunque $0 < \epsilon < 2 - \log_2 3$ si vede facilmente che $f(n) = \Omega(n^{\log_2 3+\epsilon})$.

Per concludere che $T(n) = \Theta(f(n)) = \Theta(n^2)$ usando il caso 3 del master theorem occorre dimostrare la regolarità di $f(n)$, ovvero che $af(n/b) < kf(n)$ per $0 < k < 1$, asintoticamente. Si imposta

$$af(n/b) = 5(n/3-2)^2 < k(n-2)^2$$

e si osserva

$$5(n/3-2)^2 < 5((n-2)/3)^2 = 5/9(n-2)^2$$

per cui si può scegliere $k = 5/9 < 1$ e n qualunque.

Domanda 17 Dare la definizione di $\Omega(f(n))$. Mostrare che se $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$ allora $f(n) = \Omega(h(n))$.

Soluzione: Si ha che

$$\Omega(f(n)) = \{g(n) \mid \exists c > 0.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ 0 \leq cg(n) \leq f(n)\}.$$

Se $f(n) = \Omega(g(n)$ e $g(n) = \Omega(h(n))$ allora esistono $c_1, c_2 > 0$, $n_1, n_2 \in \mathbb{N}$ tali che per ogni $n \geq n_1$

$$0 \leq c_1 g(n) \leq f(n) \tag{1}$$

e per ogni $n \geq n_2$

$$0 \leq c_2 h(n) \leq g(n) \tag{2}$$

Ne consegue che per ogni $n \geq \max\{n_1, n_2\}$, moltiplicando (2) per c_1 si ha

$$0 \leq c_1 c_2 h(n) \leq c_1 g(n) \leq f(n)$$

ovvero, indicato con $n_0 = \max\{n_1, n_2\}$ e $c = c_1 c_2$, per ogni $n \geq n_0$,

$$0 \leq ch(n) \leq f(n)$$

ovvero $f(n) = \Omega(h(n))$.

Domanda 18 Dare la definizione di $\Theta(f(n))$. Mostrare che $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.

Soluzione: Per la prima parte

$$\Theta(f(n)) = \{g(n) \mid \exists n_0 \in \mathbb{N}.\ \exists c, c_2 > 0.\ \forall n \geq n_0.\ 0 \leq c_1 f(n) \leq g(n) \leq c_1 f(n)\}.$$

Per dimostrare che $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$, ricordiamo che

1. $O(f(n)) = \{g(n) \mid \exists n_0 \in \mathbb{N}.\ \exists c > 0.\ \forall n \geq n_0.\ 0 \leq g(n) \leq cf(n)\}$
2. $\Omega(f(n)) = \{g(n) \mid \exists n_0 \in \mathbb{N}.\ \exists c > 0.\ \forall n \geq n_0.\ 0 \leq cf(n) \leq g(n)\}$.

Proviamo separatamente le due inclusioni. Se $g(n) \in \Theta(n)$ allora chiaramente $g(n) \in \Omega(f(n))$, grazie all'esistenza di n_0 e c_1, e similmente $g(n) \in O(f(n))$, grazie all'esistenza di n_0 e c_2.

Per il contrario, se $g(n) \in \Omega(f(n))$ e $g(n) \in O(f(n))$, per la prima esistono $c_1 > 0$, $n_1 \in \mathbb{N}$ tali che per ogni $n \geq n_1$ vale

$$0 \leq c_1 f(n) \leq g(n)$$

e per la seconda esistono $c_2 > 0$, $n_2 \in \mathbb{N}$ tali che per ogni $n \geq n_2$ vale

$$0 \leq g(n) \leq c_2 f(n)$$

Quindi, detto $n_0 = \max(n_1, n_2)$ per ogni $n \geq n_0$,

$$0 \leq c_1 f(n) \leq g(n) \leq c_1 f(n)$$

ovvero $f(n) \in \Theta(f(n))$.

2 Divide et impera e Ricorsione

Esercizio 1 Dato un array di interi $A[1..n]$, chiamiamo *gap* un indice $i \in [1, n)$ tale che $A[i+1] - A[i] > 1$.

i. Mostrare per induzione su n che un array $A[1..n]$ tale che $A[n] - A[1] \geq n$ (quindi $n \geq 2$) contiene almeno un gap.

ii. Fornire lo pseudocodice di una procedura ricorsiva divide et impera *gap* che dato un array $A[1..n]$ tale che $A[n] - A[1] \geq n$ restituisce un gap in A.

iii. Valutare la complessità della funzione, utilizzando il master theorem.

Soluzione: La prova per induzione è la seguente:

- $(n = 2)$ Banale, il gap è in $i = 1$.
- $(n > 2)$ Sia $A[n] - A[1] \geq n$. Allora se $A[n] - A[n-1] > 1$, si ha che $i = n - 1$ è un gap e abbiamo concluso. Altrimenti, si osserva che $A[n] - A[1] = A[n] - A[n-1] + A[n-1] - A[1]$. Quindi $A[n-1] - A[1] = (A[n] - A[1]) - (A[n] - A[n-1]) \geq n-1$, quindi per ipotesi induttiva $A[1..n-1]$ contiene un gap.

La prova funziona anche se invece che spezzare l'array in $A[1..n-1]$ e $A[n-1..n]$, lo si spezza in $A[1..\lceil n/2 \rceil]$ e $A[\lceil n/2 \rceil..n]$. Da questo segue l'algoritmo:

```
gap(A,p,r)    // p < r (almeno due elementi) , A[r] - A[p] >= r-p+1
  if p = r-1
     return p
  else
     q = (p+r)/2
     if A[q] - A[p] > q-p+1
        gap(A,p,q)
     else
        // note che allora A[r] - A[q] > r-q+1
        gap(A,q,r)
```

Si ha che $T(n) = T(n/2) + 1$, quindi il master theorem, dato che $n^{\log_b a} = n^{\log_2 1} = n^0 = \Theta(1)$, ci dice che $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$.

Esercizio 2 Scrivere una procedura di tipo divide et impera *over* che dato un array di interi distinti $A[1..n]$, ordinato in modo crescente, e un intero x restituisce l'indice del più piccolo elemento in A strettamente maggiore di x. Se nessun elemento di A soddisfa la condizione, si restituisca $n + 1$. Valutare la complessità dell'algoritmo.

Soluzione: Si realizza una funzione ricorsiva `over(A,p,r,x)` che restituisce il minimo indice $i \in [p, r]$ tale che $A[i] > x$ se un tale indice esiste, altrimenti $r + 1$.

```
over(A,p,r,x)
    if r-p < 0
       return r+1
    else
       q = floor((r+p)/2)
       if A[q] <= x
          return over(A,q+1,r,x)
       else
          return over(A,p,q-1,x)
```

La correttezza può essere dimostrata per induzione sulla lunghezza n del sottoarray, ovvero $n = |r - p + 1|$. Quando $n = 0$ quindi $r > p$ l'intervallo $[p, r]$ è vuoto, e correttamente la funzione ritorna $r + 1$. Se invece $n > 0$, ovvero $r < p$, sia $q = \lfloor p + r/2 \rfloor$. Distinguiamo due casi:

- se `A[q]>x` allora, per ipotesi induttiva, la chiamata `over(A,p,q-1,x)` ritorna il minimo indice `i` in `[p,q-1]` tale che `A[i]>x`, se un tale indice esiste, e in questo caso è anche il minimo indice in `[p,r]` con questa proprietà. Se nessun elemento in `A[p,q-1]` è maggiore di `x` correttamente ritorna $q - 1 + 1 = q$ che `[p,q-1]` sarà il minimo indice in `[p,r]` di un elemento maggiore di x.
- se `A[q]<=x` allora la chiamata `over(A,q+1,r,x)` ritorna il minimo indice `i` in `[q+1,r]` tale che `A[i]>x`, se un tale indice esiste (e in questo caso sarà anche il minimo indice in `[p,r]`), e $r + 1$ altrimenti.

La complessità è data dalla ricorrenza $T(n) = T(n/2) + c$ e quindi $T(n) = \Theta(\log n)$ dal master theorem.

Esercizio 3 Realizzare una procedure di tipo divide et impera `Max(A,p,r)` per trovare il massimo nell'array `A[p,r]`. Si assuma che l'array non sia vuoto (`p` $\leq$ `r`). Scrivere lo pseudocodice e valutare la complessità con il master theorem.

Soluzione:

```
Max(A, p, r)
    if p = r
       return A[p]
    else // p < r
       q =  (p+r)/2
       m1 = Max(A,p,q)
       m2 = Max(A,q+1,r)
       if m1<m2
                 return m2
       else
          return m1
```

Esercizio 4 Sia `A[1..n]` un array di interi distinti ordinato in senso crescente. Dimostrare che dato un qualunque indice `i`, se `A[i] > i` allora `A[j]>j` per ogni `j>i` e analogamente se `A[i] < i` allora `A[j]<j` per ogni `j<i`.

Utilizzare l'osservazione per realizzare una funzione `Fix(A)` che dato l'array di interi `A[1..n]` ordinato senza ripetizioni restituisce un indice `i` tale che `A[i]=i`, se esiste, e `0` altrimenti. Valutarne la complessità.

Soluzione: Per la prima parte sia i un indice tale che $A[i] > i$, ovvero $A[i] \geq i + 1$. Si osserva che, se $i < n$ allora $A[i+1] > A[i] \geq i+1$ ovvero $A[i+1] > i+1$. Usando questo fatto, si conclude con un ragionamento induttivo (che per ogni $j \geq 1$ vale $A[i + j] \geq i + j$).

A questo punto, un algoritmo può essere il seguente

```
Fix(A)
   return Fix-rec(A,1,n)

Fix-rec(A,p,r)
```

```
   if p <= r
      q = p+r/2
      if A[q] = q
         return q
      else if A[q] < q
         return Fix(A,q+1,r)
      else
         return Fix(A,p,q-1)
   else
      return 0
```

Il costo è dato dalla ricorrenza $T(n) = T(n/2) + c$ che ha soluzione $T(n) = \Theta(\log n)$.

Domanda 19 Scrivere una funzione ricorsiva `subseq(X,Y,m,n)` che date due sequenze $X[1..m]$ e $Y[1..n]$, di lunghezza m e n rispettivamente, verifica se X è una sottosequenza di Y e restituisce un valore booleano conseguente. Valutarne la complessità.

Soluzione: La soluzione è:

```
# subseq: dati due array X[1..m] e Y[1..n] determina se X e' una sotosequenza
# di Y

subseq(X,Y,m,n):
    if m == 0
        return True
    elif m <= n
        if X[m] == Y[n]:
            return subseq(X,Y,m-1,n-1)
        else:
            return subseq(X,Y,m,n-1)
    else
        return False
```

Per quanto riguarda la complessità si ha che ogni chiamata, esclusa la parte ricorsiva, ha costo costante e nel caso peggiore scorro tutta la seconda stringa, quindi ci sono n chiamate. Quindi il costo è $O(n)$.

Esercizio 5 Un array $A[1..n]$ di numeri si dice *alternante* se non ha elementi contigui identici (ovvero per ogni $i \leq n-1$ vale $A[i] \neq A[i+1]$) e inoltre per ogni $i \leq n-2$, vale che $a_i < a_{i+1} > a_{i+2}$ oppure $a_i > a_{i+1} < a_{i+2}$. Ad esempio gli array [1, 2, -1, 3, 2] e [5, 1, 2, -1, 3, 2] sono alternanti, mentre non lo sono [1, 2, 3] e [1, 1, 2]. Scrivere una funzione ricorsiva `alt(A,n)` che dato un array $A[1..n]$ di numeri verifica se è alternante. Valutarne la complessità.

Soluzione:

```
# alt: dato un array A[1..n] verifica se  e' alternante
alt(A,n)
    return altRec(A,n,0) or altRec(A,n,1)

altRec(A,i,dir)
    # verifica se A[1..i] e' alternante.
    # usa un ulteriore parametro per indicare se la sequenza alternante deve
```

```
    # concludersi crescendo (0) o decrescendo (1).

    if i==1
        return True
    else
        if dir==0
            return  altRec(A,i-1,1) and (A[i-1] < A[i])
        else
            return  altRec(A,i-1,0) and (A[i-1] > A[i])
```

Dato che la parte non ricorsiva è di costo costante, la complessità si può esprimere come $T(n) = 1 + T(n-1)$ che porta a $T(n) = \Theta(n)$.

Una soluzione più compatta:

```
# alt: dato un array A[1..n] verifica se  e' alternante
alt(A,n)
    return altRec(A,n,true) or altRec(A,n,false)

altRec(A,i,dir)
    # verifica se A[1..i] e' alternante.
    # usa un ulteriore parametro per indicare se la sequenza alternante deve
    # concludersi crescendo (true) o decrescendo (false).

    if i==1
        return True
    else
        return  altRec(A,i-1,not dir)
                and (A[i-1] <> A[i])
                and (dir xnor (A[i-1] > A[i])) # usa il fatto che
                                               #  true  xnor V = not V
                                               #  false xnor V = V
```

3 Ordinamento

Esercizio 6 Fornire lo pseudocodice di una procedura `min(A,B)` che dati due array `A` e `B` che siano uno la permutazione dell'altro ne trova l'elemento minimo confrontando esclusivamente elementi di `A` ed elementi di `B` (non si possono confrontare due elementi di `A` o due elementi di `B` tra loro, e non si possono fare copie degli array, ovvero la funzione deve operare con spazio costante). Valutare la complessità della funzione.

Soluzione: L'idea è quella di confrontare gli elementi di A e B, scorrendoli con due indici i, j avanzando in B quando $A[i] \leq B[j]$ e in B altrimenti. Sul lato di A l'avanzamento si fermerà ad un elemento $A[i]$ che sarà minore o uguale di tutti gli elementi di B. Il vero e proprio invariante è un po' complicato.

```
min(A,B)
  i=j=1
  // invariante:   (forall h \in [0,i) exists k in [0,j] A[h] > B[k]) or
                   (i <= n) and (forall k \in [0,j) exists h in [0,i] A[h] <= B[k])
  while (j <= n)
     if A[i] <= B[j]
        j = j+1
     else
        i = i+1
  return A[i]
```

Si noti che l'invariante implica implica $i \leq n$, quindi questo controllo nel ciclo non è necessario.

Il numero di iterazioni è al massimo $2n$ dato che uno dei due indici (i oppure j) viene sempre incrementato. Dunque la complessità è $\Theta(n)$.

Domanda 20 Realizzare una fDunzione `RevCountingSort(A,B,n,k)` che, dato un array `A[1..n]` contenente interi nell'intervallo `[0..k]`, restituisce in `B[1..n]` una sua permutazione ordinata in modo decrescente utilizzando una variante del counting sort. Valutarne la complessità.

Soluzione:

```
RevCountingSort (A, B, n, k)
  C[1..k] = 0
  for i = 1 to n
      C[A[i]] = C[A[i]]+1

   for j = k-1 downto 0 do
      C[j] = C[j] + C[j+1]

   for j = 1 to n
      B[C[A[i]]] = A[i]
      C[A[i]] = C[A[i]] - 1
```

Esercizio 7 Realizzare una funzione `Anagramma(x,y)` che date due stringhe `x` e `y` sull'alfabeto { `0, 1` }, verifica se `x` è un anagramma di `y` restituendo conseguentemente `true` o `false`. Una stringa è vista come un array di caratteri, con lunghezza data dall'attributo `len`. Ad esempio, la stringa `x` è la sequenza di caratteri `x[1] x[2] ... x[x.len]`. Valutare la complessità.

Soluzione:

i. L'implementazione è la seguente:

```
Anagramma(x,y)
     diff[0..1] = 0    # diff[v] = difference between number of v's in x and
                            #            number of v's in y, up to now

     for i=1 to x.len
              diff[x[i]]++
         diff[y[i]]--

     if (diff[0] == 0) and (x.len = y.len)
     # adding (diff[1] == 0) is useless as implied by the first two conditiosn
       return true
     else
        return false

# nota: il controllo sulla lunghezza poteva essere anticipato
```

ii. Complessità chiaramente $\Theta(n)$.

Domanda 21 Realizzare una funzione `Double(A,n)` che, dato un array `A[1,n]` ordinato in senso crescente, verifica se esiste una coppia di indici `i, j` tali che $\mathtt{A}[\mathtt{j}] = 2 * \mathtt{A}[\mathtt{i}]$. Restituisce la coppia se esiste e (0,0) altrimenti. Scrivere lo pseudocodice e valutare la complessità.

Soluzione:
Il codice può essere:

```
double(A,n) {                  // assume A[1,n] ordinato
   n = A.length                // ritorna una coppia i, j
                               // tale che 2*A[i] = A[j]
   i=1
   j=1
   while (i<=n) and (j<=n) and (2*A[i] <> A[j])
      if (2*A[i] > A[j])
         j++
      else
         i++

   if (i<=n) and (j<=n)
      return (i,j)
   else
      return (0,0)
```

È facile vedere che si mantiene l'invariante $\forall i' \in [0, i).\ \forall j' \in [0, j).\ 2 * A[i] \neq A[j]$. Da questo la correttezza segue immediatamente. Il costo è lineare (il numero di iterazioni è pari ad al più $2n$, ed ogni iterazione ha costo costante, quindi $T(n) = O(n)$.

Esercizio 8 Si dica che un array `A[1..n]` è bi-ordinato se esiste un indice `k` tale che `A[1..k]` è ordinato in senso crescente e `A[k..n]` è ordinato in senso decrescente. Realizzare un algoritmo

`top(A,n)` che dato in input un array `A` bi-ordinato con elementi distinti ne determina il valore massimo. Ad esempio su `A = (1 2 3 14 13 4)` restituisce `14`. Valutarne la complessità.

Soluzione:

```
top(A,n)
   top-rec(A,i,j)

top_rec(A,i,j)
   if (i=j)
      return A[i]
   else
      q = (i+j)/2
      if A[q] <= A[q+1]
         return top_rec(q+1,j)
      else
         return top_rec(i,q)
```

Esercizio 9 Scrivere una funzione `perm(m,n)` che dati due numeri interi m e n, maggiori o uguali di 0, verifica se uno dei due numeri può essere ottenuto permutando le cifre dell'altro. Ad esempio 915 e 159 sono uno la permutazione dell'altro mentre 911 e 19 no. Attenzione al ruolo degli zeri, ad es. 150 è la permutazione di 51, dato che $51 = 051$. Valutarne la complessità.
(*Suggerimento*: Il counting sort può fornire una ispirazione.)

Soluzione:

```
# perm: dati due interi positivi m e n verifica se uno e' la
# permutazione dell'altro.

perm(m,n):
    # use an array for counting the difference in the number of digits
    # of the two numbers. Note that we do not need to count the 0's as
    # different number of 0's can be compensated by putting 0's in front

    # D[i] = difference count for cypher i
    for i=1 to 9
        D[i] = 0

    while (m > 0)
        # compute the last digit of m
        d = m mod 10
        if (d > 0)
            D[d]++
        m = m div 10

    while (n > 0)
        # compute the last digit of m
        d = n mod 10
        if (d > 0)
            D[d]--
        n=n div 10
```

```
# returns True if D contains all zeros
i = 1
while (i<=9) and (D[i]==0)
   i++

return (i>9)
```

I primi due cicli while hanno complessità $\Theta(\log m)$ e $\Theta(\log n)$, mentre l'ultimo è $\Theta(9) = \Theta(1)$. Quindi la complessità è $\Theta(\log m + \log n) = \Theta(\log \max\{m, n\})$.

4 Heap

Domanda 22 Scrivere una funzione $sndmin(A)$ che dato in input un array A organizzato a min-heap, restituisce il successore della radice, ovvero il minimo elemento dello heap maggiore della radice. Se un tale elemento non esiste genera un errore. Assumere che A sia non vuoto e gli elementi in A siano tutti distinti.

Soluzione: È sufficiente ricordare che ogni nodo di un min-heap è minore o uguale ai discendenti per dedurre che il successore della radice, se esiste, è uno dei due figli $A[2]$ o $A[3]$.

```
sndmin(A)
  if A.heapsize > 2
       return min (A[2], A[3])
     else if A.heapsize = 2
       return A[2]
   else
     return error
```

Domanda 23 Scrivere una funzione `IsMaxHeap(A)` che dato in input un array di interi `A[1..n]` che verifica se `A` è organizzato a max-heap e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione: Versione ricorsiva

```
IsMaxHeap(A,i,j) // verifica se l'array A[i,j] e' un max-heap
                    (o meglio, se contiene un max-heap radicato in i,
                     dato che al passo ricorsivo non tutto [i,j]
                     conterra' il sottoalbero scelto
  l = 2*i
  r = 2*i+1
  if l<j
    leftOk = A[i] >= A[l] and IsMaxHeap(l, j)
  else
    leftOk = true

  if r<j
    rightOk = A[i] >= A[r] and IsMaxHeap(r, j)
  else
    rightOk = true

  return leftOk and rightOk
```

Per quanto riguarda la complessità si osservi che $T(n) = c+2T(n/2)$ per un'opportuna costante c e quindi, utilizzando il master theorem, si deduce che la complessità è $O(n)$.

Versione iterativa

```
IsMaxHeap(A,n) // verifica se l'array A[1..n] e' un max-heap
  i=2
  while (i <= n) and (A[i] <= A[i/2])
      i++
  return (i==n+1)
```

Domanda 24 Fornire lo pseudocodice della procedura `HeapExtractMin(A)` per estrarre il minimo elemento da un min-heap A, realizzato tramite un array come visto a lezione (la dimensione corrente dello heap è data dall'attributo $A.heapsize$). Discuterne la complessità.

Soluzione:

```
Heap-Extract-Min(A)    // A e' un min-heap
     if A.heapsize < 1
        error \underflow"
     else
        min = A[1]
        A[1] = A[A.heapsize]
        A.heapsize = A.heapsize - 1
        Min-Heapify(A,1)
        return min

MinHeapify(A,i)
  l = 2i, r =2i+1
  min = i
  if l <= A.heapsize and A[l] < A[m]
     min = l
  if r <= A.heapsize and A[r] < A[m]
     min = r
  if min <> i
     A[i] <-> A[m]
     MinHeapify(A,m)
```

Domanda 25 Dare la definizione di max-heap. Dato un array `A[1..12]` con sequenza di elementi `[60, 6, 45, 95, 30, 24, 15, 80, 19, 38, 21, 70]` si indichi il risultato della procedura `BuildMaxHeap` applicata ad `A`. Si descriva sinteticamente come si procede per arrivare al risultato.

Soluzione: L'array risultante (si vedano le procedure sul libro) è:

`[95, 80, 70, 60, 38, 45, 15, 6, 19, 30, 21, 24]`

5 Alberi e ricorsione

Domanda 26 Dato un albero nel quale i nodi contengono una chiave, si definisca *costo* di un cammino dalla radice ad una foglia, come la somma delle chiavi dei nodi che compaiono nel cammino. Scrivere una funzione `MaxPath(T)` che opera nel modo seguente. Prende in input un albero binario T, con radice $T.root$, e nodi x che hanno come campi $x.k$, $x.l$ e $x.r$, ovvero una chiave, il puntatore al figlio sinistro e destro, rispettivamente. Resituisce la il costo del cammino di costo massimo dalla radice ad una foglia. Valutarne la complessità.

Soluzione:

```
MaxPath(x)
   if x=nil
       return 0
   elseif x.l = nil
       return x.key + MaxPath(x.r)
   elseif x.r = nil
       return x.key + MaxPath(x.l)
   else
       return x.key + max { MaxPath(x.l), MaxPath(x.r) }
```

Domanda 27 Realizzare una procedura `Level(T)` che dato un albero binario `T`, con radice `T.root`, e nodi `x` con campi `x.left`, `x.right` e `x.key`, rispettivamente figlio destro, figlio sinistro e chiave intera, ritorna il numero di nodi per i quali la chiave `x.key` è minore o uguale al livello del nodo (la radice ha livello 0, i suoi figli livello 1 e così via). Valutare la complessità.

Soluzione:

```
// ritorna il numero di nodi y del sottoalbero radicato in x, tali che
//      y.key <= livello

Level(x, level)
   if x == nil
      return 0
   else
      left = Level(x.left, level+1)
      right = Level(x.right, level+1)
      if x.key <= level
               return left + right + 1
      else
         return left + right



// chiamata di base
Level(T)
        return Level (T.root, 0)
```

Si tratta di una visita dell'albero, quindi la complessità è lineare $\Theta(n)$.

Domanda 28 Sia T un albero binario i cui nodi x hanno i campi *x.left*, *x.right*, *x.key*. L'albero si dice un *sum-heap* se per ogni nodo x, la chiave di x è maggiore o uguale sia alla somma delle chiavi nel sottoalbero sinistro che alla somma delle chiavi nel sottoalbero destro.

Scrivere una funzione `IsSumHeap(T)` che dato in input un albero T verifica se `T` è un sum-heap e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione: La soluzione può essere

```
IsSumHeap(T)
  return IsSumHeap-rec(T.root)

IsSumHeap-rec(x) // verifica se l'albero radicato in x e' un sum-heap
                    e ritorna true/false e la somma delle chiavi nel
                    sottoalbero radicato in x
  if x = nil
     return true, 0
  else
     isSumHeapL, sumL = IsSumHeap-rec(x.left)
     isSumHeapR, sumR = IsSumHeap-rec(x.right)
     return (x.key >= sumL) and (x.key >= sumR),
            x.key + sumL + sumR
```

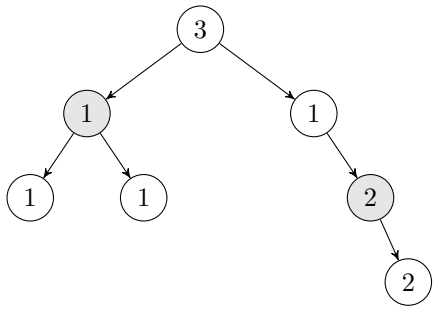
Per quanto riguarda la complessità, se l'albero è bilanciato, $T(n) = c + 2T(n/2)$ per un'opportuna costante c e quindi, utilizzando il master theorem, si deduce che la complessità è $\Theta(n)$. Se l'albero non è bilanciato, si può osservare che si tratta di una visita (costo lineare) o più precisamente scrivere la ricorrenza

$$T(n) = T(k) + T(n - k - 1) + c$$

e provare, per sostituzione, che $T(n) = an + b$ è soluzione per opportune costanti a, b.

Esercizio 10 Un nodo x di un albero binario T si dice *fair* se la somma delle chiavi nel cammino che conduce dalla radice dell'albero al nodo x (escluso) coincide con la somma delle chiavi nel sottoalbero di radice x (con x incluso). Realizzare un algoritmo ricorsivo *printFair(T)* che dato un albero T stampa tutti i suoi nodi fair. Supporre che ogni nodo abbia i campi *x.left*, *x.right*, *x.p*, *x.key*. Valutare la complessità dell'algoritmo.

Un esempio: i nodi grigi sono fair



Soluzione: L'algoritmo può essere il seguente:

```
printFair(x,path)       // x = node of the tree
                        // path=sum of the keys in the path from the root to x
```

```
                        // Action: print the fair nodes and
                       //         returns the sum of the keys in the subtree
    if (x == nil)
       return 0

    left   = printFair(x.l, path + x.key)
    right  = printFair(x.r, path + x.key)
    sumTree = left + right + x.key
    if (path == sumTree)
       print x
    return sumTree
```

e viene chiamato come `printFair(T.root, 0)`.

Si tratta di una visita, quindi con costo $O(n)$ (più precisamente ottenibile con il master theorem come soluzione della ricorrenza $T(n) = 2T(n/2) + c$.

Esercizio 11 Sia dato un albero i cui nodi contengono una chiave intera $x.key$, oltre ai campi $x.l$, $x.r$ e $x.p$ che rappresentano rispettivamente il figlio sinistro, il figlio destro e il padre. Si definisce *grado di squilibrio* di un nodo il valore assoluto della differenza tra la somma delle chiavi nei nodi foglia del sottoalbero sinistro e la somma delle chiavi dei nodi foglia del sottoalbero destro. Il grado di squilibrio di un albero è il massimo grado di squilibrio dei suoi nodi.

Fornire lo pseudocodice di una funzione `sdegree(T)` che calcola il grado di squilibrio dell'albero T (si possono utilizzare funzioni ricorsive di supporto). Valutare la complessità della funzione.

Soluzione:

```
// computes the sum of the leaf nodes of the subtree and the sdegree
// for the node x (returns two values)

sdegree(x)

  if (x == nil)
     sum = 0
     degree = 0
  elif (x.left == nil) and (x.right = nil)      # leaf
     sum = x.key
     degree = 0
  else
     suml, degreel = sdegree(x.l)
     sumr, degreer = sdegree(x.r)
     sum = suml + sumr
     degree = max { degreel, degreel, abs(suml - sumr) }

  return sum, degree
```

Esercizio 12 Si consideri un albero binario T, i cui nodi x hanno i campi $x.l$, $x.r$, $x.p$ che rappresentano il figlio sinistro, il figlio destro e il padre, rispettivamente. Un *cammino* è una

sequenza di nodi $x_0, x_1, \ldots, x_n$ tale che per ogni $i = 1, \ldots, n$ vale $x_{i+1}.p = x_i$. Il cammino è detto *terminabile* se $x_n.l = nil$ oppure $x_n.r = nil$. Diciamo che l'albero è 1-bilanciato se tutti i cammini terminabili dalla radice hanno lunghezze che differiscono al più di 1. Scrivere una funzione `bal1`(T) che dato in input l'albero T verifica se è 1-bilanciato e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione: La soluzione può basarsi su di una funzione ricorsiva che calcola per un nodo x la lunghezza massima e minima dei cammini terminabili da x

```
bal1(T)   // verifica se l'albero radicato in x e' 1-bilanciato,
          // ovvero i cammini dalla radice terminati da nil hanno
          // lunghezze che differiscono al piu di 1

          // si basa su di una funzione ricorsica che calcola la
          // lunghezza minima e massima dei cammini.
 min,max = bal1rec(T.root)

 return max-min <= 1

bal1rec(x)
 if x = nil
    return (0,0)
 else
    (left_min,left_max) = bal1(x.left)
    (right_min,r_right_max) = bal1(x.right)
    return max(left_min, right_min) + 1, max(left_max,right_max) + 1
```

Per quanto riguarda la complessità si osservi che $T(n) = c+T(k)+T(n-k-1)$ per un'opportuna costante c e quindi, la complessità è $O(n)$.

Esercizio 13 Sia T un albero binario i cui nodi x hanno i campi *x.left*, *x.right*, *x.key*. L'albero si dice *k-bounded*, per un certo valore k, se per ogni nodo x la somma delle chiavi lungo ciascun cammino da x ad una foglia è minore o uguale a k.

Scrivere una funzione `Bound(T,k)` che dato in input un albero T e un valore k verifica se `T` è k-bounded e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione:

```
k-bound(T,k)
  return k-bound-rec(T.root)

k-bound-rec(x,k)   // Verifica se l'albero radicato in x e' k-bounded
                    Ritorna true/false e la somma delle chiavi in un
                    cammino da x alla radice con somma massima
  if x = nil
     return true, 0
  else
     if x.left = nil
        isKBound, max = k-bound-rec(x.right,k)
        max  = max + x.key
     else if x.right = nil
        isKBound, max = k-bound-rec(x.left,k)
        max  = max + x.key
```

```
     else
        isKBoundL, maxL = k-bound-rec(x.left,k)
        isKBoundR, maxR = k-bound-rec(x.right,k)
        max  = max { maxR, maxL } + x.key
        isKBound =  isKBoundL and isKBoundR and max(x.key >= sumL) and max <= k

    return isKBound, max
```

La complessità è $\Theta(n)$.

Domanda 29 Scrivere una funzione *complete*(T) che dato in input un albero binario verifica se è completo (ovvero ogni nodo interno ha due figli e tutte le foglie hanno la stessa distanza dalla radice).

Soluzione:

```
complete(x) // verifica se l'albero radicato in x e' completo: ritorna
            // l'altezza dell'albero in caso positivo e -1 in caso negativo
 if x = nil
    return 0
 else
    countl = complete(x.left)
    countr = complete(x.right)

    if (countr == -1) or (countl == -1) or (countl <> countr)
       return -1
    else
       return countl+1
```

Per quanto riguarda la complessità si osservi che $T(n) = c+T(k)+T(n-k-1)$ per un'opportuna costante c e quindi, la complessità è $O(n)$.

Domanda 30 Scrivere una funzione `diff(T)` che dato in input un albero binario `T` determina la massima differenza di lunghezza tra due cammini che vanno dalla radice ad un sottoalbero vuoto. Ad esempio sull'albero ottenuto inserendo 1, 2 e 3 produce 2, su quello ottenuto inserendo 2, 1, 3 produce 0. Valutarne la complessità.

Soluzione: Il programma si basa su una una funzione ricorsiva `diff-rec(x)` che restituisce il la lunghezza del minimo e massimo cammino per il sottoalbero radicato in x

```
diff(T)
   min,max = diffRec(T.root)
   return

diff-rec(x)
   if x = nil
      return 0,0
   else
      minl, maxl = diff-rec(x.left)
      minr, maxr = diff-rec(x.right)
      return min(minl,minr)+1, max(maxl,maxr)+1
```

Per quanto riguarda la complessità si osservi che si tratta di una visita dell'albero, quindi $\Theta(n)$.

6 Tabelle Hash

Domanda 31 Si consideri una tabella hash di dimensione $m = 8$, e indirizzamento aperto con doppio hash basato sulle funzioni $h_1(k) = k \ mod \ m$ e $h_2(k) = 1 + k \ mod \ (m - 2)$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 12, 3, 22, 14, 38.

Soluzione: Si ottiene

0	-
1	14
2	-
3	3
4	12
5	-
6	22
7	38

Domanda 32 Si consideri una tabella hash di dimensione $m = 8$, e indirizzamento aperto con doppio hash basato sulle funzioni $h_1(k) = k \ mod \ m$ e $h_2(k) = 1 + 2 * (k \ mod \ (m - 3))$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 34, 12, 18, 9, 42.

Soluzione: Si ottiene

0	-
1	18
2	34
3	9
4	12
5	-
6	-
7	42

Domanda 33 Si consideri una tabella hash di dimensione $m = 8$, e indirizzamento aperto con doppio hash basato sulle funzioni $h_1(k) = k \ mod \ m$ e $h_2(k) = 1 + k \ mod \ (m - 2)$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 14, 22, 10, 16, 8.

Soluzione: Si ottiene

0	16
1	8
2	10
3	22
4	-
5	-
6	14
7	-

Domanda 34 Si consideri una tabella hash di dimensione $m = 8$, gestita mediante chaining (liste di trabocco) con funzione di hash $h(k) = k \ mod \ m$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 14, 10, 22, 18, 19.

Soluzione: Si ottiene

0		
1		
2		18 → 10
3		19
4		
5		
6		22 → 14
7		

Domanda 35 Si consideri una tabella hash di dimensione $m = 9$, e indirizzamento aperto con doppio hash basato sulle funzioni $h_1(k) = k \ mod \ m$ e $h_2(k) = 1 + k \ mod \ (m - 2)$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 12, 3, 22, 14, 38.

Soluzione: Si ottiene

0	-
1	-
2	38
3	12
4	22
5	14
6	-
7	3
8	-

7 Alberi Binari di Ricerca e Red Black Trees

Domanda 36 Realizzare una funzione $pred(x)$ che dato in input un nodo x, di un albero binario di ricerca T, restituisce il predecessore di x (oppure nil, se il predecessore non esiste). Come a lezione, supporre che ogni nodo abbia i campi *x.left*, *x.right*, *x.p*, *x.key*.

Soluzione:

```
pred(x)
    if x.left <> nil
        return max (x.left)
      else
        y = x.p
        while (y <> nil) and (x == y.left)
           x=y
           y=y.p
        return y

max (x)
    while x.right <> nil
          x = x.right
    return x
```

Domanda 37 Scrivere una funzione `IsABR(A)` che dato in input un array di interi `A[1..n]`, interpretato come albero binario (come nel caso degli heap, ogni `A[i]` è un nodo con figlio sinistro e destro `A[2i]` e `A[2i+1]`) verifica se `A` è un albero binario di ricerca. Valutarne la complessità.

Soluzione: Una versione iterativa, di tempo $O(n \log n)$ consiste nel verificare che ogni nodo sia minore o uguale dei nodi dei quali è nel sottoalbero sinistro e maggiore o uguale di quelli in cui è nel sottoalbero destro

```
IsABR(A)

  n =  A.length
  i = n
  isABR = true
  while ((i > 1) and isABR)
    j = i
    while j>1                      # risale l'albero
         if even(j)                # i nodi di indice pari sono figli sx
           isABR = isABR and A[i] <= A[j/2]
         else                      # quelli di indice dispari figli dx
           isABR = isABR and A[i] >= A[j/2]
         j = j/2
    i = i-1
  return isABR
```

E varie versioni ricorsive, la prima per ogni nodo verifica che sia maggiore o uguale al massimo del sottoalbero destro e minore o uguale del minimo del sottoalbero destro. Dato che prima controlla che tali sottoalberi siano ABR può cercare minimo e massimo in tempo $\log n$ cosa che assicura che il costo complessivo è lineare. Infatti $T(n) = 2T(n/2) + 2\log n$ che secondo il master theorem porta a $T(n) = \Theta(n)$.

```
 IsABR0(A,n)
  return IsABR0_rec(A,n,1)

IsABR0_rec(A,n,i):
  if i > n
    return True
  else:
    isABRL = IsABR0_rec(A,n,2*i)
    isABRR = IsABR0_rec(A,n,2*i+1)
    M = max(A,n,2*i)                 # massimo del sottoalbero sx (assumendolo
                                       ABR, gia' controllato!)
    m = min(A,n,2*i+1)               # minimo del sottoalbero dx (assumendolo
                                       ABR, gia' controllato!)
    return isABRL and isABRR
                  and A[i] >= m
                  and A[i] <= M


Tmin(A,n,i):
  if i>n:
    return +INFTY
  else:
    while (2*i <=n):
      i = 2*i    # scende a sx

    return A[i]
```

Si può evitare il ciclo per cercare minimo e massimo tenendo conto conto che il minimo corrisponde a $A[m]$ con m massimo valore $\leq n$ ottenuto moltiplicando i per 2, ovvero $m = i * 2^k \leq n$, con h massimo quindi

$$h = \lfloor \log_2 n/i \rfloor$$

e similmente, il massimo corrisponde a $A[M]$ con M ottenuto da i moltiplicando per 2 e sommando 1, iterativemanete, ma senza superare n. Osservando che l'operazione suddetta iterata k, ovvero $(i*2+1)*2+1\ldots)$ fatto k volte, conduce a $i*2^k+1+2+\ldots+2^{k-1}$, si ha $M = i*2^k+2^k-1 = 2^k(i+1)-1 \leq n$ con k massimo, per cui

$$k = \lfloor \log_2(n+1)/(i+1) \rfloor$$

Le altre di costo lineare:

```
# soluzione di tempo lineare

IsABR1(A,n)
  isABR,m,M = IsABR1_rec(A,n,1)     # ritorna tre valori per il sottoalbero radicato in A[i]
                                    #   - e' ABR? (true/false)
                                    #   - minimo
                                    #   - massimo
  return isABR

IsABR1_rec(A,n,i):
  if i > n
```

```
    return True, INFTY, -INFTY
  else:
    isABRL, mL, ML = IsABR1_rec(A,n,2*i)
    isABRR, mR, MR = IsABR1_rec(A,n,2*i+1)

    return isABRL and isABRR
                  and A[i] >= ML
                  and A[i] <= mR,
           min(mL, mR, A[i]),
           max(ML, MR, A[i])

// altra soluzione di tempo lineare
// effettua una visita e verifica simmetrica e che produca l'array ordinato

IsABR2(A,n)
   isABR, last = visitAndCheck(A,1,-INFTY)
                                // visita simmetrica con verifica che il prossimo
                                   // valore incontrato sia inferiore al precedente
                                // ritorna due valori
                                           //   ok
                                    //   ultimo valore prodotto
  return isABR

visitAndCheck(A,i,last)
  if i > n
    return True, last
  else:
    ok_left, last = visitAndCheck(A,2*i,last)
    // conceptually, now visit A[i]
    ok = last <= A[i]
    ok_right, last = visitAndCheck(A,2*i+1,A[i])
    return ok_left and ok and ok_right, last
```

Domanda 38 Si consideri una variante degli alberi binari di ricerca nella quale i nodi `x` hanno un campo `x.pred` (predecessore) invece che il campo `x.p` (parent). Realizzare la procedura di `Insert(T,z)` che inserisce un nodo `z` nell'albero. Valutarne la complessità.

Soluzione: La soluzione con complessità $O(h)$ dove h è l'altezza dell'albero può essere:

```
Insert(T,z)
    x = T.root
    y = nil         // father
    w = nil         // successor
    while (x <> nil)
       y = x

       if z.key < x.key
          x = x.left
          w = y
       else
          x = x.right
```

```
z.p = y        // omit if 'parent' is not a field

if y = nil
   T.root = z
else if z.key < y.key
   y.left = z
   z.prec = y.prec
else
   y.right = z
   z.prec = y

if w <> nil
   w.prec = z
```

Esercizio 14 Realizzare una procedura `BST(A)` che dato un array `A[1..n]` di interi, ordinato in modo crescente, costruisce un albero binario di ricerca di altezza minima che contiene gli elementi di `A` e ne restituisce la radice. Per allocare un nuovo nodo dell'albero si utilizzi una funzione `mknod(k)` che dato un intero `k` ritorna un nuovo nodo con `x.key=k` e figlio destro e sinistro `x.left = x.right = nil`. Valutarne la complessità.

Soluzione:

i. L'implementazione è la seguente:

```
BST(A)
   return BST-rec(A,1,n)

BST-rec(T,A,p,q)
   if p <= q
      m = floor(p+q/2)
      x=mknode(A[m])
      x.l = BST-rec(A,p,m-1)
      x.r = BST-rec(A,m+1,q)
    else
      x = nil

    return x
```

ii. Si ottiene la ricorrenza $T(n) = 2T(n/2) + \Theta(1)$ e quindi il costo è $T(n) = O(n)$.

Esercizio 15 Si consideri una estensione degli alberi binari di ricerca nei quali ogni nodo x ha anche un campo booleano $x.\mathit{even}$ che vale *true* o *false* a seconda che la somma delle chiavi nel sottoalbero radicato in x sia pari o dispari. Realizzare la procedura `Insert(T,z)` di modo che mantenga correttamente aggiornato anche il campo *even*. Valutarne la complessità.

Soluzione:

```
Insert(T,z)
    x = T.root
    y = nil
    z.even = (z.key mod 2 == 0)
```

```
    while (x <> nil)
       x.even = (x.even == z.even)    // z e‘ inserito nel sottoalbero radicato
                                      // in x, quindi si aggiunge alla somma
                                      // che sara’ pari  se la somma precedente
                                      // e z.key sono entrambi pari o entrambi
                                      // dispari e sara’ dispari altrimenti

       y = x

       if z.key < x.key
          x = x.left
       else
          w = y
          x = x.right

    z.p = y
    if y = nil
       T.root = z
    else if z.key < y.key
       y.left = z
    else
       y.right = z
```

La complessità è $\Theta(h)$.

Esercizio 16 Scrivere una funzione `range(T,k1,k2)` che dato un albero binario di ricerca T e due interi k_1 e k_2 tali che $k_1 \leq k_2$, stampa, in ordine crescente, tutte le chiavi k contenute nell'albero tali che $k_1 \leq k \leq k_2$. Valutarne la complessità.

Soluzione:

```
range(T, k1, k2)
    range_rec(T.root, k1, k2)

range_rec(x,k1,k2)
  if x <> nil
       if x.key >= k1
          range_rec(x.left, k1, k2)

       if x.key >= k1 and x.key <= k2
          print x.key

       if x.key <= k2
          range_rec(x.right, k1, k2)
```

Esercizio 17 Scrivere una funzione `RBTree(T)` che dato in input un albero binario di ricerca T, i cui nodi x, oltre ai campi $x.key$, $x.left$ e $x.right$, hanno un campo $x.col$ che può essere B (per "black") oppure R (per "red"), verifica se questo è un Red-Black tree. In caso negativo, restituisce -1, altrimenti restituisce l'altezza nera della radice. Valutarne la complessità.

Soluzione:

```
RBTree(T) // verifica se T e' un RB-tree
  if T.root == T.nil
     return 0
  elif T.root.color == R
      return -1
  else
      return RB-rec(T.root)

RB-rec(x) // calcola l'altezza nera di x e la restituisce se
             - le altezze nere dei figli sono uguali
             - se il nodo e‘ R i figli sono B
             - le foglie dei sottoalberi sono B
             altrimenti restituisce -1

  if x == T.nil     // foglia, T.nil e' automaticamente nero
     return 0

  elif (x.color == R) and ((x.l.color == R) or (x.r.color == R))
     return -1

  else
     left =  RB-rec(x.left)
     right = RB-rec(x.right)

     // se uno dei due sottoalberi non soddisfa le richieste
     // oppure entrambi i sottoalberi le soddisfano ma
     // sono non vuoti e con altezze nere diverse

     if (right == -1) or (left == -1) or (left <> right)
        count=-1
     else
        count = left
        if x.col == B
           count = count+1

        return count
```

Per quanto riguarda la complessità si osservi che $T(n) = c + T(k) + T(n-k)$ per un'opportuna costante c e quindi, la complessità è $O(n)$.

8 Programazione Dinamica

Esercizio 18 Dare un algoritmo per individuare, all'interno di una stringa $a_1 \dots a_n$ una sottostringa (di caratteri consecutivi) palindroma di lunghezza massima. Ad esempio, nella stringa "*colonna*" la sottostringa palindroma di lunghezza massima è "*olo*". Più precisamente:

i. dare una caratterizzazione ricorsiva della lunghezza massima $l_{i,j}$ di una sottostringa palindroma di $a_i \dots a_j$;

ii. tradurre tale definizione in un algoritmo (bottom up o top down con memoization) che determina la lunghezza massima;

iii. trasformare l'algoritmo in modo che permetta anche di individuare la stringa, non solo la sua lunghezza;

iv. valutare la complessità dell'algoritmo.

Soluzione: La caratterizzazione ricorsiva della lunghezza massima $l_{i,j}$ di una sottostringa palindroma di $a_{i,j} = a_i \dots a_j$ deriva dall'osservazione che

- se $i > j$, quindi $a_{i,j} = \varepsilon$ è la stringa vuota, dato che la stringa vuota è palindroma $l_{i,j} = 0$;
- se $i = j$ quindi $a_{i,j}$ è la stringa di un solo carattere, che è palindroma, vale $l_{i,j} = 1$;
- altrimenti, se $i < j$ quindi $a_{i,j}$ consta di almeno due caratteri, distinguiamo due sottocasi:
  - se $a_i = a_j$ e la sottostringa $a_{i+1,j-1}$ è palindroma, anche $a_{i,j}$ è palindroma. Quest'ultima condizione si riduce a $l_{i+1,j-1} = |a_{i+1,j-1}| = j-i-1$. In questo caso vale $l_{i,j} = j-i+1$ (lunghezza di $a_{i,j}$)
  - se invece $a_i \neq a_j$, una sottostringa palindroma non può comprendere entrambi gli estremi e quindi ci si riduce ai sottoproblemi $a_{i+1,j}$ e $a_{i,j-1}$ e si prende il massimo.

In sintesi:

$$l_{i,j} = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ l_{i+1,j-1} + 2 & \text{if } a_i = a_j \text{ and } l_{i+1,j-1} = j-i-1 \\ \max\{l_{i,j-1}, l_{i+1,j}\} & \text{otherwise } (a_i = a_j \text{ and } l_{i+1,j-1} \neq j-i-1 \text{ or } a_i \neq a_j) \end{cases}$$

Ne segue l'algoritmo che riceve in input la stringa, nella forma di un array di caratteri $A[1,n]$ e usa una matrice $L[1..n, 0..n]$ dove $L[i,j]$ rappresenta la lunghezza di una sottostringa palindroma in $A[i,j]$

```
palindrome(A, n)
  for i = 1 to n
    L[i,i-1] = 0
    L[i,i] = 1

  for len = 2 to n
    for i = 1 to n-len+1
      j = i + len - 1
      if (A[i] = A[j]) and (L[i+1,j-1] = j-i-1)
         L[i,j] = L[i+1,j-1] + 2
      else
         L[i,j] = max (L[i,j-1], L[i+1,j])

  return L[1,n]
```

Se vogliamo anche la sottostringa, occorre ricordare il massimo e dove viene raggiunto, lo facciamo mediante un array $P[1..n, 1..n]$ tale che $P[i, j]$ contenga l'indice dal quale inizia la sottostringa palindroma più lunga di $a_{i,j}$

```
palindrome(A, n)
  for i = 1 to n
    L[i,i-1] = 0
    L[i,i] = 1
    P[i,i] = i

  for len = 2 to n
    for i = 1 to n-len+1
      j = i + len - 1
      if (A[i] = A[j]) and (L[i+1,j-1] = j-i-1)
         L[i,j] = L[i+1,j-1] + 2
         P[i,j] = i
      else
         if L[i,j-1] >= L[i+1,j]
            L[i,j] = L[i,j-1]
            P[i,j] = P[i][j-1]
         else
            L[i,j] = L[i+1,j]
            P[i,j] = P[i+1][j]

   start = P[1,n]
   end   =  start + L[1,n]-1

   return A[start..end]
```

La complessità è $\Theta(n^2)$.

Una soluzione differente, non basata su programmazione dinamica, consiste nel cercare da ogni punto della stringa, una sottostringa palindroma centrata in quel punto di lunghezza dispari, che dunque includa quel carattere come centrale oppure pari, quindi che parte dalla stringa vuota.

Dato che i punti da controllare sono n e per ognuno i due casi (pari o dispari) costano al più n, ottengo comunque una complessità quadratica.

```
# extend the extremes as much as possible, with pairs of identical
# letters (starting from j=i+1 or j=i+2 at the end we are sure that
# A[i+1,j-1] is palindrome)
extend (A, i, j, n)
    while (i>=1) and (j<=n) and (A[i] == A[j]):
        i--
        j++

    return i+1, j-i-1

# search the longest palindrome substring
palindrome(A,n)

    m_start = -1    # start of a longest palindrome
    m_len   = -1    # length of a longest palindrome

    for i = 1 to n
        start, len = extend(A,i,i+1,n) # find the longest palindrome of even len
```

```
        if (m_len < len)               # centered in i
            m_len = len
            m_start = start

        start, len = extend(A,i,i+2,n)  # find the longest palindrome of odd len
        if (m_len < len)                # centered in i
            m_len = len
            m_start = start

    return A[m_start:m_start+m_len-1]
```

Esercizio 19 Sia data un'espressione $E = x_1\ op_1\ x_2\ op_2\ \ldots\ x_{n-1}\ op_{n-1}\ x_n$, con $n \geq 2$, dove ogni x_i è un intero positivo e $op_i \in \{+, *\}$ di somma o di moltiplicazione (dati). Utilizzando la programmazione dinamica si determini una parentetizzazione dell'espressione che rende il valore dell'espressione minimo. Ad esempio, l'input 7+10∗2 può essere parentetizzato come ((7+10)∗2) = 34 oppure come $(7 + (10 * 2)) = 27$. In questo caso, la parentetizzazione desiderata è quindi la seconda. Più precisamente:

i. dare una caratterizzazione ricorsiva del valore minimo $v_{i,j}$ prodotto da una parentetizzazione della sottoespressione $x_i\ op_i \ldots op_{j-1}\ x_j$;

ii. tradurre tale definizione in un algoritmo (bottom up o top down con memoization) che determina il valore minimo;

iii. trasformare l'algoritmo in modo che permetta anche di stampare l'espressione;

iv. valutare la complessità dell'algoritmo.

Soluzione: Vale la proprietà di sottostruttura ottima, ovvero se prendo una parentetizzazione ottima dell'espressione $x_i\ op_i \ldots op_{j-1}\ x_j$ e considero la posizione k delle parentesi più esterne $(x_i\ op_i \ldots op_{k-1} x_k) op_k (x_{k+1} opk + 1 \ldots op_{j-1}\ x_j)$, per ogni sottoespressione ottengo una parentetizzazione ancora ottima. Questo deriva dal fatto che la somma è monotona negli argomenti, e il prodotto lo è per numeri non negativi (quindi nelle ipotesi). Se invece avessimo avuto anche numeri negativi, occorreva tenere sia il minimo che il massimo e tener conto del segno (es. per avere il valore minimo di un prodotto quando un fattore è positivo e l'altro negativo, occorre prendere il minimo per il primo e il massimo per il secondo). A margine, si noti che se i numeri erano strettamente positivi, allora si poteva procedere con un algoritmo greedy che esegue prima i prodotti e poi le somme (tanto le operazioni sono associative). In presenza dello zero, non vale più (forse si puo' adattare?)

Sulla base della discussione precedente, la caratterizzazione ricorsiva è quindi:

$$v_{i,j} = \begin{cases} x_i & \text{if } i = j \\ \min\{v_{i,k}\ op_k\ v_{k+1,j} \mid i \leq k \leq j-1\} & \text{if } j > i \end{cases}$$

Ne segue l'algoritmo che riceve in input l'espressione, nella forma di un array di due array $x[1, n]$, $op[1, n-1]$ e usa una matrice $v[1..n, 0..n]$ dove $v[i, j]$ rappresenta il valore minimo per una parentetizzazione della sottoespressione corrispondente.

```
ParMin(x,op,n)
  for i=1 to n
    v[i,i] = x[i]

  for len = 2 to n
```

```
      for i = 1 to n - len + 1
                j = i + len - 1
          v[i,j] = +infty
          for k = i to j-1
              q =  v[i,k] op[k] v[k+1,j]
              if v[i,j] > q
                 v[i,j] = q
  return v
```

Se vogliamo anche la parentetizzazione, occorr ricordare il punto in cui si "spezza" ogni espressione e lo facciamo mediante un array $p[1..n, 1..n]$ tale che $p[i, j]$ contenga l'indice k per il quale si raggiunge il minimo.

```
ParMin(x,op,n)
  for i=1 to n
    v[i,i] = x[i]

  for len = 2 to n
      for i = 1 to n - len + 1
                j = i + len - 1
          v[i,j] = +infty
          for k = i to j-1
              q =  v[i,k] op[k] v[k+1,j]
              if v[i,j] > q
                 v[i,j] = q
                 p[i,j] = k
  return v,p
```

Quindi, per stampare la parentetizzazione:

```
print(x,op,n)
  v,p = ParMin(x,op,n)
  print_rec(x,op,1,n,p)


print_rec(x,op,i,j,p)
  if i=j
     print x[i]
  else
     k = p[i,j]
     print "("
     print_rec (x,op,i,k,p)
     print ")", op[k], "("
     print_rec (x,op,k+1,j,p)
     print ")"
```

La complessità è $\Theta(n^3)$.

Esercizio 20 Si ricordi che data una sequenza $X = x_1 \dots x_k$, si indica con X_i il prefisso $x_1 \dots x_i$. Una sottosequenza di X è $x_{i_1} \dots x_{i_h}$ con $1 \leq i_1 < i_2 < \dots i_h \leq k$, ovvero è una sequenza ottenuta da X eliminando alcuni elementi. Quando Y è sottosequenza di X si scrive $Y \sqsubseteq X$.

Realizzare un algoritmo che, date due sequenze $X = x_1 \dots x_k$ e $Y = y_1 \dots y_h$ determina una *shortest common supersequence* (SCS) ovvero una sequenza Z, di lunghezza minima, tale che $X \sqsubseteq Z$ e $Y \sqsubseteq Z$. Ad esempio per $X = abf$ e $Y = afgj$ una SCS è $abfgj$.

i. Dare una caratterizzazione ricorsiva della lunghezza $l_{i,j}$ di una SCS di X_i e Y_j e dedurne un algoritmo;

ii. trasformare l'algoritmo in modo che fornisca una SCS di X e Y;

iii. valutare la complessità dell'algoritmo.

Soluzione: La caratterizzazione ricorsiva è:

$$l_{i,j} = \begin{cases} i+j & \text{if } i=0 \text{ or } j=0 \\ l_{i-1,j-1}+1 & \text{if } i,j>0 \text{ e } x_i = y_j \\ \min\{l_{i,j-1}, l_{i-1,j}\}+1 & \text{if } i,j>0 \text{ e } x_i \neq y_j \end{cases}$$

Ne segue l'algoritmo che riceve in input le stringhe, nella forma di array di caratteri $X[1,k]$, $Y[1,h]$ e usa una matrice $L[0..k, 0..h]$ dove $L[i,j]$ rappresenta la lunghezza della minima SCS di X_i e Y_j.

```
SCSlen (X,Y,k,h)
  for i=0 to k
    L[i,0] = i
  for j=1 to h
    L[0,j] = j

  for i=1 to k
    for j = 1 to h
      if (X[i] = X[j])
         L[i,j] = L[i-1,j-1] + 1
      else
         L[i,j] = min (L[i,j-1], L[i-1,j]) + 1

  return L[k,h]
```

Se vogliamo anche la SCS occorre tener conto delle scelte effettuate per raggiungere l'ottimo. Lo facciamo mediante un array $P[1..n, 1..n]$

```
SCS-data (X,Y,k,h)
  for i=0 to k
    L[i,0] = i
  for j=1 to h
    L[0,j] = j


  for i=1 to k
    for j = 1 to h
      if (X[i] = X[j])
         L[i,j] = L[i-1,j-1] + 1
         P[i,j] = "xy"
      else
         if L[i,j-1] <= L[i-1,j]
            L[i,j] =  L[i,j-1] + 1
             P[i,j] =  y
         else
            L[i,j] =  L[i-1,j] + 1
            P[i,j] =  x
```

```
  return L, P


SCS (X,Y,i,j,P)
  if i==0
     print Y[j]
  elseif j==0
     print X[i]
  else
    if P[i,j] = xy
       SCS(X,Y,i-1,j-1,P)
       print X[i]

    elseif P[i,j] = x
       SCS(X,Y,i-1,j,P)
       print X[i]

    else
       SCS(X,Y,i,j-1,P)
       print Y[j]
```

La complessità è $\Theta(hk)$.

Esercizio 21 Si supponga di dover pagare una certa somma s. Per farlo si hanno a disposizione le banconote $b_1, \ldots, b_n$ ciascuna di valore $v_1, \ldots, v_n$. Si vuole determinare, se esiste, un insieme di banconote $b_{i_1}, \ldots, b_{i_k}$ che totalizzi esattamente la somma richiesta e che minimizzi il numero k di banconote utilizzate.

i. mostrare che vale la proprietà della sottostruttura ottima e fornire una caratterizzazione ricorsiva del costo $c(s', j)$ della soluzione ottima per il sottoproblema di dare una somma pari a s' con le banconote in $b_1, \ldots, b_j$, con $j \leq n$;

ii. tradurre tale definizione in un algoritmo (bottom up o top down con memoization) che determina il costo della soluzione ottima;

iii. trasformare l'algoritmo in modo che permetta anche di individuare la soluzione, non solo il suo costo;

iv. valutare la complessità dell'algoritmo.

Soluzione:

i. Vale la proprietà della sosttostruttura ottima, ovvero se $b_{i_1}, \ldots, b_{i_k}$ è una soluzione ottima per il problema con somma s e banconote $B = b_1, \ldots, b_n$ allora certamente $b_{i_2} \ldots b_{i_k}$ è soluzione ottima per il problema con somma $s - v_{i_1}$ e banconote $B \setminus \{b_{i_1}\}$. Infatti una soluzione migliore, ovvero con un numero di banconote inferiore a $k-1$, per il sottoproblema potrebbe essere combinata con b_{i_1} e darebbe una soluzione per il problema di partenza, con un numero di banconote $< k$ e quindi migliore di quella ottima.

   La caratterizzazione ricorsiva del costo della soluzione ottima $c(s', j)$ segue immediatamente dall'osservazione che

   - se $s' = 0$, ovvero la somma da dare è nulla, non occorre alcuna banconota $c(s', j) = 0$.

- se $s' > 0$ e $j = 0$, dato che non possiamo totalizzare una somma non nulla con 0 banconote non c'è soluzione e convenzionalmente assegnamo $c(s', j) = +\infty$
- Altrimenti, $j > 0$. Se $v_j \leq s'$ la banconota j-ma potrebbe essere utilizzata o meno, quindi si considera il minimo tra la soluzione ottima che la include e quella che non la include

  $$c(s', j) = \min\{c(s', j-1), c(s' - v_j, j-1) + 1\}$$

  Infine, se $v_j > s'$ la banconota va certamente scartata, quindi $c(s', j) = c(s', j-1)$.

In sintesi:

$$c(s', j) = \begin{cases} 0 & \text{if } s' = 0 \\ \infty & \text{if } j = 0 \text{ and } s' > 0 \\ \min\{c(s', j-1), c(s' - v_j, j-1) + 1\} & \text{if } j > 0,\ s' > 0 \text{ and } v_j \leq s' \\ c(s', j-1) & \text{altrimenti} \end{cases}$$

ii. Ne segue l'algoritmo che riceve in input la somma s da totalizzare e l'array `v[1..n]` dei valori delle banconote a disposizione,

```
banconote(s,v,n)
  for j = 0 to n
      for s' = 0 to s
          c[s',j] = -1    // valore non usato
  return banconote_rec(s,v,n,c)



banconote_rec(s',v,j,c)
   if c[s',j] = -1
      if s'=0
         c[s',j] = 0
      elseif j=0
         c[s',j] = infy    // somma impossibile da totalizzare
      elseif v[j] <= s'
         c[s',j] = min { banconote_rec(s',v,j-1,c),
                         banconote_rec(s'-v[j],v,j-1,c)+1 }
      else
         c[s',j] = banconote_rec(s',v,j-1,c)

   return c[s',j]
```

iii. Se vogliamo anche il dettaglio delle banconote

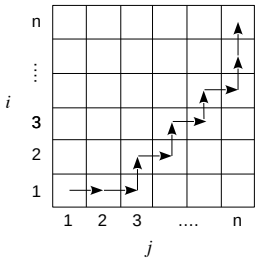
```
banconote(s,v,n)
  for j = 0 to n
      b[j] = false              // b[j] says whether the j-th note is chosen
      for s' = 0 to s
          c[s',j] = -1                // valore non utilizzato
  banconote_rec(s,v,n,c)
  return b, c

banconote_rec(s',v,j,c)
   if c[s',j] = -1
      if s'=0
         c[s',j] = 0
      elseif j=0
```

```
            c[s',j] = infty
         elseif v[j] <= s'
            q_no  = banconote_rec(s',v,j-1,b,c)
            q_yes = banconote_rec(s'-v[j],v,j-1,b,c) + 1
            if q_no <= q_yes
               c[s',j] = q_no
            else
               c[s',j] = q_yes
               b[j] = true
         else
            c[s',j] = banconote_rec(s',v,j-1,b,c)

      return c[s',j]


   // valore di ritorno:
   c[s,n] = infty   significa che la somma 's' non puo' essere ottenuta
   c[s,n] = value   da il minimo numero di banconote
```

iv. La complessità è $\Theta(sn)$.

Esercizio 22 Si supponga di avere una scacchiera `n` $\times$ `n`. Si vuole spostare un pezzo dall'angolo in basso a sinistra (1,1) a quello in alto a destra `(n,n)`.

Il pezzo può muoversi di una casella verso l'alto ($\uparrow$) o verso destra ($\rightarrow$). Un passo dalla casella `(i,j)` ha un costo `u(i,j)` se verso l'alto e `r(i,j)` se verso destra. Realizzare un algorimo `MinPath(u,r,n)` che dati in input gli array `u[1..n,1..n]` e `r[1..n,1..n]` dei costi dei singoli passi fornisce il cammino minimo. Più in dettaglio:



i. fornire una caratterizzazione ricorsiva del costo minimo di un cammino `C(i,j)` per andare dalla casella `(i,j)` alla casella `(n,n)`

ii. tradurre tale definizione in un algoritmo `MinPath(u,r,n)` (bottom up o top down con memoization) che determina il costo di un cammino minimo da `(1,1)` a `(n,n)`

iii. trasformare l'algoritmo in modo che stampi la sequenza di passi di costo minimo;

iv. valutare la complessità dell'algoritmo.

Soluzione:

i. La caratterizzazione ricorsiva del costo è:

$$C(i,j) = \begin{cases} 0 & \text{se } x = n \text{ e } y = n \\ r(i,j) + C(i,j+1) & \text{se } i = n \text{ e } j < n \\ u(i,j) + C(i+1,j) & \text{se } i < n \text{ e } j = n \\ \min\{r(i,j) + C(i,j+1), u(i,j) + C(i+1,j)\} & \text{altrimenti} \end{cases}$$

ii. Ne segue l'algoritmo che riceve in input gli array `u(1..n,1..n)` e `r(1..n,1..n)` dei costi e calcola il costo minimo di un cammino da `(1,1)` a `(n,n)`

```
// u[1..n,1..n], r[1..n,1..n] costi per andare in alto
// e a destra, rispettivamente

MinPath(u,l,n)
  C[n,n] = 0


  for j = n-1 downto 1
      C[n,j] = C[n,j+1] + r[n,j]

  for i = n-1 downto 1
      C[i,n] = C[i+1,n] + u[i,n]

for i=n-1 downto 1
      for j = n-1 downto 1
                C[i,j] = min { C[i+1,j] + u[i,j], C[i,j+1] + r[i,j] }

  return C[1,1]
```

iii. Se vogliamo anche una soluzione ottima

```
// u[1..n,1..n], r[1..n,1..n] costi per andare in alto
// e a destra, rispettivamente

MinPath(u,l,n)
  C[n,n] = 0

  for j = n-1 downto 1
      C[n,j] = C[n,j+1] + r[n,j]
      D[n,j] = 'right'

  for i = n-1 downto 1
      C[i,n] = C[i+1,n} + u[i,n]
      D[i,n] = 'up'

  for i=n-1 downto 1
      for j = n-1 downto 1
                up    = C[i+1,j] + u[i,j]
            right = C[i,j+1] + r[i,j]

            if up <= right
               C[i,j] = up
               // memorizza anche la dirazione ottima
               D[i,j] = 'up'
            else
               C[i,j] = right
               D[i,j] = 'right'

  i = j =1
  while (i < n) or (j < n)
     print (i,j)
     if D[i,j] = up
        i=i+1
     else
        j=j+1
```

iv. La complessità è $O(n^2)$.

Esercizio 23 Sia data una sequenza di città c_1 c_2 ... c_n collegate da un percorso ferroviario. Da ciascuna città c_i è possibile raggiungere c_j, con $i < j$ con un treno diretto. Sapendo che per $i, j \in \{1, \ldots, n\}$, $i < j$ il costo del biglietto del treno diretto da c_i a c_j risulta essere `b[i,j]`, determinare il costo minimo di un tragitto, con possibili cambi intermedi, da c_1 a c_n. Più in dettaglio:

i. fornire una caratterizzazione ricorsiva del costo minimo `min[i]` di un di un tragitto dalla città c_i alla città c_n;

ii. tradurre tale definizione in un algoritmo `MinTrain(b,n)` (bottom up o top down con memoization) che determina il costo di un tragitto di costo minimo dalla città c_1 alla città c_n;

iii. trasformare l'algoritmo di modo che determini anche la sequenza dei cambi necessari per ottenere il tragitto di costo minimo, privilegiando – a parità di costo – soluzioni che minimizzano il numero dei cambi;

iv. valutare la complessità dell'algoritmo.

Soluzione:

i. La caratterizzazione ricorsiva del costo è:

$$min[i] = \begin{cases} 0 & \text{se } i = n \\ min\{b[i,j] + min[j] \mid j \in \{i+1, n\}\} & \text{altrimenti} \end{cases}$$

ii. Ne segue l'algoritmo che riceve in input gli array `u(1..n,1..n)` e `r(1..n,1..n)` dei costi e calcola il costo minimo di un cammino da `(1,1)` a `(n,n)`

```
// b[1..n,1..n] con b[i,j] costo per andare da Ci a Cj

MinTrain(b,n)
  for i = n downto 1
      min[i] = b[i,n]             // costo minimo per andare da Ci a Cn
      for j = i+1 to n-1
                q = b[i,j] + min[j]
                if (q < min[i])
            min[i] = q
  return b[1]
```

iii. Se vogliamo anche una soluzione ottima

```
// b[1..n,1..n] con b[i,j] costo per andare da Ci a Cj

MinTrain(b,n)
  for i = n downto 1
      min[i] = b[i,n] // costo minimo per andare da Ci a Cn
      next[i] = n     // prossima stazione a cui fermarsi in una sol. ottima
      size[i] = 0     // num. di stazioni per raggiungere Cn in una sol. ottima
      for j = i+1 to n-1
                q = b[i,j] + min[j]
                if (q < min[i] or (q = min[i] and (size[i] > size[j]+1))
```

```
            min[i] = q
            next[i] =j
            size[i] = size[j] + 1

// stampa la sequenza delle citta'
i = 1
while (i<n) {
        print i
      i = next[i]
```

iv. La complessità è $O(n^2)$.

Esercizio 24 Si supponga che il mondo consista di n stati $S_1 \ldots, S_n$ e che spostarsi dallo stato S_i allo stato S_j imponga l'ottenimento di un visto di ingresso che ha un costo $v_{i,j}$, maggiore di 0. Realizzare un algoritmo che determini una sequenza di stati che conviene percorrere per andare dallo stato S_1 allo stato S_n, minimizzando il costo complessivo dei visti. Più precisamente:

i. Dare una caratterizzazione ricorsiva del costo complessivo minimo v_i dei visti per andare dallo stato S_i allo stato S_n;

ii. Usare la caratterizzazione al punto precedente per ottenere un algoritmo $ESP(v, n)$ che dato l'array dei costi dei visti $v[1..n, 1..n]$ determina il costo minimo dei visti per andare da S_1 a S_n;

iii. trasformare l'algoritmo in modo che determini oltre al costo minimo anche la sequenza degli stati da attraversare per ottenerlo;

iv. valutare la complessità dell'algoritmo.

Soluzione:

i. La caratterizzazione ricorsiva del costo si basa sull'osservazione che il costo v_i di un cammino minimo da S_i a S_n gode della proprietà che

$$v_i = \min\{v_{i,j} + v_j \mid i \neq j\}$$

Così però non riduco il problema originario a sottoproblemi più piccoli, ma a problemi della stessa dimensione. Una osservazione a questo punto è che il cammino minimo (dato che i costi sono positivi) non passa mai due volte per uno stato (non contiene cicli). Dunque la scelta del primo nodo del cammino, riduce l'insieme di stati nodi del grafo. Più precisamente, se indichiamo con $v_{i,N}$ il cammino minimo che conduce da S_i a S_n, usando solo i nodi in N abbiamo che

$$v_{i,N} = \min\{v_{i,j} + v_{j,N\setminus\{i\}} \mid j \in N\}$$

Si osservi che i sottoproblemi distinti sono dell'ordine dei sottoinsiemi di $\{1, \ldots, n\}$ pertanto in numero esponenziale!

Si potrebbe continuare ed ottenere una soluzione molto inefficiente.

```
// v[1..n,1..n] con v[i,j] costo per andare da Si a Sj, assumendo v[i,i] =0
// calcola
//   cost[i,l]: costo per andare da i a n con un percorso di lunghezza <= l
```

```
Min(v,n)
  for i=1 to n-1
      cost[i,0] =  infty
  cost[n,0] = 0

  for l=1 to n-1
      min = infty
      for j=1 to n
                 if v[i,j] + cost[i,N] < min
             min = v[i,j] + cost[i,N]
      cost[i,l] = min

  return cost[1,n-1]
```

Un modo per rendere l'approccio più efficiente è osservare che, invece che escludere un unico nodo dal percorso, posso escludere i percorsi di lunghezza superiore ad una certa soglia, ovvero posso considerare

$$v_{i,\ell}\text{: percorso minimo da } S_i \text{ a } S_n\text{, di lunghezza} \leq \ell.$$

In questo caso i sottoproblemi distinti, dato che il fatto che i percorsi minimi siano aciclici ci permette di limitarci a percorsi di lunghezza $\ell \leq n-1$ sono n^2.

La caratterizzazione ricorsiva:

$$v_{i,\ell} = \begin{cases} 0 & \text{se } i = n \\ \infty & \ell = 0 \text{ e } i \neq n \\ \min\{v_{i,j} + v_{j,\ell-1} \mid j \in \{1,\ldots,n\}\} & \text{altrimenti} \end{cases}$$

dove si assume che $v_{i,i} = 0$ e quindi la minimizzazione include anche $v_{i,i} + v_{i,\ell-1}$ ovvero il percorso di lunghezza $\ell - 1$.

ii. Ne segue l'algoritmo che riceve in input l'array `v(1..n,1..n)` e calcola il costo minimo di un cammino da S_1 a S_n

```
// v[1..n,1..n] con v[i,j] costo per andare da Si a Sj, assumendo v[i,i] =0
// calcola
//   cost[i,l]: costo per andare da i a n con un percorso di lunghezza <= l
Min(v,n)
  for i=1 to n-1
      cost[i,0] =  infty
  cost[n,0] = 0

  for l=1 to n-1
      min = infty
      for j=1 to n
                 if v[i,j] + cost[i,N] < min
             min = v[i,j] + cost[i,N]
      cost[i,l] = min

  return cost[1,n-1]
```

iii. Se vogliamo anche una soluzione ottima

iv. La complessità è $O(n^2)$.

Esercizio 25 Un marinaio ha n pezzi di corda di lunghezze intere $l_1, \ldots, l_n$ ($l_i \geq 2$) e deve annodarli per ottenere una corda di lunghezza esattamente d. Tenendo conto che fare un nodo richiede una lunghezza pari ad 1 (ad es. se annodo due pezzi lunghi 5 e 7 la corda che ottengo è lunga 11), individuare, se esiste, un insieme minimo di pezzi che annodati producano una corda della lunghezza d desiderata.

Più precisamente:

i. Dare una caratterizzazione ricorsiva del numero minimo $m(i, d')$ di pezzi di corda scelti in $l_1, \ldots, l_i$ che possono essere annodati per produrre la lughezza d' ($+\infty$ se non è possibile ottenere d' con nessuna combinazione);

ii. Usare la caratterizzazione al punto precedente per ottenere un algoritmo $Rope(l, n, d)$ che dato l'array delle lunghezze $l[1..n]$ determina il numero minimo di pezzi da annodare per ottenere d (se esiste);

iii. trasformare l'algoritmo in modo che restituisca oltre al numero anche l'indicazione di quali pezzi usare;

iv. valutare la complessità dell'algoritmo.

Soluzione:

i. La caratterizzazione ricorsiva, assumendo $d' > 0$ può essere:

$$m(i, d') = \begin{cases} +\infty & \text{se } i = 0 \\ 1 & \text{se } l_i = d' \\ m(i-1, d') & \text{se } d' > 0 \text{ e } l_i > d' \\ \min\{m(i-1, d'), m(i-1, d' - l_i + 1) + 1\} & \text{altrimenti (se } d' > 0 \text{ e } l_i < d') \end{cases}$$

ii. Ne segue l'algoritmo che riceve in input l'array `l[1..n]` e calcola il numero minimo di pezzi di corda da usare se esiste

```
Rope(l,n,d)
  for d' = 1 to d
      m[0, d'] = +infy

  for i = 1 to n
      for d' = 1 to d
          if (l[i] = d')
             m[i,d'] = 1
          else if (l[i] > d')
             m[i,d] = m[i-1,d']
          else
             m_in  = m[i-1, d'-l[i]+1] + 1
             m_out = m[i-1, d']
             if m_in <= m_out
                m[i,d'] = m_in
             else
                m[i,d'] = m_out
return m[n,d]
```

iii. Se vogliamo anche una soluzione ottima

```
Rope(l,n,d)
  for d' = 1 to d
```

```
        m[0, d'] = +infy

    for i = 1 to n
        for d' = 1 to d
            sol[i,d'] = false              // mantiene an array sol[i,d'] che
                                           // dice se l_i e' usato nella
                                           // soluzione ottima per i, d' (per
                                           // default non lo e')
            if (l[i] = d')
               m[i,d'] = 1
               sol[i,d'] = true
            else if (l[i] > d')
               m[i,d] = m[i-1,d']
            else
               m_out = m[i-1, d']
               m_in  = m[i-1, d'-l[i]+1] + 1
               if m_out <= m_int
                  m[i,d'] = m_out
               else
                  m[i,d'] = m_in
                  sol[i,d'] = true
   return sol
```

iv. La complessità è $O(nd)$.

Alternativamente, possiamo avere come caso base $d' = 0$

$$m(i,d') = \begin{cases} 0 & \text{se } d' = 0 \\ +\infty & \text{se } d' > 0 \text{ e } i = 0 \\ m(i-1,d') & \text{se } d' > 0 \text{ e } l_i - 1 > d' \\ \min\{m(i-1,d'), m(i-1,d'-l_i+1)+1\} & \text{altrimenti (se } d' > 0 \text{ e } l_i - 1 <= d') \end{cases}$$

in questo modo però non si tiene conto che il primo pezzo di corda non richiede di fare il nodo, per cui il valore di interesse sarà $m(d-1,n)$.

Gli algoritmi sono quindi: solo valore:

```
Rope(l,n,d)
  for i=1 to n
      m[i,0] = 0
  for d' = 1 to d
      m[0, d'] = +infy

  for i = 1 to n
      for d' = 1 to d
          if (l[i]-1 > d')
             m[i,d] = m[i-1,d']
          else
             m_in  = m[i-1, d'-l[i]+1] + 1
             m_out = m[i-1, d']
             if m_in <= m_out
                m[i,d'] = m_in
             else
                m[i,d'] = m_out
return m[n,d-1]
```

Con soluzione:

```
Rope(l,n,d)
  for i=0 to n
      m[i,0] = 0

  for d' = 1 to d
      m[0, d'] = +infy

  for i = 1 to n
      for d' = 1 to d
          sol[i,d'] = false              // mantiene an array sol[i,d'] che
                                         // dice se l_i e' usato nella
                                         // soluzione ottima per i, d' (per
                                         // default non lo e')
          if (l[i]-1 > d')
             m[i,d] = m[i-1,d']
          else
             m_out = m[i-1, d']
             m_in  = m[i-1, d'-l[i]+1] + 1
             if m_out <= m_int
                m[i,d'] = m_out
             else
                m[i,d'] = m_in
                sol[i,d'] = yes
return sol
```

Esercizio 26 Siano date n attività $a_1, \ldots, a_n$, che devono essere svolte in un'aula. Ogni attività ha tempo di inizio s_i e tempo di fine f_i, con $s_i < f_i$. Si vuole trovare un sottoinsieme di attività che possano essere svolte nell'aula, quindi senza sovrapposizioni, e che massimizzi il tempo di utilizzo dell'aula.

Più precisamente, assumendo che le attività siano ordinate in modo crescente per tempo di fine:

i. dare una caratterizzazione ricorsiva del tempo massimo $u(i, f)$ di utilizzo dell'aula, quando le attività da allocare siano $a_1, \ldots, a_i$ e l'aula sia disponibile fino al tempo f (ovvero nell'intervallo $[0, f)$);

ii. usare la caratterizzazione al punto precedente per ottenere un algoritmo $allocate(s, f, n)$ che dati gli array dei tempi di inizio e fine delle attività $s[1..n]$ e $f[1..n]$ determini il tempo di utilizzo massimo;
Sugg.: è utile osservare che nel calcolo di $u(i, f)$ i valori significativi di f sono i tempi di inizio delle varie attività e quindi è possibile limitarsi a $u(i, j)$, ovvero tempo massimo di utilizzo dell'aula, con attività $a_1, \ldots, a_i$, quando l'aula sia disponibile fino al tempo s_j;

iii. trasformare l'algoritmo in modo che restituisca oltre al tempo anche anche l'indicazione di quali attività allocare;

iv. valutare la complessità dell'algoritmo.

Soluzione:

i. La caratterizzazione ricorsiva può essere:

$$u(i,f) = \begin{cases} 0 & \text{se } i = 0 \\ u(i-1,f) & \text{se } i > 0 \text{ e } f_i > f \\ \max\{u(i-1,f), u(i-1,s_i) + f_i - s_i\} & \text{altrimenti (se } i > 0 \text{ e } f_i \leq f) \end{cases}$$

con l'osservazione che i valori significativi di f sono i tempi di inizio delle varie attività e ci si può focalizzare su $u(i,j)$, ovvero tempo massimo di utilizzo dell'aula, con attività $a_1, \ldots, a_i$, quando l'aula sia disponibile fino al tempo s_j;

$$u(i,j) = \begin{cases} 0 & \text{se } i = 0 \\ u(i-1,j) & \text{se } i > 0 \text{ e } f_i > s_j \\ \max\{u(i-1,j), u(i-1,i) + f_i - s_i\} & \text{altrimenti (se } i > 0 \text{ e } f_i \leq s_j) \end{cases}$$

ii. Ne segue l'algoritmo *allocate*(s,f,n) che dati gli array dei tempi di inizio e fine delle attività $s[1..n]$ e $f[1..n]$ determina il tempo di utilizzo massimo. Per semplificare il codice assumiamo $s[n+1] = f[n]$, così il valore ottimo complessivo sarà $u[n, n+1]$

```
Allocate(s,f,n)
  for j = 1 to n+1
      u[0, j] = 0

  for j = 1 to n+1
      for i = 1 to  j-1
          if (f[i] > s[j])
             u[i,j] = u[i-1,j]
          else
             in  = u[i-1, i] + f[i]-s[i]
             out = u[i-1, j]
             if in <= out
                u[i,j] = in
             else
                u[i,j] = out
return u[n,n+1]
```

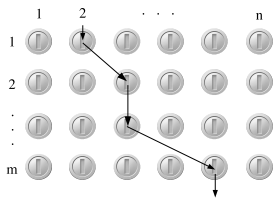
iii. Se vogliamo anche una soluzione ottima

```
Allocate(s,f,n)
  for j = 1 to n+1
      u[0, j] = 0

  for j = 1 to n+1
      for i = 1 to  j-1
          if (f[i] > s[j])
             u[i,j] = u[i-1,j]
          else
             in  = u[i-1, i] + f[i]-s[i]
             out = u[i-1, j]
             if in <= out
                u[i,j] = in
                //  s[i,j] = attivita' i prsente nella soluzione ottima del
                //           sottoproblema con attivita' 1, .., i,
                //                             intervallo [0,s_j]

                s[i,j] = true
             else
                u[i,j] = out
```

```
                s[i,j] = false

    j=n+1
    for i = n downto 1
        if s[i,j]
           print "attivita' " + i
           j = i
```

iv. La complessità è $O(n^2)$.

Esercizio 27 Mario deve attraversare una griglia come in figura, dall'alto verso il basso e per farlo, ad ogni passo salta verso il basso di una riga, spostarsi contestualmente a destra di quanto vuole. Un esempio di attraversamento è indicato in figura. Ogni casella contiene una moneta di un certo valore (possibilmente negativo) per cui nell'attraversamento Mario totalizzerà un certo guadagno.



Supponendo che la griglia abbia dimensione $m \times n$ e che per $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$ la casella (i, j) contenga una moneta di valore $V[i, j]$ realizzare un algoritmo che identifica un attraversamento di valore massimo.

Più precisamente:

i. dare una caratterizzazione ricorsiva del guadagno massimo $G[i, j]$ di un attraversamento della sottogriglia $[i..m, j..n]$;

ii. usare la caratterizzazione al punto precedente per ottenere un algoritmo `mario(V,m,n)` che dato l'array `V` con il valore delle monete determini il guadagno massimo di un attraversamento;

iii. trasformare l'algoritmo in modo che restituisca oltre al guadagno anche anche l'indicazione dell'attraversamento da seguire;

iv. valutare la complessità dell'algoritmo.

Soluzione:

i. La caratterizzazione ricorsiva può essere:

$$G[i,j] = \begin{cases} 0 & \text{se } i = m+1 \\ -\infty & \text{se } j = n+1 \\ \max\{V[i,j] + G[i+1,j], G[i,j+1]\} & \text{altrimenti} \end{cases}$$

ovvero posso decidere di saltare attraverso la casella (i, j) e passare alla riga successiva oppure non usare la casella (i, j), che significherà attraversare la sottogriglia $[i..m, j+1..n]$.

ii. Ne segue l'algoritmo che riceve in input l'array `V[1..m,1..n]` e calcola il guadagno massimo

```
Mario(V,m,m)
    allocate G[1..m+1,1..n]

    for j = 1 to n
            G[m+1,j] = 0
```

```
    for i = 1 to m
            G[i,n+1] = -infty


    for i = m to 1
        for j = n to 1
            if V[i,j] + G[i+1,j] > G[i,j+1]
               G[i,j] = V[i,j] + G[i+1,j]
            else
               G[i,j] = G[i,j+1]


     return G[1,1]
```

iii. Se vogliamo anche una soluzione ottima

```
Mario(V,m,m)
    allocate G[1..m+1,1..n]
    allocate S[1..m,1..n]

    for j = 1 to n
            G[m+1,j] = 0

    for i = 1 to m
            G[i,n+1] = -infty


    for i = m to 1
        for j = n to 1
            if V[i,j] + G[i+1,j] > G[i,j+1]
               G[i,j] = V[i,j] + G[i+1,j]
               S[i,j] = j
            else
               G[i,j] = G[i,j+1]
               S[i,j] = S[i,j+1]


     stampa_sol(S,m,n)


stampa_sol (S,m,n)
     j=1
     for i = 1 to m
         print i, S[i,j]
         j = S[i,j]
```

iv. La complessità è $O(mn)$.

Esercizio 28 Data una sequenza di numeri $X = x_1 x_2 \ldots x_n$ si vuole determinare una sottosequenza $x_{i_1} x_{i_2} \ldots x_{i_k}$ di X, crescente, ovvero tale che $x_{i_1} \leq x_{i_2} \leq \ldots \leq x_{i_k}$, di lunghezza massima. Ad esempio, se $X = 77\ \ 69\ \ 70\ \ 19\ \ 71\ \ 25\ \ 52\ \ 26\ \ 28\ \ 64$, una sottosequenza crescente di lunghezza massima è $19\ \ 25\ \ 26\ \ 28\ \ 64$

i. fornire una caratterizzazione ricorsiva della lunghezza l_i di una sottosequenza crescente di lunghezza massima che abbia come primo carattere x_i;

ii. tradurre tale definizione in un algoritmo `LIS(X,n)` (bottom up o top down con memoization) che determina la lunghezza di una sottosequenza crescente di lunghezza massima della sequenza $X[1..n]$;

iii. trasformare l'algoritmo in modo individui anche la sottosequenza, non solo la sua lunghezza;

iv. valutare la complessità dell'algoritmo.

Soluzione:

i. La caratterizzazione ricorsiva della lunghezza della LIS che inizia x_i è:

$$l_i = \begin{cases} 0 & \text{if } i > n \\ 1 + \max\{l_j \mid j \in [i+1, n] \ \wedge \ x_i \leq x_j\} & \text{if } i < n \end{cases}$$

Si noti che il secondo caso fornisce $l_i = 1$ per $i = n$.

Inoltre, se convenzionalmente fissiamo $x_0 = -\infty$, avremo che la sottosequenza crescente più lunga di X è la sottosequenza più lunga di x_0X che inizia in x_0.

ii. Ne segue l'algoritmo che riceve in input la sequenza X e la lunghezza n e calcola la lunghezza della più lunga sottosequenza crescente

```
LIS(X,n)                        // X[1..n] sequenza da elaborare
  X[0] = -INFTY
  for i = 0 to n
      L[i] = -1                 // L[i] lunghezza LIS piu' lunga di X[i..n] che
                                //      inizi in X[i]
                                // -1   valore non usato
  return LIS_rec(X,n,0,L)-1



LIS_rec(X,n,i,L)
   if L[i] = -1
         max = 0
         for j = i+1 to n       // se i > n, max = 0
         if X[i] <= X[j]:
            q = LIS_rec(X,n,j,L)
            if q > max
                max = q
         L[i] = max+1
    return L[i]
```

iii. Se vogliamo anche una soluzione ottima

```
LIS(X,n)                        // X[1..n] sequenza da elaborare
  X[0] = -INFTY
  for i = 0 to n
      L[i] = -1                 // L[i] lunghezza LIS piu' lunga di X[i..n] che
                                //      inizi in X[i]
                                // N[i] caratterere successivo di una
                                        sottosequenza otima che inizia in X[i]
                                // -1   valore non usato
```

```
LIS_rec(X,n,0,L,N)
i = 0
while N[i] <= n
      print X[N[i]]
      i = N[i]
```

```
LIS_rec(X,n,i,L,N)
   if L[i] = -1
         max = 0
         N[i] = INFTY
         for j = i+1 to n       // se i > n, max = 0
            if X[i] <= X[j]:
              q = LIS_rec(X,n,j,L,N)
              if q > max
                 max = q
                 N[i] = j
         L[i] = max+1
    return L[i]
```

iv. La complessità è $O(n^2)$.

Esercizio 29 Date due sequenze $X = x_1 \dots x_m$ e $Y = y_1 \dots y_m$ su di un alfabeto $\Sigma = \{a_1, \dots, a_\ell\}$ si definisce edit distance il costo minimo di un insieme di operazioni che trasforma la sequenza X nella sequenza Y. Le operazioni possibili sono

- *insert*, ovvero l'inserimento di un simbolo in qualche posizione in X (costo 1);
- *delete*, ovvero la cancellazione di un simbolo in qualche posizione in X (costo 1);
- *replace*, ovvero la sostituzione di un simbolo in qualche posizione in X, con costo che dipende dai simboli coinvolti: sostituire il simbolo a con il simbolo b, con $a \neq b$ ha un costo $c(a, b) > 0$.

Ad esempio per trasformare `gesto` in `gelo` posso fare una replace $\mathtt{s} \to \mathtt{l}$ e una delete di `t` con costo $c(\mathtt{s}, \mathtt{l}) + 1$, oppure una delete `s` e una replace $\mathtt{t} \to \mathtt{l}$ con costo $1 + c(\mathtt{t}, \mathtt{l})$, ecc.

Fornire un algoritmo di programmazione dinamica che dati X, Y e la funzione c dei costi di replace (per $a, b \in \Sigma$, $c(a, b)$ è il costo dell'operazione $a \to b$) calcola la edit distance tra X e Y. In dettaglio,

i. indicata con $e(i, j)$ la edit distance tra $X^i = x_1 \dots x_i$ e $Y^j = y_1 \dots y_j$, con $i \in \{0, \dots, m\}$ e $j \in \{0, \dots, n\}$, darne una caratterizzazione ricorsiva;

ii. tradurre tale definizione in un algoritmo (bottom up o top down con memoization) che determina l'edit distance;

iii. trasformare l'algoritmo in modo che fornisca anche le operazioni di edit che trasformano X in Y con costo minimo;

iv. valutare la complessità dell'algoritmo.

Soluzione: La caratterizzazione ricorsiva della edit distance $e(i, j)$ tra $X^i = x_1 \dots x_i$ e $Y^j = y_1 \dots y_j$, si ottiene osservando che se la prima stringa è vuota , se la seconda è vuota devo fare la delete della prima, altrimenti, se sono entrambe non vuote,

- se $i = 0$, ovvero la prima stringa è vuota, non posso fare altro che fare l'insert della seconda;
- se $j = 0$, ovvero la seconda stringa è vuota, non posso fare altro che fare la delete della prima;
- altrimenti, se $i, j > 0$, posso
  - se $x_i = y_j$, senza alcun costo, ridurmi X_{i-1}, Y_{j-1}, altrimenti questo richiede di pagare il costo della sostituzione;
  - fare la delete di x_i
  - fare l'insert di y_j

  alla fine, terrò la meno costosa tra le tre.

In sintesi:

$$e[i,j] = \begin{cases} j & \text{se } i = 0 \text{ e } j > 0 \\ i & \text{se } j = 0 \text{ e } i > 0 \\ \min \left\{ \begin{array}{l} e[i-1,j]+1, \\ e[i,j-1]+1 \\ e[i-1,j-1] \end{array} \right\} & \text{se } i,j > 0 \text{ e } x_i = y_j \\ \min \left\{ \begin{array}{l} e[i-1,j]+1, \\ e[i,j-1]+1 \\ e[i-1,j-1]+c(x_i,y_j) \end{array} \right\} & \text{altrimenti} \end{cases}$$

che, assumendo che $c(a,a) = 0$ posso semplificare:

$$e[i,j] = \begin{cases} \max(i,j) & \text{se } i = 0 \text{ o } j = 0 \\ \min \left\{ \begin{array}{l} e[i-1,j]+1, \\ e[i,j-1]+1 \\ e[i-1,j-1]+c(x_i,y_j) \end{array} \right\} & \text{altrimenti} \end{cases}$$

Un ulteriore semplificazione deriva dall'osservazione che se $x_i = y_j$ il match è sempre la scelta più conveniente, ovvero,

$$e[i-1,j-1] \leq \min(e[i,j-1], e[i-1,j]) + 1$$

infatti, una qualunque edit sequence che transforma X^{i-1} in Y^j può trasformare X^{i-1} in Y^{j-1} con al massimo una delete in più. E similmente, una qualunque edit sequence che transforma X^i in Y^{j-1} può trasformare X^{i-1} in Y^{j-1} con al massimo una insert in più.

Quindi:

$$e[i,j] = \begin{cases} \max(i,j) & \text{se } i = 0 \text{ o } j = 0 \\ e[i-1,j-1] & \text{se } i,j > 0 \text{ e } x_i = y_j \\ \min \left\{ \begin{array}{l} e[i-1,j]+1, \\ e[i,j-1]+1 \\ e[i-1,j-1]+c(x_i,y_j) \end{array} \right\} & \text{altrimenti} \end{cases}$$

Ne segue l'algoritmo che riceve in input X, m, Y, n e la funzione c e calcola l'edit distance.

```
edit (X, m, Y, n, c)
  for i=0 to m
    e[i,0] = i

  for j=1 to n
```

```
    e[0,j] = j

  for i=1 to m
    for j=1 to n
            if X[i] = Y[j]
           e[i,j] = e[i-1,j-1]
        else
           e[i,j] = min { e[i-1,j]+1, e[i,j-1]+1, e[i-1,j-1]+c(X[i],X[j])

   return e[m,n]
```

Se vogliamo anche la sequenza di edit ottima, occorre ricordare come il minimo viene raggiunto. Questo viene fatto tramite un array s[i,j] che ricorda l'operazione (insert, delete, replace, match) che ha portato all'ottimo.

```
edit (X, m, Y, n, c)
  e[0,0] = 0

  /* keep also s[i,j] = prossima operazione da fare per trasf. X^i in Y^j
     'i' -> insert
     'd' -> delete
     'r' -> replace
     '-' -> no op (match)
  */

  for i=1 to m
    e[i,0] = i
    s[i,0] = 'd'

  for j=1 to n
    e[0,j] = j
    s[0,j] = 'i'

  for i=1 to m
    for j=1 to n
            if X[i] = Y[j]
           e[i,j] = e[i-1,j-1]
           s[i,j] = '-'
        else

           if e[i-1,j] <= e[i,j-1]
              min = e[i-1,j] + 1
              s[i,j] = 'd'
           else
              min = e[i,j-1] + 1
              s[i,j] = 'i'

           if e[i-1,j-1] + c(X[i],X[j]) < min
              min = e[i-1,j-1] + c(X[i],X[j])
              s[i,j] = 'r'

   i=n; j=m
   while (i>0) or (j>0)
      case s[i,j]:
```

```
'i':  insert Y[j]
      j--

'd':  delete X[i]
      i--

'r':  replace X[i] by Y[j]
      i--; j--

'-':  i--; j--
```

La complessità è $\Theta(mn)$.

Esercizio 30 Nello stato di Frattalia, si è appena insediato il nuovo governo, sostenuto da due partiti, Arancio e Cobalto. La coalizione Arancio-Cobalto ha sottoscritto un contratto di governo che prevede diversi punti, alcuni proposti dagli Arancio altri dai Cobalto. Tuttavia ogni punto ha un costo economico, così che non è possibile realizzarli tutti: il debito pubblico non può oltrepassare una certa soglia. Occorre quindi scegliere un sottoinsieme di punti da realizzare e tale insieme deve essere *bilanciato*, ovvero devono essere scelti in egual numero punti degli Arancio e dei Cobalto.

Si supponga che i punti proposti dal partito Arancio siano `n` con costi $a_1, \ldots, a_n$ e analogamente i punti proposti dai Cobalto siano `m` con costi $c_1, \ldots, c_m$. Fornire un algoritmo di programmazione dinamica che individui un sottoinsieme bilanciato dei punti del contratto, di dimensione massima e tale che la somma dei costi non oltrepassi un limite fissato d. Più precisamente:

i. dare una caratterizzazione ricorsiva della cardinalità massima $p_{i,j,d}$ di un insieme bilanciato di punti di programma scelti in $a_1, \ldots, a_i$, $c_1, \ldots, c_j$ con costo $\leq d$;

ii. tradurre tale definizione in un algoritmo (bottom up o top down con memoization) che determina il numero massimo;

iii. trasformare l'algoritmo in modo che fornisca anche i punti del contratto scelti, non solo il loro numero;

iv. valutare la complessità dell'algoritmo.

Esiste la possibilità di applicare un algoritmo greedy?

Soluzione: La caratterizzazione ricorsiva della cardinalità massima $p[i, j, m]$ di un insieme bilanciato di punti di programma scelti in $a_1, \ldots, a_n$, $c_1, \ldots, c_m$ con costo $\leq d$ deriva dall'osservazione che

- se $i = 0$ oppure $j = 0$, l'unico insieme bilanciato è vuoto e quindi il numero totale sarà 0.;
- altrimenti, se $i, j > 0$, possiamo escludere il punto a_i, oppure escludere il punto c_j, oppure, se il budget lo consente, includere sia a_i che c_j

In sintesi:

$$p[i,j,m] = \begin{cases} 0 & \text{if } i = 0 \text{ oppure } j = 0 \\ \max\{p[i-1,j,d],\ p[i,j-1,d]\} & \text{if } i,j > 0 \text{ e } a_i + c_j > d \\ \max\left\{ \begin{array}{l} p[i-1,j,d],\ p[i,j-1,d], \\ p[i-1,j-1,d-a_i-c_j]+2 \end{array} \right\} & \text{if } i,j > 0 \text{ e } a_i + c_j \leq m \end{cases}$$

Ne segue l'algoritmo che riceve in input gli array di punti $a[1, n]$ e $c[1, m]$, il costo massimo d e usa una matrice $P[1..n, 1..m, 0..d]$ dove $P[i, j, d']$ rappresenta il numero massimo bilanciato di punti scelti in $a_1, \ldots, a_i$, $c_1, \ldots, c_m$ con costo $\leq d'$.

```
governo (a, c, n, m, d)
  for i=1 to n
    for d' = 0 to d
    p[i,0, d'] = 0

  for j=0 to m
    for d' = 0 to d
    p[0,j, d'] = 0

  for i=1 to n
    for j = 1 to m
        for d'=1 to to d
           if p[i-1,j,d'] >= p[i,j-1,d']
              max = p[i-1,j,d']
           else
              max = p[i,j-1,d']

           if a[i]+c[j] <= d' and p[i-1,j-1,d'-a[i]-c[j]] + 2 > max
              max  = p[i-1,j-1,d'-a[i]-c[j]] + 2

           p[i,j,d']=max

   return p[n,m,d]
```

Se vogliamo anche i punti scelti, occorre ricordare dove il massimo viene raggiunto. Dato che i punti sono scelti a coppie usiamo un array S[i,j,d'] = (i',j') che da' la prima coppia di punti scelta in una soluzione ottima (o (0,0) se non ne viene scelto nessuno)

```
governo (a, c, n, m, d)
  # i punti sono scelti a coppie
  # usa quindi S[i,j,d'] = (i',j') prima coppia di punti scelta
  # in una soluzione ottima, (0,0) se non e' possibile nessuna scelta

  for i=1 to n
    for d' = 0 to d
    p[i,0, d'] = 0
    S[i,0, d'] = (0,0)

  for j=0 to m
    for d' = 0 to d
    p[0,j, d'] = 0
    S[0,j, d'] = (0,0)


  for i=1 to n
    for j = 1 to m
        for d'=1 to to d
           if p[i-1,j,d'] >= p[i,j-1,d']
              max = p[i-1,j,d']
              S[i,j,d'] = S[i-1,j,d']
           else
              max = p[i,j-1,d']
              S[i,j,d'] = S[i,j-1,d']
```

```
        if a[i]+c[j] <= d' and p[i-1,j-1,d'-a[i]-c[j]] + 2  > max
           max  = p[i-1,j-1,d'-a[i]-c[j]] + 2
           S[i,j,d'] = (i,j)

        p[i,j,d']=max


i=n, j=m, d'=d
while (i>0 and j>0)
      i, j = S[i,j,d']

      if (i>0)
          print "a[", i, "], "c[", j, "]"
          d' = d' - a[i] - c[j]

     i--; j--
```

La complessità è $\Theta(nmd)$.

È facile vedere che si può utilizzare un algoritmo greedy che sceglie ad ogni passo coppie di punti di costo minimo. In primo luogo, si osserva, come fatto sopra, che vale la proprietà della sosttruttura ottima, ovvero una soluzione ottima per il problema consiste in una scelta di una coppia di punti, uno Arancio e uno Cobalto, diciamo a_i e c_j e un ottimo del sottoproblema con punti $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$ e $b_1, \ldots, b_{j-1}, a_{j+1}, \ldots, b_m$ con limite di spesa $d - a_i - b_j$.

Inoltre, vale la scelta greedy che cosiste nello scegliere i punti Arancio e Cobalto di costo minimo (se la somma è inferiore a d), altrimenti, nessun punto.

Ovvero, assumendo che gli array siano ordinati in modo decrescente

$$p[i,j,m] = \begin{cases} 0 & \text{if } i = 0 \text{ oppure } j = 0 \text{ oppure } a_i + c_j > d \\ p[i-1, j-1, d - a_i - c_j] + 2 & \text{altrimenti} \end{cases}$$

Questo permette di usare l'algoritmo che ordina le gli array dei punti per costo crescente, con complessità $n \log n + m \log m$. Quindi si costruisce la soluzione prendendo, ad ogni iterazione, i punti di costo minimo, finchè possibile, in tempo $O(\min(m, n))$. La complessità è dunque $\Theta(n \log n + m \log m)$, quasi invariabilmente preferibile a $\Theta(nmd)$.

Esercizio 31 Dare un algoritmo per individuare, all'interno di una stringa $a_1 \ldots a_n$ una sottosequenza (quindi una sequenza di caratteri possibilmente non consecutivi) palindroma di lunghezza massima. Ad esempio, nella stringa "*corollario*" la sottosequenza palindroma di lunghezza massima è "*orllro*". Più precisamente:

i. dare una caratterizzazione ricorsiva della lunghezza massima $l_{i,j}$ di una sottosequenza palindroma di $a_i \ldots a_j$;

ii. tradurre tale definizione in un algoritmo (bottom up o top down con memoization) che determina la lunghezza massima;

iii. trasformare l'algoritmo in modo che fornisca anche la sottosequenza, non solo la sua lunghezza;

iv. valutare la complessità dell'algoritmo.

Soluzione: La caratterizzazione ricorsiva della lunghezza massima $l_{i,j}$ di una sottosequenza palindroma di $a_{i,j} = a_i \ldots a_j$ deriva dall'osservazione che

- se $i > j$, quindi $a_{i,j} = \varepsilon$ è la stringa vuota, dato che la stringa vuota è palindroma $l_{i,j} = 0$;
- se $i = j$ quindi $a_{i,j}$ è la stringa di un solo carattere, che è palindroma, vale $l_{i,j} = 1$;
- altrimenti, se $i < j$ quindi $a_{i,j}$ consta di almeno due caratteri, distinguiamo due sottocasi:
  - se $a_i = a_j$, necessariamente a_i e a_j fanno parte di una sottosequenza palindroma massima di $a_i \dots a_j$. Infatti, supponiamo che una sottosequenza palindroma massima ω non comprenda a_i e a_j. Si osserva che necessariamente inizia per a_i oppure termina per a_j, altrimenti, se così non fosse, $a_i\omega a_j$ sarebbe una sottosequenza palindroma di lunghezza superiore. La seconda osservazione è che la lunghezza $|\omega| \geq 2$, altrimenti $a_i a_j$ sarebbe una sottosequenza palindroma più lunga di ω. Supponiamo dunque, senza perdita di generalità (l'altro caso è simmetrico) che $\omega = a_i\omega' a_k$. Dato che $a_i = a_k$, possiamo ottenere una sottosequenza della stessa lunghezza di ω (dunque ottima), sostituendo a_k con a_j, ovvero $a_i\omega' a_j$. Questo conclude la prova.
    Abbiamo dunque che $l_{i,j} = 2 + l_{i+1,j-1}$.
  - se invece $a_i \neq a_j$, una sottosequenza palindroma non può comprendere entrambi gli estremi e quindi ci si riduce ai sottoproblemi $a_{i+1,j}$ e $a_{i,j-1}$ e si prende il massimo.

In sintesi:

$$l_{i,j} = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ l_{i+1,j-1} + 2 & \text{if } a_i = a_j \\ \max\{l_{i,j-1}, l_{i+1,j}\} & \text{if } a_i \neq a_j \end{cases}$$

Ne segue l'algoritmo che riceve in input la stringa, nella forma di un array di caratteri $A[1, n]$ e usa una matrice $L[1..n, 0..n]$ dove $L[i, j]$ rappresenta la lunghezza di una sottostringa palindroma in $A[i, j]$

```
palindromeSeq (A, n)
  for i = 1 to n
    L[i,i-1] = 0
    L[i,i] = 1

  for len = 2 to n
    for i = 1 to n-len+1
      j = i + len - 1
      if (A[i] = A[j])
         L[i,j] = L[i+1,j-1] + 2
      else
         L[i,j] = max (L[i,j-1], L[i+1,j])

  return L[1,n]
```

Se vogliamo anche la sottosequenza, occorre ricordare il massimo e dove viene raggiunto, lo facciamo mediante un array $P[1..n, 1..n]$ tale che $P[i, j]$ contenga un'indicazione di come si ottiene la sottosequenza palindroma più lunga di $a_{i,j}$:

```
palindrome(A, n)
  for i = 1 to n
    L[i,i-1] = 0
    L[i,i] = 1

    # P[i,j] = r if P[i,j] = P[i+1,j]
    #        = l if P[i,j] = P[i,j-1]
```

```
    #        = c if i<j and A[i], A[j] chosen and P[i,j] = P[i+1,j-1]
    # when i=j we know that A[i] is chosen

  for len = 2 to n
    for i = 1 to n-len+1
      j = i + len - 1
      if (A[i] = A[j])
         L[i,j] = L[i+1,j-1] + 2
         P[i,j] = 'c'
      else
         if L[i,j-1] >= L[i+1,j]
            L[i,j] = L[i,j-1]
            P[i,j] = 'l'
         else
            L[i,j] = L[i+1,j]
            P[i,j] = 'r'

   return getPal(A,1,n,L,P)   #  massima sottosequenza palindroma di A[1,n]


#  massima sottosequenza palindroma di A[i,j]
getPal(A,i,j L,P)
    if (i < j)
        if P[i][j] == 'c'
            return A[i] + getPal(A,i+1,j-1, L,P) + A[j]
        elif P[i][j] == 'l'
            return getPal(A,i,j-1, L,P)
        else # P[i][j] == 'r'
            return getPal(A,i+1,j, L,P)
    elif i==j:
        return A[i]
    else # i> j
        return ''
```

In realtà si può fare a meno di utilizzare una struttura ausiliaria per memorizzare le scelte ottime, dato che sono determinate dalla forma di L:

```
palindrome(A, n)
  for i = 1 to n
    L[i,i-1] = 0
    L[i,i] = 1

  for len = 2 to n
    for i = 1 to n-len+1
      j = i + len - 1
      if (A[i] = A[j])
         L[i,j] = L[i+1,j-1] + 2
      else
         if L[i,j-1] >= L[i+1,j]
            L[i,j] = L[i,j-1]
         else
            L[i,j] = L[i+1,j]

   return getPal(A,1,n,L)   #  massima sottosequenza palindroma di A[1,n]
```

```
#  massima sottosequenza palindroma di A[i,j]
getPal(A,i,j L)
    if L[i][j] == 1
        return A[i]
    elif L[i][j] > 1
        if L[i][j] == L[i+1][j]:
            return getPal(A,i+1,j,L)
        elif L[i][j] == L[i][j-1]:
            return getPal(A,i,j-1,L)
        else # L[i][j] == L[i+1][j-1]+2 and necessarily the two chars have been chosen
            return A[i] + getPal(A,i+1,j-1,L) + A[j]
    else # when L[i][j]=0
        return ''
```

La complessità è $\Theta(n^2)$.

Esercizio 32 Realizzare, con tecniche di programmazione dinamica, un algoritmo che dato un array $A[1..n]$, non vuoto, trova un sottoarray non vuoto di somma massima, ovvero due indici i, j con $1 \leq i \leq j \leq n$ tali che $A[i] + A[i+1] + \ldots + A[j]$ sia massima. Ad esempio per [-10, 4, 1, -1, 2, -1] il sottoarray di somma massima è [4, 1, -1, 2]. Più precisamente:

i. indicato con l_j la somma massima di una sottoarray di $A[1..n]$ che termini con $A[j]$ (quindi del tipo $A[i..j]$), darne una caratterizzazione ricorsiva;

ii. tradurre tale definizione in un algoritmo (bottom up o top down con memoization) che determina la somma massima;

iii. trasformare l'algoritmo in modo che fornisca anche la sottosstringa, non solo la sua somma;

iv. valutare la complessità dell'algoritmo.

Nota: La soluzione proposta deve articolarsi nei passi sopra descritti.

Soluzione: L'osservazione fondamentale è che un sottoarray con somma massima di $A[1..n]$ che termini in $A[j]$ sarà, un sottoarray di somma massima che termina in $A[j-1]$, concatenato con $A[j]$, se il primo ha somma positiva, altrimenti semplicemente $A[j..j]$. In simboli, per $j = 1, \ldots, n$

$$l_j = \begin{cases} 1 & \text{se } j = 1 \text{ o } l_{j-1} \leq 0 \\ l_{j-1} + 1 & \text{altrimenti, ovvero se } j > 1 \text{ e } l_{j-1} > 0 \end{cases}$$

Il sottoarray con somma massima terminerà in qualche j e quindi sarà poi sufficiente massimizzare i valori trovati.

Ne segue l'algoritmo che riceve in input l'array $A[1, n]$ e usa una matrice $L[1..n]$ dove $L[j]$ rappresenta la somma di una sottoarray di lunghezza massima che termina in $A[j]$.

```
maxsum (A, n)
    L[1] = A[1]

  for j=2 to n
      if (L[j-1] > 0)
              L[j] = L[j-1] + A[j]
```

```
      else
         L[j] = A[j]

  max = A[1]
  for j=2 to n
      if L[j]>max
         max = L[j]

  return max
```

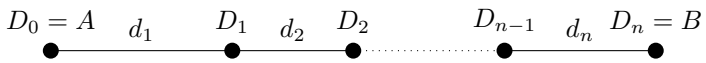
Se vogliamo anche il sottoarray, occorre ricordare il massimo e dove viene raggiunto, lo facciamo mediante un array $S[1..n]$ tale che $S[j]$ contenga l'indice dal quale inizia il sottoarray massima che termina con $A[j]$.

```
maxsum (A, n)
  L[1] = A[1]
  S[1] = 1

  for j=2 to n
      if (L[j-1] > 0)
         L[j] = L[j-1] + A[j]
         S[j] = S[j-1]
      else
         L[j] = A[j]
         S[j] = j

  max = 1
  for j=2 to n
      if L[j]>max
         max = j

  # returns the first and last index of the substring
  return S[max], max
```

La complessità è $\Theta(n)$.

9 Greedy

Esercizio 33 Si supponga di voler viaggiare dalla città A alla città B con un'auto che ha un'autonomia pari a d km. Lungo il percorso si trovano $n-1$ distributori $D_1, \ldots, D_{n-1}$, a distanze di $d_1, \ldots, d_n$ km ($d_i \leq d$) come indicato in figura



L'auto ha inizialmente il serbatoio pieno e l'obiettivo è quello di percorrere il viaggio da A a B, minimizzando il numero di soste ai distributori per il rifornimento.

i. Introdurre la nozione di soluzione per il problema e di costo della una soluzione. Mostrare che vale la proprietà della sottostruttura ottima e individuare una scelta che gode della proprietà della scelta greedy.

ii. Sulla base della scelta greedy individuata al passo precedente, fornire un algoritmo greedy `stop(d,n)` che dato in input l'array delle distanze `d[1..n]` restituisce una soluzione ottima.

iii. Valutare la complessità dell'algoritmo.

Soluzione:

Soluzione. In generale, la specifica del problema consiste in una sequenza di soste possibili $D_0(= A), D_1, \ldots, D_n(= B)$. Per una coppia di soste D_i, D_j con $i \leq j$ poniamo $d_{i,j} = \sum_{h=i+1}^{j} d_h$. Quindi una soluzione del problema è una sottosequenza di soste che porta dal punto iniziale al punto finale, senza percorrere tratti di lungheza maggiore di d ovvero:

$$S = D_{i_0} \ldots D_{i_k}$$

(con $i_0 < i_1 < \ldots < i_k$) tali che $D_{i_0} = A$ e $D_{i_k} = B$, per ogni $j \in \{0, \ldots, k-1\}$ vale $d_{i_j i_{j+1}} \leq d$. Il costo $c(S)$ è il numero $k-1$ di soste.

Sottostruttura ottima. Sia $S = D_{i_0} \ldots D_{i_k}$ una soluzione ottima per il problema $D_0, D_1, \ldots, D_n$. Se consideriamo il sottoproblema di andare da D_{i_1} a D_n, ovvero $D_{i_1}, D_{i_1+1} \ldots D_n$, allora $S_1 = D_{i_1}, \ldots, D_{i_k}$ è una soluzione ottima. Infatti, è chiaramente una soluzione. Inoltre, se ci fosse una soluzione migliore S'_1, con un numero inferiore di soste per il sottoproblema, ovvero $c(S'_1) < c(S_1)$, aggiungendo la sosta D_{i_0} otterremmo una soluzione S' migliore di S per il problema originale dato che $c(S') = c(S'_1) + 1 < c(S_1) + 1 = c(S)$.

Scelta greedy. Per il problema $D_0, D_1, \ldots, D_n$, si fissa necessariamente $i_0 = 0$, e la scelta greedy consiste nel raggiungere la sosta più lontana a distanza minore o uguale di d, ovvero definire $i_1 = \max\{j \mid d_{0,j} \leq d\}$.

Data una soluzione ottima per il problema $S = D_{j_0} \ldots D_{j_k}$, certamente $j_0 = 0 = i_0$ e $j_1 \leq i_1$. Quindi è immediato verificare che anche $S' = D_{j_0} D_{i_1} D_{j_2} \ldots D_{j_k}$ è una soluzione per il problema $D_0, D_1, \ldots, D_n$, ed è ottima, dato che $c(S') = c(S)$. (Si noti che, più precisamente, in prima istanza si potrebbe pensare che D_{i_1} potesse essere anche oltre D_{j_2}, ma questo darebbe una soluzione migliore di quella ottima, portando ad un assurdo).

Algoritmo. Ne segue l'algoritmo che riceve in la sequenza di distanze nella forma di un array $d[1, n]$ e restituisce un array $S[1..n-1]$ con le soste scelte (la prima e l'ultima sono scelte sempre, non serve indicarlo).

```
stop(d, n)
  dist = d[1]            // distanza percorsa
  for i=2 to n
    if dist + d[i] > d
       S[i-1]=1
       dist=d[i]
    else
       S[i-1] = 0

  return S
```

La complessità è $\Theta(n)$.

Esercizio 34 L'ufficio postale offre un servizio di ritiro pacchi in sede su prenotazione. Il destinatario, avvisato della presenza del pacco, deve comunicare l'orario preciso al quale si recherà allo

sportello. Sapendo che gli impiegati dedicano a questa mansione turni di un'ora, con inizio in un momento qualsiasi, si chiede di scrivere un algoritmo che individui l'insieme minimo di turni di un'ora sufficienti a soddisfare tutte le richieste. Più in dettaglio, data una sequenza $\vec{r} = r_1, \ldots, r_n$ di richieste, dove r_i è l'orario della i-ma prenotazione, si vuole determinare una sequenza di turni $\vec{t} = t_1, \ldots, t_k$, con t_j orario di inizio del j-mo turno, che abbia dimensione minima e tale che i turni coprano tutte le richieste.

i. Formalizzare la nozione di soluzione per il problema e il relativo costo. Mostrare che vale la proprietà della sottostruttura ottima e individuare una scelta che gode della proprietà della scelta greedy.

ii. Sulla base della scelta greedy individuata al passo precedente, fornire un algoritmo greedy `time(R,n)` che dato in input l'array delle richieste `r[1..n]` restituisce una soluzione ottima.

iii. Valutare la complessità dell'algoritmo.

Soluzione: Una soluzione è una sequenza $\vec{t} = t_1, \ldots, t_k$ tale che per ogni $i = 1, \ldots, n$ l'istante $r_i \in I(t_j)$ per qualche $j = 1, \ldots, k$, dove $I(t_j)$ indica l'intervallo di un'ora con inizio in t_j, Il costo è k.

Sottostruttura ottima Vale la sottostruttura ottima. Infatti se $\vec{t} = t_1 q \ldots, t_k$ è ottima per le richieste $\vec{r} = r_1, \ldots, r_n$ allora $t_2, \ldots, t_k$ è ottima per il sottoproblema $\vec{r'}$ che comprende le richieste r_i tali che $r_i \notin I(t_1)$. Infatti, se ci fosse un insieme di turni di dimensione minore di $k-1$ per servire le richieste in $\vec{r'}$, aggiungendo t_1 otterremmo una soluzione del problema originale $\vec{r}$ migliore di $\vec{t}$

Scelta greedy Assumiamo che la lista di richieste $\vec{r} = r_1, \ldots, r_n$ sia ordinata in modo crescente. La scelta greedy consiste nel considerare come primo turno $t_1 = r_1$.

Esiste sempre una soluzione ottima che la contiene. Infatti se $\vec{t'} = t'_1 \ldots t'_k$ è una qualsiasi soluzione ottima e supponiamo che i turni siano ordinati anch'essi in senso crescente, allora certamente $t'_1 \leq r_1$, dato che la prima richiesta deve essere servita. Quindi se sostituiamo t'_1 con $t_1 = r_1$, otteniamo una nuova soluzione (tutte le richieste servite da t'_1 sono anche servite da t_1!), anch'essa ottima.

Ne segue l'algoritmo che riceve in input l'array delle richieste $r[1..n]$ (che si assume non vuoto), e fornisce in uscita $t[1, k]$.

```
time(r,n)
  t[1] = r[1]
  turni=1
  for i = 2 to n
    if t[turni] < r[i]
       turni++
       t[turni] = r[i]
  return t
```

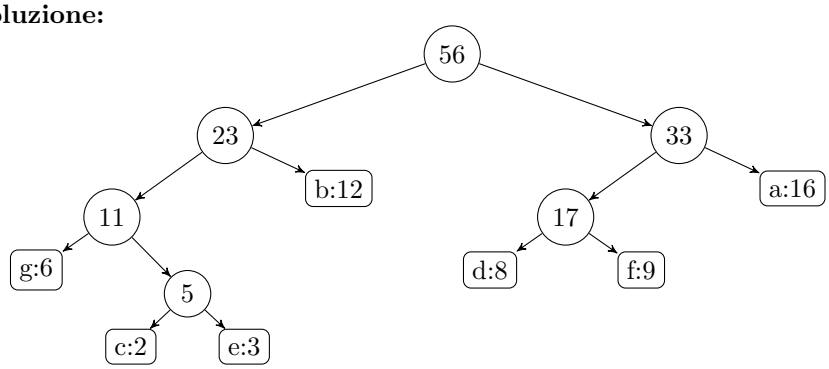
La complessità è $\Theta(n)$.

Domanda 39 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
16	12	2	8	3	9	6

Spiegare il processo di costruzione del codice.

Soluzione:

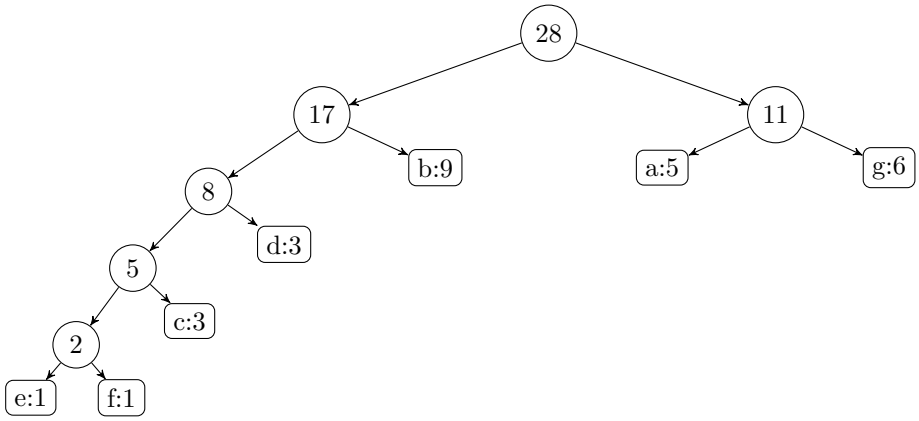


Domanda 40 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
5	9	3	3	1	1	6

Spiegare il processo di costruzione del codice.

Soluzione:

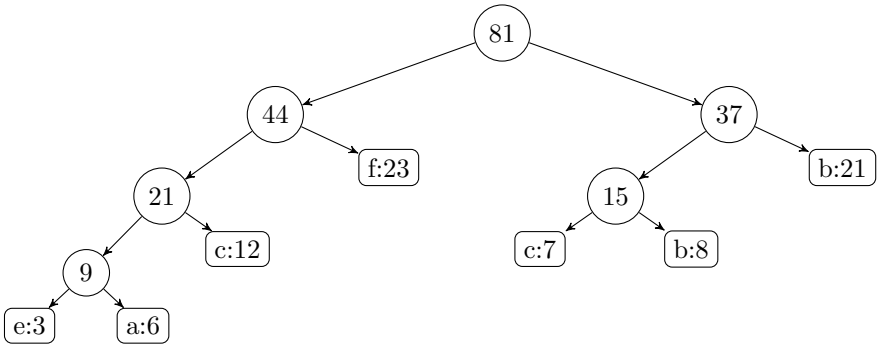


Domanda 41 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
6	21	12	8	3	23	8

Spiegare il processo di costruzione del codice.

Soluzione:

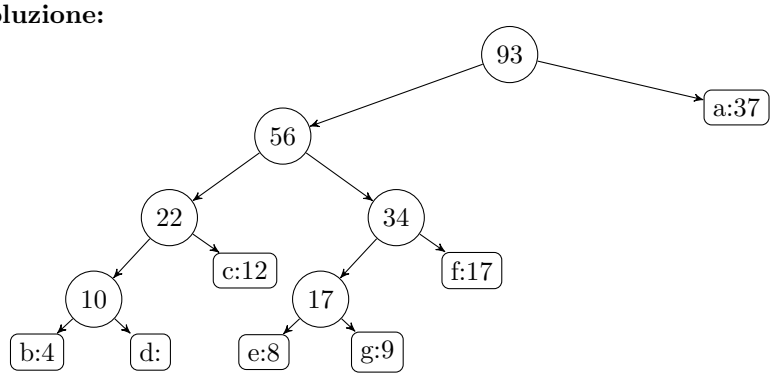


Domanda 42 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
37	4	12	6	9	17	8

Spiegare il processo di costruzione del codice.

Soluzione:

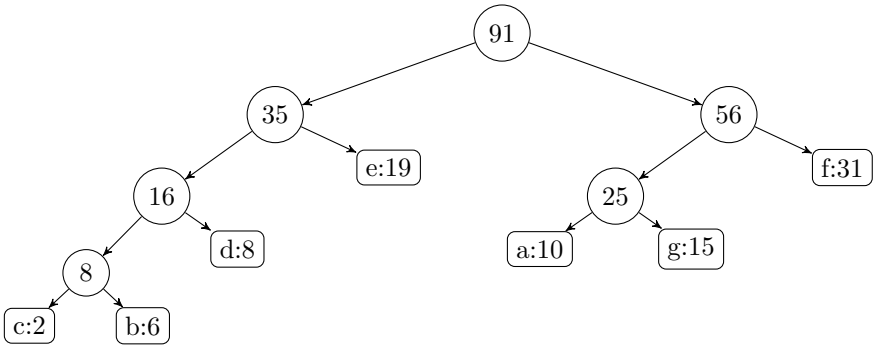


Domanda 43 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
10	6	2	8	19	31	15

Spiegare il processo di costruzione del codice.

Soluzione:

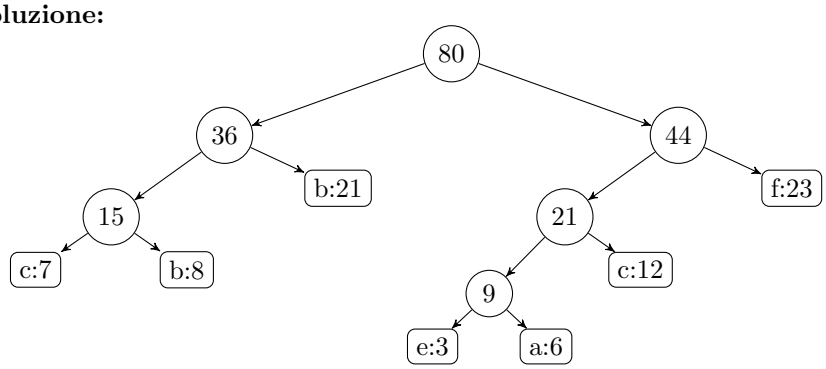


Domanda 44 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
3	8	7	12	6	23	21

Spiegare il processo di costruzione del codice.

Soluzione:

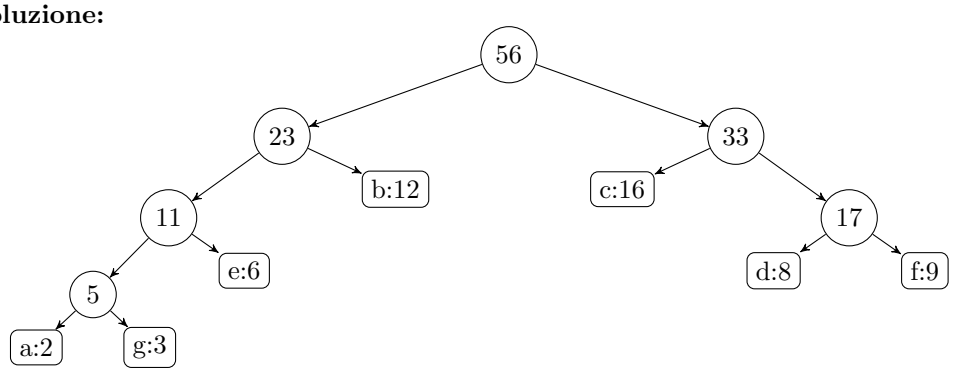


Domanda 45 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
2	12	16	8	6	9	3

Spiegare il processo di costruzione del codice.

Soluzione:



10 Analisi Ammortizzata

Esercizio 35 Progettare una struttura dati per la gestione di un insieme dinamico di interi, con operazioni

- *New*(S) crea un insieme vuoto;
- *Ins*(S, x) inserisce l'elemento x nell'insieme S;
- *Half*(S) cancella da S i $\lceil |S|/2 \rceil$ elementi più piccoli.

Si richiede che una qualsiasi sequenza di n operazioni venga eseguita in tempo $O(n \log n)$.

i. Specificare le strutture dati di supporto utilite e lo pseudo-codice delle operazioni suddette (questo può ridursi ad una chiamata di un'operazione della struttura scelta).

ii. Dimostrare, mediante un'analisi ammortizzata della complessità, che una sequenza di n operazioni costa $O(n \log n)$.

Soluzione: La struttura S può essere semplicemente un min-heap, con operazioni

```
New(S)
    crea un min-heap S
    S.size = 0

Ins(S,x)
    Heap-Insert(S,x)

Half(S,x)
    k = S.heapsize
    for i = 1 to k/2
        Heap-Extract-Min(S)
```

L'idea è che ogni operazione di inserimento paga in anticipo il costo dell'estrazione dell'elemento ($\log |S|$). Questo da' una complessità ammortizzata di $2 \log |S|$ per l'inserimento e complessità costante per *Half*, dato che le estrazioni sono già pagate. Si noti che in realtà un elemento non paga esattamente per la sua estrazione, ma per quella di un qualche altro elemento, a seconda dell'ordine relativo. Si può formalizzare con una funzione potenziale che può essere

$$\Phi(S) = \Sigma_{j=1}^{|S|} \log j = \log |S|!.$$

In questo modo, il costo delle operazioni è

	c	$\Delta\Phi$	$\hat{c}$
New	1	0	1
Ins	$1+ \log \lvert S\rvert$	$\log \lvert S\rvert$	$1+ 2 \log \lvert S\rvert$
Extract	$1+ \log \lvert S\rvert$	$- \log \lvert S\rvert$	1
Half	$1 + \Sigma_{j=1}^{\lvert S\rvert/2} logj$	$-\Sigma_{j=1}^{\lvert S\rvert/2} logj$	1

Quando la struttura è vuota, $\Phi(S) = 0$ e, per ogni S, vale $\Phi(S) \geq 0$. Pertanto per una sequenza di n operazioni, certamente $\Phi(S_n) - \Phi(S_0) \geq 0$, per cui i costi ammortizzati forniscono un upperbound per i costi reali. Una sequenza di n operazioni costa al più $\Sigma_{j=1}^{n} \log j \leq n \log n$, come desiderato.

Esercizio 36 Si consideri una struttura che chiamiamo `OderedStack` con le seguenti operazioni:

- `OEmpty(S)`: Ritorna un booleano che dice se la struttura è vuota
- `OPop(S)`: Estrae l'ultimo elemento inserito
- `OTop(S)`: Ritorna il valore dell'ultimo elemento inserito
- `OPush(S,x)`: Inserisce nello stack x, eliminando prima ogni elemento che sia maggiore di x.

Utilizzando una struttura dati stack, fornire una implementazione della suddetta struttura ovvero

i. Specificare come è definita la struttura dati e fornire lo pseudocodice dei metodi elencati.

ii. Mostrare che una sequenza di n operazioni, a partire da struttura vuota, ha costo ammortizzato $O(n)$ (e quindi ogni singola operazione ha costo ammortizzato $O(1)$).

Soluzione:

i. L'implementazione è la seguente:

```
OStack = Stack

OEmpty(S)
   return Empty(S)

OPop(S)
   return Pop(S)

OTop(S)
   return Top(S)

OPush(S,x)
   while not Empty(S) and (x < Top(S))
            Pop(S)
   Push(S)
```

ii. Per l'analisi ammortizzata si può usare il metodo del potenziale con funzione $\Phi(S) = |S|$.

In questo modo, il costo delle operazioni è

	c	$\Delta\Phi$	$\hat{c}$
`Empty`	1	0	1
`Top`	1	0	1
`Pop`	1	-1	0
`Push`	$1+k$	$-k+1$	2

Quando la struttura è vuota, $\Phi(S) = 0$ e, per ogni S, vale $\Phi(S) \geq 0$. Pertanto per una sequenza di n operazioni, certamente $\Phi(S_n) - \Phi(S_0) \geq 0$, per cui i costi ammortizzati forniscono un upperbound per i costi reali. Una sequenza di n operazioni costa al più n, come desiderato.

Esercizio 37 Progettare una struttura dati per la gestione di un insieme dinamico di interi, con operazioni

- $New(S)$ crea un insieme vuoto;

- *Ins*(S, x) inserisce l'elemento x nell'insieme S;
- *Half*(S) cancella da S i $\lceil |S|/2 \rceil$ elementi più grandi.

Si richiede che una qualsiasi sequenza di n operazioni venga eseguita in tempo $O(n \log n)$.

i. Specificare le strutture dati di supporto utilite e lo pseudo-codice delle operazioni suddette (questo può ridursi ad una chiamata di un'operazione della struttura scelta).

ii. Dimostrare, mediante un'analisi ammortizzata della complessità, che una sequenza di n operazioni costa $O(n \log n)$.

Soluzione: La struttura S può essere semplicemente un min-heap, con operazioni

```
New(S)
    crea un min-heap S
    S.size = 0

Ins(S,x)
    Heap-Insert(S,x)

Half(S,x)
    k = S.heapsize
    for i = 1 to k/2
        Heap-Extract-Min(S)
```

L'idea è che ogni operazione di inserimento paga in anticipo il costo dell'estrazione dell'elemento ($\log |S|$). Questo da' una complessità ammortizzata di $2 \log |S|$ per l'inserimento e complessità costante per *Half*, dato che le estrazioni sono già pagate. Si noti che in realtà un elemento non paga esattamente per la sua estrazione, ma per quella di un qualche altro elemento, a seconda dell'ordine relativo. Si può formalizzare con una funzione potenziale che può essere

$$\Phi(S) = \Sigma_{j=1}^{|S|} \log j = \log(|S|!).$$

In questo modo, il costo delle operazioni è

	c	$\Delta\Phi$	$\hat{c}$
New	1	0	1
Ins	$1+ \log \lvert S\rvert$	$\log \lvert S\rvert$	$1+ 2 \log \lvert S\rvert$
Extract	$1+ \log \lvert S\rvert$	$- \log \lvert S\rvert$	1
Half	$1 + \Sigma_{j=1}^{\lvert S\rvert/2} logj$	$-\Sigma_{j=1}^{\lvert S\rvert/2} logj$	1

Quando la struttura è vuota, $\Phi(S) = 0$ e, per ogni S, vale $\Phi(S) \geq 0$. Pertanto per una sequenza di n operazioni, certamente $\Phi(S_n) - \Phi(S_0) \geq 0$, per cui i costi ammortizzati forniscono un upperbound per i costi reali. Una sequenza di n operazioni costa al più $\Sigma_{j=1}^{n} \log j \leq n \log n$, come desiderato.

11 B-Alberi

Domanda 46 Dare la definizione di B-albero. Qual è la minima altezza di un B-albero con grado minimo t contenente n chiavi? Motivare le risposte.

Soluzione: Un B-albero di grado minimo t è un albero T, con radice $T.root$,tale che:

1. ogni nodo x ha attributi $x.n$ (numero di chiavi in x), $x.key_1 \leq \ldots x.key_{x.n}$ (le $x.n$ chiavi ordinate), $x.leaf$ (boleano che dice se il nodo è foglia)
2. se x non è foglia ha $x.n + 1$ figli, puntati da $x.c_1, \ldots, x.c_{x.n+1}$
3. le chiavi $x.key_1 \leq \ldots x.key_{x.n}$ di un nodo interno separano le chiavi dei sottoalberi, ovvero se $k_i i$ è una chiave qualsiasi del sottoalbero di radice $x.c_i$ allora

   $$k_1 \leq x.key_1 \leq k_2 \leq \ldots \leq k_{x.n} \leq x.key_{x.n} \leq k_{x.n+1}$$

4. le foglie sono tutte alla stessa profondità h, detta altezza del B-albero
5. Vi sono limiti inferiori e superiori al numero di chiavi in un nodo
   - ogni nodo non radice, ha almeno $t - 1$ chiavi (quindi, se interno, ha almeno t figli)
   - se l'albero non è vuoto la radice contiene almeno una chiave (quindi se non foglia, ha almeno due figli)
   - ogni nodo ha al più $2t - 1$ chiavi (quindi al più $2t$ figli, se non foglia)

L'altezza minimima del B-albero si ha quando ogni nodo ha grado massimo. Vale:

$$\begin{aligned} n &= 2t - 1 + 2t(2t-1) + (2t)^2(2t-1) + \ldots (2t)^h(2t-1) \\ &= (2t-1)\sum_{i=0}^{n}(2t)^i \\ &= (2t-1)\frac{(2t)^{h+1}-1}{2t-1} \\ &= (2t)^{h+1} - 1 \end{aligned}$$

quindi, $h = (\log_{2t}(n+1)) - 1$