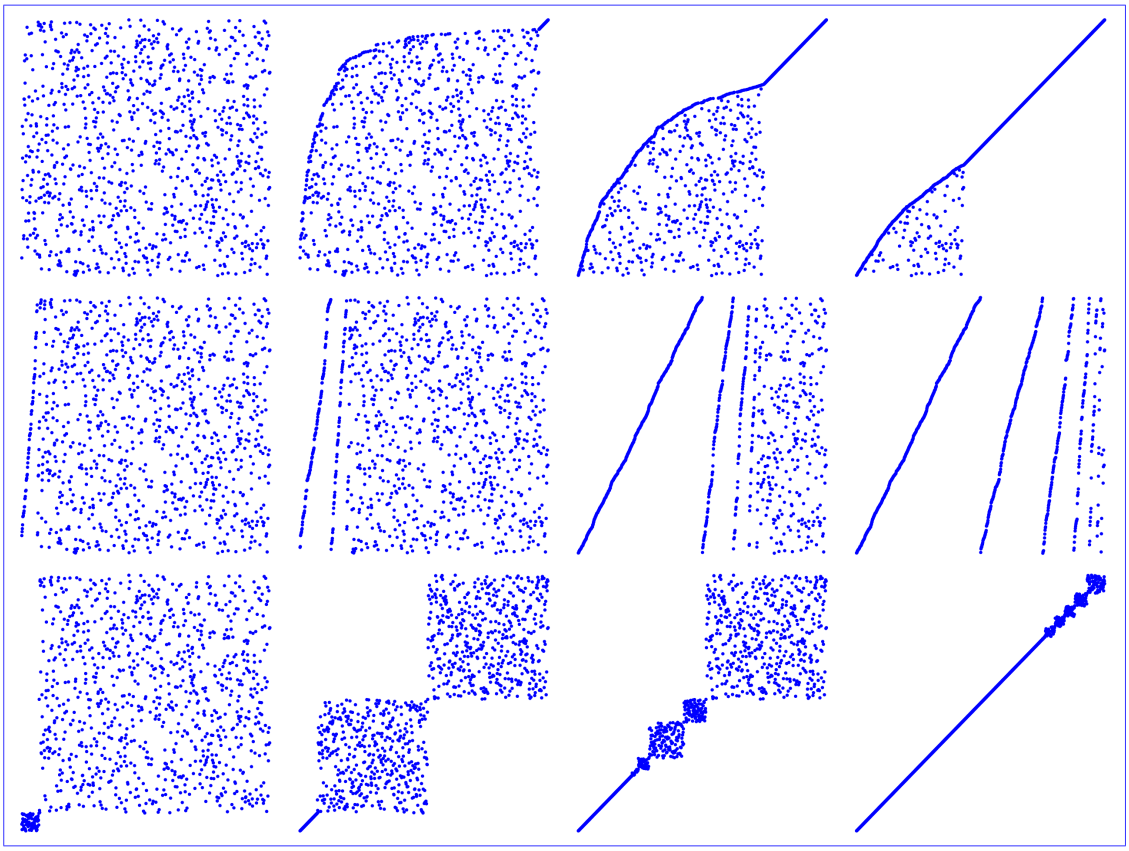
Esercizi di Algoritmi e Strutture Dati



Moreno Marzolla
http://www.moreno.marzolla.name/

Ultimo Aggiornamento: 7 novembre 2013

Commenti
Suggerimenti e correzioni sono benvenuti: scrivetemi all'indirizzo moreno.marzolla@unibo.it.

Ultima versione
La versione più recente di questo documento (in formato PDF e LibreOffice) si trova all'indirizzo http://www.moreno.marzolla.name/

Immagine di copertina
Ciascun riquadro rappresenta il contenuto di un vettore che viene ordinato con diversi algoritmi di ordinamento. Il vettore è inizializzato con una permutazione casuale degli interi compresi tra 1 e 5000. Le immagini della prima riga si riferiscono all'algoritmo BubbleSort, quelli della seconda riga all'algoritmo MergeSort, e quelli della terza riga all'algoritmo QuickSort.

Ultimo aggiornamento
7 novembre 2013

Indice

Introduzione

Questo documento raccoglie alcuni esercizi proposti a partire dall'Anno Accademico 2009/2010 nell'ambito del corso di Algoritmi e Strutture Dati, corso di laurea in Informatica per il Management, Università di Bologna.
Gli esercizi sono raggruppati per argomento; all'interno dello stesso argomento vengono presentati senza alcun ordine particolare. Alcuni degli esercizi sono tratti da prove d'esame, altri sono stati svolti a lezione. Alcuni esercizi sono tratti da Camil Demetrescu, Irene Finocchi, Giuseppe F. Italiano, *Algoritmi e strutture dati 2/ed*, McGraw-Hill, 2008, ISBN: 978 88 386 64687, o da corsi di Algoritmi svolti in altri Atenei. Naturalmente tutti gli errori sono da attribuire esclusivamente a me.
Gli algoritmi vengono presentati mediante pseudocodice, utilizzando una sintassi ispirata al linguaggio Pascal fortemente semplificato. La notazione usata dovrebbe risultare immediatamente comprensibile a chiunque sia familiare con un linguaggio di programmazione imperativo come il C/C++ o Java.

Avvertenza: questo documento è da considerarsi "work in progress", ed è soggetto a modifiche e correzioni frequenti. In questa fase vengono scoperti errori e/o imprecisioni di varia natura, per cui si raccomanda la massima cautela nell'uso del materiale.

La versione più recente è disponibile all'indirizzo http://www.moreno.marzolla.name/ . Oltre alla versione PDF, il sito ospita anche la versione editabile in formato OpenOffice/LibreOffice, in modo che chi desidera personalizzare questo documento possa farlo. Ricordo che ogni nuova versione deve comunque essere resa disponibile con licenza Creative Commons Attribution-ShareAlike 3.0 Unported (per i dettagli si veda http://creativecommons.org/licenses/by-sa/3.0/).

Bologna, 7 novembre 2013

Capitolo 1: Notazioni Asintotiche

Esercizio 1.1

Sia $f(n) = n(n+1)/2$. Utilizzando la definizione di O( ), dimostrare o confutare le seguenti affermazioni:

1. $f(n) = O(n)$
2. $f(n) = O(n^2)$

Richiamo teorico. Date due funzioni costo $f(n)$ e $g(n)$, diciamo che $f(n) = O(g(n))$ se esistono costanti $c>0$ e $n_0 \geq 0$ tali che $f(n) \leq cg(n)$ per ogni $n \geq n_0$

Soluzione. Proviamo a dimostrare se $f(n) = O(n)$. Per definizione, dobbiamo trovare una costante positiva $c>0$ tale che per ogni $n \geq n_0$ si abbia $f(n) \leq cn$. Proviamo a sviluppare la disuguaglianza:

$$\frac{n(n+1)}{2} \leq cn$$
$$\frac{n+1}{2} \leq c$$

Osserviamo che non esiste nessuna costante c che soddisfi la relazione precedente per ogni n sufficientemente grande, quindi l'affermazione $f(n) = O(n)$ è falsa.

Proviamo ora a verificare se $f(n) = O(n^2)$. Come prima, cerchiamo una costante $c>0$ per cui $f(n) \leq cn^2$ per ogni $n \geq n_0$.

$$\frac{n(n+1)}{2} \leq cn^2$$
$$\frac{n^2+n}{2n^2} \leq c$$
$$\frac{1}{2} + \frac{1}{2n} \leq c$$

Consideriamo l'ultima disuguaglianza. Per ogni $n \geq 1$ possiamo scrivere:

$$\frac{1}{2} + \frac{1}{2n} \leq \frac{1}{2} + \frac{1}{2} = c$$

per cui scegliendo $c=1$ e $n_0=1$ la relazione è verificata per ogni $n \geq n_0$. Quindi l'affermazione $f(n) = O(n^2)$ è vera.

Esercizio 1.2

Si consideri la funzione FUN(n), con $n \geq 1$ intero, definita dal seguente algoritmo ricorsivo:

```
algoritmo FUN( int n ) → int
   if (n ≤ 2) then
        return n;
   else
        return FUN(n-1) - 2*FUN(n-2);
   endif
```

1. Determinare un limite inferiore sufficientemente accurato del tempo di esecuzione $T(n)$
2. Determinare un limite superiore sufficientemente accurato del tempo di esecuzione $T(n)$

Soluzione. Il tempo di esecuzione $T(n)$ può essere calcolato come soluzione della seguente equazione di ricorrenza:

$$T(n)=\begin{cases} c_1 & \text{se } n \le 2 \\ T(n-1)+T(n-2)+c_2 & \text{altrimenti} \end{cases}$$

con c_1 e c_2 opportune costanti positive. Per studiare il valore di $T(n)$ è importante osservare che la funzione $T(n)$ è monotona crescente, ossia risulta $T(n) \geq T(n-1)$. Da questa osservazione possiamo scrivere:

$$\begin{aligned} T(n) &= T(n-1)+T(n-2)+c_2 \\ &\le 2\,T(n-1)+c_2 \\ &\le 4\,T(n-2)+2\,c_2+c_2 \\ &\le 8\,T(n-3)+4c_2+2\,c_2+c_2 \\ &\le \dots \\ &\le 2^k T(n-k)+\sum_{i=0}^{k-1} 2^i c_2 \\ &\le \dots \\ &\le 2^n T(0)+\sum_{i=0}^{n-1} 2^i c_2 \\ &= 2^n c_1+(2^n-1)\,c_2 \\ &= 2^n(c_1+c_2)-c_2 \end{aligned}$$

da cui possiamo concludere che $T(n) = O(2^n)$.
Ragionando in maniera simile, possiamo scrivere

$$\begin{aligned} T(n) &= T(n-1)+T(n-2)+c_2 \\ &\ge 2\,T(n-2)+c_2 \\ &\ge 4\,T(n-4)+2\,c_2+c_2 \\ &\ge 8\,T(n-6)+4c_2+2\,c_2+c_2 \\ &\ge \dots \\ &\ge 2^k T(n-2k)+\sum_{i=0}^{k-1} 2^i c_2 \\ &\ge \dots \\ &\ge 2^{\lfloor n/2 \rfloor} T(n-2\lfloor n/2 \rfloor)+\sum_{i=0}^{\lfloor n/2 \rfloor-1} 2^i c_2 \\ &= 2^{\lfloor n/2 \rfloor} c_1+(2^{\lfloor n/2 \rfloor}-1)\,c_2 \\ &= 2^{\lfloor n/2 \rfloor}(c_1+c_2)-c_2 \end{aligned}$$

da cui possiamo concludere the $T(n)=\Omega(2^{\lfloor n/2 \rfloor})$.

Esercizio 1.3

Scrivere un algoritmo il cui costo computazionale $T(n)$ sia dato dalla seguente relazione di ricorrenza:

$$T(n)=\begin{cases} O(1) & \text{se } n\le 10 \\ n\,T(n-1)+O(1) & \text{altrimenti} \end{cases}$$

dove n è un parametro intero positivo passato come input all'algoritmo. Non è richiesto il calcolo della soluzione della ricorrenza, né è richiesto che l'algoritmo produca un risultato di una qualsivoglia utilità pratica.

Soluzione. Un possibile algoritmo è il seguente

```
algoritmo FUN( int n ) → int
   if ( n ≤ 10 ) then
        return 1;
   else
        int a := 0;
        for i := 1 to n do
             a := a + FUN(n-1);
        endfor
        return a;
   endif
```

Esercizio 1.4

Supponiamo di disporre di un "coprocessore speciale" in grado di fondere due array ordinati, aventi complessivamente n elementi, in tempo $O(\sqrt{n})$. Determinare il costo computazionale dell'algoritmo di ordinamento MERGESORT visto a lezione, in cui la fase di fusione dei (sotto-)vettori ordinati viene realizzata mediante il coprocessore speciale di cui sopra.

Richiamo teorico. (Master Theorem) L'equazione di ricorrenza:

$$T(n)=\begin{cases} 1 & \text{se } n=1 \\ a\,T\left(\frac{n}{b}\right)+f(n) & \text{se } n>1 \end{cases}$$

ha come soluzione:

1 $T(n)=\Theta(n^{\log_b a})$ se $f(n)=O(n^{\log_b a-\epsilon})$ per un qualche $\epsilon > 0$;

2 $T(n)=\Theta(n^{\log_b a}\log n)$ se $f(n)=\Theta(n^{\log_b a})$

3 $T(n)=\Theta(f(n))$ se $f(n)=\Omega(n^{\log_b a+\epsilon})$ per un qualche $\epsilon > 0$ e se $af(n/b) \le cn$ per $c<1$ e n sufficientemente grande

Soluzione. L'equazione di ricorrenza che descrive il costo computazionale di MERGESORT, nel caso si usi il "coprocessore", è la seguente:

$$T(n)=\begin{cases} 1 & \text{se } n\le 1 \\ 2\,T(n/2)+\sqrt{n} & \text{altrimenti} \end{cases}$$

Applicando il Master Theorem, con a=2, b=2, $f(n)=\sqrt{(n)}$, si ricade nel caso 1), che ha come soluzione $T(n) = \Theta(n)$

Esercizio 1.5

Si consideri il seguente algoritmo ricorsivo:

```
algoritmo FUN( array A[1..n] di double, int i, int j ) → double
  if ( i > j ) then
      return 0;
  elseif ( i == j ) then
      return A[i];
  else
      int m := FLOOR( ( i + j ) / 2 );
      return FUN(A, i, m) + FUN(A, m+1, j);
  endif
```

L'algoritmo accetta come parametri un array $A[1..n]$ di n numeri reali e due interi i, j; l'algoritmo viene inizialmente invocato con FUN(A, 1, n) e restituisce un numero reale.

1. Scrivere la relazione di ricorrenza che descrive il costo computazionale di FUN in funzione di n
2. Risolvere la ricorrenza di cui al punto 1.
3. Cosa calcola FUN(A, 1, n) ? (spiegare a parole)

Soluzione. La funzione FUN calcola la somma dei valori presenti nel vettore $A[1..n]$. Il costo computazionale $T(n)$ soddisfa la seguente equazione di ricorrenza:

$$T(n)=\begin{cases} c_1 & \text{se } n\le 1 \\ 2\,T(n/2)+c_2 & \text{altrimenti} \end{cases}$$

che, in base al Master Theorem (caso 1, con a=2, b=2, $f(n)$=c_2) ha soluzione $T(n) = \Theta(n)$

Esercizio 1.6

Determinare in modo esplicito il costo computazionale del seguente algoritmo ricorsivo:

```
algoritmo FUN( int n ) → int
  if ( n ≤ 1 ) then
      return n;
  else
      int a := 1;
      for k := 1 to n do
          a := a*2;
      endfor
      return a + 2*FUN(n/2);
  endif
```

Soluzione. Il costo $T(n)$ soddisfa l'equazione di ricorrenza:

$$T(n)=\begin{cases} c_1 & \text{se } n\le 1 \\ T(n/2)+c_2\,n & \text{altrimenti} \end{cases}$$

che in base al caso 3) del Master Theorem (a=1, b=2, $f(n) = c_2n$) ha come soluzione $T(n) = \Theta(n)$

Esercizio 1.7

Si consideri l'algoritmo seguente, che accetta in input un intero positivo n

```
algoritmo FUN( int n ) → int
  if ( n ≤ 1 ) then
      return n;
  else
      return 10*FUN(n/2);
  endif
```

Calcolare il costo computazionale asintotico dell'algoritmo in funzione di n.

Soluzione. Il costo computazionale $T(n)$ soddisfa l'equazione di ricorrenza

$$T(n)=\begin{cases} c_1 & \text{se } n\le 1 \\ T(n/2)+c_2 & \text{altrimenti} \end{cases}$$

che in base al Master Theorem ha soluzione $T(n) = O(\log n)$

Esercizio 1.8

Si consideri il seguente algoritmo ricorsivo che accetta in input un intero $n \geq 0$

```
algoritmo FUN( int n ) → int
   if ( n > 1 ) then
        int a := 0;
        for i:=1 to n-1
             for j:=i+1 to n do
                  a := a + 2 * (i + j);
             endfor
        endfor
        for i := 1 to 16 do
             a := a + FUN(n/4);
        endfor
        return a;
   else
        return n-1;
   endif
```

Determinare il costo computazionale $T(n)$ dell'algoritmo, in funzione del parametro n.

Soluzione. L'equazione di ricorrenza è la seguente:

$$T(n)=\begin{cases} c_1 & \text{se } n\le 1 \\ 16\,T\left(\frac{n}{4}\right)+\Theta(n^2) & \text{altrimenti} \end{cases}$$

Che in base al Master Theorem (caso 2, con a=16, b=4, $f(n) = \Theta(n^2)$) ha soluzione $T(n) = \Theta(n^2 \log n)$

Esercizio 1.9

Si considerino i due algoritmi seguenti:

```
algoritmo ALGA( int n ) → int
        if ( n ≤ 1 ) then
                return 2*n;
        else
                int a := 2;
                for i := 1 to n/2 do
                        a := a + 2*i;
                endfor
                return ALGA( n/2 ) + ALGA( n/2 ) + a;
        endif
```

```
algoritmo ALGB( int n ) → int
        int a := 1;
        for i := 1 to n do
                for j := 1 to n/2 do
                        a := a + i + j;
                endfor
        endfor
        return a;
```

1. Calcolare il costo computazionale di ALGA in funzione di n.

2. Calcolare il costo computazionale di ALGB in funzione di n.

Soluzione. ALGA è un algoritmo ricorsivo, per cui è necessario innanzitutto scrivere l'equazione di ricorrenza per il tempo di esecuzione $T(n)$ in funzione della dimensione dell'input (in questo caso, il valore di n). Si ha:

$$T(n)=\begin{cases} c_1 & \text{se } n\le 1 \\ 2\,T(n/2)+c_2 n & \text{altrimenti} \end{cases}$$

(il termine $c_2 n$ deriva dal ciclo "for" che ha costo $\Theta(n)$). L'applicazione del Master Theorem (caso 2, con a=2, b=2, $f(n) = c_2 n$) da come risultato $T(n) = \Theta(n \log n)$.
ALGB invece è un algoritmo iterativo. Il corpo interno dei due cicli "for" ha costo O(1) e viene eseguito esattamente $n \times n/2$ volte; il costo complessivo dell'algoritmo è quindi $\Theta(n^2)$

Esercizio 1.10

Scrivere un algoritmo ricorsivo FUN(**array** A[1..n] di **int**, **int** i, **int** j) tale che il costo $T(n)$ dell'invocazione FUN(A, 1, n) soddisfi la seguente relazione di ricorrenza:

$$T(n)=\begin{cases} O(1) & \text{se } n\le 10 \\ T(n/2)+T(n-1)+cn & \text{altrimenti} \end{cases}$$

Soluzione. Una tra le tante possibilità è la seguente:

```
algoritmo FUN( array A[1..n] di int, int i, int j ) → int
   if ( j - i + 1 ≤ 10 ) then
        return 0;
   else
        int m := FLOOR( ( i + j ) / 2 );
        int a := FUN(A, i, m);          // costo T(n/2)
        int b := FUN(A, i, j – 1 );     // costo T(n-1)
        int c := 0;
        for k := i to j do              // costo Θ(n)
             c := c + k;
        endfor
        return a + b + c;
   endif
```

Il punto importante di questo esercizio è quello di osservare che la dimensione dell'input da considerare per l'analisi delle chiamate ricorsive è la parte di array compresa tra gli estremi i e j (questo è esattamente quello che abbiamo visto in altri algoritmi, come ad esempio l'algoritmo di ricerca binaria, oppure in algoritmi di ordinamento ricorsivi come QUICKSORT o MERGESORT). Quindi ad esempio, scrivere la condizione dell'if come

```
if (n ≤ 10) then        // SBAGLIATO
```

non sarebbe stato corretto, in quanto la dimensione dell'input (numero di elementi di A compresi tra le posizioni i e j) è $(j – i + 1)$.

Esercizio 1.11

Il professor Tritonacci ha definito la sequenza di numeri di Tritonacci Tri(n), con n = 0, 1, 2, … nel modo seguente:

$$Tri(n)=\begin{cases} 1 & \text{se } n\le 2 \\ Tri(n-1)+Tri(n-2)+Tri(n-3) & \text{altrimenti} \end{cases}$$

Il professore afferma che il seguente algoritmo ricorsivo TritonacciRic è la soluzione più efficiente per calcolare il valore dell'*n*-esimo numero di Tritonacci:

```
TritonacciRic( integer n ) → integer
  if ( n ≤ 2 ) then
      return 1;
  else
      return TritonacciRic(n-1) + TritonacciRic(n-2) + TritonacciRic(n-3);
  endif
```

Calcolare il costo asintotico dell'algoritmo TritonacciRic(*n*) in funzione di *n*. Dimostrare che il professor si sbaglia proponendo un algoritmo più efficiente per calcolare il valore di Tri(*n*).

Soluzione. Il tempo di esecuzione $T(n)$ dell'algoritmo TritonacciRic(n) soddisfa la seguente equazione di ricorrenza:

$$T(n)=\begin{cases} c_1 & \text{se } n\le 2 \\ T(n-1)+T(n-2)+T(n-3)+c_2 & \text{altrimenti} \end{cases}$$

Osservando che la funzione $T(n)$ è monotona, cioè $T(n) \ge T(n\text{-}1)$, per $n \ge 3$ possiamo scrivere:

$$T(n)=T(n-1)+T(n-2)+T(n-3)+c_2 \ge 3\,T(n-3)+c_2$$

Ripetendo k volte la sostituzione si ha:

$$T(n)=3^k T(n-3k)+3^{k-1}c_2+3^{k-2}c_2+\ldots+c_2$$

da cui si può concludere che $T(n)=\Omega(3^{\lfloor n/3 \rfloor})$ (si veda l'Esercizio1.2). Quindi l'algoritmo TritonacciRic ha costo esponenziale in *n*.
Per calcolare in modo efficiente l'*n*-esimo numero di Tritonacci possiamo applicare le stesse idee che portano ad algoritmi efficienti per il calcolo dei numeri della sequenza di Fibonacci. In particolare, l'algoritmo TritonacciIter calcola Tri(*n*) in tempo $\Theta(n)$ usando spazio O(1)

```
TritonacciIter( int n ) → int
  if ( n ≤ 2 ) then
      return 1;
  else
      int tri0 := 1, tri1 := 1, tri2 := 1, tri3;
      for i := 3 to n do
          tri3 := tri2 + tri1 + tri0;
          tri0 := tri1;
          tri1 := tri2;
          tri2 := tri3;
      endfor
      return tri3;
  endif
```

Capitolo 2: Strutture Dati Elementari

Esercizio 2.1

Scrivere un algoritmo efficiente per risolvere il seguente problema: dato un array $A[1..n]$ di $n>0$ valori reali, restituire *true* se l'array A rappresenta un min-heap binario, *false* altrimenti. Calcolare la complessità nel caso pessimo e nel caso ottimo dell'algoritmo proposto, motivando le risposte.

Richiamo teorico. Un min-heap è un albero binario in cui ad ogni nodo v è associato un valore *v*.val. Uno heap gode delle seguenti proprità: (1) l'albero è completo fino al penultimo livello; (2) le foglie sono "accatastate" a sinistra; (3) il valore associato ad ogni nodo è minore o uguale al valore associato ai figli (se esistono).
E' possibile rappresentare uno heap con n elementi mediante un array $A[1..n]$, semplicemene memorizzando i valori dello heap "per livelli". L'elemento il cui valore è memorizzato in posizione i nell'array avrà i valori dei figli memorizzati negli elementi in posizione $2*i$ e $2*i+1$, rispettivamente.

Soluzione. L'algoritmo può essere espresso come segue:

```
algoritmo CheckMinHeap( array A[1..n] di double ) → bool
  for i := 1 to n do
      if ( 2*i ≤ n && A[i] > A[2*i] ) then
          return false;
      endif
      if ( 2*i+1 ≤ n && A[i] > A[2*i+1] ) then
          return false;
      endif
  endfor
  return true;
```

Si noti i controlli (2*i≤n) e (2*i+1≤n), che sono necessari per essere sicuri che i figli dell'elemento $A[i]$ (rispettivamente $A[2*i]$ e $A[2*i+1]$) non cadano fuori dall'array. Se tali controlli sono omessi, l'algoritmo è sbagliato in quanto può causare l'indicizzazione di un array al di fuori dei limiti.
Il caso pessimo si ha quando il ciclo "for" viene eseguito interamente, ossia quando l'array $A[1..n]$ effettivamente rappresenta un min-heap. In questo caso il costo è $O(n)$.
Il caso ottimo si verifica quando la radice $A[1]$ risulta maggiore di uno dei due figli ($A[2]$ o $A[3]$, se esistono). In questo caso il ciclo "for" esce alla prima iterazione restituendo *false*, e il costo risulta O(1).

Esercizio 2.2

Il professor P. Nocchio dichiara di aver inventato una rivoluzionaria struttura dati: i PN-Heap. Un PN-Heap supporta le seguenti operazioni, tutte basate unicamente su confronti: creazione di un PN-Heap a partire da una collezione di n elementi confrontabili in tempo $O(n)$; individuazione dell'elemento minimo (o massimo) in tempo O(1); rimozione dell'elemento minimo (o massimo) in tempo O(1). I costi sono tutti nel caso pessimo.

Determinare il costo computazionale di un ipotetico algoritmo di ordinamento PN-HeapSort ottenuto combinando HeapSort con la struttura dati PN-Heap. Alla luce della risposta alla domanda precedente, vi fidate della scoperta del professor P. Nocchio?

Soluzione. Scriviamo lo pseudocodice di un ipotetico algoritmo di ordinamento PN-HeapSort, che è basato sullo schema di HeapSort ma anziché un normale heap, facciamo uso di un PN-Heap (PH):

```
algoritmo PN-HeapSort( array A[1..n] di double )
   H := PN-Heapify(A);          // Crea un PN-Heap a partire dall'array A
   for i := 1 to n do
       A[i] := H.Min();         // estrae il minimo dal PN-Heap
       H.DeleteMin();           // cancella il minimo dal PN-Heap
   endfor
```

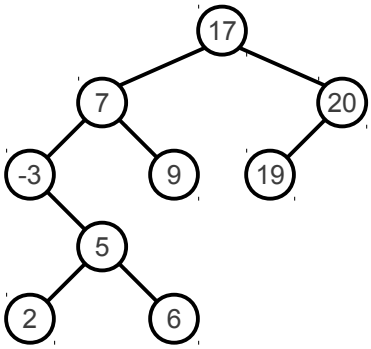
Per ipotesi, la creazione del PN-Heap richiede tempo O(n); il ciclo "for" che segue viene eseguito n volte, e ogni iterazione ha costo O(1) (in quanto per ipotesi le operazioni di ricerca del minimo e cancellazione del minimo costano entrambe O(1) in un PN-Heap). Quindi il costo del ciclo "for" è complessivamente O(n). L'algoritmo PN-HeapSort ha costo computazionale complessivo O(n).
Se i PN-Heap funzionassero come dichiarato, l'algoritmo PN-HeapSort sarebbe un algoritmo di ordinamento basato su confronti che ordina un array di n elementi in tempo O(n). E' noto però che tale algoritmo non può esistere, in quanto il limite inferiore della complessità del problema dell'ordinamento basato su confronti è $\Omega(n \log n)$. Quindi non c'è da fidarsi del professor P. Nocchio.

Esercizio 2.3

Si consideri un albero binario di ricerca non bilanciato, inizialmente vuoto. Disegnare gli alberi che risultano dopo l'inserimento di ciascuna delle seguenti chiavi numeriche: 17, 7, 9, -3, 20, 19, 5, 2, 6.

Richiamo teorico. Un albero binario di ricerca (ABR) è un albero binario in cui a ciascun nodo v è associata una chiave $v.key$. In un ABR vale la seguente proprietà: per ogni nodo v, tutte le chiavi contenute nel sottoalbero sinistro sono minori o uguali a $v.key$, e tutte le chiavi contenute nel sottoalbero destro sono maggiori o uguali a $v.key$.

Soluzione. Ci limitiamo a rappresentare l'albero finale ottenuto:



Esercizio 2.4

Si consideri un albero binario B in cui a ciascun nodo t è associata una chiave numerica (reale) t.key. Non ci sono chiavi ripetute, e tutte le chiavi appartengono all'intervallo $[a, b]$

1. Scrivere un algoritmo efficiente che dato in input l'albero B e gli estremi a e b, restituisce true se e solo se B rappresenta un albero binario di ricerca. Non è consentito usare variabili globali.

2. Calcolare il costo computazionale nel caso ottimo e nel caso pessimo dell'algoritmo di cui al punto 1. Disegnare un esempio di albero che produce un caso pessimo, e un esempio di albero che produce il caso ottimo.

Soluzione. Innanzitutto è utile ricordare che in un ABR i valori del sottoalbero sinistro di un nodo *t* sono tutti minori o uguali a t.*key*, mentre i valori del sottoalbero destro di *t* sono tutti maggiori o uguali di t.*key*. Non è quindi sufficiente confrontare *t* solamente con i propri figli. Una possibile soluzione è la seguente

```
algoritmo CheckABR( nodo t, double a, double b ) → bool
   // inizialmente t deve essere la radice di B
   if ( t == null ) then
        return true;
   else
        return ( a ≤ t.key && t.key ≤ b &&
                 checkABR(t.left, a, t.key) &&
                 checkABR(t.right, t.key, b) );
   endif
```

L'algoritmo ricorsivo verifica che per ogni nodo *v* valgano le seguenti condizioni:

- $a \leq t.key$;
- $t.key \leq b$;
- il sottoalbero sinistro è un ABR con chiavi aventi valori in $[a, t.key]$;
- il sottoalbero destro è un ABR con chiavi aventi valori in $[t.key, b]$.

Il costo nel caso ottimo è O(1), e si verifica quando la chiave presente nel figlio sinistro della radice di B viola la proprietà di ABR; il costo nel caso pessimo è $\Theta(n)$, dove n è il numero di nodi dell'albero, e si verifica quando *B* è effettivamente un ABR.

Esercizio 2.5

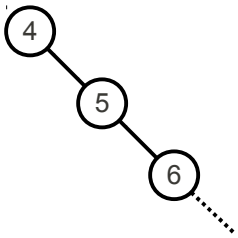
Si consideri un albero binario di ricerca *B*, non necessariamente bilanciato, in cui a ciascun nodo *t* è associata una chiave numerica (reale) *t.key*. Non ci sono chiavi ripetute.

1. Scrivere un algoritmo efficiente che dato in input l'albero *B* e due valori reali *a* e *b*, con $a<b$, restituisce il numero di nodi la cui chiave appartiene all'intervallo $[a, b]$ (estremi inclusi)
2. Calcolare il costo computazionale dell'algoritmo di cui al punto 1 nel caso ottimo e nel caso pessimo.

Soluzione. È possibile risolvere il problema utilzzando lo pseudocodice seguente:

```
algoritmo ContaNodiAB( nodo t, double a, double b ) → int
   if ( t == null ) then
        return 0;
   else
        if ( t.key < a ) then
             return ContaNodiAB(t.right, a, b);
        elseif ( t.key > b ) then
             return ContaNodiAB(t.left, a, b);
        else
             return 1 + ContaNodiAB(t.left, a, b) + ContaNodiAB(t.right, a, b);
        endif
   endif
```

Il caso peggiore si verifica quando tutte le chiavi presenti nell'albero sono comprese nell'intervallo $[a, b]$; in tale situazione, l'algoritmo ContaNodiAB è sostanzialmente equivalente (dal punto di vista del costo computazionale) ad un algoritmo di visita dell'albero (ogni nodo viene esaminato una volta). Quindi, nel caso peggiore l'algoritmo ContaNodiAB ha costo $\Theta(n)$, dove n è il numero di nodi dell'albero.
Il caso migliore si verifica ad esempio effettuando la chiamata ContaNodiAB(B, 1, 2) sull'albero *B* avente la struttura seguente:



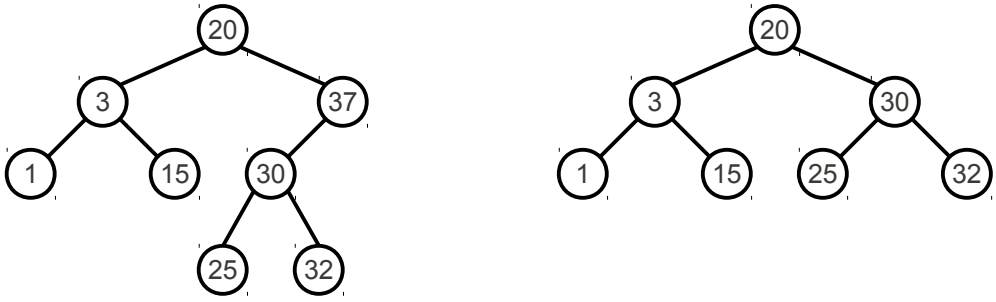
L'algoritmo termina in tempo O(1), a prescindere dal numero *n* di nodi dell'albero. Quindi il costo nel caso ottimo è O(1).

Esercizio 2.6

Si consideri un albero binario di ricerca non bilanciato, inizialmente vuoto.

1. Disegnare la struttura degli alberi risultanti dopo l'inserimento di ognuna delle seguenti chiavi, nell'ordine indicato: 20, 3, 37, 30, 1, 15, 25, 32.
2. Disegnare la struttura dell'albero dopo la rimozione del valore 37.

Soluzione. L'albero a sinistra rappresenta la situazione dopo aver inserito tutti i valori; l'albero a destra rappresenta la situazione dopo la rimozione del valore 37.



Esercizio 2.7

Si consideri un albero binario *T*, avente struttura arbitraria, in cui ciascun nodo *t* contenga un numero reale *t.val*;

1. Scrivere un algoritmo efficiente che, dato in input *T*, restituisca la somma dei valori contenuti in tutte le foglie; se l'albero è vuoto l'algoritmo restituisce zero. È vietato usare variabili globali.
2. Assumendo che l'albero *T* sia completo e tutte le foglie si trovino sullo stesso livello, determinare formalmente il costo computazionale dell'algoritmo in funzione del numero *n* di nodi.

Soluzione.

```
algoritmo SommaFoglie( nodo t ) → double
    if ( t == null ) then
        return 0;
    elseif ( t.left == null && t.right == null ) then
        return t.val;
    else
        return SommaFoglie(t.left) + SommaFoglie(t.right);
    endif
```

Osserviamo che in un albero binario completo e bilanciato con *n* nodi, il numero di nodi nel sottoalbero destro (o sinistro) della radice è (*n*-1)/2, che per *n* sufficientemente grande possiamo approssimare con *n*/2.

Pertanto, il costo $T(n)$ dell'algoritmo SommaFoglie soddisfa la seguente relazione di ricorrenza:

$$T(n)=\begin{cases} c_1 & \text{se } n\le 1 \\ 2\,T(n/2)+c_2 & \text{altrimenti} \end{cases}$$

la cui soluzione, in base al Master Theorem, è $T(n) = \Theta(n)$.

Esercizio 2.8

Si consideri un albero binario T, non necessariamente bilanciato. Ogni nodo t di T contiene un numero reale $t.val$.

1. Scrivere un algoritmo ricorsivo che accetta in input l'albero T e un intero $k\ge 0$, e restituisce la somma dei valori contenuti nei nodi che si trovano esattamente a profondità k; ricordiamo che la profondità di un nodo è pari al numero di archi del cammino che lo collega con la radice (la radice ha profondità zero). Se non esistono nodi a profondità k, l'algoritmo ritorna zero.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1 in funzione del numero n di nodi di T, nell'ipotesi in cui il valore di k sia uguale alla profondità dell'albero.

Soluzione.

```
algoritmo SommaNodiProfK( nodo t, int k ) → double
    if ( t == null ) then
        return 0;
    elseif ( k == 0 ) then
        return t.val;
    else
        return SommaNodiProfK(t.left, k-1) + SommaNodiProfK(t.right, k-1);
    endif
```

L'algoritmo viene inizialmente invocato passando come primo parametro un riferimento alla radice dell'albero. Nell'ipotesi in cui k sia uguale alla profondità dell'albero, il costo computazionale è $\Theta(n)$ in quanto l'algoritmo coincide quasi esattamente con un algoritmo di visita completa dell'albero.

Esercizio 2.9

1. Scrivere un algoritmo efficiente che, dato in input un albero binario T, restituisce true se e solo se T rappresenta un albero binario completo (ricordiamo che in un albero binario completo ogni nodo non foglia ha esattamente due figli; l'albero vuoto si assume completo).
2. Calcolare il costo computazionale dell'algoritmo di cui al punto 1, nel caso venga applicato ad un albero completo bilanciato con n nodi.

Soluzione. Una soluzione ricorsiva è la seguente

```
algoritmo AlberoBinCompleto( nodo t ) → bool
    if ( t == null ) then
        return true;      // caso base, l'albero vuoto è completo
    elseif ( t.left ≠ null && t.right ≠ null ) then
        return AlberoBinCompleto(t.left) && AlberoBinCompleto(t.right);
    elseif ( t.left == null && t.right == null ) then
        return true;      // l'albero composto da una singola foglia è completo
    else
        return false;
    endif
```

Il costo $T(n)$ dell'algoritmo applicato ad un albero binario completo bilanciato di n nodi è descritto dalla seguente relazione di ricorrenza:

$$T(n)=\begin{cases} c_1 & \text{se } n \le 1 \\ 2\,T(n/2)+c_2 & \text{altrimenti} \end{cases}$$

la cui soluzione, in base al Master Theorem, è $T(n) = \Theta(n)$.

Esercizio 2.10

Si consideri un array $Q[1..n]$ di valori reali che contiene le variazioni delle quotazioni di borsa delle azioni ACME S.p.A. durante $n \ge 1$ giorni consecutivi. Ciascun valore $Q[i]$ può essere positivo o negativo a seconda che al termine del giorno i le azioni ACME abbiano registrato un incremento o una diminuzione del loro valore rispetto alla quotazione di apertura del giorno medesimo. $Q[i]$ può anche essere zero, nel caso in cui le quotazioni delle azioni non abbiano subito variazioni.
Scrivere un algoritmo efficiente che, dato in input l'array Q, restituisca il massimo numero di giorni consecutivi in cui le azioni ACME sono risultate in rialzo. In altre parole, si chiede di determinare la lunghezza massima dei sottovettori di Q costituiti da valori contigui strettamente positivi. Ad esempio, se Q=[0.1, -0.8, 0.2, 0.3, 1.2, -0.9, 0.0, -1.7, -2] l'algoritmo deve restituire 3, in quanto esistono al più tre valori positivi consecutivi (il sottovettore [0.2, 0.3, 1.2]).
Analizzare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Definiamo un algoritmo di costo ottimo (lineare) che funziona come segue. Manteniamo due contatori, np ("numero positivi") e maxnp ("massimo numero di valori positivi") ed esaminiamo a turno tutti gli elementi di Q. np rappresenta la lunghezza della sequenza di valori positivi consecutivi che include l'elemento $Q[i]$ che stiamo esaminando. Se $Q[i] > 0$, si incrementa np e si aggiorna maxnp se necessario; se $Q[i] \le 0$, il valore di np diventa zero.

```
algoritmo QuotazioniACME( array Q[1..n] di double ) → int
    int np := 0;
    int maxnp := 0;
    for i := 1 to n do
        if ( Q[i] > 0 ) then
            np := np+1;
            if ( np > maxnp ) then
                maxnp := np;
            endif
        else
            np := 0;
        endif
    endfor
    return maxnp;
```

L'algoritmo ha costo $\Theta(n)$. Si presti attenzione al fatto che sarebbe sbagliato spostare il blocco "if (np > maxnp) … endif" nel ramo "else" dell'if più esterno; in sostanza, l'algoritmo scritto come segue non funziona:

```
algoritmo Sbagliato( array Q[1..n] di double ) → int
    int np := 0;
    int maxnp := 0;
    for i := 1 to n do
        if (Q[i] > 0) then
            np := np+1;
        else
            if ( np > maxnp ) then        // questo test è sbagliato in questa posizione!
                maxnp := np;
            endif
            np := 0;
        endif
    endfor
```

```
return maxnp;
```

in quanto produce il risultato errato in casi come Q=[0.1, 0.9, -2, 0.7, 1.2, 1.3] (*perchè?*).

Esercizio 2.11

Si consideri un albero binario T in cui a ciascun nodo v è associato un numero reale $t.val$. Dato un qualsiasi cammino che porta dalla radice ad una foglia, definiamo il costo del cammino come la somma dei valori associati a tutti i nodi attraversati (inclusa la radice e la foglia di destinazione)

1. Scrivere un algoritmo che, dato in input l'albero T, restituisce il costo massimo tra tutti i cammini radice-foglia. Nel caso di albero vuoto, l'algoritmo restituisce 0.
2. Analizzare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. È possibile risolvere il problema mediante un algoritmo ricorsivo:

```
algoritmo CamminoPesoMax( nodo t ) → double
  if ( t == null ) then
      return 0;
  else
      return t.val + max( CamminoPesoMax(t.left), CamminoPesoMax(t.right) );
  endif
```

La funzione max(a, b) restituisce il valore massimo tra a e b. L'algoritmo, applicato ad un albero binario con n nodi, ha costo $\Theta(n)$ in quanto equivale ad un algoritmo di visita dell'albero (esamina un nodo una e una sola volta). Se si assume che l'albero sia bilanciato e completo, si può fare una dimostrazione diretta scrivendo l'equazione di ricorrenza $T(n) = 2T(n/2)+O(1)$ e risolvendola tramite il Master Theorem.

Esercizio 2.12

Si consideri l'algoritmo QuickFind con euristica sul peso. Scrivere un algoritmo che crei n=16 insiemi disgiunti, ed effettui una serie di operazioni Union di cui almeno una (ad esempio, l'ultima) richieda $n/2$ cambiamenti del riferimento al padre. È possibile fare uso di operazioni Find durante l'esecuzione dell'algoritmo.

Richiamo teorico. Nelle strutture QuickFind l'operazione più efficiente (come dice il nome) è l'operazione Find. Le strutture QuickFind sono quindi rappresentate da alberi di altezza 1, in cui la radice dell'albero è il rappresentante dell'insieme. L'operazione Union consiste nel rendere le foglie di uno dei due alberi figlie della radice dell'altro; questo richiede di settare il riferimento al padre di tutte le foglie.

Esercizio 2.13

Si consideri l'algoritmo QuickUnion con euristica sul rango (altezza degli alberi QuickUnion). Scrivere un algoritmo che crei n=16 insiemi disgiunti, ed effettui una serie di operazioni Union che al termine producano un singolo albero (che corrisponde al singolo insieme {1, …, 16} di altezza massima possibile. Se necessario, è possibile fare uso di operazioni Find durante l'esecuzione dell'algoritmo.

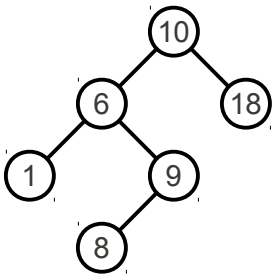
Soluzione. Ricordiamo che l'operazione union eseguita su alberi QuickUnion con euristica sul rango funziona nel modo seguente: si rende la radice l'albero più basso figlia della radice dell'albero più alto; in caso di unione di alberi aventi la stessa altezza, l'albero risultante crescerà in altezza di uno.

```
algoritmo AlberoQuickUnion( int n )
  QuickUnion UF;
  for i := 1 to n do
      UF.MakeSet(i);
  endfor
  int s := 1;
  while ( s < n ) do
```

```
        for i := 1 to n step s do
            UF.UNION(UF.FIND(i), UF.FIND(i+s)); // il nuovo rappresentante è i
        endfor
        s := s*2;
    endwhile
```

Esercizio 2.14

Si consideri un albero binario di ricerca, non bilanciato, in cui ciascun nodo contiene una chiave intera. Non ci sono chiavi ripetute. L'albero ha inizialmente la struttura seguente.

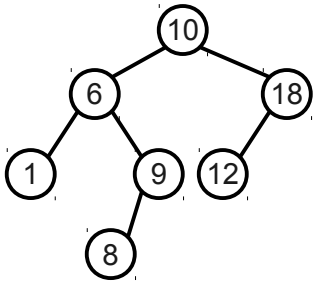


Mostrare la struttura dell'albero dopo ciascuna delle seguenti operazioni (ciascuna operazione si riferisce all'albero risultante al termine di quella precedente).

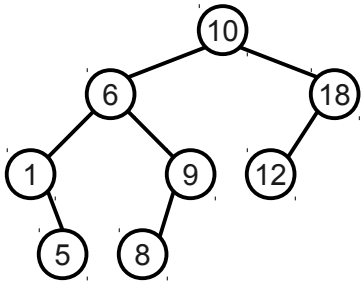
1. Inserimento 12
2. Inserimento 5
3. Cancellazione 6
4. Inserimento 23
5. Cancellazione 10
6. Inserimento 19
7. Inserimento 6

Soluzione.

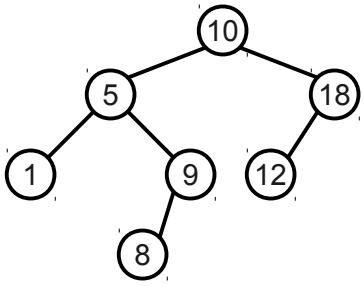
Inserimento 12



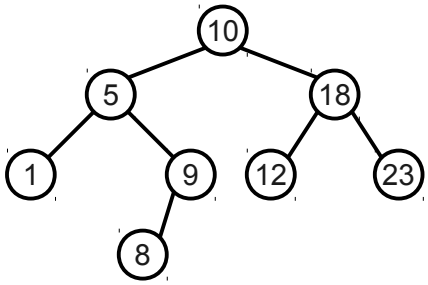
Inserimento 5



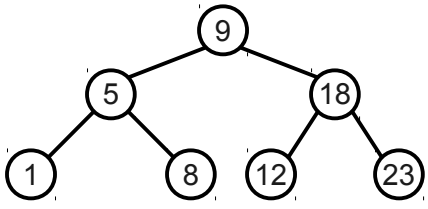
Cancellazione 6



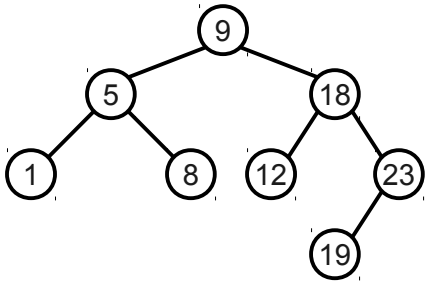
Inserimento 23



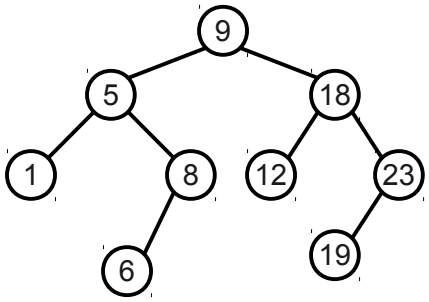
Cancellazione 10



Inserimento 19



Inserimento 6



Esercizio 2.15

Sia $P[1..n]$ un vettore non ordinato di numeri reali che rappresenta il prezzo medio giornaliero delle azioni della ACME S.p.A. durante $n \geq 1$ giorni successivi. Vogliamo calcolare il massimo numero di giorni contigui in cui il prezzo delle azioni risulta non decrescente. In altre parole, vogliamo calcolare la lunghezza massima tra tutti i sottovettori composti da elementi contigui di P che risultino ordinati in senso non decrescente. Scrivere un algoritmo per risolvere questo problema, e determinarne il costo computazionale.
Ad esempio, se P=[1, 3, 2, 5, 1, 4, 7] l'algoritmo restituisce 3, in quanto il sottovettore non decrescente composto dagli ultimi tre elementi [1, 4, 7] è ordinato in senso crescente. Altro esempio, se P=[12, 3, 5, 7, 8, 4, 6], l'algoritmo restituisce 4, in quanto il più lungo sottovettore ordinato è appunto [3, 5, 7, 8] che ha 4 elementi. Ultimo esempio, se P=[5, 4, 3, 2, 1] l'algoritmo ritorna 1, perché gli unici sottovettori non decrescenti sono i singoli elementi di P.

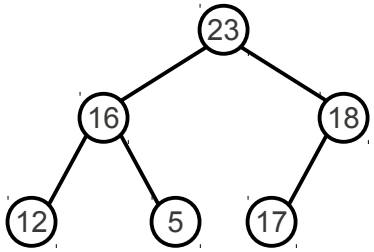
Soluzione.

```
algoritmo QuotazioneNonDecrescente( array P[1..n] di double ) → int
    int L := 1;
    int maxL := 1;
    for i := 2 to n do
        if ( P[i-1] ≤ P[i] ) then
            L := L + 1;
            if ( L > maxL ) then
                maxL := L;
            endif
        else
            L := 1;
        endif
    endfor
    return maxL;
```

Il costo dell'algoritmo è $\Theta(n)$.

Esercizio 2.16

Si consideri il max-heap seguente:

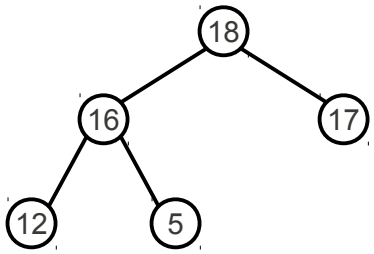


Mostrare la struttura del max-heap dopo ciascuna delle operazioni seguenti (ogni operazione viene compiuta sul min-heap risultante dall'operazione precedente, e deve essere eseguita mediante gli algoritmi descritti nel libro di testo e visti a lezione):

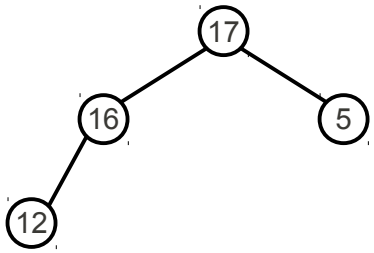
1. Cancellazione valore massimo
2. Cancellazione valore massimo
3. Inserimento 13
4. Inserimento 28
5. Cancellazione valore massimo
6. Inserimento 43

Soluzione.

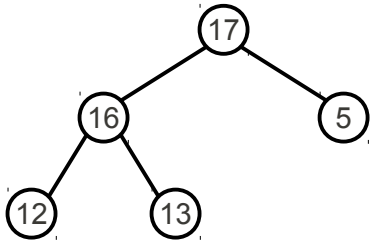
1: Cancellazione valore massimo



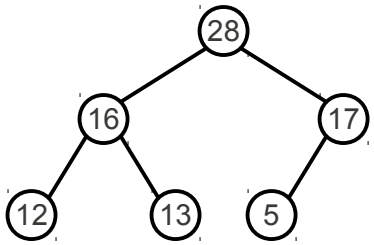
2: Cancellazione valore massimo



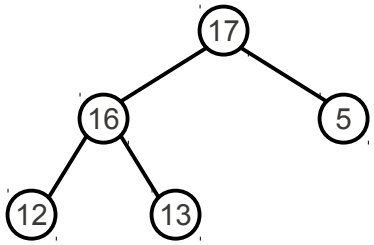
3: Inserimento 13



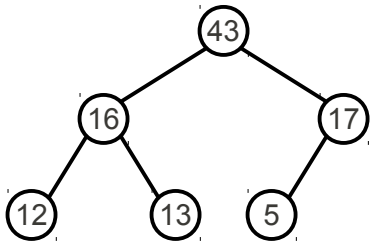
4: inserimento 28



5: Cancellazione massimo



6: Inserimento 43



Esercizio 2.17

Descrivere un algoritmo che dato un array $A[1..n]$ di interi appartenenti all'insieme $\{1, 2, \ldots k\}$, preprocessa l'array in tempo $O(n + k)$ in modo da generare una opportuna struttura dati che consenta di rispondere in tempo $O(1)$ a interrogazioni del tipo: "quanti elementi di A sono compresi nell'intervallo $[a, b]$?"(per qualsiasi $1 \leq a \leq b \leq k$, a e b interi)

Soluzione. L'idea è quella di costruire un array di interi $S[1..k]$, tale che $S[i]$ sia il numero di valori di A che

sono minori o uguali a i. Una volta costruito S, il numero di valori di V che sono compresi in $[a, b]$ è $(S[b] - S[a\text{-}1])$ se $a>1$, oppure $S[b]$ se $a=1$.
Si tratta a questo punto di capire come costruire S in tempo complessivo $O(n+k)$. L'idea è quella di procedere in due fasi: nella prima fase costruiamo un vettore $F[1..k]$ in cui $F[i]$ è il numero di volte in cui il valore i compare in A; usiamo poi F per costruire S:

```
algoritmo CostruisciS( array A[1..n] di int ) → array[1..k] di int
   array F[1..k] di int;
   array S[1..k] di int;
   // Inizializziamo F[1..k] a zero
   for i := 1 to k do
        F[i] := 0;
   endfor;
   // Prima fase: costruzione di F
   for i := 1 to n do
        F[A[i]] := F[A[i]] + 1;
   endfor
   // Seconda fase: costruzione di S
   S[1] := F[1];
   for i := 2 to k do
        S[i] := S[i-1] + F[i];
   endfor
   return S;
```

Esercizio 2.18

Scrivere un algoritmo che dati in input un array $A[1..n]$ non ordinato di n valori reali positivi (strettamente maggiori di zero) e un valore reale positivo v, restituisce *true* se e solo se esistono due valori in A la cui somma sia esattamente uguale a v. Analizzare il costo computazionale dell'algoritmo proposto.

Soluzione. Descriviamo un algoritmo banale ma non efficiente che controlla tutte le coppie $A[i], A[j]$, con $1 \leq i < j \leq n$ per verificare se $A[i] + A[j] = v$. Lo pseudocodice è il seguente

```
algoritmo EsisteSomma( array A[1..n] di double, double v ) → bool
   for i := 1 to n - 1 do
        for j := i + 1 to n do
             if (A[i] + A[j] == v) then
                  return true;
             endif
        endfor
   endfor
   return false;
```

Un errore banale ma che ho visto compiere più di una volta è quello di scrivere l'if più interno come:

```
             if (A[i] + A[j] == v) then
                  return true;
             else
                  return false;          // SBAGLIATO!!!!!
             endif
```

Questo sarebbe sbagliato, in quanto l'algoritmo terminerebbe sempre alla prima iterazione, senza mai controllare tutte le coppie.

Il costo computazionale dell'algoritmo è $O(n^2)$. Il caso pessimo si ha quando non esiste alcuna coppia di indici che soddisfano la condizione richiesta: in questo caso l'algoritmo richiede tempo $\Theta(n^2)$. Il caso ottimo si verifica quando $A[1] + A[2] = v$, e in tal caso l'algoritmo termina in tempo $O(1)$ in quanto la condizione **if**

(A[i] + A[j] == v) è immediatamente verificata.

In realtà esiste una soluzione più efficiente nel caso pessimo, che consiste nell'ordinare il vettore A in senso non decrescente, e ragionare come segue. Consideriamo i valori $A[1]$ e $A[n]$ (dell'array ordinato). Se la loro somma vale v, abbiamo terminato. Se $A[1] + A[n] > v$, sappiamo che $A[n]$ non può far parte di alcuna coppia di valori la cui somma dia v, in quanto per ogni k=1, …, n-1 avremo $A[k] + A[n] \geq A[1] + A[n] > v$, dato che $A[k] \geq A[1]$ (essendo A ordinato). Viceversa, se $A[1] + A[n] < v$, sappiamo che $A[1]$ non può far parte di alcuna coppia di valori la cui somma dia v, dato che per ogni $k = 2, \ldots, n$ si avrebbe $A[k] \leq A[n]$, da cui $A[1] + A[k] \leq A[1] + A[n] < v$.
L'algoritmo seguente risolve il problema in tempo $O(n \log n)$, che è il tempo richiesto per ordinare A usando un argoritmo di ordinamento efficiente.

```
algoritmo EsisteSomma( array A[1..n] di double, double v ) → bool
    OrdinaCrescente(A);
    int i := 1;
    int j := n;
    while ( i < j ) do
        if ( A[i] + A[j] == v ) then
            return true;
        elseif ( A[i] + A[j] < v ) then
            i := i + 1;
        else
            j := j – 1;
        endif
    endwhile
    return false;
```

Esercizio 2.19

Scrivere un algoritmo che dati in input un array $A[1..n]$ di n interi positivi restituisce *true* se e solo se esistono due elementi in A la cui somma sia esattamente uguale a 17. L'algoritmo deve avere costo $O(n)$. (Suggerimento: utilizzare un vettore di 16 valori booleani $B[1..16]$). Cosa cambia se anziché 17 si vuole determinare se esistono due elementi la cui somma sia uguale a 16?

Soluzione. L'idea è la seguente: tutti gli elementi del vettore B sono inizialmente posti a *false*. Si effettua quindi una prima scansione dell'array A. Per ogni valore $k \in \{1, \ldots, 16\}$ presente in A si pone $B[k] = true$. Al termine di questa fase, per ogni intero $k \in \{1, \ldots, 16\}$ avremo che k è presente in A se e solo se $B[k] = true$.
Facco ciò si esegue una seconda fase in cui si effettua una scansione di B, controllando se esiste un indice k tale che $B[k] = true$ e $B[17 - k] = true$. In base a quanto appena detto, se questo si verifica significa che i numeri k e $(17-k)$ sono entrambi presenti in A; dato che la loro somma è 17, l'algoritmo ritorna *true*.

```
algoritmo EsisteCoppia17( array A[1..n] di int ) → bool
    array B[1..16] di bool
    // Inizializzazione
    for k := 1 to 16 do
        B[i] := false;
    endfor
    // Prima fase
    for k := 1 to n do
        if ( A[k] < 17 ) then
            B[A[k]] := true;
        endif
    endfor
    // Seconda fase
    for k := 1 to 8 do
        if (B[k] == true && B[17-k] == true) then
            return true;
        endif
```

```
    endfor
    return false;
```

Analizziamo il costo computazionale dell'algoritmo proposto. L'inizializzazione (primo ciclo "for") ha costo O(1): si noti infatti che effettua un numero costante di iterazioni che *non* dipende dalla dimensione *n* dell'input, e ogni iterazione ha costo O(1). la "prima fase" ha costo $\Theta(n)$. Infine, la "seconda fase" ha costo O(1) in quanto anche qui viene eseguito al più 8 volte, che è una costante indipendente dalla dimensione dell'input, e ogni iterazione costa O(1). In sostanza, il costo complessivo dell'algoritmo è $\Theta(n)$.

Applicando l'algoritmo al vettore *A* di 7 elementi così composto (i numeri della prima riga rappresentano gli indici delle celle del vettore):

1	*2*	*3*	*4*	*5*	*6*	*7*
13	1	12	4	4	19	4

Si ottiene il vettore *B* seguente:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
T	F	F	T	F	F	F	F	F	F	F	T	T	F	F	F

e dato che *B*[4] = *B*[17-4] = *true*, l'algoritmo restituisce *true* (la somma dei valori 4 e 13 da 17).

Se dobbiamo cercare una coppia di valori la cui somma sia 16, anziché 17, la soluzione sopra non funziona: questo si può verificare applicando l'algoritmo appena visto (con l'ovvia modifica di cambiare opportunamente la dimensione di *B*) al vettore A = [8, 5]. Il problema è che non è più sufficiente sapere se un certo valore compare in *A*, ma anche *quante volte* compare. Infatti il numero 16 si può ottenere anche come 8+8, ma l'algoritmo EsisteCoppia17 non tiene conto del numero di volte in cui ciascun valore compare.
La soluzione corretta in questo caso consiste nel definire un vettore *B* composto da 15 elementi interi; *B*[*k*] conterrà il numero di volte in cui il valore *k* compare nell'array *A*.
L'algoritmo può essere descritto come segue:

```
algoritmo EsisteCoppia16( array A[1..n] di int ) → bool
    array B[1..15] di int
    for k := 1 to 15 do
        B[k] := 0;
    endfor
    for k := 1 to n do
        if ( A[k] ≤ 15 ) then
            B[A[k]] := B[A[k]]+1;
        endif
    endfor
    for k := 1 to 8 do
        if ( (k==8 && B[k]>1) || (B[k]>0 && B[16-k]>0) ) then
            return true;
        endif
    endfor
    return false;
```

che ha costo $\Theta(n)$ come il caso precedente.

Per dimostrare il funzionamento dell'algoritmo EsisteCoppia16, consideriamo il vettore *A* seguente:

1	*2*	*3*	*4*	*5*	*6*	*7*

12	1	12	8	4	19	4

L'algoritmo produce il vettore *B*

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
1	0	0	2	0	0	0	1	0	0	0	2	0	0	0

Dato che *B*[4]>0 && *B*[16-4]>0, l'algoritmo restituisce *true*.

Esercizio 2.20

Dato un albero binario *T*, scrivere un algoritmo che calcola la profondità di *T*; l'algoritmo deve restituire un errore in caso di albero vuoto (infatti in questo caso la profondità non è ben definita).

Richiamo teorico. La profondità di un albero è definita come il numero di archi del cammino più lungo tra la radice e una foglia. Questa definizione non si può applicare agli alberi vuoti, dato che non hanno alcun cammino.

Soluzione . Una possibile soluzione è la seguente:

```
algoritmo Profondita( nodo T ) → int
   if ( T == null ) then
        errore: albero vuoto
   endif
   // Calcola profondità del sottoalbero sinistro, se non vuoto
   int p_left := 0;
   if ( T.left ≠ null ) then
        p_left := 1 + Profondita(T.left);
   endif
   // Calcola profondità del sottoalbero destro, se non vuoto
   int p_right := 0;
   if ( T.right ≠ null ) then
        p_right := 1 + Profondita(T.right);
   endif
   // Restituisce il massimo tra la profondità del sottoalbero destro e sinistro
   if (p_left > p_right) then
        return p_left;
   else
        return p_right;
   endif
```

Esercizio 2.21

Dato un albero binario *T*, scrivere un algoritmo che calcola il numero di foglie di *T* (le foglie sono i nodi che non hanno figli). Analizzare il costo computazionale dell'algoritmo proposto. Quando si verifica il caso ottimo? Quando si verifica il caso pessimo?

Soluzione . Una possibile soluzione è la seguente:

```
algoritmo ContaFoglie( nodo T ) → int
   if ( T == null ) then
        return 0;
   else
        if ( T.left == null && T.right == null ) then
             return 1;    // il nodo T è una foglia
        else
```

```
            return ContaFoglie(T.left) + ContaFoglie(T.right);
        endif
    endif
```

Il costo dell'algoritmo eseguito su un albero con n nodi è $\Theta(n)$ (sia nel caso ottimo che nel caso pessimo l'algoritmo visita e tutti i nodi dell'albero una e una sola volta).

Esercizio 2.22

Dato un albero binario T e un intero $k \geq 0$, scrivere un algoritmo che calcola il numero di nodi che si trovano esattamente a profondità k (ricordiamo a che la radice si trova a profondità zero). Analizzare il costo computazionale dell'algoritmo proposto. Quando si verifica il caso ottimo? Quando si verifica il caso pessimo?

Soluzione. Questo esercizio è molto simile all'Esercizio 2.8. Una possibile soluzione è la seguente:

```
algoritmo ContaNodiProfK( nodo T, int k ) → int
    if ( T == null ) then
        return 0;
    else
        if (k == 0) then
            return 1;
        else
            return ContaNodiProfK(T.left, k-1) + ContaNodiProfK(T.right, k-1);
        endif
    endif
```

Il caso ottimo si verifica quando gli n nodi dell'albero sono disposti come in una catena (l'albero degenera in una lista). In questo caso l'esplorazione dell'albero riguarda solo i primi k nodi della catena e poi si interrompe; il costo è quindi $O(k)$. Il caso peggiore si verifica quando l'albero è completo fino al livello k, cioè tutti i nodi fino al livello k hanno esattamente due figli. Ricordando che un albero binario completo di altezza k ha esattamente $(2^{k+1} - 1)$ nodi, si ha che il costo computazionale in questo caso è $O(2^{k+1} - 1) = O(2^k)$.

Esercizio 2.23

Sono dati due array $A[1..n]$ e $B[1..m]$, composti da $n \geq 1$ ed $m \geq 1$ numeri reali, rispettivamente. Gli array sono emtrambi ordinati in senso crescente. A e B non contengono valori duplicati*;* tuttavia, è possibile che uno stesso valore sia presente una volta in A e una volta in B.

1. Scrivere un algoritmo efficiente che stampi a video i numeri reali che appartengono all'unione dei valori di A e di B; l'unione è intesa in senso insiemistico, quindi gli eventuali valori presenti in entrambi devono essere stampati solo una volta. Ad esempio, se A=[1, 3, 4, 6] e B=[2, 3, 4, 7], l'algoritmo deve stampare 1, 2, 3, 4, 6, 7.
2. Determinare il costo computazionale asintotico dell'algoritmo proposto, in funzione di n e m.

Soluzione. Questo problema può essere risolto in modo efficiente utilizzando un meccanismo simile all'operazione MERGE dell'algoritmo MergeSort.

```
algoritmo Unione( array A[1..n] di double, array B[1..m] di double )
    int i:= 1;          // indice array A
    int j := 1;         // indice array B
    while ( i ≤ n && j ≤ m ) do
        if ( A[i] < B[j] ) then
            print A[i];
            i := i+1;
        else if ( A[i] > B[j] ) then
            print B[j];
            j := j+1;
```

```
        else
            print A[i];
            i := i+1;
            j := j+1;
        endif
    endwhile
    // stampa la "coda" di elementi che eventualmente restano in uno dei vettori.
    // solo uno dei due cicli "while" seguenti verra' eseguito
    while ( i ≤ n ) do
        print A[i];
        i := i+1;
    endwhile
    while ( j ≤ m ) do
        print B[j];
        j := j+1;
    endwhile
```

I due array A e B vengono esaminati utilizzando due cursori, i e j. Ad ogni passo, si confronta l'elemento $A[i]$ con l'elemento $B[j]$. Se tali valori differiscono, si stampa il minimo dei due e si incrementa il corrispondente cursore. Se i valori sono uguali, se ne stampa uno (ad esempio $A[i]$) e si incrementano entrambi i cursori. Quando si raggiunge la fine di uno dei due vettori è necessario ricordarsi di stampare tutti gli elementi residui che si trovano eventualmente nell'altro. Il costo di questo algoritmo è $\Theta(n + m)$ in quanto sostanzialmente effettua una scansione di entrambi i vettori esaminando ciascun elemento una e una sola volta in tempo O(1).

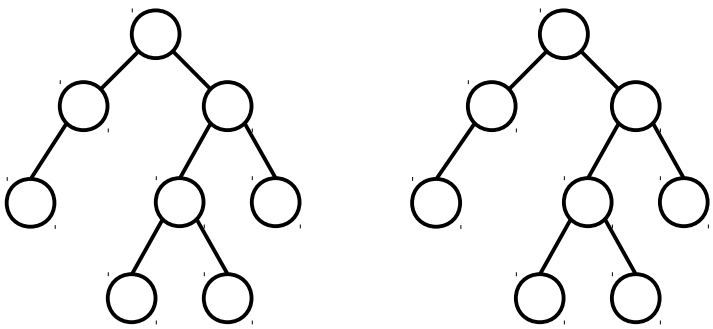
Esercizio 2.24

Si consideri una struttura QuickUnion con euristica sul rango (l'operazione Union(A, B) rende la radice dell'albero più basso figlia della radice dell'albero più alto; in caso di parità, la radice di B viene resa figlia della radice di A). Inizialmente sono presenti n=10 insiemi disgiunti {1}, {2}, … {10}.
Disegnare gli alberi QuickUnion dell'intera struttura union-find, dopo ciascuna delle seguenti operazioni:

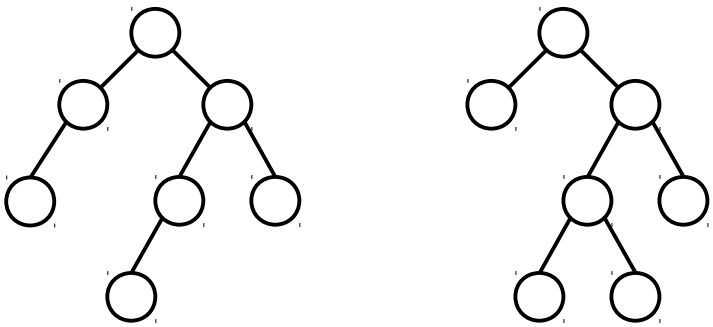
1. union(find(1), find(2))
2. union(find(3), find(4))
3. union(find(5), find(6))
4. union(find(1), find(4))
5. union(find(1), find(5))
6. union(find(7), find(8))

Esercizio 2.25

Scrivere un algoritmo che accetta in input due alberi binari $T1$ e $T2$ e restituisce *true* se e solo se $T1$ e $T2$ hanno la stessa struttura. Due alberi hanno la stessa struttura se sono entrambi vuoti, oppure se sono entrambi non vuoti e i sottoalberi destri e sinistri delle radici hanno rispettivamente la stessa struttura. Ad esempio, questi due alberi hanno la stessa struttura:



mentre questi no:



Supponendo che $T1$ e $T2$ abbiano lo stesso numero n di nodi, determinare il costo dell'algoritmo proposto nel caso peggiore e nel caso migliore; descrivere un esempio in cui si verifica il caso peggiore e uno in cui si verifica il caso migliore.

Soluzione. Una possibile soluzione è la seguente:

```
algoritmo StessaStruttura( nodo T1, nodo T2 ) → bool
   if ( T1 == null && T2 == null ) then
        return true;
   else
        // se siamo qui, allora uno tra T1 e T2 è vuoto, oppure nessuno dei due è vuoto
        if ( T1 == null || T2 == null ) then
             return false;          // uno dei due è vuoto e l'altro no
        else
             return StessaStruttura( T1.left, T2.left ) && StessaStruttura( T1.right, T2.right );
        endif
   endif
```

Se $T1$ e $T2$ sono entrambi vuoti, allora hanno la stessa struttura e l'algoritmo ritorna *true*. Se questo non è vero, significa che $T1$ e $T2$ non sono entrambi vuoti, ossia uno tra essi è vuoto, oppure nessuno dei due lo è. Se uno dei due è vuoto e l'altro no, allora i due alberi non hanno la stessa struttura. Se entrambi non sono vuoti, allora $T1$ e $T2$ hanno la stessa struttura se e solo se i sottoalberi sinistri e quelli destri hanno rispettivamente la stessa struttura.
Il costo computazionale dell'algoritmo può essere calcolato come segue. Supponiamo che entrambi gli alberi abbiano n nodi. Il caso peggiore per l'algoritmo si verifica quando $T1$ e $T2$ hanno la stessa struttura; in tal caso l'algoritmo effettua sostanzialmente una visita "sincronizzata" di entrambi, con un costo di $\Theta(n)$.
Il costo nel caso ottimo è un po' più delicato da individuare, e richiede alcune ipotesi. Nella maggior parte dei linguaggi di programmazione (incluso Java), la valutazione degli operatori booleani viene fatta in maniera "ottimizzata", nel senso che l'espressione P && Q viene valutata come segue:

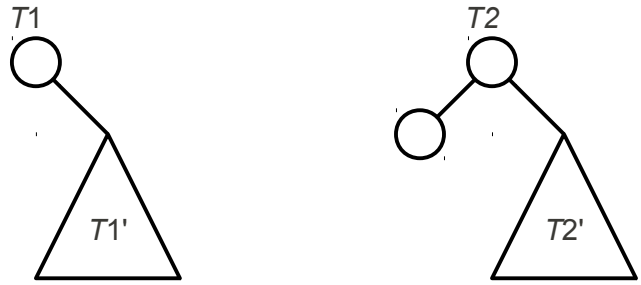
```
if (P == false) then
   return false;
else
   return Q;
```

Tale modo di procedere è detto *short-circuit evaluation* o anche *McCarthy evaluation*. Questo meccanismo ha implicazioni pratiche molto importanti, al di là di questioni di efficienza. Ad esempio, se l'espressione seguente *non* fosse valutata usando la short-circuit evaluation:

```
if ( a ≠ 0 && b/a > 0 ) then
   // fai qualcosa
endif
```

si otterrebbe un errore di divisione per zero nel caso in cui a == 0 (se la short-circuit evaluation non viene applicata, allora entrambi i lati dell'"and" logico verrebbero sempre e comunque valutati, e nel caso in cui a == 0 l'espressione b/a causa un errore).

Se gli alberi hanno ciascuno *n* nodi, e le struttura seguenti:



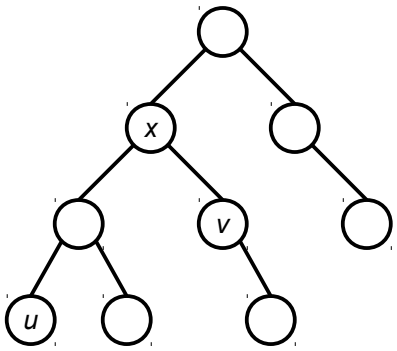
dove *T*1' ha (*n*-1) nodi, e *T*2' ne ha (*n*-2), allora la chiamata StessaStruttura(T1, T2) esegue l'istruzione:

```
return StessaStruttura( T1.left, T2.left ) && StessaStruttura( T1.right, T2.right );
```

che richiede di valutare innanzitutto il valore di StessaStruttura(T1.left, T2.left) che restituirà immediatamente *false*. Quindi, assumendo che si stia applicando la short-circuit evaluation, non sarà necessario valutare StessaStruttura(T1.right, T2.right) , perché il valore dell'argomento dell'istruzione return è già calcolato come *false*. Quindi in questo caso l'algoritmo termina dopo O(1) passi.

Esercizio 2.26

Consideriamo un albero binario *T* e due suoi nodi *u* e *v*. Si definisce *Lower Common Ancestor* (LCA) dei nodi *u* e *v*, indicato con LCA(*u*, *v*), il nodo di *T* di profondità massima che ha sia *u* che *v* come discendenti. Ad esempio, nel caso seguente il LCA di *u* e *v* è il nodo *x*:



Nel caso uno dei due sia antenato dell'altro, allora è anche il LCA di entrambi.
Scrivere un algoritmo efficiente che, dato in input l'albero *T* e due nodi *u* e *v*, restituisce il LCA di *u* e *v*. Determinare il costo asintotico dell'algoritmo proposto.

Soluzione. Proponiamo un algoritmo che opera utilizzando due stack *Su* e *Sv*. In ciascuno stack verranno inseriti i nodi presenti sul cammino da *u* e *v*, rispettivamente, fino alla radice. Al termine di questa operazione, in cima a entrambi gli stack si troverà necessariamente un riferimento alla radice dell'albero. Estraiamo contemporaneamente un elemento da entrambi gli stack; l'ultimo elemento uguale estratto da entrambi sarà il LCA. Occorre prestare attenzione al fatto che, durante questo processo, uno dei due stack

potrebbe svuotarsi (in particolare ciò accade se uno dei due nodi è antenato dell'altro, e quindi è esso stesso il LCA di entrambi).

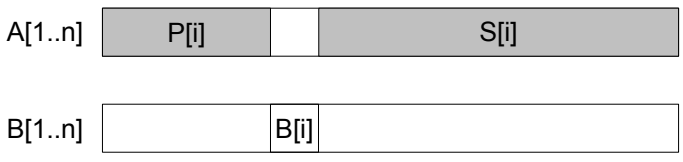
```
algoritmo LCA(nodo u, nodo v) → nodo
   Stack Su;
   Stack Sv;
   do
        Su.PUSH(u);
        u := u.parent;
   while ( u ≠ null );
   do
        Sv.PUSH(v);
        v := v.parent;
   while ( v ≠ null );
   nodo LCA;
   while( !Su.ISEMPTY() && !Sv.ISEMPTY() && Su.TOP() == Sv.TOP() ) do
        LCA := Su.TOP();
        Su.POP();
        Sv.POP();
   endwhile
   return LCA;
```

Il costo dell'algoritmo è dominato dal costo necessario per riempire gli stack, che nel caso peggiore corrisponde alla profondità massima dell'albero. Poiché nel caso peggiore un albero di n nodi ha profondità n-1 (l'albero degenera in una catena), si ha che il costo dell'algoritmo nel caso pessimo è O(n). Il caso migliore si verifica quando uno dei due nodi è la radice e l'altro è uno dei suoi figli; in questo caso il costo sarà O(1).

Esercizio 2.27

E' dato un array $A[1..n]$ composto da n numeri reali arbitrari diversi da zero. Vogliamo calcolare un secondo array $B[1..n]$, in cui $B[i] = A[1] * A[2] * \ldots * A[n] / A[i]$. Scrivere un algoritmo di costo O(n) per calcolare tutti i valori dell'array $B[1..n]$, senza effettuare divisioni e utilizzando spazio aggiuntivo O(1). (*Questo esercizio è stato proposto da Google durante i colloqui di assunzione*).

Soluzione. Notiamo innanzitutto che semplificando l'espressione del valore di $B[i]$, si ha che $B[i]$ corrisponde al prodotto di tutti gli elementi di A tranne $A[i]$



Iniziamo esaminando una soluzione che richiede tempo O(n) e spazio aggiuntivo O(n); vedremo poi come ridurre il consumo di memoria.
Definiamo due array $P[1..n]$ e $S[1..n]$. $P[i]$ contiene il prodotto dei valori del sottovettore $A[1..i\text{-}1]$; $P[1]$ vale 1. $S[i]$ contiene il prodotto dei valori del sottovettore $A[i+1..n]$; $S[n]$ vale 1. Una volta che i valori di $P[1..n]$ e $S[1..n]$ sono stati calcolati, si ha che $B[i] := P[i] * S[i]$. Il calcolo dei valori $P[i]$ e $S[i]$ si può effettuare in tempo complessivo O(n).

```
algoritmo VETTOREB( array A[1..n] di double ) → array[1..n] di double
   array P[1..n] di double;
   array S[1..n] di double;
```

```
array B[1..n] di double;
P[1] := 1;
for i := 2 to n do
    P[i] := P[i-1] * A[i-1];
endfor
S[n] := 1;
for i := n-1 downto 1 do
    S[i] := S[i+1] * A[i+1];
endfor
for i := 1 to n do
    B[i] := P[i] * S[i];
endfor
return B;
```

La soluzione precedente ha lo svantaggio di richiedere spazio aggiuntivo $\Theta(n)$. In realtà possiamo fare a meno degli array P e S, usando al loro posto due variabili reali P e S nel modo seguente:

```
algoritmo VettoreB( array A[1..n] di double ) → array[1..n] di double
  double P, S;
  array B[1..n] di double;
  P := 1;
  B[1] := 1;
  for i := 2 to n do
      P := P * A[i-1];
      B[i] := P;
  endfor
  S := 1;
  for i := n-1 downto 1 do
      S := S * A[i+1];
      B[i] := B[i] * S;
  endfor
  return B;
```

Esercizio 2.28

Si consideri una stringa memorizzata in un array $S[1..n]$ di $n \geq 1$ caratteri. La stringa S si dice palindroma se è uguale alla sua versione letta a rovescio. Ad esempio, l'array ['B', 'A', 'N', 'A', 'B'] rappresenta una stringa palindroma, mentre ['A', 'N', N', 'I'] no. Per controllare se una stringa $S[1..n]$ è palindroma è sufficiente verificare che il primo carattere risulti uguale all'ultimo, il secondo uguale al penultimo e così via.
Scrivere un algoritmo ricorsivo che, dato in input un array di caratteri $S[1..n]$, restituisce true se e solo se la stringa S è palindroma. Scrivere l'equazione di ricorrenza per il tempo di esecuzione $T(n)$ dell'algoritmo in funzione di n.

Soluzione. Definiamo l'algoritmo Palindroma(S, i, j) che restituisce true se e solo se la stringa $S[i..j]$ è palindroma. Una stringa è palindroma se è composta da al più un singolo carattere, oppure se valgono entrambe le condizioni seguenti:

1. il primo carattere è uguale all'ultimo ($S[i] = S[j]$), e
2. la stringa $S[i+1,..j-1]$ ottenuta ignorando il primo e l'ultimo carattere di $S[i..j]$ è essa stessa palindroma.

```
algoritmo Palindroma(array S[1..n] di char, int i, int j) → bool
  if ( i ≥ j ) then
      return true;
  else
      return ( S[i] == S[j] ) && Palindroma(S, i+1, j-1);
  endif
```

L'algoritmo viene invocato con PALINDROMA(S, 1, n). Osserviamo che la ricorsione avviene su una sottostringa che ha due caratteri in meno della stringa di partenza; il costo computazionale dell'algoritmo PALINDROMA è quindi dato dalla relazione di ricorrenza seguente:

$$T(n)=\begin{cases} c_1 & \text{se } n\leq 1 \\ T(n-2)+c_2 & \text{altrimenti} \end{cases}$$

Mediante il metodo della sostituzione si può verificare che la soluzione è $T(n) = O(n)$.

Esercizio 2.29

Scrivere un algoritmo che date in input due stringhe di lunghezza n rappresentate mediante due array di caratteri $A[1..n]$ e $B[1..n]$, restituisce *true* se e solo se le due stringhe sono una l'anagramma dell'altra. In altre parole, l'algoritmo deve restituire true se e solo se entrambe le stringhe contengono gli stessi caratteri, eventualmente in ordine differente. Ad esempio, A = ['b', 'a', 'r', 'c', 'a'] e B = ['c', 'r', 'a', 'b', 'a'] sono anagrammi. mentre A = ['s', 'o', 'l', 'e'] e B = ['e', 'l', 's', 'a'] non lo sono. Calcolare il costo computazionale dell'algoritmo proposto.

Soluzione. Iniziamo proponendo una soluzione non efficiente ma corretta, che si basa sul modo in cui potremmo operare per verificare manualmente. L'idea consiste nell'esaminare uno per uno i caratteri di A, verificando che siano presenti anche in B. Visto che ogni carattere può comparire in A più volte, è necessario tener conto di quali caratteri di B sono già stati "consumati", in modo da evitare di considerarli più di una volta. Allo scopo usiamo un array di booleani *flag*[1..*n*]; *flag*[*j*] = 1 se e solo se il *j*-esimo carattere di B è già stato considerato. Durante la fase di ricerca, gli elementi di B i cui flag sono settati a *true* devono essere saltati.

```
algoritmo ANAGRAMMI(array A[1..n] di char, array B[1..n] di char) → bool
    array flag[1..n] di bool;
    // Inizializzazione dell'array flag[1..n] a false
    for i := 1 to n do
        flag[i] := false;
    endfor
    for i := 1 to n do
        // cerchiamo il carattere A[i] in B
        int j := 1;
        while ( j ≤ n && (A[i] ≠ B[j] || flag[j] = true) ) do
            j := j + 1;
        endwhile
        if ( j > n ) then
            return false;          // Il carattere A[i] non esiste in B
        else
            flag[j] := true;       // il carattere A[i] esiste in B alla posizione j
        endif
    endfor
    return true;
```

Il costo dell'algoritmo è $O(n^2)$ nel caso peggiore, e $\Theta(n)$ nel caso migliore, che si verifica quando il primo carattere di A non è presente in B.
Esiste però una soluzione più efficiente che si basa sull'idea seguente: due stringhe sono una l'anagramma dell'altra se e solo se, ordinando i caratteri, esse risultano uguali. Ad esempio, considerando le stringhe 'barca' e 'craba', ordinando i caratteri otteniamo in entrambi i casi 'aabcr', da cui possiamo concludere che sono una l'anagrramma dell'altro.

Stringa originale	**Stringa ordinata**
barca	aabcr

craba	aabcr

Ordinare i caratteri di ciascuna delle due stringhe richiede tempo O(n log n) usando un algoritmo di ordinamento generico; in realtà, è possibile impiegare un algoritmo di ordinamento in tempo lineare, dato che i caratteri alfabetici sono in numero finito. Il successivo controllo dell'uguaglianza delle versioni ordinate richiede tempo O(n).

Esercizio 2.30

Dato un array $A[1..n]$ e un intero $0 \le k < n$, una k-rotazione a sinistra di A è il vettore $A[k+1]$. $A[k+2]$, … $A[n]$, $A[1]$, $A[2]$, … $A[k]$ ottenuto spostando a sinistra i suoi elementi di k posizioni, e reinserendo ciascuno di essi alla fine. Ad esempio, una 3-rotazione a sinistra del'array ABCDEFGH produce l'array DEFGHABC.
Proporre un algoritmo per ruotare un array in tempo O(n) usando spazio ulteriore O(1).

Soluzione. La soluzione più semplice sfrutta un array temporaneo tmp in cui copiare i valori di A nelle posizioni opportune.

```
RuotaSinistra( array A[1..n] di double, int k )
    array tmp[1..n] di double;
    int i;
    for i := k + 1 to n do
        tmp[i - k] := A[i];
    endfor
    for i := 1 to k do
        tmp[n – k + i] := A[i];
    endfor
    A := tmp;     // copia tmp in A
```

L'algoritmo richiede spazio Θ(n): Una soluzione molto elegante (ma tutt'altro che ovvia) per ruotare un array usando spazio O(1) si basa sulla seguente osservazione. Supponiamo che X sia il sottovettore composto dai primi k elementi di A, e Y il sottovettore composto dagli elementi rimanenti. Risulta quindi $A = XY$. Applichiamo ora una funzione swap ai due sottovettori X e Y; la funzione swap non fa altro che "rovesciare" ciascuno dei due sottovettori scambia il primo elemento con l'ultimo, il secondo con il penultimo e così via. In questo modo otteniamo il vettore $X^R Y^R$. Rovesciamo nuovamente l'intero array ottenendo $(X^R Y^R)^R = YX$, che è esattamente il risultato cercato.
Il nuovo algoritmo RuotaSinistra si basa su una funzione Swap(A, i, j) che rovescia il sottovettore $A[i..j]$.

```
Swap( array A[1..n] di double, int i, int j )
    while ( i < j ) do
        real tmp := A[i];
        A[i] := A[j];
        A[j] := tmp;
        i := i + 1;
        j := j + 1;
    endwhile

RuotaSinistra( array A[1..n] di double, int k )
    swap( A, 1, k );
    swap( A, k + 1, n );
    swap( A, 1, n );
```

L'algoritmo così ottenuto ha costo Θ(n) e richiede spazio ulteriore O(1) come richiesto dall'esercizio.

Esercizio 2.31

Una delle operazioni più comuni nei software di elaborazione delle immagini è la possibilità di ruotare una immagine di 90 gradi. Supponiamo che l'immagine sia codificata in una matrice quadrata $M[1..n, 1..n]$ di numeri interi; scrivere un algoritmo per ruotare i valori della matrice M di 90 gradi in senso orario. Ad esempio, la matrice

$$\begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix}$$

deve diventare

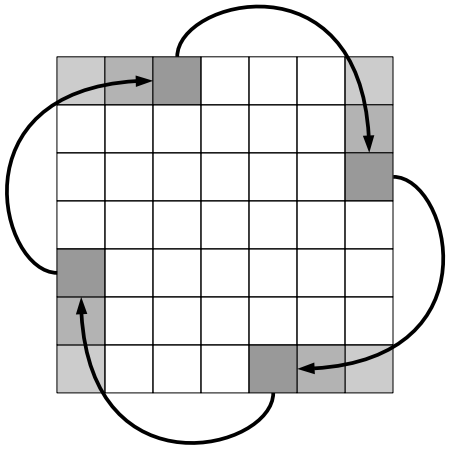
$$\begin{pmatrix} G & D & A \\ H & E & B \\ I & F & C \end{pmatrix}$$

Riuscite a risolvere il problema usando pazio aggiuntivo O(1)?

Soluzione. La soluzione più semplice richiede spazio aggiuntivo $\Theta(n^2)$ per memorizzare una matrice temporanea $R[1..n, 1..n]$ che alla fine conterrà il risultato. L'algoritmo RuotaMatrice copia la prima riga di M nell'ultima colonna di R, la seconda riga di M nella penultima colonna di R e così via. In generale, la riga i-esima di M viene copiata nella colonna $(n + 1 – i)$-esima di R.

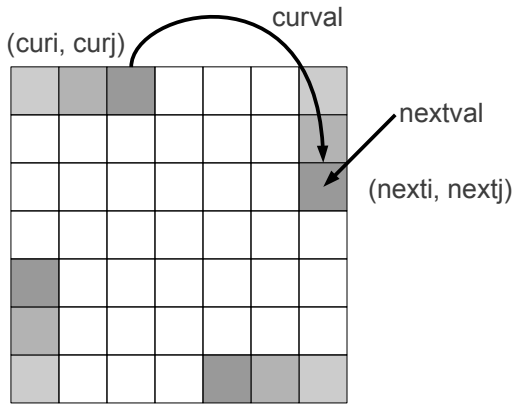
```
RuotaMatrice( array M[1..n, 1..n] di int )
   array R[1..n, 1..n] di int;
   for i := 1 to n do
        for j := 1 to n do
             R[ j, n + 1 – i ] := M[i, j];
        endfor
   endfor
   M := R;       // copia R in M
```

Il costo asintotico dell'algoritmo è $\Theta(n^2)$, in quanto il corpo del ciclo “for” più interno viene eseguito complessivamente n^2 volte.
Per ridurre lo spazio richiesto è necessario effettuare la rotazione scambiando direttamente gli elementi della matrice M secondo lo schema seguente:



Formalmente, l'elemento in posizione (i, j) deve essere spostato nella nuova posizione $(j, n + 1 - i)$; tale

operazione deve essere ripetuta quattro volte, in modo che al termine l'ultimo elemento vada ad occupare il posto lasciato libero dal primo. Nello pseudocodice facciamo uso delle variabili curi, curj per indicare le coordinate dell'elemento corrente, e nexti, nextj per indicare le coordinate dell'elemento successivo nella catena di scambi. curval denota il valore che dovrà essere scritto nella cella di posizione (nexti, nextj), mentre nextval contiene una copia del vecchio valore presente in tale cella. si noti che è necessario salvare il contenuto di *M*[nexti, nextj] prima di sovrascriverlo con curval, in modo da poterlo a sua volta copiare nella cella successiva.




```
RuotaMatrice( array M[1..n, 1..n] di int )
   for i := 1 to Floor(n / 2) do
        for j := i to n - i do
             int curi := i, curj := j, nexti, nextj, curval := M[i, j];
             for k := 1 to 4 do
                  nexti := curj;
                  nextj := n + 1 - curi;
                  int nextval := M[nexti, nextj];
                  M[nexti, nextj] := curval;
                  curval := nextval;
                  curi := nexti;
                  curj := nextj;
             endfor
        endfor
   endfor
```

Esercizio 2.32

Scrivere un algoritmo efficiente che, dato in input una lista singolarmente concatenata (ogni nodo contiene un riferimento all'elemento successivo ma non al precedente), restituisce un riferimento al *k*-esimo elemento a partire dalla fine. Ad esempio, se $k = 1$ l'algoritmo deve restituire un riferimento all'ultimo elemento; se $k = 2$ al penultimo, e così via. L'algoritmo dovrebbe effettuare una singola scansioni della lista, e utilizzare spazio aggiuntivo O(1).

Soluzione. La prima soluzione che probabilmente viene in mente consiste nell'effettuare due scansioni della lista: la prima per contare il numero *n* di elementi presenti, e la seconda per accedere all'elemento in posizione $n + 1 - k$. Con un po' di astuzia è però possibile risolvere il problema con una singola scansione. L'idea consiste nello scorrere la lista utilizzando due cursori, che chiamiamo last1 e lastk. Durante la scansione manteniamo la proprietà che il cursore lastk punti all'elemento che si trova *k* posizioni più indietro rispetto al cursore last1. I due cursori vengono fatti avanzare contemporaneamente fino a quando last1

diventerà uguale a NULL, avendo quindi raggiunto la fine della lista. A questo punto il secondo cursore conterrà il riferimento all'elemento desiderato.

```
ListaKUltimo( List L, int k ) → List
    List last1 := L, lastk := L;
    int i := 0;
    while ( last1 ≠ NULL and i < k ) do
        last1 := last1.next;
        i := i + 1;
    endwhile
    if ( i < k ) then
        errore "ci sono meno di k elementi nella lista"
    endif
    while ( last1 ≠ NULL ) then
        last1 := last1.next;
        lastk := lastk.next;
    endwhile
    return lastk;
```

Capitolo 3: Ordinamento e Ricerca

Esercizio 3.1

Considerare la seguente variante dell'algoritmo SELECTIONSORT:

```
algoritmo MIOSELECTIONSORT( array A[1..n] di double )
   for k := 1 to n-1 do
        int m := k;
        for j := k+1 to n do
             if ( A[j] ≤ A[m] ) then
                  m := j;
             endif
        endfor
        // scambia A[k] con A[m];
        double tmp := A[k];
        A[k] := A[m];
        A[m] := tmp;
   endfor
```

Ricordiamo che un algoritmo di ordinamento è *stabile* se preserva l'ordinamento relativo degli elementi uguali presenti nell'array da ordinare.
L'algoritmo MIOSELECTIONSORT è un algoritmo di ordinamento stabile? In caso negativo, mostrare come modificarlo per farlo diventare stabile. Motivare le risposte.

Soluzione. L'algoritmo MIOSELECTIONSORT è quasi identico all'algoritmo SELECTIONSORT descritto nel libro di testo, con la differenza che la condizione è stata scritta come "if ($A[j] \leq A[m]$)" (cioè usando l'operatore minore o uguale). Questo fa sì che MIOSELECTIONSORT non sia un algoritmo di ordinamento stabile, perché in caso di elementi uguali mette in prima posizione l'ultimo elemento trovato. Per renderlo stabile è sufficiente riscrivere l'if come "if ($A[j] < A[m]$)"

Esercizio 3.2

Dato un array $A[1..n]$ di $n \geq 3$ elementi confrontabili, tutti distinti, descrivere un algoritmo che in tempo O(1) determina un elemento qualsiasi dell'array che non sia né il minimo né il massimo.
A cosa può servire un algoritmo del genere? Ovvero, esiste qualche problema che abbiamo affrontato nel corso la cui soluzione potrebbe beneficiare da questo algoritmo?

Soluzione. Per risolvere il problema basta considerare i primi tre elementi di A, ossia $A[1]$, $A[2]$ e $A[3]$. Si cercano il massimo e il minimo tra questi tre, e si restituisce quello tra di loro che non è né massimo né minimo. Un tale elemento esiste sempre, perché per ipotesi tutti gli elementi sono distinti.

```
algoritmo NEMINNEMAX( array A[1..n] di elem ) → elem
```

```
int min12;          // valore minimo tra A[1] e A[2]
int max12;          // valore massimo tra A[1] e A[2]
if ( A[1] < A[2] ) then
    min12 := A[1];
    max12 := A[2];
else
    min12 := A[2];
    max12 := A[1];
endif
if ( A[3] < min12 ) then
    return min12;
elseif ( A[3] < max12 ) then
    return A[3];
else
    return max12;
endif
```

Si osservi che le variabili min12 e max12 contengono rispettivamente il valore minimo e massimo tra $A[1]$ e $A[2]$.
Un simile algoritmo può essere utilizzato per scegliere il pivot (in modo deterministico) in modo da evitare il caso pessimo nell'algoritmo QUICKSORT e QUICKSELECT, assumendo che tutti i valori in input siano distinti.

Esercizio 3.3

Un collezionista di figurine vi ha incaricati di risolvere il seguente problema. Il collezionista possiede K figurine, non necessariamente distinte. Le figurine vengono stampate con un numero di serie, impresso dal produttore. I numeri di serie sono tutti gli interi compresi tra 1 e N. La collezione sarebbe completa se ci fosse almeno un esemplare di tutte le N figurine prodotte. Purtroppo la collezione non è completa: sono presenti dei duplicati, mentre alcune figurine mancano del tutto. Il vostro compito è di indicare i numeri di serie delle figurine mancanti.

1. Scrivere un algoritmo efficiente che dato N e l'array $S[1..K]$, ove $S[i]$ è il numero di serie della i-esima figurina posseduta, stampa l'elenco dei numeri di serie mancanti. L'array S non è ordinato. Ad esempio, se N=10 e S=[1, 4, 2, 7, 10, 2, 1, 4, 3], l'algoritmo deve stampare a video i numeri di serie mancanti 5, 6, 8, 9 (non necessariamente in questo ordine)
2. Determinare il costo computazionale dell'algoritmo di cui sopra, in funzione di N e K. Attenzione: K potrebbe essere minore, uguale o maggiore di N.

Soluzione. Sfruttando il suggerimento, implementiamo l'idea che segue: definiamo un array booleano $B[1..N]$, inizialmente con tutti gli elementi posti a false. Poniamo poi $B[k] = true$ se e solo se il collezionista possiede una figurina con numero di serie k. Al termine dell'algoritmo, i valori di k per cui $B[k] = false$ rappresentano i numeri di serie delle figurine mancanti. L'algoritmo può essere descritto come segue:

```
algoritmo CercaFigurineMancanti( int N, array S[1..K] di int )
  array B[1..N] di bool
  // (1) Inizializza l'array B
  for i := 1 to N do
      B[i] := false;
  endfor
  // (2) Controlla i numeri di serie delle figurine possedute
  for i := 1 to K do
      B[S[i]] := true;
  endfor
  // (3) Stampa i numeri di serie mancanti
  for i := 1 to N do
      if ( B[i] == false ) then
          print “manca il numero di serie “ i;
      endif
  endfor
```

Analizziamo il costo computazionale dell'algoritmo. Il primo ciclo ha costo $\Theta(N)$; il secondo ha costo $\Theta(K)$; il terzo ha costo $\Theta(N)$. Il costo complessivo è quindi $\Theta(2N+K) = \Theta(N+K)$.

Esercizio 3.4

Si consideri un array $A[1..n]$ di numeri reali, non necessariamente distinti. Supponiamo che esista un intero i, $1 \le i < n$ tale che il sottovettore $A[1..i]$ sia ordinato in senso crescente, mentre il sottovettore $A[i+1..n]$ sia ordinato in senso decrescente.

1. Descrivere un algoritmo efficiente che, dato in input l'array $A[1..n]$ e il valore i, ordina A in senso crescente (è possibile utilizzare un array temporaneo durante la procedura di ordinamento).
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Il problema si risolve con una semplice variazione della procedura MERGE vista per l'algoritmo MERGESORT:

```
algoritmo OrdinaBitonica( array A[1..n] di double, int i ) → array
    array B[1..n] di double;
    int i1 := 1;
    int i2 := n;
    int j := 1; // indice in B
    while ( i1 ≤ i && i2 > i ) do
        if ( A[i1] < A[i2] ) then
            B[j] := A[i1];
            i1 := i1+1;
            j := j+1;
        else
            B[j] := A[i2];
            i2 := i2-1;
            j := j+1;
        endif
    endwhile
    while ( i1≤i ) do
        B[j] := A[i1];
        i1 := i1+1;
        j := j+1;
    endwhile
    while ( i2 > i ) do
        B[j] := A[i2];
        i2 := i2-1;
        j := j+1;
    endwhile
    return B;
```

Il costo computazionale di tale algoritmo è $\Theta(n)$

Esercizio 3.5

Si consideri il seguente algoritmo di ordinamento, che viene inizialmente invocato con la chiamata stoogesort(L, 1, n):.

```
algorithm StoogeSort( array L[1..n], int i, int j )
    if (j ≥ i) then
        return;
    endif
    if (L[j] < L[i]) then
        scambia L[i] con L[j];
    endif
```

```
if ((j - i) > 1) then
    int t = (j - i + 1)/3;
    StoogeSort(L, i  , j-t);
    StoogeSort(L, i+t, j  );
    StoogeSort(L, i  , j-t);
endif
```

1. Descrivere a parole come funziona l'algoritmo StoogeSort
2. Calcolare il costo computazionale asintotico di StoogeSort(L, 1, n) in funzione di n

Soluzione. L'algoritmo opera nel modo seguente: dopo aver sistemato nell'ordine corretto gli elementi iniziali e finale dell'intervallo $L[i..j]$, effettua tre chiamate ricorsive. Nella prima ordina gli elementi appartenenti ai primi 2/3 del sottovettore $L[i..j]$; nella seconda ordina gli elementi appartenenti agli ultimi 2/3 del sottovettore (modificato) $L[i..j]$; infine nella terza chiamata effettua nuovamente l'ordinamento degli elementi appartenenti ai primi 2/3 del sottovettore (modificato) $L[i..j]$. Quindi l'algoritmo effettua tre chiamate ricorsive, ciascuna su un input che ha lunghezza pari a 2/3 quella dell'input originale.
L'equazione di ricorrenza associata è la seguente:

$$T(n)=\begin{cases} c_1 & \text{se } n\le 2 \\ 3\,T\left(\frac{2}{3}n\right)+c_2 & \text{altrimenti} \end{cases}$$

A questo punto si può applicare il caso 1) del Master Theorem, con a=3, b=3/2 (attenzione, NON b=2/3), ottenendo

$$T(n)=\Theta\left(n^{\log_b a}\right)\approx\Theta\left(n^{2.7095}\right)$$

da cui si deduce che questo algoritmo di ordinamento non è molto efficiente.

Esercizio 3.6

Si consideri un array $A[1..n]$, non vuoto, contenente tutti gli interi appartenenti all'insieme $\{1, 2, \ldots, n+1\}$, tranne uno. L'array A è ordinato in senso crescente.

1. Scrivere un algoritmo efficiente che, dato l'array $A[1..n]$, determina il valore mancante. Ad esempio, se A=[1, 2, 3, 4, 6, 7], l'algoritmo deve restituire il valore 5.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Una soluzione non efficiente consiste nell'esaminare il vettore A dall'inizio; non appena si incontra il primo valore i tale che $A[i] \neq i$, si può concludere che il valore mancante è i. Lo pseudocodice è il seguente:

```
algoritmo TrovaValoreMancante( array A[1..n] di int ) → int
  for i:=1 to n do
      if ( A[i] != i ) then
          return i;
      endif
  endfor
  // Se siamo arrivati qui, significa che il valore mancante è n+1
  return n+1;
```

Il costo dell'algoritmo TrovaValoreMancante1 è O(n); in particolare, risulta Θ(n) nel caso pessimo (che si verifica quando il valore mancante è n+1) e O(1) nel caso oggimo (che si verifica quando il valore mancante è 1).
Sfruttando il fatto che l'array è ordinato, possiamo definire una soluzione più efficiente seguendo il suggerimento di utilizzare una variante della ricerca binaria. L'idea è la seguente: si individua la posizione

centrale m nel sottovettore $A[i..j]$. Se $A[m]$ contiene il valore m, allora sicuramente non ci sono elementi mancanti nella prima metà del sottovettore, e la ricerca continua in $A[m+1..j]$. Invece, se $A[m] \neq m$, la ricerca prosegue nella prima metà $A[i..m-1]$. Si noti che in quest'ultimo caso il valore mancante potrebbe essere m. L'aspetto cruciale di questo algoritmo è la gestione del caso base della ricorsione: se $i > j$ possiamo concludere che il valore mancante è $j+1$.

```
algoritmo TrovaValoreMancante( array A[1..n] di int, int i, int j ) → int
    if ( i > j ) then
        return i;
    else
        int m := ( i + j ) / 2;
        if ( A[m] == m ) then
            return TrovaValoreMancante(A, m + 1, j);
        else
            return TrovaValoreMancante(A, i, m - 1);
        endif
    endif
```

La precondizione di questo algoritmo, cioè la proprietà di cui devono godere i parametri di input, è la seguente:

$$\begin{array}{rll} A[k]=k & \text{per ogni} & k=1,\ldots i-1 \\ A[k]=k+1 & \text{per ogni} & k=j+1,\ldots n \end{array}$$

In altre parole, quando la funzione viene invocata, tutti gli elementi $A[k]$ che si trovano "a sinistra" della posizione i devono essere tali che $A[k] = k$; invece, tutti gli elementi $A[m]$ che si trovano "a destra" della posizione j devono essere tali che $A[k] = k+1$. Questa considerazione ci aiuta a definire cosa deve restituire l'algoritmo nel caso baso, cioè quando $i > j$. Se $i = j+1$ possiamo riscrivere la precondizione come:

$$\begin{array}{rll} A[k]=k & \text{per ogni} & k=1,\ldots i-1 \\ A[k]=k+1 & \text{per ogni} & k=i,\ldots n \end{array}$$

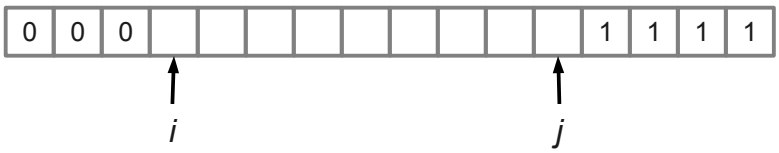
da cui risulta immediatamente che il valore mancante è i.
L'algoritmo viene invocato con TrovaValoreMancante(A, 1, n); il costo computazionale è O(log n), esattamente come per la ricerca binaria.

Esercizio 3.7

Si consideri un array $A[1..n]$ di interi in cui ciascun valore può essere esclusivamente 0 oppure 1. I valori sono presenti senza alcun ordine; è anche possibile che tutti i valori siano uguali.

1. Scrivere un algoritmo avente complessità ottima che ordini l'array A spostando tutti i valori 0 prima di tutti i valori 1. Verrà assegnato punteggio pieno ad algoritmi che scambiano elementi in A, che richiedono memoria ulteriore O(1) (quindi che non sfruttano array ausiliari), e che non sono basati sul conteggio del numero di 0 e 1 presenti.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

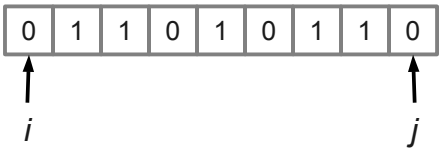
Soluzione. Questo problema è una versione più semplice del problema della "bandiera nazionale" descritto nell'Esercizio 3.12. Sfruttiamo due cursori i e j, tali che i si sposta dal primo elemento del vettore A e avanza verso la coda, mentre j si sposta dall'ultimo elemento di A e avanza verso la testa. Il nostro algoritmo esegue un ciclo incrementando ripetutamente i e/o decrementando j. Durante il ciclo, manterremo la proprietà che il sottovettore $A[1..i-1]$ è composto da tutti valori 0, il sottovettore $A[j+1..n]$ è composto da tutti valore 1, mentre il sottovettore rimanente $A[i..j]$ può contenere sia zeri che uni.



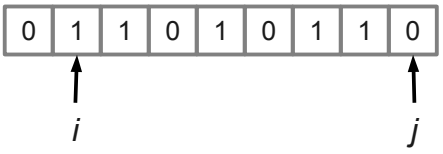
Ad ogni iterazione, esaminiamo dapprima $A[i]$. Se vale 0, si trova già dalla parte "giusta" del vettore e possiamo incrementare i passando alla prossima iterazione. Se $A[i]$ vale 1, allora esaminiamo $A[j]$; se quest'ultimo vale 1, allora si trova già dalla parte "giusta" del vettore e decrementiamo j passando alla prossima iterazione. Se le condizioni precedenti sono entrambe false significa che $A[i] == 1$ e $A[j] == 0$. In tal caso i valori si trovano dalla parte "sbagliata" del vettore, ma possiamo ripristinarli nelle posizioni corrette scambiandoli tra loro. Dopo aver effettuato lo scambio avremo $A[i] == 0$ e $A[j] == 1$, al che potremo incrementare i e decrementare j passando all'iterazione successiva.

```
algoritmo ORDINA01( array A[1..n] di int )
   int i := 1;
   int j := n;
   while ( i < j ) do
        if ( A[i] == 0 ) then
             i := i+1;
        elseif ( A[j] == 1 ) then
             j := j-1;
        else     // A[i] == 1 e A[j] == 0; scambiamo tra loro A[i] e A[j] per ripristinare l'ordine
             int tmp := A[i];
             A[i] := A[j];
             A[j] := tmp;
             i := i + 1;
             j := j - 1;
        endif
   endwhile
```

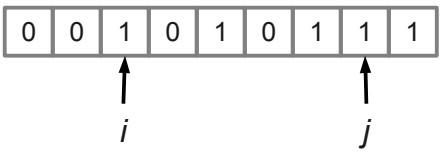
Analizziamo passo dopo passo il comportamento dell'algoritmo su questo semplice esempio:



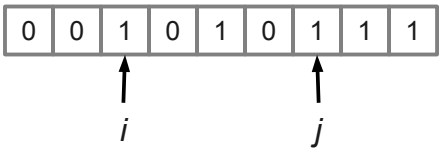
Poiché $A[i] == 0$, incrementiamo i



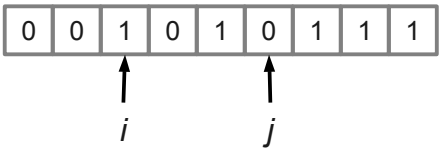
Abbiamo che $A[i] == 1$ e $A[j] == 0$: scambiamo $A[i]$ e $A[j]$, e facciamo avanzare entrambi i cursori verso il centro



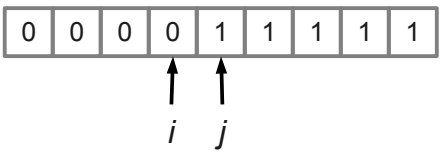
Poiché $A[j] == 1$, decrementiamo j



$A[j] == 1$, decrementiamo j



$A[i] == 1$ e $A[j] == 0$, scambiamo $A[i]$ e $A[j]$ e facciamo avanzare entrambi i cursori verso il centro



$A[i] == 0$, incrementiamo i; a questo punto $i==j$, e l'algoritmo termina.
Per calcolare il costo computazionale, osserviamo che ad ogni iterazione necessariamente incrementiamo i oppure decrementiamo j (o entrambi). Quindi dopo $\Theta(n)$ iterazioni i due cursori i e j si incroceranno e l'algoritmo termina.

Esercizio 3.8

Si considerino due array $A[1..n]$ e $B[1..m]$, contenenti n ed m valori interi, rispettivamente. Entrambi gli array sono ordinati in senso crescente, e ciascuno dei due contiene elementi distinti. È però possibile che lo stesso valore sia presente sia in A che in B.

1. Scrivere un algoritmo di complessità ottima per stampare i valori di A che **non** sono presenti in B. Ad esempio, se A=[1, 3, 4, 7, 8] e B=[2, 3, 5, 7], l'algoritmo deve stampare 1, 4, 8 (infatti i rimanenti valori di A, 3 e 7, sono presenti anche in B quindi NON devono essere stampati).
2. Analizzare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. È possibile risolvere il problema effettuando una singola scansione dei due array, sfruttando il fatto che entrambi sono ordinati. In particolare, usiamo due cursori i e j per scorrere A e B, rispettivamente. Ad ogni iterazione confrontiamo $A[i]$ e $B[j]$. Se $A[i] < B[j]$, sicuramente $A[i]$ non compare in B e può quindi essere stampato; facciamo quindi avanzare i. Se $A[i] > B[j]$, facciamo avanzare j e **non** stampiamo $A[i]$ (infatti non siamo sicuri che il valore $A[i]$ non comparirà nei prossimi elementi di B). Infine, se $A[i]$ è uguale a $B[j]$, facciamo avanzare entrambi i cursori e non stampiamo nulla. Osserviamo che, arrivando alla fine del vettore B, dobbiamo ricordarci di stampare tutti gli elementi residui di A.

```
algoritmo DIFFERENZA( array A[1..n] di int, array B[1..m] di int )
    int i := 1;
    int j := 1;
    while ( i ≤ n && j ≤ m ) do
        if (A[i] < B[j]) then
            print A[i];
            i :=i + 1;
        elseif (A[i] > B[j]) then
            j := j + 1;
        else
            i := i + 1;
            j := j + 1;
        endif
    endwhile
    while ( i ≤ n ) do
        print A[i];
        i := i + 1;
    endwhile
```

Ad ogni passo, l'algoritmo avanza di una posizione in A oppure in B (oppure in entrambi i vettori). Quindi il costo computazionale è $O(n+m)$, in quanto è possibile effettuare al più $n+m$ iterazioni complessivamente. Un esempio di input in cui si verifica il caso peggiore è il seguente:
$A = [1, 3, 5, 7, 9]$
$B = [2, 4, 6]$

Esercizio 3.9

Si consideri un array $A[1..n]$ non vuoto che contiene tutti gli interi nell'insieme $\{1, 2, \ldots, n+2\}$, tranne due. L'array non è ordinato. L'array A non contiene valori duplicati.

1. Scrivere un algoritmo di complessità ottima in grado di stampare i due numeri mancanti.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Usiamo una variante dell'algoritmo counting sort; utilizziamo un array ausilirio $B[1..n+2]$ inizializzato a false, e settiamo $B[x]$ a true se il valore x è presente in A:

```
algoritmo STAMPAMANCANTI( array A[1..n] di int )
    array B[1..n+2] di bool;
    for i := 1 to n+2 do
        B[i] := false; // B[i] = true indica che il numero i è presente in A
    endfor
    for i:=1 to n do
        B[A[i]] := true;
    endfor
    for i := 1 to n+2 do
        if ( B[i] == false ) then
            print i;
        endif
    endfor
```

Il costo computazionale è $\Theta(n)$

Esercizio 3.10

Si consideri un array $A[1..n]$ di n numeri reali distinti e un intero $k \in \{1, \ldots, n\}$; l'array non è ordinato.

1. Scrivere un algoritmo che, dati A e k, restituisca o stampi i k valori più piccoli presenti in A. Il costo dell'algoritmo proposto deve essere al più $O(kn)$. Se ritenuto necessario, è possibile fare uso anche di funzioni ausiliarie viste a lezione senza bisogno di scriverne lo pseudocodice.

2. Calcolare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Esistono diverse soluzioni a questo problema. Una soluzione non molto efficiente consiste nell'utilizzare un algoritmo di tipo SelectionSort, modificato per fermarsi dopo aver ordinato i primi k elementi di A. Lo pseudocodice è il seguente:

```
algoritmo PrimiKmin1( array A[1..n] di double, int k )
  // Si assume A non vuoto, e 1≤k≤n
  for i := 1 to k do
      int imin := i;
      for j := i+1 to n do
          if ( A[j] < A[imin] ) then
              imin := j;
          endif
      endfor
      // scambia A[imin] con A[i]
      double tmp := A[imin];
      A[imin] := A[i];
      A[i] := tmp;
      print A[i];
  endfor
```

Il costo dell'algoritmo è $\Theta(nk)$, che è compatibile con quanto richiesto dall'esercizio. Una soluzione diversa e leggermente più efficiente consiste nel creare un min-heap a partire dall'array A, ed estrarre k volte l'elemento minimo. Lo pseudocodice è il seguente (e ricorda molto l'algoritmo heapselect visto a lezione)

```
algoritmo PrimiKmin2( array A[1..n] di double, int k )
  MinHeapify(A);
  for i := 1 to k do
      v := FindMin(A);
      print v;
      DeleteMin(A);
  endfor
```

L'operazione MinHeapify(A) ha costo $O(n)$, mentre invocare k volte le operazioni FindMin (di costo $O(1)$) e DeleteMin (di costo $O(\log n)$) ha un costo complessivo di $O(k \log n)$. Il costo totale risulta quindi essere $O(n + k \log n)$.

Esercizio 3.11

Scrivere un algoritmo efficiente che, dato in input un array $A[1..n]$ di $n \geq 1$ numeri reali, permuta A in modo tale che tutti i valori negativi compaiano prima di tutti i valori positivi. Non è richiesto che i valori positivi e negativi siano a loro volta ordinati. Ad esempio, se A=[-3, 2, 4, 6, -3, -4], l'algoritmo puo' restituire, ad esempio, il vettore [-3,-3,-4,2,4,6] oppure il vettore [-3,-4,-3,6,2,4].

Esercizio 3.12

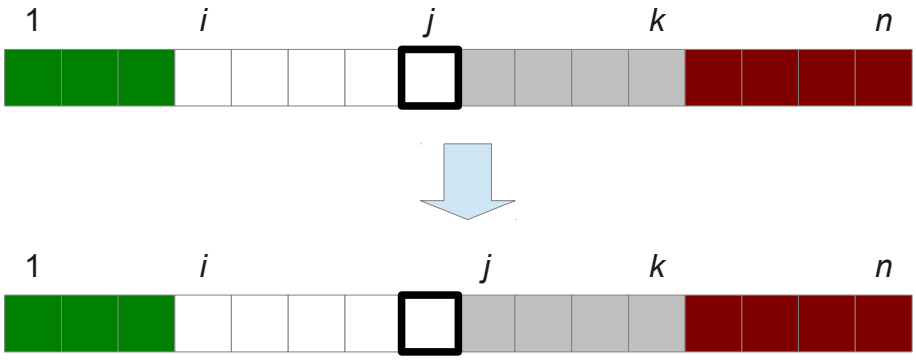
Il problema seguente fu inizialmente proposto da Edsger W. Dijkstra come *Dutch National Flag Problem*; ne proponiamo una rivisitazione in chiave italiana. Sia $A[1..n]$ un array di caratteri, i cui elementi assumono valori nell'insieme {'B', 'R', 'V'} ("bianco", "rosso" e "verde"). Scrivere un algoritmo efficiente che permuta gli elementi di A in modo tale che tutti i valori 'V' precedano tutti i valori 'B', i quali a loro volta precedano tutti i valori 'R'. L'algoritmo dovrebbe richiedere spazio aggiuntivo O(1), quindi dovrebbe limitarsi a permutare tra loro gli elementi di A senza usare un secondo array di appoggio.

Soluzione. Utilizziamo un algoritmo iterativo che esamina gli elementi di A e li sposta nella posizione corretta. L'algoritmo mantiene tre cursori i, j, k che separano gli elementi 'V', 'B' e 'R' nel modo seguente:

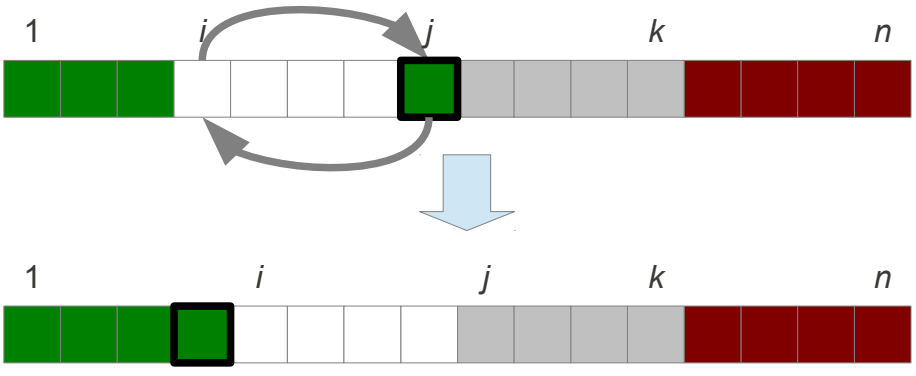
Quindi $A[1..i\text{-}1]$ rappresenta la parte verde, $A[i..j\text{-}1]$ rappresenta la parte bianca, $A[k+1..n]$ rappresenta la parte rossa e infine $A[j..k]$ rappresenta la parte del vettore ancora da esplorare. In altre parole, l'algoritmo mantiene le seguenti invarianti (ossia ad ogni ciclo mantiene vere le seguenti proprietà):

- $A[m]$ = 'V' per ogni m=1, …, i-1;
- $A[m]$ = 'B' per ogni m=i, …, j-1;
- $A[m]$ = 'R' per ogni m=k+1, … n

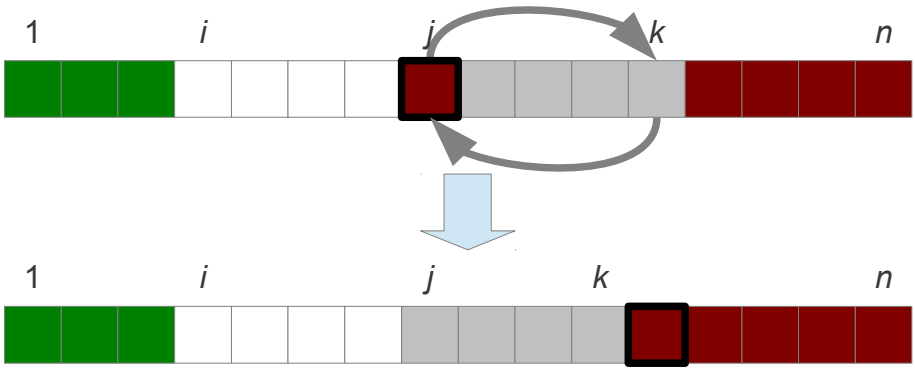
mentre il contenuto di $A[j..k]$ è inesplorato. L'idea dell'algoritmo consiste nell'esaminare il valore $A[j]$ e spostarlo, se necessario, nella posizione corretta. Occorre quindi definire in dettaglio il comportamento dell'algoritmo in base al valore di $A[j]$. Se $A[j]$ = 'B', si incrementa j senza fare nessuna ulteriore operazione, in quanto abbiamo scoperto un elemento bianco che si trova già in coda alla parte bianca del vettore. La situazione è mostrata nella figura seguente:



Se $A[j]$ = 'V', allora scambiamo tra di loro gli elementi $A[i]$ e $A[j]$, in modo da spostare $A[j]$ in coda alla parte verde; si incrementano quindi i e j. La situazione è mostrata nella figura seguente:



Infine, se $A[j]$ = 'R', si scambiano tra di loro gli elementi $A[j]$ e $A[k]$ e si decrementa k. LA situazione è mostrata nella figura seguente:



Siamo ora in grado di fornire lo pseudocodice completo dell'algoritmo della bandiera:

```
algoritmo BANDIERA( array A[1..n] di char )
    int i := 1;
    int j := 1;
    int k := n;
    char tmp;    // usata come variabile temporanea per gli scambi
    while ( j ≤ k ) do
        if ( A[j] == 'B' ) then
            j := j + 1;
        elseif ( A[j] == 'V' ) then
            tmp := A[j];
            A[j] := A[i];
            A[i] := tmp;
            i := i + 1;
            j := j + 1;
        else
            tmp := A[k];
            A[k] := A[j];
            A[j] := tmp;
            k := k - 1;
        endif
    endwhile
```

Per calcolare il costo asintotico dell'algoritmo, osserviamo che ad ogni iterazione la lunghezza della parte inesplorata (rappresentata in grigio) diminuisce sempre di un elemento. Ogni iterazione (il corpo del ciclo "while") ha costo O(1). Inizialmente la parte inesplorata è composta da tutti gli n elementi del vettore. Quindi l'algoritmo termina in $\Theta(n)$ passi.

Esercizio 3.13

Scrivere una variante dell'algoritmo MERGESORT, che chiameremo MERGESORT4(A[1..n], i, j), in cui il (sotto-)vettore $A[i..j]$ da ordinare viene suddiviso in quattro parti anziché in due. MERGESORT4 deve fare uso della funzione MERGE usuale, che fonde due sottovettori ordinati contigui. Prestare attenzione che sono necessarie tre fasi di MERGE: supponendo che il $A[i..j]$ venga suddiviso in $A[i..t1]$, $A[t1+1..t2]$, $A[t2+1..t3]$ e $A[t3+1..j]$, dopo aver ricorsivamente ordinato ciascun sottovettore è necessario effettuare dapprima la fusione di $A[i..t1]$ e $A[t1+1..t2]$, quindi la fusione di $A[t2+1..t3]$ e $A[t3+1..j]$ e infine la fusione di $A[i..t2]$ e $A[t2+1..j]$. Analizzare quindi il costo computazionale di MERGESORT4.

Soluzione. Ricordiamo che la chiamata MERGE(A, i, k, j) vista a lezione fonde i due sottovettori ordinati contigui $A[i..k]$ e $A[k+1..j]$. L'algoritmo MERGESORT4 può essere descritto dallo pseudocodice seguente:

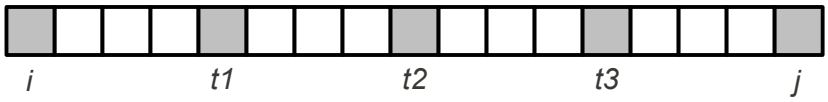
```
algoritmo MERGESORT4( array A[1..n] di double, int i, int j )
```

```
if ( i ≥ j ) then
    return;
else
    int t2 := (i + j) / 2;              // arrotondato per difetto
    int t1 := (i + t2) / 2;             // arrotondato per difetto
    int t3 := (t2 + j) / 2;             // arrotondato per difetto
    MergeSort4( A, i, t1 );             // costo T(n/4)
    MergeSort4( A, t1+1, t2 );          // costo T(n/4)
    MergeSort4( A, t2+1, t3 );          // costo T(n/4)
    MergeSort4( A, t3+1, j );           // costo T(n/4)
    Merge( A, i, t1, t2 );              // costo Θ(n/2) = Θ(n)
    Merge( A, t2+1, t3, j );            // costo Θ(n/2) = Θ(n)
    Merge( A, i, t2, j );               // costo Θ(n)
endif
```

La parte più delicata dell'algoritmo consiste probabilmente nel calcolare gli indici $t1$, $t2$, $t3$ che delimitano le quattro partizioni dell'array. Si potrebbe essere tentati di scrivere qualcosa come:

```
int t1 := (i + j) / 4;   // SBAGLIATO!
```

ma questo sarebbe errato. Ad esempio, se i=10, j=14, si avrebbe $t1$=6 il che è impossibile, in quanto si deve sempre avere $i \le t1 \le t2 \le t3 \le j$. La soluzione proposta, invece, calcola prima la posizione mediana $t2$ tra i e j, e successivamente la posizione mediana $t1$ tra i e $t2$, e la posizione mediana $t3$ tra $t2$ e j. Si faccia riferimento alla figura seguente



L'analisi del costo computazionale si può effettuare scrivendo l'equazione di ricorrenza del costo $T(n)$ necessario ad ordinare un (sotto-)vettore con n elementi. Osserviamo che ciascuna delle quattro chiamate ricorsive a MergeSort4 ha costo $T(n/4)$, in quanto opera su un sottovettore che ha dimensioni circa un quarto del sottovettore di input. Ricordando che il costo dell'operazione Merge è proporzionale alla somma delle dimensioni dei vettori da fondere, le prime due operazioni Merge hanno costo $\Theta(n/2) = \Theta(n)$, mentre l'ultima ha costo $\Theta(n)$. In sostanza, il costo complessivo delle quattro chiamate ricorsive è 4 $T(n/4)$, mentre il costo cumulativo di tutte le operazioni merge è $\Theta(n)$. Possiamo quindi scrivere:

$$T(n) = \begin{cases} c_1 & \text{se } n \le 1 \\ 4\,T(n/4) + c_2 n & \text{altrimenti} \end{cases}$$

che in base al Master Theorem (caso 2, con $a = b = 4$, $f(n) = c_2 n$) ha come soluzione $T(n) = \Theta(n \log n)$. Possiamo quindi concludere che MergeSort4 ha esattamente lo stesso costo asintotico di MergeSort.

Esercizio 3.14

Una matrice quadrata $M[1..n, 1..n]$ di $n \times n$ interi è un *quadrato latino* se ogni intero nell'insieme $\{1, \ldots, n\}$ compare una e una sola volta in ogni riga e in ogni colonna. Ad esempio, questa matrice rappresenta un quadrato latino:

1	2	3	4
2	1	4	3
4	3	2	1

3	4	1	2

Scrivere un algoritmo efficiente che, data in input una matrice quadrata $M[1..n, 1..n]$ di interi arbitrari, restituisca true se e solo se M rappresenta un quadrato latino.

Soluzione. Per risolvere questo problema può essere utile iniziare affrontando il sottoproblema più semplice che consiste nel capire se un array $A[1..n]$ di n interi contiene una e una sola volta tutti i valori in $\{1, \ldots n\}$. In dettaglio, definiamo una funzione CheckArray(**array** A[1..n] di **int**) → **bool** che restituisce *true* se e solo se A contiene una e una sola volta tutti i valori nell'insieme $\{1, \ldots n\}$. Un modo efficiente per fare ciò consiste nell'usare una strategia ispirata a counting sort: definiamo un array di booleani $F[1..n]$ inizialmente inizializzato a *false*. Per ogni valore v appartenente all'array A, controlliamo innanzitutto se v appartiene all'insieme $\{1, \ldots n\}$; in caso affermativo, controlliamo $F[v]$: se vale *false*, significa che abbiamo incontrato il valore v per la prima volta e poniamo $F[v]$ a *true*. Se $F[v]$ valeva già *true*, significa che il valore v è ripetuto. Una volta terminata l'analisi di tutti gli elementi di A senza aver mai trovato duplicati, abbiamo la certezza che A conteneva tutti e soli i valori nell'insieme $\{1, \ldots n\}$. Otteniamo quindi l'algoritmo seguente:

```
algoritmo CheckArray( array A[1..n] di int ) → bool
    array F[1..n] di bool;
    // Inizializza F
    for i:=1 to n do
        F[i] := false;
    endfor
    // Ciclo di controllo
    for i:=1 to n do
        int v := A[i];
        if ( v < 1 || v > n ) then
            return false;                  // abbiamo trovato un valore non valido
        else
            if ( F[v] == true ) then
                return false;              // abbiamo trovato un duplicato
            else
                F[v] := true;
            endif
        endif
    endfor
    return true;
```

L'algoritmo CeckArray così come descritto ha costo $\Theta(n)$, a causa della fase di inizializzazione dell'array tmp. Si noti che, nel caso in cui fosse possibile evitare l'inizializzazione, il costo sarebbe $O(n)$ in quanto l'algoritmo potrebbe terminare in tempo O(1) nel caso ottimo che si verifica quando il primo elemento di A ha un valore che non appartiene all'insieme $\{1, 2, \ldots n\}$.
A questo punto, la soluzione al nostro problema di partenza consiste nell'applicare 1a funzione check_array ad ogni riga e ogni colonna della matrice M, per controllare che restituisca sempre *true*.

```
algoritmo ControllaQuadratoLatino( array M[1..n, 1..n] di int ) → bool
    array tmp[1..n] di int;
    for i:=1 to n do
        // copia la riga i-esima di M nell'array tmp
        for j:=1 to n do
            tmp[j] := M[i, j];
        endfor
        // controlla tmp
        if ( CheckArray(tmp) == false )
            return false;
        endif
        // copia la colonna i-esima di M nell'array tmp
        for j:=1 to n do
```

```
            tmp[j] := M[j, i];
        endfor
        // controlla tmp
        if ( CheckjArray(tmp) == false )
            return false;
        endif
    endfor
    // se i controlli sono stati superati, restituisce true
    return true;
```

Il costo dell'algoritmo è $O(n^2)$. Nel caso pessimo il costo è $\Theta(n^2)$ e si verifica quando M è un quadrato latino; nel caso ottimo il costo è $\Theta(n)$ e si verifica quando già la prima riga fallisce il controllo. Attenzione che nel caso ottimo occorre comunque copiare almeno la prima riga nell'array tmp prima di effettuare il controllo, e tali operazioni richiedono entrambe tempo $\Theta(n)$.
È anche possibile dare una rappresentazione più compatta dell'algoritmo, senza usare esplicitamente la funzione esterna CheckArray: Usiamo due vettori booleani riga e col che ci permettono di controllare una riga e una colonna contemporaneamente:

```
algoritmo ControllaQuadratoLatino( array M[1..n, 1..n] di int ) → bool
    array riga[1..n] di bool;
    array col[1..n] di bool;
    int v;
    for i:=1 to n do
        // Inizializza gli array prima di ogni iterazione
        for j:=1 to n do
            riga[j] := false;
            col[j] := false;
        endfor
        for j:=1 to n do
            // Controlla la riga i-esima
            v := M[i, j];
            if ( v<1 || v>n ) then
                return false;
            else if ( riga[v] == true ) then
                return false;
            else
                riga[v] := true;
            endif
            // Controlla la colonna i-esima
            v := M[j, i];
            if ( v<1 || v>n ) then
                return false;
            else if ( col[v] == true ) then
                return false;
            else
                col[v] := true;
            endif
        endfor
    endfor
    // se i controlli sono stati superati, restituisce true
    return true;
```

Si noti che all'interno dei due cicli for annidati usiamo alternativamente i e j come indici di riga e di colonna alternativamente. Il costo dell'algoritmo è $O(n^2)$ come nel caso precedente.

Esercizio 3.15

Siete stati assunti come consulenti della Telecom per risolvere il seguente problema. E' dato un array $T[1..n]$ non ordinato di numeri interi, ciascuno dei quali rappresenta un numero di telefono di un abbonato di

Bologna. La Telecom vi informa che l'array *T* contiene poco meno di 100000 numeri di telefono, tutti compresi nell'intervallo [200001, 300000] (estremi inclusi) e nell'array non sono presenti duplicati. Scrivete un algoritmo efficiente per ordinare l'array *T*.

Soluzione. Possiamo osservare che l'array *T* è composto da numeri di 6 cifre decimali, che può quindi essere ordinato in tempo proporzionale alla sua dimensione usando Bucket Sort. In realtà esiste anche un'altra soluzione più semplice, basata su una versione specializzata di Counting Sort. Utilizziamo un array di booleani *B*[1..100000], inizializzato a false, in cui risulterà *B*[i] = *true* se e solo se il valore 200000 + *i* esiste in *T*. Dato che per ipotesi l'array *T* non contiene duplicati, non è necessario usare un array di contatori (come si usa nel Counting Sort convenzionale) per tenere traccia del numero di occorrenze di ciascun valore.

```
algoritmo OrdinaTel( array T[1..n] di int ) → array[1..n] di int
   array B[1..100000] di bool;
   for i := 1 to 100000 do
        B[i] := false;
   endfor
   for i := 1 to n do
        B[ T[i] - 200000 ] := true;
   endfor
   int j := 1;
   for i := 1 to 100000 do
        if ( B[i] == true ) then
             T[j] := 200000+i;
             j := j + 1;
        endif
   endfor
   return T;
```

Si presti particolare attenzione a come l'array *B* debba essere indicizzato; scrivere semplicemente *B*[*T*[*i*]] := *true* non sarebbe stato corretto, in quanto *T*[*i*] assume valori fuori dall'intervallo valido degli indici di *B*.

Capitolo 4: Hashing

Esercizio 4.1

Ricordiamo che una funzione hash $h: U \rightarrow \{0, 1, \ldots, m\text{-}1\}$ è detta *perfetta* se per ogni coppia di chiavi distinte x e y, si ha $h(x) \neq h(y)$.

1. Scrivere un algoritmo per il calcolo di una funzione hash perfetta in cui l'insieme universo sia costituito da sequenze di cinque caratteri scelti tra i seguenti: 'A', 'C', 'T', 'G' e il valore di m sia minimo possibile.
2. Determinare il costo computazionale dell'algoritmo di cui al punto precedente.

Soluzione. Innanzitutto osserviamo che ci sono 4^5 possibili chiavi, quindi $m=4^5=1024$. Quindi dobbiamo mappare una chiave in un numero compreso tra 0 e $4^5-1 = 1023$. L'idea è quella di considerare ciascuna chiave come un numero in base 4, dove le "cifre" del nostro sistema di numerazione sono A=0, C=1, T=2 e G=3. Sia $K[1..5]$ l'array di 5 caratteri che contiene la chiave. La funzione hash perfetta calcola il valore dell'espressione seguente:

$$h(K) = 4^4\, K[1] + 4^3\, K[2] + 4^2\, K[3] + 4^1\, K[4] + 4^0\, K[5]$$

dove supponiamo che le operazioni aritmetiche vengano svolte sostituiendo ai caratteri i numeri da 0 a 3, come riportato sopra. $h(K)$ rappresenta il valore decimale che corrisponde alla stringa K, interpretata come se fosse un numero in base 4. L'algoritmo per valutare l'espressione $h(K)$ può essere scritto in questo modo:

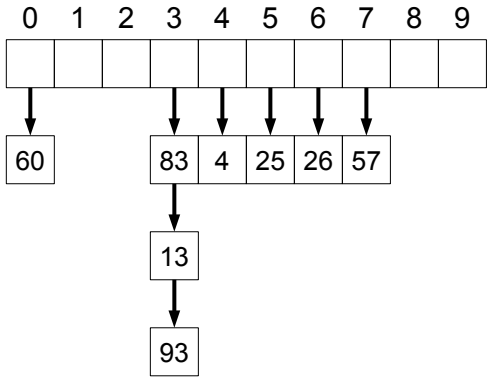
```
algoritmo HashDNA( array K[1..5] di caratteri ) → int
   int risultato := 0;
   int c; // cifra corrente
   for i:=1 to 5 do
        if (K[i] == 'A') then
             c := 0;
        elseif (K[i] == 'C' ) then
             c := 1;
        elseif (K[i] == 'T') then
             c := 2;
        else
             c := 3'
        endif
        risultato := risultato *4 + c;
   endfor
   return risultato;
```

Il costo computazionale dell'algoritmo è O(1). Infatti l'algoritmo consiste in un ciclo il cui corpo viene eseguito sempre e solo 5 volte, e il costo di ciascuna iterazione è O(1). Si noti che la dimensione dell'input è fissa e pari a 5.

Esercizio 4.2

Sia dato l'insieme di chiavi K = {25, 83, 57, 26, 13, 60, 93, 4}, e sia m=10. Si consideri una tabella hash (inizialmente vuota) di dimensione m. Mostrare come cambia la struttura della tabella ogni volta che si inserisce una delle chiavi di K, nell'ordine indicato. La funzione hash utilizzata è $h(k) = k \bmod m$. La tabella usa liste di collisione (assumere che gli elementi vengano sempre inseriti in fondo alle liste).

Soluzione. Mostriamo solo la configurazione finale:



Esercizio 4.3

Il professor Dal Caso ha inventato un nuovo tipo di funzione hash $h(x)$ che reputa efficientissima. Fissata una costante m>1, la funzione h mappa una chiave x in un intero scelto casualmente nell'insieme {1, 2, ... , m}; il valore casuale non dipende in alcun modo dal valore di x. Secondo voi è possibile realizzare una tabella hash usando la funzione $h(x)$ del professor Dal Caso? Giustificare la risposta.

Soluzione. La funzione $h(x)$ definita come sopra non può essere utilizzata per implementare una tabella hash. Infatti, per verificare se un oggetto x è presente nella tabella oppure no, sarebbe sempre necessario esaminare l'intera tabella in quanto la posizione iniziale $h(x)$ non fornisce alcuna informazione utile sulla presenza o meno di x.

Esercizio 4.4

Si consideri una tabella hash memorizzata in un vettore $A[0..m - 1]$ con m = 10 slot in cui le collisioni sono gestite mediante indirizzamento aperto. L'insieme delle chiavi è costituito da interi non negativi. La funzione hash adottata è $h(k, i) = ((k \bmod m) + i) \bmod m$, essendo k il valore della chiave e i il contatore del "tentativo di inserimento". Mostrare il contenuto del vettore A dopo ciascuna delle operazioni seguenti: (la soluzione è mostrata dopo ogni operazione)

Inserimento chiave 7

							7		

Inserimento chiave 19

							7		19

Inserimento chiave 37

							7	37	19

Cancellazione chiave 7

							DEL	37	19

Inserimento chiave 21

	21						DEL	37	19

Cancellazione chiave 37

	21						DEL	DEL	19

Inserimento chiave 17

	21						17	DEL	19

Inserimento chiave 11

	21	11					17	DEL	19

Cancellazione chiave 21

	DEL	11					17	DEL	19

Esercizio 4.5

Si consideri una tabella hash con m=10 slot, gestita mediante indirizzamento aperto. Le chiavi sono numeri interi non negativi. Il professor C. Trullo propone di generare la lista di ispezioni di una chiave k usando la funzione hash seguente:

$$h(k, i) = (k + 2 \times i) \bmod m$$

Ritenete che sia una buona idea? Giustificare la risposta.

Soluzione. Una sequenza di ispezione deve produrre una permutazione dei valori {0, 1, …, m-1} in modo da consentire l'esplorazione di tutti gli slot della tabella. La funzione $h(k, i)$ esplora inizialmente lo slot in posizione k mod m, quindi prosegue con (k+2) mod m, (k+4) mod m e così via. Purtroppo, poiché m=10, le sequenze di ispezione così generate non visitano tutti gli slot, ma solo quelli in posizione pari o dispari (in base al valore iniziale di k). Ad esempio, se k=12, la sequenza di ispezione è

2, 4, 6, 8, 0, 2, 4, 6, 8, 0

che come si vede non considera mai gli slot in posizione dispari. Possiamo quindi concludere che la funzione proposta non è una funzione hash appropriata.
E' interessante osservare che il problema si verifica solo quando m (numero di slot nella tabella) è un numero pari. Se m fosse dispari, la funzione $h(k, i)$ produrrebbe sequenze di ispezione che visitano tutti gli elementi della tabella, e quindi opererebbe nel modo corretto.

Capitolo 5: Divide et Impera

Esercizio 5.1

Si consideri il seguente problema: dato un array $A[1..n]$ di $n \geq 1$ valori reali non ordinati, restituire il valore minimo in A.

1. Scrivere un algoritmo *ricorsivo* di tipo *divide et impera* per risolvere il problema di cui sopra; non è consentito usare variabili globali.
2. Calcolare la complessità asintotica dell'algoritmo proposto, motivando la risposta;
3. Dare un limite inferiore alla complessità del problema, nell'ipotesi in cui l'algoritmo risolutivo si basi solo su confronti. Giustificare la risposta.

Soluzione. L'algoritmo può essere descritto come segue:

```
algoritmo MinDivideEtImpera( array A[1..n] di double, int i, int j ) → double
  if ( i > j ) then
      errore: vettore vuoto
  if ( i == j ) then
      return A[i];
  else
      int m := Floor( ( i + j ) / 2 );
      double v1 := MinDivideEtImpera(A, i, m);
      double v2 := MinDivideEtImpera(A, m+1, j);
      if (v1 < v2) then
          return v1;
      else
          return v2;
      endif
  endif
```

I parametri i e j indicano rispettivamente l'indice del primo e dell'ultimo elemento (estremi inclusi) in cui effettuare la ricerca. La prima invocazione dell'algoritmo quindi sarà MinDivideEtImpera(A,1,n). E' importante osservare che l'algoritmo richiede tassativamente che il vettore sia non vuoto, cioè $n \geq 1$, e restituisce un errore nel caso venga invocato con $i > j$. Se $n \geq 1$ tale condizione non si verificherà mai, cioè la ricorsione termina sempre con un vettore composto da un singolo elemento.
La complessità asintotica si ricava risolvendo la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 2T(n/2) + c_2 & \text{altrimenti} \end{cases}$$

Applicando il Master Theorem (caso 1), la soluzione è $T(n) = \Theta(n)$.
Per ricavare un limite inferiore, è sufficiente considerare che un qualsiasi algoritmo basato su confronti deve necessariamente considerare ogni elemento dell'array A almeno una volta (altrimenti se viene saltato un elemento, questo potrebbe essere il minimo e l'algoritmo non sarebbe corretto). Quindi il limite inferiore alla complessità del problema è $\Omega(n)$.

Esercizio 5.2

Si consideri un array $A[1..n]$ composto da $n \geq 1$ numeri reali, non necessariamente ordinato.

1. Scrivere un algoritmo ricorsivo, di tipo divide-et-impera, per determinare il numero di elementi in A il cui valore sia maggiore o uguale a zero;
2. Analizzare il costo computazionale dell'algoritmo proposto.

Soluzione. Una possibile soluzione è la seguente:

```
algoritmo ContaNonNegativi( array A[1..n] di double, int i, int j ) → int
  if ( i > j ) then
      return 0;     // array vuoto
  elseif ( i == j ) then
      if ( A[i] ≥ 0 ) then return 1 else return 0 endif;
  else
      int m := Floor( ( i + j )/2 );
      return ContaNonNegativi(A, i, m) + ContaNonNegativi(A, m+1, j);
  endif
```

L'algoritmo viene invocato con ContaNonNegativi(A, 1, n). Sono presenti due casi base: (i) il caso in cui il sottovettore $A[i..j]$ sia vuoto ($i > j$), e (ii) il caso in cui il sottovettore sia composto da un singolo elemento. Nel caso (i) l'algoritmo restituisce 0, in quanto in un vettore vuoto non sono presenti valori positivi. Nel caso (ii) l'algoritmo ritorna 1 oppure 0, a seconda che l'unico valore presente sia non negativo o no. Nel caso in cui $A[i..j]$ contenga più di un elemento, il sottovettore viene suddiviso in due parti approssimativamente uguali, e si restituisce la somma dei valori nonnegativi presenti nelle due metà
Il costo computazionale $T(n)$ dell'algoritmo invocato su un vettore di n elementi soddisfa l'equazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 2\,T(n/2) + c_2 & \text{altrimenti} \end{cases}$$

che in base al Master Theorem ha soluzione $T(n) = \Theta(n)$.

Esercizio 5.3

Scrivere un algoritmo ricorsivo, di tipo divide-et-impera, per calcolare la somma dei valori contenuti in un array $A[1..n]$ di valori reali. Si richiede che l'algoritmo divida ricorsivamente l'array in tre sottovettori di uguale lunghezza. Calcolare il costo computazionale dell'algoritmo utilizzando il Master Theorem.

Soluzione.

```
algoritmo Somma3( array A[1..n] di double, int i, int j ) → double
  if ( i > j ) then
      return 0;
  elseif ( i == j ) then
      return A[i];
  else
      int d1 := Floor( i + (j – i + 1)/3 );
      int d2 := Floor( i + (j – i + 1)*2/3 );
      return  Somma3( A, i, d1 - 1 ) +
              Somma3( A, d1, d2 – 1 ) +
              Somma3( A, d2, j );
  endif
```

L'algoritmo viene invocato con Somma3(A, 1, n). Osserviamo che la ricorsione presenta due casi "base": il caso di sottovettore vuoto ($i > j$), e il caso di sottovettore con un singolo elemento $A[i]$. Ad ogni chiamata ricorsiva, si calcolano le posizioni iniziali *d1* e *d2* del secondo e terzo sottovettore, rispettivamente. Fatto

questo, il sottovettore viene implicitamente scomposto in $A[i..d1\text{-}1]$, $A[d1..d2\text{-}1]$ e $A[d2..j]$, e l'algoritmo invocato ricorsivamente su ogni parte. Il calcolo di $d1$ e $d2$ è fatto in modo tale da garantisce il comportamento corretto dell'algoritmo anche quando $A[i..j]$ è composto da 3 oppure 2 elementi.
Il costo computazionale soddisfa la seguente equazione di ricorrenza:

$$T(n)=\begin{cases} c_1 & \text{se } n\le 1 \\ 3\,T(n/3)+c_2 & \text{altrimenti} \end{cases}$$

che in base al Master Theorem ha soluzione $T(n) = \Theta(n)$

Esercizio 5.4

Si consideri un array $A[1..n]$ composto da n numeri interi; l'array A è ordinato in senso non decrescente. Scrivere un algoritmo *divide et impera* per decidere se l'array contiene oppure no elementi duplicati. Analizzare il costo computazionale dell'algoritmo proposto. Secondo voi l'algoritmo potrebbe essere reso asintoticamente più efficiente?

Soluzione. L'algoritmo EsistonoDuplicati(A, i, j) restituisce *true* se e solo se esistono valori duplicati nel sottovettore $A[i..j]$

```
algoritmo EsistonoDuplicati( array A[1..n] di int, int i, int j ) → bool
   if ( i ≥ j ) then
        return false;
   else
        int m := Floor( ( i + j ) / 2 );
        return ( A[m] == A[m+1] || EsistonoDuplicati(A,i,m) || EsistonoDuplicati(A,m+1,j) );
   endif
```

Nel caso in cui il sottovettore $A[i..j]$ sia vuoto o contenga un singolo elemento (cioè quando $i \ge j$), l'algoritmo restituisce *false*, in quanto non possono esistere duplicati. Se il sottovettore contiene più di un elemento, l'algoritmo restituisce *true* se e solo se una delle tre seguenti alternative è vera:

1. I due elementi $A[m]$ e $A[m+1]$ "a cavallo" delle due metà di $A[i..j]$ sono uguali; oppure
2. Esistono duplicati nel sottovettore $A[i..m]$; oppure
3. Esistono duplicati nel sottovettore $A[m+1..j]$

In base al Master Theorem, l'algoritmo ha costo $T(n) = \Theta(n)$. Qualsiasi algoritmo deve necessariamente esaminare tutti gli elementi almeno una volta nel caso peggiore; quindi ogni algoritmo risolutivo per questo problema richiederà $\Omega(n)$ nel caso peggiore.

Esercizio 5.5

Una inversione in un array $V[1..n]$ di n numeri reali è una coppia di indici i, j tali che $i < j$ e $V[i] > V[j]$.

1. Scrivere un algoritmo iterativo che dato un array V calcola il numero di inversioni presenti in V in tempo $O(n^2)$.
2. Scrivere un algoritmo divide-et-impera che dato un array V calcola il numero di inversioni in tempo $O(n \log n)$. *Suggerimento: considerare un algoritmo simile a MergeSort; in particolare, la fase in cui si combinano i risultati parziali può essere implementata in modo simile alla operazione merge di MergeSort.*

Soluzione. Iniziamo proponendo la soluzione meno efficiente, richiesta al punto 1. L'idea è quella di confrontare tra di loro tutti gli elementi $V[i]$, $V[j]$ con $i<j$, e contare quanti di loro sono nell'ordine "errato".

```
algoritmo ContaInversioniNonEff( array A[1..n] di double ) → int
   int inv := 0;          // numero di inversioni
   for i := 1 to n-1 do
        for j := i+1 to n do
             if ( V[i] > V[j] ) then
```

```
                inv := inv + 1;
            endif
        endfor
    endfor
    return inv;
```

Il costo di questa soluzione, come richiesto è $O(n^2)$ (per la precisione, il costo è $\Theta(n^2)$ in quanto il costo nel caso ottimo coincide con quello nel caso pessimo).
Una soluzione più efficiente consiste nell'utilizzare una variante di MergeSort, come segue:

```
algoritmo ContaInversioni( array A[1..n] di double, int i, int j ) → int
    if ( i ≥ j ) then
        return 0;
    else
        int m := Floor( ( i + j ) / 2 );
        int c1 := ContaInversioni(A, i, m);
        int c2 := ContaInversioni(A, m+1, j);
        int c3 := ContaInversioniMerge(A, i, m, j);
        return c1 + c2 + c3;
    endif

algoritmo ContaInversioniMerge(array A[1..n] di double, int i, int m, int j) → int
    array tmp  := array [1..(j – i + 1)] di double;
    int c := 0;
    int k := 1;
    int i1 := i;
    int i2 := m + 1;
    while (i1 ≤ m && i2 ≤ j) do
        if (A[i1] ≤ A[i2]) then
            tmp[k] := A[i1];
            i1 := i1 + 1;
        else
            c := c + (m + 1 - i1);
            tmp[k] := A[i2];
            i2 := i2 + 1;
        endif
        k := k+1;
    endwhile
    while (i1 ≤ m) do
        tmp[k] := A[i1];
        i1 := i1 + 1;
        k := k + 1;
    endwhile
    while (i2 ≤ j) do
        tmp[k] := A[i1];
        i1 := i1 + 1;
        k := k + 1;
    endwhile
    for k=1 to (j – i + 1) do
        A[i+k-1] := tmp[k];
    endfor
    return c;
```

Esercizio 5.6

L'Agenzia Nazionale per la Valutazione del sistema Universitario della Ruritania (ANVUR) ha deciso di valutare i docenti e ricercatori universitari in vista di una promozione, utilizzando l'inedito meccanismo seguente. A ciascuno degli n ricercatori è associato un “indice di bravura” reale non negativo: più è alto, più

il ricercatore è considerato "bravo". I ricercatori sono stati ordinati in modo che i loro indici di bravura $H[1]$, $H[2]$, ..., $H[n]$ risultino in ordine non decrescente ($H[i]$ è l'indice di bravura del ricercatore i-esimo). A questo punto, l'ANVUR ha definito, usando non meglio precisate "approssimazioni", un numero reale non negativo k. Verranno promossi solo i ricercatori il cui indice di bravura sia strettamente maggiore di k.

1. Scrivere un algoritmo efficiente che, dato il vettore ordinato $H[1..n]$ e la soglia k, calcola il numero totale di ricercatori promossi.
2. Determinare il costo computazionale asintotico dell'algoritmo proposto al punto 1.

Soluzione. Iniziamo con la soluzione più semplice: effettuiamo una scansione di tutto l'array, contando quanti sono gli elementi maggiori di k:

```
algoritmo ContaPromossi1( array H[1..n] di double, double k ) → int
   int c := 0;              // contatore
   for i:=1 to n do
       if ( H[i] > k ) then c := c+1; endif;
   endfor
   return c;
```

Il costo dell'algoritmo è $\Theta(n)$.
Vediamo ora una soluzione marginalmente più efficiente. Sfruttando il fatto che l'array H è ordinato, possiamo evitare la scansione completa e fermarci al primo elemento il cui valore è $> k$; imfatti, sappiamo che tutti gli elementi successivi saranno anch'essi maggiori di k.

```
algoritmo ContaPromossi2( array H[1..n] di double, double k ) → int
   int i := 1;
   while ( i ≤ n && H[i] ≤ k ) do
       i := i + 1;
   endwhile
   return (n-i+1);
```

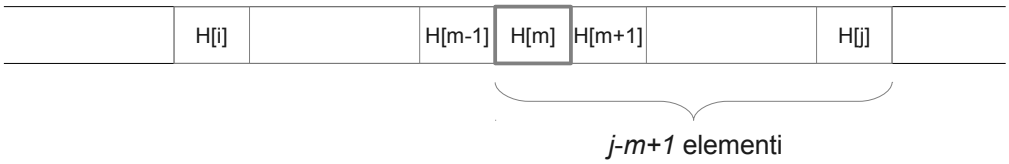
Il costo in questo caso è $O(n)$, in quanto non necessariamente si scorre l'intero array. Se tutti i ricercatori hanno indice di bravura sotto la soglia k, l'algoritmo termina in $\Theta(n)$ passi (caso pessimo). Se tutti i ricercatori hanno indice di bravura sopra la soglia k (caso ottimo), l'algoritmo richiede $O(1)$ iterazioni in quanto risulterà $H[1] > k$, e il ciclo while termina immediatamente.
Vediamo infine la soluzione efficiente, che si basa sulla ricerca binaria per contare quanti valori di H risultano strettamente maggiore di k. La funzione ContaPromossi3(H, k, i, j) restituisce il numero degli elementi del sottovettore $H[i..j]$ che risultano maggiori di k (tenendo presente che H è ordinato in senso crescente):

```
algoritmo ContaPromossi3( array H[1..n] di double, double k, int i, int j ) → int
   if ( i > j ) then
       return 0;                                          // (1)
   else
       int m := Floor( ( i + j ) / 2 );
       if ( H[m] ≤ k ) then
           return ContaPromossi3( H, k, m+1, j );         // (2)
       else
           return (j-m+1) + ContaPromossi3( H, k, i, m-1 ); // (3)
       endif
   endif
```

L'algoritmo viene inizialmente invocato con ContaPromossi3(H, k, 1, n), essendo n il numero di elementi di H. Ad ogni passo:

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo è zero (linea (1));
- se $i \leq j$ come prima cosa determiniamo la posizione m dell'elemento centrale, esattamente come si fa con la ricerca binaria. Distinguiamo due sottocasi:

- se $H[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando esclusivamente gli elementi che stanno a destra di $H[m]$, ossia quelli nel sottovettore $H[m+1...j]$ (linea (2));
- se $H[m] > k$, l'elemento centrale è sopra la soglia. Quindi, essendo l'array ordinato, tutti gli (j-m+1) elementi in $H[m..j]$ risultano sopra la soglia. Il numero di tali elementi, sommato al conteggio di quelli sopra soglia in $H[i..m-1]$ (che viene calcolato dalla chiamata ricorsiva) produce il risultato (linea (3)).



A titolo di esempio, si consideri il vettore seguente con soglia k=5 (il numero indicato sopra agli elementi del vettore rappresenta l'indice della cella).

1	*2*	*3*	*4*	*5*	*6*	*7*
2	3	5	6	6	7	8

L'indice m della posizione centrale del vettore è m=4, che contiene il valore 6. Quindi supponendo di avere invocato la funzione come calcolo di ContaPromossi3(H, 5, 1, 7), verrà eseguita la linea etichettata con (3), che calcola il risultato ricorsivamente come (7-4+1) + ContaPromossi3(H, 5, 1, 3).

Il costo computazionale $T(n)$ di questo algoritmo soddisfa l'equazione di ricorrenza $T(n) = T(n/2) + O(1)$, che in base al Master Theorem ha soluzione $T(n) = O(\log n)$

Esercizio 5.7

Si consideri un array $A[1..n]$ contenente n valori reali distinti ordinati in senso crescente. Scrivere un algoritmo efficiente che, dato l'array A e un numero reale arbitrario x, restituisce l'indice del più piccolo elemento in A che risulti strettamente maggiore di x. Se non esiste alcun valore di A che soddisfa tale vincolo, l'algoritmo deve restituire n+1 (ossia l'indice del primo elemento dopo l'ultimo presente in A). Ad esempio, se A = [1, 1.8, 2, 2.3, 3.4] e x = 2.5, l'algoritmo deve restituire 4, in quanto $A[4]$ = 2.3 risulta essere il minimo valore presente in A strettamente maggiore di x. Se A = [1, 2.5, 2.6, 3, 3.4, 5.2, 7] e x = 3.4, l'algoritmo restituisce 6, in quanto $A[6]$ = 5.2 è il minimo valore presente in A che risulti strettamente maggiore di x. Infine, se A = [1, 2, 3] e x = 3.5, l'algoritmo restituisce 4.

Soluzione. Per risolvere questo problema possiamo utilizzare una versione adeguatamente modificata dell'algoritmo di ricerca binaria. Chiamamo l'algoritmo UpperBound, nome usato dalla libreria STL del linguaggio C++ per indicare la funzione che risolve questo problema. Mentre le implementazioni STL usano prevalentemente una soluzione iterativa, mostriamo qui una implementazione ricorsiva.

```
algoritmo UpperBound( array A[1..n] di double, double x, int i, int j) → int
  if ( i > j ) then
      return i;
  else
      int m := Floor( ( i + j ) / 2 );
      if ( A[m] ≤ x ) then
          return UpperBound( A, x, m+1, j );
      else
          return UpperBound( A, x, i, m-1 );
      endif
  endif
```

La precondizione di questo algoritmo (cioè la proprietà che i parametri di input devono soddisfare) è la seguente

$$\forall k = 1,\ldots i-1 : A[k] \leq x$$
$$\forall k = j+1,\ldots n : A[k] > x$$

ossia tutti gli elementi "a sinistra" di $A[i]$ son minori o uguali a x, mentre tutti quelli "a destra" di $A[j]$ risultano strettamente maggiori di x. Si noti che inizialmente i=1 e j=n, per cui entrambe le proprietà valgono banalmente in quanto a sinistra di $A[i]$ e a destra di $A[j]$ non ci sono elementi. Si può poi verificare che la precondizione continua a essere soddisfatta per tutte le chiamate ricorsive.
La precondizione di cui sopra è utile per definire correttamente cosa restituire quando si verifica il caso base della ricorsione (ossia quando $i > j$). Quando $i = j+1$, infatti, la precondizione si può riscrivere come:

$$\forall k = 1,\ldots i-1 : A[k] \leq x$$
$$\forall k = i,\ldots n : A[k] > x$$

per cui la posizione del minimo elemento di A strettamente maggiore di x è la posizione i-esima.
Per risolvere il problema "simmetrico" di individuare la posizione del più grande valore in A che risulti strettamente minore di x possiamo usare il seguente algoritmo LowerBound

```
algoritmo LowerBound( array A[1..n] di double, double x, int i, int j ) → int
   if ( i > j ) then
        return j;
   else
        int m := Floor( ( i + j ) / 2 );
        if ( A[m] ≥ x ) then
             return LowerBound( A, x, i, m-1 );
        else
             return LowerBound( A, x, m+1, j );
        endif
   endif
```

Esercizio 5.8

Si consideri un array $A[1..n]$ contenente valori interi ordinati in senso non decrescente; possono essere presenti valori duplicati. Scrivere un algoritmo ricorsivo di tipo divide-et-impera che, dato in input A e un intero x, restituisce l'indice (la posizione) della prima occorrenza di x in A, oppure restituisce n+1 se il valore x non è presente. Ad esempio, se A=[1, 3, 4, 4, 4, 5, 6, 6] e x=4, l'algoritmo deve restituire 3, in quanto $A[3]$ è la prima occorrenza del valore 4. Modificare quindi l'algoritmo per restituire la posizione dell'*ultima* occorrenza di x in A, oppure 0 se x non è presente.

Soluzione. Il problema si può risolvere con una variante della ricerca binaria. L'algoritmo CercaPrimaOccorrenza(A, x, i, j) restituisce l'indice della prima occorrenza del valore x all'interno del sottovettore ordinato $A[i..j]$. I parametri di input dell'algoritmo devono soddisfare le seguenti precondizioni (oltre al fatto che A deve essere ordinato in senso non decrescente):

$$\forall k = 1,\ldots i-1 : A[k] < x$$
$$\forall k = j+1,\ldots n : A[k] \geq x$$

Date le precondizioni sopra e detta m la posizione centrale del sottovettore $A[i..j]$, m è il risultato cercato se $A[m] == x$, e vale una delle seguenti condizioni:

1. siamo all'inizio del sottovettore ($m == i$), oppure
2. l'elemento precedente $A[m-1]$ ha valore diverso da x.

Se nessuna delle due condizioni vale, la ricerca prosegue ricorsivamente su una delle due metà del sottovettore $A[i..j]$, in modo tale da mantenere valida la precondizione di cui sopra.

```
algoritmo CercaPrimaOccorrenza( array A[1..n] di int, int x, int i, int j ) → int
   if ( i > j ) then
        return n+1;      // valore x non trovato
   else
        int m := Floor( (i + j) / 2 );
        if ( A[m] == x && ( m == i || A[m] ≠ A[m-1] ) then
             return m;
        elseif ( A[m] ≥ x ) then
             return CercaPrimaOccorrenza( A, x, i, m-1 );
        else
             return CercaPrimaOccorrenza( A, x, m+1, j );
        endif
   endif
```

L'algoritmo "simmetrico" CercaUltimaOccorrenza(A, x, i, j) può essere utilizzato per restituire la posizione dell'ultima occorrenza del valore x nel sottovettore ordinato $A[i..j]$, oppure 0 se x non è presente.

```
algoritmo CercaUltimaOccorrenza( array A[1..n] di int, int x, int i, int j ) → int
   if ( i > j ) then
        return 0;        // valore x non trovato
   else
        int m := Floor( (i + j) / 2 );
        if ( A[m] == x && ( m == j || A[m] ≠ A[m+1] ) then
             return m;
        elseif ( A[m] ≤ x ) then
             return CercaUltimaOccorrenza( A, x, m+1, j );
        else
             return CercaUltimaOccorrenza( A, x, i, m-1 );
        endif
   endif
```

Esercizio 5.9

Si consideri un array $A[1..n]$ contenente valori reali ordinati in senso non decrescente; l'array può contenere valori duplicati. Scrivere un algoritmo ricorsivo efficiente di tipo divide-et-impera che, dato l'array A e due numeri reali qualsiasi $low < up$, calcola quanti valori di A appartengono all'intervallo $[low, up]$. Determinare il costo computazionale dell'algoritmo proposto.

Soluzione. Iniziamo con la soluzione più semplice. L'algoritmo seguente conta i valori appartenenti all'intervallo $[l, u]$ che si trovano nel sottovettore $A[i..j]$. L'algoritmo divide il sottovettore in due parti aventi circa la stessa dimensione, e invoca se stesso ricorsivamente sulle due metà

```
algoritmo ContaIntervallo( array A[1..n] di double, double low, double up, int i, int j ) →
int
   if ( i > j ) then
        return 0;
   elseif ( i == j ) then
        if ( A[i] ≥ low && A[i] ≤ up ) then return 1; else return 0; endif;
   else
        int m := Floor( ( i + j ) / 2 );
        return ContaIntervallo(A, low, up, i, m); + ContaIntervallo(A, low, up, m+1, j);
   endif
```

L'algoritmo viene invocato con ContaIntervallo(A, low, up, 1, n). Il suo costo $T(n)$ soddisfa la seguente equazione di ricorrenza:

$$T(n)=\begin{cases} c_1 & \text{se } n\le 1 \\ 2\,T(n/2)+c_2 & \text{altrimenti} \end{cases}$$

L'applicazione del Master Theorem porta alla soluzione $T(n) = \Theta(n)$. Questo vale sia nel caso ottimo che nel caso pessimo.
Una soluzione più complessa ma più efficiente consiste nell'usare gli algoritmi UpperBound e LowerBound descritti nell'Esercizio 5.7. La funzione LowerBound restituisce l'indice dell'elemento di A il cui valore viene "immediatamente prima" di *low*, mentre la funzione UpperBound restituisce l'indice dell'elemento di A il cui valore viene "immediatamente dopo" di *up*.

```
algoritmo ContaIntervallo( array A[1..n] di double, double low, double up ) → int
   int first := LowerBound(A, low, 1, n);
   int last := UpperBound(A, up, 1, n);
   return last – first – 1;
```

Il costo di questo algoritmo è O(log n) nel caso peggiore.

Esercizio 5.10

Si consideri un array $A[1..n]$ composto da $n \ge 1$ valori interi tutti distinti. Supponiamo che l'array sia ordinato in senso crescente ($A[1] < A[2] < \ldots < A[n]$).

1. Scrivere un algoritmo efficiente che, dato in input l'array A, determina un indice i, se esiste, tale che $A[i] = i$. Nel caso esistano più indici che soddisfano la relazione precedente, è sufficiente restituirne uno qualsiasi.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. È possibile utilizzare il seguente algoritmo ricorsivo, molto simile a quello della ricerca binaria (i e j rappresentano rispettivamente gli indici estremi del sottovettore $A[i..j]$ in cui effettuare la ricerca, all'inizio la funzione va invocata con i=1, j=n)::

```
algoritmo CercaI( array A[1..n] di int, int i, int j ) → int
   if (i > j) then
        errore "Non esiste alcun i tale che A[i] = i"
   endif
   int m := Floor( (i + j) / 2 );
   if ( A[m] == m ) then
        return m;
   elseif (A[m] > m ) then
        return CercaI(A,i,m-1);
   else
        return CercaI(A,m+1,j);
   endif
```

Si noti che i valori contenuti nel vettore potrebbero anche essere negativi, per cui si può anche avere il caso in cui $A[m] < m$.
Notiamo che se $A[m] < m$, l'indice i tale per cui $A[i]=i$ non potrà mai trovarsi nella prima meta' $A[i..m-1]$, in quanto per ipotesi (il vettore contiene tutti interi distinti, ed è ordinato) si avrà che $A[k] < k$ per ogni k=i, i+1, … m-1. Per rendersi conto di cio', consideriamo $A[m-1]$. Poiché l'array è ordinato e non esistono duplicati, si ha che $A[m-1] < A[m]$. Ma poiché $A[m] < m$, si ha: $A[m-1] < A[m] < m$, da cui $A[m-1] < m-1$. Lo stesso ragionamento si puo' ripetere per m-2, m-3 eccetera. Il ragionamento simmetrico si applica al caso $A[m] > m$.
L'algoritmo proposto è una semplice variante dell'algoritmo di ricerca binaria, e ha lo stesso costo computazionale O(log n).

Capitolo 6: Tecniche Greedy

Esercizio 6.1

Disponiamo di un tubo metallico di lunghezza L. Da questo tubo vogliamo ottenere al più n segmenti più corti, aventi rispettivamente lunghezza $lun[1]$, $lun[2]$, …, $lun[n]$. Il tubo viene segato sempre a partire da una delle estremità, quindi ogni taglio riduce la sua lunghezza della misura asportata.

1. Scrivere un algoritmo efficiente per determinare il numero massimo di segmenti che è possibile ottenere. Formalmente, tra tutti i sottoinsiemi degli n segmenti la cui lunghezza complessiva sia minore o uguale a L,.vogliamo determinarne uno con cardinalità massimale.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Questo problema si può risolvere con un algoritmo greedy. Ordiniamo le sezioni in senso *non decrescente* rispetto alla lunghezza, in modo che il segmento 1 abbia lunghezza minima e il segmento n lunghezza massima. Procediamo quindi a segare prima il segmento più corto, poi quello successiva e così via finché possibile (cioè fino a quando la lunghezza residua ci consente di ottenere almeno un'altro segmento). Lo pseudocodice può essere descritto in questo modo

```
algoritmo MaxNumSezioni( int L, array lun[1..n] di int ) → int
    OrdinaCrescente(lun);
    int i := 1;
    while ( i ≤ n && L ≥ lun[i] ) do
        L := L - lun[i];    // diminuisce la lunghezza residua
        i := i + 1;
    endwhile
    return i - 1;
```

L'operazione di ordinamento può essere fatta in tempo $\Theta(n \log n)$ usando un algoritmo di ordinamento generico. Il successivo ciclo while ha costo $O(n)$ nel caso peggiore. Il costo complessivo dell'algoritmo risulta quindi $\Theta(n \log n)$.
Si noti che l'algoritmo di cui sopra restituisce l'output corretto sia nel caso in cui gli n segmenti abbiano complessivamente lunghezza minore o uguale a L, sia nel caso opposto in cui nessuno abbia lunghezza minore o uguale a L (in questo caso l'algoritmo restituisce zero).

Esercizio 6.2

Siete stati assunti alla Microsoft per lavorare alla prossima versione di Word, denominata Word 2030. Il problema che dovete risolvere è il seguente. È data una sequenza di n parole, le cui lunghezze (espresse in punti tipografici, numeri interi) sono memorizzate nel vettore $w[1]$, …, $w[n]$. È necessario suddividere le parole in righe di lunghezza massima pari a L punti tipografici, in modo che lo spazio non utilizzato in ciascuna riga sia minimo possibile. Tra ogni coppia di parole consecutive posizionate sulla stessa riga viene inserito uno spazio che occupa S punti tipografici; nessuno spazio viene inserito dopo l'ultima parola di ogni riga. La lunghezza del testo su ogni riga è quindi data dalla somma delle lunghezze delle parole e degli spazi di separazione. L è maggiore della lunghezza di ogni singola parola (quindi in ogni riga può sempre essere inserita almeno una parola). Non è possibile riordinare le parole, che devono comparire esattamente

nell'ordine dato.

1. Scrivere un algoritmo efficiente che, dato in input il vettore $w[1], \ldots, w[n]$, e i valori S e L, stampi una suddivisione delle parole che minimizza lo spazio inutilizzato in ciascuna riga. Ad esempio, supponendo di avere 15 parole, l'algoritmo potrebbe stampare la stringa "[1 3][4 8][9 15]" per indicare che la prima riga contiene le parole da 1 a 3 (incluse), la seconda le parole da 4 a 8, e la terza le parole da 9 a 15.
2. Analizzare il costo computazionale dell'algoritmo proposto.

Soluzione. È possibile risolvere il problema con un semplice algoritmo greedy: si inseriscono in ciascuna riga le parole, nell'ordine indicato, finché non si supera la lunghezza massima consentita. Utilizziamo la variabile start per indicare l'indice della prima parola della riga corrente; Lres è la lunghezza residua (ancora da riempire) dalla riga corrente.

```
algoritmo FormattaParagrafo( array w[1..n] di int, int S, int L )
    int start := 1;
    int Lres := L - w[1];
    for i := 2 to n do
        if ( Lres ≥ S + w[i]) then// aggiungiamo la parola i-esima alla riga corrente
            Lres := Lres – S - w[i];
        else                                    // iniziamo una nuova riga
            print "[" + start + " " + (i-1) + "]";
            start := i;
            Lres := L - w[i];
        endif
    endfor
    print "[" + start + " " + n + "]";
```

Si noti la stampa effettuata al termine del ciclo "for", senza la quale l'algoritmo non verrebbero stampate le parole dell'ultima riga. Il costo dell'algoritmo è $\Theta(n)$.

Esercizio 6.3

Supponiamo di avere $n \geq 1$ oggetti, ciascuno etichettato con un numero da 1 a n;. l'oggetto i-esimo ha peso $p[i] > 0$. Questi oggetti vanno inseriti all'interno di scatoloni identici, disponibili in numero illimitato, ciascuno in grado di contenere un numero arbitrario di oggetti purché il loro peso complessivo sia minore o uguale a C. Si può assumere che tutti gli oggetti abbiano peso minore o uguale a C. I pesi sono valori reali arbitrari. Vogliamo definire un algoritmo che disponga gli oggetti negli scatoloni in modo da cercare di minimizzare il numero di scatoloni utilizzati. Questo genere di problema è noto col nome di *bin packing problem* ed è computazionalmente intrattabile nel caso generale; di conseguenza, ci accontentiamo di un algoritmo semplice che produca una soluzione non necessariamente ottima.

1. Scrivere un algoritmo basato sul paradigma greedy che, dato il vettore dei pesi $p[1..n]$ e il valore C, restituisce il numero di scatoloni che vengono utilizzati.
2. Calcolare il costo computazionale dell'algoritmo proposto.

Soluzione. Un algoritmo greedy molto semplice consiste nel considerare tutti gli oggetti, nell'ordine in cui sono dati. Per ogni oggetto, si controlla se può essere inserito nello scatolone corrente senza superare il limite di peso. Se ciò non è possibile, si prende un nuovo scatolone e lo si inizia a riempire.

```
algoritmo Scatoloni( array p[1..n] di double, double C ) → int
    int ns := 0;                    // numero di scatoloni utilizzati
    int i := 1;
    while ( i ≤ n ) do
        ns := ns + 1;               // iniziamo a riempire un nuovo scatolone
        double Cres := C;           // capacità residua dello scatolone corrente
        while ( i ≤ n && Cres ≥ p[i] ) do
            Cres := Cres – p[i];
            i := i + 1;
```

```
        endwhile
    endwhile
    return ns;
```

L'algoritmo fa uso della variabile intera ns, che mantiene il numero di scatoloni utilizzati, e della variabile reale Cres che indica la capacità residua dello scatolone corrente. Se la capacità residua Cres supera il peso $p[i]$ dell'oggetto i-esimo, allora tale oggetto può essere inserito nello scatolone; si provvede quindi a decrementare Cres di $p[i]$ e si passa all'oggetto successivo. Quando Cres diventa inferiore a $p[i]$, allora l'oggetto i non trova posto nello scatolone corrente, e si inizia a riempire il successivo.
Il costo computazionale dell'algoritmo proposto è $\Theta(n)$.

Esercizio 6.4

Disponiamo di $n \geq 1$ libri di varia dimensione; il libro i-esimo ha spessore $x[i]$ e altezza $y[i]$; le misure sono numeri reali. Vogliamo disporre i libri uno accanto all'altro, senza riordinarli, su scaffali di lunghezza L. Ciascuno scaffale è in grado di ospitare un numero qualsiasi di libri, purché la somma degli spessori risulti minore o uguale a L; naturalmente nessun libro ha spessore maggiore di L. Ogni scaffale deve essere riempito il più possibile. Gli scaffali vengono posizionati uno sotto l'altro in modo da non sprecare spazio in verticale. L'altezza di ciascuno scaffale è pari all'altezza massima dei libri che contiene. L'altezza complessiva occupata dall'intera collezione di libri è data dalla somma delle altezze dei singoli scaffali.
Scrivere un algoritmo greedy che, dati in input i vettori $x[1..n]$ e $y[1..n]$, e il valore di L, restituisce l'altezza totale occupata dalla collezione di libri. Analizzare il costo asintotico dell'algoritmo proposto.

Soluzione. L'algoritmo sfrutta l'idea seguente: si considerano a turno tutti i libri, nell'ordine in cui sono dati. Si posiziona ciascun libro sullo scaffale corrente finché possibile; contemporaneamente, si tiene traccia dell'altezza massima dei libri delo scaffale. Quando si è finito di riempire uno scaffale, si aggiorna l'altezza complessiva e si procede con lo riempimento dello scaffale successivo.
L'algoritmo può essere descritto nel modo seguente:

```
algoritmo SCAFFALI( array x[1..n] di double, array y[1..n] di double, double L ) → double
    double h := 0;                  // altezza complessiva di tutti gli scaffali
    int i := 1;                     // indice del libro che consideriamo.
    while ( i ≤ n ) do
        double Lres := L;           // larghezza residua scaffale corrente
        double hmax := 0;           // altezza scaffale corrente
        while ( i ≤ n && Lres ≥ x[i] ) do
            Lres := Lres – x[i];    // posiziono il libro i-esimo sullo scaffale
            if ( y[i] > hmax ) then
                hmax := y[i];
            endif
            i := i + 1;
        endwhile
        h := h + hmax;              // aggiorna ltezza scaffale corrente
    endwhile
    return h;
```

Il costo dell'algoritmo è $\Theta(n)$.

Esercizio 6.5

Un commesso viaggiatore utilizza il treno per i suoi spostamenti. Ha pianificato $n \geq 1$ visite ai suoi clienti da effettuare in n giorni diversi $g[1]$, … $g[n]$ dell'anno in corso. L'array $g[1..n]$ contiene valori interi appartenenti all'insieme $\{0, \ldots 364\}$ (dove 0 indica il primo gennaio, 364 indica il 31 dicembre); l'array non contiene valori duplicati ed è ordinato in senso crescente. Il commesso viaggiatore puo' decidere di utilizzare biglietti singoli, del costo di S euro, che valgono per un solo giorno, oppure di acquistare un abbonamento della durata di 30 giorni (incluso il giorno di emissione) del costo di A euro; si può assumere che $A < 30 \times S$.

In generale potrà risultare economicamente conveniente acquistare un mix di biglietti e abbonamenti in modo da poter effettuare tutti i viaggi minimizzando il costo.
Scrivere un algoritmo greedy che dato l'array $g[1..n]$ e i costi S e A del biglietto singolo e dell'abbonamento, determina la combinazione ottima di biglietti e abbonamenti che consentono di effettuare tutti i viaggi al costo minimo possibile.

Soluzione.

```
VIAGGI( array g[1..n] di int, real S, real A )
   int i := 1 ;
   while ( i ≤ n ) do
        int nv := 1; // n. di viaggi che posso fare acquistando un abbonamento il giorno g[i]
        int scad := g[i] + 29; // scadenza abbonamento acquistato il giorno g[i]
        int j := i+1;
        while ( j ≤ n and g[j] ≤ scad ) do
             nv := nv+1;
             j := j+1;
        endwhile
        // nv è il numero di viaggi che posso fare con un abbonamento acquistato il giorno
g[i]
        // g[j] è il giorno del primo viaggio successivo alla scadenza dell'abbonamento
acquistato il giorno g[i]
        // mi conviene un abbonamento oppure nv biglietti singoli?
        if ( nv * S < A ) then
             print "Acquisto biglietto singolo giorno ", g[i]
             i := i + 1;
        else
             print "Acquisto abbonamento giorno ", g[i]
             i := j;
        endif
   endwhile
```

Capitolo 7: Programmazione Dinamica

Esercizio 7.1

Un distributore di bibite contiene al suo interno n monete i cui valori sono rispettivamente $c[1]$, $c[2]$, …, $c[n]$. Tutti i valori sono interi positivi; è possibile che più monete presenti nel distributore abbiano lo stesso valore. Si consideri il problema di decidere se è possibile erogare, in qualsiasi modo, un resto *esattamente uguale a R* utilizzando un opportuno sottoinsieme delle n monete a disposizione; R è un intero positivo.

1. Descrivere un algoritmo efficiente per decidere se il problema ammette una soluzione oppure no.
2. Determinare il costo computazionale dell'algoritmo descritto al punto 1, motivando la risposta
3. Modificare l'algoritmo di cui al punto 1 per determinare anche quali sono le monete da erogare per produrre il resto R. Si noti che non è necessario erogare il resto con il numero minimo di monete: è sufficiente erogarlo in modo qualsiasi.

Soluzione. In assenza di ulteriori informazioni sui valori delle monete a disposizione, gli algoritmi greedy non necessariamente sono in grado di individuare una soluzione; di conseguenza l'unica possibilità è di usare la programmazione dinamica. Il problema proposto si può ricondurre al *subset-sum problem*..
Definiamo la matrice booleana $M[1..n, 0..R]$ tale che $M[i, r] = true$ se e solo se esiste un sottoinsieme delle prime i monete di valore complessivo uguale a r.
Iniziamo innanzitutto a definire i casi base. Se $i=1$ possiamo solo scegliere se usare la prima moneta oppure no. Quindi possiamo erogare solamente un resto pari a zero (non usando la moneta) oppure pari al valore della moneta, $c[1]$. Quindi per ogni $r=0, \ldots, R$ abbiamo:

$$M[1,r]=\begin{cases} true & \text{se } r=0 \text{ oppure } r=c[1] \\ false & \text{altrimenti} \end{cases}$$

Nel caso generale, avendo $i > 1$ monete a disposizione per erogare un resto r ci sono due possibilità:

1. Se $r \geq c[i]$ allora possiamo decidere di usare o meno la i-esima moneta. Se la usiamo, possiamo erogare il resto r se e solo se possiamo erogare $r - c[i]$ usando un sottoinsieme delle rimanenti i-1 monete, cioè se $M[i\text{-}1, r\text{-}c[i]]$ è *true*. Se decidiamo di non usare la moneta i-esima, il resto r è erogabile se e solo se r è erogabile usando un sottoinsieme delle restanti i-1 monete, ossia se $M[i\text{-}1, r]$ è *true*.
2. Se $r < c[i]$, non possiamo usare la moneta i-esima perché il suo valore è superiore alla somma da erogare. Quindi il resto è erogabile se e solo se lo è utilizzando le rimanenti i-1 monete.

Quindi possiamo definire $M[i, r]$, per $i=2, \ldots, n$, $r=0, \ldots, R$ come:

$$M[i,r]=\begin{cases} M[i-1,r] \vee M[i-1,r-c[i]] & \text{se } r \geq c[i] \\ M[i-1,r] & \text{altrimenti} \end{cases}$$

Il nostro problema di partenza (decidere se possiamo erogare il resto R usando un sottoinsieme delle n monete) ammette soluzione se e solo se $M[n, R] = true$. Il calcolo di tutti gli elementi della matrice $M[i, r]$

può essere effettuato in tempo $\Theta(nR)$ usando il consueto schema di programmazione dinamica:

```
algoritmo RESTO( array c[1..n] di int, int R ) → bool
    array M[1..n, 0..R] di bool;
    // inizializza M[1, r]
    for r:=0 to R do
        if ( r == 0 || r == c[1] ) then
            M[1, r] := true;
        else
            M[1, r] := false;
        endif
    endfor
    // calcola i restanti elementi della tabella
    for i := 2 to n do
        for r := 0 to R do
            if ( r ≥ c[i] ) then
                M[i, r] := M[i-1, r] || M[i-1, r-c[i]];
            else
                M[i, r] := M[i-1, r];
            endif
        endfor
    endfor
    return M[n, R];
```

Per determinare le monete da usare, facciamo ricorso ad una ulteriore matrice booleana $U[i, r]$, avente la stessa dimensione di M. $U[i, r]$ == *true* se e solo se usiamo la moneta i-esima per erogare il resto r. Occorre prestare attenzione ai casi in cui i=1: $U[1, r]$ è *false* se r=0, in quanto non usiamo alcuna moneta per erogare il resto zero, mentre è *true* se $r = c[1]$. Gli elementi di U possono essere calcolati contemporaneamente a quelli della matrice M, come segue:

```
algoritmo RESTOMAXMONETE( array c[1..n] di int, int R ) → bool
    array M[1..n, 0..R] di bool;
    array U[1..n, 0..R] di bool;
    // inizializza M[1, r] e U[1, r]
    for r:=0 to R do
        if ( r == c[1] ) then
            M[1, r] := true;
            U[1, r] := true;
        elseif ( r == 0 ) then
            M[1, r] := true;
            U[1, r] := false;
        else
            M[1, r] := false;
            U[1, r] := false;
        endif
    endfor
    // calcola i restanti elementi delle tabelle
    for i:=2 to n do
        for r := 0 to R do
            if ( r ≥ c[i] ) then
                M[i, r] := M[i-1, r] || M[i-1, r-c[i] ];
                U[i, r] := M[i-1, r-c[i] ];
            else
                M[i, r] := M[i-1, r];
                U[i, r] := false;          // non usiamo la moneta i-esima
            endif
        endfor
    endfor
    if ( M[n, R] == true ) then
```

```
        int i := n;
        int r := R;
        while ( r > 0 ) do
            if ( U[i, r] == true )  then
                print "uso la moneta" i;
                r := r - c[i];
        endif
        i := i - 1;
    endwhile
else
    print "nessuna soluzione";
endif
return M[n, R];
```

Esercizio 7.2

Supponiamo di avere n files aventi rispettivamente dimensione $F[1]$, $F[2]$, …, $F[n]$; le dimensioni sono numeri interi strettamente positivi e sono espresse in MB. Disponiamo di un CD-ROM avente capacità 650MB; sfortunatamente, il CD-ROM potrebbe non essere sufficientemente capiente per memorizzare tutti gli n files.

1. Scrivere un algoritmo efficiente per determinare il *numero massimo di files* che è possibile memorizzare sul CD-ROM senza eccederne la capacità. Non è richiesto che l'algoritmo stampi anche quali file memorizzare.
2. Analizzare il costo computazionale dell'algoritmo proposto.

Soluzione. Questo problema si puo' risolvere sia utilizzando la programmazione dinamica, che mediante una tecnica greedy.
Volendo utilizzare la programmazione dinamica, definiamo la matrice di interi $M[1..n, 0..650]$ tale che $M[i, j]$ rappresenta il massimo numero di file, scelti tra $\{1, ..., i\}$, che è possibile copiare su un supporto di capacità j MB. Per ogni j=0, …, 650 possiamo porre:

$$M[1, j] = \begin{cases} 1 & \text{se } j \geq F[1] \\ 0 & \text{altrimenti} \end{cases}$$

In generale, l'elemento $M[i, j]$, per $i = 2, \ldots n, j$=0, …, 650 può essere calcolato come:

$$M[i, j] = \begin{cases} max(M[i-1, j], M[i-1, j-F[i]]+1) & \text{se } j \geq F[i] \\ M[i-1, j] & \text{altrimenti} \end{cases}$$

In altre parole: se la dimensione $F[i]$ del file i-esimo supera la capacità j, il file non può essere copiato e quindi $M[i, j] := M[i\text{-}1, j]$. Invece, se $F[i]$ è minore o uguale alla capacità j, occorre scegliere se copiare il file oppure no. Se decidiamo di non copiarlo, il numero di files sarà $M[i\text{-}1, j]$. Se decidiamo di copiarlo, il numero di files sarà $1+M[i\text{-}1, j\text{-}F[i]]$. La scelta corretta è quella che massimizza il numero di files sul supporto.
L'algoritmo può essere descritto con lo pseudocodice seguente:

```
algorithm MaxNumFilesDinamica( array F[1..n] di int ) → int
    array M[1..n, 0..650] di int;
    // inizializza M[1, j]
    for j := 0 to 650 do
        if ( j ≥ F[1] ) then
            M[1, j] := 1;
        else
            M[1, j] := 0;
        endif
```

```
    endfor
    // Riempi il resto della matrice
    for i := 2 to n do
        for j := 0 to 650 do
            if ( j ≥ F[i] && M[i - 1, j – F[i]] + 1 > M[i - 1, j] ) then
                M[i, j] := M[i - 1, j - F[i]] + 1;
            else
                M[i, j] := M[i-1, j];
            endif
        endfor
    endfor
    return M[n, 650];
```

Il costo computazionale è $\Theta(n \times 650) = \Theta(n)$ (650 è una costante che non dipende dal numero di files, quindi può essere omessa dalla notazione asintotica).
Possiamo ottenere il risultato corretto (e ottimo) usando una tecnica greedy. L'idea è di ordinare i files in ordine non decrescente in base alla dimensione. Fatto questo, copiamo sul CD-ROM i files a partire da quello di dimensione minore, fino a quando c'è spazio.

```
algoritmo MaxNumFilesGreedy( array F[1..n] di int ) → int
    OrdinaCrescente(F);
    i := 1;         // cursore sull'array F
    D := 650;       // spazio disponibile, inizialmente 650MB
    while ( i ≤ n && D ≥ F[i] ) do
        D := D – F[i];
        i := i+1;
    endwhile
    return i-1;
```

Si noti il controllo "i ≤ n" che deve essere effettuato per evitare di accedere all'elemento $F[n+1]$ che non esiste; questo si verificherebbe nel caso in cui la somma delle dimensioni di tutti i files sia minore o uguale a 650, e quindi tutti i files possano essere memorizzati nel CD-ROM.
Il costo computazionale di MaxNumFilesGreedy è dominato dall'operazione di ordinamento, che può essere realizzata in tempo $O(n \log n)$ utilizzando un algoritmo di ordinamento generico, ad esempio MergeSort. Sfruttando il fatto che l'array $F[1..n]$ è un array di interi, sarebbe anche possibile utilizzare un algoritmo di ordinamento più efficiente che operi in tempo lineare.

Esercizio 7.3

Una emittente televisiva deve decidere quali spot mandare in onda durante un intervallo pubblicitario della durata di T secondi. L'emittente dispone di $n > 1$ spot pubblicitari; per ogni i=1, 2, ... n, sia $t[i]$ la durata (in secondi) dello spot i-esimo, e $P[i]$ il prezzo che l'inserzionista paga all'emittente se viene mandato in onda lo spot i-esimo. L'emittente può mandare in onda un sottoinsieme degli n spot a disposizione, purché la durata complessiva non superi T secondi. Ciascuno spot può essere mandato in onda al massimo una volta. Tutte le durate sono numeri interi.

1. Descrivere un algoritmo efficiente per calcolare il guadagno massimo ottenibile dall'emittente mandando in onda un opportuno sottoinsieme degli n spot. Il guadagno complessivo è rappresentato dalla somma dei prezzi $P[i]$ degli spot mandati in onda. È richiesto il calcolo della soluzione esatta
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.
3. Modificare l'algoritmo di cui al punto 1 per restituire, oltre al guadagno massimo, anche l'elenco degli spot da mandare in onda per massimizzare il guadagno.

Esercizio 7.4

Disponiamo di un tubo metallico di lunghezza L. Da questo tubo vogliamo ottenere al più n segmenti più corti, aventi rispettivamente lunghezza $lun[1]$, $lun[2]$, …, $lun[n]$, che possiamo rivendere al prezzo

rispettivamente $p[1]$, $p[2]$, …, $p[n]$. Il tubo viene segato sempre a partire da una delle estremità, quindi ogni taglio riduce la sua lunghezza della misura asportata. Tutte le lunghezze sono espresse in cm, e sono intere.

1. Scrivere un algoritmo efficiente per determinare il massimo ricavo che è possibile ottenere suddividendo opportunamente il tubo di lunghezza L. Non è necessario utilizzare l'intero tubo.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Sia $R[i, j]$ il ricavo massimo che è possibile ottenere da un opportuno sottoinsieme dei segmenti {1, …, i} a partire da un tubo di lunghezza j, con i=1, …, n, j=0, …, L. La soluzione al problema sarà quindi il valore di $R[n, L]$.
Avendo solo il primo segmento a disposizione, per ogni j=0, …, L possiamo scrivere:

$$R[1, j] = \begin{cases} p[1] & \text{se } j \geq lun[1] \\ 0 & \text{altrimenti} \end{cases}$$

In generale per ogni i=2, …, n, j=0, …, L si ha:

$$R[i, j] = \begin{cases} max\left(R[i-1, j], R[i-1, j-lun[i]] + p[i]\right) & \text{se } j \geq lun[i] \\ R[i-1, j] & \text{altrimenti} \end{cases}$$

Lo pseudocodice dell'algoritmo è il seguente.

```
algoritmo TagliaTubi( array lun[1..n] di int, array p[1..n] di double ) → double
   array R[1..n, 0..L] di double
   // Inizializza R[1, j]
   for j:=0 to L do
        if ( j ≥ lun[1] ) then
             R[1, j] := p[1];
        else
             R[1, j] := 0;
        endif
   endfor
   // Calcolo tabella di programmazione dinamica
   for i:=2 to n do
        for j:=0 to L do
             if ( j ≥ lun[i] && R[i-1, j] < R[i-1, j-lun[i]] + p[i] ) then
                  R[i, j] := R[i-1, j-lun[i]] + p[i];
             else
                  R[i, j] := R[i-1, j];
             endif
        endfor
   endfor
   return R[n, L];
```

Il costo computazionale dell'algoritmo è $\Theta(nL)$.

Esercizio 7.5

Dovete affrontare l'esame di Filologia Bizantina. L'esame si compone di n domande che valgono rispettivamente $p[1]$, ..., $p[n]$ punti. In base alla vostra esperienza, stimate che le domande richiederanno rispettivamente $t[1]$, …, $t[n]$ minuti per essere svolte. Purtroppo il tempo a vostra disposizione è di T minuti, che potrebbe essere inferiore alla somma dei tempi necessari a rispondere a tutte le domande. I punteggi e i tempi sono interi strettamente positivi.

1. Scrivere un algoritmo efficiente che, dati i vettori $p[1..n]$, $t[1..n]$ e il valore di T, restituisce il punteggio massimo che potete ottenere rispondendo correttamente ad un opportuno sottoinsieme delle n domande entro il tempo massimo di T minuti. Nota: non è richiesto di calcolare esplicitamente quali sono le domande che consentono di ottenere il punteggio massimo.

2. Calcolare il costo computazionale dell'algoritmo di cui al punto 1

Soluzione. Sia $P[i, j]$ il punteggio massimo che è possibile ottenere rispondendo correttamente ad un opportuno sottoinsieme delle domande $\{1, \ldots, i\}$ avendo a disposizione il tempo massimo di j minuti, per $i=1, \ldots, n, j=0, \ldots, T$. La soluzione al problema sarà quindi il valore di $P[n, T]$.
La tabella di programmazione dinamica è così definita. Per ogni $j=0, \ldots T$ si ha

$$P[1, j]=\begin{cases} p[1] & \text{se } j \geq t[1] \\ 0 & \text{altrimenti} \end{cases}$$

e per ogni $i=2, \ldots n, j=0, \ldots T$:

$$P[i, j]=\begin{cases} max\{P[i-1, j], P[i-1, j-t[i]]+p[i]\} & \text{se } j \geq t[i] \\ P[i-1, j] & \text{altrimenti} \end{cases}$$

per ogni $i=2, \ldots, n, j=0, \ldots, T$
L'algoritmo risolutivo può quindi essere espresso in questo modo:

```
algoritmo FILOLOGIABIZANTINA( array p[1..n] di int, array t[1..n] di int, int T ) → int
    array P[1..n, 0..T] di int
    // Inizializza P[1, j]
    for j:=0 to T do
        if ( j ≥ t [1] ) then
            P[1, j] := p[1];
        else
            P[1, j] := 0;
        endif
    endfor
    // Calcolo tabella di programmazione dinamica
    for i:=2 to n do
        for j:=0 to T do
            if ( j ≥ t[i] && P[i - 1, j] < P[i - 1, j - t[i]] + p[i] ) then
                P[i, j] := P[i - 1, j - t[i]] + p[i];
            else
                P[i, j] := P[i - 1, j];
            endif
        endfor
    endfor
    return P[n, T];
```

Il costo dell'algoritmo è $\Theta(nT)$.

Esercizio 7.6

Allo scopo di prepararvi adeguatamente all'esame di Algoritmi e Strutture Dati, decidete di dedicare 3 ore (180 minuti) per ripassare gli argomenti del corso. Durante il corso sono stati trattati n argomenti, e stimate che il tempo necessario per ripassare l'argomento i-esimo sia di $T[i]$ minuti; tutti i tempi sono numeri interi. Inoltre, ad ogni argomento i associate un numero reale positivo $V[i]$ che ne denota l'importanza (maggiore è il valore $V[i]$, più importante è l'argomento). Scrivere un algoritmo efficiente per determinare l'importanza complessiva massima degli argomenti che potete ripassare entro i 180 minuti a vostra disposizione. Non è richiesto di indicare anche quali sono gli argomenti che massimizzano l'importanza.

Esercizio 7.7

Si consideri un array $A[1..n]$ composto da n numeri reali distinti, non ordinato. Si vuole determinare la massima lunghezza di una sottosequenza di A composta da elementi in ordine crescente. Non è richiesto che

la sottosequenza sia composta da elementi contigui.
Ad esempio, supponendo che A=[4, 1,-1,3, 10, 11], possibili sottosequenze composte da valori crescenti sono [1, 3, 11], oppure [4, 10, 11], oppure [1, 3, 10, 11] (si noti che i valori vanno considerati nell'ordine in cui compaiono in A). Nel caso dell'array di esempio, l'algoritmo deve restituire il valore 4 in quanto la più lunga sottosequenza di valori crescenti ha lunghezza 4 (in particolare [1, 3, 10, 11] oppure [-1, 3, 10, 11] sono due sottosequenze di lunghezza massima).

1. Scrivere un algoritmo basato sulla programmazione dinamica per calcolare la massima lunghezza di una sottosequenza crescente. (Suggerimento: sia $L[j]$ la massima lunghezza di una sottosequenza crescente di $A[1..j]$ avente $A[j]$ come ultimo elemento)
2. Calcolare il costo computazionale dell'algoritmo di cui al punto precedente.

Soluzione. Sfruttiamo il suggerimento. Notiamo innanzitutto che $L[1] = 1$, poiché la sottosequenza composta dal singolo elemento $A[1]$ è crescente, e ha lunghezza 1. Per calcolare il generico valore $L[j]$, avendo già calcolato $L[1]$, L[2], …, $L[j\text{-}1]$ ragioniamo come segue: la più lunga sottosequenza crescente di $A[1..j]$ che termina in $A[j]$ avrà un penultimo elemento $A[k]$, per un certo $k < j$. Quindi la più lunga sottosequenza che termina in $A[j]$ sarà composta da un primo pezzo di lunghezza $L[k]$, che ha $A[k]$ come ultimo elemento, a cui si aggiunge $A[j]$. Quindi $L[j] = L[k] + 1$. Il problema è che non sappiamo quale sia il valore di k corretto. Esaminiamo quindi tutti i valori di $k < j$ per cui si verifica $A[k] < A[j]$ (ricordiamo che gli elementi della sottosequenza devono essere ordinati in senso crescente), e tra questi valori scegliamo quello per cui $L[k]$ è massimo. Nel caso in cui non esistano valori di k che soddisfano le condizioni sopra, il massimo viene posto a zero.
Il calcolo di $L[j]$ si può effettuare con la seguente equazione:

$$L[j]=\begin{cases} 1 & \text{se } j=1 \\ 1+max_{1\le k<j,\, A[k]<A[j]} L[k] & \text{altrimenti} \end{cases}$$

Una volta calcolato $L[j]$ per ogni j=1, …, n, il valore richiesto dal problema è il massimo tra tutti gli $L[j]$. Infatti, non sappiamo quale sia l'ultimo elemento della sottosequenza crescente di lunghezza massima, per cui dobbiamo esaminarle tutte.

```
algoritmo MaxLunSottosequenza( array A[1..n] di double ) → int
   // Prima fase: calcola L[j] per ogni j
   array L[1..n] di int;
   L[1] := 1;
   int Lmax := L[1];     // lunghezza massima di una sottosequenza crescente
   for j:=2 to n do
        int maxlun := 0;
        for k:=1 to j-1 do
             if ( A[k] < A[j] && L[k] > maxlun ) then
                  maxlun := L[k];
             endif
        endfor
        L[j] := 1 + maxlun;
        if ( L[j] > Lmax ) then
             Lmax := L[j];
        endif
   endfor
   return Lmax;
```

Notiamo che il corpo del ciclo for più interno fase ha costo O(1), e viene eseguito 1 + 2 + … + (n-2) + (n-1) = $\Theta(n^2)$ volte, per cui il costo complessivo dell'algoritmo è $\Theta(n^2)$.

Esercizio 7.8

Sono dati due array $A[1..n]$ e $V[1..n]$ composti da $n \geq 2$ numeri reali positivi. $A[i]$ e $V[i]$ rappresentano rispettivamente il prezzo unitario di acquisto e di vendita delle azioni di una certa società al giorno i.

Vogliamo individuare due giorni $i < j$ tali da massimizzare il guadagno unitario (o minimizzino la perdita unitaria) ottenibile acquistando una azione il giorno i e rivendendola il giorno j. È possibile vendere solo dal giorno successivo all'acquisto in poi; è consentito un singolo acquisto e una singola vendita.

1. Scrivere un algoritmo che, dati in input i vettori A e V, determini i valori di i e j che massimizzano il guadagno unitario.
2. Analizzare il costo computazionale dell'algoritmo proposto

Soluzione. Una soluzione non efficiente, ma corretta, consiste nel controllare tutte le combinazioni possibili:

```
algoritmo CompraVendi1( array A[1..n] di double, array V[1..n] di double ) → (int, int)
   double Gmax := V[2] - A[1];
   int imax := 1;
   int jmax := 2;
   for i:=1 to n-1 do
        for j := i + 1 to n do
             if ( V[j] – A[i] > Gmax ) then
                  Gmax := V[j] – A[i];
                  imax := i;
                  jmax := j;
             endif
        endfor
   endfor
   return imax, jmax;
```

Il costo di questo algoritmo è $\Theta(n^2)$; è importante osservare che (1) inizializzare Gmax (il valore del guadagno massimo) con 0 sarebbe stato sbagliato, in quanto l'algoritmo non avrebbe fornito la risposta corretta nel caso in cui tutti i guadagni fossero stati negativi (ossia tutte le possibili transazioni si concludevano in perdita); (2) dimenticarsi di inizializzare imax e jmax fuori dai cicli for sarebbe stato ugualmente sbagliato, in quanto in tal caso l'algoritmo avrebbe restituito valori indefiniti nel caso n=2.

Il problema ammette una soluzione più efficiente, che richiede tempo lineare nella dimensione degli array A e V. Una variante che richiede tempo $\Theta(n)$ e spazio aggiuntivo $\Theta(1)$ è la seguente:

```
algoritmo CompraVendi2( array A[1..n] di double, array V[1..n] di double ) → (int, int)
   double Gmax := V[2] - A[1];// guadagno massimo
   int ac := 1;
   int ve := 2;
   int imin := 1;                    // A[imin] è il minimo prezzo di acquisto in A[1..j-1]
   for j:=2 to n do
        if ( V[j] – A[imin] > Gmax ) then
             Gmax := V[j] – A[imin];
             ac := imin;
             ve := j;
        endif
        if ( A[j] < A[imin] ) then
             imin := j;
        endif
   endfor
   return imin, jmax;
```

Questo algoritmo effettua una singola scansione degli array, utilizzando tre cursori: ac e ve indicano, come prima, gli indici dei giorni in cui acquistare e vendere, rispettivamente. In più, viene utilizzato un ulteriore cursore imin, che ha il significato seguente: all'inizio di ogni iterazione del ciclo for, A[imin] è il valore minimo del sottovettore $A[1..j - 1]$. In altre parole, se decidessimo di vendere il giorno j, il valore di acquisto più basso in tutti i giorni precedenti sarebbe A[imin] in corrispondenza del giorno imin. Pertanto, volendo vendere il giorno j, otterremmo il massimo guadagno $V[j] – A$[imin]. acquistando al giorno imin. Se tale guadagno è superiore al massimo osservato fino a quel momento, aggiorniamo i cursori.

Esercizio 7.9

Dopo l'ennesimo aumento ingiustificato delle tasse, avete deciso di adottare una forma di protesta non violente che consiste nel pagare l'importo dovuto interamente in monetine. Vi recate quindi presso la vostra banca per cambiare delle banconote, ottenendo in cambio un grosso sacchetto contenente n monete i cui valori, espressi in centesimi di euro, sono rispettivamente $c[1..n]$. Per rendere ancora più significativa la vostra protesta, decidete di erogare l'importo dovuto di tasse, pari a R centesimi, usando il *massimo* numero possibile di monete. I valori delle monete e l'importo R sono tutti interi strettamente positivi.
Scrivere un algoritmo che, dato in input l'array c e il valore R, calcoli il massimo numero di monete che sono necessarie per erogare un importo esattamente pari a R, ammesso che ciò sia possibile. Analizzare il costo computazionale dell'algoritmo proposto.

Soluzione. Per risolvere questo problema si potrebbe essere tentati di utilizzare un approccio greedy "inverso" rispetto a quello adottato per risolvere il problema del resto, ossia di ordinare i valori delle monete in senso crescente, ed erogare le monete di valore minore.

```
algoritmo MaxNumMoneteSbagliato( array c[1..n] di int, int R ) → int
   OrdinaCrescente(c);
   int i := 1;    // indice della moneta che stiamo considerando
   while ( R > 0 && i ≤ n ) do
       if ( c[i] > R ) then
           errore: resto non erogabile;
       else
           R := R – c[i];
           i := i+1;
       endif
   endwhile
   if ( R > 0 ) then
       errore: resto non erogabile;
   else
       return i-1; // abbiamo utilizzato le prime i-1 monete
   endif
```

Purtroppo tale approccio non funziona: consideriamo infatti i valori seguenti (l'array è già ordinato): c = [1, 2, 2, 2, 5]. Dovendo erogare un importo R=6, è facile convincersi che il massimo numero di monete richiesto è 3 (servono le tre monete da 2). L'algoritmo MaxNumMoneteSbagliato invece sceglierebbe subito la moneta da 1, poi due monete da 2 e a quel punto non sarebbe più in gredo di erogare il residuo di 1 centesimo.
Utilizziamo quindi la programmazione dinamica. Sia $N[i, r]$ il massimo numero di monete scelte tra l'insieme $\{1, \ldots, i\}$ che sono necessarie per erogare un importo esattamente pari a r (per ogni i=1, …, n, r=0, …, R); se non è possibile erogare r, poniamo $N[i, r] = -\infty$.
Avendo solo la prima moneta a disposizione (i=1) abbiamo:

$$N[1,r]=\begin{cases} 0 & \text{se } r=0 \\ 1 & \text{se } r=c[1] \\ -\infty & \text{altrimenti} \end{cases}$$

Nel caso più generale,, il calcolo di $N[i, r]$ si effettua come:

$$N[i,r]=\begin{cases} max\{N[i-1,r],\,1+N[i-1,r-c[i]]\} & \text{se } r\geq c[i] \\ N[i-1,r] & \text{altrimenti} \end{cases}$$

In altre parole, se l'importo da erogare è minore del valore $c[i]$ della i-esima moneta, allora l'importo r dovrà essere erogato usando solo le rimenenti monete $\{1, \ldots i - 1\}$. Se r è maggiore o uguale a $c[i]$, allora abbiamo due possibilità: erogare r senza usare la moneta i-esima, oppure usare usare la moneta i-esima, ed erogare la somma residua r-$c[i]$ mediante le rimanenti monete $\{1, \ldots, i - 1\}$.
L'algoritmo può essere descritto nel modo seguente:

```
algoritmo MaxNumMonete( array c[1..n] di int, int R ) → int
    array N[1..n, 0..R] di int;
    // Calcola N[1, r]
    for r:=0 to R do
        if ( r == 0 ) then
            N[1, r] := 0;
        elseif ( r == c[1] ) then
            N[1, r] := 1;
        else
            N[1, r] := -INF;
        endif
    endfor
    // Calcola tabella di programmazione dinamica
    for i:=2 to n
        for r:=0 to R do
            // NOTA: si richiede che (1 + -INF) == -INF
            if ( r ≥ c[i]  && N[i - 1, r] < 1+N[i - 1, r - c[i]] ) then
                N[i, r] := 1 + N[i - 1, r – c[i]];
            else
                N[i, r] := N[i - 1, r];
            endif
        endfor
    endfor
    return N[n, R];
```

L'uso del valore -INF tra gli elementi della matrice $N[i, r]$ richiede una certa attenzione; è infatti necessario che l'eventuale linguaggio di programmazione usato nell'implementazione dello pseudocodice precedente garantica che 1 + (-INF) = -INF. In Java (e molti altri linguaggi che supportano l'aritmetica in virgola mobile secondo lo standard IEEE 754) è possibile usare la costante Double.NEGATIVE_INFINITY definita nella classe Double che ha esattamente le proprietà richieste. Naturalmente in questo caso bisognerebbe definire la matrice $N[i, r]$ come matrice di Double anziché di interi.
Il costo computazionale è $\Theta(Rn)$

Esercizio 7.10

Un gruppo di $n>0$ persone deve salire su un ascensore che può sostenere un peso massimo di W kg. Indichiamo con $p[1], \ldots p[n]$ i pesi (in kg) delle n persone. Tutti i pesi sono numeri interi.

1. Scrivere un algoritmo per determinare se esiste o meno un opportuno sottoinsieme di persone il cui peso complessivo sia *esattamente uguale* alla capacità dell'ascensore W (non è richiesto di indicare anche quali persone farebbero parte di tale gruppo).
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Esercizio 7.11

Una emittente televisiva deve organizzare il palinsesto di una determinata giornata. La giornata si compone di 1440 minuti (pari a 24 ore). L'emittente dispone di una lista di $n\geq 1$ programmi le cui durate in minuti sono rispettivamente $d[1], \ldots, d[n]$. Tutti i tempi sono numeri interi.

1. Scrivere un algoritmo che restituisca true se e solo se esiste un opportuno sottoinsieme degli n programmi la cui durata complessiva sia esattamente di 1440 minuti.
2. Analizzare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Il problema si risolve mediante la programmazione dinamica. Definiamo una matrice $B[1..n, 0..1440]$, tale che $B[i, j]$ = true se e solo se esiste un qualche sottoinsieme dei primi i programmi $\{1, \ldots, i\}$ la cui durata complessiva sia esattamente j.
Se i=1 abbiamo che $B[i, j] = true$ se e solo se $j = d[1]$ oppure $j = 0$. Per ogni $i=2, \ldots n, j=0, \ldots 1440$, gli altri elementi della matrice sono definiti in questo modo:

$$B[i\,,\,j]=\begin{cases} B[i-1,\,j]\vee B[i-1,\,j-d[i]] & \text{se } \; j\geq d[i] \\ B[i-1,\,j] & \text{altrimenti} \end{cases}$$

Per occupare un tempo j con un opportuno sottoinsieme dei programmi $\{1, \ldots, i\}$ possiamo: (i) riempire il tempo j senza il programma i-esimo, nel caso in cui la durata del programma i-esimo superi j, oppure (ii) scegliere se usare o meno il progamma i-esimo nella nostra allocazione.
Quindi l'algoritmo calcola tutti i valori $B[i, j]$, e restituisce $B[n, 1440]$ come soluzione.

```
algoritmo PALINSESTO(array d[1..n] di int ) → bool
   array B[1..n, 0..1440] di bool;
   // Inizializziamo B[1, j]
   for j:=0 to 1440 do
        if ( j == 0 || j == d[1] ) then
             B[1, j] := true;
        else
             B[1, j] := false;
        endif
   endfor
   // Riempiamo la matrice B
   for i:=2 to n do
        for j:=0 to 1440 do
             if ( j ≥ d[i] ) then
                  B[i, j] := B[i-1, j] || B[i-1, j-d[i]];
             else
                  B[i, j] := B[i-1, j];
             endif
        endfor
   endfor
   return B[n, 1440];
```

Il costo dell'algoritmo è $\Theta(n\times 1440) = \Theta(n)$

Esercizio 7.12

Supponiamo di voler indovinare un numero scelto da un avversario all'interno dell'insieme $\{1, 2, ..., n\}$. Ogni nostro tentativo sbagliato ci costa un numero di "gettoni" pari al valore del numero che abbiamo scelto. Per ogni tentativo, l'avversario ci può rispondere in uno di questi tre modi:

1. "Il numero che hai scelto è quello corretto".
2. "Il numero che hai scelto è maggiore di quello che ho in mente io".
3. "Il numero che hai scelto è minore di quello che ho in mente io".

Scrivere un algoritmo per decidere, ad ogni turno, quale è il numero più conveniente da scegliere in modo da minimizzare il costo totale nel caso peggiore. (Si noti che nel caso in cui il costo di ogni tentativo fosse unitario, la soluzione consisterebbe nell'usare l'algoritmo di ricerca binaria. Nel nostro caso tale algoritmo non necessariamente produrrebbe la sequenza di scelte di costo minore nel caso pessimo).

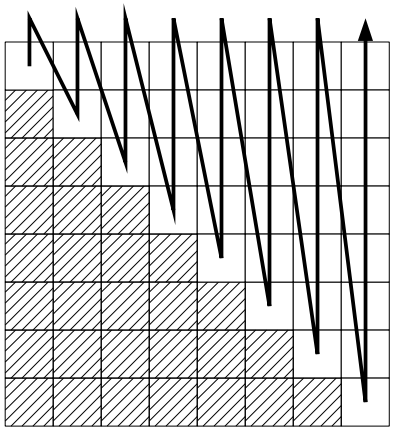
Soluzione. Questo problema è molto simile a quello della parentesizzazione ottima visto a lezione. Supponiamo di indicare con $C[i, j]$ il costo minimo (nel caso peggiore) necessario per indovinare un numero nell'insieme $\{i, ..., j\}$. Chiaramente, $C[i, i] = 0$ per ogni i=1, ..., n (per indovinare un numero in un insieme composto da un solo numero non ci sono scelte da compiere). Per calcolare $C[i, j]$ in generale, definiamo una funzione $f(i, j, k)$ aggiuntiva, che rappresenta il costo ottimo necessario per indovinare un numero nell'insieme $\{i, ..., j\}$ assumendo di usare k come tentativo iniziale.

$$f\,(i\,,\,j\,,\,k)=\begin{cases} k+C[k+1,\,j] & \text{se } \; k=i \\ k+max\,(C\,[i\,,k-1],C[k+1,\,j]) & \text{se } \; i<k<j \\ C[i\,,\,k-1]+k & \text{se } \; k=j \end{cases}$$

da cui possiamo definire $C[i, j]$, per $i < j$, come:

$$C[i, j] = min\{f(i, j, k), k = i, ..., j\}$$

Allo scopo di sapere quali sono le "mosse" migliori da fare (cioè quali numeri conviene selezionare nei vari tentativi), si può usare un'altra tabella *sel* tale che $sel[i, j]$ rappresenta l'intero k appartenente a $\{i, ..., j\}$ da usare come primo tentativo per individuare un numero compreso in quell'intervallo. Gli elementi di $sel[i, j]$ vanno riempiti con il valore di k che minimizza i valori di $C[i, j]$.
Per garantire che ad ogni passo tutti i valori necessari a calcolare $C[i, j]$ siano già stati calcolati, gli elementi di C vanno calcolati secondo lo schema seguente (le celle in grigio non vengono calcolate):



Esempio, con n=5:

Matrice $C[i, j]$

0	1	2	4	6
-	0	2	3	6
-	-	0	3	4
-	-	-	0	4
-	-	-	-	0

matrice sel$[i, j]$

1	1	2	1	2
	2	2	3	2
		3	3	4
			4	4
				5

```
algoritmo StrategiaOttima( int n )
   array sel [1..n, 1..n] di int;
   array C[1..n, 1..n] di int;
   // inizializzazione degli elementi diagonali si C[] e sel[]
   for i := 1 to n do
         C[i, i] := 0;
```

```
        sel[i, i] := i;
    endfor
    // calcoliamo i valori C[i, j] usando la programmazione dinamica
    for j:=2 to n do
        for i := j - 1 downto 1 do
            int val;
            C[i, j] := +inf; // inizializzazione a +infinito
            for k := i to j do
                if ( k == i ) then
                    val = k+C[k+1, j];
                elseif ( k == j ) then
                    val := C[i, k-1] + k;
                else
                    val := k + max( C[i, k-1], C[k+1, j] );
                endif
                if ( val < C[i, j] ) then
                    C[i, j] := val;
                    sel[i, j] := k;
                endif
            endfor
        endfor
    endfor
```

Esercizio 7.13

Consideriamo $n \geq 1$ dischi etichettati come $\{1, 2, ... n\}$. Il disco i ha diametro $d[i]$; tutti i diametri sono numeri reali positivi distinti, ma i dischi non sono ordinati in alcun modo. Vogliamo selezionare un sottoinsieme di dischi da impilare uno sull'altro, rispettando le seguenti regole:

- E' vietato posizionare un disco più grande sopra a uno più piccolo (è la stessa regola delle torri di Hanoi);
- E' vietato posizionare il disco i sopra al disco j se $i > j$.

In altre parole, è consentito solamente posizionare un disco piccolo sopra a uno grande (non viceversa), e questo purché il disco piccolo abbia un'etichetta strettamente inferiore al disco grande. Scrivere un algoritmo efficiente in grado di individuare l'altezza massima della pila di dischi che è possibile realizzare rispettando le regole di cui sopra. L'algoritmo deve anche esplicitamente indicare quali sono i dischi da impilare.
Ad esempio, supponiamo che i diametri siano 2, 5, 3, 1, 8, 10. La pila di altezza massima ottenibile è composta dai dischi 1, 3, 5, 6 (il disco 1 sta in cima, il disco 6 sta alla base), dato che $d[1] = 2 < d[3] = 3 < d[5] = 8 < d[6] = 10$.

Soluzione. Questo esercizio è del tutto simile all'esercizio 7.7 e può essere risolto allo stesso modo. Sia $P(i)$ il sottoproblema che consiste nell'impilare il maggior numero di dischi scelti tra $\{1, ... i\}$, rispettando i vincoli del testo, usando sempre il disco i come base della pila. Denotiamo con $H[i]$ la soluzione del problema $P(i)$: $H[i]$ è il massimo numero di dischi che possono essere impilati utilizzando il disco i come base.
Chiaramente $H[1] = 1$. Per ogni i=2, ... n ragioniamo come segue. La pila di altezza massima avente il disco i come base si ottiene impilando sopra a i un disco j scelto tra $\{1, ... i\text{-}1\}$ tale che $d[j] < d[i]$, e $H[j]$ sia massimo possibile. Questo può essere scritto nel modo seguente:

$$H[i] = 1 + \max\{H[j],\ \ j = 1, ...i-1,\ \ d[j] < d[i]\}$$

Nel caso in cui non esista alcun valore $j < i$ per cui $d[j] < d[i]$ (questo è il caso, ad esempio, se i diametri dei dischi sono presentati in ordine decrescente), allora $H[i] = 1$.
L'altezza massima H' della pila che è possibile ottenere sfruttando tutti i dischi sarà quindi:

$$H' = \max\{H[i],\ \ i = 1, \ldots n\}$$

Per poter determinare i dischi che fanno parte della soluzione ottima, possiamo fare uso di un vettore di interi up[1..*n*]. up[*i*] contiene l'indice del disco che viene posizionato immediatamente sopra al disco *i* per ottenere la pila di altezza *H*[*i*]; up[*i*] vale zero se *H*[*i*] = 1.

```
algoritmo PilaAltezzaMax( array d[1..n] di double ) → int
   array H[1..n] di int;
   array up[1..n] di int;
   H[1] := 1;
   up[1] := 0;
   int ibase := 1 ;   // indice del disco che funge da base alla pila di altezza massima
   for i:=2 to n do
        H[i] := 1;
        up[i] := 0;
        for j:=1 to i-1 do
             if ( d[j] < d[i] && H[j]+1 > H[i] ) then
                  H[i] := H[j] + 1;
                  up[i] := j;
             endif
        endfor
        if ( H[i] > H[ibase] ) then
             ibase := i;
        endif
   endfor
   print "Il numero massimo di dischi impilabili è " H[ibase];
   // ora stampiamo i dischi della pila di altezza massima, partendo dalla base
   i := ibase; // i rappresenta l'indice del disco
   while (i > 0) do
        print "Uso il disco " i;
        i := up[i];
   endwhile
   return H[n];
```

Ad esempio, se il *d*[1..*n*] è definito come segue, con *n*=6:

2	5	3	1	8	10

l'algoritmo calcola il seguente vettore *H*[1..*n*] delle altezze:

1	2	2	1	3	4

e il seguente vettore dei dischi "predecessori" up[1..*n*]:

0	1	1	0	2	5

In realtà è possibile ricostruire la soluzione ottima (l'elenco dei dischi da impilare) anche senza usare il vettore ausiliario up[1..*n*], ragionando come segue. Se abbiamo in qualche modo determinato che il disco *i* appartiene alla soluzione ottima, procediamo a ritroso per individuare l'indice $j < i$ tale che $H[j] = H[i] - 1$. L'indice *j* ci fornisce l'indice del disco posizionato immediatamente sopra a *i* nella soluzione ottima. Ovviamente ci si ferma non appena si trova *H*[*i*] = 1. Questo meccanismo, sebbene consenta di evitare l'uso dell'array ausiliario, ha lo svantaggio che il ciclo di stampa della soluzione è non banale e va implementato con cura.

```
algoritmo PilaAltezzaMax( array d[1..n] di double ) → int
   array H[1..n] di int;
   H[1] := 1;
   int ibase := 1 ;   // indice del disco che funge da base alla pila di altezza massima
```

```
for i:=2 to n do
    H[i] := 1;
    for j:=1 to i-1 do
        if ( d[j] < d[i] && H[j]+1 > H[i] ) then
            H[i] := H[j] + 1;
        endif
    endfor
    if ( H[i] > H[ibase] ) then
        ibase := i;
    endif
endfor
print "Il numero massimo di dischi impilabili è " H[ibase];
// ora stampiamo i dischi della pila di altezza massima, partendo dalla base
int j := ibase;
do
    i := j;
    // i = indice disco (parte della soluzione ottima) che stiamo analizzando; j = indice
disco (anch'esso parte della soluzione ottima) che viene appoggiato immediatamente
sopra a i, e che quindi analizzeremo al prossimo passo
    print "Uso il disco " i;
    if ( H[i] > 1 ) then
        do
            j := j – 1;
        while ( d[j] ≥ d[i] || H[j] ≠ H[i] – 1 )
    endif
while (H[i] > 1);
return H[n];
```

Esercizio 7.14

Si supponga di avere n files in cui il file i occupa $F[i]$ MB. Supponiamo che tutte le dimensioni $F[i]$ siano intere (ossia, ciascun file occupa un multiplo intero di un MB). Vogliamo individuare, se esiste, un sottoinsieme di file la cui dimensione complessiva sia *esattamente pari a* 650MB. In caso tale sottoinsieme esista, vogliamo anche sapere gli indici dei files che vi appartengono.

Soluzione. Consideriamo una matrice $B[i, j]$ di booleani, per ogni i=1, ..., n e per ogni j=0, ..., 650, tali che $B[i, j]$ = *true* se e solo se esiste un sottoinsieme dei primi i files $\{1, ..., i\}$ la cui dimensione complessiva sia esattamente pari a j MB.
Iniziamo considerando i casi base:

- $B[i, 0]$ = *true*, per ogni i=1, ..., n (il sottoinsieme vuoto di file ha dimensione complessiva pari a zero);
- $B[1, F[1]]$ = *true*, in quanto il sottoinsieme di $\{1\}$ composto solo dal file numero 1 ha esattamente dimensione $F[1]$. (attenzione a verificare che $F[1] \leq 650$, altrimenti l'elemento $B[1, F[1]]$ non esisterebbe)

Per quanto riguarda il generico elemento $B[i, j]$, possiamo applicare lo stesso ragionamento che abbiamo applicato per il problema dello zaino. Specificamente, $B[i, j]$ = *true* se e solo se vale almeno una delle condizioni seguenti:

- $B[i\text{-}1, j]$ = true; infatti, in questo caso significa che esiste un sottoinsieme S dei primi i-1 files la cui dimensione complessiva è j. Lo stesso sottoinsieme S è anche (per definizione) un sottoinsieme dei primi i files, quindi $B[i, j]$ = *true*;
- $B[i\text{-}1, j\text{-}F[i]\ ]$ = *true*; in questo caso, esiste un sottoinsieme dei primi i-1 files la cui dimensione complessiva è pari a j-$F[i]$. Aggiungendo il file i-esimo, si ottiene un sottoinsieme di files in $\{1, ..., i\}$ la cui dimensione complessiva è $(j\text{-}F[i]) + F[i] = j$.

Questo significa che possiamo semplicemente scrivere

$$B[i\,,\,j] = B[i-1,\,j] \vee B[i-1,\,j-F[i]]$$

È importante prestare attenzione al fatto che l'espressione $B[i\text{-}1, j\text{-}F[i]\ ]$ può essere calcolata solo se $j \geq F[i]$, altrimenti il secondo indice della matrice diventa negativo.

In conclusione, esiste un sottoinsieme degli n files la cui dimensione è esattamente pari a 650MB se e solo se $B[n, 650]$ è *true*. Per sapere quali files fanno parte di tale sottoinsieme, usiamo una ulteriore matrice booleana $P[i, j]$, tale che $P[i, j]$ = *true* se e solo se il file i-esimo fa parte dell'eventuale sottoinsieme di {1, ..., i} la cui dimensione complessiva sia esattamente pari a j (sempre se tale sottoinsieme esiste). Le matrici B e P possono essere costruite con lo pseudocodice seguente:

```
algoritmo CopiaTutto( array F[1..n] di int )
   array B[1..n, 0..650] di bool;
   array P[1..n, 0..650] di bool;
   for j := 0 to 650 to
        if ( j == 0 ) then
             B[i, j ] := true;
             P[1, j ] := false;
        elseif ( j == F[1] ) then
             B[1, j ] := true;
             P[1, j ] := true;
        else
             B[1, j ] := false;
             P[1, j ] := false;
        endif
   endif
   for i:=2 to n do
        for j:=0 to 650 do
             if ( j ≥ F[i] && B[i - 1, j - F[i] ] == true ) then
                  B[i, j] := true;
                  P[i, j] := true;     // usiamo il file i-esimo
             else
                  B[i, j] := B[i-1, j];
                  P[i, j] := false;
             endif
        endfor
   endfor
   if ( B[n, 650] == true ) then
        print “Esiste una soluzione”
        // stampa della soluzione
        j := 650;
        i := n;
        while ( j > 0 ) do
             if ( P[i, j] == true ) then
                  print “Uso file “ i;
                  j := j – F[i];
             endif
             i := i-1;
        endwhile
   else
        print “Non esiste una soluzione”
   endif
```

Esercizio 7.15

Si consideri un insieme di n persone che devono salire su un ascensore. L'ascensore è in grado di sostenere un peso massimo pari a C kg, ma è sufficientemente ampio da ospitare un numero qualsiasi di persone. Siano $p[1], \ldots, p[n]$ i pesi in kg delle n persone. Il vettore dei pesi non è ordinato. Tutti i pesi sono interi.
Scrivere un algoritmo che, dati in input il valore di C e il vettore $p[1..n]$, restituisca il numero massimo di

persone che possono salire sull'ascensore contemporaneamente, senza superare il peso massimo complessivo di C kg. In altre parole, si chiede di determinare la cardinalità di un sottoinsieme massimale delle n persone il cui peso complessivo risulti minore o uguale a C. (*Suggerimento: si può usare la programmazione dinamica oppure un algoritmo greedy*). Analizzare il costo computazionale dell'algoritmo proposto

Soluzione. L'approccio basato sul paradigma greedy è il più semplice e in questo caso si può applicare per ottenere la soluzione ottima: facciamo entrare le persone in ascensore in ordine non decrescente di peso (prima la persona che pesa di meno, poi la successiva e così via). Ci fermiamo escludendo la prima persona che, entrando in ascensore, farebbe superare la portata massima.

```
algoritmo AscensoreGreedy( int C, array p[1..n] di int ) → int
   OrdinaCrescente(P);
   int i := 1;
   while( i≤n && p[i] ≤ C ) do
        C := C - p[i];
        i := i+1;
   endwhile
   return i-1;
```

Il costo dell'algoritmo MaxPersoneGreedy è dominato dal costo dell'operazione di ordinamento, ossia $O(n \log n)$ usando un algoritmo generico (è possibile ordinare in tempo lineare usando un algoritmo che sfrutta il fatto che i pesi sono interi).
L'approccio basato sulla programmazione dinamica si basa su una matrice i cui elementi $N[i, c]$ indicano il massimo numero di persone, scelte tra le prime i, che è possibile "stipare" in un ascensore avente portata massima di c Kg, i=1, ..., n, c=0, ..., C. Si noti il "massimo numero di persone", NON "massimo peso delle persone".
Consderiamo il caso i=1, in cui c'è solo una persona (la prima) che può salire. Si ha che $N[1, c] = 1$ se e solo se $c \geq p[1]$: il numero di personc che può salire è 1 se e solo se la capacità c dell'ascensore è maggiore o uguale al peso $p[1]$ della persona. Quindi:

$$N[1,c]=\begin{cases}1 & \text{se } c \geq p[1] \\ 0 & \text{altrimenti}\end{cases}$$

Per calcolare $N[i, c]$ nel caso generico, ragioniamo come segue. Se il peso della persona i-esima supera la capacità c, allora questa non potrà certamente salire, quindi $N[i, c]$ avrà lo stesso valore del caso in cui ci siano solo le prime i-1 persone, cioè $N[i\text{-}1, c]$.
Invece, se il peso della persona i-esima è minore o uguale della capacità c, allora abbiamo due possibilità:

1. Far salire la persona i-esima; in questo caso il numero massimo di persone sull'ascensore è $N[\text{i-}1, c\text{-}p[i]] + 1$;
2. NON far salire la persona i-esima; in questo caso il numero massimo di persone è $N[i\text{-}1, c]$.

Naturalmente, tra le due opzioni precedenti sceglieremo quella che ci consente di massimizzare il valore di $N[i, c]$. Quindi possiamo definire $N[i, c]$ come:

$$N[i,c]=\begin{cases}max\{N[i-1,c], N[i-1,c-P[i]]+1\} & \text{se } c \geq p[i] \\ N[i-1,c] & \text{altrimenti}\end{cases}$$

Dopo aver compilato la tabella di programmazione dinamica in tempo $\Theta(nC)$, il risultato si leggerà nella casella $N[n, C]$.

Esercizio 7.16

Disponete di n scatole $s_1, s_2, \ldots, s_n$ a forma di parallelepipedo; la scatola s_i è larga $x[i]$, alta $y[i]$ e profonda $z[i]$. Vogliamo cercare di inserire una dentro l'altra il maggior numero possibile di scatole, senza ruotarle. La scatola s_i può essere inserita in s_j se e solo se valgono entrambe queste condizioni:

1. $i < j$ *:* la scatola esterna deve avere un numero maggiore della scatola interna

2. $x[i] < x[j]$ e $y[i] < y[j]$ e $z[i] < z[j]$: la scatola esterna deve avere dimensioni strettamente maggiori della scatola interna

Scrivere un algoritmo efficiente, basato sulla programmazione dinamica, che dati in input il valore di n e i vettori $x[1..n]$, $y[1..n]$ e $z[1..n]$, restituisca il massimo numero di scatole che possono essere inserite l'una nell'altra. Analizzare il costo computazionale dell'algoritmo proposto.

Soluzione. Definiamo con $N[k]$ il massimo numero di scatole in $\{s_1, \dots s_k\}$ che possono essere inserite l'una nell'altra, secondo le regole indicate, usando in ogni caso la scatola s_k come scatola più esterna.
Osserviamo che $N[1] = 1$. Cerchiamo ora di definire una espressione per $N[k]$, sfruttando l'idea seguente. Sappiamo che dobbiamo utilizzare la scatola s_k come scatola più esterna; quindi di sicuro $N[k] \geq 1$ (in quanto una scatola, la k-esima, viene sempre utilizzata). Dobbiamo ora inserirci dentro il massimo numero possibile di scatole in $\{s_1, ..., s_{k-1}\}$, rispettando i vincoli richiesti dall'esercizio. Supponiamo che la scatola i-esima, per un certo $i < k$, possa essere inserita dentro s_k; questo può succedere se $x[i] < x[k]$, $y[i] < y[k]$ e $z[i] < z[k]$. Quindi, se inseriamo s_i dentro s_k otterremo $N[i]+1$ scatole correttamente inserite l'una nell'altra.
Quindi: tra tutte le scatole s_i che che possono essere inserite direttamente dentro s_k, ossia che soddisfano le condizioni:

- $i<k$, e
- $x[i] < x[k]$, $y[i] < y[k]$ e $z[i] < z[k]$

scegliamo quella per cui il valore $N[i] + 1$ risulta massimo possibile. Questo meccanismo ci consente di calcolare $N[k]$ per ogni k; ricordandoci che $N[k]$ è il massimo numero di scatole inseribili l'una nell'altra, usando s_k come contenitore più esterno, si ha che la soluzione al nostro problema consiste nello scegliere al termine il valore massimo di $N[k]$.

```
algoritmo MaxNumScatole( array x[1..n] di double, array y[1..n] di double, array z[1..n] di
double ) → int
   array N[1..n] di int; // N[k] = max num. di contenitori inscatolabili usando sk
   int Nmax; // valore massimo di N[1..n]
   // Inizializzazione
   N[1] := 1;
   Nmax := N[1];
   for k:=2 to n do
        N[k] := 1;
        for i:=1 to k-1 do
             if ( x[i] < x[k] && y[i] < y[k] && z[i] < z[k] && N[i]+1 > N[k]) then
                  N[k] := N[i]+1;
             endif
        endfor
        if ( N[k] > Nmax ) then
             Nmax := N[k];
        endif
   endfor
   return Nmax;
```

Esercizio 7.17

Il Ministero per l'Istruzione e l'Università della Ruritania (MIUR) dispone di un budget di B euro (intero non negativo) che vuole usare per finanziare progetti di ricerca. Sono stati presentati n progetti di ricerca, numerati da 1 a n. Il progetto i-esimo richiede un finanziamento di $f[i]$ euro (intero non negativo). Poiché il budget B non è sufficiente per finanziare tutti i progetti, il MIUR ha deciso di far valutare ciascun progetto da un gruppo di esperti, che ha assegnato a ciascun progetto i un punteggio $p[i] \geq 0$.
Il MIUR intende finanziare il sottoinsieme di progetti tali che (i) la somma dei budget dei progetti finanziati sia minore o uguale a B, e (ii) il punteggio complessivo del sottoinsieme di progetti finanziati sia il massimo tra tutti i sottoinsiemi che soddisfano la condizione (i).

1. Scrivere un algoritmo che, dati in input il valore del budget B, le richieste di finanziamento $f[1..n]$ e i punteggi $p[1..n]$, stampa l'elenco dei progetti finanziabili in base ai requisiti descritti sopra.
2. Calcolare il costo computazionale dell'algoritmo proposto.

Esercizio 7.18

Sia $V[1..n]$ un vettore composto da n numeri reali arbitrari (possono essere presenti valori positivi e negativi). Per ogni i, j con $1 \le i \le j \le n$, sia $S(i, j)$ la somma dei valori degli elementi del sottovettore $V[i..j]$; si noti che $S(i, i) = V[i]$ per ogni i. Scrivere un algoritmo in grado di calcolare i valori $S(i, j)$, per ogni i, j, in tempo $\Theta(n^2)$. Modificare l'algoritmo per risolvere il problema di determinare il sottovettore di somma massima in tempo $\Theta(n^2)$; ricordiamo che tale problema può essere risolto in tempo $\Theta(n)$ usando la programmazione dinamica. *(Suggerimento: usare una matrice per memorizzare i valori S(i, j); è possibile riempire la matrice in maniera opportuna per ottenere il costo asintotico richiesto).*

Soluzione. Sfruttiamo un approccio ispirato alla programmazione dinamica per calcolare i costi nel modo seguente. Memorizziamo i valori $S(i, j)$ in una matrice quadrata $n \times n$. Si noti come solo la parte triangolare superiore della matrice verrà utilizzata, dato che i valori che ci interessano sono tutti e soli quelli per cui $i \le j$. Riempiamo la matrice una riga alla volta. Per ogni riga, il primo valore $S(i, i)$ è facilmente calcolato, dato che $S(i, i) = V[i]$ per definizione. Notiamo che una volta calcolato $S(i, j)$, possiamo immediatamente calcolare l'elemento $S(i, j+1) = S(i, j) + V[j+1]$. Quindi ogni elemento della matrice può essere calcolato in tempo O(1); poiché ci sono $\Theta(n^2)$ elementi, il costo computazionale dell'algoritmo sarà $\Theta(n^2)$.

```
algoritmo CALCOLAS( array V[1..n] di double) → array [1..n, 1..n] di double
    array S[1..n, 1..n] di double;
    for i:=1 to n do
        S[i, i] := V[i];
        for j:=i+1 to n do
            S[i, j] := S[i, j - 1] + V[j]
        endfor
    endfor
    return S
```

Per risolvere il problema del calcolo del sottovettore di somma massima, possiamo estendere l'algoritmo precedente come segue:

```
algoritmo SOTTOVETTORESMAX( array V[1..n] di double )
    int imax := 1; // Indice primo elemento del vettore di somma massima
    int jmax := 1; // Indice ultimo elemento del vettore di somma massima
    double Smax := V[1]; // Somma degli elementi del vettore di somma massima
    array S[1..n, 1..n] di double;
    for i:=1 to n do
        S[i, i] := V[i];
        for j:=i+1 to n do
            S[i, j] := S[i, j - 1] + V[j]
        endfor
        // Aggiorniamo il massimo per ogni riga
        for j:=i to n do
            if ( S[i, j] > Smax ) then
                Smax := S[i, j];
                imax := i;
                jmax := j;
            endif
        endfor
    endfor
    print Smax, imax, jmax
```

Esercizio 7.19

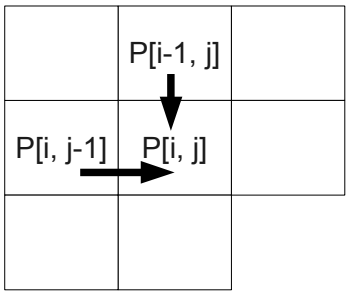
Si consideri una scacchiera quadrata rappresentata da una matrice $M[1..n, 1..n]$. Scopo del gioco è spostare una pedina dalla casella di partenza in alto a sinistra di coordinate (1, 1) alla casella di arrivo in basso a destra di coordinate (n, n). Ad ogni mossa la pedina può essere spostata di una posizione verso il basso

oppure verso destra (senza uscire dai bordi della scacchiera). Quindi, se la pedina si trova in (i, j) potrà essere spostata in $(i+1, j)$ oppure $(i, j+1)$, se possibile. Ogni casella $M[i, j]$ contiene un numero reale; man mano che la pedina si muove, il giocatore accumula il punteggio segnato sulle caselle attraversate, incluse quelle di partenza e di arrivo.
Scrivere un algoritmo efficiente che, data in input la matrice $M[1..n, 1..n]$ restituisce il massimo punteggio che è possibile ottenere spostando la pedina dalla posizione iniziale a quella finale con le regole di cui sopra. Ad esempio, nel caso seguente:

1	3	4	-1
3	-2	-1	5
-5	9	-3	1
4	5	2	-2

l'algoritmo deve restituire 16 (le celle evidenziate indicano il percorso da far fare alla pedina per ottenere il massimo punteggio che è appunto 16).

Soluzione. Possiamo risolvere il problema utilizzando la programmazione dinamica. Definiamo una matrice $P[1..n, 1..n]$ di numeri reali, tale che $P[i, j]$ rappresenta il massimo punteggio che è possibile ottenere spostando la pedina dalla cella (1, 1) fino alla cella (i, j). Una volta calcolati tutti i valori degli elementi di P, la risposta al nostro problema sarà il valore contenuto in $P[n, n]$.
Sappiamo che $P[1, 1] = M[1, 1]$. Osserviamo che le celle della prima riga della matrice possono essere raggiunte esclusivamente spostando la pedina a destra, a partire dalla posizione iniziale. Quindi $P[1, j] = P[1, j-1] + M[1, j]$, per ogni $j=2, \ldots n$. Analogamente, le celle della prima colonna possono essere raggiunte esclusivamente spostando la pedina in basso, quindi $P[i, 1] = P[i-1, 1] + M[i, 1]$ per ogni $i=2, \ldots n$.
Supponiamo ora di voler calcolare P$[i, j]$ per un certo $i>1$, $j>1$. La cella di coordinate (i, j) può essere raggiunta a partire da $(i-1, j)$, spostando la pedina verso il basso di una posizione; in tal caso avremmo che il punteggio ottenuto dopo lo spostamento è $P[i-1, j] + M[i, j]$. Alternativamente, possiamo raggiungere la cella (i, j) a partire da $(i, j-1)$ spostando la pedina verso destra; in tal caso il punteggio ottenuto è $P[i, j-1] + M[i, j]$. Si faccia riferimento alla figura seguente



Chiaramente sceglieremo la mossa che ci farà ottenere il punteggio massimo, quindi possiamo scrivere, per ogni $i=2, ..., n$ e per ogni $j=2, ... n$:

$$P[i, j] = max\{P[i-1, j], P[i, j-1]\} + M[i, j]$$

L'algoritmo seguente risolve il problema; in più, anche se non richiesto dal testo, l'algoritmo stampa il percorso che consente di ottenere il punteggio massimo. Per fare questo usiamo una seconda matrice ausiliaria $H[1..n, 1..n]$, tale che $H[i, j] = 1$ se per raggiungere la cella (i, j) ottenendo punteggio massimo abbiamo spostato a destra da $(i, j-1)$, oppure $H[i, j] = -1$ se per raggiungere la cella (i, j) abbiamo spostato in basso da $(i-1, j)$.

```
algoritmo GiocoScacchiera( array M[1..n, 1..n] di double ) → double
  array P[1..n, 1..n] di double;
```

```
array H[1..n, 1..n] di int;
int i, j;
// inizializzazione
P[1, 1] := M[1, 1];
H[1, 1] := 0;                  // non proveniamo da nessuna direzione
for j := 2 to n do             // prima riga
    P[ 1, j ] := P[ 1, j -1 ] + M[ 1, j ];
    H[ 1, j ] := 1;            // provengo da sx
endfor
for i:= 2 to n do              // prima colonna
    P[ i, 1 ] := P[ i - 1, 1 ] + M[ i, 1 ];
    H[ i, 1 ] := -1;           // provengo dall'alto
endfor
// Calcolo della soluzione
for i:= 2 to n do
    for j := 2 to n do
        if ( P[ i - 1, j ] ≥ P[ i, j – 1 ] ) then
            P[ i, j ] := P[ i - 1, j ] + M[ i, j ];
            H[ i, j ] := -1;   // provengo dall'alto
        else
            P[ i, j ] := P[ i, j – 1 ] + M[ i, j ];
            H[ i, j ] := 1;    // provengo da sx
        endif
    endfor
endfor
// Stampa la sequenza di caselle visitate, a partire dall'ultima (non richiesto
dall'esercizio); si noti che NON viene stampata la casella iniziale (1, 1).
i:= n;
j:= n;
while ( i > 1 && j > 1 ) do
    print "(", i, ",", j, ")";
    if ( H[ i, j ] == 1 ) then
        j := j - 1;
    else
        i := i - 1;
    endif
endwhile
// restituisco la soluzione (infatti l'algoritmo restituisce un numero reale)
return P[n, n];
```

Il costo dell'algoritmo è dominato dal ciclo per il calcolo degli elementi della matrice P, ed è pari a $\Theta(n^2)$.

Esercizio 7.20

[Problema basato su 10.5 del libro di testo] La CINA (Compagnia Italiana per il Noleggio di Automobili) dispone di k automobili, tutte disponibili per n giorni. Una stessa automobile può avere costo di affitto diverso in giorni diversi: indicheremo con $c[a, g]$ il costo per il noleggio dell'automobile a nel giorno g, dove $1 \le a \le k$ e $1 \le g \le n$. Un cliente si rivolge alla CINA per procurarsi un mezzo di locomozione per l'intero periodo (giorni da 1 a n, estremi inclusi). Per risparmiare e sfruttare i prezzi di noleggio migliori in ogni giornata il cliente è disposto a cambiare macchina da un giorno all'altro. Per ottenere un cambio, però, è costretto dalla CINA a pagare una penale P , che si aggiunge al costo di noleggio. Il costo di una sequenza di noleggio è quindi dato dalla somma dei costi di noleggio e delle penali pagate per i cambi. Dati k, n, i costi $c[a, g]$ e l'importo della penale P:

1. Calcolare il numero di possibili sequenze di noleggio diverse.
2. Scrivere un algoritmo che calcoli la sequenza di noleggio di costo minimo, nell'ipotesi in cui la penale P sia pari a zero.
3. Proporre un algoritmo efficiente che calcoli la spesa minima per il cliente tra tutte le possibili sequenze di noleggio, assumendo una penale $P > 0$;
4. Modificare l'algoritmo proposto al punto precedente in modo da avere in output la sequenza di

noleggio di costo minimo.

Soluzione . Il numero delle possibili sequenze di noleggio è k^n . Quindi un eventuale algoritmo risolutivo che operi di forza bruta non sarebbe praticabile.
Nel caso in cui la penale sia zero, è possibile individuare facilmente la sequenza di noleggio di costo minimo: infatti, per ogni giorno i=1, …, n basta scegliere la vettura a tale che $c[a, i]$ sia minimo. In altre parole, per ogni giorno si sceglie di noleggiare la vettura più conveniente in quel giorno.
Nel caso generale in cui la penale sia strettamente positiva, possiamo sfruttare la programmazione dinamica per individuare la sequenza di noleggio di costo minimo. Definiamo Cost[i, j] il costo minimo tra tutte le sequenze di noleggio durante i primi j giorni, tali che al giorno j-esimo usino l'auto i, per ogni j=1, …, n e i=1, …, k.. Considerando i noleggi solo per il primo giorno (j = 1), possiamo scrivere:

$$Cost[i,1]=c[i,1]$$

Nel caso di sequenze di noleggio che durano fino al giorno j > 1, l'acquirente ha a disposizione le seguenti alternative:
Se il giorno j-1 il cliente usa la stessa auto i, senza effettuare alcun cambio, ottiene un costo di noleggio pari a

$$Cost[i,j]=Cost[i,j-1]+c[i,j]$$

Se il giorno j-1 il cliente usa un'auto $m \neq i$, ottiene un costo di noleggio pari a

$$Cost[i,j]=Cost[m,j-1]+c[i,j]+P$$

Il costo Cost[i, j] della sequenza ottima di noleggio sarà quindi il minimo tra le alternative descritte sopra, ossia

$$Cost[i,j]=min_{m\neq i}\{Cost[i,j-1]+c[i,j], Cost[m,j-1]+c[i,j]+p\}$$

Il costo minimo tra tutte le sequenze di noleggio per tutti i n giorni, è dato dal valore minimo dell'ultima colonna della matrice Cost, ossia

$$min_{i=1,\ldots k}\{Cost[i,n]\}$$

Esercizio 7.21

Un bibliotecario deve disporre n libri su scaffali di larghezza L. I libri hanno larghezze x[1..n] e altezze y[1..n]; essi vanno disposti su un numero potenzialmente illimitato di scaffali senza essere riordinati, e ovviamente senza eccedere lo spazio orizzontale disponibile. Tuttavia, non è necessario riempire completamente gli scaffali, dato che è possibile lasciare quanto spazio si vuole al termine di ciascuno di essi. L'altezza di ciascuno scaffale è pari alla massima altezza dei libri che contiene; l'altezza complessiva dell'intera scaffalatura è pari alla somma delle altezze dei singoli scaffali. Tutte le misure sono numeri reali positivi. Si assuma che nessun libro abbia larghezza superiore a L.
Scrivere un algoritmo che, dati in input le dimensioni x[1..n] e y[1..n], nonché la larghezza L degli scaffali, restituisce la minima altezza complessiva.

Soluzione. Questo problema è una variante un po' più complessa dell'Esercizio 6.4; in questo caso la soluzione ottima può essere individuata applicando la programmazione dinamica.
Iniziamo col definire i sottoproblemi $P(0)$, $P(1)$, … $P(n)$ da risolvere: $P(i)$ consiste nel disporre i primi libri sugli scaffali, secondo le regole date, in modo da minimizzarne l'altezza complessiva (in $P(0)$ non è disponibile alcun libro); si noti che nella soluzione del problema $P(i)$, il libro i-esimo occuperà sempre la posizione più a destra dell'ultimo scaffale (supponendo che gli scaffali vengano riempiti da sinistra verso

destra). Il problema complessivo corrisponde col sottoproblema $P(n)$.
Definiamo $H[i]$ la soluzione ottima del problema $P(i)$, per ogni i=0, 1, … n; in altre parole, $H[i]$ è l'altezza minima complessiva degli scaffali che sono necessari per ospitare i primi i libri.
Il problema P(0) è il più semplice da risolvere, in quanto non vi sono libri a disposizione. Quindi si avrà che l'altezza della scaffalatura in tal caso sarà pari a zero: $H[0] = 0$.
Vediamo ora come calcolare la soluzione $H[i]$ del problema $P(i)$, per ogni i>0. Osserviamo che esistono diversi modi per disporre i libri sull'ultimo scaffale. La più semplice consiste nel posizionare il libro i-esimo da solo; in tal caso l'altezza complessiva sarebbe pari a $y[i] + H[i\text{-}1]$. Un'altra possibilità consiste nel posizionare i libri i e i-1 sull'ultimo scaffale, e gli altri nei precedenti. In tal caso l'altezza complessiva sarebbe pari a $\max\{y[i], y[i\text{-}1]\} + H[i\text{-}2]$. In generale, posizionando gli ultimi i-k+1 libri (quelli numerati da k a i) sull'ultimo scaffale, l'altezza *Hult* dell'ultimo scaffale è pari a

$$Hult = max\, y[k], y[+1], \ldots, y[i]$$

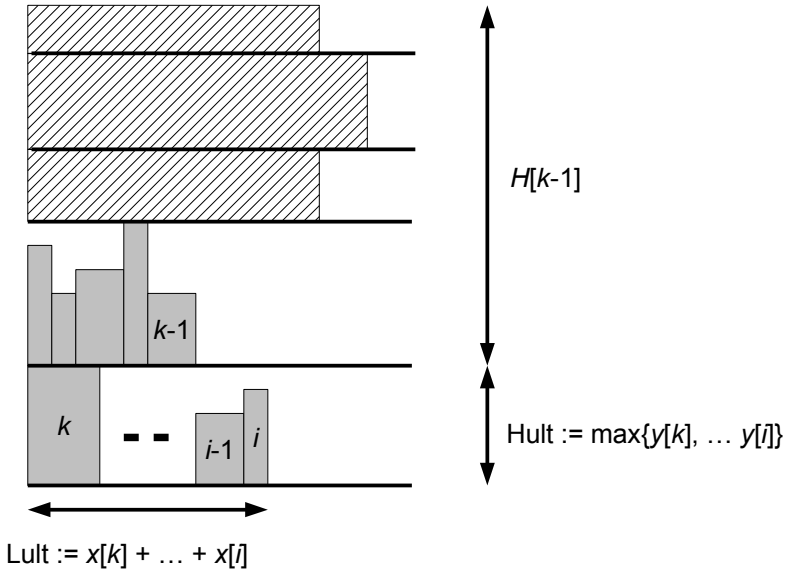
la sua larghezza *Lult* è pari a

$$Lult = x[k] + x[k+1] + \ldots x[i]$$

e l'altezza complessiva di tutti gli scaffali diventa

$$Hult + H[k-1] = max\{y[k], \ldots y[i]\} + H[k-1]$$

(si faccia riferimento alla figura seguente)



Quindi, per calcolare $H[i]$ è necessario individuare il valore di k, tra tutti qulli per cuui $Lult \le L$, che minimizza $Hult + H[k\text{-}1]$. Questo può essere realizzato dall'algoritmo seguente

```
algoritmo BIBLIOTECARIO( array x[1..n] di double, array y[1..n] di double, double L ) → double
   array H[0..n] di double;
   H[0] := 0;
   for i :=1 to n do
        double Hmin := +∞;
        double Lult := 0;              // spazio occupato dai libri sull'ultimo scaffale
        double Hult := 0;              // altezza ultimo scaffale
```

```
        int k := i;
        while ( k > 0 && Lult + x[k] ≤ L ) do
            // Posiziono il libro k sull'ultimo scaffale.
            // Per prima cosa aggiorno lo spazio orizzontale occupato
            Lult := Lult + x[k];
            // Poi controllo se l'altezza dello scaffale aumenta
            if ( y[k] > Hult ) then
                Hult := y[k];
            endif
            // A questo punto controllo se l'altezza complessiva di tutti gli scaffali diminuisce
            if ( Hult + H[k-1] < Hmin ) then
                Hmin := Hult + H[k-1];
            endif
            k := k – 1;
        endwhile
        H[i] := Hmin;
    endfor
    return H[n];
```

Capitolo 8: Grafi

Esercizio 8.1

Si consideri un grafo non orientato $G=(V, E)$ in cui a ciascun nodo $v \in V$ è associato un peso reale $w(v)$ (che puo' essere positivo o negativo). Un cammino $<v_1, v_2, ... v_k>$ si dice *monotono* se $w(v_1) < w(v_2) < ... < w(v_k)$. In altre parole in un cammino monotono i pesi dei nodi attraversati devono essere in ordine strettamente crescente.

1. Dimostrare che se $<v_1, v_2, ... v_k>$ è un cammino monotono, allora è aciclico
2. Descrivere un algoritmo efficiente che, dato in input un grafo non orientato $G=(V, E)$ con nodi pesati, e due nodi $s, d \in V$, restituisce true se e solo se esiste un cammino monotono che inizia dalla sorgente s e termina nella destinazione d. L'algoritmo deve anche stampare i nodi che compongono tale cammino (i nodi possono essere stampati nell'ordine $v_1, v_2, \ldots v_k$ oppure nell'ordine inverso $v_k, v_{k-1}, \ldots v_1$)
3. Determinare il costo computazionale dell'algoritmo descritto al punto 2, motivando la risposta

Soluzione. Supponiamo per assurdo che il cammino monotono $<v_1, v_2, ... v_k>$ contenga un ciclo, cioè che esistano interi $1 \leq i < j \leq k$ tali che il cammino possa essere scritto come $<v_1, \ldots v_i, \ldots v_j, \ldots v_k>$, e $<v_i, \ldots v_j>$ sia un ciclo (ossia, $v_i = v_j$). Poiché il cammino iniziale è monotono, lo sarà anche il sottocammino $<v_i, \ldots v_j>$. Quindi deve essere:

$$w(v_i) < w(v_{i+1}) < \ldots < w(v_j) = w(v_i)$$

da cui si conclude $w(v_i) < w(v_i)$ il che è assurdo.

Per quanto riguarda l'algoritmo che identifica se esiste un cammino aciclico che parte da s e termina in d, è possibile utilizzare una banale variazione dell'algoritmo di visita in profondità come segue.

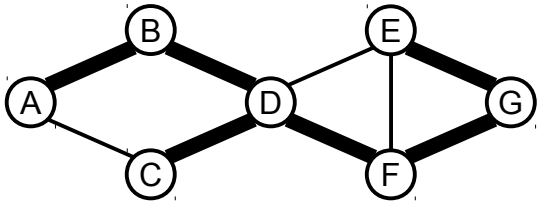
```
algoritmo CamminoMonotono( grafo G=(V, E), nodo s, nodo d ) → bool
   foreach v in V do
        v.mark := false;
   endfor
   return CamminoMonotonoRic(G, s, d);

algoritmo CamminoMonotonoRic( grafo G=(V, E), nodo s, nodo d ) → bool
   s.mark := true;
   if ( s == d ) then
        print d;
        return true;
   else
        foreach nodo x adiacente a s do
             if ( x.mark == false && w(s) < w(x) ) then
                  if ( CamminoMonotonoRic(G, x, d) ) then
                       print s;
                       return true;
                  endif
```

```
            endif
        endfor
        return false;
    endif
```

Il costo computazionale è esattamente quello di una normale visita in profondità, ossia O(n+m).

Esercizio 8.2

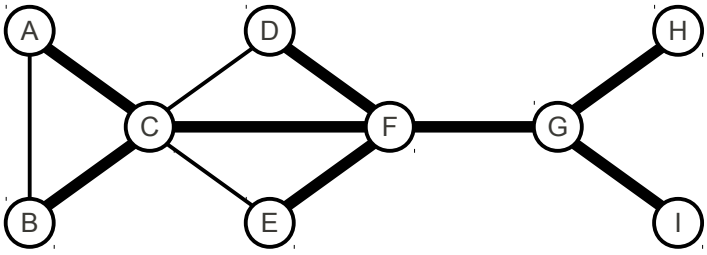
Considerare il grafo non orientato seguente, in cui gli archi in grassetto indicano uno spanning tree:



1. Lo spanning tree può essere ottenuto mediante una visita **in profondità** del grafo? In caso affermativo specificare il nodo di inizio della visita, e rappresentare il grafo mediante liste di adiacenza in modo tale che l'ordine in cui compaiono gli elementi nelle liste consenta all'algoritmo DFS di produrre esattamente lo spanning tree mostrato.
2. Lo spanning tree può essere ottenuto mediante una visita **in ampiezza** del grafo? In caso affermativo specificare il nodo di inizio della visita, e rappresentare il grafo mediante liste di adiacenza in modo tale che l'ordine in cui compaiono gli elementi nelle liste consenta all'algoritmo BFS di produrre esattamente lo spanning tree mostrato.

Esercizio 8.3

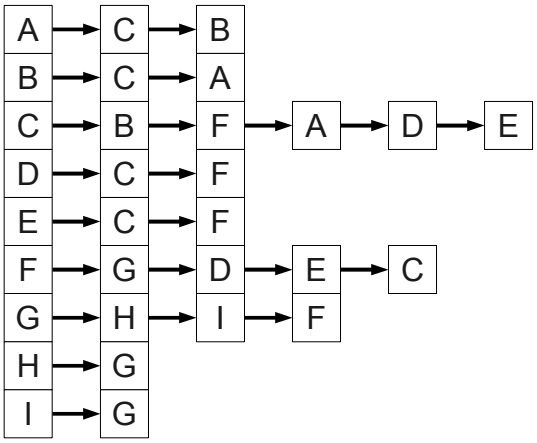
Considerare il seguente grafo non orientato:



1. Gli archi in grassetto possono rappresentare un albero di visita ottenuto mediante una visita **in profondità** del grafo? In caso affermativo specificare il nodo di inizio della visita, e rappresentare il grafo mediante liste di adiacenza in modo tale che l'ordine in cui compaiono gli elementi nelle liste consenta all'algoritmo DFS di produrre esattamente l'albero mostrato.
2. Gli archi in grassetto possono rappresentare un albero di visita ottenuto mediante una visita **in ampiezza** del grafo? In caso affermativo specificare il nodo di inizio della visita, e rappresentare il grafo mediante liste di adiacenza in modo tale che l'ordine in cui compaiono gli elementi nelle liste consenta all'algoritmo BFS di produrre esattamente l'albero mostrato.

Soluzione. La risposta è affermativa in entrambi i casi. Per la visita in profondità si può ad esempio partire da A oppure B. Per la visita in ampiezza si può partire ad esempio da F.
In entrambi i casi (al di là del nodo di partenza, che è differente per la visita DFS e BFS) si può ottenere l'albero di mostrato nel testo utilizzando, ad esempio, la seguente rappresentazione con liste di adiacenza:

Naturalmente altre soluzioni corrette erano possibili.



Esercizio 8.4

Si consideri un grafo orientato $G=(V, E)$ con n nodi, numerati da 1 a n. Il grafo è rappresentato mediante una matrice di adiacenza.

1. Scrivere un algoritmo per calcolare l'indice di un nodo avente grado uscente massimo (ossia l'indice di un nodo avente massimo numero di archi uscenti).
2. Analizzare il costo computazionale dell'algoritmo proposto.

Soluzione. La soluzione consiste nel cercare l'indice della riga avente il maggior numero di valori 1. In altre parole:

```
algoritmo TrovaNodo( array M[1..n, 1..n] di int ) → int
    int imax := 1; // nodo con grado uscente max
    int gmax := 0; // valore del grado max
    for i:=1 to n do
        int g := 0; // grado uscente del nodo corrente
        for j:=1 to n do
            g := g + M[i, j];
        endfor
        if ( g > gmax ) then
            gmax := g;
            imax := i;
        endif
    endfor
    return imax;
```

Si noti che l'esercizio chiedeva di restituire l'indice del nodo di grado massimo, non il valore del grado massimo. Quindi restituire il valore di gmax sarebbe stato sbagliato. Il costo asintotico dell'algoritmo è $\Theta(n^2)$.

Esercizio 8.5

È dato un grafo non orientato $G=(V, E)$, non necessariamente connesso, e un vertice $s \in V$. Scrivere un algoritmo che restituisca il numero di nodi che fanno parte della componente connessa di cui fa parte s

Soluzione. È sufficiente usare un algoritmo di visita (in ampiezza o profondità, a scelta) con una minima modifica. Vediamo la soluzione che usa la visita in ampiezza

```
algoritmo ContaCC( grafo G=(V, E), nodo s ) → int
    foreach v in V do
```

```
        v.mark := false;
    endfor
    int size := 1;
    QUEUE Q;
    s.mark := true;
    Q.ENQUEUE(s);
    while ( ! Q.ISEMPTY() ) do
        v := Q.DEQUEUE();
        foreach w adiacente a v do
            if ( w.mark == false ) then
                w.mark := true;
                size := size+1;
                Q.ENQUEUE(w);
            endif
        endfor
    endwhile
    return size;
```

Esercizio 8.6

Si consideri una scacchiera $M[1..n,1..n]$ di $n \times n$ elementi, con $n>1$. Ogni cella della scacchiera può contenere il valore 0 oppure 1. Se $M[i, j] = 0$, allora la cella (i, j) è libera, altrimenti è occupata da un ostacolo. Un giocatore viene posto inizialmente nella casella (1, 1), che si assume essere sempre libera. Ad ogni passo, il giocatore può muoversi in una delle caselle libere adiacenti. Specificamente, è possibile spostarsi dalla cella $M[i, j]$ ad una delle celle libere tra $M[i-1, j]$, $M[i+1, j]$, $M[i, j-1]$ e $M[i, j+1]$. Per semplicità, supponiamo che $M[i, j]$ sia definita per qualsiasi valore di i e j, e $M[i, j] = 1$ se almeno uno degli indici non è compreso tra 1 e n; quindi ad esempio, $M[0, 1] = M[1, 0] = 1$.

1. Descrivere un algoritmo efficiente in grado di determinare il numero minimo di passi necessari per spostarsi dalla cella (1, 1) alla cella (n, n), sempre che ciò sia possibile.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Questo problema si può risolvere efficientemente applicando l'algoritmo di visita in ampiezza (BFS) di un grafo non orientato. Non è necessario costruire esplicitamente il grafo da esplorare. Ciascuna cella della matrice è un nodo del grafo da esplorare; Esiste un arco tra ogni coppia di nodi adiacenti tali che entrambi i nodi corrispondano a caselle libere (cioè non occupate).
Similmente all'algoritmo BFS, utilizziamo una coda per mantenere l'elenco dei nodi ancora da esplorare. In questo caso la coda contiene coppie di interi, che rappresentano posizioni di caselle nella scacchiera. In più, utilizziamo la matrice accessoria $D[i, j]$ che contiene la minima distanza della cella (i, j) dalla cella (1, 1). Inizialmente tutte le distanze sono settate a +infinito. Ogni volta che una casella viene visitata per la prima volta, la sua distanza viene settata pari alla distanza della cella di provenienza, più uno. Per la proprietà della visita BFS questa distanza è garantita essere la minima distanza dalla sorgente della visita (ossia dalla cella (1, 1)).

```
algoritmo DISTANZAMINIMASCACCHIERA( array M[1..n, 1..n] di int ) → int
    array D[1..n, 1..n] di int;
    for i:=1 to n do
        for j:=1 to n do
            D[i, j] := +inf; // setta tutte le distanze a +inf
        endfor
    endfor
    Queue Q;
    D[1, 1] := 0;
    Q.ENQUEUE( (1, 1) ); // inserisci alla fine della coda
    while ( ! Q.ISEMPTY() ) do
        (i, j) := Q.dequeue(); // estrai dall'inizio della coda
        if (i == n && j == n) then
            return D[i, j]; // restituisce lunghezza cammino minimo
```

```
        else
            dist := D[i, j];
            if ( M[i-1, j] == 0 && D[i-1, j] == +inf ) then
                D[i-1, j] := dist+1;
                Q.ENQUEUE( (i-1, j) );
            endif
            if ( M[i+1, j] == 0 && D[i+1, j] == +inf ) then
                Q[i+1, j] := dist+1;
                Q.ENQUEUE( (i+1, j) );
            endif
            if ( M[i, j-1] == 0 && D[i, j-1] == +inf ) then
                D[i, j-1] := dist+1;
                Q.ENQUEUE( (i, j-1) );
            endif
            if ( M[i, j+1] == 0 && D[i, j+1] == +inf ) then
                D[i, j+1] := dist+1;
                Q.ENQUEUE( (i, j+1) );
            endif
        endif
    endwhile
    return +inf; // non esiste cammino
```

Per calcolare il costo dell'algoritmo osserviamo che la fase di inizializzazione della matrice $D[1..n, 1..n]$ ha costo $\Theta(n^2)$. Il calcolo del cammino viene effettuato sostanzialmente mediante una visita in ampiezza di un grafo composto da n^2 nodi (uno per ogni cella della matrice) e $2n(n\text{-}1)$ archi (*perché?*). La fase di visita ha quindi costo $O(n^2 + 2n(n\text{-}1)) = O(n^2)$. Pertanto il costo dell'algoritmo è $\Theta(n^2)$, essendo predominante la fase di inizializzazione.

Esercizio 8.7

Consideriamo un grafo orientato $G=(V, E)$ i cui archi abbiano pesi non negativi. Denotiamo con $w(u, v)$ il peso dell'arco orientato (u, v). Ricordiamo che l'algoritmo di Dijkstra per il calcolo dei cammini minimi da una singola sorgente $s \in V$ ha la seguente struttura generica:

```
algoritmo DIJKSTRAGENERICO( grafo G=(V, E), nodo s ) → albero
    inizializza D tale che Dsv = +∞ per v≠s, e Dss = 0
    T := albero formato dal solo vertice s
    while ( T ha meno di n nodi ) do
        trova l'arco (u,v) incidente su T con Dsu + w(u,v) minimo
        Dsv := Dsu + w(u,v);
        rendi u padre di v in T
    end while
    return T;
```

1. Scrivere una versione dell'algoritmo di Dijkstra che non faccia uso di una coda di priorità ma di una semplice lista, in modo che ad ogni passo esamini sistematicamente gli archi incidenti per individuare quello che minimizza la distanza.
2. Determinare il costo computazionale della variante dell'algoritmo di Dijkstra descritta al punto 1. Specificare quale struttura dati viene usata per rappresentare il grafo.

Soluzione. Supponiamo che, per ogni vertice s, la distanza tra la sorgente e s sia indicata con l'attributo $s.d$; possiamo quindi scrivere la variante dell'algoritmo di Dijkstra come segue:

```
algoritmo DIJKSTRACONLISTE( grafo G=(V, E), nodo s ) → albero
    foreach v in V do
        v.d := +∞
    endfor
    s.d := 0; // il nodo sorgente ha distanza zero da se stesso
```

```
T := albero formato dal solo vertice s
Lista L;
L.INSERT(s); // inserisci s in L
while ( ! L.ISEMPTY() ) do
    sia u il nodo di L con minimo valore di u.d // costo: O(n) nel caso peggiore
    rimuovi u da L // costo: O(1) se L è una lista doppiamente concatenata
    foreach (u, v)∈ E do
        if ( v.d == +∞ ) then
            v.d = u.d + w(u,v);
            rendi u padre di v in T;
            L.INSERT(u); // u non era nella lista, inseriscilo. Costo O(1)
        elseif ( u.d + w(u,v) < v.d ) then
            v.d := u.d + w(u,v);
            rendi u nuovo padre di v in T;
            // u era già nella lista, non va reinserito
        endif
    endfor
endwhile
return T;
```

Si noti che l'algoritmo DIJKSTRACONLISTE è leggermente più semplice da descrivere rispetto all'algoritmo di Dijkstra implementato con code di priorità visto a lezione. Infatti nel caso in cui $u.d+w(u, v) < v.d$, cioè nel caso in cui abbiamo scoperto un cammino più breve tra s e v che passa attraverso il nodo u, il nodo v non va reinserito nella struttura dati (nel nostro caso la lista), perché sicuramente vi è già contenuto.
Il costo di DIJKSTRACONLISTE è $O(n^2)$, essendo n il numero di nodi del grafo ($n=|V|$). Notiamo infatti quanto segue:

1. Il ciclo while viene eseguito al più n volte. Questo perché ad ogni iterazione viene estratto un nodo dalla lista, e una volta estratto un nodo questo non viene piú reinserito (questo è lo stesso ragionamento che abbiamo fatto per calcolare il costo computazionale dell'algoritmo di Dijkstra implementato tramite coda di priorità);
2. La ricerca del nodo u tale che $u.d$ sia minimo ha costo $O(n)$. Infatti la lista L conterrà al più n nodi (in quanto il grafo ha n nodi)
3. Il corpo del ciclo for viene eseguito $O(n)$ volte, in quanto un nodo può essere incidente al più a $n-1=O(n)$ altri nodi.

Esercizio 8.8

Un social network come Facebook può essere rappresentato mediante un grafo non orientato $G=(V, E)$, in cui ogni nodo rappresenta un utente, ed esiste un arco tra i nodi u e v se e solo se gli utenti rappresentati da u e v sono "amici"; assumiamo che se u è "amico" di v, allora anche v è "amico" di u (quindi G è un grafo non orientato). Nota: il grafo G potrebbe non risultare connesso.

1. Scrivere un algoritmo efficiente che, dato in input un grafo non orientato $G=(V, E)$ e un nodo $v \in V$, restituisce il numero di "amici di amici" di v, ossia restituisce il numero di nodi che si trovano a distanza 2 da v. Non è consentito usare variabili globali.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Questo problema si può risolvere utilizzando una versione adattata della visita in ampiezza (BFS). È anche possibile realizzare una soluzione che non fa uso esplicito di una coda, come la seguente:

```
algoritmo CONTAAMICI2( grafo G=(V,E), nodo v ) → int
  int c := 0; // numero di vicini di secondo livello
  foreach x in V do
      x.mark := false;
  endfor
  v.mark := true;
  for each u adiacente a v do // esplora vicini di v
```

```
        u.mark := true;
    endfor
    foreach u adiacente a v do // riconsidera nuovamente I vicini di v
        foreach w adiacente a u do // considera I vicini dei vicini
            if ( w.mark == false ) then
                w.mark := true;
                c := c+1; // abbiamo trovato un nuovo vicino di livello 2
            endif
        endfor
    endfor
    return c;
```

Se il grafo è rappresentato mediante liste di adiacenza, il costo nel caso pessimo è O(n+m), essendo n il numero di nodi e m il numero di archi.

Esercizio 8.9

Si consideri un grafo non orientato G=(V, E), non necessariamente connesso, in cui tutti gli archi hanno lo stesso peso positivo $\alpha > 0$.

1. Scrivere un algoritmo efficiente che, dato in input il grafo G=(V, E) di cui sopra e due nodi u e v, calcola la lunghezza del cammino minimo che collega u e v. Se i nodi non sono connessi, la distanza è infinita. Non è consentito usare variabili globali.
2. Analizzare il costo computazionale dell'algoritmo proposto al punto 1.

Soluzione. È ovviamente possibile usare l'algoritmo di Dijkstra per il calcolo dell'albero dei cammini minimi. Tuttavia, una tale soluzione NON sarebbe efficiente: una soluzione con costo computazionale inferiore consiste infatti nell'esplorare il grafo a partire dal vertice u utilizzando una visita in ampiezza (BFS, Breadth First Search). Ricordiamo che la visita BFS esplora i nodi in ordine non decrescente rispetto alla distanza (intesa come numero minimo di archi) dal nodo sorgente. Nel nostro caso, poiché tutti gli archi hanno lo stesso peso positivo α, la minima distanza di v da u è data dal minimo numero di archi che separa i due nodi, come determinato dall'algoritmo BFS, moltiplicato per α.
Lo pseudocodice seguente realizza una versione di BFS semplificata, in cui (i) non viene mantenuto esplicitamente l'albero di copertura (in quanto non richiesto dall'esercizio), e (ii) si utilizza l'attributo v.dist settato a +inf ("infinito") per denotare i nodi che non sono ancora stati esplorati (in modo tale da evitare l'ulteriore attributo booleano v.mark).
Lo pseudocodice è il seguente:

```
algoritmo DistanzaMinimaAlpha( grafo G=(V, E), nodo u, nodo v double α )→double
    // inizializza ogni nodo w come inesplorato
    foreach w in V do
        w.dist := +inf;
    endfor;
    Queue Q;
    u.dist := 0;
    Q.enqueue(u);
    while ( ! Q.isEmpty() ) do
        nodo n := Q.dequeue();
        if (n == v) then
            return n.dist * α; // abbiamo raggiunto v
        endfor
        foreach w adiacente a n do
            if (w.dist == +inf) then
                w.dist := n.dist + 1;
                Q.enqueue(w);
            endif
        endfor
    endwhile
    // se siamo arrivati qui, significa che il nodo v non è stato raggiunto
```

```
return +inf;
```

L'algoritmo esegue una visita BFS dell'intero grafo nel caso peggiore, per cui il costo computazionale è lo stesso della visita BFS, ovvia O(n + m) (n=numero di nodi, m=numero di archi, se il grafo viene implementato tramite liste di adiacenza).

Esercizio 8.10

Si consideri un grafo orientato G=(V, E), non necessariamente connesso.

1. Scrivere un algoritmo efficiente che, dato in input il grafo orientato G=(V, E) e due vertici distinti u, v, restituisce "true" se e solo se esiste un cammino orientato che parte da u e conduce in v. L'algoritmo deve ritornare "false" se tale cammino non esiste.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1. specificando anche il tipo di rappresentazione del grafo che si considera.

Soluzione. È possibile risolvere questo problema mediante un algoritmo di visita DFS o BFS (poiché non è richiesto di individuare cammini minimi, si può usare uno dei due).

Esercizio 8.11

Si consideri un grafo non orientato connesso G=(V, E) i cui archi hanno tutti lo stesso peso w>0. Volendo calcolare un Minimum Spanning Tree di G è ovviamente possibile utilizzare uno degli algoritmi visti a lezione. Sfruttando però il fatto che tutti gli archi hanno lo stesso peso, è possibile fare di meglio usando un algoritmo appositamente sviluppato.

1. Descrivere un algoritmo efficiente che, dato in input il grafo G e il peso w di tutti i suoi archi, determini un Minimum Spanning Tree di G. (Suggerimento: in realtà il valore del peso w non serve)
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Esercizio 8.12

Si consideri un grafo orientato G=(V, E) non pesato. Ricordiamo che il grafo trasposto G^T si ottiene invertendo la direzione di tutti gli archi presenti in E.

1. Supponendo che il grafo G sia rappresentato mediante matrice di adiacenza, scrivere un algoritmo avente complessità ottima in grado di modificare la matrice in modo da trasformare G nella sua versione trasposta G^T. L'algoritmo deve richiedere spazio ulteriore O(1) (non è quindi possibile ricopiare la matrice di adiacenza).
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Ricordiamo che la matrice di adiacenza M di un grafo con n nodi è una matrice quadrata di dimensione n×n, tale che M[i, j] = 1 se e solo se il grafo contiene l'arco orientato (i, j). Per calcolare il grafo trasposto è necessario invertire tutti gli archi, ossia trasformare ogni arco (i, j) nell'arco (j, i). Per fare ciò è sufficiente scambiare tra loro i valori M[i, j] e M[j, i]; in pratica questo equivale a calcolare la matrice trasposta M^T. Attenzione al ciclo interno: la variabile j ha come valore iniziale i+1: è sbagliato far partire j da 1, in quanto ogni elemento verrebbe scambiato due volte, ottenendo nuovamente la matrice iniziale.

```
algoritmo GrafoTrasposto( array M[1..n, 1..,n] di int ) → M'
   for i := 1 to n-1 do
        for j := i+1 to n do
             int tmp := M[i, j];
             M[i, j] := M[j, i];
             M[j, i] := tmp;
        endfor
   endfor
   return M; // restituisce la matrice M modificata
```

Il corpo del ciclo for più interno ha costo O(1), e viene eseguito (n-1) + (n-2) + … + 2 + 1 = $\Theta(n^2)$ volte.

Quindi il costo dell'algoritmo è $\Theta(n^2)$.

Esercizio 8.13

1. Dati un grafo non orientato $G=(V, E)$, un nodo s in V e un intero $k > 0$, scrivere un algoritmo di complessità ottima che ritorna true se e solo se la componente connessa cui appartiene il nodo s ha un numero di nodi $\leq k$ (il nodo s deve essere incluso nel conteggio, quindi se s è un nodo isolato, la componente connessa cui appartiene ha un nodo).
2. Analizzare il costo computazionale dell'algoritmo proposto.

Soluzione. Si può utilizzare una semplice variante dell'algoritmo di visita in ampiezza; si noti che non è necessario calcolare le distanze dalla sorgente, in quanto non rilevante per l'esercizio.

```
algoritmo CCMAXK( grafo G=(V, E), nodo s, int k ) → bool
   foreach v in V do
        v.mark = false;
   endfor
   Queue Q;
   int size := 0;                    // dimensione componente connessa
   s.mark := true;
   Q.ENQUEUE(s);
   while ( ! Q.ISEMPTY() ) do
        u := Q.DEQUEUE();            // rimuovi il primo nodo da Q
        size := size+1;
        if ( size > k ) then
             return false;
        endif
        foreach v adiacente a u do
             if ( v.mark == false ) then
                  v.mark := true;
                  Q.ENQUEUE(v);
             endif
        endfor
   endwhile
   return true;                      // se arriviamo qui significa che la componente connessa cui
appartiene s ha ≤ k nodi
```

Il costo è quello di una normale visita in ampiezza, ossia $O(n+m)$ nel caso peggiore (essendo n=numero di nodi e m=numero di archi).

Esercizio 8.14

Un ipotetico social network è rappresentato tramite un grafo non orientato $G=(V, E)$. I nodi rappresentano gli utenti, e ciascun arco (u, v) denota il fatto che gli utenti u e v sono "amici". Supponiamo che in tale social network tutti i messaggi inseriti da un utente siano visibili dagli amici, e anche dagli "amici degli amici".

1. Scrivere un algoritmo che, dato in input il grafo G e un nodo s, restituisce il numero di utenti che riceverebbero i messaggi inseriti dall'utente s (supponiamo che s faccia esso stesso parte del risultato, quindi ad esempio se s è un nodo isolato, l'algoritmo restituisce 1).
2. Analizzare il costo computazionale dell'algoritmo di cui al punto 1.

Soluzione. Si può applicare il consueto algoritmo di visita in ampiezza modificato in modo tale da fermarsi non appena scopre un nodo che si trova a distanza > 2; sappiamo infatti che l'algoritomo di visita in ampiezza esplora il grafo "a livelli", visitando i nodi a distanza via via crescente dalla sorgente.

```
algoritmo DIFFUSIONE( grafo G=(V, E), nodo s ) → int
   foreach v in V do
        v.mark := false;
        v.dist := +inf;
```

```
    endfor
    Queue Q;
    int count := 0;                 // numero persone che ricevono gli update
    s.dist := 0;
    s.mark := true;
    Q.ENQUEUE(s);
    while ( ! Q.ISEMPTY() ) do
        nodo u := Q.DEQUEUE();  // rimuovi il primo nodo da Q
        if ( u.dist > 2 ) then
            return count;           // appena si incontra un nodo a distanza > 2, si termina
        endif
        count := count + 1;
        foreach v adiacente a u do
            if ( v.mark == false ) then
                v.mark := true;
                v.dist := u.dist + 1;
                Q.ENQUEUE(v);
            endif
        endfor
    endwhile
    return count; // questo è necessario perché potrebbero non esistere nodi a distanza > 2
dalla sorgente, quindi l'algoritmo potrebbe non effettuare mail il return all'interno del ciclo
while
```

Il costo dell'algoritmo è $O(n+m)$, essendo in sostanza un algoritmo di visita che al limite potrebbe terminare in anticipo rispetto ad una visita BFS.

Esercizio 8.15

Una remota città sorge su un insieme di n isolotti, ciascuno identificato univocamente da un intero nell'intervallo 1, …, n. Gli isolotti sono collegati da ponti, che possono essere attraversati in entrambe le direzioni. Quindi possiamo rappresentare la città come un grafo non orientato $G=(V, E)$, essendo V l'insieme degli n isolotti ed E l'insieme dei ponti.
Ogni ponte $\{u, v\}$ è in grado di sostenere un peso minore o uguale a $W[u, v]$. La matrice W è simmetrica, e i pesi sono numeri reali positivi. Se non esiste alcun ponte che collega direttamente u e v, poniamo $W[u, v] = W[v, u] = -\infty$
Un camion di peso P si trova inizialmente nell'isolotto s (sorgente) e deve raggiungere l'isolotto d (destinazione); per fare questo può servirsi solo dei ponti che sono in grado di sostenere un peso maggiore o uguale a P. Scrivere un algoritmo efficiente che, dati in input la matrice W, il peso P, nonché gli interi s e d, restituisca il numero minimo di ponti che è necessario attraversare per andare da s a d, ammesso che ciò sia possibile. Stampare la sequenza di isolotti attraversati nel percorso di cui sopra.

Soluzione. Utilizziamo un algoritmo di visita in ampiezza modificato opportunamente per evitare di attraversare i ponti che non sosterrebbero il peso P.

```
algoritmo ATTRAVERSAISOLE( array W[1..n, 1..n] di double, double P, int s, int d ) → int
    array mark[1..n] di bool;
    array parent[1..n] di int;
    array dist[1..n] di int;
    Queue Q;
    for v := 1 to n do
        mark[v] := false;
        parent[v] := -1;
        dist[v] := +inf;
    endfor
    mark[s] := true;
    dist[s] := 0;
    Q.ENQUEUE(s);
```

```
while ( ! Q.isEmpty() ) do
    int u := Q.dequeue();
    if ( u == d ) then
        break;      // esci dal ciclo se si estrae il nodo destinazione dalla coda
    endif
    for v:=1 to n do
        if ( W[u,v] ≥ P && mark[v] == false ) then
            mark[v] := true;
            dist[v] := dist[u]+1;
            parent[v] := u;
            Q.enqueue(v);
        endif
    endfor
endwhile
if ( dist[d] == +inf ) then
    print "nessun percorso"
else
    int i := d;
    while (i ≠ -1) do
        print i;
        i := parent[i];
    endwhile;
endif
return dist[d];
```

Esercizio 8.16

Una rete stradale è descritta da un grafo orientato pesato $G=(V, E, w)$. Per ogni arco (u, v), la funzione costo $w(u, v)$ indica la quantità (in litri) di carburante che è necessario consumare per percorrere la strada che congiunge u con v. Tutti i costi sono maggiori o uguali a zero. Si consideri un veicolo che ha il serbatoio in grado di contenere C litri di carburante, inizialmente completamente pieno. Non sono presenti distributori di carburante.
Scrivere un algoritmo che, dati in input il grafo pesato G, la quantità C di carburante inizialmente presente nel serbatoio, e due nodi s e d, restituisce true se e solo se esiste un cammino che consente al veicolo di raggiungere d partendo da s, senza rimanere a secco durante il tragitto. E' possibile rappresentare il grafo come meglio si ritiene (es., mediante liste di adiacenza).

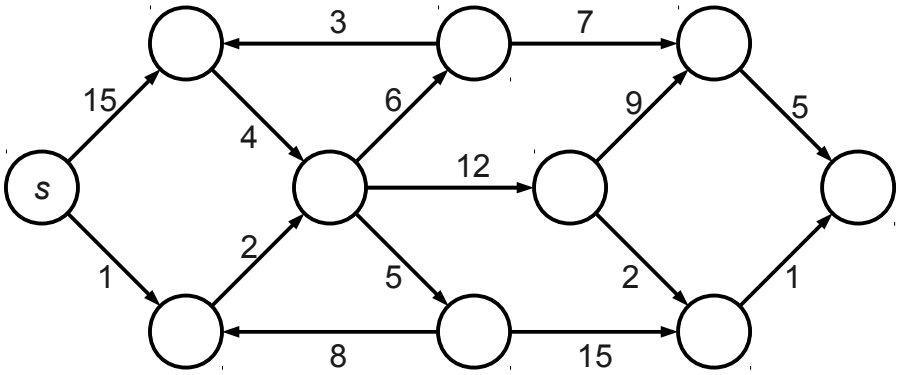
Soluzione. È sufficiente eseguire l'algoritmo di Dijkstra e calcolare il vettore $D[v]$ delle distanze minime dal nodo sorgente s a ciascun nodo v raggiungibile da s, utilizzando il consumo di carburante come peso. L'algoritmo restituisce true se e solo se $D[d]$ risulta minore o uguale a C.

Esercizio 8.17

Scrivere un algoritmo efficiente che, dato in input un grafo non orientato $G=(V, E)$, un nodo $s \in V$ e un intero $d>0$, restituisce il numero di nodi che si trovano a distanza minore o uguale a d da s (il vertice s deve essere incluso nel conteggio).

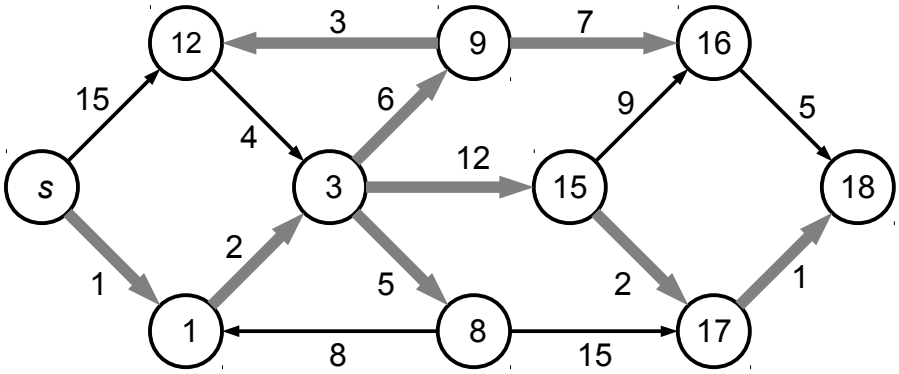
Esercizio 8.18

Si consideri il grafo orientato $G=(V, E)$, ai cui archi sono associati costi positivi come illustrato in figura:



Applicare "manualmente" l'algoritmo di Dijkstra per calcolare un albero dei cammini di costo minimo partendo dal nodo sorgente *s* (è il nodo più a sinistra in figura). Mostrare il risultato finale dell'algoritmo di Dijkstra, inserendo all'interno di ogni nodo la distanza minima da *s*, ed evidenziando opportunamente (ad esempio, cerchiando il valore del costo) gli archi che fanno parte dell'albero dei cammini minimi.

Soluzione.



Esercizio 8.19

La mappa di un videogioco è rappresentata da un grafo non orientato $G=(V, E)$: ogni nodo v rappresenta una "stanza", e ogni arco $\{u, v\}$ indica l'esistenza di un passaggio diretto dalla stanza u alla stanza v (e viceversa). Le stanze sono in tutto n, numerate con gli interi da *1* a n. L'array di interi $M[1..n]$ indica la presenza o meno di un mostro in ciascuna stanza. Se $M[v] = 0$ la stanza v è libera; se $M[v] = 1$, la stanza v ospita un mostro. All'inizio del gioco, il giocatore si trova nella stanza s; ad ogni turno può spostarsi dalla stanza corrente in una adiacente (purché libera). Scopo del gioco è di raggiungere, se possibile, la stanza di destinazione d nel minor numero di turni, attraversando esclusivamente stanze libere. Le stanze s e d sono sempre libere.
Scrivere un algoritmo che, dato in input il grafo G, l'array $M[1..n]$ e i nodi s, d, restituisca il minimo numero di turni necessari per andare da s a d, se questo è possibile. Nel caso in cui ciò non sia possibile, l'algoritmo deve restituire $+\infty$. (Nota: se $s=d$, l'algoritmo restituisce zero. Se s e d sono adiacenti, l'algoritmo restituisce 1). Determinare il costo computazionale dell'algoritmo proposto.

Soluzione. Questo esercizio poteva essere risolto mediante una semplice visita in ampiezza opportunamente modificata per evitare le stanze occupate da mostri. Sfruttiamo il fatto che i vertici sono identificati da interi nell'intervallo 1..n per scrivere il codice nel modo seguente:

```
algoritmo GIOCO( array M[1..n] di int, Grafo G=(V, E), vertice s, vertice d ) → int
    array dist[1..n] di int;
    foreach v in V do
        dist[v] := +inf;
    endfor
    Queue Q;
    dist[s] := 0;
    Q.ENQUEUE(s);
    while ( ! Q.ISEMPTY() ) do
        u := Q.DEQUEUE();
        if ( u == d ) then
            return dist[u];                    // abbiamo raggiunto la destinazione
        endif
        foreach v adiacente a u do
            if ( dist[v] == +inf && M[v] == 0 ) then
                dist[v] := dist[u] + 1;
                Q.ENQUEUE(v)
            endif
        endfor
    endwhile
    return +inf; // se arriviamo qui significa che la destinazione non è raggiungibile
```

Si noti che abbiamo semplificato l'algoritmo evitando di mantenere una apposita struttura dati (oppure un attributo booleano) per indicare quali nodi sono stati visitati. Infatti i nodi visitati sono tutti e soli quelli per cui la distanza non è +inf.
Il costo computazionale dell'algoritmo è quello di una normale visita BFS, ossia $O(n+m)$ nel caso peggiore (si noti che l'algoritmo termina immediatamente non appena raggiunge il vertice d, quindi potrebbe non visitare l'intero grafo).

Esercizio 8.20

Si consideri un grafo orientato non pesato G rappresentato mediante una matrice di adiacenza $M[1..n, 1..n]$, dove $M[i, j] = 1$ se e solo se esiste l'arco orientato (i, j). Si consideri un array di k numeri interi $C[1..k]$ tutti compresi tra 1 e n.

1. Scrivere un algoritmo efficiente che, dato in input la matrice M, l'array C, e due interi s e d, restituisce true se e solo se il vettore C rappresenta un cammino (inteso come sequenza di nodi visitati) dal nodo s al nodo d. Non è richiesto che C sia un cammino minimo, basta che sia un cammino qualunque che parte in s e termina in d.
2. Analizzare il costo dell'algoritmo proposto.

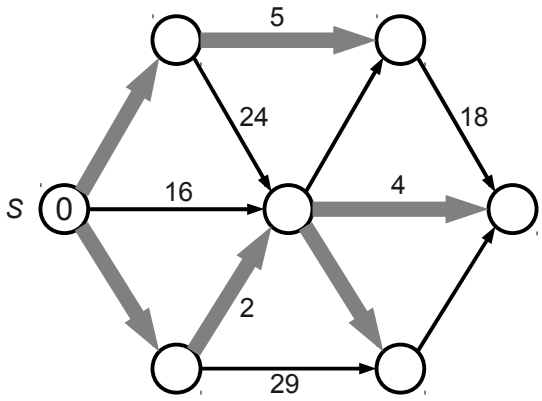
Soluzione.

```
algoritmo CONTROLLACAMMINO( array M[1..n, 1..n] di int, array C[1..k] di int, int s, int d ) →
bool
    if ( C[1] != s || C[k] != d ) then
        return false;
    endif
    for i:=1 to k-1 do
        if ( M[ C[i], C[i+1] ] != 1 ) then
            return false;
        endif
    endfor
    return true;
```

Il costo computazionale è $O(k)$

Esercizio 8.21

Si consideri il seguente grafo orientato:



Su alcuni archi sono indicati i pesi, su altri no.

1. Completare il grafo indicando dei pesi a piacere sugli archi che ne sono sprovvisti, in modo che applicando l'algoritmo di Dijkstra a partire dal nodo *S* si ottenga l'albero dei cammini minimi indicato dagli archi in grassetto.
2. Indicare all'interno di ciascun nodo la distanza minima da *S*, calcolata dall'algoritmo di Dijkstra applicato al grafo con i pesi indicati nel punto 1. Il nodo *S* è già stato etichettato con il valore 0

Capitolo 9: Esercizi di Programmazione

In questo capitolo vengono descritti, senza alcun ordine preciso, alcuni esercizi di programmazione con cui viene chiesto di progettare e implementare un algoritmo utilizzando un linguaggio di programmazione (es., Java, C, C++...).

Esercizio 9.1

Si richiede di implementare una variante dell'algoritmo QuickSort, che chiameremo QuickSort2, come segue. Definiamo un parametro intero $k \geq 0$. L'algoritmo QuickSort2 deve ordinare un vettore di n numeri interi, in senso non decrescente, in due fasi. Nella prima fase si applica l'algoritmo QuickSort normale, con la differenza che la ricorsione termina quando la dimensione del sottovettore da ordinare diventa minore o uguale a k. Ovviamente, così facendo al termine della prima fase l'array in generale potrà non risultare ordinato. Per questo motivo, nella seconda fase si completa la procedura di ordinamento eseguendo l'algoritmo BubbleSort sull'intero vettore.
Se k=0 oppure k=1, ovviamente QuickSort2 è esattamente uguale a QuickSort, nel senso che al termine della prima fase l'array risulta già ordinato e la seconda fase risulta superflua. Se $k > 1$, invece, la seconda fase risulta indispensabile per la correttezza dell'ordinamento.
Scopo di questa esercitazione è lo studio sperimentale del tempo di esecuzione dell'algoritmo QuickSort2 in funzione del parametro k. In particolare:

- Implementare in Java l'algoritmo QuickSort2 come descritto sopra (siete liberi di decidere come scegliere il pivot della fase di partizionamento);
- Scegliere una dimensione n del vettore da ordinare che sia sufficientemente grande per poter misurare i tempi di esecuzione di QuickSort2 (ad esempio $n=2^{18}$);
- Considerare diversi valori di k che siano potenze di due, partendo da 1. Ad esempio k=1, 2, 4, 8, 16... (è lasciata libertà di scegliere quanti diversi valori di k considerare, purché siano in numero congruo);
- Per ciascuno dei diversi valori di k, generare casualmente una serie (ad esempio 10) di vettori di n elementi, e misurare il tempo di esecuzione di QuickSort2 su tali vettori. Per ciascuna serie di prove calcolare il tempo medio di esecuzione di QuickSort2.
- Produrre un grafico in cui in ascissa ci siano i valori di k considerati, e in ordinata le medie dei tempi di esecuzione per ciascun set di esperimenti, come determinate nel punto precedente.
- (Facoltativo) Rispondere alla seguente domanda: quante iterazioni effettua (nel caso peggiore) l'algoritmo Bubble Sort nella seconda fase di QuickSort2? Motivare brevemente la risposta.

Esercizio 9.2

Scopo di questo progetto è il calcolo sperimentale dell'altezza media di un albero binario di ricerca (non bilanciato). Consideriamo il problema di inserire n chiavi casuali all'interno di un ABR inizialmente vuoto, e vogliamo calcolare l'altezza media dell'ABR risultante in funzione di n.
Si richiede di procedere come segue. Implementare in Java la struttura dati Albero Binario di Ricerca, in cui

ogni nodo mantiene una chiave reale (non è necessario associare ulteriori informazioni alle chiavi). La struttura dati deve supportare l'operazione di inserimento di una nuova chiave, e di calcolo dell'altezza dell'albero. L'operazione di inserimento non deve causare alcun ribilanciamento dell'albero; non è richiesto di implementare anche le funzioni di cancellazione o ricerca di chiavi.
Sia n il numero di nodi dell'albero. Si scelgano opportunamente una costante k (ad esempio, k=10) e un insieme di possibili valori per n (ad esempio, n=10, 20, 30, ... oppure n=100, 200, 300, …). Per ogni valore di n, costruire per k volte un albero binario di ricerca con n nodi, partendo dall'albero vuoto ed inserendo n chiavi reali generate casualmente con distribuzione uniforme nell'intervallo [0, 1]. Sia H_n l'altezza media dei k alberi con n nodi che sono stati generati.

1. Disegnare il grafico di H_n in funzione di n, per tutti i valori di n considerati;
2. Nella letteratura scientifica è stata data una espressione analitica di H_n, nel caso di alberi binari di ricerca contenenti una permutazione casuale degli interi {1, …, n}. Ad esempio, si veda *"A note on the height of binary search trees"* di Luc Devroye, Journal of the ACM (JACM) Volume 33 Issue 3, July 1986, disponibile accedendo dalla rete di Ateneo all'indirizzo http://dx.doi.org/10.1145/5925.5930 oppure *"The Height of a Random Binary Search Tree"* di Bruce Reed, disponibile all'indirizzo http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.1289&rep=rep1&type=pdf. Confrontare il risultato del grafico ottenuto nel punto 1 con uno dei risultati analitici presi dalla letteratura: sono compatibili?

Esercizio 9.3

Supponiamo di conoscere in anticipo le quotazioni di borsa dei prossimi n giorni. In particolare, per ogni i=1, …, n conosciamo il prezzo $A[i]$ a cui è possibile acquistare le azioni ACME inc. il giorno i. Il prezzo di vendita $V[i]$ al giorno i è sempre inferiore di 0.5 rispetto al prezzo di acquisto, ossia $V[i] = A[i] – 0.5$. Armati di tali informazioni, vogliamo determinare i giorni i, j con $1 \le i \le j \le n$ tali che acquistando le azioni al giorno i e rivendendole al giorno j, il guadagno unitario $G(i, j) := V[j] – A[i]$ sia il massimo possibile. Si noti che è possibile vendere le azioni lo stesso giorno in cui vengono acquistato, oppure in un qualsiasi giorno successivo; non è possibile vendere azioni prima di averle acquistate.
Il progetto consiste nello sviluppo di un programma Java di complessità ottima che, dati i prezzi di acquisto come specificato a breve, stampa a video il guadagno unitario massimo; non è richiesta la stampa dei giorni in cui acquistare e vendere le azioni per ottenere il guadagno di cui sopra.
Il programma deve leggere i dati di input da un file di testo avente la struttura seguente:

- la prima riga contiene il valore dell'intero n;
- seguono n righe ciascuna delle quali contiene il prezzo di acquisto $A[i]$ espresso come numero reale.

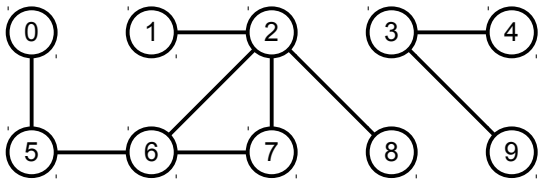
Ad esempio, il file di input potrebbe avere il contenuto seguente:

```
5
12.3
16.5
19.6
11.2
17.9
```

Nell'esempio di cui sopra, acquistando le azioni al giorno 1 al prezzo di 12.3 e rivendendole al giorno 4 al prezzo di 11.2-0.5 = 10.7 si avrebbe un guadagno (negativo!) di -1.5. Il guadagno massimo si ottiene acquistando il primo giorno al prezzo di acquisto di 12.3 e rivendendo al terzo giorno al prezzo di vendita di 19.6-0.5 = 19.1; il guadagno massimo è pertanto 19.1-12.3 = 6.8.
A titolo di esempio, nella pagina del corso è presente un file di esempio (con l'indicazione del risultato che l'algoritmo deve stampare). (*Suggerimento: si veda l'Esercizio 7.8*).

Esercizio 9.4

Un circuito stampato può essere rappresentato come un grafo non orientato $G=(V, E)$, con $V = \{0, 1, \ldots n\text{-}1\}$ l'insieme degli $n \geq 2$ "piedini" (pin) ed E l'insieme delle $m \geq 1$ connessioni che collegano tra loro coppie di pin. I segnali elettrici si propagano da un pin a tutti quelli ad esso collegati, sia direttamente che indirettamente tramite altri pin. Ad esempio, il circuito seguente ha n=10 pin e m=9 connessioni:



I pin 0 e 8 sono tra di loro collegati, mentre 1 e 9 non lo sono.
Scrivere un programma Java che legge da un file la descrizione del circuito e da un secondo file una sequenza di coppie di pin. Il programma deve stampare a video, per ogni coppia nel secondo file, se i pin sono connessi oppure no. E' richiesto che il programma costruisca l'insieme delle componenti connesse del grafo mediante una struttura Union-Find; per ogni arco $\{u, v\}$ del grafo, il programma unisce gli insiemi contenenti u e v. Terminata questa fase, due pin s e d sono connessi se e solo se find(s) == find(d). E' lasciata libertà di decidere il tipo di struttura Union-Find da implementare (QuickUnion, QuickFind, con o senza euristiche), tenendo presente che l'implementazione corretta di una euristica comporterà un punteggio maggiore in sede di valutazione. Non è consentito fare ricorso a implementazioni già pronte delle strutture Union-Find, nemmeno se presenti nella libreria standard di Java.
Il primo file di input, passato sulla riga di comando, contiene la descrizione del grafo. Ad esempio (nota: le parole in corsivo non fanno parte del file di configurazione):

```
10                  Numero di nodi n
8                   Numero di archi m
1 2                 origine, destinazione arco 0
1 6
6 7
2 7
3 4
7 8
8 4
4 0                 origine, destinazione arco m-1
```

Il secondo file contiene $q \geq 1$ coppie di pin da testare. Ad esempio:

```
6                   Numero q di coppie da controllare
1 0                 origine, destinazione coppia 0 da testare
5 0
3 1
8 1
8 2
8 9                 origine, destinazione coppia q-1 da testare
```

A fronte degli input sopra, il programma deve produrre a video l'output seguente:

```
1 0 C
5 0 NC
3 1 C
8 1 C
8 2 C
```

8 9 NC

L'output è composto da *q* righe, ciascuna contenente gli identificatori dei pin, nell'ordine in cui compaiono nel secondo file di input, e la stringa C se sono connessi, NC se non lo sono.
Analizzare il costo asintotico della prima fase di costruzione delle strutture Union-Find, e della seconda fase di esecuzione di tutte le query.

Esercizio 9.5

Disponiamo di $n \geq 1$ files le cui dimensioni (in MB) sono numeri interi positivi, minori o uguali a 650. Disponiamo anche di una scorta illimitata di CD-ROM, aventi capacità di 650MB ciascuno, su cui vogliamo copiare i files in modo da non eccedere la capienza dei supporti, e assicurandosi che nessun file venga spezzato su più supporti diversi. I files possono essere memorizzati senza rispettare alcun ordine prefissato, purché la somma delle dimensioni dei file presenti sullo stesso supporto non superi 650MB.
Realizzare un algoritmo per allocare i files sui CD-ROM senza spezzare nessun file, e minimizzando il numero di CD-ROM usati. Poiché non esistono algoritmi efficienti per risolvere questo problema, si chiede di implementare l'euristica seguente. Si riempie a turno ciascun CD-ROM, usando i files ancora da memorizzare, in modo da massimizzare lo spazio occupato senza eccedere la capienza dei supporti; per questo è richiesto l'uso di un approccio basato sulla programmazione dinamica. L'algoritmo termina quando tutti i files sono stati copiati sui supporti.
Il programma accetta sulla riga di comando il nome di un file di input, avente la struttura seguente:

8		*numero di file (n)*
progetti	143	*nome e dimensione del file 0*
esempio	242	
film1	133	
film2	181	
programma	74	
programma2	189	
documento	26	
immagine	192	*nome e dimensione del file n-1*

Il programma, eseguito sull'input precedente, deve visualizzare il seguente output:

```
Disco: 1
progetti 143
film1 133
film2 181
immagine 192
Spazio libero: 1

Disco: 2
esempio 242
programma 74
programma2 189
documento 26
Spazio libero: 119
```

In questo caso vengono utilizzati due CD-ROM; per ciascuno viene visualizzato un header Disco *X* seguito dall'elenco dei nomi dei files (in un ordine qualsiasi) con le relative dimensioni. Infine, la riga Spazio libero: *Y* indica lo spazio (in MB) ancora disponibile sul disco. Una riga vuota separa ciascun blocco dal successivo.
La relazione deve contenere una descrizione precisa dell'algoritmo di programmazione dinamica usato per riempire i CD-ROM. In particolare, indicare:

- come sono definiti i sottoproblemi;
- come sono definite le soluzioni a tali sottoproblemi;
- come si calcolano le soluzioni nei casi “base”;
- come si calcolano le soluzioni nel caso generale.

Analizzare il costo asintotico dell'algoritmo implementato (ci si riferisce all'algoritmo completo, non solo a quello di programmazione dinamica usato per riempire ciascun CD-ROM)..

Esercizio 9.6

Si consideri una scacchiera rappresentata da una matrice quadrata $M[0..n\text{-}1, 0..n\text{-}1]$ di caratteri, con $n \geq 2$. La casella di coordinate (i, j) si considera libera se contiene il carattere '.', mentre è occupata se contiene il carattere '#'. Una pedina viene posizionata inizialmente su una casella marcata con 's' (sorgente), e deve raggiungere nel minor numero di mosse la casella marcata con 'd' (destinazione), se ciò è possibile. Ad ogni mossa, la pedina può spostarsi dalla posizione (i, j) ad una delle posizioni $(i+1, j)$, $(i-1, j)$, $(i, j+1)$, $(i, j-1)$, purché (i) la posizione di destinazione sia libera (la casella 'd' si assume libera), e (ii) la posizione di destinazione sia all'interno la tabella (in altre parole, non si può uscire dai bordi).
Il programma deve accettare sulla riga di comando il nome di un file di input il cui contenuto abbia la seguente struttura:

```
10                  dimensione della scacchiera n
......##..          riga 0
##.s..##..
##....##..
####......
.......###
....######
..########
........d.
###...##..
..........          riga n-1
```

Il programma deve calcolare un cammino che consenta di spostare la pedina dalla posizione 's' alla posizione 'd' eseguendo il minor numero possibile di mosse. Se il cammino esiste, il programma deve stampare a video la scacchiera, evidenziando con il carattere '+' le caselle che fanno parte di tale percorso (incluse quelle precedentemente marcate 's' e 'd'). Immediatamente sotto, deve stampare il numero di caselle che compongono il percorso minimo, includendo anche la casella iniziale e finale del percorso (in sostanza, il programma deve stampare il numero di simboli '+' presenti sulla scacchiera).
Con riferimento al file di parametri precedente un possibile output è questo (nota: in generale possono esistere più percorsi minimi, basta calcolarne uno qualsiasi) :

```
......##..
##.++.##..
##..+.##..
####+.....
.++++..###
.+..######
.+########
.++++++++.
###...##..
..........
18
```

Se il percorso non esiste, cioè nel caso in cui la destinazione non sia raggiungibile dalla sorgente, il programma deve stampare a video il messaggio non raggiungibile.
Dare una stima del costo asintotico dell'algoritmo implementato.

Esercizio 9.7

La pianta stradale di una città è rappresentata da un grafo orientato $G=(V, E)$ composto da $n \geq 2$ nodi {0, 1,

… n-1}, che rappresentano incroci stradali, e $m \geq 1$ archi che rappresentano strade che collegano coppie di incroci. Ogni arco è etichettato con un valore reale positivo, che indica il tempo (in secondi) necessario per percorrere la strada corrispondente. Su ciascuno degli n incroci è posizionato un semaforo: prima di poter imboccare una qualsiasi delle strade che escono dall'incrocio (archi in uscita dal nodo corrispondente) è necessario aspettare che il semaforo diventi verde. Disponiamo di una funzione double attesa(int i, double t), che accetta in input l'identificativo di un nodo i (intero compreso tra 0 e n-1), e l'istante di tempo t in cui si arriva all'incrocio. La funzione restituisce un valore reale ≥ 0, che indica il tempo di attesa prima che il semaforo diventi verde. Quindi, supponendo di arrivare al nodo i al tempo T, sarà possibile imboccare uno degli archi in uscita all'istante T + attesa(i, T).
Implementare un programma per calcolare il tempo minimo necessario per andare dal nodo 0 al nodo n-1, se possibile, supponendo di trovarsi al nodo 0 all'istante t=0. Si presti attenzione al fatto che, da quanto detto sopra, si può imboccare una delle strade uscenti dal nodo 0 all'istante attesa(0, 0).
Il programma accetta un unico parametro sulla riga di comando, che rappresenta un file contenente la descrizione della rete stradale. Un esempio è il seguente:

```
5                       numero di nodi n
7                       numero di archi m
0       1       132.3   origine, destinazione e tempo di percorrenza arco 0
1       2       12.8
0       3       23.81
3       2       42.0
2       4       18.33
3       4       362.92
3       1       75.9    origine, destinazione e tempo di percorrenza arco m-1
```

Se il nodo destinazione è raggiungibile a partire dalla sorgente, il programmao stampa a video un numero reale che indica il tempo necessario per raggiungere la destinazione; il tempo deve ovviamente includere le soste ai semafori. Se il nodo destinazione non è raggiungibile dalla sorgente, il programma stampa a video il messaggio non raggiungibile. Ad esempio, considerando una ipotetica implementazione della funzione attesa() che restituisce sempre il valore 5.0 (costante), l'algoritmo applicato all'esempio precedente stamperà a video il valore:

```
99.14
```

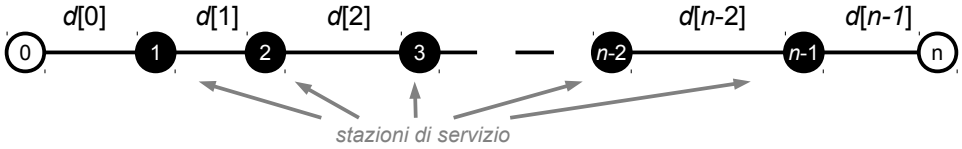
Analizzare il costo asintotico dell'algoritmo implementato.
Facoltativo: in caso di destinazione raggiungibile, stampare anche la sequenza di nodi visitati, incluso il nodo 0 e il nodo n-1, nell'ordine in cui vengono attraversati. Nell'esempio precedente, l'output completo potrebbe essere:

```
0 3 2 4
99.14
```

in quanto il cammino che consente di andare dall'incrocio 0 all'incrocio n-1 nel minor tempo possibile (assumendo 5 secondi di attesa ad ogni semaforo) è 0 → 3 → 2 → 4

Esercizio 9.8

Il dottor Emmett “Doc” Brown deve guidare la sua DeLorean lungo una strada rettilinea. L'auto può percorrere K Km con un litro di carburante, e il serbatoio ha una capacità di C litri. Lungo il tragitto si trovano n + 1 aree di sosta indicate con 0, 1, … n, con $n \geq 1$. L'area di sosta 0 si trova all'inizio della strada, mentre l'area di sosta n si trova alla fine. Indichiamo con $d[i]$ la distanza in Km tra le aree di sosta i e i+1. Nelle n-2 aree di sosta intermedie {1, 2, … n-1} si trovano delle stazioni di servizio nelle quali il dottor Brown può fermarsi per fare il pieno (vedi figura).



Tutte le distanze e i valori di *K* e *C* sono numeri reali positivi. Il dottor Brown parte dall'area 0 con il serbatoio pieno, e si sposta lungo la strada in direzione dell'area *n* senza mai tornare indietro.
Si chiede di implementare un algoritmo per calcolare il numero minimo di fermate che sono necessarie per fare il pieno e raggiungere l'area di servizio *n* senza restare a secco per strada, se ciò è possibile. Il programma legge i parametri di input da un file di configurazione, il cui nome è specificato sulla riga di comando. Il file ha la struttura seguente:

13.3	*km percorsi con un litro di carburante (K)*
30.0	*capacità del serbatoio, in litri (C)*
6	*valore di n*
150	*d[0]*
183.7	
51.9	
88.42	
247.3	
69.2	*d[n-1]*

Il programma deve stampare a video l'elenco delle aree di sosta in cui l'auto si ferma a fare rifornimento, oppure il messaggio non raggiungibile nel caso in cui il dottor Brown non possa raggiungere la destinazione senza rimanere a secco per strada. Ricordare che le aree di sosta in cui è presente una stazione di servizio sono quelle numerate come {1, 2, … *n*-1}. Nell'esempio sopra, il programma dovrà stampare:

3
5

Nel caso in cui la destinazione sia raggiungibile senza necessità di fermate intermedie, il programma non produrrà alcun output. Analizzare il costo asintotico dell'algoritmo proposto.