# Programmazione II
# Esercizi con soluzioni

Alessandro Panconesi
DSI, La Sapienza

**Esercizio.** Si consideri il seguente programma:

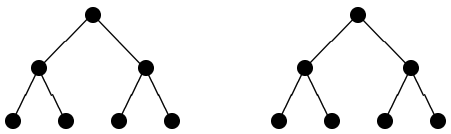
```
program P;
var i:  integer;

procedure A (k:  integer);
var i:  integer;
begin
i := 2;
i := i + k;
write(i);
k := 2;
end;

procedure C (var k:  integer);
begin
i := 2;
i := i + k;
write(i);
k := 1;
end;

begin
i:=1; A(i); C(i); write(i);
end.
```

Se il programma `P` venisse eseguito cosa stamperebbe ed in che ordine?

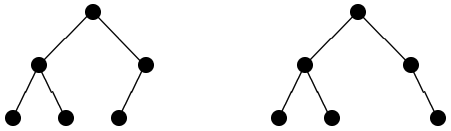
**Soluzione:** 3, 4, 1

**Esercizio.**

- Definire un tipo di dato astratto `albero` per implementare alberi binari in cui ogni nodo consiste di un campo `valore` e di due puntatori agli eventuali figli;
- Scrivere una funzione booleana PASCAL la quale, dati in ingresso due alberi binari `t1` e `t2`, determini se essi sono uguali oppure no. Due alberi sono "uguali" se possono essere sovrapposti l'uno su l'altro. Ad esempio



...sono uguali, mentre



..non lo sono.

- Qual'é la complessitá computazionale (tempo di calcolo) della vostra procedura?

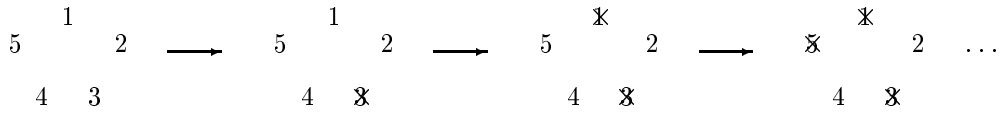
```
type
  albero = uparrownode;    node = record
    valore: integer;;
    sinistro: albero;
    destro: albero;
    end;
```

**Soluzione:**

```
function uguali (t1, t2: albero): boolean;
begin
if (t1 = nil) and (t2 = nil) then uguali := true;
else if (t1 = nil) or (t2 = nil) then uguali := false;   {uno e nil e l'altro no}
else uguali := uguali(t1.sinistro, t2.sinistro) and uguali(t1.destro, t2.destro);
end;
```

**Esercizio.**

- Scrivere una procedura PASCAL `creaLista` che crei una lista circolare di $n$ elementi in cui cioé $i$ punta ad $i+1$, per $1 \leq i < n$, ed $n$ punta ad 1.
- Scrivere una procedura `ambarabacicicoco` con due parametri $n$ ed $m$ tale che: (a) prima viene creata una lista circolare di $n$ elementi e poi (b) partendo dal primo elemento, gli elementi vengono eliminati di $m$ in $m$ uno dopo l'altro, come quando si fa la conta tra ragazzini. La procedura deve stampare gli elementi che man mano vengono eliminati. Ad esempio, `ambarabacicicoco(5, 3)` dá luogo a questa successione di liste circolari



e pertanto stamperá la sequenza di elementi eliminati:

$$3, 1, 5, 2, 4.$$

**Esercizio.**

- Scrivere una funzione PASCAL `eliminaDoppioni` che, data in ingresso una lista contenente numeri interi, elimini tutti gli eventuali doppioni. Ad esempio:

  `eliminaDoppioni([1,2,2,5,6,9,9,5,0,4])` $\Longrightarrow$ `[1,2,5,6,9,0,4]`.

**Esercizio.** Scrivere una procedura PASCAL la quale, dato come parametro di input un vettore *contenente solo 0 ed 1*, metta tutti gli 0 a sinistra e tutti gli 1 a destra. Esempio:

$$0100110010 \longrightarrow 0000001111$$

Determinate la complessitá computazionale del vostro algoritmo (giustificare la risposta!). Per il punteggio pieno il vostro programma deve avere complessitá lineare. **Soluzione.**

L'idea é quella di usare due puntatori. Il primo, denotato con `i`, é posizionato alla fine della fila di 0, mentre il secondo, `j`, si sposta da sinistra a destra alla ricerca di uno 0. Non appena questi viene individuato si scambia l'elemento puntato da `j` con quello puntato da `i+1`. L'algoritmo perció mantiene il seguente invariante: tra la posizione `i+1` e la `j-1`, estremi inclusi, non ci sono 0.

```
function aggiusta (n:  array [1..n] of integer): integer;
var i, j , t:  integer;

begin i := 0; j := 1;   (* invariante vale *)
while j <0 n do begin
  if a[j] = 0 then begin
    (* scambia a[i+1] con a[j] ed incrementa i *)
    t := a[i];  a[i] := a[j];  a[j] := t;
    i := i + 1;
    end;
  j := j + 1;
  end
end
```

**Esercizio.**

- Descrivere in modo chiaro e conciso in cosa consista la struttura dati **pila**, descriverne cioé il principio di funzionamento e le operazioni con le quali essa viene manipolata;
- Dare una implementazione PASCAL della struttura dati pila specificando sia le strutture dati che le funzioni per operare su di essa.

...in altre parole: (a) cos'è una pila e (b) come la implementereste?

**Soluzione.** La pila è una struttura dati che consente di memorizzare una lista di elementi secondo la tecnica *last in, first out* (LIFO) o, piú cristianamente, *" beati gli ultimi (ad entrare) che saranno i primi (ad uscire)"*. In soldoni quindi, una pila consiste di un insieme, possibilmente vuoto, di elementi messi uno sopra l'altro. La pila viene manipolata tramite due operazioni:

- `push(elemento, pila)`, che inserisce `elemento` in cima alla `pila`;
- `pop(pila)`, che toglie l' `elemento` in cima alla `pila`, la quale viene cosí aggiornata; l' `elemento` viene restituito come risultato dell'operazione.

Volendo implementare una pila tramite puntatori la si puó organizzare come una lista semplice, con la convenzione che il primo elemento della lista è l'elemento in cima alla pila. In questo modo è sufficiente conoscere il puntatore al primo elemento per poter effettuare sia l' inserzione di un nuovo elemento (push) che l'estrazione del primo elemento (push). Nel gestire una pila sono altresí utili le operazioni `creapila`, per la creazione di una nuova pila ed il test `pilavuota`, per verificare che una pila non sia vuota. Di seguito viene presentata una implementazione tramite puntatori.

```
type
   Pila:  ↑Elemento;
   Elemento = record
     valore:  integer;
     next:  Pila;
   end;


function empty(P: Pila):  boolean;
begin
  empty := (P = nil);
end;

procedure create(var P: Pila);
begin
  P = nil;
end;

procedure push(e:  ↑Elemento; var P: Pila);
begin
  e↑.next := P;
  P := e;
end;

function pop(var P: Pila):  ↑Elemento;
var t:  ↑Elemento;
begin
  t := P;
  if not empty(P) then
  begin
   P := P↑.next;
   t↑.next := nil; { clean dangling pointer }
  end;
  pop := t;
end;
```

**Esercizio.**

1. Definite in PASCAL una lista con puntatori;
2. Scrivete una procedura **iterativa** per stampare la lista;
3. Scrivete una procedura **ricorsiva** per stampare la lista;

**Soluzione.**

```
type
  lista = ↑ elemento;
  elemento = record
    valore:  integer;
    successivo:  lista;
  end;


procedure iterativa(l:  lista)
begin
  while (l <> nil) do
  begin
    writeln(l↑.valore);
    l := l↑.successivo;
  end
end


procedure ricorsiva(l:  lista)
begin
  if (l <> nil) do
  begin
    writeln(l↑.valore);
    ricorsiva(l↑.successivo);
  end
end
```

**Esercizio.** In questo esercizio si suppone di aver definito il seguente tipo di dato:

```
matrice = array [1..n, 1..m] of integer;
```

- Scrivere una funzione PASCAL

  ```
  somma(i: integer; a: matrice): integer
  ```

  che restituisce la somma dei valori contenuti nella riga `i`-esima della matrice `a`;

- Successivamente, scrivere una funzione PASCAL

  ```
  massimaRiga(a: matrice): integer
  ```

  che restituisce l' indice della riga di `a` di somma massima. Ad esempio, se fosse

  $$\begin{array}{rrrr} 1 & 0 & -1 & 1 \\ 1 & 0 & -2 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 0 & -2 & 2 \\ 1 & 0 & -3 & 3 \end{array}$$

  allora `massimaRiga(a)` restituirebbe il valore 3 in quanto è la terza riga ad avere somma massima. La terza riga ha infatti valore $1 + 2 + 3 + 4 = 10$, mentre tutte le altre hanno valore 1.

**Soluzione.**

```
function somma(i: integer; a: matrice): integer
var j, t: integer;
begin
  t := 0;
  for j := 1 to m do
    t := t + a[i,j];
  somma := t;
end


function massimaRiga(a: matrice): integer
var t, massimaRigaSinora, massimaSommaSinora: integer;
begin
  massimaSommaSinora := somma(1,a);
  massimaRigaSinora := 1;
  for i := 2 to n do
  begin
    t := somma(i, a);
    if (t > massimaSommaSinora) then
    begin
      massimaRigaSinora := i;
      massimaSommaSinora := t;
    end;
  end;
  massimaRiga := massimaRigaSinora;
end
```

```
function f(n:  integer):  integer;
var
  t:  array [0..max] of integer;
  i:  integer;
begin
  if (0 =< n) and (n =< max) then begin
    t[0] := 1; t[1] := 1;
    for i := 2 to n do
      t[i] := 3 * t[i-1] - t[i-2];
    f := t[n];
  end;
end;
```

**Esercizio.** Considerate la seguente situazione. Un file di nome `catalogo` contiene $n \approx 10^6$ voci. Ogni giorno viene ricevuto un file `elenco` contenente $m \approx 10^4$ voci. Deve essere messo a punto un programma che in sostanza deve fare questo: Per ogni voce in `elenco` si deve controllare se essa é presente o meno nel `catalogo`. Si prevede che il programma debba funzionare per $g \approx 10^3$ giorni di seguito, ogni giorno con un elenco diverso ma di dimensione costante pari a $m \approx 10^4$ voci. Per il programma vengono proposti due metodi:

1. Si ordina il `catalogo` e, per ogni voce $x$ dell'`elenco`, si controlla se $x$ é presente o meno nel `catalogo`;

2. Si ordina l' `elenco` e, per ogni voce $x$ del `catalogo`, si controlla se $x$ é presente o meno nell' `elenco`.

Supponendo che il confronto tra due voci e lo spostamento di una voce richiedano tempo costante, quale dei due metodi é da preferire? **Giustificare la vostra risposta.**

**Soluzione:** Per comoditá possiamo prendere 10 come base del logaritmo in quanto, come é noto, cambiando la base del logaritmo l'ordine di grandezza non cambia. Vale a dire, per ogni $a, b > 0$ costanti, $\Theta(\log_a n) = \Theta(\log_b n)$. Sostituendo per $m$ ed $n$ otteniamo

$$\textit{(tempo metodo 1)} = n \log n + g(m \log n) = 10^6\ 6 + 10^3\ 10^4\ 6$$

mentre

$$\textit{(tempo metodo 2)} = g(m \log m + n \log m) = 10^3(10^4 4 + 10^6 4)$$

Semplificando o calcolando direttamente si vede che

$$\textit{(tempo metodo 2)} > 60\textit{(tempo metodo 1)}$$

per cui il primo metodo è senz'altro da preferire.

**Esercizio.**

- Definire una struttura dati atta ad implementare un albero binario $T$.
- Scrivere una funzione booleana `verifica` per verificare se l'albero $T$ in ingresso é *ben bilanciato*, vale a dire se le lunghezze del cammino piú breve e del cammino piú lungo differiscono al piú di uno. L'implementazione deve essere tale che ogni nodo sia visitato una volta soltanto.

**Soluzione:**

```
type
   Tree:  ↑Elemento;
   Elemento = record
   valore:  integer;
   sin:  Tree;
   des:  Tree;


function bilanciato(T: Tree;):  boolean;
var min,max:  integer;
begin
  min = maxint; { inizializzazione della profonditá minima al massimo intero }
  max = 0;
  verifica(T,1,min,max);
  if (max <= min + 1) then bilanciato := true;
  else bilanciato := false;
end;

function verifica(T: Tree, prof:  integer; var min,max:  integer):  boolean;
begin
  if (foglia(T)) then { verifica se il nodo corrente é una foglia }
   if (prof < min) then
     min = prof;
   else if (prof > max) then
     max = prof;
  else begin
    if (T↑.sin ≠ nil) then verifica(T↑.sin,prof+1,min,max);
    if (T↑.des ≠ nil) then verifica(T↑.des,prof+1,min,max);
  end;
end;
```

**Esercizio.** Scrivere una procedura **ricorsiva** PASCAL la quale, data in ingresso una lista semplice di interi, la stampi in ordine inverso. Per "semplice" si intende che nella lista ogni elemento punta solo all'elemento successivo e **non** al precedente (ovviamente l'ultimo elemento punta a `nil`). Ogni elemento ha due campi, uno per valore a l'altro per il puntatore.

**Soluzione:**

```
type
   lista = ↑elemento;
   elemento = record
      value:  integer;;
      next:  lista;
      end;

procedure stampaInOrdineInverso(l:  lista);
begin
if (l <> nil) then begin
  stampaInOrdineInverso(l↑.next);
  writeln(l↑.value);
  end
end;
```

**Esercizio.** Si supponga di avere a disposizione delle funzioni siffatte:

- Una procedura `swap` con 3 parametri di ingresso: un vettore `x[1 .. n]` di numeri interi e due indici `i` e `j`; la funzione `swap` scambia di posto gli elementi $i$-esimo e $j$-esimo del vettore `x` (nel caso uno o entrambi gli indici siano al di fuori dell'intervallo `[1..n]` la funzione non fa nulla);
- Una funzione `select` con due parametri: un vettore `x[1 .. n]` di numeri interi ed un indice `i`; la funzione restituisce l'indice dell'elemento minimo del sottovettore `x[i..n]` e -1 nel caso `i` sia al di fuori dell'intervallo `[1..n]`.

Si risponda ai tre quesiti seguenti:

(a) Scrivere una procedura per ordinare un vettore di interi facendo uso di `swap` e `select`;

(b) Assumendo che una chiamata a `swap` costi tempo costante, mentre una a `select` un tempo proporzionale alle dimensioni del sottovettore `x[i..n]`, qual'é la complessitá computazionale della vostra procedura? Giustificare la vostra risposta;

(c) Specificare in che modo vadano passati i parametri a `swap` e `select`, cioé se per valore o per indirizzo.

**Soluzione.** Risposta al primo quesito:

```
procedure sort(x:  array [1..n] of integer);
var i, min:  integer;
begin
for i := 1 to n-1 do
  swap(i, select(i, x), x);
end;
```

Questa naturalmente altri non é che una implementazione del noto algoritmo *selection sort*.

Dato che una chiamata a `swap` ha costo unitario, mentre una a `select` ha costo proporzionale ad $n - i + 1$, il costo globale é dato da

$$\begin{aligned}
\sum_{i=1}^{n-1}[1 + (n - i + 1)] &= 2\sum_{i=1}^{n-1} 1 + \sum_{i=1}^{n-1}(n - i) \\
&= 2(n-1) + \sum_{i=1}^{n-1} i \\
&= 2(n-1) + \frac{n(n-1)}{2} \\
&= \Theta(n^2).
\end{aligned}$$

Il parametro `x` deve essere passato a `swap` per indirizzo, in quanto `x` stesso deve essere modificato; tutti gli altri passaggi di parametro possono (ed é bene che siano) per valore.

**Esercizio.** In seguito all'analisi di un algoritmo si scopre che il suo tempo di calcolo é dato dalla formula

$$T(n) = T(n-1) + T(n-3)$$

con condizione iniziale $T(0) = T(1) = T(2) = T(3) = 1$. Si scriva una funzione `pascal` per valutare la funzione $T$. Per ottenere il massimo del punteggio la vostra funzione deve essere efficiente dal punto di vista del tempo di calcolo.

**Soluzione.** Una semplice implementazione potrebbe essere la seguente:

```
function waste(n:  integer):  integer;
begin
if (0 =< n) and (n =< 3) then waste := 1;
else waste := waste(n-1) + waste(n-3);
end;
```

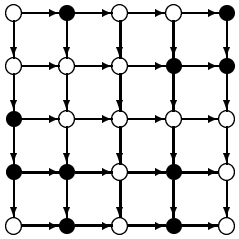
Questa implementazione ricorsiva richiede peró tempo esponenziale! (Grosso modo il suo tempo di calcolo cresce come i numeri di Fibonacci).

La seguente invece é una implementazione che impiega tempo lineare facendo uso di un vettore di appoggio:

```
function smart(n:  integer):  integer;
var
  t:  array [0..n] of integer;
  i:  integer;
begin
t[0] := 1; t[1] := 1; t[2] := 1; t[3] := 1;
for i := 4 to n do
  t[i] := t[i-1] + t[i-3];
smart := t[n];
end;
```

Queste procedure **non** effettuano il controllo dell'errore, vale a dire, se esse vengono chiamate con $n < 0$, `waste` entra in un loop infinito mentre `smart` darebbe errore cercando di accedere a `t[n]`. Come utile esercizio si apportino semplici modifiche a questi programmi per gestire possibili errori.

**Esercizio.** Si vuole implementare una struttura dati per rappresentare una matrice $n \times n$ di cosiddette *caselle*. Ogni casella é collegata con la casella immediatamente alla destra e con quella immediatamente sottostante, con eccezione naturalmente delle caselle al bordo che possono non avere il vicino di destra, quello di sotto oppure, nel caso della casella in basso a destra, nessuno dei due. Le caselle possono essere *aperte* o *chiuse*. In figura viene mostrata una possibile matrice $5 \times 5$, con le caselle aperte contrassegnate con il colore bianco e le chiuse con il colore nero.



- Descrivere il tipo di dato `casella` in linguaggio PASCAL;
- Descrivere la struttura dati `matrice` in linguaggio PASCAL atta a rappresentare una matrice di caselle di dimensione $n \times n$, dove $n$ é una costante predefinita;
- Scrivere una funzione booleana ricorsiva `trova` che restituisca `true` nel caso in `matrice` esista un cammino fatto di caselle aperte che conduca dalla casella di posizione $(0, n)$ (in alto a sinistra) alla posizione $(n, 0)$ (in basso a destra) e `false` altrimenti.

**Soluzione.** Per quanto riguarda le strutture dati:

```
var
   casella = (aperto, chiuso);
   matrice:  array [1..n, 1..n] of casella;
```

Per quanto riguarda l'algoritmo l'idea della soluzione é data dal seguente schema ricorsivo il cui scopo é quello di rispondere alla domanda *"Esiste un cammino dalla casella corrente a quella finale?"*:

**se** *la casella corrente é aperta* **allora**
  **se** *la casella corrente é quella finale* **allora** *il cammino esiste*
  **altrimenti** *il cammino esiste se e solo se*
  *esso esiste dalla casella di sotto oppure dalla casella a destra;*
**altrimenti** *il cammino non esiste.*

Nel tradurre questa idea in un programma adotteremo la convenzione che se $(i, j)$ non é contenuta all'interno della matrice, la casella (fittizia) $(i, j)$ é da considerarsi chiusa. Assumendo l'esistenza della una variabile globale `matrice` e ricordando che la dimensione della stessa é data dalla costante predefinita `n`, una possibile implementazione é la seguente,

```
function find(i, j:  integer):  boolean;
var inside, open, onSouthEastCorner :  boolean;
begin
  onSouthEastCorner := (i = n) and (j = 0);
  inside := ((0 =< i) and (i =< n) and (0 =< j) and (j =< n));
  if inside then open := (matrice[i,j] = aperto) else open := false;
  if open then
    if onSouthEastCorner then find := true
    else find := find(i+1,j) or find(i,j-1);
  else find := false;
end;
```

Notare come, anche grazie alla definizione delle variabili booleane `onSouthEastCorner`, `inside` e `open`, il programma é praticamente auto-documentante. `onSouthEastCorner` é vera se e solo se la casella corrente é quella finale (cioé $(n, 0)$); `inside` é vera se e solo se siamo all'interno della matrice, essendo possibile, come discusso, che le coordinate correnti $(i, j)$ si riferiscano ad una casella che non vi appartiene; infine, `open` é vera se e solo se la casella corrente é aperta (si ricordi che caselle fittizie, al di fuori della matrice, sono chiuse per definizione, una convenzione che semplifica la programmazione). La sua inizializzazione tramite un `if` é resa necessaria dal fatto che $(i, j)$ potrebbe essere al di fuori della matrice e pertanto il tentato accesso a `matrice[i,j]` darebbe errore.

Questa soluzione, concettualmente semplice, ha peró l'inconveniente di rivisitare inutilmente porzioni di matrice giá visitate ed infatti impiega tempo esponenziale! Come utile esercizio si apportino semplici modifiche alle strutture dati e all'algoritmo in modo da ottenere una soluzione il cui tempo di calcolo é proporzionale al numero di archi della matrice, cioé $n^2$. Un altro utile esercizio consiste nel modificare il programma in modo da fargli restituire (o stampare) l'intero cammino. Infine, si modifichi la soluzione in modo che `matrice` sia un parametro di ingresso della funzione; quali sono i pro ed i contro nell'avere `matrice` come parametro piuttosto che come variabile globale?