Prova scritta di Algoritmi e Strutture Dati (1^o anno)

Luglio '99

NOTE:

- L'ordine in cui vengono svolti gli esercizi non è rilevante (in altre parole: se avete problemi a svolgerne uno potete passare tranquillamente ai successivi e farlo dopo).
- I punti previsti per ogni esercizio si riferiscono ad uno svolgimento completamente corretto.

NOME:

COGNOME:

MATRICOLA:

==

Non scrivere qui sotto

Esercizio	Punti previsti	Punti assegnati
1	3	
2	3	
3	6	
4	8	
5	6	
6	4	
Totale	30	

Esercizio 1 (punti 3)

Consideriamo il seguente codice C che lavora su matrici 2×2:

```
#include <stdio.h>
#define  DIM 2

float A[DIM][DIM], B[DIM][DIM], C[DIM][DIM];

void ttt(float X[DIM][DIM], float Y[DIM][DIM]);
void ppp(float X[DIM][DIM], float Y[DIM][DIM], float Z[DIM][DIM]);

main()
{
   int i,j;
   for ( i=0 ; i<DIM ; i++ )
      for ( j=0 ; j<DIM ; j++)
         scanf("%f",&A[i][j]);

   ttt(A,B);
   ppp(A,B,C);

   for ( i=0 ; i<DIM; i++ )
   {
      for ( j=0 ; j<DIM; j++) printf( "%f ", C[i][j] );
      printf("\n");
   }
}

void ttt(float X[DIM][DIM], float Y[DIM][DIM])
{  int i,j;
   for ( i=0 ; i<DIM; i++ )
      for ( j=0 ; j<DIM; j++)
         Y[i][j] = X[j][i];
}

void ppp(float X[DIM][DIM], float Y[DIM][DIM], float Z[DIM][DIM])
{  int i,j,k;
   for ( i=0 ; i<DIM; i++ )
      for ( j=0 ; j<DIM; j++)
      {  Z[i][j] = 0;
         for ( k=0 ; k<DIM ; k++)
            Z[i][j] += X[i][k]*Y[k][j];
      }
}
```

Domande:

1. Descrivere in generale cosa fa la funzione ttt
2. Descrivere in generale cosa fa la funzione ppp
3. Descrivere in generale cosa fa il programma
4. Dire qual'è l'output per il seguente input 1.0 2.0 4.0 0.0

Esercizio 2 (punti 3)

Consideriamo il seguente codice C:

```
#include <stdio.h>

int a = 10;

void ppp (int * x, int * y);

int fff (int x, int y);

main()
{
  int a = 4;
  int b = 2;
  int c;
  c = fff(a,b);
  printf ("%d, %d, %d\n", a, b, c);
  ppp(&a, &b);
  printf ("%d, %d\n", a, b);
}

void ppp (int * x, int * y)
{ int a, b;
  a = (*x) * (*y);
  b = a + *y;
  *x = a * b;
}

int fff (int x, int y)
{ y *= a;
  return( x + y );
}
```

Domanda: qual'è l'output stampato dalle printf?

Esercizio 3 (punti 6)

Consideriamo il tipo di dato **Successione** di interi implementato mediante liste con puntatori.

Consideriamo l'operazione

$$\text{Purge} : \text{Succ} \times \text{Int} \longrightarrow \text{Succ}$$

Definita (a parole) come segue:

Purge(s,x) elimina da s tutte le occorrenze di x (la successione s viene modificata dall'operazione, non se ne fa una copia).

Domande:

1. Definire in pseudo-codice i tipi utilizzati per implementare le successioni.
2. Definire in pseudo-codice l'implementazione di Purge come procedura.
3. Calcolare la complessità dell'implementazione fatta per Purge in funzione della lunghezza n di s, in $\Theta()$.

Nel calcolo di complessità: non fare conti troppo dettagliati esplicitando tutte le costanti, ma non limitarsi a neppure a dare solo il risultato: bisogna spiegare come ci si arriva.

Esercizio 4 (punti 8)

Consideriamo il tipo di dato *Insieme* implementato con una tabella hash chiusa, utilizzato per contenere dati del seguente tipo (definito da una struttura C):

```
typedef struct stud {
                      int matricola;
                      char nome[20];
                    } Studente;
```

Si assume quanto segue:

- la tabella ha dimensione N = 10.000
- le matricole possono assumere valori tra 1 e 100.000
- viene fornita la seguente funzione di hashing:

  ```
  int h(s: Studente)
  {
       return( (int)(s.matricola * s.matricola / 997) % N );
  }
  ```

Domande:

1. Definire in pseudo-codice (o in C) il tipo utilizzato per l'implementazione del tipo di dato;
2. Definire in pseudo-codice (o in C) una funzione di rehashing necessaria per trattare il caso in cui un posto della tabella sia già occupato;
3. Dire come viene inizializzata una tabella vuota e che valore si mette in tabella quando un elemento viene cancellato (non si chiede di implementare le operazioni di inizializzazione e cancellazione, basta dire che effetto hanno sulla tabella);
4. Definire in pseudo-codice una primitiva che permette di cercare uno studente per matricola (il suo comportamento è simile a quello della **Member**): la primitiva prende in input un numero di matricola e deve restituire un valore Booleano (come valore della funzione o come parametro OUT) che indica se uno studente con quella matricola è in tabella o no e, in caso affermativo, deve restituirne il nome (come parametro OUT).
5. Valutare la complessità dell'algoritmo, in $\Theta()$, in funzione della dimensione N della tabella oppure del numero di studenti n presenti in tabella e dire se esiste un caso peggiore (N.B.: solo uno dei due parametri è rilevante, bisogna capire quale).

6. Dire, in breve, come cambia l'algoritmo se la chiave di ricerca è il nome (invece della matricola) senza cambiare le ipotesi del problema (non si chiede di dare l'implementazione, solo il concetto).

7. Dire quale sarebbe la complessità in quest'ultimo caso in funzione di N oppure di n e se esiste un caso peggiore (N.B.: solo uno dei due parametri è rilevante, bisogna capire quale).

Esercizio 5 (punti 6)

Consideriamo il seguente programma in pseudo-codice:

```
procedure ppp( v : array[1..n] of integer; k : integer ) /* k deve essere <=n */
{
    j, s : integer
    s ← 0
    per j = 1,...,k:
        s ← s + v[j]
    s ← s / k
}

main()
{
    aa : array[1..n] of integer
    i, j : integer
    i ← n
    while i ≥ 1 do
    {
        ppp( aa , i )
        i−−
    }
}
```

Domanda: determinare la complessità della procedura in funzione della dimensione n dell'array a.

Non fare conti troppo dettagliati esplicitando tutte le costanti, ma considerare il costo di tutte le parti di programma mettendo eventualmente in evidenza le operazioni dominanti (una volta che si sono individuate si possono ignorare le altre). Non limitarsi a dare solo il risultato: bisogna spiegare come ci si arriva.

Esercizio 7 (punti 4)

Consideriamo alberi binari con etichette a valori interi, implementati con puntatori: ogni nodo contiene un campo info di tipo intero e due campi sx e dx di tipo albero. Consideriamo la procedura seguente:

```
procedure vis( t : albero )
{
    if t non è vuoto then
    {
        scrivi(t−>info)
        if (t−>info è pari) then vis(t−>sx) else vis(t−>dx)
    }
}
```

Domanda: stimare la complessità della procedura in $\Theta()$ in funzione del numero dei nodi n dell'albero t nell'ipotesi che t sia **perfettamente bilanciato**.