

NOME:

COGNOME:

MATRICOLA:

[1] Punti 2 – Domanda a risposte multiple (sono possibili 0 o più risposte)

Un identificatore che ha un collegamento esterno:

- a. È accessibile anche da file diversi da quelli in cui è stata definito
- b. È visibile anche da file diversi da quelli in cui è stata definito
- c. In un progetto può essere definito in un solo file

[2] Punti 3 – Domanda a risposta aperta

Cosa stampa il seguente programma: _____20 6 10 30 _____

```
#include <iostream>
using namespace std ;

void f(int i, int *p, int &ri, int *&rp) {
    int *q = new int ;
    i = 10 ;
    p = q ;
    *p = i++ ;

    ri = 10 ;
    *rp = 30;
}

int main() {
    int a = 20, b=6, c=12, d=8;
    int *punt;
    punt = &b;
    int *prp;
    prp = &d;
    f(a, punt, c, prp);
    cout << a << " " << *punt << " " << c << " " << *prp << " "
    << endl;}
```

[3] Punti 3 – Domanda a risposta aperta

Cosa stampa il seguente programma: _____2_____

```
#include <iostream>
using namespace std ;

int main()
{
    cout<< funz(22,16)<<endl ;
    return 0 ;
}

int funz(int a, int b)
{
    if (b == 0) return a ;
    return funz(b, a%b);}
```

NOME:

COGNOME:

MATRICOLA:

[4] Punti 3 – Domanda a risposta aperta

Cosa stampa il seguente programma: _____ 1 1 2 3 _____

```
#include <iostream>
using namespace std ;

int main()  {

    int v[4]={1,2,3,4};
    int *p = v+1;
    int i;

    for(i=0;i<3;i++){
        (*p)--;
        p++;
    }
    for(i=0;i<4;i++){
        cout<<v[i]<< " ";
    }
}
```

[5] Punti 4 - Scrittura di codice

Un picco di una sequenza S di valori è un elemento strettamente maggiore dei suoi due elementi contigui o maggiore dell'elemento contiguo, se l'altro manca.

Esempio: Nella sequenza [4, 3, 3, 3, 0, -1, 3, -3, 4, 2], 4, 3 e 4 sono picchi.

Data una sequenza di valori interi memorizzati in una lista doppia dichiarata sotto, si scriva la *funzione ricorsiva* `int picchi(lista)` che restituisce il numero di picchi contenuti nella lista.

Esempio: Sulla sequenza [4, 3, 3, 3, 0, -1, 3, -3, 4, 2] deve restituire 3.

```
struct elem
{
    int inf;
    elem* pun ;
    elem* prev;
} ;

typedef elem* lista ;
```

NOME:

COGNOME:

MATRICOLA:

```

int picchi (lista l) {
    if (l==NULL)
        return 0;
    if ((l->prev==NULL&& l->pun->inf<l->inf) || (l->pun==NULL&& l->prev->inf<l->inf))
        return 1+picchi(l->pun);
    else
        if (l->prev->inf<l->inf&& l->pun->inf<l->inf)
            return 1+ picchi(l->pun);
        else
            return picchi(l->pun);
}

```

[6] Punti 17 - Scrittura di codice

Dato la segue dichiarazione

```

struct persona{
    char* nome;
    int id;
};

```

a. Punti 3

Si scriva la funzione `int compare(persona, persona)` che implementa la seguente relazione d'ordine:

`compare(e1,e2)=0` se nome e id coincidono

`compare (e1,e2)<0` se il nome di e2 precede il nome di e2 e oppure coincidono e l'id di e1 è minore dell'id di e2

`compare(e1,e2)>0` altrimenti

```

int compare(persona p1, persona p2) {
    if ((strcmp(p1.nome,p2.nome)==0 && p1.id==p2.id) ||
        strcmp(p1.nome,p2.nome)<0 || strcmp(p1.nome,p2.nome)>0)
        return strcmp(p1.nome,p2.nome);
    return p1.id-p2.id;
}

```

NOME:

COGNOME:

MATRICOLA:

b. Punti 2 – Scrittura di codice

Si assuma una lista ordinata di elementi di tipo `persona`. Scrivere il tipo di dato `elem` (elemento della lista) e il tipo di dato `lista` (puntatore alla testa della lista)

```
struct elem{
    persona info;
    elem* next;
};

typedef elem* lista;
```

c. Punti 4

Scrivere la *primitiva* `lista ord_insert_elem(lista, elem*)` che aggiunge un elemento ad una lista ordinata. La funzione deve usare la funzione `compare` (punto a) e le primitive `head` e `tail`.

```
lista ord_insert_elem(lista l, elem* e){
    if(l==NULL || compare(e->inf, head(l))<0){
        e->pun = l;
        return e;
    }
    lista l1=l;
    while (tail(l1)!=NULL && compare(e->inf, head(tail(l1)))>0)
        l1=tail(l1);
    e->pun = l1->pun;
    l1->pun = e;
    return l;}
}
```

d. Punti 4

Scrivere la *procedura ricorsiva* `diff(lista l1, lista l2)` che stampa l'elenco delle persone che sono presenti nella lista ordinata `l1` ma non nella lista ordinata `l2`. La procedura deve sfruttare l'ordinamento implementando un algoritmo con complessità $O(n)$.

```
void diff(lista l1, lista l2){
    if(l2==NULL){
        while(l1!=NULL){
            cout<<l1->info.nome<<" "<<l1->info.id<<endl;
            l1=l1->pun;}
        return;}
    if(compare(l1->info, l2->info)==0){
        diff(tail(l1), tail(l2));
        return;}
    }
    if(compare(l1->info, l2->info)<0){
        cout<<l1->info.nome<<" "<<l1->info.id<<endl;
        diff(tail(l1), l2);
        return;}
    diff(l1, tail(l2));}
```

NOME:

COGNOME:

MATRICOLA:

e. Punti 4

Dato la seguente dichiarazione per un binary search tree (BST)

```
struct bnode {
    persona key;
    bnode* left;
    bnode* right;
    bnode* parent;
};

typedef bnode* bst;
bst get_left(bst); //restituisce il sottoalbero sinistro

bst get_right(bst); //restituisce il sottoalbero destro
```

Scrivere la procedura `stampa_inv(bst)` che stampa l'elenco delle persone in ordine inverso (dal più grande al più piccolo). La procedura deve usare le primitive sopra definite.

```
void stampa_inv(bst b) {
    if (get_right(b) != NULL)
        stampa_inv(get_right(b));
    cout<<b->key.nome<<" "<<b->key.id<<endl;
    if (get_left(b) != NULL)
        stampa_inv(get_left(b));
}
```