```c
#include <stdlib.h>
#include <stdio.h>

/*
Dato un albero e due nodi dell'albero, restituire il minimo antenato comune (ma
c) dei due nodi.
Il mac è, tra gli antenati dei due nodi, quello che si trova il più lontano pos
sibile dalla radice.
Considero un albero con chiavi dai valori distinti
Tratto da questa esercitazione di Java:
http://www.dis.uniroma1.it/~bordino/esercitazioneBST.html
http://www.dis.uniroma1.it/~bordino/SoluzioneParte3.java
*/
typedef struct node{
    int key;
    struct node * left;
    struct node * right;
} * Node;

Node mac(Node bt, Node u, Node v) {
    // albero vuoto
    if(bt == NULL)
        return bt;
    // scorro l'albero con un albero ausiliario
    Node aux = bt;
    // mi allontano dalla radice
    while (aux != NULL) {
        // fino a che il nodo corrente è minore delle due chiavi cerco l'antena
to a dx
        if (aux -> key < u -> key && aux -> key < v -> key)
            aux = aux -> right;
        // fino a che il nodo corrente è maggiore delle due chiavi cerco l'ante
nato a sx
        else if (aux -> key > u -> key && aux -> key > v -> key)
            aux = aux -> left;
        // quando il nodo corrente risulta essere più grande di una chiave e pi
ù piccolo di un'altra
        else {
            return aux; // ho trovato l'antenato
        }
    }
    //return NULL;
}

// versione ricorsiva
Node mac_rec(Node bt, Node u, Node v) {
    // albero vuoto
    if(bt == NULL)
        return bt;
    // mi allontano dalla radice
    if (bt -> key < u -> key && bt -> key < v -> key)
        return mac_rec(bt -> right, u, v);
    // fino a che il nodo corrente è maggiore delle due chiavi cerco l'antenato
 a sx
    else if (bt -> key > u -> key && bt -> key > v -> key)
        return mac_rec(bt -> left, u, v);
    // quando il nodo corrente risulta essere più grande di una chiave e più pi
ccolo di un'altra
    else {
        return bt; // ho trovato l'antenato
    }
}
```

```c
#include <stdlib.h>
/*
Funzioni iterative minimo e successore
*/

typedef struct node{
    int key;
    struct node * p;
    struct node * left;
    struct node * right;
} * Node;

// ricerca iterativa del minimo
Node min(Node u) {
    // se è vuoto non cercare
    if(u == NULL)
        return u;
    // fino a che il nodo ha figli a sx, vai a sx, uso un albero ausiliario
    Node aux = u;
    while((aux -> left) != NULL) {
        aux = aux -> left;
    }
    // torna il nodo che non ha più figli a sx, ovvero il nodo più a sx
    return aux;
}

// ricerca successore
Node succ(Node u) {
    Node aux;
    // se è vuoto non cercare
    if(u == NULL)
        return u;
    // se il nodo non ha un figlio dx
    if(u -> right == NULL) {
        // il successore è l'antenato più a dx, scorri all'insù fino a che non
svolti a dx
        // fino a che si trova un padre che ha un nodo come figlio sx
        aux = u;
        // la condizione del padre diverso da NULL va posta prima,
        // si verifica solo quando si sta cercando un successivo a partire dal
nodo massimo
        while(aux -> p != NULL && (aux != aux -> p -> left))
            aux = aux -> p;
        return aux -> p;
    }
    // se invece ha un figlio dx, torna il minimo del sottoalbero dx
    return min(u -> right);
}
```

```c
#include <stdlib.h>
/*
Dato un albero di ricerca t, scrivere una funzione EFFICIENTE che
restituisca il numero massimo di ripetizioni di una chiave in t e
analizzarne la complessita'.

NON si possono usare strutture ausiliarie di dimensione theta(n) dove n e'
il numero dei nodi dell'albero.

Il prototipo della funzione e`:
int massimorip(Tree t)
*/

int massimorip(Node t) {
    // se è vuoto ha 0 ripetizioni
    if(t == NULL)
        return 0;

    // nodo corrente
    Node cur;
    // chiave del nodo corrente k, contatore del valore corrente cont, numero d
i occorrenze massimo max
    int k, cont, max;

    // parti dal nodo più piccolo cur, inizializza cont e max a 1
    cur = min(t);
    max = cont = 1;
    k = cur -> key;
    cur = succ(cur);
    // continua a scorrere fino al nodo massimo (quello senza successori)
    while (cur != NULL){
        // se la chiave del nodo è diversa da quella del nodo precedente k, inc
rementa cont
        if (k == cur -> key)
            cont++;
        else { // se è cambiata, se il conteggio è il più alto finora mai regis
trato, salvalo in max
            if (cont > max)
                max = cont;
            k = cur -> key; // sostituisci la vecchia chiave con la corrente
            cont = 1; // e setta a 1 il contatore
        }
        cur = succ(cur);
    }
    // potrei aver incrementato il contatore nell'ultimo ciclo
    if (cont > max)
        max = cont; // nel caso aggiorno max

    return max;
}

int main() {
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

/*
Scrivere una funzione check che dato un albero binario di ricerca verifica se e
' soddisfatta la seguente condizione:
per ogni intero k, se le chiavi k e k+2 sono nell'albero allora anche la chiave
 k+1 è nell'albero.
Il prototipo della funzione è:
int check(Node u)
Qual è la complessità in tempo della funzione?
*/

// perché valga la condizione ogni nodo con chiave k non può mai avere per succ
essore un nodo con chiave k + 2
// altrimenti, essendo un albero di ricerca, non ci sarà per forza di cose un n
odo con chiave k + 1
// al primo nodo che non rispetta questa condizione esco tornando 0

// Devo scorrere l'albero in modo iterativo, servono le funzioni massimo e mini
mo
// Per implementare la funzione successore l'albero deve avere il nodo padre!

int check(Node u) {
    // se l'alebro è vuoto torno 1 (va bene anche 0)
    if (u == NULL)
        return 1;

    // nodo corrente e suo successore
    Node cur, next;
    int checked;

    // parto dal minimo, nodo in basso a sx
    cur = min(u);
    next = succ(cur);

    checked = 1;
    // se il successore non è NULL, cioè l'albero non è ancora finito
    // e se non si è ancora trovata una sequenza di nodi che non rispetta la co
ndizione checked
    while (next != NULL && checked){
        // se il successore è il corrente + 2
        if (cur -> key + 2 == next -> key)
            checked = 0; // farà uscire il while
        // altrimenti il nodo corrente diventa il suo successore
        // e il successore diventa il succ(successore)
        else {
            cur = next;
            next = succ(next);
        }
    }
    return checked;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

/*
Sia T un albero generale i cui nodi hanno chiavi intere.
Scrivere una procedura RICORSIVA che trasforma T dimezzando i valori di tutte l
e chiavi sui livelli pari dell'albero.
Si deve utilizzare il seguente tipo per la rappresentazione di un albero binari
o:
*/

typedef struct node{
    int key;
    struct node * left;
    struct node * right;
} * Node;

void dimezza_pari2(Node u, int pari) {
    // caso base albero vuoto
    if(u == NULL)
        return;
    // se siamo ad un livello pari dimezza la chiave
    if(pari) {
        u -> key = u -> key / 2;
        // passo alle chiamate successive un livello dispari
        pari = 0;
    }
    // altrimenti passo un livello pari
    else
        pari = 1;
    dimezza_pari2(u -> left, pari);
    dimezza_pari2(u -> right, pari);
}

void dimezza_pari(Node u) {
    // il primo livello è 0, quindi pari, passo il valore 1
    dimezza_pari2(u, 1);
}
```

```c
#include <stdlib.h>
#include <stdio.h>

/*
Dato un albero binario T  scrivere una funzione efficiente in C che restituisca
  1 se per ogni nodo u di T
vale la seguente proprietà: il sottoalbero sinistro di  u ha una dimensione alm
eno doppia di quella del
sottoalbero destro di u (la dimensione è il numero di nodi  in esso contenuti),
 0 altrimenti.
Inoltre:
a. discutere la complessità della soluzione trovata.
b. dimostrare la correttezza della soluzione proposta (facoltativo).

Si deve utilizzare il seguente tipo per la rappresentazione di un albero binari
o:
*/
typedef struct node{
    int key;
    struct node * left;
    struct node * right;
} * Node;

int dim_doppia2(Node u, int * dim) {
    int dimsx, dimdx, doppiasx, doppiadx;
    // caso base albero vuoto, la dimensione dei sottoalberi è 0, vale la dimen
sione doppia
    if(u == NULL) {
        *dim = 0;
        return 1;
    }
    // per ricorsione ottengo se vale la dimensione doppia per i sottoalberi e
la loro dimensione
    doppiasx = dim_doppia2(u -> left, &dimsx);
    doppiadx = dim_doppia2(u -> right, &dimdx);
    // la dimensione dell'albero è data da 1 + la dimensione dei suoi sottoalbe
ri
    *dim = 1 + dimsx + dimdx;
    // alla radice verifico se per entrambi i sottoalberi vale la dimensione do
ppia e se vale anche nel nodo radice
    if(doppiasx && doppiadx && (dimsx >= 2 * dimdx))
        return 1;
    else return 0;
}

int dim_doppia(Node u) {
    // a priori non conosco la dimensione dei sottoalberi
    int dim;
    return dim_doppia2(u, &dim);
}
```

```c
#include <stdlib.h>
#include <stdio.h>

/*
Un nodo di un albero binario è detto pari se il numero di foglie del sottoalber
o di cui è radice è pari.
a. Progettare un algoritmo efficiente che dato un albero binario restituisca il
 numero di
nodi pari.
b. Discutere brevemente la complessità della soluzione trovata.
La rappresentazione dell'albero binario utilizza esclusivamente i campi left, r
ight e key.
*/
typedef struct node{
    int key;
    struct node * left;
    struct node * right;
} * Node;

int nodi_pari2(Node u, int * foglie) {
    int fsx, fdx, parisx, paridx;
    // caso base albero vuoto, non ha foglie, non è pari
    if(u == NULL) {
        *foglie = 0;
        return 0;
    }
    // per ricorsione ottengo il numero di nodi pari dei sottoalberi ed il loro
 numero di foglie
    parisx = nodi_pari2(u -> left, &fsx);
    paridx = nodi_pari2(u -> right, &fdx);
    // il numero di foglie di un albero è dato dalla somma delle foglie dei sot
toalberi
    (*foglie) = fsx + fdx;
    // alla radice verifico se il nodo stesso è un nodo foglia
    if(u -> left == NULL && u -> right == NULL)
        *foglie = *foglie + 1;
    // poi verifico se anche la radice è un nodo pari, cioè se ha un numero di
foglie pari
    if(((*foglie) % 2) == 0)
        return 1 + parisx + paridx;
    else // altrimenti conto solo i nodi pari dei sottoalberi
        return parisx + paridx;
}

int nodi_pari(Node u) {
    // a priori non conosciamo il numero di foglie dei sottoalberi
    int f;
    return nodi_pari2(u, &f);
}
```

```c
#include <stdlib.h>
#include <stdio.h>

/*
Un nodo di un albero binario è detto centrale se il numero di foglie del sottoa
lbero di cui è radice è pari
alla somma delle chiavi dei nodi appartementi al percorso dalla radice al nodo
stesso.
a. Scrivere una funzione efficiente in C che restituisca il numero di nodi cent
rali.
b. Discutere la complessità della soluzione trovata.
c. Se vogliamo modificare la funzione in modo che restituisca l'insieme dei nod
i centrali che tipo
di struttura dati si può utilizzare per rappresentare l'insieme?
        La complessità dell'algoritmo deve rimanere la stessa che nel caso (a).


Si deve utilizzare il seguente tipo per la rappresentazione di un albero binari
o:
*/

typedef struct node{
    int key;
    struct node * left;
    struct node * right;
} * Node;

int centrali2(Node r, int sum, int *numFoglie) {
    int fsx, fdx, csx, cdx;
    // se l'albero è vuoto ha 0  foglie e non ha nodi centrali
    if(r == NULL) {
        *numFoglie = 0;
        return 0;
    }
    // altrimenti incrementa la somma
    sum = sum + r -> key;

    // calcola ricorsivamente quanti nodi centrali hanno i sottoalberi e il num
ero delle loro foglie
    // ai sottoalberi passo il valore attuale della somma
    csx = centrali2(r -> left, sum, &fsx);
    cdx = centrali2(r -> right, sum, &fdx);

    // il numero di foglie di un albero è dato dalla somma delle foglie dei sot
toalberi
    (*numFoglie) = fsx + fdx;

    // se la radice stessa è una foglia aggiungo il suo valore
    if(r -> left == NULL && r -> right == NULL) {
        *numFoglie = *numFoglie + 1;
    }

    // se è un nodo centrale lo aggiungo al risultato
    if(*numFoglie == sum) {
        return 1 + csx + cdx;
    }
    // altrimenti torno i nodi centrali calcolati nei sottoalberi
    return csx + cdx;
}

int centrali(Node r) {
    // non conosciamo a priori il numero di foglie
    int nf;
    // ma sappiamo che la somma dei nodi è 0
    int sum = 0;
    return centrali2(r, sum, &nf);
}
```

```c
#include <stdlib.h>
#include <stdio.h>

/*
Un albero binario si dice  t bilanciato se per ogni suo nodo vale la proprietà:
 le altezze dei sottoalberi
radicati nei suoi due figli differiscono per al più t unità.
a. Dato un albero binario, scrivere una funzione  efficiente in C che restituis
ca il minimo valore  t
per cui l'albero risulti t bilanciato.
b.  Discutere la complessità della soluzione trovata.

Si deve utilizzare il seguente tipo per la rappresentazione di un albero binari
o:
*/

typedef struct node{
    int key;
    struct node * left;
    struct node * right;
} * Node;

// funzione max non definita in math.h, dati due numeri torna il maggiore dei d
ue
int max(int x, int y) {
    if(x >= y)
        return x;
    return y;
}
// funzione assoluto = abs di math, nel caso la libreria non fosse caricata
int assoluto(int x) {
    if(x < 0)
        return -x;
    return x;
}

int tBil(Node u, int *h) {
    // sbilanciamento dell'albero, sbilanciamento dei due sottoalberi e loro al
tezze
    int sbil, sbil_left, sbil_right, h_left, h_right;
    // caso base, albero vuoto, torno sbilanciamento = 0 e h = -1 perché l'alte
zza al primo livello parte da 0
    if(u == NULL) {
        *h = -1;
        return 0;
    }
    // per ricorsione ottengo l'altezza dei due sottoalberi e il loro sbilancia
mento
    sbil_left = tBil(u -> left, &h_left);
    sbil_right = tBil(u -> right, &h_right);
    // l'altezza al nodo radice è data dall'altezza massima delle altezze dei s
ottoalberi + 1
    *h = max(h_left, h_right) + 1;
    // lo sbilanciamento alla radice è la differenza in valore assoluto delle a
ltezze dei sottoalberi
    sbil = assoluto(h_left - h_right);
    // lo sbilanciamento minimo t è il massimo tra lo sbilanciamento alla radic
e e lo sbilanciamento dei sottoalberi
    return max(sbil, max(sbil_left, sbil_right));
}

int t_bilanciato(Node u) {
    // non conosciamo a priori l'altezza h dell'albero
    int h;
    return tBil(u, &h);
}
```

```c
#include <stdio.h>
#include <stdlib.h>

/*
Dato un nodo u, sia pu la sua profondita' e hu l'altezza di T(u). Diciamo che u
 è un nodo CARDINE se e solo se pu = hu.
Scrivere una funzione RICORSIVA che restituisce la lista dei nodi cardine prese
nti in un albero binario.
La complessità in tempo deve essere O(n) dove n numero dei nodi dell'albero.
Si deve utilizzare il seguente tipo per la rappresentazione di un albero binari
o:
*/

typedef struct node{
    int key;
    struct node * left;
    struct node * right;
} * Node;

// funzione ausiliaria max, torna il max tra due interi
int nmax(int x, int y) {
    if(x >= y)
        return x;
    return y;
}

int cardine2(Node u, int prof, int * h) {
    // altezza dei sottoalberi e il loro numero di nodi cardine
    int hsx, hdx, cardsx, carddx;
    // caso base albero vuoto, l'altezza è -1, non ci sono nodi cardine
    if(u == NULL) {
        *h = -1;
        return 0;
    }
    // sono su un nuovo livello, incremento la profondità
    prof++;
    // calcolo per ricorsione le altezze dei sottoalberi sx e dx, a cui passo l
a profondità appena calcolata
    // ottengo anche il loro numero di nodi cardine
    cardsx = cardine2(u -> left, prof, &hsx);
    carddx = cardine2(u -> right, prof, &hdx);
    // nel nodo radice calcolo l'altezza facendo 1 + l'altezza massima tra quel
le dei sottoalberi
    *h = 1 + nmax(hsx, hdx);
    // sommo ai nodi cardine dei sottoalberi anche il nodo radice, nel caso sia
 anch'esso un nodo cardine
    if(prof == *h)
        return 1 + cardsx + carddx;
    return cardsx + carddx;
}

int cardine(Node u) {
    // non conosciamo l'altezza dei sottoalberi ma sappiamo che la profondità p
arte da -1
    int prof = -1;
    int h;
    return cardine2(u, prof, &h);
}
```

```c
#include <stdlib.h>
#include <stdio.h>

// inserisce un elemento k in un array ordinato a di dimensioni n mantenendo l'
ordine
void insert_in_order(int a[], int n, int k) {
    int i, pos;
    // cerco la posizione dall'ultimo elemento
    // così nel caso migliore inserisco k alla fine senza spostare nessun eleme
nto
    pos = n;

    while(a[pos - 1] > k && pos > 0) // confronta al max fino al primo elemento
 a[1 - 1]
        pos--;

    // trovata la posizione sposto tutti gli elementi dall'ultimo alla posizion
e trovata compresa
    for(i = n; i > pos; i--)
        a[i] = a[i - 1];
        // l'ultimo elemento spostato è a[pos + 1] = a[pos]

    // inserisco k nello spazio libero lasciato in pos
    a[pos] = k;
}
void insertion_sort(int a[], int n) {
    // per ogni elemento di indice i a partire dal secondo,
    // inserisco quell'elemento negli elementi che lo precedono (array di dimen
sione i)
    int i;
    for(i = 1; i < n; i ++) {
        insert_in_order(a, i, a[i]);
    }
}
```

```c
#include <stdlib.h>
#include <stdio.h>

void print_a(int a[], int n);

/*
merge, dati due array ordinati a1, a2 di dimensioni n1 e n2 uniscili in modo or
dinato in un terzo array dest
*/
void merge(int a1[], int a2[], int n1, int n2) {
    // array di appoggio in cui mettere i valori in ordine
    int * aux = calloc((n1 + n2), sizeof(int)); // l'array ha dimensione n, (m
+ n - m)
    // indici
    int i, j, pos1, pos2;
    i = pos1 = pos2 = 0;
    // fino a che entrambi gli array hanno almeno un elemento confrontali
    while(pos1 < n1 && pos2 < n2) {
        // se il primo elemento di a2 è < del primo di a1 inserici quello
        if(a2[pos2] < a1[pos1])
            aux[i++] = a2[pos2++]; // usa le variabili poi incrementale con ++
        else // altrimenti fai il contrario
            aux[i++] = a1[pos1++];
    }
    // se sono rimasti elementi in a1 (quindi non sono rimasti in a2), continua
 a copiarli nell'array aux
    while(pos1 < n1) {
            aux[i++] = a1[pos1++];
    }
    // se invece rimango elementi in a2 sono già ordinati, non li copio in aux
    // quando ricopio gli elementi da aux all'array da ordinare, copio solo que
lli caricati in aux (da indice 0 ad i)
    for(j = 0; j < i; j++) {
        a1[j] = aux[j];
    }
    free(aux);
}

void merge_sort(int a[], int n) {
    // divide
    int i, m;
    m = n / 2;
    // caso base, l'array con un solo elemento è già ordinato
    if(n <= 1)
        return;
    // impera, riordina i due sotto array
    merge_sort(a, m);
    merge_sort(a + m, n - m);
    // combina, unisci gli array ordinati in un unico array
    // i due array passati a merge sono le due metà di uno stesso array!
    merge(a, a + m, m, n - m);
}
```

```c
/*
Dato un vettore v di n interi, eventualmente ripetuti, ordinati in senso cresce
nte, e un intero k,
definire una funzione occ che conta il numero di occorrenze di k in v.
La complessità della funzione deve essere O(log n).
*/
// Per rispettare la complessità O(log n) si devono usare due funzioni ausiliar
ie
// maxi e mini che calcolano rispettivamente l'indice max e min in cui c'è k
// spendendo il tempo della ricerca binaria (log n)
// dagli indici max e min otteniamo il numero di occorrenze di k

int maxi(int a[], int inf, int sup, int k) {
    int m, ris;
    // caso base array vuoto
    if(inf > sup)
        return -1; // torna un indice impossibile
    // calcola l'indice medio
    m = (sup + inf) / 2;
    // se l'elemento a metà è k
    if(a[m] == k) {
        // verifico se c'è un indice max dopo quell'elemento
        ris = maxi(a, m + 1, sup, k);
        // se c'è torna quello
        if(ris != -1)
            return ris;
        // altrimenti m è il max
        return m;
    }
    // se è più piccolo di k, ricorri a dx
    if(a[m] < k)
        return maxi(a, m + 1, sup, k);
    // se è più grande ricorri a sx
    return maxi(a, inf, m - 1, k);
}

// per mini valgono i commenti di maxi con poche differenze
int mini(int a[], int inf, int sup, int k) {
    int m, ris;
    if(inf > sup)
        return -1;
    m = (sup + inf) / 2;
    if(a[m] == k) {
        // verifico se c'è un indice min prima di quell'elemento
        ris = mini(a, inf, m - 1, k);
        // se c'è torna quello
        if(ris != -1)
            return ris;
        return m;
    }
    if(a[m] < k)
        return mini(a, m + 1, sup, k);
    return mini(a, inf, m - 1, k);
}

int num_occ(int a[], int dim, int k) {
    int max, min;
    max = maxi(a, 0, dim - 1, k);
    // se non c'è un indice massimo non ci sono occorrenze e torno 0
    if(max == -1)
        return 0;
    // altrimenti cerco anche l'indice minimo
    min = mini(a, 0, dim - 1, k);
    // il numero di occorrenze di k è il numero di indici in cui c'è k, max - (
min - 1)
    return max - min + 1;
}
```

```c
#include <stdlib.h>
#include <stdio.h>

/*
Dato un vettore v di numeri interi distinti, se i < j e v[i] > v[j], allora la coppia (i, j) è detta un'inversione di v. Utilizzando la tecnica del divide et impera, scrivere una funzione ricorsiva EFFICIENTE che restituisce il numero di inversioni in v.

Il prototipo della funzione è:
int inversioni(int v[], int inf, int sup)
(soluzione con questa firma:
https://consegne.dsi.unive.it/consegne/ChangePageAction.do?l=4&assignment=72140&ord=3)

Esempio:
In v = <2, 3, 8, 6, 1> ci sono 5 inversioni.

Qual è la complessità in tempo della funzione?

*/

// (soluzione con firma un po' diversa)
// per ottimizzare la complessità l'array viene ordinato, in questo modo è più facile contare le inversioni
// se un elemento della seconda metà è minore di uno della prima sarà minore anche dei successivi elementi della prima e viceversa
// si utilizza una variante del merge sort in cui si contano anche le inversioni
// se si ordinano la prima e la seconda metà il numero di inversioni non cambia

int count_merge(int a1[], int a2[], int n1, int n2) {
    // array di appoggio in cui mettere i valori in ordine
    int * aux = calloc((n1 + n2), sizeof(int));
    // indici
    int i, j, pos1, pos2, count;
    i = pos1 = pos2 = count = 0;
    // fino a che entrambi gli array hanno almeno un elemento confrontali
    while(pos1 < n1 && pos2 < n2) {
        // se il primo elemento di a2 è < del primo di a1 inserici quello
        if(a2[pos2] < a1[pos1]) {
            aux[i++] = a2[pos2++];
            // in questo caso incremento count, devo considerare che
            // anche tutti gli altri elementi rimasti in a1 fanno inversione con l'elemento di a2
            count = count + (n1 - pos1);
        }
        // altrimenti è maggiore, fai il contrario
        else
            aux[i++] = a1[pos1++];
    }
    // se sono rimasti elementi in a1 (quindi non sono rimasti in a2), continua a copiarli nell'array aux
    while(pos1 < n1) {
            aux[i++] = a1[pos1++];
    }
    // se invece rimango elementi in a2 sono già ordinati, non li copio in aux
    // quando ricopio gli elementi da aux all'array da ordinare, copio solo quelli caricati in aux (da indice 0 ad i)
    for(j = 0; j < i; j++) {
        a1[j] = aux[j];
    }
    free(aux);
    return count;
}

int inversioni(int a[], int n) {
```

```
    // divide
    int i, m, invsx, invdx;
    m = n / 2;
    // caso base, l'array con un solo elemento è già ordinato e non ha inversio
ni
    if(n <= 1)
        return 0;
    // impera, riordina i due sotto array e conta il loro numero di inversioni
    invsx = inversioni(a, m);
    invdx = inversioni(a + m, n - m);
    // combina, a partire dalle metà ordinate degli array ordina l'array comple
ssivo
    // somma le inversioni contate nelle metà con quelle ottenute confrontando
le due metà tra loro
    return invsx + invdx + count_merge(a, a + m, m, n - m);
}
```

```c
#include <stdlib.h>
#include <stdio.h>

/*
Scrivere una funzione range RICORSIVA che dato un albero binario di ricerca e due interi k1 e k2 tali che k1 <= k2,
restituisce un vettore contenente tutte le chiavi k ordinate in senso crescente contenute nell'albero tali che k1<= k <= k2.
Il prototipo della funzione è:
int *range(Node u, int k1, int k2, int *dim)
Qual è la complessità in tempo della funzione?
*/
typedef struct node {
    int key;
    struct node * left;
    struct node * right;
} * Node;

// quando alla radice si alloca lo spazio per l'array non si conosce la dimensione degli array dei sottoalberi
// la funzione un puntatore a variabile dim
// al termine della funzione in dim è salvata la dimensione dell'array creato
int * range(Node u, int k1, int k2, int * dim) {
    // array dei sottoalberi sx e dx e loro dimensione, contatore
    int * asx, * adx, dimsx, dimdx, i;
    // caso base albero vuoto, ha dimensione 0 e torna array = NULL
    if(u == NULL) {
        *dim = 0;
        return NULL;
    }
    // caso in cui la radice è un nodo nel range
    if(k1 <= u -> key && u -> key <= k2) {
        // calcolo per ricorsione gli array con i valori nel range nei sottoalberi sx e dx
        // e ottengo le loro dimensioni
        asx = range(u -> left, k1, k2, &dimsx);
        adx = range(u -> right, k1, k2, &dimdx);
        // la dimensione dell'array finale è la somma delle dimensioni dei due array ottenuti + 1 (la radice)
        *dim = dimsx + dimdx + 1;
        // rialloco nell'array sx lo spazio necessario a contenere la radice e l'array dx
        asx = realloc(asx, sizeof(int) * *dim);
        // copio nell'array sx la radice e incremento di 1 la dimsx
        asx[dimsx++] = u -> key;
        // copio nell'array sx gli elementi dell'array dx
        for(i = 0; i < dimdx; i++)
            asx[dimsx + i] = adx[i];
        // torno l'array sx che ora contiene tutti gli elementi
        return asx;
    }
    // nel caso in cui la radice è minore del range ricorri a dx
    if(u -> key < k1)
        return range(u -> right, k1, k2, dim);
    // se invece è maggiore del range ricorri a sx
    return range(u -> left, k1, k2, dim);
}
```