**Problema 1**

L‘algoritmo segue una strategia di Divide et Impera che consiste nel dividere l’array A[] a metà, calcolare il numero di occorrenze di k nel primo sub-array e nel secondo sub-array, e sommare i due valori ottenuti per ottenere il numero di occorrenze di k complessivo. Come caso base, si è scelto quello in cui l’array è composto da un solo elemento: in tal caso l’algoritmo restituirà il valore 1 se l’elemento dell’array coincide con k, e 0 altrimenti.

**Codice C:**

#include <stdio.h>

#include <stdlib.h>

int count(int k, int \*a, int p, int r){

int x, y, q;

if(p == r){

if(a[p] == k) return 1;

else return 0;

}

else{

q = (p+r)/2;

x = count(k,a,p,q);

y = count(k,a,q+1,r);

return x + y;

}

}

int main(void) {

// your code goes here

int ntest, k, n, ris;

int \*a;

scanf("%d",&ntest);

while(ntest > 0){

scanf("%d %d",&k,&n);

a = (int\*)malloc(sizeof(int) \* n);

for(int i = 0; i < n; i++) scanf("%d",&a[i]);

ris = count(k,a,0,n-1);

printf("%d\n",ris);

free(a);

ntest--;

}

return 0;

}

**Input di test:**

3

1 7 1 1 2 2 2 2 3

4 7 1 1 2 2 2 2 3

10 6 10 10 10 5 4 10

**Complessità:**

Calcoliamo la complessità solo della funzione *count()*, trascurando la complessità della lettura dell’input.

Scriviamo la ricorrenza per la funzione ricorsiva:

Applichiamo il metodo dell’esperto:

Ricadiamo nel primo caso del metodo dell’esperto, dunque:

**Problema 2**

L’algoritmo mantiene una somma parziale di tutti gli elementi già inseriti nella soluzione ed aggiunge un elemento X compreso tra l’elemento inserito in precedenza ed S se e solo se

* X è primo
* X + somma\_parziale ≤ S

**Codice C:**

#include <stdio.h>

#include <stdlib.h>

int is\_prime(int n){

for(int i = 2; i < n; i++){

if(n%i == 0) return 0;

}

return 1;

}

void step(int \*v, int i, int s, int n, int p, int partial\_sum){

int x;

if(i == n){ //caso base;

if(partial\_sum == s){

for(int i = 0; i < n; i++) printf("%d ",v[i]);

printf("\n");

}

}

else{

if(i == 0) x = p + 1;

else x = v[i-1] + 1 ;

while(x <= s - partial\_sum){

if(is\_prime(x)){

v[i] = x;

step(v,i+1,s,n,p,partial\_sum + x);

}

x++;

}

}

}

int main(void) {

// your code goes here

int s, n, p;

int \*v;

int ntest = 0;

while(scanf("%d %d %d",&s,&n,&p) != EOF){

printf("CASO DI TEST %d:\n",++ntest);

v = (int\*)malloc(sizeof(int)\*n);

step(v,0,s,n,p,0);

free(v);

}

return 0;

}

**Input di test:**

63 3 5

23 3 2

17 1 5

**Complessità:**

Calcoliamo la complessità dell’algoritmo, al netto della lettura dell’input.

La funzione *is\_prime(x)* ha una complessità pari a , dove con x indichiamo l’intero dato in input alla funzione.

L’algoritmo costruisce incrementalmente tutte le possibili combinazioni di N numeri primi compresi tra S e P la cui somma sia pari ad S, la complessità è dunque fattoriale.