Scritto da Gabriel

Questa guida è scritta per cercare di vedere ogni singolo problema/esercizio, riassumendo le principali tematiche: ***(tutti gli esercizi commentati, dimostrati, PRE/POST). Questi seguono un ordine sparso.***
- operazioni con alberi/liste
- pattern matching in tutte le salse
- array a più dimensioni, torte, matrici, h-fette, v-fette
- esercizi contorti e anche peggio
Il tutto discusso e ragionato, se possibile, in maniera semplice e umanamente comprensibile.
Soluzioni riprese da quelle presenti su MEGA, in molti casi modificate, ampliate, corrette e riscritte/scrritte da zero. Riprendono gli esercizi di quest'anno, appelli di tutti gli ultimi 10 anni, vari compitini.

In fondo al file viene trattata la parte ***teorica, completamente commentata****.*

Note:

- Nella trattazione del file, dove erano presenti esercizi con le code, non essendo da noi state fatte, le sostituisco con nodoE* o nodoF*, due strutture equivalenti nel nome (liste concatenate di nodi costituite da due campi, un nodo* come campo info e un nodoE* come puntatore al nodo successivo).
- La trattazione degli esercizi non è lineare (non strutturato in sezioni), ci sta un po' di tutto, temi con altri temi, coerenti in linea di massima con quanto detto e fatto.
- Se il prof consiglia funzioni ausiliarie, è meglio farle. Dovrebbe essere a scelta se farle o meno, così è secondo lui, ma non si sa mai.
- In tanti casi PRE e POST vengono omesse, per non complicare esercizi già di per sé contorti; vengono comunque commentate o quantomeno ne viene data intuitivamente un'idea.

FAQ:

- Come si fa la correttezza?

Il prof non lo ha mai spiegato, ma segue un'intuizione fattuale. Precondizione, invariante, postcondizione. Nel caso degli invarianti è la stessa roba, inizio, passo+1, fine ciclo. Spiegazione intuitiva con qualche termine formale filèse che ci sta bene.

- Come si scrivono PRE e POST?

Sono cose molto intuitive, basta dare più di un occhio alla funzione.

- Quali sono i casi ricorrenti del prof?

Sicuramente pattern matching in primis, che possono essere contigui (si interrompe quindi ad un certo punto), completi (o tutto o niente), altrimenti semplici match *en passant*. Seguono poi operazioni varie su liste ed alberi, le magiche H-fette e V-fette, il cui unico modo se non lo si visualizza visivamente è vedere la varie funzioni presenti nel file e poche altre situazioni strane.

Esercizi

Array a più dimensioni visti come se ne avessero una (2 problemi descritti qui uno in seguito all'altro)

Sotto viene affrontato un esercizio del pattern matching di un array a due dimensioni ma vista come una sola. Ad esempio, abbiamo:
T=[2, 2, 3, 2, 3, 1, 3, 1, 3, 2] e P=[0, 3, 1, 3], avremo l'array M così composto:
0 0 0 0
0 0 0 0
0 1 0 1
0 0 0 0
0 1 0 1

0 0 1 0
0 1 0 1
0 0 1 0
0 1 0 1
0 0 0 0

Usando delle parole semplici, questo array sarebbe così: 0000 0000 0101…. Ma viene visto così per facilitare secondo il prof (come no). In realtà si nota che viene considerato un pattern matching in questo modo. 4 colonne che sono gli elementi che formano P (quindi dimP) e 10 righe che formano gli elementi di T (quindi dimT). La prima colonna è tutta false perché P[0] non appare in nessun elemento di T. Invece P[1] appare nelle posizioni 2, 4, 6 e 8. Interessa trovare dei match contigui (cioè che si estendano da un certo punto in poi). Viene considerato come al solito fa Filè, il match più lungo di questo tipo.

Per rappresentare i match si usa questa struttura (la tripla, usata raramente dal caro professore).
struct tripla{int inizioT, inizioP, lung; tripla(int a=0, int b=0, int c=0){inizioT=a; inizioP=b;lung=c;}};
Veniamo quindi alla costruzione di M (essendo una matrice, si usano due cicli for). La cosa più difficile qui è capire come valorizzare un elemento dalla rappresentazione data del prof (come sempre confusa). Si noti quindi quanto scritto sopra. Di fatto, si usa dimP per scorrere i vari pezzi, seguendo l'indice i aggiunto all'elemento, ma moltiplicando per j per potersi muovere tra le singole colonne (i per muoversi tra i singoli elementi, in questo caso in orizzontale, riga per riga).

Date queste osservazioni, è possibile costruire un algoritmo (non proprio immediato).

<u>Parte iterativa</u>

```
void computeM(int*T, int*P, int dimT, int dimP, bool*M){
        for(int i=0; i<dimT; i++){
                for(int j=0; j<dimP; j++){
                        if(P[i] == T[j])
                        *(M+j*dimp+i)=true;
                        else
                        *(M+j*dimp+i)=false;
                }
}
```

Essendo booleana, questa viene riempita sulla base delle osservazioni date. Così vengono riempite righe e colonne.
Viene poi chiesto di effettuare il match tra T e P usando M nel modo dato (modo iterativo e ricorsivo).
La funzione ricorsiva è a mio dire più semplice, in quanto ce la si cava con tre funzioni ricorsive, che corrispondono a tre cicli in questo caso, in due funzioni. Di solito il prof segnala se serve o meno una funzione ausiliaria. Nel primo ciclo, si scorre per dimP e siccome si cerca la lunghezza più grande, la seconda istruzione del controllo booleano controlla se abbiamo già trovato un match di lunghezza più grande rispetto al numero di elementi rimanenti da controllare.
Nota: nelle funzioni di match, si ritorna sempre un intero, per poter confrontare nella funzione principale se ci sono stati altri match o meno, come qui sotto.

```
tripla match(bool*M,int dimP, int dimT){
tripla best;        //salva il match migliore
        for(int i=0; i<dimP && (dimP – i) > best.lung; i++){
                for(int j=0; j<dimT; j++){
                        int x=aux(M, i, j, dimP, dimT);
                        if(x > match.lung)
                        best=tripla(j, i, x);
                }
```

```
        }
}
```

Nella funzione ausiliaria, vengono analizzate le righe di dimP secondo il criterio esaminato, verificando se le singole celle di M sono true, più si scorre in contemporanea per colonne (come chiede il prof, in pratica si segue il criterio della palude, ci si sposta e si cerca di capire per righe come muoversi).

```
tripla aux(bool M; int i, int j, int dimT, int dimP){
        int L=0;
        while (i < dimT && M[i+j*dimP))
        L++;

        return L;
}
```

Articoliamo la dimostrazione di correttezza (mai mostrata una volta dal prof come fare a farne una giustamente, ma seguiamo i ragionamenti per induzione). Quindi, assumendo PRE:

(per i=0, j=0) → Si considera la prima posizione utile all'interno dei singoli array. Ci muoviamo come se avessimo due dimensioni, riga e colonna. In queste si va subito a cercare, muovendosi orizzontalmente, la presenza di un match sul primo elemento di P e T usando la funzione ausiliaria aux. In essa si va subito a cercare per ogni elemento di T se esiste muovendosi riga per riga, un corrispondente in P. Se esiste bene, altrimenti si passa al successivo.

(per i>0, j>0) → Si ripetono le stesse osservazioni di prima. Induttivamente, si hanno due casi. Si è trovato un match e nella funzione ausiliaria aux ci si sposta per righe in elementi di colonne adiacenti (da cui l'incremento di i in dimT) e si restituisce un intero che viene salvato o meno a seconda dell'occorrenza oppure o non si trovano match o un match della lunghezza uguale a quella di partenza, che viene scartato. Il ciclo viene quindi mantenuto a meno che non si trovi un match già più lungo degli elementi che stanno ora dentro all'array. In ognuno dei due casi si rispetta POST.
Alcune considerazioni da fare: si usa il principio di induzione (se vale per k, deve valere per k+1 e i successivi). Quindi: PRE → Invariante e/o dim. Induttiva → POST
Non deve essere necessariamente biblica; almeno noi, se il prof non lo è e non lo è mai, cerchiamo di essere chiari.

Ragioniamo ora sulla parte ricorsiva, con firma matchR seguente. In questo caso, il ragionamento rimane analogo. Ragiono prima sui pezzi singoli di dimP, poi sui pezzi di dimT e poi ragiono spostandomi come fatto prima per i pezzi di M nella iterativa. Si noti l'aggiunta di t e p per scorrere gli array.

<u>Parte ricorsiva</u>

```
tripla matchR(bool*M, int p, int t, int dimT, int dimP){
if(dimp==0 || dimP == p)        return tripla();
tripla x=matchR(M, p+1, dimP, t, dimT);
tripla y=aux1(M, P, dimP, T, dimT);
        if(x.lung > y.lung)
        return x;
        else
        return y;
}
//POST=ricavo il match di lunghezza maggiore con dimP elementi
```

```
tripla aux1(bool*M, int p, int t, int dimT, int dimP){
if(dimT==0 || dimT == t)          return tripla();
tripla x=aux1(M, t+1, dimT, P, dimP);
int y=aux2(M, P, dimP, T, dimT);
        if(x.lung > y)
        return x;
        else
        return tripla(t, p, y);
}
//POST=ricavo il match di lunghezza massima tra i prefissi con dimT elementi

Tripla aux2(int t, int p, int dimT, int dimP){
        If(p < dimP && t < dimT){
        Return 1 + aux2(M, p+1, t+1, dimP, dimT);
        else
        return 0;
}
//POST= massimo match tra dimP e dimT elementi
```

Per la dim. Induttiva.

Casi base: p==dimP→ritorno tripla vuota
p != dimP → inizio la ricorsione e scorro P, incrementando l'indice. Nel caso i-esimo, scorro l'array e determino, con il locale presente nella funzione e determino la lunghezza presente. Se maggiore del locale interessato la salvo, altrimenti proseguo nelle iterazioni, all'i-esimo +1 e così via fino a dimP.
Scorro quindi tutto l'array fino a dimP e post è rispettata.

Parallelamente, la seconda funzione è uguale, mi occupo di scorrere dimT rispettando POST. L'unica cosa diversa è lo scorrere l'array di riferimento M considerando entrambi i movimenti degli indici, per ricavare i sottoarray di riferimento in esso.

Nella terza funzione, alcune precisazioni.
Caso base è p==dimP o t==dimT, in quel caso ritorno 0, terminando la ricorsione.
Altrimenti calcolo un intero sulla base degli elementi presenti, non necessariamente maggiore di 0, al passo i-esimo sugli elementi t e p fino a dimT e dimP, quindi nei t+1, dimT-1, p+1, dimP-1… fino alla fine.
Rispetto in ogni caso POST.

Array a 3 dimensioni visto come una sola dimensione e pattern matching rispetto ad un intero val

In questo caso specifico, il problema affronta un array ad 1 dimensione visto come se ne avesse 3. Esso è formato dagli strati, dalle righe e dalle colonne (rispettivamente lim1, lim2, lim3). Per questo motivo si considera l'array e la dimensione finale è determinata da lim1*lim2*lim3. L'assetto di questo problema si dimostra simile ad altri.

Chiariamo uno dei concetti preferiti da Filè, le fette. Queste non sono altro che una scansione dell'array fatto a suon di mod e divisioni, sostanzialmente. Non è spiegabile in altro modo e non certo seguendo i papiri incomprensibili di chi non sa spiegarsi.

Lim1 rappresenta lo strato, con strato si intende la dimensione definita formata da lim2*lim3 elementi.
Lim2 rappresenta le righe e viene utilizzate nelle fette verticali per muoversi di riga in riga, con mod e divisione sulle altre due; Lim3 è il contrario e rappresenta le colonne e sfrutta in maniera simile ma opposta il movimento di lim2 per muoversi in quel senso.

Prendiamo ad esempio questo:

XXXX XXXX EEEE
XXXX XEEE EEEE
XXXX EEEE EEEE

Dove per X sono elementi presenti, E sono quelli non presenti, mentre è costituita da 17 elementi, chiamati dal buon prof, nele (numero elementi). La V-fetta 1 (in questo caso definita fino al quinto valore, quindi riga 1 dello strato 1) è indicata dall'evidenziazione sopra.
Sostanzialmente il problema richiede di scandire le V-Fette non vuote e trovare quante occorrenze ci siano del valore Val intero. Vogliamo semplicemente l'indice della V-Fetta (quindi 0, 1, 2 o 3) che ha questa proprietà.

Si richiede una funzione, possibilmente accompagnata da funzione ausiliaria, che faccia quanto descritto. Non avendo una soluzione in quanto non fornita dal professore per quanto da me richiesta, devo ragionare di mio.

Si richiede di tagliare queste V-Fette e di sapere che ci sono esattamente num occorrenze di un valore. Il ragionamento sarebbe di usare una funzione principale che scorre normalmente l'array. Questo sappiamo già essere formato da nele elementi, da considerare nella scrittura della funzione. Sappiamo inoltre che la generica dimensione dell'array sarà lim1*lim2*lim3, perciò le inserirei come condizioni.

Da considerare il possibile caso v-fetta vuota (vale a dire con elementi non definiti). Se num==0 potrebbe dare una corrispondenza che in realtà non ci sarebbe. Per ovviare a questo problema, una volta tagliata la v-fetta, si deve considerare il caso in cui num==0 ma la v-fetta (int) sia 0; in quel caso, semplicemente proseguo incrementando la ricerca, bloccando l'iterazione di quel ciclo e andando quindi avanti nella scansione della successiva, che cambia con l'indice i.
Personalmente, riterrei l'utilizzo di tre funzioni, una principale, una solo di supporto per il taglio delle magnifiche fette ed un'altra per capire l'effettiva lunghezza della h-fetta considerata, prendendo il suo indice.
La soluzione ricalca in parte l'esercizio 12, uno dei più contorti nella mentalità, possibilmente fatto in maniera umana e semplice.

<u>Parte iterativa</u>

```
int trovaV(int* X, int lim1, int lim2, int lim3, int nele, int val, int num){
        int occurr=0;    bool stop=false;          bool go_on=false;

        for(int i=0; i<lim1*lim2*lim3 && i<nele && !stop; i++){
                int lung_fetta=lung(lim2, lim3, nele, i);
                for(int f=0; f<lim1*lim2 && f<lung_fetta && !go_on; f++){
                        int val_fetta=cut_slice(X, lim1, lim2, lim3, i, f);

                        if(val_fetta==0 && num==0)     go_on=true;
                        if(val_fetta==val)         occurr++;
                        if(occurr==val)            stop=true;

                }
        }
        If(!stop)          return -1;
        else      return indf;
}
```

```
//se fosse una h_fetta= return X[index/lim3][slice_index][index%lim3];
int cut_slice(int X, int lim1, int lim2, int lim3, int index, int slice_index){
        return X[index/lim2][index%lim2][slice_index];
}

Int lung(int X, int lim2, int lim3, int nele, int ind_f){
        Int curr_column=0;

        curr_column = nele / lim3;
        int rest = nele % lim3;
        if (ind_f < rest)
        curr_column++;

        return curr_column;
}
```

Invariante del ciclo: 0<i<lim1*lim2*lim3 && i<nele && stop==true sse occurr==num

Personalmente quindi, l'invarianza di questo ciclo si ha sulla permanenza rispetto ai nele elementi presenti all'interno del ciclo fornito e contando sulla presenza di una possibile v-fetta, delimitata dagli indici lim1 e lim2 dentro l'array. Nel primo caso, considero l'elemento i-esimo; esso sarà necessario nelle due funzioni ausiliare definite a capire se esiste una v-fetta abbastanza lunga e se esiste il singolo elemento.

Passo i-esimo: considero i, la v-fetta nell'indice i-esimo e l'elemento val. Se presente un'occorrenza, incremento l'intero locale tenuto come riferimento.

Passo i-esimo + 1 e induttivo: scorro le v-fette iterativamente, sperando di trovare occorrenze di val. Se presenti vengono salvate. Se le occorrenze fossero uguali a num, interrompo l'iterazione uscendone prima.

Parte ricorsiva

Si considera una versione ricorsiva dello stesso problema. Non avendo la traccia su MEGA, a memoria, si utilizzava la stessa segnatura della funzione trovaV, aggiungendo il valore indF, necessario per muoversi ricorsivamente nell'array considerato.
La ricorsione utilizza comunque la funzione di supporto cut_slice per sapere a che valore mi sto riferendo.
I casi limite dell'iterazione come al solito diventano casi base.

```
int trovaV(int* X, int lim1, int lim2, int lim3, int nele, int val, int num, int indF){
        int indice=0;                //la uso come puntatore per riferirmi a dove sono ora

        if(indice>nele || indice==lim1*lim2*lim3)                return 0;
        indice=trovaV(X+1, lim1, lim2, lim3, nele, val, num, indF);
        int val_fetta=cut_slice(X, lim1, lim2, lim3, indice, indF+1);
        int lung_fetta=lung(lim2, lim3, indF, nele, indice+1);

        if(indice<lung_fetta && indice < lim1*lim2){
                if(val_fetta==val)
                fetta=trovaV(X, lim1, lim2, lim3, nele, val, num, indF+1);
                else
                return trovaV(X+1, lim1, lim2, lim3, nele, val, num, indF);

                if(indF==num)   return indF;
                else      return 0;
```

```
        }
        else{
        indice= X[lim][lim2][indice%lim3+1];
        return -1;
        }
}
```

La soluzione fornita è sempre un ragionamento. Di fatto, ho una funzione ricorsiva che ritorna un int. Questo può essere il mio indice, che mi qualifica dove sono ora. Con le funzioni che ho usato prima, rifaccio praticamente la stessa cosa. Uso trovaV per scorrere tutto l'array e capire la posizione, prendere il valore della fetta da poter utilizzare con la funziona cut_slice e la lung_fetta, per poter capire quando poter fermare la ricorsione della vetta. Se trovo una corrispondenza del valore dato rispetto alla fetta corrispondente, continuo a cercare tra le fette, altrimenti incremento l'array X e calcolo una nuova fetta, ricominciando la ricerca. Qualora avessi raggiunto la fine plausibile per la ricerca del mio indice, avanzo di una colonna usando il calcolo per mod (raggiungo la colonna data, avendo la stessa riga, lo stesso strato, ma una riga in più) perché ho raggiunto i bounds della mia condizione.

Dim. induttiva

Passo i-esimo: detengo il mio indice di riferimento e con esso taglio una v-fetta dell'array, assieme a capire quanto lunga potenzialmente può essere la fetta in uso. Fatto questo, approccio il condizionale. Questo presenta due ipotesi: la fetta che cerco rientra nella lunghezza a priori calcolata (e se completa comunque rientra nel numero di elementi contabili) e cerco di capire se il valore della fetta i-esima può essere utile. Se si, viene incrementata la fetta i-esima di indf+1, fino al limite consentito per la ricerca (lim1*lim2) o comunque il numero di elementi presenti effettivamente e calcolati da lung.
Se no, avanzo di una colonna e ritorno -1 perché non ho trovato la v-fetta che cercavo.

Passo induttivo: fino ai limiti fissati, continuo a manipolare gli indici e capire se effettivamente riesco a raggiungere la fine dell'algoritmo trovando una possibile V-fetta. Qualora la trovassi, viene salvata e confrontata con l'elemento i-esimo di riferimento, altrimenti concludo la mia ricerca.

Altro esercizio del pattern matching: due array di interi (T, P) in cui per ogni elemento match di T in P viene salvata la corrispondenza dello stesso in una lista P-esima

Se la posizione T[i] fa match con P[0], nella lista L_0 viene inserita la posizione T[i] in cui si è fatto match e così via fino a finire gli elementi dentro T. Si passa poi all'elemento successivo di P, stessa cosa e fino alla fine si procede così. Il programma sotto chiarifica questa introduzione.

Parte iterativa:

Usando la struttura, in questo caso nodoG* del tutto simile a nodoE* (puntatore a nodo, con campo info nodo e nodo next, un altro nodoE), si deve compilare la funzione:

PRE_M=(T è un array di dimT interi, P un array di dimP interi, dimT e dimP>=0)

nodoG* M(int* T, int dimT, int* P, int dimP)

POST_M=(M restituisce una lista di dimP liste tale che la lista L_i (i in [0..dimP-1]) contiene gli indici di tutte le occorrenze di P[i]in T, da sinistra a destra)

In questo esercizio, si segue la generica intuizione degli esercizi del pattern matching. Uso una funzione ausiliaria che cerca il match e la funzione principale che avanza nella lista di riferimento. Il ciclo principale

scorrerà dimP, per cercare il match si sfrutterà dimT. Si somma l'indice i per tenere conto dello spostamento per la ricerca del match nella struttura sottostante.

```
nodoG* M(int* T, int dimT, int* P, int dimP){
        nodoG* lista;
        if(!dimT || dimT < dimP || !dimP)        return 0;
        for(int i=0; i<dimP; i++){
                nodo *matched=aux(T, dimT, p+i);
                if(matched)
                push(lista, matched);
                else
                lista=new nodoG(matched, 0);
        }
}

nodo* aux(int *T, int dimT, int *P){
        nodo*lista;
        for(int i=0; i<dimT; i++){
                if(T[i]==*P)
                lista=new nodo(i);
                else
                push(lista, i);
        }
}

void push(nodoG* list, nodo*x){
        nodo* aux=list;
        while(aux->next)
        aux=aux->next;

        aux->next=x;
        return;
}
```

Invariante del ciclo: (0 <=i <=dimP) && (matched è la lista che contiene il match temporaneo sull'elemento i-esimo e sui successivi i elementi tra dimP e dimT dei due array).

Si ricerca come al solito il match nelle due liste.
Al passo 0, i=0 e l'indice di riferimento è il caso i-esimo. Approccio la funzione ausiliaria, eseguita la prima volta sull'indice e verifico la presenza o meno di un match. Se presente, creo una lista di supporto per mantenerlo.
Al passo i-esimo, proseguo la mia ricerca nell'indice +1 e verifico la presenza di un match. Se ancora nella funzione ausiliaria non presente un match proseguo, altrimenti accodo il match precedente alla lista precedente e già presente.
Se il match i-esimo + 1 risulta maggiore del match i-esimo, viene dunque salvato, altrimenti viene scartato e si passa all'elemento i-esimo + 2.
Così via fino alla fine, nei limiti stanti e quindi POST è dimostrata.

Parte ricorsiva

La funzione ricorsiva ha firma:

//PRE_FM1_ric=(G sia una lista corretta di liste tutte corrette e contenenti valori interi non negativi e crescenti, e sia L una lista corretta e possibilmente vuota di interi non negativi e i>=0)
match FM1_ric(nodoG *G, nodo *L, int i)
//POST_FM1_ric=(la funzione restituisce il valore [i, x, y] che corrisponde a MAX_S_(G,L

Anche qui è la solita funzione di match che cerca il match di lunghezza massima tra due strutture, in questo caso una lista L e la struttura nodoG* G.
È un esercizio del luglio 2014, scritto in aramaico antico dal testo originale, ma similare come trattazione a quelli presenti sopra.
Tradotto in un linguaggio umano e in tre parole, si richiede di trovare un match come una sorta di tripla, quindi (x, y, z) dove x rappresenta la lista i_esima di riferimento, y rappresenta l'indice di inizio del match e z rappresenta l'indice di fine dello stesso).
In questo caso, la struttura x, y e z è incapsulata all'interno di un tipo chiamato match, del tutto identico al tipo *tripla*.
Due pagine contro tre righe di spiegazione.

Ragionando quindi:
-la lista non c'è, quindi si restituisce (i, 0, -1), quindi indice i generico, inizio 0 e fine non precisata
-la lista c'è, quindi ragionevolmente invoco una funzione ausiliaria che cerca di capire dove sta e/o finisce questo match, restituendo un int come al solito quindi capendo se è il match di lunghezza massima.
Chiaramente mi devo spostare nella funzione locale ricorsivamente sui nodi di L per capire se trovo o meno un match.
Verifico poi se la lunghezza può essere o meno utile; se si, la salvo, altrimenti proseguo.

```
match FM1_ric(nodoG *G, nodo *L, int i){
        if(!L)     return match(i, 0, -1);

        match local=FM1_ric(G, L->next, i);
        int aux=find_match(G, L->info);
        match matched=new match(i, L->info, aux);

        if(i > aux)
        return local;
        else
        return matched;
}

int find_match(nodoG*g, int x){
        If(!G)     return 0;
        if(g->info == x)
        return 1+find_match(G->next, x+1);
        else
        return x;
}
```

La dim.induttiva della funzione segue il principio detto.

Ho una lista, da PRE conosco la validità della lista e di G. Cerco quindi per l'elemento i-esimo la presenza in una lista ausiliare la presenza del match scorrendo la lista di un l->next al passo i-esimo. Per fare ciò utilizzo

una funzione ausiliaria. Nel caso in cui si trovi un singolo match, esso verrà incrementato di 1 e aumenta la posizione indice di match per restituirla alla fine, altrimenti restituisce la lista presente ora.

Al passo i-esimo + 1 viene confermata la validità del ragionamento. Si cerca il match nell'elemento successivo alla lista fornita e presente e si ripete idealmente il ragionamento. Cerco nell'elemento l->next->next la presenza di un nodo che ha match. Se questo match esiste alla luce di quanto visto prima e se maggiore del locale presente, lo salvo e restituisco il match maggiore della funzione ausiliaria, altrimenti il match locale. Alla luce delle osservazioni dette, articolo che PRE_RIC porta correttamente a POST_RIC. Dimostrazione discorsiva questa, ma matematicamente corretta, perlomeno dati i tipi di studio.

Pattern matching tra array di interi e albero binario T

struct nodoE{nodo*info; nodoE*next; nodoE(nodo*a=0, nodoE* b=0){info=a; next=b;}};

PRE=(albero(T) ben formato, P contiene dimP>=0 elementi)
nodoE* PM1(nodo*T, int* P, int dimP)
POST=(se in T esiste un camminoche contiene unmatch di P completo e contiguo,allora PM1 restituisce una lista di dimP nodi di tipo nodoE che puntano ai nodi del cammino più a sinistra su cui esiste un tale matchdi P, altrimenti PM1 restituisce 0).

Affrontiamo il problema sia iterativamente che ricorsivamente (nonostante non venga specificato per certo come lo si debba affrontare).

Per complicarci la vita come piace al prof, partiamo da una versione iterativa.
Sappiamo che sarà necessaria una funzione ausiliaria, che cerca effettivamente il match.
In questa soluzione, dato che dobbiamo scorrere iterativamente, considero che se ci sta un match utilizzo una lista ausiliaria che mette alla propria fine (push) i nodi che hanno avuto una corrispondenza da match.

```
nodoE* PM1(nodo*T, int*P, int dimP){
        nodoE* list=new nodoE(0);
        while(T && P){
                nodoE*matched=match(T, P, dimP);
                if(matched){
                        list->info=matched;
                        list=list->next;
                        P=P+1;
                }
                T=T->next;
        }
        return list;
}

nodoE* match(nodo*T, int*P, int dimP){
        nodoE*matched=0;
        if(T->info == *P){
                matched->info=T;
                matched=matched->next;
                T=T->next;
                P=P+1;
        }
        return matched;
}
```

Qui è la versione più semplice, semplicemente scorro pezzo alla volta la lista ritorno il pezzo matchato e in entrambi casi lo restituisco, sia se corrisponde a 0, sia se non corrisponde.

Parte ricorsiva

```
nodoE* PM1(nodo*T, int*P, int dimP){
        if(!P || !T) return 0;
        if(T->info==*P)
        return PM1(T->next, P+1, dimP-1);
        else
        return PM1(T->next, P, dimP);
}
```

Senza le funzioni ausiliarie che tanto piacciono al caro professore, direi che questa è una buona soluzione. La dim. Di correttezza si completa da sola, avendo come caso base l'assenza del nodo* o array, altrimenti mi muovo ricorsivamente nel caso appunto ricorsivo assumendo l'esistenza degli altri pezzi di nodo oppure mi muovo sui singoli pezzi di array. Se trovo un match riempio la lista di riferimento altrimenti ritorno 0 come voluto.

Attraversamento di un albero binario da dx a sx in larghezza; uso della struttura nodoE*

PRE=(albero(R) è un albero binario ben formato, X e T contengonoun numero di posizioni pari al numero dei nodi di albero(R))
void scanliv(nodo*R, int& nliv, int*X, int*T)
POST=(nliv è il numero di livelli di albero(R) e X e T contengono i valori descritti nell'Esempio 1, ovviamente per l'albero(R) in input)

Lo affonto prima iterativamente; consideriamo che in soldoni avremo due strutture, in questo casi due array di interi, T che salva i singoli nodi info e X che salva il numero di nodi presenti in un livello esaminato.

```
void scanliv(nodo*R, int& nliv, int*X, int*T){
        while(R){
                if(R->right){
                        *T=r->info;
                        R=R->right;
                }
                if(R->left){
                        *T=r->info;
                        R=R->left;
                }
                //a questo punto siamo nel nodo radice dove ritorniamo e qui
                //dobbiamo prenderne l'info e poi cambiare livello
                If(R){
                        T=R->info;
                        X=X+1;
                        nliv=nliv+1;
                }
        }
}
```

Versione ricorsiva

```
void scanliv(nodo*R, int& nliv, int*X, int*T){
```

```
        if(!R)    return;
        if(R->right){
                scanliv(R->right, nliv, X+1, T);
                T=R->info;
        }
        if(R->left){
                scanliv(R->left, nliv, X+1, T);
                T=R->info;
        }
        T=R->info;
        nliv=nliv+1;
        *X=*X+1;
}
```

Dim. di correttezza:
-non esiste l'albero, albero vuoto->ritorno void, così perlomeno viene ritornata la scansione dei livelli già effettuata
-l'albero esiste e posso muovermi in due sensi predefiniti ma in larghezza; non avendo la FIFO l'unica cosa che possiamo fare è semplicemente muoversi in un senso e nell'altro (iterativamente) oppure fino a quando ci sono dei nodi (ricorsivamente), assumendo sempre che essi esistano altrimenti si ricade nel primo caso base.
La precondizione porta a questo e la postcondizione sarà dimostrata anche dal fatto che, una volta scanditi in larghezza i livelli presenti, avanzo di livello sia nei nodi che nei livelli stessi, poiché presenti da PRE e di conseguenza saranno sicuramente anche in POST.

Trovare un nodo in un albero BST (avendo campi info ed n, che ci da il nodo n-esimo dell'albero durante il suo attraversamento).

Considerato un albero BST, vogliamo che restituisca il nodo n-esimo secondo l'ordine infisso (sinistra-radice-destra). Il campo n è importante per capire dove ci stiamo muovendo, dato che diminuisce ad ogni spostamento più in profondità dell'albero (quindi determinando quanti sottonodi ha in un momento preciso un certo nodo).

PRE=(albero(r) è benformato,n>0)
nodo* trova(nodo*r, int n)
POST=(se albero(r) contiene almeno n nodi, restituisce il puntatore al nodo numero n nell'ordinamento determinato dalla visita infissa, altrimenti restituisce 0)

Versione iterativa

```
nodo* trova(nodo*r, int n){
        bool stop=false;
        while(r && !stop){
        if(n==r->num)
                stop=true;
        if(n>r->num)
                r=r->left;
        else
                r=r->right;
        }
}
```

Versione ricorsiva

```
nodo* trova(nodo*r, int n){
        if(r->num<n)    return 0;
        else{
                if(r->left){
                return trova(r->left, n);
                }
                if(r->left->num+1==n){
                return r;
                }
                if(r->right){                   //stesso spostamento che a sx
                return trova(r->right, n-(r->left-num)-1);
                }
        }
        If(n==1)
        return p;
        else
        return trova(p->right, n-1);
}
```

Il BST è un po' diverso e nell'esercizio originale non si considera l'implementazione iterativa, comunque fattibile grazie al confronto del campo num con il campo n dato e diventa effettivamente facile ricercare un campo in queste condizioni.
Induttivamente possiamo quindi ragionare dicendo, se non abbiamo la radice dell'albero lo ritorno subito, altrimenti ci muoviamo subito a sinistra e subito a destra, nel caso iterativo semplicemente un pezzo alla volta, perché me lo dice la precondizione, Ciò implica che la ricerca possa interrompersi solo nel caso base in cui non ho più elementi o sono arrivato alla radice.
Nel caso ricorsivo, invece, mi muovo allo stesso modo finché ho abbastanza elementi e, se mi muovo a destra, sottraggo lo stesso numero di nodi per avere la stessa quantità esaminata dall'altra parte.
A questo punto la ricerca va avanti sulla base dell'intero *num*>0, utilizzato solo nella versione ricorsiva, dove mi permette di spostarmi agevolmente.

Albero binario da cui devo costruire due liste (una che contiene i valori distinti di T e l'altra gli altri valori)

Solita struttura nodoE e seguente prototipo:
void filtra(nodo*R, nodoE*& tenere, nodoE*& buttare)

L'idea intuitiva dell'algoritmo risolutore è capire, oltre ad aver esaminato il classico caso della lista vuota, se l'info del nodo in questione è già stato incontrato; nel qual caso andrà in tenere, altrimenti in buttare.
La funzione qui sarà ricorsiva, ma a scopo di esercizio ne ricavo anche una funzione iterativa, brevemente commentata in quanto simile all'altra se non per due singole istruzioni.
Seguono poi due funzioni iterative, di cui si forniranno maggiori dettagli sotto, concludendo la trattazione del problema con la dimostrazione induttiva per ogni singolo algoritmo.

```
void filtra(nodo*R, nodoE*& tenere, nodoE*& buttare)
{
        If(!R)    return;
        if(check(R, tenere))      tenere=new nodoE(R, tenere);
        else buttare=new nodoE(R, buttare);
        filtra(R->left, tenere, buttare);
        filtra (R->right, tenere, buttare);
        return;
```

```
}
```

Nella variante iterativa, poco cambia, mi sposto a sx e dx con r=r->left e r=right, il resto rimane uguale.

```
bool check(nodo*R, nodoE*& tenere){
        if(tenere->info->info==R->info) return true;
        else return (R, tenere->next);
}
```

Oltre alla funzione filtra, si chiede di costruire 2 funzioni iterative:

i) una funzione iterativa buildBST che usa la lista tenere per costruire un albero BST aggiungendo, uno alla volta, i nodi di R puntati dalla lista tenere. Inoltre, la funzione, man mano che la lista tenere viene consumata,deve deallocare i suoi nodi nodoE non più utili.

ii) la funzione butta che si occupa di deallocare i nodi della lista buttare. Attenzione che si tratta di buttare sia i nodi nodoE della lista buttare che i nodi di tipo nodo che sono puntati da questi nodoE.

Mettiamoci all'opera dunque con le singole funzioni, partendo da buildBST, il quale opera già su un puntatore a nodo e per questo direi tipo di ritorno un void.
Sappiamo inoltre che per non creare garbage nello heap devo subito deallocare ciò che sto ora creando, questo dalla lista tenere. La scorro iterativemente e ricorsivamente per considerare ambedue le trattazioni del problema.

```
void buildBST(nodo* BST, nodoE* tenere){
        while(tenere){
                if(!tenere || !BST ) BST=new nodo(tenere, 0);
                nodoE*aux=tenere;
                if(tenere->info->info < BST->info){
                        BST=BST->left;
                        delete tenere;
                        tenere->next=aux->next;
                }
                else{
                        BST=BST->right;
                        delete tenere;
                        tenere->next=aux->next;
                }
                BST->info=tenere->info->info;
                tenere=tenere->next;
        }
}
```

Ho poi la funzione butta da scrivere che dealloca i nodi di buttare, dove segnala che si devono togliere naturalmente sia gli info quindi i nodo che i nodoE, indicazione poco utile perché si sa già questa cosa.

```
void butta(nodoE* buttare){
        if(!buttare)      return;
        nodoE* aux=buttare;
        delete buttare;
        buttare->next=aux->next;
}
```

Nel modo ricorsivo si richiama semplicemente la funzione e si distrugge al ritorno della ricorsione, nessuna sorpresa quindi da questo punto di vista.
Dal punto di vista delle dimostrazioni induttive, il caso base:

-albero vuoto, quindi mi fermo nel caso di butta, invece nel caso di buildBST costruisco un nodo, anche quando ho BST senza nodi, la prima volta, per poter aggiungere il nodo.
Considerando da PRE che scrivo ora (partendo quindi in butta con una lista buttare nodoE riempita di valori, e nel caso di buildBST il nodo* BST, lista che deve essere costruita e tenere lista riempita di elementi diversi da 0), ci si sposterà nelle liste concatenate fino a quando contengono elementi, quindi matematicamente fino a l->next definito e !=0.
Intuitivamente, per ogni singolo nodo lo metto nella radice perché so dove sono in un determinato momento, altrimenti grazie al puntatore ne salvo la posizione e dealloco il nodo attuale, non lasciando garbage nello heap. A questo punto passo al nodo successivo.
Nel primo caso, si considera la condizione BST, quindi della maggioranza/minoranza di un nodo rispetto all'albero maestro e cerco di capire se posso infilarlo in una determinata posizione, a sinistra se minore e a destra se maggiore di chiave.
Nel secondo caso, semplicemente devo deallocare tutto ciò che trovo, dal punto di vista di nodi info, in quanto i nodoE* mi servono per muovermi, fintanto che si mantiene la condizione iniziale cioè nodoE* next!=0. POST sono dimostrate.

Eliminare i nodi con info==y dall'inizio e dalla fine di una lista concatenata

```
nodo* elimI(nodo*L, int &n, int k, int y){
        if(!L) return 0;
        if(L->info == y){
                n++;
                nodo* x =elimI(L->next, n, k, y);
        }
        If(n==k)            return L;
        If(n > 0){
                delete L;
                n=n-1;
                return x;
        }
        else{
                L->next=x;
                return L;
        }
}
```

Nella soluzione di eliminare dalla fine, faccio direttamente un controllo al posto del classico caso base L!=0 controllando quanti nodi mi rimangono, poiché k sono i nodi ancora da cancellare, altrimenti non ne abbiamo altri e ritorno 0.
In soldoni quindi, prendo il nodo della lista spostata ricorsivamente e controllo quanti nodi mi rimangono; se me ne rimangono ancora, cancello il nodo attuale e mi sposto a quello dopo diminuendo n, altrimenti ritorno i nodi che ho (x se ne ho ancora e 0 altrimenti); in entrambi i casi si dimostra facilmente che la lista presenta dei valori, in un caso generico di esecuzione. Nel caso base chiaramente è vuota e si ritorna nullptr, altrimenti si continua a scorrere la lista alla ricerca di un possibile nodo da eliminare.

La differenza sostanziale sta nell'idea di trattazione, in quanto eliminare dalla fine significherebbe eliminare dopo aver fatto tutti i controlli e quindi come prima istruzione nella funzione, ma dopo essersi spostati; eliminare all'inizio significa cancellare immediatamente fatto lo spostamento ricorsivo.

In diversi esercizi, il caro professore utilizza, in questo caso con un senso, una variabile n, in questo caso passata per riferimento e quindi deve essere decrementata con l'apposita istruzione separata.
Essa pernette di capire come e dove spostarsi all'interno della struttura linked-list fino a quando ho nodi; se ne ho ancora vado avanti (L->next fintanto che ho n-k nodi da eliminare costituendo poi un suffisso di L valido nel caso 2 e un pre-fisso valido nel caso 1, eliminando le y-esime occorrenze n>=k, usando dei termini più formali e che si spera possano andare bene al carissimo professore).

Distribuire i nodi di una lista su due liste L1 ed L2 (in L1 quelli contenuti in A e in L2 quelli no)

Quindi:
A=[2, 4, 10] Grigio= Non ce ne frega nulla
I gialli vanno intesi come indici in L (quindi indice 2 e indice 4); facendolo coi colori si vede subito cosa si deve fare, no?

L= 3->1->2->20-> -11

Diventa: L1: 2-> -11 e L2= 3 ->1->20

L'idea originale deve sfruttare le code, noi usiamo un semplice nodo*.

Iterativamente:

```
nodo* distr_it(nodo*&L, int*A, int dimA){
        while(L && i < dimA){
                nodo*x=L;
                while(x){
                        if(A[i]==x->info)){
                                Nodo*L1=L;
                                L=L->next;
                                L1->next=0;
                                L1=conc(L1, x);
                        }
                        else{
                                Nodo*L2=L;
                                L=L->next;
                                L2->next=0;
                                L2=conc(L2, x);
                        }
                        X=x->next;
                }
                L=L->next;
                i=i+1;
        }
        //finisco prima ma rimangono ancora elementi in L
        while(L){
                Nodo*L2=L;
                L=L->next;
                L2->next=0;
                L2=conc(L2, x);
        }
        return L1;
}
```

Ricorsivamente:

```
nodo* distr_ric(nodo*&L, int*A, int dimA, int n){
        if(!L || !dimA)   return 0;
        nodo *L1, *L2;

        if(n == *A){
                L1=L;
                L=L->next;
                L1->next= distr_ric(L, A+1, dimA-1, n+1);
        }
        else{
                L=L->next;
                L2=distr_ric(L, A+1, dimA-1, n+1);
        }
}
```

Dimostrazione induttiva

Nel caso iterativo si procede, fintanto che ci sono elementi nell'array e nella lista. Al passo i-esimo, valuto se l'elemento i-esimo di A è contenuto in qualche forma in L; se risultasse contenuto a quel punto stacchiamo il nodo i-esimo da L mettendolo in L1, mentre se non risulta contenuto lo mettiamo in L2.
Ciò va avanti fino alla fine della lista.
Invece nel ricorsivo, ci sta una utilissima variabile n (è ironico ma neanche troppo; il professore carissimo la spiega come una variabile che sommata agli indici fa trovare gli elementi corrispondenti, che vuol dire tutto e niente come sempre.
In realtà sarebbe come avere i nel caso iterativo, e il problema è facile).
Quindi se l'elemento n k-esimo è uguale all'elemento i-esimo in L, si va in L1 altrimenti in L2. Simple as that.

Spezzare una lista in due liste prendendo gli indici di un array di interi A

Quindi se A = [2, 2, 1, 3] ed L = 3->1->2->20-> -11
I primi 2 di A vanno in L1, per l'indice dopo altri 2 elementi vanno in L2, per l'indice dopo 1 elemento va in L1 mentre gli ultimi 3 in L2.
Morale della favola, L1 sarà 3->1-> -11, mentre L2= 2->20

Semplice e chiaro, altro che una pagina di testo per sta robaccia.

La funzione è questa e il carissimo professore consiglia due funzioni, una che concatena due liste e l'altra che stacca i nodi, un po' come viene sempre fatto in questi casi direi, gran bella mano come sempre!

```
void Fric(nodo*L, int*A, int dimA, nodo*&L1, nodo*&L2){
        if(!dimA)         return;
        if(!L)    L1=conc(L1, L);

        f=taglia(A[0], L, resto);
        L=resto;
        resto=0;
        L1=conc(L1, f);

        s=taglia(A[1], L, resto);
```

```
        L=resto;
        resto=0;
        L2=conc(L1, f);
}

nodo* taglia (int q, nodo* L, nodo* resto){
        if(!L || q<=1)    return 0;
        while(q > 1){
                L=L->next;
                q=q-1;
        }
        If(L){
                resto=L->next;
                L->next=0;
                return resto;
        }
}

void Fiter(nodo*L, int*A, int dimA, nodo*&L1, nodo*&L2){
        while(L && dimA){
                f=taglia(A[0], L, resto);
                L=resto;
                resto=0;
                L1=conc(L1, f);

                s=taglia(A[1], L, resto);
                L=resto;
                resto=0;
                L2=conc(L1, f);

                A=A+2;
                dimA=dimA-2;
        }
}
```

È oggettivo dimostrare come la funzione data stacchi pezzi di lista, piuttosto che ogni indice, ogni due indici; nel ragionamento ricorsivo, la cosa si considera facendo due ricorsioni, una sull'elemento iniziale e l'altra sull'elemento i-esimo +1; in questo modo si considerano due metà di lista, che alla successiva iterazione saranno ordinate per il calcolo della lista e della parte restante (resto quindi), staccabile (quindi accodando alla stessa porzione resto) finché abbiamo nodi ed è soddisfatta la detta condizione.
Al nodo i-esimo fino a i=n-1 vengono staccati tutti i nodi con questa condizione, soddisfando POST (che richiede di staccare la lista, essendo POST intuitive non vengono da me segnate.

Se invece discutiamo la versione iterativa, il discorso non cambia molto, sfruttando l'iterazione, che richiede un incremento "manuale" di A, decrementando dimA della stessa quantità fornita. Se ho abbastanza elementi tali da poter eseguire l'operazione descritta, vedi sopra, è soddisfatta.

Formare pezzi di lista Crescenti e Non Allungabili (CeNA) di una lista, per esempio:

L= 2->3->4->-2->-2->3->1->1->10->11->2->0

Prendiamo i pezzi che è possibile, tipo 2-3-4, oppure altri come 2-2-3, 1-1-10-11, 2, 0.
Intuitivamente credo che l'idea sia chiara.

nodoL è come nodoE e dobbiamo restituire una lista con quelle caratteristiche. Viene consigliata una funzione ausiliaria.
La funzione ricorsiva è la seguente:

```
nodoL* Gric(nodo*L){
        if(!L)     return 0;
        nodo*x=L;
        nodo*y=tagliaric(L);
        L=y->next;
        y->next=0;
        nodoL *z=new nodoL(z, Gric(L->next);
        return z;
}

nodo *tagliaric(nodo *L){
        if(L->next && L->info <= L->next->info)
        return tagliaric(L->next);
        return L;
}
```

Per la parte iterativa abbiamo invece una funzione che mantiene i nodi formati in lungh. descrescente a parità di lunghezza.

```
nodoL* ordina(nodoL* LL){
        nodoL *x=0;
        while(LL){
                nodoL *coda=LL;
                LL=LL->next;
                coda->next=0;
                x=push_order(x, coda);
        }
}

nodo *push_order(nodoL *x, nodoL *y){
        nodoL start=x;

        if(!x)     return y;
        else if(!y)         return x;
        while (contanodi(x)<=contanodi(y) && x->next){
                x=x->next;
        }
        if(x->next){
                z->next=x->next;
                x->next=z;
                return start;
        }
        else{
                x->next=z;
                return start;
        }
}
```

Restituire il nodo k-esimo percorrendo un albero binario

In questo caso abbiamo una funzione ricorsiva che restituisce il nodo k-esimo seguendo l'ordine infisso. Si noti che in questo esercizio si utilizza anche un campo "n", che indica il numero di nodi radicati nel sottoalbero considerato un determinato nodo. Vi sono poi C, array di interi con elementi maggiori a prof. di albero (usato evidentemente per lo scorrimento come ogni tanto capita, 0 se vado a sx, 1 se vado a dx) e k, il cui valore sottratto ad m restituisce sarà il valore finale di k.

```
nodo*cerca1_infix(nodo*r, int&k,int *C){
        if(!r)     return 0;
        if(k==1)          {*C=-1; return 0;}
        if(r->left){
                *C=0;
                return cerca1_infix(r->left, k, C+1);
        }
        if(r->right){
                *C=1;
                return cerca1_infix(r->right, k, C+1);
        }
        k=k-1;
}
```

Quella sopra è una normale, classica funzione di scorrimento ricorsivo, non credo ci sia molto da commentare.

Passiamo invece ad una funzione ricorsiva chiamata *completa*, in cui bisogna completare il campo n valorizzandolo con i nodi contenuti nei sottoalberi di riferimento radicati nel nodo.
La funzione sarà la seguente (una nota che fa il prof è di non usare funzioni ausiliarie):

```
void completa(nodo*r){
        int sx, dx;
        if(!r)     return;
        completa(r->left);
        completa(r->right);

        if(r->left)
                sx=r->left->n;
        if(r->right)
                dx=r->right->n;
        r->n=sx+dx+1;             //per valorizzare la radice in ogni caso, anche con dx e sx vuoti
}
```

Viene poi richiesta una funzione iterativa che restituisce il nodo k-esimo dell'albero e il cammino dalla radice fino al nodo interessato. Della funzione detta è richiesto un invariante, che fornirò e commenterò a seguito della scrittura della funzione, con firma e implementazione sotto riportata.
Una cosa da aggiungere è che il prof specifica che l'uso di n è utile per seguire il cammino senza errori; in effetti come già visto permette di capire se un campo è giò stato visitato per esempio nella radice, senza dover eseguire lo spostamento, ma semplicemente seguendo questa sorta di sentinella.
Si sa quindi che il nodo k<= r->n verrà sempre trovato, perché ci sarà sempre una radice di suo riferimento.

```
nodo* cerca2_infix(nodo* r, int k, int*C){
        bool found=false;
```

```
        if(!r)     return 0;
        while(r && !found){
                if(r->left) {
                if(r->left->n >=k){
                        *C=0;
                        r=r->left;
                        C=C+1;
                        else{
                                k-=r->left->n;
                                if(k==1){
                                        *C=-1;
                                        found=true;
                                }
                                else{
                                        *C=0;
                                        r=r->right;
                                        C=C+1;
                                        k=k-1;
                                }

                        }

                }
                if(r->right) {
                        if(k==1){
                                found=true;
                                *C=-1;
                        }
                        else{
                                *C=1;
                                r=r->right;
                                C=C+1;
                        }
                }
        }
}
```

Detto questo, la funzione presenza un'invarianza interessante, grazie anche ad n. In poche parole smetto il ciclo qualora abbia trovato un elemento e qualora abbiamo abbastanza nodi rispetto ad r->k, tra n ed 1. Nel nodo r-k esimo in n la ricerca si conclude, altrimenti si cerca a destra e sinistra.
Più formalmente, scriviamo un intuitivo invariante di questo tipo:

R=(1<=r->k<=n && ricerco a dx/sx il nodo k-esimo considerano gli n spostamenti in ordine infisso && found==true sse k==1 a dx o sx)

Proseguendo quindi, la dim. induttiva per ogni funzione segue un principio semplice.
Partiamo dall'ultima funzione.
Considero il caso base in cui non ho abbastanza nodi, ritorno 0.
Considero invece quando ho abbastanza nodi e posso muovermi a sinistra, radice e destra. Devo però considerare come variabile n, che mi aiuta nello spostamento tra sx, radice e dx.
Se esiste il nodo sinistro, ricerco un nodo ricorsivamente solo a sx. Qualora lo trovassi interrompo la ricerca, altrimenti sottraggo lo stesso numero k di nodi percorsi e guardo la radice per trovare il nodo.
Se esiste fermo la ricerca, altrimenti vado a destra e controllo da questa parte.

Se non ho fortuna, proseguo a sinistra e controllo se ho trovato il nodo k-esimo, altrimenti proseguo ulteriormente la ricerca, guidandomi con n, il quale mi evita degli spostamenti nei nodi radice, grazie alla proprietà dell'assicurato spostamento k-esimo da n elementi presenti.

La funzione completa non è poi molto dissimile da questa, in quanto semplicemente percorriamo tutto l'albero, cercando di capire se esistono nodi a sinistra e destra. Essendo ricorsiva posso solo spostarmi e poi valorizzare n con i nodi percorsi fino a quel momento, sommando questi ultimi valori se presenti al valore 1 per formare il nodo radice (nel caso pessimo in cui entrambi siano a 0). POST verrebbe rispettata data la validità degli spostamenti presenti e le operazioni svolte sul campo n per poter ragionare.
Non cambia molto infine nella prima funzione, la quale si sposta solo se presenti i nodi valorizzando N. In questa non c'è poi molto da aggiungere a quanto descritto, vista poi la chiara spiegazione della funzione *completa* già totalmente bastevole a coprire la funzione descritta.

Eliminare le ripetizioni di in un campo info da una lista concatenata lasciando solo il primo info non ripetuto (l'implementazione originale prevedeva l'uso di una struttura coda, noi lo facciamo con una lista).

Questa tipologia di esercizio è prevedibile e occasionalmente spunta, lo facciamo *once and for all*, direi.
Viene richiesta una funzione ricorsiva sotto presente.
Una cosa interessante da considerare qui è una lista con info e un campo che segnala la posizione; questa deve essere usata per mettere i singoli nodi della stessa posizione nello stesso nodoF, lista estesa di nodi.

Se avessimo ad esempio questa lista (primo campo è la posizione nella sequenza, il secondo campo è info)

L=[0,2]->[1,0]->[2,2]->[3,0]->[4,1]->[5,2]->[6,1]->[7,2]->[8,0]->[9,1]

Vorrebbe dire che i singoli nodi (tolte le dette ripetizioni) sarebbero così:

1) [3,0] -> [8,0]
2) [6,1] -> [9,1]
3) [2,2] -> [5,2] ->[7,2]

Quello sopra è il caso di nodi della stessa posizione messi poi in una coda, come detto, quindi una FIFO con campi primo e ultimo. A scopo didattico usiamo nodoF che collega i singoli nodi.

La funzione sotto usa due funzioni ausiliarie ricorsive che sono (adattate nel caso di insR che restituiva originariamente una FIFO e accoglieva sempre una FIFO come secondo parametro):

```
1) nodoF* eliminaR(nodo*&L, int x){
        if(!L)    return 0;
        nodoF* x=eliminaR(L->next, x);
        if(L->info == x){
                nodo*y=L;
                L=L->next;
                x->next=new nodoF(x->next, y);
        }
        return x;
}

2) nodoF* insR (nodoF* Q, nodo *x){
        if(!Q)    return new nodoF (Q, x);
        Q->next=insR(Q->next, x);
        return Q;
}
```

```
nodoF* tieni_primo_ric(nodo*& L){
        if(!L)    return new nodoF(0);
        nodoF *x=eliminaR(L->next, L-info);
        nodoF* lista=tieni_primo_ric(L->next);
        lista->next=insR(lista, x->info);
        return lista;
}
```

Le tre funzioni quindi cooperano con l'ultima che toglie, ordina e restituisce usando le altre due. La funzione di inserimento è la classica funzione ricorsiva: scorri finché ne hai e riesci ad inserire, altrimenti fermati perché non hai più nodi. L'altra funzione serve ad eliminare ricorsivamente un nodo uguale ad un altro con info == x, cosa che viene fatta ed eseguita quindi in tempo lineare.
Per la parte <u>iterativa</u> richieste due funzioni controparte di quelle viste qui sopra, quindi:

```
nodoF* tieni_primo_iter (nodo*& L){
        nodoF *x=0;
        nodo *aux=L;
        while(aux){
                nodoF*y=elimina(aux->next, aux->info);
                if(y)     x=insI(x, y);
                aux=aux->next;
        }
        return x;
}
```

e poi anche

```
nodoF *insI(nodoF *Q, nodo *x){
        if(!Q)    return new nodoF(Q, x);
        while(Q && Q->info < Q->next->info){
        Q=Q->next;
        }
        return Q;
}
```

<u>Affettare un array di interi A in liste L di dimensione dimA</u>

L=4->3->10->7->2->5->6->8->9->1
A=[2,2,1,0,3]
dimA=5

F1=4->3 (perché A[0]=2)
F2=10->7 (perché A[1]=2)
F3=2 (perché A[2]=1)
F4= nulla
F5=5->6->8

<u>Iterativa</u>

```
nodoL* affettaiter(nodo*&L, int*A, int dimA){
        nodoL * list=0;
        while(L && dimA){
                nodo*x=taglia(L, A[i]);
```

```
                L=x->next;
                x->next=0;
                if(x)   L=accoda(L, x);
                else    list=new nodoL(0);
                }
        }
        return list;
}

nodo* taglia(nodo*&L, int k){
        while(L && k>1){
                L=L->next;
                k=k-1;
        }
        return L;
}

nodo *accoda(nodo*L, nodo* x){
        while(L)
        L=L->next;

        L->next=x;
        return L;
}
```

Parte ricorsiva

```
nodoL* affettaric(nodo*&L, int*A, int dimA){

        if(!L || !dimA)   return 0;
        int current=A[i];

        if(current == 0){
                nodo*x=affettaric(L, A+1, dimA-1);
                x-next=0;
                return x;
        }
        else{
                nodo*x=taglia(L, *A);
                L=x->next;
                x->next=0;
                nodoL *y=affettaric(L, A+1, dimA-1);
                return y;
        }
}

nodo* taglia(nodo*&L, int k){
        if(k==0)        return 0;
        else return taglia(L->next, k-1);
}
```

Classica funzione di taglio, con una serie di elementi positivi, un array di interi non negativo da scandire. Si presuppone ci sia ed esista ogni singolo elemento fino al limite fissato dimA, tale che si possa eseguire il

taglio senza particolari problemi. Per il resto si tratta quindi di tagliare in partizioni a-esime a lista L fintanto che rimangono degli elementi.

Esercizio che considera array a 3 dimensioni avendo due funzioni (lettura e pattern match su array di interi P)

In questo caso, si parla di V-fette, che significa scorrere l'array a 3 dimensioni tenendosi la riga (lim2) e poi scorrere tutto a mo' di divisioni e mod, sapendo che in ogni operazione di queste aumenta l'indice (per esempio 0 mod 3=0, 1 mod 3=1, 2 mod 3=2, 3 mod 3=0, quindi inizia una nuova regione.

In questo caso, l'array è questo con lim1=3 (sono 3 strati), lim2=4 (sono 4 righe) e lim3=5 (sono 5 colonne).

0 0 1 0 1	0 0 0 2 1	1 1 2 2 0
1 0 0 1 1	0 0 0 2 2	0 0 0 0 0
0 0 2 3 1	3 3 1 0 0	2 2 2 0 3
3 1 1 2 0	2 2 2 2 2	1 2 2 3 4

e considera le V-fette in orizzontale sto giro, quindi come da colori sopra e
(0,1,0), (1,0,0), (0,3,2), (3,2,1)

Essendo 15 elementi, si nota come la divisione che sarà fatta in lettura per prendersi 3 elementi è data proprio da una divisione su lim1, che in questo caso è 3 (3 strati, cioè tre quadrati, intuitivamente).

Dovrà essere composta una funzione legge con segnatura:
int legge(int*A, int lim1, int lim2, int lim3, ifstream & IN)

PRE=(lim1, lim2 e lim3 sono valori positivi, A contiene almeno lim1*lim2*lim3elementi, lo stream IN contiene o -2 o almeno lim1*lim2*lim3 valori interi)

POST=(La funzione restituisce il numero di valori letti da IN(escludendo l'eventuale sentinella), i valori letti sono inseriti in A (visto come X[lim1][lim2][lim3]) per V-fette, secondo l'ordine per tasselli. La lettura si interrompe o quando viene letta la sentinella -2 oppure quando sono stati letti lim1*lim2*lim3 valori).

L'ordine per tasselli che cita è l'ordine delle V-fette tagliate e viste in orizzontale, come ho scritto sopra (quindi 0,1,0 ecc.). In realtà è simile alla solita roba.
La funzione legge fa uso di un'altra funzione che le taglia e che scrivo subito, che è:

```
coord calc_coord(int lim1, int lim2, int lim3, int VF, int n){
        A.s=[n%lim1];
        A.r=[n/lim1];
        A.c=VF;
        return A;
}
```

La funzione sopra calcola le v-fette, che possono essere anche solo parzialmente riempite, la solita fregatura. All'atto pratico, si considera che data la scansione in corso, possa finire prima del tempo.

Andiamo a scrivere *legge* (funzione iterativa), quindi:

```
int legge(int*A, int lim1, int lim2, int lim3, ifstream & IN){
        int i=0;  bool stop=false;
        while(i<lim1*lim2*lim3 && !stop){
                for(int VF=0; VF < lim3 && !stop; VF++)
```

```
                int valore;     IN << valore;
                        if(valore == -2)  stop=true;
                        else{
                                int slice=calc_coord(lim1, lim2, lim3, VF, n);
                                //vado a riempire A visto come A[lim1*lim2*lim3]
                                A[lim2*lim3*slice.s*lim3*slice.s*slice.c]=valore;
                                i=i+1;
                        }
                }
        }
}
```

Spiegazione semplice: lim2*lim3*strato perché scorro 20 elementi per prender il pezzo successivo (ad esempio 0,1,0 prima indice 0, poi indice 1 dopo 20 elementi e indice 2 dopo altri 20 elementi).
Poi per prendersi un elemento su un nuovo strato, considerato che scorro gli strati in orizzontale prendendo 3 elementi, allora devo usare lim3.

Se invece, per esercizio, avessi voluto prendere 0,1,0,3 come v-fetta come sempre fatto avrei dovuto fare: lim1*lim3*slice.s*lim2*r*c]

Nel ciclo si considera lim3 sulle V-fette per un motivo semplice: si noti lim1*lim3=15, quindi è una scansione che considera ogni possibile elemento e perché ci sono 5 colonne, quindi dovrò avere per forza 5 come limite finale.

Da un punto di vista di correttezza, l'invariante è il solito indice, questa volta tra lim1*lim2*lim3 e il fatto che venga considerata una scansione e taglio ove presenti le V-fette. Tagliata quindi la porzione a tasselli dell'array, si considera l'input che può essere 2 e il ciclo viene interrotto oppure diverso da 2 e si taglia la v-fetta andando a riempire A come se avesse 3 dimensioni con il valore preso in input.

Più interessante è la parte ricorsiva, funzione che cerca un match dei valori negli indici di P tra le v-fette, facendo in modo che se trovo un elemento matchato in un indice di P, esso prosegua anche sull'indice successivo.
Facendo un esempio pratico, supponendo di aver letto in X:
3 0 0 - 1 0 0 - 1 3 3 - 3 3 5 e 3 3 0 – 1 1 0 – (-2)
(avendo quindi la prima v-fetta completa e la seconda incompleta)

mentre P è: 1 0 0

Si nota come il match ci sia nell'indice 3 di v-fetta 0 e poi nell'indice 5 di v-fetta 1, come sopra.
La funzione *F1* che sarà generata deve stampare i 2 indici in cui si ha avuto la stampa, quindi 3 e 5.
Se il match si interrompe prima, si stampa quello che si è trovato, altrimenti si stampa -1 perché il match deve essere ricercato solo se si è trovato un match nella v-fetta precedente, proseguendo poi nella v-fetta +1; se c'è si va avanti, altrimenti si stampa -1.
Di fatto si controlla subito l'esistenza di un match; se il match esiste e prevede il proseguimento in un'indice vfetta+1, avanzo ricorsivamente altrimenti mi fermo. C'è poi una variabile *restoelem*, occasionalmente anche *nelem* negli esercizi del prof che indica il numero di elementi presenti nelle fette.

La funzione ricorsiva quindi avrà questa firma e corpo:

```
void F1(int*A,int lim1, int lim2, int lim3, int*P,int dimP, int restoelem, int inizio, int VF, ofstream & OUT){
        if(!dimP || restoelem <=0)      return;
        int matched=match(A, lim1, lim2, lim3, restoelem, inizio, VF);
        if(matched >= (lim1*lim2-1) )   //si vede che in una V-fetta completa ci sono 11 elementi
```

```
        //se vedo che finisce prima mi fermo e devo stampare “Fine”
        OUT << “Fine”;
        if(matched != -1)
        F1(A, lim1, lim2, lim3, P+1, dimP-1, restoelem-(lim1*lim2)-(match+1-inizio), inizio, VF+1, OUT);

        //nella scans. precedente si fa restoelem – (lim*lim2) perché sono quelli gli elementi da scorrere
        //nel match di una v-fetta sottraendo poi il punto di inizio di un match
        //e incrementando la v-fetta
        //altrimenti ritorno la stessa funzione ma avanzando solo di una v-fetta

        else
        F1(A, lim1, lim2, lim3, P+1, dimP-1, restoelem-(lim1*lim2), inizio, vf+1, OUT);
}
```

A questo punto vado a cercarmi il match con la stessa tecnica di prima dentro A, controllando sempre ci siano abbastanza elementi e che l’indice iniziale di match stia dentro una v-fetta (quindi sempre lim1*lim2-1, esattamente 11 elementi)

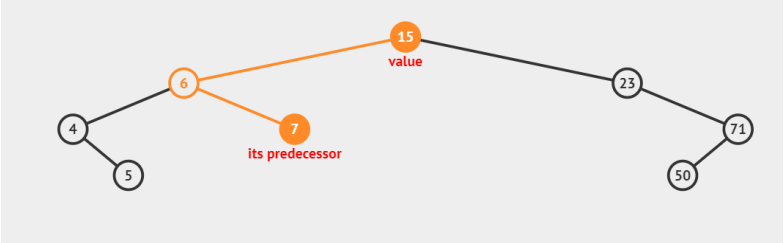
```
int match(int*A, int lim1, int lim2, int lim3, int *P, int restoelem, int inizio, int VF){
        if(restonelem <=0 || inizio > lim1*lim2-1)        return 0;

        int valore=calc_coord(lim1, lim2, lim3, VF, inizio);
        if(A[lim2*lim3*valore.s+valore.r*valore.c*lim3]==*P)
                return inizio;
        else
                return match(A, lim1, lim2, lim3, *P, restpelem-1, inizio+1, VF};
}
```

Concludendo, il problema qui principale è capire dove ci troviamo, con alcuni accorgimenti e poi la funzione si scrive; bisogna entrare nell’ottica e credo che questo esercizio faccia capire bene.
Il resto sono solo alcuni commenti di contorno; assumendo il fatto che la funzione ausiliaria calc_coord mi trovi il valore da cui ricavarmi il corrispondente elemento in A, controllo che sia una v-fetta disponibile. Se sì, allora, ritorna il punto iniziale di match, quindi l’indice, ricorrendo poi a partire da questo altrimenti ritorno semplicemente -1. Chiaro quindi che se esiste un elemento e una v-fetta grazie ai controlli dati induttivamente esiste l’elemento dopo e induttivamente ricerco il match diminuendo la complessità del problema sempre più, altrimenti termino e rispetto sempre POST.

<u>Trovare successore e predecessore in alberi BST</u>

Sappiamo la proprietà dell’albero BST, cioè una chiave rispetto alla quale inserisco a sinistra gli elementi minori e a destra gli elementi ad essa maggiori, ricorsivamente di solito. In realtà il fatto di avere un BST permette una risoluzione iterativa del problema abbastanza agevolmente.

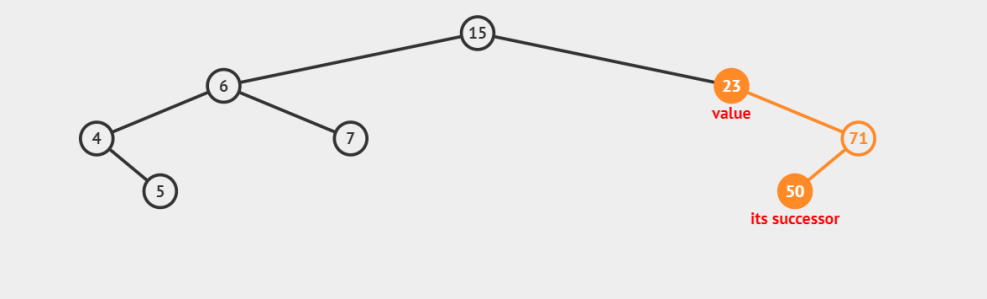
Un commento interessante nei BST è una funzione che trova il predecessore di un nodo, cioè quello che viene stampato prima del nodo attuale.



Ad esempio rispetto al BST sopra, 7 è predecessore di 15 (altro esempio, rispetto al valore 6, 5 sarà il predecessore, rispetto sempre ad un classico ordine prefisso dell'albero almeno dal punto di vista di stampa).
Per ogni nodo quindi controllo se il nodo ha un figlio a sinistra, se lo ha cerco a destra per trovare il valore più grande. Se non ha figli a destra, il valore predecessore è lui, perché in un BST lui è il valore più grande. Il ciclo mi permette di determinare se il nodo attuale era il predecessore, se non lo era, troviamo che il predecessore è il più grande tra gli elementi non a sinistra.
Soluzione iterativa, ricorsiva segue lo stesso principio perché cambia molto poco.

```
nodo *predecessor(nodo*T){
        while(T){
                nodo*parent=T;
                if(T->left)         return max(T->left);
                else{
                        nodo*prec=parent;
                        while(prec !=nullptr && prec == T->left){
                                nodo*old=p;
                                T=parent;
                                parent=old;
                        }
                if(p)   return p;
                else    return nullptr;
                }
        }
}

nodo* max(nodo *T){
        while(T->right)
        T=T->right;
return T;
}
```

Essendo quindi un BST, trovare il massimo significa semplicemente scorrere a destra; in un albero normale significa invece scorrere a destra e sinistra ricorsivamente e confrontare ogni volta chi è il minore, ritornandolo. Lo stesso vale per il maggiore.



Sopra invece un esempio del successore di 23, che è 50, con apposito algoritmo iterativo.
Il ragionamento è quasi identico a quello sopra, controlliamo chi è il nodo minore rispetto al nodo a destra e confrontiamo nella posizione destra attuale se ho un nodo a sinistra, se lo ho quello è il successore, altrimenti avevo un successore in precedenza e lo confronto alla stessa maniera di prima. Se avevo prima un successore, avanzo ricordando che il vertice attuale sarà il genitore, quindi ricerco di nuovo un valore minimo e proseguo finché ho un valore diverso, necessariamente minore a sinistra. Quello sarà il successore. Se non ne ho, termino.

```
nodo *successor (nodo*T){
        while(T){
                nodo*parent=T;
                if(T->right)        return min(T->right);
                else{
                        nodo*prec=parent;
                        while(prec !=nullptr && prec == T->right){
                                nodo*old=p;
                                T=parent;
                                parent=old;
                        }
                if(p)   return p;
                else    return nullptr;
        }
}
```

Pattern matching tra un albero BST e un array P

Sappiamo la proprietà dell'albero BST, cioè una chiave rispetto alla quale inserisco a sinistra gli elementi minori e a destra gli elementi ad essa maggiori, ricorsivamente di solito. In realtà il fatto di avere un BST permette una risoluzione iterativa del problema abbastanza agevolmente.
Supponendo per esempio di avere R albero BST e P come pattern sotto riportati:

R=[20]([10]([5]([0](_,_),[12](_,_)),[15]([13](_,_),_)),[30]([25]([22](_,_),_),[35]([33](_,_),_)))
P=[25, 22, 35, 38]

È una buona rappresentazione dell'albero senza doverlo rappresentare graficamente (lunghina da fare, per esempio si capisce che se ho un parentesi a sinistra ho un nodo sotto, tipo 10 sottoalbero sinistro di 20, 5 sottoalbero sinistro di 10 e così via).
Per esempio 0 non ha elementi né a destra ne è a sinistra, foglia quindi.
Nel qual caso è richiesta una funzione ricorsiva che attraversa in ordine prefix l'albero e cerca il match.

PRE=(albero(R) è ben formato, profondita>=0, dimP>=0 e P ha dimP elementi)
nodoP* match(nodo*R, int profondita, int*&P, int & dimP)
POST=(restituisce una lista di nodoP che corrisponde al massimo match di P nei nodi di R esaminati in ordine prefisso, la lista, P e dimP sono come descritti prima)

```
nodoP* match(nodo*R, int profondita, int*&P, int & dimP){
        nodoP*list=0;
        if(!R)  return new nodoP(0);
        else{
                if(R->info == *P){
                        P=P+1;
                        dimP=dimP-1;
                        list=new nodoP(R);
                }
                else if(R->info > *P)
                        L=conc(L, match(r->left, profondita+1, P, dimP);
                else if(R->info < *P)
                        L=conc(L, match(r->right, profondita+1, P, dimP);
                return L;
        }
}
```

La funzione *conc* viene messa a disposizione a priori e viene usata per concatenare le parti dove esiste match; la proprietà BST è utile poiché mi permette di esaminare l'albero meno volte, controllo più agilmente grazie alla proprietà considerata rispetto alla chiave. Non era richiesto dal prof il discorso di predecessore e successore, ma utile nella trattazione della struttura e presente sulle slide, oltre ad altre funzioni standard quali calcolo dell'altezza (conto quanti elementi ci sono a sinistra e destra e confronto, aggiungendo uno nel caso uno dei due pezzi sia vuoto), trovare minimo e massimo (come detto cambia se albero binario o BST), inserimento e/o cancellazione di un elemento (si tratta di capire sempre se binario o BST, comunque si procede ricorsivamente e individuato l'elemento si manipola la lista ritornando all'elemento corrente), e la stessa funzione di concatenazione (inserisco alla fine dopo aver scorso tutta la lista/albero, discorso rimane simile per entrambi, come anche inserimento e/o cancellazione).

Da un punto di vista di correttezza, il programma ragiona in tre modi: non ho più l'albero, ritorno l'unico nodoP vuoto, oppure ho info==nodo di riferimento e a questo punto avanzo mettendo nella struttura supplementare ogni singolo nodo, ciò fino alla fine. Essendo albero BST ho quindi la possibilità di andare verso sinistra, concatenando (quindi semplicemente inserendo l'elemento i-esimo alla coda della nodoP) il nodo di riferimento finché esiste e poi a destra, con simile procedimento ricorsivo.
Tutto ammesso la validità e presenza dei singoli nodi.

Pattern matching tra albero binario t e array di interi P

Abbiamo:
t= 4(6(2(3(_,_),1(_,_)),4(_,_)),5(8(4(_,_),_),9(_,11(12(_,_),_))))
P=[5,9,11]

Nella percorrenza dell'ordine prefisso sull'albero, deve essere riempito un array *path* che ha un numero di elementi maggiore della profondità di t (per forza, perché dovrà contenere almeno tutti gi elementi).
L'array path codifica in ogni momento tutti i nodi info dell'albero percorso.
Prendiamo ad esempio il cammino 000 (tutto a sinistra); l'array path sarà riempito degli elementi 4,6,2,3 come si vede sopra. Il cammino dopo sarà 001e i gli info saranno 4,6,2,1. Si prosegue così e path dovrà contenere tutti i nodi info dalla radice fino ad una foglia.
Si va avanti finché non si trova il match.

Abbiamo quindi una simpatica funzione ricorsiva:

PRE=(albero(t) è ben formato e non vuoto, P ha dimP>0 posizioni, depth >=0 e path ha più di depth + altezza(t)+1posizioni)
bool prefix(nodo* t, int*P, int dimP, int* path, int depth)
POST=(se in t non esiste alcun cammino dalla radice ad una foglia che contiene un match diP, allora prefix restituisce false, se invece un tale cammino esiste, chiamiamo c quello più a sinistra, allora prefix restituisce true e a partire da path[depth], path contiene i campi info dei nodi attraversati dalcammino c)

Andiamo quindi a scrivere la funzione:

```
bool prefix(nodo* t, int*P, int dimP, int* path, int depth)
{
        if(!t)     return false;
        path[depth]=T->info;
        if (t->left) return prefix(t->left, P, dimP, path, depth+1);
        if (t->right) return prefix(t->right, P, dimP, path, depth+1);
        //quando sono in una foglia, sono alla fine e qui potrei fermarmi ma cerco il match
        return match(path, depth+1, P, dimP);
}
```

Normale funzione di percorrenza dell'albero, unica cosa da commentare la presenza della variabile *depth* che può essere utilizzata come indice dell'info dell'albero valorizzato in quel momento.
Comunque sia, *path* viene valorizzato dalla radice ad una foglia.

Si chiede inoltre di definire una funzione iterativa che soddisfi la seguente specifica:
PRE=(P ha dimP >0 posizioni, path ha dim posizioni)
bool match(int* path, int dim, int*P, int dimP)
POST =(restituisce true sse path[0..dim-1] contiene un match contiguo e completo di P[0..dimP-1].
E dunque:

```
bool match(int* path, int depth, int*P, int dimP){
        if(!dimP || depth < dimP)       return false;
        while(i < depth && (depth – i) >= dimP){
                if(path[i] == P[j]){
                        i++;
                        j++;
                        if(j==dimP) return true;
                }
        }
        return false;
}
```

Si consideri che la sottrazione che viene fatta *depth-i* viene eseguita perché l'array P può essere considerato l'array da cui estrarre un sottomatch rispetto a path, quindi viene così ritrovato.
Per il resto direi un problema molto affrontabile, leggermente modificato rispetto alla soluzione presente per ottimizzazione di codice e resa funzionale della procedura descritta.

Pattern matching completo non contiguo di un pattern in un testo (si considerano buchi tra i match e funzione di inizio e fine match)

Abbiamo:
T=[0,2,1,0,2,0,1] e P=[0,2,0]
Secondo l'esercizio vi sono due match, il primo [0,1,3], modo poco utile di indicare che il match completo esiste sugli indici 0, 1 e 3 di T e poi sugli indici 3, 4 e 5 di T.
Si indica un bilancio con questa formula [(fine_match – in_match)+ 1 – dimP] che ad esempio nel caso del primo:
(3-0)+1-3 quindi l'ultimo indice di match meno il primo indice su cui si ha avuto match+1-dimP
(5-3)+1-3 nel caso 2 invece
La funzione da scrivere è iterativa e si deve restituire l'intero x che indica l'esistenza di un match ed è il bilancio minimo; se ha un valore lo si ritorna, altrimenti -1 equivale a dire "non ci stanno match".
Dovrà anche usare una funzione ausiliaria comunque presente al di sotto della funzione principale.
Quindi:

```
int match(char *T, int dimT, char *P, int dimP){
        int x=-1;
        int i=0, j=0;
        while(i < dimT && j < dimP){
                bool value=M(T, dimT, P, dimP, i, fine);
                if(value){
                        int bil_loc=(fine – i) + 1 - dimP;
                        if( x == -1 || bil_loc < x)          x=bil;
                }
        }
        return x;
```

```
}

bool M(char *T, int dimT, char *P, int dimP, int inizio, int fine){
        bool go_on=true;
        for(int i=inizio, int p=1; i<dimT && p <dimP && go_on; i++){
                if(T[i]==P[p]){
                p++;
                fine=i:
                }
                if(p==dimP)     go_on=false;
        }
        return go_on;
}
```

Per la parte <u>ricorsiva</u> viene richiesto di fare le stesse funzioni ma ricorsivamente. Definiremo PRE e POST di queste e inoltre la dim. induttiva di MR.

```
int matchR(char *T, int dimT, char *P, int dimP){
        if(!dimT || !dimP)      return 0;
        if(dimT < dimP)         return 0;

        bool value=M(T, dimT, P, dimP, i, fine);
        int x=matchR(T, dimT, P, dimP, inizio+1);
        int y=fine – inizio + 1 – dimP;
        if(value && x>y)
        return x;
        else    return y;
}

bool MR(char *T, int dimT, char *P, int dimP, int inizio, int fine){
        if(T[inizio] != *P)         return return MR(T, dimT, P, dimP, inizio+1, fine);;
        if(dimP)
                return MR(T, dimT-1, P+1, dimP-1, inizio+1, fine);
        else{
                fine=inizio;
                return true;
        }
}
```

L'invariante R da sopra è il seguente:
0<i<dimT && 0<j<dimP && go_on==true sse dimP esiste e non ho un match completo.

Per gli altri casi abbiamo PRE che considera come sempre tutti gli elementi definiti e maggiori di 0 se numeri o comunque di dimT o dimP elementi nel caso degli array. L'unica cosa è la variabile *inizio* che discrimina se avanzare subito tra gli indici dell'array considerato se ci sia o meno un match e permette di capire se ci sta o meno un match.
Ad ogni modo in MR semplicemente con la variabile discussa si capisce come muoversi, quindi se andare avanti ricorsivamente in T oppure se terminare la ricorsione al passio i-esimo fino a un massimo di dimP-1 ricorsioni, terminando e ponendo la variabile fine=inizio, quindi indice spostamento del match o ritornare gli array spostati nei loro singoli elementi.

Pattern matching di array a 1 dim, visto a 2 dimensioni tra T e P riempiendo un array Q scorrendo riga per riga

Solito problema discusso varie volte, ma sempre attuale direi e con una variante che merita di essere considerata.
Andiamo al sodo e rappresentiamo gli input:

T=[0,0,0 1,1,2 2,3,0 0,0,0 0,4,5] con R=5 e C=3 (5 righe e 3 colonne)
P=[1,2,3,4,5,6]

Si esamina riga per riga e si compone in output un array Q che sarà dato da [0,1,2,0,2]
Significa che:
-sulla prima riga (0,0,0) non ci sono elementi che fanno match
-sulla seconda riga solo il primo elemento di P fa match, quindi 1
-sulla terza riga abbiamo 2 elementi che fanno match (che sono 2,3)

Capito come si compone, può esserci un ipotetico caso di Q con tutti 0, naturalmente.
La cosa da considerare nell'esercizio è che se si trova un match si continua a cercare sulla stessa riga (riga i-esima) altrimenti si passa a quella successiva.
Iterativamente, vengono richieste due funzioni sotto svolte. *matchR* come da nome serve a cercare il match in una riga; restituisce un intero che sarà utilizzato nello spostamento alla riga successiva e cercare il match da essa, controllando di non aver oltrepassato i possibili elementi in una riga.

```
void F(int *T, int R, int C, int *P, int dimP, int *Q){
        int moved=0;            //la chiamo così e mi sposto per poter prendere la riga giusta
        for(int i=0; i<R && dimP>0){
                for(int j=0; j<C; j++){
                    int x=matchR(T+i*j, R, P+moved, dimP-moved);
                    Q[i]=x;
                    moved += x;     //trovato il match mi sposto con gli x elementi trovati prima
                }
        }
}
```

Chiaro come PRE qui sia il fatto di avere gli array riempiti e corretti (quindi non vuoti), comprendendo anche Q, al momento della precondizione vuoto.
Nella POST dovrà quindi essere riempito come da specifica, senza tanti giri di parole.
Si passi alla funzione sotto riportata, col compito (POST) di calcolare match in una riga, che se trovato si sposta alla riga successiva (i+1) e così via, fino alla fine. Classico ragionamento induttivo.

```
int matchR(int*riga, int dimR, int*P, int dimP){
        int matched=0;
        for(int i=0; i < dimR && dimP != 0; i++){
                if(riga[i]==P[i]){
                        P=P+1;
                        matched++;
                        dimP=dimP-1;
                }
        }
        return matched;
}
```

Chiaro come la dimostrazione induttiva possibile consideri entrambe le situazioni, quindi la sola attenzione si rivolge a *moved*, variabile che qualifica il possibile spostamento nei dimP elementi > 0 per proprietà di PRE, ammettendo l'esistenza della riga attuale e ammettendo il rientro nelle righe attuali.
Risolviamo la stessa cosa ricorsivamente (in questo caso *FR* viene già data dal prof ed è sotto riportata; si tratta di scrivere *matchRR*)

```
void FR(int *T, int dimT, int C, int *Q, int *P, int dimP){
        if(!dimT || !dimP)        return 0;
        int x=matchRR(T,C,P,dimP);
        *Q=x;
        FR(T+C, dimT-C, C, Q+1, P+x, dimP-x);
}
```

Trovo comunque utile osservare lo spostamento che viene eseguito, per C elementi rispetto a T, possibile per caso ricorsivo, ma volendo anche iterativo.

Passiamo quindi a matchRR che, dice il prof, deve invocare almeno due funzioni ausiliarie (una dovrà trovare il match a partire dalla prima riga e l'altra determina quanto lungo è il match).
Chiaro quindi che nell'analisi dobbiamo prima capire se inizia un match dalla riga attuale, altrimenti in FR si ritorna 0; altrimenti si trova il match dall'inizio, si determina una volta trovato che esiste la lunghezza e il resto da ritornare.

```
int matchRR(int *T, int C, int*P, int dimP){
        int start=find_starting_match(T, C, *P);

        if(start < C)
        return 1+link_matched(T+start+1, C-inizio-1, P+1, dimP-1);
        else
        return 0;
}

int find_starting_match(int *T, int C, int *P){
        int x=0;

        if(*T != P[0]){
        x=find_starting_match(T, C, P);
        return x;}
        else
        return x < C ? x : 0;
}

int link_matched(int *T, int C, int *P, int dimP){
        if(!C || !dimP)   return 0;

        if(*T == *P)      return 1+link_matched(T+1, C-1, P+1, dimP-1);
        else      return 0;
}
```

Quindi questo è un match ricorsivo, quindi se esiste l'indice di match primario, che vuol dire avere un punto di partenza del match, lo restituisco e da questo mi sposto per poter cercare un effettivo match tra testo e pattern, incrementando induttivamente ad ogni passo. Nel caso dell'indice iniziale ne ammetto l'esistenza rispetto al primo elemento e continuo a cercare, incrementando/decrementando del valore x per eseguire

gli spostamenti giusti. Una volta realizzati e ammessa l'esistenza del match, continuo fino a dimP elementi, naturalmente se possibile entro i bound (dimensioni array e compagnia).
I casi base sono come sempre la non esistenza di dimP, ma in questo caso anche che non ci siano colonne (C); nel qual caso ritorno 0. In tutti gli altri casi, passo allo step successivo fino alla fine, soddisfando la postcondizione.

Array con sequenze, sottosequenze e valori di input

In questo caso diverso da altri problemi trattati, dobbiamo considerare un array formato da varie sottosequenze (che terminano con -1) e tutto l'array termina con -2.
Un esempio è: 2 3 -1 0 2 3 -1 2 3 -1 0 2 -1 -1 2 3 0 -1 -2

L'uso dei colori visualizza meglio questo array chiamato C.
Per la funzione iterativa, si tratta di scrivere una funzione che alloca dinamicamente un array X e nella variabile lim risulta essere il numero di sottosequenze presenti. Dato inoltre che non si sa a priori quanto lim possa crescere, il prof richiede che IC allochi inizialmente un array di 20 posizioni ma che sia in grado, se lim diventa maggiore di 10, di allungare X di ulteriori 20 posizioni, così anche se X supera 20, quindi altre 20 posizioni… dunque, ogni 10 valori di lim si vuole l'allungamento di altre 20 posizioni..
La funzione richiesta è la seguente, con PRE e POST di fatto date dalla mia spiegazione. Si segnala come consiglio da parte di Filè funzioni iterative ausiliarie, direi effettivamente utili, una magari per allungare l'array e l'altra per trovare la lunghezza di una sottosequenza, sommando all'indice di scorrimento la lunghezza e la posizione di c per poi procedere a trovare quella successiva e poi fermandosi a -2.

Dunque:

```
int * IC(int*C, int & lim){
        int i=0;
        int X=new int[20];
        int dim=20, pos_c=0, pos_x;

        while (C[i] != -2){
        if(lim*2 >= dim)
        allunga(X, dim);

        int sub_s=conta(C, pos_c);      //sub_s sta per subsequence, secondo me nome abbastanza chiaro
        X[pos_x]=i;
        pos_x++;
        X[pos_x]=sub_s;
        i+=sub_s + 1;
        lim++;
        }
}

int conta(int *C, int pos_c){
        int i=0;
        while(*C != -1){
                pos_c++;
                i++;
        }
        return i;
}
```

```
void allunga(int *X, int &dim){
        int *aux=new int[20];
        for(int i=0; i<dim; i++){
                aux[i]=x[i];
        }
        delete [] X;
        dim+=20;
        X=K;
}
```

I cicli in questa funzione hanno come condizione di invarianza il fatto di avere un array dinamico, possibilmente illimitato nel quale trovare delle specifiche sottosequenze di valori; in un caso -2, mi accingo a trovare una singola sottosequenza terminante con -1 e salvo in due posizioni di X la posizione di inizio della sottosequenza, determinato da variabile locale e poi la lunghezza, andando avanti di volta in volta di due posizioni. Per questo fatto, la condizione di allunga dell'array viene fatta a multipli di i*2 (quindi ogni 10*i e raddoppio).
Dimostrando quindi la correttezza, noto come al di là di questa condizione, io riesca a carpire ogni sottosequenza che non termini prima di -1, per qualsiasi lunghezza possa avere, allungando poi l'indice i della lunghezza della sottosequenza per portarsi subito ad esaminare quella successiva.
In tutto questo la variabile *lim* aumenta a segnalare il numero di sottosequenze, che chiaramente devono essere prima scandite e calcolate sulla base delle osservazioni considerate.

Per il pezzo ricorsivo, invece, usando C e X, si vuole costruire un array Z di lim interi tale che Z[i] contenga il numero di sottosequenze presenti in C.

Spiego meglio il giro confuso che come sempre ne vien fuori, cercando di renderlo umanamente comprensibile per noi poveri cristi.

Usando C=2 3 -1 0 2 3 -1 2 3 -1 0 2 -1 -1 2 3 0 -1 -2 e lim=6.
Quelle evidenziate in giallo sono le sottosequenze, che sono 5 ma si considera anche una sottosequenza vuota che sarà in posizione 4 in questo caso, 6 in totale, da cui spiegato il valore di *lim*.
L'array Z richiesto dovrà essere Z=[2,2,2,1,0,3].
Per esempio Z che avrà valore 2 indica che la sottosequenza 0 (quindi 2,3) in C avrà 2 sottosequenze, quindi la sottosequenza formata da 2 e la sottosequenza formata da 3. Il prof fa notare come la sottosequenza di indice 0 sia uguale a quella di indice 2, poiché avrebbe un'influenza sulla funzione da fare, che riporto con prototipo sotto; PRE e POST fanno solo ulteriore confusione e risparmio la loro trattazione.
In parole umane, si scorre tutto l'array C determinando le singole sottosequenze e quando si confrontano 2 sottosequenze identiche, pur essendo distinte, si deve considerare.

Abbiamo quindi la funzione sotto; dobbiamo quindi utilizzare X per costruire Z e viene consigliato l'utilizzo di funzioni ausiliarie sempre ricorsive. Sulla base di quanto detto, una funzione ausiliaria scorrerà tutto fino a lim dato e l'altra confronterà le singole sottosequenze.

j qui è l'indice che scorre fino a lim-1

```
void M(int*C, int*X, int lim, int j, int *Z){
        if(j >= lim-1)      return;
        aux_M(C, X, lim, j, j+1, Z);
        M(C, X, lim, j+1, Z);
}
```

Qui scorro Y vedendo la lunghezza dei singoli pezzi e poi confrontandoli dentro C che è il riferimento base.
L'idea è di scorrere l'array X sulla base dei due indici precedenti, scorrendoli come sottoinsiemi. Se uguali, si

scorrono entrambi gli indici, altrimenti se prevale l'indice i si scorre questo come sottosequenza, altrimenti si scorre j.
Nella funzione chiamata *compare* (confronta/paragona), andrò effettivamente a vedere se vi è una somiglianza utile all'interno dell'array C sulla base degli indici presenti, altrimenti scorro ricorsivamente considerando sequenza più corta e più lunga.

```
void aux_M(int*C, int*X, int lim, int i,  int j, int *Z){
        int k=0;
        if (j==lim)         return;
        if(X[2*i+1] > X[2*j+1]){
                if(compare(C, X, i, j, k))
                Z[i]++;
        }
        if(X[2*i+1] == C[2*j+1]){
                if(compare(C, X, i, j, k))
                Z[j]++;
                Z[k]++;
        }
        if(X[2*i+1] > X[2*j+1]){
                if(compare(C, X, i, j, k))
                Z[j]++;
        }
        M1(C, X, lim, i, j+1, lim, Z);
}

bool compare (int *C, int *X, int i, int j, int k){
        if(k == X[2*i+1] || k==X[2*j+1]  return true;
        if(C[X[2*i]+k]==C[X[2*j]+k)
        return compare (C, X, i, j, k+1);
        else return false;
}
```

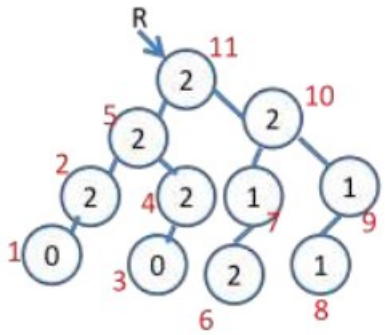
La contorsione logica del prof è sempre pesante e anche qui niente male, devo dire; al nocciolo, esaminato questo codice che è effettivamente leggermente semplificato sulla base di quello del prof (incomprensibile personalmente arrivare ad una qualunque ipotesi), è meno contorta ma comunque problema ridicolo come sempre. Ad ogni modo, si ragiona con array dentro altri array sulla base dei due indici presenti. In questo caso, Z è l'array da riempire sulla base degli indici di confronto in quel momento, scegliendo i come indice per la sottosequenza i di scansione precedente e j come sottosequenza di scansione successiva. Se hanno la stessa grandezza, incremento entrambi gli indici, altrimenti se maggiore la sottosequenza di indice i aumenta questa ultima, in maniera tale da valorizzare Z dando il maggior numero di sottosequenze presenti. Da un punto di vista induttivo, vado avanti 2*indice di riferimento per potermi correttamente inquadrare all'interno della sottosequenza non vuota (per questo poi +1), rispetto agli indici i e j. Ciò è verificato fintanto che si aumenta j ricorsivamente fino al limite dato lim.
Nella funzione *compare*, come richiesto dal prof, si cerca di confrontare le singole sottosequenze di indici dati, determinando che se due sottosequenze sono uguali come indice (quindi k che è la sottosequenza tagliata di indice attuale rispetto a quella più corta, di indice i, o quella più lunga, di indice j) si ritorna true, altrimenta si incrementa l'indice k che esamina i singoli sottocasi, ritornando false qualora falsa la condizione. Si consideri questa mezza spiegazione/intuizione la dim. di correttezza di questo "meraviglioso" esercizio.
La scrittura iterativa di questa funzione, semplicemente, cambia le ricorsioni con degli while, in quanto ricorsione terminale e quindi sostituibile facilmente (ricordiamo che la non terminal non è altrettanto facile da sostituire con un while, ma magari con funzioni più elaborate).

Il problema trattato ora sopra risale ad un appello del 2014, facente parte dei temi diversi rispetto ad array a 3 dimensioni, pattern matching, alberi, liste.

Funzione ricorsiva che da un albero binario e un k > 0 produce una lista concatenata seguendo l'ordine postfisso e cancellazione dei nodi ripetuti da esso

Tra gli appelli ed esercizi, inserisco anche questo, nella trattazione originale comprendente la struttura FIFO, ma che noi trattiamo come nodoE, per allenamento. Una precisazione dovuta è il fatto di dover prendere i nodi che sono nelle posizioni k, 2*k, 3*k, ecc.
Quindi ad esempio, se k=2, prenderò i nodi nelle posizioni 2,4,6,8,10 in ordine postfisso, ad esempio, si veda l'immagine sotto riportata.

dato l'albero



Il valore n deve essere anche utilizzato per poter creare alla fine della ricorsione un nodo alla fine della lista che stiamo costruendo; per esempio, prendendo il caso di prima, il nodo 10, alla fine della ricorsione torna al nodo 11 (radice come si nota da sopra) ed n=1, indicando che è il nodo successivo incontrato seguendo l'ordine postfisso.
Come PRE avremo il fatto di avere un albero corretto, n definito tra 1 e K e vn=n, mentre come POST la creazione di un nodoF corrispondente all'ordine postfisso dell'albero seguendo la posizione vn+k, vn+2*k, ecc.
Prototipo della funzione e sua implementazione segue:

```
nodoE* pickric_postfix(nodoE* R, int &n, int k){
        nodoE*lista=0;
        if(!R || n==k)    return new nodoE(0);
        if(R->left){
                n=n+1;
                lista->next=pickric_postfix(R->left, n, k);
        }
        if(R->right){
                n=n+1;
                lista->next=pickric_postfix(R->right, n, k);
        }
        return lista;
}
```

Nella funzione precedente, quindi, è un semplice spostamento sulla base di quello che consta dall'albero nell'ordine postfisso, tenendo traccia con n dello spostamento e ritornando la lista; se fosse stata con FIFO, poco cambiava, si inseriva una semplice *push*_end. Voglio ora cancellare i nodi ripetuti dalla lista attuale. svolgendo questa operazione in modo iterativo.

```
nodoF* tieni_primo(nodo*&L){
        nodoF* lista=0;
        nodo*prec=L, *succ=L, curr=L;
        while(prec && curr != 0){
                succ=prec;
                curr=prec->next;
                if(curr->info == prec->info){
                        succ->next=curr-next;
                        curr->next=0;
                        lista->next=new nodoF(lista, curr);
                        curr=succ;
                }
                succ=curr;
                curr=curr->next;
                prec=prec->next;
        }
        return lista;
}
```

Tutta la funzione sopra è una manipolazione iterativa sui nodi; non avendo un nodo x a cui confrontare L, bisogna operare con almeno tre nodi, in quanto con 2 sarebbe inutile e si considera una sola lista. Fondamentalmente, il nodo *prec* serve a capire il punto di partenza e in ogni momento *succ* serve a capire il nodo successivo; nel primo pezzo, per portarsi al nodo attuale, si eguagliano i due nodi e il nodo corrente sarà il successiuvo al nodo precedente; se i nodi del precedente e del successore si eguagliano, dovrò andare oltre al nodo attuale, mettendolo dentro *lista* e poi eguagliando i due nodi presenti (successore e copia del nodo attuale), andando avanti sia nella copia del precedente che del successivo.
Quindi servono entrambi per capire che nodo precede quello attuale, che diventerà il nuovo nodo attuale e che nodo succederà il nodo attuale, che sarà il nuovo link della lista attuale.

Data la spiegazione intuitiva della funzione, per invariante, essa ammette sicuramente l'esistenza di L e dei nodi copia che permangono fintanto che la lista esiste e le sue copie, perlomeno il nodo radice/genitore/precedente. Pertanto, se trovo due nodi uguali, induttivamente, avanzo scambiando copia del nodo attuale con quello successivo, altrimenti proseguo nella scansione operando uno scambio tra nodo successivo e precedente e scorrendoli entrambi secondo la specifica. Questo basta nella correttezza, naturalmente si adopererebbero termini più formali, ma non troppo dissimili da questi.

Aggiunta di nodi in fondo ad un albero binario verificando se cammino troppo lungo/troppo corto/se va bene

Senza fare esempi contorti, prendiamo il problema dentro l'albero: 00-101-1101-1
Ogni cammino termina con la sentinella -1. Potremo aggiungere nodi solamente ai cammini con una lunghezza determinata in input, cioè x, in questo caso nei cammini (00-1) e in (01-1), non in (101-1), in quanto troppo lungo nella considerazione dell'esempio.

Qui sono da fare tre funzioni:
-due funzioni di aggiunta del nodo nell'albero (*add*, una iterativa ed una ricorsiva)
-una funzione chiamata *inizio_cam* (si può decidere se farla iterativa oppure ricorsiva) che considera C (che contiene n cammini, terminanti con -1 e la fine di tutti i cammini, terminata da -2) e restituisce l'indice in cui inizia il cammino i.

Due cose da dire, si consiglia la realizzazione della parte ricorsiva prima di quella iterativa (caso add_ric) e nelle *add* se w (cammino che si forma) è troppo corto rispetto a vr (r originale, che sarebbe l'albero), si stampa "cammino troppo corto" oppure "cammino troppo lungo" seguito da x e dipende dal caso.

A questo punto, scriviamo del codice.

```
void add_iter(int* C, nodo*& r, ifstream & INP, ofstream & OUT){
	if(!r){
		if(*C == -1){
			r=new nodo(r, 0);
			INP>>"Inserisci nodo: ">>r->info;
		}
		else{
			int x;	INP>>"Inserisci la lunghezza: ">>x;
			OUT<<"Cammino troppo lungo";
		}
	}
	while(*C !=-1 && !found){
		nodo* prev=r, *next=r;
		if(next){
			if(*C==0){
				prev=next;
				prev=prev->left;
			}
			else{
				prev=next;
				prev=prev->right;
			}
			found=true;
		}
		if(found == true)		OUT<<"Cammino troppo lungo";
		if(next){
			int x;	INP>>"Inserisci la lunghezza: ">>x;
			prev->left=new nodo(x, 0);
			OUT<<"Cammino troppo corto";
		}
		else{
			int x;	INP>>"Inserisci la lunghezza: ">>x;
			prev->right=new nodo(x, 0);
		}
	}
}

void add_ric(int* C, nodo*& r, ifstream & INP, ofstream & OUT){
	if(!r){
		INP>>r->info;
		r=new nodo(r,0);
	}
	else{
		INP>>"Inserisci nodo: ">>X;
		OUT<<"Cammino troppo corto";
	}
if(*C==0)
```

```
add_ric(C+1, r->left, INP, OUT);
else
add_ric(C+1, r->right, INP, OUT);
        if(!r && *C==-2){
                INP>>"Inserisci nodo: ">>X;
                OUT<<"Cammino troppo lungo";
        }
}
```

Questo è il classico esempio di esercizio alla Filè: molto fattibile dopo averlo visto/intuito, incomprensibile a leggere il testo. Sostanzialmente io lo semplifico, ma leggendo la sua funzione non si capisce che alla fine si deve usare l'array del cammino per capire dove siamo e inserire correttamente il nodo sulla base di quello, 0 a sinistra e 1 a destra; nel caso della funzione iterativa importante capire dove siamo con due rispettivi puntatori ai nodi, precedente e successivo come nell'esercizio prima. Da segnalare anche come "troppo corto" e "troppo lungo" in realtà segnalano se siamo in un cammino vuoto (corto per forza) oppure in un cammino con più nodi del previsto (uso controllo del booleano nella iterativa).

Dal punto di vista della correttezza, si richiede una prova induttiva su add_ric, quindi:
-caso base, *C==-1, siamo alla fine e il cammino, sulla base di quanto detto e riscontrato è corto
-caso ricorsivo, il cammino si dovrà determinare a seconda dell'elemento i-esimo di C se sia troppo lungo o meno, generalmente si riprende a seconda del valore (0, a sinistra e 1 se a destra); se non ci sono più nodie ho inserito tutto ciò che potevo, siamo per forza alla fine e per esserne sicuri, segnalato con -2, capisco che è un cammino troppo lungo e mi fermo qui.

Procediamo infine con la scrittura dell'ultima funzione, con segnatura ed implementazione seguente.

```
int inizio_cam(int*C, int i){
        int indice=0;
        bool exit=false;
        while(!exit && indice < i ){
                if(C[indice]==-2)          exit=true;
                else if(C[indice]!=-1)              indice++;
        }
        if(exit)  return -2;
        return indice+1;
}
```

Si ottimizza quindi il codice al massimo, sostanzialmente scorro il cammino di i elementi considerato l'attuale indice e a seconda di questo decidiamo se andare avanti o meno; l'invarianza è garantita finché ho degli indici diversi da -1 oppure da -2.

<u>Prelevare nodi da una lista L sulla base degli indici di un array A e pattern matching tra array A e lista L</u>

Sia L=2 -> 0 -> 3 -> -2 ->10 e A=[1,2,4], la lista da prelevare è quella formata dai nodi di indice 1 (il secondo nodo), 2 (il terzo nodo) e 4 (il quinto nodo) e cioè 0 ->3->10.
Resteranno quindi i nodi 2 -> -2.

Nel problema presentato, l'unica cosa a cui bisogna effettivamente fare attenzione è il fatto che, nel caso dovessi prelevare anche il primo nodo (indice 0) dovrò mantenere un puntatore allo stesso che poi punta ai successivi.
L'implementazione originale richiedeva una struttura coda e una funzione <u>iterativa</u>.
Come sempre, usiamo una struttura alternativa da noi usata rispetto a FIFO da noi non fatta, nodoE*.

```
nodoE* preleva(nodo*&L, int*A, int nA){
        nodoE* cut=new nodoE(L), *keep=new nodoE(L);
        while(L){
                nodo*aux=L;
                L=L->next;
                aux->next=0;
                if(*aux==A[i]){
                        cut->next=new nodoE(lista, aux);
                        A=A+1;
                        nA=nA-1;
                }
                else
                keep->next=new nodoE(keep, aux);

                i=i+1;
        }

        while(L && L->next){
                cut->next=new nodoE(lista, aux);
                L=L->next;
        }
        L=cut->info;
        return cut;
}
```

Per dare un'idea intuitiva del problema, si noti l'utilizzo delle variabili *keep* e *cut*, il cui utilizzo direi è abbastanza chiaro e chiarificatore. L'invariante del ciclo è *L && L->next esistenti e nA > 0 elementi da prelevare in vL*, sapendo che è possibile prelevare tali elementi sulla base di tali osservazioni. A livello induttivo, non cambia poi molto da altre volte.

Con L=0, non si fa nulla, poi distinguiamo due casi:

-ci sono abbastanza elementi in nA e tagliamo l'elemento di quell'indice specifico, altrimenti se non corrispondente teniamo l'elemento nella lista *keep*, che ci aiuta a tenere traccia degli elementi in ogni iterazione.
-non ci sono abbastanza elementi in nA; in questo specifico sottocaso, può succedere che L termini prima del tempo, pertanto si cicla fino a quando non terminano gli elementi, prelevando quanto rimasto e restituendo infine la parte tagliata.

Per la parte ricorsiva, si considera la scrittura di una funzione che calcola il pattern matching come indicato da titolo sopra tra lista ed array.
Diamone sotto prototipo ed implementazione, ricordando che come PRE e POST, similmente alla funzione precedente ammettiamo sempre l'esistenza degli elementi nella funzione (PRE) e poi avremo L la lista che consiste dei nodi matchati (caso funzione ricorsiva) o degli elementi tenuti dopo aver tagliato L-nA elementi (funzione precedente).
Senza ulteriori indugi, si segua il principio sottostante:

```
nodo* patt_match(nodo*&L, int*A, int nA){
        if(!L)    return 0;
        if(!nA){
                nodo* end=L;
                L=0;
                return end;
```

```
        }
        if(L==*A){
                return patt_match(L->next, A+1, nA-1);
        }
        else{
                nodo*x=L;
                L=L->next;
                x->next=patt_match(L, A, nA);
                return x;
        }
}
```

Inserimento di un nodo in un albero BST (info e campo num, che contiene il num. di sottonodi)

PRE=(albero(r) è un albero binario benformato e BST in cui il campo num di ciascun nodo è corretto, Vr=albero(r), x è un qualsiasi intero)
POST=(albero(r) è benformato, BST e con i campi num corretti ed è ottenuto da Vr aggiungendo il nodo con info=x come foglia)

Sfruttando la proprietà BST, si può fare abbastanza agilmente; si consideri appunto il campo num, che essendo funzione iterativa, sarà da diminuire ad ogni ciclo fino al bound di fine e di rottura invarianza. Attenzione che all'inserimento di un nuovo nodo, si dovrà tenere in considerazione cambiando e aumentando il campo *num*.

```
void build_BST(nodo*&r, int x){
        bool inserted=false;
        while(r && !(inserted) ){
                r->num=r->num+1;
                if(r->info > x){
                        if(r->left)
                        r=r->left;
                        else{
                        r->left=new nodo(x);
                        inserted=true;
                        }
                }
                else{
                        if(r->right)
                        r=r->right;
                        else{
                        r->right=new nodo(x);
                        inserted=true;
                        }
                }
        }
}
```

Commentiamo questa funzione (invarianza e correttezza). Sulla base di quanto operato sull'albero di riferimento, intuiamo che continueremo a ciclare fintanto che l'albero esiste e, nei BST, continuiamo ad andare avanti finché non si arriva ad una foglia, dove ci si ferma e si inserisce il nodo. Si inizia da sinistra, sapendo per invarianza la lista esistente e la variabile *inserted*, che intuitivamente segnala il mancato inserimento in posizione dell'intero di interesse, posta a false.

Eseguiamo induttivamente questa operazione da sinistra, quindi, poi a destra, seguendo il classico ragionamento degli alberi; come puntualizzato sopra, necessario aumentare prima di ciclare num dell'albero, sapendo che è operazione che viene eseguita induttivamente e iterativamente (ad ogni passo). Si continua a scendere di un livello ogni qual volta si ha un nodo esistente, come da precondizione, fino ad inserire l'elemento X voluto come da postcondizione.

Match completo in una H-fetta e avendo il match eliminare i valori matchati; calcolo della distanza tra gli elementi di una H-fetta

Torniamo nel magico mondo degli array a 3 dimensioni, visti come tali perché abbiamo un'array di interi classico ad una sola dimensione, vale a dire X.
Si tratta di cercare un match completo tra un array P pattern e l'array X tagliato a regola d'arte alla Filè; unica cosa, la distanza tra gli elementi indica la loro somma (banalmente, se ho [1,2,3] la distanza è 6, se ho [2,2,8] la distanza è 12).

Non essendoci una funzione vera e propria di riferimento, lo facciamo nel main() alla old school; altra cosa si considera come sempre il fatto che l'array non sia del tutto completo e solite fole.

```
int Fetta(int lim1, int lim2, int lim3, int i){
	return i%lim3 * i/lim3 * lim2*lim3;
	// nell'ordine, operazione su strati, su colonne e su righe, moltiplicando per lim3 che
	//come in altri casi è il limite che non usiamo
}
int main()
{
	int X[400], P[20], n_ele, nP, lim1, lim2,lim3;
	cin >> n_ele;
	for(int i=0; i<n_ele; i++)
	cin >> X[i];
	cin >> lim1 >> lim2 >> lim3;

	for(int i=0; i<n_ele; i++)
	cin >> X[i];
	cin >> lim1 >> lim2 >> lim3;

	if(lim1*lim2*lim3 < n_ele)
	n_ele=lim1*lim2*lim3;
	cin >> nP;
	for(int i=0; i<nP; i++)
	cin >> P[i];

	int n_strati_pieni=n_ele/(lim2*lim3);
	int elem_ultimo_strato=n_ele%(lim2*lim3);
	int righe_complete_ultimo_strato=elem_ultimo_strato/lim3;
	int elem_ultima_riga= elem_ultimo_strato%lim3;

	int hfetta;
	cin>>hfetta;

	int lung_fetta=lim3*n_strati_pieni;

	if(hfetta < elem_ultima_riga)
	lung+=lim3;
```

```
        else
        lung+=elem_ultima_riga;

        for(int j=0; j<lung – ip; j++){
        if(P[ip] == *X(hfetta * lim3 * Fetta(lim1,lim2, lim3, i)
        //devo cancellare l'elemento che fa match, che vuol dire
        //sostituire quello che ho con l'elemento i-esimo
        X[hfetta * lim3 * Fetta(i,lim1,lim2,lim3)]=*X[hfetta * lim3 * F(i+1, lim1, lim2, lim3);
        ip=ip+1;
        }
}
```

Alla luce di quanto visto, credo che quella sopra, ripresa da un turno di un parziale del 2019, sia abbastanza eloquente e dimostra il ragionamento sulle H-fette e V-fette.

Essendo una HF, io ragiono con lim3, perché è quello che uso per potermi muovere e moltiplico per lim2 nel caso in cui voglia le righe, altrimenti una volta tagliata la magica HF con la funzione di scorrimento, essendo che deve essere definito l'array, utilizzo una serie di variabili presenti anche negli esercizi che ho messo io su MEGA di quest'anno 2021, quindi:
-calcolo degli strati definiti, dato dal numero di elementi dividendo per righe e colonne, capendo che cosa ho in quel momento
-calcolo degli elementi sull'ultimo strato, che vuol dire fare una divisione intera o modulo da righe e colonne
-calcolo delle righe complete, dato dalla divisione degli elementi dell'ultimo strato (in quanto le righe sono una dimensione più piccola) rispetto a lim3, dimensione che non stiamo usando (se fosse una magica v-fetta, sarebbe diviso lim2 quindi).
-calcolo degli elementi rimanenti, quindi facendo una divisione intera/modulo rispetto a lim3.

In buona sostanza, ragiono in cascata, il lim3 definito, il lim2 definito e lim1 definito che si ragiona al contrario; sapendo che lim2 è formato da i*lim2*lim3 elementi, lim1 è formato da i%lim3 mentre lim3 da i/lim3, per capire che strato è, poi per capire a che riga sono.

L'invariante del ciclo è quindi dato da due condizioni; il fatto di avere la lunghezza della fetta abbastanza grande rispetto ad np, diminuito dell'indice ip e il fatto di avere, nel ciclo più interno, una serie di elementi considerati nella scansione scritta e spiegata che, se uguali tra loro, permettono l'avanzamento cancellando l'elemento di match attuale (sposto la magica v-fetta avanti di 1), altrimenti scorro semplicemente l'indice ip.

Scansione in larghezza (breadth first) di un albero binario scrivendo nell'attraversamento un array T con livello ed indice del nodo raggiunto

Sia R=[20]([10]([5]([0](_,_),[10](_,_)),[15]([10](_,_),_)),[30]([25]([0](_,_),_),[35]([0](_,_),_)))
e sia x= 10. In R ci sono 3 nodi con info=10: il primo è nel livello 1 in posizione 1, mentre gli altri 2 sono entrambi nel livello 3 in posizione 2 e 3.

Riprendendo questo esempio, quindi restituiremmo una variabile *nt* intera di conta dei nodi totali=3
e come T=[1,1,3,2,3,3], ricordiamo indicanti il livello dei nodi percorsi.

La funzione si presenta come una funzione iterativa, che scriverò per esercizio anche ricorsivamente.
Seguiranno invariante e correttezza della stessa.
PRE/POST come seguono:
PRE=(albero(R) è un albero binario ben formato, x è definito, T contiene un numero di posizioni pari al doppio del numero dei nodi di albero(R))

POST=(nt è il numero di nodi con info=x in albero(R) e T contiene il livello e la posizione di ciascuno degli nt nodi nell'ordine del percorso in larghezza)
Come diverse altre volte, problema fatto con la struttura coda/FIFO e da noi affrontata con nodoE*.

```
void perlar(nodo*R, int x, int*T, int& nt){
	nodoE* list=new new nodoE(R);
	int i=0, pos=0, local_level=0;
	while(R){
		if(R->left)
		list->next=new nodoE(R->left, R->liv+1);
		else
		list->next=new nodoE(R->right, R->liv+1);

		nodoE*e=new nodoE(R);		//estraiamo il primo nodo
		if(local_level < e->L){		//teniamo il livello locale e lo aggiorniamo
			local_level++;		//con liv e pos corrente
			pos=1;
		}
		if(e->info->info == x){		//se il nodo, come da richiesta è uguale ad x
			T[i]=liv;
			T[i+1]=pos;
			nt=nt+1;
			i=i+2;
		}

		pos+=1;				//aggiorniamo liv, pos ed nt, altrimenti aumentiamo
	}					//la variabile pos, che funge da contatore.
}

void perlar(nodo*R, int x, int*T, int& nt, int pos, int liv){
	if(!R)	return;
	int l=0;
	nodoE* lista=new nodoE(R);

	perlar(R-left, x, T, nt, pos+1, liv+1);
	perlar(R->right, x, T, nt, pos+1, liv+1);

	if(lista->info->info == x){
		*T=l;
		*(T+1)=pos;
		nt=nt+1;
		T+=2;
	}
}
```

Ecco servite le due versioni, iterativa e ricorsiva del problema. Nella ricorsiva si rendeva necessario utilizzare le due variabili *pos* e *liv* come variabili utili nello spostamento ricorsivo e salvare di conseguenza, ad ogni iterazione/ricorsione, lo spostamento (sinistro o destro), tenendo conto del livello in cui mi sto spostando. Come da commenti, ecco che approccio l'idea in una maniera intelligente; prendo l'array di riferimento, salvo il primo elemento che ho estratto e controllo se questo possa essere uguale ad x; nel caso lo sia, posso aumentare i nodi totali trovare e valorizzare l'array T di conseguenza, altrimenti aumento (ric/iter) la posizione per sapere dove sono.

La correttezza è data da questo fatto; lo spostamento garantito fintanto che esiste l'albero. Qualora finissero i nodi a sinistra, si va a destra e una volta finiti tutti i nodi e al ritorno della ricorsione, si arriva avendo valorizzato T con i due campi dati.

L'invarianza prende in considerazione l'esistenza della lista, composta da un numero di livelli=left+right+1, sulla base delle osservazioni fatte. In termini più da Filè:
R=L ben formata && navigo a left/right nei livelli 0…liv-1<left+right+1

Fusione di array ordinati in uno unico (stessa cosa con una lista); ricerca del nodo/elemento minimo

Si ha un array A[limite][20] nel quale ogni riga è ordinata in modo crescente e si vuole riempire un altro array R di interi che contiene ogni elemento di A in maniera crescente.
La funzione presente che esegue questo compito è:

```
int* F(int(*A)[20], int limite){
        int*R=new int[limite*20];
        int *inizio=new int[limite], r=0, k=0;
        for(int i=0; i<limite; i++){
                inizio[i]=0;
                for(int i=0; i<limite * 20; i++){
                        M(A, inizio, limite, r, k);
                        R[i]=A[r][k];
                }
        }
        delete[] inizio;
        return R;}
```

R sostanzialmente viene usato per capire dove siamo arrivati a prelevare le righe di A.
Come si vede viene chiamata una funzione M che sarà iterativa e restituisce ad ogni invocazione gli indici r,k passati per riferimento e individua la posizione dell'elem. minimo di A tra quelli non copiati in R.

```
void M(int (*A)[20], int *inizio, int limite, int &r, int &k){
        int min_r=0, min_k=0;
        for(int i=0, r=0, k=0; i <limite*20; i++, r++, k++){
                if(inizio[i]<A[r][k])
                min_r=r;
                min_k;
        }
        if(min_r && min_k){
                r=min_r;
                k=min_k;
        }
}
```

Da quello che posso intuire dall'aramaico del prof, si cerca un minimo tra i due array negli elementi non copiati, avendo quindi questa come postcondizione, aggiungendovi anche che restituisce nulla se non trova alcun indice minimo. Di fatto, inizio è un array di scorrimento, che è sempre inizializzato a 0, mentre A è un array che permette di capire dove sta il minimo elemento. Sapendo che A è ordinato, utilizzo inizio e cerco di capire chi è il minimo salvandolo, prendendo come confronto l'array inizio che non è ordinato rispetto ad A che invece lo è.
L'invarianza è data dal fatto di non oltrepassare i limiti dell'array (limite*i), altrimenti semplicemente r e k vengono valorizzati al primo valore utile, quindi un calcolo del minimo rispetto ad inizio, array appunto minore di A che è infatti inizializzato a 0.

Per la parte ricorsiva, si ha una lista che è ordinata crescente rispetto ai suoi campi info e si vuole costruire una funzione ordinata R; come prima, bisogna trovare la lista con il primo nodo minimo di A che punta a questa lista.

La funzione sarà la seguente:

```
nodoP* G(nodoP* A){
        nodoP* G min=A;
        if(!A)    return 0;
        nodoP*ret=G(A->next);

        if(ret < min)
        return ret;
        else
        return min;
}
```

Nell'appello in questione è una funzione che scorre cercando il primo nodo minimo e restituisce quello che ha. Se la lista di partenza è 0, altrimenti dato che deve cercare il nodo minimo, se non ancora vuota, è comunque saggio confrontare ret, nodo ritornato, con min che punta al primo elemento e quindi ritornare questo.

La postcondizione sarà infatti cercare il nodo minimo tra tutti quelli della lista in questione e inuttivamente la correttezza si dimostra sapendo che se vuota, la funzione ritorna 0, altrimenti si scorre al passo next e next->next… fino ad A->next, confrontando se i prefissi di A sono potenziali nodi minimi; quindi la ricorsione termina restituendo il possibile nodo minimo della lista.

Ricerca di un match formando due liste: i nodi estratti senza match e i nodi rimasti con match

Poco da commentare rispetto ad altre situazioni, scriviamo quindi funzione, correttezza, PRE/POST

```
nodo* F(nodo*& L, char *P, int dim_P){
        if(!L || !dimP)   return 0;
        if(L->info == *P){
                nodo*ex=L;
                ex->next=F(L->next, P+1, dimP-1);
                ex->next=0;
                return ex;
        }
        else      return F(L->next, P, dimP);
}
```

Specifica che nuovi nodi non vengono creati, ma manipolati i nodi originali, tuttavia è necessario come rispecifica anche il testo stesso, creare una lista restituita col return, cosa che è stata fatta.
La PRE è il fatto di avere una lista ben formata, un array non vuoto e dim_P >0 mentre la POST indica che la funzione restituisce col return la lista dei nodi che non fanno match tenendo in L sono quello che fa match.

Eliminare un nodo con info=x da una lista di liste

Se per esempio avessimo una lista di liste LL con 3 nodi, nel qual caso Na, Nb, Nc con le liste:
Na=4->2->1 Nb=2->2 Nc=3
Volessimo eliminare i nodi con info=2 resteranno

Na=4->1 Nc=3
L'eliminazione di un nodo presuppone la sua eliminazione dallo heap, come sappiamo già anche da P2.
La stessa idea viene affrontata in entrambe le parti.
La lista di liste usa una struttura NN, che ha due campi, un nodo* lista e un puntatore a NN next

Vediamo prima la parte iterativa:

```
nodo* G(nodo *L, int x){
        while(L){
                if(L->info==x){
                        nodo*y=L;
                        L=L->next;
                        delete Y;
                }
                L=L->next;
        }
        return L;
}
```

Poi la parte ricorsiva, che considera che LL è ottenuto togliendo i nodi che contengono x e i nodi che puntano a liste vuote. Essa userà una seconda funzione ricorsiva, chiamata Gric, per eliminare ricorsivamente da L i nodi che contengono x.

```
void Fric(NN*& LL, int x){
        if(!LL)   return;
        nodo*aux=LL->lista;
        Gric(L, x);
        if(aux)   Fric(LL->next, x);
        else{
                NN*y=LL;
                LL=LL->next;
                delete y;
                Fric(LL, x);
        }
}

void Gric(nodo*&L, int x){
        if(!L)    return;
        if(L->info !=x)   Gric(L->next, x);
        else{
        nodo*y=L;
        L=L->next;
        delete y;
        Gric(L, x);
        }
}
```

Gestione di una serie di input in un array in due versioni

Dato un intero k>0, si tratta di leggere da un input continuando fino a che non si legge -2, considerando che:
-prima di leggere -2 sono stati letti al più k numeri, in quel caso si devono stampare tutti i valori letti
-prima di leggere -2 sono stati letti più di k interi, si stampano gli ultimi k valori letti prima della sentinella nell'ordine di lettura. Per fare questo, il programma deve usare un array M di k interi; nel primo caso si

inseriscono in tutte le posizioni, nel secondo caso invece, non sapendo quanti saranno i valori letti per ultimi, ma in generale non li conterrà a partire dalla prima posizione in poi. (quindi il k+1 esimo sostituirà il primo letto, il k+2 esimo sostituirà il secondo letto e seguendo così).
Si sa inoltre che si potranno inserire solo valori nuovi non presenti e non si potranno spostare valori già presenti.

Esempi pratici:
Supponiamo che l'input dia [3,1,2,4,0,-2].
Nel primo caso, stampiamo semplicemente 3,1,2,4,0 (perché poi ci sta -2)
Nel secondo caso stampa 4 e 0, perché di volta in volta viene fatta la cosa seguente:
[3, _] (il secondo elemento ancora non c'è)
[3,1] (ora sì; il prossimo andrà al posto di 3, quindi dell'elemento k)
[2,1] (quindi il successivo andrà al posto di 1, quindi dell'elemento k+1)
[2,4] (stessa cosa tolgo 2 e metto l'altro elemento)
[0,4] (che dà luogo all'ultima stampa).
Quindi questo array toglie un elemento dall'ultima posizione non considerata, in maniera circolare come segnala il testo stesso del prof.

Per la parte iterativa, si specifica la creazione di una funzione che può usarne altre iterative, di cui descriveremo invariante, pre e postcondizioni. Il prof consiglia la creazione di due funzioni ausiliarie, una per stampare i valori e l'altra per calcolare l'indice di M in cui inserire il prossimo intero letto.

```
//PRE=k num. di interi che sarà il limite dell'array dinamico M
void G(int k){
	if(k==0)		return;
	*M=new[k];	bool go_on=true;
	std::cin >> "Inserisci l'intero x: ">>x;
	bool go_on=true;
	while(go_on){
		int indice=find_index(M, k, x);
		if(indice == -2)		go_on=false;
		indice++;
		std::cin >> "Inserisci l'intero x: ">>x;
	}
	if(!indice)	stampa(0, M, indice);
	stampa(indice, M, k);
}
//POST=Restituisce un indice che sarà l'elemento i se i<k, altrimenti x se x==k. Esso sarà stampato se >0, altrimenti si ignora la stampa e si restituisce tale e quale

int find_index(int *M, int k, int x){
	int i=0;
	if(i < k)
		i++;
	else{
		i=0;
		M[i]=x;
	}
	return i;
}

void stampa(int indice, int *M, int k){
	for(int i=indice; i<k; i++){
```

```
                std::cout<<M[i]<<" "<<endl;
        }
}
```

Si noti come l'invarianza è data dal fatto di avere un elemento x!=-2 e che si estende dinamicamente in base agli elementi presenti; le due funzioni ausiliarie ragionano sul limite dei k elementi, venendo riempite iterativamente da 0 fino a k-1. Una banale per la stampa e l'altra per la gestione circolare della valorizzazione degli elementi di input.

Per la parte ricorsiva, si usa una funzione con parametro M che è l'array da gestire in maniera circolare, index posizione di M in cui inserire il prossimo intero letto, descrivendo PRE e POST (nel qual caso, tutto identico a prima, solo la cosa è fatta ricorsivamente).

La funzione sarà:

```
void Gric(int k, int *M, int index, int count){
        int x=0;
        cin>>x;

        if(x == -2){
                if(count>k)     stampa(index, M, k);
                else            stampa(0, M, index);
        }
        else{
                if(index == 0) return;
                M[index]==x;
                Gric(k, M, index+1, count+1);
        }
}
```

Pattern matching di un pattern P[n] in un testo T[n][n] (visto ad una sola dimensione)

Una novità grossa, un altro pattern matching. Si cerca P[0] nella colonna 0 di T. Se viene trovato un match, si cerca P[1] nella colonna successiva e dall'indice 0 fino ad n-1 della riga in questione e così via. Se fallisce la ricerca, fallisce anche il match. Se invece tutti gli elementi vengono trovati, il match così descritto viene salvato in un array R[n] che salva gli indici delle righe di riferimento. La variabile "n", non specificata dal testo del prof come tutte le cose utili, serve semplicemente a determinare gli elementi del pattern (sarebbe la solita dimP).
Esempio pratico: n=3, P[8,8,10], T[7,9,2][8,8,4][10,8,10] allora l'array R è uguale a [1,1,2].
Questo perché ci sarà match nell'indice 1 della seconda riga, nell'indice 1 e 2 della terza riga (se si nota come è fatto T, mi muovo per colonne ed è la solita moltiplicazione inutile che fa il prof per complicare la vita e trovare i magici match meravigliosi). Vediamolo scritto, magari.

Parte iterativa (solite robe, se trovo il match lo ritorno nell'array dato, altrimenti ritorno 0. Consigliata una magica funzione ausiliaria). Specifica di PRE/POST di funz. principale ed eventuale funz. ausiliaria. Scegliere ciclo significativo e scrivere invariante dello stesso.

```
//PRE=array di interi T e P ben definiti, n elementi del pattern P >=0
int*F(int*T, int n, int*P){
        int i=0, r_match=0;
        *R=new int[n];

        while(i<n){
```

```
                if(match(T, p, n, r_match)
                R[i]=r_match;
                else
                R[i]=0;
                i++;
        }

        return R;
}
//POST=restituisce l'array che trova gli indici in T e P del match, altrimenti ritorna 0

bool match(int *T, int*P, int n, int){
        bool stop=false;
        if(i==n) stop==true;
        if(T[i+j*n]==P[i])
        i++;
        else
        stop=false;

        return stop;
}
//POST=restituisco stop==true sse match completo, altrimenti stop==false se no match
```

Parte ricorsiva

```
//PRE=stesse di prima, più indici di riga e colonna>0 e variabile di controllo match
int* Fric(int *T, int n, int *P, int riga, int col, bool is_matched){
        *R=new int[n];

        if(n==*P){
                if(is_matched)
                return R;
                else
                return 0;
                }

        is_matched=matchR(T+col, n, P, riga, col, ind_match);
        if(is_matched){
                R[col]=ind_match;
                Fric(T, n, P+1, riga, col+1, is_matched);
        }
        else{
                R=0;
        }
        return R;
}

bool matchR(int*T, int*P, int riga, int col, int ind_match){
        if(*P == n)        return false;
        if(T[riga*col*n]==*P){
        ind_match=riga*col;
        return true;}
        return matchR(T, P, n, riga+1, col, ind_match);
```

```
}
//POST=restituisce un array R con elementi riempiti da 0 a n-1 se ci sta un match, 0 altrimenti
```

Come sempre in questi casi, match e ricerca dello stesso riga+colonna*n per determinare posizione attuale di ricerca. Per il resto considero un booleano che ferma la ricerca alla bisogna altrimenti avanza, fintanto che siamo nei bound degli array ed esiste match.

Determinare la colonna con il minimo numero di valori distinti in un array di interi A

Dato un array A di 100 elementi, determiniamo la colonna con il minimo numero di elementi distinti. Di questi ne sono definiti solo *dim*.
Ad esempio, avendo A=[0 1 1 0 1 2 0 1 2 3 0 1 0 1 0 1 0 1 0 1 0 0 1]
(con uno spazio extra ogni 10 elementi per distinguere la separazione tra le singole righe).
Dim è uguale a 23 in questo caso e la terza riga è definita da soli 3 elementi.
La colonna 0 (formata da 0 1 1 0 1 2 0 1 2 3, ha tre 0 ripetuti), la colonna successiva ne ha 2 distinti. La migliore è la prima, un solo valore ripetuto più volte.

PRE_F=(A è array10x10 con i primi dim elementi definiti,dim>0)
Int F(int A[][10], int dim, int &kol)
POST_F=F restituisce un intero vcol return e un altro intero kol,passato per riferimento, questi due valori restituiti sono tali che: (la colonna kol di A ha un numero pari a v di valori distinti e nessuna altra colonna di A ha un n. inferiore a v di valori distinti).

La funzione F deve invocare una funzione ausiliaria CC con il seguente prototipo e PRE e POST-condizioni
PRE_CC=(X è array 10x10 coni primi dim elementi definiti, dim>0)&&(c è una colonna di X che sicuramente contiene qualche elemento definito)
int CC(int X[][10],int dim, int c)
POST_CC=(CC restituisce il numero di valori distinti contenuti nella colonna c di X (considerando solo gli elementi definiti di c))

Le PRE e POST ci sono per completezza, non perché aiutino a capirci qualcosa, visto il filèse che viene qui utilizzato.
Lanciamoci quindi nella scrittura delle due magiche funzioni:

```
Int F(int A[][10], int dim, int &kol){
        int v=0, loc=0;
        bool empty_col=0;
        for(int i=0; i<10 &&!empty_col; i++){
                if(i < dim){
                loc=CC(X, dim, i);
                if(loc < v){
                        v=loc;
                        kol=i;
                }
                else
                empty_col=true;
        }
        return v;
}

int CC(int X[][10],int dim, int c){
        int el_dist=0, all_elements=0, row=0;
        while(all_elements != dim){
```

```
        for(int i=0; i<dim%10; i++){
                if(X[i][c] == X[r][c])
                el_dist++;
        }
        all_elements+=10;
        row++;
        }
}
```

Un esercizio leggermente diverso da altri, ma secondo me buono per una semplice trattazione di elementi, valori e compagnia cantante. Si noti come la scansione procede sempre se l'elemento c'è o non c'è ed anche in questo caso mi muovo in due dimensioni, tenendo conto dei movimenti con i singoli booleani. Quindi spiegando le due funzioni, uso CC che sarà minore al numero di righe (dim&^%10, quindi 3) e controllo se i singoli elementi sono uguali, contandoli e tenendo nota di quali sono uguali e quanti tra righe e colonne; ogni 10 elementi, conto gli x elementi raggiunti ed aumento la riga. Essendo che non tutti sono definiti, tale movimento deve essere considerato nella funzione principale, in quanto nella ausiliaria e come mostra il prof, bisogna scansionare ogni 10 elementi e i singoli pezzi vanno scansionati nella funz. precedente.
L'invarianza è data dall'assunzione dell'esistenza dei singoli blocchi, poi la conclusione come definita qui.

Inserire 1 al posto di 0 in una linked list e usare questa funzione per costruire un albero binario completo

Esempio: Sia L=0->1->0->0->1->1 allora, l'ultimo 0 è quello nel nodo di indice 3 (attenzione che gli indici partono da 0), quindi la lista va trasformata nella seguente: 0->1->0->1->0->0. Applicando di nuovo la funzione sulla lista appena ottenuta, avremmo: 0->1->0->1->0->1 e applicando la funzione per la terza volta: 0->1->0->1->1->0.
Chiaro che ci fermiamo non appena abbiamo tutti 1 e che la variabile b mi segnala di volta in volta posizione e operazioni da fare; se b=true abbiamo qualcosa da fare (andare avanti azzerando i successivi elementi posto l'elem. attuale ad 1), altrimenti è false. Se false, possiamo avere la lista vuota e fermarci, altrimenti se oltre ad essere false il campo info è 0, in quel caso invochiamo ricorsivamente per azzerare la lista in questione. Queste semplici osservazioni dimostrano la correttezza, sapendo che si fa solo una modifica sull'elemento i-esimo ritoccando poi tutti i successivi.

La funzione ricorsiva che realizza questa operazione è la seguente:

```
void F0(nodo*L, bool & b){
        if(!L){b=false;    L=0;}
        if(!b && !L->info){
                L->info=1;
                azzera(L->next);
                b=true;
        }
        F0(L->next, B);
}

void azzera(nodo*L){
        if(!L) return 0;
        L->info=0;
        azzera(L->next);
}
```

La funzione iterativa, riceve in input un albero ed un intero *alt* che una volta usato F0 per generarsi i cammini (quindi 1 rappresenta il nostro campo info, azzerando come commentato sopra ogni campo

successivo), aggiunge tutti i cammini non già presenti in R; detto in parole umane e non filèsi, scorro tutto l'albero e capisco se metterci un valore al campo info attuale altrimenti continuare ad iterare. Semplicissimamente, dove c'è uno 0 si va a sinistra, dove c'è un 1 a destra ma tenendo conto con il booleano dove sono attualmente, spostandomi di conseguenza e valorizzando il nodoT* di riferimento.

PRE=(albero(R)corretto, 0<prof, INP ifstream definito,R=vR, INP contiene (almeno) 2(alt+1)-1 valori)
nodoT * F1(nodoT* R, int alt, ifstream & INP)
POST=(albero(R) èottenuto da albero(vR) aggiungendo tutti i nodi non già presenti in albero(vR) di tutti i cammini di lunghezza alt, i campi info dei nodi sono letti da INP. I cammini vanno considerati partendo dal cammino con soli 0, poi applicando F0 ripetutamente)

```
nodoT * F1(nodoT* R,  int alt, ifstream & INP){
        if(!R){
        int x; INP >>"Inserisci radice: "; R=new nodo(x);
        }
        nodoT* aux=T;
        nodo*L=0;bool b=true;
        while(b && L){
                if(L->info == 0){
                        if(!T->left)
                        aux->left=new nodoT(aux->left, INP);
                        else
                        aux=aux->left;
                }
                else{
                        if(!T->right)
                        aux->right=new nodoT(aux->right, INP);
                        else
                        aux=aux->right;
                }
                L=L->next;
        }
        L0(L, b);

        return 0;
}
```

Al fine della correttezza, si noti come si scorre semplicemente a sinistra, a destra data la numerazione attuale e grazie all'altra funzione creiamo un cammino completo ed un albero, qualora siano presenti foglie e/o nodi validi nella struttura ausiliaria considerata.
Quindi:
-se nodo vuoto, inserisco la radice
-se nodo non vuoto, creo una lista per capire dove muovermi e una copia ausiliaria dell'albero, spostandomi e creando nodi di volta in volta che prendono i singoli pezzi a destra/sinistra valorizzando il nodoT* info della lista di nodi di riferimento.

L'invariante è la condizione di avere b=true sse non ho ancora inserito tutti i nodi, considerando di avere scorso tutta la lista L e inseriti tutti quanti formo quindi cammini completi.

<u>Costruzione di un albero per strati con una struttura chiamata dni</u>

Immaginiamo una dni come se fosse una coda (non approfondiremo e non usiamo nodoT* in questo specifico caso, ma usiamo pop_back e push_end); il problema è di utile trattazione per capire il movimento

dentro i magici alberi. Dobbiamo costruire alberi per strati, quindi avendo un puntatore al nodo attuale si aggiunge un numero di nodi pari a quelli puntati: sono all'inizio, ne punto 1 (radice) e aggiungo un figlio a sinistra e poi uno a destra. Punto a due nodi (figlio sinistro e figlio destro) e poi aggiungo altri due figli per ciascuno (quindi quattro). Evitando esempi filèsi che come sempre non aiutano, questa è il sunto di una pagina e mezza in un linguaggio semplice come piace a noi.

La funzione iterativa di riferimento, data una lista, aggiunge figli ai nodi dell'albero restituendo una nuova DNI che punta all'aggiunta di tutti i nodi:

```
dni creastrato(int& n, dni X0,ifstream & IN){
        dni* list=0;
        if(!X0)   {int x; IN >> X; return new dni(new nodo*(x));}
        while(X0 && n>0){
                n=n-1;
                int x; IN >> X; return new dni(x);
                nodo*aux=new nodo(x);
                dni* d=new dni(aux);
                list=push_back(list, d);
                if(X0.primo->info->left)
                X0.primo->info->left=aux;
                else
                X0.primo->info->right=aux;

                d=pop(X0);
        }
        return list;
}
```

In parole povere, sopra abbiamo una sorta di FIFO che continua ad andare avanti finché abbiamo n elementi e la FIFO, sotto nome di DNI di partenza (invariante), continuando a muoversi e valorizzare finché ci sono degli elementi a sinistra e a destra, creandosi una copia del nodo poi messa nella lista che sarà a tutti gli effetti restituita alla fine, oppure creando un nodo vuoto sulla base di quello che viene messo in input. Il risultato sarà comunque una lista completamente riempita di valori effettivi.

La funzione ricorsiva invece considera una DNI che gestisce tutti i nodi dell'albero; tradotto in italiano significa semplicemente scorrere tutti i nodi validi a sinistra e a destra restituendo una lista che punta a tutte le foglie dell'albero scorso, dopo una visita in ordine infix (infisso), quindi:

```
dni unfold(nodo* r){
        dni* lista=0;
        if(!r)     return new dni();
        if(r->left)         lista=push_back(unfold(r->left), new nodo*(r));
        else                lista=push_back(lista, unfold(r->right));
}
```

Array a tre dimensioni e una h-fetta che contiene k2 volte il valore k1

Qui il prof definisce le fette come sequenze di righe sovrapposte degli strati, almeno esiste una loro definizione. Array di riferimento è X[2][3][4] con dim=17 e significa che c'è un solo strato completamente definito, il secondo solo la prima riga tutta piena, la seconda con un solo valore mentre la terza vuota. Ricalca l'ultimo appello tra tutti, presente 60 pagine fa sostanzialmente la sua formazione.

Bando alle ciance, si chiede una funzione che date le osservazioni descritte si deve scrivere una funzione che soddisfi quanto prima enunciato; si consiglia almeno una funzione ausiliaria che conti le occorrenze di k1 nell'array di riferimento. Questa è la parte iterativa.

```
int F(int(*X)[5][10], int lim1, int dim, int k1, int k2){
	int str_pieni=dim/(5*10);
	int righe_ultimo_strato=(dim%(5*10))/10;
	int elem_ultima_riga=(dim%(5*10))%10;
	bool found=false;
	int nf=dim/10;		//quindi diviso lim3
	for(int f=0; f<nf && !found; f++){
		for(int s=0; s<str_pieni; s++){
			int count=occurr(X[s][f], 10, k1);
		}
	}
	//caso ultimo strato
	if(f < righe_ultimo_strato)	count+=occurr(X[ns[[f], 10, k1);
	else				count+=occurr(X[ns], elem_ultima_riga, k1);

	if(conta == k2){
		found=true;
	}
	return f;
}

int occurr(int *X, int dim, int k1){
	int count=0;
	for(int i=0; izdim; i++){
		if(X[i] == k1)	count++;
	}
	return count;
}
```

Per la parte ricorsiva, si usano le liste concatenate e si chiede di scrivere una funzione che tagli una porzione di lista, considerando però due limiti, k1 e k2. Se k1 è maggiore della posizione dell'ultimo nodo, oppure k2<k1, il taglio è 0.
Segue implementazione e dimostrazione induttiva:

```
nodo* cut(nodo*&L, int k1, int k2){
	nodo*list=0;
	if(!L || k1 < k2) return 0;
	if(k2)
	return cut(L->next, k1, k2-1);
	if(k1){
		nodo*x=L;
		L=L->next;
		x->next=cut(L, k1-1, k2-1);
		return x;
	}
}
```

Taglio di una lista considerate quindi le due variabili di riferimento k1 e k2; la seconda dice solo avanza ricosrivamente nella lista, la prima dice crea nodo ausilario e taglia. That's it.

Niente nodi e k1 < k2; fine.
Se invece ci sta k2, semplicemente non taglio e avanzo induttivamente, altrimenti ci sta k1, creo nodo ausiliario, ricorro su questo e lo ritorno.

Stesso problema di sopra con strati non definiti su colonne e pattern matching su albero binario

Appelli vecchi, ma che secondo me chiariscono concetti mai spiegati come si deve e comunque utile prima di scoppiare nel macello h-fette/v-fette.

Parte iterativa

```
int F(int(*X)[5][10], int lim1, int dim, int k1, int k2){
        int str_pieni=dim/(5*10), count=0;
        int righe_ultimo_strato=(dim%(5*10))/10;
        int elem_ultima_riga=(dim&(5*10))%10;
        bool found=false;
        int nf=dim/5;
        for(int f=0; f<nf; f++){
                for(int s=0; s<str_pieni; s++){
                        count+=occurr(*(X[k])+f, 5, k1);
                }
                if(righe_ultimo_strato !=0 && f<righe_ultimo_strato)
                count+= occurr(*(X[k])+f, righe_ultimo_strato+1, k1);
                else count+= occurr(*(X[k])+f, righe_ultimo_strato, k1);

                if(count == k2)
                found=true;
        }
        return f;
}
```

Per la parte ricorsiva, si segnala con -1 la fine del cammino di match, sapendo che 1 significa andare a sinistra mentre 0 andare a destra. Ordine di attraversamento è il prefisso. Come sempre viene richiesto il primo match, che significa fermarsi subito con la ricorsione, perché a semplificare per qualcuno è difficile. Il C[0]=2 rappresenta false e la presenza di nessun cammino, altrimenti si ricorre come sempre.

Ad ogni modo, la funzione è qui:

```
bool M(nodo*R, int prof, int*P, int dimP, int*C){
        if(!R)    {*C=2;   return false;}
        if(!dimP){
                if(prof ==0)
                return 0;
        else{
                *C=-1;
                return true;}
        }
        if(*P == R->info){
                P=P+1;
                dimP=dimP-1;
        }
        if(R->left && M(R->left, prof+1, P, dimP, C+1))
                return true;
        else
```

```
            *C=-1;
        if(R->right && M(R->right, prof+1, P, dimP, C+1))
            return true;
        else
            *C=-1;

}
```

Per completezza, incollo una correttezza fatta decentemente (io nella maggior parte dei casi ho sempre dato intuizioni e considerazioni che ovviamente si cerca di allungare in un esame):

Dato PRE_M, se dimP==0 allora abbiamo trovato un match ...

caso base: !R && dimP!=0
---
*Dato PRE_M, se R è un albero vuoto e non ho trovato un match completo, allora segnalo che in quel cammino non c'è match con *C=C[0]=2 e ritorno 0 → vale POST_M*

caso ricorsivo: M(R->left, P, dimP, prof+1, C+1)
---
*Dato PRE_M, devo considerere il figlio sinistro, quindi *C=0, poi assumo per ipotesi induttiva la chiamata ricorsiva M(R->left, P, dimP, prof+1, C+1):*
- se ritorna true allora c'è almeno un match completo di P[1..dimP-1] nel sottoalbero sinistro e C=[0,..,-1] → valgono PRE_M e POST_M
- se ritorna false allora non c'è match completo di P[1..dimP-1] nel sottoalbero sinistro e C=[0] diventa C[2] → valgono PRE_M e POST_M

caso ricorsivo: M(R->right, P, dimP, prof+1, C+1) && !M(R->left, P, dimP, prof+1, C+1)
---
*Dato PRE_M, devo considerare il figlio destro, quindi *C=1, poi assumo per ipotesi induttiva la chiamata ricorsiva M(R->right, P, dimP, prof+1, C+1):*
- se ritorna true allora c'è almeno un match completo (non ci sono match nel sottoalbero sinistro) di P[0..dimP-1] nel sottoalbero destro e C=[1,..,-1] → valgono PRE_M e POST_M
- se ritorna false allora non c'è match completo di P[1..dimP-1] nel sottoalbero R (nè a destra nè a sinistra) e C[1] diventa C[2]

*/

<u>Creazione di una lista e ricerca di un pattern matching rispetto ad un array P</u>

Letti dim interi costruisco la lista attraverso il parametro L e poi nella funzione sotto ricerco dei match completi. Entrambe saranno <u>ricorsive</u>.

```
void crea(nodo*& L, int dim, ifstream & INP){
        if(!L)    {int x;   INP>>x;  return;}
        int elem; INP>>elem;
        L=new nodo(elemento, 0);
        crea(L->next, dim-1; INP);
}
```

La seconda funzione presenta queste PRE e POST, che saranno dimostrate sotto:

PRE_match=(L è una lista corretta, L=vL, P ha dimP elementi definiti con dimP>0)

POST_match(se match restituisce col return un valore R diverso da 0, allora esiste un match completo di P in L e R è R(vL,P) e il valore di L è vL-P) && (se match restituisce 0 allora non c'è alcun match di P in L e il valore di L è vL).

Altra cosa del prof: *Attenzione: la condizione che dimP>0 nella PRE_match indica come segnalare al ritorno dalla ricorsione se il match è stato completato o no. Inoltre match non deve né creare né distruggere nodi.*

```
nodo* match(nodo* &L, int*P, int dimP){
        if(!L || !dimP)   return 0;
        if(L->info == *P){
                nodo*complete=L;
                L=L->next;
                complete->next=match(L.>next, P+1, dimP-1),
                return complete;
        }
        else return match(L->next, P, dimP);
}
```

Come sempre, le intuizioni base sono, no L o dimP, ritorno 0,
altrimenti induttivamente è uguale l'elemento L-esimo e quindi quello +1, fintanto che staccato il primo nodo la ricorsione prosegue, essendo un match completo, altrimenti scorro solo la lista e buonanotte sognatori.

Array ad una dimensione visto come se ne avesse due e: trovare tripla di lunghezza massima & trovare match contigui e non completi

Simile a problemi proposti negli ultimi appelli del 2021 (due appelli del 2013 questi due), quindi un array a 2 dimensioni visto come una e prendiamo le triple, che dicono dove inizia il match nella colonna, a partire da quale posizione e con quale lunghezza. A noi interessa il match di lunghezza massima assoluta tra tutte le colonne.
Sotto l'esempio, tralasciando inutili complicazioni filèsi, devo matchare tra di loro le righe con le colonne. Per esempio, prendendo colonna 2 (la terza), ho un match della prima riga fino alla posizione 5 e questo è segnalato dalla tripla "2 0 5"; non allungo il brodo per non complicare, come qualcuno che dice di semplificare con questi obbrobri di funzioni da fare.

Esempio: Sia R=8, C= 6 e T il seguente array. Si ricordi che T è ad una dimensione, ma lo rappresentiamo con R righe e C colonne per facilitare la comprensione dell'esempio essendo questo il modo di "vedere" T nell'esercizio:

T=

0 0 0 1 0 1	le triple per ciascuna riga sono queste:	2 0 5
1 1 0 1 0 0		3 0 5
0 1 0 0 1 0		0 0 6
0 0 1 1 0 1		0 2 3
1 1 0 0 1 0		4 4 4
0 1 0 1 1 1		3 2 6
0 0 0 1 0 1		2 0 5
0 1 0 1 0 1		3 2 4

Per la parte iterativa, si riporta l'utilizzo di questa funzione:

```
void It0(int*C, int R, int C, TRIPLE* Q){
        triple matched=0;        bool found=true;
        for(int i=0; i<R; i++){
                for(int j=0; j<C && !found; c++){
                        TRIPLE matched_aux=IT1(T, c, C, r, R);
                        if(matched_aux.lung > matched.lung)
```

```
                        matched=matched_aux;
                        if(matched.lung == R)    found=true;
                }
        }
        *Q=i;
        return;
}
```

Bisogna invocare la funzione ausiliaria che trova la tripla di massimo match, quindi:

```
TRIPLE It1(int*T, int r, int c, int R, int C){
        TRIPLE matched=0;
        for(int i=0; i<R && R-i>matched.lung; i++){
                triple matched_aux=findmatch(T, c, C, r, R, i);
                        if(match_aux.lung>match.lung){
                                match=match_aux;
                                if(match.lung==R)
                                completo=true;  }
        }
        return match;
}

int findmatch(int *T, int c, int C, int r, int R, int i){
        int lung=0;
        while(i<C && *(T+(i+lung)*C*c)==*(T*r*C+j){
                lung++;
                j++;
        }
}
```

Abbastanza impestato e decisamente intuibile da tutti questa megadereferenziazione giusto? Ricordo un appello dove sono passati 3 studenti su questo esercizio, ma fattibilissimo come tutti gli altri, niente da dire. La spiegazione completa di questo esercizio è tra le prime e preferisco non avvelenarmi il sangue ulteriormente.

Ad ogni modo, usiamo invece l'altra funzione che trova match contigui e non completi (scorro come primo ciclo sulle colonne perché mi interessa il più lungo su queste ultime).

```
void F(int *T, int R, int C, int* P, int dimP, int*Q) {
        int inizio=0;
        for(int i=0; i<C; i++)
                int lung=0, aux=0;
                for(int j=0; j<R; j++){
                        aux=match(T+i*j*c, i, j, C, P+inizio, inizio);
                        if(aux > lung)
                        lung = aux;
                }
        *Q=lung;
        inizio=(inizio+lung)%C;
}

int match(int*T, int r, int c, int R, int C, int*P, int dimP, int inizio){
bool found=false;
        for(int i=inizio; i<dimP && r<R && !stop; i++){
                if(*T == *P){
```

```
                    T=T+C;
                    P=P+1;
                    dimP=dimP-1;
                }
                found=true;
        }
        r++;
}
```

Per la parte ricorsiva, si riporta una funzione che deve utilizzare una funzione ricorsiva R1, che fa praticamente la stessa cosa sopra, ma in maniera meno impestata e ricorsivamente.

```
void FR(int*T, int R, int C, int*P, int dimP, int inizio, int*Q)
{
        if(!R) return;
        int MAX=R1(T,C, P+inizio, dimP-inizio); // da fare
        *Q=MAX;
        inizio=inizio+MAX;
        if(inizio==dimP) inizio=0;
        FR(T+C, R-1,C,P,dimP,inizio, Q+1);
}
```

Quindi (essendo che deve restituire un intero, ormai ad occhio è ovvio ne invochi un'altra ancora per fare i suoi calcoli di match, in maniera filèse come sempre).

```
//nell'implement. su Mega dell'esercizio, uno potrebbe essere tentato di mettere come caso base
//anche dimP=0, qui inutile perché non scorre, nella funzione aux invece si.
int R1(int *T, int *C, int *P, int dimP){
        if(!C)    return 0;
        int local_match=R1(T+1, C-1, P, dimP);
        int aux = match_aux(T, C, P, dimP);
        if(aux > local_match);
        return aux;
        else
        return local_match;
}

int aux(int *T, int C, int *P, int dimP){
        if(!C || !dimP)   return 0;
        if(*T==*P)
        return 1+aux(T+1, C-1, P+1, dimP-1);
        else
        return 0;
}
```

Riporto la prova induttiva del MEGA, che il prof ha valutato convincente; spero possa essere altrettanto per chi legge questa guida.

```
//PASSO BASE: se la riga è finita ritorno 0, in quanto non ho match in una riga vuota
// se dimP è 0 ritorno 0, in quanto il match è finito
//PASSO INDUTTIVO: assumo vere PRE_ric e POST_ric rispetto a PRE e POST
```

//alla chiamata ricorsiva passo T+1 che è array di C-1 elementi definiti, e passo P che è array di dimP elementi definiti --> ok PRE
// la chiamata ricorsiva mi restituisce la lunghezza del match massimo di P[0..dimP-1] a partire dalla posizione T+1 e esaminando le posizioni successive fino a fine riga e pongo il risultato dentro temp
// aux mi restituisce la lunghezza del match di P[0...dimP-1] dalla posizione T e pongo il risultato dentro count
//se count>temp allora il match massimo è nella posizione T e restituisco count che è lunghezza di questo match --> ok POST
//se count<temp allora il match massimo è dalla posizione T+1 alla fine della riga e restituisco temp che è lunghezza di questo match --> ok POST

Restituire la lista di punti di innesto (nodi che abbiano almeno uno dei due puntatori left e right uguali a 0)

Ciò è realizzato tramite questa struttura:
struct innesto{bool l,r; nodo*N; innesto* next;}

Citando il prof:
Ad esempio, si consideri l'albero R= 2(3(_,4(_,_)),2(3(_,_),_)) certamente le foglie con valore 4 e 3 caratterizzate dai cammini, (0,1) e (1,0), sono punti di innesto che consentono di aggiungere 2 nodi ciascuno (dato che in essi mancano entrambi i figli), ma anche i 2 figli della radice (con valori 3 e 2) hanno solo 1 figlio e quindi anch'essi sono punti di innesto.
La lista dei nodi di innesto è formata da true o false a seconda della presenza del cammino:
(l=true,r=false,N=(0))->(l=true,r=true,N=(0,1))->(l=false,r=true,N=(1))->(l=true,r=true,N=(1,0)

Si crea quindi una funzione per restituire i punti di innesto dell'albero preso in ordine infisso:

```
innesto* f0(nodo*R){
        innesto *in, in_l, in_r;
        if(!R)    return 0;
        while(R){
                if(R->left){
                        R=R->left;
                        in_l->l=true;
                        in->N=0;
                }
                if(R->right){
                        R=R->right;
                        in_r->l=true;
                        in_r->N=1;
                }
                return conc(in, in_l) || return conc(in, in_r);
        }
}
```

Si usa anche una funzione di concatenazione, classica funzione *conc*:

```
innesto *conc(innesto *a, innesto *b)
{       //PRE(lista(a) e lista(b) corrette).
        if(!a)
                return b;

        a->next=conc(a->next, b);  //fa un po' di "riattacchi" inutili
```

```
        return a;
}       //POST(ritorna lista(c) tale che lista(c)=lista(a)@lista(b)).
```

Per commentare la funzione, qui svolta in maniera iterativa, si considera che da precondizione sia verificata la presenza di una delle due parti, sinistra o destra; se siamo in una foglia, l'innesto si ferma, ho già inserito a sinistra o destra, altrimenti devo per forza andare in un qualsiasi punto libero: a sinistra, quindi lo costruisco, altrimenti a destra; se si nota la funzione è qualcosa di simile ad una funzione che percorre un cammino.

Un'altra funzione che prende il risultato della precedente, quindi una lista di innesto con n campi l/r = true, vengono letti dei valori da INP e aggiunti x nuovi nodi; la parte vecchia viene deallocata e la lista rimasta è la parte finale di tutta la lista. I punti di innesto che corrispondono alle foglie consentono di aggiungere 2 nuovi nodi; se mettiamo in input -2, termina l'esecuzione, quindi anche mentre ho messo solo uno dei figli e non anche l'altro.

```
int f1(innesto*&Inn, ifstream & INP){
        int gil=0;          IN>>gil;
        if(!inn)            return 0;
        int innex=0;        INP>>innex;

        if(INP == -2)       return 0;
        if(Inn->l && Inn->r){
                return 1+f1(inn->left, IN) || 1+f1(inn->right, IN);;
        }
        else{
                if(Inn->left);      Inn->N->left=new nodo(inn, n);
                if(Inn->right);     Inn->N->right=new nodo(inn, n);
                innesto aux=inn->next;
                delete Inn;
                Inn=aux;
                return 1+f1(inn, IN);
        }
}
```

Di fatto tutto dipende dall'input; se ho entrambi i nodi vado ad aggiungere ad una delle due parti ricorsivamente, altrimenti mi prendo solo una delle due parti esistenti con la ricorsione, proseguo al nodo successivo qualora debba deallocare, per liberare la vecchia lista e considerare solo quella nuova.

Pattern matching realizzato su un array a 3 dimensioni visto come una tra un sottoarray dello stesso e l'array stesso

Tante parole per dire che in pratica, dato questo array è una sottoporzione senza andare a crearne uno.

Esempio supponiamo che lim1=3, lim2=4 e lim3=5, che n_el=60 e che i 60 elementi letti in Y siano i seguenti già distribuiti sui 3 strati di Y:

i 3 rettangoli rappresentano un sotto-array di Y che è costituito dai 18 elementi contenuti nei 3 rettangoli, quindi 6 elementi per ciascun strato. Come mostrato dalla figura i 3 rettangoli si trovano esattamente nella stessa posizione di ciascuno strato. Infatti un sotto-array è definito da 4 quantità:

i) riga r e colonna c dell'elemento in alto a sinistra, nell'esempio r=0 e c=2

ii) l'altezza H e la larghezza L del rettangolo, nell'esempio H=3 e L=2. La profondità del sotto-array è sempre lim1 dato che Y è completo.

Sul sottoarray considerato si scansione per V-fette, quindi: 1 4 0 2 2 2 0 0 3 0 1 1 3 1 2 1 1 1
Bello pesante sto esercizio, infatti risparmio inutili aggiunte e complicazioni filèsi.

La funzione restituirà true se il sottoarray ha un match contiguo con quello normale; si usa anche un'altra funzione che calcola la posizione dell'elemento dentro il sottoarray e viene usata la magica struttura *tripla* e si sconsiglia la definizione di array di copia o altre strutture ausiliarie.
La funzione che usa la roba sotto è questo frammento di codice:

```
int count=0;
   for(int e=0; e<(h*l*lim1-dimP+1); e++)          //garantisco che a partire da "e" ci siano ancora almeno dimP elementi per poter fare match
   {
          if(match(X,r,c,h,l,lim1,lim2,lim3,P,dimP,e))
          {
          OUT<<"trovato match a partire dall'elemento "<<e<<endl;
          count++;                    //conta quanti match non sovrapposti ho trovato
          e=e+dimP-1;     //evita match sovrapposti
          }
          else
          OUT<<"a partire dall'elemento "<<e<<" non c'e' match"<<endl;
   }
```

E quindi:

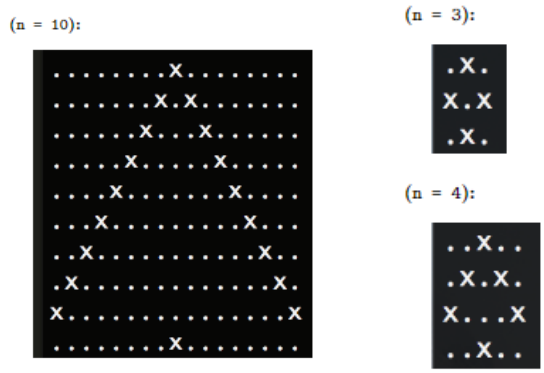
```
int*coord(int *X, int lim1, int lim2, int lim3, int el, int r, int c, int H){
          int *sub=X+r*lim3+c;              //prendo il pezzo di sottoarray con la dim che non sto usando in
                                            //quel momento e considera la scansione per righe, perché sovrapposte
                                            //se avessi HF = X+c*lim2+r;
```

```
        int str=el/(H%lim1);            //strato=visione dell'array dividendo per quello che non uso quindi
                                        //le righe e lim1, dimensione degli strati
        int row=el%H;
        int col=el/(H*lim1);            //colonna=elementi totali dividendo per le singole colonne * lim1,
limite degli strati che ho e che è il limite più simile a quello che ho.

        sub+= str*lim2*lim3 + rig*lim3 + col;    //solita scansione nelle v-fette
        return coord;
}
```

Scorro solo sulla dimensione dell'array nella funzione sotto: la scansione viene quindi utilizzata tra strati (numero totale degli elementi e divisione intera del limite lim1, di riferimento e considera solo elementi completi), righe, facendo la divisione intera per le righe, e le colonne, facendo la stessa operazione di prima degli elementi totali e dividendo per il numero di colonne moltiplicato per la dimensione che non sto usando, quindi lim1.

```
bool match(int* X,int r,int c,int H,int L,int lim1,int lim2,int lim3,int * P,int dimP,int el){
        bool found=false;
        for(int i=0; i<dimP && !found; i++){
                if(*coord(X, r, c, H, lim1, lim2, lim3, el+i) == *(P+i)
                found=true;
        }
        return found;
}
```

<u>Albero stilizzato (se input < 3, altezza non valida in output)</u>




```
int n;
cout << "Inserisci altezza";
if(n<3)  cout << "Altezza non valida";
else{
        int riga=n*2-(n/3);
        int albero=1;
        while(albero <= riga){
                punti=(riga-albero)/2;
                while(punti < albero){
                cout << ".";
                i++;
                }
                albero+=2;
        }
        //ancora una riga rimasta
```

```
        punti=(riga-1)/2;
        while(punti < albero){
                cout << ".";
                i++;
        }
}
```

Ordinare una lista in modo iterativo e ricorsivo

Quindi da L=4->2->1->5->0 si passa a 0->1->2->4->5.
Le funzioni devono essere:

```
void ordric(nodo*&L, nodo*& ord){
        if(!L)    return;
        if(!ord) L->next=ord;
        if(L->info <= ord->info){
                L->next=ord;
                ord=L;
        }
        else ordric(L->next, ord);
}
nodo* orditer(nodo* L){
        while(L){
                nodo*x=L;
                L=L->next;
                x->next=0;
                orditer_aux(L,x);
        }
}

nodo* orditer_aux(nodo* L, nodo*x){
        while(L && L->next){
                if(x->info <= L->next->info)
                L=L->next;
                if(x->info >= L->next->info)
                x->next=L->next;
                L->next=x;
        }
}
```

La prova di correttezza si basa sul fatto che si sa che una lista esiste e si assume che uno dei due elementi possa non essere crescente o non esserci comunque in un determinato caso degli elementi della lista. Nel caso ci sia, assumo tre casi:

- è minore o uguale dell'elemento dopo, vado solo avanti;
- è maggiore dell'elemento dopo, lo metto a metà e vado avanti

Riempire un array A[10][5] con nelem<=50 interi sse riga i e colonna j di A hanno entrambe elementi definiti e B[i][j]=true sse indici di riga sono crescenti (spiegato che cosa vuol dire)

Nel problema del prof, non si capisce cosa voglia dire avere indici crescenti, nel senso che usa una notazione insiemistica incomprensibile a leggerla, ma è spiegata semplicemente.
Con riga=[4,2,4,2,0]
e colonna=[0,2,4,4,2,4,0,0,2,0]

si ha che g[0]=2, g[1]=4, g(2)=5, g(3)=8, g(4)=9
Si sa che c'è il solito discorso del non tutto definito e si ha anche il fatto di avere elementio
Facile coi colori rispetto a chi non sa spiegarsi eh? Scriviamo questa magica funzione.
Al posto di restituire true e false, mettiamo direttamente false quando non si ha il discorso del crescente, true quando invece è verificata questa proprietà.

```
void lunghezza(int righe, int colonne, int nelem, int i, char dove){
        int righe_complete = nelem / colonne;
        int resto = nelem % colonne;
        if(dove == 'R'){
                if(i < righe_complete)
                return colonne;
                if(i == righe_complete)
                return resto;
                return 0;
        }
        else{
                if(i < resto)
                return righe_complete+1;
        }
}

//nel main
for(int i=0; i<10 && int LR=lung(10, 5, nelem, i, 'R');
        for(int j=0; j<5 && int LC=lung(10,5,nelem, j, 'C'); j++)

        for(int r=0; r<LR; r++){
                for(int c=inizio; c<LC; c++){
                if(A[i][r] == A[j][c])
                B[i][j]=true;
                inizio=c+i;
        }
        B[i][j]=false;
}
```

Qui si ricerca alla modalità filèse un match in indici crescenti in maniera non contigua, per cui si rende necessario utilizzare la variabile *inizio*, che mi permette di incrementare correttamente tutto quanto.
Come al solito, se esiste l'elemento nella riga e si sa che è definita, si definisce parallelamente lo stesso tipo di calcolo su una colonna e si cerca su entrambi se esiste un possibile valore, nel qual caso si ha un match, altrimenti nada.

Creare una nuova lista per ogni valore distinto in L

Esempio: sia L la seguente lista: 3->1->3->2->1->3->3->2->3, allora si vogliono ottenere 3 liste: una con i 5 nodi con info=3 di L, la seconda con i 2 nodi con info=1 di L e la terza con i 2 nodi con info=2 di L. Le 3 liste verranno inserite in unarray nodo**X nel modo seguente: X[0] deve contenere il primo nodo della lista con i 5 nodi con info=3, X[1] contiene il primo nodo della lista con i 2 nodi con info=1, e X[2] contiene il primo nodo della lista con i 2 nodi con info=2.

Si chiede di scrivere una funzione iterativa che permetta di restituire il numero di valori distinti di una lista e usa un'altra funzione ausiliaria che stacca i nodi uno alla volta e fa quanto descritto. Essendo che deve riciclare dei nodi, si dovrà creare una variabile apposita per farlo.

```
int S(nodo*L, nodo**X){
        nodo*Y=L;       int n=0;
        while(Y){
                if(*(Y)->info == L->info){
                        if(*X==0){
                                *X->info=0;
                                X->next=0;
                                n++;
                        }
                        else
                        addEnd(*Y, L);
                 }
                else{
                        while(Y->info != L->info){
                        Y=Y->next;
                        n++;}
                }
                Y=Y->next;
        }
}

void addEnd(nodo *Y, nodo *X){
        while(Y->next && Y)
        Y=Y->next;
        Y=X;
        return;
}
```

Con l'osservazione sul riciclare, ecco che si prende il nodo della lista e ad ogni iterazione si controlla se lo si ha già incontrato, nel qual caso lo si accoda ad una lista con quel valore, altrimenti si itera prendendosi il successivo valore diverso incontrato e fine dei giochi.

Lasciare e togliere dei nodi da una lista

Sia L= 0->3->2->4->100->0->200->100->11->5->2 e sia D=[0,2]->[1,3]->[3,1]->[0,2]. Le coppie di D vanno interpretate nel modo seguente, per una coppia [x,y] di D, dobbiamo lasciare x nodi di L e togliere i successivi y nodi di L. Eseguendo in questo modo quanto prescritto dai nodi di D, da L produrremo 2 liste, quella dei nodi di L che sono lasciatie quella dei nodi di L che sono tolti. Con L e D dati prima, avremo che lasciati=2->200->100->11, mentre tolti=0->3->4->100->0->5->2. Le 2 liste sono formate come segue: il primo nodo [0,2] di D richiede di lasciare 0 nodi di L e di togliere i primi 2 nodi, quindi 0->3 che infatti sono i primi 2 nodi di tolti. Quindi L è ora 2->4->100->0->200->100->11->5->2.

Si noti la presenza dei campi tieni e lascia in D per lo scopo. In L lasciamo i nodi e dall'altra li tagliamo.
La funzione iterativa è:

```
void Fiter(nodo*L, nodoD* D,nodo*& lasciati, nodo*& tolti){
        while(L && D){
                lasciati = conc(lasciati, taglia(L, D->lascia);
                tolti=conc(tolti, taglia(L, D->tolti);
                D=D->next;
        }
        lasciati=conc(lasciati, L);
}
```

Essa usa la seguente funzione ausiliaria:

```
nodo* taglia(nodo*&L, int n){
        nodo*ret=0;
        while(L && n>1){
                nodo*x=L;
                L=L->next;
                x->next=0;
                ret=conc(ret,x);
                n=n-1;
        }
        return ret;
}
```

Per la parte ricorsiva, invece si ha:

```
doppioN Fric(nodo*L,nodoD*D){
        if(!L || !D)        return 0;
        nodo *leave=tagliaric(L, D->lascia);
        nodo* cut=tagliaric(L, D->togli);
        doppioN list=Fric(L, D->next);
        list.L=concRic(l, list.L);
        list.R=concRic(l, list.R);
        return list;
}
```

La funzione Fric deve usare una funzione ricorsiva come segue:

```
nodo* tagliaric(nodo*& L, int n){
   if(!L || !n) return 0;
   if(n==1){
      nodo*x=L;
      L=L->next;
      x->next=0;
      return x;
   }
   else{ //qui n è maggiore di 1
      nodo*y=tagliaric(L->next,n-1);
      nodo*z=L;
      L=L->next;
      z->next=y;
      return z;
   }
```

Correttezza:

1) Dare l'invariante del ciclo principale di Fiter

R = L && R && esistono L->next && R->next elementi per i quali è possibile spezzarli in liste separate

2) Fornire la prova induttiva di Fric.

!D → metto solo i nodi nei lasciati

D esiste → allora vado a staccare una porzione di nodi L == D->togli && lascio una porzione di nodi L == D->lascia.

A quel punto induttivamente vale la stessa cosa fino alla fine e si ritorna la doppiaL.

3) Dare la prova induttiva di tagliaric.
L==0 || n==0→ no cut, no party
if(n==1)→ taglio l'ultimo nodo e lo ritorno
if(n > 1)→ prendo il nodo attuale chiamando ricorsivamente la funzione su L->next e n-1, poi stacco il nodo della lista L isolandolo e ritornando concatenato il nodo originale estratto.

Somma di indici che equivale ad un certo valore M in un array (iterativa e ricorsiva)

Abbiamo un intero m ed un array Y di n interi e vogliamo determinare se esiste un insieme di indici i1,...ik in [0,n-1] tale che Y*i1++Y*i2++...+Y*ik+=m. Chiamiamo un tale insieme di indici i1,...ik una soluzione per m in Y ed una rappresentazione di questa soluzione è un array R di n booleani tale che, per ogni j in [0,n-1], R[j]=true se j è nella soluzione i1,...ik altrimenti è false. Non si fa nessuna ipotesi sull'intero m e sugli interi contenuti in Y.

Scrivere una funzione iterativa bool F(int m, int* Y, int n, bool*R) che restituisca true se e solo se esiste una soluzione per m in Y e in questo caso R deve essere la rappresentazione della soluzione trovata. Se invece non ci sono soluzioni per m in Y,allora F deve restituire false (e valori qualsiasi in R)

a) Scrivere la PRE e POST della funzione F;
b) realizzare la funzione F assumendo di avere a disposizione la funzione bool add_one(bool *R, int n) che soddisfa la seguente coppia di PRE e POST.
PRE=(n >0 e R[0..n-1] è definito, e chiamiamo VAL(R) il valore rappresentato da R[0..n-1] interpretato come numero binario (con true=1 e false=0)); vedi esempio più sotto.
POST=(se VAL(R)+1 è rappresentabile come binario con n bit (cioè VAL(R)+1 < 2n) allora add_one restituisce false e modifica R in modo che rappresenti in binario VAL(R)+1, se invece VAL(R)+1 non è rappresentabile con n bit, allora add_one restituisce true)

Esempio: sia n=3 e R=[true, false, false], allora VAL(R)=4. Se invochiamo add_one con questo R e n=3, la funzione deve restituire false e R=[true,false,true] il cui VAL è 4+1. Se invece n=3 e R=[true,true,true], VAL(R)= 7 e visto che 7+1 non è rappresentabile con 3 bit, add_one deve restituire true (e qualsiasi valore per R). Intuitivamente, il valore booleano restituito da add_one ci dice se sommando 1 ad R causiamo overflow (true) oppure no (false).

Quindi se ho 100 è 4 in binario. In poche parole, controllo se ha senso in binario, altrimenti me ne esco.

```
bool add_one(bool *R, int n){
        int i=0;  bool isbinary=true;
        while(i<n){
                if(R[i]==1 && R[i+1] == 0){
                        R[i]=0;
                        R[i+1]=1;
                        isbinary=false;
                }
                else{
                R[i+1]=0;
                }
        }
        return isbinary;
}

//PRE=R array di booleani da riempire, r>0, n >0, Y array di interi definiti
bool F(int m, int* Y, int n, bool* R){
        bool go_out=false;      int sum=0;
```

```
        for(int i=0, j=0; i<n && j<n && !go_out; i++){
                if(sum == m)    go_out=true,
                sum+=Y[i];
                R[j]=add_one(R[j], i);
        }
        return go_out;
}
//POST=ritorno un booleano sse Y è soluzione per m in R
```

c) Scrivere l'invariante del ciclo principale della vostra funzione F
R=0<=i<=n && restituisce true sse ogni elemento R[0..n-1]==true

Per la parte ricorsiva, fare la stessa cosa dualmente, ma non seguendo l'idea iterativa.

```
bool F_RIC(int m, int* Y, int n, bool *R){
        if(*Y==m)       return true;
        if(n==0 || *Y==n)       return false;
        *R=add_one(*R, *Y);
        *Y=F_ric(m, Y+1, n, R);
}
```

Compitino in cui si richiedeva di trovare colonna con esattamente n match di P; copy and paste dal prof per capirci qualcosa

```
PRE=(T ha i primi n elementi definiti, n<lim2*lim3, P ha i primi dimP elementi definiti, n_m>0)
void trova_colonna(int*T,int n,int lim 1,int lim2,int lim3,int*P, int dimP, int n_occ, int& indice_c){
        int nru=n/lim3, neur=n%lim3, ntc=lim3;
        if(n<lim3)
        ntc=n;
        bool trovato=false;
        for(int c=0; c<ntc && !trovato; c++)//R
        {int lung=nru; if(c<neur) lung++;
        trovato=checkC(T+c, lung, lim3, P, dimP, n_occ);
        if(trovato) indice_c=c;
        }
}
POST=(se vediamo T come un array a 2 dimensioni X[lim2][lim3], allora indice_c ha per R-valore l'indice
minimo di una colonna di X che contiene esattamente n_m match di P[0..dimP-1], anche sovrapposti tra
loro; se nessuna colonna soddisfa questa condizione allora indice_c=-1)

bool checkC(int*inizio, int lungc, int lim3,int*P,int dimP, int n_occ){
        int conta=0;
        for(int i=0; i<lungc-dimP+1  && conta<= n_occ; i++)//R
        {
                if(match(inizio+(i*lim3), lim3, P, dimP)){
                conta++; cout<<"Match inizio="<<i<<endl;
                }
        }
        if(conta==n_occ) return true;
        else return false;
}
POST=(restituisce true sse la colonna contiene esattamente n_occ match di P)
```

R=(considerati gli elementi 0..i-1 della colonna come punti di inizio di match con P, conta=n. match di successo)

PRE=(inizio elemento di una colonna di un array[lim2][lim3], P ha dimPvalori, dimP>0)

```
bool match(int*inizio, intlim3, int*P, intdimP){
        bool ok=true;
        for(int i=0; i<dimP && ok; i++)
                if(inizio[i*lim3]!=P[i]) ok=false;
        return ok;
}
```

POST=(restituisce true sse inizio[0]=P[0], inizio[lim3]=P[1]...inizio[(dimP-1)*lim3]=P[dimP-1])

Compitino in cui si richiedeva di trovare strato con esattamente n match di P; copy and paste dal prof per capirci qualcosa

```
PRE=(T ha i primi n elementi definiti, n<=200, lim1*lim2*lim3<=200, P ha i primi
dimP elementi definiti, dimP<=20, n_m>0)
void trova_strato(int* T,int n,int lim1,int lim2,int lim3,int* P, int dimP, int n_occ, int
& indice_strato)
{
 int nrp=n/lim3, neur=n%lim3,r=0;
 bool  trovato=false;
 while(r<nrp && !trovato)//R
 {
  trovato=checkR(T+r*lim3, lim3, P ,dimP, n_occ);
  if(!trovato) r++;
 }
 if(trovato)
  {indice_strato=r/lim2;}
 else
  if(neur)
    trovato=checkR(T+nrp*lim3,neur,P,dimP,n_occ);
  if(trovato)     indice_strato=nrp/lim2;
} POST=(se vediamo T come un array a 3 dimensioni X[lim1][lim2][lim3], allora
indice_strato  ha per R-valore l'indice minimo di uno strato  di X che contiene una
riga che contiene esattamente n_m  match di P[0..dimP-1], NON  sovrapposti tra
loro; se nessuno strato soddisfa questa condizione allora indice_strato=-1)
```

```
PRE=(T ha dimr elementi definiti, P ne ha dimP)
bool checkR(int*T, int dimr, int* P, int dimP, int n_occ)
{
  int inizio=0, conta=0;
  while(inizio<dimr-dimP+1)  //R=(conta ok per 0..inizio-1)
   {
     if(match(T+inizio,P,dimP))
      {conta++;inizio=inizio+dimP;} //attenzione !!
     else
       inizio++;
   }
 if(conta==n_occ)
    return true;
 else
    return false;
}
POST=(restituisce true sse in T[0..dimr] ci sono esattamente n_occ match non
sovrapposti di P[0..dimP-1])
```

```
PRE=(T e P hanno dimP elementi definiti)

bool match(int*T, int*P, int dimP)
{
  bool ok=true;
  for(int i=0; i<dimP && ok; i++)  // R=(fino a i-2 bene, ok sse  i-1)
   if(T[i]!=P[i])
    ok=false;
  return ok;
}
POST=(restituisce true sse T[0..dimP-1]=P[0..dimP-1])
```

Inserimento ordinato di valori in liste distinte per ogni valore della lista

Se ho una lista: 3->1->3->2->1->3->3->2->3, voglio ottenere 3 liste.
Una con i cinque valori 3, una con i due valori ad 1 e la terza con 2 nodi=2.
In questa considerazione, il prof indica sempre restituire nell'ordine di apparizione, precisazione inutile perché è l'unico modo di restituirle (vuol dire che i valori distinti sono nell'ordine 3-1-2 e vanno restituite così le liste, quindi prima quella con i 3, poi quella con gli 1 e infine quella con i 2).

- Realizzare la funzione ricorsiva nodoE*insOrd(nodoE*X,nodo*y) che soddisfa la seguente PRE e POST:
PRE=(Lista(X) è ben formata e ordinata,y punta ad un nodo con campo next a 0,vX=Lista(X))
POST=(la funzione restituisce col return la lista ordinata ottenuta da vX aggiungendo ad essa il nodo y)

```
nodoE* insOrd(nodoE*X,nodo*y){
        if(!y || !X->next)          return new nodoE(y, 0);
        if(y->info < x->info->info){
                X->next=insOrd(X->next, Y);
                return X;
        }
        if(y->info > x->next->info->info && y->info < x->info->info){
                x->info->next=x;
                y->next=x->info;
                return insord(X->next, y->next);
        }
}
```

La funzione presentata realizza il distacco della lista in maniera ordinata, quindi prendendo il caso in cui il nodo sia da inserire all'inizio, sia da inserire in mezzo oppure sia da inserire dopo.
Se è minore, si prosegue ricorsivamente fino alla fine, se sta in mezzo, si collega.

Vi sarà poi una funzione ricorsiva che fa uso della precedente e restituisce la lista ordinata, quindi:

```
nodoE* SRic(nodo*L){
        nodoE* aux=0;
        if(!L)    return 0;
        else{
                nodo*x=L;
                L=L->next;
                aux->next=insord(Sric(L), x);
        }
        return aux;
```

```
}
```

Il magheggio di buttare una ricorsione dentro l'altra o, come nel caso appena presentato, di usare *insord* e poi chiamare la funzione stessa, ottimizza le chiamate ricorsive per poter usare la lista, dato che ne si estrae un nodo, la si ordina e la si ritorna ordinata correttamente.

Come al solito, data la PRE, nelle ricorsioni si considera di avere una lista esistente, ricorsivamente si stacca fino ai casi base, permettendone una corretta esecuzione fino alla fine.

Trovare l'intervallo dei valori di un albero binario (nella versione iterativa in un BST) migliore

Detto in termini normali, trovare estremo inferiore e superiore in un albero binario (parte ricorsiva) e in un albero BST (parte iterativa).

Immediato esempio pratico:
12(15(7(_,_),5(_,_)),3(4(_,_), 2(_,_))) e y=6, la coppia cercata è [5,7].

Seguendo la mia spiegazione di 3 righe, si capisce ciò che chi non sa spiegarsi dice.
In particolare, essendo due valori da restituire, si intende che questi vengano ritornati per riferimento, non ci sono altri possibili modi di eseguire questa operazione.
Siore e siori, la ricorsione:

```
void H(nodo* T, int y, int& x1, int& x2){
        if(!T){
                x1=x2=y;
                return;
        }
        if(T->info > x1 && T->info < y)
        x1=T->info;
        if(T->info < x2 && T->info > y)
        x2=T->info;
        H(T->left, x1, x2, y);
        H(T->right, x1, x2, y);
}

void H1(nodo* T, int y, int& x1, int& x2){
        if(!T){
                x1=x2=y;
                return;
        }
        while(T){
                if(T->info > x1)
                        if(T->info < y)
                        x1=T->info;
                        else
                        T=T->left;
                else{
                if(T->info < y)
                x1=T->info;
                else
                T=T->left;
                }
        }
```

}

Restituire la porzione contigua più lunga rispetto ad una lista concatenata

Data una lista concatenata L= 2->0->-1->3->0->1->2->10->2->-1, una sua sotto lista è una porzione contigua di L, come, per esempio: 3->0->1 (L= 2->0->-1->3->0->1->2->10->2->-1), 2->0->-1->3 (L= 2->0->-1->3->0->1->2->10->2->-1) e 2->-1(L= 2->0->-1->3->0->1->2->10->2->-1). In particolare a noi interessa una sottolista crescente e con dei campi inizio e fine che servono proprio a questo scopo (definiti come immaginabile all'interno della struttura dell'esercizio doppioN). Quella azzurra è quella più lunga, quella gialla un'altra dello stesso tipo.
Per la parte iterativa, utilizzo una funzione che restituisce proprio la lista così definita:

```
doppioN Fiter(nodo*L){
        doppioN local;
        while(L){
                nodo*aux=L;     int lung=0;
                if(aux->info <= aux->next->info && aux->next){
                        aux=aux->next;
                        lung++;
                }
                if(!L.inizio)       return doppioN(L,aux,lung);
                if(local.lung < lung){
                        local.inizio=L;
                        local.fine=aux;
                        local.lung=lung;
                }
                else L->next=aux;
        }
        return local;
}
```

Per la parte ricorsiva, utilizzo una funzione chiamata *Frec*, che fa praticamente la stessa cosa. La funzione userà un'altra ricorsiva chiamata *Aux*, che fa la stessa cosa. La logica è di usare una funzione che fa i confronti e nella funzione principale verifica quale sia la più lunga restituendola.

```
doppioN Frec(nodo*L){
        if(!L)     return doppioN();
        doppioN x=aux(L);
        doppioN y=Frec(L->next);
        if(x.lung >= y.lung)       return x;
        else       return y;
}

doppioN Aux(nodo*L){
        if(!L)     return doppioN();
        doppioN x=0;
        if(L->info < L->next->info){
                doppion*x=Aux(L->next);
                x.inizio=L;
                x.lung++;
                return x;
        }
        else       x=doppioN(L, L, lung=1);
```

```
}
```

Si hanno inoltre due funzioni, una iterativa *Giter* che restituisce la sottolista individuata da A e il valore di L togliendo la sottolista di prima e una ricorsiva che fa la stessa cosa, chiamata *Grec*.

```
nodo* Giter(nodo*& L, doppioN A){
        if(L->info == A){
                L=A->fine->next;
                A->fine->next=0;
                return A.inizio;
        }
        nodo* aux=L;
        while(aux && aux->next){
                if(A.fine != aux->next)
                aux=aux->next;
        }
        L=A->fine->next;
        A->fine->next=0;
        return inizio;
}

nodo* Grec (nodo*& L, doppioN A){
        if(!L || !A)        return 0;
        if(L == A.inizio){
                L=A.fine->next;
                A.fine->next=0;
                return A.inizio;
        }
        else return Grec(L->next, A);
}
```

Per completare il tutto, scriverò invariante del ciclo di *Fiter* e la dim. induttiva di *Aux* e *Frec*.

1) R=L esistente && ritorno x sse L->info < L->next->info && L->next altrimenti ritorno L;
2) Dim. induttiva Aux:
   Non esiste il nodo su cui lavorare→ ritorno la lista vuota
   Esiste, ricado nel condizionale "se minore del successivo" e scorro ricorsivamente, assumendo l'esistenza dell'elemento successivo, per poter staccare in maniera valorizzabile i nodi, altrimenti ricompongo la lista L per riferimento ricordandomi da dove partivo, quindi il nodo X. In entrambi i casi POST è valida.
3) Dim. induttiva Frec
   Non esiste la lista o una delle due→ritorno 0
   Una delle due esiste e a quel punto controllo, con l'ausilio della funzione ausiliaria appena creata come muovermi. Se minore la prima della seconda, ritorno la prima, altrimenti l'altra, perché ci interessa il discorso del crescente e quella trovata prima a sx per ricorsione.

<u>Percorrere un albero binario in larghezza producendo una lista come risultato e poi eliminazione dei nodi con info ripetuti</u>

Dato un albero binario, lo vogliamo percorrere in larghezza, cioè per livelli (prima la radice, poi i suoi figli, poi i figli dei figli e ogni livello viene percorso da sinistra a destra)producendo una lista L di nodi che puntano ai nodi dell'albero secondo l'ordine determinato dal percorso in larghezza. Quindi il primonodo di L punta alla radice, il secondo al figlio sinistro della radice (se esiste), il terzo al figlio destro (se esiste) e così via per livelli progressivi dell'albero.

Si chiede di scrivere una funzione ricorsiva,void faiRic(nodoE*&L),che soddisfi le seguenti specifiche: PRE=(L è una lista ben formata di nodoEche puntano anodi di un albero binario e la sequenza dei nodi puntati segue l'ordine del percorso in larghezzadell'albero e nessuno dei nodi puntati è discendentedi un altro dei nodi presenti, indichiamo convL il valore iniziale di L)

POST=(alla fine, L sarà la sequenza di nodoE che si ottiene aggiungendo a vL i nodi nodoE che puntano a tutti i discendentidei nodi puntati da vL ordinati secondo il percorso in larghezza)

```
void faiRic(nodoE*&L){
        if(!L)     return;
        if(L->info->left) insert(L, L->info->left);
        if(L->info->right) insert(L, L->info->right);
        faiRic(L->next);
}

void insert(nodoE*& L, int x){
        if(!L)     L->next=new nodoE(x, 0);
        else insert(L->next, x);
}

void filtraIter(nodoE*&L){
        nodoE* aux=L;
        while(L && L->next){
                check(L, L->info->info);
                L=L->next;
        }
        L=aux;
}

void check(nodo*& L, int x){
        while(L && L->next){
                if(x = L->next->next->info){
                nodo* y=L->next;
                L->next=L->next->next;
                delete y;}
                else{
                        nodo*y=L->next;
                        L->next=0;
                        delete y;
                }
        }
}
```

Teoria

<u>Dare opportune PRE e POST alla funzione F oppure dare delle POST tra le alternative:</u>

```
int F(Nodo* a) {
if(!a) return 0;
if (a->right) return 1+F(a->right);
if(a->left || a->right) return 2+F(a->left)+F(a->right);
return 3+F(a->left)+F(a->right);
```

PRE=(nodo a non vuoto)
POST=(ritorna un numero di nodi uguale alla somma dei nodi presenti a sinistra/destra più i nodi effettivi)

Si consideri la seguente funzione sulle liste concatenate e si scelga le affermazioni valide per essa. Useremo vL(n) per indicare la lista originale, mentre L(n) indica la lista finale. Il primo nodo di una lista è in posizione 0, il secondo è in posizione 1, e così via.

```
nodo* g(nodo*& n, int k, int m)
{
  if(!n) return 0;
   if(k==m)
    {
      nodo*x=n->next;
      n->next=0;
      return x;
    }
  else
    {
      nodo*x=n;
      n=n->next;
      x->next=g(n,k+1,m);
      return x;
    }
}
```

Risposte:

-L(n) consiste solo del nodo in posizione m-k di vL(n)
-restituisce col return la lista che si ottiene da vL(n) eliminando il nodo in posizione m-k, se un tale nodo c'è e altrimenti restituisce vL(n)
-se vL(n) ha lunghezza minore di m-k allora alla fine della funzione n=0

Si segue il flusso della funzione e il suo ragionamento; in effetti l'unica condizione che determina lo stacco di un nodo è k==m, però determina tutto anche in base al parametro n, che determina l'andata della ricorsione. Se la lista ha lunghezza minore la ricorsione si interrompe prima.

```
nodo* f(nodo*&L, int k){
        if(!L)    return 0;
        if(k>0)   return f(L->next, k-1);
        else{
                nodo*x=L;
                L=L->next;
                if(L)
                        x->next=f(L->next, k-1);
                else
                        x->next=0;
        }
        return x;
}
```

Risposte:
-se vLista(L) contiene più di k+1 nodi, allora alla fine della funzione della funzione Lista(L) è costituita da più di k nodi di vLista(L)
-se k è pari e vLista(L) contiene 2^k nodi, allora la funzione restituisce col return una lista di k/2 nodi
-se vLista(L) contiene k+1 nodi, allora alla fine della funzione Lista(L) contiene k nodi

---

Esercizi di stampa rispetto ai programmi

```
int X[]={1,2,3,4,5}, *Y, *Z, **Q;
Y=X+2;
*Z=*Y-2;
Q=&Y;
Z=*Q;
```

Il programma sopra è sbagliato, dato che stiamo dereferenziando Z, che non ha al momento nessun valore.

```
int*& f(int **y, int *z){
        int a=0, *b=&a;
        **y= *b;
        b=z;
        *y=b;
        return *y;
}
```

La funzione data qui sopra è corretta, in quanto tutte le assegnazioni dei puntatori/oggetti considerati puntano a cose valide/oggetti esistenti.

```
int*f(int **p){int b=3,*x=&b; **p=*x; x=*p; return x; }
int main() {int y=5, b=2,*q=&b; *f(&q)=y*2; cout<<y<<b<<*q;}
```

Il programma stampa 5 10 10 quindi è corretto, considerando che tutti i puntatori puntano ad oggetti solidi e presenti; non ci sono *dangling pointer*, che il prof riesce come sempre a confondere. Nei suoi esempi significa semplicemente che ci si riferisce a dei puntatori che non hanno un oggetto puntato oppure a variabili locali deallocate; quindi si punta a qualcosa di inesistente o che ancora non esiste.
In particolare da sopra, la variabile q punta a qualsiasi cosa arrivi da y*2 e in quanto alias ne assume facilmente il valore. Dentro la funzione f, x punta a b variabile locale, il puntatore locale viene correttamente valorizzato a b, quindi x, che viene ritornata in stato utile.

```
int*& f(int **y, int*z){
        int a=0, *b=&a;
        **y = *b;
        b=z;
        *y=*b;
        return *y;
}
```

La funzione è corretta, in quanto anche qui ogni oggetto è esistente e quindi assegnato.

```
int X[]={1,2,3,4,5}, *Y, *Z, **Q;
Y=X+2;
*Z=*Y-2;
Q=&Y;
Z=*Q;
std::cout << **Q << ' ' << X[1] << ' ' << X[2] << std::endl;
```

Stessa cosa di sopra, programma sbagliato

```
int *f(int **p){int b=3; *x=&b; x=(*p)+1; return x-2;}
int main(){
        int b[]={1,2,3,4}, *q=b+2; *f(&q)=*q;
        std::cout <<b[0]<<b[1]<<b[2]<<b[3];
}
```

Programma errato, in quanto x non esiste e compilato dà un errore di scoping.

```
int **f(int* p){int **x=&p; *(*x+1)=*p+2; return x; }
int main(){int b[]={2,3}, *q=b; **f(q)=*q+2; std::cout<<b[0]<<b[1]<<*q;}
```

La stampa produce 444, in quanto q punta all'array b e la chiamata della funzione provoca l'assegnazione ad ogni suo elemento del valore 4, cosa che viene poi anche stampata.

```
int ** F(int**p){(*p)++; return p;}
int main(){
   int a[]={0,1,2,3}, *q=&(*a);
   **F(&q)=a[3]+1;
   cout<<q[0]<<q[1]<<q[2]<<a[3];
}
```

La stampa genera 4233, in quanto q è uguale a 0, per riferimento. Poi ad F si passa q (quindi a avrà side effects). In essa si andrà semplicemente a sommare 1 al primo elemento, che assumerà poi valore 4. Quindi primo elemento valore 4 per side effects, poi gli altri due elementi sono 2 e 3 in quanto in F il puntatore del primo elemento viene spostato da 0 ad 1. Per queste motivazioni si genera la data stampa.

```
int**F(int*x){int**p=&x; (*x)++; return x;}
main(){int a=2,*q=&a; **F(q)=4; cout<<a<<endl;}
```

La funzione F contiene un errore di tipo: deve restituire un int**, ma return x; restituisce un valore int*. Quindi il programma non compila neppure. Altrimenti il main sarebbe ok rispetto ai tipi.

```
int * f (int **a){int b=10, *p=&b; *p=**a; *a=p; return *a;}
```

La funzione considerata ritorna un dangling pointer, perché come sempre non stiamo puntando a nulla in a e quindi l'assegnazione non ha senso.

---

int*&f(int *&x, int*y){int **z = &y; *z=x; return *z}

La funzione restituisce una dangling reference, perché il doppio puntatore non è inizializzato dunque eseguendo queste operazioni.

Supponiamo di avere un array int y[5][5][10] riempito con nele<=250, descrivere la scansione con h-fetta

(nele/50)*10 + (m < (nele%50/10? 10:0) + (m==(nele%50)/10? (nele%10):0)

Per le h-fette, usiamo il limite lim2*lim3 + la stessa scansione e la scansione con righe e colonne, usando la stessa dimensione.

```
nodo*F(nodo*&L, int k, int n){
        if(!L)     return 0;
        if(n%k > 0)        return F(L->next, k, n+1);
        else{
                nodo*x=L;
                L=L->next;
                x->next=f(L, k, n+1);
                return x;
        }
}
```

Risposte:
-se vLista(L) contiene n%k+1+b*k ndi con b>=0, allore la funzione restituisce col return b+1 nodi
-se vLista(L) ha n%k + b*k nodi con b>=0, allora alla fine della funzione Lista(L) ha b nodi meno di vLista(L)
-se vLista(L) non ha più di n%k nodi, allora la funzione restituisce 0 col return

---

<u>Esercizi sul capire tipi e dereferenziazioni su array</u>

Sotto alcuni esempi di dereferenziazioni: il principio base è capire che l'array di per sé è un puntatore; prendiamo per esempio A[100], che avrà tipo int*
Per esempio X[5][10], avrà tipo int (*)[10], mentre X[5][5][10] avrà tipo (*)[5][10]; per il calcolo effettivo in memoria, per esempio su X, sarà un tipo 10*4, mentre per il successivo X sarà 5*10*4.
Se dereferenziamo un puntatore otteniamo l'oggetto puntato;
attenzione solo a somme e semplificazioni.

Per esempio:
((A[-3]+3)[2] considerando array A[5][6][8][5]
Esso sarà 6*8*5*4 – 3*8*5*4 + 5*5*4, perché gli ultimi due elementi vengono direttamente sommati in quanto facenti parte della stessa dereferenziazioni (parliamo di 3 e 2 che diventa 5). Ciò viene fatto solo in caso di stessa dereferenziazione.

X[10][11][12]
-Che tipo ha?
Il tipo è (*)*11*12*dimensione tipo
-Che tipo e valore ha l'espressione (*X-5)-10?
Sapendo che *X=(*)*12, l'espressione sarà 12-5-10*dim_tipo

-Che tipo e valore ha l'espressione X[20]-3?
11*12*20 – 3*12*20

Dato char X[5][10][10][10], specificare le seguenti espressioni:
- *(X[-4]+2)-3 → -4*10*10*4 + 1*10*4
- X[-2][-2]+2 → -2*10*10*10*4 - 2*10*10*4 + 2*10*4

Dato int T[3][5][6] che tipo ha *(T[8]-2) ?

A T vengono applicate 2 *, una esplicita e l'altra contenuta nel subscripting in T[8]=*(T+8), visto che il tipo di T è: int (*)[5][6], applicando 2 stelle (dereferenziazioni) si ottiene un int*

Dato X[4][5][6][8], dare il valore dell'espressione ((*X) - 2)[2]-1.

La risposta corretta è: il tipo è int(*)[8] e il valore X-8*4

Data la seguente dichiarazione:
double X[5][4][6][6][8], indicando con X l'Rvalore di X, scegliere la risposte che contiene il tipo e il valore dell'espressione:
(((X+3)-2)-4)+3

La risposta corretta è: il tipo è: double (*) [4][6][6][8]
e il valore è: X

Data la seguente dichiarazione:
int B[3][3][4][5][6], indicando con B l'R-valore di B, scegliere la risposta che indica il tipo e il valore della seguente espressione:
(B[2]-3)[5]+4

Le risposte corrette sono:
il tipo è: int(*)[6]
il valore è: B-(3*4*5*6)*4+5*(4*5*6)*4+4*(5*6)*4,
il tipo è: int(*)[5][6]il valore è: B+2*(3*4*5*6)*4+2*(4*5*6)*4+4*(5*6)*4

---

int B[3][3][4][5][6], indicando con B l'R-valore di B, scegliere la risposta che indica il tipo e il valore della seguente espressione:
(B-2)[3][5]+4

La risposta corretta è: il tipo è: int (*)[5][6]
il valore è: B+ (3*4*5*6)*4+5*(4*5*6)*4+4*(5*6)*4

((X[0]-2)[2] con array float X[3][4][5][6][8]

X[0] è derefenziare due volte l'array poi c'è l'operazione -2+2 tra dereferenziazione e subscripting che comporta un altro passaggio. Tipo finale (*)[6][8].

```
int x=1, y=2, *p=&x, *q=&y, **Q;
p=q;
*Q=p;
**Q=*p+*q;
cout<< *p<<*q<<**Q<<endl;
```

Quale delle seguenti risposte è corretta?

*Q viene dereferenziato che non ha nessun valore e non si può fare.

```
int x=1, y=2, *p=&x, *q=&y, **Q;
p=q;
Q=&p;
**Q=*p+*q;
cout<< *p<<*q<<**Q<<endl;
```

Quale delle seguenti risposte è corretta?

p=1, successivamente l'assegnazione *Q=p diventa 2 e **Q diventa 4; per side effects la modifica si ripercuote su tutte le altre variabili.

```
void f(int* a){a[2]=a[1]; a[1]=a[0]*a[2];}
main(){int x[]={0,1,2,3,4}; f(x+2);
cout << x[0]<<x[1]<<x[2]<< x[3]<<x[4]<<endl;}
```

La risposta corretta è: stampa 0 1 2 6 3

Alla funzione f si passa il valore [2], quindi il terzo elemento, che assume poi il valore del secondo (quindi 1) e poi il secondo diventa la moltiplicazione dell'1 con il resto degli elementi, quindi diventa 2. Ritornando si ha questa assegnazione

---

```
int x=1, y=2, *p=&x, *q=&y, **Q=&p;
p=q;
*q=*p*2;
**Q=*p+*q;
cout<< *p<<*q<<**Q<<endl;
```

La risposta corretta è: stampa 8 8 8
p è 2, gli viene assegnato il valore 4 con la moltiplicazione e **Q diventa 4+4=8 che per side effects ripercuote la modifica all'indietro e si genera questo risultato.

```
void f(int*& a, int *b){int*x=a; a=b;b=x;}
main(){int x=1,y=2,*p=&x, *q=&y; f(p,q); cout << *p<<*q<<endl;}
```

La risposta corretta è: stampa 2 2

p assume il valore il valore 1, q successivamente prende il valore 2 e poi alla funzione vengono passati entrambi; le assegnazioni dall'altra parte assegnano ad entrambi i puntatori lo stesso valore, quindi diventano entrambi 2 che viene stampato.

---

int *f(int *x)(int **y=&x, c=10; *x=c; return *y;}

La funzione è corretta e non ritorna un dangling reference, perché ritorno y che può essere o non essere valorizzato, ma prendo un valore locale con c=10 e ritornando questo non viene deallocato, quindi tutto corretto.

Casting (pezzi di codice ripresi dall'ottimo sito GeeksforGeeks)

Reinterpret cast: Cambia il tipo di un puntatore; qualsiasi cosa fatta da questo casting il compilatore me lo lascia fare ma ottengo sempre dati strani (cosa che capita sempre negli esercizi del prof ed è la risposta da dare)
Static cast: Cambia un tipo X ad un tipo indicato tra parentesi come sotto:

```
int main()
{
    float f = 3.5;
    int a = f;
    int b = static_cast<int>(f);
    cout << b;
}
```

L'esempio sopra è valido, ma se cercassi di fare la cosa seguente:

```
int main()
{
    int a = 10;
    char c = 'a';

    // pass at compile time, may fail at run time
    int* q = (int*)&c;
    int* p = static_cast<int*>(&c);
    return 0;
}
```

Sto cercando di fare un casting illegale e avrei un errore:

```
[Error] invalid static_cast from type 'char*' to type 'int*'
```

Const cast: Toglie il const da una variabile; utile per esempio in funzioni dove passo dei dati non costanti (come sotto)

```
int fun(int* ptr)
{
    return (*ptr + 10);
}
```

```
int main(void)
{
    const int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    cout << fun(ptr1);
    return 0;
}
```

Se però cambiassi il valore di qualcosa dichiarato come costante, avrei un *undefined behaviour*, come nel caso riportato sotto.

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

int fun(int* ptr)
{
    *ptr = *ptr + 10;
    return (*ptr);
}

int main(void)
{
    const int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    fun(ptr1);
    cout << val;
    return 0;
}
```

Si modifica quindi quanto già dichiarato come const

Esercizi sui casting e su costanti

const int x=10, *y=&x;
int *z=const_cast<int*>(y);
(*z)++;
std::cout << x << *z << *y << endl;

Stamperà 10, 11, 11 in quanto x resta uguale, il puntatore intero z toglie il const ad y che punta al valore costante, lo dereferenzia sommandogli 1, pertanto la modifica fa side effects sui due puntatori z e y, portando ai valori qui stampati.

---

int x=10, *y=&x;
double*z=reinterpret_cast<double(y);
std::cout << *z << std::endl;

La funzione sopra, come accennato prima, darà un numero strano a compile time, perché raddoppio la size dell'oggetto puntato e lo dereferenzio, problematico quindi.

```
const int x=10, * y=&x;
int z=x;
y=&z;
z++;
cout << *y <<endl;
```

Il programma stampa semplicemente 11, in quanto viene semplicemente creato un alias della variabile in questione che viene poi successivamente modificata, senza toccare la costante di riferimento.

Esercizi su try e catch

```
void F(int z)
{
 if(z%3)
   throw true;
 else
   throw z%3;
}
main()
{
 bool doit=true; int i; cin >>i;
 try{
   while(doit && i < 20)
    {
   cout << i<< endl;
   try
    {
      F(i);
      i++;
    }
   catch(bool x)
    {if (x) i++; else doit=false;}
   i++;
    }
 }
 catch(int x)
   {cout <<i<<endl;}
}
```

La funzione sopra riportata stampa 9 9, in quanto stampa il primo 9, dopodiché chiama F una volta e vedendo che 9 % 3 non da resto, fa il throw del numero diviso tre che viene subito preso dal catch e genera quanto riportato.

Se immettessi come input 2?

```
int F(int z){
```

```
        if(z >= 3)          throw z;
        else                return z+2;
}
int main(){
        try{
                int x=3, y; cin >> y;
                try{
                        x=x+y; F(x);
                }
                catch(int y){x=x+y; throw(x);     }
        }
        catch(int x){
                cout << x;
        }
}
```

Sostanzialmente dal programma si nota che vengono eseguite due addizioni, da una parte e dall'altra e quindi al catch viene preso 2 volte due addizionato, dunque diventa 12.

Dando come input 3 cosa succede?

```
int F(int z){
        if(z >= 3) throw z;
        else return z+2;
}

int main(){
        try{
        int x=3; cin >> y;
        try{
                x=x+y+F(y);
        }
        catch(int y){
                x=x+y; throw(x);
        }
        cout << x << endl;}
        catch(int x){
                x--; cout << x << endl;
        }
}
```

Seguendo l'ordine delle chiamate darà 3 nella chiamata della funzione F(y) che ritorna il numero 6 passato all'altro blocco catch di tutta la funzione che ritorna 6-1=5.

<u>Esercizi su virgola mobile</u>

Chiarimenti sul complemento a due e trova il numero
Negli esercizi in cui chiede "che numero è X" rispetto al complemento a due si consideri di fare 256 meno il numero, tipo sotto. Se ho 67, faccio 256-67=189 (che è infatti il numero sotto; allo stesso modo con quelli dello stesso tipo.

Per il complemento a due basta prendere il numero decimale, convertirlo in binario, come qui sotto, invertire tutti i bit e sommare infine 1.
Attenzione che basta prendersi il numero e tipo 67, diventa: (letti poi dal basso verso l'alto!)
67/2=1
33/2=1
16/2=0
8/2=0
4/2=0
2/2=0
1
Quindi il numero è 1000011 e si invertono i bit, quindi 0111100, a cui si aggiunge 1, che infatti dà 10111101

Per quanto riguarda il floating point, si considerano i 3 pezzi, esponente, mantissa e bit di segno.
Se il bit di segno è 0 allora il numero è positivo altrimenti 1 per indicare che è negativo.
Per esempio, prendendo il valore 0,66, come l'esempio sotto (0 0101 0110 – segno – mantissa – esponente)

Sappiamo che è positivo, quindi subito bit a 0 all'inizio. Per calcolare la mantissa si procede prendendosi il numero e facendo i singoli prodotti per 2, determinando se 1 o 0, leggendo poi al contrario.
Esempio pratico che lo fa capire:

0,66*2=1 (si riparte poi da 0, perché sempre tra 0 ed 1).
0,32*2=0
0,64*2=1
0,28*2=0

Si noti come leggendola al contrario diventa la mantissa (0101).
Sappiamo inoltre che l'esponente è determinato da una potenza $2^{k-1}-1$.
Il bit da prendere elevato alla k è proprio quello che abbiamo appena trovato, quindi la mantissa (0101 = 5 in decimale). Per poter dare 5, deve essere 6-1.
Quindi diventerà: $2^{6-1}-1$. A noi interessa il bit k che permette di dare 5, quindi 6, che è proprio 0110.
Questa è tutta la codifica, quindi:
-segno, si vede subito
-mantissa, si fa come sopra moltiplicando per 2 e prendendosi il decimale di riferimento, azzerando se oltre 1 e leggendoli poi al contrario
-esponente, si prende il bit che permette di dare 2^k-1

Magari leggendo le parole singolarmente dice poco, ma l'esempio chiarisce tutto direi. È molto facile anche questo come argomento, basta saperlo spiegare o altrimenti vedere e provare da sé convincendosi.

Con 8 bit a disposizione, nella rappresentazione degli interi in complemento a 2, qual è il valore binario che rappresenta -67?

La risposta corretta è: 10111101

Con 8 bit a disposizione, nella rappresentazione degli interi in complemento a 2, che valore in base 10 rappresenta il valore -121?

La risposta corretta è: 135

Con 8 bit a disposizione, nella rappresentazione degli interi in complemento a 2, che valore in base 10 rappresenta il valore -8?

La risposta corretta è: 248

Nella rappresentazione floating point con 1 bit per il segno, 4 per la mantissa e per l'esponente, come verrà rappresentato il valore 0,66?

La risposta corretta è: 0 0101 0110

Nella rappresentazione floating point con 1 bit per il segno, 4 per la mantissa e 4 per l'esponente, come verrà rappresentato il valore 4,25?

La risposta corretta è: 0 0001 1001

Assumendo di avere 8 bit a disposizione per una rappresentazione di reali floating point. Gli 8 bit sono usati nel modo seguente: 1 bit per il segno, 4 per la mantissa e 3 per l'esponente. Che valore in base 10 rappresenta il valore binario 1 1 1 0 1 1 0 1?

La risposta è -7,25

Se invece devo fare l'operazione contraria come qui sopra (caso -7,25).
Noto subito il bit di segno, che è -1 perché il numero è negativo.
L'esponente è 101= 5 – 3 = 2 (dove 3 sono i bit usati per dare l'esponente come risultato).

L'esponente lo uso come potenza per denormalizzare, quindi sapendo che la mantissa è 1101 e la potenza è positva mi muovo verso destra, dunque: 1,1101 * 2^2= 111,01

Gli ultimi due bit sono 0 e 1, che sarebbero 0,5*0 e 0,25*1.
Il decimale del numero è quindi 0,25 e come si vede dall'operazione sopra ho 111, che sarebbe 7.
Il risultato finale è quindi proprio -7,25.