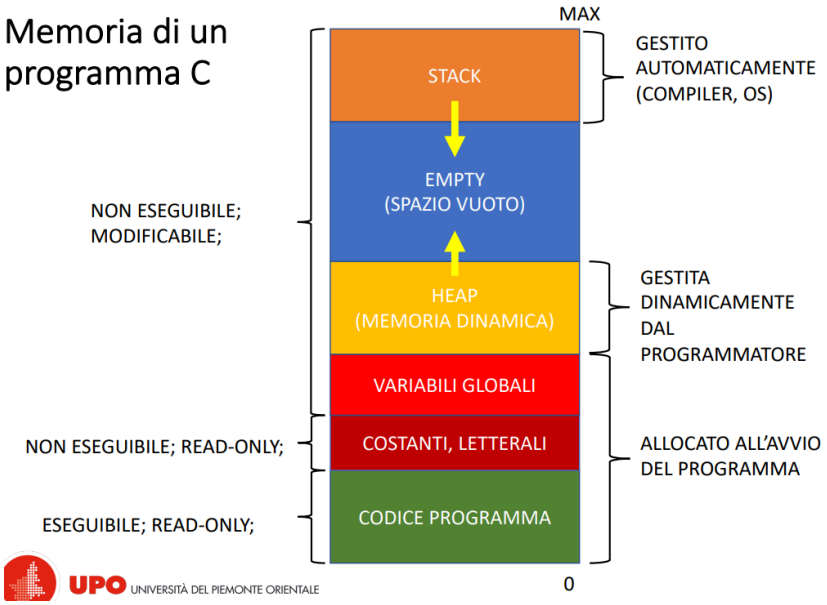
PROGRAMMAZIONE 2 FONDAMENTI

GESTIONE DELLA MEMORIA IN C

- **allocazione dinamica**: non conosco la dimensione del file o del dato in input;



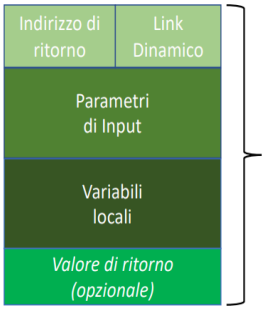
- Dimensione iniziale 0, MAX memoria massima allocata;
- **Codice programma**: linguaggio macchina;
- **Costanti, letterali**: variabili che non cambiano;
- **Variabili globali**: variabili visibili in tutto il programma, modificabile, la dimensione non può essere modificata durante l'esecuzione;
- **Heap (memoria dinamica)**: area modificabile e gestita dal programmatore;
- **Stack**: area di memoria gestita automaticamente, dinamica;
- **Empty**: spazio in cui si espandono lo stack e heap;

STACK

Struttura di dati LIFO (Last-in-first-out): l'ultimo elemento che viene inserito è il primo ad essere estratto (ex. pila di piatti);
Lo stack viene utilizzato per contenere i record di attivazione;
Il record di attivazione rappresenta l'universo della funzione (spazio di lavoro), contiene tutti i parametri formali + tutto lo spazio di memoria per le variabili;
Viene creato quando chiamato (**push**) e la funzione utilizzerà tutti i dati presenti nel record, alla fine viene rimosso (**pop**) e verrà liberata la memoria per altri record di attivazione;
Lo spazio allocato per il record è fisso;

- **indirizzo di ritorno**: indirizzo del codice dove devo tornare quando termina la funziona;
- **link dinamico**: (non lo utilizzeremo) indirizzo che contiene il record della funzione chiamante;
- **Parametri di input**: variabili utilizzate nella funzione;
- **Variabili locali**: allocato spazio nel record;
- **Valore di ritorno**: (in c non c'è) rappresenta il valore di ritorno restituito, (funziona void non ha valore di

Record di attivazione



Dimensione del record di attivazione non fissa! (cambia da funzione a funzione).

Tuttavia è sempre **possibile a priori** conoscere la dimensione del record di attivazione di una data funzione!

Perché opzionale?

ritorno), (tipicamente quando la funzione termina la computazione il record viene cancellato, per eliminare questo il c lo restituisce in un registro macchina), meglio utilizzarlo per un fattore teorico;

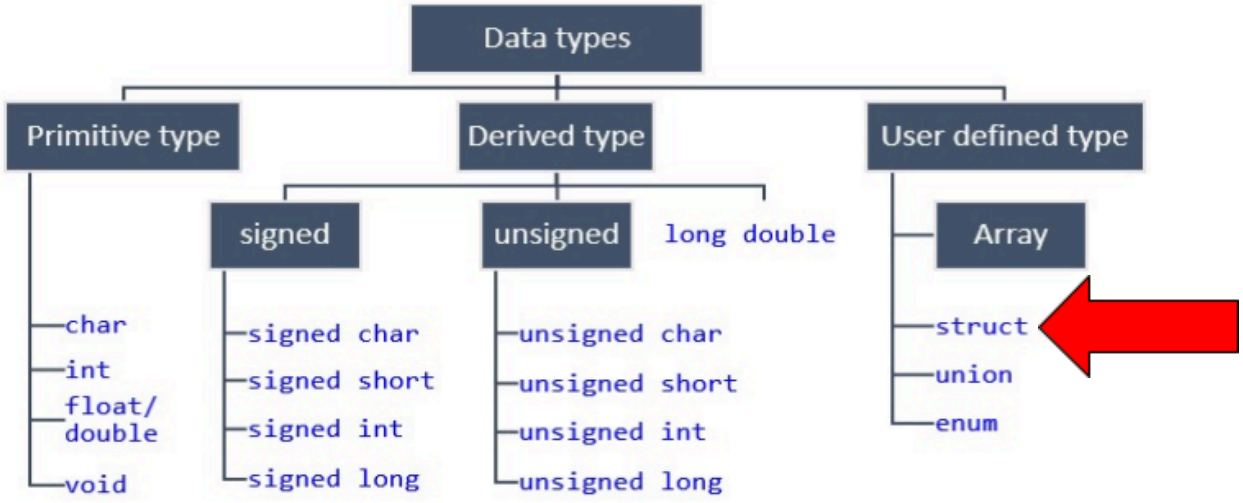
HEAP

E' uno spazio gestito dal programmatore che può far variare la dimensione durante l'esecuzione, tipicamente non viene gestita in modo sequenziale (a differenza dello stack);
L'aggiunta viene allocata nel primo spazio libero, la cancellazione può avvenire in qualsiasi momento;

STACK	HEAP
Memoria gestita **automaticamente**	Memoria gestita **manualmente**
Piccole dimensioni	**Grandi dimensioni**
L'accesso è **più veloce e facile** (cache friendly)	Può essere dispersa nella memoria, non cache friendly
Non flessibile	Flessibile
Accessi **veloci**, allocazione e deallocazione	Accessi **più lenti**, allocazione e deallocazione
Gli elementi sono **limitati ai loro threads**	Elementi accessibili in tutta l'applicazione
Il sistema operativo alloca lo spazio di memoria	Il sistema operativo è chiamato dal linguaggio in runtime

---

LE STRUTTURE (STRUCT)



Struct: permette di mettere insieme tipi e variabili di tipologia diversa;

```
struct NOME {
        Tipo1 Variabile1;
```

Tipo2 Variaibile 2;
…
TipoN VariaibileN;
};

struct punto{
float x;
float y;
};

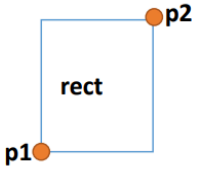
struct punto p1;
struct punto p2;

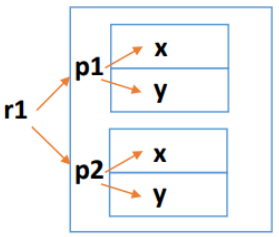
- Per inizializzare:
  struct punto p1 = {5, 7};

- Per accedere alle singole proprietà:
  nomevariabile.attributo
  p1.c oppure p1.y

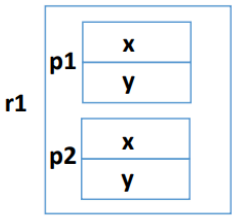
Posso **annidare le struct;**

- Possiamo anche definire struct che contengono ulteriori elementi e struct.
- Esempio: rettangolo






```
struct rect {
struct punto p1;
struct punto p2;
};
...
struct rect r1;
```



r1.p1.x = 17;

pippo = r1.p2.y;

---

typedef

Mi permette di definire o ridefinire nuovi tipi;

```c
typedef struct STUDENTE {
char cognome[20];
char nome[20];
int anno_nascita;
} Studente;

struct STUDENTE s1;
Studente s2,s3;

s1.anno_nascita = 1988;
s2.anno_nascita = 1975;
```

Passaggio di struct come parametri:
p->x (preferibile) == (*p).x

typedef double data;

data x; == double x;

COMPLESSITA'

Tempo: tempo richiesto dall'algoritmo
Spazio: memoria richiesta dall'algoritmo, numero di record di attivazione sullo stack;
Questi fattori possono essere influenzati da: tipologia di algoritmo (com'è scritto l'algoritmo), dimensione dell'input (più grande è più tempo ci i mpiego), velocità della macchina (ci vogliamo astrarre, non ci interessa analizzare la complessità prendendo in considerazione la macchina specifica);

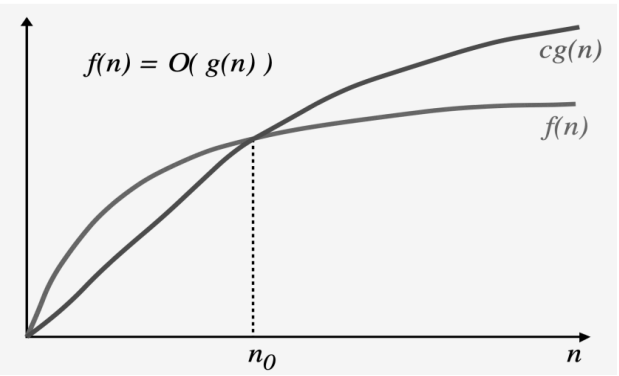
Complessità asintotica: input tendenti ad infinito;

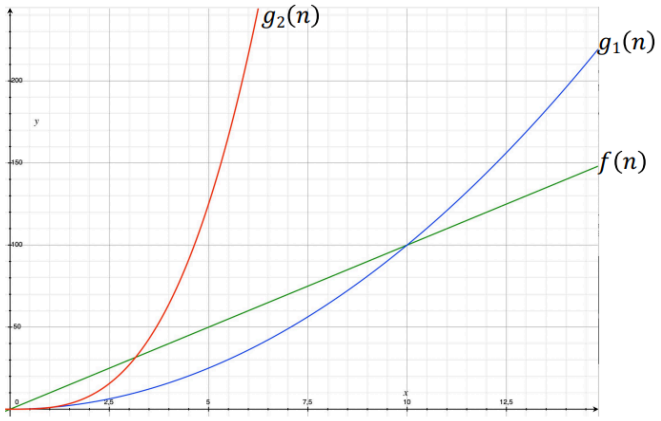
Complessità lineare: input 2 operazioni 2;
Complessità quadratica: input 2 operazioni 4;
Complessità esponenziale

Notazione O-grande

Comportamento **asintotico** delle funzioni matematiche;
Mette da parte costanti e variabili irrilevanti;
Permette di ottenere un limite superiore al comportamento asintotico;

Date due funzioni che dipendono dall'input, f(n) appartiene ad O-grande di g(n) se e solo se esiste un n0 e c>0 tale f(n)<cg(n) per ogni n>n0;
NON VALE SEMPRE [Ex. se f(n) per input piccoli può essere più grande di g(n)];
n→∞





g1(n) è più simile ad f(n) potremmo scegliere ancora meglio una costante per n (20n);

f(n) ha complessità asintotica g(n):

1) f(n) è O(g(n))
2) g(n) è la più piccola di tutte le funzioni che soddisfano la 1;

COME STABILIRE QUALE FUNZIONE E' LA PIU' PICCOLA
Date due funzioni f(n) e g(n);
f(n) è "più piccola" di g(n) se:

1) f(n) è O(g(n))
2) g(n) non è O(f(n))

Non esiste nessuna costante c per cui a partire da un n sufficientemente grande non potremo trovare g(n)<cf(n);

- 5 ha complessità asintotica... $O(1)$
- $2n + 5$ ha complessità asintotica... $O(n)$
- $3n^2 + 2n^2 + 2$ è ha complessità asintotica... $O(n^2)$
- $n^3 + 100n^2$ ha complessità asintotica... $O(n^3)$
- $3*2^n$ ha complessità asintotica... $O(2^n)$
- $3^n + 2^n + 8^n$ ha complessità asintotica... $O(8^n)$
- $3^n + n! + 8^n$ ha complessità asintotica... $O(n!)$

Funzioni iterative avranno sempre complessità in spazio O(1);

ANALISI DELLA COMPLESSITA' IN TEMPO

1) Individuare gli input e capire se questi possono influenzare la durata del programma;

Costo computazione programma = costo computazione delle singole istruzioni;
Costo(Istruzione 1) + Costo(Istruzione 2) + Costo(Istruzione N);

TIPOLOGIE DI ISTRUZIONI

- **Istruzioni Elementari**: Operazioni aritmetiche, Lettura/Scrittura di valori da/verso variabili, Condizioni logiche su un numero costante di operazioni e operatori, Operazioni di stampa o lettura da I/O …; **(COSTO COSTANTE O(1))**
- **Istruzioni Condizionali**: If, switch case …; **(COSTO = QUALE PESA DI PIU')**
- **Istruzioni iterative**: Cicli, While, Do While, For …; **(COSTO = IL NUMERO DI VOLTE IN CUI VERRA' ESEGUITO IN CICLO DIPENDE DALL'INPUT?, LA CONDIZIONE E' VERA O FALSA?, QUALI SONO E QUALE E' IL VALORE DELLE VARIABILI ALL'INGRESSO DEL CICLO?, COME VENGONO INCREMENTATE LE VARIABILI?)**

MI PONGO SEMPRE NEL CASO DEL COSTO COMPUTAZIONALE PEGGIORE

```
void function(int n, int *value){
int i;

for (i=0;i<n;i++) {
   *value += i*i;
   i = i +1;
   }
```
O(n)

```
i= 0;
```
O(1)

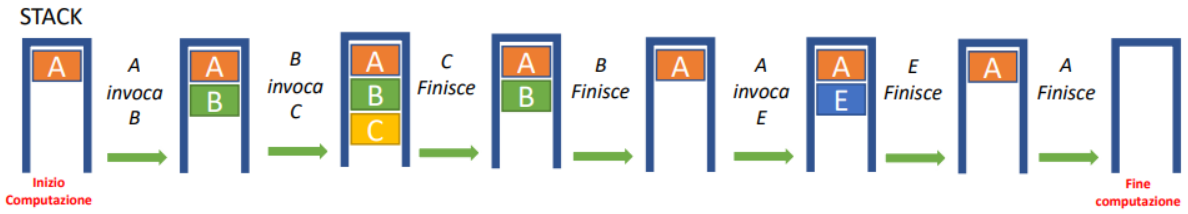
```
while (i <2*n) {
   printf("%d\n",*value);
   i+=2;
   }
```
O(n)

Nel primo for se al posto di n ci fosse stato un numero costante il costo sarebbe stato di O(1);
Se ho un ciclo annidato la complessità sarà =(n*m);

Costo complessivo = O(n); complessità asintotica lineare

COMPLESSITA' IN SPAZIO

La stima che facciamo ha a che fare con il **numero massimo di record di attivazione occuperà contemporaneamente**, non il numero di record che utilizzerà in totale;



Numero massimo di record di attivazione contemporaneamente = 3;
Vogliamo conoscere il numero di record di attivazione a livello asintotico e capire se dipende dall'input oppure no;
Se non dipende dall'input dal punto di vista asintotico sarà O(1);
Se il numero è 3 perché l'input era 3 e se fosse stato 4 sarebbero stati 4 record ecc. il valore dal punto di vista asintotico sarà O(n);
Per funzioni iterative O(1), se non vengono effettuate altre chiamate dipendenti dall'input il numero di record di attivazione sullo stack sarà costante;
Per funzioni ricorsive il valore dipenderà dal numero di chiamate ricorsive effettuate (dipendenti dall'input);

ALLOCAZIONE MEMORIA DINAMICA

Heap:

- Gestita dal programmatore;
- Gestita dinamicamente a run-time;
- Visibile globalmente da tutto il programma;
- Ad ogni richiesta si alloca quando richiesto nel primo blocco di memoria disponibile;
- Allocazione e deallocazione non sono fatte in maniera sequenziale.

malloc()

void * malloc(size_t size);

Libreria: #include <stdlib.h>
Scopo: Permette di allocare un blocco continuo di memoria di size bytes;
Input: la dimensione richiesta di memoria in bytes;
Output: un puntatore alla zona di memoria allocata **a void,** sta al programmatore attraverso un cast esplicito definire il tipo puntato;

sizeof()

sizeof (typename)

Scopo: Ottenere la dimensione di un tipo dati;
Motivazione: La dimensione delle variabili non è costante, può cambiare da macchina a macchina;
Input: un tipo (primitivo o user-defined);
Output: dimensione in bytes;

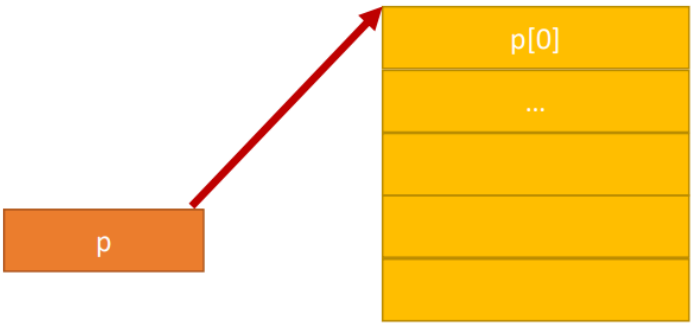
Uso combinato di malloc() e sizeof()

1. Dichiariamo un puntatore a float;
2. Richiediamo tramite la funzione malloc uno spazio di memoria
3. Per il calcolo di quanto spazio occorre utilizziamo sizeof
4. Effettuiamo un typecasting a (float *) e assegnamo l'indirizzo della zona allocata a p

```
float *p;

p = ( float * ) malloc(sizeof (float));

p = ( float * ) malloc(sizeof (float)* 5);
```



Sovrascrive la variabile p, la zona occupata non è più accessibile perché non abbiamo più il riferimento della zona; MEMORY LEAK

free()

Deallocare la memoria

void free (void *ptr);

Libreria: #include <stdlib.h>
Scopo: Deallocare un blocco di memoria precedentemente allocato;
Input: il puntatore allo spazio di memoria;
Output: nessuno;
La zona di memoria viene resa nuovamente disponibile (ma non necessariamente ripulita);
Il puntatore non punterà più ad una zona di memoria significativa;

calloc()

void * calloc (size_t count, size_t size);

Libreria: #include <stdlib.h>
Scopo: Allocare uno spazio contiguo per contente count oggetti di dimensione size bytes;
Input: numero di oggetti (count) di dimensione (size) di un oggetto;
Output: un puntatore alla zona di memoria allocata;
La memoria allocata viene riempita con byte di valore zero;

Azzerare la memoria: **memset**, **bzero**;

realloc()

void *realloc(void *ptr, size_t size);

Libreria: #include <stdlib.h>
Scopo: modificare la dimensione di un blocco di memoria precedentemente allocato;
Input: puntatore alla zona da ridimensionare, nuova dimensione;
Output: puntatore alla nuova zona di memoria allocata;
Se c'è spazio disponibile, la zona di memoria viene ridimensionata in loco (senza spostamenti) → il valore di ritorno coincide col valore del puntatore della zona da ridimensionare.
Se non c'è abbastanza spazio per ridimensionare, si cerca nella heap una nuova zona abbastanza grande, e il contenuto della vecchia zona viene copiato nella nuova.

Per allocare una matrice:
int **a → malloc(sizeof(int*)*m); poi fare una serie di for con allocazioni a vettori di dimensione n;
**a puntatore a zona di memoria che contiene altre zone di memoria;
Per deallocare devo prima deallocare i vettori e poi posso fare la free di a;

LISTE

Sequenza ordinata di elementi (nodi), contengono qualche tipo di informazione (struct, int, char ecc…), contiene il riferimento al nodo successivo (non a quello precedente);
Approccio induttivo:

1. (passo base) Una lista vuota è una lista;
2. (passo induttivo) Un nodo seguito da una lista è una lista.

Non sappiamo a priori quanti nodi ha la lista;
Per sapere quanti nodi ha bisogna scorrere la lista e contare il numero di nodi;

Le liste in C

Non esistono le liste in C;
Dobbiamo implementare noi;
LISTA DINAMICA, può crescere e decrescere in base alle necessità;
Useremo:

- struct (definire la forma del nodo);
- puntatori (mettere in evidenza la connessione con il nodo successivo).

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

typedef int DATA;

struct linked_list {
DATA d;
struct linked_list *next;
};

typedef struct linked_list ELEMENT;

typedef ELEMENT * LINK;
```

LINK = ELEMENT * = STRUCT LINKED_LIST *
Indirizzo al primo nodo, se è NULL la lista è vuota;
Quando il puntatore punta a NULL vuol dire che la lista è finita;

```c
LINK newnode(void) {
return malloc(sizeof(ELEMENT));
/* includere <stdlib.h> */
}

int main() {
LINK a;
a = newnode();
free(a);
return 0;
}
```

newnode richiede lo spazio di memoria sufficiente a contenere un nodo;
Sempre mettere la free alla fine (considerato come errore se manca)

Visita di una lista

- **Visita incondizionata**

```c
void printlis(LINK lis) {
   while (lis != NULL) {
      printf("%d\n", lis->d);
      lis= lis->next;
   }
}
```

- Ottengo una copia al puntatore alla testa della lista;
- Complessità in tempo = O(n), n numero di nodi della lista (il ciclo while dipende dall'input ed avanza di uno ad ogni ciclo);

- Complessità in spazio = O(1) (un record di attivazione della funzione).
- **SCRIVERE SEMPRE LE SPIEGAZIONI PER LE COMPLESSITA' E COSA E' N**

➢ **Visita con condizione**

```
void print_greater(LINK lis, int k) {
   while (lis != NULL) {
        if(lis->d > k ) {
           printf("%d\n", lis->d);
        }
      lis= lis->next;
   }
}
```

- Le operazioni vengono eseguite solo se si verificano determinate condizioni;
- Complessità in tempo = O(n), n numero di nodi della lista (k non influenza il numero di volte in cui verrà eseguito il ciclo while);
- Complessità in spazio = O(1), (un record di attivazione).

➢ **Visita condizionata da contatore/accumulatore**

```
void printpos(LINK lis, int x) {
   int pos=1;
   while (lis != NULL) {
      if ((pos % x)==0) {
         printf(">>>> %d\n", lis->d);
      }
      pos++;
      lis= lis->next;
   }
}
```

- La posizione non la conosciamo a priori;
- Il primo nodo sarà in posizione 1;
- Complessità in tempo = O(n), n numero di nodi della lista (x non influenza il numero di volte in cui verrà eseguito il ciclo while);
- Complessità in spazio = O(1), (un record di attivazione).

➢ **Visita di una parte di una lista**

- La parte da visitare dipende da un valore da aggiornare ad ogni passo della visita;

```
/* Esempio: stampa i primi n numeri della lista (se
ci sono) */
void print_k(LINK lis, int k) {
    int pos=1;
    while ((lis != NULL) && (pos <= k)) {
        printf("%d\n", lis->d);
        lis=lis->next;
        pos++;
    }
}
```

- Complessità in tempo = **O(min(n,k))**, n numero di nodi della lista, se il numero di nodi è < di k O(n), se il numero di nodi è > k la complessità sarà O(k);
- Complessità in spazio = O(1)

ALGORITMI CON FINESTRA

➢ **Visita con finestra**

- Analizzare elementi a gruppi;
- Con finestra si intende un intervallo;

```
int elementi_minori(LINK lis) {
   int cnt=0;
   if(lis == NULL) return 0;
   if(lis->next  == NULL) return 0;
   while (lis->next != NULL) {
       if(lis->d < lis->next->d ) cnt++;
    lis= lis->next;
   }
   return cnt;
}
```

COMPLESSITA'?

Soluzione non ottimale

```
int elementi_minori(LINK lis) {
   int cnt=0;
   while (lis! = NULL && lis->next != NULL) {
      if(lis->d < lis->next->d ) cnt++;
   lis= lis->next;
   }
   return cnt;
}
```

- Problemi: stiamo presupponendo che lis != NULL venga valutato prima di lis->next != NULL potrebbe succedere che prima valutiamo il secondo poi il primo, perché non sappiamo a prescindere come lavora il c; anche se l'ordine viene rispettato **ad ogni iterazione del while andremo a fare due valutazioni;**

RICERCA DI UN ELEMENTO

Casi: ricerca per valore, ricerca per posizione;

➢ **Ricerca per valore**
- Restituisce un puntatore al nodo con valore x nella lista;

➢ **Ricerca per posizione**
- Visita con contatore, contare il numero dei nodi, se esiste la posizione k restituisco il riferimento

➢ **Nodo precedente a nodo di valore x**
- Se lista vuota:
  - → restituisce NULL

- Se nodo contenente x in testa alla lista:
  - → restituisce NULL
- Altrimenti:
  - → restituisci predecessore del nodo

Complessità in termini di tempo: O(n), n numero di nodi della lista;
Complessità in termini di spazio: O(1).

MODIFICA DI UNA LISTA

- **Modifica del valore di tutti gli elementi o sottoinsieme di questi**
  - Visita incondizionata;
- **Aggiunta/rimozione di nodi**
  - **Inserimento in testa**
    - **ORDINE OBBLIGATORIO**
    - CREO NODO
    - COLLEGO AL PRIMO NODO
    - COLLEGO IL PRIMO RIFERIMENTO DELLA LISTA AL NUOVO NODO
    - ***NON DIMENTICARE IL NULL FINALE***
    - Complessità in tempo e spazio O(1).
  - **Inserimento in coda**
    - Complessità in tempo O(n);
    - Complessità in spazio O(1).

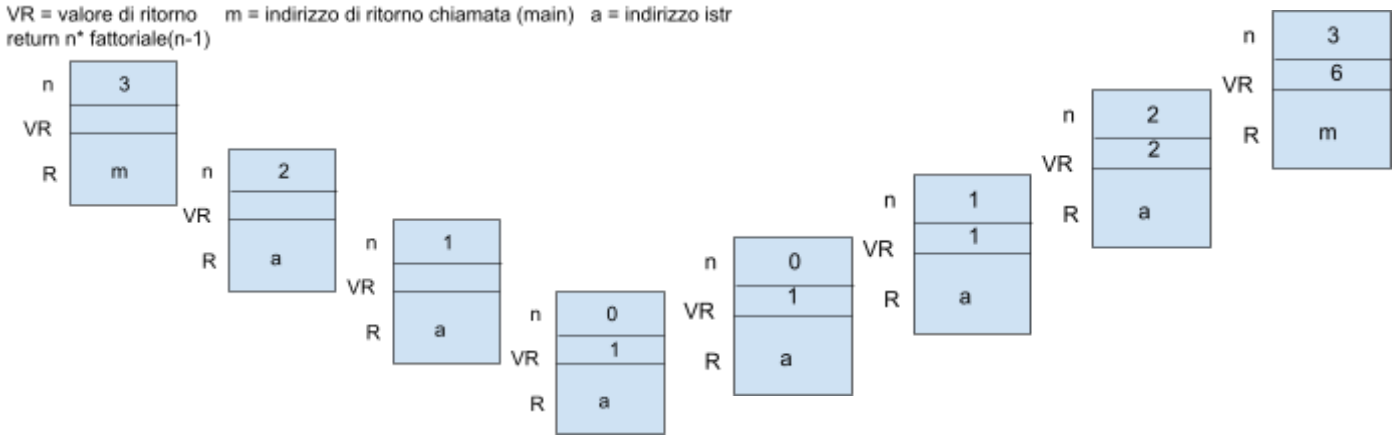
- **Creazione di una lista**
  -
- **Cancellazione di una lista**
  - **Cancellazione in testa**
    - Si può iterare per cancellare tutti i nodi della lista
      - Salvare il riferimento al primo nodo
      - Modificare la testa della lista e punto al secondo elemento
      - Liberare lo spazio di memoria (free)
    - Complessità tempo e spazio = O(1);
  - **Cancellazione in coda**
    - Se la lista ha un solo nodo la cancellazione in coda e in testa sono uguali; PER QUESTO CASO DEVO PASSARE LA LISTA PER RIFERIMENTO
    - Complessità in spazio = O(1);
    - Complessità in tempo = O(n).
  - **Cancellazione di tutti i nodi**
    - Cancellazione in testa ripetuta
    - Complessità in spazio = O(1);
    - Complessità in tempo = O(n).
    -
    - Cancellazione di tutti i nodi in coda
    - Complessità in tempo $O(n^2)$
    - [(n+1)*n]/2
  - **Cancellazione di un elemento con un determinato valore**

        - Dobbiamo sempre conoscere il nodo precedente
        - Complessità in spazio = O(1);
        - Complessità in tempo = O(n), dovuta dalla funzione findpred.
- **Operazioni su più liste**
    - **Duplicazione di una lista**
        - Complessità in spazio = O(1);
        - Complessità in tempo = O(n).
    - **Duplicazione nodi soddisfacenti certe condizioni**
        - Complessità in spazio = O(1);
        - Complessità in tempo = O(n).
    - **Visita di 2 o più liste**
        - Complessità in spazio = O(1);
        - Complessità in tempo = O(min(m,n)), lunghezza della lista più corta.
    - **Confronto in parallelo di 2 liste**
        - Complessità in spazio = O(1);
        - Complessità in tempo = O(max(m,n)), lunghezza della lista più lunga.

RICORSIONE

- **Caso base** = caso per cui si ferma la ricorsione;

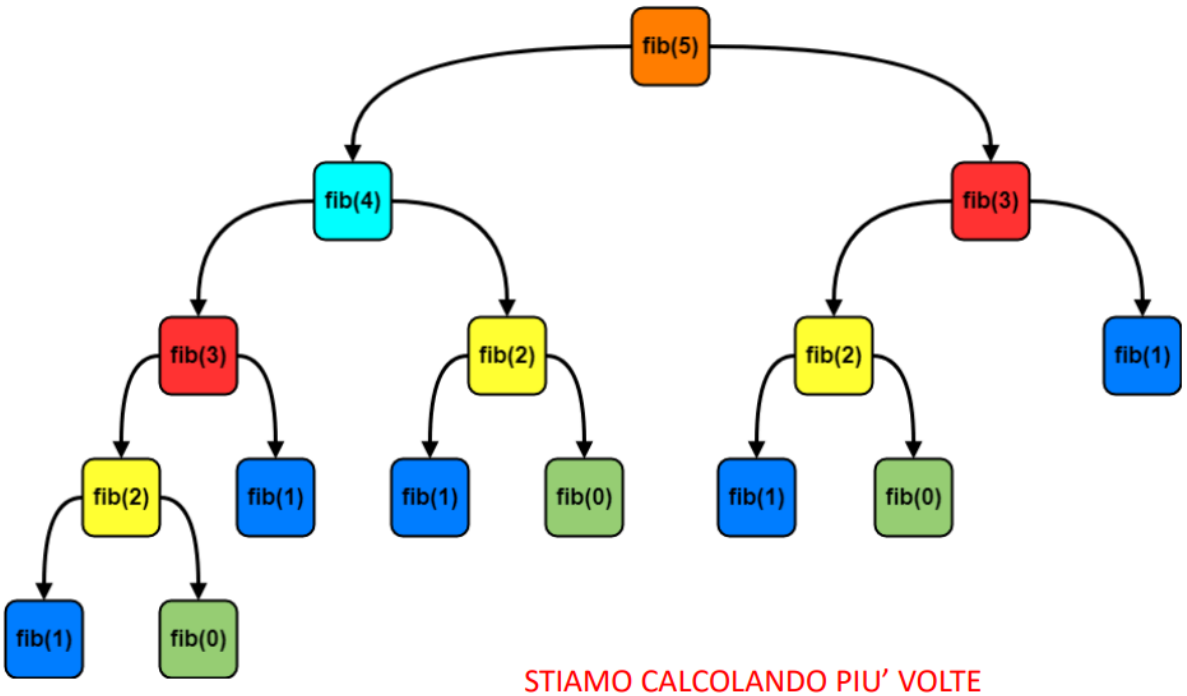
FATTORIALE



Complessità in tempo/spazio = O(n);

La complessità in tempo si calcola n chiamate di costo costante, contare il numero massimo di record di attivazione (non sempre coincide).

FIBONACCI



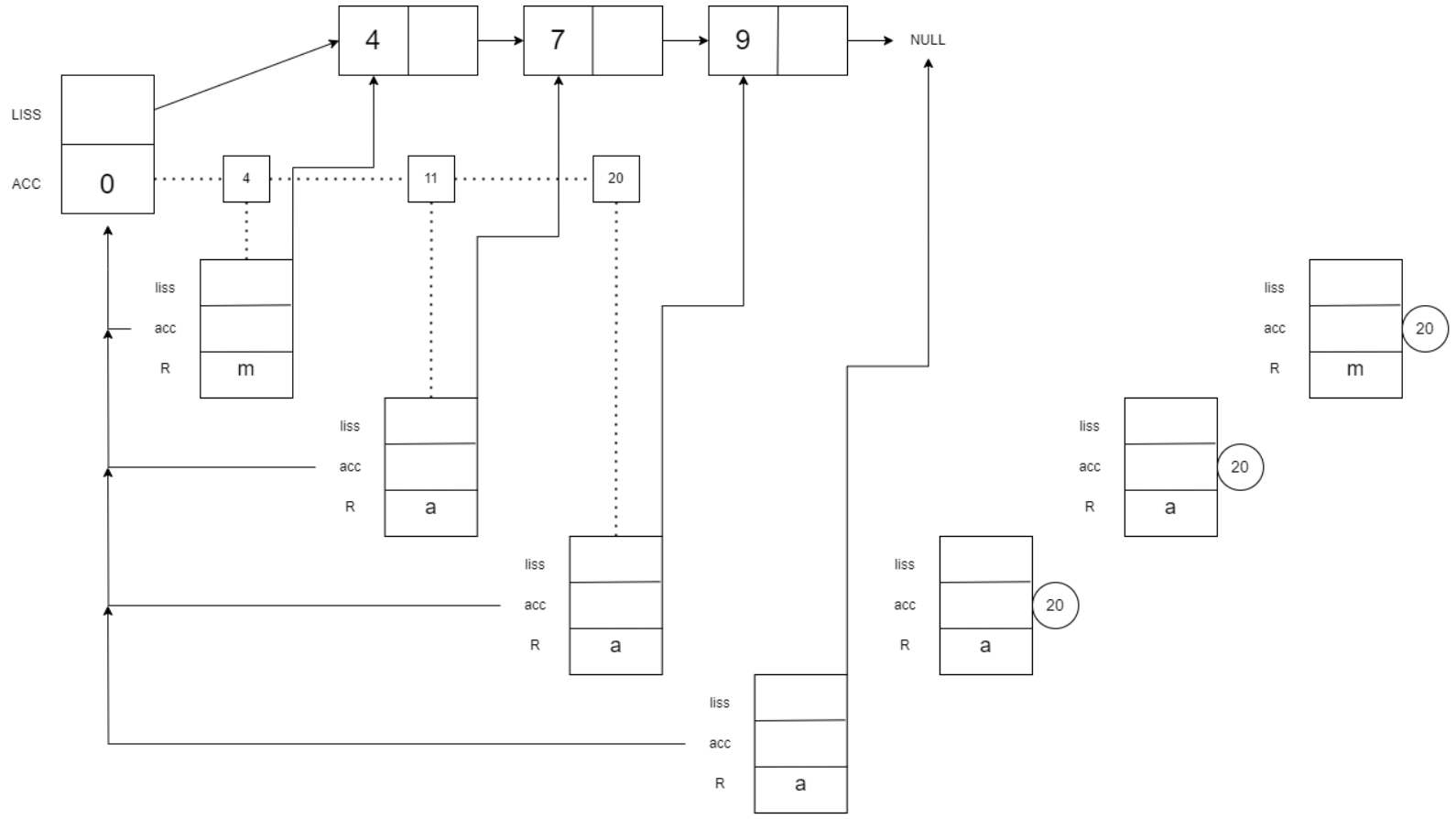
IMPORTANTE: Se dobbiamo passare dei valori condivisi nelle varie chiamate ricorsive non devo usare variabili locali ma dei puntatori a delle aree di memoria condivise.

TORRE DI HANOI
Albero delle chiamate: scendo a sinistra scambio secondo e terzo, scendo a destra scambio primo e secondo.

TIPI DI RICORSIONE

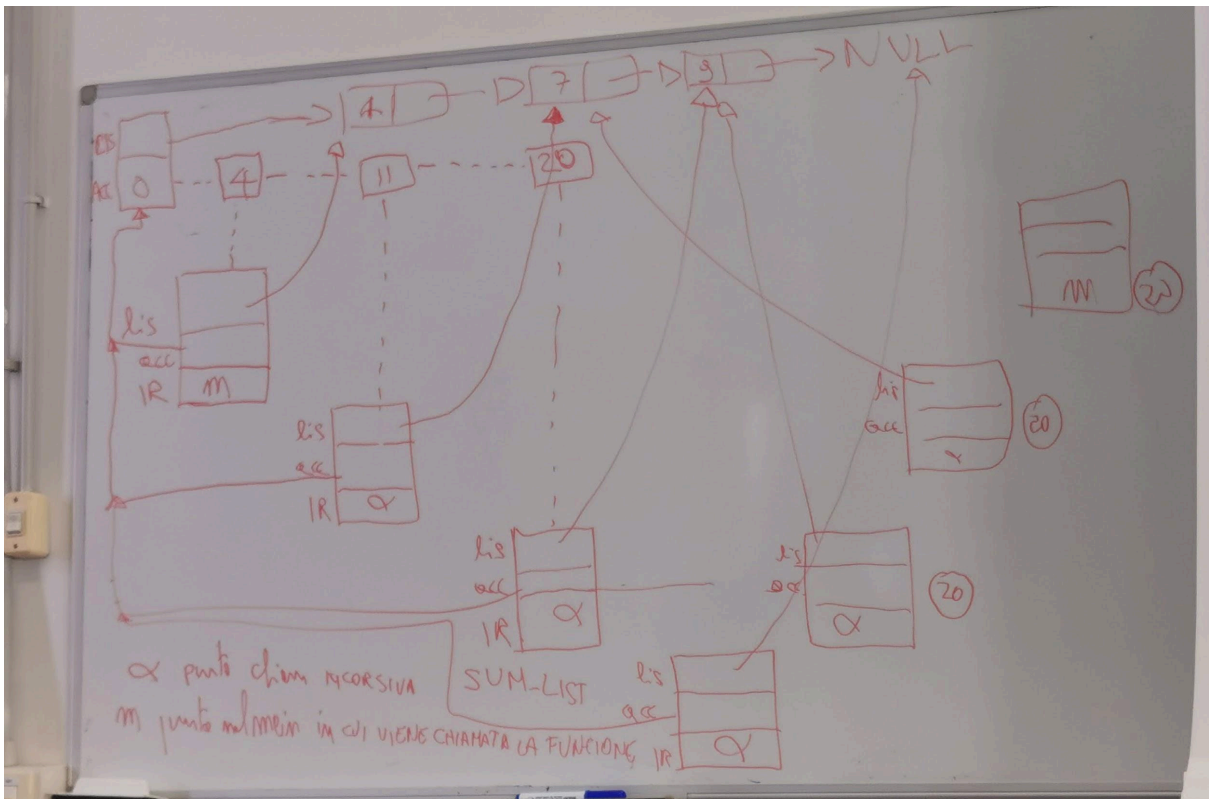
- **Ricorsione diretta**
  - Ricorsione lineare:
    - Ho solamente un ramo che si espande (il numero massimo di record di attivazione che ho sullo stack è uguale al numero totale di record che avrò sullo stack).
    - FATTORIALE
  - Ricorsione non lineare:
    - Avrò dei rami distinti che si sviluppano. Il numero totale di record di attivazione **NON** coincide con il numero massimo di record di attivazione presenti sullo stack).
    - FIBONACCI, TORRE DI HANOI
  - Ricorsione di coda
    - Quando l'ultima operazione eseguita è la ricorsione, non somme ecc.
    - Dalla ricorsione di coda posso passare ad un algoritmo iterativo.
- **Ricorsione indiretta**

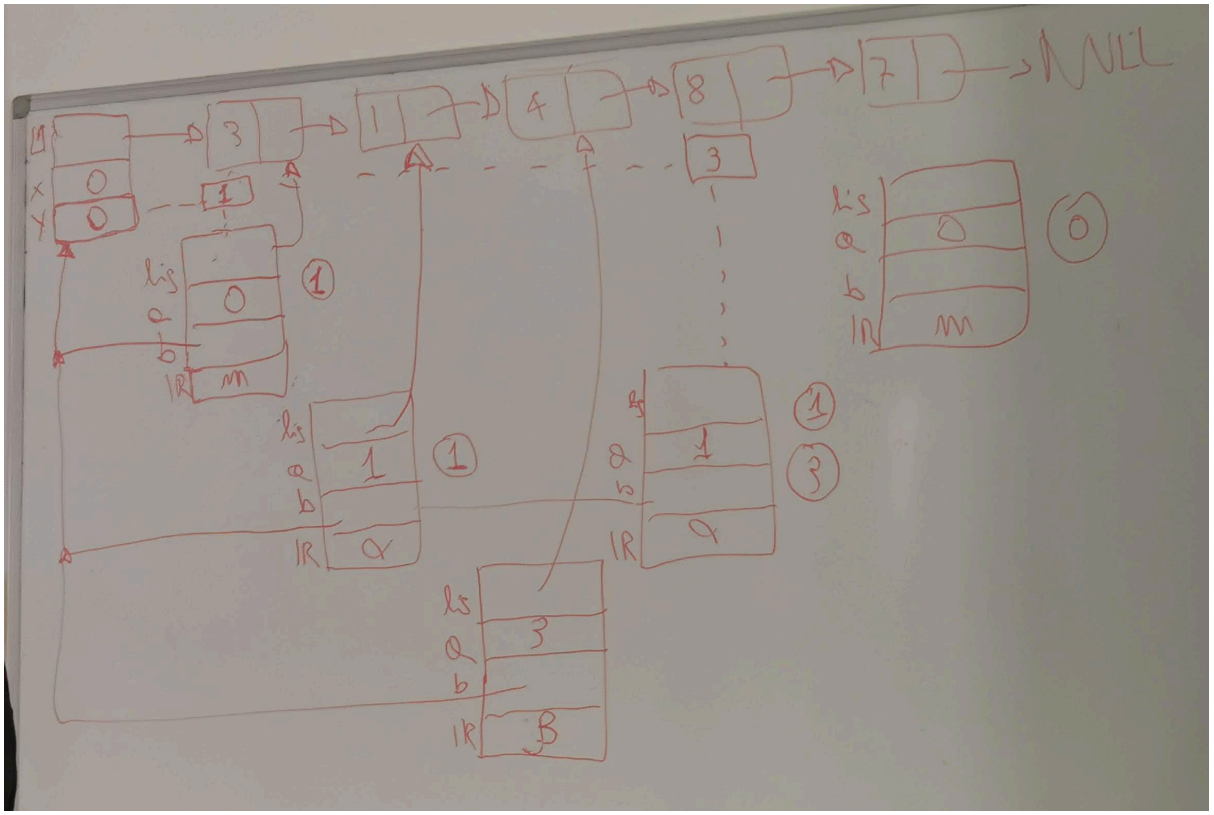
RICORSIONE SU LISTE

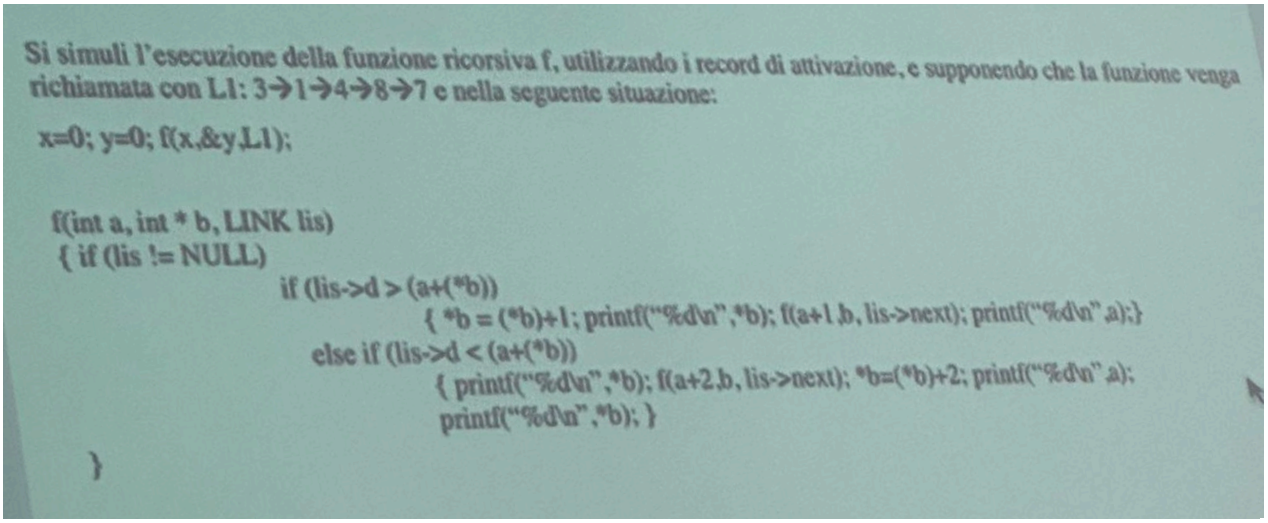


a punto chiamata ricorsiva
m punto nel main in cui viene chiamata la funzione

- **Visita**


4
7
9
NULL
0
4
11
20
lis
acc
IR
m
20
α
SUM_LIST
α punto chiamata RICORSIVA
m punto nel main in cui viene chiamata la funzione


3
1
4
8
7
NULL
0
1
3
lis
a
b
IR
m
1
α
β
0


Si simuli l'esecuzione della funzione ricorsiva f, utilizzando i record di attivazione, e supponendo che la funzione venga richiamata con L1: 3→1→4→8→7 e nella seguente situazione:

```
x=0; y=0; f(x,&y,L1);

f(int a, int * b, LINK lis)
{ if (lis != NULL)
        if (lis->d > (a+(*b))
                { *b = (*b)+1; printf("%d\n",*b); f(a+1,b, lis->next); printf("%d\n",a);}
        else if (lis->d < (a+(*b))
                { printf("%d\n",*b); f(a+2,b, lis->next); *b=(*b)+2; printf("%d\n",a);
                printf("%d\n",*b); }
}
```

Torre di Hanoi

Complessità in tempo: O(2^n)
Complessità in spazio: O(n)

Bubble sort = Insertion sort

Complessità in tempo: **O(n^2)** = [(n+1)*n]/2
Complessità in spazio: O(1)

k=r-p+1;

Mergesort

Complessità in tempo: O($n * log_2 n$) Ogni livello * k elementi scansionati nella merge.

Complessità in spazio: O($log_2 n$)

Quicksort

Algoritmo instabile, dipende dalla dimensione e da come è fatto l'input
Dobbiamo studiare la complessità nel caso migliore e nel caso peggiore
MIGLIORE:
Complessità in tempo: O($n * log_2 n$) Ogni livello * k elementi scansionati nella partition.

Complessità in spazio: O($log_2 n$)

PEGGIORE:

Complessità in tempo: O(n^2)
Complessità in spazio: O(n)