

la ricorsione

introduzione alla programmazione

il principio della ricorsione

- la ricorsione è un principio molto potente
- definizione *naïf*: una funzione ricorsiva è definita in termini di se stessa
- una definizione più rigorosa si lega al concetto di *induzione aritmetica*

esempio

- dato un problema P che deve operare su un input I di cardinali n
- ammettiamo di saper risolvere P “in modo diretto “ per n piccoli
- ammettiamo di saper dividere P in sottoparti tali per cui, se per ognuna troviamo una soluzione, allora siamo in grado di ottenere una soluzione globale per P

esempio

- allora
- *funzione* $AR(I)$ // n cardinalità
if n è sufficientemente piccolo
 risolvi P direttamente
else
 suddividi I
 risolvi ogni sottoparte attraverso la chiamata
 $AR(\text{sottoparte}(I))$
 ricombina le soluzioni
end
 chiamata ricorsiva
- **idea:** AR chiamata di volta in volta su insiemi più piccoli;
alla fine si entrerà nella parte che risolvo direttamente

osservazioni

- la ricorsione è elegante
- la ricorsione non è strettamente necessaria
*esiste sempre un'implementazione iterativa, anche se
essa può essere lunga e complessa*
- la ricorsione può risultare computazionalmente pesante

induzione aritmetica

- base - definizione/soluzione diretta
- passo induttivo - assunta nota la definizione/soluzione su un numero n , la si dimostra/definisce su un numero più grande di n

induzione aritmetica a passi

- Se valgono le seguenti clausole:
 - base - la proprietà P è vera sul numero c
 - passo induttivo - per ogni $n > c$: se P è vera per n , allora P è vera per $n+1$
- allora P è vera per ogni intero $k \geq c$

induzione aritmetica a passi - dimostrazioni

- se voglio dimostrare (tesi) che tutti gli interi $k \geq c$ hanno una proprietà P
- Allora
 - dimostro la base (in modo diretto): c ha la proprietà P
 - passo induttivo: assumo che n abbia la proprietà P e verifico che anche $n+1$ ce l'abbia

induzione aritmetica a passi - definizioni

- base: definisco la base
- passo: definisco la proprietà su $(n+1)$ in base a n

esempio - il fattoriale

- definizione:
 - $n! = n \cdot (n-1) \cdot (n-2) \dots 2 \cdot 1$
 - $0! = 1$
- base $0! = 1$
- passo per ogni $n > 0$, $n! =_{\text{def}} n(n-1)!$

esempio - il coefficiente binomiale

$$\binom{n}{k} \stackrel{def}{=} \frac{n!}{k!(n-k)!}$$

- il fattoriale può diventare facilmente molto grande (ed eccedere le dimensioni massime del tipo di dato scelto)
- questo anche nel caso in cui il coefficiente binomiale finale sia molto piccolo eg $\binom{n}{n} = 1$

esempio - il coefficiente binomiale

- base $\binom{n}{n} =_{def} \binom{n}{0} =_{def} 1$
- passo induttivo $\binom{n}{k} =_{def} \binom{n-1}{k-1} + \binom{n-1}{k}$

induzione e ricorsione

- una funzione ricorsiva dipenderà da un parametro n (numero naturale)
- la definizione della funzione seguirà il principio di induzione
 - definisco la base in modo diretto
 - passo induttivo: definisco la funzione con parametro n in termini della funzione stessa (con parametro $n-1$)

funzioni ricorsive - fattoriale

- ```
int fattoriale_iterativo (int n)
{
 if (n<0) throw ERROR;
 int aux=1;
 for (int i=1;i<=n;i++)
 aux=aux*i;
 return aux;
}
```
  - ```
int fattoriale_ricorsivo (int n)
{
    if (n<0) throw ERROR;
    if (n==0) return 1;
    else
        return n*fattoriale_ricorsivo(n-1);
}
```
- ricorsione
in coda

la ricorsione in coda è facile da “srotolare”:
$$\text{fr}(3)=3*\text{fr}(2)=3*2*\text{fr}(1)=3*2*1*\text{fr}(0)=3*2*1*1=6$$

Esempio - potenze

- Altro esempio di ritorsione in coda n^m

```
int recursive_power (int n, unsigned int m) {  
    if (m==0) return 1;  
    if(m==1) return n;  
  
    return n*recursive_power(n,m-1);  
}
```

funzioni ricorsive - coefficienti binomiali

- ```
int cbin (int n, int k)
 return(fattoriale(n)/(fattoriale(k)-fattoriale(n-k)));
```
- ```
int cbin_ricorsiva(int n, int k)
{
    if(k<0 || n<k) throw ERROR;
    if(k==0 || n==k) return 1;
    return(cbin_ricorsiva(n-1,k-1)+cbin_ricorsiva(n-1,k));
}
```

Ricorsione e sequenze

- la ricorsione si applica in modo molto naturale alle sequenze
- Tutte le operazioni che richiedono una visita di una sequenza (stampa, ricerca, calcolo di min/max, ...) possono essere progettate tramite ricorsione
- NB: ricorsione è alternativa ai cicli

Un esempio

```
void recursive_print_rev(int A[], int size ) {  
    if (size<0) throw ERROR;  
    if (size==0) {  
        cout << endl;  
        return;  
    }  
    cout<< A[size-1];    //NOTA INTERESSANTE (ERRORE TIPICO) VEDO  
    COSA SUCCEDDE SE METTO LA STAMPA DOPO LA CHIAMATA  
    recursive_print_rev(A,size-1) ;  
}
```

Un altro esempio

```
int recursive_sum( int A[], int size) {  
    if (size<=0) throw ERROR;  
    if (size==1) return A[size-1];  
    return A[size-1]+recursive_sum(A,size-1);  
}
```

- In che modo calcolare il prodotto di elementi di una sequenza?
- ... il minimo, il massimo, ...?
- (piu' difficile) in che modo verificare se una sequenza è palindrom

ricorsione e liste

- Analogamente la ricorsione si applica in modo molto naturale alle liste
- Nel caso di liste collegate semplici un modo naturale per procedere e' quello di realizzare *ricorsione in coda*:
 - *Il primo elemento viene separato da tutto il resto (la coda)*

ricorsione e liste

- Analogamente la ricorsione si applica in modo molto naturale alle liste
- esempio: ricerca per scansione sequenziale.
Riformuliamo in modo da mettere in risalto la ricorsione
 - se la lista è vuota non contiene elem -> false
 - se elem è uguale al primo elemento della lista -> true
 - cerco ricorsivamente nei successivi $(n-1)$ elementi

Liste collegate semplici

ASSUMIAMO:

```
typedef int Elem;
```

```
typedef struct cell {  
    Elem head;  
    cell * next;  
} *lista;
```

```
bool is_empty_l(const lista l)  
{  
    return (l==nullptr);  
}
```

```
void create_empty(lista &l)  
{  
    l=nullptr;  
}
```

ricorsione e liste

- ```
bool is_in(const list& l, Elem x)
{
 if (l==nullptr) return false; //base
 if (l->head==x) return true; //base
 return is_in(l->next, x);
}
```
- ```
int length(const list& l)
{
    if (is_empty_l(l)) return 0; // base
    else
        return length(l->next) + 1;
    //provate a "srotolare" le chiamate ricorsive innestate
```

Inserimento ordinato

```
void insert_elem(Elem x, lista &l)
{
    if ((l == nullptr) || (l->head > x))
    {
        // inserisci in testa
        cell * aux = new cell;
        aux->head = x; aux->next = l; l = aux;
    }
    if (l->head == x) return; // elemento gia' presente
    else
        insert_elem(x, l->next); // inserisci nel resto
}
```

Potremmo eliminare la riga in rosso per inserimenti con ripetizione

Inserimento ordinato

```
void insert_elem(Elem x, lista &l)
{
    if ((l == nullptr) || (l->head > x))
    {
        // inserisci in testa
        cell * aux = new cell;
        aux->head = x; aux->next = l; l = aux;
    }
    if (l->head == x) return; // elemento gia' presente
    else
        insert_elem(x, l->next); // inserisci nel resto
}
```

- Con il passaggio della lista per riferimento i legami aggiornati tra celle vengono mantenuti invariati da una chiamata all'altra
- Notate che in questo caso realizziamo sempre un inserimento in testa (alla sottolista contenente solo elementi maggiori di quello che vogliamo inserire)

Cancellazione di un elemento in lista ordinata (notate la somiglianza con l'inserimento)

```
bool delete_elem(Elem x, lista &l)
{
    if (l == nullptr)    return false;
    if (l->head == x)
    {
        cell *tmp = l;
        l = tmp->next;
        delete tmp;
        return true;
    }
    else
        return delete_elem(x, l->next);
}
```

reverse ricorsiva di una lista


qui le cose si complicano...

- se vogliamo ribaltare il contenuto di una lista data in input e abbiamo a disposizione una diversa lista in output l'operazione è semplice
- E' sufficiente inserire in testa alla lista di output tutti gli elementi che incontriamo nella visita della lista di input (possiamo procedere in modo iterativo o ricorsivo, provate!)
- **Se invece vogliamo riutilizzare la stessa lista di input un approccio ricorsivo è più semplice**

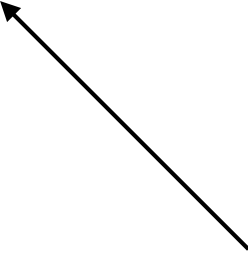
reverse ricorsiva di una lista

```
void reverse(list &l)
{
    list l1;
    if ((l == nullptr) || (l->next == nullptr)) return;
    l1 = l->next;
    reverse(l1); // capovolgi il resto
    l->next->next = l;
    l->next = nullptr;
    l = l1;
}
```

Caso base:
se la lista contiene zero o
un elemento resta invariata

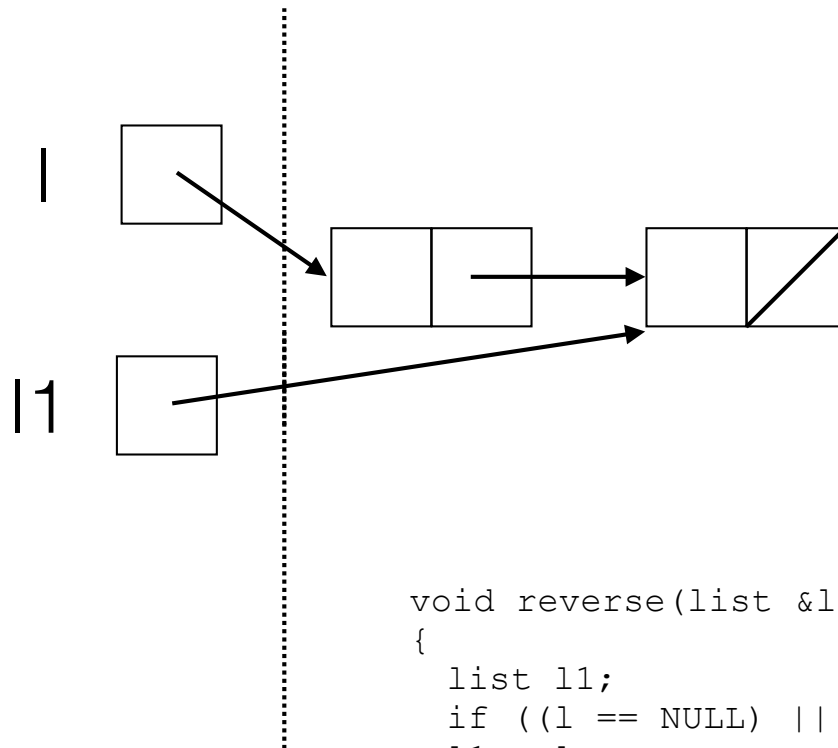


Idea per la ricorsione:
Salto il primo elemento, ribalto il resto e
poi in fondo attacco il primo elemento...



reverse ricorsiva di una lista

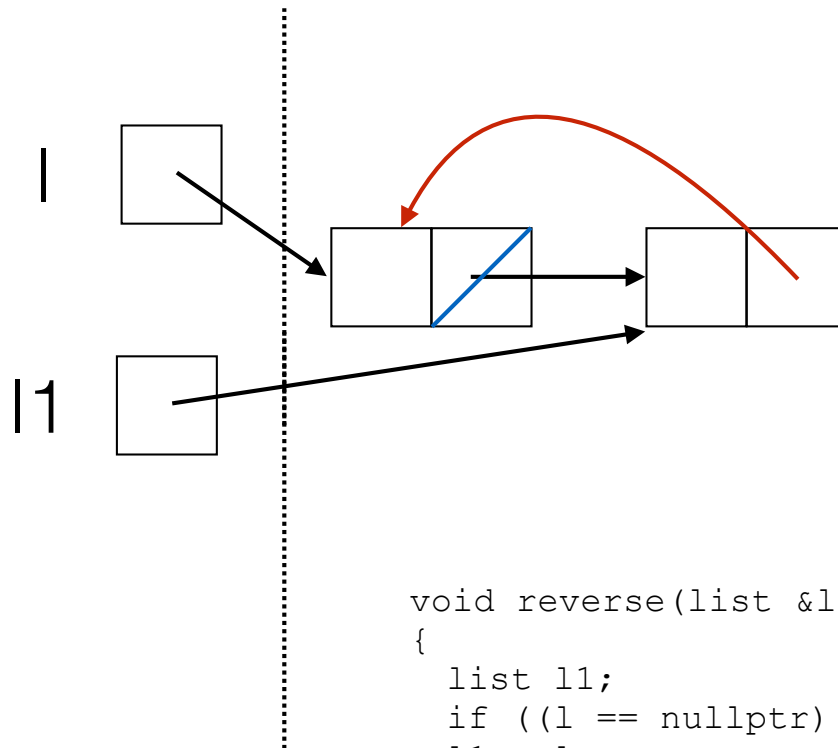
caso semplice: un solo passo di ricorsione (lista con 2 elementi)



```
void reverse(list &l)
{
    list l1;
    if ((l == NULL) || (l->next == NULL)) return;
    l1 = l->next;
    reverse(l1); // capovolgi il resto
    l->next->next = l;
    l->next = nullptr;
    l = l1;
}
```

reverse ricorsiva di una lista

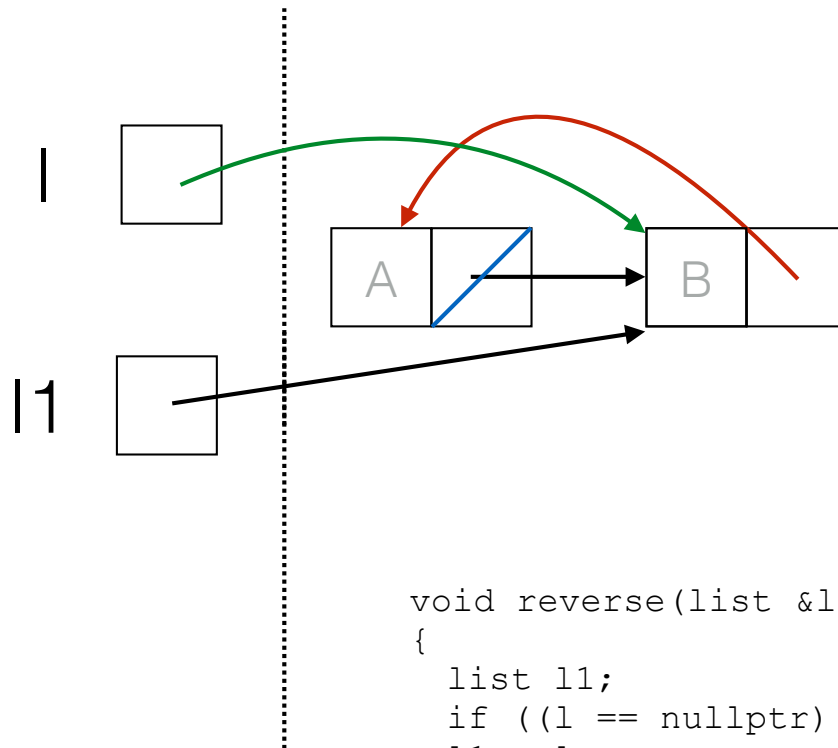
caso semplice: un solo passo di ricorsione (lista con 2 elementi)



```
void reverse(list &l)
{
    list l1;
    if ((l == nullptr) || (l->next == nullptr)) return;
    l1 = l->next;
    reverse(l1); // capovolgi il resto
    l->next->next = l;
    l->next = nullptr;
    l = l1;
}
```

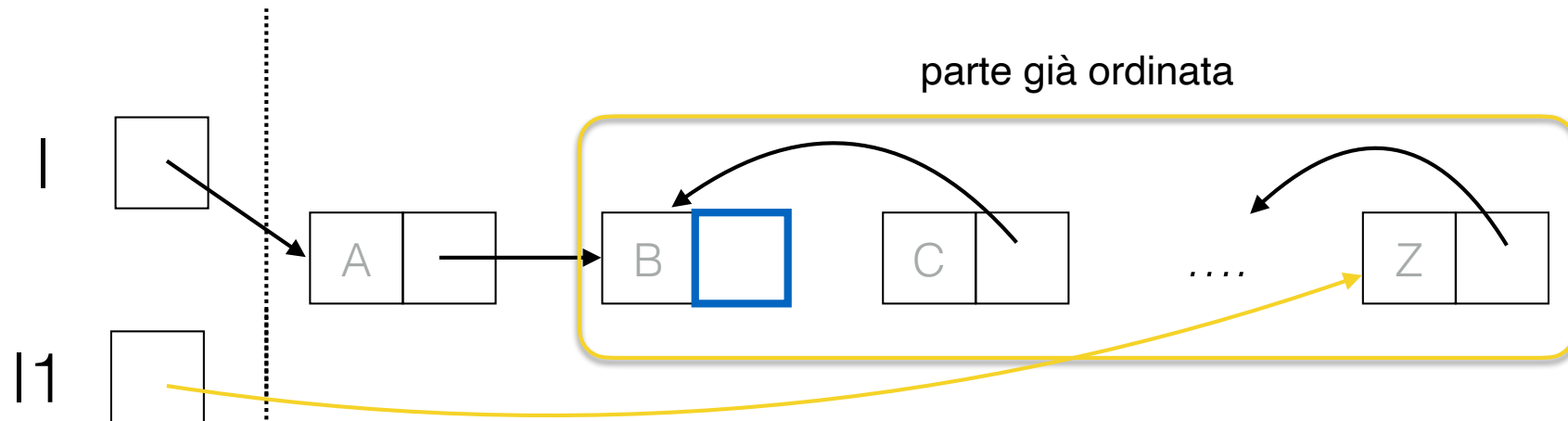
reverse ricorsiva di una lista

caso semplice: un solo passo di ricorsione (lista con 2 elementi)



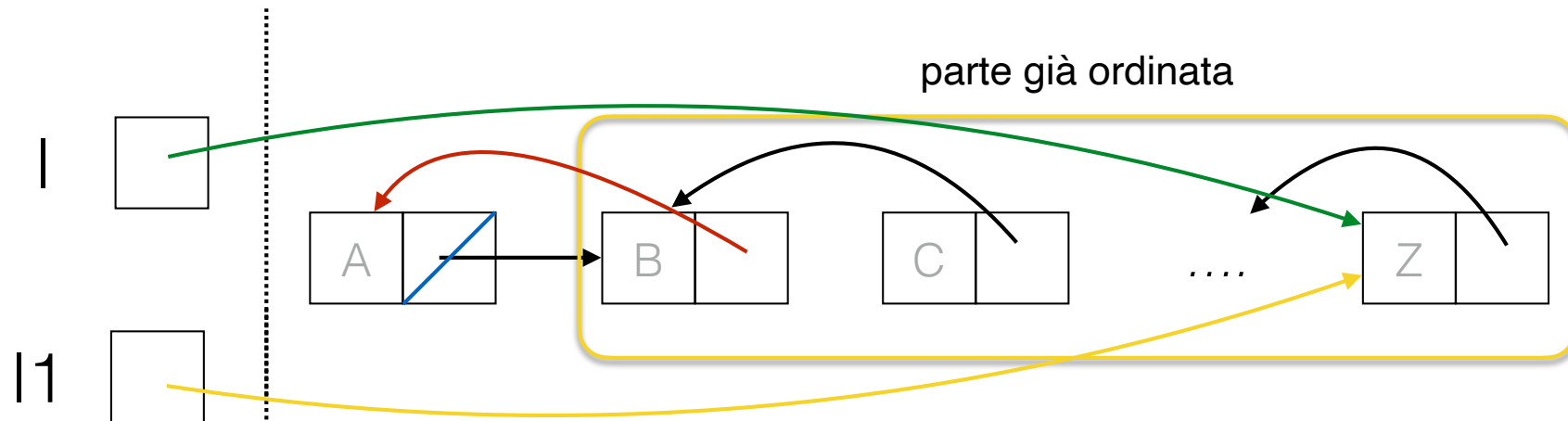
```
void reverse(list &l)
{
    list l1;
    if ((l == nullptr) || (l->next == nullptr)) return;
    l1 = l->next;
    reverse(l1); // capovolgi il resto
    l->next->next = l;
    l->next = nullptr;
    l = l1;
}
```

reverse ricorsiva di una lista



```
void reverse(list &l)
{
    list l1;
    if ((l == nullptr) || (l->next == nullptr)) return;
    l1 = l->next;
    reverse(l1); // capovolgi il resto
    l->next->next = l;
    l->next = nullptr;
    l = l1;
}
```

reverse ricorsiva di una lista



```
void reverse(list &l)
{
    list l1;
    if ((l == nullptr) || (l->next == nullptr)) return;
    l1 = l->next;
    reverse(l1); // capovolgi il resto
    l->next->next = l;
    l->next = nullptr;
    l = l1;
}
```


Tecniche divide-et-impera

- In molti casi le soluzioni ricorsive derivano in modo naturale dall'applicazione del meccanismo *divide-et-impera* (in inglese, *divide and conquer*)
1. Lo abbiamo già visto. Dato il problema P su un input I di cardinalità n
 2. **if** n è sufficientemente piccolo
 risolvi P direttamente
 else
 (a) suddividi I in sottoparti di cardinalità minore
 (b) risolvi P su ogni sottoparte
 (c) ricombina le soluzioni
 end

Tecnica divide-et-impera: esempi notevoli

- Ricerca binaria su sequenza ordinata
- Ordinamento: merge sort

Ricerca binaria e divide-et-impera

Algorithm binary_search(s, x)

if s vuota **then**

x non trovato

else

confronta x con l'elemento al centro di s, sia e

if x uguale a e **then**

x trovato

else if $x < e$ **then**

cerca x nella parte di s che precede e

else

cerca x nella parte di s che segue e

endif

endif

Ricerca binaria iterativa

Ripasso - Ricerca Binaria Iterativa

```
int binarySearch(const int list[], int length, int item)
{
    int first=0;
    int last=length-1;
    int mid;
    bool found=false;

    while(first<=last && !found)
    {
        mid=(first+last)/2;
        if (list[mid]==item)
            found=true;
        else
            if (list[mid]>item)
                last=mid-1;
            else first=mid+1;
    }
    if (found)
        return mid;
    else
        return -1;
}
```

variabili ausiliarie
marcano gli estremi (indici)
della porzione di sequenza
considerata

Ricerca binaria ricorsiva

Notare l'uso della
funzione ausiliaria!

```
bool ric_binaria_aux(const int array[N], int elem, int first, int last )
{
    if (first > last) return false; // non ho trovato
    int mid=(first+last)/2;
    if (elem == array[mid]) {
        return true; // trovato
    }
    else{
        if (elem < array[mid])
            return ric_binaria_aux(array, elem, first, mid-1);
        else
            return ric_binaria_aux(array, elem, mid+1, last);
    }
}

bool ric_binaria(const int array[N], int elem)
{
    bool b=ric_binaria_aux(array,elem, 0, N-1);
    cout << b << endl;
    return b;
}
```

i parametri di una funzione ausiliaria
marcano gli estremi (indici)
della porzione di sequenza
considerata

Merge sort: ordinamento con divide-et-impera

- Immaginiamo di avere in input una sequenza di valori da ordinare
- dividiamo la sequenza a metà
- ordiniamo ogni metà
- infine ricombiniamo le parti ordinate (merge)

Merge sort: il passo di fusione

- immaginiamo di avere le seguenti due sequenze ordinate e di volerle fondere

$X = 2, 3, 10, 20$

$Y = 4, 15, 25, 27$

$Z = 2, 3, 4, 10, 15, 20, 25, 27$

- consideriamo due variabili *posizione* una per X una per Y (indici nel caso degli array, puntatori nel caso di liste)
- confrontiamo gli elementi corrispondenti alle due posizioni, copiamo il minimo dei due nella sequenza di output

Merge sort: il passo di fusione

-

```
i = 0; j = 0; k = 0;
while ( i < p and j < q ) do
  if X[i] < Y [j] then
    Z[k] = X[i]; i + +;
  else
    Z[k] = Y [j]; j + +;
  end if
  k + +;
end while
while i < p do
  Z[k] = X[i]; i + +; k + +;
end while
while j < q do
  Z[k] = Y[j]; j + +; k + +;
end while
```


Merge sort: il passo di fusione

- osserviamo che in realtà possiamo considerare le due sequenze come porzioni diverse dello stesso array o lista

```
function merge(s, inf, med, sup)
    inf inizio prima sequenza
    med inizio seconda sequenza
    sup fine sequenza
```

PSEUDO-CODICE!

sta in piedi indipendentemente dalla scelta di
come realizzare le sequenze (array, vector o liste)

Merge sort: la struttura

- la procedura da seguire sfrutta il concetto di ricorsione e si avvale di una funzione ausiliaria

```
function mergesort(sequenza s)  
    inf=first(s)  
    sup=last(s)  
    ms(s,inf, sup)
```

due cursori delimitano
la parte di sequenza
che vogliamo ordinare

PSEUDO-CODICE!

sta in piedi indipendentemente dalla scelta di
come realizzare le sequenze

Merge sort: la struttura

- **function** ms (sequenza s, cursori inf, sup)
 if inf >=sup **then**
 return;
 else
 med=mezzo (inf, sup)
 ms (s,inf,med)
 ms (s,med+1,sup)
 merge (s,inf,med,sup)
 end if

PSEUDO-CODICE!

sta in piedi indipendentemente dalla scelta di
come realizzare le sequenze