# la ricorsione

introduzione alla programmazione

# il principio della ricorsione

- la ricorsione è un principio molto potente
- definizione *naif:* 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 cardinalità *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)* /* input I di cardinalità n */
  **if** n è sufficientemente piccolo
  risolvi P direttamente
  **else**
  suddividi I
  risolvi ogni sottoparte attraverso la chiamata
  AR(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*
- *In matematica utilizzata per definizioni e dimostrazioni*

# esempio - il fattoriale

- definizione:
  - $n! = n*(n-1)*(n-2)\ldots2*1$
  - $0!=1$
- base $0!=1$
- passo per ogni $n>0$, $n! =_{def} n(n-1)!$

# esempio - il coefficiente binomiale

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

- 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

fattoriale(int n)

int n

- una funzione ricorsiva dipenderà da un parametro
- la definizione della funzione seguirà il principio di induzione
  - definisco la base in modo diretto `if (n==0) return 1;`
  - *passo induttivo: comprende una chiamata* ***ricorsiva*** *della funzione stessa su un input semplificato*

fattoriale(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":
fr(3)=3*fr(2)=3*2*fr(1)=3*2*1*fr(0)=3*2*1*1=6

# Esempio - potenze

- Altro esempio di ricorsione in coda n^m

```
int recoursive_power (int n, unsigned int m) {

    if (m==0) return 1;
    if(m==1) return n;

    return n*recoursive_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 si una sequenza (stampa, ricerca, calcolo di min/max, …) possono essere progettate tramite ricorsione
- **NB: ricorsione è alternativa ai cicli**

# Un esempio

```
void recoursive_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 SUCCEDE SE METTO LA STAMPA DOPO LA CHIAMATA
    recoursive_print_rev(A,size-1) ;

}
```

# Un altro esempio

```
int recoursive_sum( int A[], int size) {

    if (size<=0) throw ERROR;
    if (size==1) return A[size-1];
    return A[size-1]+recoursive_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

- <u>esempio:</u> 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

# ricorsione e liste

- ```
  bool is_in(const list& l, T x)
  {
     if (l==nullptr) return false; //base
     if (l->info==x) return true; //base
     return is_in(l->next, x);
  }
  ```

- ```
  int length(const list& l)
  {
     if (is_empty(l)) return 0; // base
     else
           return length(l->next) + 1;
  ```

- analogamente per l’inserimento ordinato e la cancellazione su lista ordinata (per esercizio)

# ricorsione e liste

Notare l'uso della funzione ausiliaria!

grazie alla "valutazione lazy", se current==nullptr, l'espressione a destra dell'OR non verrà valutata

```
void insertElemInOrderAux(cell* aux, list& current, list& prev)
// assumo che questa funzione ausiliaria venga chiamata con prev != nullptr (il controllo viene fatto in
insertElemInOrder)
{
    if (current==nullptr || current->head > aux->head) // CASO BASE: devo inserire l'elemento dopo prev e prima
di current
       {
          aux->next=current;
          prev->next=aux;
       }

    else // non sono ancora nel punto giusto: richiamo ricorsivamente sulla coda della lista cambiando gli
argomenti per spostarmi in avanti di una cella nella lista
       insertElemInOrderAux(aux, current->next, current);

}



void insertElemInOrder(const Elem x, list& s)
{
    if (member(x,s))           // verifico per prima cosa se l'elemento e' gia' presente, per non ri-inserirlo
       return;

    cell *aux;   // mi preparo la cella nuova
    aux = new cell;
    aux->head=x;
    aux->next=nullptr;

    if (s == nullptr || s->head>x){  // questi sono casi particolari
        aux->next=s;
        s=aux;
    }
    else // altrimenti chiamo insertElemInOrderAux
        insertElemInOrderAux(aux, s->next, s);


}
```

# 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