

# Algoritmica

domenica 15 settembre 2019 14:14

## Sviluppo di programmi

Lo sviluppo di un programma si articola in **quattro fasi** distinte:

- **Analisi**, ossia comprensione del problema e definizione dei vincoli e delle specifiche.
  - o L'**analisi specifica cosa fa il programma**, individuando i dati in ingresso e in uscita, e i loro rispettivi vincoli.
    - I **vincoli dei dati sono la precondizione e la postcondizione**. La precondizione è relativa ai dati in ingresso dev'essere soddisfatta affinché la funzione sia applicabile; la postcondizione dev'essere soddisfatta al termine del programma ed è applicata ai dati in uscita -- significa che la postcondizione ci dice "come sono i dati di output in funzione di quelli di input".
    - È buona norma utilizzare un **dizionario dei dati**, ossia una tabella il cui schema è "Identificatore, Tipo, Descrizione". La descrizione individua il contesto in cui viene impiegato il suddetto dato.
- **Progettazione**, cioè la scelta della strategia e la conseguente formulazione di un algoritmo in pseudocodice.
  - o La **progettazione specifica come il programma fa quello che deve fare**, cioè di come effettua la trasformazione dei dati specificata.
  - La progettazione è uno processo di stepwise-refinement, cioè l'elaborazione di un algoritmo per raffinamenti successivi, articolata in definizione degli step e decomposizione funzionale.
- **Implementazione**, cioè la codifica della soluzione, il test e il debugging.
- **Esecuzione**, cioè l'applicazione dell'algoritmo sui dati reali.

Osserviamo che un algoritmo effettua una visita totale quando vengono analizzati tutti gli elementi; effettua, invece, una visita finalizzata quando la visita stessa termina al verificarsi di una determinata condizione.

## Esempio di progettazione

**Dati in ingresso:** sequenza  $s$  di  $n$  interi

**Precondizione:**  $n > 0$

**Dati in uscita:** sequenza  $s1$  di  $n$  interi

**Postcondizione:**  $s1$  è una permutazione di  $s$  dove  $\forall i \in [0, n - 2] s1_i \leq s1_{i+1}$

### Dizionario dei dati

Identificatore	Tipo	Descrizione
$s$	sequenza	Sequenza di interi in input
$s1$	sequenza	Sequenza di interi in output
$n$	intero	Numero di elementi nella sequenza
$i$	intero	Indice per individuare gli elementi nella sequenza

### Progettazione

- a. Input sequenza  $s$  in un array  $a$  di dimensioni  $n$
- b. Ordina array  $a$  di dimensione  $n$
- c. Output sequenza  $s1$  contenuta in array  $a$  di dimensione  $n$

**La progettazione può e deve contenere anche pseudocodice.**

[!] Raffinamento di (b): decomponiamo funzionalmente lo step in `input_array(a, n)`, `ordina_array(a, n)` e `output_array(a, n)`, ognuno dei quali avrà una sua analisi, una progettazione, una codifica e verifica come questo intero esempio.

Ad esempio, `ordina_array(a, n)` avrà un'analisi praticamente identica a questa, mentre come progettazione avremo uno dei vari algoritmi di ordinamento.

Questo significa che effettuo prima l'analisi e la progettazione del problema generale, per poi andare a rifare l'analisi e la progettazione sulle singole funzioni individuate nella progettazione del soluzione generale.

**In conclusione, l'analisi e la progettazione vanno fatte per ogni funzione, anche per il `main()`.**

## Esempi delle potenzialità dell'algoritmica: la ricerca binaria

Consiste nel dividere l'array in due metà e confrontare l'elemento da cercare con l'elemento centrale dell'array.

- Uguale? Trovato
- Minore? Continuo la ricerca nella prima metà dell'array.
- Maggiore? Continuo la ricerca nella seconda metà dell'array.
- Se l'elemento non è presente, l'array si ridurrà ad un solo elemento e non sarà divisibile in due.
- Nel caso peggiore, si visitano log<sub>2</sub> n elementi.

- Cercare l'elemento 7 nell'array ...



```
int binary_search(int a[], int n, int el) {  
    int start = 0, center, end = n-1;  
    while(end>=start) { // vede se c'è più di un elemento  
        center = (start+end)/2;  
        if(a[center] == el)  
            return center;  
        else {  
            if(el < a[center])  
                end = center-1;  
            else  
                start = center+1;  
            // -+1 perchè il centro lo abbiamo già considerato  
        }  
    }  
    return -1;  
}
```

# Ordinamento

venerdì 13 marzo 2020 14:45

## Introduzione

Il problema dell'ordinamento consiste nell'elencare gli elementi (**record**) di un insieme secondo una sequenza stabilita da una relazione d'ordine; tale relazione può valutare un singolo intero, le lettere dell'alfabeto, ma anche delle chiavi (**keys**): la chiave può essere un singolo campo o la combinazione di più campi del record.

Un efficiente algoritmo di ordinamento è:

- **Stabile**, due record con la stessa chiave mantengono lo stesso ordine dopo l'ordinamento.
- **In loco**, in ogni dato istante si può allocare un numero costante di variabili, oltre all'array da ordinare (cioè non interferisce col resto del programma).
- **Adattivo**, il numero di operazioni dipende dall'input, ossia dall'ammontare di record.
- **Interno**, i dati contenuti nella RAM, oppure **esterno**, con i dati contenuti su HDD.

La maggior parte degli algoritmi di ordinamento agisce per confronti.

L'approccio generale per la risoluzione di problemi computazionali secondo il metodo Divide et Impera contempla tre fasi:

- Divide: si procede alla suddivisione dei problemi in problemi di dimensione minore.
- Impera: i problemi vengono risolti in modo ricorsivo. Quando i sottoproblemi arrivano ad avere una dimensione sufficientemente piccola, essi vengono risolti direttamente tramite il caso base.
- Combina: si ricombina l'output ottenuto dalle precedenti chiamate ricorsive al fine di ottenere il risultato finale.

Gli algoritmi semplici effettuano un numero di operazioni quadratico rispetto alla taglia di input  $O(n^2)$ .

- Selection Sort
- Insertion Sort
- Bubble Sort

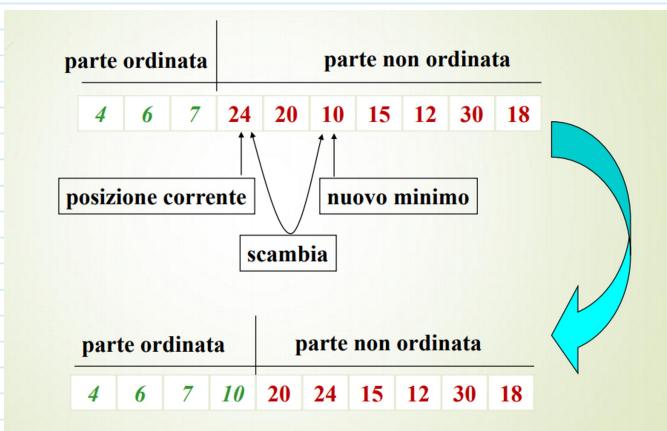
Gli algoritmi più efficienti, invece, sono i seguenti.

- Merge Sort
  - $O(n \log n)$  sempre
- Quicksort
  - $O(n \log n)$  nel caso medio
  - $O(n^2)$  nel caso peggiore

Nome	Migliore	Medio	Peggior	Memoria	Stabile	In Loco
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Sì
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)^*$	No	Sì
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Sì	No

## Selection Sort

Effettua una visita totale delle posizioni dell'array, cioè visita in sequenza tutti gli elementi.  
Per ogni posizione visitata individua l'elemento che dovrebbe occupare quella posizione.



```
void selection_sort(int *a, int n) {
    int i, m;
    for (i=0; i<n-1; i++) {
```

```

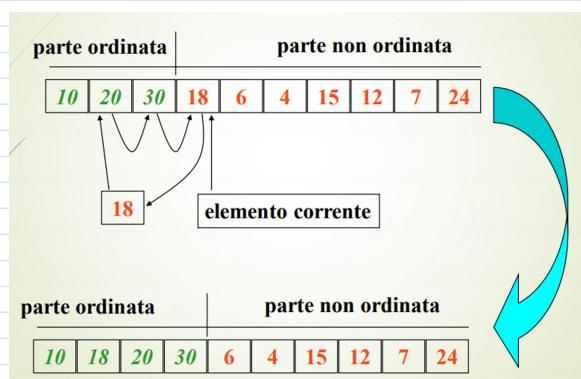
void selection_sort(int *a, int n) {
    int i, m;
    for (i=0; i<n-1; i++) {
        m = min(a+i,n-i)+i;
        swap(&a[i],&a[m]);
    }
}

```

## Insertion Sort

Effettua una visita totale dell'array. Ad ogni passo gli elementi che precedono l'elemento corrente sono ordinati.

Si inserisce l'elemento corrente nella posizione ordinata. Gli elementi precedenti maggiori sono spostati in avanti; il primissimo elemento è già ordinato.



```

void insert_sorted_array(int *a, int val, int *n){
    int i;
    for(i=*n;i>0;i--)
        if(val<a[i-1])
            a[i]=a[i-1];
        else
            break;
    a[i]=val;
    (*n)++;
}

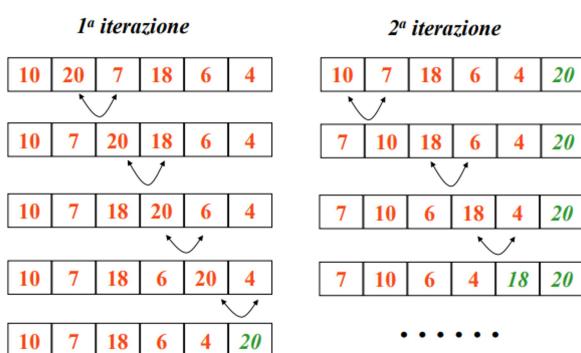
void insertion_sort(int *a, int n){
    int i;
    for(i=1;i<n;){
        insert_sorted_array(a,a[i],&i);
    }
}

```

## Bubble Sort

Si effettuano  $n-1$  visite all'array. All'i-esima, si confrontano gli elementi adiacenti dal primo al  $(n-i)$ -esimo elemento.

Elementi adiacenti che non risultano scambiati vengono ordinati. Se in una visita non vengono effettuati scambi, l'array è ordinato.



```

void bubble_sort(int *a, int n){
    int i,j;
    for(i=1;i<n;i++)
        for(j=0;j<n-i;j++)
            if(a[j]>a[j+1])
                swap(&a[j],&a[j+1]);
}

```

## Quick Sort

Il Quick Sort è ricorsivo e usa il metodo divide et impera. È un algoritmo in loco, non stabile e l'idea di base è ordinare un array con delle Partitions.

- Scegli un elemento detto **pivot**.
- Metti a sinistra gli elementi  $\leq$  pivot, a destra  $>$ .

Ad esempio, con  $pivot = 17$ , valutiamo la seguente situazione:

A questo punto, i due puntatori avanzano dalle loro rispettive posizioni. Quando si incroceranno, la procedura ricomincerà da capo.



- **Divide:** partiziona il vettore  $A[P..R]$  in due sottovettori rispetto ad un pivot  $x$ , il quale si troverà nella posizione  $q$ .
- **Impera:** quicksort sul sottovettore sinistro  $[P .. Q-1]$  e poi sul destro  $[Q+1 .. R]$ .
  - Passo base: il vettore ha 1 elemento; è quindi già ordinato

Trattandosi di un algoritmo di ordinamento basato sui confronti, analizziamo il numero di confronti in funzione del numero di oggetti da ordinare. Tutti i confronti sono eseguiti nella procedura **partition**. Quando **partition** è eseguita su un sottoarray di lunghezza  $m$ , vengono eseguiti  $m$  confronti (ogni elemento è confrontato una volta con il pivot).

Se l'array da ordinare viene partizionato in due array di dimensione  $r$  e  $n-r$ , abbiamo che il numero di confronti.

$$T(n) = 0 \text{ se } n = 1, T(r) + T(n - r) + n, \text{ se } n > 1.$$

Questa ricorrenza non è del tipo che si può ricondurre ai casi presentati, perché  $r$  cambia ad ogni chiamata della procedura.

L'**efficienza del quicksort** è legata al bilanciamento delle partizioni.

- Nel caso peggiore, si ha un vettore da  $n - 1$  elementi e l'altro da 0.
- Nel caso migliore, si hanno due vettori da  $\frac{n}{2}$  elementi.
- Nel caso medio, si hanno due vettori di dimensioni diverse.

**Il bilanciamento è legato alla scelta del pivot.**

Nel **caso peggiore**, il pivot è il minimo o il massimo dell'array. Con questa scelta del pivot, il caso peggiore si presenta solo quando l'array è già ordinato, a prescindere dal suo ordine. Ergo  $T(n)=T(n-1)+n$ ,  $T(1)=1$  e  $T(n)=O(n^2)$ . Il lavoro di combinazione lineare è quadratico con  $n$ .

Nel **caso migliore**, l'array viene partizionato sempre in due regioni uguali:  $T(n) = 2T(n/2) + n$ ,  $T(1) = 1$  e  $T(n) = O(n \log n)$ . Il lavoro di combinazione è lineare con costo logaritmico.

Nel **caso medio**, se le partizioni sono molto sbilanciate, il costo può risultare molto alto. Se le partizioni sono troppo sbilanciate, si ha un costo logaritmico di combinazione lineare, cioè asintoticamente uguale al caso ottimo.

Il pivot si determina a caso, generando un numero casuale  $i$  con  $p \leq i \leq r$ , e scambiando  $A[1]$  e  $A[i]$ , usando come pivot  $A[1]$ . Il pivot è quindi l'indice. Si può dimostrare matematicamente che il tempo di esecuzione del quicksort con pivot casuale è  $O(n \log n)$  con probabilità molto vicina ad 1. In altre parole, devo essere molto sfortunato per avere un tempo di esecuzione asintoticamente superiore a quest'ultimo.

Tuttavia, la generazione di un indice casuale è molto costoso, perciò si adotta la strategia del "medio di 3", ossia considerare gli elementi che sono nella prima e nell'ultima posizione dell'array, e l'elemento che si trova in una posizione mediana. Si fa poi la media fra questi 3 numeri, e se ne estrae l'indice.

```

void quicksort(int a[MAX],int primo,int ultimo){
    int i, j, pivot, temp;
    /*pivot -- inizialmente il pivot è il primo elemento
    primo e ultimo sono le due variabili che servono per scorrere l'array
*/
    if(primo<ultimo){
        pivot=primo;
        i=primo;
        j=ultimo;

        while(i<j){
            while(a[i]<=a[pivot]&&i<ultimo)
                i++;
            while(a[j]>a[pivot])
                j--;
            if(i<j){
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }

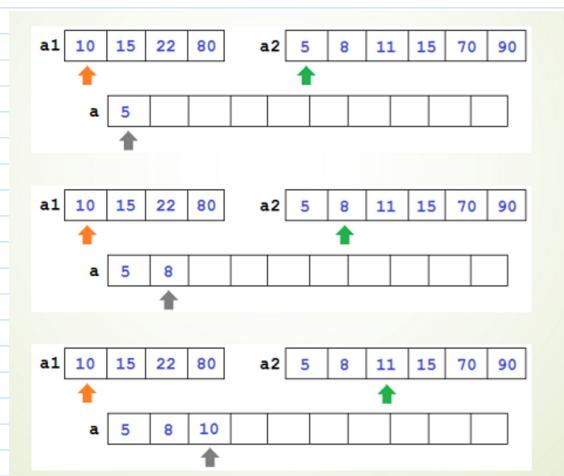
        temp=a[pivot];
        a[pivot]=a[j];
        a[j]=temp;
        quicksort(a,primo,j-1);
        quicksort(a,j+1,ultimo);
    }
}

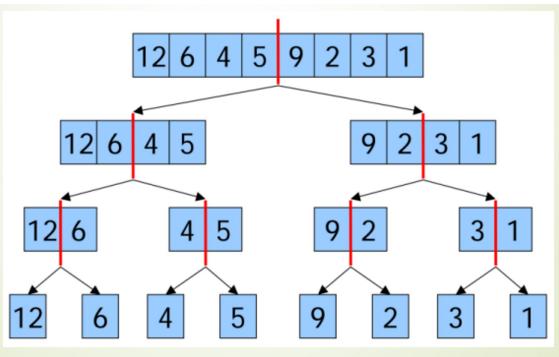
```

## Merge Sort

Inventato da Von Neumann nel 1945. Richiede spazio ausiliario  $O(n)$ . Usa il metodo divide et impera. Il costo è  $T(n) = O(n \log n)$ .

- Scorro  $a_1, a_2$  con indici  $i, j$ .
- Confronto  $a_1[i]$  e  $a_2[j]$ , finchè  $i < n_1, j < n_2$ .
- Altrimenti inserisco  $a_2[j]$  in  $a$  e incremento  $j$ .
- Riverso tutti gli elementi restanti in  $a_1 \vee a_2$  in  $a$ .
- **Impera:** merge sort su sottovettore destro, poi sinistro, condizione base con  $p = r$  o array ordinato.
- **Combina:** merge per fondare i due sottovettori ordinati in un vettore ordinato.





```

void mergeSort(int a[], int p, int r) {
    int q;
    if (p < r) {
        q = (p+r)/2;
        mergeSort(a, p, q);
        mergeSort(a, q+1, r);
        merge(a, p, q, r);
    }
    return;
}

```

```

void merge(int a[], int p, int q, int r) {
    int i, j, k=0, b[max];
    i = p;
    j = q+1;

    while (i<=q && j<=r) {
        if (a[i]<a[j]) {
            b[k] = a[i];
            i++;
        } else {
            b[k] = a[j];
            j++;
        }
        k++;
    }
    while (i <= q) {
        b[k] = a[i];
        i++;
        k++;
    }
    while (j <= r) {
        b[k] = a[j];
        j++;
        k++;
    }
    for (k=p; k<=r; k++)
        a[k] = b[k-p];
    return;
}

```

# Astrazione

venerdì 27 marzo 2020 09:26

## Introduzione

L'**astrazione** è un procedimento mentale che consente di evidenziare le caratteristiche pregnanti di un problema e offuscare gli aspetti che si ritengono secondari rispetto ad un determinato obiettivo.

L'astrazione (sui dati e sulle funzionalità) ha la finalità di **ampliare l'insieme dei modi di operare sui tipi di dati già disponibili, attraverso la definizione di nuovi operatori e dati**; astraendo un programma, riusciamo a delegare ad un sottoprogramma le funzionalità secondarie, le quali sono usabili indipendentemente dall'algoritmo.

Nel C, un tipo di dati è definito da un insieme di operazioni previste sui valori compresi nel dominio dei valori. I tipi di dati possono essere primitivi ed aggregati; oltre questi, vi sono i puntatori.

## Abstract Data Types (ADT)

I tipi di dati astratti estendono i dati pre-esistenti, e vengono definiti distinguendo Specifica ed Implementazione.

### Specifiche

- *Definizione del tipo di dati*
- *Definizione dell'insieme degli operatori*
  - *Sintattica (nomi, tipi)*
  - *Semantica (valori, vincoli)*

### Implementazione

- *Codifica di quanto definito nella specifica in un linguaggio di programmazione usando dati primitivi e costrutti di default*
- *È spesso nascosta al programmatore seguendo il principio dell'incapsulamento (information hiding)*

Gli ADT vengono definiti usando una pratica tabella.

	Sintattica	Semantica
<b>Tipi di dati</b>	- Nome dell'ADT - Tipi da dati già usati	- Insieme dei valori
<b>Operatori (una riga per ogni operatore)</b>	- Nome dell'operatore - Tipi di dati di input e di output	Funzione associata all'operatore - Precondizioni - Postcondizioni

Sintattica	Semantica
Nome del tipo: Punto Tipi usati: Reale	Dominio: Insieme delle coppie (ascissa, ordinata) dove ascissa e ordinata sono numeri reali
creaPunto (reale, reale) → punto	creaPunto(x, y) = p • pre: true • post: p = (x, y)
ascissa (punto) → reale	ascissa(p) = x • pre: true • post: p = (x, y)
ordinata (punto) → reale	ordinata(p) = y • pre: true • post: p = (x, y)
distanza (punto, punto) → reale	distanza(p1, p2) = d • pre: true • post: d = sqrt( (ascissa(p1)-ascissa(p2))^2 + (ordinata(p1)-ordinata(p2))^2 )

## Pseudo-generics

Gli **pseudo-generics** sono uno strumento che permettono la **definizione di un tipo parametrizzato**, il quale viene esplicitato successivamente in fase di compilazione o linking secondo le necessità. Consentono di eseguire l'algoritmo ed applicare gli ADT su tipi di dati diversi. La potenzialità è quella di implementare algoritmi e ADT che siano in grado di funzionare di volta in volta col tipo di dato desiderato.

Ese. item-punto, item-str [...]

# Strutture

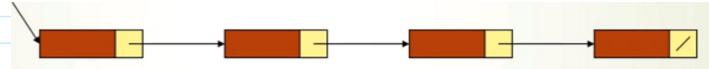
giovedì 2 aprile 2020 12:52

## List

Le **liste** sono particolari ADT. Sono caratterizzate da sequenze di elementi di un determinato tipo (anche il generico **Item**), in cui è possibile aggiungere o togliere elementi. **Ogni elemento di una lista è un record con un campo puntatore che serve da collegamento per il record successivo.** Si accede alla struttura attraverso il puntatore al primo record. Il campo puntatore dell'ultimo record contiene **NULL**.

A differenza di un array, le liste hanno dimensione variabile e l'accesso è diretto solo al primo elemento della lista. Per accedere al generico elemento, occorre scandire sequenzialmente gli elementi della lista.

### Efficienza computazionale



- Con le **liste**, per accedere all'i-esimo elemento occorre scorrere la lista dal primo all'i-esimo elemento (tempo massimo proporzionale al *size*). Inoltre, è possibile eliminare un elemento o aggiungerlo uno alla testa direttamente (tempo costante).
- Con gli **array**, ogni elemento è accessibile usando l'indice (tempo costante). Per inserire o cancellare un elemento occorre shiftare gli altri elementi (tempo massimo proporzionale al *size*).

## Stack

Una **pila** è una sequenza di elementi di un determinato tipo in cui è possibile aggiungere o togliere elementi esclusivamente da un unico lato: quello superiore. La pila è una struttura dati lineare a dimensione variabile, in cui si può accedere direttamente solo al primo elemento della lista. Non è possibile accedere ad un elemento diverso dal primo se non dopo aver eliminato tutti gli elementi che lo precedono (e quindi inseriti dopo).

La pila è gestita, inoltre, in modalità **LIFO** (*Last-In-First-Out*), cioè l'ultimo elemento inserito nella sequenza sarà il primo ad essere eliminato.

È possibile implementare le pile sia con array che con liste.

## Queue

Una **queue** o coda è una sequenza di elementi di un determinato tipo in cui tali elementi si aggiungono da un lato e si tolgono dall'altro lato (head). La coda è una struttura dati lineare a dimensione variabile. Si può accedere direttamente solo alla testa della lista e non è possibile accedere ad un elemento diverso dalla testa se non dopo aver eliminato tutti gli elementi che lo precedono, e cioè quelli inseriti prima.

La sequenza viene gestita con la modalità **FIFO** (*First-In-First-Out*): il primo elemento inserito nella sequenza sarà il primo ad essere eliminato.

È possibile implementare le code sia con array circolari che con liste concatenate.

## Graphs

Un **grafo** orientato  $G$  è una coppia  $\langle N, A \rangle$  dove  $N$  è un insieme finito non vuoto di nodi e  $A \subseteq NxN$  è un insieme finito di coppie ordinate di **nodi**, detti **archi**, spigoli o linee. Se  $\langle u_i, u_j \rangle \in A$ , nel grafo vi è un arco che passa fra i due nodi.

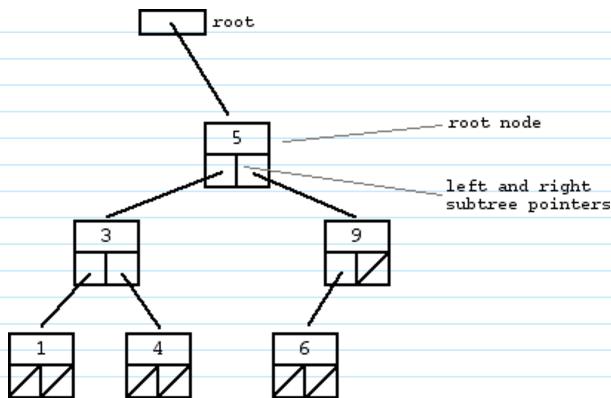
## Binary Trees

Ogni nodo di un **albero** ha un unico arco entrate, tranne la **radice**. Ogni nodo può avere zero o più archi uscenti; i nodi senza archi uscenti sono detti **foglie**. Un arco dell'albero induce una relazione padre e figlio e a ciascun nodo è solitamente associato un valore (**Item**) detto **etichetta** del nodo.

- Il **grado** di un nodo è il numero di figli del nodo.
- L'**ordine** dell'albero è il massimo grado fra tutti i nodi.
- Il **cammino** è una sequenza di nodi.
- La **lunghezza** del cammino è la quantità di archi in un cammino.
- Il **livello** di un nodo è la lunghezza del cammino dalla radice a tale nodo.
- L'**altezza** dell'albero è la lunghezza del più lungo cammino.

A differenza dei grafi, gli alberi sono **grafi diretti aciclici**, in cui per ogni nodo esiste un solo arco entrate (tranne che per la radice).

Se esiste un cammino che va da un nodo ad un altro, tale cammino è unico; inoltre, in un albero esiste un solo cammino che va dalla radice a qualunque altro nodo. Infine, dato un nodo, i suoi discendenti costituiscono un albero detto **sottoalbero**.



Gli alberi binari sono **alberi i cui nodi hanno al più 2 figli**, i quali sono chiamati sottoalberi sinistri e destri.

Un albero binario è tale se è vuoto oppure se è definito come una terna  $< S, R, D >$ , dove  $r$  è la radice e  $s, d$  sono i sottoalberi sinistri e destri.

Un albero binario è definito come puntatore ad un nodo: se è vuoto, il puntatore è nullo; se non lo è, punta al nodo radice.

Ogni nodo contiene l'item e i sottoalberi sinistri e destri.

```

struct node {
    Item etichetta;
    struct node *alberosx;
    struct node *alberodx;
};

typedef struct node *Btree;
Btree T = NULL;

```

Si noti che **un albero binario è costruito in maniera bottom-up**, cioè partendo dalle foglie.

La visita di un albero può essere effettuata in tre modi:

- **Visita in pre-ordine**: indago il contenuto del nodo, poi i suoi sottoalberi.
- **Visita in post-ordine**: indago i sottoalberi del nodo, poi il contenuto del nodo stesso.
- **Visita simmetrica**: indaga prima il sottoalbero sinistro, poi analizza la radice, e poi il sottoalbero destro.

# Ricorsione

martedì 21 aprile 2020 09:48

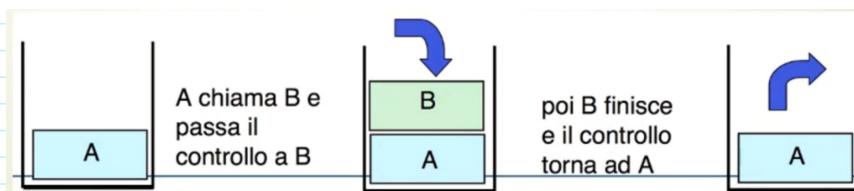
## Record di attivazione

Il **record di attivazione**, o **mondo della funzione**, è un **blocco di memoria** che contiene i *parametri formali*, le *variabili locali*, l'*indirizzo di ritorno* (che indica il punto di memoria a cui tornare nel codice del chiamante al termine della funzione), il *link dinamico* (cioè il collegamento al record di attivazione del chiamante) e l'*indirizzo del codice della funzione* (un puntatore alla primissima istruzione della funzione).

Il record di attivazione è associato ad una funzione *f*. Viene creato al momento dell'invocazione della funzione e permane in memoria per tutto il tempo in cui la funzione è in esecuzione. Viene, infine, distrutto - cioè deallocated - al termine dell'esecuzione. La dimensione del record di attivazione varia in base alla funzione, ma è comunque fisso e sempre calcolabile data una funzione.

L'area di memoria in cui vengono allocati i record di attivazione vengono gestiti con delle liste LIFO, ossia degli **Stack**, nelle quali **ogni item è un record di attivazione**.

Se la funzione *A* chiama la funzione *B*, lo stack evolve nel modo seguente:



## Programmazione ricorsiva

Un **sottoprogramma ricorsivo** richiama direttamente o indirettamente sé stesso. I linguaggi che gestiscono la ricorsione lo fanno mediante i record di attivazione sopraindicati; operativamente, risolvere un problema con un approccio ricorsivo comporta:

- Identificazione del caso base *n*, con soluzione nota.
- Espressione della soluzione al caso generico per *n* + 1.

Si noti che la funzione chiamata è detta *Servitore*, e quella chiamante *Cliente*.

In genere, **una funzione matematica è definita ricorsivamente quando nella sua definizione compare un riferimento a sé stessa**. È basata sul principio di **induzione matematica**.

Esempi noti sono il fattoriale e Fibonacci.

## Processo computazionale ricorsivo

I risultati iterativi, partoriti dai **processi computazionali iterativi**, vengono sintetizzati in avanti: la caratteristica fondamentale di un processo computazionale iterativo è che ad ogni passo è disponibile un *risultato parziale*; dopo *k* passi, si ha a disposizione il risultato parziale relativo al caso *k*. Questo non è vero nei processi computazionali ricorsivi, in cui nulla è disponibile finché non si è giunti fino al caso elementare.

Si noti che i processi computazionali iterativi si possono realizzare anche tramite funzioni ricorsive, e si basano sulla disponibilità di una variabile detta *accumulatore*, destinata ad esprimere in ogni istante la soluzione corrente. Una ricorsione realizzata tramite iterazioni è detta **ricorsione apparente** o **ricorsione tail**.

A livello di prestazioni, le iterazioni sono tendenzialmente più efficienti. Tuttavia, sotto un profilo ingegneristico, la ricorsione è più leggibile.

## Funzione fattoriale

```
long int factorial(int n) {
    if(n >= 1)
        return (n * factorial(n-1));
    else
        return 1;
}
```

## Funzione MCD

```
int mcd(int a, int b) {
    if(b == 0)
        return 0;
    else
        return mcd(b, a%b)
```

```
}
```

## Stringhe palindrome

// Ricorsione

```
int palindroma(char *str) { // Es. bob, sos, onorarono
    char *s = strdup(s, str);
    if(strlen(s) <= 1)
        return 1; // Stringa palindroma
    else {
        int i = strlen(s) - 1;
        if(s[0] != s[i])
            return 0; // Stringa non palindroma
        else {
            s[strlen(s)-1] = '\0';
            palindroma(s+1);
        }
    }
}
```

// Ricorsione Tail

```
int palindromaTail(char *s) { // Per generare i, j iniziali
    return palindromaIterTail(s, 0, strlen(s) - 1);
}

int palindromaIterTail(char *s, int i, int j) {
    if(i < j) {
        if(s[i] != s[j])
            return 0;
        return palindromaIterTail(s, i + 1, j - 1);
    }
    return 1;
}
```

# Complessità

martedì 28 aprile 2020 10:03

## Complessità computazionale

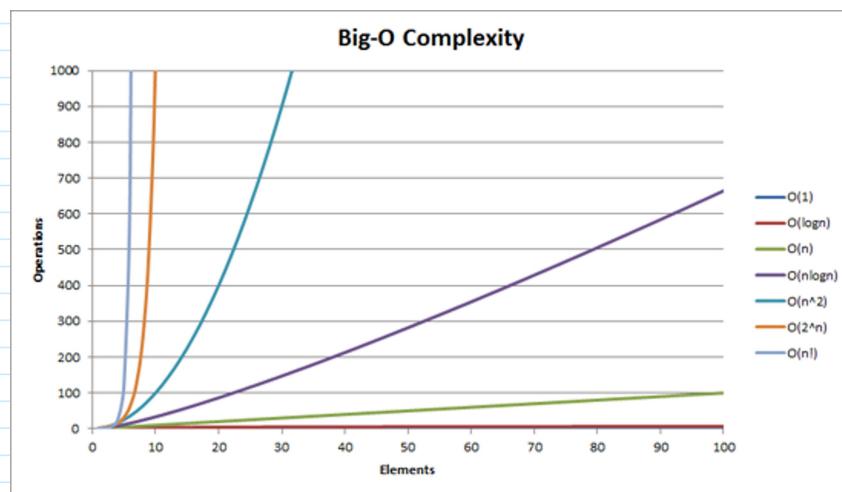
L'analisi della complessità è la stima del costo degli algoritmi in termini di risorse di calcolo. Con "risorse di calcolo" si intedono in particolar modo il **tempo e lo spazio di memoria necessari per l'esecuzione di un algoritmo**. Tale unità di misura è influenzata da alcune variabili che influenzano il tempo d'esecuzione, come la *macchina usata*, la *dimensione dei dati* e la *configurazione* di questi ultimi.

Per sovvenire a certe necessità, è stato elaborato un modello astratto per la valutazione del tempo d'esecuzione che rispetta alcune caratteristiche, come l'indipendenza dalla macchina usata, la stima in funzione della taglia dell'input e la valutazione nel caso peggiore di configurazione dei dati. Inoltre, questo modello permette di valutare il comportamento asintotico.

Una **macchina astratta** ha istruzioni e condizioni atomiche a costo unitario.

- Le strutture di controllo hanno un costo pari alla somma dei costi dell'esecuzione delle istruzioni interne, più la somma dei costi delle condizioni.
- Le chiamate a funzione hanno un costo pari al costo di tutte le sue istruzioni e condizioni; il passaggio dei parametri, invece, ha costo nullo.
- Istruzioni e condizioni con chiamate e a funzioni hanno costo pari alla somma del costo delle funzioni invocate più uno.

La **taglia dell'input** è, ad esempio, il numero di elementi in un vettore, quello di nodi in un albero e quello di archi più quello dei nodi in un grafo. Se  $n$  è la dimensione dell'input,  $\log_2 n$  è la sua taglia  $d$  in bit.



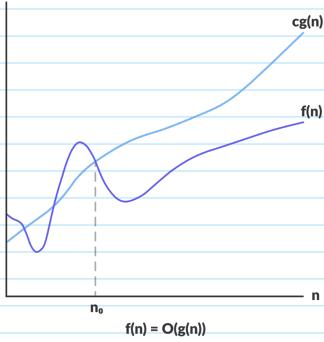
## Comportamento asintotico

Nell'analizzare la complessità di tempo di un algoritmo siamo interessati a come aumenta il tempo al crescere della taglia  $n$  dell'input. Siccome per valori piccoli il tempo richiesto è comunque poco, ci interessa soprattutto il comportamento per valori grandi, cioè il comportamento asintotico: si intende, dunque, il comportamento al crescere della dimensione dell'input all'infinito.

Per far ciò, si suddividono gli algoritmi in classi di complessità:

$a$	Costante
$an + b$	Lineare
$an^2 + bn + c$	Quadratica
$a\log_2 n + h$	Logaritmica
$a^n$	Esponenziale
$n^n$	Esponenziale

In effetti, calcoli alla mano, notiamo che le classi esponenziali impiegano più tempo di tutti e che in un algoritmo composto da più classi computazionali, quello veramente considerato è il più grande. Per grandi dimensioni dell'input, non notiamo sostanziali differenze fra le classi; mentre per grandi taglie di input, gli algoritmi lineari e quadratici, così come quelli logaritmici, impiegano decisamente meno tempo rispetto alle classi esponenziali.



## Funzioni per la complessità computazionale

Siano  $f, g$  funzioni dai naturali ai reali positivi.

$f(n) \in O(g(n))$  se esistono due costanti positive  $c, n_0: n \geq n_0, f(n) \leq cg(n)$ .

Applicata alla funzione di complessità  $f(n)$ , la funzione  $O$  ne limita superiormente la crescita e fornisce quindi un'indicazione della bontà dell'algoritmo.

$f(n) \in \Omega(g(n))$  se esistono due costanti positive  $c, n_0: n \geq n_0, cg(n) \leq f(n)$ .

Applicata alla funzione di complessità  $f(n)$ , la funzione  $\Omega$  ne limita inferiormente la crescita, indicando così che il comportamento dell'algoritmo non è migliore di un comportamento assegnato.

In conclusione, il limite asintotico stretto  $\Omega(n) \neq \Theta(n) \neq \Omega(n) \neq \{f(n): \exists c_1, c_2 > 0 \wedge n_0: \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$ .

## Complessità e ricorsione

Negli algoritmi ricorsivi, la soluzione di un problema si ottiene applicando lo stesso algoritmo ad uno o più sottoproblemi. La complessità viene espressa nella forma di una relazione di ricorrenza: in effetti, la funzione di complessità è definita in termini di sé stessa su una dimensione inferiore dei dati.

Per la **valutazione della complessità** si considerino:

- Il lavoro di combinazione (cioè la preparazione delle chiamate ricorsive e la combinazione dei risultati ottenuti): lineare, costante, ...
- La forma dell'equazione di ricorrenza, che può avere o meno una partizione dei dati.
- Il numero di chiamate ricorsive nella funzione.

Se l'algoritmo applica ripetutamente un certo insieme di istruzioni la cui complessità all'i-esima esecuzione vale  $f_i(n)$ ; con  $g(n)$  numero di ricorsioni, la complessità è  $\Omega(g(n)f(n))$

Si noti che un'istruzione di costo di esecuzione  $f(n)$  è dominante quando viene eseguita  $g(n)$  volte, con  $f(n) \leq a * g(n)$ . Se un algoritmo ha un'operazione dominante, allora è  $\Omega(g(n))$

### a. Lavoro di combinazione costante

- i.  $T(n) = a_1 T(n - 1) + a_2 T(n - 2) + \dots + a_n T(n - h) + b, \quad n > h$ 
  - 1) Esponenziale con  $n$ : se sono presenti almeno 2 termini (cioè se l'algoritmo contiene almeno due chiamate ricorsive).
  - 2) Lineare con  $n$ : se è presente un solo termine (una singola chiamata ricorsiva).

$$\text{ii. } T(n) = aT\left(\frac{n}{p}\right) + b, \quad n > 1$$

- 1)  $\log n$ , se  $a = 1$  (singola chiamata ricorsiva)
- 2)  $n^{\log_p a}$ , se  $a > 1$

### b. Lavoro di combinazione lineare

- i. Quadratico per  $n$ :  $T(n) = T(n - h) + b * n + d$ , per  $n > h$

$$\text{ii. } T(n) = aT\left(\frac{n}{p}\right) + bn + d$$

- 1) Lineare con  $n$ , se  $a < p$
- 2)  $n \log n$ , se  $a = p$
- 3)  $n^{\log_p a}$ , se  $a > p$

## Complessità dei problemi

Studiare la complessità di un problema, ossia quello che un algoritmo risolve, è molto diverso dallo studiare la complessità di un algoritmo. Per poter dire che un problema ha complessità  $\Omega(g(n))$  basta trovare un qualsiasi algoritmo che lo risolva con  $\Omega(g(n))$  per poter matematica affermare che un problema è  $\Omega(g(n))$ , occorre dimostrare matematicamente che tutti i possibili algoritmi lo risolvano al meglio come  $\Omega(g(n))$

Per limitare superiormente un problema basta trovare almeno un algoritmo di tale complessità.

Per limitare inferiormente bisogna studiare ogni possibile soluzione (il problema, in linea teorica, potrebbe essere risolto in tempo costante, ma si può

sempre dimostrare il contrario). Quando la complessità di un algoritmo è pari al limite inferiore di complessità determinato da un problema, l'algoritmo si dice ottimo (in ordine di grandezza).

Tali limiti si individuano per dimensione dei dati ed eventi contabili.

- **Dimensione  $n$  dei dati:** se nel caso peggiore occorre analizzare tutti i dati allora  $\Omega(n)$  è un limite inferiore alla complessità del problema. Un esempio è la ricerca di un elemento o del massimo in un array.
- **Eventi contabili:** la ripetizione di un evento a un dato numero di volte è essenziale per la risoluzione di un problema. Esempio: la generazione di tutte le permutazioni di  $n$  oggetti.

# Hashing

venerdì 8 maggio 2020 18:27

## Tabelle Hash

Una **tabella hash** è una struttura dati usata per mettere in corrispondenza una data chiave con un dato valore.

Le chiavi e i valori possono appartenere a diversi tipi primitivi o strutturati. Ad esempio, è possibile associare l'età (intero) ad una persona utilizzando il nome (stringa). E' possibile utilizzare un array a patto che si associno prima gli indici interi alle persone e poi l'età agli indici.

Alcuni linguaggi implementano di base questa funzionalità con gli "array associativi".

In molte applicazioni è necessario che un insieme dinamico fornisca solamente le seguenti operazioni:

- *INSERT(key, value)*: inserisce un elemento nuovo, con un certo valore (unico) di un campo chiave.
- *SEARCH(key)*: determina se un elemento con un certo valore della chiave esiste; se esiste, lo restituisce.
- *DELETE(key)*: elimina l'elemento identificato dal campo chiave, se esiste.

Non è, ad esempio, necessario dover ordinare l'insieme dei dati o restituire l'elemento massimo, o il successore. Prelevo tutto grazie alla key.

Definiamo come:

$U$  - Universo di tutte le possibili chiavi

$K$  - Insieme delle chiavi effettivamente memorizzate

Se l'universo delle chiavi è piccolo e le chiavi sono intere, allora è sufficiente utilizzare una **tabella ad indirizzamento diretto**.

Una tabella ad indirizzamento diretto corrisponde al concetto di **array**. Ad ogni chiave è possibile corrispondere una posizione, o slot, nella tabella. Una tabella restituisce il dato memorizzato nello slot di posizione indicato tramite la chiave in tempo  $O(1)$ .

Se le chiavi non sono intere e/o l'universo delle possibili chiavi è molto grande, non è possibile o conveniente utilizzare il metodo delle tabelle a indirizzamento diretto, magari per causa della limitatezza delle risorse di memoria.

- Se  $|K|$  è circa  $|U|$ , non spremiamo troppo spazio e nel caso peggiore ci costa  $O(1)$ .
- Se  $|K| \ll |U|$  la soluzione non è praticabile.

Le tabelle hash permettono di mediare i requisiti di memoria ed efficienze nelle operazioni.

L'**hashing** permette di impiegare una quantità ragionevole sia di memoria che di tempo per operare un compromesso fra i casi precedenti.

Con il **metodo di indirizzamento diretto**, un elemento con chiave  $k$  non viene memorizzato in posizione  $k$ .

Con il **metodo hash**, un elemento con chiave  $k$  viene memorizzato in posizione  $h(k)$ .

## Funzione Hash

La funzione  $h()$  è detta **funzione hash**.

Il suo scopo è definire una corrispondenza fra  $U$  e una tabella hash  $T[0 .. M - 1]$  con  $M \ll |U|$ .

$h : U \rightarrow \{0, 1, \dots, M - 1\}$

$h$  è iniettiva, cioè due chiavi distinte possono produrre lo stesso valore hash; ogni qualvolta  $h(k_i) = h(k_j) \neq k_j$  si verifica una **collisione**.

Occorre dunque minimizzare il numero di collisioni ottimizzando la funzione, e gestire le collisioni residue.

Per risolvere il problema delle collisioni si impiegano principalmente due strategie:

- *Metodo di concatenazione*
- *Metodo di indirizzamento aperto*

## Metodo di concatenazione

L'idea è di mettere tutti gli elementi che collidono in una lista concatenata.

Le caratteristiche di una funzione hash rispettano il **criterio di uniformità semplice**: il valore hash di una chiave è uno dei valori in modo equiprobabile.

Un altro requisito è che l'hash dovrebbe utilizzare tutte le cifre della chiave per produrre un valore hash. Se le chiavi hanno dimensione troppo grande per poter essere rappresentati come interi, si rende **modulare la funzione hash**.

## Metodo di indirizzamento aperto

L'idea è di memorizzare tutti gli elementi nella tabella stessa.

In caso di collisione si memorizza l'elemento nella posizione successiva. Si genera quindi un nuovo valore hash fino a trovare una posizione vuota dove inserire l'elemento. Si estende la funzione hash perché generi non solo un valore hash ma una sequenza di scansione.

$h : U \times \{0, 1, \dots, M - 1\} \rightarrow \{0, 1, \dots, M - 1\}$

Con questo metodo, le chiavi devono rispettare la proprietà di uniformità delle funzioni hash: per ogni chiave, la sequenza di scansione generata da  $h$  dev'essere una qualunque delle permutazioni del suo dominio. Poiché è molto difficile implementare questo metodo, si procede per approssimazione.