# Appunti di Algoritmi e Strutture Dati

## Domande e risposte

### 1) Cosa è un algoritmo? Quali sono le sue caratteristiche?

Un algoritmo è una sequenza ordinata di passi o istruzioni finiti necessari per la risoluzione del problema.
Le caratteristiche dell'algoritmo sono:
→ Ogni istruzione deve concretamente realizzabile dall'esecutore, a cui è affidato l'algoritmo.
→ Le istruzioni che compongono l'algoritmo devono essere precise e non ambigue in modo che non lasciano dubbi nell'interpretazione da parte dell'esecutore.
→ Ogni istruzione, se pur concretamente eseguibile e non ambigua, deve avere una durata limitata nel tempo.
→ Ogni istruzione deve produrre un risultato osservabile.
→ Ogni istruzione deve produrre sempre il medesimo effetto se eseguita a partire dalle stesse condizioni iniziali (carattere deterministico).
→ Le istruzioni devono essere elementari, cioè non ulteriormente scomponibili rispetto alle capacità dell'esecutore.
Gli algoritmi devono seguire delle determinate regole e deve essere:
→ Finito: Un algoritmo deve essere composto da un numero finito di istruzioni e deve rappresentare un punto di inizio, dove incomincia la parte risolutiva, e un punto di fine, dove finisce l'algoritmo.
→ Esaustivo: deve essere completo ed esaustivo, ovvero che per tutti i casi che si possono verificare durante l'esecuzione deve essere indicata la soluzione da seguire.
→ Riproducibile: Ogni successiva esecuzione dello stesso algoritmo con i medesimi dati iniziali deve produrre sempre i medesimi risultati finali.

### 2) Definizione di Problema Computazionale

Dati un dominio di input e un dominio di output, un problema computazionale è rappresentato dalla relazione matematica tra un insieme di istanze (dati di input) e un insieme di soluzioni (dati di output). Permette di stabilire formalmente la relazione desiderata tra l'input e l'output di un algoritmo.

### 3) Cosa si intende per pseudocodice?

Lo pseudocodice è un linguaggio che rappresenta un algoritmo ad alto livello ed è considerato come "una via di mezzo" fra il linguaggio di programmazione e il

linguaggio naturale.

Esempio

```
...
conta = 1;
finchè (conta<10)
      se (voto<6)
            print("bocciato");
      altrimenti
            stampa("promosso");
      conta++;
....
```

## 4) Come possono essere classificati gli algoritmi?

Un algoritmo possiede due classificazioni:

1 - classificazione degli algoritmi basata sul numero di passi che possono eseguire contemporaneamente, in cui essi si classificano in:

→ algoritmi sequenziali: eseguono un solo passo alla volta;

→ algoritmi paralleli: possono eseguire più passi per volta, avvalendosi di un numero prefissato di esecutori;

2 - classificazione degli algoritmi basata sul modo in cui risolvono le scelte, in cui essi si classificano in:

→ algoritmi deterministici: ad ogni punto di scelta, intraprendono una sola via determinata in base ad un criterio prefissato (p.e. considerando il valore di un'espressione aritmetico-logica);

→ algoritmi probabilistici: ad ogni punto di scelta, intraprendono una sola via determinata a caso (p.e. lanciando una moneta o un dado);

→ algoritmi non deterministici: ad ogni punto di scelta, esplorano tutte le vie contemporaneamente (necessitano di un numero di esecutori generalmente non fissabile a priori).

Un buon algoritmo è fatto da:

→ correttezza: porta sempre a termine correttamente il suo compito

→ efficienza: compie la sua funzione nel minor numero possibile di passi.

Viene utilizzato il sistema di analisi asintotica per analizzarli indipendentemente dalla macchina sul quale vengono fatti funzionare.

## 5) Cos'è un problema e come può essere classificato?

Sia I un insieme di istanze ed S un insieme di soluzioni, un problema P è una relazione che ad ogni istanza I associa le relative soluzioni.

$P \subseteq I \times S$

Un problema P può essere espresso in tre diverse forme:

→ problema di decisione: richiede una risposta binaria rappresentante il soddisfacimento di qualche proprietà o l'esistenza di qualche entità, quindi S = {0,1};
→ problema di ricerca: richiede di trovare una generica soluzione in corrispondenza di ciascuna istanza dei dati di ingresso;
→ problema di ottimizzazione: richiede di trovare la soluzione ottima rispetto ad un criterio prefissato in corrispondenza di ciascuna istanza dei dati di ingresso.
→ I problemi possono essere classificati come i più facili o più difficili in funzione del tempo di esecuzione del migliore algoritmo che li risolve.
→ Un problema è detto decidibile se esiste un algoritmo che produce la corrispondente soluzione in tempo finito per ogni istanza dei dati di ingresso del problema.
→ Un problema è detto indecidibile se non esiste nessun algoritmo che produce la corrispondente soluzione in tempo finito per ogni istanza dei dati di ingresso del problema. In tal caso, sarà possibile calcolare la soluzione in tempo finito solo per alcune delle istanze dei dati di ingresso del problema, mentre per tutte le altre istanze non è possibile decidere quali siano le corrispondenti soluzioni.

## 6) Cosa si intende per struttura dati?

Una struttura dati è un insieme di dati logicamente correlati e opportunamente memorizzati, per i quali sono definiti degli operatori di costruzione, selezione e manipolazione.
Le strutture dati basata sulla loro occupazione di memoria:
→ strutture dati statiche: la quantità di memoria di cui esse necessitano è determinabile a priori (array);
→ strutture dati dinamiche: la quantità di memoria di cui esse necessitano varia a tempo d'esecuzione e può essere diversa da esecuzione a esecuzione (liste, alberi, grafi).

## 7) Correttezza di un algoritmo iterativo con l'invariante di ciclo

→ L'invariante di ciclo è un utile strumento che permette di dimostrare la correttezza di un ciclo. I componenti sono:
1 – Inizializzazione: la condizione è vera all'inizio del ciclo.
2 – Conservazione: la condizione è vera all'inizio del ciclo, allora rimane vera al termine (quindi prima della successiva iterazione);
3 – Inizializzazione: quando il ciclo termina, l'invariante deve rappresentare la

"correttezza" dell'algoritmo.

Esempio: Calcolo fattoriale

Dato il seguente algoritmo che calcola il fattoriale di un numero:

```
int fatt(int n) {
    int fattoriale = 1;

    for (int i=n;i>0;i--)
        fattoriale*=i;

    return fattoriale;
}
```

L'invariante di ciclo è:

```
(fattoriale = 1 x 2 x ... x n) and (1<=i<=n+1)
```

## 8) Che cosa si intende per complessità computazionale?

Dati due algoritmi A1 e A2, che risolvono lo stesso problema P, si dice che A1 è più efficiente di A2 se A1 viene eseguito più velocemente di A2 (si dice che A1 è meno complesso di A2).

La complessità computazionale di un problema P è la complessità in tempo dell'algoritmo più efficiente che risolve il problema.

## 9) Complessità degli algoritmi: notazione per esprimere la complessità asintotica

→ Gli algoritmi che risolvono lo stesso problema decidibile vengono confrontati sulla base della loro efficienza, misurata attraverso il loro tempo d'esecuzione.

→ Il tempo d'esecuzione di un algoritmo viene espresso come una funzione della dimensione dei dati di ingresso, di solito denotata con T(n).

→ La caratterizzazione della dimensione n dei dati di ingresso dipende dallo specifico problema. Può essere il numero di dati di ingresso oppure la quantità di memoria necessaria per contenere tali dati.

→ Il tempo d'esecuzione di un algoritmo può essere calcolato in tre diversi casi:

a) caso ottimo (caso migliore): è determinato dall'istanza dei dati di ingresso che minimizza il tempo d'esecuzione, quindi fornisce un limite inferiore alla quantità di risorse computazionali necessarie all'algoritmo;

b) caso pessimo (caso peggiore): è determinato dall'istanza dei dati di ingresso che massimizza il tempo d'esecuzione, quindi fornisce un limite superiore alla quantità di risorse computazionali necessarie all'algoritmo;

c) caso medio: è determinato dalla somma dei tempi d'esecuzione di tutte le

istanze dei dati di ingresso, con ogni addendo moltiplicato per la probabilità di occorrenza della relativa istanza dei dati di ingresso.
→ Le funzioni tempo d'esecuzione dei vari algoritmi che risolvono lo stesso problema decidibile vengono dunque confrontate considerando il loro andamento al crescere della dimensione n dei dati di ingresso. Ciò significa che, all'interno delle funzioni, è possibile ignorare le costanti moltiplicative e i termini non dominanti al crescere di n.

## 11) Complessità computazionale algoritmo iterativo

Il tempo di esecuzione di un'istruzione iterativa è il prodotto tra il massimo numero di volte in cui viene eseguito il ciclo e il tempo necessario per eseguire il corpo del ciclo e valutare la condizione.

Esempio 1: Somma degli elementi di un array

Dato il seguente metodo

```
int sumVect(int vet[]) {
    int sommaVettori = 0;

    for (int i=0;i<10;i++)
        sommaVettori+=vet[i];

    return sommaVettori;
}
```

Calcolo

```
int sommaVettori = 0;   → c1
for (int i=0;i<10;i++)
    sommaVettori+=vet[i];   → n*c2
return sommaVettori;   → c3
```

Ora per calcolare la complessità dell'algoritmo si procede nel seguente modo:

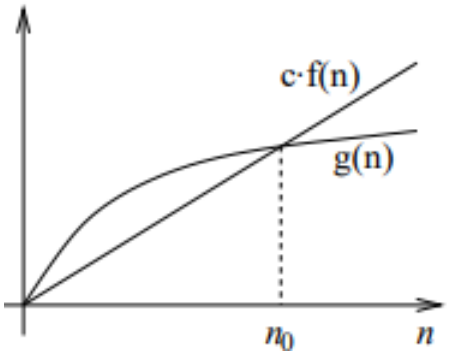
```
T(n) = c1 + n*c2 + c3
T(n) = n*c2 + c1 + c3
a = c2
b = c1 + c3
T(n) = a*n + b
```

## 12) Notazioni asintotiche: O, Ω e Θ

Esistono tre notazioni asintotiche:

→ notazione asintotica O: rappresenta il limite superiore asintotico;

$$g(n) = O(f(n)) \iff \exists c, n_0 > 0.\ \forall n \geq n_0.\ g(n) \leq c \cdot f(n)$$

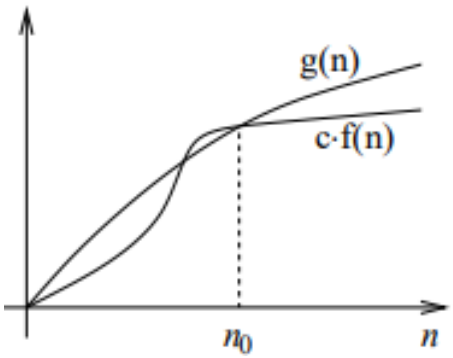



$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \quad \Leftrightarrow \quad f(n) = O(g(n))$$

→ notazione asintotica Ω: rappresenta il limite inferiore asintotico;

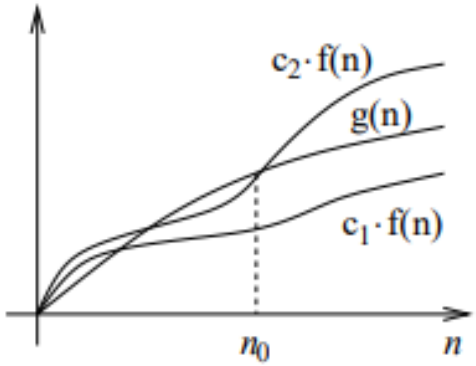
$$g(n) = \Omega(f(n)) \iff \exists c, n_0 > 0.\ \forall n \geq n_0.\ g(n) \geq c \cdot f(n)$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \quad \Leftrightarrow \quad f(n) = \Omega(g(n))$$




→ notazione asintotica Θ: rappresenta il limite asintotico stretto.

$$g(n) = \Theta(f(n)) \iff \exists c_1, c_2, n_0 > 0.\ \forall n \geq n_0.\ c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$




$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = k \quad \Leftrightarrow \quad f(n) = \Theta(g(n))$$

## 13) Scale asintoti

| Funzione | Descrizione |
|---|---|
| $T(n) = O(1)$ | complessità costante |
| $T(n) = O(\log n)$ | complessità logaritmica |
| $T(n) = O(n)$ | complessità lineare |
| $T(n) = O(n \cdot \log n)$ | complessità pseudo lineare |
| $T(n) = O(n^2)$ | complessità quadratica |
| $T(n) = O(n^3)$ | complessità cubica |
| $T(n) = O(n^k), k > 0$ | complessità polinomiale |
| $T(n) = O(\alpha^n), \alpha > 1$ | complessità esponenziale |

## 14) Come calcolare la complessità di un algoritmo?

Le regole per calcolare la complessità di un algoritmo:

1 – la complessità di un'istruzione di assegnamento, di confronto o di lettura/scrittura è assunta essere costante:

$$T(n) = \begin{cases} 1 = O(1) & \text{se priva di chiamate di funzioni} \\ T'(n) + 1 = O(f'(n)) & \text{se contiene chiamate di funzioni eseguibili in } T'(n) = O(f'(n)) \end{cases}$$

2 – il tempo d'esecuzione di una sequenza di una sequenza di istruzioni è pari alla somma del tempo di esecuzione delle singole istruzioni:

$$T(n) = \sum_{i=1}^{s} T_i(n) = O(\max\{f_i(n) \mid 1 \leq i \leq s\})$$

3 – il tempo di esecuzione di un'istruzione di selezione è dato dal tempo necessario alla valutazione della condizione più il tempo necessario per eseguire le istruzioni del ramo più complesso:

il tempo di valutazione dell'espressione $\beta$ è $T_0(n) = O(f_0(n))$;
il tempo d'esecuzione dell'istruzione $S_1$ è $T_1(n) = O(f_1(n))$;
il tempo d'esecuzione dell'istruzione $S_2$ è $T_2(n) = O(f_2(n))$;

$$T(n) = T_0(n) + op(T_1(n), T_2(n)) = O(\max\{f_0(n), op(f_1(n), f_2(n))\})$$

4 – il tempo di esecuzione di un'istruzione iterativa è il prodotto tra il massimo numero di volte in cui viene eseguito il ciclo e il tempo necessario per eseguire il corpo del ciclo e valutare la condizione:

il numero di iterazioni è $i(n) = O(f(n))$;
il tempo di valutazione dell'espressione $\beta$ è $T_1(n) = O(f_1(n))$;
il tempo d'esecuzione dell'istruzione $S$ è $T_2(n) = O(f_2(n))$;

$$T(n) = i(n) \cdot (T_1(n) + T_2(n)) + T_1(n) = O(f(n) \cdot \max\{f_1(n), f_2(n)\})$$

$$T(n) = \sum_{j=1}^{i(n)} (T_1(n,j) + T_2(n,j)) + T_1(n, i(n)+1)$$

5 – il tempo di esecuzione di un sottoprogramma è uguale al tempo di esecuzione del sottoprogramma.

## 15) Array: definizione e caratteristiche

→ Un array è una struttura dati statica e omogenea, cioè i cui elementi non variano di numero a tempo d'esecuzione e sono tutti dello stesso tipo.
→ Gli elementi di un array sono memorizzati consecutivamente e individuati attraverso la loro posizione. L'indirizzo di ciascuno di essi è determinato dalla somma dell'indirizzo del primo elemento dell'array e dell'indice che individua l'elemento in questione all'interno dell'array.
→ Poiché l'accesso a ciascun elemento di un array è diretto, l'operazione di lettura o modifica del valore di un elemento di un array ha complessità asintotica $O(1)$ rispetto al numero di elementi dell'array.

## 16) Problema della visita

Dato un array e la dimensione n, si vuole attraversare tutti i suoi elementi esattamente una volta.

Algoritmo

```
void visitaArray(int a[], int n) {
    for (int i=0;i<n;i++) {
        cout<<a[i];
        if (i<(n-1))
            cout<<", ";
    }
}
```

Complessità

$$T(n) = 1 + n \cdot (1 + d + 1) + 1 = (d+2) \cdot n + 2 = O(n)$$

## 17) Problema della ricerca: ricerca lineare array

Gli elementi dell'array vengono attraversati uno dopo l'altro nell'ordine in cui sono memorizzati finché il valore cercato non viene trovato e la fine dell'array non viene raggiunta.

Algoritmo:

```
int ricercaLineareElemento(int a[], int n, int searchEl) {
    for (int i=0;i<n;i++) {
        if (a[i] == searchEl)
            return i;
    }
    return -1;
}
```

Complessità

1 – Caso migliore

$$T(n) = 1 + 1 = 2 = O(1)$$

2 – Caso peggiore

$$T(n) = 1 + n \cdot (1 + 1) + 1 = 2 \cdot n + 2 = O(n)$$

## 18) Problema della ricerca: ricerca binaria per array ordinati

L'idea della ricerca binaria è di confrontare il valore cercato col valore dell'elemento che sta nella posizione di mezzo dell'array (supponendo che questo sia ordinato). Se i due valori sono diversi, si continua la ricerca solo nella metà dell'array che sta a sinistra (risp. destra) dell'elemento considerato se il suo valore è maggiore (risp. minore) di quello cercato.

Complessità

1 – Caso migliore

Il caso migliore si verifica quando il valore cercato è presente nell'elemento che sta nella posizione di mezzo dell'array. In questo caso la complessità asintotica è $O(1)$ come per il caso ottimo dell'algoritmo di ricerca lineare.

2 – Caso peggiore

Il caso peggiore si verifica quando lo spazio di ricerca viene ripetutamente diviso a metà fino a restare con un unico elemento da confrontare con il valore cercato. Dato k il numero di iterazioni, il quale coincide con il numero di dimezzamenti dello spazio di ricerca, nel caso pessimo $n/2^k = 1$ da cui $k = \log_2 n$. La complessità computazionale è

$$T(n) = 1 + k \cdot (1 + 1 + 1 + 1) + 1 = 4 \cdot k + 2 = 4 \cdot \log_2 n + 2 = O(\log n)$$

## 19) Problema dell'ordinamento

→ Gli algoritmi di ordinamento consentono di solito una rapida ricerca di valori all'interno di un array attraverso le chiavi degli elementi.

→ È possibile dimostrare che il problema dell'ordinamento non può essere risolto con un algoritmo di complessità asintotica inferiore a quella pseudo lineare.

→ Gli algoritmi di ordinamento basati su confronti ordinano elementi confrontando i vari elementi di input. L'albero di decisione rappresenta il confronto eseguito da un algoritmo di ordinamento quando opera su un input di una data dimensione.

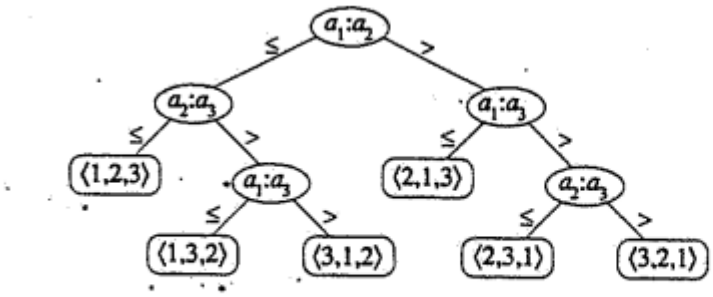



Figura 9.1 *L'albero di decisione per l'insertion sort che opera su tre elementi. Vi sono* 3! = 6 *possibili permutazioni degli elementi di input, quindi l'albero di decisione deve avere almeno* 6 *foglie.*

→ Ogni algoritmo che ordina un array di n elementi, la complessità computazionale è:

$$T(n) = \Omega(n \cdot \log n)$$

→ La dimostrazione è riportata qui sotto:

Poiché un albero binario di altezza $h$ non ha più di $2^h$ foglie, si ha

$n! \leq 2^h$ ,

che, passando ai logaritmi, implica

$h \geq \lg(n!)$ ,

poiché la funzione lg è monotona crescente. Dall'approssimazione di Stirling (2.11), si ha

$n! > \left(\frac{n}{e}\right)^n$ ,

dove $e = 2.71828\ldots$ è la base dei logaritmi naturali; da cui

$$\begin{aligned} h &\geq \lg\left(\frac{n}{e}\right)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) . \end{aligned}$$

■

***Corollario 9.2***

Il merge sort e lo heapsort sono ordinamenti per confronti asintoticamente ottimi.

## 20) Algoritmo: Insertsort

→ L'algoritmo di Insertsort è un algoritmo di ordinamento iterativo che al generico passo i si ha l'array di viso in una sequenza di destinazione a[0], …, a[i-1] già ordinata e una sequenza di origine a[i], …, a[n-1] ancora da ordinare.

→ L'obiettivo è di inserire il valore contenuto in a[i] al posto giusto nella sequenza di destinazione facendolo scivolare a ritroso, in modo da ridurre la sequenza di origine di un elemento.

Algoritmo

```
void insertSort(int ar[], int n) {
    int valore_ins;
    int j;

    for (int i=0;i<n;i++) {
        for (valore_ins=ar[i],j=i-1;((j>=0) && (a[j]>valore_ins)); j--)
            a[j+1] = a[j];
        if (j+1 != i)
            a[j+1] = valore_ins;
    }
}
```

## Complessità

Caso migliore: si verifica quando l'array è già ordinato.

$$T(n) = 1 + (n-1) \cdot (1+1+0 \cdot (1+1+1)+1+1+1)+1 = 5 \cdot n - 3 = O(n)$$

Caso peggiore: si verifica quando l'array è inversamente ordinato.

In questo caso il numero di iterazioni nell'istruzione for interna è proporzionale ad i, quindi la complessità é:

$$T(n) = 1 + \sum_{i=1}^{n-1} (1+1+i \cdot (1+1+1)+1+1+1+1)+1$$

$$T(n) = 2 + (n-1) \cdot 6 + 3 \cdot \sum_{i=1}^{n-1} i$$

$$T(n) = 6 \cdot n - 4 + 3 \cdot \frac{(n-1) \cdot n}{2}$$

$$T(n) = 1.5 \cdot n^2 + 4.5 \cdot n - 4$$

$$T(n) = O(n^2)$$

## Simulazione algoritmo

Dato il seguente array

| 42 | 38 | 11 | 75 | 99 | 23 | 84 | 67 |
|---|---|---|---|---|---|---|---|

Ordinamento

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 42 | 38 | 11 | 75 | 99 | 23 | 84 | 67 | (1 scambio) |
| 38 | 42 | 11 | 75 | 99 | 23 | 84 | 67 | (2 scambi) |
| 11 | 38 | 42 | 75 | 99 | 23 | 84 | 67 | (0 scambi) |
| 11 | 38 | 42 | 75 | 99 | 23 | 84 | 67 | (0 scambi) |
| 11 | 38 | 42 | 75 | 99 | 23 | 84 | 67 | (4 scambi) |
| 11 | 23 | 38 | 42 | 75 | 99 | 84 | 67 | (1 scambio) |
| 11 | 23 | 38 | 42 | 75 | 84 | 99 | 67 | (3 scambi) |

| 11 | 23 | 38 | 42 | 67 | 75 | 84 | 99 |
|---|---|---|---|---|---|---|---|

Fine

## 21) Algoritmo: Selesort

→ L'algoritmo di Selesort è un algoritmo di ordinamento iterativo che, come Insertsort, al generico passo i vede l'array diviso in una sequenza di destinazione a[0], . . . , a[i – 1] già ordinata e una sequenza di origine a[i], . . ., a[n - 1] ancora da ordinare.

→ L'obiettivo è di selezionare l'elemento della sequenza di origine che contiene il valore minimo e di scambiare tale valore con il valore contenuto in a[i], in modo da ridurre la sequenza di origine di un elemento

### Algoritmo

```
void selectSort(int ar[], int n) {
    int valore_min;
    int indice_valore_min;
    int i;
    int j;

    for (i = 0;i<n-1;i++) {
        indice_valore_min = i;
        valore_min = a[i];
        j=i+1;
        for (;(j<n);j++) {
            if (a[j]<valore_min) {
                valore_min = a[j];
                indice_valore_min = j;
            }
        }
        if (indice_valore_min!=i) {
            a[indice_valore_min] = a[i];
            a[i] = valore_min;
        }
    }
}
```

### Caratteristiche

→ Selesort è stabile in quanto nel confronto tra a[j] e valore min viene usato < al posto di <=.

→ La complessità asintotica non dipende dalla disposizione iniziale dei valori negli elementi dell'array.

→ Si denota con h ∈ {1, 3} il tempo d'esecuzione di una delle due istruzioni if.

### Complessità

$$
\begin{aligned}
T(n) &= 1 + \sum_{i=1}^{n-1} (1+1+(n-i)\cdot(1+h+1)+1+h+1)+1 \\
&= 2 + (n-1)\cdot(h+4) + (n-1)\cdot(h+2)\cdot n - (h+2)\cdot\sum_{i=1}^{n-1} i \\
&= (h+2)\cdot n^2 + 2\cdot n - (h+2) - (h+2)\cdot\tfrac{(n-1)\cdot n}{2} \\
&= (0.5\cdot h+1)\cdot n^2 + (0.5\cdot h+3)\cdot n - (h+2) = O(n^2)
\end{aligned}
$$

Simulazione algoritmo

Dato il seguente array

| 42 | 38 | 11 | 75 | 99 | 23 | 84 | 67 |
|---|---|---|---|---|---|---|---|

Procedimento

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 42 | 38 | 11 | 75 | 99 | 23 | 84 | 67 | (1 scambio) |
| 11 | 38 | 42 | 75 | 99 | 23 | 84 | 67 | (1 scambio) |
| 11 | 23 | 42 | 75 | 99 | 38 | 84 | 67 | (1 scambio) |
| 11 | 23 | 38 | 75 | 99 | 42 | 84 | 67 | (1 scambio) |
| 11 | 23 | 38 | 42 | 99 | 75 | 84 | 67 | (1 scambio) |
| 11 | 23 | 38 | 42 | 67 | 75 | 84 | 99 | (0 scambi) |
| 11 | 23 | 38 | 42 | 67 | 75 | 84 | 99 | (0 scambi) |
| 11 | 23 | 38 | 42 | 67 | 75 | 84 | 99 | |

## 22) Ricorsione

→ I metodi ricorsivi sono metodi che contengono al loro interno chiamate a sé stessi. Per risolvere un problema, il metodo ricorsivo scompone i problemi in sotto problemi (divide et impera) e richiama sé stesso su di essi per poi, alla fine, mettere insieme i vari risultati e giungere alla soluzione.

→ Ogni metodo ricorsivo è riconducibile ad una soluzione iterativa.

→ A volte però una soluzione ricorsiva risulta decisamente più adeguata e per questo motivo che Wirth definì quando algoritmo risulta essere intrinsecamente ricorsivo: un algoritmo si dice intrinsecamente ricorsivo quando la sua versione iterativa (sempre esistente) necessita della manipolazione esplicita di uno stack di chiamate, operazione che spesso rende l'essenza del programma estremamente difficile da capire.

Esempio 1:

1) l'algoritmo di ricerca dicotomica è un algoritmo non intrinsecamente ricorsivo;

2) l'algoritmo della stampa in order è un algoritmo intrinsecamente ricorsivo;

Si ha l'obiettivo di effettuare una ricerca dicotomica e una stampa in order.

La versione ricorsiva di entrambi gli algoritmi è:

→ ricerca dicotomica: si confronta l'elemento x da cercare con la radice: se l'albero è vuoto, la ricerca termina, altrimenti se x<radice si effettua un richiamo ricorsivo sul sotto albero sinistro, altrimenti sul sotto albero destro;

→ stampa in order: controlla, partendo dalla radice, se esiste su un sotto albero sinistro: si chiama ricorsivamente a sinistra, si stampa la radice e, se esiste, si richiama il sotto albero destro. Esso stampa tutti gli elementi in ordine crescente e, per questo motivo, che la stampa viene detta in order.
La versione iterativa di entrambi gli algoritmi è:
→ ricerca dicotomica: ogni discesa lungo un sotto albero l'altro può essere trascurato;
→ stampa in order: ogni discesa nel sottoalbero sinistro si deve memorizzare il fatto che il sottoalbero destro non è ancora stato scandito. Occorre quindi utilizzare una struttura LIFO (Last In Fist Out), ovvero lo STACK.

## 23) Complessità algoritmi ricorsivi

Un algoritmo ricorsivo ha la seguente struttura:

```
soluzione risolvi(problema p) {
      soluzione s1,s2,s3,...,sn;
      if (p_is_semplice)
            <calcola_soluzione_s_di_p>
      else {
            <dividi p in p1, p2, p3,..., pn>
            s1 = risolvi(p1);
            s2 = risolvi(p2);
            s3 = risolvi(p3);
            ....
            sn = risolvi(pn);
            <combina s1, s2, s3, ..., sn per ottenere p>
      }
      return(s);
}
```

→ Il tempo di esecuzione di un algoritmo ricorsivo viene risolto attraverso una relazione di ricorrenza le cui incognite costituiscono una successione di numeri interi positivi corrispondenti ai valori di T(n).
→ Una relazione di ricorrenza lineare di ordine k è, a coefficienti costanti e omogenea ha forma:

$$\begin{cases} x_n = a_1 \cdot x_{n-1} + a_2 \cdot x_{n-2} + \ldots + a_k \cdot x_{n-k} & \text{per } n \geq k \\ x_i = d_i & \text{per } 0 \leq i \leq k-1 \end{cases}$$

in cui l'n-esima incognita $x_n$ è espressa come combinazione lineare delle k incognite che la precedono.
Ci sono due casi:
1 – se k = 1, la soluzione è:

$$x_n = \begin{cases} d_0 = O(1) & \text{se } a_1 = 1 \\ d_0 \cdot {a_1}^n = O({a_1}^n) & \text{se } a_1 > 1 \end{cases}$$

2 – se k > 1, si procede nel seguente modo:

a) assumo che esistano delle soluzioni in forma chiusa del tipo $x_n = c \cdot z^n$ con c != 0 != z;

b) con la sostituzione si ha che :

$c \cdot z^{n-k} \cdot (z^k - \sum_{i=1}^{k} a_i \cdot z^{k-i}) = 0$ e quindi $z^k - \sum_{i=1}^{k} a_i \cdot z^{k-i} = 0$ essendo $c \neq 0 \neq z$.

La relazione è quindi:

$$x_n = \sum_{j=1}^{k} c_j \cdot {z_j}^n = O(\max\{{z_j}^n \mid 1 \leq j \leq k\})$$

→ Una relazione di ricorrenza lineare, di ordine costante k, a coefficienti costanti e non omogenea ha forma:

$$\begin{cases} x_n = a_1 \cdot x_{n-1} + a_2 \cdot x_{n-2} + \ldots + a_k \cdot x_{n-k} + h & \text{per } n \geq k \\ x_i = d_i & \text{per } 0 \leq i \leq k-1 \end{cases}$$

Ponendo

$y_n = x_n + h/(\sum_{j=1}^{k} a_j - 1)$

si ha che:

$x_n = O(y_n)$

→ Una relazione di ricorrenza lineare, di ordine non costante ha forma:

– costante, ha forma:

$$\begin{cases} x_n = a \cdot x_{n/b} + c & \text{per } n > 1 \\ x_1 = d \end{cases}$$

e soluzione:

$$x_n = \begin{cases} O(\log_b n) & \text{se } a = 1 \\ O(n^{\log_b a}) & \text{se } a > 1 \end{cases}$$

– lineare, ha forma:

$$\begin{cases} x_n = a \cdot x_{n/b} + (c_1 \cdot n + c_2) & \text{per } n > 1 \\ x_1 = d \end{cases}$$

e soluzione:

$$x_n = \begin{cases} O(n) & \text{se } a < b \\ O(n \cdot \log_b n) & \text{se } a = b \\ O(n^{\log_b a}) & \text{se } a > b \end{cases}$$

– polinomiale, ha forma:

$$\begin{cases} x_n = a \cdot x_{n/b} + (c_p \cdot n^p + c_{p-1} \cdot n^{p-1} + \ldots + c_1 \cdot n + c_0) & \text{per } n > 1 \\ x_1 = d \end{cases}$$

e soluzione:

$$x_n = O(n^{\log_b a}) \quad \text{se } a > b^p$$

## 24) Algoritmo: Mergesort

Mergesort è un algoritmo di ordinamento ricorsivo che Data una porzione di un array, la divide in due parti della stessa dimensione a cui applicare l'algoritmo stesso, poi fonde ordinatamente le due parti.

## Algoritmo

```
void mergeSort(int ar[], int sx, int dx, int n) {
    int ar3 [n];
    if (sx<dx) {
        middle = (sx+dx)/2;
        mergeSort(ar,sx,middle,n);
        mergeSort(ar,middle,dx,n);
        merge(ar,sx,middle,dx);
    }
}

void merge(int V1[], int sin, int mid, int dex) {
    int V2[dex-sin+1];
    int i = sin;
    int j = mid + 1;
    int k = 0;

    while (i<=mid && j<=dex) {
        if (V1[i]<=V1[j]) {
            V2[k] = V1[i];
            i++;
        }
        else {
            V2[k] = V1[j];
            j++;
        }
        k++;
    }

    while(i<=mid) {
        V2[k] = V1[i];
        i++;
        k++;
    }

    while(j<=dex) {
        V2[k] = V1[j];
        j++;
        k++;
    }

    for (int k=0;k<sizeof(V2);k++)
        V1[sin+k-1] = V2[k];
}
```

## Caratteristiche

→ Mergesort è stabile in quanto nel confronto tra a[i] e a[j] effettuato nella

fusione ordinata viene usato <= anziché <.

→ Mergesort non opera sul posto in quanto nella fusione ordinata viene usato un array di appoggio (b) il cui numero di elementi `e proporzionale al numero di elementi dell'array da ordinare.

→ Il tempo di esecuzione di questo algoritmo è:

$$\begin{cases} T(n) = 2 \cdot T(n/2) + (c_1 \cdot n + c_2) & \text{per } n > 1 \\ T(1) = 1 \end{cases}$$

$$T(n) = O(n \cdot \log n)$$

Simulazione Input

| 42 | 38 | 11 | 75 | 99 | 23 | 84 | 67 |
|---|---|---|---|---|---|---|---|

| 42 | 38 | 11 | 75 |
|---|---|---|---|

| 42 | 38 |
|---|---|

| 42 | | 38 |
|---|---|---|

| 38 | 42 |
|---|---|

| 11 | 75 |
|---|---|

| 11 | | 75 |
|---|---|---|

| 11 | 75 |
|---|---|

| 11 | 38 | 42 | 75 |
|---|---|---|---|

| 99 | 23 | 84 | 67 |
|---|---|---|---|

| 99 | 23 |
|---|---|

| 23 | | 99 |
|---|---|---|

| 23 | 99 |
|---|---|

| 84 | 67 |
|---|---|

| 84 | | 67 |
|---|---|---|

| 67 | 84 |
|---|---|

| 23 | 67 | 84 | 99 |
|---|---|---|---|

| 11 | 23 | 38 | 42 | 67 | 75 | 84 | 99 |
|---|---|---|---|---|---|---|---|

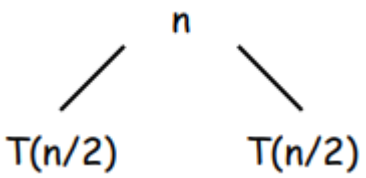
## 25) Albero di ricorsione

Un albero di ricorsione è un albero in cui ogni nodo rappresenta il costo di un sottoproblema da qualche parte nell'insieme delle chiamate ricorsive
Considerando la ricorrenza:

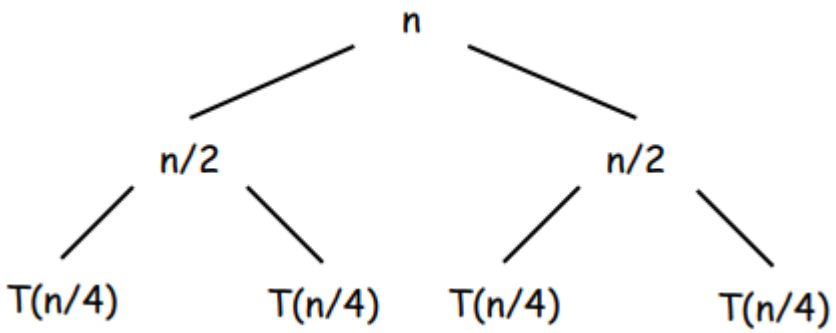
$$T(n) = \begin{cases} n + 2T(n/2) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

Il procedimento utilizzato per costruire l'albero delle ricorrenze per T(n) è il seguente

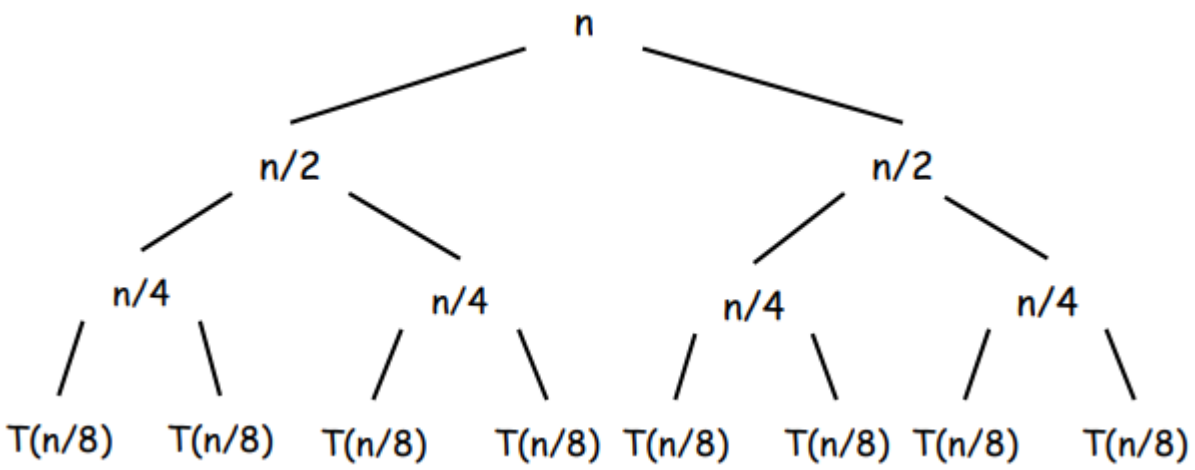
1)



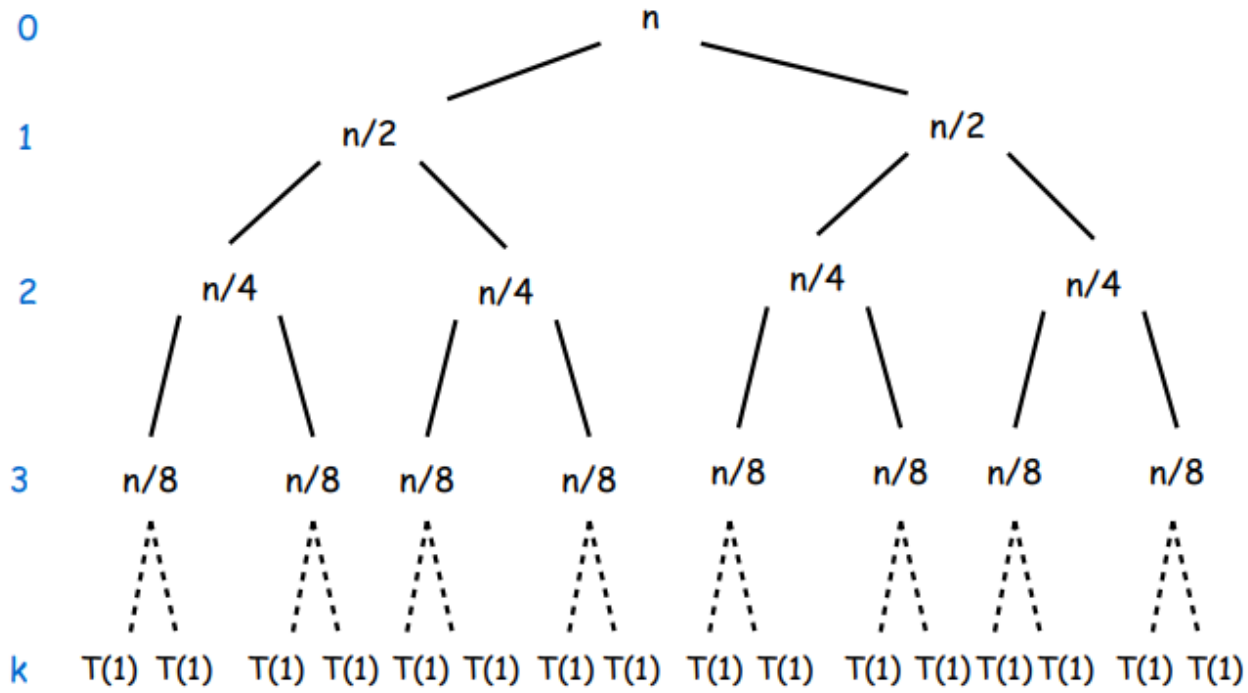

2)




3)



4)




Caratteristiche:

Sia i un dato livello dell'albero di ricorsione.

→ Quanti nodi ci sono al livello i?

Vi sono esattamente $2^i$ nodi;

→ Quale è il costo di un nodo al livello i?

ogni nodo al livello i costa $n/2^i$ nodi. Il costo complessivo dei nodi al livello i è:

$$\#nodi(i) \times costo\_nodo(i) = 2^i \times n/2^i = n$$

→ Quale è il costo complessivo della chiamata = T(n)?

$$T(n) = \sum_{j=0}^{\#livelli} costo\_livello(j) = \sum_{j=0}^{\#livelli} n = n\,(\#livelli + 1)$$

→ Quale è il numero di livelli?

Con T(1) si ha:

$$1 = n/2^k$$

$$n/2^k = 1$$

$$2^k = n$$

$$\#livelli = k = \log_2 n$$

Quindi

$$T(n) = n\,(\log_2 n + 1) = \Theta(n \log_2 n)$$

## 26) Teorema dell'esperto e divide et impera

Permette di analizzare algoritmi basati sulla tecnica del divide et impera:

→ divisione il problema (di dimensione n) in a sotto-problemi di dimensione n/b;

→ risoluzione sotto – problemi ricorsivamente;

→ ricombinazione soluzioni.

La relazione di ricorrenza è data da:

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

Più precisamente:

1) Se $f(n) = O(n^{\log_b a - \varepsilon})$ per qualche costante $\varepsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$;

2) Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \log_2 n)$;

3) Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per qualche costante $\varepsilon > 0$ e se, per qualche costante $c < 1$, $af(n/b) \leq cf(n)$, allora $T(n) = \Theta(f(n))$

Esempi

1)

$$T(n) = n + 2T(n/2)$$

$a = b = 2$, $\log_b a = 1$
$f(n) = \Theta(n) = \Theta(n^{\log_b a})$
(caso 2 del teorema master)

$$\Downarrow$$

$$T(n) = \Theta(n^{\log_b a} \log_2 n) = \Theta(n \log_2 n)$$

2)

$$T(n) = c + 9T(n/3)$$

$a = 9$, $b = 3$, $\log_b a = \log_3 9 = 2$
$f(n) = c = O(n) = O(n^{\log_b a - \varepsilon}) = O(n^{2-\varepsilon})$ con $\varepsilon = 1$
(caso 1 del teorema master)

$$\Downarrow$$

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

3)

$$T(n) = n + 3T(n/9)$$

$a = 3,\ b = 9,\ \log_b a = \log_9 3 = 1/2\ (3 = \sqrt{9} = 9^{1/2})$
$f(n) = n = \Omega(n) = \Omega(n^{\log_9 3+\varepsilon})$ con $\varepsilon = 1/2$
inoltre,
$af(n/b) = 3f(n/9) = 3(n/9) = 1/3n \leq cf(n)$, per $c = 1/3$
(caso 3 del teorema master)

$$\Downarrow$$

$$T(n) = \Theta(f(n)) = \Theta(n)$$