COSTO COMPUTAZIONALE

COMPETIZIONE FRA ALGORITMI PER IL PREZZO PIÙ BASSO

La leggenda di Sessa inventore del gioco degli scacchi ...Sessa: «mio buon sovrano vorrei come ricompensa che tu mi facessi dare i chicchi di grano necessari per riempire le 64 caselle del mio scacchiere, in ogni casella vorrei si mettesse un numero di chicchi doppio rispetto a quelli della precedente» Sovrano: «Saresti tanto sciocco da formulare una richiesta così modesta?»

GESTIONE DELL'INFORMAZIONE NEI PROBLEMI DELLA PRIMA UNITÀ

CONFRONTO TRA ALGORITMI:

1. Confrontare gli algoritmi per trovare m e M
2. Il problema della ricerca del I e II sembrava analogo al precedente, invece si è riusciti a trovare un metodo migliore.

I miglioramenti per ora sono intesi come **miglioramenti in termini di confronti**

COSA FA LA DIFFERENZA?

La gestione delle informazioni

Vediamo come.

Prima osservazione:

Si potrebbe dire che un confronto non è detto che contenga solo un elemento di informazione. L'informazione che si sfrutta una volta effettuato il confronto, dipende dal problema che si deve risolvere.

1) nel primo problema il primo confronto contiene un solo elemento di informazione: A > B
2) nel secondo problema sempre il primo confronto contiene due elementi di informazione: A > B e B < A
3) nel terzo caso contiene anche una terza informazione: A > B e B < A e B segue immediatamente A

Il significato di quando detto sopra è il seguente

1) nel problema della ricerca del solo Massimo o del solo minimo, il confronto contiene un solo elemento di informazione che ci è utile: A > B, cioè B non può essere il Massimo.
2) nel secondo problema, ricerca del Massimo e del minimo, contiene invece due elementi di informazione utili: A > B e B < A, cioè B non può essere il Massimo e A non può essere il minimo.
3) nel terzo problema, ricerca del primo e secondo, un confronto contiene una terza informazione: B ha perso solo con A, sulla base di quel confronto non si può ancora escludere che B possa essere secondo.

Riassumendo le informazioni contenute in un unico confronto sono le seguenti:

A>B (il soggetto è A)

A potrebbe essere massimo, A non può essere minimo;

B<A (il soggetto è B)

B non può essere massimo, B potrebbe essere minimo, B potrebbe essere secondo;

Proprio le considerazioni sopra esposte porteranno a migliorare l'algoritmo per i tre casi distinti.

Per convincercene esaminiamo gli algoritmi mettendo in evidenza a cosa sono dovuti i miglioramenti ottenuti

---

Nel primo problema una volta fatto il primo confronto si può eliminare il perdente, c'è infatti uno che perde.

Negli altri due una volta fatto il primo confronto non posso eliminare nessun elemento, cioè me ne devo ricordare, non ci sono infatti perdenti.

Infatti se elimino uno dei due, se non ricordo in qualche modo sia le informazioni su A che le informazioni su B, quindi sia A > B sia B < A, spreco informazione.

Esaminiamo come si procede per trovare il M e il m nel modo più ingenuo.

Trovo il massimo del primo insieme, poi tolgo il massimo trovato dall'insieme e trovo il minimo tra gli elementi rimasti.

Del primo confronto ho utilizzato solo una delle due possibili informazioni, perché ho utilizzato l'algoritmo per trovare il massimo: n-1 + n-2, ottengo così 2n-3 confronti.

Se invece utilizzo il metodo che fa più economia di confronti, per n superiore a 3, posso notare che utilizzo tutte e due le informazioni implicite.

Facciamo l'esempio di 5 oggetti

AB CD E

Se dal confronto ottengo A > B, li metto in ordine M e m

AB

Se dal confronto ottengo C > D, metto anche questi due in ordine

CD

Poi sfrutto sia la conoscenza dei massimi che dei minimi per continuare

---

È interessante esaminare come mai un tale metodo efficace nel caso della ricerca del M e del m, non funzioni altrettanto bene per la ricerca del "I e II", come mai si fanno più passaggi del necessario?

C'e' una informazione che non viene sfruttata utilizzando il metodo per la ricerca del M e del m.
Del primo confronto si deve utilizzare anche l'importante acquisizione che se A > B, non solo provvisoriamente A è massimo e B è minimo, ma anche A e B sono uno successivo dell'altro, ed è proprio questo che farà la differenza.

È sempre vero che ottengo un algoritmo con 2n-3 passi se trovo il massimo dell'insieme e poi butto via il massimo trovato e trovo il massimo dell'insieme meno un oggetto (n-1+n-2), infatti come per il caso della ricerca del M e del m, non sfrutto una parte dell'informazione contenuta in un confronto.

Riflettendo sul metodo utilizzato per l'algoritmo più veloce, si può notare che è proprio avere la possibilità di riconsiderare tutti quelli che hanno perso con il massimo che rende l'algoritmo ottimale, perché è proprio il momento di non perdere di vista l'informazione che uno è successivo dell'altro. Questo è importante solo se uno dei due è il massimo, altrimenti si può buttare via uno dei due elementi senza problemi, non ci porta alcuna informazione, è proprio sconfitto.
Invece avere perso con il massimo non è una vera e propria sconfitta: si può aspirare al secondo posto.
Come si era detto il contenuto di un confronto ha tre aspetti: A > B, B < A, A precede immediatamente B, nel senso che abbiamo chiarito.

Alcune considerazioni sulla ricerca del *m* e *M* e del *I* e *II* di un insieme

Supponiamo di avere un insieme di 5 oggetti, in un vettore, e di dover trovare il I e II dell'insieme
Li confrontiamo a due a due e mettiamo il maggiore a destra e il minore a sinistra, li scambiamo di posto.

6 2 5 3 7	$\mathbf{A_1\ \ A_2\ \ A_3\ \ A_4\ \ldots\ A_n}$
6>2	$\mathbf{A_1<A_2}$
2 6 5 7 6	$\mathbf{A_2\ \ A_1\ \ A_3\ \ A_4\ \ldots\ A_n}$
6>5	$\mathbf{A_1>A_3}$ ...
2 5 6 3 7	
6>3	
2 5 3 6 7	
6<7	
2 5 3 6 7	

In questo caso

7 è il massimo e 6 il secondo perché 6 ha vinto con tutti gli altri escluso il 7

Con n-1 passi abbiamo trovato il massimo e il secondo CASO MIGLIORE

Se però il massimo si fosse trovato nella prima posizione si avrebbe avuto il caso peggiore.

7 6 2 5 3 $\mathbf{A_1\ A_2\ A_3\ A_4\ \ldots\ A_n}$

si avrebbe avuto la seguente sequenza

6 7 2 5 3 $\mathbf{A_2\ A_1\ A_3\ A_4\ \ldots\ A_n}$

6 2 7 5 3

6 2 5 7 3

6 2 5 3 7 $\mathbf{A_1\ A_3\ \ldots\ldots\ldots\ A_{max}}$

con n-1 passi si sarebbe trovato il massimo e non avremmo nessun altra informazione, perché tutti gli altri elementi hanno perso solo con il massimo.

Per trovare il secondo dobbiamo trovare il massimo dei quattro oggetti rimasti.

n-1 + n-2 = 2n-3 CASO PEGGIORE

Devo distribuire meglio i confronti, in modo tale da non sprecare informazioni

Esaminiamo un metodo come quello che segue.

Mettiamo gli elementi dell'insieme considerato in un **vettore**.

$$\mathbf{A_1\ A_2\ A_3\ A_4\ A_5\ A_6\ A_7\ \ldots\ A_m\ \ldots\ A_{m+1}\ \ldots\ \ldots\ A_{n-1}\ A_n.}$$

Si considera l'elemento centrale $\mathbf{A_m}$ e si confrontano un elemento a sinistra e uno a destra nel modo indicato dalle frecce, a distanza **m** quindi.

Il vincitore lo si mette a sinistra e il perdente a destra, si può concludere che il massimo deve per forza stare a sinistra e il minimo deve per forza stare a destra di $\mathbf{A_m}$.

Deve essere così, infatti gli elementi a sinistra hanno vinto almeno con un elemento e gli elementi a destra hanno perso almeno con un elemento.

Aver perso almeno con un elemento significa non poter essere il massimo, e aver vinto almeno con un elemento significa non poter essere il minimo.

Questo metodo è molto utile per trovare il Massimo e il minimo, ma non fa seguire invece la strada più semplice nel caso della ricerca del primo e secondo.

Il motivo è di facile comprensione.
Nel caso della ricerca del M e del m l'aver costruito i due sottoinsiemi come sopra, isola la zona in cui si deve trovare il massimo e quella i cui si deve trovare il minimo con $m = n/2$ passaggi, dopodiché si trova il massimo a sinistra e il minimo a destra, rispettivamente con $m - 1 + m - 1$ passaggi e in tutto si ottiene il seguente numero di confronti
$m + m + m = 3\,m = 3m/2$
Tenendo conto poi se l'insieme ha un numero pari o dispari, si arriva alla formula esatta.

Nel caso della ricerca del massimo e del minimo non ho sprecato informazione, mi è servito a risparmiare passi aver separato, nel modo visto, gli oggetti dell'insieme.

Invece per la ricerca del secondo massimo, devo tenere presente che potrebbe essere sia un elemento dell'insieme di destra che un elemento dell'insieme di sinistra.
Questo rende poco significativa la suddivisione fatta, spreco passaggi nella ricerca del secondo massimo

m passaggi per suddividere l'insieme
m-1 per trovare il massimo dell'insieme dei vincenti,
supponiamo che il massimo sia $\mathbf{A_k}$
So che l'elemento del gruppo di sinistra che ha perso con il massimo è $\mathbf{A_{k+m}}$**.**
(Degli altri non mi interessa perché hanno perso con un elemento che a sua volta ha perso con il massimo, quindi sono effettivamente sconfitti, non possono concorrere neanche per il secondo posto).
Posso trovare facilmente il massimo dell'insieme $m – 2$ e poi confrontarlo con A_{k+m}
$m + m – 1 + m – 2 + 1 = 3m - 2$
Il numero di confronti non è il migliore.

Ho però un importante indizio:
le posizioni nel vettore possono essere utilizzate per rappresentare informazioni aggiuntive.

Infatti l'aver messo gli elementi in un vettore mi ha permesso di ricordare che A_{k+m} ha perso con il massimo.

Seguendo questa strada, anche questo modo di procedere permette di trovare l'algoritmo ottimale.

Un aspetto molto importante, oltre il numero dei confronti è costituito dalle altre operazioni che devono essere fatte per organizzare (non perdere) e accedere all'informazione acquisita con i confronti.

Il problema è che anche se una certa informazione è potenzialmente disponibile (si può evincere dai risultati parziali già calcolati) non è detto che lo sia in maniera immediata, diretta.

- ✓ **Quali risultati la implicano?**
- ✓ **Quando sono stati calcolati?**
- ✓ **Dove sono stati messi?**

COSTO DELLA COMPUTAZIONE

In informatica ha un'importanza fondamentale l'indecidibilità del problema dell'arresto.

Il *problema dell'arresto* consiste nel chiedersi se la computazione di un algoritmo A sui dati D abbia termine oppure no. Si può dimostrare che il problema dell'arresto non è decidibile, cioè non esiste un algoritmo H che è in grado di risolvere il problema dell'arresto per qualunque A e per qualunque D.

L'indecidibilità del problema mette in evidenza come non sia possibile costruire un dispositivo che riconosca i problemi non risolubili e li blocchi prima che prenda avvio la computazione.

L'uso del calcolatore ha un costo in termini di tempo e memoria, che dipende essenzialmente dal tipo di rappresentazione dei dati e dalla scelta dell'algoritmo che risolve il problema.

Nell'ambito poi dei problemi algoritmicamente risolubili è di fondamentale importanza distinguere quelli che lo sono in un tempo ragionevole e quelli che richiederebbero un tempo inaccettabile

L'analisi della complessità computazionale consente di stimare il fattore di crescita dei costi temporali in funzione dei dati di input.

I problemi indecidibili sono non numerabili.

LE PRESTAZIONI DEGLI ALGORITMI

Le prestazioni degli algoritmi possono essere confrontate con diversi metodi. Uno di questi consiste, semplicemente, nell'eseguire molti test per ogni algoritmo e nel confrontarne i tempi di esecuzione misurati.Un altro metodo consiste nel confrontare i fattori di crescita delle stime di costo..

Per chiarire meglio il concetto supponiamo di dover risolvere un problema su un insieme di n elementi utilizzando un microsecondo per una operazione di confronto.
La seguente tabella indica il tempo necessario alla determinazione della risoluzione del problema se il tempo necessario è proporzionale a n, n^2, n^3, 2^n, 3^n

	n = 20	**n = 50**	**n = 100**
n	0,00002 secondi	0,00005 secondi	0,0001 secondi
n^2	0,0004 secondi	0,0025 secondi	0,01 secondi
n^3	0,008 secondi	0,125 secondi	1 secondi
2^n	1,05 secondi	35, 7 anni	$4{,}3*10^{14}$ secoli
3^n	58 minuti	$2*10^8$ secoli	$1{,}6*10^{32}$ secoli

Si vede chiaramente che esiste una notevole differenza tra i problemi che richiedono algoritmi eseguibili in tempi che dipendono da potenze di n e quelli che richiedono tempi di tipo esponenziale.

Per algoritmi di complessità n! il tempo sarebbe nettamente superiore a quello richiesto dalle funzioni esponenziali. Ad esempio il numero delle permutazioni di n oggetti, cioé n!, cresce tanto rapidamente che già per n=30 il loro numero ha 33 cifre.
Se non si usa un algoritmo ottimale il problema di disporre n oggetti in un determinato ordine nella memoria del calcolatore può risultare insolubile.

Se affermiamo che il tempo di calcolo è $\boldsymbol{O}(n)$ (O grande di *n*) significa che, all'aumentare di *n*, il *tempo* è proporzionale al numero di elementi *n* nella lista. Di conseguenza ci aspettiamo che il tempo di ricerca risulti triplicato se la dimensione della lista aumenta di un fattore tre.
Possiamo notare questo anche dalla tabella sopra riportata.

Se un algoritmo richiede $\boldsymbol{O}(n^2)$ tempo, allora il suo tempo di esecuzione non cresce più del quadrato della dimensione della lista.

La tabella che segue confronta i valori con cui crescno le funzioni **lg*n*, *n*lg*n*, $\boldsymbol{n^2}$** con **n**=numero degli elementi

n	lg*n*	*n*lg*n*	n^2
1	0	0	1
16	4	64	256
256	8	2.048	65.536
4.096	12	49.152	16.777.216
65.536	16	1.048.565	4.294.967.296
1.048.476	20	20.969.520	1.099.301.922.576
16.775.616	24	402.614.784	281.421.292.179.456

Un fattore di crescita $\boldsymbol{O}(\lg n)$ si ha per gli algoritmi simili alla ricerca binaria. La funzione lg (logaritmo in base 2) cresce di uno quando *n* raddoppia. Si ricordi che, con la ricerca binaria, è possibile ricercare fra il doppio dei valori con un confronto in più. Pertanto la ricerca binaria è un algoritmo di complessità $\boldsymbol{O}(\lg n)$.

Se i valori nella tabella rappresentano microsecondi, allora un algoritmo $O(\lg n)$ può impiegare 20 microsecondi per processare 1.048.476 elementi, un algoritmo $O(n^2)$ può impiegare fino a 12 giorni!

Un altro esempio interessante, e semplice, di riduzione della complessità, è quello del metodo di Horner per il calcolo del valore del polinomio in una variabile, che riduce il numero delle operazioni da effettuare e permette di ridurre la complessità da $O(n^2)$ a $O(n)$.

CALCOLO DEL VALORE DI UN POLINOMIO

METODO ORDINARIO

b	$ax + b$	$ax^2 + bx + c$	$ax^3 + bx^2 + c$	$ax^4 + bx^3 + cx^2 + dx + e$
0	(1 + 1)	2 + (1 + 2)	2 + 3 + (1 + 3)	2 + 3 + 4 + (1 + 4)

Si vede che all'aumentare del grado del polinomio di una unità il numero delle operazioni aumenta di un numero pari a (1 + grado del polinomio)
Infatti si aggiunge sempre una addizione e una moltiplicazione e una potenza che dà un numero di operazioni pari al grado del polinomio meno uno.

Quindi se N è il numero delle operazioni si ottiene

$$N = 2 + 3 + 4 + ... + n + 1 = 1 + (1 + 2 + 3 + 4 + ... + n) + \underbrace{(1+1+1+1+1+ ...)}_{n-1 \text{ volte}} = 1 + n*(n+1)/2 + n-1$$

$N = (n^2 + 3n)/2$ n = grado del polinomio
complessità $\boldsymbol{O}(n^2)$

METODO DI HORNER

$ax + b$	$(ax + b)x + c$	$((ax + b)x + c)x + d$	$(((ax + b)x + c)x + d)x + e$
2	2 + 2	2 + 2 + 2	2 + 2 + 2 + 2

All'aumentare del grado si ha sempre un aumento di due operazioni.

Si capisce inoltre che si risparmiano operazioni perché se devo fare
5*3 + 5*7 + 5*9 e faccio invece 5*(3 + 7 + 9) si vede con chiarezza che la moltiplicazione per 5 invece di farla 3 volte la faccio solo due volte

$N = 2n$ n = grado del polinomio (complessità $\boldsymbol{O}(n)$)

RICORSIONE: EFFICIENTE O INEFFICIENTE

In molti casi una funzione ricorsiva (che richiama sè stessa), è la soluzione più rapida ad un problema, ma usare una funzione non ricorsiva potrebbe rendere molto piu' efficiente lo stesso procedimento. Le funzioni ricorsive hanno fatto la loro comparsa sulla scena molto tempo fa praticamente con il primo linguaggio che era dotato di funzioni.

Una funzione ricorsiva è una funzione che, per svolgere il proprio compito, richiama se stessa. Ad ogni richiamo la "profondità" dell'elaborazione aumenta, fino a quando lo scopo viene raggiunto e risalendo lungo la catena di chiamate ricorsive, la funzione termina l'esecuzione restituendo il valore alla funzione chiamante.

Un esempio di una funzione ricorsiva è la visualizzazione dei file contenuti in una directory.
Invece di costruire un sistema che 'visiti' ogni sotto-directory presente, si crea una funzione 'generica' che legge una directory e visualizza ogni file. Se il file e' una directory, la funzione richiama se stessa per analizzare il contenuto della directory stessa.

Questa logica non tiene conto di possibili problemi. Il codice in questo caso e' molto semplice, ma ci si chiede se è efficiente.
Per cominciare a progettare una funzione ricorsiva, occorre pensare se il "lavoro" da svolgere può essere spezzato in una sequenza di mini-operazioni identiche. Inoltre occorre pensare a quando la singola operazione dovrà concludersi e restituire il controllo all'unità chiamante.

Il numero di chiamate ricorsive della funzione non e' ipotizzabile a priori e questo è un possibile problema, un altro possibile problema è se la funzione utilizza altre risorse oltre alla memoria del sistema quando tali risorse sono in quantità limitata , perché se il numero di richiami è superiore ad un certo limite è possibile un fallimento di chiamata.

Non sempre le funzioni ricorsive sono una risposta efficiente, però potrebbe non esserci una soluzione efficiente.

Sfortunatamente, anche se è sempre possibile, non tutte le funzioni sono però facilmente 'trasformabili' da ricorsive a non-ricorsive. In alcuni casi la funzione non può essere trasformata senza snaturare completamente la logica risolutiva..

Un esempio emblematico di uso inefficiente della ricorsione è descritto qui di seguito, in relazione alla successione di Fibonacci così definita.

1) Fib(0)=0
2) Fib(1) = 1
3) Fib(n) = Fib(n-1) + Fib(n-2) per n 2

Che dà luogo alla successione 0, 1, 1, 2, 3, 5, 8, 13, 21,. . .
Le relazioni delle definizioni si traducono immediatamente nel seguente programma

```
Function FIB (N:integer) : integer;
Begin
  If (N=0) or (N=1) then return N
      else return FIB(n-1) + FIB(n-2)
  endif
end FIB
```

Per calcolare Fib(n-1) occorre prima calcolare Fib(n-2) che viene calcolato 2 volte, ... Fib(n-4) viene calcolato 5 volte
Si può far vedere che Fib(n-i) viene calcolato Fib(i) volte.
Questo non si verifica nella versione iterativa, perché vengono conservati gli ultimi due valori calcolati.

```
If N=0 or N=1
then return N
else PENULT:=0;
ULT:=1;
for i:=2 to n do
H:= ULT;
ULT:=ULT+PENULT;
PENULT:=H
endfor
return ULT
endif
end FIB;
```

Si può però osservare come sia meno immediato capire, rispetto a ll'algoritmo ricorsivo, ciò che il programma deve calcolare.

C'è comunque la possibilità di scrivre un algoritmo efficiente anche utilizzando la ricorsione.

ALGORITMO RICORSIVO EFFICIENTE

```
Function FIB_EFFICIENTE( N: integer ) : integer;

Begin
  return FIB_REC( 0,1, 0,N )
End FIB_EFFICIENTE

Fuction FIB_REC( A,B, K,N: integer ) : integer;

Begin
  If  (K = N)  Then
    return A
  Else
    return FIB_REC( B,A+B, K+1,N )
  End
End FIB_REC
```

Se si escludono questi casi particolare, il maggior numero di volte non c'è una differenza marcata tra l'efficienza dell 'algoritmo ricorsivo e di quello iterativo.

La ricorsione è comunque uno strumento espressivamente più potente per risolvere algoritmicamente problemi.

RICERCA BINARIA

L'algoritmo di ricerca in un albero binario segue un approccio ricorsivo. Un albero binario è una struttura logica in cui ogni singolo elemento (nodo) ha due collegamenti ad altrettanti nodi. Gli alberi binari sono utilizzati per cercare elementi in modo efficiente. Tuttavia, affinché la ricerca sia efficiente, è necessario che l'albero sia, in qualche modo, "bilanciato". Un albero binario è bilanciato se l'altezza dei sottoalberi sinistro e destro, relativi a qualunque nodo, non è troppo diversa (le definizioni precise sono tecnicamente complicate).

La funzione che analizza l'albero in cerca di un valore può essere scritta con la ricorsione in modo molto semplice. Ci sono due casi in cui la funzione deve ritornare: il primo caso è quando il nodo è identico a quello cercato, il secondo caso è quando il nodo non ha nessun 'figlio', cioè quello che cerchiamo non è qui

Se l'abero e' perfettamente bilanciato, per localizzare un qualunque nodo sono richiesti meno di $\log_2(n)$ ricerche, dove n e' il numero di elementi dell'albero. Per un albero di 65.000 elementi sono sufficienti solo 16 confronti. Nel caso di un albero molto sbilanciato, il nostro algoritmo non si comporta altrettanto bene, potrebbe essere necessario in questo caso il controllo di ogni singolo nodo dell'albero.

Trasformare la ricerca binaria in un algoritmo non-ricorsivo e' un lavoro abbastanza semplice. Il nostro algoritmo infatti non richiede espressamente la ricorsione. La funzione ritorna in due soli casi:o ha trovato il nodo o non lo ha trovato. Non c'e' motivo di "ritornare all'inizio".

Si è già avuto modo di osservare che una ricerca in un insieme ordinato di un milione di elementi, che inizia dal punto centrale della sequenza e procede nella metà selezionata, permette di ridurre il numero di passi da 2^{20} a 20 rispetto a una ricerca sequenziale che scorre tutti gli elementi della sequenza ordinata.

La complessità si riduce da lineare quindi, pari alla grandezza del numero se la ricerca è sequenziale, a logaritmica, pari quindi alla lunghezza del numero se è binaria, come si è detto a proposito degli alberi binari bilanciati.

ALGORITMI DI ORDINAMENTO ED EFFICIENZA DELLA RICORSIONE

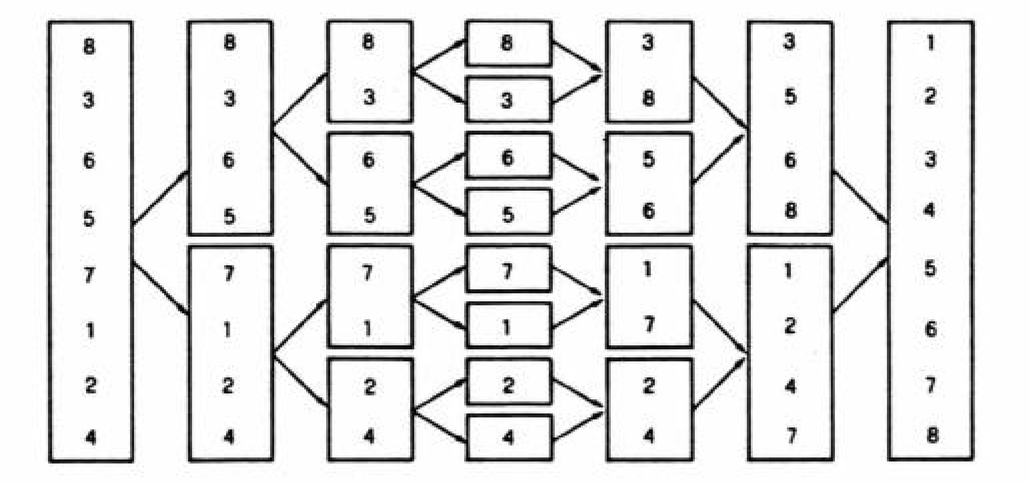
DIVIDE ET IMPERA

I fautori della ricorsione possono citare diversi esempi a loro favore. Particolarmente interessanti da citare sono gli algoritmi di ordinamento Quick Sort e Merge Sort

In particolare Merge Sort per ordinare un vettore di N elementi richiede in tutti i casi un numero di operazioni dll'ordine di *NlogN.*

La chiave della superiorità dei procedimenti di ordinamento che utilizzano la ricorsione come Merge Sort è nell' aver utilizzato il principio del **DIVIDE ET IMPERA** che può essere illustrato con la figura seguente che visualizza le partizioni e le fusioni.

MERGE SORT



DIVIDE ET IMPERA

Viene utilizzata la strategia del DIVIDE ET IMPERA anche dall'algoritmo di ordinamento QUICK SORT, l'ordinamento cosiddetto *veloce*

ALGORITMI DI ORDINAMENTO ED EFFICIENZA

.L'ordinamento per selezione (Selection Sort) ha complessità quadratica rispetto alla lunghezza dell'array da ordinare, come si è mostrato nella seconda unità.
Si può allora concludere che l'ordinamento ha complessità al più quadratica.

Esistono algoritmi con complessità minore? Oppure esistono algoritmi che pur avendo complessità quadratica nel caso peggiore, hanno una complessità migliore nel "caso medio"?
Anche questo è importante.

L'algoritmo **Selection Sort** fa pochi scambi (in numero lineare: n-1), le sue caratteristiche computazionali sono legate ai confronti, e quindi il grosso del peso computazionale ricade proprio sui confronti.
È un algoritmo lento.

L'ordinamento per fusione ha complessità asintotica ***O*(*nlogn*).** L'ordinamento per fusione è asintoticamente più efficiente degli altri algoritmi di ordinamento studiati.

È stato infatti dimostrato che non esistono algoritmi di ordinamento di array, basati su confronti, che abbiano complessità asintotica migliore dell'ordinamento per fusione.

Se confrontiamo Merge Sort con Quick Sort calcolando numerose volte il tempo di esecuzione di un algoritmo, si può notare che **Quick Sort è mediamente più veloce di Merge Sort.**

Come mai?
Infatti **nel caso "medio" il costo esatto è minore di quello dell'ordinamento per fusione**, perché ha un fattore moltiplicativo inferiore.

Il caso peggiore ha una probabilità infima di verificarsi, se si ha a che fare con grandi numeri.
Quick Sort è chiamato infatti ordinamento veloce.

Selection Sort fa meno scambi di Quick Sort, però quest'ultimo fa molti meno confronti ed è decisamente più veloce.

NELLA SEGUENTE TABELLA SONO RIPORTATI DEI RISULTATI TIPICI

Quello che ci interessa sono comunque i rapporti tra i tempi nei vari casi (i valori relativi) e non i valori assoluti.

N ALGORITMO	1000	10 000	100 000	1 000 000
SELECTION SORT	0,03	2,40	250	
BUBBLE SORT	0,08	5,52	550	
INSERTION SORT	0,01	1,80	180	
MERGE SORT	0,01	0,03	0,31	3,63
QUICK SORT	0,01	0,04	0,24	2,42

Per approfondire visitare il sito del prof. Luca Cabibbo dell'Università di Roma)

www.//.dia.uniroma3.it/~cabibbo/fiji/capitoli/pdf/24_ordinamento.pdf

ALTRI ESEMPI POSSIBILI

Agli studenti si possono fare anche altri esempi di riduzione della complessità:

Facciamo l'elenco di alcuni:

- Calcolo del M.C.D. di due numeri con l'algoritmo di Euclide: questo algoritmo permette di determinare il M.C.D. di due numeri naturali ed è molto più efficiente del metodo che richiede di elencare tutti i divisori dei due numeri considerati
- Camminando per le vie di Manhattan (unità del prof. Antoniali)
- La torre di Hanoi
- L'algoritmo del contadino russo
- ..