

A proposito di ordinamento #1 (e di Google Search)

da Alessandro Baricco <the Game>

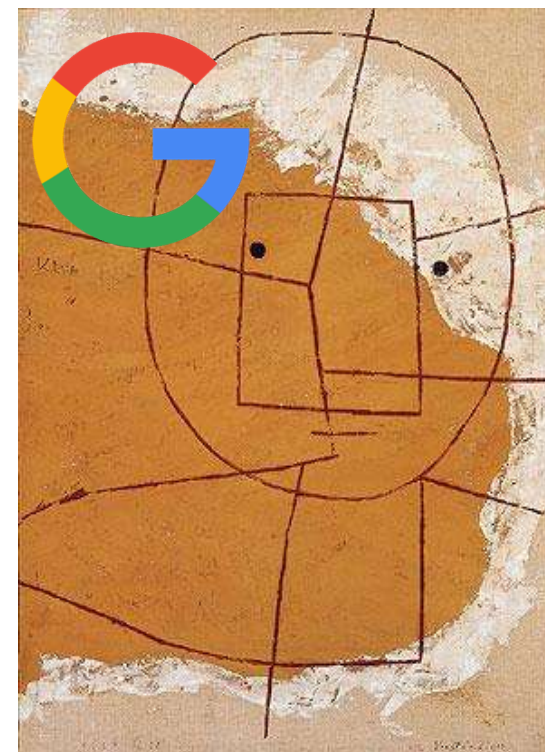
«1998. Due studenti 24enni della Stanford University (Sergj Brin e Larry Page) lanciano un motore di ricerca che, oggi, è il sito web più visitato al mondo. Allora c'erano poco più di 600.000 siti: loro trovarono il modo di farti trovare, in meno di un secondo, tutti quelli che contenevano l'informazione cercata, ordinandoteli in ordine di importanza. La cosa stupefacente è che continuano ad essere in grado di farlo adesso che i siti sono più di un miliardo e 200 milioni. Usando una metafora: se i browser ti procuravano i velieri per viaggiare nel grande mare del web, se Yahoo! ti suggeriva rotte e pericoli, Brin e Page in un colpo misero al servizio di qualsiasi navigatore un mappamondo in cui c'erano tutti i porti del pianeta, ordinati per importanza, confortevolezza e vocazione commerciale. Sapevano dirti quelli in cui si mangiava meglio, dove il prezzo del pepe era più basso e dove i bordelli erano i migliori.»



... continua

A proposito di ordinamento #2 (e di Google Search)

«... Indicizzare e gerarchizzare le pagine web, proposito più che logico, quasi ovvio. Il difficile era come scegliere quelle da mettere in testa alla classifica. La logica tradizionale avrebbe suggerito di prendere degli esperti che segnalassero i siti migliori: ma Google lavorava su numeri tali che la cosa era improponibile; soprattutto Google era oltre: istintivamente mirava a saltare passaggi e mediazioni, cercando una presa diretta sul mondo. Allora fecero una delle mosse più rivoluzionarie che stanno nel cuore della mutazione attuale: sarebbero state le scelte dei diversi utenti a sancire cosa era meglio e cosa peggio. Dove andava più gente, quello era il posto migliore. Viene stabilito un principio che sarà poi decisivo: il parere di milioni di incompetenti è più affidabile, se sei in grado di leggerlo, di quello di un esperto. Sparisce qualsiasi forma di 'sacerdote', resta la presenza vigile di un sistema remoto e le correnti generate da flussi collettivi di enormi dimensioni.»



*da Umberto Eco <pape
satan aleppe>*

«il dramma di Internet è
che ha promosso lo
scemo del villaggio a
portatore di verità.»

L'intelligenza collettiva

E' una forma di intelligenza distribuita, non individuale (dal filosofo Pierre Lévy: *l'intelligenza collettiva. Per un'antropologia del cyberspazio*).

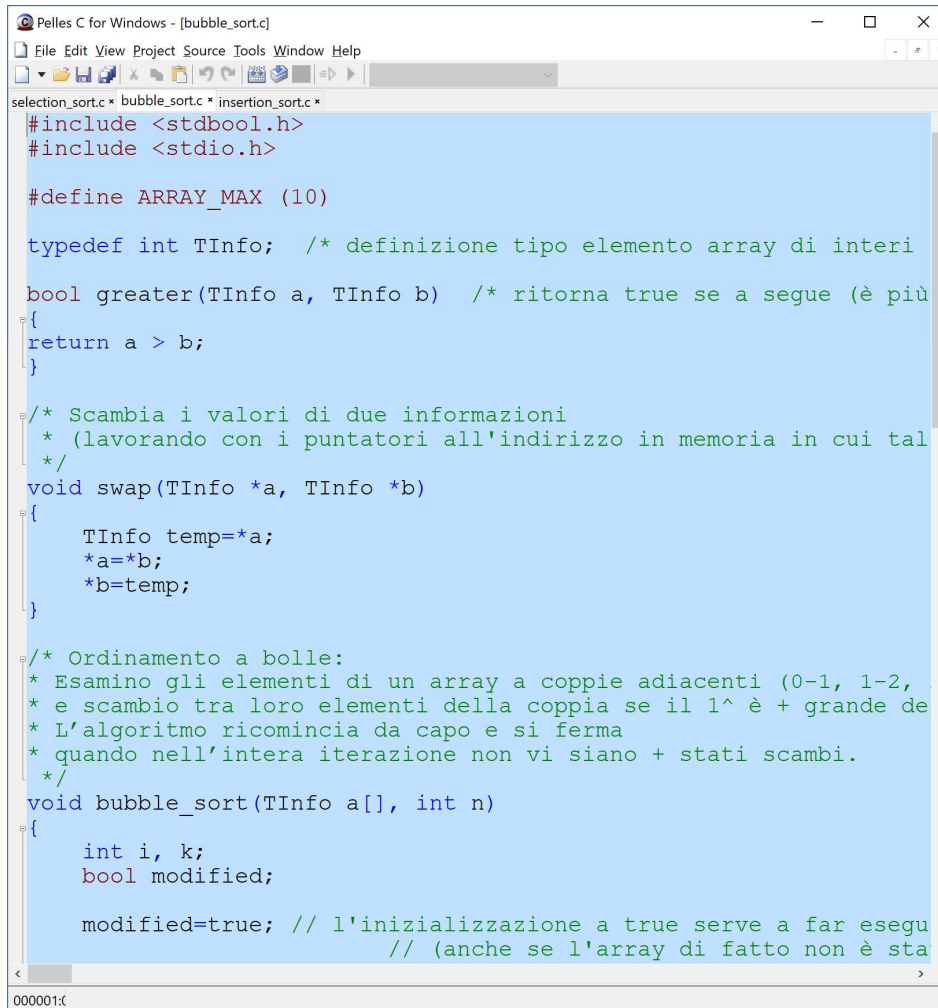
La diffusione delle tecniche di comunicazione su supporto digitale ha permesso la nascita di nuove modalità di legame sociale, non più fondate su appartenenze territoriali, relazioni istituzionali, o rapporti di potere, ma sul radunarsi intorno a centri d'interesse comuni, sul gioco, sulla condivisione del sapere, sull'apprendimento cooperativo, su processi aperti di collaborazione.

Il sapere è sempre diffuso - "nessuno sa tutto, ognuno sa qualcosa" - e "la totalità del sapere risiede nell'umanità« nel suo complesso. Tutta l'esperienza del mondo, quindi, coincide con ciò che le persone condividono.



Un esperimento: *il recipiente di fagioli da contare*

Algoritmi di ordinamento #1a



```
Pelles C for Windows - [bubble_sort.c]
File Edit View Project Source Tools Window Help
selection_sort.c * bubble_sort.c * insertion_sort.c *
#include <stdbool.h>
#include <stdio.h>

#define ARRAY_MAX (10)

typedef int TInfo; /* definizione tipo elemento array di interi
bool greater(TInfo a, TInfo b) /* ritorna true se a segue (è più
{
return a > b;
}

/* Scambia i valori di due informazioni
 * (lavorando con i puntatori all'indirizzo in memoria in cui tal
 */
void swap(TInfo *a, TInfo *b)
{
    TInfo temp=*a;
    *a=*b;
    *b=temp;
}

/* Ordinamento a bolle:
 * Esamino gli elementi di un array a coppie adiacenti (0-1, 1-2,
 * e scambio tra loro elementi della coppia se il 1^ è + grande de
 * L'algoritmo ricomincia da capo e si ferma
 * quando nell'intera iterazione non vi siano + stati scambi.
 */
void bubble_sort(TInfo a[], int n)
{
    int i, k;
    bool modified;

    modified=true; // l'inizializzazione a true serve a far esegui
                  // (anche se l'array di fatto non è sta
000001c
```

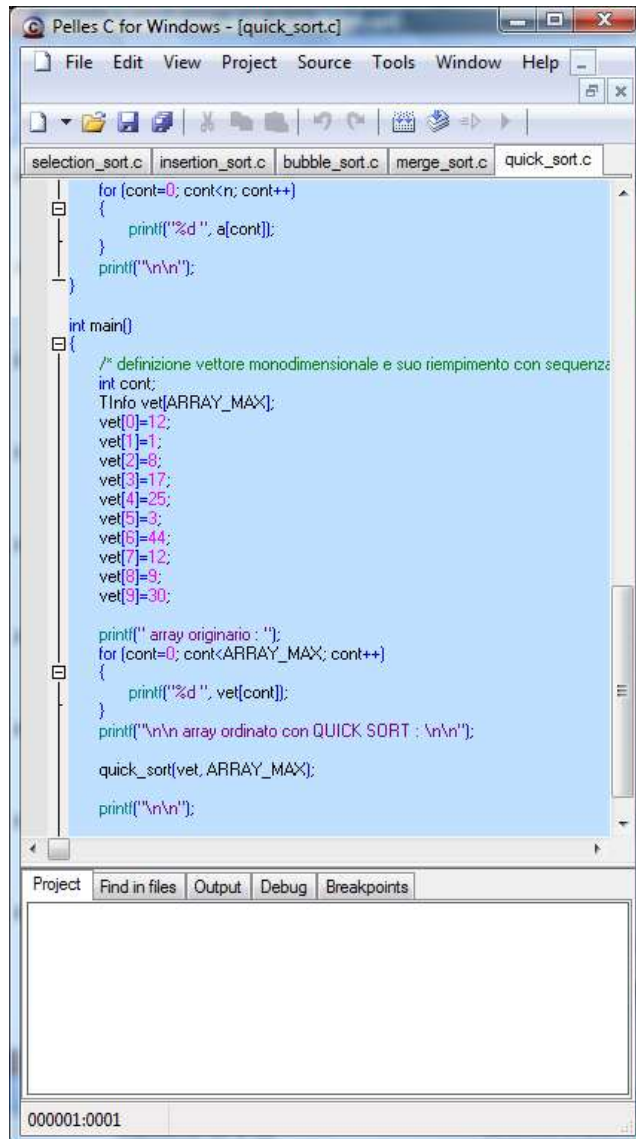
VARI LISTATI IN C CHE IMPLEMENTANO GLI ALGORITMI DI ORDINAMENTO ITERATIVI STUDIATI (selection_sort.c / insertion_sort.c / bubble_sort.c).

In tutti i casi partiremo dallo stesso vettore di interi **vet[12, 1, 8, 17, 25, 3, 44, 12, 9, 30]**, di dimensione 10, ricevuto in input da codice nella funzione MAIN.

Algoritmi di ordinamento #1b

tipo	procedura	complessità
Selection sort	Cerco il minimo tra gli elementi di un array e lo metto al primo posto. Poi cerco il minimo tra gli elementi rimanenti e lo metto al secondo posto ... e così via.	di semplice implementazione ma adatto solo per array piccoli n-1 iterazioni $T(n)=\Theta(n^2)$ <i>non migliora se array è già ordinato</i>
Insertion sort	Prendo uno alla volta gli elementi di un array e li inserisco in ordine in un nuovo array (scorrendolo all'indietro). Spostando in avanti tutti gli elementi seguenti rispetto a quello inserito posso evitare l'uso di un nuovo array (effettuando un inserimento sul posto). Devo tenere traccia di dove sono arrivato e prendere dal primo degli elementi non ancora esaminati/spostati fino all'ultimo.	di semplice implementazione – richiama la funzione di inserimento_in_ordine (che svolge tutto il lavoro) $T_{best}(n)=\Theta(n)$ $T_{worst}(n)=\Theta(n^2)$ <i>efficiente anche quando array è quasi ordinato</i>
Bubble sort	Esamino gli elementi di un array a coppie adiacenti (0-1, 1-2, 2-3, etc.) e scambio tra loro elementi della coppia se il 1 ^a è + grande del 2 ^a . L'algoritmo ricomincia da capo e si ferma quando nell'intera iterazione non vi siano + stati scambi.	di comprensione non immediata n-1 scansioni al massimo $T_{best}(n)=\Theta(n)$ $T_{worst}(n)=\Theta(n^2)$ <i>usabile se il numero di elementi fuori posto (specie numeri piccoli in fondo all'array) è limitato, e tale il numero di scansioni. Altrimenti si può implementare lo shaker sort, metodo che alterna scansioni inizio-fine con scansioni fine-inizio.</i>

Algoritmi di ordinamento #2a



```
selection_sort.c | insertion_sort.c | bubble_sort.c | merge_sort.c | quick_sort.c
for (cont=0; cont<n; cont++)
{
    printf("%d ", a[cont]);
}
printf("\n\n");

int main()
{
    /* definizione vettore monodimensionale e suo riempimento con sequenza
    int cont;
    TInfo vet[ARRAY_MAX];
    vet[0]=12;
    vet[1]=1;
    vet[2]=8;
    vet[3]=17;
    vet[4]=25;
    vet[5]=3;
    vet[6]=44;
    vet[7]=12;
    vet[8]=9;
    vet[9]=30;

    printf(" array originario : ");
    for (cont=0; cont<ARRAY_MAX; cont++)
    {
        printf("%d ", vet[cont]);
    }
    printf("\n\n array ordinato con QUICK SORT : \n\n");

    quick_sort(vet, ARRAY_MAX);

    printf("\n\n");
}
```

VARI LISTATI IN C CHE IMPLEMENTANO GLI ALGORITMI DI ORDINAMENTO RICORSIVI STUDIATI (merge_sort2.c / quick_sort.c).

In tutti i casi partiremo dallo stesso vettore di interi **vet[12, 1, 8, 17, 25, 3, 44, 12, 9, 30]**, di dimensione 10, ricevuto in input da codice nella funzione MAIN.

Esecuzione quick sort

```
Prompt dei comandi
C:\Users\andrea_sergiacomi>"C:\Users\andrea_sergiacomi\Desktop\HOMEworks\Algoritmi Strutture Dati (NEW)\FOGGIA VENTO\2020\listati\05\quick_sort.exe"
array originario : 12 1 8 17 25 3 44 52 9 30

chiamata alla funzione PARTITION
elemento pivot : 12
scambio 1
scambio 8
scambio 3
scambio 9
Primo quick_s
9 1 8 3 -- 12
chiamata alla funzione PARTITION
elemento pivot : 9
scambio 1
scambio 8
scambio 3
Primo quick_s
3 1 8 -- 9
chiamata alla funzione PARTITION
elemento pivot : 3
scambio 1
Primo quick_s
1 -- 3
caso base
Secondo quick_s
8
caso base

array (alla fine) ordinato con QUICK SORT :

1 3 8

Secondo quick_s
3 8 9
caso base

array (alla fine) ordinato con QUICK SORT :

1 3 8 9

Secondo quick_s
44 52 25 30
chiamata alla funzione PARTITION
elemento pivot : 17
Primo quick_s
-- 17
caso base
Secondo quick_s

chiamata alla funzione PARTITION
elemento pivot : 44
scambio 25
scambio 30
Primo quick_s
30 25 -- 44
chiamata alla funzione PARTITION
elemento pivot : 30
scambio 25
Primo quick_s
25 -- 30
caso base
Secondo quick_s
30
caso base

array (alla fine) ordinato con QUICK SORT :

25 30

Secondo quick_s
44 52
caso base

array (alla fine) ordinato con QUICK SORT :

25 30 44 52

array (alla fine) ordinato con QUICK SORT :

17 25 30 44 52

array (alla fine) ordinato con QUICK SORT :

1 3 8 9 12 17 25 30 44 52
```

Algoritmi di ordinamento #2b

tipo	procedura	complessità
Merge sort	<p>Divido un array a metà in 2 sottoarray (che ordino chiamando ricorsivamente per ciascuno la stessa funzione merge_sort). In un vettore di appoggio riporto, a partire dal primo elemento, l'elemento più piccolo tra i primi elementi di ciascuno dei due sottoarray ordinati; poi nel secondo posto riporto l'elemento più piccolo tra i primi elementi rimanenti in ciascuno dei due sottoarray, e così via fino alla fine.</p>	<p>notevolmente migliore degli algoritmi finora presentati</p> <p>$T(n) = \Theta(n \cdot \log n)$</p> <p><i>l'algoritmo di fusione non effettua un ordinamento sul posto, in quanto richiede un array di destinazione distinto dai due array di partenza (esistono versioni dell'algoritmo che aggirano questa limitazione, ma non per gli array e l'implementazione diventa comunque molto più complessa). La creazione di strutture dati temporanee richiedono ulteriore tempo di esecuzione ed occupazione di memoria aggiuntiva per un aggravio computazionale pari a $\Omega(n)$</i></p>
Quick sort	<p>Per evitare di utilizzare array temporanei, partiziono l'array e mediante swap (scambio) riordino gli elementi in base ad un elemento pivot (per semplicità il primo elemento del vettore originario) che dovrà risultare maggiore uguale di tutti gli elementi precedenti e minore uguale dei successivi. Ottengo così 3 sottoinsiemi all'interno dello stesso array: il sottoinsieme con l'elemento pivot, quello che contiene gli elementi inferiori e quello che contiene gli elementi superiori. Ordinando anche questi 2 segmenti utilizzando ricorsivamente la stessa funzione quick_sort, alla fine otterrò un unico array ordinato.</p>	<p>Nel caso migliore ha una complessità temporale pari al merge sort (ma senza occupazione di memoria aggiuntiva). Nel caso peggiore (che si verifica quando il pivot è l'elemento minimo o massimo dell'array, sia nella chiamata iniziale che nelle successive chiamate ricorsive) è pari agli altri algoritmi</p> <p>$T_{\text{best}}(n) = \Theta(n \cdot \log n)$ $T_{\text{worst}}(n) = \Theta(n^2)$</p> <p><i>A differenza di insertion o bubble sort se array è già ordinato, poiché nelle implementazioni si sceglie per convenzione il primo o l'ultimo elemento quale pivot, il quick sort si rivelerebbe estremamente inefficiente (si potrebbe rimediare scegliendo come pivot l'elemento centrale o un elemento a caso)</i></p> <p>47</p>