QUICKSORT

a cura di Pietro Busetti, Rachele Canel e Lorenzo Spaiardi
Realizzato nell'ambito del ***progetto Archimede***
con la supervisione dei Proff. Fabio Breda, Gabriele Rizzo, Valentina Fabbro, Francesco Zampieri
I.S.I.S.S. M.Casagrande, Pieve di Soligo, a.s. 2013/14

Il quicksort è un algoritmo di ordinamento ricorsivo in place che si basa sul paradigma "divide et impera"; La base del suo funzionamento è l'utilizzo ricorsivo della procedura partition. Per poter apprendere al meglio il funzionamento dell'algoritmo dobbiamo analizzare la definizione parola per parola:

- algoritmo: insieme di istruzioni finite che devono essere eseguite per portare a termine un dato compito e per raggiungerte un risultato definito in precedenza;
- ordinamento ricorsivo: algoritmo in cui l'esecuzione dello stesso su un insieme di dati comporta la semplificazione (o suddivisione) e l'applicazione dello stesso algoritmo agli insiemi di dati semplificati.
- algoritmo in place: un algoritmo si dice in place quando i dati in ingresso sono sovrascritti con il risultato prodotto durante l'esecuzione dello stesso;
- paradigma "divide et impera": procedimento secondo il quale un problema viene diviso ricorsivamente in due o più sottoproblemi fino a renderli di facile svolgimento per poi combinare i risultati per ottenere la risoluzione del problema;
- procedura partition: processo alla base del quicksort che individua il pivot all' interno di un insieme di numeri (l'intero processo verrà spiegato meglio insieme alla struttura dell'algoritmo).

Possiamo quindi dire che il quicksort è un insieme di istruzioni che una volta aquisito un vettore con dei valori interi, richiama se stesso più volte ottenendo come risultato un insieme di numeri ordinati.

Ora riportiamo l'algoritmo scritto con il linguaggio di programmazione C++:

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <time.h>
4  using namespace std;
5  const int INPUT_SIZE=100000;
```

In questa prima immagine possiamo notare vari elementi tipici del C++: i primi che ci compaiono alla vista sono delle librerie che servono a includere vari elementi necessari allo svolgimento del programma; successivamente viene dichiarata una costante chiamata "INPUT_SIZE", questa avrà valore 100000 in tutto il programma, quindi se sarà richiamata in una funzione o nel main avrà sempre lo stesso valore.

```
7  void print(int *input)
8  {
9          for(int i=0;i<INPUT_SIZE;i++)
10         cout<<input[i]<<" ";
11         cout<<endl;
12 }
```

In questa seconda immagine invece è presente la procedura per stampare a video un vettore, la procedura inizia con la scritta "void" seguita dal nome della procedura, in questo caso "print", seguito dal nome del parametro che la funzione utilizza per eseguire i procedimenti, poi, all'interno delle parentesi graffe, si inizializza un ciclo di tipo for che stampa a video in sequenza ogni elemento del vettore.

```
38 void quicksort(int *input,int p,int r)
39         {
40                 if(p<r)
41                         {
42                                 int j=partition(input,p,r);
43                                 quicksort(input,p,j-1);
44                                 quicksort(input,j+1,r);
45                         }
46         }
```

Questa procedura si chiama "quicksort" serve a scambiare i valori all'interno del vettore nel modo in cui andremmo a vedere. I parametri iniziali sono il vettore input, il numero intero "p" e il numero intero "r" corrispondenti al numero 0 e numero "INPUT_SIZE", cioè gli estremi del vettore. La funzione inizia con un comparatore logico ("if") che verrà eseguito solo se il valore "p" è minore del valore "r", se ciò è confermato il programma richiamerà la funzione 'partition" che avrà come parametri il vettore input e i valori "p" e "r", il valore di questa funzione verrà assegnato alla variabile 'j" inizializzata ad intero, il nostro pivot. Quindi in poche parole il programma verifica che gli estremi del vettore non si sovrappongano. In seguito, trovato il valore 'j" il programma sarà rieseguito ricorsivamente e quindi richiamerà se stesso, nella prima chiamata avrà come parametri il vettore input, il valore "p" e il valore "j-1"; La seconda chiamata avrà come parametri il solito vettore input e le due variabili "j-1" e "r". La procedura non sarà risolta finché non sarà trovato il caso base perché essa è ricorsiva.

```
14  int partition(int *input,int p,int r)
15      {
16          int pivot=input[r];
17          int tmp;
18          while(p<r)
19              {
20                  while(input[p]<pivot)
21                  p++;
22                  while(input[r]>pivot)
23                  r--;
24                  if(input[p]==input[r])
25                  p++;
26              else if(p<r)
27              {
28                  tmp=input[p];
29                  input[p]=input[r];
30                  input[r]=tmp;
31              }
32              }
33          return r;
34      }
```

Questa funzione è il vero e proprio cuore dell'algoritmo; inizia prendendo dal main 3 valori: il vettore "input", "p" la prima posizione del vettore e "r", l'ultima posizione del vettore. Presi questi 3 elementi la funzione dichiara una variabile pivot e le assegna il valore del vettore nella posizione "r", poi viene dichiarata un altra variabile; successivamente viene eseguito un ciclo che esegue determinate istruzioni solo fino a quando la condizione tra parentesi è verificata. All'interno di questo ciclo vengono eseguiti altri due cicli, il primo si esegue fino a quando il valore in corrispondenza della variabile "p" è minore della variabile pivot, se questo è vero allora la variabile "p" viene incrementata di un unità; il secondo ciclo ha un procedimento molto simile al primo solo che analizza la variabile "r". Dopo questi due cicli otteniamo una comparazione tra il valore del vettore in posizione "p" e il valore in posizione "r": se questi due valori sono uguali allora la variabile "p" viene incrementata di un unità; altrimenti, come si vede in questa immagine, se "p" è minore di "r" allora il valore in corrispondenza di "p" viene scambiato con il valore in corrispondenza di "r". Questa funzione trova il nostro pivot scorrendo tutto il vettore analizzando le posizioni dei vertici, quando gli estremi vengono scambiati il valore in quella posizione viene assegnato alla variabile "pivot"; in seguito vengono presi tutti gli elementi del vettore in posizione minore del pivot e se sono maggiori del suo valore allora vengono spostati nella parte a destra del vettore

```
48  int main()
49          {
50                  int input[INPUT_SIZE];
51                  clock_t start;
52
53
54                  srand(time(NULL));
55                  for(int x=0;x<INPUT_SIZE;x++)
56                  {
57                      input[x]=(rand()%10000);
58                  }
59
60                  cout<<"input: ";
61                  print(input);
62                  start = clock();
63                  quicksort(input,0,INPUT_SIZE-1);
64
65                  cout<<"output: ";
66                  print(input);
```

All'interno del main è presente un vettore con grandezza "INPUT_SIZE" contenente numeri interi e start inizializzato in modo che in un successivo momento sarà in grado di calcolare il tempo. A questo punto assegnamo a ogni componente del vettore un numero casuale generato dalla funzione rand, i numeri generati saranno sempre compresi tra 0 e 9999. Il programma manda in output la parola "input: " e richiama la funzione "print" che ha come parametro il vettore input. A questo punto inizia a scorrere il tempo che servirà per calcolare quanto il programma impiega a svolgere l'operazione di ordinamento. La chiamata alla funzione "quicksort" successiva ha come parametri il vettore input, il numero iniziale zero e il numero finale "INPUT_SIZE" e sarà il fulcro del nostro programma cioè quella funzione ricorsiva in grado di ordinare i numeri. Successivamente il programma da in output "output: " e richiama la funzione "print" che stamperà a video tutti i valori ordinati dalla funzione precedente.

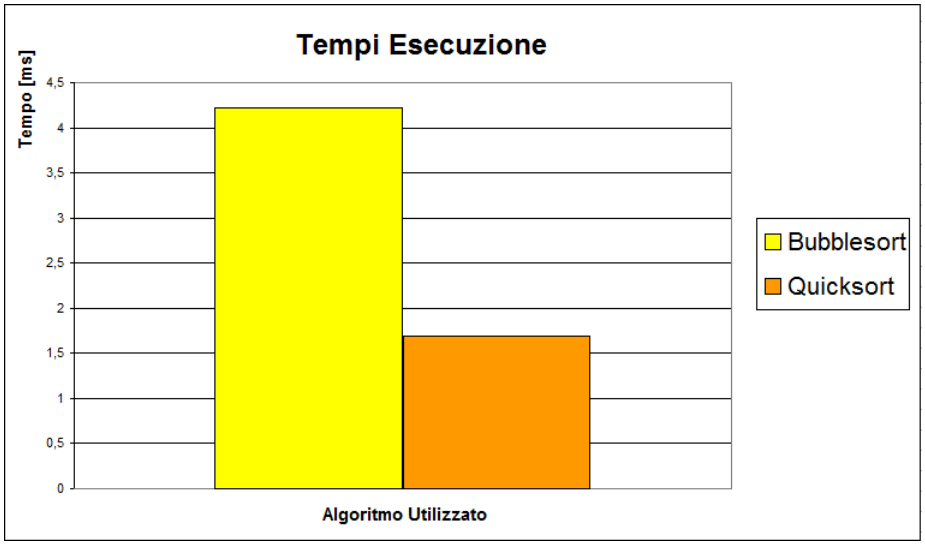
```
68                  clock_t end = clock();
69                  double diff = (double) (end-start)/CLOCKS_PER_SEC*1000.0;
70                  cout<<endl;
71                   cout<<"Il tempo impiegato a eseguire l'operazione è: "<<diff<<endl;
72                  return 0;
73          }
74
```

Il programma fa terminare il conteggio del tempo perché l'ordinamento è terminato e manda in output il valore calcolato del tempo in modo che l'utente si renda conto della durata effettiva del programma, a questo punto il programma è terminato, ciò si può notare dalla parola "return 0".

Ora per dimostrare l'efficacia di questo metodo lo confrontiamo con un algoritmo di ordinamento iterativo: il bubblesort.

```
1  void bubbleSort(int x[])
2      {
3      int temp = 0;
4      int j = x[length-1];
5      while(j>0)
6        {
7        for(int i=0; i<j; i++)
8          {
9          if(x[i]>x[i+1])
10           {
11           temp=x[i];
12           x[i]=x[i+1];
13           x[i+1]=temp;
14           }
15         }
16       j--;
17       }
18     }
```

Questo algoritmo per ordinare i numeri di un vettore usa un procedimento molto diverso rispetto al quicksort, analizza il vettore più e più volte trovando ogni volta il numero minore ecludendolo dal vettore e inserendolo in ordine; questo metodo anche se a prima vista sembra il più efficace, risulta essere il più lento e sconveniente del punto di vista del tempo, risultando anche il più lento come si può vedere dai grafico qui riporato:



Per denotare il divario fra i due algoritmi si deve sapere che questo grafico rappresenta i logaritmi dei tempi medi quindi le misure sono più vicine tra di loro, se avessimo fatto il grafico con le medie reali il grafico sarebbe risultato impossibile da leggere a causa del divario troppo elevato, visto che il tempo

medio per il quicksort si aggira intorno ai 48 millisecondi per 10000 numeri da ordinare, mentre per il bubblesort, sempre con gli stessi numeri da ordinare, ci impiega circa 16606 millisecondi. Per comprendere al meglio la differenza tra i due algoritmi basta notare che il numero di passaggi per ordinare 10 numeri è molto differente, peril quicksort si hanno 21 passaggi, mentre per il bubblesort ne abbiamo ben 44; se con solo dieci numeri il divario è così notevole per ordinare 10000 numeri la differenza tra i due algoritmi sarà immensa. Per prelevare i dati sperimentali abbiamo utilizzato un pc con CPU 3.10 GHz con 4.00 GB di RAM.