Algoritmi di Ordinamento basati sul confronto
Analisi sperimentale con
Implementazione in C++ e Descrizione in Noweb

Federico Bertolli

July 19, 2001

1 Struttura del progetto

L'obiettivo del progetto é la sperimentazione dell'efficienza computazionale dei quattro algoritmi di ordinamento seguenti

- Mergesort
- Quicksort
- Heapsort
- Shellsort

L'applicativo é scritto in C++ utilizzando una forte suddivisione in classi per rendere piú semplice la comprensione del codice e per avere una netta separazione delle varie parti.
Per ogni algoritmo implementato infatti, é stata creata una classe contenente i propri metodi e attributi. Il programma principale invece é un semplice sorgente che non utilizza la programmazione OO ma solamente richiama al suo interno, tutti i metodi delle classi dichiarate.
Un altro programma di supporto é stato creato per ottenere delle sequenze numeriche casuali e ordinate per testare gli algoritmi.

1.1 Descrizione degli algoritmi utilizzati

1.1.1 Mergesort

L'idea che sta alla base dell'ordinamento per fusione *Mergesort*, é molto semplice e si basa sull'osservazione che dal punto di vista temporale, o del numero di operazioni da eseguire, costa molto meno ordinare due sequenze con metá elementi piuttosto che l'intera successione da cui le due sottosequenze sono state estratte. Infatti prendendo come base altri algoritmi di ordinamento come *Bubblesort* o *Insertionsort* che, avendo complessitá media $O(n^2)$, di fronte a una sequenza di $n = 100$ elementi, impiegherebbero circa 100^2 operazioni; se invece si dovesse procedere all'ordinamento di due sottosequenze da 50 elementi, il numero di operazioni complessive scenderebbe a: $50^2 = 2500 * 2 = 5000$. Se poi si riesce a fondere in una sequenza ordinata due sequenze anch'esse ordinate in tempo lineare, allora il costo complessivo dell'ordinamento si ridurrebbe a:

1. Ordinamento di due sequenze da 50 elementi: 5000
2. Fusione di due sequenze ordinate di 50 elementi: 100
3. Costo totale: 5100

Se le stesse operazioni fossero eseguite su 4 sottosequenze estratte dalla successione da ordinare, si avrebbe un'ulteriore riduzione pari a: $25^2 = 625 * 4 = 2500$ a cui aggiungere il costo di 3 fusioni: 2 di 25 elementi che creano due sequenze ordinate da 50 elementi, ed una finale che partendo dalle due sottosequenze appena create di 50 elementi, crea una sequenza ancora ordinata di 100 elementi. Il costo complessivo si puó stimare essere pari a $2500 + 200 = 2700$. Per sfruttare questa idea l'ordinamento Mergesort opera dividendo una sequenza di n elementi non ordinati in sottosequenze ordinate di lunghezza crescente, che a loro volta vengono fuse tra loro (mantenendo l'ordinamento) in nuove sottosequenze. L'efficienza di questo metodo é molto alta in quanto agisce in un tempo che si aggira, nel caso peggiore intorno a $O(n\ log\ n)$. Tuttavia per poter operare, richiede la disponibilitá aggiuntiva di un'area di memoria temporanea di ordine $O(n)$, il che ne limita l'uso e rappresenta una controindicazione al suo utilizzo soprattutto nel caso si operi su grandi masse di dati. Il procedimento sfrutta la possibilitá di fondere tra loro due successioni giá ordinate in una nuova successione anch'essa ordinata impiegando un tempo $O(n)$.

- Implementazione dell'algoritmo MergeSort

  Il seguente metodo *Sort*, divide il vettore A di n elementi in 2 sequenze contenenti $n/2$ elementi ciascuna. Successivamente la procedura richiama se stessa 2 volte: una sulla prima sequenza e una sulla seconda, specificando come parametri i nuovi indici di inizio e fine sequenza. Per effetto della ricorsione, alla fine vengono create n sequenze con ognuna un singolo elemento.
  Quando le chiamate ricorsive "rientrano" viene invocato il metodo *Merge* che si occupa di fondere 2 sequenze ordinate ottenendone una sola anch'essa ordinata.

  ⟨*MergeSort*⟩≡

```
void MergeSort::Sort(int* A,int first,int last)
{
  int medium;

  if(first<last)
  {
    medium=(first+last)/2;
    Sort(A,first,medium);
    Sort(A,medium+1,last);
    Merge(A,first,last,medium);
  }
}
```

Per fondere le due sequenze ordinate, la procedura *Sort* utilizza il seguente metodo *Merge* che prende come parametri in ingresso il vettore A e 3 indici che indicano rispettivamente da dove inizia e dove termina la prima e la seconda sequenza (la prima da $A[first]$ a $A[medium]$ e la seconda da $A[medium+1]$ a $A[last]$). Inoltre c'é l'uso di un vettore di appoggio B che serve come serbatoio temporaneo dei dati.
Ad ogni iterazione sono confrontati gli elementi $A[i]$ e $A[j]$. Il minore dei due viene copiato in $B[k]$ (con k inizializzato a 0) e sono incrementati l'indice k e l'indice dell'elemento minore (i o j). Questo processo si conclude quando una delle due sequenze viene esaurita. Se ció accade con la prima sequenza (cioé $i > medium$) gli eventuali elementi non considerati della seconda (da $A[j]$ a $A[last]$) si trovano giá nelle posizioni corrette dell'ordinamento finale. Se invece é stata esaurita la seconda sequenza, (cioé $i \leq medium$) gli elementi non scanditi (da $A[i]$ a $A[medium]$) della prima, vengono immediatamente spostati nelle ultime posizioni di B (da $B[k]$ a $B[last]$). Si fa notare che non é possibile scrivere direttamente nelle ultime posizioni di A perché ci potrebbe essere una perdita di dati:

– Es:
  Gli elementi non considerati della prima sequenza vengono scritti in fondo ad A.

  1. sequenza $[3, 5, 7]$
  2. sequenza $[0, 1]$

  In questo caso la seconda sequenza é copiata in B (in $B[0]$ e $B[1]$), mentre la prima, non essendo stata scandita viene direttamente copiata nelle ultime 3 posizioni di A. Tale vettore ha dimensione 5 perció scrivendo il 3 in terzultima posizione si elimina l'elemento 7 che fa parte dei dati da spostare. Successivamente si copia il 5 in penultima posizione e l'ipotetico 7 all'ultima. Come giá detto, l'elemento 7 non esiste piú dato che al suo posto c'é il 3, quindi tale valore viene copiato due volte ottenendo la seguente parte di vettore errata:
  $[., ., 3, 5, \underline{3}]$.

Infine, la porzione scritta di B é copiata in A, a partire dalla posizione $first$, riuscendo cosí ad ottenere $A[first..last]$ ordinato.

⟨*Merge*⟩≡

```
void MergeSort::Merge(int* A,int first,int last,int medium)
{
  int i,j,k,h;
  int* B;
  //Alloco la memoria del vettore ausiliario
  B=new(int[last-first+1]);

  i=first;
  j=medium+1;
  k=0;

  while((i<=medium)&&(j<=last))
  {
    if(A[i]<=A[j])
    {
      B[k]=A[i];
      ++i;
    }
    else
    {
      B[k]=A[j];
      ++j;
    }
    ++k;
  }

  //k e' la posizione in cui e' arrivato il vettore B

  //I valori non considerati della prima meta' devo scriverli in B comunque!
  //Questo per non avere perdita di informazione
  for(h=i;h<=medium;++h)
  {
    B[k]=A[h];
    ++k;
  }

  //i risultati li voglio cmq in A e copio i soli valori scritti in B
  for(int y=first;y<k+first;++y)
  {
    A[y]=B[y-first];
  }

  delete [] B;
}
```

1.1.2 Heapsort

L'algoritmo di ordinamento per fusione precedentemente spiegato, presenta come si é visto, un tempo di esecuzione che, sia nel caso peggiore che nel caso medio, si aggira intorno a $O(n\ log\ n)$, pur richiedendo peró, uno spazio aggiuntivo di dimensione n, proporzionale cioé al numero di elementi da ordinare. L'algoritmo di ordinamento *heap* ha la caratteristica di richiedere uno spazio aggiuntivo costante, con un tempo di elaborazione che, sia nel caso peggiore che nel caso medio, si aggira intorno a $O(n\ log\ n)$. L'algoritmo *heap* esegue l'ordinamento in loco mantenendo in ogni istante un numero costante di elementi al di fuori dell'array da ordinare, a tal fine viene usata una struttura dati esterna denominata **heap** con funzione di deposito intelligente delle informazioni durante l'esecuzione. La struttura dati heap assume la veste di un array, ma puó essere visto nel contempo anche come un albero binario completo. Questa analogia tra le due strutture é dovuta al fatto che ogni nodo di un albero binario puó essere fatto corrispondere ad un elemento di un array heap in cui viene inserito il valore contenuto nel nodo. L'albero é completo per cui é riempito su tutti i livelli, tranne, al massimo, quello piú basso dove il riempimento procede da sinistra verso destra. Per quanto riguarda l'array che rappresenta lo heap (definito per semplicitá A), si considerino le seguenti caratteristiche che ne definiscono la dimensione e la lunghezza. Con il termine dimensione, si individua il numero massimo di elementi che possono essere memorizzati nella struttura ad array, mentre con il termine di lunghezza si indica invece il preciso numero di elementi che in ogni istante vi sono mantenuti. Un array heap é caratterizzato dal fatto di mantenere sempre la radice dell'albero binario nella prima posizione; mentre per gli altri nodi sono definite le seguenti relazioni: Se i individua l'indice di un nodo, la posizione all'interno dello heap del nodo padre, del figlio sinistro e di quello destro risulta essere:

- Padre = $\lfloor i/2 \rfloor$
- Figlio SX = $[2i]$
- Figlio DX = $[2i + 1]$

Gli alberi heap possiedono inoltre una proprietá aggiuntiva che impone loro un ordinamento parziale che stabilisce che per ogni nodo i diverso dalla radice, deve essere sempre soddisfatta la relazione $A[padre(i)] \geq A[i]$ che impone a ogni nodo di possedere un valore minore o uguale di quello del proprio padre. Questa proprietá ha come conseguenza che la radice di un albero heap mantiene sempre il valore piú grande tra quelli presenti nell'albero, mentre i sotto-alberi che si generano da qualsiasi nodo contengono valori piú piccoli di quelli del nodo in origine. L'altezza (numero di archi attraversati) di un albero heap é una funzione logaritmica in quanto l'albero é binario completo; cosí le operazioni che agiscono su un albero heap hanno un tempo di esecuzione che é al piú proporzionale all' altezza dell'albero,e quindi una complessitá di tipo logaritmico $O(log\ n)$. Riassumendo, si puó dire che, mentre un albero heap rappresenta un albero parzialmente ordinato in quanto possiede la proprietá che impone ai nodi figli di avere un valore minore di quello del nodo genitore, un array heap rappresenta invece, semplicemente una struttura lineare in grado di mantenere efficientemente al proprio interno un albero heap sfruttando le tre proprietá che legano la posizione dei nodi con gli indici dell'array. L'algoritmo di Heapsort ordina un array di dati di lunghezza n agendo in due fasi: nella prima trasforma l'array originario disordinato in un array heap, cioé in un albero parzialmente ordinato; nella seconda, selezionando ripetutamente l'elemento piú grande presente nello heap e spostando tale elemento nella parte finale dell'array, al fine di creare cosí una sequenza ordinata. L'heap che si viene cosí ad ottenere risulta essere via via sempre piú piccolo. La scelta del valore piú grande é facile da realizzare poiché, in base alle proprietá dello heap, questo valore si trova sempre nella prima posizione in quanto radice dell'albero. Lo scambio che ne risulta comporta la necessitá di procedere ad una probabile risistemazione dell'albero restante.

Durante questa fase l'array risulta essere logicamente diviso in due parti: nella prima viene mantenuto l'albero heap dei restanti dati ancora da ordinare, nella seconda sono presenti gli elementi piú grandi giá ordinati in modo non decrescente.

- Implementazione dell'algoritmo HeapSort

  Il metodo *Sort* chiama al suo interno la funzione *CostruisciHeap* che si occupa di ristrutturare il vettore di partenza A in modo che rispetti le proprietá di uno heap. Tale ristrutturazione avviene grazie alla procedura *RestauraHeap* che si occupa di ricostruire la sezione di A specificata da due indici i e j passati come parametro, se l'elemento in posizione i viola le proprietá dello heap (cioé se $A[i]$ é minore di almeno uno dei suoi figli).
  RestauraHeap é un metodo ricorsivo che inizialmente controlla se $A[i]$ ha un figlio nella porzione $A[i..j]$. Se $i > \lfloor j/2 \rfloor$, allora $A[i]$ é una foglia e il metodo termina, altrimenti cerca $A[k]$ come il massimo dei due figli di $A[i]$, $A[2i]$ e $A[2i+1]$ (o solo $A[2i]$ se l'altro non esiste). Se $A[i] < A[k]$ scambia $A[i]$ con $A[k]$ e richiama se stesso sulla porzione $A[k..j]$.

  $\langle$*RestauraHeap*$\rangle\equiv$

```
void HeapSort::RestauraHeap(int i, int j)
{
  int k,temp;

  if(i<=j/2)
  {
    k=2*i;
    if(k<j)
      if(heap[k-1]<heap[k])
        ++k;

    if(heap[k-1]>heap[i-1])
    {
      temp=heap[i-1];
      heap[i-1]=heap[k-1];
      heap[k-1]=temp;
      RestauraHeap(k,j);
    }
  }
}
```

Dato il vettore A contenente gli n elementi da ordinare, é possibile ottenere da esso un vettore heap grazie alla funzione *CostruisciHeap* che richiama la procedura *RestauraHeap* ripetutamente, dalle foglie verso la radice dell'albero (*RestauraHeap* permuta le componenti di A in modo che siano verificate le proprietá dello heap). Le chiamate a tale funzione *RestauraHeap* sono effettuate solo per $i \leq \lfloor n/2 \rfloor$ perché la porzione $A[\lfloor n/2 \rfloor + 1..n]$ é formata da foglie che per definizione sono giá heap costituiti da un solo elemento ciascuno.

⟨*CostruisciHeap*⟩≡
```
void HeapSort::CostruisciHeap(int n)
{
  for(int i=n/2;i>0;--i)
    RestauraHeap(i,n);
}
```

Una volta terminata la *CostruisciHeap*, nella prima posizione del vettore A si trova l'elemento massimo. Questo puó essere scambiato direttamente con quello in ultima posizione, essendo il posto che gli compete nella sequenza ordinata finale. A causa dello scambio peró il vettore potrebbe non soddisfare piú le proprietá dello heap e va perció invocata la funzione *RestauraHeap* sui primi $n-1$ elementi. A questo punto il nuovo primo elemento puó essere scambiato con il penultimo ma nuovamente deve essere chiamata la *RestauraHeap* sui primi $n-2$ elementi per ripristinare il vettore heap. Iterando il procedimento si ottiene il vettore A ordinato in modo crescente.

⟨*HeapSort*⟩≡
```
void HeapSort::Sort(int** A,int n)
{
  int i,temp;

  heap=*A;

  CostruisciHeap(n);

  for(i=n;i>1;--i)
  {
    temp=heap[i-1];
    heap[i-1]=heap[0];
    heap[0]=temp;
    RestauraHeap(1,i-1);
  }
}
```

1.1.3 Quicksort

L'algoritmo che sta alla base dell'ordinamento *Quick* é stato ideato da Hoare e presenta un comportamento con costo medio di ordine $O(n\ log\ n)$ senza richiedere spazi temporanei aggiuntivi; solo nel caso pessimo la sua complessitá puó divenire $O(n^2)$. L'algoritmo si fonda sull'idea di scegliere uno degli elementi della successione da ordinare, che da ora verrá denominato con il termine di *"pivot"*. Questo valore sará usato per effettuare una partizione dei valori in due gruppi distinti: nel primo gruppo (di sinistra) saranno inseriti i valori minori del valore di pivot, mentre alla destra saranno inseriti i valori maggiori o uguali sempre al valore di pivot. Questa operazione di spostamento presenta una complessitá di ordine $O(n)$. Si ottengono cosí due successioni disgiunte costituite, la prima da valori sicuramente minori di quelli presenti nella seconda successione; le due successioni possono pertanto essere ordinate in forma ricorsiva riapplicando su ambedue, lo stesso procedimento che porta alla scelta di un nuovo elemento di pivot ed a una nuova suddivisione degli elementi in altre due successioni di dimensione ovviamente minore di quella originaria. Il processo termina quando le successioni da ordinare saranno costituite da zero o un elemento. Un punto fondamentale per il buon andamento del metodo é rappresentato dalla scelta dell'elemento che dovrá essere utilizzato come pivot, in quanto dovrebbe avere un valore mediano rispetto alla successione a cui deve essere applicato, in modo da poter dividere i dati in due successioni formate all'incirca dallo stesso numero di componenti. La scelta del valore da usare come pivot richiederebbe una preventiva scansione della successione, questo puó essere compiuto scrivendo un'apposita funzione il cui calcolo potrebbe peró portare ad un notevole rallentamento; per evitare questo si potrebbe operare una scelta di natura piú semplicistica e piú veloce: prendere un elemento qualsiasi dalla successione (ad esempio il primo o un elemento preso a caso grazie a una funzione random che genera numeri casuali con distribuzione uniforme) sulla base dell'ipotesi che essendo la successione disordinata, é probabile che questa scelta in media non faccia prendere un valore tale da sbilanciare le due successioni di numeri maggiori e minori del pivot (cioé che una delle due successioni abbia dimensione molto maggiore dell'altra); oppure si potrebbero considerare gli elementi primo, ultimo e centrale e fra loro considerare come pivot il mediano. L'algoritmo di base puó essere cosí sintetizzato:

1. Scegliere un elemento p dalla successione S da usare come pivot
2. Partizionare la successione $S - p$ in due nuove partizioni disgiunte tali che $(S_1 = v \in S - p \mid v < p)$ e $(S_2 = v \in S - p \mid v \geq p)$
3. Terminare creando la successione che si ottiene applicando:
    - nuovamente lo stesso algoritmo su S_1
    - nuovamente lo stesso algoritmo su S_2

Come é gia stato detto, la scelta dell'elemento pivot costituisce una condizione fondamentale perché l'algoritmo possa fornire delle buone prestazioni; la scelta di usare il primo elemento puó essere valida solo se la successione da ordinare contiene valori di tipo casuale. Nel caso contrario questa scelta potrebbe portare a individuare un valore che non riesce a dividere la successione in due parti di pari dimensione: facilmente gli elementi potrebbero confluire piú su una successione rispetto che sull'altra. Nel caso ci sia la possibilitá di agire su sequenze giá parzialmente o totalmente ordinate, la scelta del pivot deve essere fatta in modo diverso per non imbattersi in questo problema che porterebbe la complessitá al caso pessimo $O(n^2)$. Una strategia alternativa consiste nel selezionare come pivot l'elemento *mediano*, valore questo che purtroppo é costoso da calcolare. Una buona stima peró si puó ottenere estraendo tre elementi dalla successione (ad esempio il primo, l'ultimo e il centrale) e considerare come pivot il mediano fra questi. Questa strategia riesce a trattare in modo positivo i casi in cui possono essere presenti successioni ordinate senza che le operazioni di ordinamento richiedano grossi aggravi di tempo.

- Implementazione dell'algoritmo QuickSort

Il seguente metodo *Sort* riceve in ingresso la porzione di vettore da ordinare. In primo luogo invoca la funzione *Perno* che si occupa di selezionare un elemento (detto *Pivot*) e di risistemare quelli restanti nel seguente modo: gli elementi minori del pivot alla sua sinistra mentre i maggiori di esso alla sua destra. Successivamente *Sort* chiama ricorsivamente se stessa due volte. Questo per eseguire il medesimo procedimento sugli elementi minori del pivot e su quelli maggiori. Le chiamate ricorsive si susseguono fino ad operare su porzioni di vettore con dimensione pari a 1. Quando tali chiamate saranno tutte rientrate, si avrá il vettore A ordinato.

$\langle$*QuickSort*$\rangle\equiv$
```
void QuickSort::Sort(int* A,int first,int last)
{
  //conterra' il nuovo valore del perno dopo la chiamata alla Perno()
  int j;

  if(first<last)
  {
    j=Perno(A,first,last);
    Sort(A,first,j-1);
    Sort(A,j+1,last);
  }
}
```

Come giá detto, la procedura *Perno* sceglie un elemento rispetto al quale poi "riarrangiare" quelli restanti. Per fare ció peró deve prima prenderne in considerazione uno con il quale iniziare la computazione. La scelta di tale elemento iniziale é cruciale per l'efficienza della funzione. Fra le molteplici possibilitá di scelta sono state considerate le seguenti:

1. Primo elemento
   Si considera come valore iniziale il primo elemento del vettore A.

   $\langle$*Scelta 1*$\rangle\equiv$
   ```
   //In x metto il primo valore
     x=A[first];
   ```

2. Elemento casuale
   Utilizzando un generatore di numeri casuali interi (con distribuzione uniforme) viene generato un valore i compreso nell'intervallo definito dagli indici di inizio e fine della porzione di vettore considerata.
   Viene preso come valore iniziale l'elemento $A[i]$.

   $\langle$*Scelta 2*$\rangle\equiv$
   ```
     //Calcolo l'elemento casuale
     i=first + (int)((last-first+1.0)*rand()/(RAND_MAX+1.0));

     //scambio l'elemento casuale precedentemente calcolato con il primo
     temp=A[first];
     A[first]=A[i];
     A[i]=temp;

     //in x metto il primo valore che ora sara' quello casuale
     x=A[first];
   ```

3. Mediano fra tre
   Si considerano tre elementi a piacere del vettore e si prende come iniziale il mediano fra i tre. Nell'implementazione i tre indici scelti sono stati il primo, l'ultimo e il centrale.

   ⟨*Scelta 3*⟩≡

```
//Salvo in x,y,z rispettivamente il primo elemento,
//l'ultimo e quello centrale
x=A[first];
y=A[(first+last)/2];
z=A[last];

//Ora li confronto scegliendo il mediano dei tre
if(((x>y)&&(x<z)&&(y<z))||((x<y)&&(x>z)&&(y>z)))
{
  //Prendo il primo
  //in x c'e' gia' il valore corretto
}
else if(((x<y)&&(x<z)&&(y>z))||((x>y)&&(x>z)&&(y<z)))
{
  //Prendo il terzo
  A[first]=z;
  A[last]=x;
}
else //if(((x<y)&&(x<z)&&(y<z))||((x>y)&&(x>z)&&(y>z)))
{
  //Prendo il centrale
  A[first]=y;
  A[(first+last)/2]=x;
}

x=A[first];
```

4. Mediano fra tutti
   La scelta del valore di partenza é decisa grazie all'algoritmo di **Ricerca lineare del punto medio**. Questo algoritmo, sfruttando la procedura *Perno* (che ha costo $O(n)$), riesce a trovare il k^{esimo} elemento piú piccolo senza ordinare il vettore. Se si pone $k = n/2$ il problema si riduce a trovare appunto il MEDIANO. La procedura *Perno* scambia tra loro elementi della porzione di A considerata e restituisce l'indice j del perno tale che gli elementi precedenti ad esso nel vettore siano minori, mentre quelli successivi siano maggiori.

   ⟨*Scelta 4*⟩≡

```
//scambio il primo valore con il mediano calcolato da Select
x=Select(A,first,last,(last-first)/2);

temp=A[first];
A[first]=A[x];
A[x]=temp;

m=A[first];
```

La seguente procedura *Select* richiama la *Perno* individuando cosí il k^{esimo} elemento piú piccolo della porzione $A[first..last]$ dove $q = j - first + 1$ é il numero di elementi di $A[first..j]$. Se $k \leq q$ allora *Select* é richiamata su $A[first..j]$, se invece $k > q$ su $A[q+1..last]$, per cercare peró il $(k-q)^{esimo}$ elemento piú piccolo.

$\langle Select \rangle \equiv$

```
int QuickSort::Select(int* A,int first,int last,int k)
{
  int q,j;

  if(first==last)
    return(first);
  else
  {
    j=Perno(A,first,last);

    //numero di elementi minori del perno relativi
    //al sottovettore considerato
    q=j-first+1;

    if(k<=q)
      return(Select(A,first,j,k));
    else
      return(Select(A,j+1,last,k-q));
  }
}
```

Una volta scelto l'elemento iniziale (d'ora in poi x) si procede in questo modo: gli altri elementi del vettore vengono via via suddivisi in tre sequenze: quelli minori di x, quelli maggiori e quelli non ancora esaminati. La *Perno* utilizza due indici i e j. L'indice i punta al primo elemento y non ancora considerato, mentre j dá la posizione finale in cui va spostato il perno. Ad ogni iterazione, viene controllato il valore di y: se $y \geq x$ allora y non viene spostato. Viceversa, se $y < x$ allora y viene scambiato con il primo elemento tra quelli maggiori o uguali a x. Alla fine x é permutato con $A[j]$.

$\langle Perno \rangle \equiv$

```
int QuickSort::Perno(int* A,int first,int last)
{
  int x,i,j,temp;

  //In x metto il primo valore del perno in base alle quattro
  //possibilita' specificate sopra
  x=...;
  j=first;

  for(i=first+1;i<=last;++i)
    if(A[i]<x)
    {
      ++j;
      temp=A[j];
      A[j]=A[i];
      A[i]=temp;
    }

  //scambio l'elemento in prima posizione con quello in posizione perno
  A[first]=A[j];
  A[j]=x;
  return(j);
}
```

1.1.4 Shellsort

Questo algoritmo ideato da Shell, rappresenta un'estensione dell'algoritmo di ordinamento *Insertionsort*. Per spiegare il suo funzionamento si considerino dei giocatori di carte: per ordinare una mano, il giocatore considera le carte una per volta ad esempio da sinistra verso destra; ogni volta che ne considera una la inserisce fra le precedenti giá ordinate nella posizione corretta traslando verso destra le carte con valore maggiore. La complessitá della procedura dipende dal numero di inversioni (un'inversione é data da una coppia di elementi a_i, a_j *t.c.* $i < j$ *e* $a_i > a_j$) della sequenza iniziale. Il caso peggiore si ha quando il vettore di partenza é ordinato alla rovescia e il numero di inversioni della sequenza iniziale é quindi massimo. In questo caso ad ogni iterazione dell'algoritmo l'elemento considerato deve essere sempre portato in prima posizione con conseguente complessitá $O(n^2)$. Viceversa, per sequenze ordinate la complessitá é $O(n)$ poiché il numero di inversioni é nullo. La pecca dell'*Insertionsort* é quella di poter spostare solo elementi tra loro adiacenti, richiedendo pertanto $O(n)$ tempo per trasferire un elemento piccolo dall'estremitá destra a quella sinistra della sequenza. Il suo pregio d'altronde é quello di richiedere poco tempo se la sequenza é quasi ordinata. Per eliminare la pecca e mantenere il pregio, si puó pensare di spostare elementi che non sono adiacenti, ma distanti h (con $h > 1$). Shell, ha concretizzato questo ragionamento creando l'estensione dell'Insertionsort chiamato appunto *Shellsort*. L'idea di base consiste nell'usare una sequenza di distanze decrescenti, in modo da ordinare in primo luogo elementi che sono fra loro lontani, e solo successivamente prendere in considerazione elementi vicini, che grazie agli ordinamenti precedenti dovrebbero risultare quasi ordinati. La procedura implementata ordina elementi distanti h (con h minore ad ogni iterazione fino ad arrestare la computazione quando ha valore 1), in modo da creare h sequenze ordinate indipendenti fra loro ma comunque legate dal fatto che un singolo valore puó essere coinvolto in molte di esse. Da un'analisi della complessitá risulta che l'algoritmo di ordinamento Shellsort costa tempo $O(n\sqrt{n})$ nel caso pessimo.

- Implementazione dell'algoritmo ShellSort

  La seguente procedura *Sort* riceve in ingresso il vettore da ordinare A e la sua dimensione n. In primo luogo si cerca h (inizializzato a 1) il piú grande possibile (ovviamente minore di n). Successivamente viene considerata la sequenza con gli elementi "distanti fra loro h" e si esegue un ordinamento a ritroso fra questi. Una volta terminata l'operazione su tale sequenza, si decrementa h ($h = \lfloor h/3 \rfloor$) e si ripete l'ordinamento. Iterando il procedimento fino ad ottenere $h = 1$, si riesce ad avere il vettore A ordinato. Si noti che l'ultimo passo dell'iterazione non é altro che un *InsertionSort* eseguito sull'intero vettore A in cui peró ci si aspettano pochi elementi da scambiare.
  Si possono considerare molteplici valori di h perché, assieme ai dati di ingresso, anch'essi influenzano la complessitá dell'algoritmo.
  Nell'implementazione sono stati scelti i seguenti valori:
  $h = (3^i - 1)/2$ con $i = t, t-1, ..., 1$ dove t é il massimo indice s t.c. $(3^{s+1} - 1)/2 > n$

⟨*ShellSort*⟩≡

```
void ShellSort::Sort(int* A,int n)
{
  int i,j,k,h;
  bool out;

  h=1;
  //cerco h il piu' grande possibile rispetto a n
  while(h<=n)
    h=3*h+1;
  h/=3;

  while(h>=1)
  {
    for(i=h+1;i<=n;++i)
    {
      k=A[i-1];
      j=i;
      out=false;

      while((A[j-h-1]>k)&&(!out))
      {
        A[j-1]=A[j-h-1];
        j-=h;
        if(j<=h)
        {
          j+=h;
          out=true;
        }
      }
      if(out==true)
        A[j-h-1]=k;
      else
        A[j-1]=k;
    }
    //consideriamo ora il caso con h minore
    h/=3;
  }
}
```

2 Testing degli algoritmi

Gli algoritmi sono stati testati con numeri interi ordinati e casuali, quest'ultimi ottenuti con un generatore a distribuzione uniforme avente range $[0..2^{16}-1]$.
Sono state effettuate prove ripetute per ogni insieme di dati, non riscontrando differenze significative tra le varie esecuzioni. Questo puó essere dato dall'imprecisione della misurazione dei cicli di CPU o dal fatto che i vari algoritmi, non richiedendo operazioni di I/O e operando solo in memoria, sono molto stabili rispetto al tempo impiegato. Infatti il tempo di salvataggio dei dati su file non viene conteggiato.
I risultati sono stati rappresentati tramite 10 grafici. Questi presentano sull'asse delle ascisse le *cardinalitá* dei vettori sui quali é avvenuta l'elaborazione, mentre sull'asse delle ordinate il tempo di calcolo espresso come *cicli di CPU*.

2.1 Testing con Numeri Casuali

Il test é stato eseguito con sequenze di cardinalitá:

1. *Grafico 1*
   Da 0 a 100000 con numeri presi a intervalli di 5000.
2. *Grafico 2*
   Da 0 a 100000 con numeri presi a intervalli di 5000.
3. *Grafico 3*
   Da 0 a 500000 con numeri presi a intervalli di 10000.
4. *Grafico 4*
   Da 0 a 500000 con numeri presi a intervalli di 10000.

2.2 Testing con Numeri Ordinati in modo <u>crescente</u>

Il test é stato eseguito con sequenze di cardinalitá:

1. *Grafico 5*
   Da 0 a 25000 con numeri presi a intervalli di 5000.
2. *Grafico 6*
   Da 0 a 100000 con numeri presi a intervalli di 5000.
3. *Grafico 7*
   Da 0 a 25000 con numeri presi a intervalli di 5000.

2.3 Testing con Numeri Ordinati in modo decrescente

Il test é stato eseguito con sequenze di cardinalitá:

1. *Grafico 8*
   Da 0 a 25000 con numeri presi a intervalli di 5000.

2. *Grafico 9*
   Da 0 a 100000 con numeri presi a intervalli di 5000.

3. *Grafico 10*
   Da 0 a 25000 con numeri presi a intervalli di 5000.

3 Discussione dei risultati ottenuti

3.1 Legenda

Le legende dei seguenti 10 grafici contengono degli acronimi. Il loro significato é riportato qui di seguito:

Acronimo	Significato
ms.r	MergeSort
hs.r	HeapSort
ss.h	ShellSort
qs.h	QuickSort 1 (con Perno = primo elemento)
qsr.h	QuickSort 2 (con Perno = casuale)
qsm3.h	QuickSort 3 (con Perno = stima della mediana con tre elementi)
qspm.h	QuickSort 4 (con Perno = mediana degli elementi)

3.2 Numeri Casuali

3.2.1 1^o *esperimento*

Il *Grafico 1* presenta le prestazioni dei quattro algoritmi **Mergesort, Heapsort, Shellsort e Quicksort 1**. Si puó notare molto chiaramente che **Quicksort 1** risulta essere nettamente il piú veloce. Fra gli altri tre **Shellsort** sembra prevalere su **Mergesort** e **Heapsort** che circa si equivalgono. Il peggiore é comunque **Mergesort**.

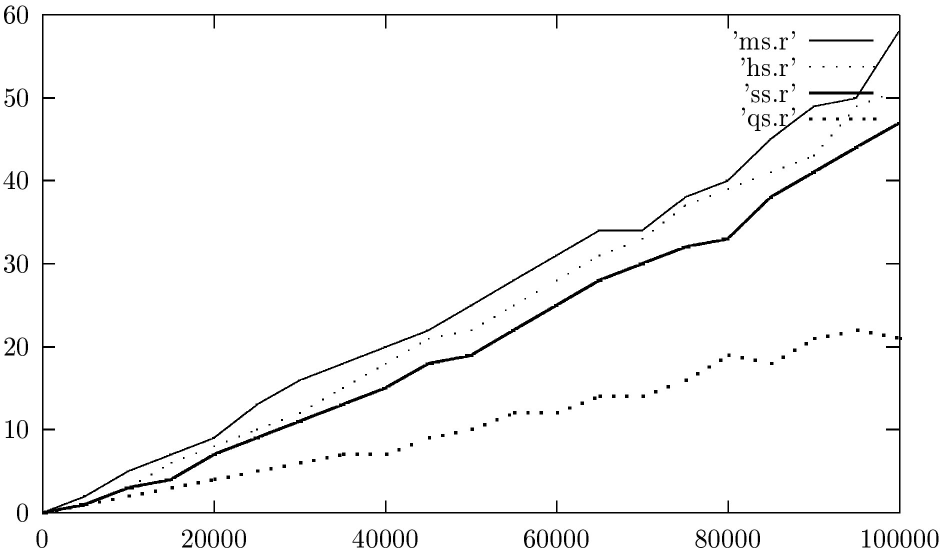

Grafico 1

3.2.2 2^o *esperimento*

Il *Grafico 2* presenta le prestazioni dell'algoritmo **Quicksort** nelle quattro varianti: **Quicksort 1, Quicksort 2, Quicksort 3** e **Quicksort 4**.
Si puó notare, che per sequenze con numeri casuali, **Quicksort 1** e **Quicksort 3** prendono il sopravvento sugli altri due, mantenendo i loro tempi computazionali molto minori di **Quicksort 2** e **Quicksort 4**.
Quicksort 1 e **Quicksort 3** inoltre, hanno un andamento circa uguale che permette di preferirne uno all'altro indifferentemente.
Quicksort 2 e **Quicksort 4** pur avendo qualche intersezione fra le due curve, sembrano equivalersi fino a sequenze di cardinalitá 80000.
Il peggiore risulta essere comunque **Quicksort 4** che per valori superiori a 90000 presenta una lieve impennata.

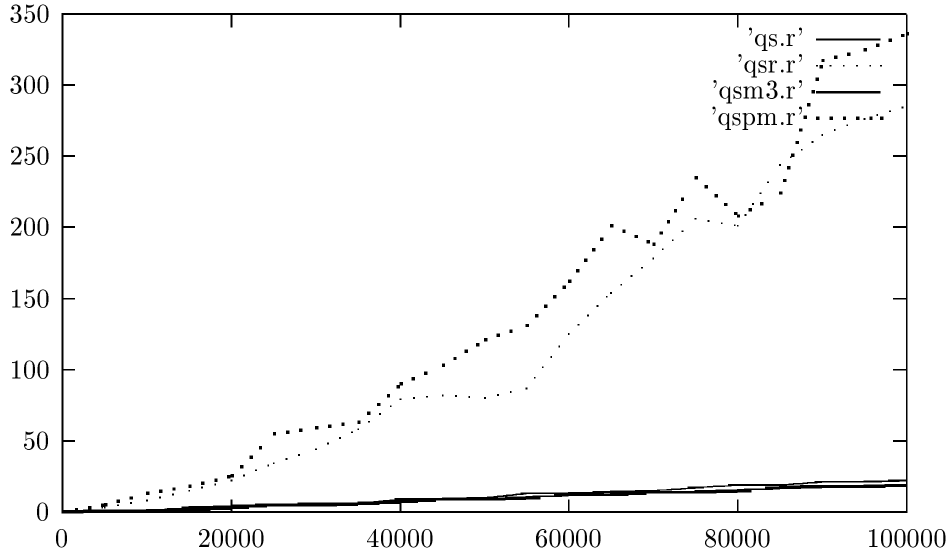

Grafico 2

3.2.3 3^o *esperimento*

Il *Grafico 3* presenta le prestazioni dei quattro algoritmi **Mergesort, Heapsort, Shellsort e Quicksort 1**, questa volta peró sono state considerate delle sequenze con cardinalitá molto piú alte: fino a 500000 valori.
Quicksort 1 é ancora nettamente il migliore presentando una curva con un'inclinazione veramente lieve rispetto alle altre.
Si riesce a vedere, grazie alla parte finale del grafico, che **Shellort**, per valori molto elevati, aumenta parecchio il tempo di computazione non essendo piú competitivo con **Mergesort** e **Heapsort**.
Questi ultimi invece sembrano equivalersi, anche se quello migliore fra i due pare essere **Mergesort**.

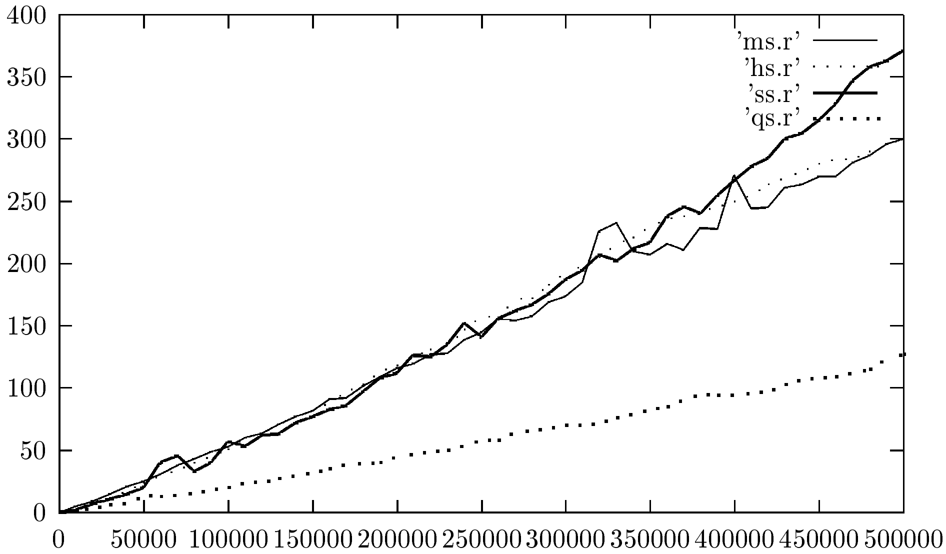

Grafico 3

3.2.4 4^o *esperimento*

Il *Grafico 4* presenta le prestazioni dell'algoritmo **Quicksort** nelle quattro varianti: **Quicksort 1, Quicksort 2, Quicksort 3** e **Quicksort 4** con sequenze peró con cardinalitá molto maggiori al *Grafico 2*, infatti qui si arriva a computare vettori di 500000 elementi.
Si puó notare, che nuovamente, **Quicksort 1** e **Quicksort 3** prendono il sopravvento sugli altri due, mantenendo i loro tempi computazionali molto minori di **Quicksort 2** e **Quicksort 4**.
Quicksort 1 e **Quicksort 3** inoltre, hanno degli andamenti circa uguali che ancora, non ne fanno preferire uno all'altro. **Quicksort 2** e **Quicksort 4** pur avendo delle curve, che oltre a essere molto irregolari, sono anche intersecate fra loro, sembrano avere in media, circa lo stesso comportamento anche se pare essere leggermente peggiore il secondo.

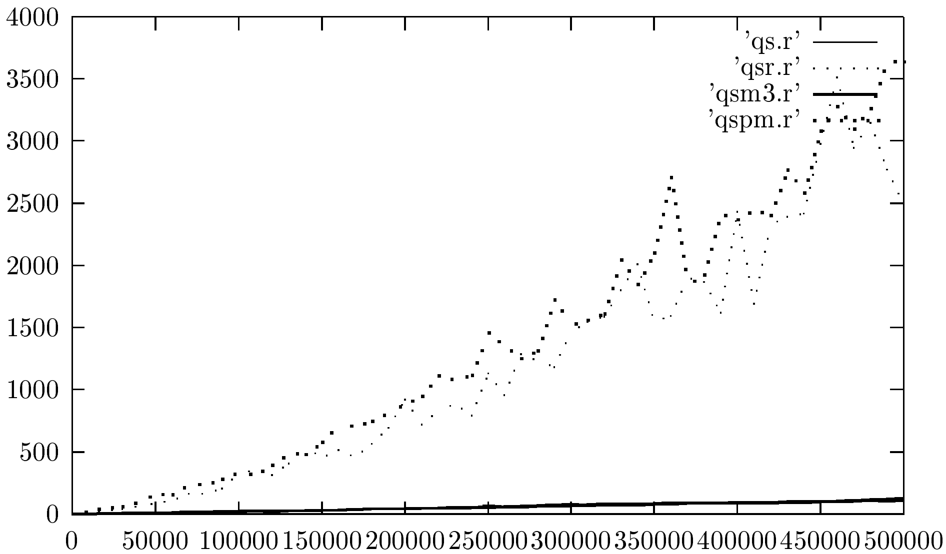

Grafico 4

3.3 Numeri Ordinati in modo *Crescente*

3.3.1 5^o *esperimento*

Il *Grafico 5* rappresenta il costo computazionale dei quattro algoritmi di ordinamento:
Mergesort, Heapsort, Shellsort e **Quicksort 1**
testati con sequenze contenenti numeri ordinati in modo crescente.
Si noti come **QuickSort 1** raggiunga il caso pessimo abbassando drasticamente la sua efficienza. Gli altri tre algoritmi infatti hanno tempi computazionali cosí lievi rispetto al primo da non venir neanche visualizzati.

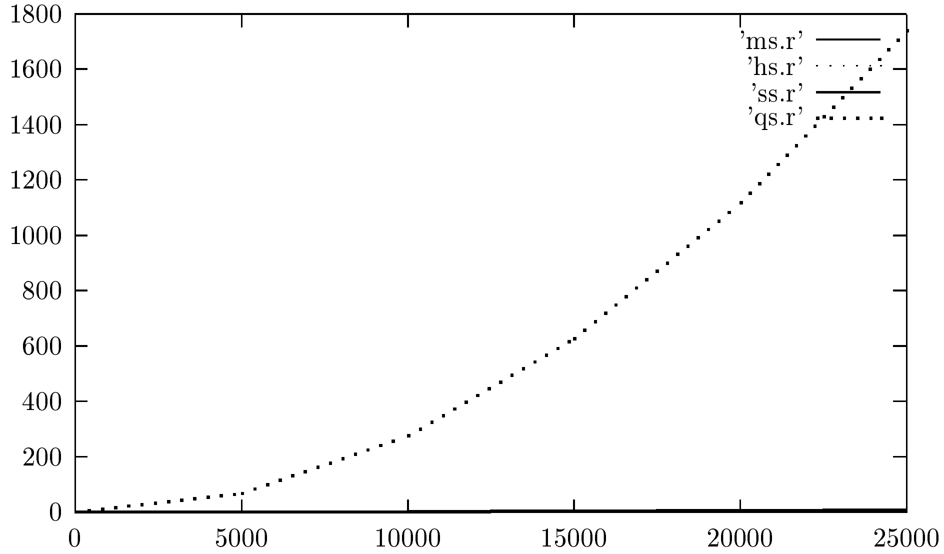

Grafico 5

3.3.2 6^o *esperimento*

Il *Grafico 6* rappresenta il costo computazionale dei tre algoritmi di ordinamento:
Mergesort, Heapsort, Shellsort senza **Quicksort 1** perché, dal *Grafico 5*, si é visto com'é il suo comportamento con sequenze contenenti numeri ordinati in modo crescente.
Si noti come **ShellSort**, abbassi pesantemente il tempo di calcolo distruggendo letteralmente gli altri due algoritmi **Mergesort** e **Heapsort**, che piú o meno si comportano come nel *Grafico 1.*

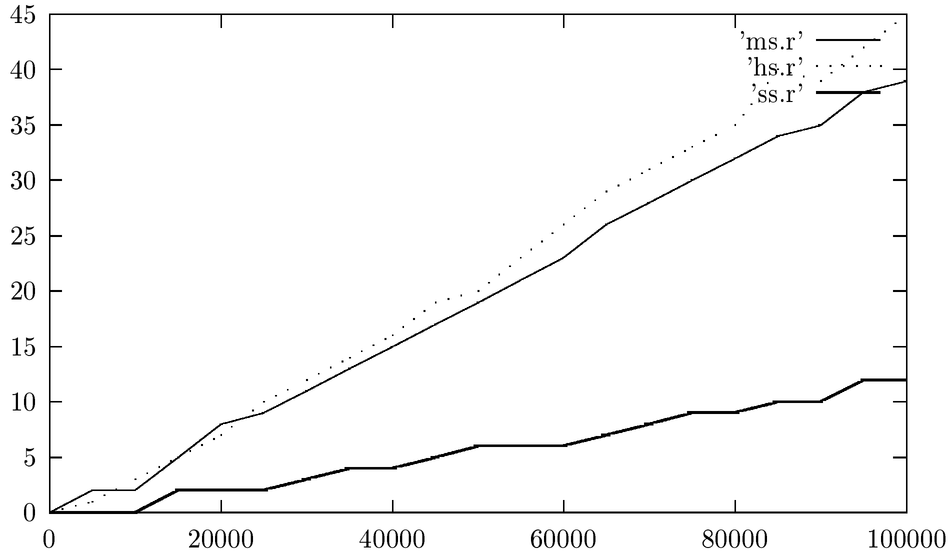

Grafico 6

3.3.3 *7º esperimento*

Il *Grafico 7* presenta le prestazioni dell'algoritmo **Quicksort** nelle quattro varianti:
Quicksort 1, Quicksort 2, Quicksort 3 e **Quicksort 4**
Si riesce a vedere molto chiaramente che il peggiore é **Quicksort 4**, seguito da **Quicksort 1**. Molto migliore é invece **Quicksort 2** che per il valore di fondoscala 25000 riesce a diminuire di circa 1300 cicli di CPU i tempi computazionali dei precedenti due. Il migliore peró é **Quicksort 3**, che ha tempi computazionali cosí bassi da non venir neanche visualizzato.

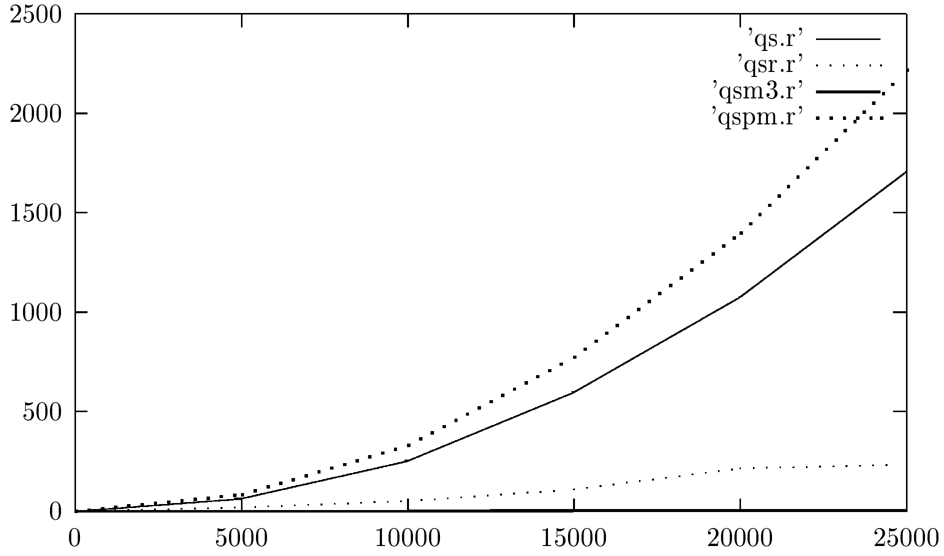

Grafico 7

3.4 Numeri Ordinati in modo *Decrescente*

3.4.1 *8º esperimento*

Il *Grafico 8* rappresenta le prestazioni dei quattro algoritmi di ordinamento:
Mergesort, Heapsort, Shellsort e **Quicksort 1**
testati con sequenze contenenti numeri ordinati in modo decrescente.
In questo caso si ripresenta una situazione analoga a quella riscontrata nel *Grafico 5*. Anche qui **QuickSort 1** risulta essere di gran lunga il peggiore con valori cosí elevati da non permettere la visualizzazione degli altri tre algoritmi.

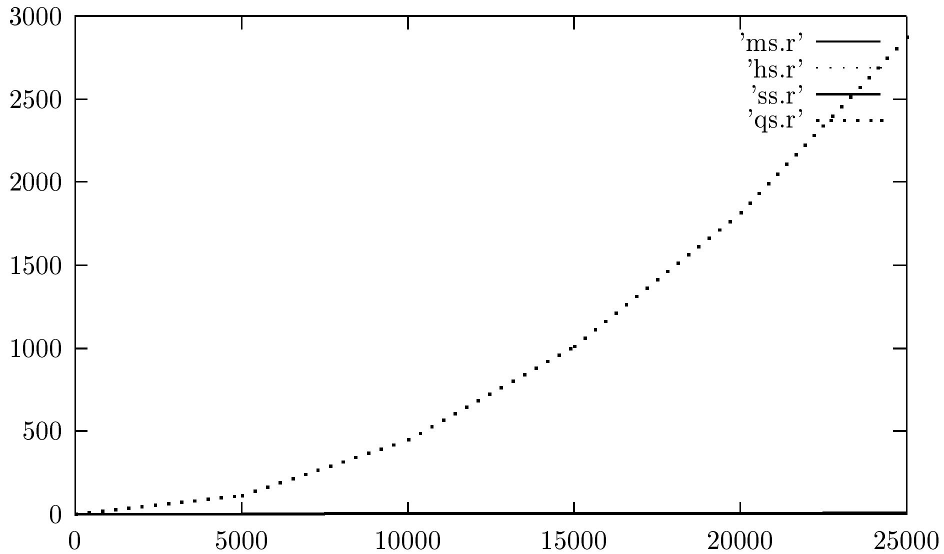

Grafico 8

3.4.2 9^o *esperimento*

Il *Grafico 9* rappresenta il costo computazionale dei tre algoritmi di ordinamento:
Mergesort, Heapsort, Shellsort senza **Quicksort 1** perché, dal *Grafico 8*, si é visto com'é il suo comportamento con sequenze contenenti numeri ordinati in modo decrescente.
Anche in questo caso sembra ripresentarsi la situazione successa nel *Grafico 6*, infatti **ShellSort**, riesce ad abbassare drasticamente il suo tempo di calcolo rispetto a **Mergesort** e **Heapsort**. Fra questi ultimi, anche se sembrano avere un andamento molto simile, risulta leggermente migliore **Heapsort**.

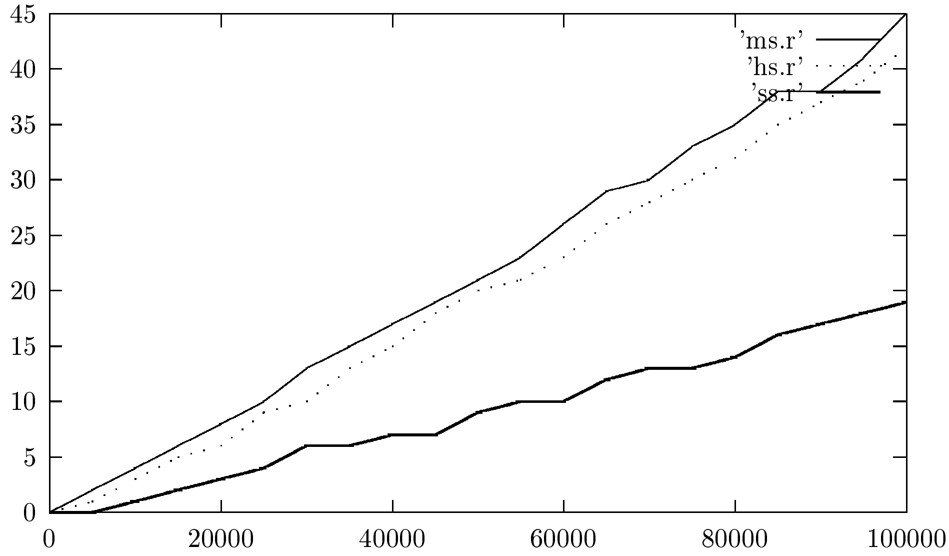

Grafico 9

3.4.3 10^o *esperimento*

Il *Grafico 10* presenta le prestazioni dell'algoritmo **Quicksort** nelle quattro varianti:
Quicksort 1, Quicksort 2, Quicksort 3 e **Quicksort 4**
Si riesce a vedere molto chiaramente che il peggiore é sempre **Quicksort 4**, seguito da **Quicksort 1**. Anche in questo caso c'é un'analogia con il *Grafico 7*, anche se, con sequenze contenenti numeri ordinati in modo decrescente, peggiora di molto le sue prestazioni **Quicksort 3** (che nel *Grafico 7* era il migliore) portando cosí al primo posto **Quicksort 2**

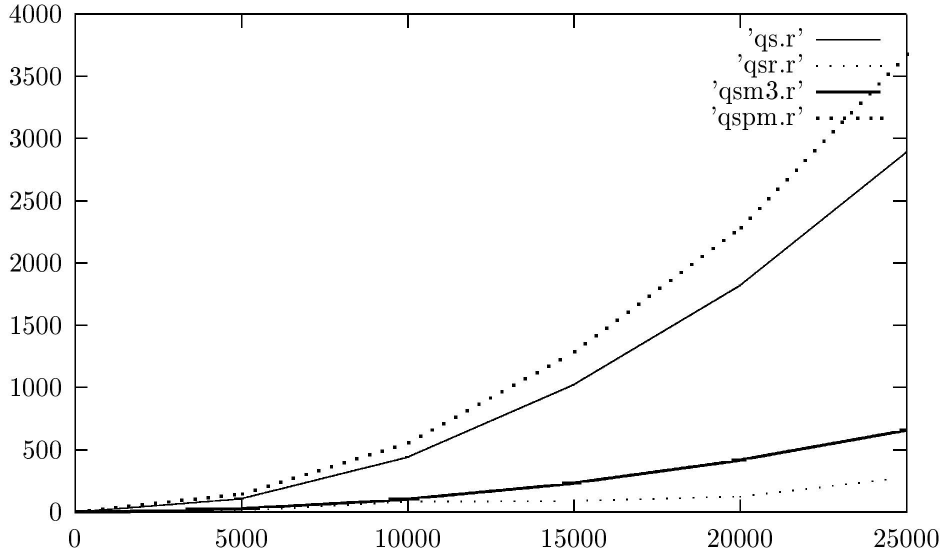

Grafico 10

4 Testing dell'applicativo

Per verificare che effettivamente il programma funzioni correttamente é necessario:

- Controllare che i valori di ingresso sui quali effettuare l'ordinamento, siano gli stessi che si ottengono in uscita dopo tale operazione.
- Validare l'ordinamento ottenuto, cioé controllare che i dati siano realmente ordinati in modo crescente.
- Considerare i risultati di tutti gli algoritmi implementati e verificare che gli ordinamenti ottenuti corrispondano.

Per testare il programma sono stati creati i seguenti semplici applicativi di supporto, che permettono di dare una risposta ai tre punti sopra specificati.

- Implementazione del primo programma *check1*

  Il programma acquisisce i nomi dei due file ed estrapola la dimensione dei dati che sono contenuti al loro interno. Se tali dimensioni sono diverse avvisa l'utente e termina, altrimenti considera ad uno ad uno tutti i valori del primo file, e con costo $O(n^2)$ controlla se tale valore é presente nell'altro file. Se cosí non é termina, altrimenti conferma che i dati sono gli stessi.

- Implementazione del secondo programma *check2*

  Il programma acquisisce il nome del file contenente i dati da controllare. Preleva il primo elemento e lo salva in number1. Successivamente considera il valore seguente e lo salva in number2. Confronta poi questi due elementi e se il secondo non é maggiore del primo, il programma lo segnala e termina. Se invece il primo é minore del secondo, in number1 viene copiato il valore di number2 e in quest'ultimo va l'elemento successivo. Si confrontano nuovamente questi due valori, e ripetendo il procedimento, si riesce a testare l'intero file. Il costo é $O(n)$ perché si devono scandire tutti gli elementi almeno una volta.

- Implementazione del terzo programma *check3*

  Il programma acquisisce i nomi dei due file ed estrapola la dimensione dei dati che sono contenuti al loro interno. Se tali dimensioni sono diverse avvisa l'utente e termina, altrimenti considera ad una ad una le coppie di elementi dei due file (quei valori cioé che si trovano alla stessa posizione) e se questi sono diversi i due ordinamenti non corrispondono e il programma termina, altrimenti conferma la correttezza dell'ordinamento. Il costo computazionale é $O(n)$ perché si considerano tutti gli elementi dei due file una sola volta.

⟨*check1.cpp*⟩≡

```
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_STR 250

int main(int argc,char* argv[])
{
  fstream of1;
  fstream of2;
  char file[MAX_STR];

  int size1;
  int size2;

  cout << "Inserire il nome del primo file (con valori ordinati): ";
  cin >> file;
  of1.open(file, ios::in|ios::binary);
  cout << "Inserire il nome del secondo file (con valori da ordinare): ";
  cin >> file;
  of2.open(file, ios::in|ios::binary);

  if((!of1)||(!of2))
  {
    cout << "Attenzione. Problemi nella apertura dei due files!!\n";
    exit(-1);;
  }

  of1.getline(file,MAX_STR,'\n');

  of1 >> size1;
  of2 >> size2;

  if(size1!=size2)
  {
    cout << "\nAttenzione: I due file sono diversi!";
    exit(-1);
  }

  of1.getline(file,MAX_STR,'\n');

  int number1;
  int number2;
  bool found;

  while(!of1.eof())
  {
    of1 >> number1;
    found=false;

    of2.seekg(1, ios::beg);
```

```
    for(int k=0;k<=size2;++k)
    {
      of2 >> number2;
      //cout << number2 << endl;
      //fflush(stdout);
      if(number1==number2)
      {
        found=true;
        break;
      }
    }

    if(found==false)
    {
      cout << "\nAttenzione: Non e' stato trovato il valore: " << number1 << endl;
      exit(-1);
    }
  }

  of1.close();
  of2.close();

  cout << "\nI due file contengono gli stessi dati\n";

  exit(0);
}
```

⟨*check2.cpp*⟩≡

```
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_STR 250

int main(int argc,char* argv[])
{
  fstream of1;
  char file[MAX_STR];
  int size1;
  cout << "Inserire il nome del file (con i valori ordinati): ";
  cin >> file;
  of1.open(file, ios::in|ios::binary);

  if(!of1)
  {
    cout << "Attenzione. Problemi nella apertura del file!!\n";
    exit(-1);;
  }
  of1.getline(file,MAX_STR,'\n');
  of1 >> size1;
  of1.getline(file,MAX_STR,'\n');

  int number1;
  int number2;

  of1 >> number1;
  while(!of1.eof())
  {
    of1 >> number2;
    if(number1>number2)
    {
      cout << "\nAttenzione: Non hanno rispettato l'ordinamento i due seguenti valori:\n";
      cout << number1 << endl << number2 <<endl;
      exit(-1);
    }
    number1=number2;
  }

  of1.close();
  cout << "\nL'ordinamento e' corretto\n";
  exit(0);
}
```

⟨*check3.cpp*⟩≡

```
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_STR 250

int main(int argc,char* argv[])
{
  fstream of1;
  fstream of2;
  char file[MAX_STR];

  int size1;
  int size2;

  cout << "Inserire il nome del primo file: ";
  cin >> file;
  of1.open(file, ios::in);
  cout << "Inserire il nome del secondo file: ";
  cin >> file;
  of2.open(file, ios::in);

  if((!of1)||(!of2))
  {
    cout << "Attenzione. Problemi nella apertura dei due files!!\n";
    exit(-1);;
  }

  of1.getline(file,MAX_STR,'\n');
  of2.getline(file,MAX_STR,'\n');
  of1 >> size1;
  of2 >> size2;

  if(size1!=size2)
  {
    cout << "\nAttenzione: I due file sono diversi!";
    exit(-1);
  }

  of1.getline(file,MAX_STR,'\n');
  of2.getline(file,MAX_STR,'\n');

  int number1;
  int number2;
  int i=3;

  while(!of1.eof())
  {
    ++i;
    of1 >> number1;
    of2 >> number2;
```

```
    if(number1!=number2)
    {
      cout << "\nAttenzione: I due file sono diversi alla linea " << i << endl;
      fflush(stdout);
      exit(-1);
    }
  }

  of1.close();
  of2.close();

  cout << "\nI due file sono uguali\n";

  exit(0);
}
```

5 Appendice

Questa sezione comprende tutto il codice e le scelte di progetto.
Si informa che é possibile ottenere tutti i sorgenti e l'eseguibile semplicemente digitando "make exec" a linea di comando.

Testi utilizzati
ALGORITMI E STRUTTURE DI DATI - Alan Bertossi - UTET Libreria
ELEMENTI DI PROGRAMMAZIONE CON IL C++ - Franco Crivellari - FrancoAngeli

5.1 Main Program

Programma principale contenente le istanze delle quattro classi:
Mergesort Quicksort Heapsort Shellsort

La seguente porzione di codice permette di prelevare da file le sequenze numeriche sulle quali applicare gli algoritmi di ordinamento. Oltre all'ordinamento vengono anche salvati i risultati relativi al tempo di utilizzo di CPU.

⟨*esame3.cpp*⟩≡
```
#include <fstream.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/times.h>
#include <string.h>
#include <stdlib.h>

#include "mergesort.h"
#include "quicksort.h"
#include "heapsort.h"
#include "shellsort.h"

#define MAX_STR 250
#define MAX_LEN 500

//Serie di funzioni C utilizzate per la gestione del video
extern "C" {
  char Tasto(void);
  void Stampa(const char*,int,int);
  void Pulisci(void);
  void InitGrafica(void);
  void EndGrafica(void);
}


int* GetInfo(int* size)
{
  char file[MAX_STR];
  int* A;
  fstream f;
```

```
  Stampa("Inserire il nome del file contenente i valori da ordinare:",11,2);
  cin >> file;
  Stampa("Inizio Computazione",13,2);
  Stampa("Attendere prego...",14,2);

  f.open(file, ios::in);
  if(!f)
  {
    Stampa("Attenzione. Problemi in apertura del file",15,2);
    Tasto();
    EndGrafica();
    return NULL;
  }

  //acquisisco quanti numeri ho da ordinare
  f >> (*size);

  //alloco la memoria necessaria
  A=new(int[(*size)]);

  int i=0;
  while((!f.eof())&&(i<(*size)))
  {
    f >> A[i];
    ++i;
  }

  f.close();
  return A;
}


int main(int argc,char* argv[])
{
  struct tms tms1;
  struct tms tms2;
  fstream of1;
  fstream of2;
  fstream of3;
  fstream of4;
  fstream of5;
  fstream of6;
  fstream of7;

  int size;

  MergeSort* MS =new MergeSort();
  QuickSort* QS =new QuickSort();
  HeapSort*  HS =new HeapSort();
  ShellSort* SS =new ShellSort();

  int* A;
```

```
 char scelta;
 int ok;

 InitGrafica();

while(scelta!=27)
{
 Pulisci();

 Stampa("Algoritmi di Ordinamento - Federico Bertolli - Esame di Lab. di Algoritmi 1",2,2);
 Stampa("Scegliere quale dei seguenti algoritmi applicare",3,2);
 Stampa(" a  -> MergeSort",5,2);
 Stampa(" b  -> HeapSort",6,2);
 Stampa(" c  -> ShellSort",7,2);
 Stampa(" d  -> QuickSort",8,2);
 Stampa(" e  -> QuickSort R",8,20);
 Stampa(" f  -> QuickSort M3",9,2);
 Stampa(" g  -> QuickSort PM",9,20);
 Stampa("Esc -> Exit",10,2);

 Stampa("Scelta: ",11,2);

 scelta=Tasto();

 switch(scelta)
 {
   case 'a': A=GetInfo(&size);

             if(A==NULL)
               continue;

             //of1.open("ms.r", ios::app|ios::out);
             of1.open("ms.r", ios::out);
             if(!of1)
             {
               Stampa("Attenzione. Errore nella creazione del file di output!",15,2);
               Tasto();
               EndGrafica();
               exit(-1);;
             }
             of1 << "Cardinalita' | tempo di calcolo\n";
             times(&tms1);
             MS->Sort(A,0,size-1);
             times(&tms2);
             of1 << size << " " << tms2.tms_utime - tms1.tms_utime << endl << endl;

             //salvo i risultati su file
             for(int i=0;i<size;++i)
               of1 << A[i] << endl;

             of1.close();
             Stampa("I risultati sono stati salvati nel file 'ms.r'",16,2);
```

```
          Tasto();

          break;

case 'b': A=GetInfo(&size);

          if(A==NULL)
            continue;

          //of2.open("hs.r", ios::app|ios::out);
          of2.open("hs.r", ios::out);
          if(!of2)
          {
            Stampa("Attenzione. Errore nella creazione del file di output!",15,2);
            Tasto();
            EndGrafica();
            exit(-1);;
          }
          of2 << "Cardinalita' | tempo di calcolo\n";
          times(&tms1);
          HS->Sort(&A,size);
          times(&tms2);
          of2 << size << " " << tms2.tms_utime - tms1.tms_utime << endl << endl;

          //salvo i risultati su file
          for(int i=0;i<size;++i)
            of2 << A[i] << endl;

          of2.close();
          Stampa("I risultati sono stati salvati nel file 'hs.r'",15,2);
          Tasto();

          break;

case 'c': A=GetInfo(&size);

          if(A==NULL)
            continue;

          //of3.open("ss.r", ios::app|ios::out);
          of3.open("ss.r", ios::out);
          if(!of3)
          {
            Stampa("Attenzione. Errore nella creazione del file di output!",15,2);
            Tasto();
            EndGrafica();
            exit(-1);;
          }

          of3 << "Cardinalita' | tempo di calcolo\n";
          times(&tms1);
          SS->Sort(A,size);
```

```
          times(&tms2);
          of3 << size << " " << tms2.tms_utime - tms1.tms_utime << endl << endl;

          //salvo i risultati su file
          for(int i=0;i<size;++i)
            of3 << A[i] << endl;

          of3.close();
          Stampa("I risultati sono stati salvati nel file 'ss.r'",15,2);
          Tasto();

          break;

case 'd': A=GetInfo(&size);

          if(A==NULL)
            continue;

          //of4.open("qs.r", ios::app|ios::out);
          of4.open("qs.r", ios::out);
          if(!of4)
          {
            Stampa("Attenzione. Errore nella creazione del file di output!",15,2);
            Tasto();
            EndGrafica();
            exit(-1);;
          }
          of4 << "Cardinalita' | tempo di calcolo\n";
          times(&tms1);
          QS->Sort(A,0,size-1,0);
          times(&tms2);
          of4 << size << " " << tms2.tms_utime - tms1.tms_utime << endl << endl;

          //salvo i risultati su file
          for(int i=0;i<size;++i)
            of4 << A[i] << endl;

          of4.close();
          Stampa("I risultati sono stati salvati nel file 'qs.r'",15,2);
          Tasto();

          break;

case 'e': A=GetInfo(&size);

          if(A==NULL)
            continue;

          //of5.open("qsr.r", ios::app|ios::out);
          of5.open("qsr.r", ios::out);
          if(!of5)
          {
```

```
        Stampa("Attenzione. Errore nella creazione del file di output!",15,2);
        Tasto();
        EndGrafica();
        exit(-1);;
      }
      of5 << "Cardinalita' | tempo di calcolo\n";
      times(&tms1);
      QS->Sort(A,0,size-1,1);
      times(&tms2);
      of5 << size << " " << tms2.tms_utime - tms1.tms_utime << endl << endl;

      //salvo i risultati su file
      for(int i=0;i<size;++i)
        of5 << A[i] << endl;

      of5.close();
      Stampa("I risultati sono stati salvati nel file 'qsr.r'",15,2);
      Tasto();

      break;

case 'f': A=GetInfo(&size);

      if(A==NULL)
        continue;

      //of6.open("qsm3.r", ios::app|ios::out);
      of6.open("qsm3.r", ios::out);
      if(!of2)
      {
        Stampa("Attenzione. Errore nella creazione del file di output!",15,2);
        Tasto();
        EndGrafica();
        exit(-1);;
      }
      of6 << "Cardinalita' | tempo di calcolo\n";
      times(&tms1);
      QS->Sort(A,0,size-1,2);
      times(&tms2);
      of6 << size << " " << tms2.tms_utime - tms1.tms_utime << endl << endl;

      //salvo i risultati su file
      for(int i=0;i<size;++i)
        of6 << A[i] << endl;

      of6.close();
      Stampa("I risultati sono stati salvati nel file 'qsm3.r'",15,2);
      Tasto();

      break;

case 'g': A=GetInfo(&size);
```

```
              if(A==NULL)
                continue;

              //of7.open("qspm.r", ios::app|ios::out);
              of7.open("qspm.r", ios::out);
              if(!of7)
              {
                Stampa("Attenzione. Errore nella creazione del file di output!",15,2);
                Tasto();
                EndGrafica();
                exit(-1);;
              }
              of7 << "Cardinalita' | tempo di calcolo\n";
              times(&tms1);
              QS->Sort(A,0,size-1,3);
              times(&tms2);
              of7 << size << " " << tms2.tms_utime - tms1.tms_utime << endl << endl;

              //salvo i risultati su file
              for(int i=0;i<size;++i)
                of7 << A[i] << endl;

              of7.close();
              Stampa("I risultati sono stati salvati nel file 'qspm.r'",15,2);
              Tasto();

              break;
  }
 }
 EndGrafica();
 exit(0);
}
```

5.2 Classe MergeSort

Files che dichiarano e definiscono la classe

- mergesort.h
- mergesort.cpp

La seguente porzione di codice contiene la dichiarazione della classe **Mergesort**

$\langle$*mergesort.h*$\rangle\equiv$

```
#ifndef MERGESORT_H
#define MERGESORT_H

#include <fstream.h>

class MergeSort
{
  private:
    void Merge(int*,int,int,int);

  public:
    MergeSort();
    ~MergeSort();

    void Sort(int*,int,int);
};

#endif
```

La seguente porzione di codice contiene la definizione della classe **Mergesort**

⟨*mergesort.cpp*⟩≡

```
//implementazione della classe Merge

#include "mergesort.h"

MergeSort::MergeSort()
{
}

MergeSort::~MergeSort()
{
}

void MergeSort::Sort(int* A,int first,int last)
{
  int medium;

  if(first<last)
  {
    medium=(first+last)/2;
    Sort(A,first,medium);
    Sort(A,medium+1,last);
    Merge(A,first,last,medium);
  }
}

void MergeSort::Merge(int* A,int first,int last,int medium)
{
  int i,j,k,h;
  int* B;
  //Alloco la memoria del vettore ausiliario
  B=new(int[last-first+1]);

  i=first;
  j=medium+1;
  k=0;

  while((i<=medium)&&(j<=last))
  {
    if(A[i]<=A[j])
    {
      B[k]=A[i];
      ++i;
    }
    else
    {
      B[k]=A[j];
      ++j;
    }
    ++k;
  }
```

```
  //k e' la posizione in cui e' arrivato il vettore B

  //I valori non considerati della prima meta' devo scriverli in B comunque!
  //Questo per non avere perdita di informazione
  for(h=i;h<=medium;++h)
  {
    B[k]=A[h];
    ++k;
  }

  //i risultati li voglio cmq in A e copio i soli valori scritti in B
  for(int y=first;y<k+first;++y)
  {
    A[y]=B[y-first];
  }

  delete [] B;
}
```

5.3 Classe Quicksort

Files che dichiarano e definiscono la classe

- quicksort.h
- quicksort.cpp

La seguente porzione di codice contiene la dichiarazione della classe **Quicksort**

⟨*quicksort.h*⟩≡

```
#ifndef QUICKSORT_H
#define QUICKSORT_H

#include <time.h>
#include <stdlib.h>
#include <iostream.h>

class QuickSort
{
  private:
     //funzione che considera il primo elemento come perno
     int Perno(int*,int,int);
     //funzione che considera l'elemento mediano come perno
     int PernoPM(int*,int,int);
     //Funzione per la ricerca lineare del punto medio
     int Select(int*,int,int,int);

     //funzione che considera un elemento casuale come perno
     int PernoR(int*,int,int);
     //funzione che considera l'elemento mediano fra il primo, l'ultimo e il mezzo come perno

int PernoM3(int*,int,int);
  public:
     QuickSort();
     ~QuickSort();

     void Sort(int*,int,int,int);
};

#endif
```

La seguente porzione di codice contiene la definizione della classe **Quicksort**

⟨*quicksort.cpp*⟩≡

```
//implementazione della classe Quick

#include "quicksort.h"

QuickSort::QuickSort()
{
  srand(time(NULL));
}

QuickSort::~QuickSort()
{
}



/*
  Il parametro "mode" serve per specificare quale funzione perno utilizzare
  0 -> normale
  1 -> casuale
  2 -> mediano fra primo, ultimo e mezzo
  3 -> mediano fra tutti
*/
void QuickSort::Sort(int* A,int first,int last,int mode)
{
  //conterra' il nuovo valore del perno dopo la chiamata alla Perno()
  int j;

  switch(mode)
  {
    case 0: if(first<last)
            {
              j=Perno(A,first,last);
              Sort(A,first,j-1,0);
              Sort(A,j+1,last,0);
            }
            break;
    case 1: if(first<last)
            {
              j=PernoR(A,first,last);
              Sort(A,first,j-1,1);
              Sort(A,j+1,last,1);
            }
            break;
    case 2: if(first<last)
            {
              j=PernoM3(A,first,last);
              Sort(A,first,j-1,2);
              Sort(A,j+1,last,2);
            }
            break;
    case 3: if(first<last)
```

```
            {
              j=PernoPM(A,first,last);
              Sort(A,first,j-1,3);
              Sort(A,j+1,last,3);
            }
            break;
  }
}


//Perno implementata con la ricerca lineare del punto medio
int QuickSort::PernoPM(int* A,int first,int last)
{
  int x,m,l,temp;

  //scambio il primo valore con il mediano calcolato da Select
  x=Select(A,first,last,(last-first)/2);

  temp=A[first];
  A[first]=A[x];
  A[x]=temp;

  m=A[first];
  l=first;

  for(int i=first+1; i<=last; i++)
  {
    if(A[i]<m)
    {
      l++;

      //scambio l e i
      temp=A[l];
      A[l]=A[i];
      A[i]=temp;
    }
  }

  //scambio il primo con l
  temp=A[first];
  A[first]=A[l];
  A[l]=temp;

  m=l;
  return m;
}


int QuickSort::Select(int* A,int first,int last,int k)
{
  int q,j;
```

```
  if(first==last)
    return(first);
  else
  {
    j=Perno(A,first,last);
    //numero di elementi minori del perno relativi al sottovettore considerato
    q=j-first+1;
    if(k<=q)
      return(Select(A,first,j,k));
    else
      return(Select(A,j+1,last,k-q));
  }
}

int QuickSort::Perno(int* A,int first,int last)
{
  int x,i,j,temp;

  //In x metto il primo valore
  x=A[first];
  j=first;

  for(i=first+1;i<=last;++i)
    if(A[i]<x)
    {
      ++j;
      temp=A[j];
      A[j]=A[i];
      A[i]=temp;
    }

  //scambio l'elemento in prima posizione con quello in posizione perno
  A[first]=A[j];
  A[j]=x;
  return(j);
}

int QuickSort::PernoR(int* A,int first,int last)
{
  int x,i,j,temp;

  //Calcolo l'elemento casuale
  i=first + (int)((last-first+1.0)*rand()/(RAND_MAX+1.0));

  //scambio l'elemento casuale precedentemente calcolato con il primo
  temp=A[first];
  A[first]=A[i];
  A[i]=temp;

  //in x metto il primo valore che ora sara' quello casuale
  x=A[first];
  j=first;
```

```
  for(i=first+1;i<=last;++i)
    if(A[i]<x)
    {
      ++j;
      temp=A[j];
      A[j]=A[i];
      A[i]=temp;
    }

  //scambio l'elemento in prima posizione con quello in posizione perno
  A[first]=A[j];
  A[j]=x;
  return(j);
}

int QuickSort::PernoM3(int* A,int first,int last)
{
  int x,y,z,i,j,temp;

  //Salvo in x,y,z rispettivamente il primo elemento, l'ultimo e quello centrale
  x=A[first];
  y=A[(first+last)/2];
  z=A[last];

  //Ora li confronto scegliendo il mediano dei tre
  if(((x>y)&&(x<z)&&(y<z))||((x<y)&&(x>z)&&(y>z)))
  {
    //Prendo il primo
    //in x c'e' gia' il valore corretto
  }
  else if(((x<y)&&(x<z)&&(y>z))||((x>y)&&(x>z)&&(y<z)))
  {
    //Prendo il terzo
    A[first]=z;
    A[last]=x;
  }
  else //if(((x<y)&&(x<z)&&(y<z))||((x>y)&&(x>z)&&(y>z)))
  {
    //Prendo il centrale
    A[first]=y;
    A[(first+last)/2]=x;
  }

  x=A[first];
  j=first;

  for(i=first+1;i<=last;++i)
    if(A[i]<x)
    {
      ++j;
      temp=A[j];
```

```
        A[j]=A[i];
        A[i]=temp;
      }

    //scambio l'elemento in prima posizione con quello in posizione perno
    A[first]=A[j];
    A[j]=x;
    return(j);
  }
```

5.4 Classe Heapsort

Files che dichiarano e definiscono la classe

- heapsort.h
- heapsort.cpp

La seguente porzione di codice contiene la dichiarazione della classe **Heapsort**

$\langle$*heapsort.h*$\rangle\equiv$

```
#ifndef HEAPSORT_H
#define HEAPSORT_H

class HeapSort
{
  private:
    int* heap;

    void RestauraHeap(int,int);
    void CostruisciHeap(int);

  public:
    HeapSort();
    ~HeapSort();

    void Sort(int**,int);
};

#endif
```

La seguente porzione di codice contiene la definizione della classe **Heapsort**

⟨*heapsort.cpp*⟩≡

```
//implementazione della classe HeapSort

#include "heapsort.h"

HeapSort::HeapSort()
{
}

HeapSort::~HeapSort()
{
}

void HeapSort::RestauraHeap(int i, int j)
{
  int k,temp;

  if(i<=j/2)
  {
    k=2*i;
    if(k<j)
      if(heap[k-1]<heap[k])
        ++k;

    if(heap[k-1]>heap[i-1])
    {
      temp=heap[i-1];
      heap[i-1]=heap[k-1];
      heap[k-1]=temp;
      RestauraHeap(k,j);
    }
  }
}

void HeapSort::CostruisciHeap(int n)
{
  for(int i=n/2;i>0;--i)
    RestauraHeap(i,n);
}


void HeapSort::Sort(int** A,int n)
{
  int i,temp;

  heap=*A;

  CostruisciHeap(n);

  for(i=n;i>1;--i)
  {
```

```
    temp=heap[i-1];
    heap[i-1]=heap[0];
    heap[0]=temp;
    RestauraHeap(1,i-1);
  }
}
```

5.5 Classe ShellSort

Files che dichiarano e definiscono la classe

- shellsort.h
- shellsort.cpp

La seguente porzione di codice contiene la dichiarazione della classe **Shellsort**

$\langle$*shellsort.h*$\rangle\equiv$

```
#ifndef SHELLSORT_H
#define SHELLSORT_H

class ShellSort
{
  private:

  public:
    ShellSort();
    ~ShellSort();

    void Sort(int*,int);
};

#endif
```

La seguente porzione di codice contiene la definizione della classe **Shellsort**

⟨*shellsort.cpp*⟩≡

```
//implementazione della classe ShellSort

#include "shellsort.h"

ShellSort::ShellSort()
{
}

ShellSort::~ShellSort()
{
}

void ShellSort::Sort(int* A,int n)
{
  int i,j,k,h;
  bool out;

  h=1;
  //cerco h il piu' grande possibile rispetto a n
  while(h<=n)
    h=3*h+1;

  h/=3;

  while(h>=1)
  {
    for(i=h+1;i<=n;++i)
    {
      k=A[i-1];
      j=i;
      out=false;

      while((A[j-h-1]>k)&&(!out))
      {
        A[j-1]=A[j-h-1];
        j-=h;
        if(j<=h)
        {
          j+=h;
          out=true;
        }
      }
      if(out==true)
        A[j-h-1]=k;
      else
        A[j-1]=k;
    }
    //consideriamo ora il caso con h minore
    h/=3;
  }
```

```
}
```

5.6 Generator

- generator.cpp

La seguente porzione di codice permette la generazione automatica di *numeri casuali* con il relativo salvataggio su file di testo.

⟨*generator.cpp*⟩≡

```
#include <fstream.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char* argv[])
{
  fstream f;
  int n;
  char file[100];

  cout << "\nInserire il nome del file in cui inserire i numeri:\n";
  cin >> file;
  cout << "\nInserire quanti numeri generare:\n";
  cin >> n;

  f.open(file, ios::trunc|ios::out);
  if(!f)
  {
    cout << "\nErrore nella creazione del file!\n";
    exit(-1);
  }

  f << n << endl;

  srand(time(NULL));
  for(int i=0;i<n;++i)
    f << rand() << endl;

  f.close();
}
```

5.7 Modulo Utilita

Il modulo *Utilita* permette una gestione migliore della grafica modalitá testo sui terminali Linux per rendere il programma piú user friendly.

Files che dichiarano e definiscono il modulo C

- utilita.h
- utilita.c

La seguente porzione di codice contiene la dichiarazione del modulo **Utilita**

$\langle$*utilita.h*$\rangle\equiv$

```
#ifndef UTILITA_H
#define UTILITA_H

#include <curses.h>
#include <stdlib.h>


//Funzione che consente di acquisire un char con la sola pressione del tasto
//senza la visualizzazione dello stesso a video e senza #13 di conferma
char Tasto(void);

//Funzione che permette di sampare un char* alle coordinate (x,y) desiderate
void Stampa(const char*,int,int);

//Funzione che pulisce il video
void Pulisci(void);

//Funzione che serve per l'inizializzazione del video per poter usare le
//funzionalita' della curses.h
void InitGrafica(void);

//Funzione che chiude la finestra attiva
void EndGrafica(void);

#endif
```

La seguente porzione di codice contiene la definizione del modulo **Utilita**

⟨*utilita.c*⟩≡
```
#include "utilita.h"

//Inizializzazione grafica
void InitGrafica(void)
{
 initscr ();
 cbreak ();
 refresh ();
}

//Chiusura grafica
void EndGrafica(void)
{
 endwin();
}

//Funzione che permette l'acquisizione di un char con la semplice pressione del tasto
char Tasto (void)
{
  char c;
  noecho ();
  c=(char) getch ();
  echo ();
  return(c);
}

//Funzione che stampa un char* alle coordinate volute
void Stampa(const char* s,int x,int y)
{
 move(x,y);
 addstr(s);
 //Attenzione
 refresh();
 echo ();
}

//Funzione che pulisce lo schermo
void Pulisci(void)
{
 clear();
 refresh();
}
```