Corso di Laboratorio di Algoritmi - AA 2000 - 2001
Algoritmi di ordinamento
Counting Sort, Radix Sort e Bucket Sort

Paolo Zotti 62SI

December 17, 2001

1 Struttura del progetto

1.1 La relazione

Questa relazione fornisce un'ampia introduzione allo studio di alcuni algoritmi presentandoli e trattandoli in modo approfondito. Inoltre rende la loro progettazione ed analisi accessibili a lettori di ogni livello.
La caratteristica principale di questa classe di algoritmi è quella di ordinare i dati senza usare confronti e scambi.
Ho cercato di mantenere le spiegazioni a livello elementare senza sacrificare l'ampiezza della trattazione o il rigore matematico.
Ogni sezione ad esclusione della parte introduttiva, presenta un algoritmo, una tecnica di progettazione, un listato in linguaggio C++ e alcuni test effettuati allo scopo di verificare diverse proprietà degli algoritmi stessi.
Gli algoritmi sono descritti con uno pseudo codice pensato per essere leggibile da chiunque abbia fatto un po' di programmazione, e sono supportati da ampie spiegazioni ed esempi.
Poichè l'efficienza va considerata determinante nei criteri di progettazione di un algoritmo, ho incluso un'analisi del tempo di esecuzione.
I 3 algoritmi di ordinamento presi in analisi sono i seguenti:

- **Counting Sort**
- **Radix Sort**
- **Bucket Sort**

1.2 Il codice

L'applicativo di questo progetto è stato scritto in C++. Ho voluto una forte suddivisione in classi per rendere piú semplice la comprensione del codice e per avere una netta separazione delle varie parti.
Infatti, per ogni algoritmo implementato è stata costruita una classe contenente i propri metodi ed attributi.

- Per l'algoritmo di ordinamento **Counting Sort** è stata implementata una classe che ordina i numeri di tipo *int* in un range $[0..(2^{10} - 1)]$;
- per l'algoritmo di ordinamento **Radix Sort** è stata implementata una classe che ordina i numeri di tipo *int* in un range $[0..(2^{31} - 1)]$;
- per l'algoritmo di ordinamento **Bucket Sort** è stata implementata una classe che ordina i numeri di tipo *int* in un range$[0..(2^{31} - 1)]$ usando come algoritmo di ordinamento intermedio **Radix Sort**.
  Decidere di progettare **Bucket Sort** (utilizzando per l'ordinamento intermedio, un algoritmo non basato sul confronto) ho scoperto essere un errore perchè và contro la natura stessa di questo algoritmo.
  In nessun libro di testo da me analizzato è specificata questa problematica.
  Precisamente, nel testo "Introduzione agli algoritmi" di Thomas Cormen, nella sezione riservata a Bucket Sort, (pag 170, sez 9.4), lo pseudo codice utilizza per l'ordinamento intermedio, Insert Sort, che basa il suo funzionamento sul confronto.
  Nel testo non è specificata ne spiegata la necessità o meglio, l'obbligo, di usare per l'ordinamento intermedio, un algoritmo basato sul confronto e vice versa, l'assurdità di usare per l'ordinamento intermedio un algoritmo non basato sul confronto.
  Anche nel testo "The Art of Computer Programming" di Donal E. Knuth, (pag 169 sez 5.2.5) non vi è alcun riferimento a questo problema.
  Lo stesso dicasi per il materiale recuperato dalla rete.
  Sarebbe interessante capire se si tratta di una "svista concettuale" o di qualcosa di piú serio.
  Comunque ho ampiamente dimostrato nella se 3.3.9 che Le spiegazioni a questo problema sono nella sezione relativa ai test su **Bucket Sort**.
  La scelta di progetto effettuata ha le seguenti motivazioni:
  - **Radix Sort**, oltre ad essere un ottimo algoritmo, opera in un range molto ampio, $[0..(2^{31} - 1)]$;
  - la ricerca studia e prende in analisi gli ordinamenti non basati sul confronto.
    Anche se **Bucket Sort** per sua natura deve ordinare tipi di dato in virgola mobile e deve implementare un algoritmo di ordinamento intermedio basato sul confronto, (si capirà in seguito il perché), ho deciso di implementarlo con **Radix Sort** per non uscire dall'argomento di questa relazione.
    Come si vedrà nei test, sperimentare il comportamento di **Bucket Sort** nell'ordinare i dati usando un algoritmo intermedio non basato sul confronto ha dato dei risultati curiosi. Dato che l'algoritmo **Bucket Sort** richiede come supporto una lista a doppio concatenamento, ho realizzato la classe *Lista* che crea e gestisce oggetti di tipo **Liste a doppio concatenamento**;
- il listato principale che coordina tutto il programma è *Main*.
- per ottenere delle sequenze numeriche casuali necessarie agli algoritmi di ordinamento ho creato una classe chiamata *Rand* che utilizza un generatore di numeri casuale in un range $[0..(2^{31} - 1)]$.

1.3 Testing degli algoritmi

Solo attraverso la sperimentazione è possibile determinare se le prestazioni degli algoritmi verificano il tempo di esecuzione ricavato attraverso formule matematiche, e a tal fine è stato necessario eseguire alcuni test.

- I numeri usati nei test sono stati ottenuti con un generatore a distribuzione uniforme avente range $[0..(2^{31}-1)]$;
- gli algoritmi sono stati testati con numeri interi casuali, ordinati in modo crescente e in modo decrescente;
- i test sono stati effettuati su una macchina montante un processore "Intel" Pentium 3 con clock a 750 MHz e 64 MB RAM e il sistema operativo "Linux Mandrake Versione 8.0";
- al fine di ottenere validi risultati sperimentali ogni test, è stato eseguito 5 volte, infatti i dati relativi ai grafici sono la media matematica dei risultati ottenuti dai 5 esperimenti.
  Non ho riscontrato differenze significative tra le 5 esecuzioni.
  Questo può 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 caricamento e salvataggio dei dati su file non viene conteggiato;

Per ulteriori dettagli, si veda ogni singolo esperimento.

1.4 Ottenere la documentazione e l'eseguibile

È possibile ottenere:

- tutti i sorgenti, l'eseguibile e la documentazione digitando "make all" a linea di comando;
- tutti i sorgenti e l'eseguibile, **main**, digitando "make code" a linea di comando;
- la documentazione, **relazione.dvi e relazione.ps** digitando "make rel" a linea di comando.

2 Introduzione all'ordinamento

Qualsiasi procedura computazionale ben definita che prende alcuni valori come input e produce alcuni valori come output è chiamata algoritmo.
Un **algoritmo** è quindi una sequenza di passi computazionali che trasformano l'input nell'output.
L'**ordinamento**, consiste nel disporre un gruppo di informazioni in ordine crescente o decrescente utilizzando un algoritmo.
In genere quando si esegue un ordinamento, come chiave dell'ordinamento viene utilizzata solo una parte delle informazioni disponibili.
La **chiave** è quella parte dei dati che determina la posizione relativa dei vari elementi quando ordinati.
Quindi nel confronto fra gli elementi viene utilizzata solo la chiave, ma nello scambio dei dati viene trasferita l'intera struttura.
Ad esempio, in un database anagrafico, si puó decidere di ordinare i record utilizzando come chiave, il nome, il cognome o la data di nascita.
Cambiando la chiave i record risulteranno ordinati in modo diverso.
Vi sono due diverse categorie di algoritmi di ordinamento, una che ordina oggetti ad accesso casuale come gli array o i file ad accesso diretto, l'altra che ordina oggetti sequenziali quali file su disco o su nastro o liste concatenate.
In questa sede, verrà trattata solamente la prima categoria.

2.1 Le classi degli algoritmi di ordinamento

Vi sono tre metodi generali per ordinare le informazioni:

- scambio;
- selezione;
- inserimento;

per comprendere appieno il funzionamento di questi tre metodi, si provi a immaginare un mazzo di carte.
Per ordinare le carte utilizzando un algoritmo di:

- **scambio**: si devono disporre le carte allineate sul tavolo e quindi scambiare le carte non ordinate, procedendo fino a ordinare l'intero mazzo;
- **selezione**: si devono spargere le carte sul tavolo, poi si deve selezionare la carta con il valore piú basso, estrarla dal mazzo e tenerla in mano. Quindi, dalle carte che rimangono sul tavolo si deve selezionare la carta piú bassa e posizionarla dietro a quella già in mano. Questo processo continua fino ad avere in mano tutte le carte. Al termine del processo tutte le carte in mano risulteranno ordinate;
- **inserimento**: si devono tenere tutte le carte in mano. Poi si deve posare una carta alla volta sul tavolo inserendola nella posizione corretta. Quando saranno state scartate tutte le carte il mazzo sarà ordinato.

2.2 La complessità degli algoritmi di ordinamento

Per risolvere un problema, spesso sono disponibili diversi algoritmi. Come scegliere il migliore?
Un criterio generalmente adottato, consiste nel valutare la validità di un algoritmo in base alla quantità di risorse utilizzate per il calcolo.
La risorsa di interesse dominante è il tempo richiesto per eseguire le operazioni elementari, ma a volte è considerato anche lo spazio necessario per memorizzare e manipolare i dati.
In pratica, non è completamente soddisfacente o significativo misurare il tempo di calcolo di un algoritmo, con il numero di secondi richiesto per eseguire una procedura su un calcolatore elettronico. Infatti tale tempo, dipende pesantemente dai dati in ingresso, dal linguaggio in cui la procedura è descritta, dalla qualità con cui viene tradotto automaticamente dal compilatore in una sequenza di istruzioni macchina e dalla natura e velocità del calcolatore elettronico.
Quando possibile, si preferisce fare riferimento anche, od in alternativa, ad una misura astratta che tenga

maggiormente in considerazione il metodo di risoluzione con cui l'algoritmo effettua la computazione.
Poichè i problemi da risolvere hanno una dimensione che dipende dalla grandezza dei dati in ingresso, viene spontaneo esprimere il tempo di calcolo come il costo complessivo delle operazioni elementari in funzione della dimensione n dei dati in ingresso.
Sono considerate **elementari**, le operazioni aritmetiche, logiche, di confronto e assegnamento.
Il costo delle operazioni è valutato principalmente nel caso **peggiore** cioè sul dato di ingresso meno favorevole.
Anche se il caso peggiore potrebbe, in linea di principio, verificarsi molto raramente, viene effettuata perché presenta il vantaggio di garantire che l'algoritmo non richiederà mai per nessun dato di ingresso di dimensione n un tempo maggiore. Nel valutare il tempo di calcolo $T(n)$ di una procedura, è in genere molto difficile quantificare con esattezza il numero di operazioni elementari eseguite. Questa difficoltà viene aggirata valutando il numero di operazioni in ordine di grandezza , cioè esprimendolo come limitazione della funzione $T(n)$ al tendere all'infinito della dimensione n, trascurando le costanti moltiplicative ed addittive. Si parla di complessità computazionale asintotica in ordine di grandezza (o brevemente, complessità) di una procedura. A tal fine si usano le notazioni "O", "Ω", "Θ" definite come segue:

- "O", indica una limitazione superiore cioè il caso peggiore;
- "Ω", indica una limitazione inferiore cioè il caso migliore;
- "Θ", indica una limitazione sia superiore che inferiore, specificando cioè che il tempo di calcolo nel caso peggiore e migliore si equivalgono, come è il caso della classe di algoritmi analizzati in questa ricerca.

3 Descrizione degli algoritmi utilizzati

3.1 Counting Sort

Counting Sort si basa sull'ipotesi che ognuno degli n elementi di input sia un intero nell'intervallo da 1 a k, dove k è un intero non troppo grande.
L'idea di base di **Counting Sort** è quella di determinare per ogni elemento x in input, il numero di elementi minori di x. Questa informazione può essere usata per porre l'elemento x direttamente nella sua posizione nell'array di output.
Per esempio, se vi sono 10 elementi minori di x allora x va messo in undicesima posizione dell'array di output. Questo schema deve essere modificato per gestire situazioni in cui alcuni elementi abbiano lo stesso valore infatti, non si vuole metterli tutti nella stessa posizione.
Counting Sort è veloce perché fa delle ipotesi sull'input infatti, assume che l'input consista di numeri interi in un piccolo intervallo.
Una proprietà importante di **Counting Sort**, è quella di essere un algoritmo di ordinamento *stabile*, cioè elementi con lo stesso valore compaiono nell'array di output nello stesso ordine in cui compaiono in quello di input. In pratica, lo spareggio tra due elementi con lo stesso valore avviene secondo la regola per cui qualunque numero compaia per primo nell'array di input appare per primo nell'array di output.
Sui testi che o analizzato, questo algoritmo viene descritto e implementato sempre allo stesso modo, come di seguito riportato:
Lo pseudo codice di Counting Sort è il seguente, dove assumo che:

- input[n], array che contiene i numeri da ordinare;
- ordinati[n], array che contiene i numeri ordinati;
- temp[k], array temporaneo di lavoro;
- n, numero degli elementi da ordinare;
- k, massimo numero trovato nell'array input[ ];
- si suppone che la prima posizione di ogni vettore sia vettore[1].

Counting Sort (input,n,k)

```
 1    )   for i = 1 to n
 2    )     do ordinati[i]=0
 3    )   for i = 1 to k
 4    )     do temp[i]=0
 5    )   for j =1 to n
 6    )     do temp[input[j]]= temp[input[j]]+1
 7    )   for i = 2 to k
 8    )     do temp[i]=temp[i]+temp[i-1]
 9    )   for j = n to 1
10    )     do ordinati[temp[input[j]]] = input[j]
11    )     temp[input[j]] = temp[input[j]]-1
```

Dopo l'inizializzazione alle linee 1 - 2 - 3 - 4, si analizza ogni elemento di input alle linee 5 e 6. Se il valore di un elemento di input è i, si incrementa $temp[i]$.
Così , dopo le linee 5 e 6, $temp[i]$ contiene il numero di elementi di input uguali ad i per ciascun intero $i = 1, 2, ..k$.
Alle linee 7 e 8, si determina per ciascun $i = 1, 2, ...k$, quanti elementi dell'input sono minori o uguali ad i. Questo viene fatto determinando la somma cumulata degli elementi dell'array $temp[]$.
Infine alle linee 9-10 e 11, si pone ogni elemento $input[j]$ nell'array di output $ordinati[]$, nella sua corretta posizione.
Se tutti gli n elementi sono distinti, allora la prima volta che viene eseguita la linea 9, per ciascun $input[j]$, il valore $temp[input[j]]$ è la posizione finale corretta di $input[j]$ nell'array di output $ordinati[]$.
Infatti vi sono $temp[input[j]]$ elementi minori o uguali di $input[j]$.

Visto che gli elementi potrebbero non essere distinti, si decrementa $temp[input[j]]$ ogni volta che si pone un valore $array[j]$ nell'array $ordinati[]$, questo fa si che il prossimo elemento di input con un valore uguale ad $input[j]$, se esiste, vada nella posizione immediatamente precedente nell'array $ordinati[]$.

Il tempo impiegato dalla sopra descritta procedura è il seguente:

- Ciclo for nelle linee 1 e 2: O(n);
- Ciclo for nelle linee 3 e 4: O(k);
- Ciclo for nelle linee 5 e 6: O(n);
- Ciclo for nelle linee 7 e 8: O(k);
- Ciclo for nelle linee 9-11: O(n);
- Tempo totale: O(k+n).

Per comprendere a livello pratico il **modo di operare** dell'algoritmo di ordinamento **Counting Sort**, si segua l'esempio sottostante dove assumo che:

- A[n]: array contenente una sequenza di numeri, dove ogni elemento di $A[]$ è un intero positivo, non piú grande di k.
- n: dimensione dell'array A[ ].
- temp[k]: array di supporto
- k: max numero contenuto in A[ ].
- ordinati[n]: array che conterrà i numeri completamente ordinati

Supponiamo che nell'array di input, $A[]$, vi siano 8 elementi, dove ogni elemento di $A[]$ è un intero positivo, non piú grande di $k = 6$:

A[8]=

3	6	4	1	3	4	1	4

La prima operazione che **Counting Sort** deve eseguire, dopo le inizializzazioni, è quella di inserire in $temp[]$, le ricorrenze dei numeri trovati in $A[]$.

- Alla prima posizione, cioè $A[0]$, dell'array contenente i numeri da ordinare, troviamo il numero 3 quindi andremo ad incrementare di una unità la terza posizione, cioè $temp[2]$, dell'array di supporto. Ricordo che in realtà, la prima posizione di un array, è array[0] e non array[1];
- Alla seconda posizione, cioè $A[1]$, dell'array contenente i numeri da ordinare, troviamo il numero 6 quindi andremo ad incrementare di una unità la sesta posizione, cioè $temp[5]$, dell'array di supporto;
- Alla terza posizione, cioè $A[2]$, dell'array contenente i numeri da ordinare, troviamo il numero 4 quindi andremo ad incrementare di una unità la quarta posizione, cioè $temp[3]$, dell'array di supporto;
- .....a seguire per tutti i numeri contenuti in $A[]$.

Alla fine di questo ciclo, $temp[]$ contiene i seguenti elementi:

temp[6]=

2	0	2	3	0	1

Questo sta a significare che nell'array $A[]$ ci sono:

- 2 ricorrenze del numero 1;

- 0 ricorrenze del numero 2;
- 2 ricorrenze del numero 3;
- 3 ricorrenze del numero 4;
- 0 ricorrenze del numero 5;
- 1 ricorrenze del numero 6.

La seconda operazione da fare, è quella di determinare per ciascun $i = 1, 2, ...k$, quanti elementi dell'input sono minori o uguali ad i. Questo viene fatto determinando la somma cumulata degli elementi dell'array *temp*[].

Alla fine di questo ciclo, *temp*[] contiene i seguenti elementi:

temp[6]=

2	2	4	7	7	8

Ora, visto che di ogni elemento dell'insieme da ordinare, conosciamo posizione, cioè quanti elementi minori o uguali ci sono, dato nella casella dell'array di supporto, non ci rimane che mettere in maniera ordinata i numeri nell'array ordinati[ ], ottenendo il seguente risultato:

ordinati[8]=

1	1	3	3	4	4	4	6

3.1.1 Come ho implementato Counting Sort

Ho variato lievemente la "*classica*" procedura di ordinamento di **Counting Sort**, progettando l'algoritmo in modo che potesse ordinare anche degli interi negativi. Un'altra variazione apportata è stata il modo di inserire le ricorrenze dei numeri trovati in input nell'array di supporto *supporto*[].
Ho limitato il range dei numeri interi in input da $MIN = 0$ a $MAX = [2^{10} - 1]$, dove MIN e MAX sono rispettivamente il limite inferiore e superiore dell'intervallo specificato.
Se si desidera aumentare il range dei numeri in input sarà sufficiente modificare i valori di MIN e MAX all'interno del file sorgente *countingsort.cpp*.
E' necessario prestare molta attenzione nell'aumentare il range dei numeri in input perché questo algoritmo richiede interi in un piccolo intervallo, pena il blocco della macchina. **Counting Sort** deve dimensionare un array con il max numero trovato nell'array di input, se il numero troppo grande il computer potrebbe bloccarsi.

Vediamo con un esempio, il mio modo di procedere.
Nel codice assumo che:

- array[dim], array che contiene i numeri da ordinare;
- ordinati[dim], array che contiene i numeri ordinati;
- supporto[range], array temporaneo di lavoro;
- dim, numero degli elementi da ordinare;
- MAX=$[2^{10} - 1]$, limite superiore dell'intervallo dei numeri da ordinare
- MIN=$-[2^{10} - 1]$, limite inferiore dell'intervallo dei numeri da ordinare
- range, range dell'insieme numerico da ordinare=$MAX - MIN + 1$

Supponiamo di avere un array di input, $Array[8]$, dove ogni elemento di $Array[]$ è un intero positivo, non piú grande di MAX e non piú piccolo di MIN:

Array[ ]=

3	6	4	-1	1	3	4	1	4

Dopo l'inizializzazione degli array occorrenti, viene analizzato ogni elemento in input. Se il valore di un elemento di input è i, si incrementa $supporto[i - MIN]$, e questo viene eseguito con la seguente parte di codice:

```
for( x=0;x<dim;x++ ){
   supporto[array[ x ]-MIN ]++;
}
```

Successivamente, dato che in array[ ] troviamo:

- alla prima posizione, cioè $Array[0]$, il numero 3, andremo a incrementare di una unità la [$terza-min$] posizione dell'array di supporto, cioè $supporto[1003]$;
- alla seconda posizione, cioè $Array[1]$, troviamo il numero 6, andremo ad incrementare di una unità la [$sesta - min$] posizione, cioè $supporto[1006]$,dell'array di supporto;
- alla terza posizione, cioè $Array[2]$, troviamo il numero 4, andremo a incrementare di una unità la [$quarta - min$] posizione, cioè $supporto[1004]$,dell'array di supporto;
- alla quarta posizione, cioè $Array[3]$, troviamo il numero -1, andremo a incrementare di una unità la [$(-1) - min$] posizione, cioè $supporto[999]$, dell'array di supporto;
- a seguire per tutti i numeri contenuti in $Array[]$.

Alla fine di questo ciclo, $supporto[]$ contiene i seguenti elementi:
(per motivi di spazio, sulla figura riporterò solamente le posizioni da $supporto[999]$ a $supporto[1006]$ che contengono le ricorrenze dei numeri trovati in input, tutti gli altri valori di $supporto[]$ sono infatti uguali a0).

$supporto[range]=$								
	1	0	2	0	2	3	0	1

Questo sta a significare che in $Array[]$ ci sono:

- 1 ricorrenza del numero -1;
- 0 ricorrenze del numero 0;
- 2 ricorrenze del numero 1;
- 0 ricorrenze del numero 2;
- 2 ricorrenze del numero 3;
- 3 ricorrenze del numero 4;
- 0 ricorrenze del numero 5;
- 1 ricorrenze del numero 6.

Così , dopo questo ciclo, $supporto[i]$ contiene il numero di elementi di input uguali ad i per ciascun intero $i = MIN..MAX$.
La seconda operazione da eseguire, è quella di determinare per ciascun $i = MIN, ...MAX$, quanti elementi dell'input sono minori o uguali ad i, operazione effettuata dal seguente codice:

```
//secondo ciclo for che calcola la posizione di scrittura
  for( x=0,k=0;x<range;x++){
     int tmp;
     tmp=supporto[ x ];
     supporto[ x ]=k;
     k+=tmp;
  }
```

Alla fine di questo ciclo, *supporto*[999], ..., [1007], contengono i seguenti elementi:

supporto[range]=

0	1	1	3	3	5	8	8	9

Se nell'array in input, tutti i numeri non sono distinti, numeri uguali verranno scritti in posizioni uguali in *ordinati*[].
Per evitare questo, si incrementa di una unità ogni casella di *supporto*[] visitata, dando così alla successiva ricorrenza dello stesso numero, la sua nuova giusta posizione in *ordinati*[].

```
//terzo ciclo for che mette i numeri in array[  ] in maniera ordinata
  for( x=0;x<dim;x++ ){
     ordinati[ supporto[ array[ x ]-MIN ] ]=array[ x];
     supporto[ array[x]-MIN ]++;
  }
```

Il tempo impiegato da **Counting Sort** è il seguente:

- Caso migliore e peggiore: "Θ(dim+range);
- Tempo totale: Θ(dim+range).

3.1.2 Esperimento: testing dell'algoritmo Counting Sort

In questo esperimento sono stai effettuati 120 test, di cui;

- 5 test in cui sono stati ordinati n numeri **interi casuali**. Dei valori temporali ottenuti dai 5 test è stata calcolata la media matematica riportata sul grafico;
- 5 test in cui sono stati ordinati n numeri **interi ordinati in modo crescente**. Come sopra;
- 5 test in cui sono stati ordinati n numeri **interi ordinati in modo decrescente**. Come sopra;
- il valore di n varia da 500.000 a 5.000.000 con incrementi di 500.000.

L'insieme numerico usato nell'esperimento è stato ottenuto con un generatore a distribuzione uniforme in un range $[0..(2^{10}-1)]$. I risultati ottenuti sono riportati nella tabella seguente, dove si tenga presente che:

- la colonna "quantità", rappresenta la quantità n dei numeri ordinati;
- la colonna "casuali", rappresenta il tempo medio impiegato dall'algoritmo per ordinare la sequenza di n numeri interi casuali;
- la colonna "ordinati", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi ordinati in modo crescente;
- la colonna "inversi", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi ordinati in modo decrescente;

CountingSort			
Quantità	Casuali	Ordinati	Inversi
500.000	16	11	12
1.000.000	22	28	27
1.500.000	33	39	39
2.000.000	50	49	48
2.500.000	60	62	61
3.000.000	71	71	72
3.500.000	82	81	88
4.000.000	99	93	98
4.500.000	110	112	111
5.000.000	132	121	127

3.1.3 Note sul grafico

Il grafico presenta sull'asse delle ascisse le cardinalità dei vettori sui quali è avvenuta l'elaborazione, mentre sull'asse delle ordinate il tempo di calcolo espresso in centesimi di secondo.
Le tre barre verticali rappresentano rispettivamente:

- il tempo medio occorso all'algoritmo per ordinare la sequenza numerica di numeri casuali in un range $[0..(2^{10}-1)]$;
- il tempo medio occorso all'algoritmo per ordinare la sequenza numerica di numeri ordinati in modo crescente in un range $[0..(2^{10}-1)]$;
- il tempo medio occorso all'algoritmo per ordinare la sequenza numerica di numeri ordinati in modo decrescente in un range $[0..(2^{10}-1)]$;

3.1.4 Scopo dell'esperimento

Verificare e dimostrare che:

- dato che questo algoritmo di ordinamento è basato sulla distribuzione e usa metodi, per ordinare l'insieme dei dati, non basati sul confronto e lo scambio, l'ordine dei numeri in input non dovrebbe influire sulle prestazioni dell'algoritmo;
- il tempo necessario all'ordinamento cresca in modo lineare al crescere di n, dove n rappresenta la quantità dei numeri da ordinare.

3.1.5 Risultati dell'esperimento

Si può notare molto chiaramente dal grafico, che l'algoritmo di ordinamento **Counting Sort** risulta essere assolutamente indipendente dalla distribuzione dei numeri in input, infatti i tempi di ordinamento nei casi migliore e peggiore sono molto simili. Questo risultato è determinato dal fatto che l'algoritmo effettua il medesimo numero di operazioni sia nel caso migliore che nel caso peggiore. Si nota altresì dal grafico che il tempo cresce in modo lineare.

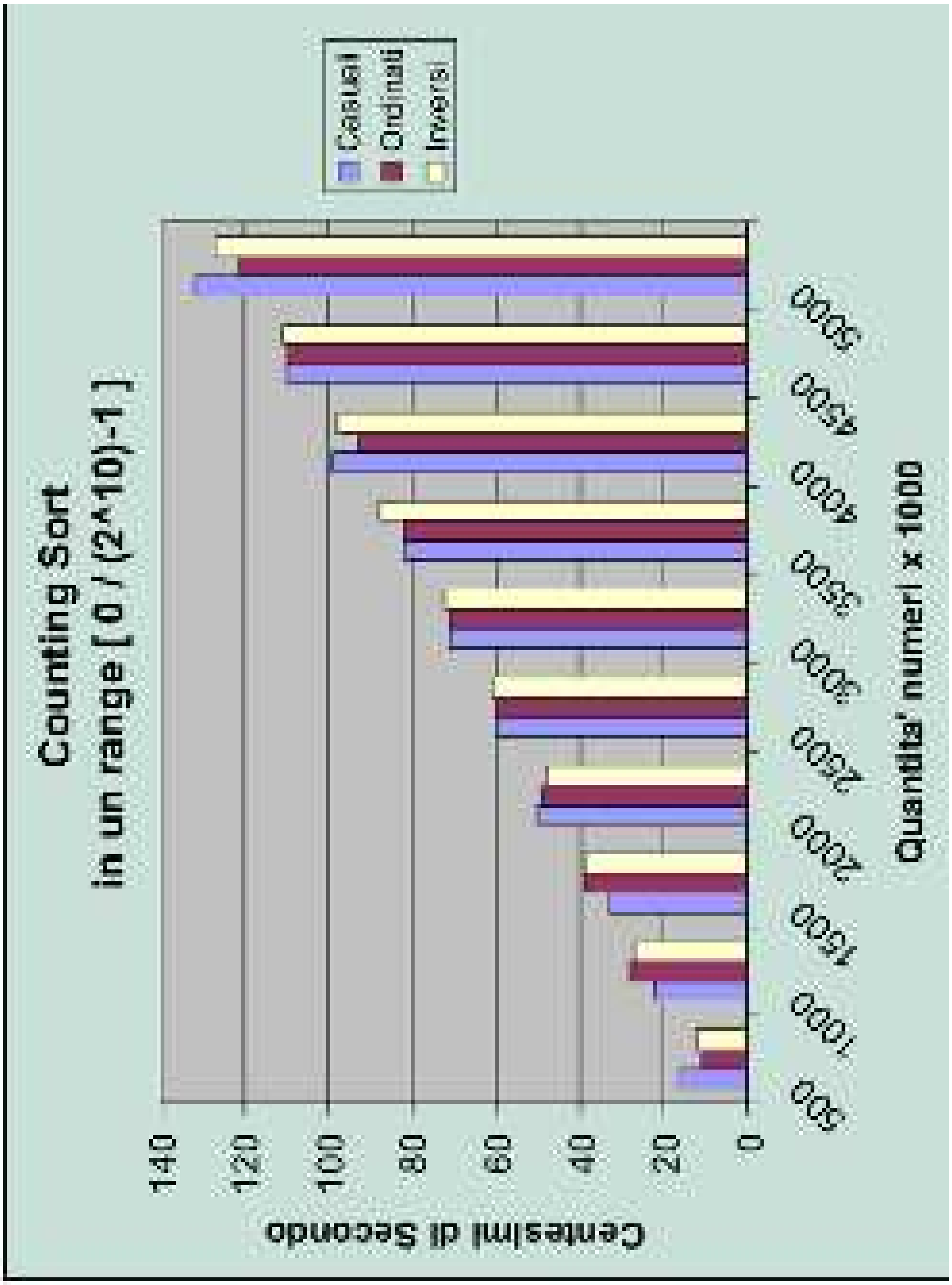

Figure 1: Grafico 2

3.2 Radix Sort

Radix Sort è l'algoritmo usato dalle macchine ordinatrici di schede che si utilizzavano in passato.
Le schede sono suddivise in 80 colonne e ogni colonna può essere perforata in una delle 12 righe. La macchina ordinatrice può essere programmata meccanicamente perché esamini una data colonna di ogni scheda, e inserisca la scheda in uno dei 12 contenitori a seconda di quale riga è stata perforata. Un operatore può quindi raccogliere le schede, contenitore per contenitore, in modo che le schede con la prima riga perforata siano messe sopra le schede con la seconda riga perforata, ...e così via.
Per le cifre decimali, in ogni colonna sono usate solo 10 cifre (le altre 2 sono usate per codificare caratteri non numerici). Un numero di d cifre dovrebbe allora occupare un campo di d colonne. Poichè l'ordinatrice di schede può controllare solo una colonna alla volta, il problema di ordinare n schede contenenti ciascuna un numero di d cifre, richiede un algoritmo di ordinamento.
Intuitivamente, si potrebbe desiderare di ordinare i numeri rispetto alla loro cifra piú significativa, ordinare poi ricorsivamente, ognuno dei contenitori risultanti e quindi, ordinare i blocchi in modo ordinato.
Radix Sort, ordina le schede in modo contro intuitivo, ordinando prima sulla cifra meno significativa. Le schede, sono quindi combinate in un unico blocco, con le schede nel contenitore 0, prima delle schede nel contenitore 1, prima delle schede nel contenitore 2...e così via. Quindi l'intero blocco è ordinato di nuovo sulla seconda cifra meno significativa e ricombinato allo steso modo. Il processo continua fino a che le schede sono state ordinate per tutte le d cifre: a quel punto, le schede sono completamente ordinate. Quindi per l'ordinamento sono richieste solo d passate del blocco.
Facendo un altro esempio, supponiamo di avere un array i cui elementi sono numeri interi ognuno composto da d cifre.
L'ordinamento, consiste nell'utilizzare un qualsiasi algoritmo di ordinamento, che ordini l'array usando come chiave la cifra meno significativa.
Quando l'intero blocco è ordinato, si considera la seconda cifra meno significativa e si ordina il blocco allo stesso modo.
Il processo continua finchè i numeri sono stati ordinati per tutte le d cifre. A quel punto, i numeri sono completamente ordinati.
Se l'algoritmo utilizzato per l'ordinamento intermedio è stabile, cioè, ha la proprietà che numeri con lo stesso valore appaiono nell'array di uscita nello stesso ordine di ingresso, avremo alla fine un array ordinato. In definitiva, **Radix Sort**, ordina i numeri in modo contro intuitivo, cioè non *naturale*, ordinando prima sulla cifra meno significativa, poi sulla seconda cifra meno significativa, e così via fino alla cifra piú significativa. La cosa essenziale in quest'algoritmo è che l'ordine delle cifre rimanga stabile. In un comune calcolatore, che è una macchina sequenziale ad accesso diretto, il **Radix Sort** è talvolta usato per ordinare record di informazioni con chiavi multiple, che comprendono piú campi. Per esempio, si potrebbe far eseguire un algoritmo di ordinamento con una funzione di confronto che operi su due date in questo modo: confronta gli anni e, in caso di parità, confronta i mesi e se c'è ancora parità confronta i giorni.
Alternativamente, si potrebbero ordinare le informazioni tre volte con un ordinamento stabile: prima sul giorno, poi sul mese e infine sull'anno.

Il pseudo codice di Radix Sort è molto semplice. La seguente procedura assume che ogni elemento nell'array input di n elementi abbia d cifre, dove la cifra 1 è quella di ordine piú basso mentre la cifra d è quella di ordine piú alto.
Nello pseudo codice di Radix Sort sotto riportato assumo che:

- A[n], array che contiene i numeri da ordinare;
- n, numero degli elementi da ordinare;
- d, numero di cifre che compongono il numero.

Radixsort(A,d)

```
1) for i =1 to d
2)   do usa un ordinamento stabile per ordinare l'array "A[ ]" sulla cifra "i";
```

Per vedere come opera praticamente l'algoritmo di ordinamento **Radix Sort**, si segua l'esempio sottostante. Nell'esempio, per rendere al lettore la comprensione piú semplice, ho utilizzato dei numeri interi composti da d cifre, dove d è costante.
Supponiamo di avere un array di input, $A[7]$, dove ogni elemento di $A[]$ è un intero positivo, composto da 3 cifre, compreso in un range [100...999]:

A[7]=329,457,657,839,436,720,355.

Radixsort			
Originale	I Ordine	II Ordine	III Ordine
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

- La prima operazione che deve eseguire **Radix Sort**, è quella di effettuare l'ordine usando come chiave di ordinamento la cifra meno significativa dei numeri stessi. Logicamente ad ogni ordinamento sulla chiave (cifra meno significativa) deve corrispondere uno spostamento di tutta la struttura (il numero in esame) per questo, è importante che l'ordine venga eseguito da un algoritmo di ordinamento stabile.
  Al primo passaggio i numeri risultano parzialmente ordinati come riportato nella seconda colonna della tabella;

- la seconda operazione che deve eseguire **Radix Sort** è quella di effettuare l'ordine usando come chiave di ordinamento la seconda cifra meno significativa. Al secondo passaggio i numeri risultano parzialmente ordinati come in tabella alla terza colonna;

- la terza operazione che deve eseguire **Radix Sort** è quella di effettuare l'ordine, usando come chiave di ordinamento la terza cifra meno significativa. Al terzo passaggio i numeri risultano definitivamente ordinati come in tabella alla quarta colonna.

In questo caso, dato che i numeri interi presi in analisi sono composti da 3 cifre ciascuno, sono sufficienti 3 passaggi per ottenere un ordine definitivo.
Quindi in un array[ ] contenente n numeri interi composti ognuno di d cifre, occorrono d passaggi per ordinare completamente il vettore.

3.2.1 Come ho implementato Radix Sort

Nell'implementazione di questa classe in codice C++, ho trovato delle difficoltà perché nessuna funzione di questo linguaggio ci da la possibilità di estrarre da un numero di tipo qualsiasi le cifre di cui è composto.
Una soluzione poteva essere quella di convertire il numero in una stringa in modo da poter estrarre ad una ad una le cifre, ma implementando questa soluzione l'efficienza di questo algoritmo cala drasticamente.
Inoltre, come poter ordinare un insieme composto da numeri con un numero di d cifre dove d è variabile?
Ho trovato una soluzione a questo problema, pensando a come sono rappresentati i numeri nella memoria del calcolatore.
L'idea è assumere che un numero intero n per esempio a 64 bit, è composto di 8 cifre. Ogni cifra è composta da 8 bit in base 256.
Per esempio il numero: 0x0329A90000000000
lo si puo immaginare come composto di 8 cifre: 0x03, 0x29, 0xA9, 0x00, 0x00, 0x00, 0x00, 0x00.

A questo punto per estrarre le cifre dal numero n, non ci resta che usare le funzioni per operare sui bit, che il linguaggio C++ implementa:

- $n >> 0 -- > I\ cifra$;
- $n >> 8 -- > II\ cifra$;
- $n >> 16 -- > III\ cifra$.
- $n >> 24 -- > IV\ cifra$...ecc.

Ora dopo aver risolto il problema di estrazione delle cifre dal numero, non rimaneva che implementare un algoritmo di ordinamento *lineare* che eseguisse l'ordinamento dei numeri usando come chiavi di ordinamento le cifre componenti i numeri, partendo da quella meno significativa fino a quella piú significativa.

```
<<Radixsort>>=
define CIFRA 8
define SHIFT 4
define BASE (1<<SHIFT )

int* Radixsort::RadixsortI( int array[ ],int dim){
  for( int x=0;x<CIFRA;x++ ){
     OrdinaLineare( array,dim,x );
  }
  return array;
}
```

Nel codice implementato, la funzione *RadixsortI*, si occupa, attraverso un ciclo for, di fare in modo che l'algoritmo che esegue l'ordinamento intermedio, *OrdinaLineare*, ordini i numeri contenuti nell'array di input *array*[], usando come chiave di ordinamento le cifre componenti il numero partendo da quella meno significativa fino a quella piú significativa.
Vediamo ora di capire come funziona Ordina Lineare:
La funzione riceve come parametri un array contenente i numeri da ordinare, la sua dimensione e un numero intero, che servirà a calcolare la cifra significativa che verrà usata come chiave dell'ordinamento. Per prima cosa si inizializzano gli array temp[ ] con la dimensione dell'array input A[ ] e supporto[ ] con la dimensione BASE attraverso la funzione Init( ).

```
<<OrdinaLineare>>=
void Radixsort::OrdinaLineare(int array[ ],int dim,int cifre

  temp=Init( temp,dim );
  supporto=Init( supporto,BASE );
```

Successivamente, viene eseguito un ciclo for, che analizza gli elementi in input e riempie l'array *supporto*[]. Se il valore di un elemento è i, viene incrementato $array[i] >> (SHIFT * cifre))mod(BASE)]$.

```
//riempie supporto[  ]
  for( int x=0;x<dim;x++ ){
     supporto[ ( array[ x ]>>(SHIFT*cifre ))mod( BASE )]++;
  }
```

La variabile $cifre$ assume un valore che va da 0 a 8 con incrementi di una unità.

- Quando la variabile $cifre = 0$, $SHIFT * cifre = 0$. Questo significa che il valore dello $shift$ a destra è pari a 0 bit e di conseguenza, il valore di supporto[i] rimane inalterato. In questo caso, l'ordine viene eseguito in base alla cifra meno significativa.
- Quando la variabile $cifre$ =1, $SHIFT * cifre = 8$. Questo significa che il valore dello $shift$ a destra è pari a 8 bit e di conseguenza, il valore di *supporto*[i] viene shiftato a destra di 8 bit. In questo caso, l'ordine viene eseguito in base alla seconda cifra meno significativa.

- Quando la variabile $cifre = 2$, $SHIFT * cifre = 16$. Questo significa che il valore dello $shift$ a destra è pari a 16 bit e di conseguenza, il valore di *supporto*$[i]$ viene shiftato a destra di 16 bit. In questo caso, l'ordine viene eseguito in base alla terza cifra meno significativa.

Questo procedimento continua fino alla cifra piú significativa.

La prossima parte del codice, serve a riempire supporto[ ] con la giusta posizione dei numeri in *ordinati*[].

```
//trova posizione
  somma=supporto[ 0 ];
  supporto[ 0 ]=0;
  for( x=1;x<BASE;x++){
     int tmp;
     tmp=supporto[ x ];
     supporto[ x ]=somma;
     somma+=tmp;
  }
//per gestire situazioni in cui alcuni elementi abbiano lo stesso valore, si
incrementa la posizione.
  for( x=0;x<dim;x++  ){
     temp[ supporto[(array[ x]>>(SHIFT*cifre ))mod BASE ]++]=array[ x ];
  }
//rimette in array numeri parzialmente ordinati
  for(x=0;x<dim;x++){
     array[ x ]=temp[ x ];
  }
  delete supporto;
  delete supporto;
  }
```

3.2.2 Tempo impiegato da Radix Sort

L'analisi del tempo di esecuzione dipende da quale ordinamento stabile viene usato come algoritmo di ordinamento intermedio.
Il mio *ordinaLineare*:

- Primo ciclo for: $O(cifre * n)$.
- Secondo ciclo for: $O(cifre * n)$.
- Terzo ciclo for: $O(cifre * n)$.
- Quarto ciclo for: $O(cifre * n)$.
- ....
- Tempo totale: $O(cifre * n)$

I tempi di ordinamento nel caso migliore e peggiore, si equivalgono.

3.2.3 Note supplementari al codice e all'esperimento di Radix Sort

Nell'implementazione del codice, come ho già accennato, ho deciso di considerare costante il numero di cifre di cui ogni elemento intero è composto, per le seguenti ragioni:

- **Radix Sort** deve essere usato per ordinare insiemi numerici composto da numeri molto grandi. Mi è sembrato superfluo gonfiare ulteriormente il codice di questa classe visto che se si ha la necessità di ordinare degli interi in un range molto basso è sicuramente pi'u proficuo usare **Counting Sort**;
- l'ordine dei numeri in un range molto basso viene eseguito puramente a scopo sperimentale.

3.2.4 Esperimento: testing dell'algoritmo Radix Sort

Nel terzo esperimento sono stai effettuati 240 test, di cui;

- 5 test in cui sono stati ordinati 500.000 numeri **interi casuali** in un range $[0..(2^{10}-1)]$. Dei valori temporali ottenuti dai 5 test è stata calcolata la media matematica riportata sul grafico;
- 5 test in cui sono stati ordinati n numeri **interi ordinati in modo crescente** in un range $[0..(2^{10}-1)]$. Come sopra;
- 5 test in cui sono stati ordinati n numeri **interi ordinati in modo decrescente** in un range $[0..(2^{10}-1)]$. Come sopra;
- 5 test in cui sono stati ordinati n numeri **interi casuali** in un range $[0..2^{31}-1]$. Come sopra.
- 5 test in cui sono stati ordinati n numeri **interi ordinati in modo crescente** in un range $[0..2^{31}-1]$. Come sopra;
- 5 test in cui sono stati ordinati n numeri **interi ordinati in modo decrescente** in un range $[0..2^{31}-1]$. Come sopra;
- il valore di n varia da 500.000 a 5.000.000 con incrementi di 500.000.

L'insieme numerico usato nell'esperimento è stato ottenuto con un generatore a distribuzione uniforme in un range $[0..2^{31}-1]$. I risultati ottenuti sono riportati nella tabella seguente, dove si tenga presente che:

- la colonna "quantità", rappresenta la quantità n dei numeri ordinati;
- la colonna "casuali 10", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi casuali in un range $[0..(2^{10}-1)]$;
- la colonna "ordinati 10", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi ordinati in modo crescente in un range $[0..(2^{10}-1)]$;
- la colonna "inversi 10", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi ordinati in modo decrescente in un range $[0..(2^{10}-1)]$;
- la colonna "casuali 31", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi casuali in un range $[0..2^{31}-1]$;
- la colonna "ordinati 31", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi ordinati in modo crescente in un range $[0..2^{31}-1]$;
- la colonna "inversi 31", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi ordinati in modo decrescente in un range $[0..2^{31}-1]$;

RadixSort						
Q.tà num.	Casuali 10	Ordinati 10	Inversi 10	Casuali 31	Ordinati 31	Inversi 31
500.000	1.21	1.32	1.26	1.25	1.27	1.248
1.000.000	2.42	2.36	2.65	2.43	2.44	2.48
1.500.000	3.73	3.46	3.35	3.57	3.52	3.54
2.000.000	5.01	5.05	4.72	4.97	4.96	4.92
2.500.000	6.04	6.23	5.82	6.06	6.07	6
3.000.000	7.47	8.01	7.98	7.73	7.827.76	
3.500.000	8.4	8.02	8.54	8.29	8.26	8.36
4.000.000	9.61	9.23	9.65	9.49	9.45	9.53
4.500.000	10.08	10.87	10.89	10.83	10.85	10.84
5.000.000	11.92	12.04	11.08	11.82	11.78	11.66

3.2.5 Note sul grafico

Il grafico presenta sull'asse delle ascisse le cardinalità dei vettori sui quali è avvenuta l'elaborazione, mentre sull'asse delle ordinate il tempo di calcolo espresso in decimi di secondo.
Le 6 barre verticali rappresentano rispettivamente:

- il tempo medio occorso all'algoritmo per ordinare la sequenza numerica usando numeri casuali in un range $[0..2^{31}-1]$;
- il tempo medio occorso per ordinare la sequenza numerica usando numeri ordinati in modo crescente in un range $[0..2^{31}-1]$;
- il tempo medio occorso per ordinare la sequenza numerica usando numeri ordinati in modo decrescente in un range $[0..2^{31}-1]$;
- il tempo medio occorso per ordinare la sequenza numerica usando numeri casuali in un range $[0..(2^{10}-1)]$;
- il tempo medio occorso per ordinare la sequenza numerica usando numeri ordinati in modo crescente in un range $[0..(2^{10}-1)]$;
- il tempo medio occorso per ordinare la sequenza numerica usando numeri ordinati in modo decrescente in un range $[0..(2^{10}-1)]$;

3.2.6 Scopo dell'esperimento

Verificare e dimostrare che:

- dato che questo algoritmo di ordinamento è basato sulla distribuzione e usa metodi, per ordinare l'insieme dei dati, non basati sul confronto e lo scambio, l'ordine dei numeri in input non dovrebbe influire sulle prestazioni dell'algoritmo;
- il tempo necessario all'ordinamento cresca in modo lineare al crescere di n, dove n rappresenta la quantità dei numeri da ordinare.
- visto che il codice di questa classe considera ogni intero a lunghezza costante, la grandezza dei numeri in input non dovrebbe influire sulle prestazioni dell'algoritmo.

3.2.7 Risultati dell'esperimento

- Si può notare molto chiaramente dal grafico, che l'algoritmo di ordinamento **Radix Sort** risulta essere assolutamente indipendente dalla distribuzione dei numeri in input, infatti i tempi di ordinamento nei casi migliore e peggiore sono molto simili. Questo risultato è determinato dal fatto che l'algoritmo effettua il medesimo numero di operazioni sia nel caso migliore che nel caso peggiore;
- si nota altresì dal grafico che il tempo cresce in modo lineare;
- come ci si aspettava, visto che il codice C++ da me implementato, considera fisso il numero di cifre di cui ogni elemento è composto, il tempo di calcolo di **Radix Sort** non è influenzato dalla grandezza dei numeri in input, infatti sia che l'insieme numerico ordinato appartenga ad un range $[0..2^{31}-1]$ o $[0..(2^{10}-1)]$, i risultati sono simili.

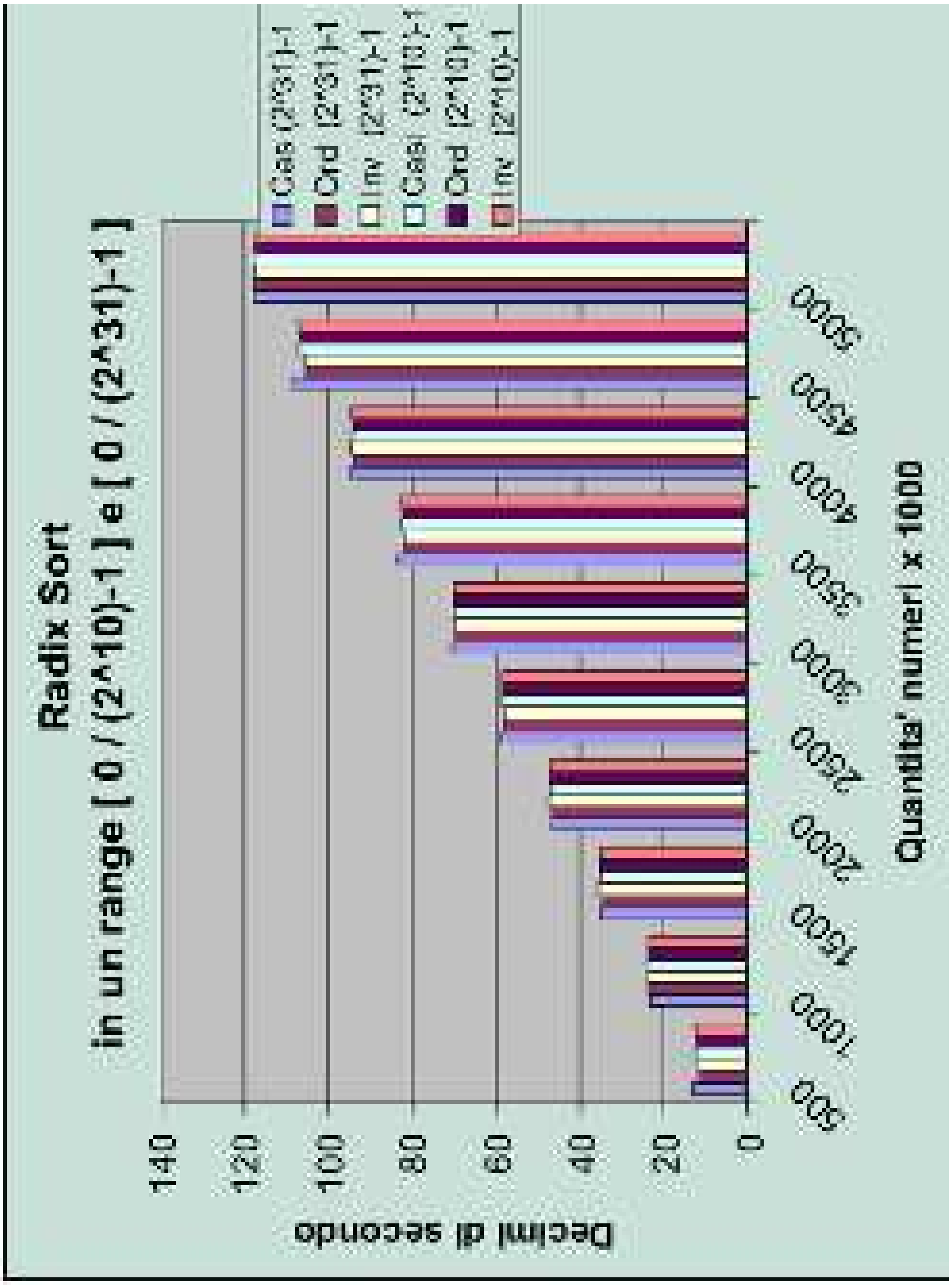

Figure 2: Grafico 3

3.3 BucketSort

L'idea generale di **Bucket Sort** è di dividere un intervallo prefissato, in n sotto intervalli di uguale dimensione, detti *bucket* (cestini), e quindi distribuire gli n elementi da ordinare nei *bucket*. Sotto l'assunzione che gli elementi dell'input siano uniformemente distribuiti nell'intervallo specificato, non ci si aspetta che molti numeri cadano nello stesso *bucket*. Per produrre l'output, si ordinano semplicemente i numeri di ogni *bucket* e quindi si elencano gli elementi di ogni *bucket* prendendoli in ordine.
Il codice di **Bucket Sort** assume che l'input sia un array di n elementi e che ogni elemento, $array[i]$, dell'array soddisfi $min <= array[i] < max$, dove min e max, sono rispettivamente il limite inferiore e superiore dell'intervallo specificato.
Il codice richiede un array ausiliario, $liste[cestini]$, di liste a doppio concatenamento dove ogni elemento di $liste[cestini]$ rappresenta un *bucket*. Nello pseudo codice di **Bucket Sort** sotto riportato assumo che:

- array[n], array che contiene i numeri da ordinare;
- n, numero degli elementi da ordinare;
- liste[cestini], array di liste a doppio concatenamento che contiene i numeri divisi per *bucket*.

$BucketSort(array, n)$

```
1 )  for i = 0 to n-1
      do trova in quale lista va inserito l'elemento array[                              i
      do inserisci array[ i ] nella lista appropriata
2 )  for i = 0 to cestini-1
      do ordina la lista liste[ i ] con RadixSort
3 )  concatena insieme le liste liste[ 0 ], liste[1 ], ..., liste[ cestini-1 ] in
quest'ordine
```

Come **Counting Sort**, **Bucket Sort** è veloce perché fa delle ipotesi sull'input.
Bucket Sort assume che l'input sia generato da un processo casuale che distribuisce gli elementi in modo uniforme sull'intervallo designato.

Per vedere come opera praticamente l'algoritmo di ordinamento **Bucket Sort**, si segua l'esempio sottostante. Supponiamo di dividere l'intervallo [0...1], in 10 sotto intervalli di uguale dimensione, e quindi distribuire gli n elementi da ordinare nei *bucket*. Supponiamo inoltre di avere un array di input, $A[10]$, dove ogni elemento di $A[]$ è un double positivo compreso nell'intervallo prefissato:

A[10]= 0.78, 0.17, 0.35, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68

La prima operazione che deve eseguire **Bucket Sort**, è quella di estrarre ad uno ad uno gli n elementi contenuti in $A[]$ e inserirli nella lista giusta. Osservando la tabella si può notare che:

Bucket Sort						
bucket						
Lista 0						
Lista 1	−− >	0.17	−− >	0.12		
Lista 2	−− >	0.26	−− >	0.21	−− >	0.23
Lista 3	−− >	0.35				
Lista 4						
Lista 5						
Lista 6	−− >	0.68				
Lista 7	−− >	0.78	−− >	0.72		
Lista 8						
Lista 9	−− >	0.94				

- i numeri double 0.12 e 0.17 vengono inseriti in $lista[1]$
- i numeri double 0.21, 0.23 e 0.26 vengono inseriti in $lista[2]$
- il numero double 0.35 viene inserito in $lista[3]$
- il numero double 0.68 viene inserito in $lista[6]$
- i numeri double 0.72 e 0.78 vengono inseriti in $lista[7]$
- il numero double 0.94 viene inserito in $lista[9]$

La seconda operazione che deve eseguire **Bucket Sort** è quella di ordinare i numeri di ogni *bucket*, con un algoritmo di ordinamento adatto, in questo caso, visto che l'insieme numerico in input è composto da tipi double, si può utilizzare *InsertSort*. In tabella, si può notare il risultato di questa operazione.

Bucket Sort						
bucket						
Lista 0						
Lista 1	−− >	0.12	−− >	0.17		
Lista 2	−− >	0.21	−− >	0.23	−− >	0.26
Lista 3	−− >	0.35				
Lista 4						
Lista 5						
Lista 6	−− >	0.68				
Lista 7	−− >	0.72	−− >	0.78		
Lista 8						
Lista 9	−− >	0.94				

La terza ed ultima operazione da eseguire, è quella di concatenare insieme le liste liste[0], liste[1], ..., liste[n-1] in quest'ordine copiando gli elementi nell'array finale *ordinati*[], ottenendo il seguente risultato:

Ordinati[10]= 0.12, 0.17, 0.21, 0.21, 0.23, 0.26, 0.35, 0.68, 0.72, 0.78, 0.94

3.3.1 Come ho implementato Bucket Sort

La classe *Bucketsorti*, ordina i numeri di tipo *int*.
Come algoritmo di ordinamento intermedio ho scelto **Radix Sort** (vedi introduzione). La prima operazione che ho eseguito in fase di implementazione del codice, è stata quella di implementare la classe *Listai*, che crea e gestisce attraverso vari metodi, una lista a doppio concatenamento che serve di supporto a *Bucketsorti*.
La distribuzione degli n elementi da ordinare si suppone uniforme sull'intervallo $[0...(2^{31}-1))$, quindi ho scelto di dividere questo intervallo in 214.748 sotto intervalli uguali, visto che il massimo numero intero generabile è 2.147.483.647.
Il codice di *BucketSorti*, prima di tutto definisce il numero di $CESTINI$, dove $CESTINI$ è il numero dei sotto intervalli.
Definisce poi il valore di $COSTANTE$ dove $COSTANTE$ è la costante con cui il numero in input viene diviso per calcolare il numero del cestino dove lo stesso numero verrà memorizzato.
Crea infine un array di liste a doppio concatenamento, *arraydiliste*[*cestini*] di dimensione *cestini*. Ogni lista di *arraydiliste*[*cestini*], conterrà i numeri in input appartenenti allo specifico sotto intervallo.

```
<<bucketsorti>>
#define CESTINI 214748
#define COSTANTE 10001
ListaI array_di_Listai[ CESTINI ];
```

Viene inizializzato poi, attraverso la funzione *Init*, l'array ordinati[ ] che conterrà la sequenza ordinata dei dati in input. Il passo successivo è quello di riempire le liste con gli n numeri in input. Questa operazione viene svolta dalla funzione *RiempiLista*(), che esamina il numero in input, calcola in quale lista va inserito e lo inserisce.

```
<<RiempiLista>>=
void bucketsort::RiempiLista( double pieno[ ], int n ){
  int posizione;
  for( x=0;x<n;x++ ){
     posizione=( int)(pieno[ x ]/COSTANTE );//calcola in quale cestino va messo il numero
     array_di_liste[ posizione ].Inserisci(pieno[ x ]
  }
}
```

Dopo aver riempito le liste, si esamina la lista i, dove i=[0...CESTINI-1] e:

- viene calcolata la lunghezza della lista;
- se la lista ha lunghezza=0, non esegue nessuna operazione;
- se la lista ha lunghezza=1, copia in ordinati[ ] il valore presente nella lista;
- se la lista ha lunghezza >1, copia tutti i numeri interi presenti nella lista in un array di supporto *supporto*[] e ordina questo array con l'algoritmo di ordinamento *RadixSort*();
- ricopia poi il contenuto di *supporto*[] in *ordinati*[] nella giusta posizione.

```
  for( x=0;x<CESTINI;x++ ){//scandaglia tutte le liste da lista: [ 0 ]-[ CESTINI ]
    lung_lista=array_di_Listai[ x ].DaiLunghezza( );
    switch(lung_lista){
      case 0: //la lista \'e vuota, non fa nulla
      break;

      case 1://ha 1 elemento che viene inserito in ordinati[ ]
      array_di_Listai[ x ].VaiAlPrimo( );
      ordinati[ pos_ord ]=array_di_Listai[ x ].Leggi( );
      pos_ord++;
      break;

      default://ha >1 elementi che passo a Radix Sort
      array_di_Listai[ x ].VaiAlPrimo( );
      supporto=Init( supporto,lung_lista );
      for( int y=0;y<lung_lista;y++  ){//mette la lista esaminata in supporto
      supporto[ y ]=array_di_Listai[ x ].Leggi( );
      array_di_Listai[ x ].Prossimo( );
      supporto=r.RadixsortI(supporto, lung_lista );
      for( int n=0;n<lung_lista;n++ ){
      ordinati[ pos_ord ]=supporto[ n ];
      pos_ord++;
    }
  delete supporto;
  return ordinati;
}
```

Per analizzare il costo di **Bucket Sort** con **Radix Sort**, sia n la variabile casuale che denota il numero di elementi memorizzati nel cestino lista[i] e $cifre$ il numero di cifre, (costante=8), di cui sono composti i numeri.
Radix Sort impiega un tempo $O(cifre * n)$ per ordinare gli elementi del cestino lista[i].
Vi sono d cestini, quindi:
tempo totale $= O(d(cifre * n))$.

3.3.2 Esperimento: testing dell'algoritmo Bucket Sort

Questo esperimento, non era necessario e viene effettuato solo a scopo dimostrativo.
Se l'algoritmo usato per l'ordinamento intermedio ha un tempo che cresce linearmente, è ovvio che anche il tempo impiegato da **Bucket Sort** deve crescere in modo lineare. Infatti questo algoritmo non fa altro che ordinare ogni bucket con **Radix Sort** che ha un tempo lineare.
Nel quarto esperimento sono stai effettuati 120 test, di cui;

- 5 test in cui sono stati ordinati n numeri **interi casuali** in un range $[0..(2^{10}-1)]$. Dei valori temporali ottenuti dai 5 test è stata calcolata la media matematica riportata sul grafico;
- 5 test in cui sono stati ordinati n numeri **interi ordinati in modo crescente** in un range $[0..(2^{10}-1)]$. Come sopra;
- 5 test in cui sono stati ordinati n numeri **interi ordinati in modo decrescente** in un range $[0..(2^{10}-1)]$. Come sopra;
- il valore di n varia da 500.000 a 5.000.000 con incrementi di 500.000.

L'insieme numerico usato nell'esperimento è stato ottenuto con un generatore a distribuzione uniforme in un range $[0..2^{31}-1]$. I risultati ottenuti sono riportati nella tabella seguente, dove si tenga presente che:

- la colonna "quantità", rappresenta la quantità n dei numeri ordinati;
- la colonna "casuali ", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi casuali in un range $[0..2^{31}-1]$;
- la colonna "ordinati", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi ordinati in modo crescente in un range $[0..2^{31}-1]$;
- la colonna "inversi", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi ordinati in modo decrescente in un range $[0..2^{31}-1]$;

BucketSort			
Quantità	Casuali	Ordinati	Inversi
500.000	2.05	1.22	2.32
1.000.000	37.85	43.54	42.58
1.500.000	52.24	48.34	50.21
2.000.000	71.56	67.24	68.53
2.500.000	89.31	92.39	87.36
3.000.000	103.25	104.88	105.32
3.500.000	129.15	128.78	127.28
4.000.000	163.48	160.56	161.2
4.500.000	188.07	184.29	185.32
5.000.000	205.78	201.99	202.23

3.3.3 Note sul grafico

Il grafico presenta sull'asse delle ascisse le cardinalità dei vettori sui quali è avvenuta l'elaborazione, mentre sull'asse delle ordinate il tempo di calcolo espresso in secondi.
Le 3 barre verticali rappresentano rispettivamente:

- il tempo medio occorso all'algoritmo per ordinare la sequenza numerica usando numeri casuali in un range $[0..2^{31}-1]$;
- il tempo medio occorso per ordinare la sequenza numerica usando numeri ordinati in modo crescente in un range $[0..2^{31}-1]$;
- il tempo medio occorso per ordinare la sequenza numerica usando numeri ordinati in modo decrescente in un range $[0..2^{31}-1]$;

3.3.4 Scopo dell'esperimento

Verificare e dimostrare che:

- dato che questo algoritmo di ordinamento è basato sulla distribuzione e usa metodi per ordinare l'insieme dei dati non basandosi sul confronto e lo scambio, l'ordine dei numeri in input non deve influire sulle prestazioni dell'algoritmo, quindi il tempo necessario all'ordinamento deve crescere in modo lineare.

3.3.5 Risultati dell'esperimento

Si può notare molto chiaramente dal grafico, che l'algoritmo di ordinamento **Bucket Sort** risulta essere assolutamente indipendente dalla distribuzione dei numeri in input. Il risultato era quello che ci si aspettava, infatti, come ho già detto, **Bucket Sort** non fa altro che ordinare i numeri presenti in ogni cestino, usando l'algoritmo **Radix Sort** che ha un tempo di ordinamento lineare.
Da notare, il notevole incremento temporale rispetto agli altri algoritmi, causato dalla gestione delle liste che rappresentano i bucket.

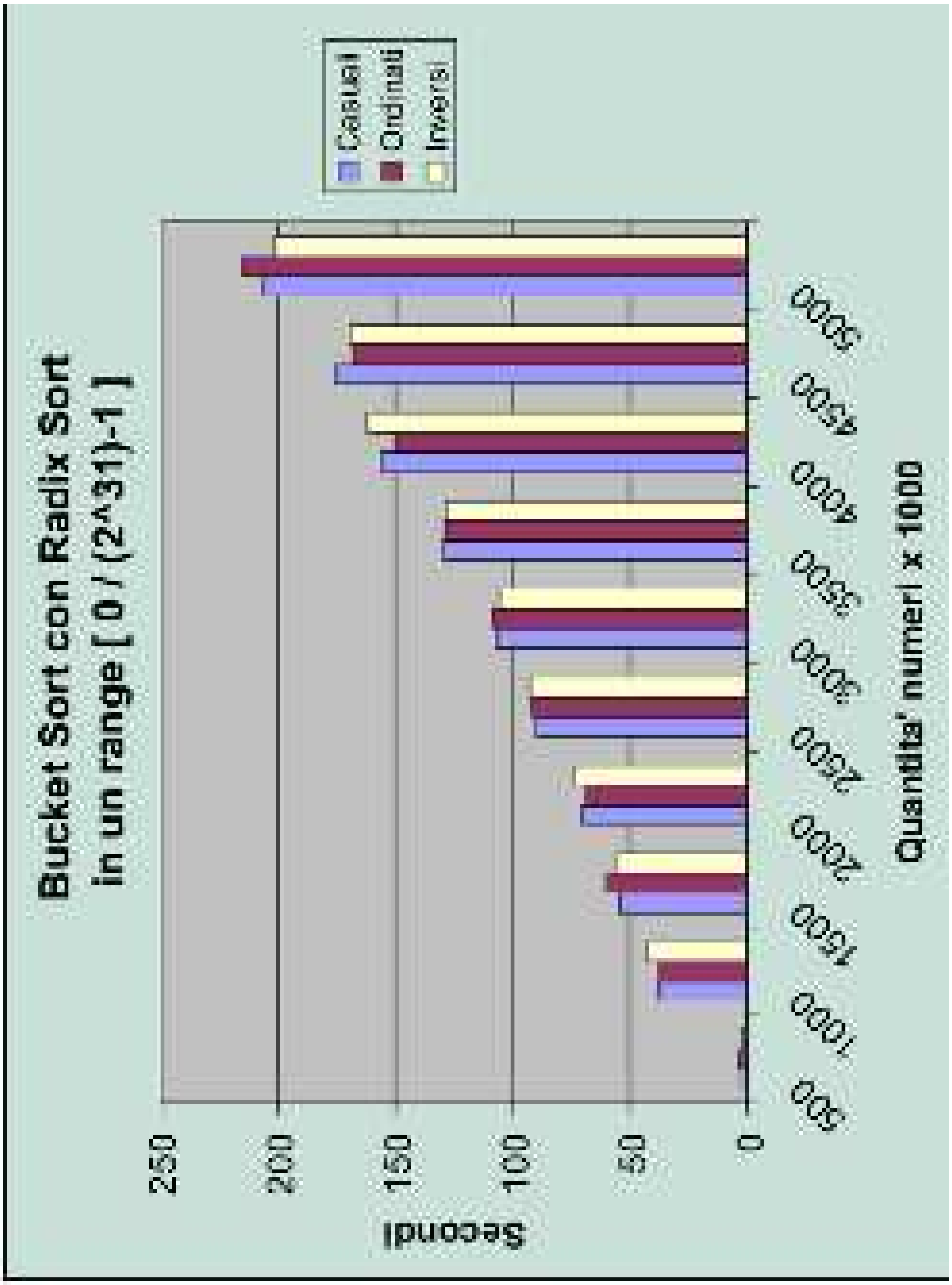

Figure 3: Grafico 4

3.3.6 Esperimento: testing dell'algoritmo Bucket Sort

Nel quinto esperimento sono stai effettuati 120 test, di cui;

- 5 test in cui sono stati ordinati n numeri **interi casuali** in un range $[0..2^{31}-1]$ usando nell'algoritmo un solo cestino, cioè memorizzando tutti i numeri in input in un solo cestino. Dei valori temporali ottenuti dai 5 test è stata calcolata la media matematica riportata sul grafico;
- 5 test in cui sono stati ordinati 500.000 numeri **interi casuali** in un range $[0..2^{31}-1]$ usando nell'algoritmo tutti i cestini disponibili, cioè distribuendo in maniera uniforme l'insieme dei numeri in input nei 214.748 cestini. Come sopra;
- il valore di n varia da 500.000 a 5.000.000 con incrementi di 500.000.

L'insieme numerico usato nell'esperimento è stato ottenuto con un generatore a distribuzione uniforme in un range $[0..2^{31}-1]$. I risultati ottenuti sono riportati nella tabella seguente, dove si tenga presente che:

- la colonna "quantità", rappresenta la quantità n dei numeri ordinati;
- la colonna "1 Cestino", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi casuali in un range $[0..2^{31}-1]$ usando 1 solo cestino;
- la colonna "*All* Cestini", rappresenta il tempo medio occorso all'algoritmo per ordinare la sequenza di n numeri interi casuali in un range $[0..2^{31}-1]$ usando tutti i 214.748 cestini disponibili;

BucketSort		
Quantità	1 Cestino	All Cestini
500.000	2.36	13.1
1.000.000	38.29	137.6
1.500.000	54.16	>800
2.000.000	70.74	>800
2.500.000	90.3	>800
3.000.000	107.02	>800
3.500.000	129.29	>800
4.000.000	156.1	>800
4.500.000	175.87	>800
5.000.000	206.41	>800

3.3.7 Note sul grafico

Il grafico presenta sull'asse delle ascisse le cardinalità dei vettori sui quali è avvenuta l'elaborazione, mentre sull'asse delle ordinate il tempo di calcolo espresso in secondi.
Le 2 barre verticali rappresentano rispettivamente:

- il tempo medio occorso per ordinare la sequenza numerica usando 1 solo cestino;
- il tempo occorso per ordinare la sequenza numerica usando tutti i cestini disponibili;

Dato che in questo esperimento non era necessario rilevare esattamente i tempi di ordinamento, nei test dove sono state usate tutte le liste disponibili, visto che il tempo occorso era notevolmente elevato, ho interrotto ogni test che superava gli 800 secondi.

3.3.8 Scopo dell'esperimento

Lo scopo di questo esperimento, è quello di dimostrare che implementare l'algoritmo **Bucket Sort** usando per l'ordinamento intermedio un algoritmo non basato sul confronto è assurdo.

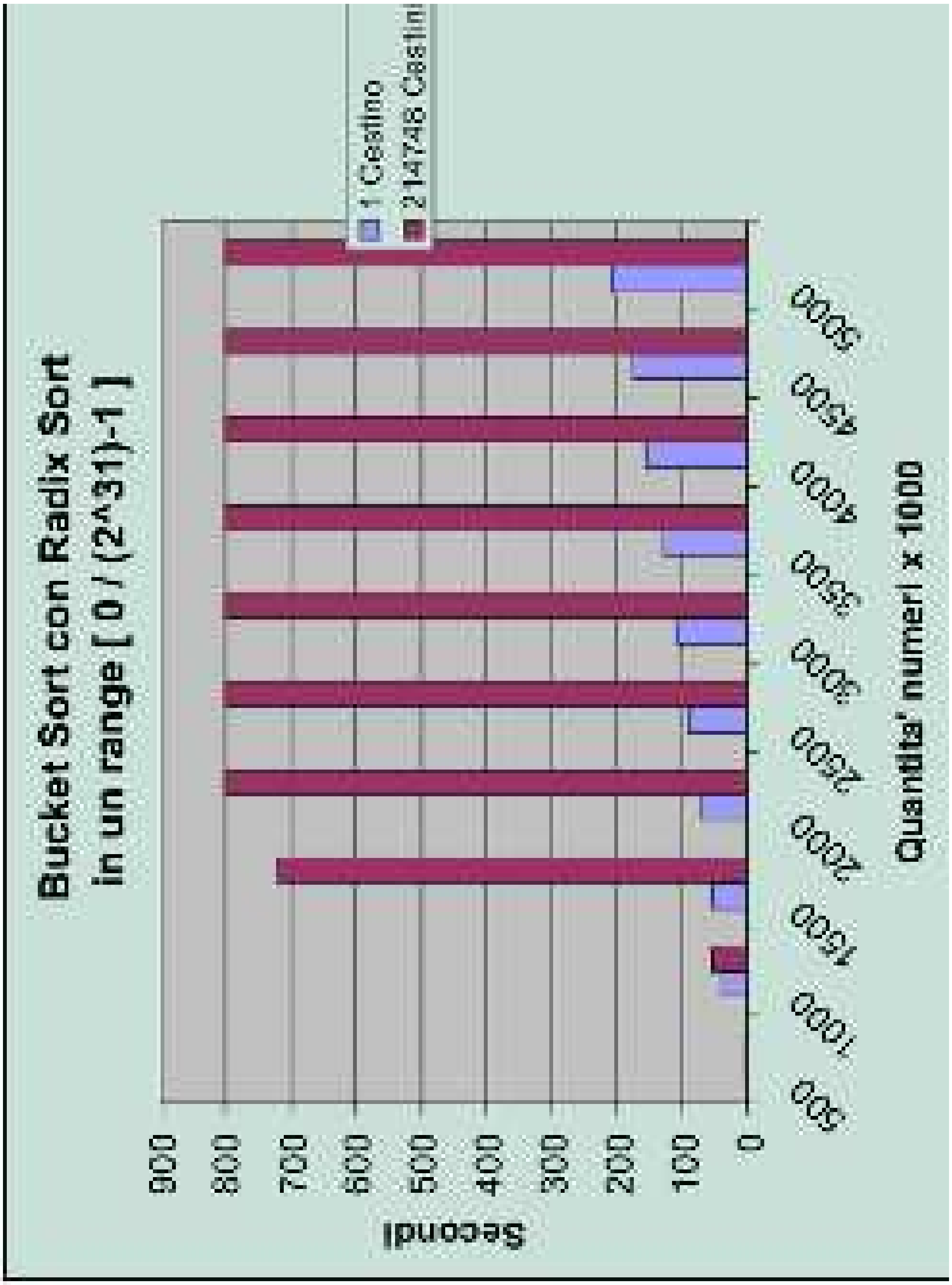

Figure 4: Grafico 5

3.3.9 Risultati dell'esperimento

Come abbiamo dimostrato nel precedente esperimento, l'ordine dei numeri in input non influisce sulle prestazioni dell'algoritmo, di conseguenza il tempo di calcolo cresce in modo lineare.
Non è quindi necessario eseguire un ordine parziale distribuendo gli n elementi nei 214748 cestini, riducendo il numero delle presenze in ognuno di essi, perché il tempo impiegato da **Radix Sort** è lineare. Infatti, sia che i numeri siano uniformemente distribuiti in tutti i 214.748 cestini disponibili, sia che siano tutti memorizzati in un solo cestino, il tempo di calcolo cresce in entrambi i casi in modo lineare. Inoltre, il tempo occorrente alla distribuzione dei numeri in input nei 214.748 cestini, fà addirittura aumentare all'algoritmo il tempo di calcolo.
Escludendo il tempo utilizzato nella gestione dei cestini e nella distribuzione degli n numeri negli stessi, le prestazioni sono equivalenti sia nel caso che si utilizzi 1 solo cestino, sia che si utilizzino tutti i cestini. La conseguenza di questa affermazione è che (nel caso in cui l'algoritmo intermedio abbia un tempo lineare) conviene implementare **Bucket Sort** usando 1 solo cestino. A questo punto è solo una perdita di tempo sistemare i numeri nel cestino per poi ordinarli con **Radix Sort**, è piú intelligente farli ordinare direttamente da **Radix Sort**.
Bucket Sort è nato per ordinare sequenze numeriche in virgola mobile. Usando questo tipo di dati, non è possibile implementare per l'ordinamento intermedio, algoritmi come **Counting Sort** o **Radix Sort**, ma diventa necessario rivolgersi ad algoritmi che basano il loro funzionamento sul confronto e lo scambio. Infatti, implementando **Bucket Sort** con un algoritmo di questo tipo, il numero dei cestini influisce in maniera determinante sul tempo di calcolo perché maggiore è il numero dei cestini, minore sarà il tempo di calcolo. Per dimostrare quest'affermazione, si consideri a titolo esemplificativo di implementare nell'algoritmo di ordinamento **Bucket Sort**, per l'ordinamento intermedio, **Insert Sort**, che basa il suo funzionamento sul confronto. Il tempo impiegato da questo algoritmo dipende pesantemente dal numero di presenze in ogni cestino e dall'ordine con cui i numeri sono memorizzati negli stessi. Assumendo che ci siano n elementi nel bucket dobbiamo scorrere $n-1$ elementi. Per ogni elemento, occorre esaminare e spostare altri $n-1$ elementi, per cui risulta una complessità $O(n^2)$. Con una tale complessità, è conveniente che la presenza numerica in ogni cestino sia la piú bassa possibile.
A titolo di curiosità si osservi il prossimo grafico, che presenta un esperimento in cui **Bucket Sort** ordina una sequenza casuale di 1.500.000 numeri interi utilizzando come algoritmo di ordinamento intermedio **Insert Sort**.
Per l'ordinamento di un insieme di numeri fino a circa 1 milione i tempi sono molto vicini allo zero, ma superando il milione di elementi il tempo di calcolo esplode.
Infatti, quando la quantità di numeri è relativamente bassa, la distribuzione numerica in ogni cestino è di conseguenza molto bassa e **Insert Sort** deve eseguire un numero bassissimo di confronti e scambi. Quando la quantità di numeri presenti in ogni cestino comincia ad aumentare il tempo occorrente all'ordinamento cresce con un andamento quadratico, supponendo un ordine casuale all'interno del cestino, visto l'aumentare dei confronti e scambi che **Insert Sort** deve effettuare. La conseguenza è una crescita smisurata del tempo di calcolo.

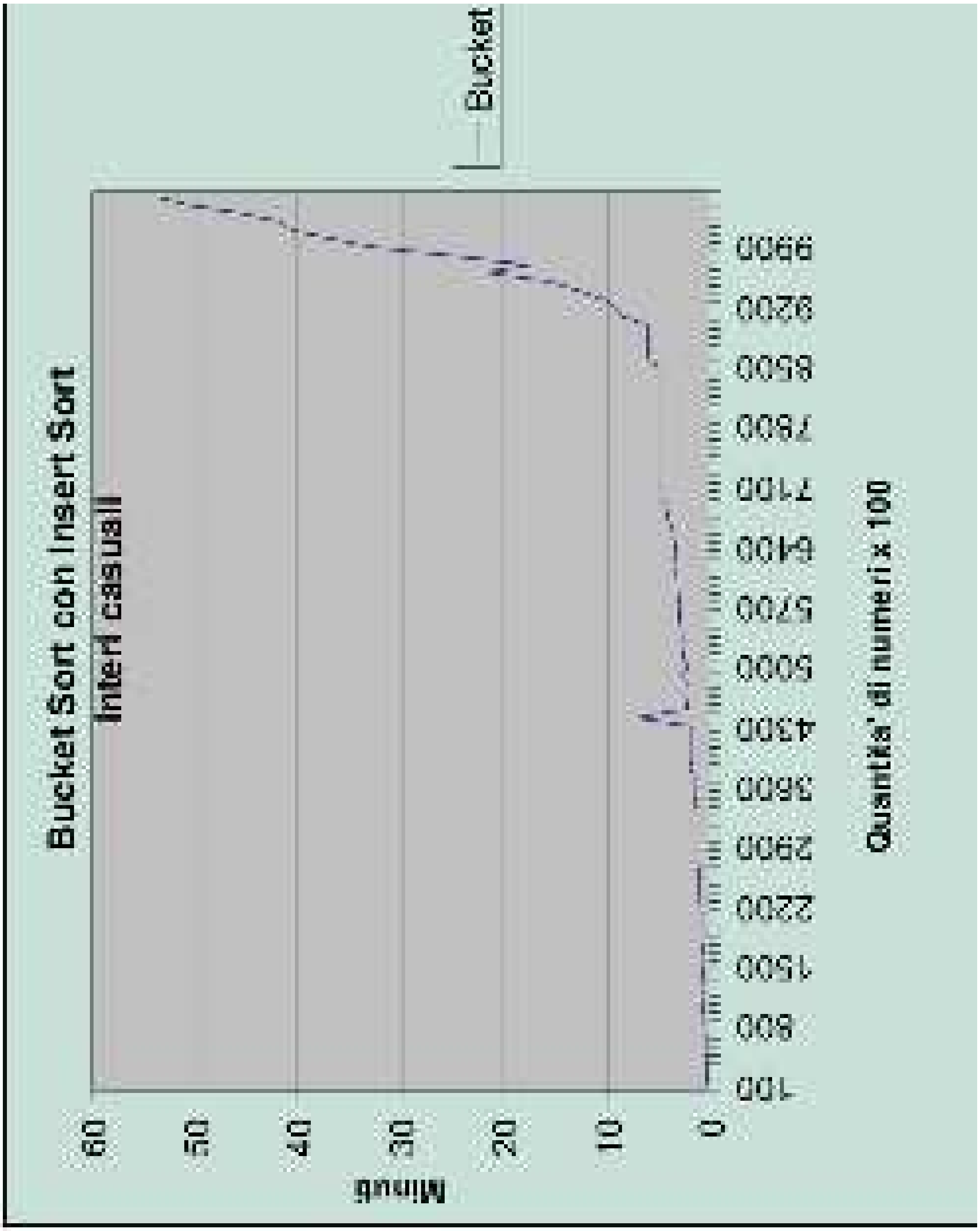

Figure 5: Grafico 6

4 I numeri casuali

La generazione dei numeri casuali è un argomento complesso che non credo sia il caso di trattare in modo approfondito in questa sede.
Viene comunque spontaneo porsi una domanda basilare, cioè cosa significa dire che una sequenza di numeri è casuale?
Consideriamo le due sequenze numeriche formate da 0 e 1:

- a: 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
- b: 1 0 0 1 1 0 1 1 0 1 1 1 0 0 1 0 0 0 0 1

Intuitivamente siamo portati ad affermare che la sequenza a non è casuale poichè è riconoscibile una regolarità o, detto in un altro modo, un algoritmo in grado di generarla, del tipo "scrivi dieci volte 1 0". Al contrario, per la sequenza b, non siamo in grado di discernere alcuno schema soggiacente nè predire lo sviluppo successivo. Non sembra sussistere alcuna regola che presieda alla formazione della sequenza e, quindi, attribiuamo la sua generazione ad un evento casuale. Per esempio, possiamo pensare di aver lanciato in aria per venti volte una moneta non truccata e aver trascritto 0 ad ogni uscita testa e 1 ad ogni uscita croce. A simili argomentazioni , però si può controbattere osservando che nulla vieta di ritenere che anche la sequenza a possa essere stata generata associando 0 e 1 ai risultati dei lanci della stessa moneta che, in tal caso, si sarebbero manifestati con testa e croce alternativamente per venti volte.
Consideriamo altre due sequenze formate dai primi dieci numeri interi:

- c: 0 1 2 3 4 5 6 7 8 9
- d: 1 8 1 3 1 9 4 9 8 5

è immediato ritenere la sequenza c non casuale e casuale la sequenza d. Eppure, anche la sequenza, d apparirà non casuale, qualora si conosca il metodo con cui è stata generata. In particolare, si è scritto in lettere la sequenza c e poi si sono sommate le cifre corrispondenti, nell'alfabeto italiano, alle lettere con cui si sono scritti i numeri precedenti: infine si sono sommate le cifre che formano la somma precedente e, nel caso che il risultato superi 9, si è eseguito una somma ulteriore. Pertanto le due sequenza c e d sono concettualmente equivalenti: l'unica distinzione risiede nella diversità dell'algoritmo di generazione. È dunque la casualità un concetto soggettivo, basato sul fatto di sapere o non sapere come la sequenza è stata prodotta? Se così fosse quello che è casuale per una persona all'oscuro del metodo di generazione non lo è per altri che ne sono a conoscenza.
Esiste tuttavia una via per definire oggettivamente la casualità di una sequenza di numeri?
Il fatto che si possano produrre solo numeri casuali che, proprio per il fatto di essere generati attraverso un algoritmo, casuali non sono, appare in un qualche modo paradossale. La verità è che noi li riteniamo casuali fintanto che rimaniamo all'oscuro delle "regole del gioco" con cui sono stati ottenuti: regole che possono essere scoperte, qualora si sottoponga la sequenza ad opportuni test statistici. Sembrerebbe conveniente, pertanto, ricorrere a fenomeni fisici intrinsecamente non deterministici per generare sequenze di eventi casuali, da usarsi direttamente oppure per produrre tabelle da cui estrarre ogni volta che si desidera successioni di numeri a caso.
Oggi i metodi di generazione maggiormente usati sono i cosiddetti metodi congruenziali, così chiamati perchè si basano sulla nozione di congruenza.
Dati tre numeri interi x, y, z si dice che "x è congruo ad y modulo z" e si scrive $x = y\ (mod\ z)$ se x è la parte intera del resto della divisione tra y e z.
Per generare numeri casuali, si distinguono tre tipi di metodi congruenziali:

- Metodoivo: $r_i + 1 = r_i + r_i - 1(mod\ m)$;
- Metodo moltiplicativo: $r_i + 1 = ar_i(mod\ m)$;
- Metodo misto: $r_i + 1 = (ar_i + b)(mod\ m)$;

dove r_i, a, b, m sono interi non negativi e r_i è compreso tra 0 ed m. Vengono chiamati: a moltiplicatore, b incremento e m modulo, mentre r_i, valore iniziale della sequenza, è detto seme. Tutti questi metodi danno origine a sequenze periodiche, con periodo minore o al più uguale ad m. Se si desiderano numeri casuali razionali nell'intervallo [0,1), questi si ottengono facilmente mediante la divisione:

- $u_i = r_i / m$

Per ciò che riguarda il periodo, si cerca sempre di scegliere m in modo tale da assicurarsi una sequenza abbastanza lunga; in ogni caso, la lunghezza del periodo deve essere superiore alla quantità di numeri casuali richiesti per la soluzione del problema che interessa. Quando la lunghezza del periodo è uguale ad m, si dice che il generatore ha periodo pieno.
Oltre a dar luogo ad un periodo sufficientemente lungo, è auspicabile che il generatore possegga un'alta velocità di generazione, vale a dire che sia in grado di effettuare il calcolo dell'espressione, ad esempio,

- $(ar_i + b)(mod\ m)$ il più rapidamente possibile.

4.1 La classe Rand

Questa classe, è stata progettata per generare sequenze numeriche casuali, di tipo intero, in quantità e range desiderati. La scelta del range di generazione è stata disabilitata, vista l'inutilità ai fini del progetto, ma alcuni ritocchi alla classe possono abilitare questa procedura.
La quantità di numeri da generare è sempre decisa dall'utente. L'insieme dei numeri generati, viene salvato in un file. La classe, può generare anche numeri negativi, sempre previo ritocco del codice.
Dopo aver riempito il file con i numeri desiderati dall'utente, il programma inserisce automaticamente alla fine dello stesso, una stringa contenenti dei dati riepilogativi, ad uso eslusivo della funzione main( ), che se ne servirà per identificare quali sono il max e min numero generati e di quanti numeri è composto l'insieme numerico.

Esempio:

`&1m297M100d`

- 1m: 1 è il minimo numero presente nel file
- 297M: 297 è il massimo numero presente nel file
- 100d: 100, totale dei numeri presenti nel file.

I caratteri speciali, sono stati inseriti necessariamente nella stringa, per permettere al programma di capire quali sono i numeri "significativi".
La funzione "cuore" di questa classe è la seguente:

```
long int RandNumberI(long int max, long int min) {

   long int valore=0;
   valore=min + ( long int ) ( ( max-min ) * ( double( rand( )   ) / RAND_MAX ) );
   dimpos++;
   return valore;
}
```

Questa funzione, il cui codice originale è stato scritto da Romeo Rizzi, il mio docente, funziona in modo esemplare come si vedrà dai test effettuati.

4.2 Esperimento: verifica del generatore di numeri casuali

4.2.1 Scopo dell'esperimento

Il primo esperimento, comprendente 3 test, è stato eseguito per verificare che il generatore di numeri casuali crei un insieme di numeri interi, effettivamente distribuiti nell'intervallo $[0..2^{31}-1]$.
A tale scopo sono state eseguite le seguenti operazioni:

- ll generatore crea 20.000.000 di elementi di tipo intero in un range $[0..2^{31}-1]$;
- i numeri vengono riversati nei 214.748 cestini che utilizza l'algoritmo di ordinamento **Bucket Sort**;
- viene contata la quantità di numeri presente in ogni cestino.
- l'operazione viene ripetuta altre 2 volte.
- viene calcolata la media in ogni cestino, riportata sul grafico.

Se il generatore di numeri casuali, crea numeri interi uniformemente distribuiti nell'intervallo prefissato, nei cestini dovremmo trovare quantità piú o meno uguali di numeri interi.

4.2.2 Note sul grafico

Il grafico presenta sull'asse delle ascisse la quantità di numeri presenti nei cestini, mentre sull'asse delle ordinate il numero del cestino n, dove n=[1...214748].
Per ovvi motivi, è stata rilevata e riportata sul grafico la quantità rilevata di1 cestino ogni 4, esattamente dei cestini: [1...4...8...12...214748].

4.2.3 Risultati dell'esperimento

Si può notare molto chiaramente dal grafico, che ogni cestino di **Bucket Sort** contiene da 70 a 120 presenze, questo prova che il generatore di numeri casuali, crea numeri interi uniformemente distribuiti nell'intervallo prefissato.

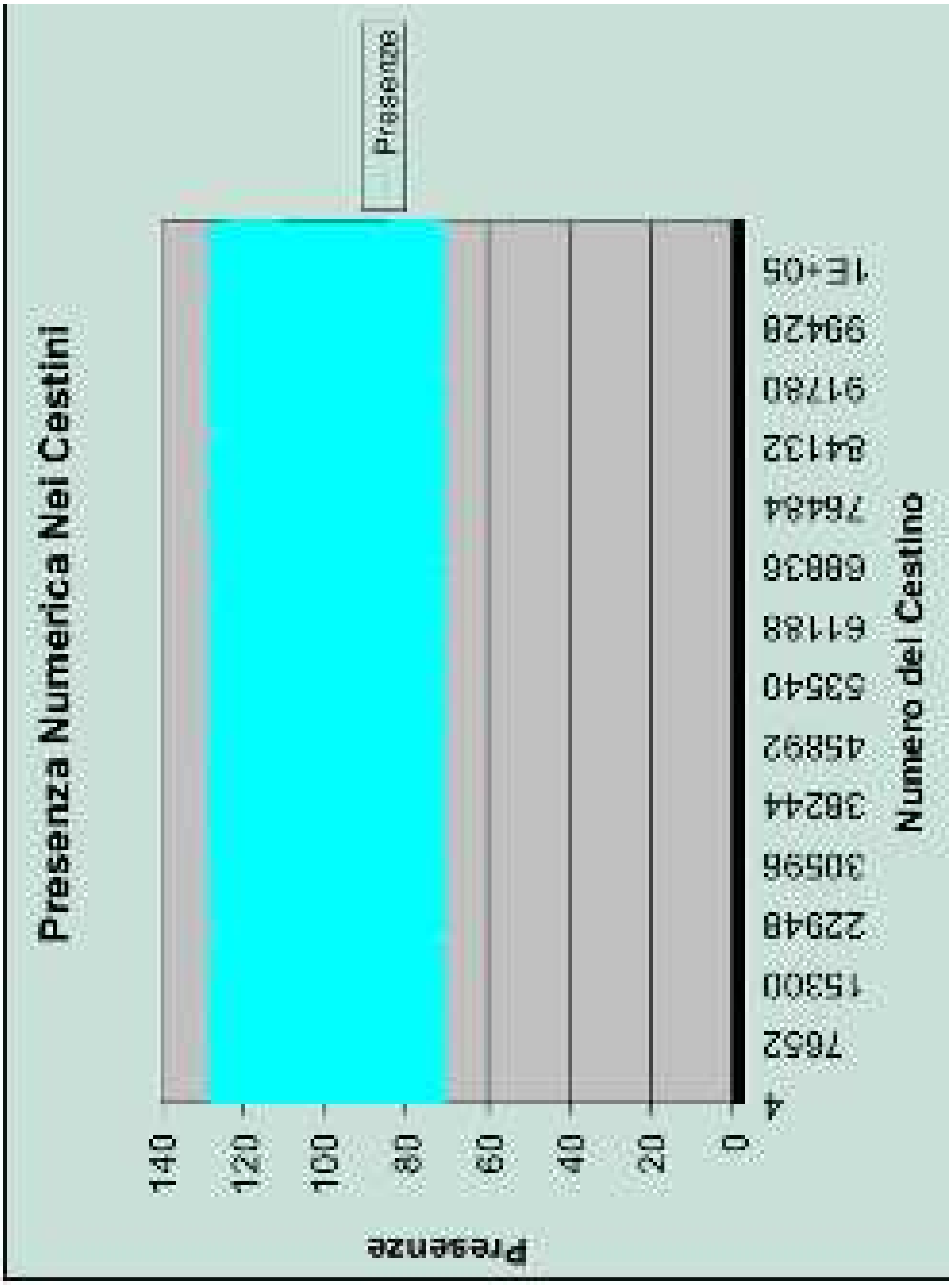

Figure 6: Grafico 1

5 Le Liste concatenate

Nell'implementazione del codice di **Bucket Sort**, ho creato una classe che genera e manipola oggetti di tipo *Lista*, che servono di supporto a questo algoritmo quindi introdurrò brevemente queste strutture di dati.
Le code e gli stack, hanno due caratteristiche in comune: entrambi, prevedono rigide regole per l'accesso ai dati e usano operazioni di estrazione, per loro natura distruttive.
In altre parole, per accedere ad un oggetto in uno stack o una coda è necessario rimuoverlo e questo ne provoca la distruzione, sempre che l'elemento non sia stato memorizzato altrove. Inoltre gli stack e le code utilizzano aree di memoria contigue.
A differenza di uno stack o di una coda, una lista concatenata può essere letta in modo piú flessibile poichè ogni struttura di informazioni, contiene un collegamento all'oggetto successivo come in una catena.
Inoltre un'operazione di estrazione da una lista non distrugge alcun elemento. Quindi per cancellare un elemento sarà necessario implementare una funzione specifica.
Il concatenamento delle liste può essere semplice o doppio.
Una lista a concatenamento semplice contiene un collegamento che porta all'oggetto successivo.
In una lista a doppio concatenamento, contiene un collegamento sia all'oggetto successivo che all'oggetto precedente.
La scelta fra liste a semplice o doppio concatenamento dipende dall'applicazione dalla quale vengono indicate.

5.1 Liste a concatenamento semplice

In una lista a concatenamento semplice, ogni oggetto deve contenere un collegamento con l'oggetto successivo. In genere un oggetto è costituito da una classe che contiene i campi delle informazioni piú un puntatore per il concatenamento.
Principalmente vi sono due modi per realizzare una lista a concatenamento semplice.
Il primo consiste nel semplice inserimento di ogni nuovo oggetto al termine della lista.
Il secondo prevede l'inserimento di un nuovo oggetto in una determinata posizione della lista, ad es. per conservare l'ordinamento ascendente della lista. Questa scelta determina il modo in cui deve essere codificata la funzione di inserimento: Per esempio, in una lista concatenata che preveda solo l'inserimento di nuovi oggetti alla fine della lista, ad ogni oggetto, deve essere associato un puntatore all'oggetto successivo.
Quando si deve inserire un nuovo oggetto in una lista a concatenamento semplice, possono verificarsi tre situazioni.
L'elemento può divenire il primo elemento della lista, oppure trovarsi fra altri due elementi, o essere l'ultimo elemento della lista.
C'è da considerare che se si cambia il primo elemento della lista, si dovrà anche aggiornare il punto di ingresso alla lista. Per evitare questo, si può utilizzare come primo elemento una sentinella. In questo caso, si deve scegliere un valore che sia sempre il primo della lista ed evitare quindi di modificare il punto di ingresso.
Lo svantaggio di questo metodo consiste nell'impiego di una ulteriore posizione di memoria per conservare la sentinella anche se, al giorno d'oggi, questo non rappresenta sicuramente un problema.

5.2 Liste a doppio concatenamento

Le liste a doppio concatenamento, prevedono un concatenamento con l'oggetto successivo e un altro con l'oggetto precedente.
L'aggiunta di un nuovo concatenamento offre numerosi vantaggi. Il piú importante è la possibilità di leggere la lista in entrambe le direzioni.
Il secondo vantaggio è importante solo se si verifica qualche problema. Poichè l'intera lista può essere letta utilizzando i concatenamenti in avanti o all'indietro, nel caso in cui un concatenamento dovesse essere errato, si potrà comunque ricostruire la lista impiegando l'altro concatenamento.
Si può inserire un nuovo elemento in questa lista in tre posizioni, come primo, come mediano o come ultimo.

Come le liste a concatenamento semplice, anche le liste a doppio concatenamento vanno generate con una funzione che inserisca ogni nuovo elemento nella posizione corretta della lista.

5.3 La classe Listai

La classe Listai, rappresenta una lista collegata a doppio concatenamento di oggetti di tipo *nodo*, ed è adatta a contenere numeri di tipo intero.
La variabile di istanza piú importante e la variabile *sentinella*, puntatore al nodo sentinella. Un'altra variabile di istanza di tipo $nodo*$, chiamata *current*, designa il nodo della lista su cui attualmente si è posizionati.

5.4 Le funzioni della classe Lista

- Prossimo( ): va al prossimo nodo della lista
- Precedente(): va al nodo precedente della lista
- VaiAlPrimo( ): va al primo nodo della lista
- VaiFineLista(): va alla fine della lista
- EVuota(): controlla se la lista è vuota
- EPrimo(): controlla se è il primo elemento della lista
- FineLista(): controlla se si è alla fine della lista
- Scrivi(double o int): scrive il dato nel nodo corrente
- Leggi: legge il dato del nodo corrente
- Inserisci(double o int): inserisce un nodo contenente il dato di tipo intero immediatamente prima del nodo designato da *current*. A tale scopo viene creato, con un'istruzione new, un oggetto *newnodo* di tipo *nodo* contenente il dato intero.
  Ai valori dei puntatori $newnodo >next$ e $newnode > prev$ vengono assegnati rispettivamente i valori *current* e $current > prev$ (l'indirizzo del nodo che precede *current*). Poi viene aggiornato il valore del $current > prev$ e il valore del puntatore *next* del nodo che precedeva *current* viene posto uguale a *newnode*; dopo tutte queste operazioni di assegnazione ed aggiornamento, il valore di *lunghezza* (la lunghezza attuale della lista, ovvero il numero di nodi escluso *sentinella*) viene incrementato di 1;
- Cancella(): rimuove il nodo designato da *current*, operazione permessa solo nel caso in cui *current* non coincida con *sentinel*, e restituisce il dato contenuto nel nodo rimosso. A tale scopo $current > data$ viene salvato in una variabile temporanea *tmp* e $current > next$ viene salvato in un puntatore temporaneo *tmpp* in modo che, alla fine, *current* sia il nodo che seguiva il vecchio *current*.
  In seguito vengono aggiornati i valori di $current > next > prev$ e di $current > prev > next$, puntatori dei nodi rispettivamente successivo e precedente a *current* e *current* viene distrutto con un'istruzione delete.

6 conclusioni

Un algoritmo "efficiente" deve svolgere il suo compito in un tempo accettabile utilizzando le risorse a disposizione in modo intelligente e senza sprechi. Algoritmi ideati per risolvere lo stesso problema spesso differiscono vistosamente per quanto riguarda la loro efficienza e queste differenze possono essere piú significative della differenza tra un calcolatore molto lento e un supercomputer. Infatti, se proviamo ad ordinare 10.000.000 di numeri ordinati in modo decrescente, con **Radix Sort**, su un computer dotato di processore intel 286, otterremo un tempo di ordinamento estremamente basso rispetto ad un ordinamento della stessa sequenza numerica, effettuato sul computer piú veloce del mondo usando **Insert Sort**. L'esempio vuole dimostrare due cose fondamentalmente:

- prima di tutto che progettare algoritmi efficienti è importante quanto progettare sistemi hardware sempre piú veloci;
- Infine, che scegliere l'algoritmo appropriato in base al problema da risolvere, significa usare risorse e tempo in modo intelligente.

Gli algoritmi di ordinamento, da quanto ho potuto constatare, sono un argomento un po' sottovalutato dai programmatori perché quando si devono ordinare i dati, a detta di un amico, professionista nel settore, molti richiamano semplicemente la funzione standard *qsort*() contenuta nella libreria di molti linguaggi di programmazione come il C e il C++, basata sull'algoritmo di ordinamento *QuickSort*. Questo algoritmo non può rappresentare la risposta universale per tutte le esigenze di ordinamento perché non è possibile applicare una funzione generica come *qsort*() a tutte le situazioni, pertanto, il suo funzionamento puó risultare piuttosto inefficiente rispetto ad una routine di ordinamento specifica.
Esistono infatti diversi approcci all'ordinamento che possono rivelarsi utili in varie situazioni, pertanto, sarebbe utile che ogni programmatore avesse a disposizione e usasse un'ampia varietà di algoritmi di ordinamento.
Il perché lo abbiamo ampiamente dimostrato in questa relazione, infatti:

- *CountingSort* è risultato essere il piú veloce nel range $[0..(2^{10}-1)]$, ma la sua limitazione lo penalizza pesantemente.
- *RadixSort* si è dimostrato un ottimo algoritmo di ordinamento sia per l'ampiezza del range sia per i tempi di calcolo relativamente bassi.
- *BucketSort* al contrario, abbiamo scoperto e verificato che non deve essere progettato usando per l'ordinamento intermedio un algoritmo non basato sul confronto.
  Sappiamo inoltre che **Bucket Sort** usato per ordinare numeri in virgola mobile è un ottimo algoritmo.

Quindi la scelta di uno di questi, dipende esclusivamente dal campo di applicazione.
Scrivere software in modo professionale, significa cercare di utilizzare al meglio le risorse a disposizione, senza sprechi e sopra tutto cercando di abbassare al minimo i tempi di calcolo.

Come ho scritto all'inizio di questa relazione, ribadisco il concetto che ogni buon programmatore dovrebbe cercare di usare un'ampia categoria di algoritmi di ordinamento e applicarli in modo appropriato a seconda del problema da risolvere.
Il lettore che sperava di trovare una risposta definitiva alla fatidica domanda:

- quale algoritmo si è dimostrato il piú efficiente in termini di utilizzo delle risorse e tempo di calcolo?

rimarrà sicuramente deluso perché tutti gli algoritmi analizzati in questa ricerca sono degli ottimi "algoritmi di ordinamento" progettati in modo intelligente, e un'eventuale classifica, non puó essere generale ma riferita solamente ad uno specifico problema da risolvere.

7 Manuale d'uso

Il programma, una volta lanciato,inizia con un menú comprendente 3 opzioni:

- Crea nuovo file con sequenze numeriche casuali;
- Ordinamento;
- ordinamento inverso;
- exit.

L'opzione *Crea...*, permette la creazione di un file, senza estensione, contenente una sequenza numerica casuale di n elementi di tipo int, dove n viene decisa dall'utente. La scelta di questa opzione, dopo aver digitato il nome del nuovo file, da accesso al sotto menú:

- Counting Sort..., riempie il file con una sequenza di numeri casuali di tipo *int* in un range $0...2^{10}-1$, adatti ad essere ordinati con l'algoritmo di ordinamento **Counting Sort**.
- Radix Sort..., riempie il file con una sequenza di numeri casuali di tipo *int* in un range $0...2^{31}-1$, adatti ad essere ordinati con l'algoritmo di ordinamento **Radix Sort**.
- Bucket Sort..., riempie il file con una sequenza di numeri casuali di tipo *int* in un range $0...2^{31}-1$, adatti ad essere ordinati con l'algoritmo di ordinamento **Bucket Sort**.

L'opzione *Ordinamento*, permette di effettuare un ordinamento crescente di una sequenza di numeri prelevata da un file, con l'algoritmo di ordinamento desiderato. Si tenga presente che, l'ordine decrescente serve solamente a ordinare il file in modo decrescente, per testare l'algoritmo di ordinamento nel caso peggiore. Con questa opzione il timer temporale non viene attivato.
L'opzione "Ordinamento inverso", permette di effettuare un ordinamento decrescente di una sequenza di numeri prelevata da un file. Si tenga presente che, l'ordine decrescente serve solamente a ordinare il file in modo decrescente, per testare l'algoritmo di ordinamento nel caso peggiore. Con questa opzione il timer temporale non viene attivato.
L'opzione *Ordinamento* da accesso al sotto menú:

- Counting Sort: Ordina il file usando l'algoritmo Counting Sort
- Radix Sort: Ordina il file usando l'algoritmo Radix Sort
- Bucket Sort: Ordina il file usando l'algoritmo Bucket Sort

Nel file originale, vengono infine salvati tutti i dati relativi all'ordinamento.

8 I sorgenti C++ del progetto

8.1 Main

⟨*main.cpp*⟩≡

```
#include <string.h>
#include "bucketsorti.h"
#include "radixsort.h"
#include "countingsort.h"
#include "rand.h"

clock_t start, finish;
Rand num;
Countingsort c;
Radixsort r;
Bucketsorti bi;
ifstream Finput;
ofstream Foutput;

char *stringa, *NomeFile;
int  minint, maxint, w, dim, *numeri_int, scelta_menu;
double durata;

void DatiStringa();
void ApriFileS();
void ChiudiFileS();
void ChiudiFileL();
void ApriFileL();
void ControlloNomeFile();
int  InputInt(int);
void RiempiArray();
void Menu();
void Menu_algoritmi();
void Ordinamento(int);

void main (){

        NomeFile=new char[100];
        stringa=new char[100];
        int scelta_algoritmo;

        Menu();
        scelta_menu=InputInt(4);
        if (scelta_menu==4){cout<<"Fine programma..."; exit(1);}

        switch(scelta_menu){

                case 1:
                        ControlloNomeFile();
                        num.RiempiFile(NomeFile);
                        main();
                break;

                case 2:
                        ControlloNomeFile();
```

```
                ApriFileL();
                DatiStringa();
                ChiudiFileL();
                ApriFileL();
                RiempiArray();
                ChiudiFileL();
                Menu_algoritmi();
                scelta_algoritmo=InputInt(4);
                if (scelta_algoritmo==4){main();}
                Ordinamento(scelta_algoritmo);
                delete numeri_int;
            break;

            case 3:
                ControlloNomeFile();
                ApriFileL();
                DatiStringa();
                ChiudiFileL();
                ApriFileL();
                RiempiArray();
                ChiudiFileL();
                numeri_int=r.RadixsortI(numeri_int,dim);
                ApriFileS();
                for(int x=dim-1;x>0;x--){
                    Foutput<<'x'<<numeri_int[x]<<"\n";
                }
                Foutput<<"%";
                for(int y=0;y<w;y++){
                    Foutput<<stringa[y];
                }
                cout<<"\nOrdinamento decrescente eseguito...";
                ChiudiFileS();
                delete numeri_int;
            break;

    }

}
////////////////////////////////////////////////////////////////////////////////
void ControlloNomeFile(void){
    int valido;

    cout <<"\n"<<"Digita il nome del file: ";
    do {
        cin>>NomeFile;
        valido = (strlen(NomeFile)>= 1) && (strlen(NomeFile)<= 100);
        if (! valido) {
            cout <<"\n"<<"Errore nel nome del file!";
            cout <<"\n"<<"Deve avere da 1 a 100 caratteri! ";
            cout <<"\n"<<"Nome del file: ";
          }
    }while (! valido);
}
////////////////////////////////////////////////////////////////////////////////
void Ordinamento(int scelta_algoritmo){
```

```
        switch(scelta_algoritmo){

                                case 1:
                                        start=finish=0;
                                        start=clock();
                                        numeri_int=c.CountingsortI(numeri_int,dim);
                                        finish=clock();
                                        durata=(double)(finish-start)/CLOCKS_PER_SEC;
                                        ApriFileS();
                                        Foutput<<"\n"<<"File ordinato con l'algoritmo di ordinamento C
Sort:";

                                        Foutput<<"\n"<<"Numeri testati: "<<dim;
                                        Foutput<<"\n"<<"Numero minimo: "<<numeri_int[0];
                                        Foutput<<"\n"<<"Numero massimo: "<<numeri_int[dim-1];
                                        Foutput<<"\n"<<"Durata ordinamento: "<<durata<<" secondi.";
                                        ChiudiFileS();
                                break;

                                case 2:
                                        start=finish=0;
                                        start=clock();
                                        numeri_int=r.RadixsortI(numeri_int,dim);
                                        finish=clock();
                                        durata=(double)(finish-start)/CLOCKS_PER_SEC;
                                        ApriFileS();
                                        Foutput<<"\n"<<"File ordinato con l'algoritmo di ordinamento R
                                        Foutput<<"\n"<<"Numeri testati: "<<dim;
                                        Foutput<<"\n"<<"Numero minimo: "<<numeri_int[0];
                                        Foutput<<"\n"<<"Numero massimo: "<<numeri_int[dim-1];
                                        Foutput<<"\n"<<"Durata ordinamento: "<<durata<<" secondi.";
                                        ChiudiFileS();
                                break;

                                case 3:
                                        start=finish=0;
                                        start=clock();
                                        numeri_int=bi.BucketsortI(numeri_int,dim);
                                        finish=clock();
                                        durata=(double)(finish-start)/CLOCKS_PER_SEC;
                                        ApriFileS();
                                        Foutput<<"\n"<<"File ordinato con l'algoritmo di ordinamento B
                                        Foutput<<"\n"<<"Numeri testati: "<<dim;
                                        Foutput<<"\n"<<"Numero minimo: "<<numeri_int[0];
                                        Foutput<<"\n"<<"Numero massimo: "<<numeri_int[dim-1];
                                        Foutput<<"\n"<<"Durata ordinamento: "<<durata<<" secondi.";
                                        ChiudiFileS();
                                        break;

                        }
}
////////////////////////////////////////////////////////////////////////////////
void Menu(void){
        cout <<"\nAlgoritmi di ordinamento - Paolo Zotti 62SI -"<<"\n";
        cout <<"\n\nSelezionare opzione"<<"\n";
```

```
        cout <<"\n1= Crea nuovo file con sequenze numeriche casuali ";
        cout <<"\n2= Ordinamento ";
    cout <<"\n3= Ordinamento inverso ";
        cout <<"\n4= Exit";
        cout <<"\n\nDigitare scelta: ";
}
////////////////////////////////////////////////////////////////////////////////
void Menu_algoritmi(void){
        cout <<"\n\nSelezionare tipo ordinamento:"<<"\n";
        cout <<"\n1= Counting Sort ";
        cout <<"\n2= Radix Sort";
        cout <<"\n3= Bucket Sort";
        cout <<"\n4= Exit";
        cout <<"\n\nDigitare scelta: ";
}
////////////////////////////////////////////////////////////////////////////////
int InputInt(int n){
        int valido, num;

        do {
                cin>>num;
                valido =(num >= 1 && num <= n);
                if (!valido) {
                        cout <<"\n"<<"Error! (1-"<<n<<"): ";
                }
        } while (!valido);
        return num;
}
////////////////////////////////////////////////////////////////////////////////
void ApriFileS(){
        Foutput.open(NomeFile);
        if(!Foutput){
                cerr<<"\n"<<"ATTENZIONE: errore nella creazione del file "<<NomeFile;
                exit(-1);
        }
}
////////////////////////////////////////////////////////////////////////////////
void ApriFileL(){
        Finput.open(NomeFile);
        if(!Finput){
                cerr<<"\n"<<"ATTENZIONE: errore nell'apertura del file "<<NomeFile;
        exit(-1);
        }
}

////////////////////////////////////////////////////////////////////////////////
void ChiudiFileL(){
        Finput.close();
}
////////////////////////////////////////////////////////////////////////////////
void ChiudiFileS(){
        Foutput.close();
}
////////////////////////////////////////////////////////////////////////////////
void RiempiArray(){
```

```
        numeri_int=new int[dim];
        for(int x=0;x<dim;x++){
                numeri_int[x]=0;
                Finput >> numeri_int[x];
        }
}
////////////////////////////////////////////////////////////////////////////////
void DatiStringa(){
        char car;
        int z=0;
        int n=0;
        char temp[5][50];//vettore bid restano residui
//riempio la stringa stringa[] con i dati statistici del file
        while (!Finput.eof()){
                Finput.get(car);
                if (car=='%'){
                        while (!Finput.eof()){
                                Finput.get(car);
                                stringa[z]=car;
                                z++;
                                w++;
                        }break;
                }
        }
//cerco il numero minimo
        while (stringa[n]!='m'){
                temp[2][z]=stringa[n];
                z++;
                n++;
        }
        n++;
        minint=atoi(temp[2]);
        z=0;
//cerco il numero massimo
        while (stringa[n]!='M'){
                temp[3][z]=stringa[n];
                z++;
                n++;
        }
        n++;
        maxint=atoi(temp[3]);
        z=0;
//cerco la dimensione
        while (stringa[n]!='d'){

                temp[4][z]=stringa[n];
                z++;
                n++;
        }
        dim=atoi(temp[4]);
}
```

8.2 Classe Countingsort

Files che dichiarano e definiscono la classe

- countingsort.h
- countingsort.cpp

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

⟨*countingsort.h*⟩≡

```
class Countingsort{

        private:
                int *supporto, *ordinati;
                int* Init(int [],int);
                void RiempiSupporto(int [], int [], int);
                void AddPrev(int a[], int);
                void Ordina(int [], int [], int [], int);

        public:
                Countingsort(){};
                ~Countingsort(){};
                int* CountingsortI(int[], int);
};
```

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

⟨*countingsort.cpp*⟩≡

```
#include "countingsort.h"
#include <stdlib.h>
#include <fstream.h>
#define MIN 0//MIN num per countingsort
#define MAX 1023//MAX num per countingsort
#define RANGE MAX-MIN+1

////////////////////////////////////////////////////////////////////////////////
int* Countingsort::CountingsortI(int array[],int dim){

        ordinati=Init(ordinati,dim);
        supporto=Init(supporto,RANGE);

        for(int x=0;x<dim;x++){
        supporto[array[x]-MIN]++;
        }

        for(int y=0,k=0;y<RANGE;y++){//calcola posizione di scrittura
                int tmp;
                tmp=supporto[y];
                supporto[y]=k;
                k+=tmp;
        }

        for(int z=0;z<dim;z++){
                ordinati[(supporto[array[z]-MIN])]=array[z];
                supporto[array[z]-MIN]++;
        }

        delete supporto;
        delete array;
        return ordinati;
}

////////////////////////////////////////////////////////////////////
int* Countingsort::Init(int *pa,int dim){
        pa=new int[dim];
        for(int x=0;x<dim;x++){
                pa[x]=0;
        }
        return pa;
}
```

8.3 Classe Radixsort

Files che dichiarano e definiscono la classe

- radixsort.h
- radixsort.cpp

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

⟨*radixsort.h*⟩≡

```
class Radixsort{

        private:
                int *supporto, *ordinati;
                int* Init(int [],int);
                void OrdinaLineare(int array[],int n,int cifre);

        public:
                Radixsort(){};
                ~Radixsort(){};
                int* RadixsortI(int [], int);
};
```

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

⟨*radixsort.cpp*⟩≡

```
#include "radixsort.h"
#define CIFRA 8
#define SHIFT 8
#define BASE (1<<SHIFT)

////////////////////////////////////////////////////////////////////////////////
int* Radixsort::RadixsortI(int array[],int dim){

        for(int x=0;x<CIFRA;x++){
                OrdinaLineare(array,dim,x);
        }
        return array;
}
////////////////////////////////////////////////////////////////////////////////
void Radixsort::OrdinaLineare(int array[],int dim,int cifre){

        ordinati=Init(ordinati,dim);
        supporto=Init(supporto,BASE);

        for(int x=0;x<dim;x++){
                supporto[(array[x]>>(SHIFT*cifre))%(BASE)]++;
        }
//trova posizione
        int somma=supporto[0];
        supporto[0]=0;
        for(int y=1;y<BASE;y++){
                int tmp;
                tmp=supporto[y];
                supporto[y]=somma;
                somma+=tmp;
        }
//incrementa ad ogni visita
        for(int z=0;z<dim;z++){
                ordinati[supporto[(array[z]>>(SHIFT*cifre))%BASE]++]=array[z];
        }
//rimette in array numeri parzialmente ordinati
        for(int q=0;q<dim;q++){
                array[q]=ordinati[q];
        }

        delete ordinati;
        delete supporto;
        }

////////////////////////////////////////////////////////////////////
int* Radixsort::Init(int *pa,int dim){
        pa=new int[dim];
        for(int x=0;x<dim;x++){
                pa[x]=0;
        }
        return pa;
}
```

8.4 Classe Bucketsorti

Files che dichiarano e definiscono la classe

- bucketsorti.h
- bucketsorti.cpp

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

⟨*bucketsorti.h*⟩≡

```
class Bucketsorti{

private:
        void RiempiLista(int [], int);
        int *supporto, *ordinati;
        int* Init(int [],int);

public:
        Bucketsorti(){};
        ~Bucketsorti(){};
        int* BucketsortI(int[],int);
};
```

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

⟨*bucketsorti.cpp*⟩≡

```
#include "listai.h"
#include "bucketsorti.h"
#include "radixsort.h"
#define CESTINI 214748
#define COSTANTE 10001
ListaI array_di_listei[CESTINI];

///////////////////////////////////////////////////////////////////////////////
int* Bucketsorti::BucketsortI(int array[], int dim){
        Radixsort r;
        int pos_in_ordinati;
        int lung_lista=0;//lunghezza lista[x]
        pos_in_ordinati=0;//prima posizione libera di ordinati[]
        ordinati=Init(ordinati,dim);
        RiempiLista(array,dim);

        for(int x=0;x<CESTINI;x++){
                lung_lista=array_di_listei[x].DaiLunghezza();
                switch(lung_lista){

                        case 0: //la lista  vuota, non fa nulla
                        break;

                        case 1://ha 1 solo elemento che viene inserito in ordinati[]
                                array_di_listei[x].VaiAlPrimo();
                                ordinati[pos_in_ordinati]=array_di_listei[x].Leggi();
                                pos_in_ordinati++;
                        break;

                        default://la lista ha >1 elementi che passo a Radix Sort
                                array_di_listei[x].VaiAlPrimo();
                                supporto=Init(supporto,lung_lista);
                                for(int y=0;y<lung_lista;y++){
                                        supporto[y]=array_di_listei[x].Leggi();
                                        array_di_listei[x].Prossimo();
                                }
                                supporto=r.RadixsortI(supporto, lung_lista);
                                for(int n=0;n<lung_lista;n++){
                                        ordinati[pos_in_ordinati]=supporto[n];
                                        pos_in_ordinati++;
                                }
                delete supporto;
                }
        }

        return ordinati;
}

///////////////////////////////////////////////////////////////////////////////
void Bucketsorti::RiempiLista(int pieno[], int n){

        int posizione;
```

```
        for(int x=0;x<n;x++){
                posizione=(int)(pieno[x]/COSTANTE);//calcola in quale cestino va messo il numero
                array_di_listei[posizione].Inserisci(pieno[x]);
        }
}

/////////////////////////////////////////////////////////////////////
int* Bucketsorti::Init(int *pa,int dim){
        pa=new int[dim];
        for(int x=0;x<dim;x++){
                pa[x]=0;
        }
        return pa;
}
```

8.5 Classe Rand

Files che dichiarano e definiscono la classe

- rand.h
- rand.cpp

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

⟨*rand.h*⟩≡

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

class Rand{

        private:
                long int mini, maxi;
                int dim;
                int InputInt(int );
                long int RandNumberI(long int, long int);
                void ApriFile(char []);
                void ChiudiFile(char []);
                void Menu();

        public:
                Rand(){}
                ~Rand(){}
                void RiempiFile(char []);
                ofstream Foutput;
};
```

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

⟨*rand.cpp*⟩≡

```
#include "rand.h"
#define MINIMO 0//minimo numero generabile
#define MASSIMO 2147483647//massimo numero generabile

////////////////////////////////////////////////////////////////////////////////
void Rand::RiempiFile(char NomeFile[]){

        long int minint=0, maxint=0;
        int scelta=0, quantinumeri=0, tempi=0, x=0, y=0;
        dim=0;
        mini=MINIMO;
        maxi=MASSIMO;
        Menu();
        scelta=InputInt(3);
        if (scelta==3){cout<<"Fine programma!!!";exit(1);}
        srand (time(NULL));
        ApriFile(NomeFile);
        switch(scelta){

                case 1://riempie il file di numeri interi 0/(2^10)-1
                        minint=0;
                        maxint=1023;
                        cout <<"\nQuanti numeri inserisco nel file (MAX=5.000.000)?: ";
                        quantinumeri=InputInt(5000000);
                        for( x = 0; x < quantinumeri; x++){
                                Foutput<<RandNumberI(minint,maxint)<<"\n";
                        }
                        tempi=mini;mini=maxi;maxi=tempi;

                        Foutput<<"%"<<mini<<"m"<<maxi<<"M"<<dim<<"d";
                        ChiudiFile(NomeFile);
                        break;

                case 2://riempie il file di numeri interi 0/2^31-1
                        minint=0;
                        maxint=2147483647;
                        mini=minint;
                        maxi=maxint;
                        cout <<"\nQuanti numeri inserisco nel file (MAX=5.000.000)?: ";
                        quantinumeri=InputInt(5000000);
                        for( y = 0; y < quantinumeri; y++){
                                Foutput<<RandNumberI(minint,maxint)<<"\n";
                        }
                        tempi=mini;mini=maxi;maxi=tempi;

                        Foutput<<"%"<<mini<<"m"<<maxi<<"M"<<dim<<"d";
                        ChiudiFile(NomeFile);
                        break;
        }

}
////////////////////////////////////////////////////////////////////////////////
void Rand::Menu(){
```

```
        cout <<"\n\n"<<"Scegli con quale tipo di dato rimpire il file...\n";
        cout <<"\n"<<"1=Interi in un range [0 / (2^10)-1]";
        cout <<"\n"<<"2=Interi in un range [0 / (2^31)-1]";
        cout <<"\n"<<"3=Esci dal programa";
        cout <<"\n\n"<<"Digitare scelta: ";
}
////////////////////////////////////////////////////////////////////////////////
int Rand::InputInt(int n)
{
        int valido;
        int num;
        do {
                cin>>num;
                valido =(num >= 1 && num <= n);
                if (!valido) {
                cout <<"\n"<<"Error! (1-"<<n<<"): ";
                }
        } while (!valido);

        return num;
}

////////////////////////////////////////////////////////////////////////////////
long int Rand::RandNumberI(long int max, long int min) {

        long int valore=0;
        valore=min + (long int) ( (max-min) * (double(rand()) / RAND_MAX) );
        dim++;
        if (valore<maxi){maxi=valore;}
        if (valore>mini){mini=valore;}

        return valore;
}

////////////////////////////////////////////////////////////////////////////////
void Rand::ApriFile(char NomeFile[]){

        Foutput.open(NomeFile,ios::out);
        if(!Foutput){
                cerr<<"Errore nella creazione del file!";
                exit(-1);
        }
}

////////////////////////////////////////////////////////////////////////////////
void Rand::ChiudiFile(char NomeFile[]){Foutput.close();}
```

8.6 Classe Listai

Files che dichiarano e definiscono la classe

- listai.h
- listai.cpp

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

⟨*listai.h*⟩≡

```
//original code of Battiti-Brunato, unitn.it
#include <fstream.h>
#include <stdlib.h>

class nodo{

public:

        nodo *prev, *next;
        int data;
};


class ListaI{

public:

        nodo *sentinella, *current;
        int lunghezza;

        ListaI();
        ~ListaI();

        void Prossimo();
        void Precedente();
        void VaiAlPrimo();
        void VaiFineLista();
        int  EVuota();
        int  EPrimo();
        int  FineLista();
        void Scrivi(int);
        int  Leggi();
        void Inserisci(int);
        int  DaiLunghezza();
        int  Cancella();
};
```

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

⟨*listai.cpp*⟩≡

```
//implementazione della classe
#include "listai.h"

///////////////////////////////////////////////////////////////////////////////
ListaI::ListaI(){
                sentinella= new nodo;
                sentinella->next=sentinella;
                sentinella->prev=sentinella;
                current=sentinella;
                lunghezza=0;

        }

///////////////////////////////////////////////////////////////////////////////
ListaI::~ListaI(){
                VaiAlPrimo();
                while(!FineLista())
                        Cancella();
                delete sentinella;
        }

///////////////////////////////////////////////////////////////////////////////
        void ListaI::Prossimo(){
                if(FineLista()){
                        cerr<<"errore in Next: sei oltre l'ultimo nodo!"
                                <<endl;
                        exit(-1);
                        }
                else
                current=current->next;
        }

///////////////////////////////////////////////////////////////////////////////
        void ListaI::Precedente(){
                if(EPrimo()){
                        cerr<<"errore in Precedente: sei sul primo nodo!"
                                <<endl;
                        exit(-1);
                        }
                else
                current=current->prev;
        }

///////////////////////////////////////////////////////////////////////////////
        void ListaI:: VaiAlPrimo(){
                current=sentinella->next;
        }

///////////////////////////////////////////////////////////////////////////////
        void ListaI::VaiFineLista(){
                current=sentinella;
        }
```

```
/////////////////////////////////////////////////////////////////////////////////
        int ListaI::EVuota(){
                return(sentinella==sentinella->next);
        }

/////////////////////////////////////////////////////////////////////////////////
        int ListaI::EPrimo(){
                return(current==sentinella->next);
        }

/////////////////////////////////////////////////////////////////////////////////
        int ListaI::FineLista(){
                return(current==sentinella);
        }

/////////////////////////////////////////////////////////////////////////////////
        void ListaI::Scrivi(int d){
                if(FineLista()){
                        cerr<<"errore in Scrivi: sei oltre l'ultimo nodo!"
                                <<endl;
                        exit(-1);
                        }
                else
                current->data=d;
        }

/////////////////////////////////////////////////////////////////////////////////
        int ListaI::Leggi(){
                if(FineLista()){
                        cerr<<"errore in Leggi: sei oltre l'ultimo nodo!"
                                <<endl;
                        exit(-1);
                        }
                else
                return(current->data);
        }

/////////////////////////////////////////////////////////////////////////////////
        void ListaI::Inserisci(int d){
                nodo *newnodo=new nodo;

                newnodo->next=current;
                newnodo->prev=current->prev;
                newnodo->data=d;
                current->prev=newnodo;
                newnodo->prev->next=newnodo;
                lunghezza++;
        }

/////////////////////////////////////////////////////////////////////////////////
        int ListaI::DaiLunghezza(){
                return lunghezza;
        }

/////////////////////////////////////////////////////////////////////////////////
```

```
int ListaI::Cancella(){
        if(FineLista()){
                cerr<<"errore in Cancella: sei oltre l'ultimo nodo!"
                        <<endl;
                exit(-1);
                }
        int tmp=current->data;
        nodo* tmpp=current->next;

        current->next=current->prev;
        current->prev=current->prev;

        delete current;
        lunghezza--;
        current=tmpp;
        return tmp;
}
```