1. Radix Sort.
Tra gli algoritmi di sorting più celebri figura il QuickSort. Molti fra voi avranno già sentito questo nome.
Nel corso degli anni QuickSort è stato ampiamente studiato ed applicato alle situazioni più disparate.
Una implementazione di QuickSort la potete trovare addirittura nel set di funzioni standard del linguaggio C, sebbene detta implementazione lasci a desiderare.
QuickSort è una funzione ricorsiva, la cui filosofia è: *Dividi e Conquista*.
E' una strategia efficace che può dare risultati strabilianti.
Tuttavia nel caso in cui l'array da riordinare sia di grosse dimensioni, QuickSort è costretta ad invocare se stessa ricorsivamente un elevato numero di volte.
Se è vero che ad ogni iterazione la funzione lavora su 2 subset sempre più piccoli dei precedenti (invocando se stessa 2 volte), è anche vero che ad ogni nuova ricursione la funzione causa un allargamento dello stack necessario per contenere tutte le istanze di QuickSort al momento in esecuzione.
Tutto ciò provoca un degrado prestazionale che progredisce e regredisce ciclicamente fino al completamento del lavoro.
Trattandosi inoltre di un algoritmo fondamentalmente comparativo, QuickSort soffre particolarmente in situazioni ad entropia elevata.
In parole semplici, quanto più è disordinata la sequenza di dati, tante più sono le ricursioni effettuate da QuickSort.
Pertanto QuickSort è meglio utilizzato su moderate quantità di dati.

Laddove QuickSort fallisce, un altro algoritmo di sorting, sconosciuto alle grandi masse, eccelle superlativamente.
Questo algoritmo trova applicazione nei casi in cui la velocità di esecuzione sia un *must*, e la mole di dati da trattare sia considerevole (o comunque superiore alla mole ottimale per QuickSort).

Il suo nome è RadixSort.
E' il Re incontrastato del sorting, e come tutti i Re si dà un sacco di arie e preferisce scomodarsi solo quando ce n'è effettiva necessità. Spacconate a parte, per mettere in moto RadixSort avete un costo di attivazione da pagare in termini di calcolo.
Se usato su quantità *irrisorie* di elementi, RadixSort non solo è sprecato, ma risulta controproducente per voi, in quanto i costi di attivazione tendono a superare i guadagni in velocità.

Indicativamente parlando, se avete da riordinare solo un centinaio di elementi, gli altri algoritmi di sorting, come QuickSort (il vice-Re), sono candidati migliori.
Ma nel momento in cui il numero di elementi sale, il divario in prestazioni tra RadixSort ed ogni altro algoritmo aumenta vertiginosamente!

Il principio su cui si basa RadixSort è di una semplicità disarmante.
La parola *Radix* significa radice (niente a che vedere con la Radice Quadrata).
Radice è un sinonimo di cifra.
L'algoritmi quindi effettua un sorting basato sulle cifre che compongono gli elementi da riordinare.
RadixSort non è un algoritmo comparativo, pertanto il livello di entropia della sequenza da riordinare è totalmente ininfluente.

2. Teoria e pratica.
Come prima cosa dovete sapere che RadixSort lavora effettuando un numero di passaggi pari al numero di radici di cui si compone l'elemento con magnitudine maggiore.
Più semplicemente, RadixSort effettua un numero di passaggi pari alla quantità di cifre che compone il

numero più grande tra quelli che volete riordinare.

Considerate la sequenza di numeri:

```
10, 51, 54, 88, 190, 207
```

E' evidente che il numero più grande è 207, composto da 3 cifre (2, 0 e 7).
Quindi il numero di passaggi da effettuare con RadixSort (in questo caso) sarà 3.

Sto per illustrarvi il procedimento utilizzato da RadixSort, così come lo potreste fare voi con carta e penna.
Prima però (e solo per maggiore chiarezza) è opportuno uniformare la sequenza di numeri del nostro esempio, così che siano tutti a 3 cifre.
Mi limiterò ad anteporre degli Zeri iniziali, dove necessario:

```
010, 051, 054, 088, 190, 207
```

Naturalmente è anche il caso di mescolare questi numeri alla rinfusa:

```
190, 051, 054, 207, 088, 010
```

Per poter procedere con il sorting a mano ci servono dei *contenitori* per accomodare temporaneamente i vari elementi che vogliamo riordinare.
Che cosa rappresentino esattamente questi contenitori ve lo spiegherò in seguito.
Per ora vi basti sapere che servono per spostare i nostri numeri durante il procedimento.

E quanti contenitori servono?
Ne occorrono tanti quanti sono gli stati che può assumere una cifra.

I numeri del nostro esempio sono tuti in base 10 (per comodità nostra).
Nella numerazione decimale, ogni cifra che compone un numero può assumere 10 stati diversi, cioè da 0 a 9. Quindi useremo 10 contenitori.

Se i nostri numeri fossero stati in binario, avremmo usato 2 contenitori.
Se invece fossero stati in esadecimale, avremmo usato 16 contenitori.
Scolpitevi nella mente questo dettaglio.

Immaginate i contenitori come tanti piatti sui quali andremo a posare i nostri numeri.
Eccoli qui.

```
---  ---  ---  ---  ---  ---  ---  ---  ---  ---
"0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"
```


Ciascuno di essi è etichettato con il "nome" di una cifra.
Come ho detto prima RadixSort riordina per cifre, ed esegue sempre un numero di passaggi pari alla quantità di cifre dell'elemento più grande.
Avendo numeri a 3 cifre, effettueremo 3 passaggi.

Ad ogni passaggio considereremo una radice (o cifra) ben precisa.
Al 1° passaggio considereremo solo la radice delle Unità.
Al 2° passaggio considereremo solo la radice delle Decine.
Al 3° passaggio considereremo solo la radice delle Centinaia.

Alla fine del 3° passaggio avremo ottenuto la sequenza di numeri riordinata.

1° Passaggio: ordina in base alle Unità

```
Input: 190, 051, 054, 207, 088, 010

010
190  051            054            207  088
---  ---  ---  ---  ---  ---  ---  ---  ---  ---
"0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"

Output: 190, 010, 051, 054, 207, 088
```

Che cosa ho appena fatto?
Ho letto la sequenza di input da sinistra a destra.
Basandomi sui valori delle loro Unità, ho inserito i numeri nei contenitori appropriati.
Quindi:

- `190 (Unita: 0) --> contenitore "0"`
- `051 (Unità: 1) --> contenitore "1"`
- `054 (Unità: 4) --> contenitore "4"`
- `207 (Unità: 7) --> contenitore "7"`
- `088 (Unità: 8) --> contenitore "8"`
- `010 (Unità: 0) --> contenitore "0" (impilato sopra il 190)`

Finito il passaggio, ho estratto i numeri dai contenitori per rimetterli in sequenza lineare.
L'ordine di estrazione è: dal contenitore di sinistra al contenitore di destra.
E per ogni contenitore: dal numero più in basso al numero più in alto.

La sequenza di output del 1° passaggio è quindi:

```
190, 010, 051, 054, 207, 088
```

che diventa la sequenza di input per il passaggio successivo.

2° Passaggio: ordina in base alle Decine

```
Input: 190, 010, 051, 054, 207, 088

                       054
207  010               051            088  190
---  ---  ---  ---  ---  ---  ---  ---  ---  ---
"0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"

Output: 207, 010, 051, 054, 088, 190
```

Come nel passaggio precedente, ho letto la sequenza di input da sinistra a destra.
Ma stavolta ho inserito i numeri nei contenitori basandomi sulle Decine.

- `190 (Decina: 9) --> contenitore "9"`

- 010 (Decina: 1) --> contenitore "1"
- 051 (Decina: 5) --> contenitore "5"
- 054 (Decina: 5) --> contenitore "5" (impilato sopra lo 051)
- 207 (Decina: 0) --> contenitore "0"
- 088 (Decina: 8) --> contenitore "8"

Come prima, ho estratto i numeri dai contenitori per rimetterli in sequenza lineare.
L'ordine di estrazione è sempre: dal contenitore di sinistra al contenitore di destra.
E per ogni contenitore è sempre: dal numero più in basso al numero più in alto.

La sequenza di output del 2° passaggio è quindi:

```
207, 010, 051, 054, 088, 190
```

che diventa la sequenza di input per il passaggio successivo.

3° Passaggio: ordina in base alle Centinaia

```
Sequenza: 207, 010, 051, 054, 088, 190

088
054
051
010  190  207
---  ---  ---  ---  ---  ---  ---  ---  ---  ---
"0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"

Output: 010, 051, 054, 088, 190, 207
```

Come nei passaggi precedenti, ho letto la sequenza di input da sinistra a destra.
Ma stavolta ho inserito i numeri nei contenitori basandomi sulle Centinaia:

- 207 (Centinaia: 2) --> contenitore "2"
- 010 (Centinaia: 0) --> contenitore "0"
- 051 (Centinaia: 0) --> contenitore "0" (impilato sopra lo 010)
- 054 (Centinaia: 0) --> contenitore "0" (impilato sopra lo 051)
- 088 (Centinaia: 0) --> contenitore "0" (impilato sopra lo 054)
- 190 (Centinaia: 1) --> contenitore "1"

Per l'ultima volta ho estratto i numeri dai contenitori per rimetterli in sequenza lineare.
L'ordine di estrazione non cambia mai.
Sempre dal contenitore di sinistra al contenitore di destra, e sempre dal numero più in basso al numero più in alto.

La sequenza di output del 3° e ultimo passaggio è quindi:

```
010, 051, 054, 088, 190, 207
```

Cioè la sequenza finale, riordinata.
Applausi, grazie :)

2.1 Non sarà così facile.
E' più che probabile che abbiate compreso lo svolgimento a mano già al primo colpo.
Magari lo avete riletto una volta o due, tanto per sentirvi sicuri.
Ma sappiate che la parte facile termina qui. La trasposizione informatica dell'algoritmo esegue in un colpo solo una sequenza di operazioni quantomai ostica da digerire e comprendere.
Se anche vi mostrassi subito il codice, infarcito di commenti, non ci capireste niente.

Farò del mio meglio per spiegarvi in dettaglio cosa-succede-dove (e perchè).
Ma le spiegazioni sono lunghe e noiose.
Vi chiedo di aver pazienza e leggere tutto con attenzione. Non ve ne pentirete.

3. Istogrammi e Indici.
Non lo avrete notato (perchè non è intuitivo per niente) ma tutte le volte che abbiamo inserito i numeri nei contenitori, abbiamo composto degli Istogrammi di distribuzione delle cifre.
In 3 passaggi (perchè 3 erano le cifre) abbiamo costruito 3 Istogrammi diversi: uno per le Unità, uno per le Decine e uno per le Centinaia.
Ciascun Istogramma ci dice con quali e quante cifre abbiamo a che fare per quel dato passaggio.

Non ci avete capito niente? Date un'occhiata al 3° passaggio.
Il contenitore "0" contiene 4 numeri. Quindi per quella radice (le Centinaia), l'Istogramma ci dice che abbiamo 4 cifre "0".
Il contenitore "1" contiene 1 numero. L'istogramma ci dice che abbiamo 1 cifra "1".
Lo stesso vale per la cifra "2", ce n'è una sola.
Infine l'Istogramma ci dice che abbiamo 0 cifre per i contenitori da 3 a 9.

Quindi, rileggendo per bene l'Istogramma, per la terza radice abbiamo la seguente distribuzione:

- `4 cifre 0`
- `1 cifra 1`
- `1 cifra 2`
- `0 cifre 3`
- `0 cifre 4`
- `0 cifre 5`
- `0 cifre 6`
- `0 cifre 7`
- `0 cifre 8`
- `0 cifre 9`

Se vi viene più facile, pensate agli Istogrammi come a dei *contatori* per il numero di occorrenze delle varie cifre.
Naturalmente effettuando la sommatoria di tutti i contatori in un Istogramma, otteniamo sempre il numero di elementi dell'intera sequenza:

```
#Elementi: 4 + 1 + 1 + 0 + 0 + 0 + 0 + 0 + 0 + 0 = 6
```

Proseguiamo.
C'è un'ulteriore informazione che si può estrapolare dall'esame di ciascun passaggio.
Date sempre un'occhio al 3° passaggio.
Osservando il numero di elementi presenti in ciascun contenitore, è possibile ricavare una Tabella di Indici. C'è anche chi la chiama *Tabella di Offset*, perchè i valori contenuti in essa sono allo stesso

tempo Indici e Offset. Noi la chiameremo Tabella degli Indici.

Guardate al contenitore "0", sempre nel 3° passaggio.
Quanti numeri contiene?
Sono 4, e cioè: 010, 051, 054 e 088.
E dopo quante estrazioni abbiamo finalmente estratto il primo (e unico) numero che risiedeva nel contenitore "1" (cioè il numero 190)?
Dopo 4 estrazioni, cioè solo dopo aver estratto tutti i numeri del contenitore precedente.
Quindi l'Indice del primo numero del contenitore "1", è 4.
Similmente, possiamo affermare che l'indice del numero 010 è 0 (perchè viene estratto per 1°).
L'indice dello 051 è 1 (estratto per 2°), l'indice dello 054 è 3, e così via...

3.1 Doppia indicizzazione.
RadixSort deve gran parte della sua straordinaria velocità, al fatto che, per ogni passaggio, la posizione finale di ciascun elemento nell'array **è già nota**.
Oguno degli elementi da riordinare raggiunge infatti la propria posizione finale in *n* spostamenti, dove *n* è il numero di passaggi effettuati.
Questa invidiabile proprietà non è presente in nessun altro algoritmo di sorting.

Ma come fa RadixSort a conoscere con esattezza la posizione finale di un elemento, se non effettua alcuna comparazione con gli altri elementi?
Il segreto risiede nelle informazioni contenute negli Istogrammi.

La trasposizione informatica dell'algoritmo utilizza gli Istogrammi per comporre una Tabella di Indici.
La Tabella di Indici permette di indicizzare a colpo sicuro nell'array di destinazione, e trovare così in una scheggia di tempo la posizione finale di ciascun elemento.

Adesso non confondetevi le idee, ma è doveroso precisare che ciò che continuo a chiamare "Tabella" in realtà non è una tabella.
Bensì è un array mono-dimensionale, contenente tanti elementi quanti sono i diversi stati che può assumere una cifra.

Niente panico.
Nel nostro esempio abbiamo usato numeri in base 10 (perchè sono comodi per noi).
Base 10 significa che ogni cifra può assumere 10 stati diversi, cioè da 0 a 9.
Quindi la Tabella degli Indici che ricaviamo dal nostro esempio contiene 10 elementi.
Non uno di più, non uno di meno.

Al variare della base della numerazione, varia anche il numero di indici nella Tabella.
In base 16 sarebbero 16 indici, in base 2 sarebbero 2 indici, etc etc.

Ora... questa Tabella di Indici viene utilizzata come se fosse un array di puntatori.
Si tratta di una doppia indicizzazione.

Sarò più chiaro:
Dopo aver composto la Tabella degli Indici, RadixSort esamina la sequenza di input.
Di ogni numero viene considerata solo la radice (o cifra) pertinente al passaggio in corso.
Questa cifra viene usata per indicizzare all'interno della Tabella degli Indici.
L'elemento puntato, nella Tabella, contiene a sua volta un numero.
Questo numero identifica un elemento della sequenza di output del passaggio in corso.
E RadixSort muove il numero, a colpo sicuro, dalla sequenza di input a quella di output.
Il procedimento continua fino all'esaurimento degli input.

Se avete difficoltà a capire, provate a tracciare uno schema su carta.
Se avete capito, non mi resta che spiegarvi come viene composta la Tabella degli Indici.

3.2 Composizione degli Indici.
Facciamo un breve passo indietro.
Riporto le 3 sequenze di input per i 3 passaggi effettuati nel nostro esempio.
Osservatele bene:

```
1° Passaggio. Input: 190, 051, 054, 207, 088, 010
2° Passaggio. Input: 190, 010, 051, 054, 207, 088
3° Passaggio. Input: 207, 010, 051, 054, 088, 190
```

L'input del 1° passaggio era una sequenza completamente disordinata.
Il 1° passaggio ha riordinato gli elementi tenendo conto delle radici delle Unità.
L'output riordinato del 1° passaggio è divenuto l'input da riordinare per il 2° passaggio.
Il 2° passaggio ha riordinato gli elementi tenendo conto delle radici delle Decine.
L'output riordinato del 2° passaggio è divenuto l'input da riordinare per il 3° passaggio.
Il 3° passaggio ha riordinato gli elementi tenendo conto delle radici delle Centinaia.
L'output riordinato del 3° passaggio è divenuto la sequenza finale riordinata.

Ripeto ora le stesse 3 sequenze di input, ma usando una colorazione differente:

```
1° Passaggio. Input: 190, 051, 054, 207, 088, 010
2° Passaggio. Input: 190, 010, 051, 054, 207, 088
3° Passaggio. Input: 207, 010, 051, 054, 088, 190
```

Notato niente?
Escludendo l'input del 1° passaggio, in quanto sequenza non ancora rimaneggiata dall'algoritmo, abbiamo che all'inizio del 2° passaggio le Unità degli elementi sono già messe in ordine crescente (0, 0, 1, 4, 7, 8).
All'inizio del 3° passaggio abbiamo che non solo le Unità, ma anche le Decine sono già messe in ordine crescente (07, 10, 51, 54, 88, 90).
In un 4° ipotetico passaggio avremmo che sia le Unità che le Decine che le Centinaia sarebbero messe in ordine crescente.
Ma il nostro esempio si è fermato alla fine del 3° passaggio (che per l'appunto potremmo considerare come l'inizio di un 4° ipotetico passaggio).

Torniamo a noi.
Ogni passaggio si preoccupa di riordinare gli elementi **solo** in base alla radice sotto esame, forte del fatto che le radici precedenti risultano automaticamente riordinate dai passaggi precedenti (come appare evidente da quanto appena dimostrato).

Dovendo così preoccuparsi di **una sola radice alla volta**, ogni passaggio trova nel proprio Istogramma tutte le informazioni necessarie per svolgere la propria parte di lavoro, senza per questo compromettere il buon esito dei passaggi precedenti.

Riporto ora la configurazione dei contenitori riempiti nel 3° passaggio del nostro esempio:

```
088
054
051
010  190  207
```

```
---  ---  ---  ---  ---  ---  ---  ---  ---  ---
"0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"
```

Come potete notare, per lo stato "0" esistono 4 elementi.
Per lo stato "1" esiste 1 elemento, così come per lo stato "2".
Tutte gli altri stati (da "3" a "9") possiedono 0 elementi (sono vuoti).
Queste sono le informazioni contenute in un Istogramma (in questo caso nell'Istogramma del 3° passaggio).

Da queste informazioni costruiamo la relativa Tabella degli Indici.
Vi ricordo che il numero di elementi nella Tabella è pari al numero di stati (nel nostro caso 10).

Quindi, la nostra Tabella degli Indici contiene 10 elementi (ancora vuoti):

```
Elemento: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Indice:   ?, ?, ?, ?, ?, ?, ?, ?, ?, ?
```

Iniziamo dallo stato "0" (associato all'elemento 0 della Tabella).
Non essendo preceduto da altri stati, lo stato "0" assume il valore 0.

```
Elemento: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Indice:   0, ?, ?, ?, ?, ?, ?, ?, ?, ?
```

Passiamo allo stato "1" (associato all'elemento 1 della Tabella).
Lo stato **precedente** allo stato "1", contiene 4 elementi (010, 051, 054, 088).
(l'Istogramma ci dice che -per le Centinaia- abbiamo 4 cifre "0")
Quindi lo stato "1" assume il valore 4.
A questo valore dobbiamo poi aggiungere il valore contenuto nello stato precedente.
Ma lo stato precedente (lo stato "0") ha valore 0. Quindi 0 + 4 = 4.

```
Elemento: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Indice:   0, 4, ?, ?, ?, ?, ?, ?, ?, ?
```

Passiamo allo stato "2" (associato all'elemento 2 della Tabella).
Lo stato precedente allo stato "2", contiene 1 elemento (190).
(l'Istogramma ci dice che abbiamo 1 cifra "1")
Quindi lo stato "2" assume il valore 1.
A questo valore dobbiamo poi aggiungere il valore contenuto nello stato precedente.
Lo stato precedente (lo stato "1") ha valore 4. Quindi 1 + 4 = 5.

```
Elemento: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Indice:   0, 4, 5, ?, ?, ?, ?, ?, ?, ?
```

Passiamo allo stato "3" (associato all'elemento 3 della Tabella).
Lo stato precedente allo stato "3", contiene 1 elemento (207).
(l'Istogramma ci dice che abbiamo 1 cifra "2")
Quindi lo stato "3" assume il valore 1.
A questo valore dobbiamo poi aggiungere il valore contenuto nello stato precedente.
Lo stato precedente (lo stato "2") ha valore 5. Quindi 1 + 5 = 6.

```
Elemento: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Indice:   0, 4, 5, 6, ?, ?, ?, ?, ?, ?
```

Passiamo allo stato "4" (associato all'elemento 4 della Tabella).
Lo stato precedente allo stato "4", contiene 0 elementi (è vuoto).
(l'Istogramma ci dice che abbiamo 0 cifre "3")
Quindi lo stato "4" assume il valore 0.
A questo valore dobbiamo poi aggiungere il valore contenuto nello stato precedente.
Lo stato precedente (lo stato "3") ha valore 6. Quindi 0 + 6 = 6.

```
Elemento: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Indice:   0, 4, 5, 6, 6, ?, ?, ?, ?, ?
```

Tagliamo corto. Da qui in poi gli stati sono tutti vuoti. Ripetendo il processo fino alla fine scopriamo che assumono tutti il valore 0 + 6 = 6.
Quindi la Tabella composta per il 3° passaggio è:

```
Elemento: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Indice:   0, 4, 5, 6, 6, 6, 6, 6, 6, 6
```

3.3 Utilizzo degli Indici.
Ora che sappiamo come comporre la Tabella degli Indici, vediamo come utilizzarla.
Vi ho già spiegato il discorso della doppia indicizzazione.
Quindi quanto segue (abbinato al contenuto della Tabella degli Indici) dovrebbe risultarvi di comprensione immediata.

Sempre con riferimento al 3° passaggio del nostro esempio (quello che riordina in base alle Centinaia), descrivo a parole il ragionamento algoritmico seguito nella trasposizione informatica di RadixSort:

Estrai il 1° elemento dall'array di input (010). Le sue Centinaia sono 0.
Leggi l'elemento 0 della Tabella degli Indici. Contiene il valore 0.
La posizione finale dell'elemento 010, nell'array di output è quindi 0.
Incrementa di 1 l'elemento della Tabella che abbiamo appena letto.
L'elemento 0 della Tabella passa così dal valore 0 al valore 1.

Estrai il 2° elemento dall'array di input (051). Le sue Centinaia sono 0.
Leggi l'elemento 0 della Tabella degli Indici. Contiene il valore 1.
La posizione finale dell'elemento 051, nell'array di output è quindi 1.
Incrementa di 1 l'elemento della Tabella che abbiamo appena letto.
L'elemento 0 della Tabella passa così dal valore 1 al valore 2.

Estrai il 3° elemento dall'array di input (054). Le sue Centinaia sono 0.
Leggi l'elemento 0 della Tabella degli Indici. Contiene il valore 2.
La posizione finale dell'elemento 054, nell'array di output è quindi 2.
Incrementa di 1 l'elemento della Tabella che abbiamo appena letto.
L'elemento 0 della Tabella passa così dal valore 2 al valore 3.

Estrai il 4° elemento dall'array di input (088). Le sue Centinaia sono 0.
Leggi l'elemento 0 della Tabella degli Indici. Contiene il valore 3.
La posizione finale dell'elemento 088, nell'array di output è quindi 3.
Incrementa di 1 l'elemento della Tabella che abbiamo appena letto.
L'elemento 0 della Tabella passa così dal valore 3 al valore 4.

Estrai il 5° elemento dall'array di input (190). Le sue Centinaia sono 1.

Leggi l'elemento 1 della Tabella degli Indici. Contiene il valore 4.
La posizione finale dell'elemento 190, nell'array di output è quindi 4.
Incrementa di 1 l'elemento della Tabella che abbiamo appena letto.
L'elemento 1 della Tabella passa così dal valore 4 al valore 5.

Estrai il 6° elemento dall'array di input (207). Le sue Centinaia sono 2.
Leggi l'elemento 2 della Tabella degli Indici. Contiene il valore 5.
La posizione finale dell'elemento 207, nell'array di output è quindi 5.
Incrementa di 1 l'elemento della Tabella che abbiamo appena letto.
L'elemento 2 della Tabella passa così dal valore 5 al valore 6.

Tutti gli elementi sono stati estratti dall'array di input ed inseriti nell'array di output.
Trattandosi del 3° e ultimo passaggio del nostro esempio, abbiamo che la sequenza finale riordinata è:

```
010, 051, 054, 088, 190, 207
```

Applausi.
Oh! Forzaa! Fate vedere che apprezzate :)

4. Partire preparati.
Sappiamo tutto. Possiamo scrivere del codice funzionale.
Ma senza un paio di ottimizzazioni (niente di complicato) il nostro algoritmo non può far sfoggio di tutta la velocità di cui è capace. Invece noi vogliamo che corra!

4.1 Base 256.
Ricordate quando ho spiegato di quanti contenitori avremmo avuto bisogno per riordinare a mano?
E' la base della numerazione a determinare il numero di contenitori.
Contemporaneamente è la base a determinare quanti diversi stati può assumere una singola cifra (o radice).
Ed è sempre la base (seppur indirettamente) a dettare il numero di cifre, e quindi il numero di passaggi da effettuare con RadixSort.
Tra le numerazioni comunemente note, l'esadecimale è senza dubbio quella con la base più grande (base 16).
Ma nulla ci vieta di salire ben oltre la base 16, e ridurre così il numero di cifre/passaggi.

Considerate il numero: 4294967296. Lo avete riconosciuto, è 2 alla 32esima.
Lo stesso numero, ma espresso in esadecimale, diventa: `0xFFFFFFFF`.
Aumentando la base (da 10 a 16), il numero di cifre è diminuito (da 10 a 8).

Ma sappiamo bene che `0xFFFFFFFF` altro non rappresenta che 4 byte uno dietro l'altro.
Ogni byte contiene 8 bit, e 8 bit esprimono 256 stati (2 alla 8ava).
E allora perchè non fare finta che i nostri numeri siano in base 256?
In questo modo tagliamo **drasticamente** il numero di cifre/passaggi da effettuare con RadixSort.
Solo 4 anzichè 10, per interi da 32 bit.
Solo 8 anzichè 20, per interi da 64 bit. Volete mettere il guadagno?
Tuttavia questo articolo tratta esclusivamente gli interi da 32 bit.

4.2 Tutti insieme appassionatamente... però non tutti.
Ad ogni passaggio effettuato corrisponde un Istogramma per quel dato passaggio.

Quindi *n* passaggi = *n* Istogrammi. Semplice.
Ma i passaggi (e così gli Istogrammi) si basano su singole radici (o cifre).
Dal momento che quando leggiamo un numero abbiamo a disposizione tutte le cifre che lo compongono, nulla ci vieta di comporre gli Istogrammi tutti insieme in un colpo solo.
In questo modo analizziamo l'intera sequenza di numeri una volta sola anzichè tante quanti sono i passaggi richiesti.

Per la Tabella degli Indici possiamo fare un discorso diverso.
E' vero che essa dipende da un singolo Istogramma (come abbiamo visto).
Ed è vero che, come per gli Istogrammi, abbiamo tante Tabelle di Indici quanti sono i passaggi.
Sarebbe possibile approfittare di questo fatto per comporre le Tabelle di Indici tutte insieme, proprio come faremmo per gli Istogrammi... tuttavia non ne approfitteremo.
Per 3 ottime ragioni:
1) ciascuna Tabella di Indici, così come ciascun Istogramma, ci costa memoria.
2) le Tabelle di Indici, a differenza degli Istogrammi, sono compilabili *al volo*.
3) esistono varianti di RadixSort nelle quali la composizione contemporanea di tutte le Tabelle necessarie è quantomai problematica.
(Spiegherò ogni cosa a tempo debito. Per ora fidatevi delle mie parole e accontentatevi di comporre le Tabelle degli Indici una alla volta.)

4.3 Riutilizzare gli array.
Avete visto come per ogni passaggio i numeri da riordinare siano *usciti* dall'array di input per *entrare* nell'array di output.
Ma quanti array servono davvero? Se ci pensate, durante lo svolgimento di ciascun passaggio erano solo due gli array utilizzati contemporaneamente.
Quindi ci servono sempre e solo due array, indipendentemente dal numero di passaggi.

Questi array devono poter contenere l'intera sequenza di numeri da riordinare.
Uno dei due lo abbiamo già: è quello di input dato in pasto al 1° passaggio.
Quindi aggiungiamo solo un altro array, gemello del primo, ma vuoto.

Durante l'esecuzione dell'algoritmo il nostro codice scambierà di posto i puntatori ai due array, simulando un *trasferimento istantaneo* di tutti gli elementi da un array all'altro.
(l'output diverrà l'input, l'input verrà riordinato nell'output, l'output diverrà l'input...)

5. Sorting a 32 bit - unsigned integer.
Finalmente del codice!!
Se avete letto e capito tutto fino a qui (siete stati bravi), quanto segue non vi riserverà sorprese:

```
// [desc]
//       RadixSort su interi senza segno a 32 bit.
//       NON usare per interi con segno!
//
// [in,out]
//         pSort : Array di elementi da riordinare.
// [in]
//     pTmp : Array temporaneo, e gemello di pSort.
//            La funzione lo utilizza come memoria di
//            scambio per depositare temporaneamente i
//            numeri riordinati tra un passaggio e l'altro.
// [in]
```

```
//     Count : Numero di elementi da riordinare.
//
void Radix32u (DWORD* pSort, DWORD* pTmp, DWORD Count)
{
    // Array statico grande abbastanza da contenere tutti gli
    // Istogrammi che occorrono, più una singola Tabella di
    // Indici.
    // 4 passaggi = 4 Istogrammi.
    // Ogni Istogramma richiede 1 KB di memoria (256 DWORD).
    // Anche la Tabella degli Indici richiede 1 KB.
    // Totale: 5 KB.
    // Tutto ruota intorno al numero 256 (cioe: base 256).
    static DWORD Array [256 * (4 + 1)];

    // Azzera il contenuto di tutti gli Istogrammi.
    // Gli Istogrammi sono contatori. Devono partire da 0.
    ::ZeroMemory (Array, 256 * sizeof (DWORD) * 4);

    // Prepara alcuni puntatori per indicizzare nell'Array.
    DWORD* pHist0 = &Array [0];    // Istogramma 1^ radice.
    DWORD* pHist1 = &Array [256];  // Istogramma 2^ radice.
    DWORD* pHist2 = &Array [512];  // Istogramma 3^ radice.
    DWORD* pHist3 = &Array [768];  // Istogramma 4^ radice.
    DWORD* pIndex = &Array [1024]; // Tabella degli Indici.

    // Compone tutti gli Istogrammi.
    for (DWORD i = 0; i < Count; ++i)
    {
        DWORD Value = pSort [i];
        ++pHist0 [(Value >> 0 ) & 0xFF];
        ++pHist1 [(Value >> 8 ) & 0xFF];
        ++pHist2 [(Value >> 16) & 0xFF];
        ++pHist3 [(Value >> 24) & 0xFF];
    }

    // Punta al 1° Istogramma.
    DWORD* pHistX = pHist0;

    // Tante iterazioni quanti sono i passaggi.
    for (DWORD Bit = 0; Bit < 32; Bit += 8)
    {
        // Compone la Tabella degli Indici.
        pIndex [0] = 0;

        for (DWORD i = 0; i < (256 - 1); ++i)
        {
            pIndex [i + 1] = pIndex [i] + pHistX [i];
        }

        // Riordina gli elementi.
        for (DWORD i = 0; i < Count; ++i)
        {
            DWORD Value = pSort [i];
            pTmp [pIndex [(Value >> Bit) & 0xFF]++] = Value;
        }

        // Scambia i puntatori degli array.
        DWORD* pSwap = pSort;
```

```
                pSort = pTmp;
                pTmp  = pSwap;

        // Punta all'Istogramma successivo.
        pHistX += 256;
    }
}
```

Ma come? Tutto qui?
Tutto qui :) Ve l'avevo detto che RadixSort è di una semplicità disarmante.

5.1 Mettetelo alla prova.
Per testare la funzione **Radix32u()** potete copiare quanto segue nella vostra main:

```
#if defined _DEBUG
    #define halt __asm { int 3 }
#else
    #define halt
#endif // _DEBUG

// 10K elementi: acqua fresca...
// Dategli MOLTO di più se volete metterlo alla frusta.
const DWORD kNumElem = 10000;

// Istanzia un array e lo popola di valori pseudo-casuali.
DWORD* pSort = new DWORD [kNumElem];
for (DWORD i = 0; i < kNumElem; ++i)
{
    pSort [i] = ::rand ();
}

// Istanzia un array di lavoro per il RadixSort.
DWORD* pTmp = new DWORD [kNumElem];

LARGE_INTEGER Freq, Start, Stop;
::QueryPerformanceFrequency (&Freq);

// Questo è un punto di interruzione software (solo debug).
halt;
// Approfittatene per controllate che pSort contenga
// numeri a casaccio.

// Tempo di inizio.
::QueryPerformanceCounter (&Start);

// RadixSort... corri, CORRI!!!
Radix32u (pSort, pTmp, kNumElem);

// Tempo di fine.
::QueryPerformanceCounter (&Stop);

// Quanto ci abbiamo messo?
float Tmsec = float (double (Stop.QuadPart - Start.QuadPart) / double
(Freq.QuadPart)) * 1000.0f;

// Questo è un punto di interruzione software (solo debug).
halt;
```

```
// Tmsec: tempo impiegato, in millesimi di secondo.
// pSort: la sequenza di numeri riordinata.
// pTmp:  spazzatura. Buttatelo via.

delete [] pTmp;
delete [] pSort;
```

6. Un momento! E gli interi con segno??
I più intraprendenti fra voi si saranno accorti che qualcosa non va nella **Radix32u()**.
Se le date in pasto degli interi signed, la sequenza riordinata risulta sballata.

Spiego subito l'arcano.
RadixSort è come un carroarmato. Non va per il sottile. Lui guarda alla maschera dei bit dei vostri numeri e li riordina. Il **segno** di un numero non sa nemmeno dove sta di casa.
Dobbiamo dirglielo noi.

6.1 Capire il problema del segno.
Il segno di un intero è sempre indicato dall'MSB (Most Significant Bit), cioè il bit più significativo, cioè il 32° bit, cioè il bit più a << sinistra << nella vostra maschera.
(Parentesi...
Le nostre CPU utilizzano la codifica **Little Endian**, nella quale l'MSB è quello più a sinistra. Sappiate che esiste anche la codifica **Big Endian**, tipica dei processori Motorola, nella quale l'MSB è invece quello più a destra. Un esempio di utilizzo di codifica Big Endian lo potete trovare nei file musicali MIDI, con estensione ***.mid**
... chiusa parentesi)

Ecco qui 32 bit, suddivisi in 4 Byte:

```
0000.0000 | 0000.0000 | 0000.0000 | 0000.0000
4° Byte     3° Byte     2° Byte     1° Byte
```

Il bit del segno è quello marcato in rosso. Come potete vedere risiede nel 4° Byte.
Quindi il *bug* che si verifica nella funzione **Radix32u()** si annida nel 4° passaggio.

Il bug si verifica perchè i numeri signed positivi hanno il bit del segno impostato a 0.
Mentre i numeri signed negativi hanno il bit del segno impostato a 1.

Ne consegue che RadixSort vede i numeri negativi come maggiori dei numeri positivi.
Pertanto nell'array finale riordinato vedrete apparire la sequenza dei numeri positivi seguita dalla sequenza dei numeri negativi.

6.2 Correggere il problema del segno.
Occorre innanzitutto sapere quanti sono i numeri negativi nella sequenza.
Contarli è facile (e veloce). Dobbiamo rileggere una parte dell'Istogramma del 4° Byte.
Più precisamente dobbiamo rileggere gli ultimi 127 contatori, cioè da 128 a 255.
Perchè proprio quelli?

Isoliamo per un attimo il 4° Byte del nostro intero signed:

```
0000.0000
4° Byte
```

Secondo la numerazione binaria (visto che si tratta di bit), sappiamo che i numeri che fanno uso dell'ultimo bit (il segno) sono quelli maggiori di 127.
Infatti fino a 127 (che sia signed o unsigned), abbiamo la seguente maschera di bit:

```
0111.1111 = 127
```

Nella quale il bit del segno è sempre spento.
Dal valore 128 in avanti (cioè fino al 255), il bit del segno è sempre acceso:

```
1000.0000 = 128
1111.1111 = 255
```

Ne consegue che se vogliamo sapere con quanti valori negativi abbiamo a che fare, dobbiamo eseguire la sommatoria dei valori contenuti nell'Istogramma del 4° Byte, per gli stati che vanno da 128 a 255.
Non a caso ho detto *stati*. Il nostro RadixSort ragiona in base 256. Per lui le cifre hanno 256 stati.

Una volta che conosciamo la quantità di numeri negativi nella sequenza, non facciamo che aggiustare la Tabella degli Indici.
Sono sempre gli Indici a dettare le posizioni finali dei nostri numeri.
Dobbiamo fare sì che il 1° numero negativo sia memorizzato nel 1° elemento dell'array, mentre il 1° numero positivo deve essere spostato subito in coda all'ultimo numero negativo.
Non ci avete capito niente?

Osservate. Questa è una sequenza errata:

- `  1 (indice 0)`
- `  2 (indice 1)`
- `  3 (indice 2)`
- `-4 (indice 3)`
- `-3 (indice 4)`

Abbiamo un totale di 5 elementi.
Sappiamo che 2 sono negativi. Quindi gli altri 3 sono positivi.

Correggiamo la Tabella degli Indici:
Aggiungiamo 2 agli indici degli elementi positivi.
Sottraiamo 3 dagli indici degli elementi negativi.

- `  1 (indice 0 +2 = 2)`
- `  2 (indice 1 +2 = 3)`
- `  3 (indice 2 +2 = 4)`
- `-4 (indice 3 -3 = 0)`
- `-3 (indice 4 -3 = 1)`

Se riscriviamo il tutto, dando retta ai nuovi indici, otteniamo la sequenza corretta:

- `-4 (indice 0)`
- `-3 (indice 1)`
- `  1 (indice 2)`
- `  2 (indice 3)`

- 3 (indice 4)

La buona notizia è che non c'è bisogno di **correggere** la Tabella degli Indici.
Possiamo compilarla al volo con i valori **già** corretti (come vi mostrerò a breve).
Per quanto riguarda il calcolo di quanti valori negativi abbiamo, il costo è irrisorio.
Il tutto si traduce in un loop velocissimo che esegue 127 somme, e solo al 4° passaggio.
Nei primi 3 passaggi, il problema del segno non si pone.

7. Sorting a 32 bit - signed integer.
Ripropongo la funzione **Radix32u()**, con le dovute modifiche perchè riordini correttamente anche gli interi con segno.
Chiamerò la nuova funzione: **Radix32s()**.
Il fatto che il tipo di dati degli array sia ancora DWORD (anzichè long) non conta.
RadixSort se ne frega del tipo di dati. Lui guarda alla maschera dei bit.

```
// [desc]
//       RadixSort su interi con segno a 32 bit.
//       Utilizzabile ANCHE per interi senza segno.
//
// [in,out]
//         pSort : Array di elementi da riordinare.
// [in]
//     pTmp : Array temporaneo, e gemello di pSort.
//            La funzione lo utilizza come memoria di
//            scambio per depositare temporaneamente i
//            numeri riordinati tra un passaggio e l'altro.
// [in]
//     Count : Numero di elementi da riordinare.
//
void Radix32s (DWORD* pSort, DWORD* pTmp, DWORD Count)
{
    // Array statico grande abbastanza da contenere tutti gli
    // Istogrammi che occorrono, più una singola Tabella di
    // Indici.
    // 4 passaggi = 4 Istogrammi.
    // Ogni Istogramma richiede 1 KB di memoria (256 DWORD).
    // Anche la Tabella degli Indici richiede 1 KB.
    // Totale: 5 KB.
    // Tutto ruota intorno al numero 256 (cioe: base 256).
    static DWORD Array [256 * (4 + 1)];

    // Azzera il contenuto di tutti gli Istogrammi.
    // Gli Istogrammi sono contatori. Devono partire da 0.
    ::ZeroMemory (Array, 256 * sizeof (DWORD) * 4);

    // Prepara alcuni puntatori per indicizzare nell'Array.
    DWORD* pHist0 = &Array [0];    // Istogramma 1^ radice.
    DWORD* pHist1 = &Array [256];  // Istogramma 2^ radice.
    DWORD* pHist2 = &Array [512];  // Istogramma 3^ radice.
    DWORD* pHist3 = &Array [768];  // Istogramma 4^ radice.
    DWORD* pIndex = &Array [1024]; // Tabella degli Indici.

    // Compone tutti gli Istogrammi.
    for (DWORD i = 0; i < Count; ++i)
```

```
{
    DWORD Value = pSort [i];
    ++pHist0 [(Value >> 0 ) & 0xFF];
    ++pHist1 [(Value >> 8 ) & 0xFF];
    ++pHist2 [(Value >> 16) & 0xFF];
    ++pHist3 [(Value >> 24) & 0xFF];
}

// Punta al 1° Istogramma.
DWORD* pHistX = pHist0;

// Tante iterazioni quanti sono i passaggi.
for (DWORD Bit = 0; Bit < 32; Bit += 8)
{
    if (Bit != (3 * 8)) // Esegue per: Bit=0, Bit=8, Bit=16
    {
        // +-------------------------------------------------+
        // | I primi 3 passaggi sono indipendenti dal segno |
        // +-------------------------------------------------+

        // Compone la Tabella degli Indici.
        pIndex [0] = 0;

        for (DWORD i = 0; i < (256 - 1); ++i)
        {
            pIndex [i + 1] = pIndex [i] + pHistX [i];
        }
    }
    else // Esegue per: Bit=24
    {
        // +-------------------------------------------+
        // | L'ultimo passaggio deve considerare il segno |
        // +-------------------------------------------+

        // Conta il numero di valori negativi.
        DWORD NegCount = 0;

        for (DWORD i = 128; i < 256; ++i)
        {
            NegCount += pHistX [i];
        }

        // Compone la Tabella degli Indici.
        // Quello che prima era 1 ciclo da 256 iterazioni,
        // adesso è spezzato in 2 cicli da 128 iterazioni.

        // Indici dei numeri positivi (da 0 a 127).
        pIndex [0] = NegCount;

        for (DWORD i = 0; i < 127; ++i)
        {
            pIndex [i + 1] = pIndex [i] + pHistX [i];
        }

        // Indici dei numeri negativi (da 128 a 255).
        //
        // Se NegCount != 0 (abbiamo valori negativi)
        //   pIndex [128] assume il valore 0.
```

```
            // Se NegCount = 0 (non abbiamo valori negativi)
            //   pIndex [128] assume il valore standard.
            //
            pIndex [128] = (NegCount)
                           ? 0
                           : pIndex [127] + pHistX [127];

            for (DWORD i = 128; i < (256 - 1); ++i)
            {
                pIndex [i + 1] = pIndex [i] + pHistX [i];
            }
        }

        // Riordina gli elementi.
        for (DWORD i = 0; i < Count; ++i)
        {
            DWORD Value = pSort [i];
            pTmp [pIndex [(Value >> Bit) & 0xFF]++] = Value;
        }

        // Scambia i puntatori degli array.
        DWORD* pSwap = pSort;
               pSort = pTmp;
               pTmp  = pSwap;

        // Punta all'Istogramma successivo.
        pHistX += 256;
    }
}
```

8. Sì maa... ci sarebbe un piccolo dettaglio...
C'è un'ultima cosa che è bene spiegarvi prima che possiate camminare da soli.
I due RadixSort che vi ho proposto lavorano su interi da 32 bit.
Ma come ho detto prima, RadixSort non guarda al tipo dei vostri dati.

Avete già provato a dargli in pasto un array contenente valori float?
Per favore, fatelo. Assemblate un piccolo array da 10 float.
Usate solo float positivi e invocate entrambe le funzioni **Radix32u()** e **Radix32s()**.
Il vostro array viene riordinato correttamente da entrambe le funzioni.

Ora fate un altro test.
Sempre 10 elementi, ma stavolta usate un mix di 5 float positivi e 5 negativi.
Sorpresa! Il vostro array **non** viene riordinato correttamente ne dall'una ne dall'altra funzione.

8.1 Allora cosa succede?
Niente panico!
Se pensate che il problema sia ancora legato al segno dei numeri: avete visto giusto.
Solo che questa volta la soluzione è un filino più complessa.

Osservate la sequenza di valori float:
(tra parentesi avete le traduzioni in esadecimale)

- `0.5f (0x3F000000)`

- 1.0f (0x3F800000)
- 2.0f (0x40000000)
- 2.5f (0x40200000)
- -0.5f (0xBF000000)
- -3.5f (0xC0600000)
- -3.6f (0xC0666666)

Come potete vedere, le maschere di bit sono effettivamente messe in ordine crescente (dal più piccolo al più grande). Solo che la codifica dei float gioca un brutto scherzo.
Non solo ci ripropone il problema del segno (proprio come per gli interi con segno), ma fa anche sì che i numeri negativi siano ordinati al contrario (dal più grande al più piccolo).
(Parentesi...
Il tipo di dati **float** è codificato secondo i dettami dello standard internazionale registrato con il nome di: **IEEE 754 Single Precision Floating Point**.
La spiegazione del funzionamento di tale codifica esula dallo scopo del presente articolo.
Se volete documentarvi, una semplice ricerca con google produrrà tutto il necessario.
... chiusa parentesi)

Attenti a non confondervi: **-0.5f** è maggiore di **-3.5f**. E' giusto.
Se guardate alle traduzioni esadecimali, **-0.5f** appare minore di **-3.5f**. E' sbagliato.

Ecco spiegato perchè nemmeno la **Radix32s()** riesce a ordinarli correttamente.
Non solo i valori negativi sono da visualizzare prima dei valori positivi, ma la loro sequenza deve anche essere invertita.
Come nel caso della **Radix32s()** anche questo problema è risolvibile mettendo mano alla Tabella degli Indici.

8.2 E il bit del segno?
Nella codifica floating point il bit del segno risiede nell'ultimo Byte, proprio come accade per gli interi.
Pertanto correggiamo solo la Tabella degli Indici dell'ultimo passaggio.

8.3 Quindi la soluzione è...?
... spezzata in 2 parti. Haha!
Vedo di spiegarmi.
Oltre a modificare la Tabella degli Indici, è necessario poi tenere conto che gli indici dei float negativi sono stati inseriti nella Tabella al rovescio, cioè partendo dal fondo e risalendo verso la cima.

Quando poi rileggeremo la tabella al dritto, allora quegli indici inseriti al rovescio (cioè quelli per i float negativi) risulteranno andare in ordine decrescente.
Invece gli indici relativi ai float positivi risulteranno andare in ordine crescente.
Dovremo tenere conto di questo dettaglio, o saranno guai.

Quindi se in fase di riordinamento finora avevamo sempre post-incrementato i nostri indici dopo averli usati, adesso dovremo fare l'esatto opposto, e pre-decrementarli prima di usarli.

Non ci avete capito niente?
Mi riferisco al loop che ha luogo subito dopo la composizione della Tabella degli Indici:

```
    // Riordina gli elementi.
    for (DWORD i = 0; i < Count; ++i)
    {
        DWORD Value = pSort [i];
```

```
        pTmp [pIndex [(Value >> Bit) & 0xFF]++] = Value;
    }
```

Lo vedete il post-incremento?
Nelle funzioni **Radix32u()** e **Radix32s()** avevamo sempre post-incrementato, perchè gli indici nelle Tabelle erano sempre stati inseriti in ordine crescente.

Adesso che avremo una parte degli indici inseriti in ordine decrescente, dovremo effettuare il pre-decremento prima di usare tali indici. Cioè:

```
    pTmp [--pIndex [(Value >> Bit) & 0xFF]] = Value;
```

Ma dovremo pre-decrementare solo con i float negativi.
Con qualunque altro valore dobbiamo post-incrementare, come abbiamo sempre fatto.

9. Sorting a 32 bit - float negativi.
Ecco a voi la terza e ultima variante di RadixSort: quella capace di riordinare anche i float negativi. Per via delle modifiche apportate, questa funzione è saggio utilizzarla **solo** in presenza di almeno un float negativo.

Se nella vostra sequenza appaiono solo float positivi, potete riordinarli in meno tempo tramite la funzione **Radix32u()**. Ma ci eravate arrivati da soli, vero?

```
// [desc]
//       RadixSort per sequenze contenenti (anche) float negativi.
//       Utilizzabile ANCHE per sequenze di soli float positivi.
//
// [in,out]
//         pSort : Array di elementi da riordinare.
// [in]
//     pTmp : Array temporaneo, e gemello di pSort.
//            La funzione lo utilizza come memoria di
//            scambio per depositare temporaneamente i
//            numeri riordinati tra un passaggio e l'altro.
// [in]
//     Count : Numero di elementi da riordinare.
//
void Radix32f (DWORD* pSort, DWORD* pTmp, DWORD Count)
{
    // Array statico grande abbastanza da contenere tutti gli
    // Istogrammi che occorrono, più una singola Tabella di
    // Indici.
    // 4 passaggi = 4 Istogrammi.
    // Ogni Istogramma richiede 1 KB di memoria (256 DWORD).
    // Anche la Tabella degli Indici richiede 1 KB.
    // Totale: 5 KB.
    // Tutto ruota intorno al numero 256 (cioe: base 256).
    static DWORD Array [256 * (4 + 1)];

    // Azzera il contenuto di tutti gli Istogrammi.
    // Gli Istogrammi sono contatori. Devono partire da 0.
    ::ZeroMemory (Array, 256 * sizeof (DWORD) * 4);

    // Prepara alcuni puntatori per indicizzare nell'Array.
```

```
DWORD* pHist0 = &Array [0];    // Istogramma 1^ radice.
DWORD* pHist1 = &Array [256];  // Istogramma 2^ radice.
DWORD* pHist2 = &Array [512];  // Istogramma 3^ radice.
DWORD* pHist3 = &Array [768];  // Istogramma 4^ radice.
DWORD* pIndex = &Array [1024]; // Tabella degli Indici.

// Compone tutti gli Istogrammi.
for (DWORD i = 0; i < Count; ++i)
{
    DWORD Value = pSort [i];
    ++pHist0 [(Value >> 0 ) & 0xFF];
    ++pHist1 [(Value >> 8 ) & 0xFF];
    ++pHist2 [(Value >> 16) & 0xFF];
    ++pHist3 [(Value >> 24) & 0xFF];
}

// Punta al 1° Istogramma.
DWORD* pHistX = pHist0;

// Tante iterazioni quanti sono i passaggi.
for (DWORD Bit = 0; Bit < 32; Bit += 8)
{
    if (Bit != (3 * 8)) // Esegue per: Bit=0, Bit=8, Bit=16
    {
        // +-------------------------------------------------+
        // | I primi 3 passaggi sono indipendenti dal segno |
        // +-------------------------------------------------+

        // Compone la Tabella degli Indici.
        pIndex [0] = 0;

        for (DWORD i = 0; i < (256 - 1); ++i)
        {
            pIndex [i + 1] = pIndex [i] + pHistX [i];
        }

        // Riordina gli elementi.
        for (DWORD i = 0; i < Count; ++i)
        {
            DWORD Value = pSort [i];
            pTmp [pIndex [(Value >> Bit) & 0xFF]++] = Value;
        }

        // Punta all'Istogramma successivo.
        pHistX += 256;
    }
    else // Esegue per: Bit=24
    {
        // +-----------------------------------------------+
        // | L'ultimo passaggio deve considerare il segno |
        // +-----------------------------------------------+

        // Conta il numero di valori negativi.
        DWORD NegCount = 0;

        for (DWORD i = 128; i < 256; ++i)
        {
            NegCount += pHistX [i];
```

```
            }

            // Compone la Tabella degli Indici.
            // Quello che prima era 1 ciclo da 256 iterazioni,
            // adesso è spezzato in 2 cicli da 128 iterazioni.

            // Indici dei numeri positivi (da 0 a 127).
            pIndex [0] = NegCount;

            for (DWORD i = 0; i < 127; ++i)
            {
                pIndex [i + 1] = pIndex [i] + pHistX [i];
            }

            // Indici dei numeri negativi (da 128 a 255).
            // Cioè no. Da 255 a 128. Li memorizziamo al contrario!
            pIndex [255] = pHistX [255];

            for (DWORD i = 254; i > 127; --i)
            {
                pIndex [i] = pIndex [i + 1] + pHistX [i];
            }

            // Riordina gli elementi.
            for (DWORD i = 0; i < Count; ++i)
            {
                DWORD Value = pSort [i];

                // Il bit del segno è impostato?
                //   sì : float negativo.
                //   no : float positivo.
                if (Value & (1 << 31))
                {
                    // --pIndex[] = Pre-decrementa l'indice.
                    pTmp [--pIndex [(Value >> Bit) & 0xFF]] = Value;
                }
                else
                {
                    // pIndex[]++ = Post-incrementa l'indice.
                    pTmp [pIndex [(Value >> Bit) & 0xFF]++] = Value;
                }
            }
        }

        // Scambia i puntatori degli array.
        DWORD* pSwap = pSort;
               pSort = pTmp;
               pTmp  = pSwap;
    }
}
```

10. Conclusione.

In questo articolo avete fatto la conoscenza dell'algoritmo di sorting più veloce che esista. Avete imparato il semplice meccanismo di sorting da esso utilizzato, e siete poi passati alla comprensione della difficile logica dell'algoritmo in versione informatica.

Vi siete scontrati con i problemi che ha l'algoritmo in presenza dei 3 principali formati numerici. Avete imparato a riconoscere questi problemi e li avete eliminati, ottenendo così le 3 varianti dell'algoritmo:

Radix32u(), per interi senza segno, e per float solo positivi.
Radix32s(), per interi con e senza segno, e per float solo positivi.
Radix32f(), per interi senza segno, e per float positivi e/o negativi.

A questo punto potete affermare con assoluta certezza di conoscere vita, morte e miracoli di RadixSort. Avete visto che è in grado di riordinare velocemente qualunque sequenza di numeri in qualunque formato.
Quanto più lunga è la sequenza di numeri, tanto maggiore è il guadagno in velocità nell'utilizzare RadixSort al posto di altri algoritmi di sorting.

Nelle implementazioni qui proposte abbiamo trattato esclusivamente il sorting di elementi da 32 bit. Ma RadixSort è facilmente adattabile ad elementi di lunghezza arbitraria.
E' sufficiente modificare il numero di passaggi.

Tutto ciò che la vostra immaginazione è in grado di ricondurre ad un numero, RadixSort lo può riordinare. L'algoritmo trova applicazione in diversi campi dell'informatica.
Non ultimo: la Computer Graphics.

Ma non è compito mio darvi idee su come potete impiegarlo :)
Io mi fermo qui. Spero che la lettura sia stata di vostro gradimento.

-Kelesis

[eof]