Insertion Sort e Radix Sort

Lorenzo Pesci

6 Luglio 2020

1 Introduzione

Vogliamo analizzare le differenze fra gli algoritmi di Insertion Sort e Radix Sort per ordinare un numero crescente di valori.
Entrambi gli algoritmi sono algoritmi di ordinamento, che ricevono in input una sequenza di numeri casuali e restituiscono in output una sequenza di numeri ordinati.

1.1 Insertion Sort

Insertion Sort é l'algoritmo piú semplice per il risolvere il problema dell'ordinamento, e risulta efficiente per input di piccole dimensioni.
Riceve in input un array A, i cui numeri sono ordinati sul posto: ossia in ogni istante ci sono al piú un numero costante di valori memorizzati all'esterno di A.

Pseudocodice:

```
INSERTION-SORT (A)
1 for j = 2 to length[A]
2       key = A[j]
3       i = j - 1
4       while i > 0 and A[i] > key
5               A[i + 1] = A[i]
6               i = i - 1
7       A[i + 1] = key
```

Nel nostro esperimento, lo pseudocodice proposto viene tradotto in linguaggio Python nel file insertionsort.py

1.2 Radix Sort

Radix Sort é un algoritmo che riceve in input un array A in cui ogni valore é composto di d cifre. Inizialmente ordina A usando la cifra meno significativa di ogni numero come chiave di ordinamento, poi ripete l'ordinamento utilizzando come chiave la seconda cifra significativa e cosí via, finché non saranno state considerate tutte le d cifre.
Radix Sort si avvale di un algoritmo di ordinamento ausiliario, e fondamentale per la sua correttezza é che questo algoritmo ausiliario sia stabile, quindi Counting Sort é un ottimo candidato.

Pseudocodice:

```
RADIX-SORT(A,d)
1 for i = 1 to d
2       usa un ordinamento stabile per ordinare A usando come chiave la cifra i
```

Nello pseudocodice di Radix Sort si suppone che la cifra 1 sia la meno significativa e la cifra d la piú significativa.
Nel nostro esperimento, lo pseudocodice proposto viene tradotto in linguaggio Python nel file radixsort.py

2 Descrizione esperimenti e documentazione del codice

Gli esperimenti sono stati svolti su un MacBook Pro con sistema operativo macOS Catalina, processore 2,2 GHz 6-Core Intel Core i7 e 16 GB di RAM.

Eventuali numeri casuali sono stati generati dalla funzione *random.randint()* importata dalla libreria *random*.

Il codice è articolato in quattro file: *insertionsort.py*, *radixsort.py*, *plot.py*, *main.py*.

- Nel file **insertionsort.py** é realizzato in linguaggio Python l'algoritmo di Insertion Sort che verrá utilizzato per i nostri esperimenti.

- Nel file **radixsort.py** é realizzato in linguaggio Python l'algoritmo di Radix Sort che verrá utilizzato per i nostri esperimenti.

- Nel file **plot.py** sono realizzati tutti i grafici sugli esperimenti condotti sia su Insertion Sort che su Radix Sort.

- Nel file **main.py** é realizzato il codice che esegue i test su entrambi gli algoritmi. In particolare sono stati eseguiti dei test diversi per entrambi gli algoritmi riguardanti in problema dell'ordinamento.

3 Presentazione dati sperimentali

3.1 Insertion Sort

Sull'algoritmo di Insertion Sort sono stati eseguiti i test su un numero crescente di elementi, partendo da array di cento elementi fino a raggiungere un array con centomila elementi, analizzando il tempo di esecuzione dell'algoritmo.
In particolare sono stati realizzati tre esperimenti considerando il caso migliore, il caso peggiore e il caso medio di Insertion Sort, ovvero rispettivamente ordinamento su array ordinato, array ordinato al contrario e array generato randomicamente.

Di seguito é riportata la **tabella** del tempo di esecuzione di Insertion Sort sui tre casi.

Numero di Elementi	**100**	**1000**	**10000**	**100000**
Array Random	0.00027	0.02481	2.01623	199.041
Array Ordinato	1.5020e-05	0.00010	0.00103	0.01214
Array Ordinato Al Contrario	0.00040	0.03872	3.82862	409.713

Tabella 1: Tabella tempo di esecuzione di Insertion Sort

Come si evince dai risultati ottenuti l'algoritmo ed i test implementai rispettano la teoria, ovvero il caso migliore si ha con array ordinato, il caso medio si ha con array generato randomicamente ed infine il caso peggiore si ha con array ordinato al contrario.

Di seguito sono riportati i **grafici** di Insertion Sort per ognuno dei tre casi analizzati.

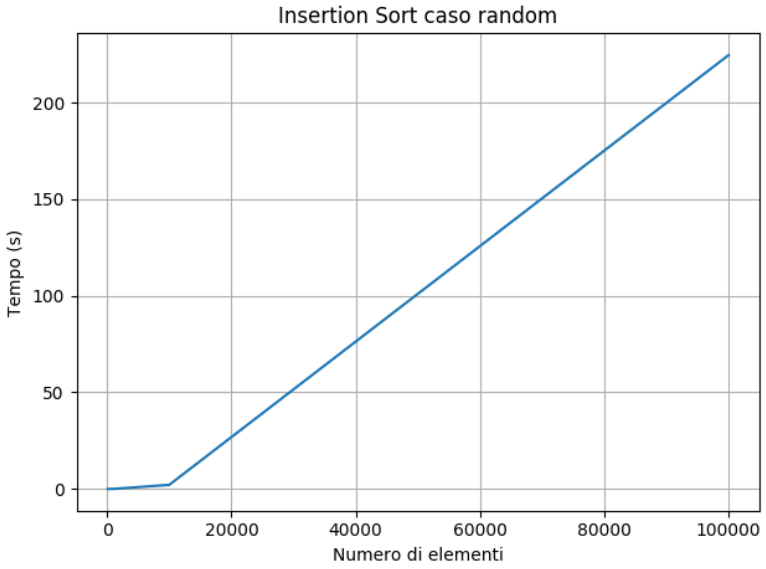

Figura 1: Tempo di esecuzione di Insertion Sort su array random.

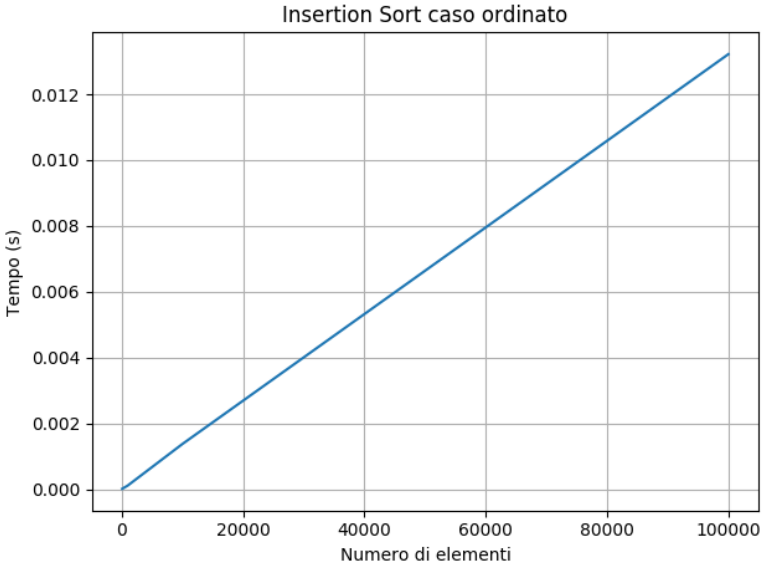

Figura 2: Tempo di esecuzione di Insertion Sort su array ordinato.

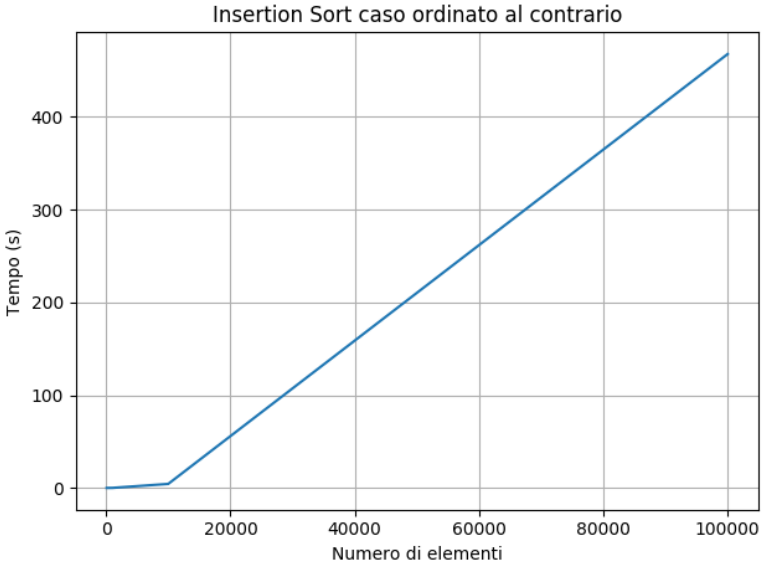

Figura 3: Tempo di esecuzione di Insertion Sort su array ordinato al contrario.

3.2 Radix Sort

Per l'algoritmo di Radix Sort sono stati condotti due tipi di esperimenti.

3.2.1 Primo esperimento

Il primo esperimento é stato condotto variando la base con cui viene rappresentato un numero. In particolare sono state utilzzate la base 2, la base 5 e la base 10, ovvero la base decimale con la quale abbiamo a che fare ogni giorno nella vita quotidiana. Oltre alla base del numero, l'esperimento tiene conto di un numero crescente di elementi, ovvero siamo partiti da cento elementi in base 2, base 5 e base 10 fino ad arrivare a centomila elementi in base 2, base 5 e base 10.
Di seguito é riportata la **tabella** riassuntiva dei risultati ed il **grafico** di confronto dei risultati.

Numero di Elementi	**100**	**1000**	**10000**	**100000**
Base 2	0.00135	0.01310	0.14778	1.46151
Base 5	0.00042	0.00403	0.04126	0.39812
Base 10	0.00029	0.00323	0.03747	0.33712

Tabella 2: Tabella tempo di esecuzione di Radix Sort

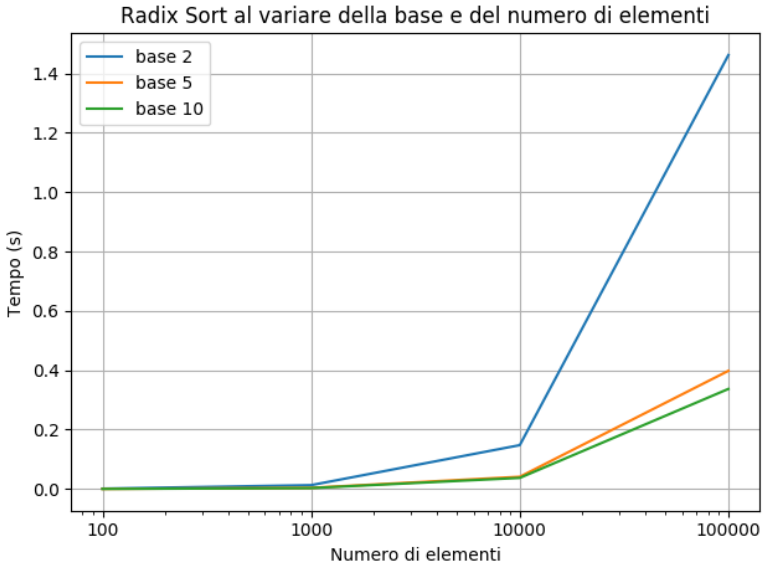

Figura 4: Tempo di esecuzione di Radix Sort in base 2, base 5 e base 10.

Come ci aspettavamo dai dati teorici, Radix Sort risulta piú efficente sui numeri in base 10, poiché ordina numeri su piú cifre significative ma su un numero inferiore di bit.

3.2.2 Secondo esperimento

Per la seconda parte dell'esperimento abbiamo quindi utlizzato la base 10 come notazione per rappresentare i nostri numeri. In questo esperimento abbiamo utlizzato array che partivano da cento elementi fino ad arrivare ad un milione di elementi variando il numero massimo da ordinare.
Di seguito é riportata la **tabella** riassuntiva dei risultati ed il **grafico** di confronto dei risultati.

Numero di Elementi	100	1000	10000	100000
Da 1 a 100	0.00030	0.00370	0.03335	0.33719
Da 1 a 1000	0.00038	0.00451	0.06099	0.42153
Da 1 a 10000	0.00045	0.00486	0.07889	0.52280
Da 1 a 100000	0.00053	0.00535	0.06909	0.63612
Da 1 a 1000000	0.00061	0.00671	0.08480	0.68152

Tabella 3: Tabella tempo di esecuzione di Radix Sort al variare del numero massimo da ordinare e del numero di elementi.

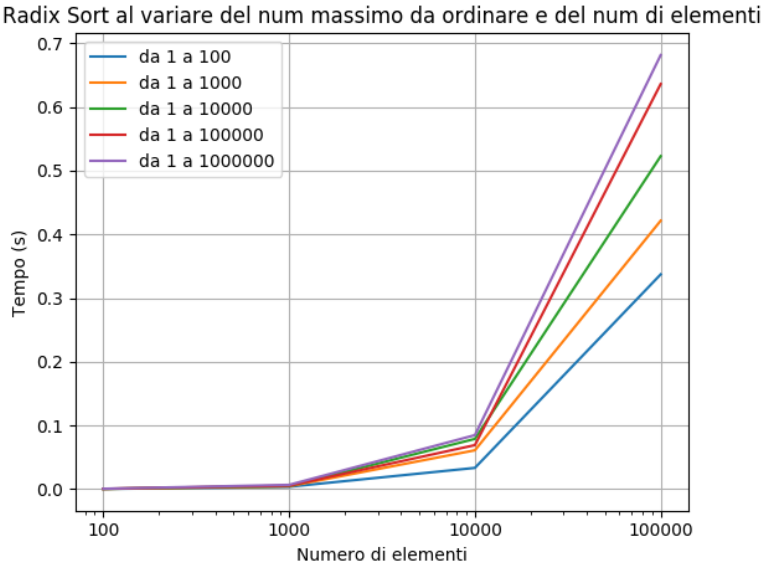

Figura 5: Radix Sort al variare del numero massimo da ordinare e del numero di elementi.

Infine, come nel caso di Insertion Sort, sono stati condotti gli esperimenti inserendo in input la stessa sequenza di array nel caso in cui l'array fosse ordinato o ordinato al contrario.

Numero di Elementi	100	1000	10000	100000
Array Ordinato	8.296e-05	0.00105	0.01455	0.21112
Array Ordinato Al Contrario	5.960e-06	2.002e-05	0.00018	0.00181

Tabella 4: Tabella tempo di esecuzione di Radix Sort su array ordinato e array ordinato al contrario

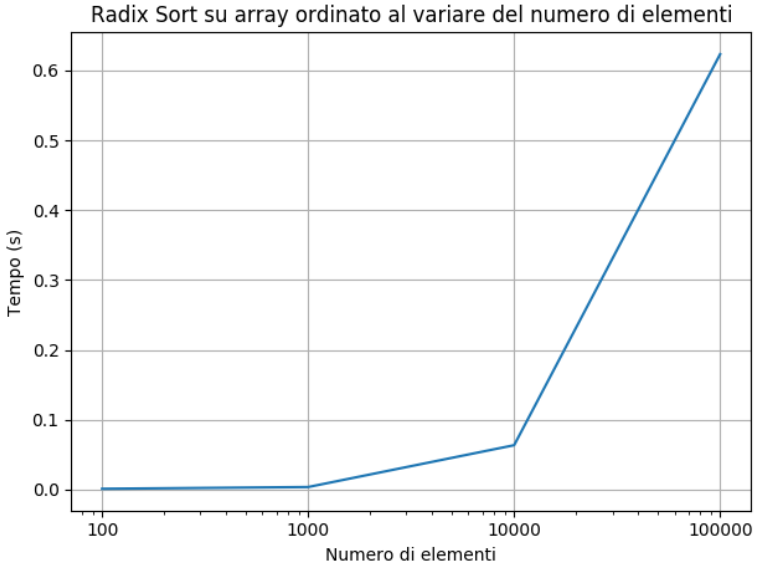

Figura 6: Tempo di esecuzione di Radix Sort su array ordinato.

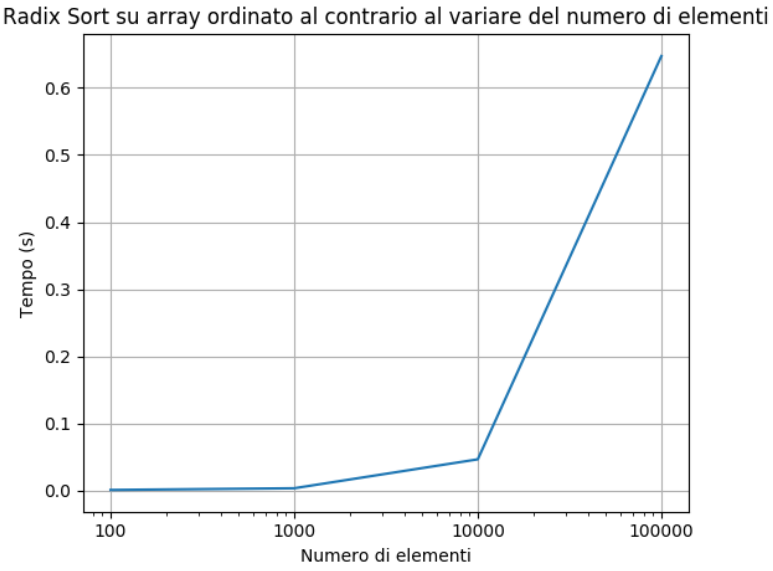

Figura 7: Tempo di esecuzione di Radix Sort su array ordinato al contrario

4 Analisi dei risultati ottenuti

Dai nostri test possiamo vedere come il caso di Array Ordinato con Insertion Sort è quello che impiega il minor tempo di esecuzione. Tale risultato si poteva intuire anche dalla teoria, avendo un costo computazionale pari a $\theta(n)$.
Negli altri casi invece Radix Sort è generalmente più veloce e tale rapidità rimane costante in tutti i casi, questo è dovuto alla sua complessità totale pari a $\theta(n)$.
Una grande diversità tra i tempi di esecuzione si può notare nel caso di Array Random e Array Ordinato al Contrario con 100000 elementi, dove il Radix Sort esegue l'ordinamento in poco più di un decimo di secondo, invece l'Insertion Sort impiega qualche minuto per portare a termine l'operazione.
Quindi generalmente, avendo un tempo totale di esecuzione $\theta(n)$, il Radix Sort è un algoritmo che impiega con i nostri dati sempre meno di un secondo per completare l'ordinamento.
Con tali esperimenti possiamo dunque comprendere quanto sia grande la differenza tra avere un costo computazionale pari a $\theta(n)$ o $\theta(n^2)$.

5 Conclusioni

Anche se Radix Sort e Counting Sort vengono eseguiti in un tempo asintoticamente lineare, algoritmi di ordinamento sul posto vengono comunque preferiti a questi due algoritmi.
Infatti quest'ultimi eseguono l'ordinamento sul posto senza utilizzare un array ausiliario o produrre un nuovo array per il risultato. Infine i fattori costanti nascosti nella notazione $\theta(n)$ possono produrre un tempo di esecuzione superiore rispetto a un ordinamento per confronti.

TEMPO DI ESECUZIONE	**INSERTION SORT**	**RADIX SORT**
Array Random	$\theta(n^2)$	$\theta(n)$
Array Ordinato	$\theta(n)$	$\theta(n)$
Array Ordinato Al Contrario	$\theta(n^2)$	$\theta(n)$

Tabella 5: Tabella dei Costi