Prima prova pratica in itinere

Ingegneria degli Algoritmi 2018/2019

Progetto : 2 Tutor di riferimento : Luca Pepè Sciarria

Studente : Simone Festa, matricola 0251679

IMPLEMENTAZIONE DEL SAMPLE MEDIAN SELECT

La realizzazione dell'algoritmo **SampleMedianSelect** è basata sulla select deterministica. In particolare, per la creazione del *sottoinsieme V* di *m* elementi, viene partizionata la lista di elementi in *m* gruppi, prelevando in modo random un indice da ciascun gruppo ed aggiungendo l'elemento di indice " *i* " presente nella lista nel sottoinsieme V.

Ottenuto un sottoinsieme V di m elementi, procedo con il calcolo del mediano, utilizzandolo come pivot. La partizione verrà eseguita con riferimento a tale pivot.

Le restanti linee di codice sono fedeli alla select deterministica, come l'implementazione del TrivialSelect per liste di piccole dimensioni o le partizioni rispetto ad un determinato perno.

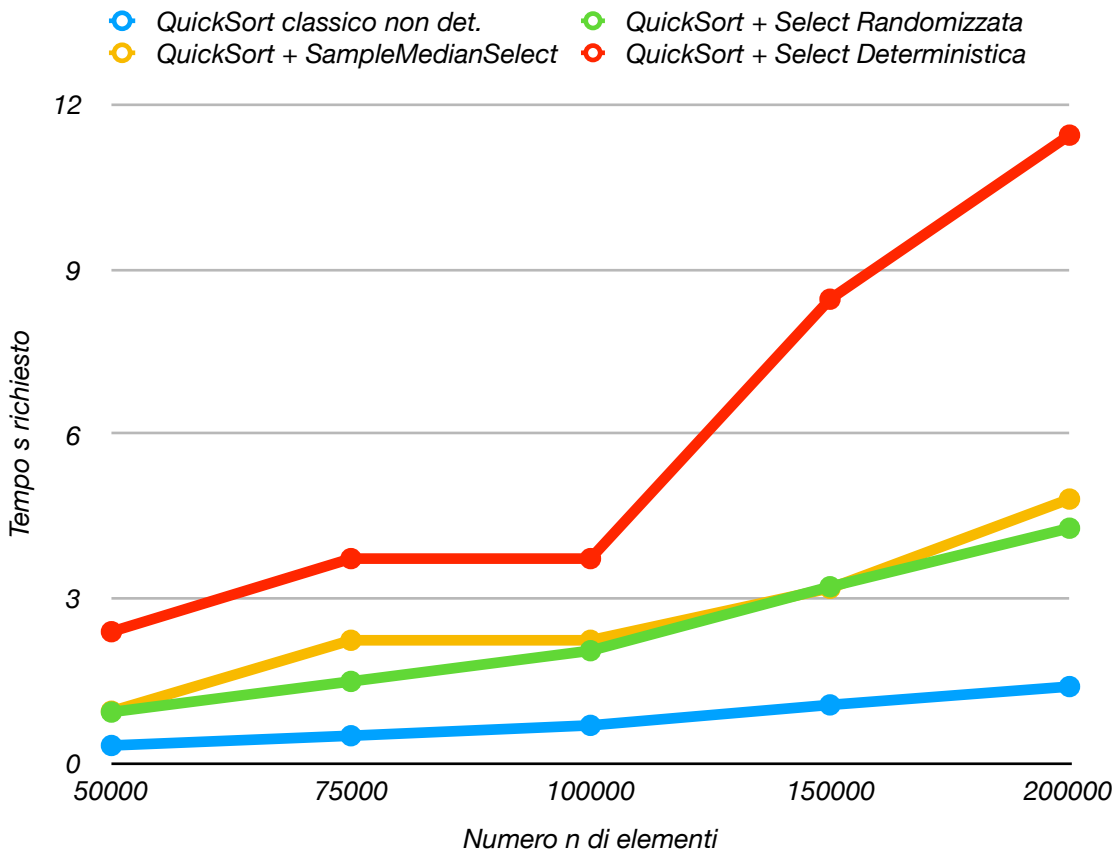
SCELTA DEL PARAMETRO 'M'

Per la scelta del parametro "*m*", cioè del numero di elementi del sottoinsieme V, oltre ad aver eseguito varie prove con l'obiettivo di poter confrontare il comportamento del programma al variare di tale parametro, viene eseguito più volte lo stesso test per limitare la presenza di eventi casuali atti a migliorare (o peggiorare) la tempistica. Di seguito è allegata una tabella mostrante i vari tempi di esecuzione per alcuni valori "*m*" utilizzati nel QuickSort con SampleMedianSelect. I confronti vengono anche effettuati su dimensioni di liste diverse.

	1° test (in secondi)	2° test (in secondi)	3° test (in secondi)	Media aritmetica
n = 100000 ; m = 1	2,676010847091675	1,8974528312683105	1,9490768909454346	*2,17418018976847*
n = 100000 ; m = 5	2,0427310466766357	2,1158721446990967	2,025563955307007	*2,06138904889425*
n = 100000 ; m = 9	2,3147921562194824	2,405780792236328	2,4166431427001953	*2,37907203038534*
n = 200000 ; m = 1	4,249889135360718	3,9460840225219727	4,031553030014038	*4,07584206263224*
n = 200000 ; m = 5	4,341506004333496	4,226111173629761	4,366738319396973	*4,31145183245341*
n = 200000 ; m = 9	4,768217086791992	4,874582052230835	4,7082178592681885	*4,78367233276367*

Come possiamo notare, al variare di *m* non abbiamo sempre una netta differenza tra i vari test, ma possiamo dire che la scelta di un *m* piccolo sembri comportare un minore impiego di tempo. Utilizzeremo allora **m = 5** (poiché con *m* = 1 si prenderebbe come sottoinsieme V tutta la lista, mentre *m* = 9 non risulta essere ottimale) consci del fatto che potrebbe non esistere un *m* univoco per ogni lista, ma che esso potrebbe cambiare con quest'ultima.

CONFRONTO TRA QUICKSORT

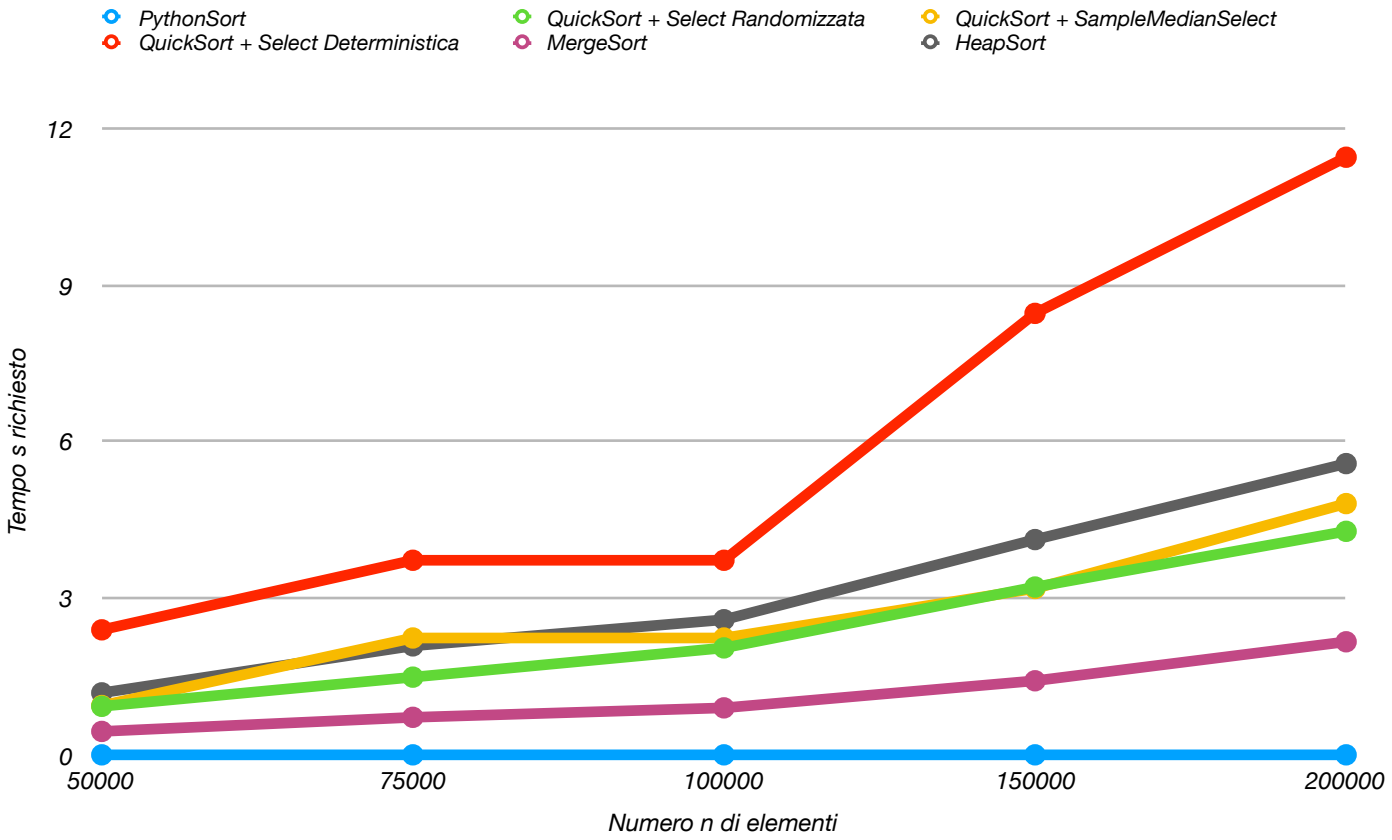


Elementi → Algoritmo ↓	50000	75000	100000	150000	200000
QuickSort classico	0,320067167282	0,497707843780	0,68663763999	1,05960202217	1,39344573020
QuickSort + Select Randomizzata	0,928756952285	1,48746800422	2,04729795455	3,21291208267	4,27950215339
QuickSort + SMedianSelect	0,944205045700	2,23725414276	2,23725414276	3,18437004089	4,81013679504
QuickSort + Select Deterministica	2,39330315589	3,72527718544	3,72527718544	8,45216298103	11,441623210

NB : Il tempo, nella tabella e nel grafico, è espresso in secondi.

Notiamo come il QuickSort classico richieda meno tempo di tutti, mentre il più lento risulta essere il QuickSort con Select Deterministica. Osserviamo che, nella maggior parte dei casi, un algoritmo QuickSort che opera con scelte random impieghi meno tempo rispetto ad una scelta deterministica. Ovviamente potrebbero essere sempre presenti dei casi limiti, ma tale eventualità viene ridotta dalla randomicità stessa.

CONFRONTO CON GLI ALTRI ALGORITMI



Elementi → Algoritmo ↓	50000	75000	100000	150000	200000
PythonSort	0	0,00000019209289	0,00000009536743	0	0
QuickSort + Select Randomizzata	0,928756952285767	1,48746800422668	2,04729795455933	3,21291208267212	4,27950215339661
QuickSort + SMedianSelect	0,944205045700073	2,23725414276123	2,23725414276123	3,18437004089355	4,81013679504395
QuickSort + Select Deterministica	2,39330315589905	3,72527718544006	3,72527718544006	8,45216298103333	11,441623210907
MergeSort	0,448452949523926	0,718372106552124	0,899311065673828	1,41792893409729	2,16446805000305
HeapSort	1,18669009208679	2,08612298965454	2,58701586723328	4,12123870849609	5,57591915130615

NB : Il tempo, nella tabella e nel grafico, è espresso in secondi.

Come si evince dal grafico con annessa tabella, non sono presenti i seguenti algoritmi: BubbleSort, InsertionSort e SelectionSort. Il motivo è dato dall'elevato costo computazionale e dalla richiesta eccessiva di tempo per la produzione di un risultato:
Nel nostro caso base (50000 elementi), tali algoritmi hanno prodotto i seguenti risultati:
SelectionSort tempo richiesto : 149,12068128585815 secondi;
InsertionSort tempo richiesto : 152,7231481075287 secondi;
Il *BubbleSort* non è riuscito a produrre risultati in questo lasso temporale (e superiore).

COMMENTO FINALE

La prima osservazione ha come soggetto il metodo Sort() di Python, un algoritmo non analizzato interamente nel corso di " Ingegneria degli Algoritmi" ma che si è messo in mostra per la sua velocità. Risulta difficile ottenere dei dati sul tempo di esecuzione, poiché per liste piccole il tempo richiesto è " 0 ", mentre per liste più grandi vi sono difficoltà nelle creazioni delle liste stesse. Discutendo dell'operato dei vari QuickSort, notiamo come un approccio più deterministico generi un'attesa maggiore. In particolare, la scelta di un valore *m* piccolo rispetto alla dimensione della lista nel SampleMedianSelect sembra non influenzare nettamente il tempo di esecuzione.

Allegato a questa relazione è presente il codice sorgente contenente le varie implementazioni dell'algoritmo QuickSort con SampleMedianSelect, Select deterministica e Select randomizzata. In tale codice non sono presenti altri algoritmi di ordinamento usati per i confronti, poiché sarebbe una riproposizione di linee di codice già note. Sono invece presenti tutte le funzioni, strutture ed algoritmi che hanno richiesto modifiche e controlli. Per i vari confronti ho semplicemente aggiunto le nuove implementazioni dell'algoritmo QuickSort al file "Sorting.py" contenente i precedenti algoritmi di ordinamento, producendo in questo modo i risultati qui riportati. Il metodo Sort di Python è stato semplicemente richiamato, poiché incluso come algoritmo di ordinamento standard in Python.