A.A.
2018
2019

PRIMA PROVA PRATICA IN ITINERE INGEGNERIA DEGLI ALGORITMI

PROGETTO 2

MONTENEGRO ALESSANDRO 0252225
CALAVARO MARCO 0252844

INDICE

NOTA

I test sono stati svolti su un DELL XPS 15 9560 con sistema operativo Ubuntu Mate 18.04.1 LTS e processore Intel Core i7 7700HQ 2,80 GHz.

Nei grafici mostrati in questo elaborato, i valori presenti sull'asse delle ascisse e delle ordinate sono rispettivamente le dimensioni degli array considerati, chiaramente in termini di numero di elementi, e il tempo di esecuzione in secondi.

Per comodità alleghiamo i tabulati dei test delle funzioni in un file .xlsx a parte, in quanto abbiamo realizzato delle tabelle molto grandi. In questo pdf alleghiamo solamente i grafici più significativi.

1. PROGETTO

Il progetto richiedeva di modificare l'algoritmo di ordinamento ***quickSort*** in modo tale che non scegliesse il pivot su cui partizionare in maniera casuale, bensì in maniera deterministica, avvalendosi di un nuovo algoritmo di selezione, ***sampleMedianSelect***. Quest'ultimo si basa sull'algoritmo ***select*** di Floyd e Rivest, modificandone però il modo in cui effettua la partizione, infatti, generando un sottoinsieme V di grandezza **m** in modo casuale dall'array di partenza, trovandone il mediano, il nuovo algoritmo di selezione effettua la partizione intorno ad esso. Per la ricerca dell'elemento desiderato, ovvero quello in posizione k, ci si sposta ricorsivamente sulla partizione contenente tale posizione. Alla fine del ciclo di esecuzione del ***sampleMedianSelect***, verrà restituito l'elemento in posizione k, che abbiamo scelto essere il mediano della lista, impiegato poi nel ***quickSort*** come pivot. Una volta implementato tale algoritmo di ordinamento, che chiameremo ***Modify-_Quick_Sort,*** lo confronteremo con gli altri algoritmi di ordinamento.

2. SCELTE IMPLEMENTATIVE

Segue la descrizione dettagliata di ogni funzione impiegata nel codice.

Modify_Quick_Sort: per l'implementazione dell'algoritmo ***Modify_Quick_Sort*** abbiamo scelto come pivot da considerare per la partizione, il mediano dell'array da ordinare, in quanto in tal modo l'algoritmo ricorre su due sotto array di uguale lunghezza diminuendo il tempo di esecuzione totale, infatti si fa in modo che si verifichi sempre il caso migliore del ***quickSort*** classico.

randomChoice: per la creazione del sotto array randomico V di dimensione m, abbiamo implementato la funzione ***randomChoice*** che, data una lista, una dimensione, e due indici (che rappresentano l'indice destro e sinistro) genera una nuova lista, di dimensione pari a quella passata come parametro, di elementi casuali della lista in ingresso, considerata solamente tra i due indici. Si è fatto uso della funzione della libreria ***random***, ***sample*** (list, k) che restituisce una lista di lunghezza k di elementi distinti, scelti dalla sequenza di input. La creazione della funzione ***randomChoice*** è stata necessaria in quanto così si riesce a lavorare solo sulla porzione di lista interessata e l'utilizzo solo di ***sample*** rendeva l'algoritmo instabile.

partitionDet: la funzione ***partitionDet*** è stata presa dal codice visto a lezione, più specificatamente da **Selection.py**. Viene impiegata nel nostro codice sia per partizionare l'array in ***sampleMedianSelect*** rispetto al mediano dell'insieme randomico V, sia in ***Modify_Quick_Sort*** quando si partiziona rispetto al mediano dell'intero array.

sampleMedianSelect: la procedura ***sampleMedianSelect*** è la modifica dell'algoritmo ***select*** secondo i canoni indicati nella traccia del progetto. In primis si fa un controllo sull'indice dell'elemento da selezionare affinché esso si trovi tra 0 e la lunghezza della lista**.** La scelta del parametro **m,** è discussa nel paragrafo '3'. Per trovare il mediano di V, si riutilizza in modo ricorsivo ***sampleMedianSelect***. Una volta determinato il pivot dell'algoritmo di selezione, si partiziona attorno ad esso e si va ad effettuare la ricerca

dell'elemento desiderato in modo ricorsivo, così come avveniva nell'algoritmo ***select*** classico. Infine, abbiamo scelto di lasciare il valore di **minLen** a 3, come nel codice visto a lezione, in quanto esso è il valore ottimale per eseguire il ***trivialSelect***, da noi impiegato per la selezione del k-esimo elemento nel passo base.

Sort: la funzione ***Sort*** accetta come unico parametro in ingresso la lista da ordinare e chiama ***Modify_Quick_Sort*** con la stessa lista, '0' e 'lunghezza lista - 1' come indici left e right, in tal modo si comincia a ordinare tenendo conto di tutta la lista. In seguito, si calcola come **long** la lunghezza della porzione di lista da considerare tramite la sottrazione di right e left, effettuando un controllo che alla fine dell'ordinamento permetterà di uscire dalla ricorsione. La funzione chiama ***sampleMedianSelect*** con parametri new e int(long/2), dove new è la copia della porzione della lista considerata e int(long/2) chiede all'algoritmo di selezione di andare a cercare il mediano di new. Abbiamo pensato di passare come argomento una copia della lista piuttosto che la lista stessa perché in tal modo si evita di disordinare ulteriormente la lista, in quanto nella selezione sono effettuate delle partizioni. Una volta identificato il pivot, la funzione effettua una partizione e due chiamate ricorsive come fa ***quickSort*** di consueto.

3. Test

Il codice di testing si trova nel file **Testing.py**.

Abbiamo creato diverse funzioni chiamate ***test*** che si muovono tenendo conto della stessa logica, cambiano solamente la funzione per ordinare e il file *.csv* in cui sono salvati i tabulati.

salvaInFile: salva i risultati del test in un file .csv.

createList: crea una lista di elementi in base ad un parametro, chiamato **type,** che ne determina l'ordinamento in modo crescente, decrescente o randomico.

timeTest: data una lista e una funzione di ordinamento, calcola il tempo di esecuzione dell'algoritmo su quell'input.

I test che verranno illustrati nel paragrafo 5, sono stati effettuati su un array randomico con elementi tra 0 e 'steps' (ovvero la dimensione dell'array passato). Inoltre, per poter confrontare gli algoritmi su gli stessi parametri, abbiamo impostato il random seed ad un valore fisso, in modo tale da ricreare la stessa lista data una certa lunghezza.

Infine, abbiamo impostato diversi range per effettuare i test, in particolare, per dimensioni grandi vi è un salto grande (second e third), in quanto si passa da 50'000 elementi a 100'000, per questo motivo nei grafici risulta esserci un'impennata verso i valori finali. Abbiamo fatto ciò per osservare il comportamento degli algoritmi con array di ingenti dimensioni.

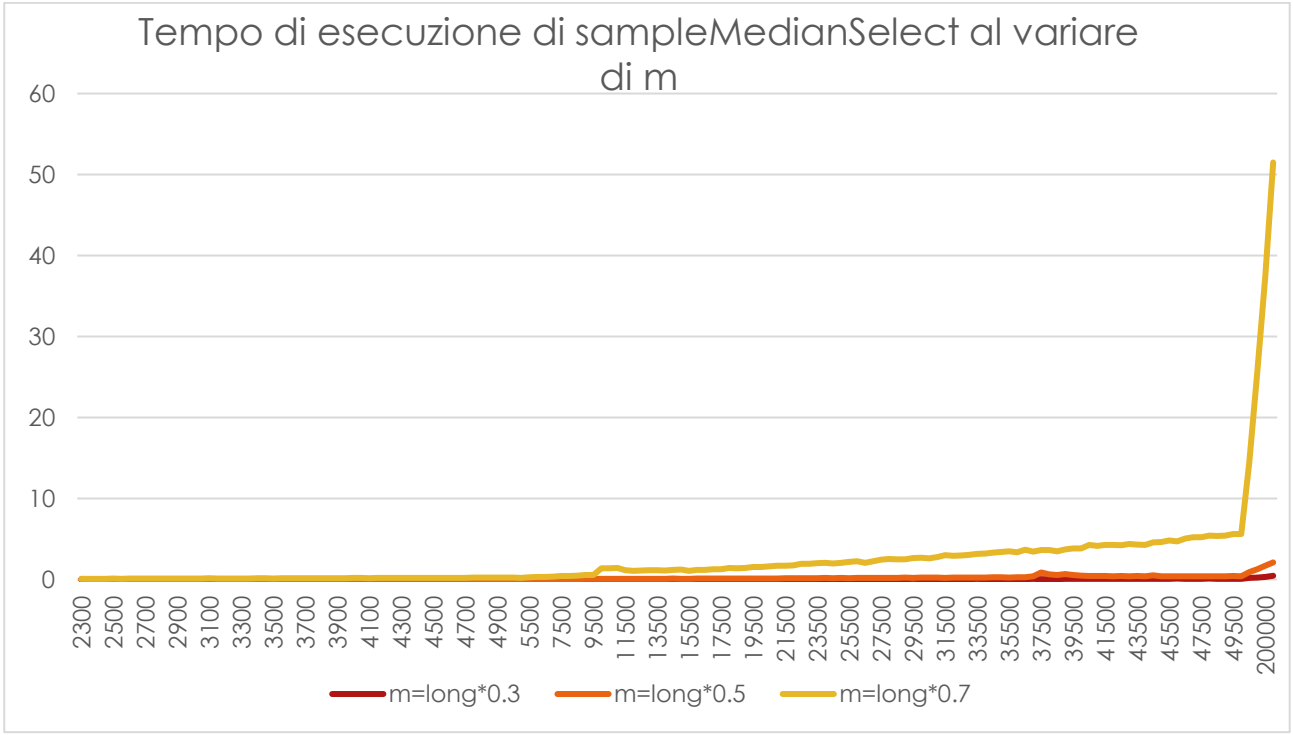
4. Scelta della dimensione del sotto-array randomico

La scelta della lunghezza **m** del sotto-array randomico V, avviene in funzione degli indici passati come parametri (left e right).

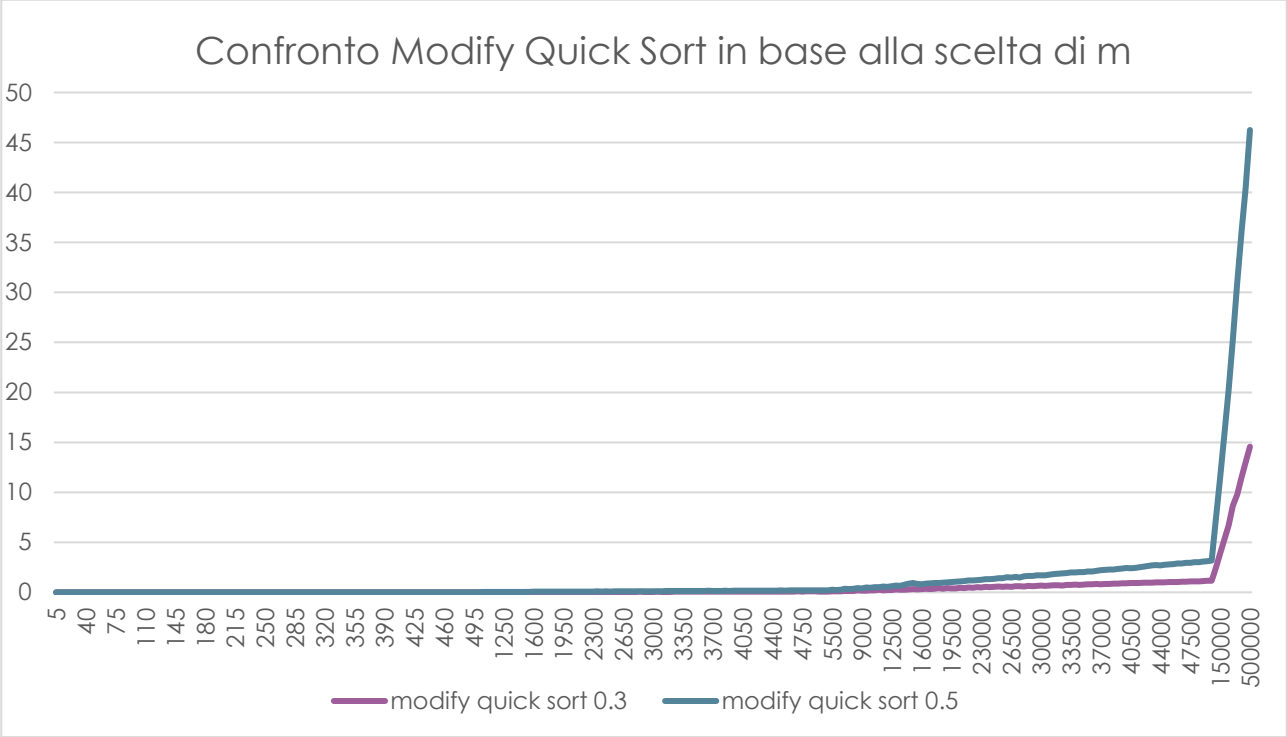
Se avessimo lasciato **m** fisso, esso avrebbe dovuto assumere un valore minore o uguale a quello di **minLen**, solo in questo caso l'algoritmo avrebbe funzionato correttamente. Infatti, se si assegnasse a **m** un valore maggiore di **minLen**, si ricadrebbe in una ricorsione infinita, poiché la dimensione del sotto-array non diminuirebbe e non si arriverebbe mai al caso base. Infine, ponendo un valore fisso di **m** accettabile, non si avrebbe una "vera" ricorsione, infatti, si effettuerebbe sempre il caso base, selezionando l'elemento desiderato tramite ***trivialSelect***.

Come si può evincere dai test, più piccolo è il valore di **m** e più l'algoritmo risulta competitivo, tuttavia ridurlo eccessivamente porterebbe a considerare una campionatura non adeguata. Inoltre, abbiamo riscontrato che con m = 1 l'algoritmo ***sampleMedianSelect*** si "trasforma" in ***quickSelect*** randomico.

Abbiamo scelto di impostare il valore di *m = long*0.3* in quanto rende l'algoritmo molto veloce, andando però a considerare una porzione di array significativa. Scegliendo come grandezza del sotto-array *m = long*0.5 (o long*0.7)* il tempo di esecuzione aumenta di molto, soprattutto nel secondo caso, ma la campionatura risulta molto di più precisa. In conclusione, il primo valore di **m** presentato risulta quello ottimale, in quanto già con il secondo valore, il tempo di esecuzione aumenta di molto.



Come si può notare la funzione ***sampleMedianSelect***, così come ***Modify_Quick_Sort***, lavora meglio con *m=long*0.3* per i motivi prima illustrati. Inoltre, scegliendo un valore minore, il tempo di esecuzione diminuirebbe di molto, ma si andrebbe a lavorare su una porzione di array non molto significativa.



5. Confronto con gli altri algoritmi di ordinamento

In base ai test effettuati, appare chiaro che anche ***Modify_Quick_Sort*** è un algoritmo di ordinamento ottimale, infatti, è molto più veloce degli algoritmi che ordinano in tempo quadratico, come ***Bubble Sort, Insertion Sort*** e ***Selection Sort***, e competitivo con quelli che ordinano in tempo logaritmico.

Come si può notare dai grafici riportati nella pagina successiva, quando gli algoritmi che ordinano in $O(n^2)$ hanno un'impennata, ***Modify_Quick_Sort*** rimane nella zona degli algoritmi che ordinano in O(nlogn).

Quando confrontiamo più da vicino il nostro algoritmo con quelli più rapidi, si nota che, pur mantenendo un tempo di esecuzione molto basso, esso è più lento degli altri.

In particolare, il ***Quick Sort*** classico ordina molto più velocemente e il metodo ***Sort*** di python, non solo batte il nostro ***Modify_Quick_Sort***, ma si è rivelato essere l'algoritmo di ordinamento più veloce in assoluto.