Simone Demuro
65879

Relazione per il corso Algoritmi e Strutture Dati: Algoritmi di ordinamento

Nella presente relazione focalizzandoci sui risultati pratici allegati si discuteranno cinque tra i più noti algoritmi di ordinamento dati.

I criteri di misurazione delle performances presi in considerazione sono: il numero di confronti tra elementi, il numero di scambi effettuati tra questi e il tempo richiesto per completare l'ordinamento, al netto delle operazioni preliminari (allocazione memoria, generazione dei dati di test, etc.). Tali misurazioni sono state effettuate su quattro diversi schemi di input: {ordinato, parzialmente ordinato, inversamente ordinato, casuale} al variare della dimensione degli stessi, con: {500, 1.000, 2.000, 5.000, 10.000, 20.000, 50.000} elementi. In totale sono stati effettuati 140 test (5 algoritmi × 4 schemi × 7 dimensioni).

Dalla tabella allegata si evincerà come i dati raccolti rispecchino abbastanza fedelmente la loro stima formale del costo computazionale.

Selection Sort: In favore della sua semplicità implementativa, che lo rende un algoritmo di rilievo soprattutto a fini didattici è al contempo uno dei meno efficienti. Confrontano tutti gli elementi tra loro e cercando ogni volta il minore, la sua complessità computazionale è infatti $O(n^2)$ per ogni tipo di sequenza in Input penalizzando molto in termini di tempo. Per quanto riguarda la complessità a livello di spazio non richiede l'utilizzo di memoria aggiuntiva.

Insertion Sort: Nel raro caso si debba avere il compito di "ordinare" sequenze già ordinate, questo algoritmo ha le performances migliori con un costo computazionale di $O(n)$ e nessuno scambio. In scenari medi ha una complessità di $O(n^2)$. Funziona confrontando l'elemento i-esimo solo con i suoi predecessori e, nel caso, scambiarlo con uno di questi se non è al suo posto.

Heap Sort: Come riporta il nome, sfrutta la caratteristica della struttura heap di essere ordinata per prelevare a ogni iterazione l'elemento radice, quindi "riadattare" la struttura, fino a che questa non sarà vuota. Il suo comportamento e le sue performances in termini di tempo sono prossime a quelle degli algoritmi di ordinamento ottimi a seguire.

Quick Sort: Sicuramente si tratta di uno dei più famosi algoritmi di ordinamento consiste nello spostare alla sua sinistra tutti gli elementi minori e a destra tutti quelli maggiori, di un elemento chiamato pivot, si ripetere poi l'operazione sui sottogruppi di destra e di sinistra finché le sotto sequenze conteranno un elemento solo. La sua implementazione non spicca per semplicità, le sue performance possono cambiare in base alla scelta che si fa del pivot. Il suo costo computazionale medio è di $O(n \log(n))$, le performances degradano in caso di input ordinati o inversamente ordinati o parzialmente, fino a $O(n^2)$. Si sono sperimentate problematiche (risolte mediante modifiche alle opzioni di compilazione) di stabilità del programma in presenza di input molto grandi, ma si attribuiscono le cause alla sua implementazione ricorsiva che potrebbe aver generato troppe chiamate sul call-stack.

Merge Sort: Il merge sort è un algoritmo di ordinamento ottimo in quanto, per ogni sequenza in ingresso, la sua complessità risulta sempre $O(n \log(n))$ che ne rappresenta il limite asintotico stretto. La sua implementazione si presta bene alla ricorsione dando luogo ad un'implementazione che si distingue per raffinatezza. Esso ripartisce la sequenza in due sotto-sequenze dalle medesime dimensioni, ripete l'operazione a destra e sinistra fino a che si hanno delle sequenze unitari, quindi ordinate. A questo punto si combinano tra di loro andando a ritroso fondendosi, ottenendo un input ordinato.

Dai test effettuati si osserva inoltre che fino a quando non si sono raggiunte dimensioni elevate di input tutti gli algoritmi svolgono il loro compito in tempistiche non apprezzabili nemmeno dal tramite il calcolo del tempo di CPU espresso in unità clock_t tramite la funzione clock() di time.h. Al raggiungimento di ingenti quantità di dati in ingresso ha senso stimare almeno empiricamente o statisticamente se questi siano già parzialmente ordinati, inversamente ordinati o se non si possano fare astrazioni di questa sorta. Inoltre è da considerare (soprattutto in base al tipo di dato che si vuole ordinare) il quantitativo di memoria primaria che si ha a disposizione, nel caso fosse carente optare per algoritmi che non copiano i dati ma agiscono sull'input.