UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato di in **Algoritmi e Strutture Dati**

Analisi e confronto delle prestazioni di Counting Sort e QuickSort

Anno Accademico 2020/2021

Domenico Fordellone

Introduzione

Qualsiasi algoritmo di ordinamento basato sul confronto richiede nel caso peggiore un tempo di esecuzione non lienare del tipo $\Omega(n*\log(n))$. Gli algoritmi di ordinamento in tempo lineare garantiscono un tempo di esecuzione lineare anche nel caso peggiore. Per dare questa garanzia si fanno delle assunzioni sugli elementi da ordinare. In questo elaborato verranno presentati due algoritmi, uno basato sui confronti, l'altro non basato sui confronti. Verrà poi condotta un'analisi sui rispettivi tempi di esecuzione al fine di evidenziare le differenze tra i due algoritmi.

Indice

Capitolo 1

Counting Sort

Counting Sort è un algoritmo di ordinamento lineare, ciò vuol dire che non si basa sul confronto per ordinare gli elementi. Questo algoritmo assume che i valori da ordinare siano interi siano compresi nel range [0,k]. Non è un algoritmo che ordina sul posto, infatti richiede una quantità di memoria proporzionale alla quantità di memoria richiesta per memorizzare il vettore da ordinare. Si utilizzano due vettori di appoggio:

- B, di lunghezza n, che mantiene i valori ordinati
- C, di lunghezza k+1, che indica le occorrenze di ciascuno valore compreso tra 0 e k.

Come ordino gli elementi senza confornarli tra loro?

- Per ciascun intero i compreso tra 0 e k, si contano quanti elementi pari ad i ci sono nel vettore da ordinare
- Per ciascun intero i compreso tra 0 e k, si determinano quanti elementi minori o uguali ad i ci sono nel vettore da ordinare
- Ciò ci indica in che posizione deve stare ciascun elemento.

Questo è un algoritmo di ordinamento stabile: se sono presenti due o più elementi con lo stesso valore nel vettore da ordinare, questi si troveranno nel vettore ordinato nella stessa posizione relativa tra di loro.

1.1 Pseudocodice

```
Counting-Sort (A,B,k)
for i ← 0 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]]+1
// C[i] è il numero di occorrenze di i
for i ← 1 to k
    do C[i] ← C[i]+C[i-1]
// C[i] è il numero di elementi ≤ i
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]]-1
```

Figura 1.1: Pseudocodice di Counting Sort

L'algoritmo riceve in ingresso il vettore da ordinare, il vettore in cui mettere gli elementi ordinati e k che identifica l'intervallo di valori validi.

1.2 Analisi Complessità

In ogni ciclo for facciamo delle operazioni che richiedono un tempo costante, la complessità dell'algoritmo sarà data dal numero di iterazioni dei diversi cicli for. Il primo ed il terzo ciclo for eseguono un numero di iterazioni proporzionale a k, il secondo ed il quarto eseguono un numero di iterazioni proporzionale a n. Complessivamente l'algoritmo ha un tempo di esecuzione del tipo $\Theta(n + k)$. Si distinguono due casi:

- k<n: nella somma prevale il termine n, otteniamo un tempo di esecuzione lineare $\Theta(n)$. Per rientrare in questo caso il numero di elementi da ordinare deve essere maggiore del numero di elementi ammissibili nel vettore, ovvero nel vettore devono essere presenti degli elementi uguali

- k>n: nella somma prevale il termine k, otteniamo un tempo di esecuzione lineare $\Theta(n + k)$. In questo caso vedremo che Counting Sort non è più efficiente di un algoritmo basato sul confronto.

1.3 Implementazione C++

```
void CountingSort(int* A, int* B, int n, int k) {
    int* C;
    C = new int[k + 1];

    for (int i = 0; i <= k; i++) {
        C[i] = 0;
    }

    for (int j = 0; j < n; j++) {
        C[A[j]] = C[A[j]] + 1;
    }

    for (int i = 1; i <= k; i++) {
        C[i] = C[i] + C[i - 1];
    }

    for (int j = n - 1; j >= 0; j--) {
        B[C[A[j]] - 1] = A[j];
        C[A[j]] = C[A[j]] - 1;
    }

    delete[] C;
}
```

Figura 1.2: Counting Sort implementato in C++

L'algoritmo è stato implementato in C++, segue la stessa logica dello pseudocodice, l'unica cosa che cambia è la gestione degli indici.

Al fine di condurre un'analisi sui tempi di esecuzione generati dall'algoritmo al crescere della dimensione del problema in ingresso ed evidenziare le differenze tra i vari casi è stato implementato un main in cui vengono generati vettori con una dimensione che varia nel range [100,100000]. Per ogni dimensione vengono effettuate 50 misurazioni diverse i cui tempi di esecuzione esecuzione sono stati mediati per ottenere dei valori più realistici.

I diversi risultati ottenuti vengono salvati in un file txt che sarà poi analizzato da un script MATLAB al fine di graficare l'anadamento del tempo di esecuzione al crescere della dimensione del problema in ingresso.

Nel main si analizza sia il best case che il worst case di Counting Sort. Per il pri-

mo viene considerato un valore di k«n, infatti, per ogni dimensione, come k viene preso un valore casuale minore di n. Nel secondo caso k assume un valore sempre maggiore della dimensione n del problema in ingresso, nello specifico k varia nel range [0,2n].

```
int main() {

    double num_test = 50.0;
    int sizes[times];
    int index = 0;
    double somma_CS = 0;
    double tempo_CS = 0;
    double risultati_CS[times];

    for (int n = 100; n <= 100000; n = n + 100) {
        sizes[index] = n;
        somma_CS = 0;
        int* A = new int[n];
        int* B = new int[n];
        srand(time(NULL));

        for (int i = 0; i < 50; i++) {
            int k = rand() % n;                 //best case:  k<n
            //int k = rand() % n + n;           //worst case: k>n

            for (int j = 0; j < n; j++) {       //creo vettore da ordinare
                A[j] = rand() % (k + 1);        //restituisce un valore nel range [0,k]
            }

            auto start_CS = chrono::high_resolution_clock::now();
            CountingSort(A, B, n, k);
            auto end_CS = chrono::high_resolution_clock::now();
            somma_CS += chrono::duration_cast<chrono::nanoseconds>(end_CS - start_CS).count();

        }
        tempo_CS = somma_CS / num_test;

        risultati_CS[index] = tempo_CS;
        index++;

        delete[] A;
        delete[] B;
    }

    return 0;
}
```

Figura 1.3: mainCS implementato in C++

```
//Scrittura su file
ofstream output;
output.open("best_case_CS.txt", ios::out | ios::app);
//output.open("worst_case_CS.txt", ios::out | ios::app);
if (output.is_open()) {
    for (int i = 0; i < times; i++) {
        output << sizes[i] << "," << risultati_CS[i];
        output << endl;
    }

}
output.close();
```

Figura 1.4: scrittura su file implementata in C++

1.3.1 Analisi dei tempi di esecuzione

Andiamo ora ad analizzare i risultati ottenuti con il software MATLAB, in particolare verrà rappresentato l'andamento del tempo di esecuzione nei vari casi. Inoltre, verranno scelte opportune costanti al fine di dimostrare la legge di crescita asintotica vista in teoria.

Best-case

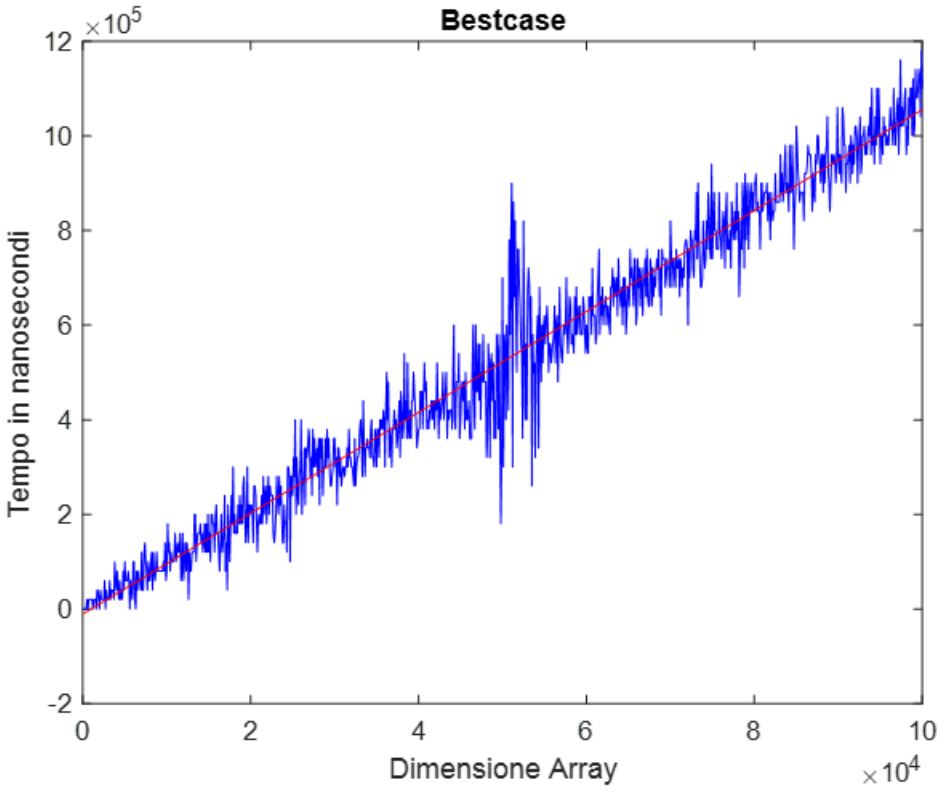


Figura 1.5: best-case Counting Sort in MATLAB

La funzione che descrive la retta che approssima l'andamento del tempo di esecuzione è:

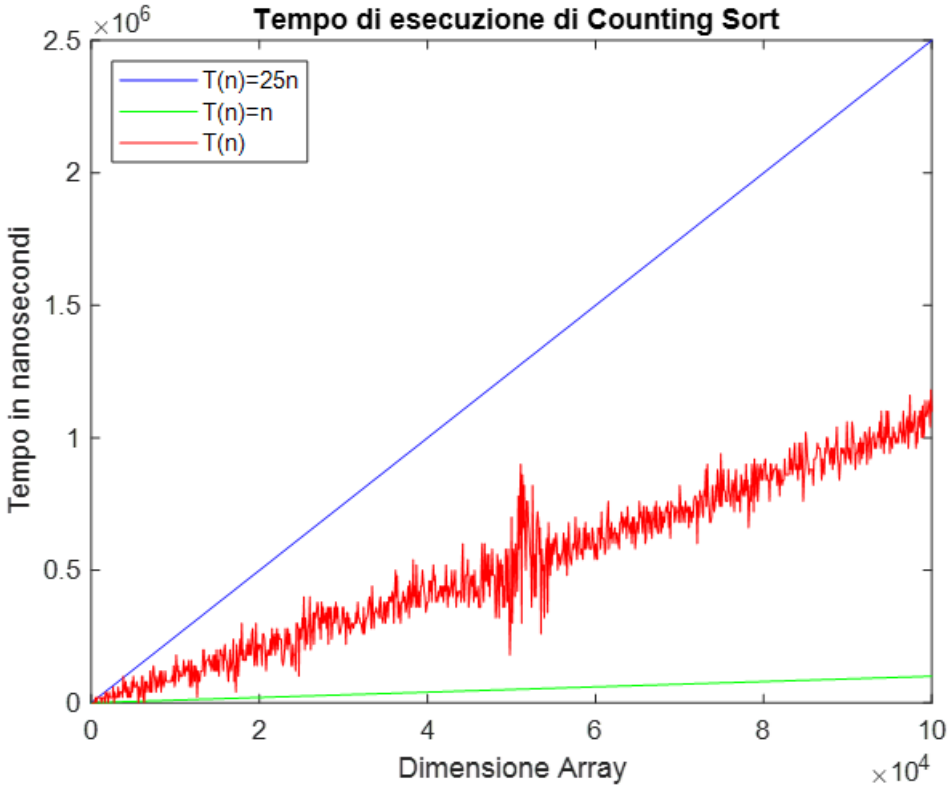
$$f(n) = 10.6642n - 10735 \tag{1.1}$$

Poiché il tempo di esecuzione del Counting Sort nel caso migliore è $\Theta(n)$ bisogna quindi risolvere il seguente sistema di disequazioni:

$$\begin{cases} 10.6642n - 10735 \leq c_2 n \\ 10.6642n - 10735 \geq c_1 n \geq 0 \end{cases} \tag{1.2}$$

La disequazione è positiva per $n \geq 1006.6$.

Scegliendo $n_0 = 2000$, $c_1 = 1$ e $c_2 = 25$, la curva che descrive il tempo di esecuzione rientra nel limite teorico:



Worst-case

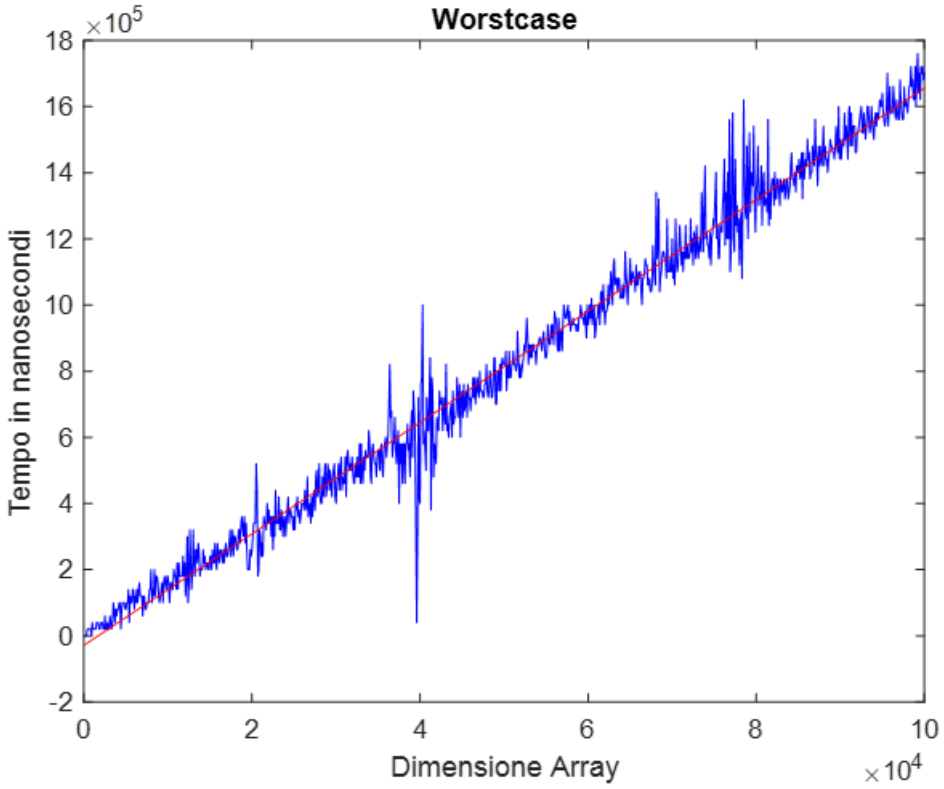


Figura 1.6: worst-case Counting Sort in MATLAB

La funzione che descrive la retta che approssima l'andamento del tempo di esecuzione è:

$$f(n) = 16.8476n - 28852 \tag{1.3}$$

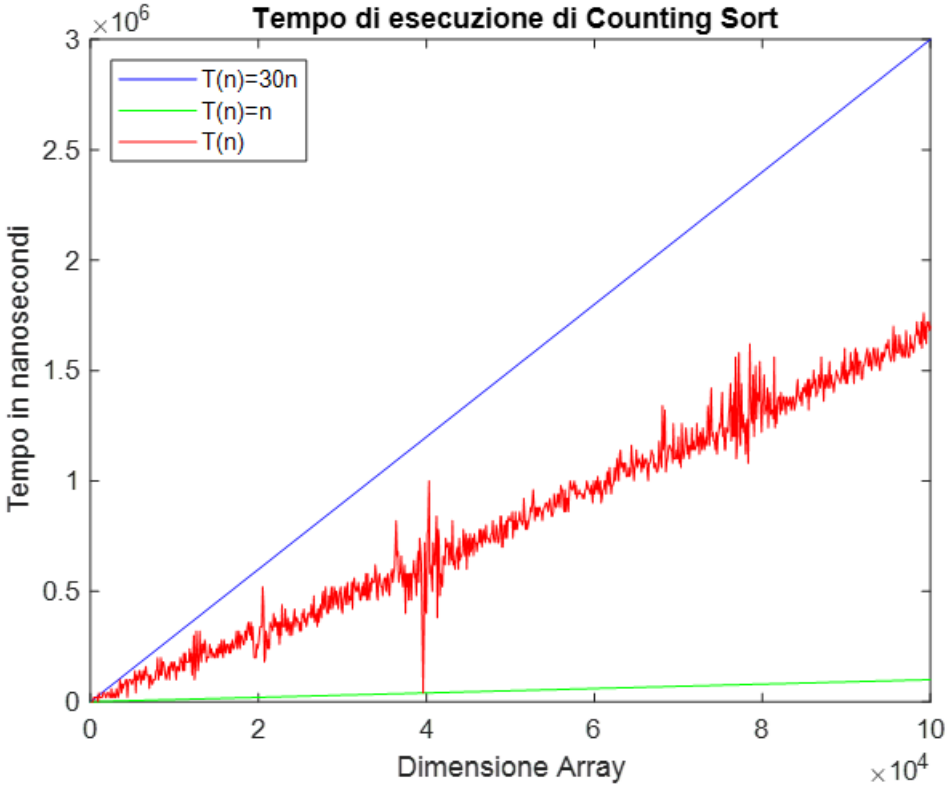
Poiché il tempo di esecuzione del Counting Sort nel caso peggiore è $\Theta(n+k)$ bisogna quindi risolvere il seguente sistema di disequazioni:

$$\begin{cases} 16.8476n - 28852 \leq c_2 n \\ 16.8476n - 28852 \geq c_1 n \geq 0 \end{cases} \tag{1.4}$$

La disequazione è positiva per $n \geq 1712.53$.

Scegliendo $n_0 = 3000$, $c_1 = 1$ e $c_2 = 30$, la curva che descrive il tempo di esecuzione

rientra nel limite teorico:



Confronto tra worst e best case

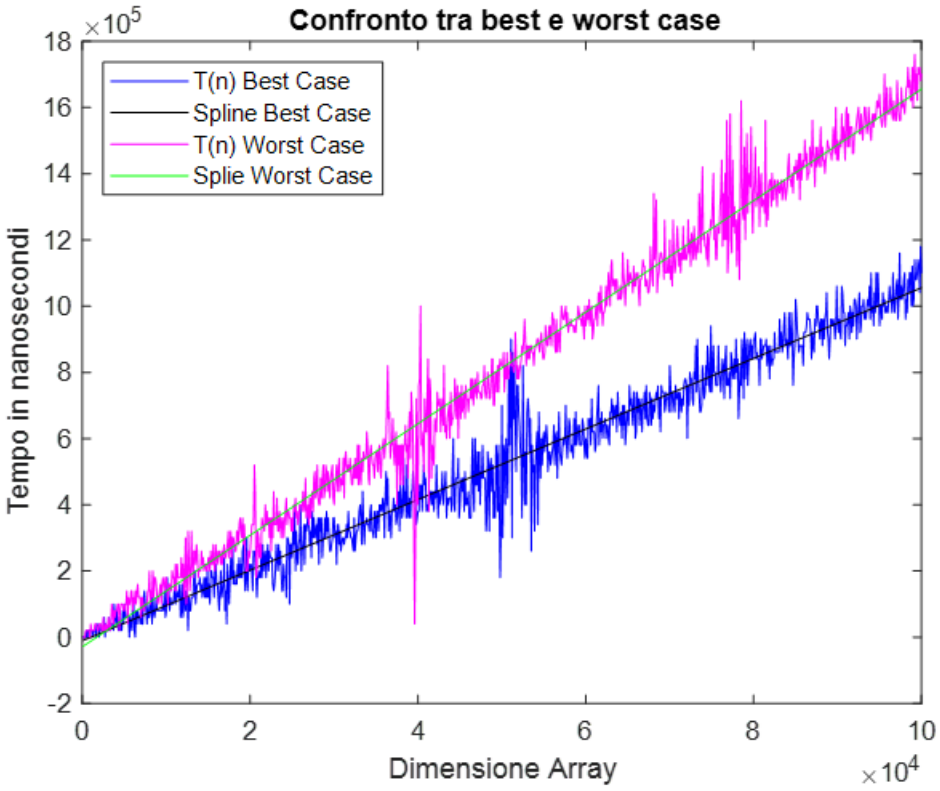


Figura 1.7: worst-case Counting Sort in MATLAB

Come si può notare dal grafico le prestazioni tra i due casi peggiorano notevolmente, ne consegue che il valore di k assume un ruolo fondamentale per la determinazione della legge di crescita del tempo di esecuzione.

Capitolo 2

QuickSort

QuickSort è un algoritmo di ordinamento basato sul confronto che prevede nel caso peggiore un tempo di esecuzione del tipo $\Theta(n^2)$, mentre nel caso migliore prevede un tempo di esecuzione del tipo $O(n * log(n))$. Per QuickSort ha senso considerare il caso medio, in quanto si presenta molto spesso nella pratica ed ha un tempo di esecuzione del tipo $O(n * log(n))$.
Al di là della complessità asintotica, quello che si riscontra per QuickSort è che i fattori nascosti nella notazione asintotica sono piuttosto piccoli, ecco perchè nella pratica risulta essere uno dei più efficienti (a meno che non rientriamo nel caso peggiorem caso abbastanza singolare).
QuickSort prevede una diversa applicazione del paradigma del **divide et impera**, in particolare utilizza una funzione chiamata Partition che sceglie un elemento detto **pivot** e riorganizza gli elementi del vettore in maniera tale da avere alla sinistra del pivot tutti gli elementi minori o uguali del pivot ed alla destra del pivot tutti gli elementi maggiori del pivot, si creano quindi due problemi da risolvere ricorsivamente. A differenza di MergeSort qui il costo per ricombinare le soluzioni è costante.

2.1 Pseudocodice QuickSort

```
Quicksort (A,p,r)
if p<r
    then q ← Partition (A,p,r)
    Quicksort (A,p,q-1)
    Quicksort (A,q+1,r)
```

Figura 2.1: Pseudocodice di QuickSort

Riceve in ingresso il vettore A e gli indici che identificano il sottovettore corrispondente all'istanza che vogliamo risolvere.
La ricorsione termina quando ci troviamo nel caso banale, ovvero un vettore di dimensione uno. Se non ci troviamo in questo utlimo caso viene chiamata la funzione Partitin che restituisce la posizione in cui va inserito il pivot, grazie alla quale identifico i due sottoproblemi da risolvere ricorsivamente.
Vediamo la funzione Partition:

```
Partition (A,p,r)
x ← A[r]
i ← p-1
for j ← p to r-1
    do if A[j]⩽x
        then i ← i+1
        exchange A[i]↔A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

Figura 2.2: Pseudocodice della Partition

Riceve in ingresso il vettore A e gli indici che rappresentano la porzione del vettore da partizionare attorno al pivot.

2.2 Analisi complessità

La complessità di QuickSort dipende dal costo di tutte le Partition che vengono eseguite.

Nel **caso banale**, che si verifica quando la sequenza in ingresso contiene un solo elemento, QuickSort non fa niente, invece negli altri casi QuickSort suddivide il vettore in due sottovettori tramite la Partition ed invoca ricorsivamente sé stesso sulle due partizioni create.

Distinguiamo vari casi.

2.2.1 Worst case

Questo caso si verifica quando ad ogni invocazione della Partition abbiamo un partizionamento bilanciato, ovvero quando scegliamo come pivot o l'elemento più grande oppure l'elemento più piccolo del vettore. Si generano due partizioni, una di dimensione nulla, l'altra di dimensione n-1. Il tempo di risoluzione della partizione nulla è costante, rimane quindi un solo sottoproblema da risolvere.

Si dimostra che in questo caso il tempo di esecuzione è del tipo $\Theta(n^2)$.

2.2.2 Best case

Si verifica quando ad ogni partizionamento scegliamo il pivot in maniera tale che le partizioni abbiano la stessa dimensione.

Si dimostra che in questo caso il tempo di esecuzione è del tipo $O(n * log(n))$.

2.2.3 Average case

Si dimostra che qualunque partizionamento con rapporto costante produce un tempo di esecuzione la cue legge asintotica è del tipo $O(n * log(n))$. Maggiore sarà il rapporto, maggiore saranno i fattori moltiplicativi nascosti. Notiamo quindi che

nel caso medio otteniamo la stessa legge di crescita del tempo di esecuzione per il caso banale.

2.3 Implementazione C++

```
void QuickSort(int* A, int p, int r) {
    if (p < r) {
        int q = Partition(A, p, r);
        QuickSort(A, p, q - 1);
        QuickSort(A, q + 1, r);
    }
}
```

Figura 2.3: QuickSort implementato in C++

```
int Partition(int* A, int p, int r) {
    int x = A[r];
    int i = p - 1;
    for (int j = p; j < r; j++) {
        if (A[j] <= x) {
            i = i + 1;
            swap(A[i], A[j]);
        }
    }
    swap(A[i + 1], A[r]);
    return i + 1;
}
```

Figura 2.4: Partition implementata in C++

L'algoritmo è stato implementato in C++, segue la stessa logica dello pseudocodice, l'unica cosa che cambia è la gestione degli indici.

Al fine di condurre un'analisi sui tempi di esecuzione generati dall'algoritmo al crescere della dimensione del problema in ingresso ed evidenziare le differenze tra i vari casi è stato implementato un main in cui vengono generati vettori con una dimensione che varia nel range [10,10000]. Per ogni dimensione vengono effettuate 50 misurazioni diverse i cui tempi di esecuzione esecuzione sono stati mediati per ottenere dei valori più realistici.

I diversi risultati ottenuti vengono salvati in un file txt che sarà poi analizzato da un script MATLAB al fine di graficare l'anadamento del tempo di esecuzione al crescere della dimensione del problema in ingresso.

Nel main si analizza sia l'average case che il worst case di QuickSort. Per il worst case forniamo in ingresso un vettore già ordinato con Counting Sort.

```
int main() {

    double num_test = 50.0;
    int sizes[times];
    int index = 0;
    double tempo_QS = 0;
    double somma_QS = 0;
    double risultati_QS[times];

    for (int n = 10; n <= 10000; n = n + 10) {
        sizes[index] = n;
        somma_QS = 0;
        int* A = new int[n];
        int* Q = new int[n];
        srand(time(NULL));

        for (int i = 0; i < 50; i++) {
            int k = rand() % n;

            for (int j = 0; j < n; j++) {           //creo vettore da ordinare
                A[j] = rand() % (k + 1);
            }

            //CountingSort(A, Q, n, k);             //worst-case QS: Q sarà un vettore ordinato

            auto start_QS = chrono::high_resolution_clock::now();
            QuickSort(A, 0, n-1);
            //QuickSort(Q, 0, n-1);                 //QS riceve un vettore già ordinato
            auto end_QS = chrono::high_resolution_clock::now();
            somma_QS += chrono::duration_cast<chrono::nanoseconds>(end_QS - start_QS).count();

        }
        tempo_QS = somma_QS / num_test;

        risultati_QS[index] = tempo_QS;
        index++;

        delete[] A;
        delete[] Q;
    }
    return 0;
}
```

Figura 2.5: mainQS implementato in C++

2.3.1 Analisi dei tempi di esecuzione

Andiamo ora ad analizzare i risultati ottenuti con il software MATLAB, in particolare verrà rappresentato l'andamento del tempo di esecuzione nei vari casi. Inoltre, verranno scelte opportune costanti al fine di dimostrare la legge di crescita asintotica vista in teoria.

Average case

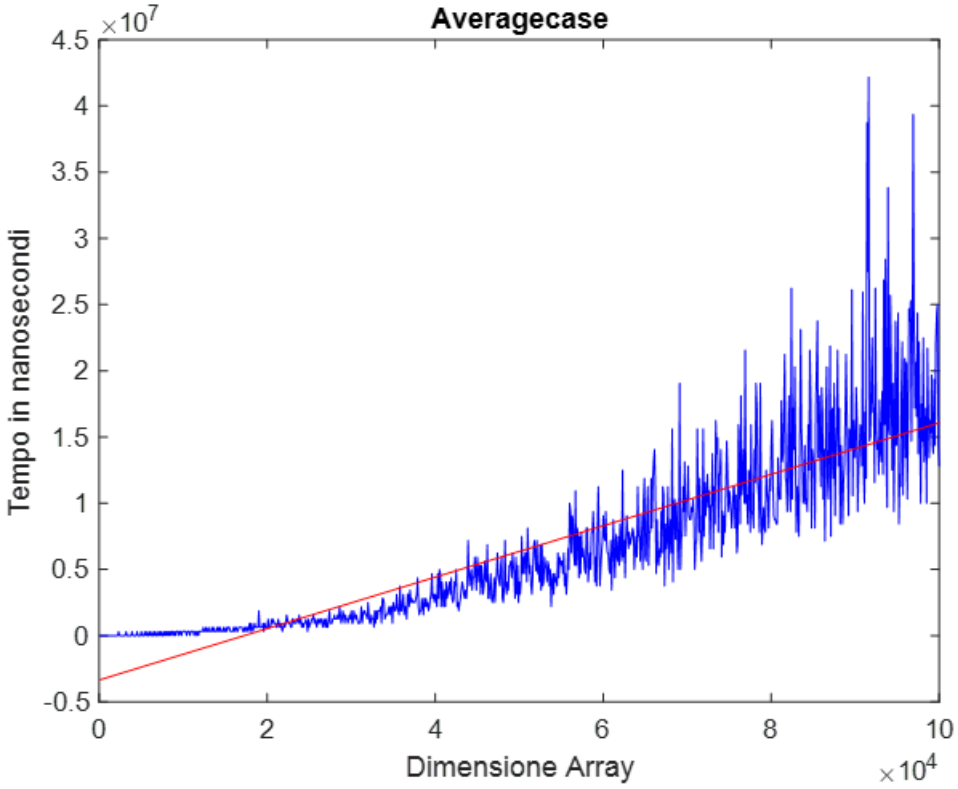


Figura 2.6: average-case QuickSort in MATLAB

La funzione che descrive la retta che approssima l'andamento del tempo di esecuzione è:

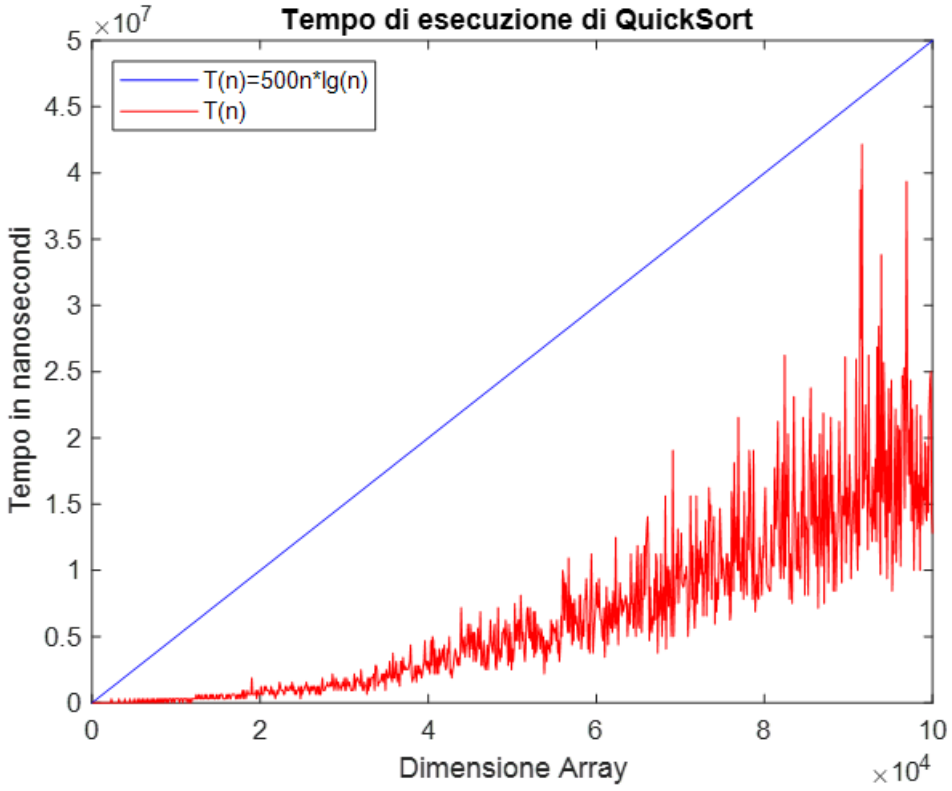
$$f(n) = 194.1570n * lg(n) - 33505 \tag{2.1}$$

Poiché il tempo di esecuzione di QuickSort nel caso medio è $O(n * log(n))$ bisogna quindi risolvere la disequazione:

$$\begin{cases} 0 \leq 194.1570n * lg(n) - 33505 \leq c_1 n * lg(n) \end{cases} \tag{2.2}$$

La disequazione è positiva per $n \geq 172.57$.

Scegliendo $n_0 = 300$ e $c_1 = 500$ la curva che descrive il tempo di esecuzione rientra nel limite teorico:



Worst case

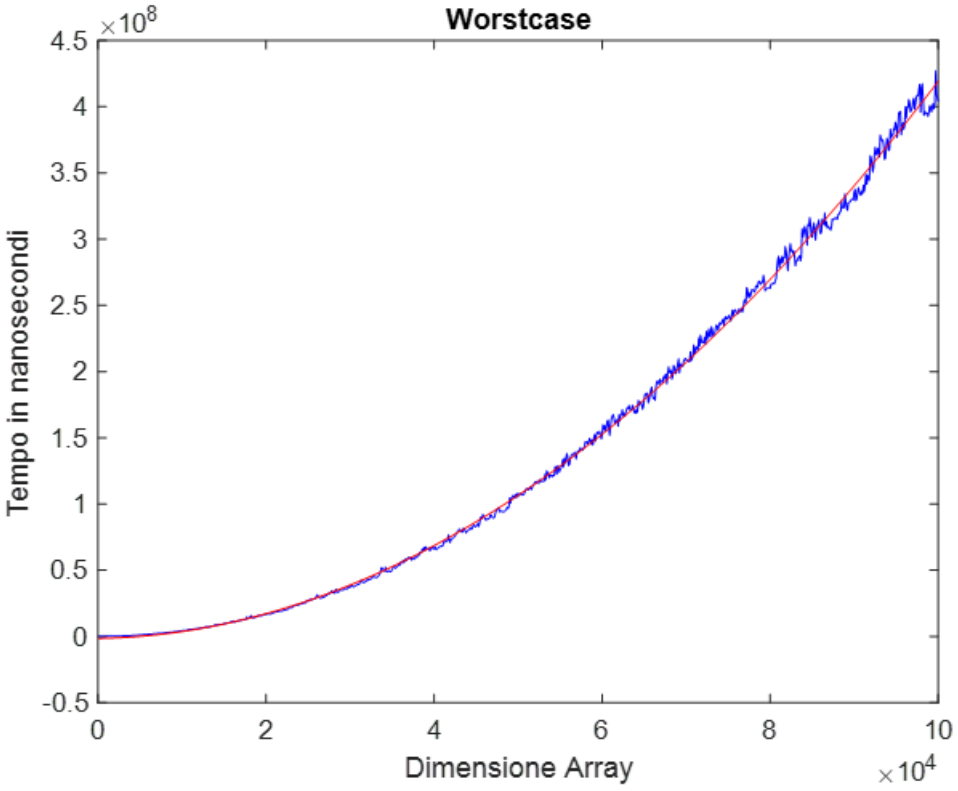


Figura 2.7: worst-case QuickSort in MATLAB

La funzione che descrive la retta che approssima l'andamento del tempo di esecuzione è:

$$f(n) = 0.0409n^2 + 117.1566n - 15373 \tag{2.3}$$

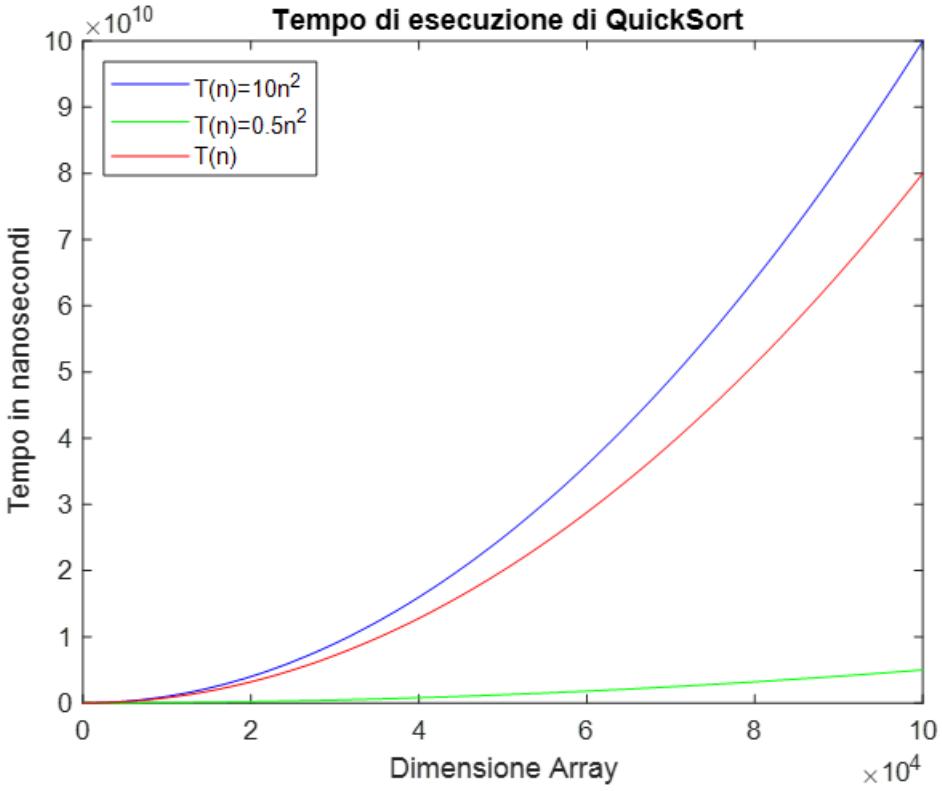
Poiché il tempo di esecuzione di QuickSort nel caso peggiore è $\Theta(n^2)$ bisogna quindi risolvere il seguente sistema di disequazioni:

$$\begin{cases} 0.0409n^2 + 117.1566n - 15373 \leq c_2 n^2 \\ 0.0409n^2 + 117.1566n - 15373 \geq c_1 n^2 \geq 0 \end{cases} \tag{2.4}$$

La disequazione è positiva per $n \geq 1558.45$.

Scegliendo $n_0 = 3000$, $c_1 = 0.5$ e $c_2 = 10$, la curva che descrive il tempo di

esecuzione rientra nel limite teorico:



Confronto tra worst case ed average case

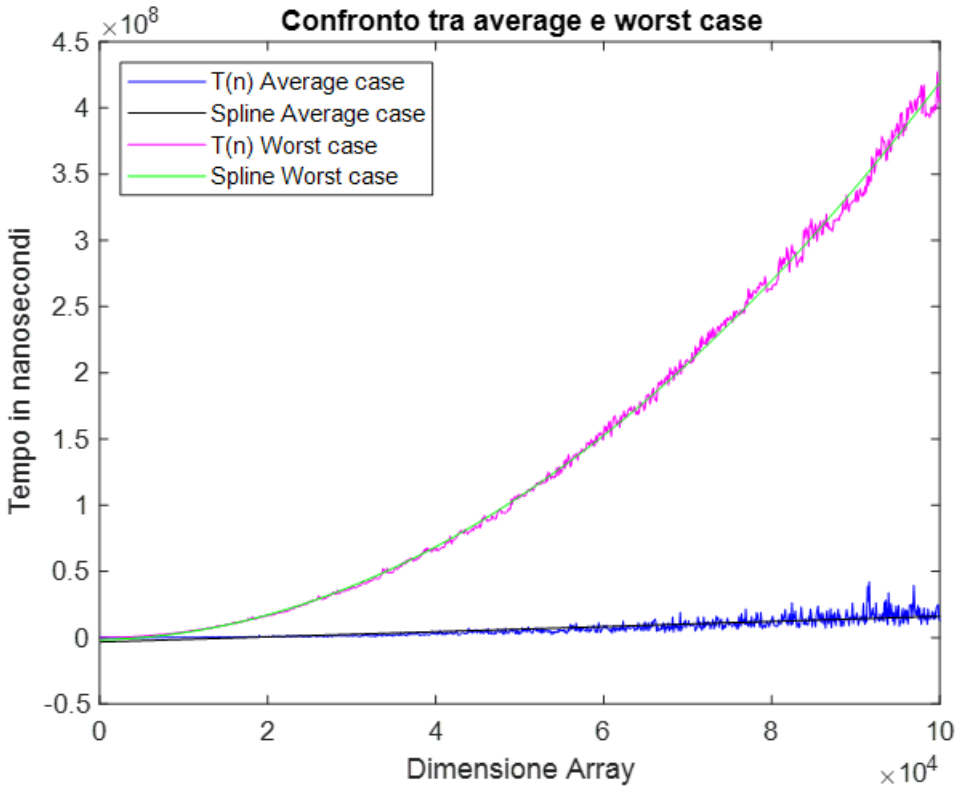


Figura 2.8: worst-case QuickSort in MATLAB

Come si può notare dal grafico le prestazioni tra i due casi peggiorano notevolmente.

Capitolo 3

Confronto tra QuickSort e Counting Sort

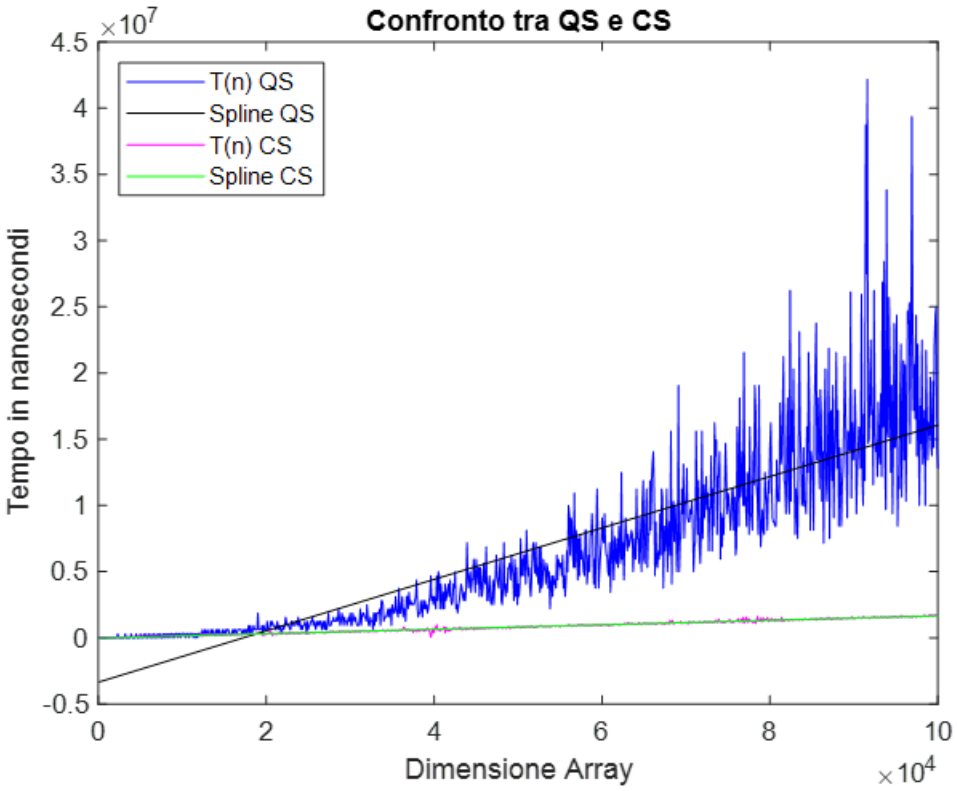
Dal momento che Counting Sort non ordina nemmeno sul posto, vogliamo capire quanto è efficiente utilizzare Counting Sort rispetto ad un algoritmo di ordinamento basato sul confronto.
Questo dipende dai valori che assumo n e k, infatti si evidenziano vari casi:

- n<k: per k non molto grande Counting Sort risulta più efficiente di QuickSort
- n«k: per n molto più piccolo di k QuickSort risulta più efficiente di CountingSort, infatti in questa situazione i cicli for presenti in Counting Sort fanno molte iterazioni a vuoto, invece QuickSort che confronta soltanto gli elementi nel vettore impiega meno tempo
- n>k: se andiamo a graficare le curve dei tempi di esecuzione dei due algoritmi notiamo che al crescere di n QuickSort farà molte più operazioni di Counting Sort.

In generale, maggiore è k, maggiore è il minimo valore di n per cui Counting Sort risulta più efficiente di QuickSort.

3.1 Analisi dei tempi di esecuzione

Vengono riportati sul grafico gli andamenti dei tempi di esecuzione relativi a QuickSort e Counting Sort.



Andando ad analizzare nel dettaglio il grafico si può affermare che per n minore di 11500 QuickSort è più efficiente di Counting Sort, per valori di n maggiori Counting Sort è più efficiente di QuickSort.