QuickSort e RadixSort
esame Algoritmi e strutture dati

Farina Giorgio - Mat. M63/000861

27 giugno 2019

Indice

Capitolo 1

Quicksort

1.1 L'algoritmo di ordinamento Quicksort

1.1.1 Descrizione dell'algoritmo: un approccio "Divide et impera"

Quicksort è un algoritmo di ordinamento basato sul confronto che utilizza un approccio divide et impera.

In particolare tramite la funzione "Partition" divide ricorsivamente il vettore in due sotto vettori con rispettivamente elementi minori e maggiori di un Pivot.

In questo elaborato il Pivot sarà l'ultimo elemento del vettore.

Il caso base è il caso in cui un sottovettore è costituito da un elemento.

Il "backward" della ricorsione, ovvero la combinazione delle soluzioni, richiede una complessità costante, in quanto dopo l'ultima partition il vettore è ordinato sul posto.

1.1.2 Ordinamento sul posto

La funzione Partition che effettua la partitione del vettore in due sottoarray è implementata in modo tale da operare stesso sul vettore di partenza.

1.1.3 Codice in Java

Nel list.2.1 è stato riportato il codice Java di Quicksort.

```
1  package Algoritmi;
2
3  public class QuickSort {
4    public static void quickSort(int A[], int p, int r)
5    {
6      int q;
7      if(p<r)
8      {
9        q = partition(A,p,r);
10       quickSort(A,p,q-1);
11       quickSort(A,q+1,r);
12     }
```

```
13    }
14    private static int partition(int A[], int p, int r)
15    {
16      int x=A[r];
17      int i=p-1;
18      for (int j=p; j<=r-1; j++)
19      {
20        if(A[j]<=x)
21        {
22          i++;
23          exchange(A[i],A[j]);
24        }
25      }
26      exchange(A[i+1],A[r]);
27    return i+1;
28    }
29
30    private static void exchange(int e1, int e2)
31    {
32      int temp;
33      temp=e1;
34      e1=e2;
35      e2=temp;
36    }
37
38  }
```

Listing 1.1: Quicksort in codice Java

1.1.3.1 La funzione Partition e la sua correttezza

La funzione partition ha come paramentri di ingresso, il vettore che deve partizionare A, e i due indici p e r. Si ha quindi informazione anche sul pivot che sarebbe l'elemento A[r].

Si è voluto implementare un algoritmo iterativo che ad ogni passo rispetti il seguente invariante:

Ad ogni passo k il sottovettore di A su cui si sta operando al passo k deve essere l'unione di quattro sotto-aree mutuamente esclusive che rispettino le seguenti proprietà:

- un'area con elementi minori o uguali di A[r]
- un'area con elementi maggiori di A[r]
- un'area costituita dagli elementi non ancora analizzati
- un'area che contiene il pivot.

Affinchè l' invariante sia rispettata saranno utilizzati due ulteriori indici i e j per delimitare le aree nel modo seguente: - A[p]...A[i] è un sottoarray con elementi minori o uguali di A[r]

- A[i+1] A[j-1] un'area con elementi maggiori di A[r]

Affinchè l'invariante sia rispettata prima della prima iterazione le due aree devono essere nulle: - (i-p) +1=0 implica che l'indice i sarà inizializzato a (p-1)

- (j-1)- (i+1) +1=0 dunque j deve essere inizializzato a i+1, ovvero a p. Inoltre j potrà essere incrementato fino a r-1 in modo tale che il pivot A[r] avrà partizionato tutti i restanti elementi di A.

Affinchè l'invariante sia rispettata durante il ciclo: L'elemento di posto j (il primo non analizzato) viene confrontato con il pivot.

Se è maggiore, è sufficiente incrementare la barriera tra seconda e terza area, ovvero incrementare j, che è fatto dal for automaticamente.

Se è minore o uguale, deve essere spostato nella prima area. Per farlo, si devono fare le seguenti operazioni:

- Si incrementa i (si sposta barriera tra prima e seconda area)
- Si scambia A[i] con A[j]
- Si incrementa j (si sposta barriera tra seconda e terza area)

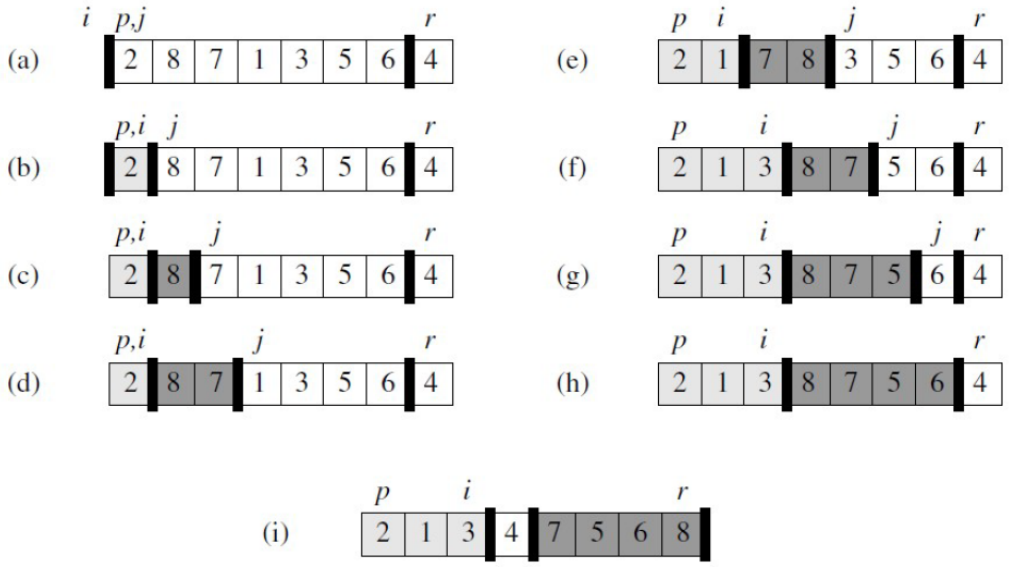


Figura 1.1: Esempio di Partition

1.2 Esecuzione di Quicksort nel caso peggiore

1.2.1 La compessità nel caso peggiore

E' stato individuato un caso in cui l'algoritmo si potrebbe comportare nel caso peggiore, ovvero quando il vettore è ordinato o contrordinato

In questo caso essendo un algoritmo ricorsivo possiamo calcolarci la complessità computazionale con la ricorrenza che espreme il tempo di esecuzione in funzione dello stesso tempo di esecuzione in funzione di una dimensione minore.

Nel caso preso in esame, è pari a:
T(n) = T(n-1) + T(0) + Θ(n) = T(n-1) + Θ((n)
Con l'albero di ricorrenza si scopre che questa è pari alla somattaria dei primi n numeri naturali e quindi è quadratica.

Successivamente tramite il metodo di sostituzione si può dimostrare che effettivamente il tempo di esecuzione nel caso peggiore appartiene a un insieme di funzioni limitato asintoticamente sia superiormente che inferiormente, per n sufficentemente grande, da una funzione n^2 a meno di una costante moltiplicativa.
T(n)=$\Theta(n^2$)

1.2.2 Esecuzione del caso peggiore

Come si può osservare dalla fig3.1 sono state individuate le due funzioni quadratiche che limitano asintoticamente il tempo di Quick Sort nel caso peggiore.
Le due costanti moltiplicative sono c1= 1.01 e c2= 0.07.

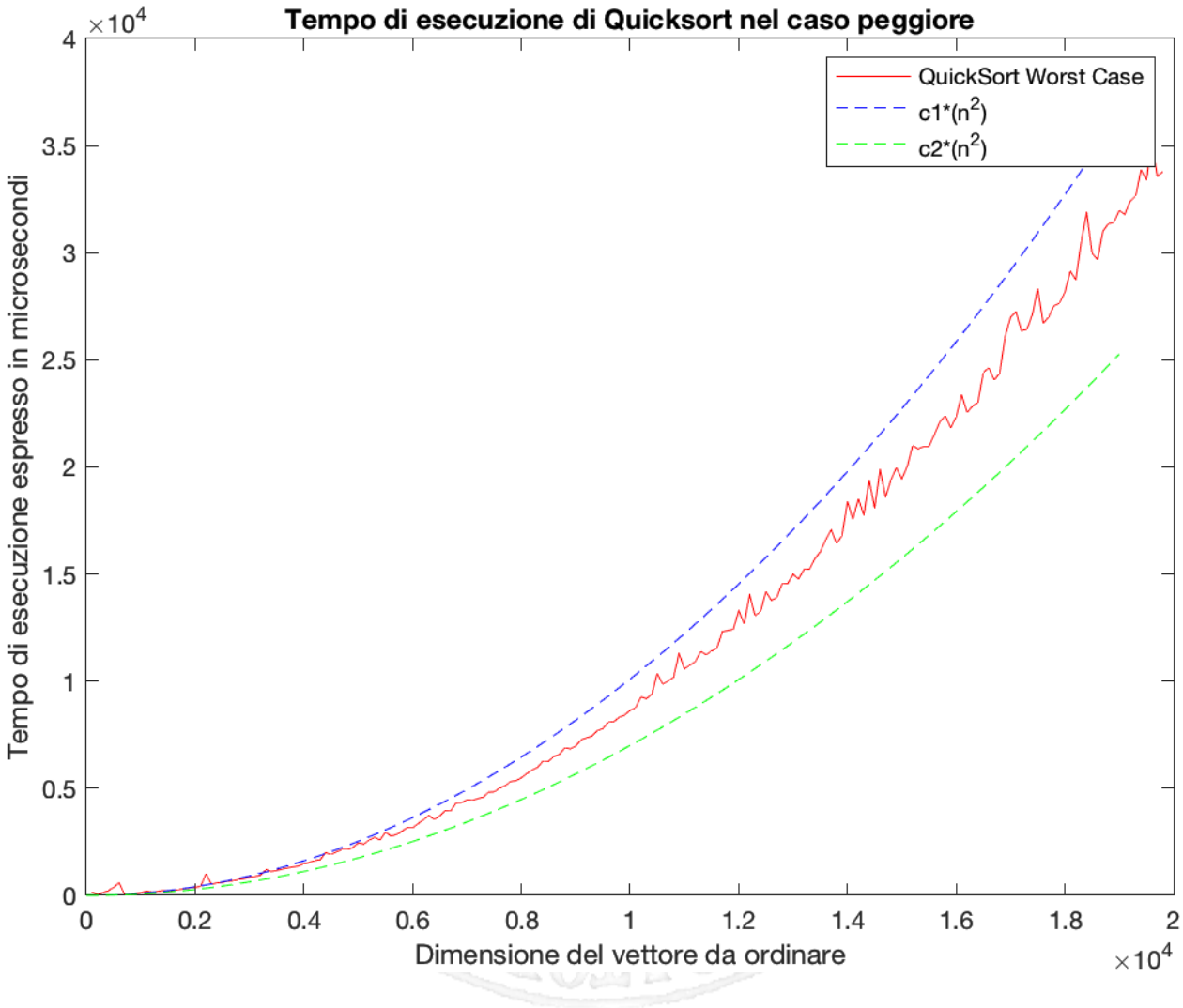


Figura 1.2: Tempo di esecuzione di QuickSort nel caso peggiore

```
plot(testWorstQuickSort.dimensione,testWorstQuickSort.tempo,'-r');
x = 0:19000;
y1 = 1.01*(x.^2)/10^4;
y2 = 0.07*(x.^2)/10^3;
hold on;
plot(x,y1,'--b',x,y2,'--g');
title("Tempo di esecuzione di Quicksort nel caso peggiore")
xlabel("Dimensione del vettore da ordinare");
ylabel("Tempo di esecuzione espresso in microsecondi")
legend("QuickSort Worst Case","c1*(n^2)","c2*(n^2)");
```

Listing 1.2: Codice in Matlab

1.3 Esecuzione di Quicksort nel caso caso medio

1.3.1 La compessità nel caso medio

Se il vettore non è ordinato, si possono verificare altri casi, tutti che hanno complessità del tipo O(nlg(n)) o Θ(nlg(n)).

Infatti, se si avesse un partizionamento completamente bilanciato ad ogni iterazione, ossia quando il partizionamento produce un sottoproblema con $\lfloor n/2 \rfloor$ elementi e uno con $\lceil n/2 \rceil$-1 elementi, si avrebbe il caso migliore il cui tempo di esecuzione sarebbe:

$T(n) \leq 2T(n/2) + \Theta(n)$

Esso ha una soluzione immediata grazie al caso 2 del teorema dell'esperto, cioè

$T(n) = O(nlg(n))$

Se, invece, si considera una partizione molto sbilanciata, dove ad ogni iterazione si hanno due sottoproblemi la cui dimensione è sempre nel rapporto 9:1, utilizzando l'albero di ricorrenza si dimostra ancora una volta che il tempo di esecuzione è

$T(n) = O(nlg(n))$

Il tempo medio di esecuzione di Quicksort è più vicino al caso migliore che non al caso peggiore.

Nel caso medio, tipicamente partizioni buone e cattive si alternano.

Intuitivamente, se si alternano casi migliori e casi peggiori, il tempo di esecuzione complessivo è quello del caso migliore. Infatti, il costo Θ(n-1) della partizione cattiva viene assorbito nel costo Θ(n) della partizione buona, e la partizione risultante è buona. Ciò vuol dire che avere una partizione cattiva e poi una buona equivale ad averne una sola buona.

1.3.2 Esecuzione nel caso medio

Si mostra in questo paragrafo che le prestazioni di QuickSort di vettori costituiti da elementi generati random ha nel caso medio complessità $T(n) = \Theta(nlg(n))$.

L'algoritmo è stato testato nello stesso modo in cui è stato testato il caso peggiore, ma questa volta si genereranno vettori costituiti da elementi casuali, e inoltre poichè questa volta non si troverà quasi mai nel caso peggiore, non abbiamo problemi di stack overflow (perchè il numero di chiamate ricorsive è molto minore) e quindi si è pensato di aumentare la dimensione massima a 70000.

Il passo è stato settato a 500 e il numero di prove all'interno di un passo a 10 per poi fare la media.

In fig. 3.1 sono state riportate sia le due funzioni che delimitavano superiormente e inferiormenteil tempo di esecuzione nel caso peggiore sia il tempo di esecuzione del caso medio con le rispettive funzioni che le delimitano asintoticamente.

In particolare è le due costanti moltiplicative per nlog(n) sono c3= 0.5 e c4= 0.6.

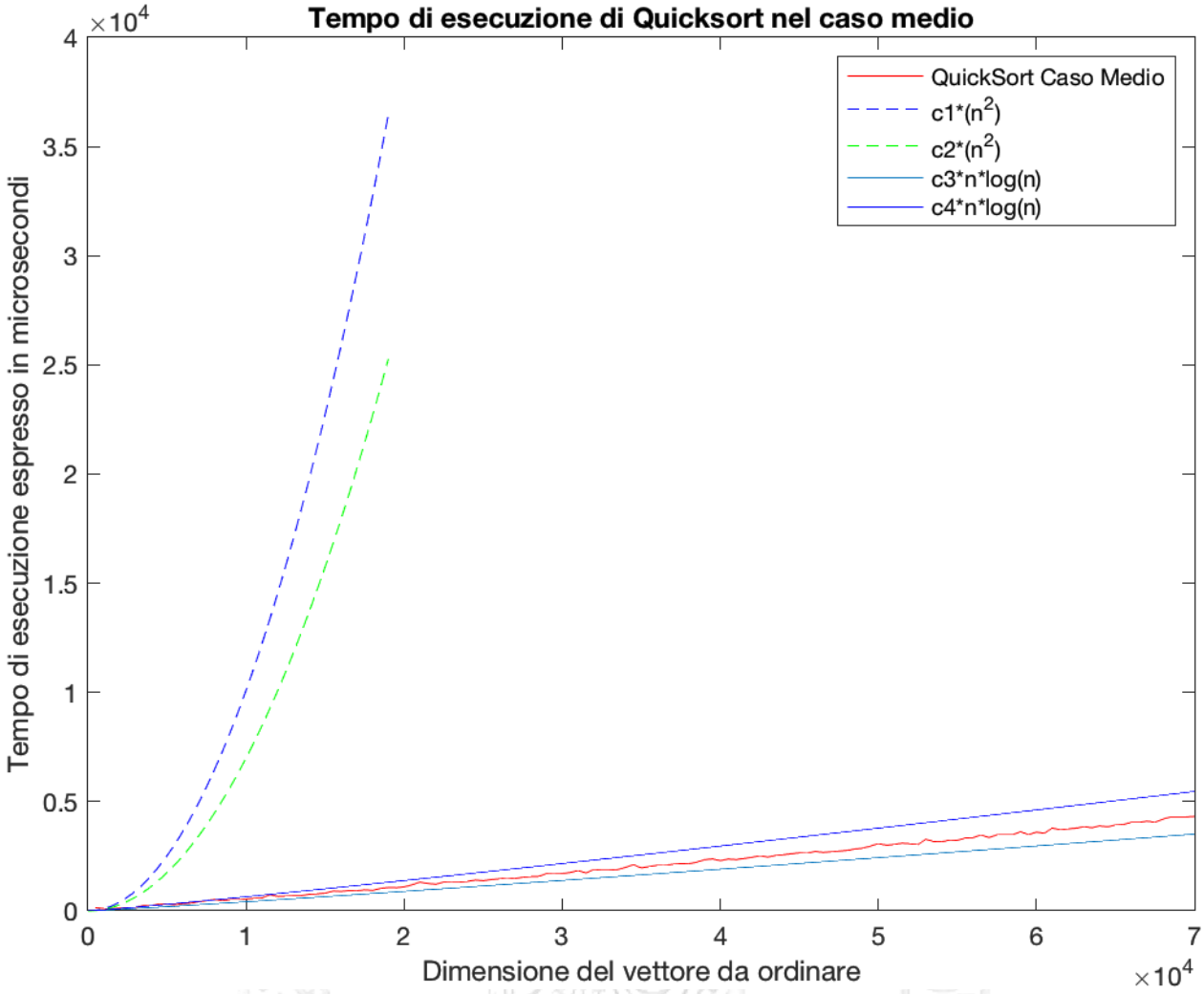


Figura 1.3: Tempo di esecuzione di QuickSort nel caso peggiore

```
plot(testAvgQuickSort.dimensione,testAvgQuickSort.tempo,'-r');
x = 0:19000;
x1 = 0:70000;
y1 = 1.01*(x.^2)/10^4;
y2 = 0.07*(x.^2)/10^3;
y3 = 0.5*10^(-2)*x1.*log(x1);
y4 = 0.6*10^(-2)*x1.*log(x1);
hold on;
plot(x,y1,'--b',x,y2,'--g',x1,y3,x1,y4,'-b');
```

```
10 title("Tempo di esecuzione di Quicksort nel caso medio")
11 xlabel("Dimensione del vettore da ordinare");
12 ylabel("Tempo di esecuzione espresso in microsecondi")
13 legend("QuickSort Caso Medio","c1*(n^2)","c2*(n^2)","c3*n*log(n)","c4*n
       *log(n)");
```

Listing 1.3: Codice in Matlab

Capitolo 2

RadixSort

2.1 Radix Sort: un algoritmo lineare

2.1.1 Descrizione dell'algoritmo

Il Radix Sort è un algoritmo di ordinamento che assume di lavorare con vettori i cui elementi hanno d cifre.

Gli elementi sono ordinati a partire dalla cifra meno significativa, usando un algoritmo stabile (come il Counting Sort).

La proprietà di stabilità dell'algoritmo Counting Sort è essenziale.

Stabilità significa che se ci sono più occorrenze dello stesso valore, l'ordine che si aveva prima dell'applicazione dell'algoritmo stabile sarà preservato anche dopo l'applicazione.

Quindi la proprietà di stabilità ci permette di preservare scelte di ordinamento fatte in precedenza (ad esempio su una cifra meno significativa di quella corrente) da un altro algoritmo su attributi diversi da quelli correntemente analizzati .

2.1.2 Counting Sort

Counting Sort è un algoritmo stabile.

L'algoritmo Counting Sort come tutti gli altri algoritmi lineari per garantire una complessità computazionale inferiore rispetto agli altri algoritmi di ordinamento basati sul confronto deve assumere delle ipotesi sull'insieme dei valori in ingresso.

In particolare l'ipotesi a cui fa riferimento Counting Sort è che i numeri interi in ingresso appartengano a un intervallo di valori da 0 a k per un totali di k+1 valori possibili.

Senza entrare nel merito dell'algoritmo, è facile verificare che il suo tempo di esecuzione appartiene all'insieme di funzioni limitate strettamente da una funzione lineare all'ordine predominante tra k e la dimensione n dell'array in ingresso.

E' un algoritmo che non ordina sul posto.

2.2 Analisi di complessità

2.2.1 Analisi di complessità

Indicando con d le cifre significative, poichè Radix Sort consiste nell' applicare in modo iterativo dalla cifra meno significativa alla più significativa Counting Sort (su tutti gli elementi del vettore), la sua complessità computazionale è esprimibile secondo questa formula:

T(n)= dΘ(n+k)

In questo elaborato si è voluto approfondire la seguente intuizione: sfruttare come ipotesi il vincolo intrinseco al calcolatore che ci impone di rappresentare tutti i numeri interi su b bit.

Ovviamente applicare Counting Sort su valori rappresentabili su b bit richiede un tempo di esecuzione elevatissimo poichè i possibili valori rappresentabili su b bit sono 2^b-1 oltre che un enorme spreco di memoria.

Allora l'idea di radixSort è proprio quella di partizionare questi bit in dei gruppi (le cifre) per poi applicare su ogni gruppo di bit Counting Sort.

Supponendo di utilizzare r bit per ogni cifra, si hanno in questo modo b/r gruppi di r bit.

Quindi il tempo di esecuzione è possibile esprimerlo in questo modo:

T(n)= ($\frac{b}{r}\Theta$(n+2^r).

Allora r è l'unica variabile da poter modificare affinchè si abbia una complessità computazionale lineare.

Per garantire un tempo di esecuzione lineare r deve essere definita in questo modo (ricordando che r non può essere maggiore di b) :

- se b è più piccolo di lg n allora si dovrebbe scegliere r=b

- se b è maggiore di lg n allora si dovrebbe scegliere r= lg n. In questo caso se si sceglie r<lg n comunque si ha una complessità computazionale pressochè lineare ma la costante moltiplicativa b/r cresce. Invece se si sceglie b>lg n la complessità computazione aumenta velocemente.

2.3 Implementazione

2.3.1 Descrizione dell'implementazione

Si ipotizza che i numeri interi del vettore non abbiano limitazioni se non quelle dovute alla rappresentazione sul calcolatore.

In questo caso java rappresenta gli interi su 32 bit.

r è un parametro di ingresso.

k è la base con cui si sta lavorando, quindi se scelgo ad esempio r=4 bit è come se lavorassi in base 16.

Ad esempio si supponga di voler individuare la quarta cifra di a :

- individuo la quarta cifra in base k del valore a in questo modo:

($\frac{a}{k^3}$)mod k

Quindi CountingSort avendo come parametri di ingresso sia exp = k^i sia la base k ordinerà solo secondo la cifre (i+1)-esima i valori del vettore in ingresso.

Si è pensato di mettere come condizione di uscita del ciclo for la condizione in cui exp ha una differenza, in termini di numero di bit, dall'oveflow minore di r. Se exp dovesse superare il massimo numero rappresentabile diverrà uguale a zero o minore di zero (così è gestito l'overflow di interi in java).

2.3.2 Codice in Java

```
1
2       static void radixsort(int arr[], int n, int r)
3       {
4         int k;
5         k = 1 << r;
6         System.out.println(k);
7         for (int exp=1;exp>0;exp*=k) {
8           countSort(arr, n, exp,k);
9         }
10      }
```

Listing 2.1: Radix Sort

```
1
2         static void countSort(int arr[], int n, int exp, int k)
3       {
4           int output[] = new int[n]; // output array
5           int i;
6           int count[] = new int[k];
7           Arrays.fill(count,0);
8
9
10          for (i = 0; i < n; i++)
11              count[ (arr[i]/exp)%k ]++;
12
13
14          for (i = 1; i < k; i++)
15              count[i] += count[i - 1];
16
17
18          for (i = n - 1; i >= 0; i--)
19          {
20              output[count[ (arr[i]/exp)%k ] - 1] = arr[i];
21              count[ (arr[i]/exp)%k ]--;
22          }
23
24
25          for (i = 0; i < n; i++)
26              arr[i] = output[i];
27      }
```

Listing 2.2: Counting Sort

2.3.3 r sub-ottimale

Per scegliere r in modo ottimale non si calcolerà il logaritmo binario di n. Si è pensato che potrebbe essere troppo oneroso o comunque, facendo parte della libreria Math, non si conosce a priori la sua complessità computazionale.

Piuttosto ad r verrà assegnata la posizione del bit più alto di n ("Integer.toBinaryString(arr.length).length()-1").

In questo modo r sarà sempre minore o uguale al logaritmo binario di n, ovviamente non si ha il caso ottimo, ma la differenza principalmente consisterà in un incremento della costante moltiplicativa della funzione lineare, si vedrà che comunque l'algoritmo si comporta piuttosto bene.

2.4 Casi di test

2.4.1 Analisi Radix Sort con r sub-ottimo

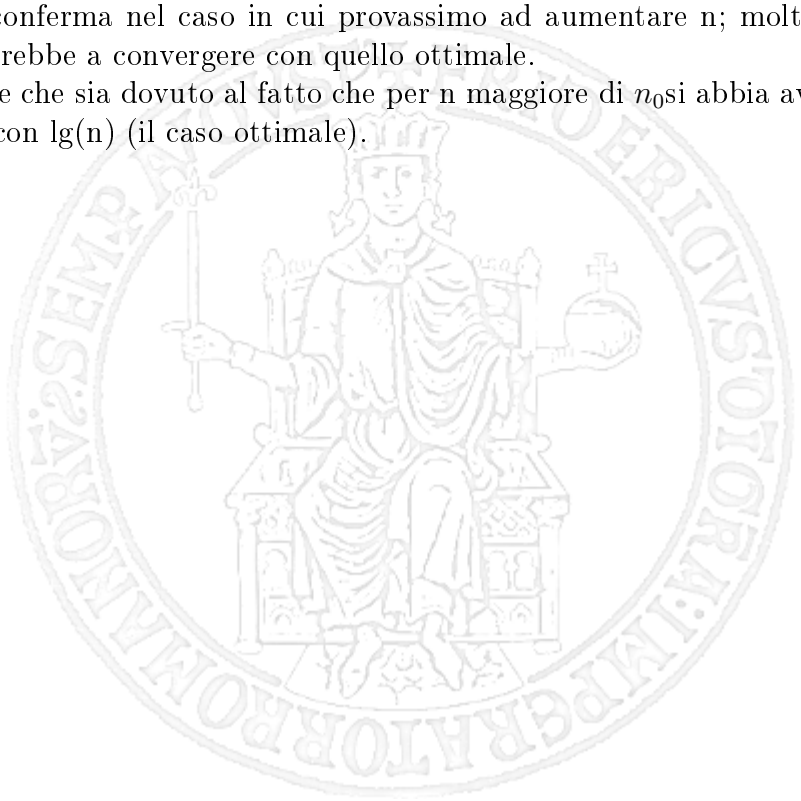
Come è possibile osservare in fig. 3.1 sono stati individuati due funzioni lineari con costanti moltiplicative rispettivamente pari a 425 e 300 che da un certo n_0=3,5*10^4 in poi limitano il tempo di esecuzione .

Il perchè si abbia un comportamento peggiore (ovviamente rapportato alla dimensione) si pensa sia dovuto all'approssimazione che si è fatta di r.

Per dimensioni di n "piccole" la costante moltiplicativa b/r (dove r potrebbe diminuire con velocità logaritmica) avrà un peso relativo maggiore sul tempo di esecuzione (che dovrebbe essere lineare).

Tale ipotesi ha conferma nel caso in cui provassimo ad aumentare n; molto probabilmente il caso sub-ottimo andrebbe a convergere con quello ottimale.

E' poco probabile che sia dovuto al fatto che per n maggiore di $n_0$si abbia avuto sempre il caso in cui r coincidesse con lg(n) (il caso ottimale).



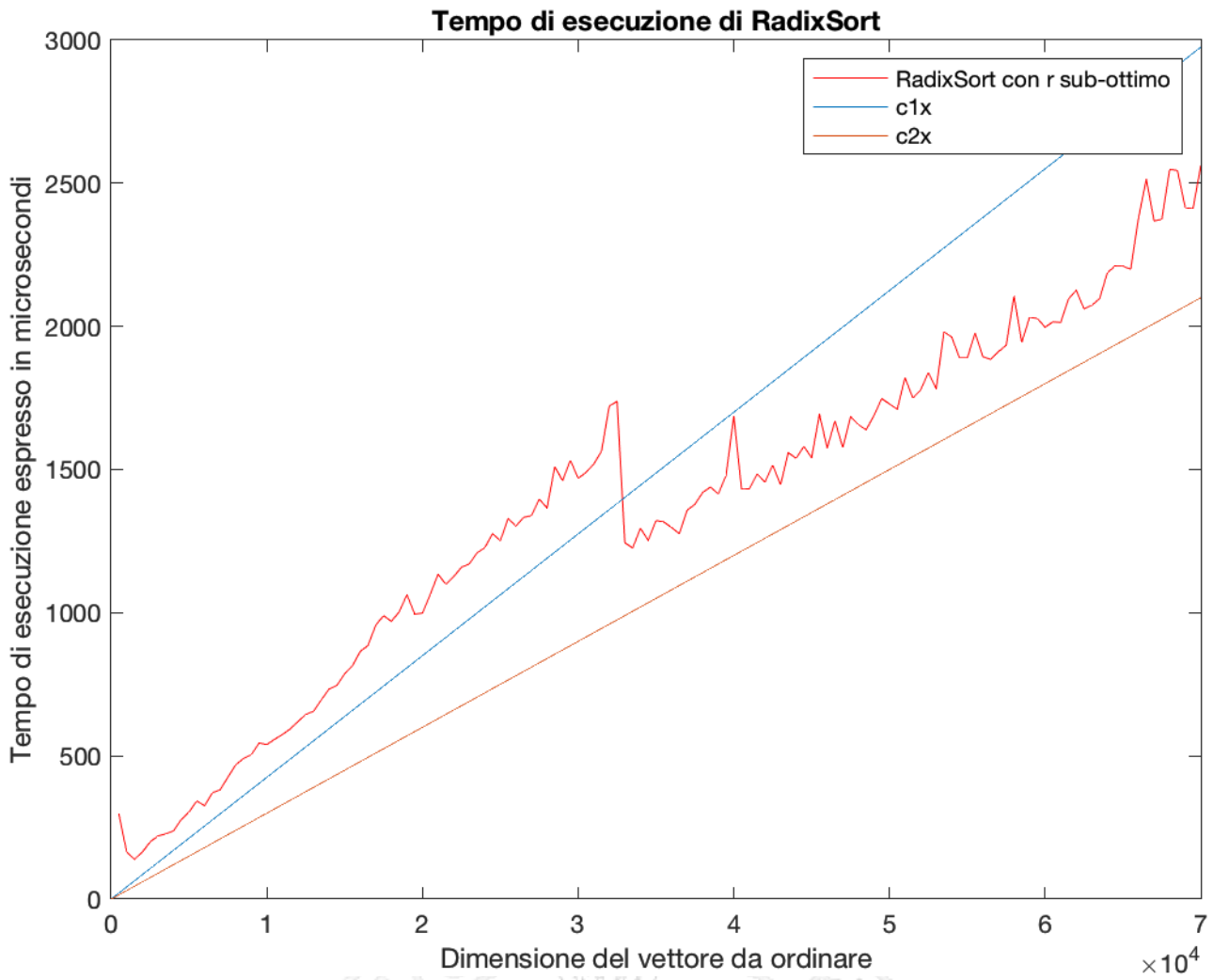


Figura 2.1: Tempo di esecuzione di QuickSort nel caso peggiore

2.4.2 Radix Sort r uguale a 1

Come si poteva intuire dalla formula della complessità scegliendo r pari a un numero finito e non dipendente dalla dimensione, in particolare in questo caso pari a 1, quello che si fa non è altro che aumentare la costante moltiplicativa della crescita lineare del tempo di esecuzione.

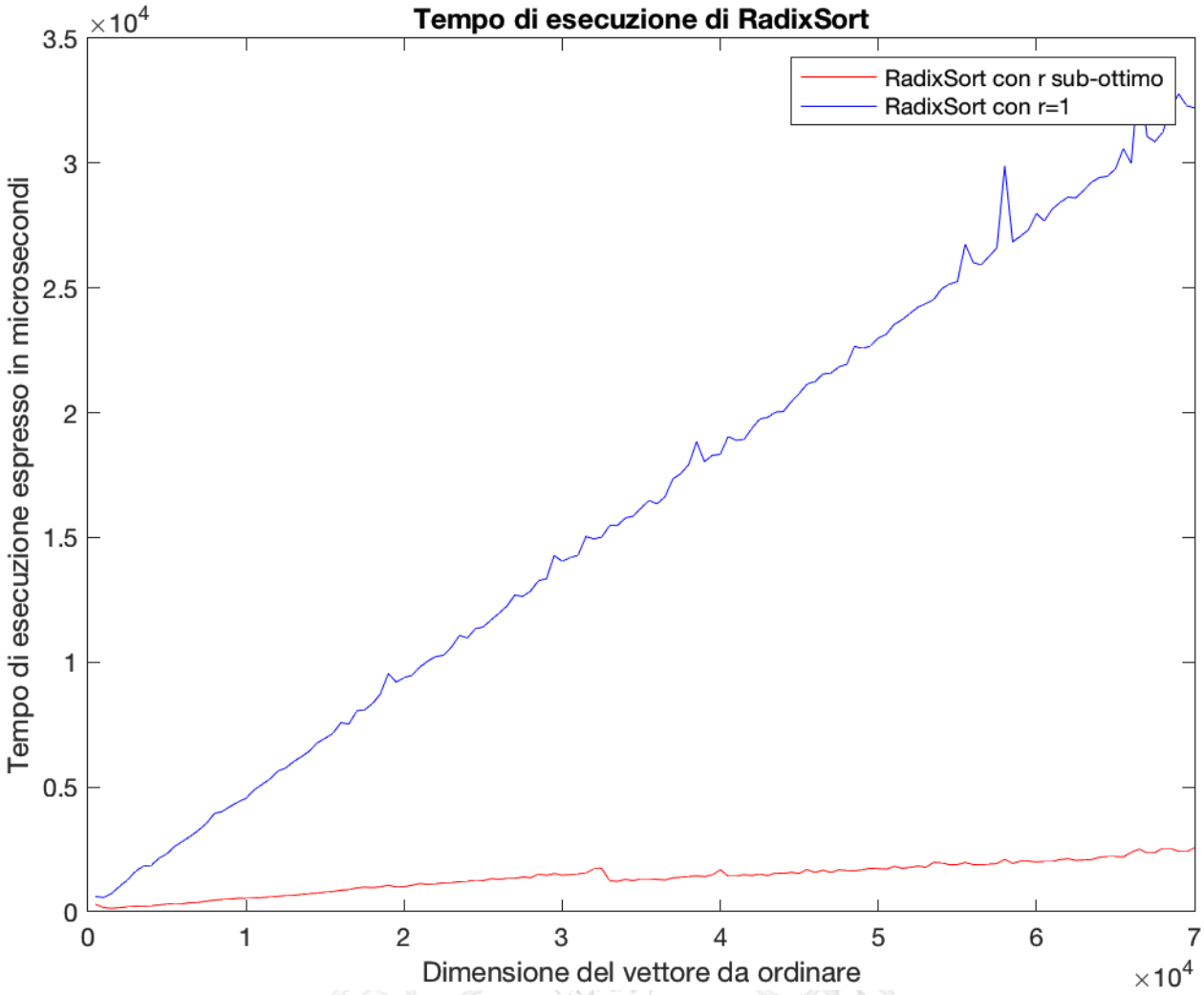


Figura 2.2: Tempo di esecuzione di QuickSort nel caso peggiore

2.4.3 Radix Sort con r maggiore del caso sub-ottimo di 10 unità

Invece per r maggiore del caso sub-ottimo di sole 10 unità il tempo di esecuzione aumenta in modo non lineare. Si osservi che è stato scelto un r maggiore del caso sub-ottimo di 10 unità per ogni dimensione, quindi r in questo caso dipende dalla dimensione.

Questo esempio riportato in Fig. 2.3quindi mostra per ogni dimensione del vettore in ingresso cosa succede se si scegliesse una r anche solo di 10 unità più grande.

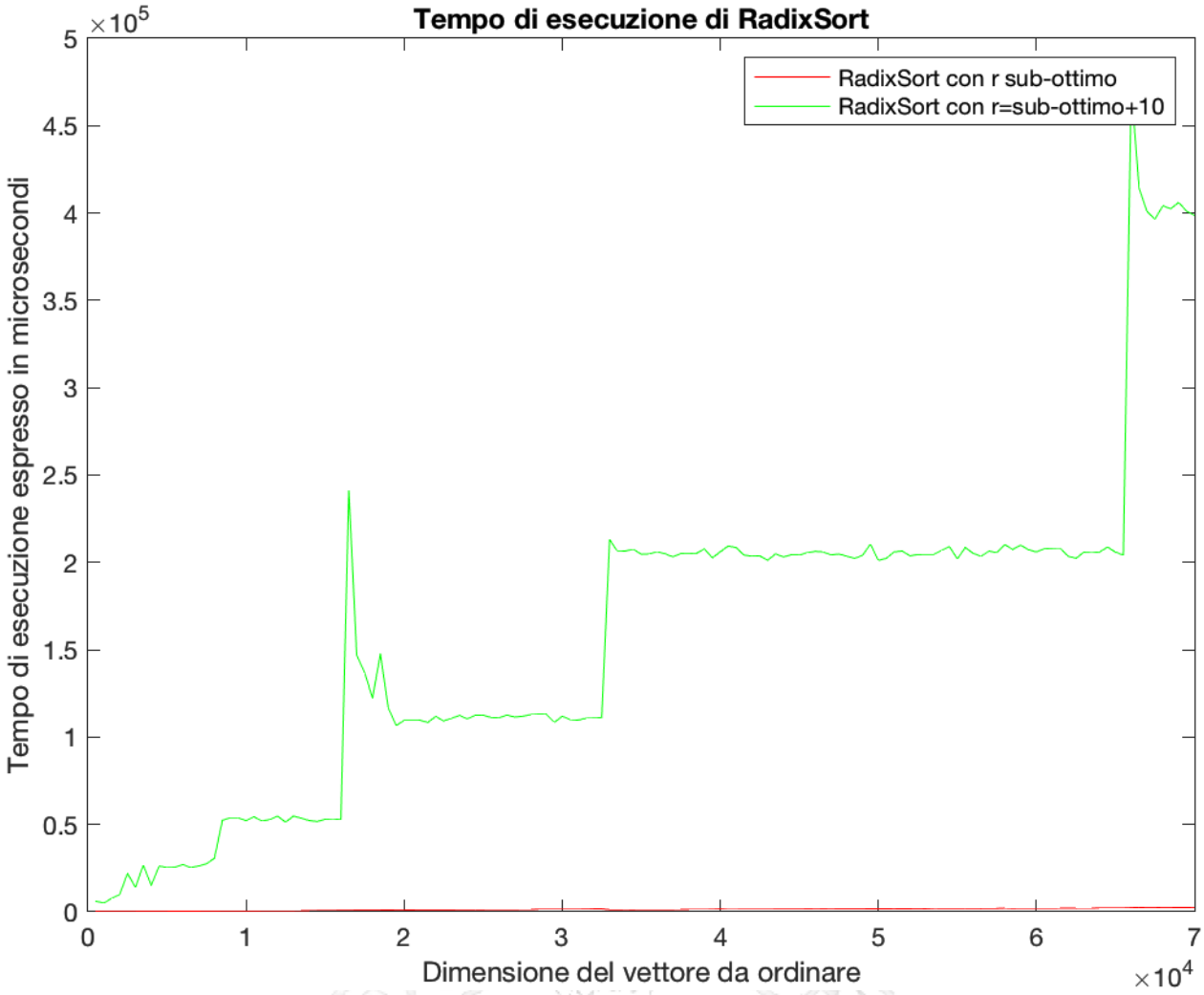


Figura 2.3: Tempo di esecuzione di QuickSort nel caso peggiore

2.4.4 Confronto tra i casi di test

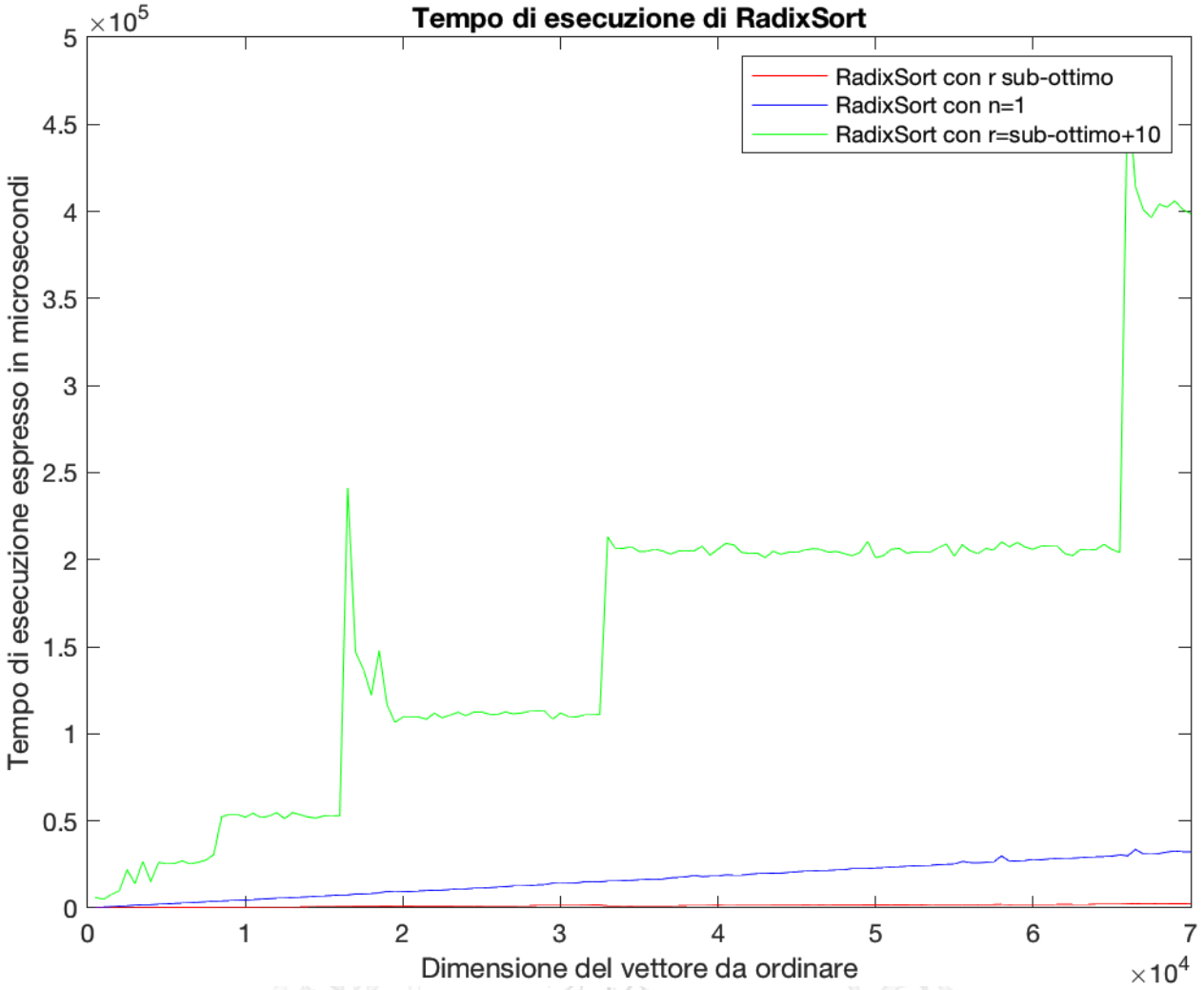


Figura 2.4: Tempo di esecuzione di QuickSort nel caso peggiore

Capitolo 3

Confronto tra QuickSort e RadixSort

3.1 Confronto tra QuickSort nel caso medio e RadixSort

3.1.1 Confronto dei tempi di esecuzione

Per n minore di 3.4*10^4 i tempi di esecuzione di Radix Sort e Quick Sort hanno più o meno la stessa crescita. Successivamentela RadixSort mostra la sua potenziale tendenza alla linearità.

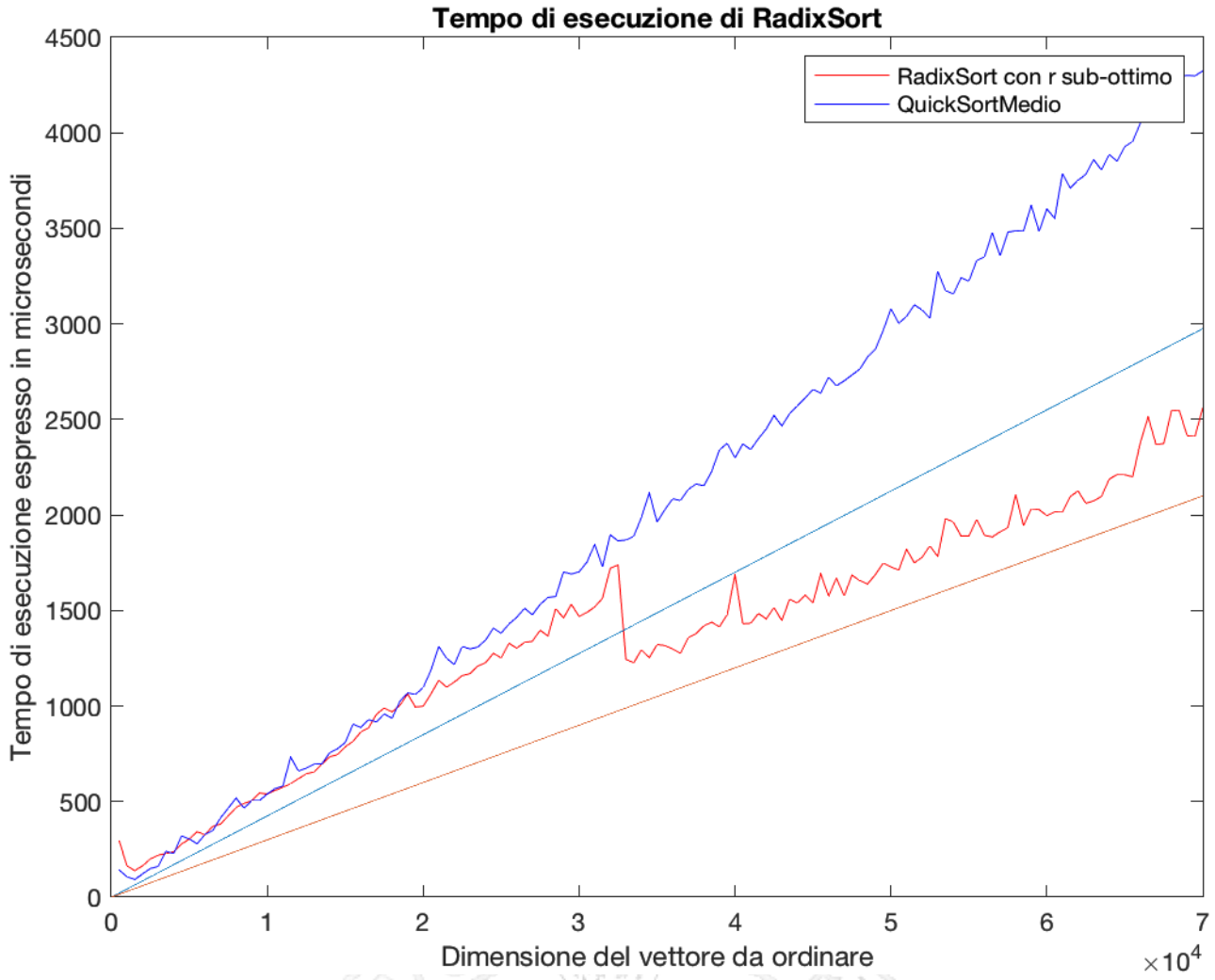


Figura 3.1: Tempo di esecuzione di QuickSort nel caso peggiore