UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

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

Elaborato **Algoritmi e strutture dati**

Heapsort

Anno Accademico 2020/21

Candidato:

Michele Maresca

matr. M63001151

Indice

Introduzione

Il presente elaborato affronta il problema dell'ordinamento, si intende cioè, trovare un algoritmo che prenda come dati di input un array di n elementi e produca in uscita una sua permutazione tale che gli elementi rispettino una relazione d'ordine. Più precisamente, "data una sequenza di *n* valori di ingresso $(a_1, a_2, a_3, \ldots, a_n)$, determinare una permutazione $(a'_1, a'_2, a'_3, \ldots, a'_n)$ della sequenza di ingresso tale che $(a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_n)$".

Le soluzioni a tale problema sono molteplici. Esse si differenziano in base alle seguenti proprietà:

- **Occupazione di memoria**: si distinguono algoritmi che ordinano sul posto, per i quali i valori sono riordinati all'interno dell'array stesso, con al più un numero costante di essi memorizzati al di fuori dell'array in ogni momento, oppure algoritmi che non ordinano sul posto, per i quali c'è la necessità di ulteriore memoria ausiliaria per eseguire l'operazione;
- **Tempo di esecuzione**: il tempo che impiega un algoritmo per restituire l'output desiderato in relazione alla dimensione del problema, ovvero alla lunghezza della sequenza da ordinare nel caso di algoritmi di ordinamento.

La particolare soluzione al problema, che sarà oggetto dell'elaborato, è l'algoritmo Heap-Sort.

Quest'ultimo si basa su una particolare struttura dati chiamata **Heap**.

Un Heap è una sequenza di elementi memorizzati in un supporto che consente l'accesso diretto agli elementi in un tempo costante caratterizzato dal fatto che gli elementi sono disposti in maniera tale da verificare la condizione:

- Max-Heap, `A[i] ≤ A[Parent(i)]` per ogni `i > 1`, nella radice c'è il valore più grande;

- Min Heap, `A[i] ≥ A[Parent(i)]` per ogni `i > 1`, nella radice c'è il valore più piccolo.

Un modo grafico per vedere la disposizione degli elementi contenuti nella struttura dati Heap è utilizzare un albero binario (quasi) completo. Per questo motivo viene fatto riferimento agli elementi presenti all'interno della struttura dati con il termine di nodi, e quindi si parlerà di nodo radice della struttura, figlio di destra di un nodo e figlio di sinistra di un nodo.

È stato associato alla struttura dati un attributo fondamentale, `heap-size[A],` che indica il numero di elementi presenti nell'Heap.

Si possono, inoltre, definire tre funzioni ausiliari, che restituiscono l'indice del figlio di sinistra, di destra e del padre di un generico nodo i-esimo. Tali funzioni sono le seguenti:

- `Left(i) = 2*i`

- `Right(i) = 2*i+1`

- `Parent(i) = ⌊i/2⌋`

Capitolo 1: Soluzione

La soluzione fa uso di tre funzioni. La prima è la **Max-Heapify**. Tale funzione prende in input l'array, e la posizione dell'elemento che deve essere analizzato. Inoltre, richiede come precondizione che il figlio di sinistra e il figlio di destra del nodo i-esimo, sulla quale è applicato l'algoritmo, debbano essere entrambi radici di un Max-Heap, e produca come post-condizione un Max-Heap con radice nella posizione in analisi. Tale funzione è illustrata nel seguente pseudocodice:

```
Max-Heapify (A, i)
l ← Left(i)
r ← Right(i)
largest ← i
If l ≤ heap_size[A] and A[l]>A[i]
    then largest ← l
If r ≤ heap_size[A] and A[r]>A[largest]
    then largest ← r
If largest ≠ i
    then Exchange A[i] ↔ A[largest]
    Max-Heapify (A, largest)
```

Il primo obiettivo della funzione è quello di individuare la posizione dell'elemento più grande tra l'elemento *i*-esimo in questione, il figlio di destra di *i* e il figlio di sinistra di *i*. Individuata la posizione essa sarà inserita nella variabile ausiliaria *largest*. Se tale elemento non è già in posizione *i*, viene effettuato uno swap tra le posizioni di *i* e *largest*, e viene effettuata una chiamata ricorsiva a Max-Heapify con input A e largest.

La seconda funzione necessaria per il funzionamento dell'algoritmo è la
Build-Max-Heap. Essa prende in input l'array, visto come sequenza generica, e produce come post-condizione lo stesso array organizzato in modo tale da soddisfare le proprietà di un Max-Heap con radice in posizione 1. Tale funzione è illustrata nel seguente pseudocodice:

```
Build-Max-Heap (A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
    do Max-Heapify (A, i)
```

Questa funzione in prima istanza rende la dimensione dell'Heap uguale a quella dell'array.
Dopodiché si considera che i nodi foglia risultano banalmente radici di un Max-Heap. Di conseguenza, per costruire iterativamente il Max-Heap con radice in posizione 1, si itera la funzione di Max-Heapify partendo dall'ultimo nodo non foglia dell'array fino a tornare alla posizione 1. L'ultimo nodo non foglia viene trovato attraverso l'istruzione `⌊length[A]/2⌋` poiché esso è il padre dell'ultimo nodo della struttura presente in posizione `lenght[A]`.

L'ultima funzione è **Heap-Sort**, che prende in input l'array da ordinare e produce in uscita l'array ordinato. Tale funzione è illustrata nel seguente pseudocodice:

```
Heap-Sort (A)
Build-Max-Heap (A)
For i ← length[A] downto 2
    Exchange A[1] ↔ A[i]
    Heap-size[A] ← heap-size[A]-1
    Max-Heapify (A,1)
```

Essa crea in prima istanza la struttura dell'Heap, tramite la chiamata di Build-Max-Heap, dopodiché viene scambiata iterativamente la radice con l'ultimo elemento dell'Heap, viene ridotta la sua dimensione e viene ristabilita la proprietà di Max-Heap tramite una chiamata di Max-Heapify.

Capitolo 2: Analisi di complessità

In questa sezione, è analizzata la complessità temporale asintotica di ciascuna delle tre funzioni sfruttate dalla Heap-Sort. Per quanto riguarda il tempo di esecuzione di Max-Heapify si considera:

- Il tempo costante per individuare il massimo tra tre valori, ovvero tra il nodo i-esimo, il suo figlio di sinistra e il suo figlio di destra;
- Un tempo per eseguire la chiamata ricorsiva sullo stesso problema con dimensione minore.

Per quanto riguarda il tempo di esecuzione della chiamata ricorsiva, bisogna realizzare un'analisi più accurata. Allo scopo di considerare il caso peggiore, si ricava il caso in cui la dimensione del sotto-problema da risolvere durante la ricorsione sia la più grande possibile. Si distinguono due casi:
1. Un caso in cui l'albero con radice in i sia completo, quindi con ultimo livello pieno;
2. Un caso in cui l'albero con radice i non sia completo e presenti l'ultimo livello pieno a metà.

2.1 Ultimo livello pieno

Si indica con n la dimensione del problema originario, con m la dimensione del sotto-albero con più elementi e h l'altezza del nodo considerato nel problema originario.

$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \qquad (2.1)$$

$$m = \sum_{i=0}^{h-1} 2^i = 2^h - 1 \qquad (2.2)$$

$$\frac{m}{n} = \frac{2^h - 1}{2^{h+1} - 1} = \frac{2^h - 1}{2 \cdot 2^h - 1} \overset{h\to\infty}{\rightarrow} \frac{1}{2} \qquad (2.3)$$

In questo caso, il fattore di riduzione del sotto-problema rispetto al problema originario di partenza, nel caso peggiore, ovvero facendo tendere $h \to \infty$, è $\frac{1}{2}$ quindi il tempo di esecuzione per risolvere il sotto-problema sarebbe, come limite asintotico superiore, $O(^n/_2)$.

2.2 Ultimo livello pieno a metà

Si considera il secondo caso. La situazione sarà quindi descritta in tal modo:

$$n = (2^h - 1) + (2^{h-1} - 1) + 1 = 3 \cdot 2^{h-1} - 1 \qquad (2.4)$$

$$m = (2^h - 1) \qquad (2.6)$$

$$\frac{m}{n} = \frac{2^h - 1}{3 \cdot 2^{h-1} - 1} = \frac{2 \cdot 2^{h-1} - 1}{3 \cdot 2^{h-1} - 1} \overset{h\to\infty}{\rightarrow} \frac{2}{3} \qquad (2.6)$$

Facendo tendere $h \to \infty$ il rapporto tenderà a $\frac{2}{3}$, allora la complessità ha come limite superiore asintotico $O({}^{2n}/_{3})$. Questo limite rappresenta il Worst Case. Si ottiene in questo modo la ricorrenza che descrive il tempo di esecuzione della funzione Max-Heapify, $T(n) \leq T({}^{2n}/_{3}) + \theta(1)$, la quale può essere risolta utilizzando il metodo dell'esperto, dato che rientra nel secondo caso. Si ottiene, in questo modo, che la complessità della Max-Heapify è $O(\lg(n))$.

Per quanto riguarda la funzione Build-Max-Heap, è opportuno notare che essa evoca iterativamente $\lfloor \texttt{length[A]/2} \rfloor = \lfloor \texttt{n/2} \rfloor$ volte la funzione Max-Heapify, quindi si potrebbe pensare che la complessità asintotica sia *n* volte la complessità asintotica di Max-Heapify $O(n\ lg(n))$. Quest'ultimo, però, non è un limite stretto, poiché solo una volta la Max-Heapify è invocata sulla radice ed è applicata quindi a sequenze di *n* elementi, le altre volte è applicata su nodi che non sono radici di alberi con *n* elementi, ma ne presenteranno un numero inferiore. La complessità asintotica della Max-Heapify dipende dall'altezza dell'albero con radice in *i* al quale è applicata.

Un Heap di *n* elementi ha altezza $\lfloor \log(n) \rfloor$ e ha al più $\lceil n/2^{h+1} \rceil$ elementi di altezza h.

Esso presenta $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ foglie (nodi di altezza 0) e un numero di nodi di altezza 1 pari alla metà del numero di foglie, ovvero $\lceil n/2^2 \rceil$, quindi dato che Max-Heapify ha una complessità O(h) per nodi di altezza h, la complessità della funzione Build-Max-Heap è:

$$T(n) = \sum_{h=1}^{\lfloor lg\ n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=1}^{\lfloor lg\ n \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(2n) = O(n) \qquad (2.7)$$

Si può concludere che la complessità di Heap-Sort è $O(n\ lg(n))$, data dalle *n* – 1 invocazioni di Max–Heapify di complessità $O(lg(n))$. che dominano la complessità $O(n)$. della Build-Max-Heap.

Capitolo 3: Simulazione

Nell'ultima sezione è stato implementato l'algoritmo Heap-Sort, mediante il linguaggio di programmazione C++, allo scopo di testare la sua efficienza tramite dei casi di test, in modo tale da poter verificare sperimentalmente i dati teorici ottenuti. Il codice della funzione Heap-Sort è descritto da *Codice 3.1*.

```cpp
void heapsort(std::vector<int> &A){
    buildmaxheap(A);
    int heapsize=int(A.size());
    for(int i=int(A.size())-1;i>=1;i--){
        int key=A[0];
        A[0]=A[i];
        A[i]=key;
        heapsize=heapsize-1;
        maxheapify(A,heapsize,0);
    }

}
```

Codice 3.1

Il Codice 3.2 descrive la funzione Build-Max-Heap.

```cpp
void buildmaxheap(std::vector<int> &A){
    int heapsize=int(A.size());
    for(int i=int((A.size()-1)/2);i>=0;i--){
        maxheapify(A,heapsize,i);
    }
}
```

Codice 3.2

Il Codice 3.3 descrive la funzione Max-Heapify.

```
void maxheapify(std::vector<int> &A,int
heapsize, int index){

    int l=Left(index);
    int r=Right(index);
    int largest=index;

    if(l<heapsize and A[l]>A[index]){
        largest=l;
    }
    if(r<heapsize and A[r]>A[largest]){
        largest=r;
    }
    if(largest != index){
        int key=A[index];
        A[index]=A[largest];
        A[largest]=key;
        maxheapify(A,heapsize,largest);
    }

}
```

Codice 3.3

Infine, le funzioni ausiliare per ottenere il padre, il figlio sinistro e il figlio destro di un nodo sono descritte da Codice 3.4

```
int Left(int index){
    return index*2;
}
```

Codice 3.4 (a)

```
int Right(int index){
    return index*2+1;
}
```

Codice 3.4 (b)

```
int Parent(int index){
    return int(index/2);
}
```

Codice 3.4 (c)

È stato implementato un Main nel quale è stato generato un vettore di numeri interi casuali tra 0 e 10000 mediante la funzione `std::uniform_distribution` fornita dalla standard library. Essa produce valori interi casuali i, distribuiti uniformemente sull'intervallo chiuso [a, b], cioè distribuiti secondo la funzione di probabilità discreta $P(i/a,b) = \frac{1}{b-a+1}$.
È stata utilizzata la funzione `std::chrono::steady_clock::now` fornita dalla standard library, in particolare, è una funzione membro della classe `std::chrono::steady_clock` dell'header `<chrono>`. Questo clock non è correlato all'ora dell'orologio da parete (ad esempio, può essere l'ora dall'ultimo riavvio) ed è più adatto per misurare gli intervalli. La funzione restituisce un time_point che rappresenta il valore corrente dell'orologio.

Sono stati effettuati inizialmente 27 esperimenti con dimensioni crescenti del vettore da ordinare, in un range che va da 5 elementi fino a 100.000.000 elementi. I tempi di esecuzioni, espressi in millisecondi, con la relativa dimensione del vettore ordinato sono espressi in Tabella 3.1.

DIMENSIONE VETTORE	**TEMPO DI ESECUZIONE HEAPSORT-IMPLEMENTATO(ms)**
5	**0,0009388**
10	**0,0017486**
50	**0,0101254**
100	**0,0226778**
500	**0,1429164**
1000	**0,3260002**
3000	**1,2024216**
5000	**2,1245464**
7000	**2,9530094**
10000	**4,1916348**
30000	**15,37134**
50000	**26,9145058**
70000	**38,5308412**
100000	**51,7215358**
300000	**167,7064158**
500000	**298,1242086**
700000	**464,2221804**
1000000	**735,5433656**
3000000	**2444,644286**
5000000	**4269,117877**
7000000	**6126,384851**
10000000	**8948,497922**
30000000	**32636,1247**
50000000	**51493,01619**
70000000	**74660,0328**
100000000	**110165,8207**

Tabella 3.1

I risultati della Tabella 3.1 sono rappresentati attraverso il Grafico 3.1.

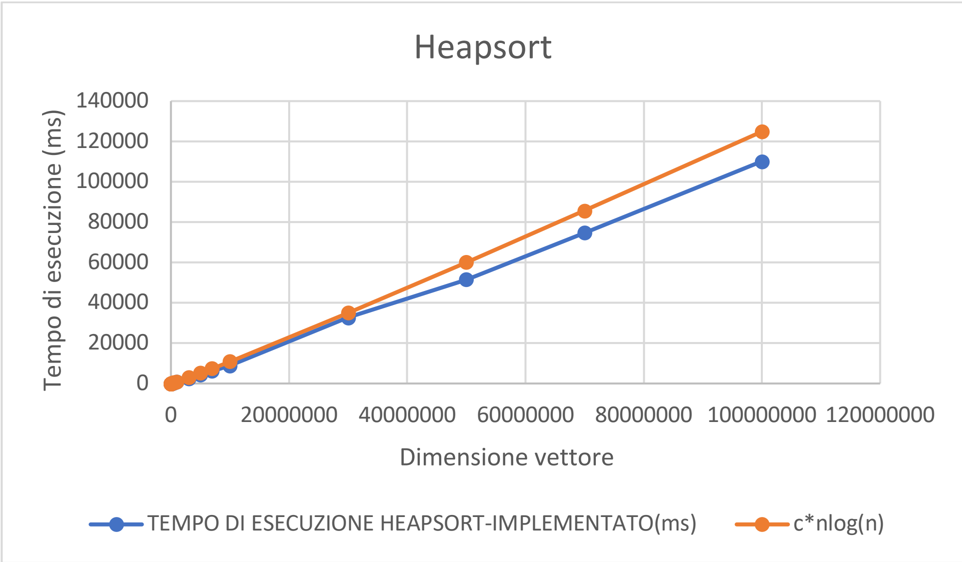

Grafico 3.1

La curva blu rappresenta la curva dei tempi di esecuzione dell'algoritmo heap-sort.
Allo scopo di ottenere una funzione analitica dei tempi di esecuzione misurati durante gli esperimenti è stata tracciata una linea di tendenza lineare, ottenendo in questo modo la funzione (3.1).

$$f(n) = 0.0011(n) - 337.85 \qquad (3.1)$$

Dove n rappresenta la dimensione del vettore da ordinare.
Risolvendo la disequazione (3.2) è stato possibile procedere nel calcolo della costante positiva c e della dimensione del vettore n_0 dalla quale è valida la definizione di notazione O.

$$f(n) = 0.0011(n) - 337.85 \leq c \cdot n \cdot \log(n) \qquad (3.2)$$

Si è ottenuto il risultato (3.3).

$$n > 0, \qquad c \geq \frac{\ln(2) \cdot (0.0011\, n - 337.85)}{n \cdot \ln(n)} \qquad (3.3)$$

Con la scelta di $n_0 = 50$ e $c = 0.000047$ è quindi verificata la definizione di notazione O.
Infatti, dal Grafico 3.1 è possibile osservare che la curva può essere maggiorata dalla funzione $c \cdot n\log(n)$, con c costante positiva scelta pari a 0.000047. Si può affermare che la complessità asintotica dell'algoritmo heap-sort è $O(n\, lg(n))$.

3.1 Confronto con Heapsort della standard library

In questa sezione viene confrontato l'algoritmo Heap-sort implementato nella sezione precedente con l'algoritmo heapsort implementato mediante le funzione fornite dalla Standard library del C++.
Le funzioni della Standard library utilizzate sono `std::make_heap` e `std::sort_heap`.
La prima, `void make_heap( RandomIt first, RandomIt last )`; costruisce un max-heap nell'intervallo [first, last). La versione della funzione utilizzata sfrutta l'operatore < per confrontare gli elementi. Essa ha una complessità asintotica $3 \times O(n)$.
La seconda, `void sort_heap( RandomIt first, RandomIt last );` converte il max-heap [first, last) in un intervallo ordinato in ordine crescente. L'intervallo risultante non ha più la proprietà heap. La versione della funzione utilizzata sfrutta l'operatore < per confrontare gli elementi. Essa ha complessità asintotica $2 \times n \times log(n)$.

Il Codice (3.5) sfrutta le funzioni `std::make_heap` e `std::sort_heap` per implementare l'algoritmo Heap-sort.

```
void heap_sort(std::vector<int> &A){
    make_heap(A.begin(),A.end());
    sort_heap(A.begin(),A.end());
   }
```

Codice 3.5

Sono stati effettuati lo stesso numero di esperimenti con gli stessi vettori utilizzati per la sezione precedente, ottenendo la Tabella 3.2 che relaziona le dimensioni dei vettori e i tempi di esecuzione, in millisecondi, dell'algoritmo di ordinamento.

DIMENSIONE VETTORE	**TEMPO DI ESECUZIONE HEAPSORT-IMPLEMENTATO(ms)**	**TEMPO DI ESECUZIONE HEAPSORT-LIBRERIA(ms)**
5	**0,0009388**	**0,0012054**
10	**0,0017486**	**0,0025052**
50	**0,0101254**	**0,0156668**
100	**0,0226778**	**0,030381**
500	**0,1429164**	**0,1824244**
1000	**0,3260002**	**0,4415062**
3000	**1,2024216**	**1,525608**
5000	**2,1245464**	**2,8103312**
7000	**2,9530094**	**4,0753494**
10000	**4,1916348**	**5,8218184**
30000	**15,37134**	**19,7054392**
50000	**26,9145058**	**34,012523**
70000	**38,5308412**	**48,9003298**
100000	**51,7215358**	**68,9718832**
300000	**167,7064158**	**236,6614056**
500000	**298,1242086**	**395,1583428**
700000	**464,2221804**	**641,4302164**
1000000	**735,5433656**	**995,6930314**
3000000	**2444,644286**	**3128,467152**
5000000	**4269,117877**	**5344,711962**
7000000	**6126,384851**	**7764,718494**
10000000	**8948,497922**	**11201,6799**
30000000	**32636,1247**	**40207,5693**
50000000	**51493,01619**	**66652,2075**
70000000	**74660,0328**	**94740,9232**
100000000	**110165,8207**	**138051,8937**

Tabella 3.2

I risultati della Tabella 3.2 sono rappresentati attraverso il Grafico 3.2.

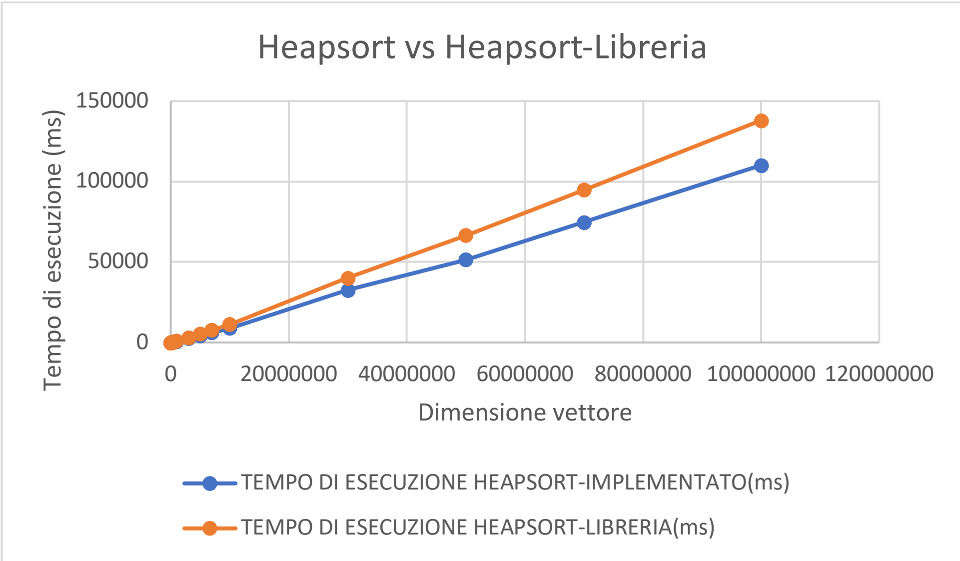

Grafico 3.2

Dal Grafico 3.2 è possibile osservare che i due algoritmo presentano lo stesso andamento asintotico, $O(n\,lg(n))$. La versione di Heap-sort implementata attraverso le funzione della Standard Library presenta dei tempi di esecuzione più elevati all'aumentare di n, numero di elementi del vettore da ordinare. Questo è dovuto alle costanti nascoste che non appaiono nella notazione O, le quali comunque influiscono sul tempo di esecuzione.

4.2 Confronto con Counting-Sort

In quest'ultima sezione è stata confrontata la versione di Heap-sort implementata con un algoritmo di ordinamento lineare, non basato sul confronto. L'algoritmo lineare scelto è Counting-Sort. Questo presenta una complessità asintotica O(K + n), con n numero di elementi presenti all'interno del vettore da ordinare e K cardinalità del range di valori degli interi presenti all'interno del vettore da ordinare.

Il Codice 3.6 descrive l'implementazione dell'algoritmo Counting-Sort.

```
std::vector<int>
countingsort(std::vector<int> A, int k){
std::vector<int> B{A};
int C[k+1];
for(int i=0;i<=k;i++)
C[i]=0;
for(int j=0;j<A.size();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=A.size()-1;j>=0;j--){
B[C[A[j]]-1]=A[j];
C[A[j]]=C[A[j]]-1;
}
return B;
}
```

Codice 3.6

È stato prima verificato sperimentalmente che Counting-Sort avesse una complessità asintotica lineare, utilizzandolo per ordinare lo stesso insieme di vettori utilizzato nelle sezioni precedenti.
È stato fatto in modo che gli interi contenuti all'interno del vettore da ordinare appartenessero al range di valori che va 0 a 10000. In particolare, è stato fatto in modo che per ogni istanza del vettore da ordinare ci fosse un intero pari a 10000, in modo tale che K fosse sempre pari a 10000 in tutti gli esperimenti.
I tempi di esecuzione, espressi in millisecondi, e le relative dimensioni del vettore da ordinare sono riassunti nella seguente Tabella 3.3.

DIMENSIONE VETTORE	TEMPO DI ESECUZIONE COUNTINGSORT-IMPLEMENTATO(ms)
5	**0,0651754**
10	**0,0738956**
50	**0,0759306**
100	**0,079801**
500	**0,0896836**
1000	**0,1091838**
3000	**0,5749636**
5000	**0,329325**
7000	**0,4314814**
10000	**0,484701**
30000	**1,2801806**
50000	**2,9436746**
70000	**2,7935646**
100000	**4,1737568**
300000	**12,3293636**
500000	**19,0743604**
700000	**30,4949234**
1000000	**38,9247354**
3000000	**114,5450678**
5000000	**194,3551782**
7000000	**269,4595734**
10000000	**384,8764696**
30000000	**1265,132511**
50000000	**2074,642125**
70000000	**2908,150252**
100000000	**4104,28445**

Tabella 3.3

I risultati della Tabella 3.3 sono rappresentati attraverso il seguente Grafico 3.3.

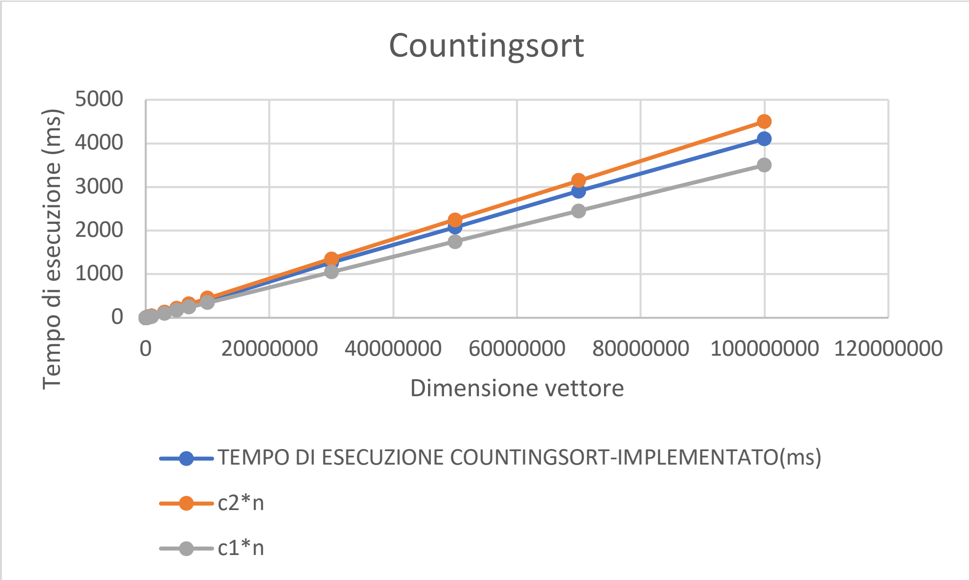

Grafico 3.3

La curva blu rappresenta la curva dei tempi di esecuzione dell'algoritmo Counting-Sort. Allo scopo di ottenere una funzione analitica dei tempi di esecuzione misurati durante gli esperimenti è stata tracciata una linea di tendenza lineare a partire dai tempi di esecuzione misurati per $n \geq 10000$ = K, perché è per K = O(n) che l'algoritmo presenta complessità lineare. Si è ottenuto, in questo, modo la funzione (3.4).

$$f(n) = 4 \cdot 10^{-5}n - 2.9071 \qquad (3.4)$$

Dove n rappresenta la dimensione del vettore da ordinare.
Risolvendo la disequazione (3.5) è stato possibile procedere nel calcolo delle costanti positive c_1 e c_2 e della dimensione del vettore n_0 dalla quale è valida la definizione di notazione θ.

$$c_1 \cdot n \leq 4 \cdot 10^{-5}n - 2.9071 \leq c_2 \cdot n \qquad (3.5)$$

Si è ottenuto il risultato (3.6).

$$n \geq -\frac{145355}{2(25000 \cdot c_1 - 1)}, \quad 0 < c_1 < \frac{1}{25000}, \qquad c_2 \geq \frac{1}{25000} \qquad (3.6)$$

Con la scelta di $n_0 = 581420$ e $c_1 = 0.000035$ e $c_2 = 0.000045$ è quindi verificata la definizione di notazione θ.

Infatti, anche dal Grafico 3.3 è possibile osservare che la curva, per dimensioni del vettore maggiori o uguali a K, $n \geq$ K, può essere maggiorata dalla funzione $c_2 \cdot n$ ed è limitata inferiormente dalla funzione $c_1 \cdot n$ per dimensioni del vettore n maggiori di 581420, con c_1 e c_2 costanti positive scelte pari a 0,000035 e 0,000045 rispettivamente. Si può affermare che la complessità asintotica dell'algoritmo Counting-Sort è $\theta(n)$ nel caso in cui $n \geq$ K.

Dal punto di vista dell'utilizzo della memoria, Heap-Sort risulta più efficiente poiché ordina sul posto, ovvero non ha bisogno di ulteriore memoria per operare, mentre Counting-Sort ha bisogno di due vettori ausiliari, B con dimensione pari ad A, vettore da ordinare, e C con dimensione pari a K cardinalità del range di valori che assumono gli interi da ordinare.
Inoltre, Counting-Sort, a differenza di Heap-Sort pone anche delle pre-condizioni su come devono essere gli elementi che costituiscono il vettore da ordinare, ovvero essi devono essere interi compresi nel range [0, K], mentre Heap-Sort non pone questo limite. Successivamente si procede con il confronto dei tempi di esecuzioni dei due algoritmi.
Per un corretto confronto di Heap-sort con Counting Sort sono stati utilizzati vettori di numeri interi per soddisfare la pre-condizione di Counting-Sort.
I risultati del confronto di Heap-Sort con Counting-Sort sono riassunti nella Tabella 3.4.

DIMENSIONE VETTORE	**TEMPO DI ESECUZIONE HEAPSORT-IMPLEMENTATO (ms)**	**TEMPO DI ESECUZIONE COUNTINGSORT-IMPLEMENTATO (ms)**
5	0,001	0,06
10	0,002	0,066
50	0,011	0,071
100	0,022	0,083
500	0,185	0,133
1000	0,409	0,428
5000	1,94	0,289
10000	4,639	0,434
50000	24,951	2,396
100000	55,387	3,57
500000	277,829	18,301
1000000	654,209	35,262
5000000	3872,04	178,36
10000000	9022,01	366,012
50000000	51799,9	2001,72
100000000	109783	4224,81

Tabella 3.4

I risultati della Tabella 3.4 sono rappresentati attraverso il Grafico 3.4.

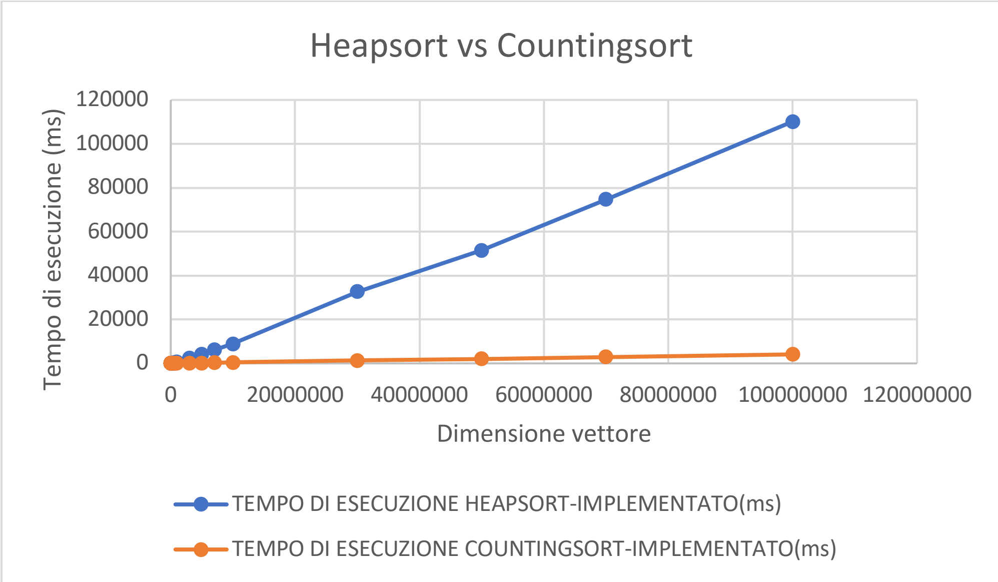

Grafico 3.4

Il K utilizzato per tutti gli esperimenti è K=10000.
Dal Grafico 3.4 è possibile osservare che, all'aumentare della dimensione del vettore Counting-Sort ha prestazioni decisamente migliori rispetto ad Heap-Sort, questo è spiegato anche dalle diverse complessità asintotiche, lineare per Counting-Sort e $O(n\ lg(n))$ per Heap-Sort.

Il risultato, però, è valido solo per $n \geq K$, infatti è in queste condizioni che Counting-Sort esprime la sua maggiore efficienza. Successivamente è stata fatta un'analisi per risaltare questo concetto.
In Tabella 3.5 sono riassunti i risultati del confronto di Heap-Sort e Counting-Sort in condizione $n < K$ con K=250 per tutti gli esperimenti.

DIMENSIONE VETTORE	TEMPO DI ESECUZIONE HEAPSORT-IMPLEMENTATO(ms)	TEMPO DI ESECUZIONE COUNTINGSORT-IMPLEMENTATO(ms)
5	**0,0009388**	**0,0651754**
10	**0,0017486**	**0,0738956**
50	**0,0101254**	**0,0759306**
100	**0,0226778**	**0,079801**
500	**0,1429164**	**0,0896836**
1000	**0,3260002**	**0,1091838**
3000	**1,2024216**	**0,5749636**
5000	**2,1245464**	**0,329325**
7000	**2,9530094**	**0,4314814**
10000	**4,1916348**	**0,484701**
30000	**15,37134**	**1,2801806**
50000	**26,9145058**	**2,9436746**
70000	**38,5308412**	**2,7935646**
100000	**51,7215358**	**4,1737568**
300000	**167,7064158**	**12,3293636**
500000	**298,1242086**	**19,0743604**
700000	**464,2221804**	**30,4949234**
1000000	**735,5433656**	**38,9247354**
3000000	**2444,644286**	**114,5450678**
5000000	**4269,117877**	**194,3551782**
7000000	**6126,384851**	**269,4595734**
10000000	**8948,497922**	**384,8764696**
30000000	**32636,1247**	**1265,132511**
50000000	**51493,01619**	**2074,642125**
70000000	**74660,0328**	**2908,150252**
100000000	**110165,8207**	**4104,28445**

Tabella 3.5

I risultati della Tabella 3.5 sono rappresentati attraverso il seguente Grafico 3.5.

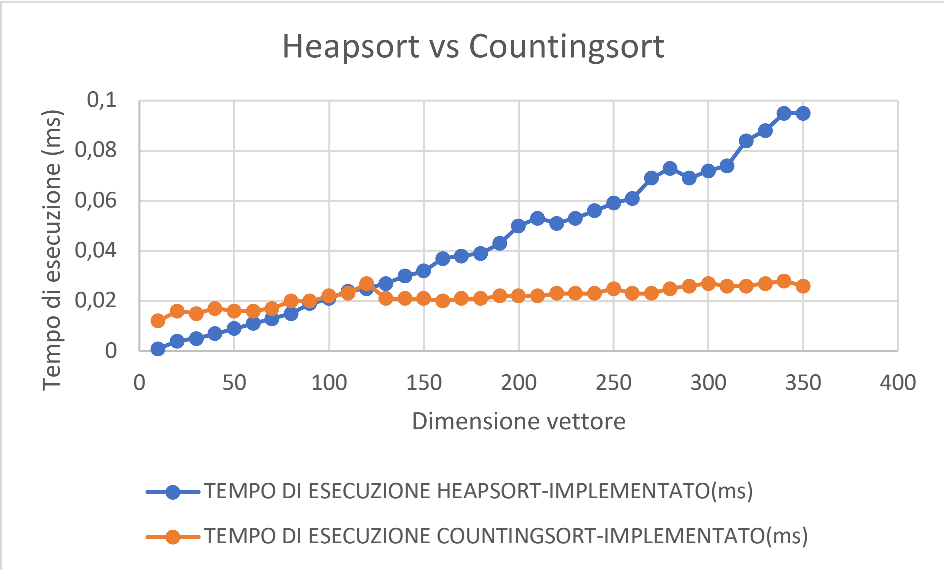

Grafico 3.5

Dal Grafico 3.5 è possibile confermare che per valori di $n \leq$ K Counting-Sort non ha un andamento lineare e le prestazioni di Heap-Sort sono migliori. Ma come visto precedentemente non appena si ha che $n >$ K, Counting-Sort assume un andamento lineare e gli risulta più efficiente di Heap-Sort, come è possibile constatare nuovamente attraverso l'ultimo insieme di esperimenti effettuati con K=250, e riassunti in Tabella 3.6.

DIMENSIONE VETTORE	**TEMPO DI ESECUZIONE HEAPSORT-IMPLEMENTATO (ms)**	**TEMPO DI ESECUZIONE COUNTINGSORT-IMPLEMENTATO (ms)**
50	0,009	0,016
100	0,021	0,022
150	0,032	0,021
200	0,05	0,022
250	0,059	0,025
300	0,072	0,027
350	0,095	0,026
400	0,103	0,04
450	0,114	0,046
500	0,149	0,03
550	0,149	0,035
600	0,161	0,039
650	0,174	0,04
700	0,205	0,054
750	0,229	0,057
800	0,22	0,046
850	0,263	0,051
900	0,261	0,049
950	0,294	0,056
1000	0,322	0,057

Tabella 3.6

I risultati della Tabella 3.6 sono rappresentati attraverso il seguente Grafico 3.6.

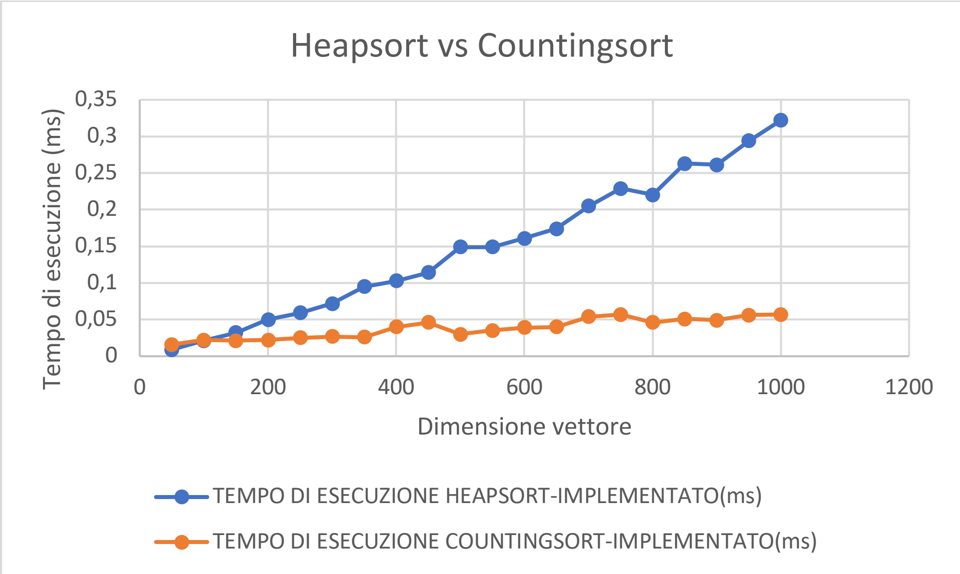

Grafico 3.6

Conclusioni

Nell'elaborato è stata valutata la complessità asintotica dell'algoritmo Heap-Sort, ed è risultato $O(n\,lg(n))$. È stato successivamente verificato il risultato ottenuto sperimentalmente, implementando l'algoritmo in C++. È stato confrontata la versione dell'algoritmo implementata con la versione implementata mediante le funzioni fornite dalla Standard Library e è stato osservato che le due implementazioni presentano lo stesso andamento asintotico $O(n\,lg(n))$. Infine, è stato confrontato Heap-Sort con l'algoritmo Counting-Sort e si è osservato che, per dimensioni del vettore maggiori o uguali alla cardinalità degli interi presenti al suo interno, Counting-Sort presenta prestazioni migliori, ma ciò non vale per dimensioni del vettore inferiori alla cardinalità degli interi presenti al suo interno.