Università degli Studi di Napoli
Federico II
Facoltà di Ingegneria Informatica

Algoritmi di ordinamento: Counting Sort Bucket Sort Heap Sort

Tesina di
Algoritmi e Strutture Dati

A cura di:
CIRIELLO VINCENZO 885/327
DI LUCA GIUSEPPE 885/326
DI PAOLO VINCENZO 885/328

1. COUNTING SORT

1.1 Pseudocodice

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to lunghezza[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] adesso contiene il numero di elementi uguale a i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] adesso contiene il numero di elementi minore o uguale a i.
 9  for j ← lunghezza[A] downto 1
10       do B[C[A[j]]] ← A[j]
11          C[A[j]] ← C[A[j]] − 1
```

Il ***Counting Sort*** è un algoritmo di ordinamento senza confronti: questo, come vedremo, ci permette di avere l'ordinamento del vettore di input in un tempo praticamente lineare. Lo schema di funzionamento è di immediata semplicità: noto a priori l'elemento massimo di un vettore di interi (nella firma della funzione esso è identificato con k), si costruisce un vettore ausiliario C di lunghezza k+1. Ogni elemento di quest'ultimo rappresenterà il numero di occorrenze che ha il suddetto elemento all'interno del vettore di input A. Dopo un primo passo di inizializzazione, infatti, ogni elemento i-esimo del vettore C viene incrementato ogniqualvolta sia presente il numero i all'interno del vettore A. Nel passo successivo, dunque, si sommano i valori degli elementi successivi presenti in C; in tal modo otteniamo, alla fine di quest'ultimo ciclo, un vettore che contiene, a meno di valori uguali, le posizioni che dovranno avere nel vettore di output B gli elementi di A. Nel caso di valori uguali in A, il problema viene risolto nell'ultimo ciclo, per mezzo degli iterati decrementi degli elementi di C.

1.2 Implementazione

La stesura del codice nel linguaggio C rispecchia quella dello pseudocodice. Essa si differenzia solo per le modifiche dovute al fatto che l'indicizzazione degli array, nello specifico linguaggio, parte dal valore 0 e non da 1.

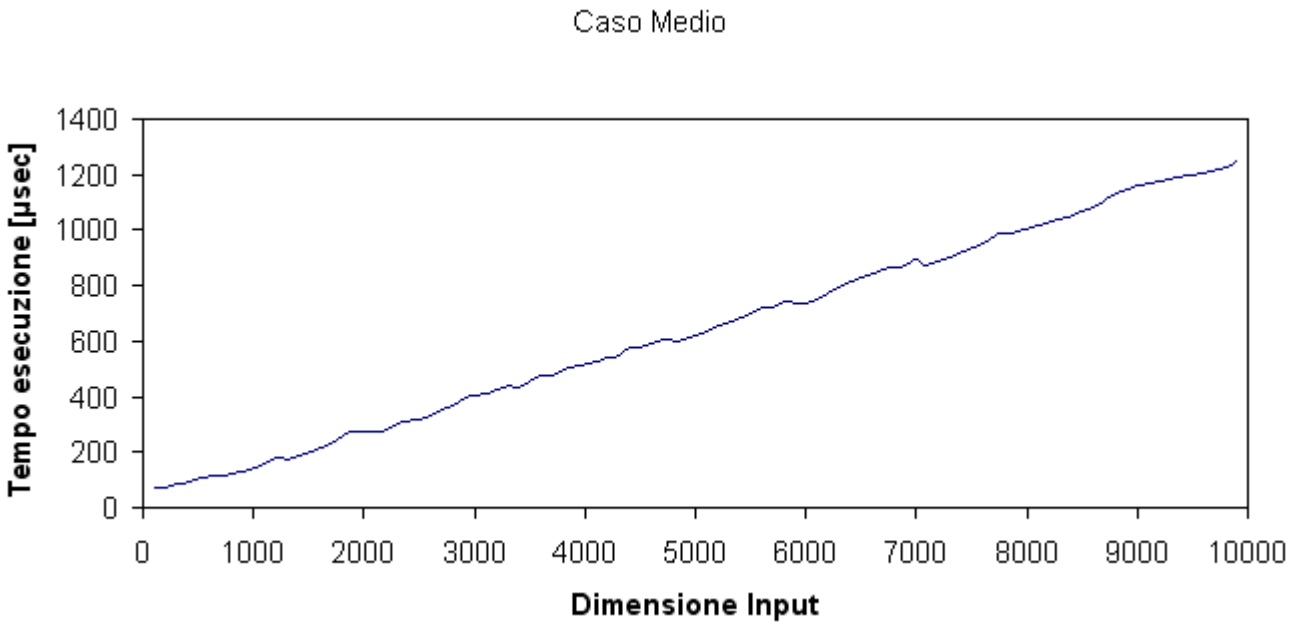
```c
// counting.h

#ifndef COUNTING_H
#define COUNTING_H

void counting_sort(int* A, int Alen, int* B, int k);

#endif
```

```c
// counting.c

#include "counting.h"
#include <stdio.h>

void counting_sort(int* A, int Alen, int* B, int k){


	int i;
	int C[k];

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

	for(j=0; j<Alen; j++)
		C[A[j]] = C[A[j]]+1;  //C[j] contiene il numero
				       // di elementi di A pari a j

	for(i=1; i<k; i++)
		C[i] = C[i]+C[i-1];  //C[j] contiene il numero di elementi
				     // minori o uguali a j

	for (j=Alen-1; j>=0; j--){

		B[C[A[j]]-1] = A[j]; //Sul libro porta B da 1 a Alen,
				     //mentre il nostro B va 0 ad Alen-1
				     //perciò si è inserito il -1 come si può vedere
		C[A[j]] = C[A[j]]-1;

	}

	printf("\n\n");
}
```

```c
// main.c

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include "counting.h"


int main(int argc, char** argv){


	FILE* out;
	out = fopen("result.csv","a");
	int* A = malloc(sizeof(int)*atoi(argv[1]));

	srand(time(0));

	int i;
	for(i=0;i<atoi(argv[1]);i++)
		A[i] = rand() %atoi(argv[1]);

	//Calcoliamo il massimo elemento contenuto
	// nel vettore A generato in automatico
	int max = A[0];

	for(i=1; i<atoi(argv[1]); i++)

		if(A[i]>max) max = A[i];

	int* B = malloc(sizeof(int)*atoi(argv[1]));

	struct timeval t1,t2;

	gettimeofday(&t1,NULL);
	counting_sort(A,atoi(argv[1]),B,max+1);
	gettimeofday(&t2,NULL);

	int usec;
	usec = (t2.tv_sec*1000000 + t2.tv_usec) -
                                              (t1.tv_sec*1000000 + t1.tv_usec);

	printf("Size: %d, USeconds:  %d\n",atoi(argv[1]),usec);
	fprintf(out,"%d\t%d\n",atoi(argv[1]),usec);
	fclose(out);

	free(A);
	free(B);
	return 0;
}
```

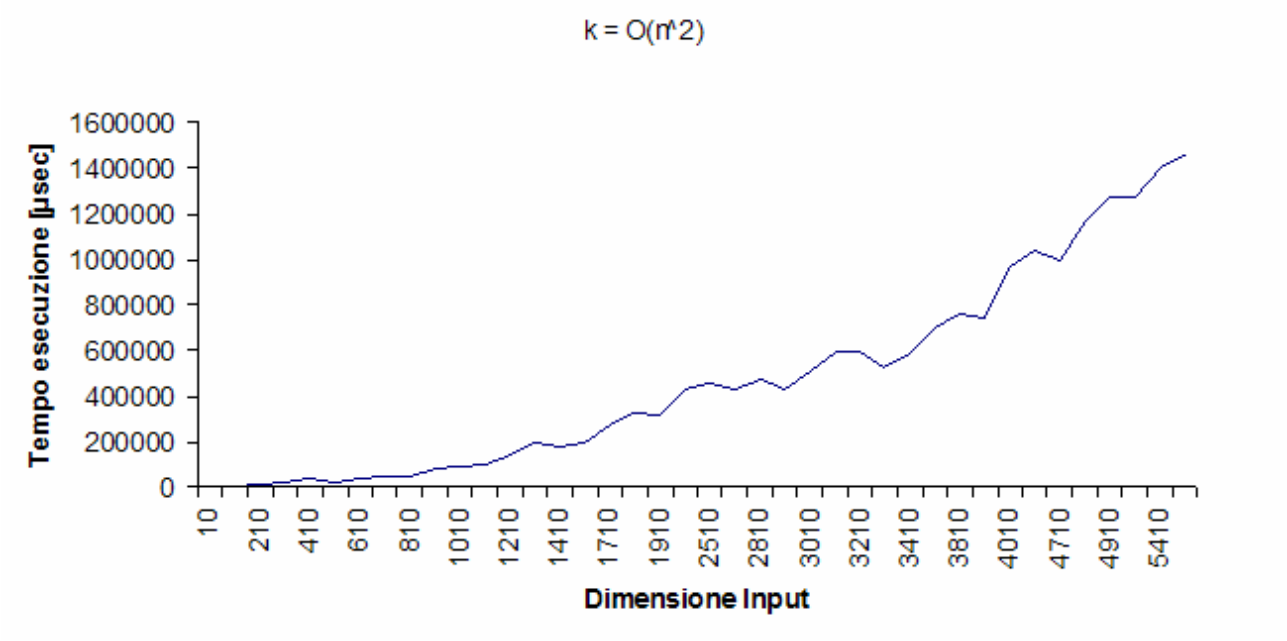
1.3 Analisi delle prestazioni

Quando $k = O(n)$, l'ordinamento viene effettuato nel tempo Θ(n). Ciò è facilmente comprensibile osservando dallo pseudocodice che il primo e il terzo ciclo *for* (righe 1 e 6) hanno complessità $\Theta(k)$ mentre il secondo e il quarto (righe 3 e 9) hanno complessità $\Theta(n)$. Sommando questi valori, l'algoritmo in definitiva ha una complessità pari a $\Theta(n+k)$ da cui, per $k = O(n)$ abbiamo l'andamento è proprio Θ(n). Il grafico mostrato nella figura seguente rispecchia abbastanza fedelmente l'andamento ideale.



Il grafico è stato costruito utilizzando vettori di input da 10 a 10010 elementi, generati casualmente dalla macchina tra 0 ed n, con un passo di 100 elementi alla volta. Si fa notare che per avere un grafico che rispecchiasse più fedelmente possibile l'andamento medio, sono stati eliminati i valori di picco del grafico, ovvero quelli ottenuti per dimensioni degli array in corrispondenza delle quali il tempo di elaborazione risultasse significativamente distante dal valore medio.

Quando, invece, $k = O(n^2)$, all'interno della espressione di complessità è k a prendere il sopravvento, e a far avere alla procedura un andamento di tipo quadratico. Il grafico seguente è stato ottenuto modificando *main.c* in modo da avere un valore di k pari esattamente a n^2: notiamo immediatamente l'andamento parabolico che il tempo di esecuzione assume.



2. BUCKET SORT

2.1 Pseudocodice

```
BUCKET-SORT(A)
1  n ← lunghezza[A]
2  for i ← 1 to n
3       do inserisci A[i] nella lista B[⌊nA[i]⌋]
4  for i ← 0 to n − 1
5       do ordina la lista B[i] con insertion sort
6  concatena ordinatamente le liste B[0], B[1], . . . , B[n − 1]
```

Bucket sort suppone che l'input sia generato da un processo casuale che distribuisce gli elementi uniformemente nell'intervallo [0, 1[.
Il concetto che sta alla base dell'algoritmo è quello di dividere l'intervallo [0*,* 1[in *n* sottointervalli della stessa dimensione, detti ***bucket***, nei quali vengono distribuiti gli *n* valori di input. A questo scopo lo pseudocodice che definisce l'algoritmo suppone che l'input sia un array *A* di *n* elementi e richiede un array ausiliario *B*[*0...n−1*] di liste concatenate (bucket). L'algoritmo procede semplicemente ordinando i valori in ogni bucket tramite un ordinamento di tipo insertion sort. L'output viene infine prodotto concatenando in ordine i vari bucket in un array.

2.2 Implementazione

L'implementazione è stata ottenuta seguendo fedelmente lo pseudocodice. In particolare, la funzione *floor* (riga 3) è stata realizzata utilizzando l'operatore di casting a intero; per quanto riguarda, invece, le liste B[i], si è scelto di implementarle mediante liste doppiamente concatenate con sentinella. Si fa notare che, per agevolare l'ordinamento dei bucket è stata realizzata una procedura, la *list_to_array*, che converte una lista in un vettore i cui elementi siano pari ai valori contenuti nel campo *key* di ogni elemento della lista stessa. Per ottenere le liste ordinate, dunque, è stato richiamato l'algoritmo insertion sort per le liste convertite, ovvero per gli array summenzionati. La procedura *list_free*, infine, è stata implementata per la corretta deallocazione delle liste concatenate, e viene richiamata appena prima di uscire dalla procedura *bucket_sort*, su tutti gli elementi del vettore di liste B.

```c
// lista.h

#ifndef LISTA_H
#define LISTA_H

#define NIL -1

typedef struct nodo NODO;

struct nodo{

	NODO* next;
	NODO* prev;
	float key;

};


NODO* init(float key);
void list_insert(NODO* lista, NODO* x);
void list_delete(NODO* x);
NODO* list_search(NODO* lista, float key);
void list_to_array(NODO* lista, float* v);
int list_length(NODO* lista);
void list_free(NODO* lista);

#endif
```

```c
// lista.c

#include <stdlib.h>
#include <stdio.h>
#include "lista.h"

NODO* init(float key){

	NODO* n = malloc(sizeof(NODO));
	n->key = key;
	n->prev = n;
	n->next = n;
	return n;

}


void list_insert(NODO* lista, NODO* x){

	x->next = lista->next;
	lista->next->prev = x;
	lista->next = x;
	x->prev = lista;

}
```

```
void list_delete(NODO* x){

	x->prev->next = x->next;
	x->next->prev = x->prev;

	free(x);
}


NODO* list_search(NODO* lista, float key){


	NODO* x = lista->next;
	while(x!=lista && x->key!=key)
		x = x->next;
	return x;

}



void list_to_array(NODO* lista, float* v){

	NODO* x = lista->next;

	int i = 0;
	while(x!=lista){

		v[i] = x->key;
		x = x->next;
		i++;
	}

}

int list_length(NODO* lista){

	int count = 0;
	NODO* x = lista->next;
	while(x!=lista){
		count++;
		x = x->next;
	}
	return count;

}

void list_free(NODO* lista) {

	NODO* x;

	while (x != lista) {
		x = lista->next;

		list_delete(x);
	}
}
```

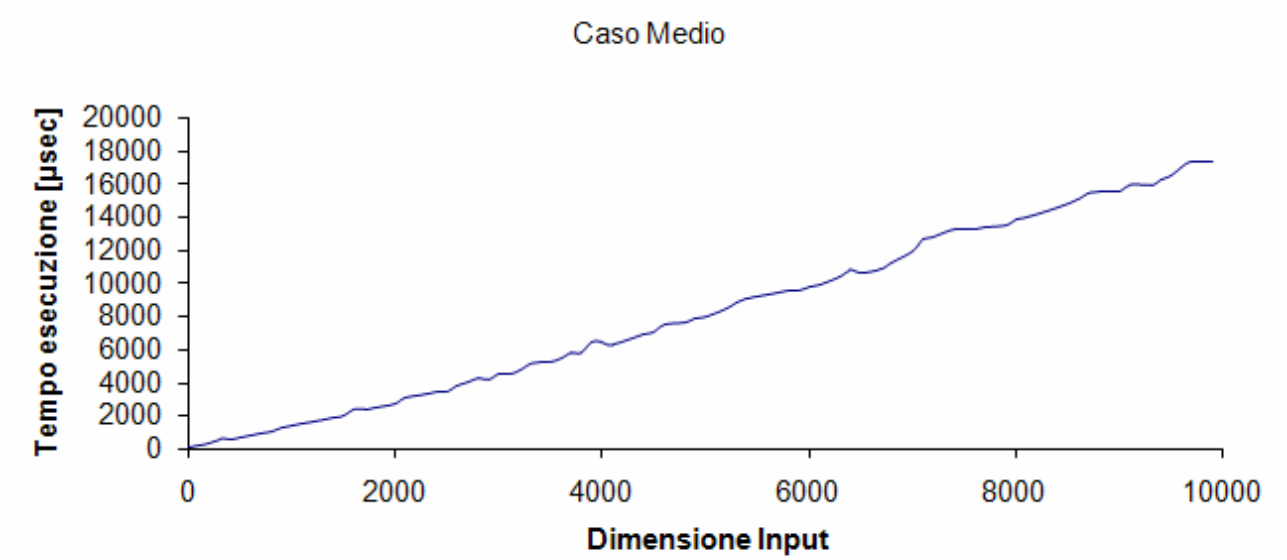
```c
// insertion.h

# ifndef INSERTION_H
# define INSERTION_H

# include <stdio.h>

void insertion_sort (float* v, int len);

void print (float* v, int len);

# endif


// insertion.c

#include "insertion.h"

void insertion_sort (float* v, int len) {

     float key = 0;
     int i = 0;

     int j;
     for (j = 1; j < len; j++) {

          key = v[j];

          i = j - 1;
          while (i >= 0 && v[i] > key) {
                v[i+1] = v[i];
                i = i - 1;
          }

          v[i+1] = key;

     }

}

void print (float* v, int len) {

     int i;
     for (i = 0; i < len; i++)
         printf("%f ",v[i]);
     printf("\n");

}

// bucket_sort.h

#ifndef BUCKET_SORT_H
#define BUCKET_SORT_H

void bucket_sort(float* A,int len);

#endif
```

```c
// bucket_sort.c

#include "bucket_sort.h"
#include "lista.h"
#include "insertion.h"
#include <stdlib.h>

void bucket_sort(float* A, int len){

	int i;
	NODO* B[len];
	for(i=0; i<len; i++)

		B[i] = init(NIL); //associa la sentinella ad ogni elemento
				  // del vettore B di liste


	for(i=0; i<len; i++) //effettua l'inserimento degli elementi in B
			     // secondo bucket_sort

		list_insert(B[(int)(len*A[i])],init(A[i]));

	int k=0; //indice che scorre l'array A (originario)
	for(i=0; i<len; i++){ //ordina le liste degli elementi di B

		int vlen = list_length(B[i]);

		float* v = malloc(sizeof(float)*vlen);

		list_to_array(B[i],v); //porta gli elementi delle liste B[i]
				       // in un array temporaneo

		insertion_sort(v,vlen);

		int j=0;
		for(j;j<list_length(B[i]);j++){

			A[k] = v[j];	//concatenamento degli array ordinati
					 // e posizionamento nell'array A
			k++;
		}

		free(v);
	}

	for (i=0; i<len; i++)
		list_free(B[i]);

}
```

```c
// main.c

#include "insertion.h"
#include "lista.h"
#include <time.h>
#include <stdlib.h>

int main(int argc, char** argv)
{

    FILE* out;
    out = fopen("result.csv","a");
    float* A = malloc(sizeof(float)*atoi(argv[1]));

    srand(time(0));

    int i;
    for(i=0;i<atoi(argv[1]);i++)
            A[i] = ((float) (rand() % 100))/100;

    struct timeval t1,t2;

    gettimeofday(&t1,NULL);
    bucket_sort(A,atoi(argv[1]));
    gettimeofday(&t2,NULL);

    int usec;
    usec = (t2.tv_sec*1000000+t2.tv_usec) - (t1.tv_sec*1000000+t1.tv_usec);

    printf("Size: %d, USeconds:  %d\n",atoi(argv[1]),usec);
    fprintf(out,"%d\t%d\n",atoi(argv[1]),usec);
    fclose(out);

    free(A);

    return 0;
}
```

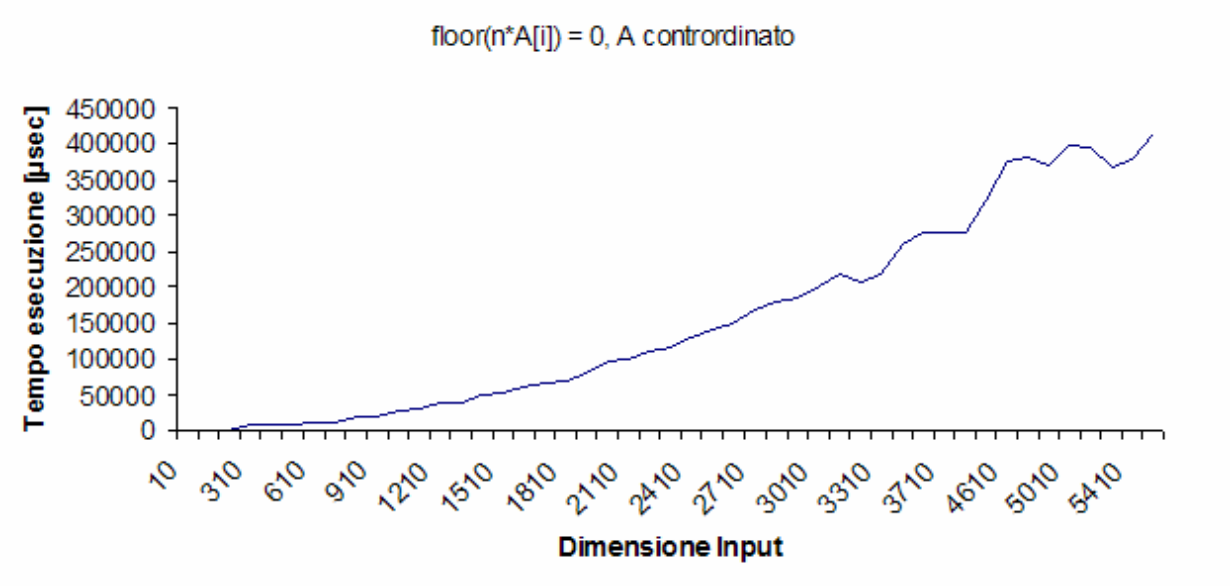
2.3 Analisi delle prestazioni

Il tempo di esecuzione atteso di bucket sort è un Θ(n) quando l'input è estratto da una distribuzione uniforme. Come counting sort, bucket sort è veloce perchè fa un'ipotesi sull'input. Mentre counting sort suppone che l'input sia formato da interi in un piccolo intervallo, bucket sort ipotizza invece che gli input siano uniformemente distribuiti nell'intervallo [0, 1[.
Per analizzare il tempo di esecuzione, notiamo che tutte le righe, tranne la 5, richiedono un tempo $O(n)$ nel caso peggiore. Il tempo totale richiesto dalle n chiamate di insertion sort nella riga 5 risulta essere pari a Θ(n); ne consegue, pertanto, che il tempo d'esecuzione complessivo è Θ(n). Anche in questo caso, il grafico mostrato nella figura seguente rispecchia abbastanza fedelmente l'andamento ideale.



Il grafico è stato costruito utilizzando vettori di input da 10 a 10010 elementi, generati casualmente dalla macchina tra 0 ed n, con un passo di 100 elementi alla volta. Si fa notare che per avere un grafico che rispecchiasse più fedelmente possibile l'andamento medio, sono stati eliminati i valori di picco del grafico, ovvero quelli ottenuti per dimensioni degli array in corrispondenza delle quali il tempo di elaborazione risultasse significativamente distante dal valore medio.

Allorché, invece, il vettore di input è composto da elementi che non rispettano il vincolo di uniformità all'interno dell'intervallo [0, 1[, la maggior parte degli elementi di A vengono inseriti in una sola lista. A questo punto è l'insertion sort chiamato su questa lista a determinare la complessità dell'algoritmo che ricordiamo, nel caso peggiore, presenta un tempo di esecuzione dell'ordine di $O(n^2)$. Nell'esempio seguente, si sono generati valori per il vettore A, tali che $floor(n \cdot A[i]) = 0 \ \forall i$ (in modo tale che tutti gli elementi fossero inseriti nella stessa lista concatenata). Non solo, ma i valori generati erano perfettamente contrordinati, in modo da ottenere perfettamente il caso peggiore per l'insertion sort: è evidente infatti, nel grafico, l'andamento quadratico del tempo di esecuzione.



3. HEAP SORT

3.1 Pseudocodice

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4     then massimo ← l
 5     else  massimo ← i
 6  if r ≤ heap-size[A] and A[r] > A[massimo]
 7     then massimo ← r
 8  if massimo ≠ i
 9     then scambia A[i] ↔ A[massimo]
10          MAX-HEAPIFY(A, massimo)

BUILD-MAX-HEAP(A)
1  heap-size[A] ← lunghezza[A]
2  for i ← ⌊lunghezza[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)

HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← lunghezza[A] downto 2
3       do scambia A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          MAX-HEAPIFY(A, 1)
```

Un ***heap (binario)*** è una struttura dati composta da un array che possiamo considerare come un albero binario quasi completo. Ogni nodo dell'albero corrisponde a un elemento dell'array che memorizza il valore del nodo. Tutti i livelli dell'albero sono completamente riempiti, tranne eventualmente l'ultimo che può essere riempito da sinistra fino a un certo punto. Un *max-heap* è un heap tale che *A[parent[i]] > A[i]*: è evidente che l'elemento A[1] contiene il massimo valore.
La procedura ***max-heapify***, dato un nodo *i* di un heap *A*, modifica il sottoalbero con radice *i* rendendolo un max-heap; la procedura ***build-max-heap*** richiama la *max-heapify* in maniera tale da rendere l'array di input un max-heap.
L'algoritmo ***heapsort*** inizia utilizzando la procedura *build-max-heap* per costruire un max-heap dell'array di input *A*[*1...n*], dove *n* è pari alla lunghezza di A. Poiché l'elemento più grande dell'array è memorizzato nella radice *A*[1], esso può essere inserito nella sua posizione finale corretta scambiandolo con *A*[*n*]. Scartando il nodo *n* dall'heap (diminuendo *heap-size*[*A*]), si nota che *A*[*1...(n−1)*] può essere facilmente trasformato in un max-heap. I figli della radice restano max-heap, ma la nuova radice potrebbe violare la proprietà del max-heap. Per ripristinare questa

proprietà, tuttavia, basta una chiamata alla procedura max-heapify(A,1), che lascia un max-heap in $A[1...(n-1)]$. L'algoritmo heapsort poi ripete questo processo per il max-heap di dimensione $n-1$ fino a un heap di dimensione 2.

3.2 Implementazione

L'implementazione rispecchia fedelmente lo pseudocodice, eccetto che per le funzioni *left*, *right* e *parent*, che sono state adattate al particolare linguaggio utilizzato.

```
// heap_sort.h

#ifndef HEAP_SORT_H
#define HEAP_SORT_H

void max_heapify(int* A, int i);
void build_max_heap(int* A,int Alen);
void heap_sort(int* A, int Alen);
int parent(int i);
int left(int i);
int right(int i);

#endif
```

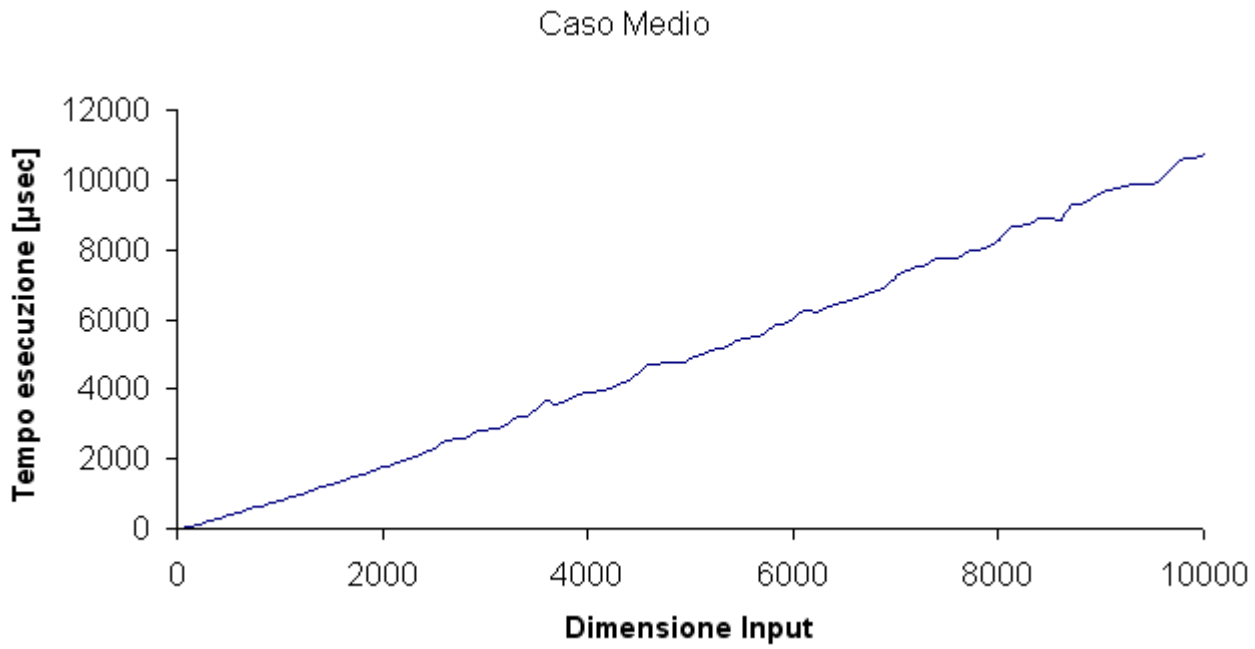
```
// heap_sort.c

#include "heap_sort.h"

static int heap_size;

void max_heapify(int* A, int i){

	int max;
	int l = left(i);
	int r = right(i);
	if (l<heap_size && A[l]>A[i])
		max = l;
	else
		max = i;
	if (r<heap_size && A[r]>A[max])
		max = r;
	if (max!=i){
		int tmp;
		tmp = A[i];
		A[i] = A[max];
		A[max] = tmp;
		max_heapify(A,max);
	}
}
```

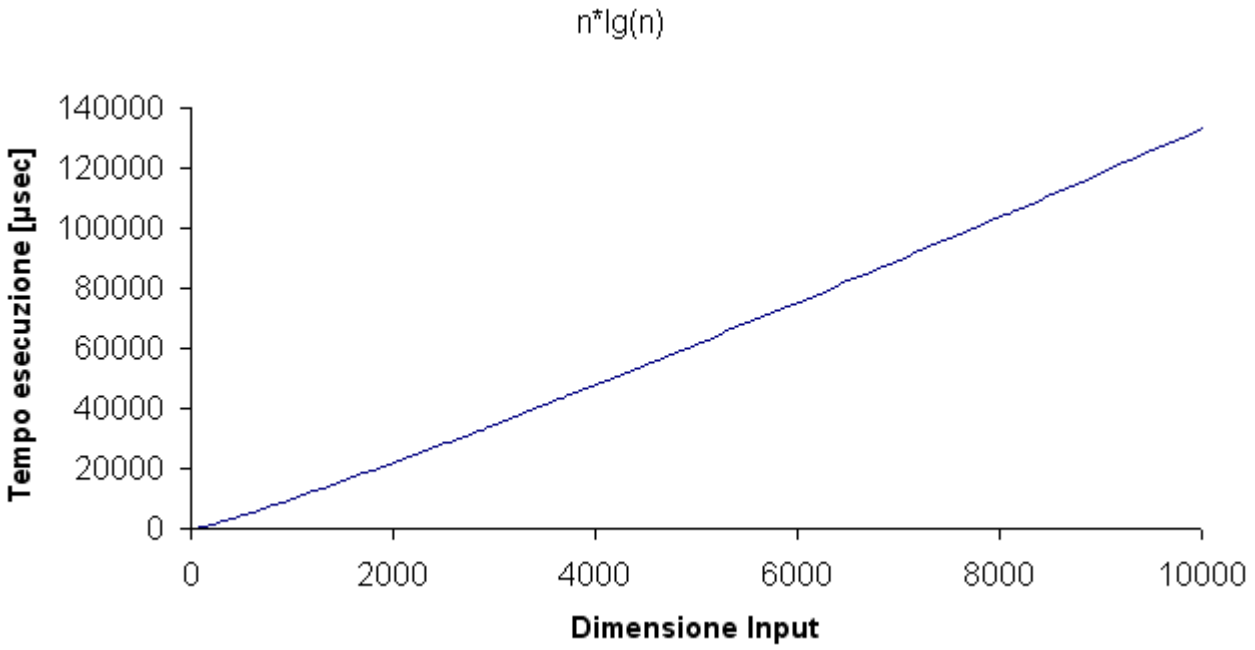
```
void build_max_heap(int* A,int Alen){

	heap_size = Alen;
	int i;

	for(i=Alen/2-1; i>=0; i--)
		max_heapify(A,i);
}


void heap_sort(int* A, int Alen){

	build_max_heap(A,Alen);
	int i;
	for(i = Alen-1; i>0; i--){
		int tmp;
		tmp = A[0];
		A[0] = A[i];
		A[i] = tmp;
		heap_size--;
		max_heapify(A,0);
	}
}


int parent(int i){

	return ((i-1)/2);

}


int left(int i){

	return (2*i+1);

}


int right(int i){

	return (2*i + 2);

}
```

```c
// main.c

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include "heap_sort.h"
#include <math.h>

int main(int argc, char** argv){

	FILE* out;
	srand(time(0));
	int* A;
	int size = atoi(argv[1]);

	out = fopen("results.csv","a");

		A = malloc(sizeof(int)*size);

		int k;
		for(k=0; k<size;k++)
			A[k] = rand()%size;

		struct timeval t1,t2;
		gettimeofday(&t1,NULL);
		heap_sort(A,size);
		gettimeofday(&t2,NULL);

		int usec;
		usec = (t2.tv_sec*1000000 + t2.tv_usec) -
						(t1.tv_sec*1000000 + t1.tv_usec);

		printf("\nSIZE: %d, Time elapsed useconds: %d \n",size,usec);

		fprintf(out,"%d\t\t%d\n",size,usec);

	fclose(out);
	free(A);
	return 0;

}
```

3.3 Analisi delle prestazioni

Per il caso 2) del teorema Master la complessità della procedura *max-heapify* è pari a *O(lg(n))*. Per quanto riguarda, invece, la *build-max-heap* si ha che un limite asintoticamente stretto della complessità è dato da *O(n)*. Dunque la procedura *heapsort* impiega un tempo O(n · lg(n)), in quanto la chiamata di *build-max-heap* impiega un tempo *O(n)* e ognuna delle n-1 chiamate alla *max-heapify* impiega un tempo *O(lg(n))*. In questo caso, il grafico mostrato nella figura seguente risulta essere maggiorato da n · lg(n).





Il grafico è stato costruito utilizzando vettori di input da 10 a 10010 elementi, generati casualmente dalla macchina tra 0 ed n, con un passo di 100 elementi alla volta. Si fa notare che per avere un grafico che rispecchiasse più fedelmente possibile l'andamento medio, sono stati eliminati i valori di picco del grafico, ovvero quelli ottenuti per dimensioni degli array in corrispondenza delle quali il tempo di elaborazione risultasse significativamente distante dal valore medio.

3. ANALISI COMPARATIVA

In questo lavoro sono stati considerati tre algoritmi di ordinamento molto veloci: counting sort, bucket sort e heap sort.

Il primo assume come ipotesi che i valori del vettore da ordinare siano compresi fra 0 e k. Se k è dello stesso ordine di n (lunghezza del vettore di input) l'algoritmo viene eseguito in un tempo lineare. Se k invece, come è stato mostrato, è di ordine superiore ad n, la complessità aumenta insieme all'ordine di k.

Il secondo assume come ipotesi che i valori del vettore di input siano uniformemente distribuiti tra 0 e 1. Secondo queste ipotesi il tempo di esecuzione atteso è dello stesso ordine di n. Nel peggiore dei casi, invece, se i valori sono addensati attorno a uno stesso valore, l'algoritmo si trasforma nell'insertion sort il quale, nel caso peggiore, ha una complessità di $O(n^2)$.

Il terzo, infine, non assume nessuna ipotesi sul vettore di input, ma per alcune dimensioni del vettore (quando l'ultimo livello del max-heap corrispondente è pieno a metà), può avere un tempo di esecuzione dell'ordine di $O(n \cdot lg(n))$.

In conclusione, entrambi il primo e il secondo algoritmo, sono molto buoni quando è sempre verificata una particolare condizione del vettore da ordinare. Quando queste condizioni peggiorano, la complessità dei due algoritmi aumenta visibilmente. L'heap sort, invece, risulta buono quando non è possibile effettuare nessuna ipotesi sul vettore di input (per di più opera sul posto); del resto, l'utilizzo della ricorsione e il comportamento dell'algoritmo per alcune dimensioni del vettore da ordinare, rendono questo algoritmo non sempre il migliore da utilizzare.

Segue il grafico di confronto fra i tempi effettivi in microsecondi dei tre algoritmi: come si vede l'andamento è lineare per tutti gli algoritmi, ma la pendenza è visibilmente minore per il counting sort, mentre quella dell'heapsort è circa la metà di quella del bucket sort.