Relazione per il Progetto di Laboratorio di Algoritmi e Strutture Dati

Daniele Ferroli – Matricola: 137357

ferroli.daniele@spes.uniud.it

Corso di Studi: Informatica

Anno 2018/2019

Sommario

Introduzione

Consegna del progetto

$Si\ considerino\ n\ valori\ razionali\ positivi\ (pesi) w_1, \dots, w_n e\ si\ indichi\ con\ W\ la\ loro\ somma$:
$W = \sum_{i=1}^{n} w_i$. $Chiamiamo\ mediana\ (inferiore)\ pesata\ di\ w_1, \dots, w_n, il\ peso\ w_k\ tale\ che$:

$$\sum_{w_i < w_k} w_i < \frac{W}{2} \leq \sum_{w_i \leq w_k} w_i$$

Sono necessari due programmi basati sullo stesso codice. Il primo programma è quello che risolve il problema: deve prendere i dati dallo standard input e scrivere il risultato sullo standard output risolvendo il problema.

Il secondo programma serve per la misurazione dei tempi per i grafici: dovrà produrre dei dati casuali e misurarli secondo le specifiche degli appunti delle lezioni.

Definizione del problema

Il primo programma deve calcolare la mediana inferiore pesata di n valori razionali positivi.

La mediana pesata inferiore è il numero che, sommato ai suoi elementi precedenti ordinati, supera o è uguale alla loro somma W diviso due.

Specifiche per il formato dell'input e dell'output

L'input del programma sarà una sequenza di numeri razionali separati da una virgola (ignoriamo gli spazi) e terminante con un punto. L'input verrà fornita da linea di comando (standard input).

L'output dovrà venire scritto a terminale (standard output) e dovrà consistere esclusivamente in uno degli elementi in input.

Soluzione proposta

La soluzione proposta utilizza un algoritmo di ordinamento (Merge Sort) per risolvere il problema. Utilizzando la definizione precedente di mediana inferiore pesata, vengono sommati gli elementi in input in una variabile chiamata W. Viene ordinato l'insieme e a partire dall'elemento più piccolo si somma elemento dopo elemento fino a quando la sommatoria supera la metà o è uguale a W/2.

Linguaggio di programmazione utilizzato

Il linguaggio di programmazione utilizzato per scrivere l'algoritmo è il C.

Specifiche per il formato dell'input e dell'output

In input vengono lette fino a 15 cifre dopo la virgola e viene stampato in output il risultato con altrettante cifre.

Algoritmo utilizzato per il risolvere il problema

L'algoritmo calcola la mediana pesata inferiore utilizzando un algoritmo di ordinamento.

In caso in cui non trovi l'elemento (impossibile) ritorna -1 per segnalare l'errore.

```
1	double weighted_median_sort(double *ptr, int size){
2		double W = sum_partitions(ptr, 0, size-1);
3		sort(ptr, size);
4		double sum = 0;
5		for (int i=0; i<size; i++){
6			sum += ptr[i];
7			if (sum >= W/2){
8				return ptr[i];
9			}
10		}
11		return -1;
12	}
```

Il metodo sum_partitions calcola la somma di una partizione di un insieme a partire dall'indice p fino all'indice q.

```
1	double sum_partitions (double *ptr, int p, int q){
2		double sum = 0.0;
3		for (int i=p;i<=q;i++){
4			sum += ptr[i];
5		}
6		return sum;
7	}
```

Non viene riportato l'implementazione del codice di sort perché è l'implementazione di Merge Sort vista a lezione.

Algoritmo utilizzato per prendere l'input

L'algoritmo per la gestione dell'input inizializza un puntatore di double a una size prefissata iniziale.

Man mano che gli input vengono letti, se necessario viene incrementato lo spazio di memoria che viene puntato dalla variabile array che contiene l'input.

scanf() si occupa di rilevare i double nell'input, tiene conto che tra un double e un altro ci sia sempre almeno uno spazio o una virgola. Siccome le specifiche dell'input prevedono sempre una virgola in mezzo ma indefiniti spazi tra un numero e l'altro, è sufficiente questa regola per la gestione dell'input.

Una volta letto l'input parte l'algoritmo.

```
1   void start_project(){
2       int size = 1;
3       double *array = malloc(size*sizeof(double));
4       int read = 0;
5       while (scanf("%lf%*[ ,]", &array[read]) >= 1) {
6           if (++read == size){
7               size *= 2;
8               array = realloc(array, size*sizeof(double));
9           }
10      }
11  start_algorithm_print(array, read);
12  }
```

Calcolo della complessità

La complessità del codice dipende dai metodi chiamati durante la risoluzione del metodo weighted_median_sort(double *ptr, int size) e dal metodo stesso.

I metodi utilizzati (riportando l'intestazione) sono:

```
1   double weighted_median_sort(double *ptr, int size);
2   double sum_partitions(double *ptr, int p, int q);
3   void sort(double *ptr, int size);
4   void mergesort(double *ptr, int p, int q, int size);
5   void merge(double *ptr, int p, int q, int r, int size);
```

A seguire sono presenti i calcoli della complessità:

Merge

```
1   void merge (double *ptr, int p, int q, int r, int size){          COSTO
2     double B[size];
3     int i = p;
4     int j = r + 1;
5     for (int k = p; k <= q; k++){                                    Θ(q-p)
```

```
6           if (i<=r && j<=q) {
7              if (ptr[i] <= ptr[j]) {
8                 B[k] = ptr[i];
9                 i = i + 1;
10             } else if (ptr[i] > ptr[j]) {
11                B[k] = ptr[j];
12                j = j + 1;
13             }
14          } else if (i<=r){
15             B[k]=ptr[i];
16             i=i+1;
17          } else if (j<=q){
18             B[k]=ptr[j];
19             j=j+1;
20          }
21       }
22       for (int k = p; k<=q; k++){                    Θ (q-p)
23          ptr[k]=B[k];
24       }
25    }
```

Il costo totale di merge è Θ(n).

MergeSort

```
1     void mergesort (double *ptr, int p, int q, int size) {     COSTO
2        if (p < q) {
3           int r = p + (q - p) / 2;                            c
4           mergesort(ptr, p, r, size);                         T(n/2)
5           mergesort(ptr, r + 1, q, size);                     T(n/2)
6           merge(ptr, p, q, r, size);                          c*n
7        }
8     }
```

Il costo totale di mergesort è:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

$Utilizzando\ l'albero\ delle\ chiamate\ ricorsive\ il\ costo\ e'$:

$$T(n) = c * n + c * n + \cdots + c * n = c * n * \log_2 n = \ \Theta(\mathrm{n}\log_2 n)$$

Sort

La complessità del metodo verrà commentata riga per riga:

```
1    void sort (double *ptr, int size){                    COSTO
2       mergesort(ptr, 0, size-1, size);                   Θ(nlogn)
3    }
```

Sum Partitions

La complessità del metodo verrà commentata riga per riga:

```
1    double sum_partitions(double *ptr, int p, int q){     COSTO
2       double sum = 0.0;                                  O(1)
3       for (int i = p; i<=q; i++){                        Θ(q-p)
4          sum += ptr[i];                                  O(1)
5       }
6       return sum;                                        O(1)
7    }
```

Il costo in totale nel caso peggiore è la somma dei vari costi:

$$T(n) = O(1) + \Theta(\mathrm{q} - \mathrm{p}) * O(1) + O(1)$$
$$= \Theta(\mathrm{q} - \mathrm{p})$$

Siccome il costo dipende dal range di 'p' e 'q' e in weighted_median_sort viene chiamato su un array di dimensione n ne segue che il costo è sempre Θ(n).

Weighted Median Sort

La complessità del metodo nel caso peggiore verrà commentata riga per riga:

```
1    double weighted_median_sort(double *ptr, int size){   COSTO
2          double W = sum_partitions(ptr, 0, size-1);      Θ(n)
3          sort(ptr, size);                                Θ(nlogn)
4          double sum = 0;                                 O(1)
5          for (int i=0; i<size; i++){                     Θ(n)
6                sum += ptr[i];
7                if (sum >= W/2){
8                      return ptr[i];
9                }
10         }
11         return -1;
12   }
```

Il costo in totale nel caso peggiore è la somma dei vari costi:

$$T(n) = \Theta(\mathrm{n}) + \Theta(\mathrm{nlogn}) + \mathrm{O}(1) + \Theta(\mathrm{n})$$
$$= \Theta(\mathrm{nlogn})$$

Nel caso medio il costo del for è $\mathrm{O}(n)$, ciò non cambia però il costo complessivo.
Nel caso migliore del for (quando viene eseguito una sola volta) il costo è $\Theta(1)$, ma vale lo stesso discorso del caso medio.

Dimostrazione di correttezza

La correttezza di MergeSort è quella vista a lezione.

Ora verrà dimostrato prima Sum Partitions poiché viene utilizzato da Weighted Median Sort.

Sum Partitions

Per dimostrare la correttezza di Sum Partitions, verrà utilizzata una dimostrazione per induzione sulla dimensione dell'array, cioè della differenza tra q e p.

Il predicato è la sommatoria dei q-p elementi che è uguale a un certo k.

$$P(n) = \sum_{i=p}^{q} A(i) = k$$

Caso Base: $size = 0 \rightarrow$
$p\ e\ q\ non\ possono\ essere\ uguali\ perche' implicherebbe\ la\ presenza\ di\ un\ elemento, quindi\ p = 0\ e\ q = -1$

$$siccome\ sum = 0.0\ all'inizio\ del\ metodo\ e\ il\ ciclo\ for\ non\ viene\ eseguito$$
$$\rightarrow\ viene\ restituito\ 0.0\ che\ e' corretto.$$

Passo induttivo: $(P(n) = k) \rightarrow (P(n+1) = k + A(q+1))$

$$\left(\sum_{i=p}^{q} A(i) = k\right) \rightarrow \left(\sum_{i=p}^{q+1} A(i) = k + A(q+1)\right)$$

Sommo al primo membro da entrambi i dati A(q+1)

$$\left(\sum_{i=p}^{q} A(i) + \mathrm{A(q+1)} = k + \mathrm{A(q+1)}\right) \rightarrow \left(\sum_{i=p}^{q+1} A(i) = k + A(q+1)\right)$$

$$\left(\sum_{i=p}^{q+1} A(i) = k + \mathrm{A(q+1)}\right) \rightarrow \left(\sum_{i=p}^{q+1} A(i) = k + A(q+1)\right)$$

CVD

Weighted Median Sort

Per dimostrare la correttezza di Weighted Median Sort, verrà utilizzata una dimostrazione per induzione sulla dimensione dell'array.

Per semplicità siccome la size = 0 implicherebbe che il risultato del metodo fosse -1, impossibile, partiremo da size = 1.

Caso Base: $size = 1 \rightarrow \left(W = \sum_{i=0}^{size-1} A(i)\ \ AND\ A\ e'\ ordinato\right) \rightarrow W = A(0)$

$sum = 0$
$inizia\ il\ ciclo\ for$
$sum = A(0) + A(1) + \cdots + A(size-1) \rightarrow sum = A(0)$

$if\ \left(sum \geq \frac{W}{2}\right) ritorna\ A[i]$

$Siccome\ sum = A(0)\ e\ W = A(0), \qquad e' sempre\ vero\ che\ A(0) \geq \frac{A(0)}{2}$

$Quindi\ ritorna\ A(0)\ che\ e' correttamente\ la\ mediana\ inferiore\ pesata$

Passo induttivo: $P(n) \rightarrow P(n+1)$

$size = n \rightarrow W = \sum_{i=0}^{n} A(i)\ \ e\ A\ e'\ ordinato$

$sum = 0, inizia\ il\ ciclo\ for$

$sum = A(0) + A(1) + \cdots$

$siccome\ sum\ e' la\ sommatoria\ di\ ogni\ elemento\ e\ a\ un\ certo\ punto\ verra'\ superato\ \frac{W}{2}$

$verra'\ restituito\ l'elemento\ che\ fa\ superare\ o\ e'\ uguale\ a\ \frac{W}{2}\ nella\ sommatoria$

$Che\ coincide\ con\ la\ definizione\ di\ media\ inferiore\ pesata\ fornita\ all'inizio\ della\ relazione$

La stessa cosa si applica quando size = n+1

Analisi dei tempi

Per svolgere l'analisi dei tempi si sono utilizzati gli algoritmi provvisti dalle dispense.
È stato fondamentale implementare l'algoritmo numero 9 poiché tramite quello si poteva effettuare la misurazione dei tempi.

```
1       void misurazione (void (**handler_of_methods)(double*, int, double*), int number_of_methods,
2                   double *array, int size,
3                   double t_min, int numero_campioni, double zalpa, double delta_input, double *seed){
4           double original_seed = *seed;
5           double t = 0;
6           double sum_2 = 0;
7           double cn = 0;
8           double delta=0, e=0;
9           do {
10              for (int i=1; i<=numero_campioni; i++){
11                  double m = tempo_medio_netto(handler_of_methods, number_of_methods, array, size, t_min,
seed);
12                  t = t + m;
13                  sum_2 += m*m;
14              }
15              *seed = original_seed;
16              cn += numero_campioni;
17              e = t / cn;
18              double s = sqrt(sum_2/cn-(e*e));
19              delta = (1/sqrt(cn))*zalpa*s;
20          } while (!(delta<delta_input));
21          printf("%5d\t%lf\t%lf\n", size, e, delta);
22      }
```

Al metodo misurazione viene passato un gestore di metodi che in base al metodo passato chiama uno o l'altro metodo, questo per l'algoritmo numero 5 che ha due varianti, quella che calcola il numero di ripetizioni necessarie senza e con l'algoritmo.

Il numero di metodi indica che ci sono quindi due funzioni da passare: prepara e start_algorithm.
Viene poi passato l'array, la sua dimensione, il tempo minimo, il numero di campioni, il valore della funzione di distribuzione normale (zalpha), il Δ in questo caso chiamato delta_input e infine il numero del seed.

Il metodo funziona in questo modo: viene eseguito l'algoritmo per numero_campioni volte, ogni volta si calcola il tempo medio netto e viene salvato in una variabile t.
Siccome l'algoritmo andava a modificare il numero del seed, esso è stato ripristinato ogni volta al suo valore originario per generare successivamente gli stessi valori.

Viene poi calcolata la varianza campionaria e il nuovo delta, il metodo viene ripetuto finché la misurazione non è precisa almeno quanto delta_input.

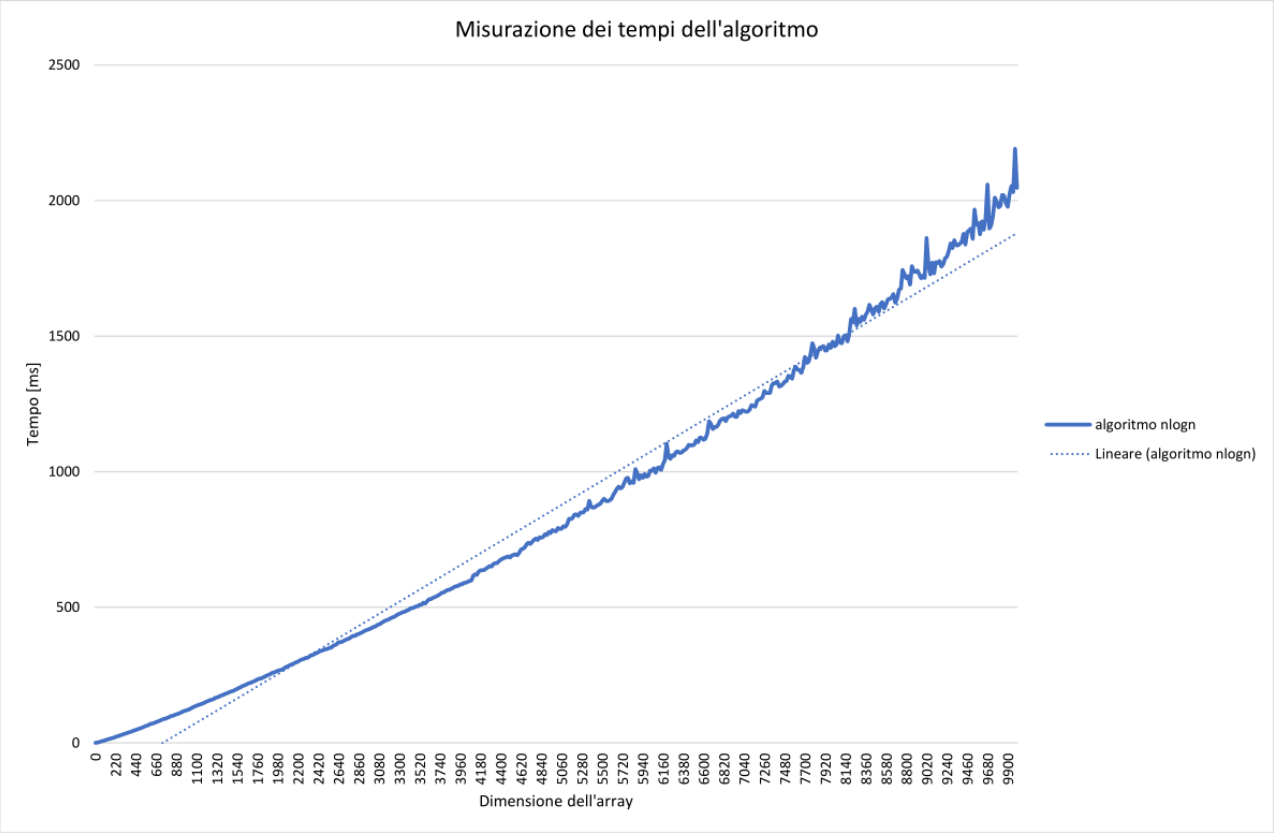
Per eseguire i test sono stati usati i seguenti valori:

- TOLLERANZA = 0.02
- VARIANZA = 0.02
- NUMERO_CAMPIONI = 10
- ZALPHA = 2.33
- SEED = 133769420

Le specifiche tecniche del computer utilizzato sono:

- CPU: i5 3570k
- RAM: 24 GB
- OS: Ubuntu 19.10

Il grafico che si è andato a ottenere tramite l'esecuzione di uno script in bash esegue la misurazione a partire da zero fino a diecimila con un incremento di venti sulla dimensione dell'input per ottenere un totale di cinquecento misurazioni.



Il grafico presenta degli errori verso la fine della misurazione dovuti all'esecuzione contemporanea di altri programmi nella macchina.

Il grafico segue un andamento asintotico simile a una funzione nlogn, quindi si può concludere che la misurazione è andata a buon fine.

Come compilare il progetto

Il progetto è stato testato su Ubuntu 19.10 ed è necessario avere installato clang.

Per compilarlo è stato scritto uno script per mettere insieme tutti i file.

```
1    clang -c sum.h sum.c
2    clang -c sort.h sort.c
3    clang -c weighted_median.h weighted_median.c sum.h sort.h
4    clang -c algorithm.h algorithm.c weighted_median.h
5    clang -c analisi_stima_dei_tempi.h analisi_stima_dei_tempi.c algorithm.h
6    clang -c main.c weighted_median.h algorithm.h analisi_stima_dei_tempi.h
7    clang -o main.out main.o algorithm.o analisi_stima_dei_tempi.o sort.o sum.o weighted_median.o -lm
```

Assicurarsi che lo script compile.sh abbia i permessi di lettura, scrittura ed esecuzione.

Per eseguire lo script basta scrivere sul terminale ./compile.sh oppure copiare ogni singola riga dello script ed eseguirne una alla volta.

Una volta compilato il file, viene generato un file main.out
Esso è il binario del progetto.

Riassumendo per compilare ed eseguire il progetto sono necessari questi comandi:

```
sudo chmod 755 compile.sh
./compile.sh
./main.out
```

E' noto che è presente un warning alla fine della compilazione, è voluto così.

Conclusione

Il calcolo della mediana pesata inferiore è stato fatto utilizzando un algoritmo di complessità:

$$\theta(nlogn)$$

Durante il progetto si sono riscontrati problemi nella misurazione dei tempi, poiché l'implementazione di getTime() in pseudocodice è stata tradotta in clock() in C.

Questa istruzione comporta una problematica, il tempo minimo rilevato era di 1 µs, il quale è fin troppo basso per una CPU che lavora in GHz quindi ns.