Università degli Studi di Udine

Relazione di Progetto

ALGORITMI E STRUTTURE DATI (MA0006)

Amos Cappellaro
matricola n°**134059**
`cappellaro.amos@spes.uniud.it`

31 Dicembre 2019

Indice

1 Introduzione

1.1 Compilazione ed esecuzione

Per compilare e mandare in esecuzione il progetto sono sufficienti i seguenti due comandi da terminale:

```
javac Progetto.java
java Progetto
```

Il programma, dopo aver avviato l'esecuzione, chiederà di fornire in input - da linea di comando - una sequenza di numeri razionali separati da una virgola e terminante con un punto. Una volta inserita e computata la sequenza, il programma restituirà il valore della *mediana (inferiore) pesata* nello standard output.

Viene illustrato di seguito un esempio di esecuzione del programma:

```
Input:
0.1 , 0.35 , 0.05, 0.1, 0.15 , 0.05, 0.2 .
Output:
0.2
```

1.2 Esposizione del problema

Sia S = $\{w_1,...,w_n\} \in \mathbb{Q}^+$ una sequenza di valori razionali positivi, chiamati anche *pesi*, e sia W la loro somma: W $= w_1 + ... + w_n = \sum_{i=1}^{n} w_1$. Viene posto il problema di trovare il peso w_k, detto *mediana (inferiore) pesata*, tale che

$$\sum_{w_i < w_k} w_i < W/2 \leq \sum_{w_i \leq w_k} w_i.$$

Si tratta pertanto di voler trovare quel peso w_k tale per cui:

- la somma dei pesi strettamente minori a w_k sia strettamente minore alla somma totale dei pesi dimezzata,
- e la somma dei pesi minori o uguali a w_k sia maggiore o uguale alla somma totale dei pesi dimezzata.

In altre parole, la **mediana pesata** w_k è quel peso che, una volta aggiunto alla somma di tutti i pesi più piccoli di w_k, essa diventerà maggiore della metà della somma totale dei pesi. Nel caso in cui si verificasse una situazione di *"parità"*, dove cioè la somma di tutti i pesi più piccoli di w_k compresa di w_k è pari alla metà della somma totale dei pesi, si ottengono due mediane pesate, di cui una è w_k. In questo frangente w_k viene chiamata **mediana *inferiore* pesata**. È necessario inoltre specificare che la mediana inferiore pesata consiste necessariamente in uno dei pesi forniti.

2 Soluzione proposta e correttezza

Acquisita una stringa dallo Standard Input, si esegue innanzitutto il *parsing* della stringa per inizializzare e popolare un array di tipo `double`. Questa procedura viene gestita dal metodo `importFromStdIn(String str)`. Viene poi eseguito il metodo `getWeightedMedian(double[] a, int p, int r)`, nel quale risiede l'algoritmo per trovare e restituire la *mediana (inferiore) pesata* da una sequenza di valori contenuti in `a`. Quest'ultimo utilizza un ulteriore algoritmo, chiamato `randomizedPartition(double[] a, int p, int r)`.

2.1 importFromStdIn

Questo metodo si occupa principalmente di eseguire il *parsing* della stringa fornita in input: nello specifico, elimina innanzitutto caratteri superflui come gli spazi ed il `'.'` al termine della stringa; dopodichè suddivide, attraverso il metodo `split()`, tale stringa ad ogni `','`. Solo in questo momento avviene la vera e propria conversione in `double` di ogni valore immagazzinato come `String`, grazie al metodo `parseDouble()`.

2.2 getWeightedMedian

A questo algoritmo, fondamentalmente strutturato a partire da una rivisitazione dell'algoritmo `QuickSelect`, viene assegnato il compito più importante: trovare e restituire la *mediana (inferiore) pesata.*

Algorithm 1 $getWeightedMedian(A, p, r)$

if $(p == r)$ **then** ▷ Caso base
    return $A[p]$
end if
$q \leftarrow$ randomizedPartition(A, p, r)

$leftWeight \leftarrow$ sumElements$(A, 0, q - 1)$ ▷ Somma pesi < q
$rightWeight \leftarrow$ sumElements$(A, q + 1, A.length - 1)$ ▷ Somma pesi > q
$totalWeight \leftarrow$ sumElements$(A, 0, A.length - 1)$ ▷ Somma totale dei pesi

if $(leftWeight/totalWeight < 0.5$ and $(leftWeight + A[q])/totalWeight \geq 0.5)$ **then**
    return $A[q]$
else
    if $(leftWeight > rightWeight)$ **then**
        $rightWeight \leftarrow$ rightWeight + A[q]/totalWeight
        return getWeightedMedian(A, p, q-1)
    else if $(leftWeight < rightWeight)$ **then**
        $leftWeight \leftarrow$ leftWeight + A[q]/totalWeight
        return getWeightedMedian(A, q+1, r)
    end if
end if

L'idea alla base dell'algoritmo è la seguente:

- si comincia con il caso base. Se l'indice di partenza e di arrivo sono uguali, la porzione di array $A[p..r]$ che stiamo considerando è composta solamente da un elemento. È naturale quindi concludere che si debba ritornare l'unico elemento a disposizione $A[p]$.
- Altrimenti avviene la chiamata a RandomizedPartition che partiziona l'array $A[p..r]$ in due subarray $A[p..q-1]$ e $A[q+1..r]$ tali che ogni elemento di $A[p..q-1]$ è minore o uguale a $A[q]$, che a sua volta è minore di ogni elemento in $A[q+1..r]$. L'elemento $A[q]$ viene comunemente chiamato ***pivot***. Vengono consecutivamente computati, grazie ad un banale metodo chiamato SumElements, i pesi dei subarray $A[p..q-1]$ e $A[q+1..r]$, e dell'array $A[p..r]$, e salvati rispettivamente nelle variabili *leftWeight*, *rightWeight* e *totalWeight*.

  A questo punto, se le condizioni della definizione vengono rispettate, abbiamo trovato la *mediana (inferiore) pesata*, corrispondente al *pivot* $A[q]$. Alternativamente, si può verificare una delle due situazioni: *leftWeight* è maggiore di *rightWeight*, oppure *leftWeight* è minore di *rightWeight*.

  - Nel primo caso, conoscendo la definizione di *mediana (inferiore) pesata*, possiamo intuire che il valore che stiamo cercando risiede in uno dei pesi minori o uguali al pivot. È possibile dunque escludere sia il pivot, sia tutti i pesi maggiori del pivot. Si può procedere pertanto incrementando *rightWeight* del peso relativo del pivot (*A[q]/totalWeight*), e ricercare la mediana pesata all'interno del subarray $A[p..q-1]$ chiamando ricorsivamente $getWeightedMedian(A, p, q-1)$.
  - Nel secondo caso, specularmente, intuiamo che il valore cercato risiede in uno dei pesi maggiori del pivot. Anche qui, dunque, possiamo escludere sia il pivot, sia tutti i pesi minori o uguali al pivot. Si può procedere pertanto incrementando *leftWeight* del peso relativo del pivot, e ricercare la mediana pesata all'interno del subarray $A[q+1..r]$ chiamando ricorsivamente $getWeightedMedian(A, q+1, r)$.

2.2.1 randomizedPartition

Un algoritmo chiave, di fondamentale importanza per GetWeightedMedian è senz'altro RandomizedPartition, una variante di Partition la cui correttezza - dimostrata attraverso la tecnica dell'invariante - è data per scontata in quanto vista a lezione. Si è deciso di utilizzare la versione randomizzata anziché quella classica (Partition) puramente per una questione di prestazioni, spiegata più in dettaglio nel capitolo successivo. Molto semplicemente RandomizedPartition si differenzia da Partition per il criterio di selezione del *pivot*, elemento attorno al quale partizionare l'array: se Partition seleziona sistematicamente come *pivot* l'ultimo elemento dell'array, RandomizedPartition ne sceglie uno in maniera randomizzata (usufruendo dell'*Algoritmo 8* degli appunti) e lo scambia di posizione con l'ultimo elemento nell'array. Da qui, il loro comportamento è il medesimo: ad uno ad uno, tramite l'utilizzo di due indici ausiliari i e j, viene confrontato ogni elemento con il *pivot* e riposizionato in base all'esito del confronto. All'ultimo passo, viene scambiata la posizione del *pivot* con quella dell'elemento ad indice $i+1$: il *pivot* viene, cioè, interposto tra gli elementi minori o uguali ad esso, raccolti alla sua sinistra, e gli elementi maggiori ad esso, raccolti alla sua destra.

Algorithm 2 *randomizedPartition*(A, p, r)

```
index ← random(p, r)                          ▷ Valore random tra p ed r
pivot ← A[index]
exchange A[index] with A[r]

i ← p − 1
for j = p to r − 1 do
    if A[j] ≤ pivot then
        i ← i + 1
        exchange A[i] with A[j]
    end if
end for
exchange A[i + 1] with A[r]
return i + 1
```

3 Complessità degli algoritmi

Si analizzerà ora la complessità dei metodi presenti nelle classi che compongono il progetto. Come convenzione, assunta in input una stringa, si indicherà con n il numero di valori (o *pesi*) individuati nella stringa, ovvero contenuti nell'array generato a partire da essa.

3.1 Classe RandomGenerator

get Questo metodo restituisce un numero random compreso tra 0 e 1, ed aggiorna il seme fornito nel momento in cui viene istanziato un oggetto di `RandomGenerator`. Tale metodo, dopo l'inizializzazione di alcune costanti e di alcune variabili, esegue un numero pressoché limitato di operazioni con costo costante; più nello specifico, esegue anche un'operazione che utilizza il metodo `ceil` della classe `Math`, anch'esso di costo costante. La complessità di questo metodo risulta quindi $\Theta(1)$.

3.2 Classe Progetto

importFromStdIn Questo metodo esegue il *parsing* della stringa in input. Indichiamo con $|s|$ la lunghezza della stringa s, ovvero il numero di caratteri che la compongono. Innanzitutto esso scandisce la stringa *"pulendola"* da caratteri superflui: grazie al metodo `substring` della classe `String`, che copia la parte di stringa a cui siamo interessati con complessità lineare $O(|s|)$, otteniamo una nuova stringa senza il punto che termina la sequenza di valori inseriti; dopodiché, tramite il metodo `replaceAll` della classe `String` vengono eliminati tutti gli spazi presenti nella stringa, con complessità lineare $\Theta(|s|)$; infine, per mezzo del metodo `split` della classe `String`, la stringa viene separata nei valori che la compongono separati dal carattere '`,`', ottenendo quindi un array di `String`, ancora una volta con complessità $\Theta(|s|)$. Complessivamente, quindi, la stringa viene scandita tre volte con complessità $\Theta(|s|)$.

A questo punto avviene il vero e proprio processo di *parsing*, convertendo i valori da `String` a `double`: ciò è possibile chiamando il metodo `parseDouble` della classe `Double` per ogni singolo valore da convertire. Anche qui, la complessità dipende dalla lunghezza s - da intendere questa volta come lunghezza della stringa che compone un singolo valore, perciò $\Theta(|s|)$.

Si può dunque concludere che, complessivamente, il metodo in analisi ha complessità $\Theta(|s|)$.

swap Questo banale metodo scambia due elementi di posizione in un array di tipo `double`. Essendo composto da sole tre operazioni con costo costante, esso ha complessità $\Theta(1)$.

sumElements Questo metodo ha il semplice compito di sommare i valori (o *pesi*) degli elementi in un array di tipo `double`, dati un indice di partenza p e un indice di arrivo r. La complessità, pertanto, dipende dal numero di elementi tra p ed r da sommare. Questo metodo ha perciò complessità $\Theta(r\text{-}p)$. Più specificamente, se questo metodo viene utilizzato per trovare la somma di tutti gli n valori (o *pesi*) degli elementi presenti nell'array, chiaramente si ha una complessità $\Theta(n)$.

randomizedPartition Questo metodo ripartisce l'array raggruppando i valori (o *pesi*) minori o uguali da una parte ed i valori (o *pesi*) maggiori dall'altra, scelto un valore dell'array detto *pivot*. Il metodo in analisi fa uso del metodo `floor` della classe `Math`, di costo costante, ed il metodo `get` della classe `RandomGenerator` - per assegnare ad un valore random dell'array il ruolo di *pivot* - anch'esso di costo costante, come visto in precedenza. È comunque necessario, in qualsiasi caso, passare su ogni elemento dell'array: il metodo ha pertanto complessità $\Theta(n)$.

getWeightedMedian Questo metodo svolge il compito più importante, trovare il valore corrispondente alla *mediana (inferiore) pesata*.

L'algoritmo che risiede in questo metodo risente della scelta del *pivot*: se, costantemente, vengono scelti dei *"buoni" pivot* - tali per cui l'insieme nel quale cercare la mediana pesata si restringe sensibilmente - l'insieme di ricerca si dimezza ad ogni iterazione (o chiamata ricorsiva); si può dunque dire, per induzione, che in questo caso si otterrebbe una complessità lineare $O(n)$, data dalla complessità lineare di ogni iterazione per un numero costante di iterazioni. Nel caso vengano invece scelti costantemente *pivot "cattivi"* - tali per cui l'insieme di ricerca della mediana pesata si restringe solamente di un elemento alla volta - si ricadrebbe nel caso pessimo, di complessità $O(n^2)$. Ciò accade, ad esempio, con l'algoritmo QUICKSORT, quando utilizza la subroutine classica PARTITION per il partizionamento, e l'array fornito in input è già completamente ordinato.

Nasce da qui, dunque, la scelta di usare la versione randomizzata di PARTITION, chiamata RANDOMIZEDPARTITION, per il metodo in analisi. Abbiamo capito che utilizzare PARTITION può risultare, in situazione di particolari input, dispendioso in termini di complessità temporale. Grazie alla versione randomizzata RANDOMIZEDPARTITION, invece, risulta estremamente improbabile la scelta di *pivot "cattivi"* ad ogni iterazione. Tanto è vero che considerare ancora possibile il caso pessimo per il metodo in analisi GETWEIGHTEDMEDIAN, fondamentalmente, perde di senso. Altra motivazione a sostegno della scelta, inoltre, sta nel fatto che RANDOMIZEDPARTITION non perde di prestazioni rispetto alla versione classica PARTITION.

Oltre alla scelta cruciale di RANDOMIZEDPARTITION, il metodo utilizza il metodo SUMELEMENTS, di complessità $\Theta(n)$, per computare le somme delle partizioni e della somma totale dei *pesi*. Vi è poi un numero limitato di operazioni con costo costante. È possibile concludere, pertanto, che il metodo ha complessità $\Theta(n)$.

4 Analisi empirica dei tempi

Considerando l'analisi di complessità formulata nel capitolo precedente, abbiamo osservato che per trovare la *mediana (inferiore) pesata* dato un set di n valori (o *pesi*), si abbia un costo $\Theta(n)$. Constatato che - scelto un *pivot* per la partizione in maniera casuale - perde sostanzialmente di senso ragionare in termini di caso pessimo, risulta utile discutere la complessità considerando i casi ottimo e medio.

- **Caso ottimo**: il *pivot* selezionato in maniera casuale nella prima chiamata di RANDOMIZEDPARTITION coincide proprio con il valore cercato, la *mediana (inferiore) pesata*. Il set di valori viene scandito interamente una sola volta; si avrà perciò una complessità pari a $\Theta(n)$.

- **Caso medio**: ad ogni chiamata ricorsiva di GETWEIGHTEDMEDIAN, il set di *pesi* si riduce, mediamente, a volte di un fattore pari a $\frac{1}{2}$ (qualora venga scelto un *"buon" pivot*), altre volte solamente di un elemento (nel caso in cui venga scelto un *"cattivo" pivot*). Questo comportamento ci consentirà di ottenere una complessità nell'ordine di quella che si verifica nel caso ottimo, ovvero $\Theta(n)$.

Ciò che ci si aspetta sarà pertanto una complessità lineare sia per il caso ottimo che per il caso medio. In entrambi i casi sono state fatte le misurazioni dei tempi - tramite l'utilizzo dell'`Algoritmo 9` degli appunti - di computazione della *mediana (inferiore) pesata* dati degli input con dimensioni $1 \leq |input| \leq 120$, contenenti cioè da 1 a 120 valori razionali positivi.

Osservando la selezione casuale del pivot da parte di RANDOMIZEDPARTITION, non è possibile stabilire a priori quando si possa verificare il caso ottimo piuttosto che il caso medio: le misurazioni che andremo ad ottenere, quindi, potranno essere interpretate in termini di *caso medio*. Per le misurazioni con l'`Algoritmo 9` sono stati dati in input, dunque, gli stessi valori sia per il caso ottimo che per il caso medio: in particolare, per il parametro $tMin$ si è scelto un valore in funzione del rapporto tra *granularità del sistema* δ, ed un *errore massimo* tollerabile Δ. Osservata una *granularità del sistema* δ mai eccedente i 3 millisecondi (misurata numerose volte, grazie all'`Algoritmo 4` degli appunti), e scelto un *errore massimo* $\Delta = 0.02$, si è deciso di assegnare al parametro in input $tMin$ un valore arbitrario di $150ms$, il valore più piccolo utilizzabile in modo da risultare maggiore o uguale a tale rapporto,

Le misurazioni dei tempi sono state effettuate utilizzando un *MacBook Pro* con processore *Intel Core i5 dual-core* a 3,1*GHz*, 8*GB* di memoria RAM e sistema operativo *macOS Catalina* (versione 10.15.2).

Viene presentato di seguito il grafico ottenuto (i tempi rappresentati sull'asse delle ordinate vanno intesi in *millisecondi*, mentre sull'asse delle ascisse è riportato $|input|$, la dimensione dell'input):

.

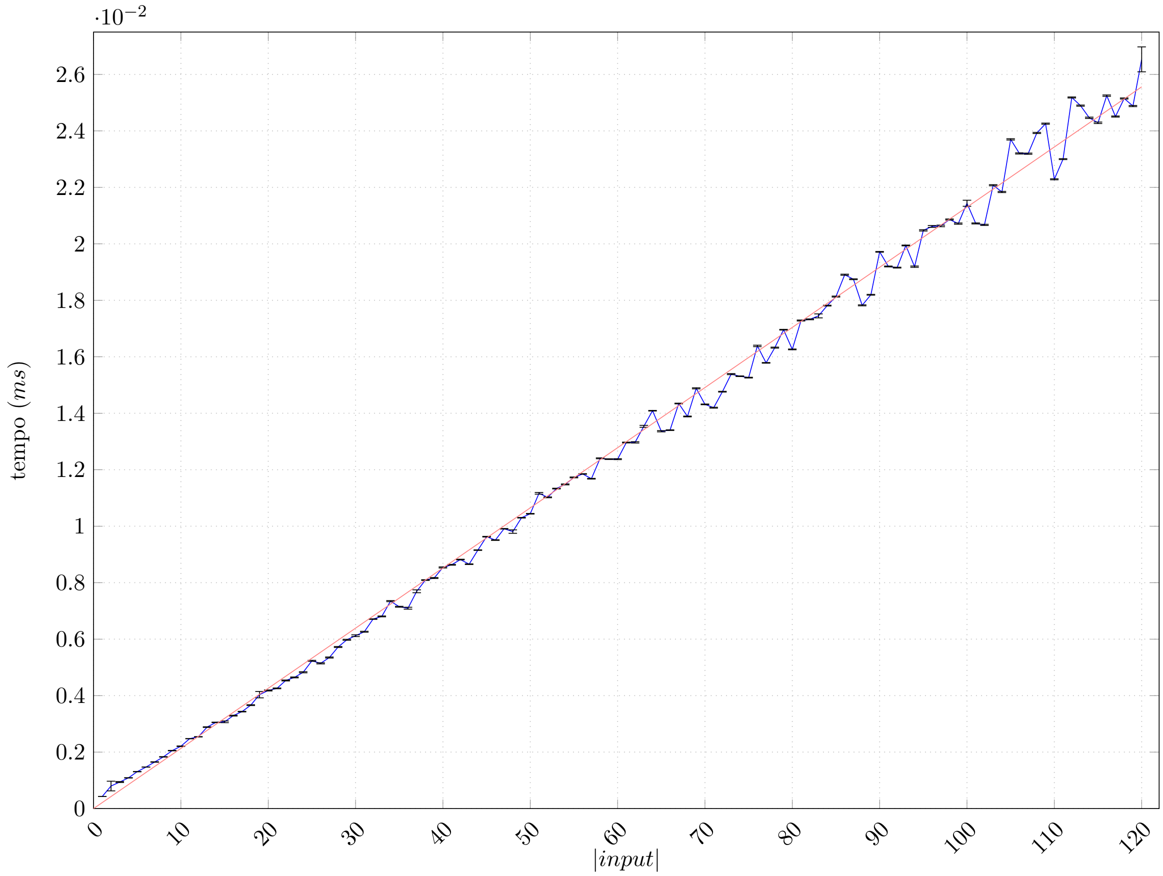

Figura 1: Caso Medio ($y = 2.13 \cdot 10^{-4} \cdot x$)

5 Conclusione

L'analisi empirica dei tempi, nell'insieme, si è dimostrata piuttosto coerente ed in linea con i valori dei costi teorizzati nell'analisi asintotica della complessità. Si nota infatti un andamento pressoché lineare, come ipotizzato: il grafico, infatti, ricalca in maniera relativamente accurata una linea di tendenza (nel grafico evidenziata in rosso) con un andamento riconducibile alla funzione $y = 2.13 \cdot 10^{-4} \cdot x$. Si può dire inoltre che, avendo il termine di primo grado x un coefficiente piuttosto piccolo, l'algoritmo dia complessivamente delle buone prestazioni. Si può concludere che i valori acquisiti in fase di analisi empirica dei tempi, grazie all'`Algoritmo 9`, siano affidabili.

Appendice A Tabella delle misurazioni

|input|	Tempo	Errore	|input|	Tempo	Errore
1	4.306518379645777E-4	0.0	61	0.012963392660496434	1.4150544728575535E-5
2	7.98121346102149E-4	1.706355990755906E-4	62	0.01296930166430764	2.3560288241371484E-5
3	9.393113093880953E-4	1.7046185546800508E-5	63	0.01353714798777111	3.468963822831551E-5
4	0.0010882107497191236	6.131817924628295E-6	64	0.01409006834867891	1.0804152212845825E-5
5	0.0013072177478078008	5.4965614986448884E-6	65	0.013363970681754397	2.1693968630137197E-5
6	0.001468629160616181	1.4235229100136477E-6	66	0.013401658924980176	1.1905958222871978E-5
7	0.00164905334135465	7.708887703449044E-6	67	0.014345391145409093	1.1638383873263139E-5
8	0.001831085381169046	7.477786399631685E-6	68	0.013886603314829679	1.322754096407888E-5
9	0.0020526568220643002	7.547143719327897E-6	69	0.014883827365191674	1.848488408381786E-5
10	0.0022074960895802076	9.678194764436242E-6	70	0.014314854341557008	1.2375830197724695E-5
11	0.002475088215793429	3.962289675454076E-6	71	0.014196689300759355	1.552154208956748E-5
12	0.002542534106766043	5.991274663497088E-6	72	0.014763849333365747	1.3204216185780497E-5
13	0.0028852695647399824	1.1072274834373513E-5	73	0.015385725173642506	1.844705932307563E-5
14	0.0030517477574799915	8.533582973415269E-6	74	0.015311915612431276	1.0963658212128265E-5
15	0.0030723501443008566	3.064749226366328E-5	75	0.015262129727815198	1.2803024910394725E-5
16	0.0032912472870287113	1.4080199921099497E-5	76	0.016384545265605604	2.573983726330814E-5
17	0.003433759883535305	1.0007109732330221E-5	77	0.015785120203944413	1.0957215923401133E-5
18	0.0036625153510626324	1.5344336797507852E-5	78	0.01632393937942677	1.7591899661835042E-5
19	0.004034470001335836	1.1258057386275238E-4	79	0.016958998585144363	1.2398178254435522E-5
20	0.004182898613410163	1.7246956838090057E-5	80	0.016262082777157193	1.3539953913758172E-5
21	0.00425685546375848	1.3152625574640746E-5	81	0.017284913264031525	1.7005717423159404E-5
22	0.004536422856997315	1.6034696317555345E-5	82	0.01732643609894896	1.4768558904068727E-5
23	0.004645383366416834	1.5507446547590635E-5	83	0.01744925797844096	7.051236103773932E-5
24	0.004827461313347186	1.916829421116868E-5	84	0.0178125188567637	1.3686157711644243E-5
25	0.005232743104288836	1.7457686485028662E-5	85	0.018131831741266785	1.7539601058306258E-5
26	0.005144223364858465	1.98327253120201E-5	86	0.01890605059469891	1.897911906116737E-5
27	0.0053494924614879	1.8980743118479056E-5	87	0.018746197464881646	1.4520187227813802E-5
28	0.0057220224907867795	1.7788415515480553E-5	88	0.017822791078910407	1.7348429410264112E-5
29	0.00597736325255259	1.6937075283429607E-5	89	0.018196676976839635	1.4363768704286148E-5
30	0.006121927805168206	3.16180001973016E-5	90	0.019716839578995723	1.3212809170481784E-5
31	0.006262236109938159	1.6208282816191122E-5	91	0.01919892043713319	1.4537109905236118E-5
32	0.0067101502170796795	9.835999890357076E-6	92	0.019157843268457146	1.4336323049069926E-5
33	0.006806157053655505	1.5186601895674933E-5	93	0.01994259274622121	1.3969116622170288E-5
34	0.007346365044770296	1.8857733844242913E-5	94	0.01919084493193094	2.279821911677109E-5
35	0.007147372954270763	1.5414971132336512E-5	95	0.020477632867597906	2.318759211868338E-5
36	0.007095626935716052	3.419904644179636E-5	96	0.020616441131743355	3.282869035680797E-5
37	0.0076955757422507015	5.137781857666522E-5	97	0.02063439643217131	2.776096207404334E-5
38	0.008092789404891332	1.2273306374021636E-5	98	0.020861981950525268	1.907502927261536E-5
39	0.008165124011397691	1.7986177270516893E-5	99	0.020715883087274813	2.22447999152138E-5
40	0.008540421199347819	1.9486549175413254E-5	100	0.021437121931084723	1.0679127862031999E-4
41	0.008633723741126758	1.4605148622459476E-5	101	0.020723813508721732	1.7902876059190065E-5
42	0.008818976110251035	1.1330445530239464E-5	102	0.020672055091920435	1.998660848993229E-5
43	0.008654745980952745	1.1597245021609324E-5	103	0.022076063447386662	1.935931873884863E-5
44	0.009153126042274072	9.750580627585528E-6	104	0.021836424248773415	2.1726064284839667E-5
45	0.009629094858476076	1.1004713793242271E-5	105	0.02370040203124054	2.3102601011428284E-5
46	0.009508353558481498	1.0962638051052495E-5	106	0.02320235579574732	2.052660257278428E-5
47	0.009909452469226315	8.896974661909225E-6	107	0.023192356311703983	2.0914011229065783E-5
48	0.009809700899104117	6.13524822689609E-5	108	0.023926798503153154	2.091715446334925E-5
49	0.010299905767778061	9.969247358264345E-6	109	0.0242554567670874	1.956645830018217E-5
50	0.010440828747127597	1.0751948980314085E-5	110	0.022286172304570705	1.758191426179367E-5
51	0.011164969224037468	3.0324909968664642E-5	111	0.022999468494174904	1.6459851193427116E-5
52	0.01102261966012957	1.4603819966718522E-5	112	0.025184390489545194	1.8376745957600477E-5
53	0.011332890319389733	1.2176566562568706E-5	113	0.02489391679634609	2.0191592891830873E-5
54	0.011480920303565163	1.598794749321532E-5	114	0.02446395816539252	2.271536323893958E-5
55	0.011728717209774371	1.4975053422337722E-5	115	0.024285772663148006	2.628189653505318E-5
56	0.011850469525417332	9.881866494736377E-6	116	0.02524860959149752	2.3666628409888436E-5
57	0.011682019351113274	1.2122870776458265E-5	117	0.024508048659612855	1.8949901993140732E-5
58	0.012405108158335103	1.532608564611262E-5	118	0.025148671321603137	1.9457425896526878E-5
59	0.012375471346920816	1.2487300153671743E-5	119	0.02488024967185727	2.144035716883102E-5
60	0.012374169575455563	1.6442461635378876E-5	120	0.026534543045230143	4.4143956581477027E-4