Zestawienie empirycznej ilości porównań algorytmu Merge-Sort z wynikiem Master Theorem

Kacper Pawłowski

19 sierpnia 2012

Streszczenie

W tym raporcie zostały zebrane wyniki badań dotyczących pomiarów ilości porównań algorytmu Merge-Sort poprzez wykonanie 100 sortowań dla każdego n z zakresu $1 - 10^5$. Zebrane wyniki zostały porównane z wynikiem otrzymanym poprzez zastosowanie twierdzenia o rekurencji uniwersalnej. Opracowano także wzór dla kresu dolnego i górnego ilości porównań.

1 Wprowadzenie

Definicja 1.1 *(Divide and conquer)*
Metoda algorytmiczna polegająca na podziale problemu algorytmicznego na mniejsze podproblemy i rozwiązaniu ich.

Definicja 1.2 *(Big O-Notation)*

$$f(x) = O(g(x)) \leftrightarrow (\exists c > 0)(\exists N \in \mathbb{N})(\forall n > N)(f(n) \leq c \cdot g(n))$$

Sortowanie przez scalanie jest idealnym przykładem algorytmu opartego na technice "divide and conquer" ("dziel i zwyciężaj"). Zbiór liczb zostaje podzielony rekurencyjnie na mniejsze podzbiory, które następnie przy bardzo małych zbiorach są z łatwością sortowane i scalane. W algorytmach tego typu do obliczenia złożoności obliczeniowej nieoceniona zdaje się znajomość Master Theorem (twierdzenie o rekurencji uniwersalnej). Pseudokod sortowania przez scalanie jest dostępny w wielu publikacjach, między innymi w pozycji [CLRS12]. W tym tekście znajduje się najważniejsza część użytej w pomiarach implementacji, którą można (a nawet trzeba!) zoptymalizować celem uzyskania lepszej wydajności.

Raport jest zestawieniem empirycznie zmierzonej średniej ilości porównań przez algorytm z wynikiem wyliczonym. Merge-sort będąc algorytmem sortowania opartym na porównywaniu elementów ma złożoność większą bądź równą $O(n \log n)$. Wynika to z faktu, że w drzewie porównań istnieje $n!$ liści, a więc najkrótsza ścieżka wynosi w tym drzewie $O(\log(n!))$, co daje oszacowanie ilości porównań. Wiążąc ze wzorem Stirlinga jest to $O(n \log n)$ porównań.

2 Równanie rekurencyjne

Liczba porównań przy scalaniu n elementów wynosi $O(n)$. Każdy zbiór jest dzielony na dwa mniejsze (w idealnym przypadku na dwa równe). Równanie rekurencyjne dla ilości porównań w tym sortowaniu można, więc zapisać jako:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Rozwiązanie poniżej rekurencji umożliwia twierdzenie o rekurencji uniwersalnej (poniżej zacytowane z jednej z najlepszych publikacji związanych z analizą algorytmów):

Twierdzenie 2.1 *(Master Theorem [SD10])*
Jeśli $T(n) = aT(\lceil \frac{n}{b} \rceil) + O(n^d)$ *dla pewnych stałych* $a > 0, b > 1$ *oraz* $d \geq 0$*, to:*

$$T(n) = \begin{cases} O(n^d), & gdy\ d > log_b a, \\ O(n^d log n), & gdy\ d = log_b a, \\ O(n^{log_b a}), & gdy\ d < log_b a \end{cases}$$

Zmienne dla Master Theorem: $a = 2, b = 2, d = 1$

Otrzymany wzór zwarty rekurencji:

$$T(n) = O(n \log_2 n)$$

Można ograniczyć z góry liczbę porównań dla zbioru n-elementowego. Liczba porównań przy scalaniu dwóch list, które zawierają w sumie n elementów wynosi maksymalnie $n - 1$. Rekurencja dla ograniczenia górnego liczby porównań:

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n - 1$$

Ograniczenie dolne polega na określeniu minimalnej liczby porównań dla scalania dwóch list, z których suma elementów wynosi n. W takim wypadku występuje $\lfloor \frac{n}{2} \rfloor$ porównań. Otrzymana rekurencja dla ograniczenia dolnego liczby porównań:

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \lfloor \frac{n}{2} \rfloor$$

Taki przypadek będzie zachodził dla wszystkich równych elementów.

3 Wykonanie symulatora

Symulator sortowania przez scalanie został wykonany w języku programowania C++. Za zliczanie ilości porównań dla poszczególnych elementów zbioru odpowiada specjalnie napisana klasa "element", w której przeciążono operatory porównania celem zebrania danych statystycznych. Mergesort sortuje w rzeczywistości obiekty typu *element*, których kluczami są wartości elementów dla zbioru sortowania. Wartości te są losowane z rozkładu jednostajnego.

3.1 Kod źródłowy klasy *element*

Kod klasy *element*:

```
1  #include<cstdlib>
2  #include<iostream>
3
4  using namespace std;
5
6  template< class T > class element
7  {
8      private:
9          T value;
10         int compare;
11
12     public:
13         element<T>();
```

```
14          void setValue(T);
15          void resetCounter();
16          T getValue();
17          int howCompare();
18          void inc();
19
20          bool operator<(element<T>&);
21          bool operator>(element<T>&);
22  };
23
24  template< class T > element<T>::element()
25  {
26      this->resetCounter();
27  }
28
29  template< class T > void element<T>::setValue(T val)
30  {
31      this->value = val;
32  }
33
34  template< class T > T element<T>::getValue()
35  {
36      return this->value;
37  }
38
39  template< class T > void element<T>::resetCounter()
40  {
41      this->compare = 0;
42  }
43
44  template<class T > bool element<T>::operator>(element<T>& e)
45  {
46      this->inc();
47      e.inc();
48
49      return (this->value > e.getValue());
50  }
51
52  template<class T > bool element<T>::operator<(element<T>& e)
53  {
54      this->inc();
55      e.inc();
56
57      return (this->getValue() < e.getValue());
58  }
59
60  template<class T > int element<T>::howCompare()
61  {
62      return this->compare;
63  }
64
65  template<class T > void element<T>::inc()
66  {
67      ++this->compare;
68  }
69
70  template< class T > void swap2(T& a, T& b)
71  {
72      T c = a;
73      a = b;
74      b = c;
75  }
```

element.cpp

3.2 Kod źródłowy funkcji sortujących - typowa implementacja

Kod funkcji `merge()` i `mergesort()`:

```
1  void merge(int p, int s, int k, element<int>* e)
2  {
3
4      if(k-p == 1)
5      {
6          if(e[k] < e[p])
7              swap2(e[k], e[p]);
8      }
9
10     else if(k-p > 1)
11     {
12
13         element<int>* tmp = new element<int>[k-p+1];
14         int i=0, j=0, i_w=s-p+1, j_w=k-s;
15
16         while(i < i_w && j < j_w)
17             tmp[i+j] = ((e[i+p] < e[s+1+j]) ? e[p+(i++)] : e[s+(j++)+1]);
18
19         while(i < i_w || j < j_w)
20             tmp[i+j] = ((j < j_w) ? e[s+1+(j++)] : e[p+(i++)]);
21
22         for(int z=0; z < (i_w + j_w); z++)
23             e[z+p] = tmp[z];
24
25         delete[] tmp;
26
27     }
28
29 }
30
31
32 void mergeSort(int p, int k, element<int>* e)
33 {
34
35     int s = (p+k)/2;
36
37     if(k > p)
38     {
39         mergeSort(p, s, e);
40         mergeSort(s+1, k, e);
41         merge(p, s, k, e);
42     }
43
44 }
```

merge.cpp

4 Wyniki otrzymane z symulatora

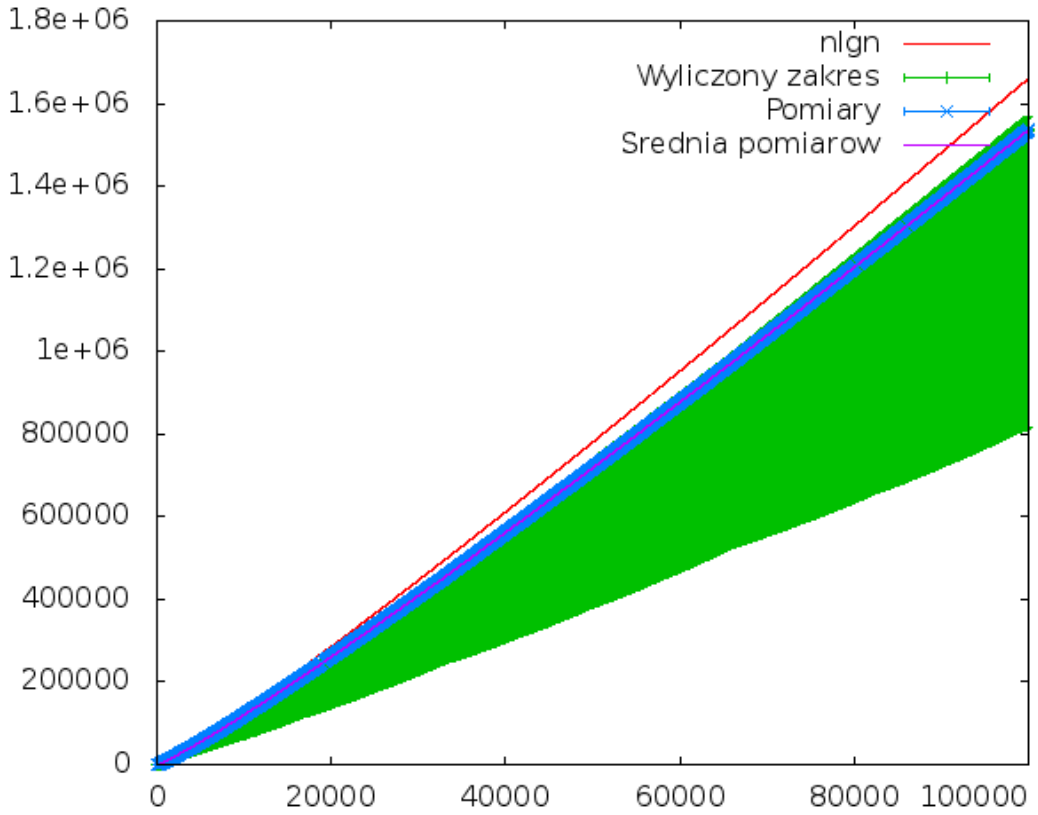
4.1 Tabela z danymi

n	nlg_2n	Średnia	Minimum	Maximum	Wyliczone min.	Wyl. max.
1	0	0	0	0	0	0
2	2	1	1	1	1	1
3	4,755	2,69	2	3	2	3
4	8	4,72	4	5	4	5
5	11,61	7,03	5	8	5	8
10	33,219	22,55	19	25	15	25
50	282,193	221,16	214	231	133	237
64	384	305,33	294	313	192	321
100	664,386	541,92	528	555	316	573
128	896	736,04	720	749	448	769
200	1528,771	1281,2	1256	1300	732	1345
256	2048	1726,28	1704	1748	1024	1793
500	4482,892	3854,69	3831	3892	2216	3989
512	4608	3961,22	3917	3992	2304	4097
1000	9965,784	8705,65	8649	8742	4932	8977
1024	10240	8948,95	8902	9003	5120	9217
2000	21931,569	19412,75	19356	19464	10864	19953
2048	22528	19942,09	19866	20007	11264	20481
5000	61438,562	55231,5	55138	55308	29804	56809
6400	80920,68	72958,62	72826	73088	39424	75009
10000	132877,124	120444,22	120291	120572	64608	123617
16384	229376	208647,89	208479	208775	114688	212993
20000	285754,248	260904,72	260617	261181	139216	267233
30000	446180,246	408593,62	408352	408812	219504	417233
32768	491520	450077,01	449797	450296	245760	458753
40000	611508,495	561800,48	561499	562102	298432	574465
50000	780482,024	718201,34	717794	718570	382512	734465
60000	952360,493	877184,88	876799	877511	469008	894465
65535	1048558,557	965695,11	965365	966061	524272	983025
65536	1048576	965712,23	965323	966064	524288	983041
65537	1048593,443	965725,66	965397	966209	524289	983058
70000	1126654,711	1038959,94	1038682	1039415	555200	1058929
80000	1303016,99	1203583,32	1203077	1204149	636864	1228929
90000	1481187,364	1369463,42	1369038	1369870	723680	1398929
99998	1660927,943	1536326,52	1535794	1536835	815005	1568895
99999	1660945,995	1536367,03	1535944	1536819	815014	1568912
100000	1660964,047	1536344,66	1535965	1537023	815024	1568929

Tablica 1: Wycinek tabeli z wynikami badań

Szczegółowe wyniki pomiarów można znaleźć pod adresem:
`https://dl.dropbox.com/u/35548543/merge/log.ods`.

4.2 Wykres



5 Uwagi i wnioski

- Dla każdego n losowane były liczby z przedziału od 1 do 40000 z rozkładu jednostajnego.
- Oprócz wyniku w Big O-Notation opracowano również ograniczenie dolne i górne ilości porównań dla zbioru n-elementowego.
- Wynik otrzymany przy użyciu Master Theorem jest wynikiem w notacji "wielkie O". Porównując z otrzymanymi wynikami okazuje się, że liczba porównań zawsze jest mniejsza od $nlog_2n$, z tego powodu, że liczba porównań przy scaleniu n elementów należy do przedziału $[\lfloor \frac{n}{2} \rfloor, n-1]$. Stała c zbiega do wartości bliskiej 1.
- Średnia porównań jest znacznie wyższa od wyliczonej wartości minimalnej i (stosunkowo) niewiele mniejsza od maksymalnej.
- Początkowo klasa *element* zliczała osobno porównania, dla których dany element okazał się mniejszy, większy lub równy, lecz dla wymogów tego zadania było to rozwiązanie zbędne, a przy tym bardzo czasochłonne.
- Łatwy overloading operatorów spowodował wybrane do tej implementacji języka programowania C++. Dla osiągnięcia większej wydajności powinno się jednak wykorzystać "czysty" język C wybierając inny mechanizm zliczania porównań.
- Narzucone wymagania (symulowanie sortowania dla każdego n z zakresu 1 - 10^5 po 100 razy) powodują, że zbieranie danych przez symulator trwa bardzo długo (nawet kilka tygodni).

Lepszym pomysłem od zastosowanego byłoby programowanie rozproszone. Zdecydowano, że część wyników będzie zbierana na innych maszynach (ułatwiały to dane wejściowe, gdzie definiowało się zakres i ilość sortowań).

- W aplikacji, dla osiągnięcia lepszej wydajności, powinno się użyć wątków. Uruchamiając program w dwóch wywołaniach dla innych danych wejściowych (zakres i ilość sortowań) osiągane były znacznie lepsze rezultaty.

Literatura

[CLRS12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Wprowadzenie do algorytmów*. Wydawnictwo Naukowe PWN, 2012.

[SD10] Umesh Vazirani Sanjoy Dasgupta, Christos Papadimitriou. *Algorytmy*. Wydawnictwo Naukowe PWN, 2010.