# Expected Number of remaining elements in the other array when the elements in one of the array is exhausted in merge algorithm

## Analysis

Problem:

Given two sorted arrays ***A1*** and ***A2***, which contain elements ***m*** and ***n*** respectively, merge them arrays into one sorted array. What's the expected number of elements in the other array when elements in one of the arrays is exhausted? Assume that all the elements are distinct and the elements and order of elements are random.

Solution:

Without loss of generality, we assume that . The possible number of remaining elements in ***A2*** is :

**n, n – 1, …, n – m**

There are ***C(m, m + n)*** ways to distribute ***(m + n)*** elements into the two subarrays ***A1*** and ***A2***.

If the number of remaining elements in ***A2*** is ***(n – i)***, then the ***(m – i)***th smallest element, say ***a***, must resides in ***A1***, and all the other elements in ***A1*** must be less than ***a***. There are ***C(m + i – 1, m -1)*** such cases. That is the possibility that the number of remaining elements in ***A2*** is ***(n – i)*** is :

So the expected number of remaining elements in ***A2***, denoted by , is:

By similar reasoning, the expected number of remaining elements in ***A1***, denoted by , is:

We now consider the special case when ***n = m + 1***.

Under this case, the expected number of remaining elements in ***A1*** and ***A2***, denoted by , is :

We will now calculate the value of .

Define as:

and as :

And we have the following :

Note that .

This indicates that is less than 2, and greater than 1.

# Expected number of successful optimization test

## Analysis

Problem:

Given an array where the elements in it are distinct, we will use an optimized version of merge sort to sort it. Two optimization strategies will be used.

1. ***cutoff*** value : If the number of elements is less than the cutoff value, then insertion sort will be used

2. avoiding merge: After we have sorted the left subarray and the right subarray. If the last element in the left subarray is not greater than the first element in the right subarray, then the array is sorted and no merge is needed.

The question is : what's the expected number of the times merge is avoided? Assume that the elements in the array are random and the order is also random.

Solution:

Let's say we have an array ***A***, the number of elements in it is ***N***.

Let denote the expected number of the times merge is avoided when we are sorting an array with ***N*** elements. The possibility that all the resides in the first half and the other elements resides in the last half is . It's not hard to derive the recursive formula:

.

We will look at the base case. When the number of elements is one less the cutoff value, insertion sort will be used, that is:

, when .

And .

The exact value of is hard to calculate as far as my knowledge is concerned. Only the example with ***cutoff*** equals to 10 will be given. And only the case which the number of elements in the array are , where , will be considered.

Then we have = 1 / 462, = ,...,.

## Codes

The codes will be written in C++, and some features of C++11 is used. The version of c++ compiler that support C++11 features is required. G++4.8 support almost all of the new C++11 features.

**#include** <map>

**#include** <random>

**#include** <algorithm>

**#include** <iostream>

/\*

\* MergeAnyalysis.cpp

\*

\* Created on: Dec 29, 2013

\* Author: Stiff Liu

\*/

**class** MergeAnaysis {

std::map<**unsigned** **int**, std::pair<**unsigned** **int**, **unsigned** **int**>> statistics;

**unsigned** **int** optimizationCount = 0;

**bool** val1 = **true**;

**public**:

/\*\*

\* insertion sort

\* C++ version of Algorithm 2.2 in book "Algorithms 4th edition".

\* @param array The array to sort.

\* @param n Number of elements in the array.

\* @param comparator The "less" relation between two elements in the array.

\*

\* best-case cost of this algorithm is O(n), such as when the array is already sorted.

\* average-case cost of this algorithm is O(n \* n) if the elements in the input array

\* aren't all identical and are randomly ordered.

\* worst-case of this algorithm is O(n \* n), such as when the array is in reversed order.

\*/

**template**<**class** **T**, **class** **Comparator**>

**void** **insertionSort**(**T** a, **unsigned** **int** n, **Comparator** comparator) {

**for** (**decltype**(n) i = 1; i < n; i++) {

**for** (**auto** j = i; j > 0 && comparator(a[j], a[j - 1]); j--) {

std::swap(a[j], a[j - 1]);

}

}

}

/\*\*

\* Merge two sorted array into one sorted array.

\* If {@code ar1} and {@code ar2} contains the some "equal" elements,

\* the elements in {@code ar1} will be put before the elements in {@code ar2}

\* in the merged sorted array.

\* This is important for the stability of algorithms like merge sort.

\*

\* @param ar1 One of the arrays to be merged

\* @param n1 Number of elements in array {@code ar1}

\* @param ar2 The other array to be merged

\* @param n2 Number of elements in array {@code ar2}

\* @param output The merged sorted array.

\* @param comparator The "less than" relation between every two element.

\*

\* The expected number of elements remaining in the other array when the first

\* array is exhausted is about 2(1- 1/ceil(n/2)).

\*/

**template**<**class** **T**, **class** **U**, **class** **Comparator**>

**void** **merge**(**T** ar1, **unsigned** **int** n1, **T** ar2, **unsigned** **int** n2, **U** output,

**Comparator** comparator) {

**decltype**(n1) i = 0;

**decltype**(n2) j = 0;

**auto** count = n1 + n2;

**decltype**(count) k = 0;

**while** (i < n1 && j < n2) {

**if** (comparator(ar2[j], ar1[i])) {

output[k] = ar2[j];

++j;

} **else** {

output[k] = ar1[i];

++i;

}

++k;

}

**if** (val1) {

**auto**& tmp = statistics[count];

++tmp.first;

tmp.second += (n1 - i) + (n2 - j);

}

**while** (i < n1)

output[k++] = ar1[i++];

**while** (j < n2)

output[k++] = ar2[j++];

}

/\*\*

\* Top-down merge sort.

\* @param ar The array to be sorted.

\* @param n Number of elements in the array.

\* @param output An auxiliary array that will be used by this sorting procedure,

\* the length of it should be at list the same as {@code ar}.

\* @comparator The "less than" relation between any two element in the array

\*

\* After returned from this procedure, the elements in {@code ar} will be sorted.

\*

\* Note the template argument {@code optimize} indicates whether to used the

\* optimized version of the algorithm. {@code cutoff} is a threshold value,

\* when {@code n} is not greater than this value insertion sort will be used.

\*/

**template**<**class** **T**, **class** **U**, **class** **Comparator**, **bool** **optimize** = **true**,

**unsigned** **int** **cutoff** = 10>

**void** **mergeSort**(**T** ar, **unsigned** **int** n, **U** output, **Comparator** comparator) {

//if only one element no need to sort

**if** (n <= 1)

**return**;

//if use the optimized version and the elements to be sorted is less than the "cutoff" value

//insertion sort will be used.

**if** (**optimize** && n <= **cutoff**) {

insertionSort(ar, n, comparator);

**return**;

}

**decltype**(n) n1 = n / 2;

**decltype**(n) n2 = n - n1;

**T** ar2 = ar + n1;

//sort the left half

mergeSort(ar, n1, output, comparator);

//sort the right half

mergeSort(ar2, n2, output + n1, comparator);

//if use the optimized version and the smallest element in the right half

//is greater than the largest element in the left half, then no need to merge.

//The possibility that this optimization succeeds depend on the "cutoff" value,

//For a cutoff value of "11", the possibility is less than 0.002

// (about 1.0/C(cutoff, cutoff / 2), where C(m,n) is the combination of "m" and "n"

//Several tests have validated this assertion.

**if** (**optimize** && !comparator(ar[n1], ar[n1 - 1])) {

++optimizationCount;

**return**;

}

//merge the left and left sorted subarrays.

merge(ar, n1, ar2, n2, output, comparator);

**for** (**decltype**(n) i = 0; i < n; ++i)

ar[i] = output[i];

}

/\*\*

\* Generate uniformly distributed random unsigned integers between {@code 0} and {@code maxValue}

\* in the range [{@code begin), {@code end}).

\*/

**template**<**class** **ForwardIterator**>

**static** **void** **randUInts**(**ForwardIterator** begin, **ForwardIterator** end,

**unsigned** **int** maxValue) {

std::random\_device rd;

std::uniform\_int\_distribution<**decltype**(maxValue)> generator(0,

maxValue);

std::generate(begin, end, [&] {**return** generator(rd);});

}

**int** **test**(**int** argc, **char** \*argv[]) {

**unsigned** **int** start = 10;

**for** (**unsigned** **int** i = 0; i < 100; ++i, start += 10) {

**unsigned** **int** values[start], aux[start];

*randUInts*(values, values + start, 2 \* start);

val1 = **true**;

mergeSort(values, start, aux, std::less<**unsigned** **int**>());

start += 10;

}

**for**(**auto** val : statistics){

std::cout << val.first << ", " << val.second.second / (**double**)val.second.first << '\t';

}

std::cout << std::**endl** << "----------------------------------------------------------------" << std::**endl**;

statistics.clear();

val1 = **false**;

**auto** i = 1;

**for** (start = 11; start <= (1 << 25) \* 5; start \*= 2, i \*= 2) {

**unsigned** **int** \*values = **new** **unsigned** **int**[start];

**unsigned** **int** \*aux = **new** **unsigned** **int**[start];

optimizationCount = 0;

*randUInts*(values, values + start, 2 \* start);

mergeSort(values, start, aux, std::less<**unsigned** **int**>());

std::cout << "number of elements : " << start << ", expected : " << i / 462.0 << ", actual : " << optimizationCount << std::**endl**;

**delete** []values;

**delete** []aux;

}

**return** 0;

}

};

**int** **main**(**int** argc, **char** \*argv[]) {

MergeAnaysis one;

**return** one.test(argc, argv);

}

Sample output:

11, 1.73449 12, 1.66453 13, 1.76422 14, 1.7165 15, 1.73294 16, 1.70642 17, 1.86131 18, 1.75912 19, 1.79348 20, 1.73106 21, 1.83696 22, 1.82482 23, 1.78832 24, 1.88768 25, 1.93985 26, 1.8913 27, 1.96715 28, 1.76642 29, 1.8913 30, 1.8566 31, 1.77273 32, 2.01351 33, 1.94118 34, 1.86111 35, 2.12903 36, 2.05556 37, 1.69118 38, 1.67647 39, 1.70833 40, 2.03333 41, 2.02778 42, 2.07353 43, 2.10294 44, 1.90278 45, 1.77419 46, 2.04167 47, 2.02941 48, 1.77941 49, 1.75 50, 1.98361 51, 1.91667 52, 1.97059 53, 2.08824 54, 1.83333 55, 1.85484 56, 1.97222 57, 1.70588 58, 1.85294 59, 1.91667 60, 1.98333 61, 1.84722 62, 1.79032 63, 1.875 64, 1.72222 65, 1.92857 66, 1.77778 67, 1.83333 68, 1.94444 69, 1.5 70, 2.15385 71, 1.5 72, 1.72222 73, 1.88889 74, 2 75, 1.92857 76, 1.72222 77, 1.33333 78, 1.94444 79, 2.44444 80, 2.41667 81, 1.66667 82, 1.5 83, 1.44444 84, 2.33333 85, 1.92857 86, 2.44444 87, 1.66667 88, 2.22222 89, 1.72222 90, 1.69231 91, 2.22222 92, 1.66667 93, 2 94, 1.88889 95, 1.5 96, 1.77778 97, 1.94444 98, 2.66667 99, 1.77778 100, 1.5 101, 1.77778 102, 1.83333 103, 1.77778 104, 1.61111 105, 1.85714 106, 1.77778 107, 1.88889 108, 1.94444 109, 2 110, 1.30769 111, 1.83333 112, 2.33333 113, 2.61111 114, 1.72222 115, 1.71429 116, 1.83333 117, 2.61111 118, 1.66667 119, 2.05556 120, 1.66667 121, 2 122, 1.44444 123, 2.05556 124, 1.77778 125, 1.625 126, 2.5 127, 4 128, 2 129, 2.16667 130, 1 131, 2 132, 2.5 133, 1 134, 1.66667 135, 5 136, 3.16667 137, 1 138, 3.5 139, 1.5 141, 1.83333 142, 1.75 143, 1 144, 1.16667 145, 3.5 146, 1.83333 147, 1.75 148, 1.75 149, 1.66667 150, 5 151, 1.5 152, 1.25 153, 4 154, 1.83333 155, 1 156, 1.5 157, 1.75 158, 2.5 159, 1.66667 161, 2 162, 2 163, 2.5 164, 1.66667 165, 1 166, 2.5 167, 1.25 168, 1.75 169, 2 170, 1 171, 1.83333 172, 1.25 173, 1.75 174, 2.5 175, 1 176, 1.83333 177, 1.75 178, 3.5 179, 2.33333 181, 1 182, 2 183, 2.75 184, 1.33333 185, 1.5 186, 1.5 187, 3.5 188, 2.5 189, 1.83333 190, 2 191, 1.83333 192, 1 193, 1.75 194, 1.66667 195, 2 196, 1.83333 197, 2 198, 2 199, 3.16667 201, 2.16667 202, 1.25 203, 1 204, 1.83333 205, 1 206, 2 207, 3 208, 1.25 209, 2.16667 210, 1 211, 1.5 212, 1.75 213, 2 214, 2.5 215, 4.5 216, 1.83333 217, 1.5 218, 2.25 219, 1.5 221, 2.33333 222, 1.25 223, 2.5 224, 1.5 225, 1.5 226, 2.16667 227, 1 228, 2.25 229, 2 230, 1 231, 1.66667 232, 2.5 233, 1.5 234, 1 235, 1.5 236, 1 237, 2 238, 1.5 239, 1.66667 241, 1.66667 242, 2 243, 1.75 244, 2.33333 245, 2.5 246, 2.5 247, 2.25 248, 1.5 249, 1.5 250, 1 252, 1 253, 3.5 255, 2 257, 1 258, 1 262, 2.5 263, 1 265, 4.5 267, 1.5 268, 2.5 270, 5 272, 2 273, 1 275, 2.5 277, 1.5 278, 1 282, 1.5 283, 1.5 285, 1.5 287, 3 288, 3 290, 1 292, 1.5 293, 1.5 295, 1 297, 2 298, 3.5 302, 1.5 303, 1 305, 3 307, 1 308, 2 310, 3 312, 1.5 313, 1 315, 2 317, 2 318, 4.5 322, 1 323, 1 325, 3 327, 1.5 328, 2 330, 4 332, 2.5 333, 2 335, 2 337, 1.5 338, 2 342, 4 343, 3 345, 4.5 347, 1 348, 1.5 350, 1 352, 1.5 353, 5 355, 2 357, 2 358, 4 362, 2.5 363, 3 365, 1 367, 1 368, 1.5 370, 1 372, 1 373, 1.5 375, 1 377, 1.5 378, 1 382, 1.5 383, 2 385, 2.5 387, 1 388, 2 390, 1 392, 4 393, 1.5 395, 1.5 397, 1 398, 1 402, 2.5 403, 1.5 405, 2 407, 1 408, 3 410, 1 412, 1.5 413, 2 415, 1.5 417, 1.5 418, 1 422, 2 423, 2 425, 1.5 427, 2 428, 2.5 430, 2 432, 1.5 433, 1 435, 1 437, 1.5 438, 2 442, 2.5 443, 2.5 445, 1.5 447, 4 448, 1 450, 1 452, 1 453, 1.5 455, 1 457, 1.5 458, 1.5 462, 2 463, 1.5 465, 1.5 467, 3.5 468, 2 470, 5 472, 2 473, 1.5 475, 1 477, 1 478, 1 482, 2 483, 2 485, 6 487, 2.5 488, 1.5 490, 1 492, 1.5 493, 2 495, 1.5 497, 2 498, 2 505, 1 510, 1 515, 1 525, 2.5 530, 2 535, 4 545, 2.5 550, 3 555, 3.5 565, 1.5 570, 1 575, 3.5 585, 1 590, 2 595, 1 605, 2 610, 1 615, 1 625, 6 630, 1 635, 1 645, 1.5 650, 3 655, 2 665, 3 670, 1 675, 2.5 685, 2 690, 1 695, 1.5 705, 1.5 710, 1 715, 2 725, 4 730, 1 735, 3 745, 3 750, 1 755, 1.5 765, 2.5 770, 1 775, 1 785, 2.5 790, 1 795, 2 805, 1 810, 1 815, 1.5 825, 2 830, 1 835, 2 845, 1.5 850, 2 855, 2 865, 2 870, 1 875, 2.5 885, 2.5 890, 2 895, 3 905, 1.5 910, 1 915, 1.5 925, 3 930, 5 935, 2.5 945, 1 950, 1 955, 1.5 965, 1.5 970, 1 975, 3.5 985, 1 990, 1 995, 1 1010, 1 1030, 2 1050, 1 1070, 1 1090, 2 1110, 1 1130, 4 1150, 2 1170, 4 1190, 3 1210, 1 1230, 1 1250, 2 1270, 1 1290, 1 1310, 4 1330, 1 1350, 1 1370, 1 1390, 1 1410, 2 1430, 6 1450, 3 1470, 2 1490, 2 1510, 2 1530, 1 1550, 2 1570, 1 1590, 2 1610, 1 1630, 3 1650, 1 1670, 1 1690, 1 1710, 5 1730, 1 1750, 3 1770, 1 1790, 1 1810, 3 1830, 1 1850, 1 1870, 2 1890, 2 1910, 3 1930, 1 1950, 1 1970, 1 1990, 1

----------------------------------------------------------------

number of elements : 11, expected : 0.0021645, actual : 0

number of elements : 22, expected : 0.004329, actual : 0

number of elements : 44, expected : 0.00865801, actual : 0

number of elements : 88, expected : 0.017316, actual : 0

number of elements : 176, expected : 0.034632, actual : 0

number of elements : 352, expected : 0.0692641, actual : 0

number of elements : 704, expected : 0.138528, actual : 0

number of elements : 1408, expected : 0.277056, actual : 0

number of elements : 2816, expected : 0.554113, actual : 0

number of elements : 5632, expected : 1.10823, actual : 0

number of elements : 11264, expected : 2.21645, actual : 4

number of elements : 22528, expected : 4.4329, actual : 5

number of elements : 45056, expected : 8.8658, actual : 4

number of elements : 90112, expected : 17.7316, actual : 18

number of elements : 180224, expected : 35.4632, actual : 29

number of elements : 360448, expected : 70.9264, actual : 73

number of elements : 720896, expected : 141.853, actual : 122

number of elements : 1441792, expected : 283.706, actual : 252

number of elements : 2883584, expected : 567.411, actual : 545

number of elements : 5767168, expected : 1134.82, actual : 1081

number of elements : 11534336, expected : 2269.65, actual : 2269

number of elements : 23068672, expected : 4539.29, actual : 4503

number of elements : 46137344, expected : 9078.58, actual : 9072

number of elements : 92274688, expected : 18157.2, actual : 18239