An straightforward in-place merge algorithm and its corresponding in-place merge sort

Berenger Bramas Quentin Bramas

11 Sept. 2012 - Draft V1
15 Dec. 2014 - Beta V1.1

Abstract

In this paper we present an in-place merge algorithm which can be used to implement an in-place merge sort. To our knowledge, this algorithm has never been proposed. However, merge sort has been studied for a long time it is highly probable that someone had already implemented a similar approach. The space complexity of the algorithm is $O(1)$ since it only needs to perform swaps within the array. The time complexity is $O(m^2 + n)$ and the algorithm perform no more than $O(m + n)$ comparisons, with m and n the lengths of the two partitions to merge. The main asset of our algorithm is it simplicity in term of implementation and understanding since it is very closed to the merge algorithm with external buffer. In annex, we provide a *C* implementation of our algorithm and a merge sort based on it.

1 Introduction

The problem of in-place merging is defined as follow. Having two sorted arrays A and B of size m and n respectively, we want to obtain a sorted array C of size $m + n$ with the values from A and B inside. A basic approach to solve this problem relies on an external buffer of size $m+n$ to construct the resulting array C without changing the sources A and B. However, if memory is drastically limited or if allocation/deallocation is costly or even if the given arrays are small, having a in-place algorithm to perform this operation could be a need.

The topic has been studied since old time relatively to computers age. But more attention has been given to merge sort (instead of merging) and we can find nice studies ([1], [2]) which proposed in-place merge sorts with $O(nlogn)$ complexities. However our study is different in term of complexity because we focus only and merging and just play with it in a real merge sort. Moreover we also feel that our approach is very intuitive and our rearrange algorithm powerful.

In the first section of the paper we describe different cases depending on the values in A and B. Then, we introduce a so-called rearrange algorithm which is the more complicated part of the wall system. Finally, after detailing the complexity, we show some results and provide an implementation of the algorithm.

2 Merging algorithm

2.1 Merging with external buffer

Merging two sorted arrays in a third one is a classic algorithm. One can set an index on both source arrays and read with an iterative sequential access. We compare both indexed values and copy the smallest value into the destination array before incrementing the corresponding index. If all the values of A are smaller than the first value of B, this algorithm simply copies A into C and then B into C. Moreover, if one of the arrays has been completely proceed, the only remaining operation is to copy the second without any comparisons. In the worst case the algorithm needs to perform one comparison to proceed one value so we have a maximal of $O(m + n)$ comparisons. In term of complexity, both iterators are accessing all the values of the arrays, but only one is incremented at a time so we have $O(m + n)$.

2.2 Merging in-place

From now we consider that the input arrays A and B are stored continuously in memory, see Figure 1. For simplicity we can refer to the global array as the $Input$. Like the merging with external buffer, we know that the first value of the resulting array will be $A(0)$ or $B(0)$. If $B(0)$ is the smallest value it should be moved a the front of the array, at the position of $A(0)$. However, when we move some data we

must ensure that no information are lost. That is why we are allowed to perform only swap operations which lets us save any values within the working array in 0(1) space complexity.

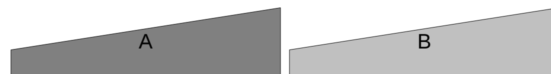


Figure 1: Input of the algorithm

All values of A smaller than $B(0)$ In the case of having all the values of A smaller than the first value of B, the array is already sorted we have nothing to do. See Figure 2.

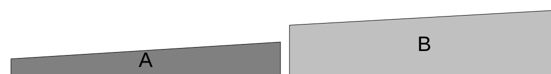


Figure 2: $A < B$

All values of A greater than the last of B In this case, we need to move the values of B in front and the values of A at the end Figure 3. This is the objective of our rearrange algorithm presented in Section 2.3.

Figure 3: $A > B$

Some values of A are smaller than the first of B In this case, we should leave all the values of A that are smaller than the first of B ($B(0)$) because they are already at the correct position. As shown in Figure 4 we leave the front of A in-place and renamed the remaining part A. Then we are in the case: Some values of B are smaller than the first of A.

Figure 4: $A(0 : L) < B(0)$

Some values of B are smaller than the first of A We note L the number of values of B that are smaller than the first values of A ($A(0)$). We need to exchange these values in order to put them at their correct positions. Since we need to save the values from A that are replaced, we simply swap between A and B as shown in Figure 5.

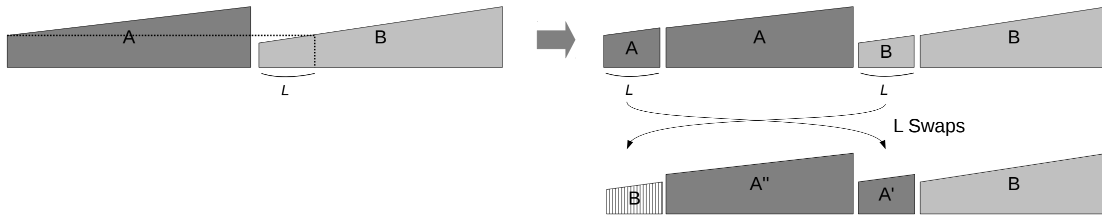


Figure 5: $B(0 : L - 1) < A(0)$ ($P3$)

The values of B that have been moved will not be used anymore since they are at their correct positions. So we end up with three partitions and called this configuration $P3$. The first one, A'' is composed of the values of A that have not been moved. The second one, A' is composed of the L values of A that have been moved and we know that the values in A' are smaller than the values in A'' The last one, contains the remaining values of B (which is our B partition for the next steps).

All the values of A' are smaller than the first values of B In this case, we know that the values of A' should be stored in front because they are smaller than $B(0)$ and by definition A' is smaller than A''. We can simply exchange the values between A' and the first ones of A'' (Figure 6). After this stage we end in the same configuration as $P3$.

Some values of A' are smaller than the first values of B In this case, we know that the first values of A' that are smaller than $B(0)$ should be moved in the front of the buffer. We proceed as previously by swapping values between A' and

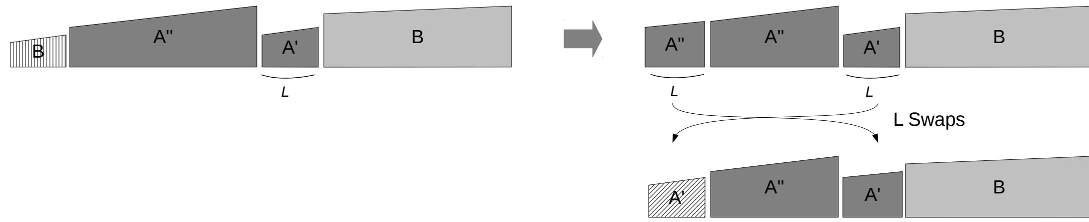


Figure 6: $A' < B(0)$

A''. However, we end-up in a new configuration called $P4$ with four partitions. The values of A' that have been moved in front which will not be touched again since they are at their correct positions. The first useful partition is A''' which is the rightest part of the previous A''. Then comes A'' which contains the first values of the previous A'' that have been swapped. In third, we have A' which contains the values of A' that have not been swapped because they are greater than $B(0)$. Finally we have B.

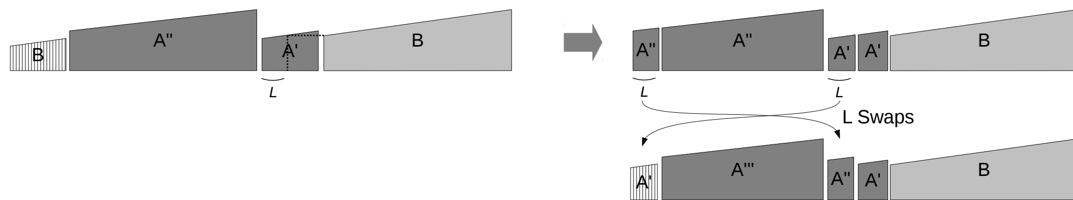


Figure 7: $A'(0 : L - 1) < B(0)$ $(P4)$

In this configuration we have $A' < A'' < A''$ and $B(0) < A'(O)$. If the second condition was false we would have been in the case of: All the values of A' are smaller than the first values of B.

The next operation could be to insert the first values of B in front which will be swapped the first values of A'''. We know that the values of A''' that will be moved are greater that A' and A'' but it will create a new partition. We do not want to create some kind of recursive partitions and so it is not the good way to proceed. To avoid this situation, we rearrange A' and A'' to have only one sorted partition. This process is presented in the next section and is called in-place rearrange.

Once the rearrange is done, we end up in configuration $P3$: $B(0)$ is the smallest values and we have only two others partitions A' and A''. It leads us back to the

case: Some values of B are smaller than the first of A.

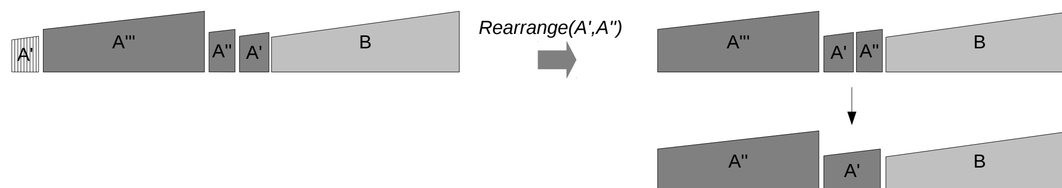


Figure 8: $P4$ to $P3$

End of the algorithm The algorithm is over if there are no more values in A or B. If there is no more values in A there is nothing to do and the values from B are originally in the correct position. If there is no more values from B we might need to perform a rearrange between A' and A''.

2.3 In-place Rearrange

As input, the rearrange algorithm is taking an array composed of two partitions A and B of lengths L_A and L_B respectively. It aims to inverse the partitions, moving the partition in front in the back and the one in the back in the front. But it must keep the internal order of the partition.

In our case, the partition are sorted arrays and we know that the values from the left partition are greater than the partition in the right. Therefore, the output is a complete sorted array.

In the next paragraphs, the input can been seen as two distinct arrays A and B or a single array *Input* of length $L_I = L_A + L_B$. The resulting array of size L_I is called *Output* but when talking about in-place operation *Input* and *Output* refer to the same element.

Using an external buffer Using an external buffer makes this problem very easy. We can copy all the values directly in a external buffer as shown in Figure 9. We can even use a buffer of the size of the smallest partition and copy the smallest partition in it. Then, it is possible to shift the largest partition in the right place. Finally, we copy back the values from the temporary buffer into the array. One

can define the operation as:

$$\forall i, i \leq L_I, Output(i) = B(i) = Input(L_A + i), i \leq L_B \tag{1}$$
$$= A(i - L_B) = Input(i - L_B), L_B \leq i \tag{2}$$

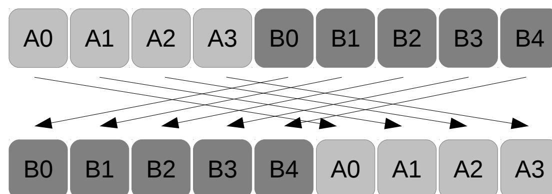


Figure 9: Rearrange with an external buffer

However, in our case we would like to do it in-place. The next paragraphs describe two of the possibilities to solve this problem.

2.3.1 In-place rearrange by shifting

In the first approach, we move the smallest partition, let say A, to the appropriate position. We swap the values of the smallest partition with the one that are currently stored in the buffer and which belong to B. Then the smallest partition will remain untouched because it has been moved to the correct place. We end up with the same problem but with different data, the big partition has been split in two parts, the part that has been moved to leave the space and the part that has not changed. Therefore, we can call the algorithm again because these two partitions need to be rearranged. Despite its simplicity this algorithm has potentially a big extra-cost even if it is not needed to do it recursively. On the other hand, it has a sequential memory access which can make it extremly efficient on current modern architectures. Figure 10 presents several iterations of this approach.

2.3.2 In-place rearrange by rolling

The second approach minimizes the number of data movements at the cost of irregular accesses. We already know where each values of the input array should be move to. The values of A should be shift to the right by L_B, whereas the values of B should be shifted to the left by L_A. The Figure 11 shows the objective of the rearrange by rolling.

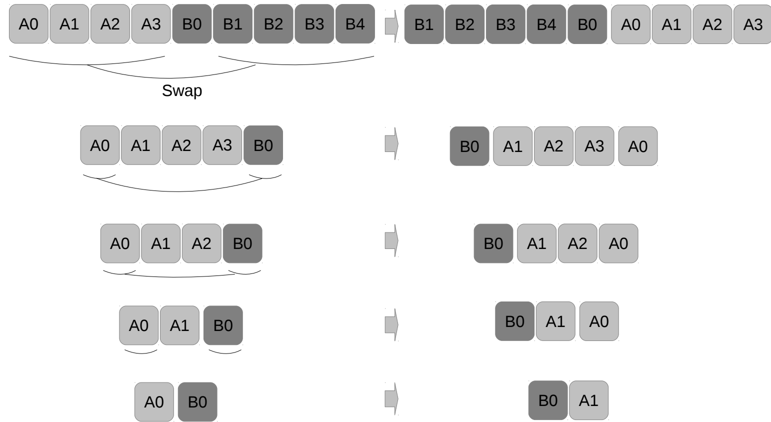


Figure 10: Rearrange with shifts

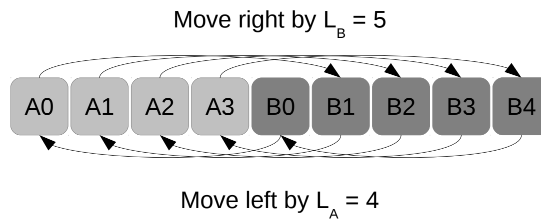


Figure 11: Rearrange with rolls

If one of the two partition is longer than the other, we will shift some of its values inside its original interval. For example, if $L_A < L_B$, all the values of A will go on position originally taken by B, whereas some values of B will go on the previous part of A and some others on the previous part of B. This come from the interval relations, for values of A $Input(O : L_A - 1) \rightarrow Output(L_B : L_I - 1)$ and for values of B $Input(L_A : L_I - 1) \rightarrow Output(0 : L_B - 1)$.

From this relation, we are able to move the elements one after the other. Starting from $A(0)$, we shift it to the right by L_B, $Input(0) \rightarrow Input(L_B)$. In order not to loose data we save the destination value $Input(L_B)$ into a temporary. If the saved value is a value from A (which can happen if $L_A > L_B$) we repeat the operation by shifting it to the right. Otherwise we saved of value from B and we shift it to the left by L_A. The complete algorithm is illustrated in Figure 12.

To guarantee that this algorithm is completed we need to ensure that all the values have been moved and moved only once. We prove in the next paragraph that the cycle goes back to the first position of the loop after several iterations.

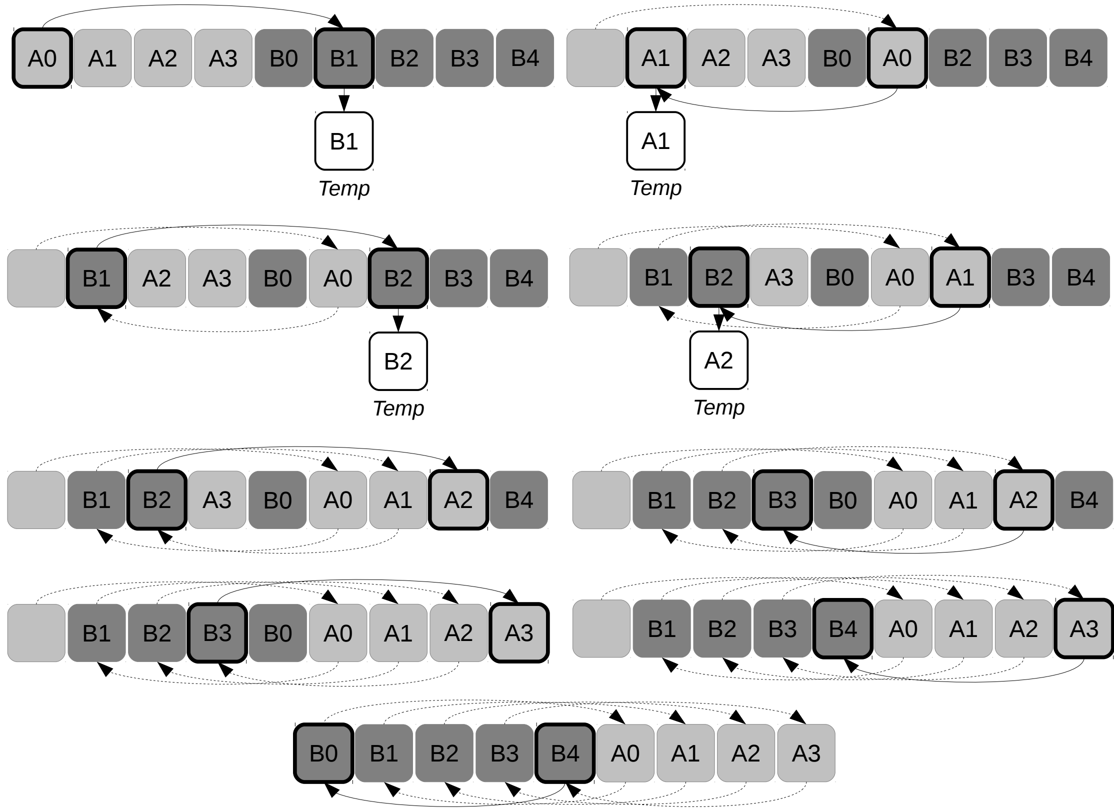


Figure 12: $P4$ to $P3$

Then, we can erase the value with the one in the temporary buffer since it had been moved at the first iteration. The cycle stops there but the process is not over and some values may have not been moved yet. We need to perform others cycles but from another starting index as illustrated in Figure 13.

Proof We consider the case where $L_A < L_B$ but the inverse can be proved similarly. To understand the rolling process one can follow these steps:

1. We start from $i_0 = s$, in our case $s = 0$ for the first iteration
2. We increase by L_B to move to the right, $i_{k+1} = i_k + L_B$
3. We decrease by L_A possibly several times to move to the left, $i_{k+1} = i_k - hL_A$

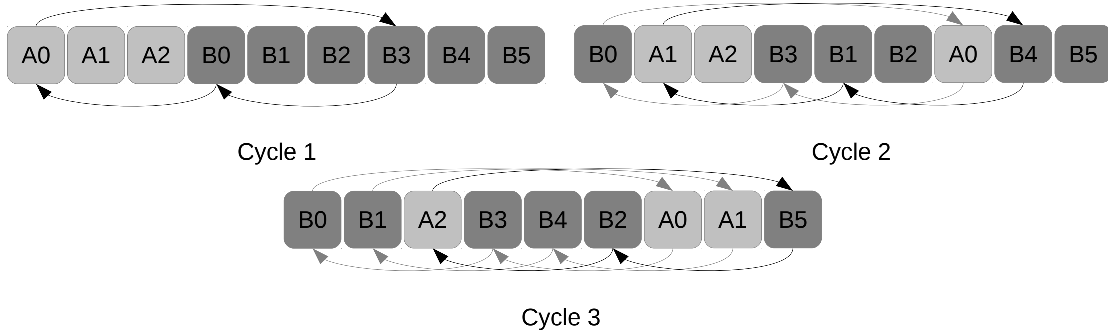


Figure 13: Example with 3 cycles need to complete the process

In order to find a cycle, we need to find a equality between the first index and a new one:

$$i_k = x_1 * L_B - x_2 * L_A = s = 0 \tag{3}$$

$$x_1 * L_B = x_2 * L_A \tag{4}$$

Solving this equation is related to Least common multiple (LCM called Plus Petit Commun Multiple PPCM in French). We have to find x_1 and x_2 integers smallest as possible but greater than 0. In the worse case, $x_1 = L_A$ and $x_2 = L_B$, $L_A * L_B = L_B * L_A$. Such case means that L_A values from A have been shifted by L_B positions to the right. The partition A has exactly L_A elements, so in this case all elements are moved before the index finished its cycle. And the same for B which contains L_B values.

We could have $x_1 \neq L_A \Leftrightarrow x_2 \neq L_B$. In the case of $x_1 \neq L_A$ and $x_2 \neq L_B$, we still have $i_0 = x_1 * L_B - x_2 * L_A, x_1 < x_2$ which means that some elements have not been moved as the cycle is finished. To complete the algorithm, we need to perform more cycles but starting at different positions. In any cases cycles will move x_1 values from A and will replace them by values from B.

Let $i_0 = 0$, $i_1 \ldots i_{n-1}$, the consecutive indexes in partition A. Starting from 0 we need the same number k of L_A translations to the right to obtain the next index and we have $i_0 < i_1$, $i_1 < i_2 \ldots$. However the last loop we need one more

L_A translation because $i_0 < i_{n-1}$. Therefore we have:

$$i_0 = 0 \tag{5}$$
$$i_1 = L_B - k \times L_A + i_0 \tag{6}$$
$$i_2 = L_B - k \times L_A + i_1 = 2i_1 \tag{7}$$
$$i_3 = L_B - k \times L_A + i_2 = 3i_1 \tag{8}$$
$$\vdots \tag{9}$$
$$i_c = L_B - (k+1) \times L_A + i_{c-1} = ci_1 - L_A = i_0 = 0 \tag{10}$$
$$\tag{11}$$

We have shown from the LCM that $c = x_1$. It follows $ci_1 = L_A$ and that c values from A have been moved (and replaced). The space between the values in A is $t = i_1 - i_0 = L_A/x_1$.

If we start a cycle from a different position with $i'_0 = i_0 + j$, we just translate these indexes, $i'_1 = i_1 + j$, $i'_2 = i_2 + j$ .... This is given by:

$$i'_1 = L_B - k \times L_A + j = i_1 + j \tag{12}$$
$$i'_2 = L_B - k \times L_A + i_1 = 2i_1 + j \tag{13}$$
$$i'_3 = L_B - k \times L_A + i_2 = 3i_1 + j \tag{14}$$
$$\vdots \tag{15}$$
$$i'_c = L_B - (k+1) \times L_A + i_{c-1} = ci_1 + j \tag{16}$$
$$\tag{17}$$

Finally, when a cycle started at position j is finished, every values of A at positions $j+k*t$ have been moved. And all elements with indexes $\neq j+k*t$ remain unchanged. In order to proceed all the elements we need to do t consecutive cycles starting from positions 0 to $t-1$ In the case of $t = 1$ only one cycle is sufficient and will rearrange the full partition.

We have shown that from the $LCM(L_B, L_A)$ we can obtain x_1 and x_2 with $x_1 L_B = x_2 L_A$. Then in order to find $t = i_1 = L_A/x_1$:

$$
\begin{aligned}
t &= L_A/x_1 && (18)\\
&= L_B L_A / L_B x_1 && (19)\\
&= LMC(L_B, L_A) GMD(L_B, L_A)/L_B x_1 \qquad \text{Since } L_B L_A = GMD \times LMC && (20)\\
&= GMD(L_B, L_A) \qquad \text{Since } L_B x_1 = LMC && (21)
\end{aligned}
$$

Finally we have $t = LMC(L_A, L_B)$ Remark : we have LMC for least common multiple (PPCM - Plus petit commun multiple in French), GMD for greatest common divisor (PGCD - Plus grand commun diviseur in French).

3 Complexity

3.0.3 In-place rearrange by shifting

When rearranging two partitions A and B of sizes L_A and L_B, if $L_A = L_B$ we simply swap the elements of A and B which gives a linear complexity. But in most of the case $L_A \neq L_B$ and we have several iterations. Consider $L_A < L_B$ the first iteration $k = 0$ swaps L_A elements. Then we end with two new partitions A^{k+1} and B^{k+1} having $L_A^{k+1} = Min(L_B^k - L_A^k, L_A^k)$ and $L_B^{k+1} = Max(L_B^k - L_A^k, L_A^k)$ as sizes. So the algorithm is over when $L_A^{k+1} = L_B^{k+1}$ or when of the partition is zero length. In any case at each swap operation only one value is moved to the right place. So the complexity is linear but only one data is written at the correct position at each iteration. However the memory access pattern is linear.

3.0.4 In-place rearrange by rolling

In this case each value will be read once and moved directly to its correct position. We have $t = LMC(L_A, L_B)$ cycles and in each of them we moved $x_1 + x_2$ values. So the complexity is also linear, but the memory access pattern is irregular. In fact, from one cycle to the next one the spatial difference is only one but inside a cycle each loops may access different parts of the array. One can imagine to implement a parallel version with t threads (one thread per cycle).

3.0.5 In-place Merging

If $A < B$ or $B < A$ it is easy to see that the complexity is linear because there is nothing to do or we just need to perform a rearrange in order to swap the partition. In the case where we move the same values of A and B the complexity is linear too because no rearrange is invoked in this case. Otherwise the complexity increases every time we need to call rearrange and the worst complexity is $O(m^2 + n)$ with $m = L_A$ and $n = L_B$.

This situation happens when we have to call rearrange every time we swap a value. For example if half the items of B have been swapped with the items of A we end up in $P3$. Then if we need to insert one value of A and one value of B consecutively we will need to perform one rearrange before inserting one value of B. But the average complexity is $O(m * l + n)$ with l the average size of blocks that are moved.

4 Results

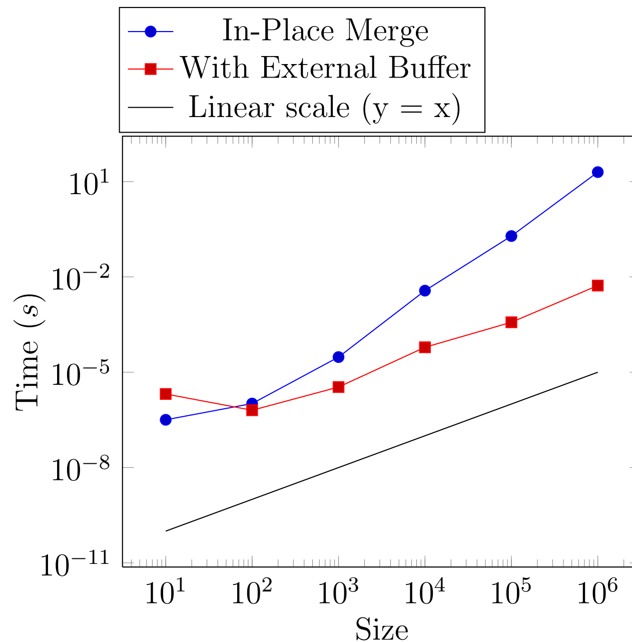
We have measured the execution time needed by the merge algorithm to proceed two arrays. We took arrays from 10 to 1000000 and take 1/4, 2/4 and 3/4 of for the length of partition A. We also put the time of merging using an external buffer but we include the time needed for the allocation, the deallocation and the copy of the results. As shown in Figure 4 the execution time is almost linear as expected. One interesting information is that the in-place merge is faster for very small arrays.

5 Licence

This code is given under the LGPL licence without any warranty.

References

[1] Practical In-Place Merging, BING-CHAO HUANG, MICHAEL A. LANGSTON , Communications of the ACM (1988)



[2] PRACTICAL IN-PLACE MERGESORT, JYRKI KATAJAINEN, TOMI PASANEN, JUKKA TEUHOLA, Nordic Journal of Computing (1996)

Implementation

In-place rearrange of the same size

We just need to flip values of the array:

```
for(int idx = 0 ; idx < lengthLeftPart ; ++idx){
    swap(array, idx, idx + lengthLeftPart);
}
```

In-place rearrange with shifting

```
// size of the partitions at first iteration
int workingLeftLength  = lengthLeftPart;
int workingRightLength = lengthRightPart;
```

```
4  // while the partitions have different sizes and none of them are
       null
5  while(workingLeftLength != workingRightLength && workingLeftLength
       && workingRightLength){
6      // if the left partition is the smallest
7      if(workingLeftLength < workingRightLength){
8          // move the left parition in the correct place
9          for(int idx = 0 ; idx < workingLeftLength ; ++idx){
10             swap(array, idx, idx + workingRightLength);
11         }
12         // the new left partition is now the values that have been
               swaped
13         workingRightLength = workingRightLength -
               workingLeftLength;
14         //workingLeftLength = workingLeftLength;
15     }
16     // if right partition is the smallest
17     else{
18         // move the right partition in the correct place
19         for(int idx = 0 ; idx < workingRightLength ; ++idx){
20             swap(array, idx, idx + workingLeftLength);
21         }
22         // shift the pointer to skip the correct values
23         array = (array + workingRightLength);
24         // the new left partition is the previous right minus the
               swaped values
25         //workingRightLength = workingRightLength;
26         workingLeftLength  = workingLeftLength -
               workingRightLength;
27     }
28 }
29 // if partitions have the same size
30 for(int idx = 0 ; idx < workingLeftLength ; ++idx){
31     swap(array, idx, idx + workingLeftLength);
32 }
```

In-place rearrange with rolling

```
int coefLeft  = 0;
int coefRight = 0;
// get the ppcm
const int ppc = ppcm(lengthLeftPart, lengthRightPart, &coefLeft, &
    coefRight);
// this is "t" in the attached repport
const int end = lengthLeftPart / coefRight;
// we need to do end cycle
for(int idx = 0 ; idx < end ; idx += 1){
    // store the first value
    int value = array[idx];
    // keep track of the first index to know when the cycle is
        over
    int idxToMove = idx;
    do{
        // While we have to do move to right
        while(idxToMove + lengthRightPart < totalSize){
            idxToMove += lengthRightPart;
            swap(&array[idxToMove], &value);
        }
        // while we have to do move to the left
        while(0 <= idxToMove - lengthLeftPart){
            idxToMove -= lengthLeftPart;
            swap(&array[idxToMove], &value);
        }
        // the cycle is over when we go back to the first index
    }while(idxToMove != idx);
}
```

In-place merging

```
void mergeInPlace(int array[], const int sizeArray, const int
    centerPosition ){
    int idxLeft   = 0;
    int idxRight  = centerPosition;

    // Skip left small values
    // case: some values of A are smaller than the first of B
```

```
 7      while( idxLeft < centerPosition && array[idxLeft] <= array[
            idxRight] ){
 8          ++idxLeft;
 9      }
10      // Where to store left values
11      int idxBuffer = centerPosition;
12      // While there are left values && there are right values
13      while(idxLeft < idxRight &&  idxRight < sizeArray){
14          // if the first value of A is smaller than the first of B
15          // we know this test will be false the first time
16          if(idxBuffer != idxRight && array[idxBuffer] <= array[
                idxRight]){
17              int offsetPosition = idxBuffer;
18              // while values from A are smaller
19              while(idxLeft < idxBuffer && offsetPosition < idxRight
                    &&
20                    array[offsetPosition] <= array[idxRight] ){
21                  swap(array, idxLeft++, offsetPosition++ );
22              }
23              // if the values in the buffer have not all moved
24              if(offsetPosition != idxRight){
25                  // We are in P4, we need to go back in P3
26                  reorderRolling(array + idxBuffer, offsetPosition -
                        idxBuffer, idxRight - idxBuffer);
27                  //reorderShifting(array + idxBuffer,
                        offsetPosition - idxBuffer, idxRight -
                        idxBuffer);
28              }
29              // all the value of A'' have been moved
30              if( idxLeft == idxBuffer ){
31                  // the buffer becomes the new A
32                  idxBuffer = idxRight;
33                  // we move the pointer as long as values from A
                        are smaller
34                  while( idxLeft < idxBuffer && array[idxLeft] <=
                        array[idxRight] ){
35                      ++idxLeft;
36                  }
37              }
38          }
39          else{
```

```
40          // The value we have to compare to is the first of A''
                or the first from the buffer
41          const int leftSmallValue = (idxBuffer == idxRight ?
                array[idxLeft] : array[idxBuffer]);
42          // Swap B and A values
43          while( idxLeft < idxBuffer && idxRight < sizeArray
44                  && array[idxRight] <= leftSmallValue ){
45              swap(array, idxLeft++, idxRight++ );
46          }
47          // If we have moved all the values of A
48          if(idxLeft == idxBuffer){
49              // A is now the buffer
50              idxBuffer = idxRight;
51              // Move the pointer as long as A is smaller
52              while( idxLeft < idxBuffer && array[idxLeft] <=
                    array[idxRight] ){
53                   ++idxLeft;
54              }
55          }
56      }
57  }
58  // it is over because no more right values && there are values
        between left and buffer
59  if(idxLeft != idxRight && idxLeft != idxBuffer){
60      reorderRolling(array + idxLeft, idxBuffer - idxLeft,
            idxRight - idxLeft);
61      //reorderShifting(array + idxLeft, idxBuffer - idxLeft,
            idxRight - idxLeft);
62  }
63 }
```

In-place merge sort

```
1 void inPlaceMergeSort(int array[], const int Size){
2     for(int sizeOfPartitions = 1 ; sizeOfPartitions < Size ;
          sizeOfPartitions *= 2){
3          for(int startingIdx = 0 ; (startingIdx+sizeOfPartitions) <
               Size ; startingIdx += (sizeOfPartitions*2)){
```

```
4             int sizeOfSecondPartiton = sizeOfPartitions;
5             if((startingIdx + sizeOfSecondPartiton +
                 sizeOfPartitions) > Size) sizeOfSecondPartiton =
                 Size - (startingIdx + sizeOfPartitions);
6             mergeInPlace(&array[startingIdx], (sizeOfPartitions+
                 sizeOfSecondPartiton), sizeOfPartitions);
7         }
8     }
9 }
```