CSC3100 Assignment 1 Report

CHEN Ang (*118010009*)

Problem 1: Merge Sort

Problem 1 asks for an implementation of the Merge Sort algorithm. The algorithm takes a positive integer N and N rows of unsigned integer as input, and outputs N rows of Merge-Sorted numbers in ascending order.

Solution

The plain, classical Merge Sort is implemented as required. That is, the algorithm divide the problem `MERGE-SORT` into two `MERGE-SORT` sub-routines which recursively sort two sub-arrays, and calls `MERGE` to merge the two sorted sub-arrays together. The `MERGE` function is implemented by creating two auxiliary arrays as copies of two sub-arrays, comparing the elements in the auxiliary arrays, and merging them into the original array.

Other Possible Solutions

The possible solutions of this problem are limited as it requests the implementation of a specific algorithm, namely Merge Sort. We shall discuss here one possible variation of the plain Merge Sort by exploiting the fact that all numbers all of type `unsigned int`, which could reduce the auxiliary space complexity to $\Theta(1)$ under some cases (omitting the $\Theta(\log n)$ stack space).

If we set

$$M := \max_{1 \le i \le n} A[i] + 1$$

we could in fact store two positive integers $a < M, b \in \mathbb{N}$ at $A[i]$ by assigning

$$A[i] \leftarrow a + Mb$$

Notice that

$$A[i]/M = b$$

since $a < M$, and that

$$A[i] \bmod M = a$$

since $a < M$. Following this strategy we could potentially reduce the space complexity to constant in `MERGE` by the following so-called `MOD-MERGE`

```
M = max(A) + 1

MOD-MERGE(A, p, q, r):
    L = p
    R = q + 1
    i = p
    while L <= q and R <= r
        if (A[L] % M <= A[R] % M)          /* compare "a"s in original array */
```

```
            A[i] = A[i] + (A[L] % M) * M  /* store A[L] % M ("a") as the "b" */
            L = L + 1
        else
            A[i] = A[i] + (A[R] % M) * M  /* store A[R] % M ("a") as the "b" */
            R = R + 1
        i = i + 1
    if (L > q)
        while (R <= r)
            A[i] = A[i] + (A[R] % M) * M  /* store A[R] % M ("a") as the "b" */
            R = R + 1
            i = i + 1
    else
        while (L <= q)
            A[i] = A[i] + (A[L] % M) * M  /* store A[L] % M ("a") as the "b" */
            L = L + 1
            i = i + 1
    for i = p..r
        A[i] = A[i] / M                   /* extract all the "b"s */
```

Notice that no copies of sub-arrays are needed in the `MOD-MERGE` routine, and hence the revised `MOD-MERGE-SORT` would only require $\Theta(1)$ auxiliary space. Also, since the `MOD-MERGE` routine is also linear in time, the revised algorithm has the same recurrence relation and same $\Theta(n \log n)$ running time as the original Merge Sort.

Comparison

The reason the plain Merge Sort is implemented instead of the revised version is because of an obvious shortcoming of the Mod-Merge routine: it only works when the number stored in the original array A are relative small compared to the data-type limit. Specifically, let us consider an `unsigned int A[n]`. if the maximum element m is

$$m = \sqrt{2^{32}} = 2^{16}$$

Storing it in the array as b would require at least an integer range of (without considering a)

$$Mm = (m+1)m > m^2 = 2^{32}$$

, which would already cause an overflow for `unsigned int`. In the extreme, if the maximum element m is near the limit of an `unsigned int`, namely

$$m = 2^{32} - 1$$

We at least need an integer range of (without considering a)

$$Mm = (m+1)m = 2^{64} - 2^{32}$$

If we consider a, whose maximum value is $2^{32} - 1$, we need exactly an `unsigned long long A[n]` array to prevent possible overflow, which requires doubling the integer size of the original array. In this case we are actually back to the $\Theta(n)$ auxiliary space.

Unfortunately, in our 6 test examples the largest element $m \approx 10^9 \approx 2^{30}$, which is nearly at the limit of an `unsigned int`. So we are in the degenerate $\Theta(n)$ case and do need `unsigned long long A[n]` if we take the alternative approach.

Testing

We tested two approaches on the OJ platform and indeed obtained nearly identical linear result in terms of space and time complexity.

Plain Merge Sort (1.071s, 6.36 MB)

Execution Results

Test case #1:	AC	0.036s	1.63 MB	(10/10)
Test case #2:	AC	0.052s	1.63 MB	(10/10)
Test case #3:	AC	0.037s	1.63 MB	(10/10)
Test case #4:	AC	0.087s	3.27 MB	(10/10)
Test case #5:	AC	0.078s	3.27 MB	(10/10)
Test case #6:	AC	0.079s	3.27 MB	(10/10)
Test case #7:	AC	0.085s	3.27 MB	(10/10)
Test case #8:	AC	0.120s	4.04 MB	(10/10)
Test case #9:	AC	0.226s	5.59 MB	(10/10)
Test case #10:	AC	0.269s	6.36 MB	(10/10)

Resources: 1.071s, 6.36 MB
Final score: 100/100 (100.0/100 points)

Mod Merge Sort (1.042s, 6.36 MB)

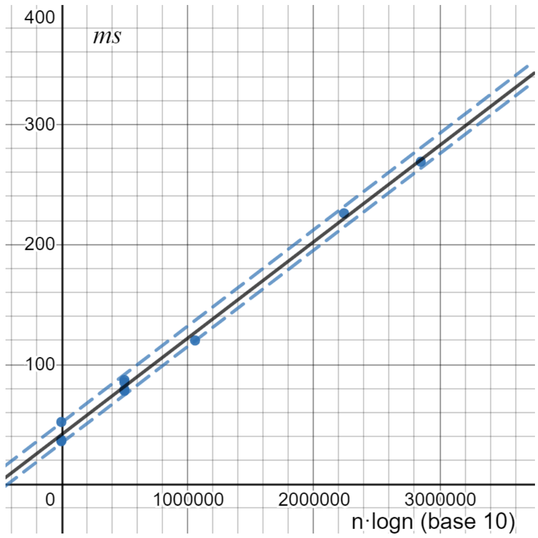
Execution Results

Test case #1:	AC	0.041s	1.63 MB	(10/10)
Test case #2:	AC	0.042s	1.63 MB	(10/10)
Test case #3:	AC	0.045s	1.63 MB	(10/10)
Test case #4:	AC	0.082s	3.27 MB	(10/10)
Test case #5:	AC	0.087s	3.27 MB	(10/10)
Test case #6:	AC	0.081s	3.27 MB	(10/10)
Test case #7:	AC	0.080s	3.27 MB	(10/10)
Test case #8:	AC	0.125s	4.04 MB	(10/10)
Test case #9:	AC	0.206s	5.59 MB	(10/10)
Test case #10:	AC	0.251s	6.36 MB	(10/10)

Resources: 1.042s, 6.36 MB
Final score: 100/100 (100.0/100 points)

Now, since the Plain Merge Sort has almost the same performance under our test cases and is conceptually easier and more friendly in terms of readability, we preferred it over the Mod revised version.

- Running time of the plain Merge Sort in ms against $n \log_{10} n$, showing the $\Theta(n \log n)$ complexity



Possible Improvement

We could in fact realize a fully in-place Merge Sort algorithm. The rough idea is that we only sort half of the array each time and merge them not by creating copies of sub-arrays, but by using the other half of the array as auxiliary space for merging. We show the key idea by an example. Say we wish to merge the right half of the array, we use the left half as the auxiliary space:

```
                                                                       L
                p q   r         i           L q R            i           q R
   [x x x x | 1 3 2 4]  =>  [x x x x | 1 3 2 4]  =>  [1 x x x | x 3 2 4]

                  L                             L                            L
             i         q   R               i       q   R                  i q      R
=> [1 2 x x | x 3 x 4]  =>  [1 2 3 x | x x x 4]  =>  [1 2 3 4 | x x x x]
```

Also note that, unlike the traditional copying approach, we actually exchange the content of the to-be-merged sub-arrays and the auxiliary space. So we need to exchange the content of the merged sub-array and the auxiliary sub-array to get the desired

```
[x x x x | 1 2 3 4]
```

This leads to an easy implementation of a `half_merge_sort` function which is very much like the traditional Merge Sort but only sort a half of the array via the help of the other half.

The next crucial step (proposed by Katajainen, Pasanen, and Teuhola in 1996), is that instead of sorting the right half of the remaining unsorted array recursively, which would lead to insufficient auxiliary space for merging, we sort the left half using `half_merge_sort`

```
[5 6 | x x | 1 2 3 4]
```

Now we can merge `[5 6]` and `[1 2 3 4]` to the rightmost of the array, starting from the first `x` in the auxiliary space with no problem:

```
     p q         r     s        L q   i     R     s        L q     i     R   s
    [5 6 | x x | 1 2 3 4]  =>  [5 6 | x x | 1 2 3 4]  =>  [5 6 | 1 x | x 2 3 4]

                                                  R                           R
     L q         i   R s        L q           i   s        L q             i s
=>  [5 6 | 1 2 | x x 3 4]  =>  [5 6 | 1 2 | 3 x x 4]  =>  [5 6 | 1 2 | 3 4 x x]

                       R
       q   L           s i
=>  [x x | 1 2 | 3 4 5 6]
```

We don't even need to do any exchange!

```
[x x | 1 2 3 4 5 6]
```

We see that the size of the unsorted sub-array halves after each cycle, and once we reach the base case below there is no need for further calls of `half_merge_sort`:

```
[x | 1 2 3 4 5 6 7]
```

We can directly insert `x` to the correct position in linear time and obtain the final sorted array

```
[1 2 3 4 5 6 7 8]
```

Katajainen et al. showed in their work that this in-place Merge Sort has $\Theta(n \log n)$ running time.

Problem 2: Prefix Expression

Problem 2 asks to evaluate prefix expressions of positive integers. A prefix expression $[\mathrm{pEx}]$ could either be a single number, or of the form

$$[[\mathrm{operator}][\mathrm{pEx}_1][\mathrm{pEx}_2]]$$

which, in infix notation, is

$$\mathrm{iEx}([[\mathrm{operator}][\mathrm{pEx}_1][\mathrm{pEx}_2]]) = \mathrm{iEx}([\mathrm{pEx}_1])\ \ [\mathrm{operator}]\ \ \mathrm{iEx}([\mathrm{pEx}_2])$$

In this problem, the operators are restricted to be one of $+, -, *$ to avoid fractional values.

The algorithm takes in a positive integer N and N different operators or positive integers which constitute the whole prefix expression.

Solution

First off, it is easy for a computer to evaluate a postfix expression $[\mathrm{qEx}]$, which is either a single number, or of the form

$$[[\mathrm{qEx}_1][\mathrm{qEx}_2][\mathrm{operator}]]$$

which, converted to infix notation, is

$$\mathrm{iEx}([[\mathrm{qEx}_1][\mathrm{qEx}_2][\mathrm{operator}]]) = \mathrm{iEx}([\mathrm{qEx}_1])\ \ [\mathrm{operator}]\ \ \mathrm{iEx}([\mathrm{qEx}_2]) \tag{1}$$

For example,

$$[3, 1+5, 2*-]$$

is what in infix notation

$$[[3, 1+], [5, 2*]-] \rightarrow (3+1) - (5*2)$$

To evaluate a post-fix expression, we can use a stack for storing the operands in a LIFO fashion. Whenever we encounter an operator, we pop two consecutive operands from the stack as **operand 2 and operand 1 (note the order!)** respectively and evaluate the intermediate expression

$$[\mathrm{operand\ 1}]\ [\mathrm{operator}]\ [\mathrm{operand\ 2}]$$

The result is then pushed back to the stack for further evaluation. If the expression is indeed valid, at the end of the process we shall be left with a single number in the stack, which is our answer. So an obvious thought would be to try to convert our prefix expression into an equivalent postfix one and feed that into the algorithm. The question is: How?

We notice that when we reverse a prefix expression

$$[\mathrm{pEx}] = [[\mathrm{operator}][\mathrm{pEx}_1][\mathrm{pEx}_2]]$$

we get

$$[\mathrm{pEx}]^R = \left[[\mathrm{pEx}_2]^R[\mathrm{pEx}_1]^R[\mathrm{operator}]\right]$$

with $[\mathrm{pEx}_1], [\mathrm{pEx}_2]$ being shorter prefix expressions than $[\mathrm{pEx}]$. Thus by strong induction on length $N \in \{1, 3, 5, 7, \cdots\}$ of any prefix expression, one can show that the reversed expression of a prefix expression is actually a postfix expression (the base cases being $N = 1$ and 3, which can be checked easily). But how do we know if they are indeed equivalent? When we transform the reversed expression (which is postfix) into infix notation, we get

$$\left[[\mathrm{pEx}_2]^R[\mathrm{pEx}_1]^R[\mathrm{operator}]\right] \rightarrow \mathrm{iEx}\left([\mathrm{pEx}_2]^R\right)\ \ [\mathrm{operator}]\ \ \mathrm{iEx}\left([\mathrm{pEx}_1]^R\right) \tag{2}$$

If we compare that to the infix notation of the original prefix expression

$$[[\mathrm{operator}][\mathrm{pEx}_1][\mathrm{pEx}_2]] \rightarrow \mathrm{iEx}([\mathrm{pEx}_1])\ \ [\mathrm{operator}]\ \ \mathrm{iEx}([\mathrm{pEx}_2]) \tag{3}$$

we see that the positions of two expressions are switched. So they are in fact NOT equivalent. However, this is in the sense of (1). If we were to switch the definition of (1) to

$$\mathrm{iEx}([[\mathrm{qEx}_1][\mathrm{qEx}_2][\mathrm{operator}]]) = \mathrm{iEx}([\mathrm{qEx}_2])\ \ [\mathrm{operator}]\ \ \mathrm{iEx}([\mathrm{qEx}_1]) \tag{1'}$$

Then (2) becomes

$$\left[[\mathrm{pEx}_2]^R[\mathrm{pEx}_1]^R[\mathrm{operator}]\right] \to \mathrm{iEx}\left([\mathrm{pEx}_1]^R\right)\ \ [\mathrm{operator}]\ \ \mathrm{iEx}\left([\mathrm{pEx}_2]^R\right) \tag{2'}$$

accordingly. This is in the order of (3), with $[\mathrm{pEx}_1]$, $[\mathrm{pEx}_2]$ being two shorter prefix expressions than $[\mathrm{pEx}]$. We, again, may use strong induction on the length $N \in \{1, 3, 5, 7, \cdots\}$ of the prefix expressions to show that any prefix expression is equivalent to its reversed expression in sense of $(1')$ with the same base cases as before. Of course, if we want to evaluate a postfix expression in definition $(1')$ using a stack, we need to treat consecutive operands popped from the stack as **operand 1 and operand 2** instead.

The solution then becomes clear: We reverse the input strings by pushing them into a first stack S_a. This effectively transforms the expression into its postfix equivalence in definition $(1')$. We then evaluate the postfix expression using the amended method discussed above, via the help of a second stack S_b.

$$[\mathrm{pEx}] \xrightarrow{S_a} [\mathrm{pEx}]^R \equiv [\mathrm{qEx}] \xrightarrow{S_b} \mathrm{Result}$$

Note that the original expression is valid if and only if its equivalent postfix expression is valid. Therefore we may test the validity of the prefix expression by examining the evaluation process of the equivalent postfix expression. There are exactly two types of possible invalidities for a postfix expression:

1. Missing operand: $[[\mathrm{qEx}][\ \cdot\ ][\mathrm{operator}]]$

   This can be caught by checking if S_b is empty when trying to pop an operand.

2. Missing operator: $[[\mathrm{qEx}_1][\mathrm{qEx}_2][\ \cdot\ ]]$

   This can be caught by checking if the final size of S_b is larger than 1.

The time complexity of the algorithm is $\Theta(n)$. Reversing the expression requires $n = \Theta(n)$ pushes. Whenever we pop out an operand from S_a, and push it to S_b, taking $\Theta(1)$ time. If we pop out an operator from S_a, we pop twice from S_b, evaluate the expression (assuming constant time), and push once to S_b, which also takes $\Theta(1)$ time. Assuming validity of the expression, the algorithm should finish exactly (also adding the one single pop) when S_a becomes empty, i.e., after n pops from S_b. Hence the time complexity of the algorithm is always

$$\Theta(n) + n \cdot \Theta(1) + \Theta(1) = \Theta(n)$$

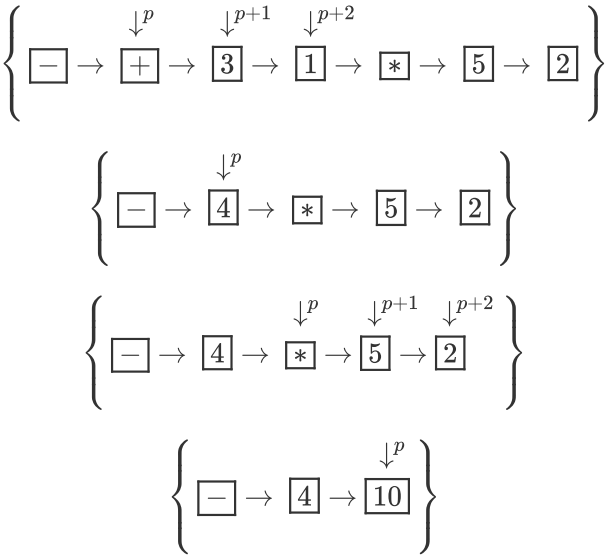
if the input expression is valid. If the expression could be invalid, the algorithm only stops earlier in the process, which leads to $O(n)$ complexity.

Other Possible Solutions

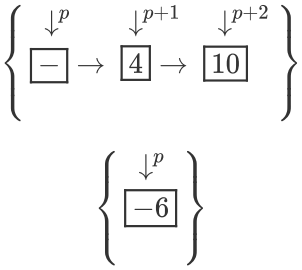
One other possible solution is that we insist to evaluate the prefix expression in order of the input. We may first store the expression into a singly linked list, e.g.,

$$[-+3, 1*5, 2] \implies \left\{\boxed{-} \to \boxed{+} \to \boxed{3} \to \boxed{1} \to \boxed{*} \to \boxed{5} \to \boxed{2}\right\}$$

We scan through the entire list, stop whenever we meet an operator and check if the next two elements are numbers. If they are, we evaluate the result and replace the three-element expression in our list by the result.



However, we see that the expression is yet fully evaluated as the length of the list is not 1. We need to do another round:



Now we know the expression has been fully evaluated by the fact that the length of the list has reached 1. We print out the head of the list as our final result. We can check the validity of an expression by comparing the length of the list before and after one scan. If the length fails to decrease, we know the expression must be invalid.

```
PREFIX_EVAL_LIST(L)
    /* building the list takes θ(N) time */
    Let L be a non-empty singly linked list containing the prefix expression.

    /* valid expressions must have odd length */
    if L.length % 2 == 0
        raise "INVALID"
    else
        prev_len = ∞
        curr_len = L.length

        /* dominated by the while-loop, O(l) where l = length of current list */
        while curr_len < prev_len
            /* expression fully evaluated */
            if curr_len == 1
                if L.head.val is a number
                    return L.head.val
                else
                    raise "INVALID"
            /* expression yet fully evaluated */
            else
                p0 = L.head
                p1 = p0.next
                p2 = p1.next
                /* scanning, takes θ(l) where l = length of current list */
```

```
            while p2 != NULL
                if (p0.val is an operator) and (p1.val, p2.val are numbers)
                    /* evaluate the expression */
                    p0.val = EVALUATE(p1.val, p0.val, p2.val)

                    /* delete two middle nodes */
                    p0.next = p2.next
                    L.length = L.length - 2

                    /* advance all pointers by 3 */
                    p0 = p0.next
                    p1 = p0 == NULL? NULL : p0.next
                    p2 = p1 == NULL? NULL : p1.next
                else
                    p0 = p1
                    p1 = p2
                    p2 = p2.next
            /* update length records */
            prev_len = curr_len
            curr_len = L.length

        /* length fails to decrease */
        raise "INVALID"
```

Comparison

The above solution is intuitive and easy to understand. However, its performance downgrades drastically when the "depth" of the expression increases. Consider the following worst case scenario:

$$[\overbrace{+ + + \cdots +}^{\lfloor n/2 \rfloor} \overbrace{1, 1, 1, \cdots, 1}^{\lceil n/2 \rceil}]$$

Unable to utilize the result of a previous evaluation later in one scan, our algorithm can only evaluate one expression in a go, resulting in minimum decrease, 2, of the expression length. The time complexity in this case would be

$$\sum_{l \in \{n, n-2, \cdots, 1\}} \Theta(l) = \Theta\left(\frac{(n+1)^2}{4}\right) = \Theta(n^2)$$

One way of fixing this issue is to use a doubly linked list instead, and backtrack by 2 (since new evaluation is possible only in this range) whenever we make an evaluation.

Take the worst-case example,

$$\left\{ \cdots \leftrightarrow \boxed{+} \leftrightarrow \boxed{+} \leftrightarrow \underset{\downarrow p}{\boxed{+}} \leftrightarrow \underset{\downarrow p+1}{\boxed{1}} \leftrightarrow \underset{\downarrow p+2}{\boxed{1}} \leftrightarrow \boxed{1} \leftrightarrow \cdots \right\}$$

$$\left\{ \cdots \leftrightarrow \boxed{+} \leftrightarrow \boxed{+} \leftrightarrow \underset{\downarrow p}{\boxed{2}} \leftrightarrow \boxed{1} \leftrightarrow \cdots \right\}$$

$$\left\{ \cdots \leftrightarrow \underset{\downarrow p \leftarrow p-2}{\boxed{+}} \leftrightarrow \boxed{+} \leftrightarrow \boxed{2} \leftrightarrow \boxed{1} \leftrightarrow \cdots \right\}$$

$$\left\{ \cdots \leftrightarrow \boxed{+} \overset{\downarrow p}{\leftrightarrow} \boxed{+} \leftrightarrow \boxed{2} \overset{\downarrow p+1}{\leftrightarrow} \boxed{1} \overset{\downarrow p+2}{\leftrightarrow} \cdots \right\}$$

With backtracking, we can use our new result to evaluate expressions without taking a second scan, which should drop the worst-case time complexity to $\Theta(n)$. However this implementation is far too messy and its correctness cumbersome to prove. Since the two-stack approach already has $O(n)$ complexity in all cases, we shall not adopt this approach.

Testing

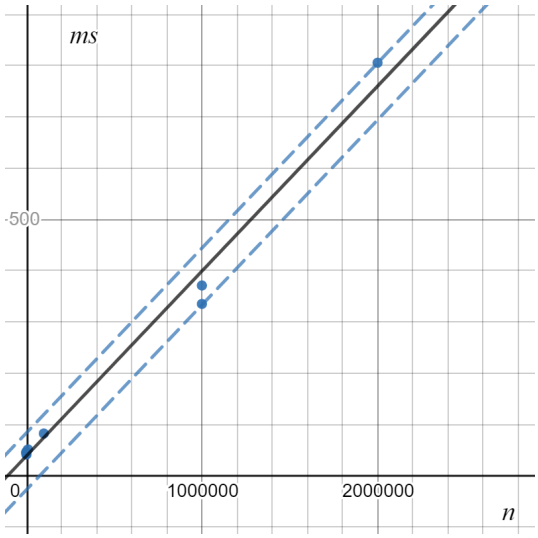
Execution Results

Test case #1:	AC	0.045s	1.63 MB	(10/10)
Test case #2:	AC	0.046s	1.63 MB	(10/10)
Test case #3:	AC	0.042s	1.63 MB	(10/10)
Test case #4:	AC	0.045s	1.63 MB	(10/10)
Test case #5:	AC	0.041s	1.63 MB	(10/10)
Test case #6:	AC	0.051s	3.21 MB	(10/10)
Test case #7:	AC	0.082s	6.05 MB	(10/10)
Test case #8:	AC	0.371s	33.64 MB	(10/10)
Test case #9:	AC	0.335s	33.64 MB	(10/10)
Test case #10:	AC	0.806s	64.06 MB	(10/10)

Resources: 1.865s, 64.06 MB
Final score: 100/100 (100.0/100 points)

We tested the program on the OJ platform and verified that the algorithm is at worst linear in time.

- Running time of the prefix expression evaluation algorithm in ms against n, showing the $O(n)$ complexity.



Possible Improvement

There can be no possible improvement of the algorithm in term of asymptotic notations, since any valid evaluation must be done after reading all the input operators and operands (we do not have any prior knowledge of them), requiring $\Omega(n)$ time. However, it is possible to decrease the actual running time in some special cases. For example, notice that any valid prefix expression must have that

$$\#\text{operand} - \#\text{operator} = 1 \iff 2 \cdot \#\text{operand} - 1 = N$$

a fact that can be shown by induction on the length of expressions. Therefore we could detect any invalidity before evaluating an expression if we counted the numbers of operands and operators respectively when scanning the input, and compare their final difference to 1. This could potentially save time in practice because the evaluation of expressions, particularly multiplication (adding + shifting in CPU, which we assumed to be atomic), is likely more costly than mere counting. If the number of multiplications needed in the evaluation is large, or even more costly operations are involved in the expression, we would expect the an significant save of time.

Reference

> Practical In-Place Mergesort. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.22.8523&rep=rep1&type=pdf