Ending merge sort halfway can improve its efficiency

HuangZhiDun
231870201@smail.nju.edu.cn

Abstract

Merge sort is one of the most commonly used and most efficient sorting algorithms. Usually we compile it using recursive calls until the array to be sorted has only one element. However, in some cases, if we end the recursion halfway and use insertion sort instead, we may get the result faster.

Keywords: Merge Sort, Insertion Sort, Optimization

ACM Reference Format:
HuangZhiDun. 2023. Ending merge sort halfway can improve its efficiency. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

When handling large-scale data, it is quite common to use recursion to break down a big problem into multiple smaller problems, thus reducing the size of the data to be dealt with at one time. Merge sort is one such example, where the array to be sorted is continuously divided into two parts until each sub array has the size one, and then they are merged with each other. This kind of approach, known as the "Divide and Conquer"[1] method, has significant advantages when dealing with large scale data. However, it is not necessary to divide the array when the data size is sufficiently small. And it would be faster to use direct methods like insertion sort. This article will explore, under certain hardware conditions, when the speed of insertion sort becomes faster than merge sort for a given data size after being divided.

2 Key Ideas

Let's look at an example here, where the merge sort algorithm[2] and the insertion sort algorithm[3] are asked to deal with an array that is suffciently small(has the length of 5 in this case). We'll implement these two algorithms in python, and let's see which algorithm takes less steps to sort the array out.

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

According to the interpreter,it takes 56 steps for the insertion sort to sort out the array in the worst case.

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)
    return merge(left_half, right_half)
def merge(left, right):
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    while i < len(left):
        merged.append(left[i])
        i += 1
    while j < len(right):
        merged.append(right[j])
        j += 1
    return merged
```

According to the interpreter,it takes 120 steps for the merge sort to sort out the array in the worst case.

Here's a template for how to combine merge sort with insertion sort properly.

```
def sort(arr):
```

```
if len(arr) <= CONSTANT:
    return insertion_sort(arr)
else:
    return merge_sort(arr)
```

3 Technical Meat

So far, we have only been measuring the efficiency of algorithms based on the vague concept of "fast" or "slow". In order to compare the efficiency of algorithms more accurately, we quantify the time required by algorithms using the concept of complexity.
First, we introduce the big o notation O. Here's it definition. Let F and G be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $F(x)$ is $O(G(x))$ if there are constants C and k such that $|F(x)| \leq |kG(x)|$ whenever $x > k$.
As you can see, the big o notation O is only useful when dealing with sufficiently large n for it only cares about situations when $x > k$.[4]

Initially, let's calculate the complexity of merge sort when it meets an array with the length n which is totally reversed.(Worst Case)

Assume it'll take $f(n)$ steps for an array with the length of n to be completely sorted by merge sort.
For the first step, we have

$$f(n) = 2f(n/2) + n$$

Taking it a step further in recursion.

$$f(n/2) = 2f(n/4) + n/2$$

And as this goes on, eventually we'll get

$$f(n) = nf(1) + n\log_2 n$$

And as we all know, arrays with only one element don't need to be sorted, so $f(1) = 1$

Finally, we get $f(n) = O(n\log_2 n)$

Then let's look at insertion sort.(Also the worst case, also an array with the the length n.

In this case, each element in the array needs to be compared for n - 1 times. So $f(n) = n(n-1)/4$, which is $O(n^2)$. Now we reach the conclusion that when dealing with large scale data, merge sort comes much faster.

Then ,let's focus on how many times each expression are called when dealing with these two algorithms.

We should analyze under the same hardware conditions, so let's assume all assignment statements take time t1, all comparison statements take time t2, all return statements take time t3, and all function creation statements take time t4.

The for expression gets called n times. And the assignment expression to follow gets called $(n-1)$ times each, which is $2(n-1)$ times in total.The while expression, cause it contains two comparing statement, takes $(n-1) * (n-2)$ times. The last assigning statement gets called $(n-1)$ times. To sum up, it takes$(n^2 - 2n + 2) * t1 + (n^2 + n - 2) * t2$.

As for the merge sort, because of the existence of logarithm, we cannot accurately list out how many times each expressions are called, we can only induct the final answer.

At each level of recursion, the if len(arr) <= 1 comparison is performed once. This occurs for a total of $\log_2 n$ levels.

The assignment operations during the merging process occur a total of n times.

The return arr statement is executed once for each recursive call in the base case, which happens a total of n times.

The functions included also get called n times in total.

So, we get the total time consumed to be $n * t1 + \log_2 n * t2 + n * t3 + n * t4$.

For different computers and compilers, $t1, t2, t3, t4$ vary. When $(n^2 - 2n + 2) * t1 + (n^2 + n - 2) * t2$ is smaller than $n * t1 + \log_2 nt2 + n * t3 + n * t4$, we can use insertion sort instead of merge sort for higher efficiency.

4 Reference

References

[1] Bentley, J. L. (1980). Multidimensional divide-and-conquer. Communications of the ACM, 23(4), 214-229..
[2] Cole, R. (1988). Parallel merge sort. SIAM Journal on Computing, 17(4), 770-785.
[3] Sort, I. (2006). Insertion Sort. Sort, 9(4), 2..
[4] Danziger, P. (2010). Big o notation. Source internet: http://www. scs. ryerson. ca/mth110/Handouts/PD/bigO. pdf, Retrieve: April, 1(1), 6.