Multithreaded Merge Sort

Merge sort using multiple threads.

Merge Sort

Merge sort is a typical text-book example of a recursive algorithm and the poster-child of the divide and conquer strategy. The idea is very simple: we divide the array into two equal parts, sort them recursively, then combine the two sorted arrays. The base case for recursion occurs when the size of the array reaches a single element. An array consisting of a single element is already sorted.

The running time for a recursive solution is expressed as a *recurrence equation*. An equation or inequality that describes a function in terms of its own value on smaller inputs is called a recurrence equation. The running time for a recursive algorithm is the solution to the recurrence equation. The recurrence equation for recursive algorithms usually takes on the following form:

Running Time = Cost to divide into n subproblems + n * Cost to solve each of the n problems + Cost to merge all n problems

In the case of merge sort, we divide the given array into two arrays of equal size, i.e. we divide the original problem into sub-problems to be solved recursively.

Following is the recurrence equation for merge sort:

Running Time = Cost to divide into 2 unsorted arrays + 2 * Cost to sort half the original array + Cost to merge 2 sorted arrays

sort half the original array — cost to merge 2 sorted arrays

$$T(n) = Cost\ to\ divide\ into\ 2\ unsorted\ arrays + 2 * T(\frac{n}{2}) + Cost\ to\ merge\ 2\ sorted\ arrays when n > 1$$

$$T(n) = O(1)\ \ when\ n = 1$$

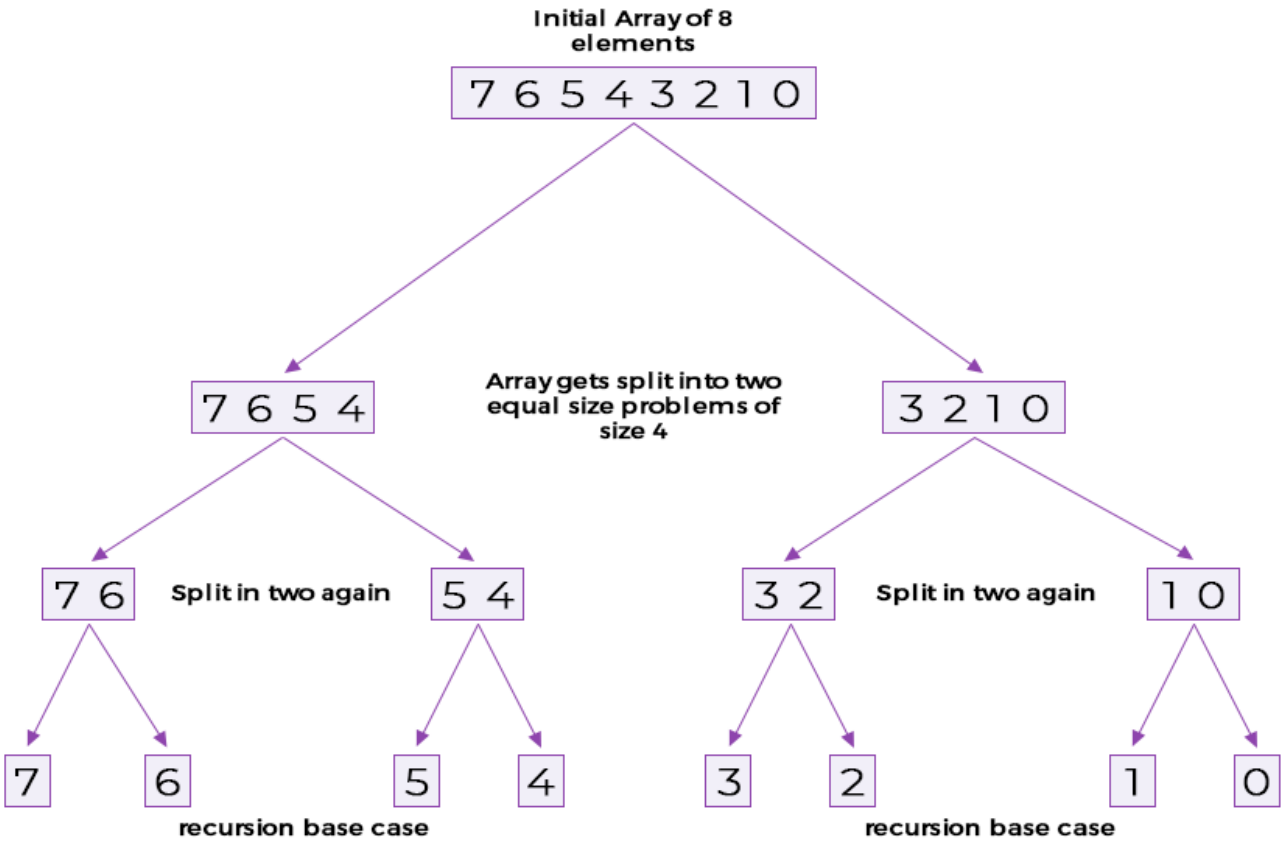
Remember, the *solution* to the recurrence equation will be the *running time* of the algorithm on an input of size n. Without getting into the details of how we'll solve the recurrence equation, the running time of merge sort is

$$O(nlgn)$$

where *n* is the size of the input array.

Merge Sort Recursion Tree

Below is a pictorial representation of how the merge sort algorithm works:



Merge sort lends itself very nicely to parallelism. We can also use the multiprocessing module to create multiple sub-processes to implement merge sort, but in this example, we'll stick to using the threading module. Note that the subdivided problems or subarrays don't overlap with each other so each thread can work on its assigned subarray without worrying about synchronization with other threads. There is no data or state being shared between threads. There's only one caveat: we need to make sure that peer threads at each level of recursion finish before we attempt to merge the subproblems.

Let's first implement the single-threaded version of Merge Sort and then attempt to make it multithreaded. Note that merge sort can be implemented without using extra space but the implementation

becomes complex, so we'll allow ourselves the luxury of using extra space and stick to a simple-to-follow implementation.

```
0.  def merge_sort(start, end, input):
1.      global scratch
2.
3.      if start == end:
4.          return
5.
6.      mid = start + math.floor((end – start) / 2)
7.
8.      # sort first half
9.      merge_sort(start, mid, input)
10.
11.      # sort second half
12.     merge_sort(mid + 1, end, input)
13.
14.     # merge the two sorted arrays
15.     i = start
16.     j = mid + 1
17.
18.     for k in range(start, end + 1):
19.         scratch[k] = input[k]
20.
21.     k = start
22.     while k <= end:
23.
24.         if i <= mid and j <= end:
25.             input[k] = min(scratch[i], scratch[j])
26.
27.             if input[k] == scratch[i]:
28.                 i += 1
29.             else:
30.                 j += 1
31.
32.         elif i <= mid and j > end:
33.             input[k] = scratch[i]
34.             i += 1
35.         else:
36.             input[k] = scratch[j]
37.             j += 1
```

```
38.

39.         k += 1
```

In the above single-threaded code, the opportunity to parallelize the processing of each sub-problem exists on **line 9** and **line 12**. We create two threads and allow them to carry on processing the two subproblems. When both are done we combine the solutions. Note that the threads work on the same array but on completely exclusive portions of it. There's no chance of synchronization issues coming up.

Below is the multithreaded code for Merge sort. Note that the code is slightly different than the single-threaded version to account for changes required for concurrent code.

```
def merge_sort(start, end, input):
    global scratch

    if start == end:
        return

    mid = start + math.floor((end – start) / 2)

    # sort first half
    worker1 = Thread(target=merge_sort, args=(start, mid, inpu
t))

    # sort second half
    worker2 = Thread(target=merge_sort, args=(mid + 1, end, in
put))

    worker1.start()
    worker2.start()
    worker1.join()
    worker2.join()

    # merge the two sorted arrays
    i = start
    j = mid + 1

    for k in range(start, end + 1):
        scratch[k] = input[k]

    k = start
    while k <= end:

        if i <= mid and j <= end:
            input[k] = min(scratch[i], scratch[j])

            if input[k] == scratch[i]:
                i += 1
            else:
                j += 1
```

```
        elif i <= mid and j > end:
            input[k] = scratch[i]
            i += 1
        else:
            input[k] = scratch[j]
            j += 1

        k += 1
```

```
import random
import math
from threading import Thread
scratch = None


def merge_sort(start, end, input):
    global scratch

    if start == end:
        return

    mid = start + math.floor((end - start) / 2)

    # sort first half
    worker1 = Thread(target=merge_sort, args=(start, mid, input))

    # sort second half
    worker2 = Thread(target=merge_sort, args=(mid + 1, end, input))

    worker1.start()
    worker2.start()
    worker1.join()
    worker2.join()

    # merge the two sorted arrays
    i = start
    j = mid + 1

    for k in range(start, end + 1):
        scratch[k] = input[k]

    k = start
    while k <= end:

        if i <= mid and j <= end:
            input[k] = min(scratch[i], scratch[j])

            if input[k] == scratch[i]:
                i += 1
            else:
                j += 1

        elif i <= mid and j > end:
            input[k] = scratch[i]
            i += 1
        else:
            input[k] = scratch[j]
            j += 1

        k += 1
```

```
def create_data(size):
    unsorted_list = [None] * size
    random.seed()

    for i in range(0, size):
        unsorted_list[i] = random.randint(0, 1000)

    return unsorted_list


def print_list(lst):
    end = len(lst)
    for i in range(0, end):
        print(lst[i], end=" ")


if __name__ == "__main__":
    size = 12
    scratch = [None] * size
    unsorted_list = create_data(size)

    print_list(unsorted_list)
    merge_sort(0, size - 1, unsorted_list)
    print("\n\n")
    print_list(unsorted_list)
```

We create two threads on lines **16** and **19** and then wait for them to finish on **lines 23-24**.

← Back

Asynchronous to Synchronous Problem

Next →

Epilogue

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/multithreaded-merge-sort__interview-practise-

problems__python-concurrency-for-senior-engineering-interviews)