ECE374 Fall2020

Lab3: Optimized Merge Sort

Name: Zhang Yichi 3180111309

Oct. 20^{th} 2020

1 Introduction

In this lab, we will optimize the merge sort by modifying the sorting of small size problems. Instead of dividing the problem size to 1 by recursion then merge them, we stop when the leaves contain less than k elements and apply insertion sort to sort the sublists since insertion sort is more efficient in small size problems.

2 Python Code

```
def insertionsort(vec):
    for i in range(1,len(vec)):
        num = vec[i]
        j = i - 1
        while j >= 0 and vec[j] > num:
            vec[j+1] = vec[j]
            j -= 1
        vec[j+1] = num

def merge(vec,mid):
    i = 0
    j = mid + 1
    n = len(vec)
    new_vec = []
    for _ in range(n):
        if j > n:
            new_vec.append(vec[i])
            i += 1
```

```
        elif i > mid:
            new_vec.append(vec[j])
            j += 1
        elif vec[i] > vec[j]:
            new_vec.append(vec[j])
            j += 1
        else:
            new_vec.append(vec[i])
            i += 1
    vec = new_vec

def mergesort(vec,k):
    if len(vec) > k:
        insertionsort(vec)
        return
    mid = len(vec) // 2
    mergesort(vec[0:mid],k)
    mergesort(vec[mid:],k)
    merge(vec,mid)
```

3 Test Example

Here, 20 random floating numbers are generated by np.random.randn. We use the optimized merge sort with $k = 3$ (the best k is around 60 to 80, here we just choose a k to test the algorithm) to sort them and output the result.

```
vec = np.random.randn(20)
print(f"Before sorting {vec}")
print()
mergesort(vec,3)
print(f"After sorting {vec}")
```

```
Before sorting [-1.84244125  1.84361641 -1.56826877  0.14843406 -1.15003046  0.05074022
  0.78465603 -0.89999407  0.72486114 -0.85188455 -0.443877    0.33648184
  1.02684664 -1.58713181 -0.87028225  0.04094225  0.30284608  0.77462343
 -0.41739672  0.32193534]

After sorting [-1.84244125 -1.58713181 -1.56826877 -1.15003046 -0.89999407 -0.87028225
 -0.85188455 -0.443877   -0.41739672  0.04094225  0.05074022  0.14843406
  0.30284608  0.32193534  0.33648184  0.72486114  0.77462343  0.78465603
  1.02684664  1.84361641]
```

Figure 1: Example inputs and outputs for 20 random floating numbers

4 Indication

We first build a normal merge sort where we divide the lists in half, merge sort the two sublists and finally merge them back to one list.

For merging the two sublists, we take the first one of the elements in both sublists, compare them, add the smaller one to a new list and take the latter one element of the smaller one and repeat the compare, add-to-new-list, take-new-element process until one sublist is empty. After that, we add all the rest elements in the nonempty sublist to the new list.

The modification is that instead of dividing the problem size to 1, we stop when the problem size is less than k and call the insertion sort to sort the sublists.

For the insertion sort, we take the elements one by one in order, compare them with the ones ahead, find the place they should be and insert them into those places.

5 Complexity Analysis

For the original merge sort, the height of the recursion tree is $\frac{n}{2^h} = 1 \Rightarrow h = logn$

For the optimized merge sort, the height of the recursion tree is $\frac{n}{2^h} = k \Rightarrow h = log\frac{n}{k}$ approximately since k is not necessarily be a power of 2.

The width of one layer of the recursion tree is n, the total number of the elements in the list, which stays the same.

The time complexity of the insertion sort is $O(k^2)$

The time complexity of the merge sort is $O(nlog\frac{n}{k}) + O(k^2)$

To best optimize the time complexity, we need $O(k^2) \leq O(nlog\frac{n}{k})$ and therefore the final time complexity will be $O(nlog\frac{n}{k}) = O(nlogn)$.