Assignment : Fuzzy Sorting of Intervals
Team Members : Amit Shetty and Vamshi Goud
Course : ITCS 6114 – Data Structures and Algorithms

Source Code:

Language : Python

```python
"""
ITCS 6114  - Fuzzy Interval Sorting
"""


def perform_fuzzy_sort(intervals_list, start, end):
    """
    Perform the sorting operation by calculating the pivot in this case the intersection and spliting the input list
    into 2 based on the intersection
    :param intervals_list:
    :param start: start list index for the sub list
    :param end: end list index for the sub list
    """
    if start < end:
        intersection = calculate_intersection(intervals_list, start, end)
        right_partition = perform_right_partition(intervals_list, start, end, intersection)
        left_partition = perform_left_partition(intervals_list, start, right_partition, intersection)
        perform_fuzzy_sort(intervals_list, start, left_partition - 1)
        perform_fuzzy_sort(intervals_list, right_partition + 1, end)

def calculate_intersection(intervals_list, start, end):
    """
    Calculates the intersections of the ranges to return an interval otherwise returns the interval at the end of the list
    based on Lomuto's paritioning algorithm
    :param intervals_list:
    :param start:
    :param end:
    :return: intersection of the interval items
    """
    intersection = intervals_list[end]
    for i in range(start, end):
        if intervals_list[i][0] <= intersection[1] and intervals_list[i][1] >= intersection[0]:
            if intervals_list[i][0] > intersection[0]:
                intersection = (intervals_list[i][0], intersection[1])
            if intervals_list[i][1] < intersection[1]:
                intersection = (intersection[0], intervals_list[i][1])
    return intersection

def perform_right_partition(intervals_list, pivot, end, intersection):
    """
    Splits the sublist based on the location of the intersection from its left side

    :param intervals_list:
    :param pivot:
    :param end:
    :param intersection:
    :return: New parition index for the right side of the intervals list
    """
    right_part_start = pivot - 1
    for j in range(pivot, end):
```

```python
        if intervals_list[j][0] <= intersection[0]:
            right_part_start += 1
            intervals_list[right_part_start], intervals_list[j] = intervals_list[j], intervals_list[right_part_start]
    #Swap the variables
    intervals_list[right_part_start + 1], intervals_list[end] = intervals_list[end], intervals_list[right_part_start + 1]
    return right_part_start + 1

def perform_left_partition(intervals_list, start, right, intersection):
    """
    Splits the sublist based on the location of the intersection from its left side

    :param intervals_list:
    :param start:
    :param right:
    :param intersection:
    :return: New partition index for the left side of the parition list
    """
    left_part_start = start - 1
    for j in range(start, right):
        if intervals_list[j][1] < intersection[1]:
            left_part_start += 1
            intervals_list[left_part_start], intervals_list[j] = intervals_list[j], intervals_list[left_part_start]
    # Swap the variables
    intervals_list[left_part_start + 1], intervals_list[right] = intervals_list[right], intervals_list[left_part_start + 1]
    return left_part_start + 1

def perform_fuzzy_interval_test_cases(case_number, list_of_intervals):
    print("\nTest Case #{}\n\nInput Elements:\n".format(case_number))
    print(*list_of_intervals, sep='\n')
    perform_fuzzy_sort(list_of_intervals, 0, len(list_of_intervals) - 1)
    print("\nTest Case #{}\n\nOutput Elements:\n".format(case_number))
    print(*list_of_intervals, sep='\n')

def main():
    # Test Case 1
    list_of_intervals_1 = [(5,7),
                           (1,3),
                           (4,6),
                           (8,10)]
    perform_fuzzy_interval_test_cases(1, list_of_intervals_1)

    # Test Case 2
    list_of_intervals_2 = [(6, 7),
                           (9, 11),
                           (13, 14),
                           (3, 7),
                           (11, 15),
                           (13, 14),
                           (12, 14),
                           (14, 15),
                           (9, 15),
                                  (5, 7),
                                  (7, 9),
                                  (1, 5),
                                  (1, 9),
                                  (6, 10)]
           perform_fuzzy_interval_test_cases(2, list_of_intervals_2)

       if __name__ == '__main__':
           main()
```

```
/Users/amitshetty/.pyenv/versions/3.7.2/bin/python /Users/amitshetty/PycharmProjects/PythonSandbox/03_Fuzzy_Interval_Sort/fuzzyintervalsort.py

Test Case #1

Input Elements:

(5, 7)
(1, 3)
(4, 6)
(8, 10)

Test Case #1

Output Elements:

(1, 3)
(4, 6)
(5, 7)
(8, 10)

Test Case #2

Input Elements:

(6, 7)
(9, 11)
(13, 14)
(3, 7)
(11, 15)
(13, 14)
(12, 14)
(14, 15)
(9, 15)
(5, 7)
(7, 9)
(1, 5)
(1, 9)
(6, 10)

Test Case #2

Output Elements:

(1, 5)
(6, 10)
(5, 7)
(7, 9)
(6, 7)
(1, 9)
(3, 7)
(9, 11)
(12, 14)
(13, 14)
(14, 15)
(11, 15)
(13, 14)
(9, 15)

Process finished with exit code 0
```

Analysis of run time of fuzzy interval sort:

- The algorithm for the fuzzy interval sort is similar to that of quick sort.
- The run time in average case is O (nlgn) which is similar to that of quick sort average case run time.
- But when all the intervals combine which happens during the best case analysis the run time is O(n) ie; linear run time.
- The for loop makes n comparisons in the perform_left_partition function and these comparisons are made through the left part
- Similarly the for loop in the calculate_intersection and perform_right_partition functions also makes n comparisons.
- Summing up all these values we get the best case running time and the worst case running time ie; O(n) since we are guaranted to have n operations.
- The depth that is expected is lg n since the base values are just changed by a constant factor and therefore the algorithm in general has a running time of O (nlg n).
- The run time for the worst case would be O(nlg n) in this case all the elements would overlap with the pivot and they would fall into the following intervals and no recursion occurs on the empty intervals.
- Average case is similar to that of worst case as the overlaps between the intervals would not be maximum.
- The average case run time is O(nlg n).
- So the conclusion of the analysis is that the run time depends on the number of overlaps between the intervals the more the number of overlaps between the intervals the lesser the run time. Therefore if the number of overlaps are less the run time is comparatively higher.