Assignment 3 CSCI4041 Wyatt Kormick
DUE: Thursday, October 13, 10:00pm (+1hr55min grace period) Fall 2016

IF YOUR ASSIGNMENT IS NOT ACCEPTED BY TURNITIN BECAUSE IT DOES NOT CONTAIN 120+ WORDS OF DIGITAL TEXT, 3 POINTS WILL BE DEDUCTED FROM YOUR HOMEWORK. If it is rejected for other reasons, you will not incur a point deduction.

1. Fuzzy Sorting of Intervals

Consider a sorting problem in which you are given *n* closed intervals of the form [a, b], where a ≤ b. A fuzzy sort of these intervals creates an ordering $\{[a_1,b_1], [a_2,b_2], \ldots, [a_n,b_n]$ such that for each interval, there exists a c_i in $[a_i, b_i]$ satisfying $c_1 \le c_2 \le \ldots \le c_n$. For example, given the intervals { [2,3], [1,5], [4,8], [2,9] } the following orderings constitute a fuzzy ordering:

- { [1,5], [2,9], [2,3], [4,8]} because we can select c_i for each as { 1, 2, 3, 4 }
- { [2,3],[4,8],[1,5],[2,9] } because we can select c_i for each as { 2, 4, 5, 9 }

a. Write pseudocode for a randomized algorithm for fuzzy sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left end-points (the a_i values), but it should take advantage of overlapping intervals to improve the running time. As the intervals overlap more and more, the problem of fuzzy-sorting the intervals becomes progressively easier. Your algorithm 'should take advantage of such overlapping to the extent that it exists.

FUZZY-SORT Algorithm found at end of assignment

b. Describe how your algorithm runs in expected linear time THETA(n) when all intervals overlap, such that there exists some x in $[a_i,b_i]$ for all *i*.

The algorithm in this case will attempt to find the element that exists in every interval. This element, if it exists, will be in the middle partition, and as such when the algorithm attempts to FUZZY-SORT the partitions to the left and right, the functions will exit, doing no work.

2. Understanding Bounds on Comparison Sort Algorithms

Show that there is no comparison sort whose running time is linear for at least half of the n! inputs of length n.

The only time a running time is linear in a binary decision tree for a specific value is when the value is within a certain distance to the root.

Using theorem 8.1, we can say that $\frac{n!}{2} \le n! < 2^h$

$h \ge \log(\frac{n!}{2})$ = log(n!)-log(2)=log(n!)-1 = Θ(n*log(n))

What about a fraction of 1/n of the inputs of length n?

The same is true for 1/n inputs of length n

From Th 8.1: $\frac{n!}{n} \le n! < 2^h$

$h \ge \log\left(\frac{n!}{n}\right) = \log(n!) - \log(n) = \theta(n * \log(n))$

3. **Write pseudocode for an algorithm** that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range [a..b] in O(1) time. Briefly justify a tight running time bound of THETA(n+k) preprocessing time.

Algorithm found at end of assignment

Preprocessing has two for loops that go through k operations, and a for loop that does n operations for O(n+2k) which is bound by Theta(n+k)

4. **Show that the following "greedy" strategy does not always produce an optimal solution.**
   a. Consider a collection of objects of differing weights and prices. The unit cost can be expressed as $ per pound. The problem is to maximize profit by selecting the subset of elements that together weigh less than W, but have maximal value over all subsets. The greedy strategy is to continually choose the element from those remaining with the highest unit cost until the weight limit is reached.

      Consider the following example:

      [[Weight, Unit Cost, Total Cost][5g, $2.20/g, $11] [2g, $2.25/g, $4.50] [1g, $1/g, $1]]

      If our weight w is 5g, the greedy solution would take 2g+2g+1g for a profit of $10. A better solution would be to take just take the 5g for $11.

   c. Similarly, consider coins with values 1, 5, 7, and 15 cents. If you are given some value x > 1 and asked to produce a minimally sized collection of coins that together sum to x, show that a greedy strategy of continually choosing the highest denomination that is less than the remaining sum will not produce an optimal solution.
      a. A counterexample would be 19 cents. The greedy solution would be to take 15+1+1+1+1, which is 5 coins, when a better solution would be to take 7+7+5 for 3 coins. Many amounts with a remainder of 4 from the largest denomination will be troublesome since the only way to make four is to take 1+1+1+1.

5. **Application of Dynamic Programming**

You wish to exchange currency c_1 for c_n. Instead of a direct exchange, it might be beneficial to make a series of exchanges through other currencies. You want to select a series of exchanges that maximize the units of c_n. Consider the following:

- There exists n currencies and an exchange rate r_{ij} for each trade from c_i to c_j, such that $d*r_{ij}$ = units of c_j after exchanging d units of c_i.
- No commission is charged for an exchange,
- You can trade each currency at most once,
- There is a fixed order to the currencies which trades must adhere to (for example, c_1 to c_3 to c_8 to c_{10} is allowed but c_1 to c_8 to c_3 to c_{10} is not allowed because 1, 8, 3, 10 is not ordered).

a. Briefly describe how to derive a solution using a brute force approach and justify a runtime upper bound.

In order, a currency can either be exchanged to or not exchanged to with no other option, as they cannot go backwards in order, so the total number of permutations of exchange possibilities is similar to 2^n. The brute force method would make recursive calls in order to test, in order from c1 to cn, the different permutations. If a specific permutation is currently being tested and is already worse than the current best one, that iteration would exit.

b. Write the recursive definition for the best series of exchanges (see equation 15.7 as an example).

c. Consider a modification in which a commission m_k is charged for k trades, where for all k, m_k is arbitrary (unknown). Show that this modification means the problem does not necessarily exhibit optimal substructure.

Pseudocode

```
//Assuming The Intervals are nested arrays of length two inside a larger array
INTERSECTION-SELECTION(A,L,R)
    If L<R
        k =RANDOM-INT(L,R) //Random int generates a random integer in the given range
        l = A[k][0]
        r = A[k][1]
        for i = 0 to A.length-1
            if A[i][0] >= 1 and A[i][1] <= r
                l = MAXIMUM(l, A[i][0])
                r = min(r, A[i][1])
        return l

FUZZY-PARTITION(A,L,R,j)
    if L < R
        lft, mid, rt, i = L
        while i < R
            left = A[i][0]
            right = A[i][1]
            if left <= j and j <= right
                Swap A[i] with A[mid]
                mid++
            else if right < j
                Swap A[mid] and A[i]
                Swap A[mid] and A[lft]
                lft++
                mid++
            i++
    returns lft, mid

FUZZY-SORT(A,L,R)
    if L < R
        j = INTERSECTION-SELECTION(A,L,R)
        mlft, mrt = FUZZY-PARTITION(A,L,R,j)
        FUZZY-SORT(A,L,mlft)
        FUZZY-SORT(A,mrt,R)
```

```
PREPROCESSING(A)
    Let k be the largest element in A
    Let C[0...k] be a new array
    for i = 0 to k                  //O(k)
        C[i] = 0
    for j = 1 to A.length           //O(n)
        C[A[j]] = C[A[j]]+1
    for i = 1 to k                  //O(k)
        C[i] = C[i]+C[i-1]
    return C
//Exactly the same as the preprocessing part
//    of COUNTING-SORT
//Returns an array that contains the sum of the count of each
//    element in A from 0 to k

QUERY(A, a, b)
    Let C[] be a new array
    if b > a
        C = Preprocessing(A) //O(n+2k)
        return C[b + 1] - C[a+1] //O(1)
//Subtracts the total amount of elements at the lower end
//from the total amount at the upper end to get the amount
//between
```