CSOR W4231: Analysis of Algorithms (sec. 001) - Problem Set #3

Hongmin Zhu (hz2637) - hz2637@columbia.edu

October 23, 2019

Problem 1

Algorithm:

fuzzy-sort can be implemented by following the general structure of *quick-sort*, which is quick-sorting the left endpoints of each intervals. And this general case will take worst-case running time of $\Theta(n \log n)$. However, we can improve this by considering the fact that overlapping intervals do not require sorting. Because in such situations, we can always choose c_i, c_j from two overlapping intervals that satisfy $c_i <= c_j$. If all the intervals overlap, which is the best case, the running time will be $\Theta(n)$.

So, instead of partitioning the inputs into 2 parts as the *quick-sort*, we can partition input intervals into 3 parts: *left*, *middle*, *right*. We first pick a random pivot interval as the region to split the inputs. Intervals that are smaller than the pivot will be swap into *left*, intervals that are bigger than pivot will be swap into *right*, and intervals inside *middle* overlap, which we consider as the same. For example: two intervals $\boldsymbol{i}([a_i, b_i])$, $\boldsymbol{j}([a_j, b_j])$, there are three situations:

(1) $\boldsymbol{j} < \boldsymbol{i}$, if $b_j < a_i$
(2) $\boldsymbol{j} > \boldsymbol{i}$, if $a_j > b_i$
(3) $\boldsymbol{j} = \boldsymbol{i}$, others

Since intervals inside *middle* are the same, we only need to sort *left* and *right*, the algorithm can be more efficient when there are more intervals in the inputs.So we modify *quick-sort* to fit this problem by changing PARTITION to return an interval.

Pseudo-code:

```
PARTITION (A, B, p, r)
 1  y ← RANDOM(p, r)
 2  exchange A[y] ↔ A[r]
 3  exchange B[y] ↔ B[r]
 4  a ← A[r]
 5  b ← B[r]
 6  i ← p-1
 7  j ← r
 8  k ← p
 9  while k < j and k < r do
10     if B[k] < a do
11        i ← i+1
12        exchange A[i] ↔ A[k]
13        exchange B[i] ↔ B[k]
14        k ← k+1
15     if A[k] > b do
16        j ← j-1
17        exchange A[j] ↔ A[k]
18        exchange B[j] ↔ B[k]
19     else do
20        a ← max(A[k], a)
21        b ← min(B[k], b)
22        k ← k+1
23  exchange A[i+1] ↔ A[r]
24  exchange B[i+1] ↔ B[r]
25  return i+1, j
```

```
FUZZY-SORT (A, B, p, r)
1  if p < r do
2     left, right ← PARTITION(A, B, p, r)
3     FUZZY-SORT(A, B, p, left)
4     FUZZY-SORT(A, B, right, r)
```

Analysis:

On lines 1 through 3 of PARTITION, we select a random pivot interval as the initial region of overlap ***middle*** ($[a,\ b]$). On lines 4 through 22, we loop over all the inputs, except the one we set as the initial region. At each iteration, we see what is the relationship between current interval with the region ***middle***, if the current interval is smaller than region ***middle***, we put it into ***left***, if the current interval is bigger than region ***middle***, we put it into ***right***, if it overlap ***middle***, we update region of overlap to $[a,\ b] = [a,\ b] \cap [a_k, b_k]$. So, PARTITION will always divide inputs into three regions. The FUZZY-SORT part is identical to *quick-sort*, after we get the *middle* region, we

will recursively call FUZZY-SORT on the left part and right part.

Running Time:

The worst case will be there is no overlap between intervals, which means the *middle* region will only contains one interval. And since the *middle* region is chosen randomly, recursion runs on *left* and *right*, which have the size of $\lfloor \frac{n}{2} \rfloor$, and PARTITION has the worst-case running time of $\Theta(n)$, so the recurrence for the running time is

$$T(n) \leq 2T(n/2) + \Theta(n)$$

which can be solved to $\Theta(nlogn)$. If in the best case, the intervals all overlap at c_*, every interval will be within *middle*, *left* and *right* will be empty. As a result, there will be no recursion, the running time will then be $\Theta(n)$.

Problem 2

Algorithm:

Use *radix-sort* to sort the strings from most significant digits to least significant digits. In order to achieve $O(m)$ running time, we should sort each digit at most once. We first use *counting-sort* to sort the most significant digit, we will then have strings with first letter 'a' to strings with first letter 'z', which divides all strings into several blocks. Then for each block, we recursively run *counting-sort* on the second digit of current block. Because strings in each block have the same first letter, so the overall ordering will not be changed. After second digit is sorted, there will be more blocks of second digit from 'a' to 'z', we then recursively run *counting-sort* on the third digit of each block of second digit. So the idea is to combine *recursion* and *counting-sort*.

Because the length of strings are different, so when sorting a specific digit, some strings might have letters in that digit, some don't. So to handle this, we will treat those empty digits as the smallest, which means they will come first in sorting.

Pseudo-code:

```
SORT (A, L, p, r, d)
 1  if p+1 ≤ r
 2     for i ← 0 to 26
 3        do C[i] ← 0
 4     for i ← p to r
 5        if L[i] < d
 6           do C[0] ← C[0] + 1
 7        do C[A[i].charAt(d) − 'a' + 1] ← C[A[i].charAt(d) − 'a' + 1] + 1
 8     for i ← 1 to 26
 9        do C[i] ← C[i] + C[i − 1]
10     for i ← p to r
11        if L[i] < d
12           temp[C[0] + +] = A[i]
13        temp[C[A[i].charAt(d) − 'a' + 1] + +] = A[i]
14     for i ← p to r
15        A[i] = temp[i + p]
16     for i ← 1 to 26
17        SORT (A, L, p + C[i], p + C[i + 1] − 1, d+1)
```

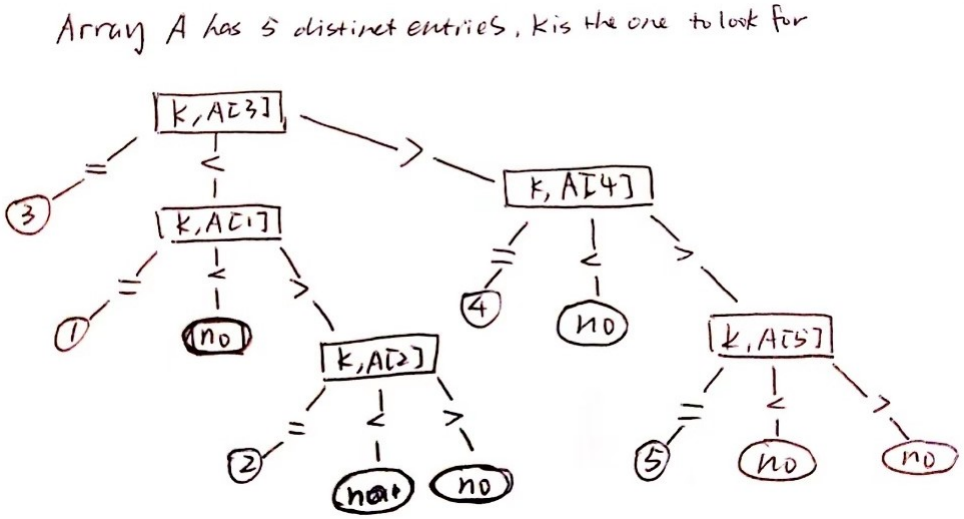
Analysis:

The algorithm is based on *counting-sort* code, instead of sorting from the least digit as the example in the class, we sort strings from most significant digit. After sorting

each digit, the strings with the same first letter will be grouped together, each group, or we can say "bucket" corresponds to letter 'a' to 'z'. In each bucket, the algorithm recursively sort the next digit, and again, strings will be grouped into buckets with same digit. So, using this method, we can make sure that each character of strings is *counting-sorted* only once, and since m is the sum of the length of strings, the total running time is $O(m)$.

Problem 3

(a) A is a sorted array with distinct entries, and since *binary-search* works well on sorted arrays, we start with this searching algorithm and see its decision tree. Suppose the array contains 5 entries, below is the decision tree of *binary-search*:



from the decision tree, we can know that each comparison will have three outputs $<, =, >$, and some of the outputs are duplicated, i.e. the "no" output appears at many different leaves. So in total, there are 6 unique leaves (5 keys and no if there is no match). So, for any *comparison-based* search algorithm searching an item x in A, we can know:

(1) there will be $n + 1$ distinct comparison outputs in the worst case.
(2) the decision tree will have at least $n + 1$ leaves.
(3) each node will have three children $(<, =, >)$

So since a ternary tree of height h will have at most 3^h leaves, and we have n entries in A, we know that the height of corresponding decision tree will have height at least $log_3 n$, and $log_3 n \in \Omega(logn)$, we can say that for any *comparison-based* search algorithm, we have a lower bound of $\Omega(logn)$.

(b) if the output is just "yes" or "no", then the decision tree of *comparison-based* searching algorithm on array A will have

(1) $n + 1$ distinct leaves ("yes" for all keys and a "no")
(2) each node will have two children $(=, \neq)$

So, the height of corresponding decision tree will have height at least $log_2 n$, and $log_2 n \in \Omega(logn)$, we can say that for any *comparison-based* search algorithm that outputs "yes" or "no", we have a lower bound of $\Omega(logn)$.

Problem 4

Algorithm:
We use dynamic programming to solve this problem. Pre-processing steps are below:

(1) add 0 to the front of array L and n to the end of array L
(2) sort array L in ascending order

Now, we can define the problem as follows: suppose $L[i...j]$ indicates indexes i from j of array L. We define sub-problem (i, j) to be the best break of $String[L[i+1], L[j]]$, and for this sub-problem, we care about how to break $L[i+1, j-1]$. And we say the break point of (i, j) is $k, i < k < j$, we can divide problem (i, j) to two sub-problems (i, k) and (k, j). Therefore we can know the equation:

$$cost[i,j] = \begin{cases} 0 & j - i \leq 1 \\ \min_{i<k<j} cost[i,k] + cost[k,j] + (L[j] - L[i]) & j - i > 1 \end{cases}$$

To be more specific, if $j - i \leq 1$, the string is $String[L[i+1], [j]]$, if the length is 1, then we can't break it, so the cost is 0; if the length is two, there is no break point from array L inside string, so we don't break it, the cost is 0. if $j - i > 1$, the length of string is more than 1, so there could be break points inside the string, the total cost now is $L[j] - L[i]$, which is the cost to break this string, plus *minimum cost to break sub-strings*. So we can build to tables to keep track the cost and the break points.

Pseudo-code:

```
COST (L, n, m)
 1  add 0 to the front of L and n to the end of L
 2  sort array L in ascending order
 3  create 2D matrix cost[m][m] and points[m][m], both start from 1

    // initialize base cases
 4  for i ← 1 to m − 1
 5     do cost[i][i] = 0
 6     do cost[i][i + 1] = 0
 7  cost[m][m] = 0

    // general cases
 8  for len ← 3 to m
 9     for i ← 1 to m − len + 1
10        j = i + len − 1
11        cost[i][j] = ∞
12        for k ← i + 1 to j − 1
13           if cost[i][k] + cost[k][j] < cost[i][j]
```

```
14              cost[i][j] ← cost[i][k] + cost[k][j]
15              points[i][j] ← k
16          cost[i][j] ← cost[i][j] + L[j] − L[i]
```

```
POINTS (L, P, points, i, j)
1   if j − i > 1
2     point ← points[i][j]
3     add point to array P
4     POINTS (L, P, points, i, k)
5     POINTS (L, P, points, k, j)
```

So, minimum cost is $cost[1][m]$, and sequence of breaks are stored in array P.

Analysis:

The algorithm COST above uses three *for-loop*, in the worst-case, each *for-loop* will take $\theta(m)$ time, so the overall running time will be $\theta(m^3)$. The algorithm POINTS outputs the sequence of break points, its running time is $\theta(m)$.