CSOR W4231: Homework 3

Problem 1 (20 points, Graded by Niloofar and Rabia)

a. The number of ways to divide $2n$ numbers into two sorted lists with n numbers is

$$\binom{2n}{n} = \frac{(2n)!}{n!(2n-n)!} = \frac{(2n)!}{n!n!}$$

This counts the number of ways to select n numbers from a list of $2n$ numbers to form the first list and the second list is just formed from the remaining numbers.

b. The decision tree of an algorithm used to merge the two sorted lists must be able to output every permutation of the items on the two sorted lists in the input, thus the number of leaves of the decision tree must be at least the number of these permutations. Our decision tree will be a binary tree containing internal nodes that only compare elements that are not from the same list. From (a) we see that total number of permutations of inputs, which are the number of leaves in our decision tree, is $\frac{(2n)!}{n!n!}$. Thus, the minimum height of the tree must be

$$h = \lg \frac{(2n)!}{n!n!}$$

Using Sterling's Approximation we see that:

$$\begin{aligned}
\frac{(2n)!}{n!n!} &= \frac{\sqrt{2\pi \cdot 2n}\left(\frac{2n}{e}\right)^{2n}\left(1+\Theta\left(\frac{1}{n}\right)\right)}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\left(1+\Theta\left(\frac{1}{n}\right)\right)\right)^2} \\
&= \frac{2\sqrt{\pi n}\left(\frac{2n}{e}\right)^{2n}\left(1+\Theta\left(\frac{1}{n}\right)\right)}{2\pi n\left(\frac{n}{e}\right)^{2n}\left(1+\Theta\left(\frac{1}{n}\right)\right)^2} \\
&= \frac{2^{2n}}{\sqrt{\pi n}}\frac{1}{\left(1+\Theta\left(\frac{1}{n}\right)\right)}
\end{aligned}$$

Substituting that into our minimum height equation we have:

$$\begin{aligned}
h &= \lg \frac{2^{2n}}{\sqrt{\pi n}}\frac{1}{\left(1+\Theta\left(\frac{1}{n}\right)\right)} \\
&= 2n - \lg\left(\sqrt{\pi n}\left(1+\Theta\left(\frac{1}{n}\right)\right)\right) \\
&= 2n - o(n)
\end{aligned}$$

Since the height of the tree corresponds to the number of comparisons, we see that we must perform at least $2n - o(n)$ comparisons.

c. Suppose there is a merging algorithm in which two consecutive items from different lists are not compared. Suppose we run this algorithm on two lists with consecutive elements that are not compared. Because elements are consecutive in the final list, they behave exactly the same with respect to comparison to other elements. So, imagine an input that is the same as the original input, but with the two consecutive elements swapped between the lists. Then the algorithm will place them in the same order, except with the two consecutive elements swapped - this is a wrong order. So, the algorithm is incorrect, and by contradiction, any correct algorithm must compare consecutive elements from different lists.

d. Fix any sorting algorithm, and consider the input lists $[a_1, ..., a_n]$ and $[b_1, ..., b_n]$. Suppose that the final sorted order is $[a_1, b_1, a_2, b_2, \; ..., a_n, b_n]$. From part *c.*, we know that a_i must be compared to b_i for i from 1 to n, and b_i must be compared to a_{i+1} for i from 1 to $n-1$. So, at least $2n-1$ comparisons must occur.

Problem 2 (15 points, Graded by Aditya)

Form an array A of size n indexed from 1 to n. Each element of this array is considered to be a bucket (a list), initialized to an empty list.
Step 1: Read off all the lists L_i, adding for each element j of each list L_i, a pair (i, j) to the end of list $A[j]$. (We could have added also just i to $A[j]$ instead of the pair (i, j).)
We now have an array of "buckets" with elements.
Step 2: Now, construct empty lists $L'_1, .., L'_n$. Read off the elements in the buckets of the array A sequentially. First read off the list $A[1]$, then $A[2]$, and so on. For each element (i, j) in each list $A[j]$, add element j to the end of list L'_i.

The algorithm is a variant of bucket sort, but we need to keep track of which list the elements belong to. The algorithm is like radix sort of all the pairs $(i, j), j \in L_i$, where Step 1 does the bucket sorting according to the second component j, and Step 2 does the bucket sorting according to the first component i.

Correctness: After Step 1, each bucket $A[j]$ contains one element (i, j) for every occurrence of j in list L_i. In Step 2, the elements are inserted to the lists L'_i in order because we process the lists $A[j]$ in increasing order of j and we append the elements to the end of the lists L'_i. Thus, the lists L'_i are the sorted equivalents of L_i.

Time complexity. The initialization takes time $O(n)$. Step 1 takes time $O(n + m)$ because there are n lists and the sum of their lengths is m. Step 2 takes also time $O(n + m)$ for the same reason. In total, the algorithm takes time $O(n + m)$.

Problem 3 (15 points, Graded by Xingyu and Zhenchen

S1: Take the first element of each list and put them in a min-heap. Along with the element we will also keep track of which list we took it from. (If a list is empty, there is nothing to do for that list.)
Repeatedly do the following:
S2: Take out the minimum element from the heap.
S3: Insert next element off the list where that element came from unless it is empty.
S4: Go back to S2. Continue the process until the heap is empty (every list is empty now).

The pseudo-code of the algorithm is given below. It takes as input an array $L[1 \dots k]$ of k sorted lists, where $L[i]$ points to the first node of the i-th list, and outputs the sorted list *Result* of all the elements. We assume that every node of each input list has a key field with the element and a next field that points to the next node of the list.

```
function MergeLists($L[1 \dots k]$)
    $Result$ = empty list
    $h = minHeap()$
    for $i = 1$ to $k$ do
        if $L[i] \neq NIL$ then
            $h.insert(L[i].key, i)$
            $L[i] = L[i].next$
        end
    end
    while $!h.isEmpty()$ do
        $(curMin, i) = h.ExtractMin()$
        Append $curMin$ to $Result$
        if $L[i] \neq NIL$ then
            $h.insert(L[i].key, i)$
            $L[i] = L[i].next$
        end
    end
    return $Result$
```

We prove the following loop invariant for the while loop in the algorithm: at every iteration i, the Result list contains the $i-1$ smallest elements in sorted order, the heap contains the smallest element of each list that has not been output in the Result list yet, and the root of the heap contains the ith smallest element out of all the k lists.

Initialization: Prior to the first iteration, we insert the first element of each of the k lists into a min-heap. Since the lists are sorted, these are the smallest elements of each list. The smallest element overall must be the first element of one of the lists. Thus, after insertion into the min-heap, the root must be the smallest overall element.

Maintenance: Assuming the invariant holds before iteration i, we want to show that it holds before iteration $i+1$. At iteration i, we extract the root of the heap, which we assumed is the ith overall smallest value, append it at the end of *Result*, and insert to the heap the next element from the list the root element came from. At this stage, the heap contains for each list either the smallest element that has not been inserted in the result list yet, or no elements if all of them have been inserted in the result list. Since the i smallest elements have already been inserted into the result list, the heap must contain the $(i+1)$th smallest element at its root.

Termination: At the end when the heap becomes empty, all the lists are empty, $i = n+1$, and by the invariant, all the n elements are in the Results list in sorted order. Thus, the algorithm outputs the correct sorted list.

For the runtime analysis, we observe that the heap never contains more than k elements. Indeed, we insert the k first elements of the lists before the while loop. Then, during each iteration, we extract an element and possibly insert another element, so the size of the heap never increases. Thus, insertion and extraction from the heap take $O(\log k)$ time in the worst case. Every element is inserted once in the heap and extracted once. Since there are n elements in total, these operations take time $O(n \log k)$ in total. In addition to the heap operations, there is constant more work in each iteration of the for loop and of the while loop, thus $O(k + n)$ additional time. Hence the total time is $O(k + n + n \log k) = O(k + n \log k)$. If the lists are nonempty, $k \leq n$ and the total time is $O(n \log k)$. If the lists may be empty, then it is possible that $k > n$, and the total time is $O(k + n \log k)$.

Remark: There is another algorithm with the same time complexity, that uses divide and conquer, like Merge Sort: Divide the k lists into two sets of approximately $k/2$ lists each (i.e., one set has $\lfloor k/2 \rfloor$ and the other $\lceil k/2 \rceil$). Recursively merge the lists in each set, and then merge the two resulting sorted lists, using the linear-time Merge algorithm. Since every recursive call divides the number of lists by 2, the recursion tree has height $\log k$. In each level of the recursion tree we merge disjoint lists of elements, thus the time spent at each level of the tree is $O(n)$. Since there are $\log k$ levels and k leaves, the total time is $O(k + n \log k)$.

Problem 4 (15 points, Graded by Shenzhi and Ziqing)

We present the following algorithm, where T is the tree node corresponding to the root of the subtree currently being considered (for the initial call, it would be the root of the tree):

```
1  function print-subtree($T, x, y$):
2      if $T == null$ then
3          return;
4      if $x \leq T.key \leq y$ then
5          print-subtree ($T.left, x, y$);
6          print ($T.key$);
7          print-subtree ($T.right, x, y$);
8      else if $T.key < x$ then
9          print-subtree ($T.right, x, y$);
10     else
11         print-subtree ($T.left, x, y$);
```

Correctness:
This algorithm essentially does an in-order traversal on the subtree that contains nodes with keys between x and y. The correctness thus stems from the correctness of in-order traversal. Looking at the 4 possible cases in more details we have that:

1. If the node is *null*, then return as there is nothing to do.
2. If the key of T is between x and y, then we recurse on the left subtree (as it may have other keys between x and y), print the current key and recurse on the right subtree. This is done in this order so that the printed keys are in sorted order.
3. If the key at the node is smaller than x, then only the right subtree of T may contain keys between x and y, so make a recursive call on the right subtree.
4. If the key at the node is larger than y, then only the left subtree of T may contain keys between x and y, so make a recursive call on the left subtree.

These 4 cases ensure that the in-order traversal is limited to the part of the tree that contains keys in $[x, y]$.

Runtime:
We show that there can only be at most h recursive calls from line 9 (case 3) and at most h from line 11 (case 4). To see this, suppose we get to case 3 for two different nodes in the tree m and n such that m is not an ancestor of n and n is not an ancestor of m. Let z be the lowest common ancestor of m and n, and assume without loss of generality that m is in z's left subtree and n is in z's right subtree. Since n is in case 3, we have $n.key < x$, and since n is in the right subtree of z, it follows that $z.key \leq n.key < x$, hence z is also in case 3. But this would mean that a recursive call was made only on z's right subtree, which does not contain m; hence the algorithm would not visit m leading to a contradiction. Thus, each call corresponding to case 3 must be made by nodes along the same path in the tree. The same reasoning applies for case 4. Thus, in both cases the work total runtime for the corresponding nodes is $O(h)$.
Finally for each node in case 2, two recursive calls are made and the current node key is printed. Since there are s such nodes, the total work done for that case is $O(s)$.
Therefore, the total runtime of the algorithm is $O(2h + s) = O(h + s)$.

Problem 5 (15 points, Graded by Malhar)

We can use a Red Black Tree to implement Best Fit heuristic. Each node in the tree represents a bin that has already been used, whose key is the size of remaining capacity. Each time there comes a new item, we will first search for the best fit bin in this tree, if there is one, we will put the item into this bin and reinsert the node with updated capacity into tree to maintain the order property; if there is not, we can create a new bin, put the item into this new bin, and insert this node into RB tree. The algorithm is illustrated as follows:

```
function SearchBestFitBin($(root, s)$)
    $x \leftarrow root$
    $node \leftarrow NULL$
    while $x \neq NULL$ do
        if $x.key < s$ then
            $x \leftarrow x.right$
        else
            $node \leftarrow x$
            $x \leftarrow x.left$
        end
    end
    return $node$
```

```
function BestFit($(S, B)$)
    $root \leftarrow$ new RBTree
    $k \leftarrow 0$
    for $i \leftarrow 1$ to $S.length$ do
        $bestBin \leftarrow$SearchBestFitBin($(root, S[i])$)
        if $bestBin == NULL$ then
            $k \leftarrow k + 1$
            $Bin[i] \leftarrow k$
            $newBin \leftarrow$ new TreeNode
            $newBin.key \leftarrow B - S[i]$
            $newBin.id \leftarrow k$
            $INSERT(root, newBin)$
        else
            $Bin[i] \leftarrow bestBin.id$
            $DELETE(root, bestBin)$
            $bestBin.key \leftarrow bestBin.key - S[i]$
            $INSERT(root, bestBin)$
        end
    end
```

Justification: The bins are numbered in the order they were created $1, \ldots, k$, where k is the number of bins created so far. We use an array *Bin* to record the bin of each item. Every RBTree node has two fiels, one is *key*, which equals the remaining capacity of this bin; the other is *id*, which is the index (the number) of this bin. Operations like DELETE and INSERT are just the same as the corresponding methods for a regular RBTree.

In the best fit algorithm, we need to find the best-fit bin for each item, which is defined as the fullest bin that still has enough room for the current item. Function $SearchBestFitBin(root, s)$ is to find the best fit bin from the RBTree for item s. To prove the correctness of this function, for a node x whose key is smaller than s, the space left is not enough for s. So we just iteratively evaluate $x.right$. If $x.key \geq s$, which means x might be a choice for best fit bin, but a bin with smaller space than x might also fit s, so we record x as a potential result, and iteratively move to $x.left$ to find a smaller fit for s. When a NULL is found, the function terminates and return the best-fit bin.

If the best-fit bin returned by $SearchBestFitBin(root, s)$ is not a NULL, we will add item s to this bin by setting accordingly the corresponding entry in the Bin array, and use $DELETE$ and $INSERT$ to maintain the order and color properties of RBTree. And if the best-fit bin is NULL, that means no bin in the RBTree is a fit for s, we will create a new bin, put the item in it, calculate its remaining space, and insert it into RBTree. The order and color properties are still kept by $INSERT$ method.

Time Complexity: In a balanced binary search tree like RBTree, $SearchBestFitBin$, $DELETE$ and $INSERT$ all run in at most $O(h) = O(\log k)$ time as k is the final number of nodes in the RBTree. Hence, for each item, we need $O(\log k)$ time to process, so the total time will be $O(n \log k)$.

Problem 6 (20 points, Graded by Ananth)

Data Structure:

The data structure can be a red-black tree (or other balanced search tree) with auxiliary information. Each node of the tree has following attributes:

- `(birthdate, salary)` (key of the tree): birthdate and salary the node corresponds to. The comparison of key is based on birthdate first. If the birthdates of both keys are equal, it will then compare based on salary.
- `emp`: the number of the employees with the (birthdate, salary) pair.
- `ttlemp`: the total number of employees in the subtree rooted at current node.
- `ttlsal`: the total salary of employees in the subtree rooted at current node.

Here, `ttlemp` and `ttlsal` are the additional information used in expediting the queries `Count(d)` and `AvgSal(d)` in the tree. These attributes of node x follow the two equations:

$$x.ttlemp = x.emp + x.left.ttlemp + x.right.ttlemp$$

$$x.ttlsal = x.salary \times x.emp + x.left.ttlsal + x.right.ttlsal$$

Algorithm:

- **Insertion and Deletion**

  The `Insertion` and `Deletion` of a `(bday, sal)` pair is similar to `RB-Insert` and `RB-Delete` in textbook (chapter 13 Red-Black Trees). There are two differences between the modified algorithm and the original one. First, we need to maintain the attributes, `ttlemp` and `ttlsal`, of all nodes during insertion and deletion. Second, when key `(bday, sal)` already exists, we only need to update the node's `emp` instead of adding or removing a node (we can delete the node physically if `emp` becomes 0).

  Correctness and runtime: The attributes of each node depends on only the information in the node and its left and right children. According to Theorem 14.1 (Augmenting a red-black tree) in textbook, we can maintain the attributes in all nodes during insertion and deletion within $O(\log n)$ time.

```
function MinBday():
    $x = T.root$;
    if $x == NIL$ then
        return $NIL$;
    while $x.left \neq NIL$ do
        $x = x.left$;
    end
    return $x.birthdate$;
```

- `MinBday`
  Correctness and runtime: The key of our search tree is `(birthdate, salary)` pair. Therefore the minimum element in the tree has the minimum birthdate. The time complexity is the height of the balanced tree, $O(\log n)$.

- **`Count(d)` and `AvgSal(d)`**

  To know the average salary of employees with birthdate $\leq d$, we need to know the number and the total salary of these employees.

```
function count-and-salary($d$):
    $x = T.root$;
    $count = 0$;
    $salarysum = 0$;
    while $x \neq NIL$ do
        if $x.birthdate > d$ then
            $x = x.left$;
        else
            $count = count + x.left.ttlemp + x.emp$;
            $salarysum = salarysum + x.left.ttlsal + x.salary \times x.emp$;
            $x = x.right$;
        end
    end
    return $count$, $salarysum$;

function Count($d$):
    $count, salarysum =$ count-and-salary($d$);
    return $count$;

function AvgSal($d$):
    $count, salarysum =$ count-and-salary($d$);
    return $salarysum/count$;
```

Correctness and runtime:

When node x's `birthdate` $> d$, all the employees with birthdate $\leq d$ are in x's left tree. When node x's `birthdate` $\leq d$, x and the subtree rooted at $x.left$ are all with birthdate $\leq d$, and they contributes $x.emp + x.left.ttlemp$ to `Count(d)`. However, there can also be employees with birthdate $\leq d$ in x's right tree. We need to add the number of these employees to `Cound(d)` unless x has no right child. Our algorithm calculates `Count(d)` starting from the tree's root. So `Count(d)` is the number of all employees with birthdate $\leq d$. The same logic applies to the calculation of total salary.

The algorithm terminates when reaches a leaf node. Its runtime is the height of the tree, $O(\log n)$.