

## Q1.

Since we use simple uniform hashing, the average number of elements that hash to the same array would be  $\alpha = \frac{n}{m}$ , where  $n$  is the total number of elements in the hash table, and  $m$  is the number of slots.

(a). Given that we use sorted arrays as the chaining components for the hash table:

On average, the time complexity for search would be  $O(1 + \log(\alpha))$ , Here,  $O(1)$  is the time taken to compute  $h(k)$ , and  $O(\log(\alpha))$  is the time to search for the key inside the array at slot  $h(k)$ . We achieve logarithmic runtime as we could perform binary search on the sorted array, since arrays allow random access. However, if  $\alpha$  is very small, we would not have much benefit from the usual way of using linked list as the chaining component (runtime being  $O(1 + \alpha)$ ). But in the case that  $n \gg m$ , then the runtime would improve.

(b). Given that we need to run a merge sort every time we insert an element:

On average, the time complexity for insertion would be  $O(1 + \alpha \log(\alpha))$ , where  $O(1)$  is the time to compute  $h(k)$ , and  $O(\alpha \log(\alpha))$  is the time to perform the merge sort on the array at slot  $h(k)$ . If we do not keep track of the current number of elements in the array, then we also have an additional cost of  $O(\alpha)$  to search to the end for an empty spot, otherwise it would just take  $O(1)$  time to append. Or if we consider the cost to expand the array size, then we need  $O(\alpha)$  to copy the old array into the new one. Yet in the end, the insertion time would be worse than the original scheme of using linked lists, which only costs  $O(1)$  time to prepend the key.

(c). Deletion with sorted arrays would remain the same as using linked lists, i.e.  $O(1 + \alpha)$ .  $O(1)$  is the time to compute  $h(k)$ , and  $O(\alpha)$  is the total cost for deletion. To delete, we first need to search for the key, so it takes  $O(\log(\alpha))$  to perform binary search, then remove the key by shifting all elements greater than the deleted key one slot to the left. In the worst case, the deleted key is the first element, so we need to shift all the remaining elements, which is  $O(\alpha)$ . Since  $O(\log(\alpha)) \in O(\alpha)$ , we have the overall cost for search-delete-shift as  $O(\alpha)$ . With linked lists, deletion is also  $O(\alpha)$ . We need to traverse the list to search for the previous element of the target, then change its pointer to the next element of the target ( $O(1)$ ), and free memory for the target ( $O(1)$ ). In the worst case, the target is the last element, so we require  $O(\alpha)$ .

## Q2.

His claim is not correct for all situations. There is no guarantee that the leaves of a complete binary tree are on the same level. There could be missing right-most nodes on the last level. As illustrated in Figure 1, all internal nodes of a complete binary tree have two children, but the leaves here are not on the same level, meaning that if we color all nodes as black, the number of black nodes from root to a leaf is not the same for all paths. This violates the red black tree property:

For each node, all paths from this node to each of its leaves contain the same number of black nodes.

Hence, it is not a red black tree in this case.

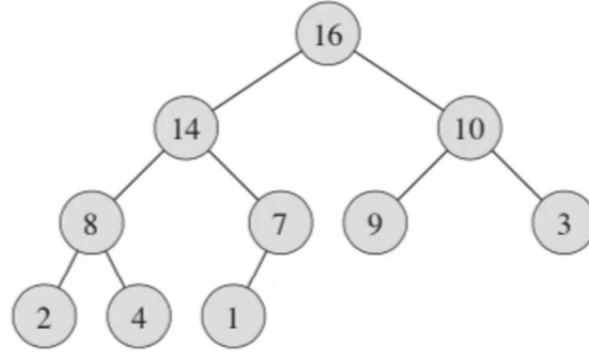


Figure 1: missing right-most nodes

However, if we have a perfect binary tree, which is a complete binary tree with all leaves on the same level (assuming leaves are nil), then it is valid red black tree:

1. the root is black
2. leaves are black nils
3. there is no consecutive red nodes along any path since there are no red nodes at all
4. all paths from a node to its leaves contain the same number of black nodes.

### Q3.

An avl tree is a self balancing binary search tree that has the property that, for any node  $x$  in the tree,  $x$  has at most two children, and  $x.left.key \leq x.key \leq x.right.key$ . Although for an avl tree, we might need rotation for insertion and deletion operations in order to maintain the balance factor, we still guarantee that the bst property holds. So the node with the minimum key is the left-most node in the tree. We could find it by traversing down the tree from the root, and always choosing the left subtree to traverse, until we find the left subtree to be nil, then the current node contains the minimum key. Similarly, the node with the maximum key is the right-most node in the tree. Beginning from the root, we always traverse the right subtree, until the right subtree is nil, then the current node contains the maximum key.

### Q4.

(a). Yes, we can solve it by dynamic programming.

Since the flat fee  $C$  remains constant for any sequence, we can just focus on the cost sum of flights in the sequence. In addition, since we need to find the cheapest sequence from any city- $i$  to city- $n$ , the problem can be generalized to an all pairs shortest paths problem. We have to find cheapest sequence for all pairs, despite that the problem only asks for  $(i, n)$ . This is because the subsequence of a cheapest sequence is also a cheapest sequence by itself. We need to construct a cheapest-sequence tree (similar to shortest-path tree) for the ease of restoring sequences at the end. This algorithm will only need to be run quarterly when the prices of direct flights change. After that, we can simply answer customers' queries by looking up the cheapest-sequence tree. The underlying algorithm is Floyd-Warshall algorithm.

**Sub-problems:** cheapest sequence to travel from city- $i$  to city- $j$  where  $i, j \in [1, n]$  using intermediate cities from the set  $\{1, \dots, k\}$ , and its length.

**Relation between sub-problems:**

To build the cheapest sequence from city- $i$  to city- $j$  when the intermediate cities are taken from the set  $\{1, \dots, k\}$ , we notice that the sequence can either go through city- $k$ , or not. If it does, then the subsequences from city- $i$  to city- $k$ , and from city- $k$  to city- $j$  would have intermediate cities taken from the set  $\{1, \dots, k-1\}$ . If it does not, then the sequence itself would have intermediate cities from  $\{1, \dots, k-1\}$ . Written as formulas,

- On minimum cost:

$$C_k(i, j) = \begin{cases} P(i, j) & k = 0 \\ \min(C_{k-1}(i, j), C_{k-1}(i, k) + C_{k-1}(k, j)) & k \geq 1 \end{cases}$$

where  $C_k(i, j)$  denotes the minimum cost for travelling from city- $i$  to city- $j$  when the intermediate cities are from the set  $\{1, \dots, k\}$

$P(i, j)$  denotes the price of direct flight from city- $i$  to city- $j$ . If there is no direct flight between two cities, the price would be  $\infty$ .

- On cheapest sequence:

For a pair of source and destination cities  $(i, j)$ , we store the next city of city- $i$  on the cheapest sequence.

$$N_k(i, j) = \begin{cases} j & \text{if } P(i, j) \neq \infty \wedge k = 0 \\ N_{k-1}(i, j) & \text{if } C_{k-1}(i, j) < C_{k-1}(i, k) + C_{k-1}(k, j) \wedge k \geq 1 \\ N_{k-1}(i, k) & \text{otherwise } \wedge k \geq 1 \end{cases}$$

where  $N_k(i, j)$  is the next city of city- $i$  on a cheapest sequence with intermediate cities taken from the set  $\{1, \dots, k\}$

**(b).** We use a bottom up approach to find the cheapest sequence and its cost for all pair of cities. To do so, we first represent the problem by constructing a graph  $G$ . Each city is a vertex, and we draw a directed edge  $(i, j)$  if there is a direct flight from city- $i$  to city- $j$ . The cost function is  $P$  that takes two cities as input and return the price of the flight.  $C[1..n][1..n]$  is the output table that stores the minimum costs.  $O[1..n][1..n]$  is the output table that stores the cheapest-sequence tree.

---

**Algorithm 1** All Pairs Cheapest Sequence  $(G, P)$ 

---

```
1: let  $C[1..n][1..n]$  be the minimum cost table
2: let  $O[1..n][1..n]$  be the cheapest sequence table
3: for  $i = 1$  to  $n$  do ▷ initialization
4:   for  $j = 1$  to  $n$  do
5:      $C[i][j] = \infty$ 
6:      $O[i][j] = \text{Nil}$ 
7: for each directed edge  $(i, j) \in G.E$  do
8:    $C[i][j] = P(i, j)$ 
9:    $O[i][j] = j$ 
10: for  $i = 1$  to  $n$  do
11:    $C[i][i] = 0$ 
12:    $O[i][i] = i$ 
13: for  $k = 1$  to  $n$  do
14:   for  $i = 1$  to  $n$  do
15:     for  $j = 1$  to  $n$  do
16:       if  $C[i][j] > C[i][k] + C[k][j]$  then
17:          $C[i][j] = C[i][k] + C[k][j]$ 
18:          $O[i][j] = O[i][k]$ 
19: Return  $C, O$ 
```

---

To answer customers' queries on the cheapest sequence from city- $i$  to city- $n$  and its cost, call Find Sequence  $(C, O, i, n)$

---

**Algorithm 2** Find Sequence  $(C, O, u, v)$ 

---

```
1: if  $O[u, v] = \text{Nil}$  then
2:   Return No Sequence
3: let  $l$  be the output list that contains the cheapest sequence
4:  $l = [u]$ 
5: while  $u \neq v$  do
6:    $u = O[u][v]$ 
7:    $l.append(u)$ 
8: Return  $l, C[u, v]$ 
```

---

(c). Time complexity is  $O(n^3)$ . The Floyd-Warshall algorithm has 3 **for** loops on line13-18. The outermost **for** loop iterates on  $k$  which gradually builds cheapest sequences based on the set of intermediate cities  $\{1..k\}$  until  $k = n$ . The two inner **for** loops iterates over all pairs of cities to build sequences for them. The initialization on line3-6 is  $O(n^2)$ . The time to initialize the costs and successor cities for  $k = 0$  on line7-12 is  $O(n^2)$ . This is because in the worst case, cities are fully connected, and the number of directed edges is  $n \times (n - 1) = O(n^2)$ . This algorithm is only required to run once per quarter when the price of direct flights changes. To reconstruct the sequence by looking up the table, we only need  $O(n)$  time since

there can at most be  $n$  cities in a cheapest sequence (no repeated cities, i.e. no cycles). Hence, at other times when prices do not change, we only need  $O(n)$  time to find the cheapest sequence. Otherwise, we need to both reconstruct the table and find the sequence, which is in total  $O(n^3)$  time.

## Q5.

(a). First, we simplify the problem into a longest decreasing subsequence (LDS) problem by sorting. We sort based on the first parameter, i.e. radius, in the descending order. This ensures that we could choose a longest decreasing sequence out of the original unsorted  $n$  boxes, by simply finding the LDS of the sorted list. Note that here we cannot solve the LDS based on height purely, since in the case where radius are the same and height are in the descending order, we might mistakenly contribute boxes of the same radius to the LDS. We could avoid this by also sorting the height in the ascending order for the boxes of the same radius, and then find the LDS based on height only. Now we can find the LDS of the sorted sequence by dynamic programming.

### Sub-problems:

We use a modified sub-problem (referenced from [?]) to solve the LDS in  $O(n \log n)$  time. The sub-problems are decreasing subsequence of varying lengths starting from 1. We build longer subsequence based on the end element of the current subsequences. For subsequences of the same lengths, we store the end element that is known to be the largest at the current point of time. This ensures that we could build the subsequence the longest possible. The LDS is the longest subsequence found at the end.

### Relation between sub-problems:

Suppose  $dp[1..n]$  is an array that we use to store the end elements of LDS, where  $dp[i]$  stores the end element of LDS of length  $i$ . Now if  $P_j = \langle (r_1, h_1), (r_2, h_2), \dots, (r_j, h_j) \rangle$  is a prefix of the sorted input list, with which we have used to build subsequence of different lengths so far, and if  $dp[1..k]$  where  $k < n$  is the prefix of the array that has been filled (meaning that we have found subsequence of length from 1 to  $k$ ). Then for the next element  $(r_{j+1}, h_{j+1})$ , we want to build subsequences in the following ways:

- Update the end element of subsequences of length  $m$  where  $m \leq k$  to be the largest so far:

$$dp[m] = (r_{j+1}, h_{j+1}) \quad \text{if } h_{j+1} > dp[m].h \\ \wedge \forall l \in [1, m-1]. dp[l].h > h_{j+1}$$

- Build a longer subsequence if it can be appended to the current longest subsequence

$$dp[k+1] = (r_{j+1}, h_{j+1}) \quad \text{if } \forall l \in [1, k]. dp[l].h > h_{j+1}$$

(b) Let  $P$  denote the input list  $[(r_1, h_1), (r_2, h_2), \dots, (r_n, h_n)]$ .  $P_{sorted}$  denotes the input list after sorting. In the following pseudocode, we access the  $i^{th}$  element in  $P_{sorted}$  by  $P_{sorted}[i]$ , and access  $r_i$  by  $P_{sorted}[i].r$  and  $h_i$  by  $P_{sorted}[i].h$ . Also,  $prev[1..n]$  is an array where  $prev[i]$  is the index of the predecessor of  $P_{sorted}[i]$  in a decreasing subsequence.  $dp[1..n]$  is an array where  $dp[i]$  is the current largest end element of a subsequence of length  $i$ .  $idx[1..n]$  is an array where  $idx[i]$  is the index of the element  $dp[i]$  in  $P_{sorted}$ .

---

**Algorithm 3** Bottom-up Longest Strictly Decreasing Sequence( $P$ )

---

```
1: let  $dp[1..n]$  be an empty array
2: let  $prev[1..n]$  be an empty array
3: let  $idx[1..n]$  be an empty array
4: MergeSort ( $P, 1, n$ )
5: for  $i = 1$  to  $n$  do
6:    $prev[i] = \text{Nil}$ 
7:  $dp[1] = P_{sorted}[1]$  ▷ initialize the  $dp$  array
8:  $idx[1] = 1$ 
9:  $length = 1$  ▷ keep track of the length of the current LDS
10: for  $i = 1$  to  $n$  do
11:   if  $dp[length].h > P_{sorted}[i].h$  then ▷ append to the current LDS
12:      $dp[length + 1] = P_{sorted}[i]$ 
13:      $idx[length + 1] = i$ 
14:      $prev[i] = idx[length]$ 
15:      $length = length + 1$ 
16:   else
17:      $k = \text{BinarySearch}$  ( $dp, P_{sorted}[i], 1, length$ )
18:     ▷ return the index of largest element  $\leq P_{sorted}[i]$  in  $dp[1..length]$ 
19:     if  $dp[k].h < P_{sorted}[i].h$  then ▷ no need to update when elements are the same
20:        $dp[k] = P_{sorted}[i]$  ▷ update the largest end element of subsequence of length  $k$ 
21:        $idx[k] = i$ 
22:       if  $k > 1$  then ▷ subsequence of length 1 has no predecessor
23:          $prev[i] = idx[k - 1]$ 
24: let  $O[1..length]$  be the output sequence
25:  $i = idx[length]$  ▷ index of the last element of the LDS
26:  $l = length$ 
27: while  $prev[i] \neq \text{Nil}$  do ▷ restore the LDS backwards
28:    $O[l] = P_{sorted}[i]$ 
29:    $i = prev[i]$ 
30:    $l = l - 1$ 
31:  $O[l] = P_{sorted}[i]$ 
32: return  $O$ 
```

---

The Merge function for the MergeSort is modified as follows

---

**Algorithm 4** Merge in descending order ( $P, p, q, r$ )

---

```
1:  $n_1 = q - p + 1$ ;  $n_2 = r - q$ 
2: let  $L = [1..n_1 + 1]$  and  $R = [1..n_2 + 1]$ 
3: for  $i = 1$  to  $n_1$  do
4:    $L[i] = P[p + i - 1]$  ▷ copy of lower subarray
5: for  $i = 1$  to  $n_2$  do
6:    $R[i] = P[q + i]$  ▷ copy of upper subarray
7:  $L[n_1 + 1] = \infty$ ;  $R[n_2 + 1] = \infty$ 
8:  $i = 1$ ;  $j = 1$ 
9: for  $k = p$  to  $r$  do
10:  if  $L[i].r > R[j].r$  then ▷ descending order based on  $r$ 
11:     $P[k] = L[i]$ 
12:     $i = i + 1$ 
13:  else if  $L[i].r = R[j].r$  then
14:    if  $L[i].h \leq R[j].h$  then ▷ then ascending order based on  $h$ 
15:       $P[k] = L[i]$ 
16:       $i = i + 1$ 
17:    else
18:       $P[k] = R[j]$ 
19:       $j = j + 1$ 
20:  else
21:     $P[k] = R[j]$ 
22:     $j = j + 1$ 
```

---

BinarySearch on a sorted descending array  $A$  that returns the index of largest element smaller than or equal to the target  $k$ , assuming that  $k$  is within the range of  $A$ , is as follows:

---

**Algorithm 5** BinarySearch ( $A, k, p, q$ )

---

```
1: while  $p \leq q$  do
2:    $mid = \lfloor \frac{p+q}{2} \rfloor$ 
3:   if  $A[mid].h = k.h$  then ▷ comparison based on height
4:     Return  $mid$ 
5:   else if  $A[mid].h > k.h$  then
6:      $p = mid + 1$ 
7:   else
8:     if  $mid = p$  then
9:       Return  $mid$ 
10:     $q = mid$ 
11: Return Nil
```

---

(c). Time complexity is  $O(n \log n)$ . Initialization of arrays takes  $O(n)$  time. Performing mergesort on the input list takes  $O(n \log n)$  time. Then line 10-23 builds the subsequences by iterating through the sorted input list. For each iteration, we either append the element to the current LDS, or perform a binary search to find the position in the  $dp$  array to update the end element. Binary search takes  $O(\log n)$  since subsequences will at most be length  $n$ , and in fact it is always smaller than  $n$  if we ever need to update the end element of established subsequences. We can perform binary search at most  $n$  times, so the cost for building LDS is  $O(n \log n)$ . In fact if we update end element  $n$  times, we can never build longer subsequences than length 1. So the cost will definitely be  $O(n \log n)$ . Then to restore the LDS from backwards, we iterate at most  $n$  times because its length is at most  $n$ . The overall time complexity would be  $O(n) + O(n \log n) + O(n \log n) + O(n) = O(n \log n)$ .

## Q6.

(a). The following algorithm relies heavily on the assumption that all  $m_{ij}$  are integer numbers, and assuming this is allowed in the problem specification. Given that  $k_i$  is the number of proposals for the  $i^{th}$  approach, and  $m_{ij}$  is the funding requested for the  $j^{th}$  proposal for the  $i^{th}$  approach, where  $j \in [1, k_i]$ . We would use a  $dp$  table to compute the choices of proposals. In the following psuedocode, entry  $dp[i][j]$  is a pair of values  $\langle x, y \rangle$  where  $x$  is a funding sum (potentially many possibilities) requested by the previous  $(i - 1)$  approaches, given that  $j$  is the exact funding requested by all  $i$  approaches; and the second value  $y$  is the corresponding index of the proposal for the  $i^{th}$  approach such that  $x + m_{iy} = j$ . If there is no set of proposals for the first  $i$  approaches that sum to  $j$  exactly,  $dp[i][j]$  would be  $\langle 0, 0 \rangle$ . The reason for keeping the fund sum of the previous  $(i - 1)$  proposals is to trace the selected proposals backwards at the end.



---

**Algorithm 6** Set of proposals ( $n, F, m$ )

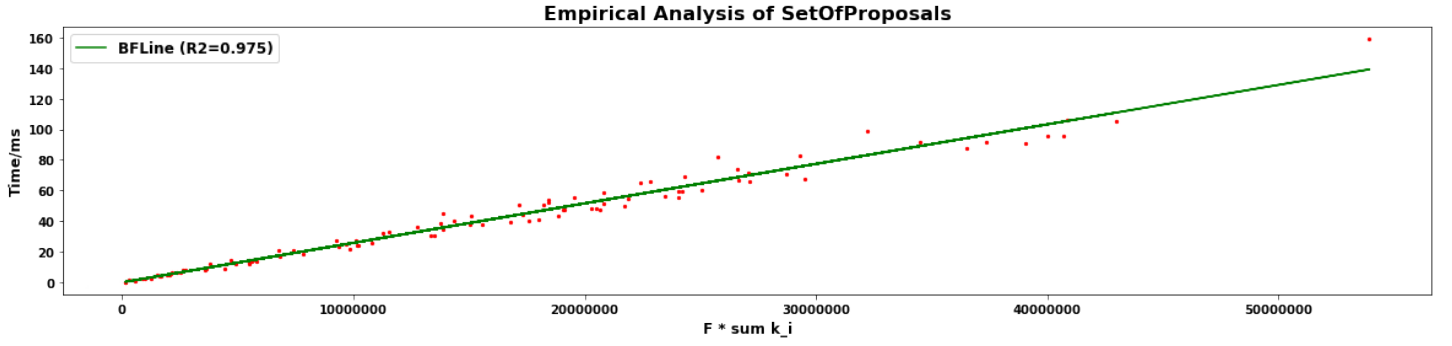
---

```
1: let  $dp[1..n][1..F]$  be an empty table ▷ assume all entries are initialized to  $\langle 0, 0 \rangle$ 
2: for  $j = 1$  to  $k_1$  do
3:    $dp[1][m_{1j}] = \langle 0, j \rangle$  ▷ initialize the pair for the first approach
4: for  $i = 1$  to  $n - 1$  do
5:   for  $j = 1$  to  $F$  do
6:     if  $dp[i][j] \neq \langle 0, 0 \rangle$  then
7:       for  $y = 1$  to  $k_{i+1}$  do
8:         if  $j + m_{(i+1)y} < F$  then
9:            $dp[i + 1][j + m_{(i+1)y}] = \langle j, y \rangle$ 
10: let  $seqProposals[1..n]$  be the indices of the selected proposals
11: let  $reqFund[1..n]$  be the fund requested by the selected proposals
12: let  $totalFund$  be the total fund distributed
13: for  $j = F$  to  $1$  do
14:   if  $dp[n][j] \neq \langle 0, 0 \rangle$  then ▷ check the last row for the maximum fund distributed
15:      $totalFund = j$ 
16:      $sum = j$ 
17:     for  $i = n$  to  $1$  do ▷ back trace the sequence
18:        $seqProposals[i] = dp[i][sum].second$ 
19:        $reqFund[i] = m_{i(seqProposals[i])}$ 
20:        $sum = dp[i][sum].first$ 
21:     break
22: return  $totalFund, seqProposals, reqFund$ 
```

---

(b). Time complexity is  $O(F \times \sum_{i=1}^n k_i)$ . In line 4-9, there are 3 layers of **for** loops. The outermost **for** loop at line 4 and the innermost loop at line 7 in total would have  $\sum_{i=1}^{n-1} k_{i+1} = \sum_{i=2}^n k_i = O(\sum_{i=1}^n k_i)$  iterations. This is then multiplied by  $F$  because of the **for** loop at line 5. The loop body in line 8-9 is constant time operations, so the loops have cost  $O(F \times \sum_{i=1}^n k_i)$ . The initialization in line 2-3 costs another  $O(k_1)$  operations, and to output the final set of proposals in line 13-21, we have to traverse the last row of the dp table, which would at most iterate  $F$  times if there is no solution. If there is a solution, then we have another  $O(n)$  operations to trace backwards the sequence of proposals. Hence, in sum there should be  $O(F \times \sum_{i=1}^n k_i) + O(k_1) + O(F) + O(n)$ , which is dominated by  $O(F \times \sum_{i=1}^n k_i)$ . Notice that if we also count the initialization cost at line 1, which is  $O(n \times F)$ , in the end we still get  $O(F \times \sum_{i=1}^n k_i)$  as the time complexity since  $n \leq \sum_{i=1}^n k_i$ .

(d). Empirical analysis:



This plotted graph is based on a data set of 100 samples. We observe a linear relationship between the time taken in  $ms$ , and the number  $(F \times \sum_{i=1}^n k_i)$ . The best fit line has a  $R^2$  coefficient of 0.975, which means that the linear model fits the data set well. Hence, we could conclude that the time complexity of the algorithm is some constant times  $F \times \sum_{i=1}^n k_i$ , and hence there is  $c, n_0 > 0$  such that  $T(n) \leq c(F \times \sum_{i=1}^n k_i)$  for all  $n \geq n_0$ . Time complexiy is therefore  $O(F \times \sum_{i=1}^n k_i)$  and the empirical analysis matches with our theoretical analysis.