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 α 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. Yet in the end, the insertion time would be worse than the original scheme of using linked lists, with which it 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 ($O(1)$), 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.

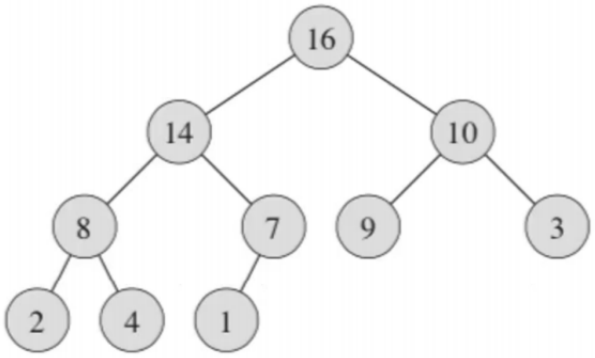


Figure 1: missing right-most nodes

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. The problem is therefore reduced to the normal SSSP problem.
Sub-problems: cheapest sequence to travel from city-i to city-x using at most k hops where $k \in [1, n-1]$ and $x \in [1, n]$, $x \neq i$. We have to search the cheapest sequence for all cities despite that the problem only asks for the case when destination is city-n. This is because the subsequence of a cheapest sequence to city-n is also a cheapest sequence on its own.

Relation between sub-problems:
Let E denote the set of all direct flights between cities and $(u, v) \in E$ if there is a direct flight from city-u to city-v. Let $d_k(i, u)$ denote the lowest cost from city-i to city-u using at most k hops. $P(u, v)$ is the cost of the direct flight from u to v.

$$d_k(i, v) = \min_{(u,v)\in E}(d_{k-1}(i, u) + P(u, v))$$

(b). We use a bottom up approach to find the cheaptest sequence and its cost. To do so, we first represent the problem by constructing a graph G. Each city is a vertex, and we draw a directed edge (u, v) if there is a direct flight from city-u to city-v. The cost function is P that takes two cities as input and return the price of the flight. The source city is s, and the destination city in the sequence we look for is d. We maintain the cheapest cost to travel from source s to vertex v by $v.c$ and the precedessor of vertex v by $v.\pi$. The underlying algorithm is the Bellman-Ford algorithm. To find the solution to the original problem, we would call Cheapest Sequence (G, P, i, n).

Algorithm 1 Cheapest Sequence (G, P, s, d)

```
1:  for each vertex v ∈ G.V do
2:      v.c = ∞
3:      v.π = Nil
4:  s.c = 0                                              ▷ initialize source
5:  for i = 1 to |G.V| − 1 do
6:      for each edge (u, v) ∈ G.E do
7:          if v.c > u.c + P(u, v) then
8:              v.c = u.c + P(u, v)
9:              v.π = u
10: Find the vertex x that represents city-d
11: v = x                                  ▷ we are going to trace the sequence backwards
12: Declare an empty linked list L                    ▷ used to output the cheapest sequence
13: while v.π ≠ Nil do                    ▷ loop terminates when v = s, since s.π = Nil
14:     prepend (L, v)
15:     v = v.π
16: prepend (L, s)
17: Return L
```

(c). Time complexity is $O(|VE|)$. Line 1-3 initializes all vertices, and this takes $O(|V|)$ time. The outer **for** loop of line 5-9 iterates $|V| - 1$ times. The nested **for** loop of line 6-9 iterates $|E|$ times per pass. So the total complexity of the loops is $O(|VE|)$. To retrieve the final output, the **while** loop iterates at most $|V|$ times. Hence the overall complexity is $O(|V|) + O(|VE|) + O(|V|) = O(|VE|)$. Since there could be direct flights between any two cities, $|E| = O(|V|^2)$. Also $|V| = n$, time complexity is therefore $O(n^3)$.

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 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 $P_j = \langle (r_1, h_1), (r_2, h_2), \ldots, (r_j, h_j) \rangle$ 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}) that we want to build LDS with:

- Update the end element of LDS 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 LDS 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 $[(r_1, h_1), (r_2, h_2), \ldots, (r_n, h_n)]$. P_{sorted} denotes the input list after sorting. In the following psuedocode, 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 end element of a subsequence of length i. $idx[1..n]$ is an array where $idx[i]$ is the index of the element $dp[i]$.

Algorithm 2 Bottom-up Longest Strictly Decreasing Sequence(P)

let $dp[1..n]$ be an empty array
let $prev[1..n]$ be an empty array
let $idx[1..n]$ be an empty array
MergeSort $(P, 1, n)$
for $i = 1$ to n **do**
    $prev[i] = \text{Nil}$
$dp[1] = P_{sorted}[1]$ ▷ initialize the dp array
$idx[1] = 1$
$length = 1$ ▷ keep track of the length of the current LDS
for $i = 1$ to n **do**
    if $dp[length].h > P_{sorted}[i].h$ **then** ▷ append to the current LDS
        $dp[length + 1] = P_{sorted}[i]$
        $idx[length + 1] = i$
        $prev[i] = idx[length]$
        $length = length + 1$
    else
        $k =$ **BinarySearch** $(dp, P_{sorted}[i], 1, length)$
        ▷ return the index of largest element $\leq P_{sorted}[i].h$ in $dp[1..length]$
        if $dp[k].h < P_{sorted}[i].h$ **then**
            $dp[k] = P_{sorted}[i]$ ▷ update the largest end element
            $idx[k] = i$
            if $k > 1$ **then** ▷ first element in a subsequence has no predecessor
                $prev[i] = idx[k - 1]$
let $O[1..length]$ be the output sequence
$i = idx[length]$ ▷ index of the last element of the LDS
$l = length$
while $prev[i] \neq \text{Nil}$ **do** ▷ restore the LDS backwards
    $O[l] = P_{sorted}[i]$
    $i = prev[i]$
    $l = l - 1$
$O[l] = P_{sorted}[i]$
return O

The Merge function for the MergeSort is modified as follows

Algorithm 3 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] = ∞; R[n_2 + 1] = ∞
8: i = 1; j = 1
9: for k = p to r do
10:    if L[i].r > R[i].r then                      ▷ descending order based on r
11:        P[k] = L[i]
12:        i = i + 1
13:    else if L[i].r = R[i].r then
14:        if L[i].h ≤ R[i].h then                  ▷ then ascending order based on h
15:            P[k] = L[i]
16:            i = i + 1
17:        else
18:            P[k] = R[i]
19:            j = j + 1
20:    else
21:        P[k] = R[i]
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 4 BinarySearch (A, k, p, q)

```
1: while p ≤ q do
2:     mid = ⌊(p+q)/2⌋
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 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.

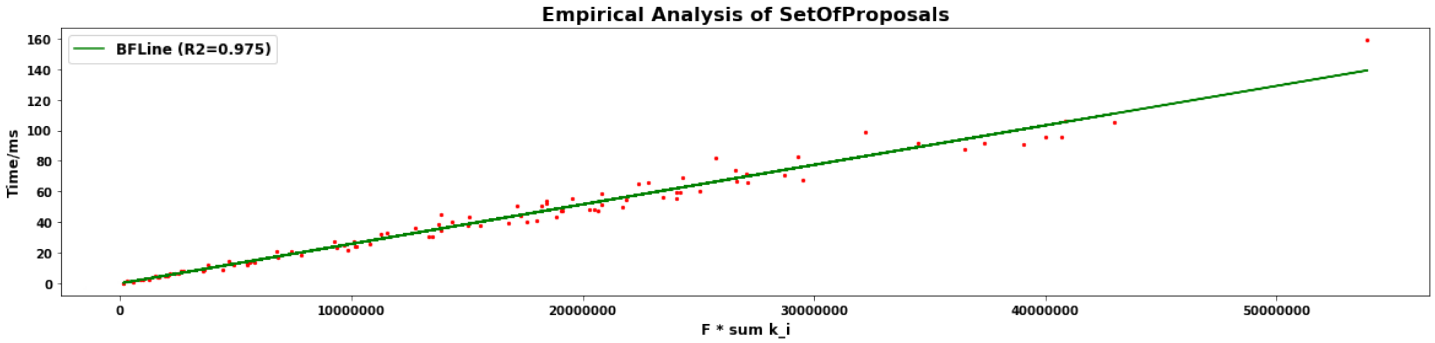
(a). Given that k_i is the number of proposals for the i^{th} approach, and m_{ij} is the funding requested for the j^{th} proposal for the i^{th} approach, where $j \in [1, k_i]$. The following algorithm relies heavily on the assumption that all m_{ij} are integer numbers, and assuming this is allowed in the problem specification. 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 5 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: $\quad dp[1][m_{1j}] = \langle 0, j\rangle$ ▷ initialize the pair for the first approach
4: **for** $i = 1$ to $n - 1$ **do**
5: $\quad$ **for** $j = 1$ to F **do**
6: $\quad\quad$ **if** $dp[i][j] \neq \langle 0, 0\rangle$ **then**
7: $\quad\quad\quad$ **for** $y = 1$ to k_{i+1} **do**
8: $\quad\quad\quad\quad$ **if** $j + m_{(i+1)y} < F$ **then**
9: $\quad\quad\quad\quad\quad dp[i + 1][j + m_{(i+1)y}] = \langle j, y\rangle$
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: $\quad$ **if** $dp[n][j] \neq \langle 0, 0\rangle$ **then** ▷ check the last row for the maximum fund distributed
15: $\quad\quad totalFund = j$
16: $\quad\quad sum = j$
17: $\quad\quad$ **for** $i = n$ to 1 **do** ▷ back trace the sequence
18: $\quad\quad\quad seqProposals[i] = dp[i][sum].second$
19: $\quad\quad\quad reqFund[i] = m_{i(seqProposals[i])}$
20: $\quad\quad\quad sum = dp[i][sum].first$
21: $\quad\quad$ **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.