BBM408 Second Assignment

Umut Güngör
21946198

22 May 2023

Question 1

Consider a hotel that places visitors to rooms after hashing (non-uniformly) their names and assigning them to the room with their hash result. For example, if your name is hashed to 7, then you have to stay in room 7. However, if the room is busy, you are placed on a waiting list for that room. Each visitor is expected to spend 1 to 5 days in the room, with a uniform distribution. Assuming there are 10 rooms in the hotel, and an average of 4 visitors arrive every day, what is the expected waiting time for the first visitor arriving on day t + 1, if the hash function produces the following hashes with the given proportions?

Hash:	1 2 3 4 5 6 7 8 9 10
Proportion:	4 3 3 2 2 2 1 1 1 1

Solution:

In order to calculate the expected waiting time for the first visitor arriving on day t + 1, we have to calculate the expected waiting time for each room separately.

The sum of all proportions is: 20
The ratio for room 1 is: 4/20
The expected length of stay in room 1 is an average of 1 to 5 days: 3
Therefore, the expected waiting time for **room 1** is: (4/20) * 3 = 0.6
Similarly for the other rooms:
For **room 2** is: (3/20) * 3 = 0.45
For **room 3** is: (3/20) * 3 = 0.45
For **room 4** is: (2/20) * 3 = 0.3
For **room 5** is: (2/20) * 3 = 0.3
For **room 6** is: (2/20) * 3 = 0.3
For **room 7** is: (1/20) * 3 = 0.15
For **room 8** is: (1/20) * 3 = 0.15
For **room 9** is: (1/20) * 3 = 0.15
For **room 10** is: (1/20) * 3 = 0.15

From these calculations, we can calculate the expected waiting time for the visitor. We multiply each room's ratio with its proportion, then sum up all these products. Lastly, divide this sum by the total proportion.

If we say r_i to ratio for i^{th} room and, h_i to proportion for i^{th} room, then

$\sum_{i=1}^{10} r_i * h_i$ will be our summation formula.

The total expecting waiting time is: $\frac{\sum_{i=1}^{10} r_i * h_i}{20}$

Which equals to $\frac{7.5}{20}$.

Hence, the expected waiting time for the first visitor arriving on day t + 1 is **0.375** days which corresponds to **9** hours.

Question 2

a.) Consider a stack of fixed capacity k, where we copy the stack contents to another stack for backup purposes every k operations. For example, if k is 5, after 4 push and 1 pop operations, we copy all stack contents to another stack (assume the stack is implemented using an array and you are just copying the array contents, assuming a cost of 1 per item, whole array is always copied). If the stack is full, the next push operation is ignored, and if the stack is empty, the next pop operation is ignored. In both cases, the cost is assumed to be 0. Show that the amortized cost of n stack operations is O(n) using accounting method.

b.) Now consider a dynamic stack (doubles in size when full) and solve the problem, again. You now make a backup immediately before each expansion.

Solution:

a.) In order to solve this problem, we use accounting method with amortized cost.

$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$

Where c_i is the actual cost for an operation and $\hat{c}_i$ is the amortized cost for an operation. So, if the amortized cost for n operations is O(n), then the actual cost of the n operations will also be O(n).

We have two operations: push and pop.

For push operation:

– The actual cost of a push operation is 1 unless there is a move operation. If there is a move operation, the cost will be k.

– We assign the amortized cost of a push operation as 2. Because, when we apply every k^{th} operation, we have to k move operations. 1 for push operation and 1 for move operation.

For pop operation:

– The actual cost of a pop operation is 1 unless there is a move operation. If there is a move operation, the cost will be k.

– We assign the amortized cost of a pop operation as 2. Because, when we apply every k^{th} operation, we have to apply k move operations even if the stack is not full. 1 for pop and 1 for move operation.

We assign the amortized cost for an operation which may be push or pop as 2. Since we apply n stack operations, the cost of these n operations would be 2n.

Thus, the amortized cost of n stack operations is $\boldsymbol{O(n)}$.

b.)We apply the same logic to this part.

For push operation:

– The actual cost of a push operation is 1 unless there is an expansion. If there is an expansion, the cost will be the size of the dynamic stack.

– We assign the amortized cost of a push operation as 3. Because when there is an expansion, we have to move the elements and double the size of the stack. 1 for push operation, 1 for move operation and 1 for doubling the stack.

For pop operation:

– The actual cost of a pop operation is 1. There cannot be any expansion after pop operation.

– We assign the amortized cost of a pop operation as 3. This value would be an overestimate for pop operation. However, this will help our future calculations.

We assign the amortized cost for an operation which may be push or pop as 3. Since we apply n stack operations, the cost of these n operations would be 3n. Since the actual costs of these operations are less than the amortized costs, the actual cost of these n operations would be less than 3n.

Hence, the amortized cost of n stack operations is $\boldsymbol{O(n)}$.

Question 3

Given n distinct integers in a list (assume odd n), you pick 3 of them randomly and return the median of these 3 as the median of the list. What is the probability that this approach will result in a median within 1 position tolerance of the actual median?

Solution:

We have a list of n integers. If we sort this list, the index of the median will be $\frac{n+1}{2}$. We pick 3 elements from the list and return their median. This returned median should equal to the value at the index of $\frac{n+1}{2}$, $\frac{n+1}{2}-1$ or $\frac{n+1}{2}+1$ in the ordered list. So, if we calculate this probability and divide it by C(n,3), we will solve the problem. We divide it by C(n,3) because C(n,3) is the probability of picking 3 numbers.

The probability of the returned median equals to the value at the index of $\frac{n+1}{2}-1$ which is equal to $\frac{n-1}{2}$ in the ordered list would be:
We pick 3 numbers and return their median value. So, if we want this returned median value equals to the value at the index of $\frac{n-1}{2}$ in the ordered list, we pick the value at the index of $\frac{n-1}{2}$, one value from less than this value and one value from greater than this value. Because the median of these 3 chosen values is what we are looking for. The probability of these 3 values is:
$\frac{n-3}{2} * \frac{n+1}{2}$.
Because we pick the value at the index of $\frac{n-1}{2}$ in the ordered list. There will be $\frac{n-3}{2}$ numbers that are less than this value and $\frac{n+1}{2}$ numbers that are greater than this value. The probability is the multiplication of these values.

We can apply the same logic for the other two indices.

The probability of the returned median equals to the value at the index of $\frac{n+1}{2}$ in the ordered list would be:

$\frac{n-1}{2} * \frac{n-1}{2}$.

Similarly, the probability of the returned median equals to the value at the index of $\frac{n+1}{2}+1$ which is equal to $\frac{n+3}{2}$ in the ordered list would be:

$\frac{n+1}{2} * \frac{n-3}{2}$.

In the end, we sum up all these three probabilities and divide them by C(n,3).

$$\frac{\frac{n^2-2n-3}{4}+\frac{n^2-2n+1}{4}+\frac{n^2-2n-3}{4}}{\frac{n*(n-1)(n-2)}{6}}$$

After doing some algebra, this equals to:

$\frac{9n^2-18n-15}{2n^3-6n^2+4n}$

Hence, the probability is: $\boldsymbol{\frac{9n^2-18n-15}{2n^3-6n^2+4n}}$.

Question 4

Assume that you are given a list of n numbers such that, if sequentially divided into groups of k numbers (assume n is a multiple of k), the numbers in group i are guaranteed to be greater than the numbers in groups 1...i-2 For example, if the list is grouped as:
a - b - c - d - e - f - g
where each of a, b, etc. is a group k numbers, then all the numbers in a, b and c are less than all the numbers in e.
Design an algorithm to show that this list can be fully sorted in linear time with respect to n, for small k.

Solution:

To sort the list, we will use an algorithm that is similar to merge sort. We start with the 1^{st} and 2^{nd} groups. We sort these groups within themselves. After that, we merge them. Next, we sort the 4^{th} group within itself. We can just merge the 4^{th} group to the end of the merged first and second groups without any comparison. Because all numbers in 4^{th} group are greater than all numbers in 1^{st} and 2^{nd} groups. After, this merge we have to merge the 3^{rd} group to the last merged one. Firstly, we sort the 3^{rd} group itself. Then, we merge it with the last merged one. For 6^{th} group we merge easily like 4^{th} one. Because all numbers in the 6^{th} group are greater than all the numbers in the 1^{st}, 2^{nd}, 3^{rd} and 4^{th} groups. Merging the 5^{th} is similar to merging the 3^{rd} one. This process goes like that. The complexity of this algorithm would be: We have n/k groups and we sort each of them. The complexity of these sorts is: $O(n/k * klgk)$ which equals to $O(nlgk)$. We merge n items, so the complexity of the merges is: $O(n)$. The total complexity of this algorithm is: $O(nlgk) + O(n)$. When k is small, we can neglect it. Thus, this algorithm sorts the list in $\boldsymbol{O(n)}$ time.