ShanghaiTech University

CS240 Algorithm Design and Analysis

Fall 2021

Problem Set 4

Due: 23:59, Dec.17, 2021

1. Submit your solutions to Gradescope (www.gradescope.com).
2. In "Account Settings" of Gradescope, set your FULL NAME to your Chinese name and enter your STUDENT ID correctly.
3. If you want to submit a handwritten version, scan it clearly. Camscanner is recommended.
4. When submitting your homework, match each of your solution to the corresponding problem number.

Problem 1:

To implement a TABLE-DELETE operation, it is simple enough to remove the specified item from the table. It is often desirable, however, to contract the table when the load factor(number of elements in a table/table size) of the table becomes too small, so that the wasted space is not exorbitant. Table contraction is analogous to table expansion: when the number of items in the table drops too low, we allocate a new, smaller table and then copy the items from the old table into the new one. Similar to table doubling, we halving its size when its load factor drops below 1/4 in TABLE-DELETE, but in this problem,we contract a table by multiplying its size by 2/3 when its load factor drops below 1/3. Using the potential function

$$\Phi(\mathrm{T}) = \mid 2 * T.num - T.\ \mathrm{size}\ \mid$$

show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded above by a constant.

Solution:

Consider first the case when the load factor does not drop below 1/3,

$$\begin{aligned}
\alpha_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + |2num_i - \mathrm{size}_i| - |2num_{i-1} - size_{i-1}| \\
&= 1 + |2num_i - \mathrm{size}_i| - \mid 2(\ \mathrm{num}\ _i + 1) - size_i \mid \\
&\leq 1 + \mid 2num_i - size_i - (2(\ \mathrm{num}\ _i + 1) - size_i) \mid \quad \text{// (reverse) triangle inequality} \\
&= 1 + \mid -2| \\
&= 3
\end{aligned}$$

Now consider the case when the load factor drops below 1/3, i.e.

$$num_i + 1 = num_{i-1}$$
$$size_i = \frac{2}{3}\,\mathrm{size}_{i-1}$$
$$\mathrm{size}_{i-1} = 3num_{i-1}$$

Therefore,

$$\begin{aligned}
\alpha_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (\text{ num }_i + 1) + |\ 2 \text{ num }_i - \text{size}_i\ | - |2 \text{ num }_{i-1} - size_{i-1}\ | \\
&= (\text{ num }_i + 1) + (-2 \text{ num }_i + size_i) - (-2 \text{ num }_{i-1} + \text{size}_{i-1}) \\
&= (\text{ num }_i + 1) + (-2 \text{ num }_i + 2(\text{ num }_i + 1)) - (-2(\text{ num }_i + 1) + 3(\text{ num }_i + 1)) \\
&= (\text{ num }_i + 1) + 2 - (\text{ num }_i + 1) \\
&= 2
\end{aligned}$$

Problem 2:

Assume you are creating an array data structure that has a fixed size of n. You want to backup this array after every so many insertion operations. Unfortunately, the backup operation is quite expensive, it takes n time to do the backup. Insertions without a backup just take 1 time unit. How frequently can you do a backup and still guarantee that the amortized cost of insertion is O(1) ? Prove that you can do backups in O(1) amortized time. Use the potential method for your proof.

Solution:

1.You can backup the array after every n insertions

2. Let $\phi_i = i \bmod n$. Then when i mod n = 0, ai = n + 0 - (n - 1) = 1. When i mod n != 0, $a_i = 1 + (i \bmod n) - ((i-1)) \bmod n = 2$

Problem 3:

There is a way to implement a queue Q with two stacks S_1 and S_2. To insert an element e , we simply push it into S_1. To delete an element , we pop each element in S_1 and then push it into S_2 when S_2 is empty. If S_2 is not empty, we pop the element in S_2 directly. Assume that the cost of push and pop is 1. Here is an example of insert a, insert b, delete.

	$S_1 =$ []	$S_2 =$ []	
INSERT(a)	$S_1 =$ [a]	$S_2 =$ []	
INSERT(b)	$S_1 =$ [b a]	$S_2 =$ []	
DELETE()	$S_1 =$ []	$S_2 =$ [a b]	"dump"
	$S_1 =$ []	$S_2 =$ [b]	"pop" (returns a)

Figure 1: Example of Q3

Start with no element in the queue, give the best upper bound you can on the amortized cost using accounting method when there are arbitrary insert and delete .

Solution:

The tightest amortized upper bounds are 3 units per insertion, and 1 unit per removal. We will prove this 2 ways (using the accounting and potential methods; the aggregate method seems too weak to employ elegantly in this case). (We would also accept valid proofs of 4 units per insertion and 0 per removal, although this answer is looser than the one we give here.)

Here is an analysis using the accounting method: with every insertion we pay 3: 1 is used to push onto S1, and the remaining 2 remain attached to the element just inserted. Therefore every element in S1 has 2 attached to it. With every removal we pay 1, which will (eventually) be used to pop the desired element off of S2. Before that, however, we may need to dump S1 into S2; this involves popping each element off of S1 and pushing it onto S2. We can pay for these pairs of operations with the 2 attached to each element in S1.

Problem 4:

Suppose we have a device that generates a sequence of independent fair random coin-flips, but what we want is a six-sided dice that generates the values 1,2,3,4,5,6 with equal probability. Give an algorithm that does so using $\frac{11}{3}$ coin-flips on average and explain the expected coin-flips your algorithm need to use.

Solution:

algorithm: generate 3 times and combine them as a binary number($eg: 000, 001, ..., 111$) and use 001 ˜110 to represent number 1 ˜6.

If we get 000, then we set the first bit of next round as 0 and generate 2 more times. Then we recursive the result.

If we get 111, then we set the first bit of next round as 0 and generate 2 more times. Then we recursive the result.

Problem 5:

Consider a 3-coloring problem in which you need to maximize the number of satisfied edges. In a graph $G = (V, E)$, edge (u, v) is *satisfied* if the node u and v are assigned with different colors. Design a randomized algorithm that the expected number of edges it satisfies is at least $\frac{2}{3}$ of the optimal number and prove it.

Solution:

Pick a color for each vertex randomly and independently in random. For each $e \in E$, $Pr[e \text{ is satisfied}] = \frac{2}{3}$. Let $X_e = 0$, if E is not satisfied and $X_e = 1$ otherwise. So

$$E[\text{number of satisfied edges}] = E[\sum_{e \in E} X_e] = |E| \cdot E[X_e] = \frac{2}{3}|E|$$

And obviously, $|E| \geq c^*$, so $E[numberofedges] = \frac{2}{3}|E| \geq \frac{2}{3}c^*$

Problem 6:

a) Design a scheme to generate unbiased coin flips using a biased coin (the coin has a probability p to generate a HEAD and you do not know the p). You are required to generate an unbiased result with the expected number of flipping the coin up to $\frac{1}{p(1-p)}$.

b) Modify your scheme to make it more efficient, i.e., try to get more bits every time you toss the coin (in expectation).

Solution:

(a) We rely on the fact that flips are independent, even if biased. To try to generate one bit, flip the coin twice. If you get heads followed by tails (HT) then output heads. If tails followed by heads (TH) output tails. If HH or TT, report failure.

Conditioned on having succeeded (gotten HT or TH) the probability that you got HT is equal to the probability you got TH, so the output you produce is unbiased.

If you fail, you try again, repeating until you succeed. The probability that you succeed in one trial is just $2p(1-p)$ (probability of one head and one tail). So

the number of trials you perform before succeeding has a geometric distribution; its expectation is $1/[2p(1-p)]$. Since we toss the coin twice at each trial, the expected number of total coin tosses is $1/[p(1-p)]$.

(b) The following solution owes to [Elias72]. This one is tricky. Any reasonable effort is sufficient for full credit.

Before we tackle the problem itself, let us consider the following scenario: suppose you are given a number r drawn uniformly at random from $\{1, \ldots, N\}$. How many unbiased bits can you output from such a sample? The following scheme will turn out to be asymptotically optimal: Suppose that the binary representation of N is as follows:

$$N = \alpha_m 2^m + \alpha_{m-1} 2^{m-1} + \cdots + \alpha_1 2^1 + \alpha_0 2^0$$

where $\alpha_i \in \{0, 1\}, \alpha_m = 1$, and $m = \lfloor \log N \rfloor$. We assign output strings to the N numbers, so that for every $i > 0$ with $\alpha_i = 1$, we map 2^i of the numbers to all the binary strings of length i. The exact assignment does not matter, but since every input number is equally likely, we are assured that the output bits are unbiased and independent random bits. Note that if N is odd, i.e. $\alpha_0 = 1$, one of the input numbers will not be assigned to any output - if we encounter it, we output nothing. Luckily, this case will become increasingly unlikely as N grows.

It remains to be seen how many bits we expect to output using this scheme. Due to the above construction, if $\alpha_i = 1$ for some $i > 0$, then we output a length i bit string withprobability $2^i/N$. So the expected output length is equal to

$$\begin{aligned} E_N := E[\ \text{ bits output }] &= \sum_{i=0}^{m} i \cdot \alpha_i \cdot \frac{2^i}{N} \\ &= \frac{1}{N} \left(\sum_{i=0}^{m} m \cdot \alpha_i \cdot 2^i - \sum_{i=0}^{m} (m-i) \cdot \alpha_i \cdot 2^i \right) \\ &= m - \frac{1}{N} \sum_{i=0}^{m} (m-i) \cdot \alpha_i \cdot 2^i \end{aligned}$$

We have $N \geq 2^m$, and $\sum (m-i)\alpha_i 2^i \leq \sum (m-i) 2^i = 2^{m+1} - m \leq 2^{m+1}$. This implies

$$\log N \geq m \geq E_N \geq m - \frac{2^{m+1}}{2^m} = m - 2 \geq \log N - 3$$

Let us now apply this observation to the problem we really want to solve, extracting bits from n biased coin flips. Suppose we flip the coin n times and see k heads. There are $\binom{n}{k}$ different different positions those k heads can take

among the n flips (that is, which of the k flips came up heads), and each is equally likely. So, using the above scheme, we can map them to bit sequences with an expected length of about $\log \left(\begin{array}{c} n \\ k \end{array} \right)$. Since the probability that we see k heads is $p^k(1-p)^{n-k} \left(\begin{array}{c} n \\ k \end{array} \right)$, we can conclude that the expected number E of output bits is

$$\sum_{k=0}^{n} p^k(1-p)^{n-k} \left(\begin{array}{c} n \\ k \end{array} \right) \log \left(\begin{array}{c} n \\ k \end{array} \right) - 3 \leq E \leq \sum_{k=0}^{n} p^k(1-p)^{n-k} \left(\begin{array}{c} n \\ k \end{array} \right) \log \left(\begin{array}{c} n \\ k \end{array} \right)$$

To bound this quantity will use the following two facts:

Fact 1 (Weak law for binomial distributions) For all $p \in [0,1], \varepsilon > 0$ there exists $n_0 \in \mathbb{N}$, such that for all $n \geq n_0, a(1-\varepsilon)$ fraction of all length n sequences of p-biased coin flips contain between $n(p-\varepsilon)$ and $n(p+\varepsilon)$ heads. Fact 2

$$\lim_{n \to \infty} \frac{1}{n} \log \left(\begin{array}{c} n \\ n \cdot p \end{array} \right) = H(p) := p \cdot \log \frac{1}{p} + (1-p) \cdot \log \frac{1}{1-p}$$

This second fact follows easily from $\log n! \approx n \log n -$ a consequence of Stirling's Formula; see Proposition B.1 in the MR book.

This first fact implies that (1) asymptotically grows like $\log \left(\begin{array}{c} n \\ np \end{array} \right)$, [1] which according tothe second fact converges to $nH(p)$, [2] which proves that our scheme obtains the optimum expected output length. (c) This is a (the) classic result from information theory [Shannon48]. The basic idea behind the proof is the following. By Fact 1 from part (b), we know that as n becomes large, almost all sequences have about np heads in them. So the input can be approximated by just $\left(\begin{array}{c} n \\ np \end{array} \right)$ sequences of equal probability. The 'best' we can do to get unbiased bits is to map all of these sequences to different output strings from some $\{0,1\}^k$. Since we only have $\left(\begin{array}{c} n \\ np \end{array} \right)$ sequences to map from, we have that $k \leq \log \left(\begin{array}{c} n \\ np \end{array} \right)$, which by Fact 2 from above implies that $k \leq nH(p)$ in the limit.