ShanghaiTech University

CS240 Algorithm Design and Analysis

Fall 2024

Problem Set 4

Due: 23:59, Jan. 5, 2024

1. Submit your solutions to the course Gradescope.
2. If you want to submit a handwritten version, scan it clearly.
3. Late homeworks submitted within 24 hours of the due date will be marked down 25%. Homeworks submitted more than 24 hours after the due date will not be accepted unless there is a valid reason, such as a medical or family emergency.
4. You are required to follow ShanghaiTech's academic honesty policies. You are allowed to discuss problems with other students, but you must write up your solutions by yourselves. You are not allowed to copy materials from other students or from online or published resources. Violating academic honesty can result in serious penalties.

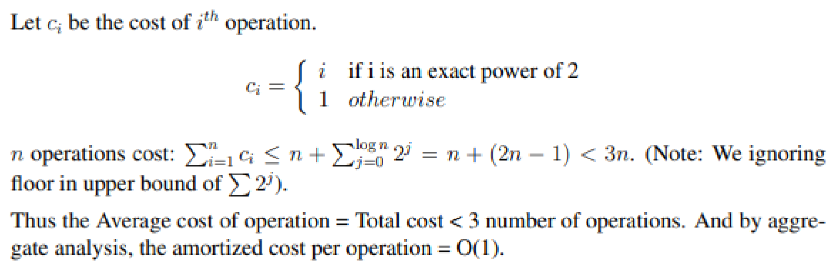
Problem 1:

Once upon a time, in a magical kingdom, there was a data structure that needed to be maintained by a group of skilled wizards. They performed a sequence of n operations on this data structure. The cost of each operation was determined by a mysterious rule: if the operation number was an exact power of 2 (like 1, 2, 4, 8, ...), the cost was equal to the operation number itself; otherwise, the cost was just 1.

The king of the kingdom asked the wizards to calculate the average cost of performing these operations, also known as the amortized cost per operation. You as the wizards of the kingdom, please use aggregate analysis or accounting methods to determine the amortized cost per operation.

Solution:

Aggregate analysis:



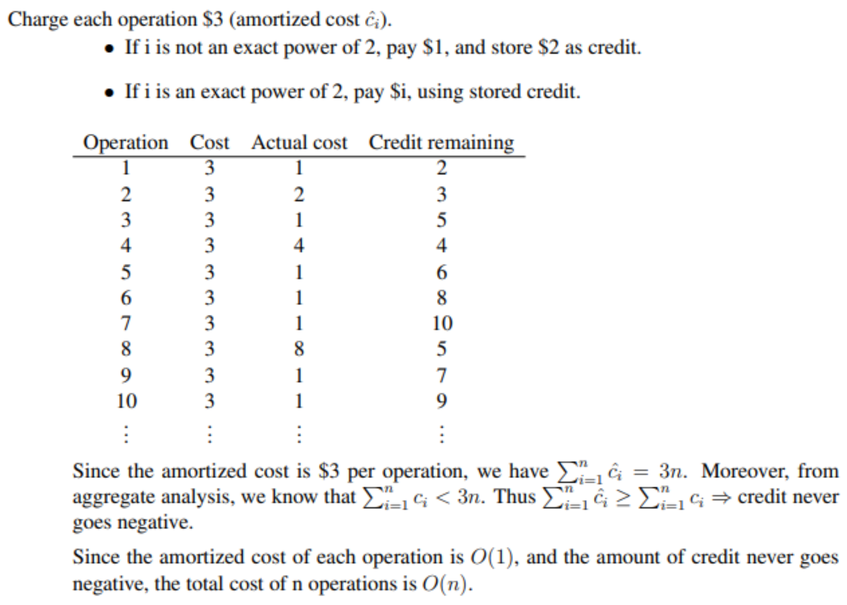
Let c_i be the cost of i^{th} operation.

$$c_i = \begin{cases} i & \text{if i is an exact power of 2} \\ 1 & otherwise \end{cases}$$

n operations cost: $\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\log n} 2^j = n + (2n - 1) < 3n$. (Note: We ignoring floor in upper bound of $\sum 2^j$).

Thus the Average cost of operation = Total cost < 3 number of operations. And by aggregate analysis, the amortized cost per operation = O(1).

Accounting method:



Charge each operation \$3 (amortized cost $\hat{c}_i$).

- If i is not an exact power of 2, pay \$1, and store \$2 as credit.
- If i is an exact power of 2, pay \$i, using stored credit.

Operation	Cost	Actual cost	Credit remaining
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
⋮	⋮	⋮	⋮

Since the amortized cost is \$3 per operation, we have $\sum_{i=1}^{n} \hat{c}_i = 3n$. Moreover, from aggregate analysis, we know that $\sum_{i=1}^{n} c_i < 3n$. Thus $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \Rightarrow$ credit never goes negative.

Since the amortized cost of each operation is $O(1)$, and the amount of credit never goes negative, the total cost of n operations is $O(n)$.

Problem 2:

Suppose a shipping company needs to process packages in a queue-like manner, but they only have access to stack-based storage units. One way to simulate a queue is to use two stacks S_1 and S_2. When a new package arrives, it is added to stack S_1. To process a package (dequeue), the system first checks if stack S_2 is empty. If it is, all packages from S_1 are transferred to S_2 one by one (this is called a "dump"). Then, pop from S_2.

For example, if the company processes the following operations: `INSERT(a)`, `INSERT(b)`, `DELETE()`, the results are:

Operation	Stacks State	
	$S_1 = []$	$S_2 = []$
`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`)

Suppose each stack operation (push or pop) takes 1 unit of time. Transferring n packages from S_1 to S_2 (a "dump") requires $2n$ units of work: n pops and n pushes.

(a) Suppose that (starting with an empty system) the company performs 3 arrivals (insertions), 2 removals (deletions), 3 more arrivals, and 2 additional removals. What is the total cost of these 10 operations, and how many packages remain in each stack at the end?

(b) If a total of n arrivals and n removals are performed in some order, what is the maximum possible time for a single operation? Provide an example sequence of operations that achieves this maximum time, and specify which operation causes it.

(c) Suppose the company processes an arbitrary sequence of arrivals and removals, starting with an empty system. What is the amortized cost of each operation? Provide the tightest possible (non-asymptotic) bounds. Use the accounting method to justify your answer. In particular, determine how much to charge (x) for an arrival and (y) for a removal. Prove your answer.

Solution:

(a) The total cost is $3 + (6 + 2) + 3 + (1 + 6 + 1) = 22$. At the end, S_1 has 0 packages, and S_2 has 2.

(b) An arrival always takes 1 unit, so a removal must cause our worst-case cost. No more than n packages can ever be in S_1, and no fewer than 0 packages can be in S_2. Therefore the worst-case cost is $2n + 1$: $2n$ units to dump, and one extra to pop from S_2. This bound is tight, as seen by the following sequence: perform n insertions, then n removals. The first removal will cause a dump of n packages plus a pop, for $2n + 1$ work.

(c) The tightest amortized upper bounds are 3 units per insertion, and 1 unit per removal.

With every insertion we pay \$3: \$1 is used to push onto S_1 and the remaining \$2 remains attached to the package just inserted. Therefore every package in S_1 has \$2 attached to it. With every removal we pay \$1, which will (eventually) be used to pop the desired package off of S_2. Before that, however, we may need to dump S_1 into S_2; this involves popping each package off of S_1 and pushing it onto S_2. We can pay for these pairs of operations with the \$2 attached to each package in S_1.

Problem 3:

Suppose you have $2n$ balls and 2 bins. For each ball, you throw it randomly into one of the bins. Let X_1 and X_2 denote the number of balls in the two bins after this process. Prove that for any $\varepsilon > 0$, there is a constant $c > 0$ such that the probability $\Pr[X_1 - X_2 > c\sqrt{n}] \leq \varepsilon$.

Solution:

The mean for X_2 is n , so in order to have $X_1 - X_2 > c\sqrt{n}$, we need $X_2 < E[X_2] - \frac{c}{2}\sqrt{n} = (1-\delta)E[X_2]$ for $\delta = \frac{c}{2\sqrt{n}}$. Plugging this into the Chernoff lower bound, the probability this happens is

$$e^{-\frac{1}{2}\delta^2 E[X_2]} = e^{-c^2/4}.$$

This can be made smaller than a constant ε by choosing the undetermined constant c large enough.

Problem 4:

Suppose there is a basic random function rand50() that returns 0 or 1 with equal probability (i.e., each outcome has a 50% chance of occurring). We want to construct a new random function Rand75() using calls to rand50() only—and without using any additional library functions or floating-point arithmetic—such that Rand75() returns 1 with probability 75% and 0 with probability 25%. Moreover, our goal is to **minimize the number of calls** to rand50().

1. Provide a specific randomized algorithm (i.e., show how to call rand50() and how to process the returned values) to achieve this goal.

2. Prove the correctness of your proposed algorithm by showing:

- The probability that the algorithm returns 1 is indeed 75%, and the probability that it returns 0 is 25%.
- The algorithm uses as few calls to rand50() as possible (or give a justification that the number of calls is near-optimal under these constraints).

Hint: You may consider operating on two independent random bits from rand50() using an appropriate bitwise operation (e.g., bitwise OR) along with a suitable procedure to achieve the desired probabilities.

Solution:

Call rand50() twice, obtaining two return values denoted by X and Y. Perform a bitwise OR operation on X and Y:

$$Z = X \,\mathrm{OR}\, Y.$$

Use Z as the output of Rand75(). In other words, if $Z = 1$, then return 1; if $Z = 0$, then return 0.

Correctness Proof Sketch

1. Since X and Y are independent and each takes the value 0 or 1 with probability 50%, we can list the four possible pairs:

   $$(X, Y) \in \{(0,0), (0,1), (1,0), (1,1)\}.$$

   Each pair occurs with probability 25%.

2. The bitwise OR of X and Y is 0 if and only if $(X, Y) = (0, 0)$, which has a probability of 25%. In all other cases, the result of OR is 1, for a total probability of 75%. Therefore, the algorithm outputs 1 with probability 75% and 0 with probability 25%.

Minimizing the Number of Calls

To see why calling rand50() only once cannot achieve a transformation from 50% to 75%, note that a single bit cannot directly encode a change from 50% to 75%. Hence, at least two calls are required. Since this algorithm uses exactly two calls, it meets the requirement of being optimal or near-optimal in terms of the number of calls.

Problem 5:

A delivery company needs to load n packages into trucks. Each truck has a maximum load capacity of C, and the weight of the n packages is denoted as $\{w_1, w_2, \ldots, w_n\}$, where $w_i \le C$ $(1 \le i \le n)$.

Design a polynomial-time 2-approximation algorithm to solve this problem and prove the correctness of your algorithm.

Solution:

Algorithm: First, initialize all the trucks as empty. Then, take each package one by one and place it into the first truck that can accommodate it.

The basic idea of the algorithm is to find the first truck that can accommodate package i. Its time complexity is $O(n^2)$.

Without loss of generality, assume the capacity of each truck is C, and C_i represents the total volume of packages packed in the i-th truck. Using the strategy, we have $C_i + C_j > 1$. Suppose the approximate solution for the bin packing problem is k, then:

- If k is even, $C_1 + C_2 + \cdots + C_k > k/2$.
- If k is odd, $C_1 + C_2 + \cdots + C_k > (k-1)/2$.

Adding these two equations, we get:

$$2\sum_{i=1}^{k} C_i > \frac{2k-1}{2}$$

The optimal solution to the bin packing problem is m. At optimal packing, all trucks are fully used to pack all packages, i.e.,

$$m = \sum_{i=1}^{m} C_m = \sum_{i=1}^{n} s_i$$

Thus, we have:

$$2\sum_{i=1}^{k} C_i = 2\sum_{i=1}^{n} s_i = 2m > \frac{2k-1}{2}$$

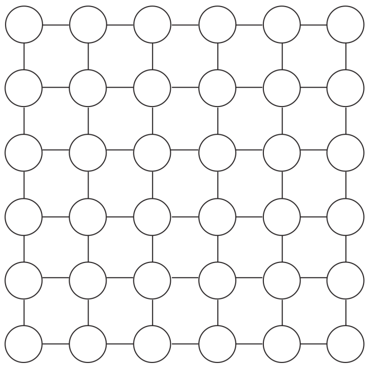
This implies:

$$\frac{k}{m} < 2$$

Therefore, the approximation ratio of the algorithm is less than 2.

Problem 6:

Consider an $n \times n$ grid graph G, as shown below.



Each node v in G has a weight $w(v) > 0$. You want to choose an independent set of nodes with maximum total weight. That is, you want to choose a set of nodes S with maximum total weight such that for any $v \in S$, none of v's neighbors are in S. To do this, consider the following greedy algorithm. Let V be the set of all nodes in G. Choose the node in V with the largest weight (breaking ties arbitrarily), add it to the independent set, then remove the node and all its neighbors from V. Repeat this process until V is empty. Let S be the output of this algorithm. Solve the following problems.

1. Let T be any independent set in G. Show that for each node $v \in T$, either $v \in S$, or there is a neighbor v' of v with $v' \in S$ and $w(v) \leq w(v')$.

2. Show that the greedy algorithm is a 4-approximation.

Solution:

(a) If v is not in S, then v must be deleted by the heaviest first algorithm. According to the algorithm, one of its neighbors must be in S, and its neighbor must have largest weight in the remaining nodes at that time, which means $w(v) \leq w(v')$.

(b) Consider any other independent set T, and let $w(T)$ stands for its weight. For each node v in S, it either also belongs to T, or not. If it belongs to T, then $w(T)$ also contains $w(v)$. Otherwise, $w(T)$ may (at most) contains all the four neighbors of of v, which have total weights $w < 4w(v)$. So summing over all the nodes in S, $w(T) \leq 4w(S)$.