ShanghaiTech University

CS240 Algorithm Design and Analysis

Fall 2023

Problem Set 4

ZHAOYU

Due: 23:59, Jan. 19, 2024

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:

If the set of stack operations included a MULTIPUSH operation, which pushes k items onto the stack. Analyze the amortized cost of stack operations (including PUSH, POP, MULTIPOP and MULTIPUSH).

```
MULTIPUSH(S, a, k)
   While k > 0
      PUSH(S, a[k])
      k = k - 1
```

Solution:

The time complexity of such a series of operations depends on the number of pushes (pops vise versa) could be made. Since one MULTIPUSH needs $\theta(k)$ time, performing n MULTIPUSH operations, each with k elements, would take $\theta(kn)$ time, leading to amortized cost of Θ(k)

Problem 2:

Suppose we perform a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 3, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

Solution:

$$let \quad k = floor(\log_3 n)$$

$$SumCost = \sum_{i=1}^{k} 3 * \frac{(3^k - 1)}{3 - 1} + n - k$$

$$amortized\ \ cost\ \ per\ \ operation = \frac{SumCost}{n}$$

amortized cost per operation = O(1)

Problem 3:

Given a set of positive integers, $A = a_1, a_2, ..., a_n$. And a positive integer B. A subset $S \in A$ is GOOD if

$$\sum_{a_i \in S} a_i \leq B$$

Given an approximation algorithm that it returns a GOOD subset whose total sum is at least half as large as the maximum total sum of any GOOD subset, with the running time at most $O(nlogn)$

Solution:

First sort A in non-increasing order and then relabel the numbers a1, a2, . . . , an in this order. The run the algorithm.

Algorithm 1 Pseudocode for Algorithm

$S \leftarrow \emptyset$

$T \leftarrow 0$

for $i \leftarrow 1$ to n **do**

  if $T + a_i \leq B$ **then**

    $S \leftarrow S \cup \{a_i\}$

    $T \leftarrow T + a_i$

  end if

end for

return S

Prove:

The proof is by contradiction. Suppose that there is a feasible subset S^* such that the total sum of S (the subset returned by the algorithm) is less than half the total sum of S^*. Then there is an element $a \in S^*$ such that $a \notin S$. Now let us consider why a is not added to S. The answer is simple: a is not added to S because when a is considered by the algorithm, adding a to the elements currently in S causes the total sum to exceed B. Therefore, the total sum of S plus a is greater than B. Hence, either (i) the total sum of S is greater than $B/2$, in which case we have proved the claim, or (ii) $a > B/2$. So we assume (ii) holds. Since the algorithm considers elements in non-increasing order, the elements already in S when a was considered are also $> B/2$. Thus, the total sum of S is $> B/2$.

Problem 4:

An undirected graph G = (V, E) with node set V and edge set E is given. The goal is to color the edges of G using as few colors as possible such that no two edges of the same color are incident to a common node. Let OPT(G) denote the minimum number of different colors needed for coloring the edges of G.

Show that there exists a Greedy algorithm that needs at most 2 · OPT(G)-1 different colors for any graph G. Prove that your algorithm always obtains a valid solution, i.e., no two edges of the same color are incident to a common node

Solution:

The greedy algorithm can be defined as follows:

For each edge, choose a color that has not yet been used on its adjacent edges. If possible, choose an existing color; if not, introduce a new color.

Proving the Validity of the Algorithm

Lemma 1: At any point, if an edge needs to be colored, then there are at most $\deg(v) - 1$ colors that have already been used on other edges adjacent to its endpoints, where $\deg(v)$ is the maximum degree of its endpoints.

Lemma 2: The greedy algorithm will never choose a new color for an edge if there is an already used color available.

Proof This is a direct consequence of the greedy strategy, which always prefers an already used color.

Suppose $\mathrm{OPT}(G)$ is the minimum number of colors needed for edge coloring of graph G. According to Lemma 1, for any node v, there are at most $\deg(v) - 1$ colors used on adjacent edges, and $\deg(v) - 1 \leq \mathrm{OPT}(G) - 1$.

Therefore, for each edge, the greedy algorithm needs to consider at most $\mathrm{OPT}(G)$ different colors (the $\mathrm{OPT}(G) - 1$ already used plus one new color). Since the algorithm always prefers an already used color, this means in the worst-case scenario, the algorithm will use $2 \cdot \mathrm{OPT}(G) - 1$ colors.

Problem 5:

Given a function rand2() that returns 0 or 1 with equal probability, implement rand3() using rand2() that returns 0, 1 or 2 with equal probability. Minimize the number of calls to rand2() method. Prove the correctness.

Solution:

Algorithm 2 code for algorithm

while True **do**
  $R1 \leftarrow rand2()$
  $R2 \leftarrow rand2()$
  $R3 \leftarrow 2 * R1 + R2$
  if R3 ≤ 2 **then**
    return $R3$
  end if
end while

Proof of Correctness:

The function 'rand2()' is called twice, generating two bits. Each bit can be 0 or 1 with equal probability. Therefore, the combination of these two bits (00, 01, 10, 11) is also generated with equal probability. Each of these combinations corresponds to a decimal number (0, 1, 2, 3), so each number in the range [0, 3] has an equal probability of $\frac{1}{4}$.

Since we need a range of [0, 2], the number 3 (binary 11) is discarded. This means that the algorithm only returns 0, 1, or 2. Each time we generate a number, there's a $\frac{3}{4}$ chance that it's in the desired range and a $\frac{1}{4}$ chance that we need to retry. The probability of getting 0, 1, or 2 remains equal because the process of generating these numbers is independent and identically distributed.

The expected number of calls to 'rand2()' can be calculated. On average, the 'while' loop runs $\frac{4}{3}$ times (since there's a $\frac{3}{4}$ probability of getting a number in the desired range). Each loop makes 2 calls to 'rand2()'. Therefore, the expected number of calls to 'rand2()' is $2 \times \frac{4}{3} = \frac{8}{3}$, which is less than 3 calls on average.

Problem 6:

Assume that you have a function `randM()` which returns an integer between 0 and $M-1$ (inclusive) with equal probability. Write an algorithm using the `randM()` function to implement a `randN()` function, where N is not necessarily a multiple of M, but `randN()` needs to return an integer between 0 and $N-1$ with equal probability.

Solution:

Algorithm 3 Generate a random number between 0 and $N-1$ using $randM()$

$k \leftarrow 0$

while $M^k < N$ **do**

  $k \leftarrow k+1$

end while

loop

  $number \leftarrow 0$

  for $i \leftarrow 0$ to $k-1$ **do**

    $number \leftarrow number \times M + \text{RANDM}$

  end for

  if $number < N \times \left(\lfloor \frac{M^k}{N} \rfloor\right)$ **then**

    return $number \bmod N$

  end if

end loop

The generated number is uniformly distributed between 0 and $M^k - 1$, and we only accept numbers that fall within the largest multiple of N that is smaller than M^k. This ensures that each number in the range [0, $N-1$] has an equal probability of being chosen.

The efficiency of the algorithm depends on the ratio of N to M. The closer M^k is to N, the fewer the number of expected rejections. In the worst case, the number of rejections can be high, especially if N is much smaller than M^k.

The modulo operation ($number//N$) ensures that the result is always within the range [0, $N-1$], even if 'number' is larger than N.