ShanghaiTech University

CS240 Algorithm Design and Analysis

Fall 2023

Problem Set 4

Due: 23:59, Jan. 19, 2023

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 total cost of stack operations depends on the number of pushes. One MULTIPUSH operation needs $O(k)$ time, and n MULTIPUSH operations needs $O(kn)$ time. Therefore the amortized cost is $O(k)$.

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 c_i be the cost of operation i. Then we have

$$
\begin{aligned}
\sum_{n}^{i=1} c_i &= \sum_{i=1}^{\lfloor \log_3 n \rfloor} 3^i + \sum_{i \leq n \text{ and } i \text{ is not power of } 3} 1 && (1)\\
&\leq \left(\sum_{i=1}^{\lfloor \log_3 n \rfloor} 3^i \right) + n && (2)\\
&= \frac{3^{1+\lfloor \log_3 n \rfloor} - 3}{2} + n && (3)\\
&\leq 3n + n = O(n) && (4)
\end{aligned}
$$

The amortized cost per operation is $O(1)$.

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:

- $Result = []$
- Sort A in reverse order, $A' = a_{s_1}, a_{s_2}, ..., a_{s_n}$. ($\mathbf{O(nlogn)}$)
- Traverse A'. If $a_{s_i} + sum(Result) \leq B$, append a_{s_i} into $Result$ ($\mathbf{O(n)}$)
- return $Result$

Obviously, $Result$ is GOOD. Now prove $Result$ is at least half as large as the maximum total sum of any GOOD subset. Consider two situations:

If $a_{s_n} \in Result$, means all $a_i \leq B$ is in $Result$. $Result$ is the GOOD subset with the maximum total sum.

Else, we assume $a_{s_j} \in Result$ and $a_{s_{j+1}} \notin Result$. Thus, $a_{s_{j+1}} + sum(Result) > B$.

And $a_{s_j} \in Result$, $a_{s_j} > a_{s_{j+1}}$. So $sum(Result) > B/2$. $Result$ is at least half as large as the maximum total sum of any GOOD subset.

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:

Number all colors and use the color with the number as small as possible.

Sort all edges according to the number of neighbors. Start traversing from the edge with few neighbors. Give each edge the smallest possible color.

Prove:

Assume the biggest degree of a node in G is N. Obviously, $N \leq OPT(G)$ So the biggest number of neighbor an edge can have is less than $2N-2$. According to how we choose colors, the number of color is no bigger than $2N-1$. Thus the number of color is no bigger than $2OPT(G)-1$. So the algorithm needs at most 2 · OPT(G)- 1 different 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:

The idea is to use expression 2rand2() + rand2(). It returns 0, 1, 2, 3 with equal probability. To make it return 0, 1, 2 with equal probability, we eliminate the undesired event 3.

$$P(0) = P(1) = P(2) = P(3) = 0.25$$

Thus, 0, 1, 2 are in the same probability.

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:

The idea is to use `randM()` to generate a range of numbers large enough and then adjust this range to the desired 0 to $N - 1$ interval. The specific steps are as follows:

1. Determine a number K which is a power of M such that $K \geq N$.
2. Use `randM()` to generate a number within the range of K. This can be thought of as generating a number in base M, with each call to `randM()` providing one base M digit.
3. If the generated number is less than the largest multiple of N that is less or equal to K, then take the remainder of this number modulo N as the result.
4. If the generated number is greater than or equal to this threshold, discard it and regenerate.

The pseudocode is as follows:

```
def randN():
    # The largest multiple of N less than or equal to K
    max_val = (K // N) * N
    while True:
        num = 0
        # Generate a number within the range of K
        for _ in range(logarithm of K to the base M):
            num = num * M + randM()
            if num >= max_val:
                break
        if num < max_val:
            # Return a number between 0 and N-1
            return num % N
```

The correctness of this algorithm is based on the following:

- Each call to `randM()` generates a uniformly random digit.
- The generated numbers are uniformly distributed within the range 0 to $K - 1$ because K is a power of M.
- Discarding numbers larger than or equal to the largest multiple of N that is less than or equal to K ensures the remainder is also uniformly distributed.
- This guarantees that the algorithm, when terminated, returns a number in the range 0 to $N - 1$ that is also uniformly distributed.