CS-GY 6033: Design and Analysis of Algorithms I

Fall 2024

Instructor: Nairen Cao **NYU Tandon School of Engineering**

Problems	**1**	**2**	**3 or 4**	**5 (ungraded)**	**6 (ungraded)**	**7 (Bonus)**	**Total**
Max. Score	40	40	20	0	0	10	100

Table 1: Score Summary

Homework #2

Assigned: 9/27/2024 **Due: 10/10/2024**

General Instructions Write all solutions clearly, concisely, and legibly. It is okay to provide hand-written and scanned/photographed solutions as long as they are clearly legible. If typing, using LaTeX is recommended.

You should generally describe algorithms in words and high-level pseudocode. Specifically, emphasize the goal of each step in your algorithm rather than the coding details. After giving a high-level description, provide more detail on the implementation of each step where appropriate.

The purpose of having these problem sets is to enhance your mathematics and programming skills in the scope of designing algorithms. These problems may be quite challenging, so discussions with classmates are encouraged. However, you should spend at least half an hour thinking about the problems individually before discussing them. Although discussions (and consulting ChatGPT and WolframAlpha) are allowed, you need to write the solution in your own words and by yourself. Please acknowledge any person or reference you discuss with or consult from.

Special Instructions

- Problems 3 and 4 are about using data structures to construct new data structures. You only need to finish one of these two problems.
- Problems 5 and 6 are designed to enhance the understanding of hash tables and priority queues. If you are not familiar with how to use them, you can try them. They are ungraded.
- Problem 7 is about the analysis of the hash table and is super challenging. I will give you a 10-point bonus for this problem, while 100 points will give you full credit.

Problem 2-1. Exercises on graph searches

The following problem parts are unrelated.

(a) (10 points) What changes about the details of BFS if we represent the graph by an adjacency matrix instead of an adjacency list? What is the running time of the modified adjacency-matrix-based algorithm? Explain your answer—don't just state a bound.

(b) (10 points) Give a counterexample to the following conjecture: if there is a path from u to v in a directed graph G, then any depth-first search must result in $v.s \le u.f$, where $v.s$ denotes the start time of v and $u.f$ denotes the finishing time of u.

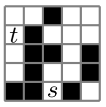
There are counterexamples as small as 3 vertices. The smaller and more concise the counterexample, the better. If your example graph is too large, you may lose some credit. Note that you must explain why your counterexample is indeed a counterexample to the conjecture, i.e., you should explain how DFS would execute.

(c) (10 points) You're presented as input an $n \times n$ 2D grid $G[1\,..\,n][1\,..\,n]$ of cells which are either clear (white) or blocked (black), modeling a maze where you can move horizontally or vertically (not diagonally) through white squares.

Describe and analyze a good algorithm for determining if there's a path from a given starting cell $s = (i, j)$ to a given ending cell $t = (i', j')$.

You may apply an algorithm that we've learned as a black box. Note however that the problem is not stated as a graph — if you want to apply a graph algorithm, you would have to describe what the corresponding graph is.

For example, for this grid with displayed locations s and t, the answer should be "No."



Your runtime bound should be a function of n, where the grid consists of n^2 cells.

(d) (10 points) There are two types of professional wrestlers: "good guys" and "bad guys." Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n wrestlers and a list of r pairs of wrestlers for which there are rivalries.

Give an $O(n + r)$-time algorithm to determine whether it is possible to designate some of the wrestlers as good guys and the remainder as bad guys such that each rivalry is between a good guy and a bad guy. If it is possible to perform such a designation, your algorithm should produce it.

Note that this problem does not explicitly provide a graph as input — your solution would need to specify what the graph would be.

Problem 2-2. Exercises on Strongly Connected Components

The following problem parts are unrelated.

(a) (10 points) Describe an algorithm to compute the transpose of a directed graph, given in adjacency list representation, in $O(V + E)$ time. In this case, I want you to also provide pseudocode.

(b) (15 points) Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second DFS and scanned the vertices in *increasing* finishing times.

Does this simpler algorithm always produce correct results? If yes, justify your answer. If no, provide and explain a counterexample.

(c) (15 points) Given a directed graph $G = (V, E)$, explain how to create another graph $G' = (V, E')$ such that (i) G' has the same strongly connected components as G, (b) G' has the same component graph as G, and (c) E' is as small as possible. Describe a fast algorithm to compute G' and analyze its running time.

(d) (Ungraded) A directed graph $G = (V, E)$ is ***semiconnected*** if, for all pairs of vertices $u, v \in V$, we have $u \rightsquigarrow v$ and/or $v \rightsquigarrow u$. Give an efficient algorithm to determine whether or not G is semiconnected. Prover you algorithm is correct and analyze its running time.

Hint: first consider what you would do on a directed acyclic graph.

Problem 2-3. Constructing a Minimum Stack and Queue

You are tasked with constructing a Minimum Stack and Queue that efficiently supports standard operations and a min() operation that returns the minimum element in constant time. **For each part, you are required to:**

•Provide the algorithm for the given task.

•Analyze the time complexity of each operation.

(a) Design a data structure called Minimum Stack that supports the following operations:

- **Push(x):** Add element x to the top of the stack.
- **Pop():** Remove the element from the top of the stack.
- **Top():** Return the top element of the stack.
- **Min():** Return the minimum element in the stack in constant time.

(b) Construct a Minimum Queue that supports:

- **Enqueue(x):** Add element x to the end of the queue.
- **Dequeue():** Remove the element from the front of the queue.
- **Min():** Return the minimum element in the queue in constant time.

Problem 2-4. Hash Table with Random Access

Design a data structure that supports the following operations efficiently:

•**Insert(x):** Insert element x into the data structure.

•**Remove(x):** Remove element x from the data structure.

•**Contains(x):** Check if element x is in the data structure.

•**GetRandom():** Return a random element from the data structure with equal probability.

Requirements:

•Each operation should have an average time complexity of $O(1)$.

•You may use data structures such as dynamic arrays, lists, and hash tables.

(a) Provide algorithms for Insert, Remove, Contains, and GetRandom.

(b) Analyze the time complexity of each operation.

Problem 2-5. (Ungraded) Hash Table Application

Given: An array of integers `A` and an integer k.
Task: Return one subarray whose sum equals k. If such a subarray does not exist, output ”No”.
A subarray is a contiguous, non-empty sequence of elements within an array.

Example

Input: `A` = [1, -2, 3, -4, 5], $k = 1$
Output: [-2, 3]
Explanation: There are several subarrays that sum to 1: [1], [-2, 3], [-4, 5]. You only need to output one of them.

(a) Provide an algorithm for finding such a subarray.

(b) Analyze the time complexity of your algorithm.

Problem 2-6. (Ungraded) Priority Queue Application

The task is to design a data structure that supports the following operations efficiently:

- `addNum(int num)`: Add a number to the data structure.
- `findMedian() -> float`: Return the median of all elements so far.
- You may use data structures such as max or min heap directly.

The median is the middle value in a sorted list of integers. If the list contains an even number of integers, the median is the average of the two middle values. There will be at least one element before calling `findMedian`.

Example:

Input:

```
["addNum", "addNum", "findMedian", "addNum", "addNum", "findMedian"]
[[6], [10], [], [2], [3], []]
```

Output:

```
[null, null, 8.0, null, null, 5.0]
```

Explanation:

- `addNum(6)` is called.
- `addNum(10)` is called.
- `findMedian()` is called and returns 8.0. The numbers so far are [6, 10].
- `addNum(2)` is called.
- `addNum(3)` is called.
- `findMedian()` is called and returns 5.0. The numbers so far are [2, 3, 6, 10].

(a) Provide algorithms for `addNum`, `findMedian`, you can use heap without the implementation.

(b) Analyze the time complexity of each operation.

Problem 2-7. (Bonus) Hash Table Analysis: Linear Probing with uniform hashing function

In this problem, we will analyze the performance of a hash table using linear probing for collision resolution.

Assume that the hash function h is uniform random, meaning that:

$$\Pr[h(e) = i] = \frac{1}{m}$$

for any key e and any index $i \in \{0, 1, 2, \ldots, m-1\}$. We assume that n elements have already been inserted into the hash table, and let S represent the set of elements in the hash table. Hence, $|S| = n$.

Define the **load factor** α as:

$$\alpha = \frac{n}{m}$$

where n is the number of inserted elements and m is the size of the hash table. Assume that α is a constant. We are interested in analyzing the relationship between the load factor α and the access time for a randomly chosen element from S.

The following algorithm describes how we insert an element into the hash table using linear probing with uniform hashing:

To better understand the average cost of accessing elements, consider the following scenario. For each slot i in the hash table, we initialize a counter $C_i = 0$. Whenever we insert an element, we query the

Algorithm 1 Insert with Linear Probing and Uniform Hashing

```
1: Input: Key $e$, Hash table $T$ of size $m$
2: Output: Insert key $e$ into $T$
3: $i \leftarrow h(e)$                                  ▷ Hash the key $e$ to index $i$
4: while $T[i] \neq$ None do
5:     $i \leftarrow (i+1) \mod m$                      ▷ Use linear probing if the slot is occupied
6: end while
7: $T[i] \leftarrow e$                                  ▷ Insert the key $e$ into the empty slot
8: return
```

hash function to obtain a random slot, and if the slot is occupied, we move to the next slot (using linear probing). We increment the counters for all slots we touch during the probing process. For instance, if the hash function gives us index 4, but slots 4, 5, and 6 are occupied while slot 7 is empty, we insert the element in slot 7 and increment C_4, C_5, C_6, C_7 by 1.

(a) After finishing all insertions, the counters C_i follow a certain distribution. What is the expected access time for retrieving a random element from the hash table in terms of the counters C_i? Specifically, derive the expected access time based on the number of slots probed during an insertion.

(b) Let $B_j(i)$ be the indicator that after n insertions, $C_i = j$, meaning that slot i has been touched j times. More formally, $B_j(i) = 1$ if the slot i is touched j times after all insertions, and $B_j(i) = 0$ otherwise. How are the following probabilities related?

$$\Pr[B_j(i) = 1] \quad ? \quad \Pr[B_j(i+1) = 1]$$

Use this relation to establish a connection between $E[C_i]$ and $E[C_{i-1}]$.

(c) Let $N(i) = |\{e \in S \mid h(e) = i\}|$ represent the number of elements whose hash function value equals i. In other words, $N(i)$ is the number of elements that are hashed to the i-th slot of the hash table. For large enough n and m, show that:

$$\Pr[N(i) = k] \approx e^{-\alpha}\frac{\alpha^k}{k!}$$

where $\alpha = \frac{n}{m}$ is the load factor, and e is the natural base of the exponential function.

(d) Consider the event $B_j(i) = 1$, which indicates that slot i is touched j times during the whole insertion process. For a fixed i, if $B_j(i)$ occurs, what can you infer about the states of slots $i-1$ and the number of elements hashed to slot $i-1$, denoted by $N(i-1)$? Let $q_j = \Pr[B_j(i) = 1]$ be the probability that slot i is touched j times, and $p_k = \Pr[N(i) = k]$ be the probability that k elements are hashed to slot i. Explain the equation:

$$q_j = q_0 p_j + q_1 p_j + q_2 p_{j-1} + \cdots + q_{j+1} p_0$$

Hint: Analyze how the fullness of slot i depends on the state of slot $i-1$ and the number of elements hashed to it.

(e) Now assume that n is infinite. From the previous problem, we can derive a set of equations for q_j using the probabilities $q_{j+1}, q_j, \ldots, q_0$ and the corresponding probabilities $p_j, p_{j-1}, \ldots, p_0$, for $j \in [0, n]$. We are interested in computing the expected value of C_i, which is the number of slots touched during the insertion of an element:

$$E[C_i] = \sum_{j=1}^{\infty} j q_j.$$

To compute $\sum_{j=1}^{\infty} jq_j$, we will apply a less formally defined method using generating functions.

To begin, let's focus on computing q_0. One approach to find q_0 is by using generating functions. Define the following generating function for the probabilities p_k:

$$f(x) = \sum_{k=0}^{\infty} p_k x^k,$$

where $p_k = e^{-\alpha}\frac{\alpha^k}{k!}$. By expanding the function, show that $f(x)$ is a well-known result of Taylor expansion.

(f) Now, define a new generating function for the sequence q_k:

$$g(x) = \sum_{k=0}^{\infty} q_k x^k.$$

Given the previous problems and the structure of the system, show that $g(x)$ is given by:

$$g(x) = \frac{q_0 e^{\alpha x}(x-1)}{xe^{\alpha} - e^{\alpha x}}.$$

(g) Next, use L'Hôpital's rule to compute q_0 from the expression for $g(x)$. Specifically, evaluate the limit of $g(x)$ as $x \to 1$ to obtain q_0. What's q_0? Does it make sense?

(h) Next, compute $g'(1)$. First, compute $g'(x)$ and then use L'Hôpital's rule to evaluate $g'(x)$ as $x \to 1$.