You are allowed to collaborate on this homework. Please limit yourself to groups of two (or at most three) people. We adhere to a “whiteboard” collaboration policy: you should discuss with your partner on whiteboards (or equivalents) but then should go away and write down your solutions yourself. You must not share written work. You should write down the names of the person(s) you collaborate with. In general, you should solve the homework problems using only to material we refer you to (or put on the course page), and not books or other online resources. If you have reason to use any resources we point you to (except the course notes), please cite them.

A word of advice: try to solve the problem by yourself before working with your partner(s). That is the way to improve your problem-solving skills, which is one of taking this course.

Homeworks will be due at 11:59pm on the due date on gradescope. Corrections and changes will appear on the course webpage and on Piazza, please check them regularly.

Exercises

Exercises are for fun and edification, please do not submit. But please do try to solve them, we may need ideas from there later in the course!

1. **(Array Maintenance)** You are asked to implement a dynamic array: one that can handle inserts and deletes, without needing an upper bound on its length.

   (a) First, inserts-only. At each time you have \( n \) elements in the array which is of length \( L \). Initially \( n = 0 \) and \( L = 1 \). If you insert an element when \( n = L \), i.e., the current array is full, you allocate a new array of length \( 2L \), copy over the \( n \) elements from the old array to the new one, and de-allocate the old array. Since you now have room in the current array, insert the new element into the array (thus making \( n \) increase to \( n + 1 \)). The cost of all the copying/deallocating in going from \( L \) to \( 2L \) is \( cL \) (the length of the old array) for some constant \( c \), and the cost of inserting the element is 1. E.g., a sequence of inserts starting from the beginning will cost \( 1, c + 1, 1, 2c + 1, 1, 4c + 1, 1, 1, 1, \ldots \). Show that the amortized cost of a sequence of \( n \) inserts will be \( O(c) \).

   (Succinctly: we maintain \( L = \max(1, 2^{\lceil \log_2 n \rceil}) \); that is, \( L \) equals \( n \) raised to the closest higher power of 2.)

   (b) Now consider maintaining the array under both inserts and deletes. We want that \( L \approx n \) at all times, so if the current array size \( L \gg n \) then we allocate a shorter array of size \( L' \approx n \), copy the \( n \) elements over, and de-allocate the old array. The cost of this is \( dL \) for some constant \( d > 0 \). Show that the above strategy of maintaining an array equal to \( n \) rounded up to the next higher power of 2 is bad, if there are both inserts and deletes. Namely, show a sequence of \( n \) inserts and deletes, which can incur \( \Omega(n^2) \) cost according to this strategy.

   Now give a strategy to maintain arrays so that \( L \) always lies in the interval \([n, 4n]\) such that the amortized cost is at most \( O(c + d) \), i.e., a constant.

2. **(Coin-Flipping)**

   (a) Suppose you have access to a bent coin, that randomly outputs a 1 with some fixed but unknown probability \( p \in (0, 1) \), and a 0 with probability \( 1 - p \). Give an procedure
that uses flips of this bent coin to implement a fair coin: i.e., a procedure that outputs “heads” with probability exactly 1/2. What’s the expected number of times you flip the bent coin?

(b) Now suppose you have access to a fair coin! Given the binary representation for some value \( p \in [0, 1] \), give an algorithm for implementing a coin with bias \( p \) that only flips the fair coin few times. What is the expected number of times you flip the fair coin?

(c) You want to sample uniformly at random from the set of all \( n \)-bit strings that are balanced, i.e., that contain exactly \( n/2 \) 0’s and \( n/2 \) 1’s. You do the following: take a uniform random sample \( \omega \) from the set of all \( n \)-bit strings. output \( \omega \) if it is balanced, else reject, and sample again. Show that the expected number of times you sample an \( \omega \) is \( O(\sqrt{n}) \). (Hint: Stirling’s formula.) You should also try to show that the expected number of samples is \( \leq n + 1 \), by arguments from first principles.

3. Forest for the Trees

Here are slight variants on some familiar algorithms.

(a) Given a connected undirected graph with non-negative edge lengths, you want to pick some of its edges so that you have exactly \( k \) connected components in the graph. Now among all edges whose endpoints lie in distinct components, consider the one with the shortest length. (Think of the \( k \) components as a clustering, and call the length of this edge the “separation” of the clustering.) You want a clustering whose separation is as large as possible. This clustering is often called single-linkage clustering. 

Show how to use the minimum spanning tree in general (or Kruskal’s algorithm in particular) to solve this problem.

(b) Given a directed graph with edge weights \( w_e \geq 0 \), and two nodes \( s \) and \( t \), you want to find a path from \( s \) to \( t \) whose “bottleneck” weight is as large as possible. (Given a path, the bottleneck weight is the least weight among the edges on the path.) Show how to alter Dijkstra’s algorithm slightly to solve this problem in essentially the same amount of time.

Problems

Please solve problems #4 and #5, and any two of the first three problems.

1. Fast Multiplication (20 pts)

Given two \( n \)-bit strings, you are asked to output their product in the form of a binary string. For instance, if you received 101 and 11 as inputs, you should output 1111. The time complexity of the algorithm in this problem is the number of basic bitwise operations, such as AND, OR, XOR. You may also assume that you have access to primitive operations of addition and subtraction and bit-shifts of \( n \)-bit strings which use \( O(n) \) basic bitwise operations.

(a) Show the naïve algorithm we learnt from elementary school (potentially kindergarten) has time complexity \( O(n^2) \).

(b) Show a faster algorithm with time complexity \( o(n^2) \) and state the time complexity in \( O \) notation.
Hint 1: Use the same trick as we did in class to solve matrix multiplication recursively.

Hint 2: Multiplying two 2-bit numbers can be done with four 1-bit multiplications. (If you’re multiplying \(ab\) and \(cd\), where each letter denotes a single bit then you can multiply \(ac, ad, bc, bd\) and then return \(4ac + 2ad + 2bc + bd\). Note that we don’t count multiplying by 2 and 4, since that can be done using bit shifts.) Can you do it with just 3?

2. Triangle-Counting Again (20 pts)
Here’s an algorithm that given a graph \(G = (V, E)\), counts the number of triangles in time \(O(m^{1.5})\). (For this problem, assume you can check in constant time whether any edge \((i, j)\) exists in \(G\) or not.) Fix some parameter \(\alpha \geq 1\), and let \(H \subseteq V\) be the “heavy” nodes in \(G\) that have degree at least \(\alpha\).

- For every triple of nodes in \(H\), check if they form a triangle.
- For every node \(u\) in \(L := V \setminus H\), for every pair of edges \((u, v), (u, w)\) incident to \(u\), check if \(\{u, v, w\}\) forms a triangle.

Give a suitable choice of \(\alpha\) (it could be a function of \(m\) and \(n\)), and complete the algorithm description to output the number of triangles in \(G\) in \(O(m^{1.5})\) time. Prove your runtime.

Observe that this algorithm is faster than the matrix-multiplication based approach when \(m \ll n^2\). If matrix multiplication takes \(O(n^\omega)\) time for some constant \(\omega \in [2, 3)\), for what values of \(m\) will this algorithm be faster?

3. The Mathematician Formerly Known as Leonardo di Pisa (20 pts)
Suppose instead of using powers of two, we now represent integers as the sum of Fibonacci numbers. That is, rather than representing a number as an array of bits, we keep an array of “Fibbits” so that \((x_k x_{k-1} \ldots x_1)_F\) denotes the number \(\sum_{i=1}^{k} x_i F_i\). As an example, the Fibonacci number \((1101)_F = F_4 + F_3 + F_1 = 1 + 2 + 3 = 6\). Recall that the Fibonacci numbers satisfy the recurrence \(F_0 = 0, F_1 = 1\) and \(F_{n+2} = F_{n+1} + F_n\).

a) Show that every positive integer \(N\) can be represented as a Fibonacci number.

b) Give an algorithm to increment a Fibonacci number in constant amortized time. I.e., starting from the all-zeros state and performing \(n\) increments incurs a cost of \(O(n)\).

4. Sampling and Accounting (30 pts)

(a) (20 points) Suppose we are storing income data for a set of people, and want to be able to draw random samples from the set that meet certain criteria. In particular, we are given tuples of the form \((\text{name}, \text{income})\), and we want to perform queries of the form \(\text{randomSample}(\text{lower}, \text{upper})\), which returns, uniformly at random, the name of someone with income between \(\text{lower}\) and \(\text{upper}\) (inclusive). In addition, we want to be able to insert and delete \((\text{name}, \text{income})\) tuples to and from the data. Describe a data structure that can perform all these operations in expected time \(O(\log n)\), where \(n\) is an upper bound on the number of inserts plus deletes.

(b) (10 points) Now suppose we are keeping track of changes to our bank account. We are given transactions, each of which looks either like \((++, \text{time} t)\) or like \((-−, \text{time} t)\): the former means we added $100 to our account at time \(t\), and the latter that we withdrew $100 from our account at time \(t\). For simplicity, assume that the time of each transaction is distinct, and the account is initially empty.
We must support the ability to add and remove transactions to and from the record, and to query the value of the account at any point during the history. Show how to satisfy all of these operations, again in expected time $O(\log n)$, where $n$ is the total number of operations.

5. **The Trippy Mind Meld** (30 pts)

In this problem we consider the problem of taking two treaps $T_1$ and $T_2$ and forming a single treap, whose keys are precisely the union of the sets of keys of $T_1$ and $T_2$, with the same priority for each key in the new treap as in this element’s initial treap. We assume that all the keys in $T_1$ are strictly smaller than the keys in $T_2$, and that all the priorities of both $T_1$ and $T_2$ are chosen independently. Let $n_1$ and $n_2$ be the number of keys in $T_1$ and $T_2$, respectively.

a) Write recursive pseudo-code to meld $T_1$ and $T_2$, say, $Meld(T_1, T_2)$. Feel free to use the following functions, and to draw pictures:

- $\text{root}(T) =$ the root of $T$.
- $\text{p}(\text{key } k) =$ the priority of key $k$.
- $\text{left}(T) =$ the left subtree of $T$.
- $\text{right}(T) =$ the right subtree of $T$.

b) Now we want to analyze the expected runtime for this merge. As a starting point, give an asymptotic formula for the expected length of the right-most path in $T_1$ (i.e. the path down the tree starting at the root and following right children until reaching either a leaf or a node with only a left child). You should use the following random variable:

$$V_i = \begin{cases} 
1 & \text{if key } k_i \text{ is on the right-most path of } T_1, \\
0 & \text{otherwise}
\end{cases}$$

(Assume the keys in $T_1$ are denoted $k_1, k_2, \ldots, k_{n_1}$.) Argue that a similar bound also holds for the expected length of the left-most path.

c) Use your answer to Part 5b to prove an upper bound of $O(\log n_1 + \log n_2)$ on the number of priority comparisons, and for the overall running for your meld operation.