Required Problems

1. **Tail Inequalities**
   **[15 Points]**
   (a) **[10 Points]** Prove the following theorem:
   \[
   \text{Theorem 0.1} \quad \text{Let } X_1, X_2, \ldots, X_n \text{ be independent coin flips such that for } 1 \leq i \leq n, \text{ we have } \Pr[X_i = 1] = p_i, \text{ where } 0 < p_i < 1. \text{ Then, for } X = \sum_{i=1}^{n} X_i, \mu = E[X] = \sum_{i} p_i \text{ and for any } \delta > 0, \]
   \[
   \Pr[X > (1 + \delta)\mu] < \left[ \frac{e^{\delta}}{(1 + \delta)^{(1+\delta)}} \right]^\mu,
   \]
   and
   \[
   \Pr[X < (1 - \delta)\mu] < \exp\left( -\frac{\mu\delta^2}{2} \right).
   \]
   (b) **[5 Points]** Consider a collection of \( n \) random variables \( X_i \) drawn independently from the geometric distribution with mean \( 2 \) – that is, \( X_i \) is the number of flips of an unbiased coin up to and including the first head. Let \( X = \sum X_i \). Derive an upper bound as small as possible on the probability that \( X > (1 + \delta)(2n) \) for any fixed \( \delta \).

2. **Tournament without a winner**
   **[10 Points]**
   Consider a tournament on \( n \) teams, in which each pair of teams play against each other once and a winner is always declared. Suppose we try to rank the teams in some total order based on the outcome of the tournament. Say that a game agrees with the ranking we have
chosen if the team we ranked better won. Prove that for sufficiently large $n$, there is a possible set of outcomes such that no ranking agrees with more than 51% of the games. (Hint: Pick the winner in a game randomly and use the results of exercise 1 above.)

3. **Fuzzy Sorting of Intervals** (Based on CLRS 7-6)  
**[10 Points]**
Consider a sorting problem in which the numbers are not known exactly. Instead, for each number, we know an interval on the real line to which it belongs. That is, we are given $n$ closed intervals of the form $[a_i, b_i]$, where $a_i \leq b_i$. The goal is to fuzzy-sort these intervals, i.e., produce a permutation $< i_1, i_2, \ldots, i_n >$ of the intervals such that there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \cdots \leq c_n$.

(a) **[5 Points]** Design an algorithm for fuzzy-sorting $n$ intervals. Your algorithm should have the general structure of an algorithm that quicksort the left endpoints (the $a_i$’s), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals gets easier and easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.)

(b) **[5 Points]** Argue that your algorithm runs in expected time $\Theta(n \log n)$ in general, but runs in expected time $\Theta(n)$ when all of the intervals overlap (i.e., when there exists a value $x$ such that $x \in [a_i, b_i]$ for all $i$). Your algorithm should not be checking for this case explicitly; rather, its performance should naturally improve as the amount of overlap increases.

4. **Approx Max Cut**  
**[5 Points]**
Given a graph $G = (V, E)$ with $n$ vertices and $m$ edges, describe an algorithm that runs in $O(n)$ times, and output a cut $S \subseteq V$, such that the expected number of edges in the cut is $\geq M/2$, where $M$ is the number of edges in the maximum cut, where the number of edges in the cut is $|(S \times (V \setminus S)) \cap E|$.

5. **Modified Partition** (Based on CLRS 7.4-6)  
**[5 Points]**
Consider modifying the Partition procedure by randomly picking three elements from array $A$ and partitioning about their median. Approximate the probability of getting at worst an $\alpha$-to-$$(1 - \alpha)$ split, as a function of $\alpha$ in the range $0 < \alpha < 1$.

6. **Sorting Random Input**  
**[10 Points]**
Let $a_1, \ldots, a_n$ be $n$ real numbers chosen independently and uniformly from the range $[0, 1]$.

- **[5 Points]** Describe an algorithm with an expected linear running time that sorts the numbers.
- **[5 Points]** Show that the linear running time is with high probability.
7. Estimating quantities

[10 Points]

(a) [5 Points] Assume that you are given a function $\text{RandBit}$ that returns a truly random bit. However, you do not know what is the probability $p$ that $\text{RandBit}$ returns 1. Describe an algorithm (and prove its correctness), as fast as possible, that receives parameters $\varepsilon, \delta$, and outputs a number $x$, such that with probability $\geq 1 - \delta$ we have $p \leq x \leq p + \varepsilon$. Namely, the program estimates the value of $p$ “reliably”.

(b) [5 Points] Let $G = (V, E)$ be a graph with $n$ vertices and $m$ edges. Assume that the only way you can know whether there is an edge between vertices $u$ and $v$ is to probe the graph $G$ and ask whether there is an edge $uv \in E$ in constant time. You are given parameters $\varepsilon > 0$ and $\delta > 0$. Describe an algorithm, as fast as possible, that outputs a number $k$ which is a good estimate of the number of edges of $G$. Namely, such that $m \leq k \leq m + \varepsilon n^2$ with probability larger than $1 - \delta$.

8. (Extra credit question) [10 Points]

Provide a subquadratic ($o(n^2)$ time) deterministic algorithm for the nuts and bolts matching problem. Your solution should be self-contained.
Practice Problems

1. [20 points] Let’s analyze the number of random bits needed to implement the operations of a treap. Suppose we pick a priority $p_i$ at random from the unit interval. Then the binary representation of each $p_i$ can be generated as a potentially infinite series of bits that are the outcome of unbiased coin flips. The idea is to generate only as many bits in this sequence as is necessary for resolving comparisons between different priorities. Suppose we have only generated some prefixes of the binary representations of the priorities of the elements in the treap $T$. Now, while inserting an item $y$, we compare its priority $p_y$ to other’s priorities to determine how $y$ should be rotated. While comparing $p_y$ to some $p_i$, if their current partial binary representation can resolve the comparison, then we are done. Otherwise, the have the same partial binary representations (upto the length of the shorter of the two) and we keep generating more bits for each until they first differ.

   (a) Compute a tight upper bound on the expected number of coin flips or random bits needed for a single priority comparison. (Note that during insertion every time we decide whether or not to perform a rotation, we perform a priority comparison. We are interested in the number of bits generated in such a single comparison.)

   (b) Generating bits one at a time like this is probably a bad idea in practice. Give a more practical scheme that generates the priorities in advance, using a small number of random bits, given an upper bound $n$ on the treap size. Describe a scheme that works correctly with probability $\geq 1 - n^{-c}$, where $c$ is a prespecified constant.

2. [20 points] Consider a uniform rooted tree of height $h$ (every leaf is at distance $h$ from the root). The root, as well as any internal node, has 3 children. Each leaf has a boolean value associated with it. Each internal node returns the value returned by the majority of its children. The evaluation problem consists of determining the value of the root; at each step, an algorithm can choose one leaf whose value it wishes to

   (a) [5 points] Describe a deterministic algorithm that runs in $O(n)$ time, that computes the value of the tree, where $n = 3^h$.

   (b) [10 points] Consider the recursive randomized algorithm that evaluates two subtrees of the root chosen at random. If the values returned disagree, it proceeds to evaluate the third sub-tree. Show the expected number of leaves read by the algorithm on any instance is at most $n^{0.9}$.

   (c) [5 points] Show that for any deterministic algorithm, there is an instance (a set of boolean values for the leaves) that forces it to read all $n = 3^h$ leaves. (hint: Consider an adversary argument, where you provide the algorithm with the minimal amount of information as it request bits from you. In particular, one can devise such an adversary algorithm.).

3. [20 points] Assume that you are given a data-structure $DS$ (i.e., a black box) so that given $n$ points $p_1, \ldots, p_n$ in $\mathbb{R}^d$ one can build it in $T(n)$ time a data-structure that supports nearest-neighbor search in $Q(n)$ time. Namely, given a query point $q \in \mathbb{R}^d$, one can compute the point $p_i$ which is nearest to $q$ among all the points $p_1, \ldots, p_n$. Formally, $\text{dist}(p_i, q) \leq \text{dist}(p_j, q)$ for $j = 1, \ldots, n$.

   Furthermore, assume that one can delete a point from $DS$ in $D(n)$ time (namely, this point would no longer be considered in answering nearest-neighbor queries). Deleting a point that does not exist in the data-structure is allowed, and does not do anything.
Describe how to construct a data-structure that support insertions (i.e., you are allowd to use the black-box described above as a building block in the new data-structure). The new data-structure should have the following performance:

- Build an empty data-structure in $O(1)$ time.
- Insertion takes $O((T(n)/n) \log n)$ amortized time.
- Deletion takes $O(D(n) \log n)$ time.
- Nearest neighbor query takes $O(Q(n) \log n)$ time.

Here $n$ is the overall number of insertions/deletions performed. (Hint: Maintain several data-structures $DS_1, \ldots, DS_k$ and simulate insertions by performing rebuilds.). Show that if $T(n) = n^2$ then the amortized insertion time is in fact $O(T(n)/n)$.

4. **[20 points]** You are given two sorted arrays of real numbers $A[1..n], B[1..n]$ (say, sorted in increasing order). Consider the set $C = \{C_{ij}\}$ of $n^2$ numbers that can be represented as $C_{ij} = A[i] + B[j]$ (to make things simple, assume that all those $n^2$ numbers are different), for $i = 1, \ldots, n$ and $j = 1, \ldots, n$.

(a) (3 point) Describe an $O(n^2)$ time algorithm that receives $k$, and return the $k$-th smallest element in $C$.

(b) (15 points) Describe an $O(n \log^2 n)$ expected time algorithm that receives $k$, and return the $k$-th smallest element in $C$. (Hint: consider $C$ to be written as an implicit matrix of size $n \times n$, and observe that this matrix has a lot of useful properties).

(c) (2 points) Describe an $O(n \log n)$ expected time algorithm that receives $k$, and return the $k$-th smallest element in $C$.

5. **[10 points]**

(a) Given a binary search tree $T$ with $n$ nodes with values stored in each node, describe an algorithm that prints all the values of $T$ from smallest to largest in $O(n)$ time.

(b) Describe such an algorithm that uses only $O(1)$ space, assuming that each node in $T$ has a pointer to its parent.

(c) Describe an algorithm that receives a tree $T$ and outputs a balanced binary tree $T'$ with the same values stored in it. The algorithm should work in linear time.

6. **[10 points]** Show how to implement a queue with two ordinary stacks so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$

7. **[10 points]** *(This problem is required only for graduate students. No extra credit would be given for undergraduates submitting a solution for this question)*

Design a data structure to support the following two operations for a set $S$ of integers:

- `INSERT(S, x)` inserts $x$ into $S$.
- `DELETE-LARGER-HALF(S)` deletes the largest $\lceil S/2 \rceil$

Explain how to implement this data structure so that any sequence of $m$ operations runs in $O(m)$ time.
8. [10 points] [This problem is required only for graduate students. No extra credit would be given for undergraduates submitting a solution for this question]

You have to provide a data-structure that support the following operations:

(a) Create an empty set.
(b) Insert an integer number into a set.
(c) Delete an integer number from a set.
(d) Search - given an integer number decide if the number is inside the set.
(e) Merge two sets into a new set (the two old sets are destroyed, and can not be used any more).

Describe how to implement such a data-structure so that the price of each operation is $O(\log n)$ amortized. Namely, after performing a sequence of $n$ operations, the overall running time is $O(n \log n)$. (Remember to prove everything - bounds, correctness, etc.)

9. [10 points] [This problem is required only for graduate students. No extra credit would be given for undergraduates submitting a solution for this question] Consider an ordinary binary search tree augmented by adding to each node $x$ the field $\text{size}[x]$ giving the number of keys stored in the subtree rooted at $x$. Let $\alpha$ be a constant in the range $1/2 < \alpha < 1$. We say that a given node $x$ is $\alpha$-balanced if

$$\text{size}[\text{left}[x]] \leq \alpha \cdot \text{size}[x]$$

and

$$\text{size}[\text{right}[x]] \leq \alpha \cdot \text{size}[x]$$

The tree as a whole is $\alpha$-balanced if every node in the tree is $\alpha$-balanced.

Suppose that INSERT and DELETE are implemented as usual for an $n$-node binary search tree, except that after every such operation, if any node in the tree is no longer $\alpha$-balanced, then the subtree rooted at the highest such node in the tree is rebuilt so that it becomes 1/2-balanced. i.e., as balanced as it can be.

We shall analyze this rebuilding scheme using the potential method. For a node $x$ in a binary search tree $T$, we define

$$\Delta(x) = |\text{size}[\text{left}[x]] - \text{size}[\text{right}[x]]|$$

and we define the potential of $T$ as

$$\Phi(T) = c \cdot \Sigma \Delta(x) \text{ for all } x \text{ in } T \text{ such that } \Delta(x) \geq 2$$

where $c$ is a sufficiently large constant that depends on $\alpha$

(a) Argue that any binary search tree has non-negative potential and that a 1/2-balanced tree has potential 0.

(b) Suppose that $m$ units of potential can pay for rebuilding an $m$-node subtree. How large must $c$ be in terms of $\alpha$ in order for it to take $O(1)$ amortized time to rebuild a subtree that is not $\alpha$-balanced?
(c) Show that inserting a node into or deleting a node from an $n$-node $\alpha$-balanced tree costs $O(\log n)$ amortized time.

10. Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer $k$. Describe an algorithm to find the $k$th smallest element in the union of $A$ and $B$. (For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, our algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicates. For full credit, your algorithm should run in $\Theta(\log n)$ time. [Hint: First try to solve the special case $k = n$.]

11. Say that a binary search tree is augmented if every node $v$ also stores $|v|$, the size of its subtree.

(a) Show that a rotation in an augmented binary tree can be performed in constant time.

(b) Describe an algorithm $\text{SCAPEGOATSELECT}(k)$ that selects the $k$th smallest item in an augmented scapegoat tree in $O(\log n)$ worst-case time.

(c) Describe an algorithm $\text{SPLAYSELECT}(k)$ that selects the $k$th smallest item in an augmented splay tree in $O(\log n)$ amortized time.

(d) Describe an algorithm $\text{TREAPSELECT}(k)$ that selects the $k$th smallest item in an augmented treap in $O(\log n)$ expected time.

12. (a) Prove that only one subtree gets rebalanced in a scapegoat tree insertion.

(b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree. (Recall that $I(v) = \max\{0, |T| - |s| - 1\}$, where $T$ is the child of greater height and $s$ the child of lesser height, and $|v|$ is the number of nodes in subtree $v$. A perfectly balanced tree has two perfectly balanced subtrees, each with as close to half the nodes as possible.)

(c) Show that you can rebuild a fully balanced binary tree from an unbalanced tree in $O(n)$ time using only $O(\log n)$ additional memory.

13. Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do not use the potential method (it makes it much more difficult).

14. A multistack consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the $i$th stack $S_i$ can hold up to $3^i$ elements. Whenever a user attempts to push an element onto any full stack $S_i$, we first move all the elements in $S_i$ to stack $S_{i+1}$ to make room. But if $S_{i+1}$ is already full, we first move all its members to $S_{i+2}$, and so on. Moving a single element from one stack to the next takes $O(1)$ time.
Making room for one new element in a multistack.

(a) [1 point] In the worst case, how long does it take to push one more element onto a multistack containing \( n \) elements?

(b) [9 points] Prove that the amortized cost of a push operation is \( O(\log n) \), where \( n \) is the maximum number of elements in the multistack. You can use any method you like.

15. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with \( 4^n \) leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after \( 2n \) moves, when the token lands on a leaf. If the final leaf is black, you die; if it’s white, you will live forever. You move first, so Death gets the last turn.

You can decide whether it’s worth playing or not as follows. Imagine that the nodes at even levels (where it’s your turn) are \( \lor \) gates, the nodes at odd levels (where it’s Death’s turn) are \( \land \) gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for \( \text{True} \) and \( \text{False} \). If the output at the top of the tree is \( \text{True} \), then you can win and live forever! If the output at the top of the tree is \( \text{False} \), you should challenge Death to a game of Twister instead.

(a) (2 pts) Describe and analyze a deterministic algorithm to determine whether or not you can win. [Hint: This is easy!]
(b) (8 pts) Unfortunately, Death won’t let you even look at every node in the tree. Describe a randomized algorithm that determines whether you can win in $\Theta(3^n)$ expected time. [Hint: Consider the case $n = 1$.]

16. (a) Show that it is possible to transform any $n$-node binary search tree into any other $n$-node binary search tree using at most $2n - 2$ rotations.

(b) Use fewer than $2n - 2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n - 6$ rotations, and there are pairs of trees that are $2n - 10$ rotations apart. These are the best bounds known.

17. Faster Longest Increasing Subsequence (LIS)
   Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. [Hint: In the dynamic programming solution, you don’t really have to look back at all previous items. There was a practice problem on HW 1 that asked for an $O(n^2)$ algorithm for this. If you are having difficulty, look at the solution provided in the HW 1 solutions.]

18. Amortization
   (a) Modify the binary double-counter (see class notes Sept 12) to support a new operation $\text{SIGN}$, which determines whether the number being stored is positive, negative, or zero, in constant time. The amortized time to increment or decrement the counter should still be a constant.

   [Hint: Suppose $p$ is the number of significant bits in $P$, and $n$ is the number of significant bits in $N$. For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update $p$ and $n$ in $O(1)$ time.]

   (b) Do the same but now you can’t assume that $p$ and $n$ can be updated in $O(1)$ time.

19. Amortization
   Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of ‘fits’, where the $i$th least significant fit indicates whether the sum includes the $i$th Fibonacci number $F_i$. For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string. This is not the same representation as on Homework 0.]

20. Detecting overlap
   (a) You are given a list of ranges represented by min and max (e.g., $[1,3]$, $[4,5]$, $[4,9]$, $[6,8]$, $[7,10]$). Give an $O(n \log n)$-time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.

   (b) You are given a list of rectangles represented by min and max x- and y-coordinates. Give an $O(n \log n)$-time algorithm that decides whether or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). [Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.]
21. Comparison of Amortized Analysis Methods

A sequence of $n$ operations is performed on a data structure. The $i$th operation costs $i$ if $i$ is an exact power of 2, and 1 otherwise. That is operation $i$ costs $f(i)$, where:

$$f(i) = \begin{cases} 
  i, & i = 2^k, \\
  1, & \text{otherwise}
\end{cases}$$

Determine the amortized cost per operation using the following methods of analysis:

(a) Aggregate method
(b) Accounting method
(c) Potential method