Задача 1-1 (5 баллов). Пусть $t_r(I)$ обозначает время рандомизированного алгоритма на входе I при значениях случайных бит r. Для заданного множества входов $\mathcal{I}$ рассмотрим две "меры сложности":

$$A = \max_{I \in \mathcal{I}} E_r \left[t_r(I) \right], \qquad B = E_r \left[\max_{I \in \mathcal{I}} t_r(I) \right].$$

Можно ли утверждать, что одно из чисел A и B всегда не меньше другого?

Решение. Используя неравенство Йенсена, можно показать, что $B \geqslant A$, что следует из вогнутости функции $\max_{I \in \mathcal{I}} t_r(I)$ по случайным битам r. Вогнутость функции max в данном контексте следует из свойств функций максимума в пространстве случайных величин.

Поэтому ожидание максимума всегда не меньше максимума ожиданий.

Задача 1-2 (15 баллов). Рассмотрим произвольное (корректное) решающее дерево для задачи сортировки n ключей, в котором все листья являются достижимыми. Найдите точную (не асимптотическую!) нижнюю оценку на *наименьшую* из глубин листьев данного дерева, т.е. такую функцию $g(n)$, что в любом подобном дереве наименьшая глубина листа не меньше $g(n)$, и одновременно для любого n существует такое дерево, в котором наименьшая глубина листа равна $g(n)$.

Решение. Эта глубина соответствует минимальному числу сравнений, необходимых для сортировки ключей в наилучшем случае.

Каждое сравнение может определить порядок между двумя элементами, и для установления полного порядка среди n элементов необходимо как минимум $n - 1$ парных сравнений (то есть, нельзя меньше), что выступает глубиной минимального листа.

При этом, для любого набора из n ключей всегда найдется такое дерево, в котором существует путь, в котором каждое сравнение ведет в одну и ту же сторону (т.е., $\forall a, b \in \{a_1, ..., a_n\}$ сравнение $a < b$ либо только удовлетворяется, либо нет), при этом, сравнения не повторяются. Поэтому, для достижения отсортированного массива из n ключей выполняется $n - 1$ сравнений.

В результате, $g(n) = n - 1$; то есть минимальная глубина листа не меньше $n - 1$ и может быть равна $n - 1$.

Задача 1-3 (15 баллов). Рассмотрим задачу слияния двух упорядоченных последовательностей длины m и n, $m \geqslant 2n$. Докажите (используя модель решающего дерева) нижнюю оценку $\Omega(n \log \frac{m}{n})$ на количество сравнений, необходимых для решения данной задачи в худшем случае.

Решение. При слиянии массивов нам нужно выбрать n позиций из $m + n$ для элементов из B, так как остальные позиции будут заняты элементами из A. Число способов выбрать n позиций из $m + n$ равно C_{n+m}^{n}, что равно количеству различных способов размещения элементов A и B в результирующей последовательности, а соответственно, минимально возможное количество листьев корректного дерева.

Из двоичного ветвления решающего дерева следует, что глубина дерева не меньше $\log(C_{n+m}^n)$. Имея ввиду то, что $C_{n+m}^n \approx \frac{(n+m)^n}{n^n}$ по формуле Стирлинга, получаем, что

$$\Omega(\log(C_{n+m}^n)) = \Omega(\frac{(n+m)^n}{n^n}) = \Omega((\frac{m}{n})^n) = \Omega(n \log \frac{m}{n}).$$

Задача 1-4 (30 баллов). Рассмотрим алгоритмическую задачу с конечным множеством входов $\mathcal{I}$ и произвольное вероятностное распределение на нем. Рассмотрим также некоторое конечное семейство $\mathcal{A}$ детерминированных алгоритмов для ее решения и вероятностное распределение на нем. Обозначим через $f_A(I)$ время работы алгоритма A на входе I.

1. Докажите неравенство

$$\min_{A \in \mathcal{A}} E_I \left[f_A(I) \right] \leqslant \max_{I \in \mathcal{I}} E_A \left[f_A(I) \right],$$

где матожидания взяты относительно соответствующих распределений. Сформулируйте данное неравенство в терминах соотношения сложности в среднем (по входам) и рандомизированной сложности.

Решение. В силу дискретности распределений величин $I \in \mathcal{I}$ и $A \in \mathcal{A}$ получаем, что $E_I(f_A(I)) = \sum_{I \in \mathcal{I}} P(I) f_A(I)$, а также $E_A(f_A(I)) = \sum_{A \in \mathcal{A}} Q(A) f_A(I)$, с соответствующими таблицами распределения P и Q.

Значит, требуемое неравенство следует из очевидной цепочки сравнений:

$$\min_{A \in \mathcal{A}} E_I \left[f_A(I) \right] = \min_{A \in \mathcal{A}} \sum_{I \in \mathcal{I}} P(I) f_A(I) \leqslant \sum_{A \in \mathcal{A}} \sum_{I \in \mathcal{I}} P(I) Q(A) f_A(I) = E_{I,A}(f_A(I)),$$

а также

$$E_{I,A}(f_A(I)) = \sum_{I \in \mathcal{I}} \sum_{A \in \mathcal{A}} P(I) Q(A) f_A(I) \leqslant \max_{I \in \mathcal{I}} \sum_{A \in \mathcal{A}} Q(A) f_A(I) = \max_{I \in \mathcal{I}} E_A \left[f_A(I) \right].$$

Само же неравенство можно интерпретировать как то, что время, затрачиваемое лучшим в среднем (по входу) алгоритмом, всегда не больше, чем время, затрачиваемое на наихудший вход некоторого алгоритма "среднячка".

Это показывает связь между детерминированными алгоритмами с минимальной средней сложностью и рандомизированными алгоритмами с максимальной ожидаемой сложностью по всем входам.

2. Придумайте и сформулируйте определение *рандомизированного* алгоритма сортировки в модели решающих деревьев. Докажите в этой модели оценку $\Omega(n \log n)$ для сложности произвольного рандомизированного алгоритма сортировки, основанного на сравнении ключей.

Решение. Рандомизированность алгоритма сортировки в модели решающих деревьев можно интерпретировать как выбор случайных элементов сравнения за место детерминированных индексов элементов.

В ванильном детерминированном алгоритме сортировки в модели решающих деревьев из-за $n!$ количества перестановок и необходимости (для корректности) наличия соответствующих (как минимум) $n!$ листьев, получаем $\log_2 n!$ глубину дерева, которая асимптотически (по формуле Стирлинга) равна $n \log n$.

Рандомизированный алгоритм может давать разные деревья на одном и том же входе в зависимости от draw случайного числа, но ожидаемая глубина его выполнения всё равно ограничена снизу $\Omega(n \log n)$, поскольку даже при наличии случайности, алгоритм должен учитывать все возможные перестановки входов, что требует по меньшей мере $\log_2(n!)$ шагов.

Задача 1-5 (20 баллов). Предложите реализацию *стека* на основе (одного) массива, которая поддерживает операции добавления в конец и удаления из конца. Требуется, чтобы *емкость* (количество выделенных ячеек памяти) стека в любой момент времени отличалась от фактического размера не более чем в константу раз, а учетная сложность операций добавления в конец и удаления из конца была константной.

Решение. Будет использоваться динамический массив с изменением размера (увеличением/уменьшением) размера в константу раз (напр., в 2 раза) при достижении `size`'а `capacity` или при уменьшении `size`'а в константу раз (минус единица) от `capacity`.

Это позволяет обеспечить, чтобы емкость стека была не более чем в константу раз больше его фактического размера и чтобы учетная сложность операций добавления и удаления была $O(1)$. При этом, стоимость операций textttpush и `pop` имеют амортизированную сложность $O(1)$.

Для доказательства амортизированной стоимости операций `push` и `pop` в стеке с динамическим изменением размера используем **метод банкира** (accounting method). Каждой операции присваивается амортизированная стоимость 3 единицы, что покрывает её фактическую стоимость и накапливает "кредит"для более затратных операций изменения размера.

Если при добавлении (`push`) массив не заполнен, фактическая стоимость составляет 1, а оставшиеся 2 единицы идут в "кредит". Если массив заполнен, емкость удваивается и элементы копируются, что стоит n операций (где n — текущий размер). Однако накопленные $2n$ единиц кредита с предыдущих `push` покрывают затраты на удвоение.

Аналогично, при удалении (`pop`) амортизированная стоимость 3 покрывает 1 единицу на удаление и оставляет 2 в "кредит". Если размер уменьшается до capacity$/4$, емкость делится пополам и требуется n операций для копирования, что компенсируется накопленным кредитом $2n$.

Таким образом, амортизированная стоимость операций `push` и `pop` — 3, обеспечивая амортизированную сложность $O(1)$.

Задача 1-6 (20 баллов). Предложите реализацию *очереди* на основе (одного) массива, которая поддерживает операции добавления в конец и удаления из начала. Требуется, чтобы емкость очереди в любой момент времени отличалась от фактического размера не более чем в константу раз, а учетная сложность операций добавления в конец и удаления из начала была константной.

Решение. Также используем **метод банкира** (accounting method).

Для реализации очереди на основе одного массива, которая поддерживает добавление в конец (`enqueue`) и удаление из начала (`dequeue`), используем динамический массив с циклическим буфером и динамическим изменением размера.

Используем два указателя: `front`, указывающий на начало очереди, и `rear`, указывающий на следующий доступный для добавления элемент. Начальная емкость равна 1.

При добавлении (`enqueue`), если массив заполнен, удваиваем его емкость. Элементы копируем в новый массив так, чтобы `front` всегда был на позиции 0. Это требует n операций, где n — текущий размер очереди. Назначаем амортизированную стоимость добавления в 3 единицы, где 1 покрывает само добавление, а 2 идут в "кредит"на удвоение емкости в будущем. Накопленный кредит $2n$ с предыдущих добавлений покрывает затраты на копирование при увеличении размера.

При удалении (`dequeue`), если после удаления размер становится равным четверти емкости, уменьшаем емкость вдвое и копируем элементы, что требует n операций. Амортизированная стоимость удаления также равна 3. При обычном удалении используется 1 единица на перемещение `front`, а 2 идут в "кредит". Эти $2n$ единиц кредита покрывают затраты на уменьшение емкости, когда это необходимо.

Для циклического массива `enqueue` и `dequeue` могут обходить конец массива с помощью операции по модулю. Это позволяет эффективно использовать всю емкость.

Таким образом, амортизированная сложность операций `enqueue` и `dequeue` — $O(1)$, так как кредит с дешевых операций компенсирует редкие дорогостоящие операции изменения размера. Емкость очереди всегда остаётся в константное количество раз больше фактического размера.

Задача 1-7 (30 баллов). Покажите, как *деамортизировать* операции вставки в конец вектора, т.е. добиться того, чтобы операции добавления в конец и чтения элемента по индексу требовали $O(1)$ времени в *худшем* случае. Совет: по мере добавления новых элементов необходимо параллельно копировать уже имеющийся массив в массив увеличенного размера. Делать это следует с такой скоростью, чтобы в тот момент, когда меньший массив окажется заполнен, мы могли за время $O(1)$ выполнить переключение на новый массив.

Решение. Для деамортизации операций вставки в конец динамического массива и обеспечения того, чтобы добавление элемента и чтение по индексу имели сложность $O(1)$ в худшем случае, используем стратегию параллельного копирования.

1. Процесс начинается с аллокации 2х массивов с константными длинами k и $k * 2$, где $k >= 2$ и k - четное. Обозначим меньший массив за S, а больший за L.

2. В каждый момент времени имеем два массива с константной `capacity` n и $2n$ у S и L соответственно. Пользователь взаимодействует с S. Пока $\texttt{size}_S <$ половины $\texttt{capacity}_S$, делаем `push` лишь в S.

3. Как только $\texttt{size}_S$ достигает половины $\texttt{capacity}_S$, делаем такой же `push` в S. Создаем `index`$= 0$ на элементы массива S и делаем их `push` в массив L дважды, увеличивая `index`.

4. Когда S достигает своей $\texttt{capacity}_S$, "переключаем"пользователя на L, который к этому моменту имеет все элементы S и достиг половины $\texttt{capacity}_L$. Теперь $S := L$, старый S деаллоцируется и новый L с удвоенной $\texttt{capacity}_S$ аллоцируется. Здесь считаем, что операции аллокации и деаллокации памяти константны.

Благодаря этому процессу переключение между старым и новым массивом происходит плавно и не требует $O(n)$ времени в момент переключения. Таким образом, использование параллельного копирования позволяет деамортизировать вставку в конец вектора, обеспечивая гарантированное время $O(1)$ на каждую операцию добавления, даже в момент изменения емкости массива.