5) вероятностный анализ алгоритмов (сложность в среднем, сложность для почти всех входов), вероятностные алгоритмы и их анализ (проверка тождеств с помощью метода Монте – Карло, вероятностное округление), дерандомизация, вероятностные вычисления (вероятностная машина Тьюринга, полиномиальный вероятностный алгоритм проверки простоты числа);

Вероятностный анализ алгоритмов на примере оценки среднего времени работы алгоритмов сортировки и хэширования.

Говоря неформально, алгоритм (algorithm) — это любая корректно определенная вычислительная процедура, на вход (input) которой подается некоторая величина или набор величин, и результатом выполнения которой является выходная (output) величина или набор значений. Таким образом, алгоритм представляет собой последовательность вычислительных шагов, преобразующих входные величины в выходные. Алгоритм также можно рассматривать как инструмент, предназначенный для решения корректно поставленной вычислительной задачи (computational problem). В постановке задачи в общих чертах задаются отношения между входом и выходом. В алгоритме описывается конкретная вычислительная процедура, с помощью которой удается добиться выполнения указанных отношений.

Обычно мерой эффективности алгоритма является скорость и время, в течение которого он выдает результат.
Пусть по входным данным (входу) x алгоритм A вычисляет результат (выход) y. Обозначим функции затрат времени и памяти данного алгоритма A на входе x, как $C_A^T(x), \quad C_A^S(x)$.
Определение. Временной и пространственной сложностями алгоритма A называют функции числового аргумента

$$T_A(n) = \max_{\|x\|=n} C_A^T(x), \quad S_A(n) = \max_{\|x\|=n} C_A^S(x).$$

Более полно, каждая такая сложность имеет сложность в худшем случае.

Рассмотрим конечное множество $X_n = \{x : \|x\| = n\}$ входов размера n. Будем предполагать, что каждому $x \in X_n$ приписана некоторая вероятность $P_n(x)$. ($P_n(x) \in [0;1], \quad \sum_{x \in X_n} P_n(x) = 1$)

На заданном таким образом вероятностном пространстве затраты алгоритма A на входе x размера s (т.е. $C_A^T(x), \quad C_A^S(x)$) являются случайными величинами.
Определение.
Сложностью в среднем называют математическое ожидание соответствующей случайной величины:

$$\bar{T}_A(\cdot) = \sum_{x \in X_n} P_n(x) C_A^T(x) \quad \bar{S}_A(\cdot) = \sum_{x \in X_n} P_n(x) C_A^S(x)$$

Упражнение. Покажите, что для любого алгоритма A при любом распределении вероятностей на множестве входов $X_n = \{x : \|x\| = n\}$:

$$\bar{T}_A(n) \le T_A(n), \quad \bar{S}_A(n) \le S_A(n)$$

В качестве примера рассмотрим задачу, которая часто возникает на практике – задача сортировки.

Задача сортировки:
Имеется произвольный массив $\mathbf{a} = (a_1, ..., a_n)$. Требуется путем сравнений упорядочить этот массив таким образом, чтобы элементы расположились в порядке возрастания, то есть $a_{i_1} \leq, ..., \leq a_{i_n}$.

Рассмотрим алгоритмы для решения этой задачи и оценим их время работы. Заметим, что для вычисления сложности в среднем потребуется вероятностное пространство на множестве всех входов длины n.

Упражнение. Покажите, что для этой задачи можно перейти от бесконечного множества входов размера n, к конечному пространству перестановок длины n (обозначим его, как Π_n). Обычно предполагается, что на таком пространстве введено равномерное распределение, т.е. $\forall \mathbf{a} \in \Pi_n \;\; P(\mathbf{a}) = \frac{1}{n!}$

Указание. На выполнение сортировки с помощью сравнений сами значения элементов $a_1, ..., a_n$ не оказывают влияния, важен их относительный порядок. То есть алгоритмы не различают входные данные: -5, 10, -17 и 2, 3, 1.

Рассмотрим **простые схемы сортировки**.
Сортировка вставками.
На i-ом шаге вставляется элемент a_i в нужную позицию среди уже отсортированных предыдущих (i-1) элементов.

Псевдокод:
For i=2 to n do
Переместить a_i на позицию $j \leq i$:
$a_i < a_k$ для $j \leq k < i$ и
Либо $a_i \geq a_{j-1}$, либо j=1

Техническое замечание. Иногда полезно ввести элемент a_0, который меньше всех остальных. Как бы постулировать существование константы $-\infty$. Как будет выглядеть псевдокод в таком случае?

Задача. Оцените временную сложность работы алгоритма вставками.
Замечание. Считаем, что затраты алгоритма идут операцию сравнения, то есть требуется определить сложность алгоритма по числу перемещений. Можно также вводить сложность по числу перемещений. Последнее имеет смысл, если в качестве входных данных у нас, например, массив длинных последовательностей символов, соответственно операция перемещения такой последовательности на новое место достаточно затратная. Но на практике можно избежать сложностей, связанными с такими входными данными, работая с массивом указателей. (см. Техническое замечание ко второй задаче)
Указание. Несложно показать, что время работы в худшем случае, когда список отсортирован по убыванию, есть $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, т.е. $\Omega(n^2)$.
Покажем, что в среднем алгоритм работает также квадратичное время.
Введем случайные величины $\xi_n^1, \xi_n^2, ..., \xi_n^{n-1}$, определенные так, что значение ξ_n^i для $\mathbf{a} = (a_1, ..., a_n) \in \Pi_n$ равно затратам на i-ом шаге алгоритма сортировки вставками, применяемой к $\mathbf{a}$. Тогда интересующее нас математическое ожидание есть

$$\bar{T}(n) = E(\xi_n^1 + \xi_n^2 + ... + \xi_n^{n-1}) = E\xi_n^1 + E\xi_n^2 + ... + E\xi_n^{n-1}$$

Найдем $E\xi_n^i$, $1 \le i < n$. Рассмотрим события $H_n^{k,i+1}$, $k = 0,1,...,i$, состоящие в том, что в среди первых i элементов перестановки **a** (т.е. $a_1,...,a_i$) содержится ровно k элементов меньших a_{i+1}. Эти события образуют разложение пространства Π_n.Применяя формулу полного математического ожидания, получим

$$E\xi_n^i = \sum_{k=0}^{i} E(\xi_n^i \mid H_n^{k,i+1}) P(H_n^{k,i+1}) = \frac{1}{i+1} \sum_{k=0}^{i} E(\xi_n^i \mid H_n^{k,i+1}) = \frac{1}{i+1} \left(\sum_{k=1}^{i} (i-k+1) + i \right) = \frac{i}{2} + 1 - \frac{1}{i+1}.$$

Здесь учитывалось, что если перестановка принадлежит пространству $H_n^{k,i+1}$, то

$$\xi_n^i = \begin{cases} i-k+1, & k > 0; \\ i & k = 0. \end{cases}$$

Значит для искомой сложности в среднем, получаем:

$$\bar{T}(n) = n - 1 + \sum_{i=1}^{n-1} \left(\frac{i}{2} - \frac{1}{i+1} \right) = \frac{n^2}{4} + O(n).$$

Чем хорош рассмотренный алгоритм?
Хоть этот алгоритм требует квадратичного по размеру входного массива времени работы, он не требует дополнительной памяти, т.е. $S(n) = O(1)$ (требуется константная дополнительная память на то, чтобы поменять местами два элемента массива).

Рассмотрим еще один алгоритм сортировки – **посредством выбора**.
На i-ом этапе сортировки выбирается наименьший элемент из $a_i,...,a_n$ и меняется местами с a_i. В результате после i-го этапа все элементы $a_1,...,a_i$ будут отсортированы.
For i=1 to n-1 do

Выбрать среди элементов $a_i,...,a_n$ наименьший

Поменять его местами с a_i

Задача. Оценить время работы алгоритма сортировки посредством выбора.
Указание. В силу выбора наименьшего элемента в подмассиве $a_i,...,a_n$ алгоритму требуется выполнить (n-i) операций сравнения вне зависимости от входных данных. Поэтому время работы алгоритма есть $\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$.

Чем хорош этот алгоритм сортировки?
Существуют различные критерии эффективности алгоритмов. Применимость того или иного алгоритма зависит также от типа входных данных. Если мы имеем дело с символьным массивом, то операция перестановки элементов массива «трудоемка», поэтому нужно учитывать число перестановок алгоритма (как и говорилось в замечаниях к первой задаче). Рассмотренный алгоритм сортировки посредством выбора выполняет O(n) перестановок, в отличие от предыдущего алгоритма сортировки вставками (там $O(n^2)$). Причина: в алгоритме посредством выбора элементы «перескакивают» через большой ряд других элементов, вместо того, чтобы последовательно меняться с ними местами, как это делает сортировка вставками.

Техническое замечание. Вообще, если нужно упорядочить длинные записи, целесообразно работать не с массивом записей, а с массивом указателей на них, используя для сортировки любой подходящий алгоритм. В этом случае переставляются не записи, а их

указатели. После – за линейное время можно по массиву указателей восстановить отсортированные записи.

Предыдущие алгоритмы сортировки выполняются за квадратичное время как в среднем так и в худшем случае. Рассмотрим теперь алгоритмы, требующие выполнения $O(n \log n)$ времени. Однако следует заметить, что значение n, начиная с которого эти алгоритмы эффективнее простых методов сортировки, зависит от различных факторов (в частности от конкретной задачи и вычислительного окружения).

Рассмотрим алгоритм сортировки **слиянием**. В основе – процедура *merge*, которая в качестве входных данных использует два отсортированных массива (списка) меньшего размера, а на выходе получает объединенный отсортированный список. Процедура просматривает эти списки поэлементно, начиная с наименьших элементов. На каждом шаге наименьший элемент из двух сравниваемых (наименьшие элементы каждого из списков) удаляется из своего списка помещается в выходные данные. Псевдокод этого алгоритма:

```
Mergesort(A):
  If (|A| == 1)
     если список из одного элемента, то сортировать нечего
     Return A
  Else
          Разбиваем на два списка
      L = A[1..|A|/2]
      R = A[|A|/2+1..|A|]
      Return merge (mergesort(L), mergesort(R) );
End


Merge (L, R)
   If L[0] < R[0]
     Добавляем в суммарный список S элемент L[0] и удаляем его из  L
   Else
      Добавляем в суммарный список S элемент R[0] и удаляем его из  R
Return S
```

Задача. Оценить количество дополнительной памяти, требуемой алгоритму слиянием.
Указание. Дополнительная память требуется на этапах: слияния двух подмассивов в один результирующий и напротив – получения из одного массива два подмассива (списка). То есть линейная память от размера входа.

Задача. Оценить время работы этого алгоритма.
Этот алгоритм основан на принципе «разделяй и властвуй». Если алгоритм рекурсивно обращается к самому себе, время его работы часто описывается с помощью рекуррентного уравнения, или рекуррентного соотношения, в котором полное время, требуемое для решения всей задачи с объемом ввода n, выражается через время решения вспомогательных подзадач. Затем данное рекуррентное уравнение решается с помощью определенных математических методов, и устанавливаются границы производительности алгоритма.
Можно записать следующее рекуррентное неравенство, ограничивающее сверху T(n):

$$T(n) \le \begin{cases} c_1, & \text{если} \quad n = 1; \\ 2T\left(\frac{n}{2}\right) + c_2 n, & \text{если} \quad n > 1; \end{cases}$$

Константа c_1 соответствует фиксированному количеству шагов, выполняемых алгоритмом над списком длины 1 (обычно полагают $c_1 = 1$). Если $n > 1$, то время выполнения процедуры *mergesort* можно разбить на две части. Первая часть состоит из проверки, что $n \neq 1$, разбиения списка на две части (равные - при предположении, что n является степенью двойки) и вызова процедуры *merge*. Можно выбрать такую константу c_2, что время выполнения этой части процедуры будет ограничено величиной $c_2 n$. Вторая часть процедуры состоит из двух рекурсивных вызовов этой процедуры для списков длины $\frac{n}{2}$, которые требуют времени $2T\left(\frac{n}{2}\right)$.

Полученное соотношение для анализа времени выполнения алгоритма в худшем случае работает при предположении, что n является степенью двойки. Можно оценить $T(n)$ для любых n. Например, для практически всех алгоритмов можно предположить, что $T\left(2^i\right) < T(n) < T\left(2^{i+1}\right)$, где $2^i < n < 2^{i+1}$. Более того, можно показать, что для нечетных n можно заменить $2T\left(\frac{n}{2}\right)$ на $\frac{1}{2}\left(T\left(\frac{n+1}{2}\right) + T\left(\frac{n-1}{2}\right)\right)$.

Оценка решения рекуррентного соотношения методом подстановки:

$$T(n) \le 2T\left(\frac{n}{2}\right) + c_2 n \le 2\left(2T\left(\frac{n}{4}\right) + c_2 \frac{n}{2}\right) + c_2 n = 2T\left(\frac{n}{4}\right) + 2c_2 n \le \cdots \le 2^i T\left(\frac{n}{2^i}\right) + ic_2 n$$

Предположим, что n является степенью двойки $n = 2^k$. Тогда

$$T(n) \le 2^k T(1) + kc_2 n \le c_1 n + + c_2 n \log_2 n$$

То есть $T(n)$ имеет порядок роста не более, чем $O(n \log n)$ (см. приложение).

Несмотря на асимптотически оптимальное поведение алгоритма сортировки слиянием, на практике он используется редко, так как требует в процессе своей работы дополнительные массивы (см. предыдущую задачу), что редко допустимо в реальных приложениях. Следующий алгоритм, несмотря на то, что он имеет время работы в худшем случае хуже, чем рассмотренный, чаще используется, так как требует лишь O(1) дополнительной памяти.

Быстрая сортировка. Алгоритм также основан на парадигме «разделяй и властвуй». Здесь выбирается из элементов массива опорный элемент, относительно которого переупорядочиваются все остальные элементы. Желательно выбрать опорный элемент близким к значению медианы, чтобы он разбивал список на две примерно равные части. Переупорядочивание элементов относительно опорного, происходит так, что все переставленные элементы, лежащие левее опорного, меньше его, а те, что правее – больше или равны опорному. Далее процедура быстрой сортировки рекурсивно применяется к левому и правому списку для их упорядочивания по отдельности.

Псевдокод основной процедуры алгоритм, примененный к подмассиву A[p..r] выглядит следующим способом (здесь в качестве опорного элемента выбирается A[r]):

```
Quicksort (A, p, r)

1. if  p<r
2.       then q = Partition(A, p, r)
3.       Quicksort(A, p, q - 1)
```

4. Quicksort(A, q + 1, r)

Чтобы выполнить сортировку всего массива A, вызов процедуры должен иметь вид Quicksort(A, 1, |A|).

Процедура Partition (A, p, r) переупорядочивает элементы подмассива, не использую дополнительной памяти (имеется в виду, что дополнительная память есть O(1)):

Partition (A, p, r)

1. x = A[r] // Опорный элемент
2. i = p-1
3. for j = p to r - 1
4. if A[j] ≤ x
5. then i = i + 1
6. Обменять A[i] и A[j]
7. Обменять A[i + 1] и A[i]
8. return i + 1

Таким образом, процедура Partition поддерживает в подмассиве A [p..r]; четыре области: все элементы подмассива A [p..i] меньше либо равны опорному элементу,
все элементы подмассива A [i+ 1..j - 1] больше опорного, A[j..r-1]произвольные (непросмотренные элементы) и A[r] – опорный элемент.

Упражнение. Покажите, что процедура переупорядочивания имеет временную сложность, пропорциональную длине обрабатываемого списка.
Указание. Каждый элемент массива назначается опорным не более чем один раз.

Задача. Оценить время работы алгоритма быстрой сортировки в худшем случае.
Наихудшие входные данные для описанного алгоритма быстрой сортировки (предполагается, что в качестве опорного элемента выбирается последний элемент обрабатываемого массива) – элементы уже упорядоченные по возрастанию. На каждом шаге в этом случае процедура Partition порождает две подзадачи размера n-1 и 0 (опорный элемент не входит в новые подзадачи!).
Действительно, пусть T(n) время работы алгоритма в наихудшем случае. Тогда справедливо рекуррентное соотношение:

$$T(n) = \max_{0 \le q \le n-1} \{T(q) + T(n-1-q)\} + \Theta(n),$$

где $\Theta(n)$ - время, требуемое на процедуру переупорядочивания (Partition).

Замечание. $f(n) = \Theta(g(n)) \Leftrightarrow \exists c_1, c_2 : c_1 g(n) \le f(n) \le c_2 g(n)\ \forall n \ge N$

Для решения рекуррентного уравнения применим метод подстановки: $T(n) \le cn^2$.

$$T(n) \le c \max_{0 \le q \le n-1} \{q^2 + (n-1-q)^2\} + \Theta(n).$$

Так как выражение $q^2 + (n-1-q)^2$ достигает своего максимума на концах интервала $q = 0, q = n-1$, то $T(n) \le c(n-1)^2 + \Theta(n) = cn^2 - c(2n-1) + \Theta(n) \le cn^2$, поскольку константу c можно выбрать настолько большой, что слагаемое $c(2n-1)$ доминировало над $\Theta(n)$. Таким образом, для оценки времени работы алгоритма быстрой сортировки выполняется оценка $T(n) = O(n^2)$.

Аналогично можно показать, что $T(n)=\Omega(n^2)$ (т.е. $\exists c: T(n)\ge cn^2$) есть решение рекуррентного выражения $T(n)\le c\max\limits_{0\le q\le n-1}\left\{q^2+(n-1-q)^2\right\}+\Theta(n)$. Откуда следует, что $T(n)=\Theta(n^2)$ - оценка времени работы быстрой сортировки в худшем случае.

В «благоприятном» случае процедура Partition разделяет входной массив на два почти одинаковых подмассива. (с длинами $\left\lfloor\frac{n}{2}\right\rfloor$, $\left\lceil\frac{n}{2}\right\rceil-1$). Можно показать, что время работы алгоритма в этом случае есть $\Theta(n\log n)$.

Задача. Оценить время работы алгоритма быстрой сортировки в среднем.
(Ход рассуждений, как т в анализе в среднем времени работы сортировки посредством вставками.) Введем случайную величину ξ_n на Π_n, равную числу сравнений быстрой сортировки на $\mathbf{a}=(a_1,...,a_n)\in\Pi_n$. (В обозначениях, принятых в начале параграфа случайная величина есть $\xi_n=C^T(\mathbf{a})$.) В качестве разбиения вероятностного пространства, рассмотрим события K_n^i, $i=1,...,n$ - перестановки, для которых разбивающий элемент занимает i-ую позицию в упорядоченном массиве. Применим формулу полного математического ожидания:

$$\bar{T}(n)=E\xi_n=\sum_{i=1}^{n}E\left(\xi_n\mid K_n^i\right)P\left(K_n^i\right)=\frac{1}{n}\sum_{i=1}^{n}E\left(\xi_n\mid K_n^i\right)$$

При условии, что выполняется событие K_n^i, опорный элемент сравнивается с другими элементами массива, разбивая его на два подмассива длиной i и $n-i$ соответственно, к которым потом рекурсивно и применяется алгоритм:

$$E\left(\xi_n\mid K_n^i\right)=n-1+E\xi_{i-1}+E\xi_{n-i}$$

Таким образом, получаем:

$$\bar{T}(n)=\frac{1}{n}\sum_{i=1}^{n}\left[n-1+\bar{T}(i-1)+\bar{T}(n-i)\right]=n-1+\frac{2}{n}\sum_{i=0}^{n-1}\bar{T}(i)$$

Для того чтоб показать, что выполняется $\bar{T}(n)=O(n\log n)$, т.е. существует некоторая константа c, что $\bar{T}(n)\le cn\log n$ покажем, что:

$$\sum_{k=1}^{n-1}k\log k\le\log\frac{n}{2}\sum_{k=1}^{\left\lceil\frac{n}{2}\right\rceil-1}k+\log n\sum_{k=\left\lfloor\frac{n}{2}\right\rfloor}^{n-1}k=\log n\sum_{k=1}^{n-1}k-\sum_{k=1}^{\left\lceil\frac{n}{2}\right\rceil-1}k=$$

$$=\frac{n(n-1)}{2}\log n-\frac{\left\lfloor\frac{n}{2}\right\rfloor\left(\left\lceil\frac{n}{2}\right\rceil-1\right)}{2}\le\frac{1}{2}n^2\log n-\frac{n^2}{8}$$

Подстановкой в рекурсивное соотношение убеждаемся:

$$\bar{T}(n)\le n-1+\frac{2}{n}c\left[\frac{1}{2}n^2\log n-\frac{n^2}{8}\right]=cn\log n+(n-1)-c\frac{n}{4}\le cn\log n\le cn\log n$$

Выбором соответствующей константы, можно добиться, чтобы слагаемое $c\frac{n}{4}$ доминировало над $n-1$.

Упражнение. Покажите аналогично, что $\bar{T}(n)=\Omega(n\log n)$.

Таким образом, асимптотически точная оценка среднего времени работы алгоритма быстрой сортировки для принятого предположения о входных данных - $\Theta(n\log n)$.

См. приложение.

В предыдущих задачах предполагалось, что данные хранятся в массиве. Операции проверки принадлежности, удаление и добавление на такой структуре данных имеют сложность линейную по числу элементов в структуре. **Хэширование** – один из способов улучшить оценки для упомянутых операций в среднем (в худшем случае сложность будет также линейна по числу элементов).
Таким образом, еще одно из приложений вероятностного анализа в информатике и программировании - оценки работы хэширования.
Задача заключается в представлении некоторого множества элементов типа T, причем число их заведомо меньше n. Введем на множестве типа
T функцию h (хэш-функцию), принимающую значения $0,...,k-1$ ($k<n$). На такую функцию можно смотреть, как на способ свести вопрос о хранении одного большого множества к вопросу о хранении нескольких меньших (списков). В i-ом $0 \le i < k$ подмножестве хранятся те элементы t, для которых $h(t)=i$. Вопрос о проверке принадлежности, добавлении или удалении для большого множества сводится к такому же вопросу для одного из меньших. Эти меньшие множества удобно хранить с помощью ссылок; их суммарный размер равен числу элементов хэшируемого множества. Такую структуру данных будем называть хэш-таблицей. Идея – выбрать хэш-функцию такой, чтобы списки были почти одинаковые. Заранее не известно, какие значения будут хранить в хэш-таблице, но обычно можно выбрать хэш-функцию такой, чтобы значения $h(t)$ можно было считать случайной величиной, равномерно распределенной множестве $0,...,k-1$ и независимой от хэш-кодов других элементов.

Задача. Пусть в хэш-таблицу уже помещено n элементов. Оцените временную сложность поиска нового $(n+1)$-го элемента.
Если n элементов уже помещено в таблицу, то их взаимное расположение зависит только от соответствующих хэш-кодов $h_1, h_2, ..., h_n$. Все k^n возможных последовательности $h_1, h_2, ..., h_n$ считаются равновероятными; случайная величина X, равная числу операций сравнения при поиске элемента в таблице зависит только от этой последовательности.
Случай 1: поступивший элемент отсутствует в таблице
Вероятностное пространство, отвечающее данному случаю, состоит из k^{n+1} элементарных событий $\omega = (h_1, h_2, ..., h_n, h_{n+1})$, где h_j - хэш-код j-го элемента, а h_{n+1} - хэш-код нового элемента, которого не удалось найти. С учетом предположений о выборе хэш-функции $P(\omega) = \frac{1}{k^{n+1}}$ для любого такого ω. Безуспешный поиск требует по одной операции сравнения для каждого элемента списка h_{n+1}. То есть

$$X = [h_1 = h_{n+1}] + [h_2 = h_{n+1}] + ... + [h_n = h_{n+1}]$$

Так как $P\{[h_j = h_{n+1}]\} = \frac{1}{k}$ для $1 \le j \le n$, получим:

$$EX = E[h_1 = h_{n+1}] + E[h_2 = h_{n+1}] + ... + E[h_n = h_{n+1}] = \frac{n}{k}.$$

То есть сложность в среднем улучшилась в k раз, чем без хэширования.

Случай 2: поступивший элемент присутствует
Вероятностное пространство для случая успешного поиска уже другое, это множество элементарных событий вида $\omega = (h_1, h_2, ..., h_n; m)$, где m $(1 \le m \le n)$ - индекс искомого

элемента (h_m - его хэш-код). Если s_j - вероятность того, что ищется j-ый помещенный в таблицу элемент ($s_j = \frac{1}{n}$), то $P(\omega) = \frac{s_m}{k^n}$ для события $\omega = (h_1, h_2, ..., h_n; m)$. Заметим, что $\sum_{\omega} P(\omega) = \sum_{m=1}^{n} s_m = 1$, то есть действительно корректное распределение вероятностей. Случайная величина X в случае успешного поиска равна l, если новый элемент является l-ым в своем списке. Следовательно,

$$X = [h_1 = h_m] + [h_2 = h_m] + ... + [h_m = h_m].$$

Далее

$$EX = E[h_1 = h_m] + E[h_2 = h_m] + ... + E[h_m = h_m]$$

В отличие от 1 случая, здесь число слагаемых – случайная величина. Воспользуемся тождеством Вальда:

$$EX = E[h_j = h_m] E[m].$$

Так как $E[m] = \sum_{i=1}^{n} i s_i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}$, $E[h_j = h_m] = \frac{1}{k}$ $1 \le j \le n$, то $EX = \frac{n+1}{2k}$.

Таким образом, сложность поиска для хэширования, оценивается в среднем как $\Theta\left(\frac{n}{k}\right)$.

Рандомизированные алгоритмы.

Знание информации о распределении входных данных может помочь проанализировать поведение алгоритма в среднем случае. Однако часто встречаются ситуации, когда такими знаниями мы не располагаем, и анализ среднего поведения невозможен. Тогда есть возможность использовать рандомизированные алгоритмы.

Определение. Алгоритм с элементом случайности, реализуемый обращениями к генератору случайных чисел, называются рандомизированными.

Отличие вероятностного анализа (анализа работы в среднем детерминированного алгоритма) от рандомизированных алгоритмов заключается в том, что число шагов работы детерминированного алгоритма для любых конкретных входных данных фиксировано. В рандомизированном алгоритме число шагов зависит от результата рандомизации (генератора случайных чисел), то есть затраты алгоритма на конкретном входе теперь становятся случайными.

Замечание. Правомерен взгляд на рандомизированный алгоритмы, при котором каждому возможному входу сопоставляется вероятностное пространство обычных детерминированных алгоритмов.

Задача. Покажите, что временная сложность рандомизированного алгоритма быстрой сортировки, где разбивающий элемент выбирается случайно, допускает оценку $O(n \log n)$.

Указание. Вывод аналогичен вероятностному анализу (оценки в среднем) для детерминированного алгоритма быстрой сортировки, только теперь усреднение проводится не по всем входным данным, а по всем возможным результатам работы генератора случайных чисел. Совпадению результатов можно дать объяснение: случайный выбор разбивающего элемента, предпринимаемый на каждом этапе сортировки, эквивалентен тому, что мы меняем порядок исходного массива случайным

образом и затем применяем «обычную» быструю сортировку. Однако не следует путать, что усреднения проводятся на различных вероятностных пространствах. Рандомизированную сортировку можно улучшить (уменьшить константу для $n\log n$), если разбивающий элемент определять путем случайного выбора без возвращения трех элементов массива и последующего определения второго по величине из этих трех.

Насколько хороша оценка среднего времени работы алгоритмов сортировки $O(n\log n)$?

Задача. Покажите, что функция $\log_2 n!$ является нижней границей сложности в среднем класса алгоритмов сортировки посредством *сравнений*.

Определим дерево решений алгоритма. Узлам дерева соответствует программное «состояние» - наши знания об упорядочивании входных данных, которое можно получить на данном этапе программы. То есть, узлу соответствует несколько упорядочиваний исходных данных – истинный порядок не известен. В листьях – результат сортировки. Высота листа – число ребер в пути от корня к листу. Высота дерева – максимум высот его листьев. Любое дерево решений, описывающее алгоритм сортировки массива длины n, должно иметь n! листьев. Высота двоичного дерева с n! листьями равна $\log n!$. Использую формулу Стирлинга, получим, что сортировка посредством сравнений требует в худшем случае времени $\Omega(n\log n)$.

Покажите, что для произвольного двоичного дерева с k листьями средняя глубина листьев не меньше $\log k$.

Указание. Если после корня дерево разбивается на два поддерева с m и k-m листьями, то средней глубины исходного дерева получим соотношение

$$\frac{m}{k}\log m+\frac{k-m}{k}\log(k-m)+1=\frac{m\log 2m+(k-m)\log 2(k-m)}{k}$$

Покажите, что минимум достигается при $m=\frac{k}{2}$ и равен $\log k$.

Задача поиска k-ой порядковой статистики.

Рекурсивное применение процедуры, основанной на методе быстрой сортировки, позволяет быстро (в среднем) находить k-ую порядковую статистику. Задача вычисления порядковых статистик состоит в следующем: дан список (массив) из n чисел, необходимо найти значение, которое стоит в k-ой позиции в отсортированном в возрастающем порядке списке.

Замечание. В некоторых случаях решение задачи вычисления порядковых статистик находится за линейное время. Например, нахождение минимального (максимального) элемента требует времени порядка $O(n)$. Достаточно пройтись по всему массиву, обновляя рекордное значение экстремума.

Модифицируем алгоритм быстрой сортировки:

Выбираем опорный элемент

Делим список на две группы. В первой – элементы меньше опорного, во второй – больше либо равны.

Если размер (число элементов) первой группы больше либо равен k, то к ней снова применяется эта процедура. Иначе – вызывается процедура для второй группы.

Так же, как и метод быстрой сортировки, это решение задачи поиска порядковой статистики в самом худшем случае потребует времени не менее $\Omega(n^2)$. Но в среднем процедура работает значительно быстрее, чем быстрая сортировка, а именно - $O(n)$. Принципиальная разница между этими алгоритмами заключается в том, что когда процедура быстрой сортировки вызывается два раза, процедура для задачи поиска порядковой статистики вызывается только один раз.

Покажите, используя ту же технику, что и при анализе в среднем алгоритма быстрой сортировки, что среднее время работы такого алгоритма линейно.

Указание.
Покажите, что выполняется оценка среднего времени работы алгоритма: $E[T(n)] \le E\left\{\sum_{k=1}^{n} X_k \left[T\left(\max(k-1, n-k)\right) + O(n)\right]\right\} \le \frac{2}{n} \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} E[T(k)] + O(n)$.

Где учитывается, что

$$\max(k-1, n-k) = \begin{cases} k-1, k \ge \left\lceil \frac{n}{2} \right\rceil, \\ n-k, k < \left\lceil \frac{n}{2} \right\rceil \end{cases}$$

$\sum_{k=1}^{n} T(\max(k-1, n-k)) = 2 \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} T(k)$. См. приложение.

Приложение к рекуррентным соотношениям.

Теорема. Пусть $a \ge 1$ и $b > 1$- константы, $f(n)$ - произвольная функция, $T(n)$ - функция, определенная на множестве неотрицательных целых чисел с помощью рекуррентного соотношения: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

где выражение $\frac{n}{b}$ интерпретируется либо как $\left\lfloor \frac{n}{b} \right\rfloor$, либо как $\left\lceil \frac{n}{b} \right\rceil$. Тогда асимптотическое поведение функции $T(n)$ можно выразить следующим образом.

1. Если $f(n) = O(n^{\log_b a - \varepsilon})$ для некоторой константы $\varepsilon > 0$, то $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. Если $f(n) = \Theta\left(n^{\log_b a}\right)$, то $T(n) = \Theta\left(n^{\log_a b} \lg n\right)$.
3. Если $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для некоторой константы $\varepsilon > 0$, и для некоторой константы $c < 1$ и достаточно больших n выполнено: $af\left(\frac{n}{b}\right) \le cf(n)$, то $T(n) = \Theta\left(f(n)\right)$.

Вероятностные алгоритмы.

Рассмотрим на примере задач разрешения (в некоторой литературе употребляется термин - распознавания), каковые, по определению могут иметь только два ответа:«0» или «1» - «нет» или «да», как применение вероятностных алгоритмов уменьшает временные затраты на решение задачи.

Вначале введем следующую терминологию для классов сложностей алгоритмов.

RP (Random Polynomial time):

Если $x \notin L$ с вероятностью 1 алгоритм выдает $x \notin L$.

Если $x \in L$ с вероятностью большей $\frac{1}{2}$ алгоритм выдает $x \in L$.

co-RP :

Если $x \in L$ с вероятностью 1 алгоритм выдает $x \in L$.

Если $x \notin L$ с вероятностью большей $\frac{1}{2}$ алгоритм выдает $x \notin L$.

ZPP=RP $\cap$ co-RP **(zero-error probability polynomial time)**:

Безошибочно распознают x, работают в среднем за полиномиальное время.

BPP (bounded-error probability polynomial time):

Если $x \in L$ с вероятностью не меньшей, чем $\frac{3}{4}$, алгоритм выдает $x \in L$.

Если $x \notin L$ с вероятностью не большей, чем $\frac{1}{4}$, алгоритм выдает $x \in L$.

Классическим примером вероятностного алгоритма является алгоритм Фрейвалда для **задачи проверки перемножения матриц**.

Даны три матрицы A, B, C размера $n \times n$. Требуется проверить равенство $AB = C$.

Простой детерминированный алгоритм перемножает матрицы A, B и сравнивает результат с C. Время работы такого алгоритма при использовании обычного перемножения матриц составляет $O(n^3)$, при использовании быстрого - $O(n^{2,376})$. Вероятностный алгоритм Фрейвалда с односторонней ошибкой проверяет равенство за время $O(n^2)$.

Описание вероятностного алгоритма типа Монте-Карло проверки равенства AB = C:

1. взять случайный вектор $x \in \{0,1\}^n$
2. вычислить $y = Bx$
3. вычислить $z = Ay$
4. вычислить $t = Cx$
5. если $z = t$ вернуть «да», иначе «нет».

Иными словам, алгоритм проверяет выполнение равенства $A(Bx) = Cx$.

Лемма. Покажем**,** что для предъявленного алгоритма выполняется

$P\{\text{"да"} \mid AB = C\} = 1,$

$P\{\text{"нет"} \mid AB \neq C\} \geq 1/2.$

То есть вероятность ошибки не более $1/2$, при чем алгоритм ошибается только в случае $AB \neq C$ (co-RP).

Доказательство.

Если $AB = C$, то из описания алгоритм вернет «да».

Рассмотрим случай $AB \neq C$.

Алгоритм ошибается, если для x:

$(AB - C)x = 0$.

При этом матрица $AB - C$ ненулевая.

Обозначим ее за матрицу D. Пусть $d = (d_1, ..., d_n)$ - ненулевая строка матрицы D. Без ограничения общности можно считать, что $d_1 \neq 0$.

$$P\{Dx=0\} \le P\{d^T x=0\} = P\left\{x_1 = -\frac{\sum_{i=2}^{n} d_i x_i}{d_1}\right\}.$$

Последнее равно $1/2$, если $-\frac{\sum_{i=2}^{n} d_i x_i}{d_1} \in \{0,1\}$, иначе равно 0.

Лемма доказана.

Заметим, что вероятность ошибки $\left(\frac{1}{2}\right)$ можно уменьшить засчет времени решения задачи. Рассмотрим, что будет, если запустить алгоритм дважды. В качестве ответа на задачу возьмем конъюнкцию ответов обоих алгоритмов. Тогда в силу независимости их работы получим

$$P\{\text{"да"} \mid AB \ne C\} = P\{\text{"да" в первом алгоритме} \mid AB \ne C\} \cdot$$

$$\cdot P\{\text{"да во втором алгоритме"} \mid AB \ne C\} < \frac{1}{2}\frac{1}{2} = \frac{1}{4}.$$

Примененный метод называется «вероятностным усилением» или «вероятностной амплификацией» (от amplification).

Отметим, что рассмотренная задача является задачей разрешения, то есть требуемый на нее ответ есть «0» или «1» («нет» или «да»). Приведенный алгоритм является вероятностным алгоритмом с односторонней ошибкой.

Рассмотрим еще две задачи такого же типа.

Задача сравнения двух строк.

Требуется сравнить две битовые строки a, b, затратив как можно меньше информации. Основная идея – сравнивать не сами строки, а функции от них. Так сравниваются $a \bmod p$ и $b \bmod p$, для некоторого простого числа p. Для этого требуется передать $2\log_2 p$ бит информации.

Описание алгоритма сравнения строк:

1. Пусть $|a| = |b| = n$, $N = n^2 \log_2 n^2$
2. Выбираем случайное простое число p из интервала $[2..N]$
3. Выдать «да», если $a \bmod p = b \bmod p \Leftrightarrow (a-b) \equiv 0 \bmod p$, иначе выдать «нет».

Лемма. Этот алгоритм является вероятностным алгоритмом с односторонней ошибкой и вероятностью ошибки $O\left(1/n\right)$.То есть справедливо, что

$$P\{\text{"да"} \Leftrightarrow (a-b) \equiv 0, \bmod(p) \mid a = b\} = 1,$$

$$P\{\text{"да"} \Leftrightarrow (a-b) \equiv 0, \bmod(p) \mid a \ne b\} = O\left(1/n\right),$$

Т.е. класс co-RP. При этом необходимое количество переданных бит равно $O\left(\log_2 n\right)$.

Доказательство.

В основе доказательства лежит асимптотический закон распределения простых чисел:

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1,$$

где $\pi(n)$ - функция распределения простых чисел, равная количеству простых чисел, не превосходящих n.

Алгоритм может ошибаться, когда $a \neq b$. Число таких простых чисел p, при которых алгоритм ошибочно выдает "да" $\Leftrightarrow (a-b) \equiv 0 \bmod p$ не превосходит n.

Поэтому $P\{\text{"да"} \Leftrightarrow (a-b) \equiv 0 \bmod p \mid a \neq b\} \leq \dfrac{n}{N/\ln N} = \dfrac{n\left(\ln n^2 + \ln \log_2 n^2\right)}{n^2 \ln n^2} = O\left(\dfrac{1}{n}\right)$.

При этом число передаваемых битов оценивается, как

$$2\log_2 p \leq 2\log_2 N = 2\log_2\left(n^2 \log_2 n^2\right) = O(\log_2 n)$$

Лемма доказана.

Проверка простоты числа.

Согласно малой теореме Ферма, если N - простое число и целое a не делится на N, то

$$a^{N-1} \equiv 1 \bmod N \qquad (*)$$

Проверку $(*)$ для некоторого a называют тестом Ферма. Тест Ферма для задачи проверки простоты числа работает с гарантией только в одном направлении. Если при каком-то a сравнение $(*)$ нарушается, то можно утверждать, что N - составное. Вопрос только в том, как найти для составного N целое a, не удовлетворяющее $(*)$. Можно, например, пытаться найти необходимое число a, испытывая все целые числа, начиная с 2. Или попробовать выбирать эти числа случайным образом на отрезке $1 < a < N$. К сожалению, такой подход не всегда дает то, что хотелось бы. Имеются составные числа N, обладающие свойством $(*)$ для любого целого a с условием $(a, N) = 1$ (a и N - взаимно простые). Такие числа называются числами Кармайкла.

Рассмотрим, например число $561 = 3 \times 11 \times 17$. Так как 560 делится на каждое из чисел $2, 10, 16$, то с помощью малой теоремы Ферма легко проверить, что 561 есть число Кармайкла. Недавно была решена проблема о бесконечности множества таких чисел. Тест Ферма, как тест на простоту имеет фундаментальный изъян, при чем не такой уж маленький: чисел Кармайкла, меньших 10^{17}, более полумиллиона.

В 1976г. Миллер предложил заменить проверку $(*)$ проверкой несколько иного условия. Если N - простое число, то $N - 1 = 2^s t$, где t нечетно, то согласно малой теоремы Ферма для каждого a с условием $(a, N) = 1$ хотя бы одна из скобок в произведении

$$\left(a^t - 1\right)\left(a^t + 1\right)\left(a^{2t} + 1\right) \times \ldots \times \left(a^{2^{s-1}t} + 1\right) = a^{N-1} - 1$$

делится на N. Обращение этого свойства можно использовать, чтобы отличать составные числа от простых.

Пусть N - нечетное составное число, $N - 1 = 2^s t$, где t нечетно. Назовем целое число a, $1 < a < N$ «хорошим» для N, если нарушается одно из двух условий:

I) N не делится на a

II) $a^t \equiv 1 \bmod N$ или существует целое k, $0 \leq k < s$ такое, что $a^{2^k t} \equiv -1 \bmod N$.

Ясно, что для простого числа N не существует «хороших» чисел a. Если же N составное число, то согласно теореме Рабина (ссылка??) их существует не менее $\frac{3}{4}(N-1)$. Таким образом, тест Рабина-Миллера (проверка условий I, II для некоторого a) также работает с гарантией только в одном направлении. Число N, не проходящее тест, - составное. В противном случае N может быть простым с вероятностью не большей $\frac{1}{4}$. Важно, что для такого теста нет аналогов чисел Кармайкла.

Теперь можно поострить вероятностный алгоритм, отличающий составные числа от простых.

Описание алгоритма проверки простоты числа:

1) Выберем случайным образом число a, $1 < a < N$ и проверим для этого числа указанные выше свойства I и II.
2) Если хотя бы одно из них нарушается, то число N составное.
3) Если выполнены оба условия I и II, возвращаемся к шагу 1)

Из сказанного выше следует, что составное число не будет определено как составное после однократного выполнения шагов 1)-3) с вероятностью не большей $\frac{1}{4}$. А вероятность не определить его после k повторений не превосходит $\left(\frac{1}{4}\right)^k$, т.е. убывает очень быстро.

То есть справедлива

Лемма. При однократном выполнении предъявленного алгоритма выполняется

$$P\{\text{"простое"} \mid N - \text{простое}\} = 1,$$

$$P\{\text{"составное"} \mid N - \text{составное}\} \geq 3/4.$$

Комментарии.

Задача возведения в степень имеет эффективное решение, основанное на двоичном представлении показателя степени.

Замечание.

Позже Миллер усилил результат:

Теорема Миллера. Если справедлива расширенная гипотеза Римана и N проходит тест при любом a: $1 < a < 2\log^2 N$, то N - простое.

То есть можно получить полиномиальный тест на простоту с нулевой вероятностью ошибки (ZBP) в предположении о справедливости расширенной гипотезы Римана.

Позже был открыт полиномиальный алгоритм AKS (Agrawal, Kayal, Saxena) проверки числа на простоту, не требующий каких-либо предположений. Основная идея опирается на следующий факт: натуральное N, при условии Í Î Ä$(N,a)=1$, является простым в том и только в том случае, когда $(x-a)^N \equiv x^N - a \mod N$. Далее необходимый перебор значительно сокращается после деления последнего тождества на некоторый полином вида $x^r - 1$. Переходят к проверке тождества $(x-a)^N \equiv x^N - a \pmod{N, x^r - 1}$.

Вероятностные алгоритмы.

Вначале будет рассказано о применении теории вероятностей к модному и перспективному направлению информатики – интерактивные доказательства (где речь

идет о решении дискретных задач двумя участниками: «Verifier» - «Prover»; «Алиса» - «Боб»; «Оракул» - «Вычислитель», - в рамках упорядоченного диалога).

Пример: Алисе известен изоморфизм φ графов G_0 и G_1. Но она посылает Бобу граф $H = \psi(G_0)$, либо $H = \psi(G_1)$, где ψ - некоторый другой изоморфизм, не равный φ. Боб бросает монетку и просит изоморфизм либо $H \sim G_0$, либо $H \sim G_1$. В первом случае Алиса посылает ψ, во втором - $\varphi\psi^{-1}$. Таких парий разыгрывается N штук.

Если φ - действительно изоморфизм $G_0 \sim G_1$, то все проверки Боба будут положительны. Если φ - блеф, то с вероятностью 2^{-N} хотя бы одна проверка обнаружит это (та проверка, в которой Боб попросил $H \sim G_1$, тогда как $H = \psi(G_0)$, или напротив Боб попросил $H \sim G_0$ в то время, как $H = \psi(G_1)$).

Интересно здесь то, что Алиса убедила Боба в $G_0 \sim G_1$ так и не огласив самого изоморфизма φ. Поэтому в криптографии этот факт находит применение: если φ - пароль, диалог можно вести даже в открытую, - что служит примером системы с нулевым разглашением.

Совокупность задач, которые могут быть решены за полиномиальное время с помощью интерактивных доказательств (взаимодействий), образует **класс IP.**

Другой класс задач, являющийся также разновидностью интерактивного взаимодействия, - **PCP (Probabilistic Checkable Proof System)**:

1) Система представляет собой тандем Вычислителя V с Оракулом P, который для любой индивидуальной задачи распознавания с описанием x и ответом «да» - располагает ее решением, доказательством $P(x)$. Точнее говоря, строчкой $P(x)$ доказательства.
2) Вычислитель осуществляет проверку нескольких битов строки $P(x)$ (Вычислитель ограничен в своих запросах фрагментами строки доказательства), и принимает доказательство, если задача x действительно имеет ответ «да».
3) Ложное доказательство (если решение задачи «нет») Вычислитель принимает с вероятностью не более $\frac{1}{2}$ (можно заменить любой константной $p \in (0;1)$, ибо повторением процедуры вероятность ошибки понижается сколь угодно).

Класс сложности обозначается $PCP[r(\cdot), q(\cdot)]$, где r и q целочисленные функции, если V «подбрасывает монетку» не более $r(|x|)$ раз и запрашивает строки $P(x)$ не более $q(|x|)$ бит.

Теперь рассмотрим, применение вероятностных приближенных алгоритмов для решения **NP-трудных задач** (предполагается выполнение гипотезы $P \neq NP$). Рассматривается класс перечислительных задач. Алгоритм для решения таких задач в качестве выхода выдает целое неотрицательное число, являющееся числом решения задачи.

Рассмотрим задачу **максимальная выполнимость (MAX-SAT)**

Даны m скобок конъюнктивной нормальной формы (КНФ) с n переменными. Найти значения переменных, максимизирующее число выполненных скобок.

0.5-приближение.

Следующее утверждение по существу дает приближенный вероятностный алгоритм решения.

Для любых m скобок существуют значения переменных, при которых выполнено не менее $\frac{m}{2}$ скобок.

Предположим, что каждой переменной приписаны значения 0 или 1 независимо и равновероятно. Для $1 \le i \le m$ пусть $Z_i = 1$, если i-ая скобка выполнена, и $Z_i = 0$ в противном случае.

Для каждой дизъюнкции (скобки) с k литералами (переменными или их отрицаниями) вероятность, что эта дизъюнкция не равна 1 при случайном приписывании значений переменных, равна 2^{-k}. Значит вероятность того, что скобка рана 1, есть $1-2^{-k} \ge \frac{1}{2}$ и математическое ожидание $EZ_i \ge \frac{1}{2}$. Отсюда математическое ожидание числа выполненных скобок (равных 1) равно $\sum_{i=1}^{m} EZ_i \ge \frac{m}{2}$. Это означает, что есть приписывание значений переменным, при котором $\sum_{i=1}^{m} Z_i \ge \frac{m}{2}$

Говорят, что вероятностный приближенный алгоритм гарантирует точность C, если для всех входов I

$$\frac{Em_A(I)}{m_0(I)} \ge C,$$

где $m_0(I)$ - оптимум, $m_A(I)$ - значение, найденное алгоритмом, и решается задача максимизации.

Таким образом, описанный приближенный вероятностный алгоритм для задачи максимальной выполнимости дает точность $\frac{1}{2}$.

0.63-приближение.

Можно подойти к этой задаче (максимальная выполнимость) по-другому, переформулировав ее в задачу целочисленного линейного программирования (ЦЛП). Каждой скобке C_j поставим в соответствие булеву переменную $z_j \in \{0,1\}$, которая равна 1, если скобка C_j выполнена; каждой входной переменной x_i сопоставляем переменную y_i, которая равна 1, если $x_i = 1$, и равна 0 в противном случае. Обозначим C_j^+ индексы переменных в скобке C_j, которые входят в нее без отрицания, а через C_j^- - множество индексов переменных, которые входят в скобку с отрицанием. .Тогда задача максимальной выполнимости эквивалентна следующей задача ЦЛП:

$$\sum_{j=1}^{m} z_j \to \max$$

$$\sum_{i \in C_j^+} y_i + \sum_{i \in C_j^+} (1 - y_i) \ge z_j \quad j = 1..m$$

$$y_i, \quad z_j \in \{0,1\} \quad \forall i = 1..n, \; j = 1..m$$

Рассмотрим и решим задачу линейной релаксации целочисленной программы:

$$\sum_{j=1}^{m} \hat{z}_j \to \max$$

$$\sum_{i \in C_j^+} \hat{y}_i + \sum_{i \in C_j^+} (1 - \hat{y}_i) \geq \hat{z}_j \quad j = 1..m$$

$$\hat{y}_i, \quad \hat{z}_j \in [0,1] \quad \forall i = 1..n, j = 1..m$$

Пусть $\hat{y}_i$, $\hat{z}_j$ - решение линейной релаксации. Ясно, что $\sum_{j=1}^{m} \hat{z}_j$ является верхней оценкой числа выполненных скобок для данной КНФ.

Рассмотрим вероятностный алгоритм решения задачи максимальной выполнимости, где каждая переменная y_i независимо принимает значения 0 или 1 уже не с равными вероятностями, а с вероятностью $\hat{y}_i$ принимает значение 1 (и 0 с вероятностью $1 - \hat{y}_i$). Такой метод называется вероятностным округлением.

Докажем следующее утверждение:

Пусть в скобке C_j имеется k литералов. Вероятность того, что она выполнена при вероятностном округлении, не менее $\left(1 - \left(1 - \frac{1}{k}\right)^k\right)\hat{z}_j$.

Доказательство. Без ограничения общности можно предположить, что все переменные в скобке входят в нее без отрицания. Пусть эта скобка имеет вид: $x_1 \vee \cdots \vee x_k$. Из ограничений линейной релаксации следует, что

$$\hat{y}_1 + \cdots + \hat{y}_k \geq \hat{z}_j$$

Скобка C_j остается невыполненной, только если каждая из переменных $\hat{y}_i$ округляется в 0. Поскольку каждая переменная округляется независимо, это происходит с вероятностью $\prod_{i=1}^{k}(1 - \hat{y}_i)$. Остается показать, что

$$1 - \prod_{i=1}^{k}(1 - \hat{y}_i) \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right)\hat{z}_j$$

Выражение в левой части достигает минимума при $\hat{y}_1 = \ldots = \hat{y}_k = \frac{\hat{z}_j}{k}$. Остается показать, что $1 - (1 - z)^k \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right)z$ для всех положительных целых k и $0 \leq z \leq 1$.

Поскольку $f(x) = 1 - \left(1 - \frac{x}{k}\right)^k$ - вогнутая функция, для доказательства того, что она не меньше линейной функции на отрезке, достаточно проверить это нестрогое неравенство на концах этого отрезка, т.е. в точках $x = 0$ и $x = 1$.

Использую тот факт, что $1 - \left(1 - \frac{1}{k}\right)^k \geq 1 - \frac{1}{e} > 0.63$ для всех положительных целых k, получаем, что справедлива следующая

Теорема Для произвольной КНФ среднее число скобок, выполненное при вероятностном округлении, не меньше $1 - \frac{1}{e} > 0.63$ от максимально возможного числа выполненных скобок.

0.75-приближение

Теперь опишем общую идею, которая позволит получить вероятностный приближенный алгоритм с точностью $\frac{3}{4}$.

Идея: на данном входе запускаем два алгоритма и выбираем лучшее из решений. В качестве двух алгоритмов рассматриваем:

1) округление каждой переменной независимо в 0 или 1 с вероятностью $\frac{1}{2}$;

2) вероятностное округление решения линейной релаксации соответствующей задачи ЦЛП.

Пусть n_1 - математическое ожидание числа выполненных скобок для первого алгоритма, и n_2 - математическое ожидание числа выполненных скобок для второго алгоритма.

Теорема.

$$\max\{n_1, n_2\} \geq \frac{3}{4}\sum_j \hat{z}_j$$

Доказательство. Поскольку всегда $\max\{n_1, n_2\} \geq \frac{n_1 + n_2}{2}$, достаточно показать, что $\frac{n_1 + n_2}{2} \geq \frac{3}{4}\sum_j \hat{z}_j$. Пусть S_k обозначает множество скобок, содержащих ровно k литералов, тогда

$$n_1 = \sum_k \sum_{C_j \in S_k} (1 - 2^{-k}) \geq \sum_k \sum_{C_j \in S_k} (1 - 2^{-k})\hat{z}_j$$

Из предыдущей теоремы

$$n_2 \geq \sum_k \sum_{C_j \in S_k} \left(1 - \left(1 - \frac{1}{k}\right)^k\right)\hat{z}_j$$

Следовательно,

$$\frac{n_1 + n_2}{2} \geq \sum_k \sum_{C_j \in S_k} \frac{\left(1 - 2^{-k}\right) + \left(1 - \left(1 - \frac{1}{k}\right)^k\right)}{2}\hat{z}_j$$

Простое вычисление показывает, что $\left(1 - 2^{-k}\right) + \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \geq \frac{3}{2}$ для всех натуральных k и, значит,

$$\frac{n_1 + n_2}{2} \geq \frac{3}{4}\sum_k \sum_{C_j \in S_k} \hat{z}_j = \frac{3}{4}\sum_j \hat{z}_j$$

В следующем разделе будет описан 0.75-приближенный детерминированный алгоритм решения этой задачи.

Дерандомизация.

Оказывается, в некоторых случаях вероятностные алгоритмы могут быть «дерандомизированы», т.е. конвертированы в детерминированные алгоритмы. Один из общих методов, позволяющих сделать это, называется методом условных вероятностей.

Опишем этот подход на следующей задаче: имеется величина $X(\mathbf{x})$, где в булевом векторе $\mathbf{x}=(x_1,...,x_n)$ компоненты являются независимыми случайными величинами, причем $P\{x_i=1\}=p_i$, $P\{x_i=0\}=1-p_i$.

Так в задаче о максимальной выполнимости КНФ $X(x_1,...,x_n)$ равно числу невыполненных скобок в КНФ при вероятностном округлении.

Требуется найти булев вектор $\hat{\mathbf{x}}$, для которого выполнено неравенство

$$X(\hat{\mathbf{x}}) \le EX$$

Обозначим через $X(\mathbf{x} \mid x_1=d_1,...,x_k=d_k)$ новую случайную величину, которая получена их X фиксированием значений первых k булевых переменных.

Рассмотрим покомпонентную стратегию определения искомого вектора $\hat{\mathbf{x}}$. Для определения его первой компоненты вычисляем значения $f_0=EX(\mathbf{x} \mid x_1=0)$ и $f_1=EX(\mathbf{x} \mid x_1=1)$. Если $f_0<f_1$ полагаем $x_1=0$, иначе полагаем $x_1=1$. При определенной таким образом первой компоненты (обозначим ее d_1) вычисляем значение функционала $f_0=EX(\mathbf{x} \mid x_1=d_1,x_2=0)$ и $f_1=EX(\mathbf{x} \mid x_1=d_1,x_2=1)$. Если $f_0<f_1$ полагаем $x_2=0$, иначе полагаем $x_2=1$. Фиксируем вторую координату (обозначая ее d_2) и продолжаем описанный процесс до тех пор, пока не определится последняя компонента решения.

Найденный вектор будет удовлетворять требованию минимизации оценки математического ожидания. Рассмотрим первый шаг алгоритма. Имеем:

$$\begin{aligned}
EX &= P\{x_1=1\}\,EX(\mathbf{x} \mid x_1=1)+P\{x_1=0\}\,EX(\mathbf{x} \mid x_1=0)= \\
&= p_1 EX(\mathbf{x} \mid x_1=1)+(1-p_1)EX(\mathbf{x} \mid x_1=0)\ge \\
&\ge p_1 EX(\mathbf{x} \mid x_1=d_1)+(1-p_1)EX(\mathbf{x} \mid x_1=d_1)= \\
&= EX(\mathbf{x} \mid x_1=d_1)
\end{aligned}$$

Продолжая эту цепочку неравенств для каждого шага алгоритма, получаем на n-ом шаге:

$$EX \ge EX(\mathbf{x} \mid x_1=d_1,...,x_n=d_n)$$

Но $EX(\mathbf{x} \mid x_1=d_1,...,x_n=d_n)=X(\mathbf{x} \mid x_1=d_1,...,x_n=d_n)$.

Таким образом, изложенный общий метод позволяет осуществить «дерандомизацию», если есть эффективный алгоритм вычисления условных математических ожиданий (или условных вероятностей).

Применим метод условных вероятностей, к задаче максимальной выполнимости КНФ. Математическое ожидание числа невыполненных скобок равно:

$$EX=\sum_{j=1}^{m} P_j,$$

$$P_j = P\left\{\sum_{i\in C_j^+} x_i+\sum_{i\in C_j^-}(1-x_i)=0\right\}$$

Важный вопрос заключается в том, как эффективно вычислять условные математические ожидания. Предположим, что значения первых k переменных уже определены и I_0 - множество индексов тех переменных, значения которых равно 0, а I_1 - множество индексов тех переменных, значения которых равно 1. Если $I_0\cap C_j^-\ne\varnothing$ или $I_1\cap C_j^+\ne\varnothing$, то $P_j^k=P\left\{\sum_{i\in C_j^+} x_i+\sum_{i\in C_j^-}(1-x_i)=0 \mid x_1=d_1,...,x_k=d_k\right\}=0$. В противном случае

$$P_j^k=\prod_{i\in C_j^+\setminus I_0}(1-p_i)\prod_{i\in C_j^-\setminus I_1} p_i$$

Для полученного вектора $(d_1,...,d_n)$ выполнено $X\left(x_1=d_1,...,x_n=d_n\right)\le EX\le\frac{1}{e}m$

Таким образом, мы находим допустимый 0-1 вектор с гарантированной верхней оценкой для целевой функции.

Рассмотрим еще одну NP-трудную задачу.

Задача DNF-COUNT. $f(x_1,...,x_n)=C_1\vee\cdots\vee C_m$ - булева формула в дизъюнктивной нормальной форме (ДНФ), где каждая скобка C_i - есть конъюнкция $L_1\wedge\cdots\wedge L_{k_i}$ k_i литералов (литерал есть либо переменная, либо ее отрицание). Набор значений переменных $\mathbf{a}=(a_1,...,a_n)$ называется выполняющим для f, если $f(a_1,...,a_n)=1$. Найти число выполняющих наборов для данной ДНФ.

Рассмотрим алгоритм, основанный на стандартном методе Монте-Карло.
Пусть
V - множество всех двоичных наборов длины n.
G - множество выполняющих наборов.

1) Проведем N независимых испытаний:
   Выбираем случайно $v_i\in V$ (в соответствии с равномерным распределением)
   $y_i=f(v_i)$. Заметим, что $P\{y_i=1\}=\frac{|G|}{|V|}=p$
2) Рассмотрим сумму независимых случайных величин $Y=\sum_{i=1}^{N}y_i$. В качестве аппроксимации $|G|$ возьмем величину $\frac{Y}{N}|V|$.

Для дальнейшего анализ потребуется следующие неравенства больших уклонений:
Пусть $X_1,...,X_n$ - независимые случайные величины, принимающие значения 0 или 1, при этом $P\{X_i=1\}=p,\quad P\{X_i=0\}=1-p$. Тогда для $X=\sum_{i=1}^{N}X_i$ и для любого $0<\delta<1$, выполнены неравенства

$$P\{X>(1+\delta)EX\}\le e^{-\frac{\delta^2}{3}EX}$$

$$P\{X<(1-\delta)EX\}\le e^{-\frac{\delta^2}{2}EX}$$

Воспользовавшись этим утверждением, оценим качество аппроксимации решения в приведенном алгоритме Монте-Карло.

$$P\left\{(1-\delta)\frac{|G|}{|V|}N\le Y\le(1+\delta)\frac{|G|}{|V|}N\right\}>1-2e^{-\frac{\delta^2}{3}N\frac{|G|}{|V|}}$$

Потребуем, чтобы вероятность хорошей аппроксимации была не меньше $1-\varepsilon$, получим

$$2e^{-\frac{\delta^2}{3}N\frac{|G|}{|V|}}<\varepsilon$$

$$N>\frac{3}{\delta^2}\frac{|V|}{|G|}\ln\frac{2}{\varepsilon}$$

Правда, полученная оценка не столь хороша, так как $p = \frac{|G|}{|V|}$ может быть экспоненциально мало (например, если функция равна 1 лишь в одной точке), и тогда число требующихся шагов будет экспоненциально велико. Можно модифицировать алгоритм, чтобы он работал полиномиальное число шагов.
Эту задачу (DNF-COUNT) можно переформулировать в других терминах.

Рассмотрим множество двоичных слов длины n $\Omega = \{0,1\}^n$. Цилиндром будем называть такое подмножество $A \subseteq \{0,1\}^n$, если существует такой набор индексов $1 \le i_1 < i_2 < ... < i_k \le n$ и чисел $u_1, u_2, ..., u_k \in \{0,1\}$ таких, что выполняется условие: $\mathbf{x} = (x_1, ..., x_n) \in A$ тогда и только тогда когда $x_{i_p} = u_p$ для всех $p = 1, ..., k$. Неформально, цилиндр – множество двоичных слов, некоторые символы которых фиксированы.

Задача об объединении цилиндров (CILINDERS-UNION).
В пространстве $\Omega = \{0,1\}^n$ заданы m цилиндров $C_1, ..., C_m$. Найти объем их объединения V.

Действительно, каждая конъюнкция задает цилиндр. Набор переменных является выполняющим тогда и только тогда, когда он принадлежит хотя бы одному из цилиндров, заданных конъюнкциями.

FPRAS (Fulli Polynomial Randomized Approximation Scheme)

Для предыдущих задач перечисления мы рассматривали приближенные вероятностные алгоритмы их решения с гарантированной оценкой точности (С-приближенные алгоритмы).
Теперь хотим построить FPRAS. Это вероятностный алгоритм, который на вход получает помимо входных данных положительное рациональное число ε. Алгоритм с вероятностью не менее $\frac{3}{4}$ должен выдать правильный ответ с относительной погрешность не более, чем ε. Время работы алгоритма должно быть полиномиальным по размеру входных данных и $\frac{1}{\varepsilon}$. Вероятность $\frac{3}{4}$ можно увеличить до $1-\delta$, если выполнить алгоритм $O(\ln(1-\delta))$ раз и выдать среднее арифметическое полученных чисел. Это следует из неравенства Чернова.
Чтобы построить алгоритм FPRAS нужно сгенерировать случайную величину X с математическим ожиданием, равным V (объем объединения цилиндров). Алгоритм выглядит таким образом:

1. Выберем натуральное число k.
2. Сгенерируем k раз случайную величину X, обозначим их $X_1, ..., X_k$.
3. В качестве ответа выдадим среднее арифметическое $\frac{X_1 + ... + X_k}{k}$.

Воспользуемся неравенством Чебышева для нахождения такого натурального числа k, чтоб выполнялись соответствующие требования на вероятность приближенного решения:

$$P\left\{\left|\frac{X_1+...+X_k}{k}-V\right|<\varepsilon V\right\}\geq\frac{3}{4}$$

$$P\left\{\left|\frac{X_1+...+X_k}{k}-V\right|<\varepsilon V\right\}=1-P\left\{\left|\frac{X_1+...+X_k}{k}-V\right|\geq\varepsilon V\right\}\geq 1-\frac{DX}{k(\varepsilon V)^2}\geq\frac{3}{4}.$$

$$k\leq 4\frac{DX}{(\varepsilon V)^2}$$

Хотим выбрать такую случайную величину X, удовлетворяющую условиям:

1. $EX=V$
2. X генерируется за полиномиальное от n и m время.
3. величина $\frac{DX}{V^2}$ ограничена полиномом от n и m.

Случайная величина, удовлетворяющая первым двум требованиям, строится так:

а) Выберем равновероятно $x\in\{0,1\}^n$

б) Тогда положим $X=\begin{cases}2^n, & x \text{ принадлежит объединению цилиндров}\\ 0, & \text{иначе}\end{cases}$

Вычислим дисперсию: $DX=2^{2n}\frac{V}{2^n}-V^2=V(2^n-V)$. Если $V=\Theta(1)$, то $\frac{DX}{V^2}=\Theta(2^n)$. Но если $V=\frac{2^n}{poly(n,m)}$, то $\frac{DX}{V^2}$ уже является полиномиальной от n и m.

Построение случайной величины для случая большого ответа.

Пусть H какое-то множество, удовлетворяющее следующим требованиям:

1) Размер множества H можно вычислить за полиномиальное время.
2) За полиномиальное время можно выбрать $x\in H$ равновероятно.

Пусть G некоторое подмножество множества H, проверку на принадлежность к которому можно выполнить за полиномиальное время, и $\frac{|G|}{|H|}=\frac{1}{poly(n,m)}$. Тогда для оценки $|G|$ можно построить такую случайную величину X:

а) Выбирается равновероятно $x\in H$

б) Положить $X=\begin{cases}|H|, & x\in H\\ 0, & x\notin H\end{cases}$.

Воспользуемся этими рассуждениями для нашей задачи.

В качестве множества H возьмем объединение всех цилиндров, как мультимножество, то есть $H=\{(x,i):x\in C_i\}$. Генерировать $x\in H$ будем в два этапа. Первый – выбираем цилиндр (C_j будем выбирать с вероятностью, пропорциональной его объему). Второй – равновероятно выбираем элемент внутри цилиндра. Опишем множество G. Если x принадлежит объединению цилиндров, то в H может лежать несколько пар вида (x,i). Выберем «каноническую» пару (с наименьшим номером i) и поместим в множество G. Принадлежность G можно проверять за полиномиальное время. $\frac{|G|}{|H|}\leq\frac{1}{m}$, так как каждый элемент x входит не более, чем в m пар. Приведенные рассуждения позволяют построить FPRAS для $V=|G|$.

Анализ для «почти всех входов».

Задача о покрытии. Дано конечное множество X из m элементов и система его подмножеств $S_1,...,S_n$. Требуется найти минимальную по числу подмножеств подсистему $S_1,...,S_n$, покрывающую все множество объектов.

В общем случае, вводится функция стоимости $c:\ S_1,...,S_n \to \mathbb{R}^+$, приписывающая каждому множеству из $S_1,...,S_n$ положительное значение (стоимость). Требуется найти минимальное по стоимости покрытие.

$J \subseteq \{1,...,n\}$

$\bigcup_{j\in J} S_j = X$

$\sum_{j\in J} c(S_j) \to \min$

Или если стоимость всех подмножеств одинакова, то $|J| \to \min$. (Будем рассматривать этот простой случай, оговаривая, как модифицировать алгоритм на произвольную функцию стоимости.)

Жадный алгоритм. Точность O(lnm).

Рассмотрим жадный алгоритм, который на каждом шаге добавляет максимально покрывающее еще не покрытые элементы подмножество. В общем случае, ищется подмножество с минимальной удельной стоимостью $\bar{c}$, с которой покрываются «новые» (еще не покрытые элементы). (То есть удельная стоимость некоторого подмножества на некотором шаге алгоритма есть отношение обычной стоимости этого подмножества к числу еще не покрытых на данном шаге алгоритма элементов.)

Покажем, что жадный алгоритм гарантирует точность $O(\ln m)$.

Пусть X_k - число непокрытых элементов после k-ого шага алгоритма. M - размер минимального покрытия. Имеем:

$$X_{k+1} \le X_k - \frac{X_k}{M} = X_k\left(1-\frac{1}{M}\right).$$

$$X_k \le m\left(1-\frac{1}{M}\right)^k \le m\exp\left(-\frac{k}{M}\right).$$

Найдем наименьшее натуральное k, при котором $m\exp\left(-\frac{k}{M}\right)<1 \Leftrightarrow\ k > M\ln m$. Откуда следует результат.

Для общего случая функции стоимостей:

Обозначим Ω_{opt} - оптимальное покрытие и пусть $\mathrm{OPT} = c(\Omega_{opt})$ - его стоимость. Покрытие, построенное жадным алгоритмом - Ω. Припишем каждому элементу $x \in X$ потенциал $\pi(x)$ равный удельной стоимости множества, впервые покрывшего этот элемент в процессе работы жадного алгоритма. Ясно, что $\sum_{x\in X}\pi(x) = c(\Omega)$.

Покажем, что если элемент $x \in X$ был покрыт на i-ом шаге, то $\pi(x) \le \frac{\mathrm{OPT}}{m-i+1}$. Допустим, что это не так, тогда на i-ом шаге удельная стоимость множества $S^{(i)}$ относительно текущего покрытия Ω_i будет не меньше $\bar{c}_{\Omega_i}\left(S^{(i)}\right) \ge \frac{\mathrm{OPT}}{|X\setminus\Omega_i|}$. Посчитаем удельные стоимости множеств, входящих в Ω_{opt} относительно Ω_{i-1}. Покажем, что среди множеств оптимального покрытия Ω_{opt} найдется множество с удельной стоимостью меньшей

$\bar{c}_{\Omega_i}\left(S^{(i)}\right)$. Пусть для всех $Y \in \Omega_{opt}$, не содержащихся целиком в Ω_i, справедливо $\bar{c}_{\Omega_i}(Y) > \dfrac{\mathrm{OPT}}{|X \setminus \Omega_i|}$. Но тогда справедливо

$$\mathrm{OPT}=\sum_{Y\in\Omega_{opt}} c(Y) > \sum_{Y\in\Omega_{opt}} |Y \setminus \Omega_i| \frac{\mathrm{OPT}}{|X \setminus \Omega_i|} = \frac{\mathrm{OPT}}{|X \setminus \Omega_i|} \sum_{Y\in\Omega_{opt}} |Y \setminus \Omega_i| \geq \frac{\mathrm{OPT}}{|X \setminus \Omega_i|} |X \setminus \Omega_i| = \mathrm{OPT}.$$

Получили противоречие. Итак, оптимальное покрытие содержит подмножество, с удельным весом меньшим $\bar{c}_{\Omega_i}\left(S^{(i)}\right)$, что противоречит эвристике жадного алгоритма.

Значит, выполняется неравенство $\pi(x) \leq \dfrac{\mathrm{OPT}}{m-i+1}$.

$$c(\Omega) = \sum_{x\in X} \pi(x) \leq \sum_{i=1}^{m} \frac{\mathrm{OPT}}{m-i+1} = \mathrm{OPT} \cdot O(\ln m).$$

Можно построить алгоритм, основанный на решении задачи линейного программирования, который дает лучшее решение, если известно, что каждый элемент множества X покрывается не более, чем f подмножествами, т.е. $\forall x \in X \quad \left|\left\{j \in \{1,...,n\} : x \in S_j\right\}\right| \leq f$.

f-приближение для задачи покрытия.

Сформулируем задачу о покрытии (Set Cover) (простой вариант) в терминах булевых матриц и целочисленного линейного программирования:

$$\begin{cases} \mathbf{cx} \to \min, \\ \mathbf{Ax} \geq \mathbf{b}, \\ \forall j \; x_j \in \{0,1\}. \end{cases}$$

Здесь переменные $x_1,...,x_n$ соответствуют включению подмножеств $S_1,...,S_n$ в решение-покрытие, матрица $\mathbf{A}$ - матрица $m \times n$ инцидентности, $\mathbf{c} = (1...1)^T \in \mathbb{R}^n$, $\mathbf{b} = (1...1)^T \in \mathbb{R}^m$ - векторы стоимости и ограничений.

Упражнение. Что значат m строк ограничений? Чему равна целевая функция? Что характеризует столбец матрицы инцидентности? (в терминах множеств в объектов)

Разрешив переменным принимать вещественные значения, перейдем от ЦЛП к ЛП:

$$\begin{cases} \mathbf{cx} \to \min, \\ \mathbf{Ax} \geq \mathbf{b}, \\ \forall j \; x_j \geq 0. \end{cases}$$

Эта задача (ЛП) решается за полиномиальное время. Пусть $\bar{\mathbf{x}}$ - ее решение. Построим решение $\mathbf{x}$ для ЦЛП по такому правилу: если $\bar{x}_j < \dfrac{1}{f}$, то $x_j = 0$, иначе $x_j = 1$. Покажем, что такое решение будет f-приближением задачи о покрытии.

Вначале покажем, что полученное решение $\mathbf{x}$ является допустимым. То есть в терминах множеств: покажем, что полученное решение действительно является покрытием. Из условия, что $\mathbf{A}\bar{\mathbf{x}} \geq \mathbf{b}$ и в каждой строке матрицы инцидентности не более f единиц, следует, что для каждого $i \in \{1,...,m\}$ найдется $j \in \{1,...,n\}$: $a_{ij} = 1, \bar{x}_j \geq \dfrac{1}{f}$, а значит $x_j = 1$, что и значит, что элемент включен в покрытие.

Так как $\mathbf{x} \leq f\bar{\mathbf{x}}$, то $\mathbf{cx} \leq f\mathbf{c}\bar{\mathbf{x}} \leq f \cdot \mathrm{OPT}$, что и требовалось.

Анализ жадного алгоритма «для почти всех входов» задачи о покрытии.

Покажем, что для «типичных данных» решение жадный алгоритма почти равно минимальному покрытию.
Снова воспользуемся формулировкой задачи в терминах ЦЛП.
Пусть элементы матрицы инцидентности – независимые случайные величины с бернулевским распределением: $P\{a_{ij}=1\}=p,\ P\{a_{ij}=0\}=1-p$.
Для решения задачи применим жадный алгоритм.

Теорема. Пусть для случайной матрицы **A**, определенной выше, выполнены соотношения:

$$\forall \gamma > 0:$$
$$\frac{\ln n}{m^{\gamma}} \underset{n\to\infty}{\to} 0,$$
$$\frac{\ln m}{n} \underset{n\to\infty}{\to} 0.$$

Тогда для $\forall \varepsilon > 0:\ P\left\{\frac{Z}{M} \le 1+\varepsilon\right\} \underset{n\to\infty}{\to} 1$, где Z – решение жадного алгоритма, M – величина минимального покрытия.
Доказательство.
Сначала покажите, что почти наверное величина минимального покрытия не меньше $l_0(\delta) = -\left\lceil (1-\delta)\frac{\ln m}{\ln(1-p)} \right\rceil$, т.е. $P\{M \ge l_0\} \to 1$. Для этого введем случайную величину X, равную числу покрытий размера $l_0(\delta)$. И покажем, что $P\{X \ge 1\} \to 0$. Для этого достаточно (из неравенства Чебышева-Маркова) показать, что $EX = C_n^{l_0}(1-(1-p)^{l_0})^m \to 0$.
Далее, покажите верхнюю оценку размера покрытия жадным алгоритмом на почти всех входах.
Для этого воспользуйтесь неравенством для больших уклонений:
$Y_i = \begin{cases} 1, & p; \\ 0, & 1-p \end{cases}$, тогда $P\left\{\left|\sum_{i=1}^{n} Y_i - np\right| > \delta np\right\} \le 2\exp\left\{-\frac{\delta^2}{3}np\right\}$.
Из этого утверждения покажите, что среднее число «плохих» столбцов (у которых число единиц менее, чем $(1-\delta)np$) стремится к нули с ростом n.
Пусть N_t - число непокрытых строк после t-ого шага жадного алгоритма. Имеем:

$$N_t \le N_{t-1} - \frac{N_{t-1}(1-\delta)pn}{n} = N_{t-1}(1-(1-\delta)p)^t = m(1-(1-\delta)p)^t.$$

Откуда максимальное t, при котором есть еще непокрытый элемент:

$$m(1-(1-\delta)p)^t \Rightarrow t \le -\frac{\ln m}{\ln(1-(1-\delta)p)}$$

Откуда получается верхняя оценка мощности жадного покрытия в типичном случае:

$$P\left\{Z \le -\frac{\ln m}{\ln(1-(1-\delta)p)}\right\} \to 1$$

Комбинируя, с нижней оценкой минимального покрытия, получим, что $\forall \varepsilon > 0: P\left\{\frac{Z}{M} \le 1+\varepsilon\right\} \underset{n\to\infty}{\to} 1$. Воспользуйтесь тем, что $\forall \varepsilon > 0\ \ \exists \delta:$

$$\frac{Z}{M} \le \frac{\ln(1-p)}{(1-\delta)\ln(1-p(1-\delta))} \le (1-\delta)^{-1} + o(1) \le 1+\varepsilon.$$