ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
«ПРИДНЕСТРОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. Т. Г. ШЕВЧЕНКО»

Физико-технический институт
Факультет информатики и вычислительной техники

Кафедра программного обеспечения вычислительной техники

А.В. КИРСАНОВА

ПРАКТИКУМ ПО РЕШЕНИЮ ЗАДАЧ ПО ДИСЦИПЛИНЕ «АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ»

Учебно-практическое пособие

Тирасполь
Издательство Приднестровского университета
2026

УДК 004.657
ББК 32.81я73

Рецензенты:
С. Г. Федорченко, доцент кафедры программного обеспечения вычислительной техники ФТИ ГОУ ПГУ им. Т.Г. Шевченко
С. Б. Бабенко, директор по развитию ЗАО «ТирАЭТ»

Кирсанова А. В.
Практикум по решению задач по дисциплине «Алгоритмы обработки данных»: Учебно-практическое пособие [Электронный ресурс] / А.В. Кирсанова; ГОУ «Приднестровский государственный университет им. Т. Г. Шевченко», Физико-технический институт; Факультет информатики и вычислительной техники. – Тирасполь: Изд-во Приднестр. ун-та, 2026. – 70 с.
Минимальные системные требования: CPU (Intel/AMD) 1,5 ГГц/ОЗУ 2 ГГб/HDD 450 Мб/1024*768/Windows 7 и старше/ Internet Explorer 11/Adobe Acrobat Reader 6 и старше.

В пособии описан подход к решению задач по дисциплине «Алгоритмы обработки данных». Даны пояснения к решению множества примеров, приведены по несколько возможных решений большого количества задач, дан список формулировок тренировочных задач для самостоятельной работы. Пособие предназначено для студентов направления «Программная инженерия» и «Информационные системы и технологии».

УДК 004.657
ББК 32.81я73

Рекомендовано Учебно-методическим советом ПГУ им. Т. Г. Шевченко

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время неотъемлемой частью практически всех сфер деятельности – от бизнеса и науки до образования и искусства стало умение решать задачи по программированию. Среди востребованных в этой области компетенций можно выделить способность формулировать задачу, анализировать её и находить эффективное алгоритмическое решение. Это требует определенных технических и интеллектуальных навыков, а также наличия логического и критического мышления.

Во-первых, программирование лежит в основе большинства инноваций. Искусственный интеллект, большие данные, кибербезопасность, автоматизация и робототехника строится на умении писать и оптимизировать код. Компании по всему миру нуждаются в специалистах, которые умеют не просто знать язык программирования, а именно решать задачи – понимать, как превратить требования и данные в работающую программу.

Во-вторых, решение задач по программированию развивает универсальные когнитивные способности: системное мышление, внимательность к деталям, умение разбираться в сложных структурах и моделировать процессы. Эти качества полезны не только программистам, но и представителям любых профессий, связанных с анализом информации и принятием решений.

Кроме того, в образовательной и профессиональной среде решение задач – это основной способ проверки знаний и практических навыков. Именно через задачи обучающийся учится применять теорию на практике, улучшает алгоритмическое мышление и становится способным быстро адаптироваться к новым технологиям и языкам программирования.

Умение решать задачи по программированию – это не просто технический навык, а ключ к успешной деятельности. Оно развивает мышление, открывает широкие профессиональные возможности и помогает быть конкурентоспособным в условиях стремительного технологического прогресса.

1 ОБЩИЕ СВЕДЕНИЯ О ЖАДНЫХ АЛГОРИТМАХ

Основная идея жадных алгоритмов заключается в том, что последовательно на каждом шаге необходимо делать жадный выбор, то есть последовательно брать самое оптимальное на данный момент значение, то есть на каждом шаге выбирается локально оптимальное решение – такое действие, которое в данный момент кажется наилучшим, в надежде, что последовательность таких выборов приведёт к глобально оптимальному решению.

Очевидно, что такая стратегия не всегда срабатывает. Рассмотрим несколько таких примеров и научимся распознавать ситуации, когда применение жадных алгоритмов оправданно и осмысленно.

Жадные алгоритмы похожи на динамическое программирование, где тоже часто решается задача оптимизации. Например, задача о рюкзаке (упаковать рюкзак максимальной стоимости) может быть решена и жадной стратегией и с помощью динамического программирования:

– задача о рюкзаке с повторениями (есть предметы);

– непрерывная задача о рюкзаке (есть вещества, поэтому можно взять любое количество).

Пример: пусть дан рюкзак вместимостью W=100 и три предмета с заданными стоимостью и объемом.

Таблица 1 – Исходные данные для задачи о рюкзаке

	стоимость	объем	удельная стоимость
1	120	60	2
2	80	50	1,6
3	80	50	1,6

Если взять I предмет, то больше ничего не поместится, суммарная стоимость окажется равной 120. А если взять II предмет, то III-й поместится и стоимость будет равна 160.

Свойство оптимальности подзадач соблюдается и в жадных алгоритмах и в динамическом программировании. Рассмотрим решение задач.

2 РЕШЕНИЕ КЛАССИЧЕСКИХ ЗАДАЧ С ПОМОЩЬЮ ЖАДНЫХ АЛГОРИТМОВ

2.1 Задача о выборе заявок

Формулировка: имеется прямая, на которой отмечено несколько отрезков, которые могут «пересекаться». Необходимо выбрать максимальное количество непересекающихся отрезков. Считать, что если отрезки пересекаются по концу, то они являются непересекающимися.

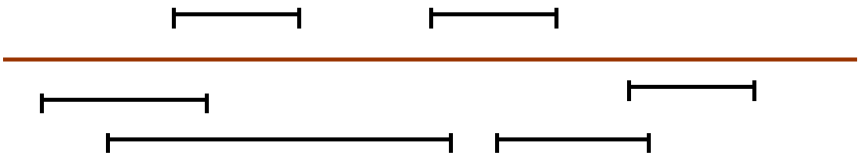


Рисунок 1 – Демонстрация условия задачи о выборе заявок

Жизненная постановка задачи такова: в некоторой фирме имеется комната для переговоров (для совещаний) и все отделы фирмы должны проводить совещания именно в этой комнате. Отрезки – это длительность одного совещания с известным началом и концом совещания. Необходимо удовлетворить максимальное количество заявок, то есть максимизировать количество заявок.

Стратегия решения такова: взять заявку, которая заканчивается раньше всех. Потом удалить все пересекающиеся с ней заявки.

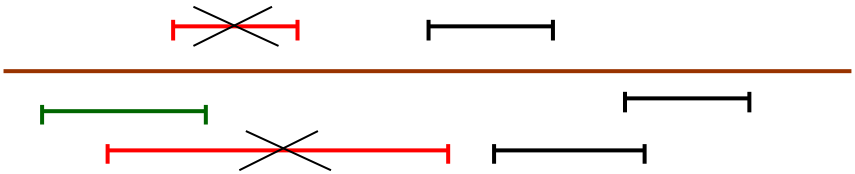


Рисунок 2 – Первое удаление в задаче о выборе заявок

Затем берем следующую заявку с минимальным временем окончания.

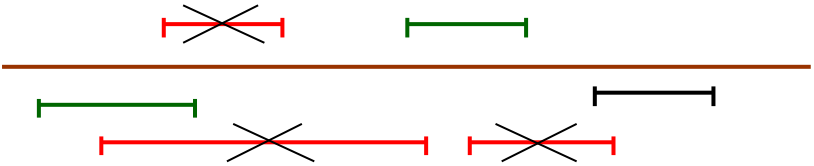


Рисунок 3 – Второе удаление в задаче о выборе заявок

Итак, шаги жадного алгоритма следующие:

1) отсортировать заявки по возрастанию по времени окончания;

2) взять текущую заявку с минимальным временем окончания и удалить все те заявки, с которыми текущая пересекается.

2.2 Задача о минимальном покрывающем дереве

Формулировка: в заданном взвешенном неориентированном связном графе G, удалить все лишние ребра так, чтобы G оставался связным, а сумма весов оставшихся ребер была минимальной.

Например, имеется компьютерная сеть, необходимо убрать лишние провода, так чтобы суммарная длина оставшихся проводов была минимальная, а сеть оставалась связной.

Жадный алгоритм решает эту задачу, выбирая ребро минимального веса.

Лемма: Пусть множество S – это подмножество множества вершин графа: $S \subseteq E$ и S – подмножество некоторого МОД. Пусть также имеется разрез ($V1$, $V2$), такой что S не пересекает этот разрез. (Разрез означает, что множество вершин разбито на 2 части $V1$ и $V2$ и нет ребер из $V1$ в $V2$ и наоборот.) Изобразим это графически:

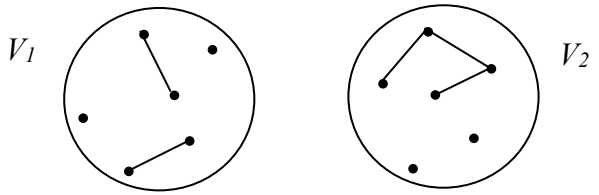


Рисунок 4 – Разрез множества

Показаны только ребра, принадлежащие S.

Пусть, e – самое легкое ребро графа G, пересекающее разрез. Тогда его надо добавить в S, то есть к множеству S можно приписать это ребро: $S \cup \{e\}$, а это значит, что получено подмножество некоторого оптимального решения, то есть подмножество некоторого минимального покрывающего дерева (МПД).

Доказательство:

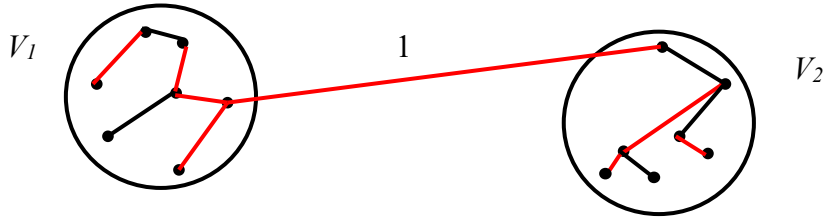


Рисунок 5 – Ребро между разрезами множества

Черным цветом нарисованы вершины и ребра множества S. Известно, что S – это подмножество некоторого покрывающего дерева. Дорисуем его красным цветом. Нарисуем ребро e, которое должно идти из $V1$ в $V2$.

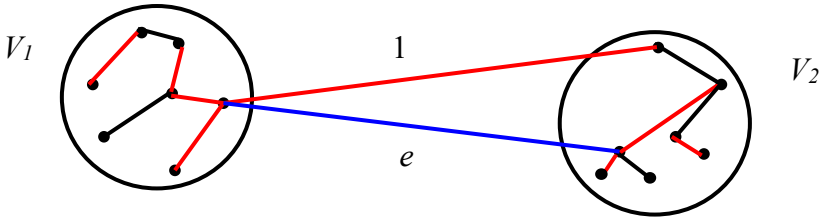


Рисунок 6 – Ребра между разрезами множества

Если ребро 1 и есть e, то доказывать ничего не надо: уже нарисовано МПД и ребро e в него входит, то есть $S \cup \{e\}$ – подмножество МПД.

Предположим, что e – это какое-то другое ребро (синим цветом изображено на рисунке). Тогда в дереве образовался цикл и

более того: понятно, что найдется какое-то другое ребро в этом дереве, которое соединяет одну часть с другой.

Рассмотрим дерево, в котором ребро 1 удалено.

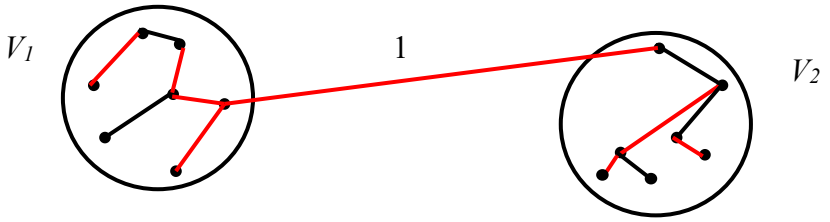


Рисунок 7 – Удаление ребра между разрезами множества

С одной стороны, это обязано быть деревом, то есть связность не нарушена. Осталось дерево и суммарный его вес не увеличился, т.к e – это самое легкое ребро (по условию). Значит, то что получилось и есть МПД. Ч.т.д.

2.3 Задача выполнимости формул Хорна (HORN SAT)

Формулировка: в общем виде задачи выполнимости формул Хорна формулируются так: существуют ли такие значения переменных, при которых формула Хорна истинна.

То есть имеется некоторая конъюнкция формул

$$F = (a \vee b \vee \overline{c}) \wedge (a \vee b \vee d) \wedge (a \vee \overline{e}) \wedge ... \wedge (e \vee \overline{b}) .$$

Необходимо найти выполняющий набор.

Формула Хорна – это конъюнкция дизъюнктов Хорна. Чтобы формула Хорна была истинной, необходимо, чтобы в каждой скобке была хотя бы одна 1. Если в каждой скобке не более трёх литералов, то задача о выполнимости такой формулы называется задачей о 3-выполнимости, если не более, чем 2, то 2-выполнимости. Но задача 3-выполнимости – *NP*-полная, а 2-выполнимости решается за полиномиальное время.

Каждую скобку назовём дизъюнктом Хорна – скобка, в которой не более одного положительного литерала $(k \le 1)$.

I тип: $\bar{a} \vee \bar{b} \vee \bar{c}$ или $\bar{a} \vee \bar{b} \vee c$; $k \le 1$

$$\bar{a} \vee \bar{b} \vee c = (\overline{a \wedge b}) \vee c = (a \wedge b) \to c$$

т.е. дизъюнкт Хорна I типа – импликация.

Если $a = 0$ или $b = 0$, то формула выполнима. Если $a = 1$ и $b = 1$, то обязательно и $c = 1$, чтобы формула была выполнима.

Здесь может быть сколько угодно отрицательных литералов, а положительный только один $(k \le 1)$.

Например: $\bar{a} \vee \bar{b} \vee c = (a \wedge b) \to c$

$\bar{a} \vee c = a \to c$

$c =$ конъюнкция из нуля литералов $\to c$ или $1 \to c$ или просто $\to c$.

Когда говорят о формуле Хорна, часто пишут именно так: из пустого множества предпосылок следует. Т.е. $\to c$ – это корректный дизъюнкт Хорна.

Итак, дизъюнкты типа $\bar{a} \vee \bar{b} \vee c$, в которых есть хотя бы один положительный литерал. И есть ещё фраза, что из множества переменных $\bar{a} \vee \bar{b} \vee \bar{c}$ хотя бы одна должна быть ложной, чтобы её отрицание было истинным и тогда формула будет выполнимой.

С помощью этих литералов можно написать некоторые логические фразы. Например, язык Пролог основан на дизъюнктах Хорна. В него синтаксически заложена естественным образом фраза о том, что из некоторого множества литералов следует какой-то литерал. Например, если у Васи живёт собака, и собака живёт в доме у человека, который курит, значит Вася курит.

Пусть есть формула Хорна, в которой каждый из дизъюнктов имеет вид $\bar{a} \vee \bar{b} \vee c$ или $\bar{a} \vee \bar{b} \vee \bar{c}$. Проверим существует ли выполняющий набор для формулы $F = (\bar{a} \vee \bar{b} \vee \bar{c}) \wedge (b) \wedge (\bar{b} \vee c)$, в каждой скобке которой не более одного положительного литерала. Для удобства размышления перепишем эту формулу на языке импликации: $F = (\bar{a} \vee \bar{b} \vee \bar{c}) \wedge (\to b) \wedge (b \to c)$

Если $a = b = c = 0$, то $F = 0$, то есть b обязано быть 1.

Далее будем заменять 0 на 1 только там, где необходимо это сделать. Нигде единица на ноль заменена не будет. Итак: $a = 0, b = 1, c = 0$. Теперь проблема в конъюнкте: $b \to c$. Поэтому

$c=1$. Теперь имеем $a=0, b=c=1$. Рассмотрим скобку $\bar{a} \vee \bar{b} \vee \bar{c}$. Она истинна. Значит вся формула выполнена.

Шаги жадного алгоритма:

1) рассмотрим все скобки типа $x \to y$; присвоим всем переменным y единицу.

2) если все скобки выполнимы, то формула выполнима, иначе повторяем шаг 1.

3) рассмотрим скобки, где нет положительных литералов и если хоть одна из них не выполнена, то её можно и не выполнить, следовательно вся F не выполнима.

За линейное время определена выполнимость формулы Хорна.

Скобки просматриваются по очереди и присваивается 1 тем литералам, которые сделают скобку выполнимой, т.е. правые части импликации должны равняться 1.

Затем пересматриваются те скобки, в которых встречалась эта переменная в левой части импликации и там всё исправляется, если необходимо. Но при правильной организации это всё выполняется за линейное время. Это реализовано в языке Прологе.

2.4 Задача о покрытии множествами

Приближённое решение *NP*-полной задачи «Покрытие множествами» (*set cover*). Пусть дано множество $U = \{U_{1,\dots}, U_n\}$ и семейство его подмножеств $E = \{S_{1,\dots}, S_к\}$, где $S_i \subseteq U$. То есть E – это множество подмножеств множества U.

В сумме все множества покрывают U. Задача о покрытии множествами (ЗПМ) состоит в том, чтобы найти минимальный набор подмножеств, покрывающий всё множество U.

Пример: имеется набор новых городов, находящихся на каком-то расстоянии друг от друга. В этих городах подрастают дети, которым необходимо учиться в школах, но школы пока надо строить не во всех городах сразу. Однако детям нельзя ездить дальше, чем на 30 км. То есть на рисунке с городами обозначаем множества радиусом r = 30 км. Так сделаем для всехгородов и находим минимальное покрывающее множество.

Это *NP* – задача. Поэтому неизвестно ни одного полиномиального алгоритма для решения этой задачи. Значит решаем эту

задачу приближенно жадным алгоритмом. На каждом шаге выбираем множество, покрывающее максимальное количество все еще не покрытых элементов.

Можно показать, что если самое оптимальное покрытие содержит p множеств, то жадный алгоритм найдет решение, содержащее не более $p \cdot \log n$ множеств.

Доказательство:

Пусть n_t – количество непокрытых элементов после t шагов. Так как все эти элементы покрываются p множествами из оптимального покрытия, то найдется множество, покрывающее хотя бы $\frac{n_t}{p}$ из них.

На следующем шаге:

$$n_{t+1} \le n_t - \frac{n_t}{p} = n_t \left(1 - \frac{1}{p}\right).$$

Затем для n_t можно записать такое же неравенство, и n_t станет n_{t-1}, а $\left(1 - \frac{1}{p}\right)$ взведется в квадрат. И так далее. В итоге

$$n_t \le n_0 \left(1 - \frac{1}{p}\right)^t < n_0 \left(e^{-\frac{1}{p}}\right)^t = n \cdot e^{-\frac{t}{p}}.$$

Заметим, что n_0 – количество элементов после нуля операций, следовательно $n_0 = n$, т.е. все элементы $n_t \le n \cdot e^{-\frac{t}{p}}$.

При $t = p \cdot \ln n$, то $n_t \le n \cdot e^{-\frac{p \ln n}{p}} = n \cdot e^{-\ln n} = n \cdot \frac{1}{n} = 1$

2.5 Задача о минимальном количестве тестов

Пусть имеется некоторое сложное электронное устройство и есть набор тестов, которые позволяют проверить это устройство. Тест № 1 проверяет подсистемы x_1, x_3 и x_5; № 2 – x_1, x_2, x_5; № 3 – x_3 и x_4; № 5 – x_3 и x_5.

Вся система $U = \{x_1, x_2, x_3, x_4, x_5\}$.

$t_1 = \{x_1, x_3, x_5\}$;

$t_2 = \{x_1, x_2, x_5\}$;

$t_3 = \{x_3, x_4\}$;

$t_4 = \{x_3, x_5\}$;

$T = \{t_1, t_2, t_3, t_4\}$

Нельзя выпускать устройство, у которого хоть одна из подсистем не работает. Надо проверить все системы, потратив минимальное время. Надо найти такое множество $S \subset T$, чтобы объединение давало множество U'.

Для данного примера – это тесты №2 и №3. Тестом №1 проверим модули 1, 3, 5; тестом №2 – 3 и 4 модули.

Эта задача *NP*-полная, так как найти в произвольном случае оптимальное множество тестов, которое покрывает всё множество U за полиномиальное время невозможно. Получить терпимый результат, в разумное количество раз, отличающийся от оптимального, можно с помощью жадного алгоритма.

В начале положим, что $S = \varnothing$, т.е. что в начале никакие тесты не проводятся. И каждый раз выбирается такой тест t_i, у которого пересечение с множеством U максимальное, т.е. выбирается тест, который проверяет наибольшее количество подсистем из тех, которые ещё не были проверены.

Итак, в начале никакой тест не выбран. Выбирается любой, который покрывает максимальное количество подсистем, например t_1; остались две непокрытые подсистемы: x_2 и x_4. Теперь выбирается такой тест, который проверяет максимальное подмножество из непроверенных. Например t_2. Теперь проверены x_1, x_3, x_5, x_2.

Не проверен x_4. Выбирается t_3. Ответ: 3 теста (вместо двух в оптимальном случае).

Оценим, насколько жадный алгоритм дал решение хуже оптимального. Сделаем это так. Представим, что существует набор тестов, который имеет размер *optimum,* и обозначим *opt*; и будем отслеживать итерации, т.е. как выбирается очередное множество. Для этого будем следить за величиной *m* – число непроверенных элементов (размер непокрытой части множества) на данный момент.

Пусть за одну итерацию получено m'. Оценим на сколько улучшится (уменьшится) размер непокрытого множества, сколько подсистем точно проверятся (покроется). Для этого подумаем о следующем: существует ответ, в котором *opt* элементов, т.е. есть такие *opt* множеств, которые покрывают всё множество U'. В частности, эти *opt* множеств покрывают и все *m* элементов, непокрытых в данный момент. Значит, по принципу Дирихле один из них (t_2 и t_3) покрывает хотя бы $\frac{m}{opt}$ элементов (т.е. $\geq \frac{m}{opt}$). Если каждый из них покрывает меньше чем $\frac{m}{opt}$, то, учитывая, что их всего *opt*, они вместе покрывают меньше, чем *m*. То есть из непокрытых на данный момент элементов они покрывают меньше *m*. Но по условию *opt* элементов покрывают всё множество. Получено противоречие. Следовательно, по принципу Дирихле, один из оптимальных покрывает хотя бы $\frac{m}{opt}$ элементов.

Итак, существует какой-то тест, который покрывает $\frac{m}{opt}$ подсистем. А выбирается максимальный, то есть тот тест, который покрывает максимальное количество элементов. Значит сейчас покроется хотя бы $\frac{m}{opt}$ элементов. Возможно, выберется самый оптимальный, тогда всё идеально. А если нет, то в любом случае, выбранный тест покроет хотя бы $\frac{m}{opt}$ элементов. После этого

$$m' \leq m - \frac{m}{opt};\ m' \leq m\left(1 - \frac{m}{opt}\right).$$

И так необходимо поступать после каждой итерации. Изобразим это графически.

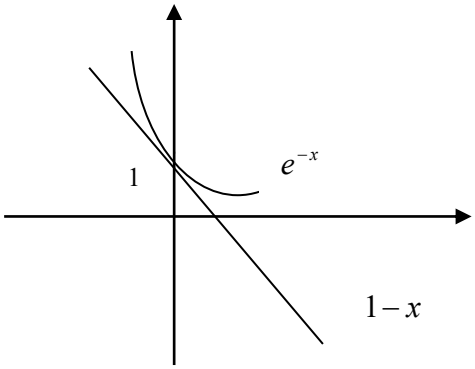


Рисунок 8 – Графическая интерпретация уменьшения количества тестов

Заметим, что $e^{-x} \geq 1-x$, то есть $1-\frac{1}{opt} \leq e^{-\frac{1}{opt}}$. Значит $m' \leq m \cdot e^{-\frac{1}{opt}}$.

Т.е. количество непокрытых элементов на каждой итерации уменьшается в $e^{-\frac{1}{opt}}$ раз. Теперь совершим такое следующее количество операций: $opt \cdot \ln|U|$, то есть организуем цикл от 1 до $opt \cdot \ln|U|$, а значит выполним указанное число итераций. Затем посмотрим, что происходило с множеством непокрытых элементов.

Вначале все были непокрыты: $|U|$ и каждый раз умножали на $e^{-\frac{1}{opt}}$ и в итоге количество непокрытых элементов стало меньше либо равно $|U| \cdot e^{-\frac{1}{opt} \cdot opt \cdot \ln|U|} = |U| \cdot e^{-\ln|U|} = |U| \cdot |U|^{-1} = 1$.

После $opt \cdot \ln|U|$ операций, количество непокрытых элементов будет ≤ 1. А после $1 + opt \cdot \ln|U|$ итераций количество непокрытых элементов будет < 1, т.е. $m < 1$, значит размер множества непокрытых элементов < 1, следовательно размер множества – нуль. Т.е. все элементы покрыты после $1 + opt \cdot \ln|U|$ итераций.

Итак, после такого нехитрого, но довольно элегантного математического анализа, понятно, что действуя жадным алгоритмом и сделав $1 + opt \cdot \ln|U|$ итераций, покрыто всё множество. Естественно, что цикл будет не от 1 до $opt \cdot \ln|U|$, а выбирается очередное множество пока не всё покрыто. Реально алгоритм закончится раньше, чем $1 + opt \cdot \ln|U|$ итераций, но жадный алгоритм точно сделает не больше, чем $1 + opt \cdot \ln|U|$ итераций. Значит, в данной задаче возьмём тестов не больше, чем $1 + opt \cdot \ln|U|$ штук.

Время работы полиномиальное, причём найденный ответ будет не хуже оптимального, чем в $\ln|U|$ раз, где $|U|$ – количество тестируемых подсистем. Если 1000 подсистем, то в $\ln 1000$, т.е. в 10 раз.

2.6 Построение минимального остовного дерева

Необходимо построить МПД (МОД). Приведем сначала псевдокод алгоритма Крускала.

```
Kruskal (G)
X=∅
отсортировать рёбра G по возрастанию веса
     для всех рёбер e∈E в отсортированном порядке, если
     добавление e в X не создаёт цикла X = X∪{e}.
```

На примере разберем, как этот алгоритм работает.

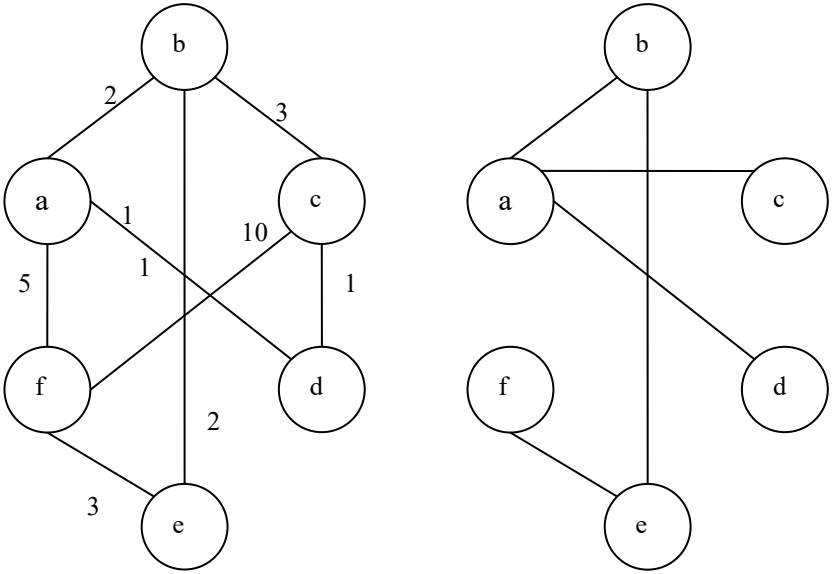


Рисунок 9 – Пример для построения МОД

Список отсортированных рёбер исходного графа и их веса:

$$\{a,c\},\{a,d\},\{c,d\},\{a,b\},\{b,e\},\{b,c\},\{f,e\},\{a,f\},\{c,f\}.$$
$$1,\quad 1,\quad 1,\quad 2,\quad 2,\quad 3,\quad 3,\quad 5,\quad 10.$$

Теперь перемещаясь по списку отсортированных рёбер, добавляем $\{a,c\}$, $\{a,d\}$, дальше предлагается $\{c,d\}$, но это ребро нельзя брать, так как будет цикл. Затем $\{a,b\}$, $\{b,e\}$; Нельзя $\{b,c\}$, затем $\{f,e\}$. Имеется 6 вершин и уже 5 рёбер, т.е. если ещё хоть 1 ребро добавим – получим цикл. Значит дальше перебирать не имеет смысла.

Оценим время работы. Время сортировки

$$\mathrm{O}\left(|E|\log|E|\right)=\mathrm{O}\left(|E|\log|V|\right).$$

Изменим псевдокод алгоритма Крускала.

```
Kruskal (G)
X=∅
отсортировать рёбра G по возрастанию веса
для всех рёбер {u,v} ∈ E  в отсортированном порядке,
  если  find(u) ≠ find(v)
      union(u,v)
      X = X ∪ {{u,v}}
```

Первые скобки означают, что это множество, состоящее из одного ребра $\{u,v\}$; внутренние фигурные скобки означают, что ребро неориентированное.

Структура данных, которая поможет в этой задаче называется системой непересекающихся множеств. Сначала этих множеств было *n* (количество вершин). Затем их начали объединять. *Union* объединяет, добавляет вершину в компоненту связности. По умолчанию предполагается, что у каждого множества (КС) есть свой представитель, а *find* находит его. Т.е. в строке $find(u) \neq find(v)$ проверяется, если представитель КС, содержащий вершину *u* и представитель КС, содержащий вершину *v* один и тот же, то *u* и *v* лежат в одном множестве, т.е. в одной КС.

Выясним как это быстро реализовать. Необходимо найти такую структуру данных, которая позволяет *Union* реализовать за константное время, а *find* за $\log|v|$. Общее время будет $|E|\log|V|+|E|\log|v|$, т.е. представим систему непересекающихся множеств. Изменив немного эту структуру данных, шаг 2 сможем выполнять за время $|E|\cdot\log^*|v|$, где $\log^*$ – это итерированный логарифм, т.е. это количество раз, которое надо взять логарифм от числа, чтобы он стал меньше 1. Это очень медленно растущая функция. Она не более 5 для всех разумных чисел, которые могут существовать, например, для количества частиц во Вселенной. То есть нет чисел на практике, для которых $\log^*$ будет больше 5. Вопрос только в том, зачем уменьшать второе слагаемое, первое и так будет доминировать. Однако это необходимо сделать. Если рёбра

были уже изначально даны в отсортированном порядке; кроме этого могло оказаться, что для сортировки достаточно не $|E|\log|v|$ времени, а всего $|E|$. Если, например, веса не очень большие, то числа от 1 до *n* можно отсортировать за $O(n)$. И тогда общее время $O(|E|+|E|\log^*|v|)$.

Итак, начнём с конструирования этой структуры данных – системы непересекающихся множеств. Над ней должны быть определены операции:

- *make set(x)* – создать множество из одного элемента;
- *find(x)* – найти представителя множества;
- *union(x, y)* – объединить множества, в которых лежат *x* и *y*.

Такую структуру можно реализовать на массиве, операция *make set* – создать массив из одного элемента; *find(x)* – найти представителя массива (например, первый элемент). Другими словами, необходимо, чтобы каждый элемент массива «знал» про начало этого массива (например, у каждого элемента массива есть ссылка на первый элемент). Т.е. *find(x)* работает за константное время.

Union(u, v) склеивает 2 массива, кроме этого необходимо перенаправить ссылки элементов второго массива. В худшем случае, это отработает за $O(v)$. Т.е. если топорно всё это делать, то время $union\,O(|v|)$, а их произойдёт *v* штук, т.е. получим v^2. Необходимо оптимизировать данный процесс – меньший массив приклеить к большему. Теперь *union* будет работать время $|v|\log|v|$ вместо v^2. Проследим за «судьбой» какого-то отдельного элемента. Он сначала был один. Для каждого элемента нашего множества перекидываний ссылок будет не больше, чем $\log|v|$ штук, так как каждый раз, перекидывая ссылку, уменьшаем хотя бы в 2 раза размер множества, в котором этот элемент находится. Пусть элемент находится в множестве. Если этот элемент к чему-то приклеим (со всем множеством), то значит, весь массив приклеим к чему-то, чей размер больше, т.е. размер множества, содержащего элемент, увеличится хотя бы вдвое.

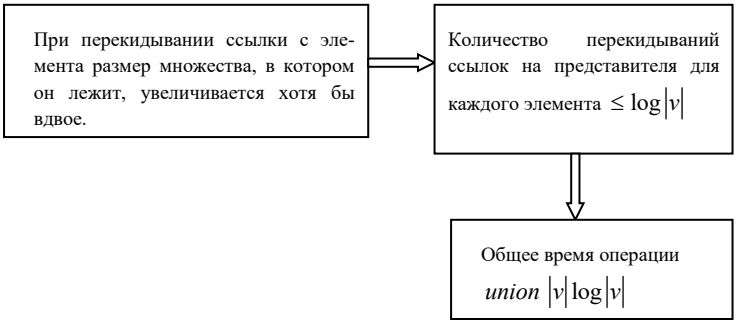


Рисунок 10 – Схема алгоритма построения МОД

Рассмотрим несколько другую реализацию системы непересекающихся множеств (СНМ), которая тоже даст время $v \cdot \log v$, но потом её необходимо доработать, и появится $\log^*$. Причём эта реализация тоже проста.

Покажем на примере. Пусть есть элементы, в каждом из которых хранится ссылка на родителя, но в начале – на него самого.

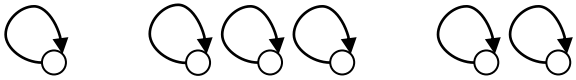


Рисунок 11 – Элементы, в каждом из которых хранится ссылка на себя

Если дальше понадобится объединить 1 и 3 вершину. Для удобства перерисуем её в другое место, но в памяти ничего не двигаем. Затем “кидаем” ссылку.

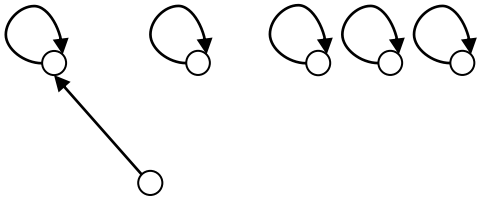


Рисунок 12 – Элементы, в каждом из которых хранится ссылка на родителя

Если теперь надо объединить с вершиной 2, то перерисуем:

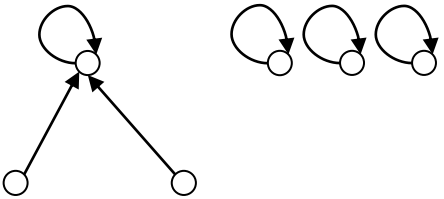


Рисунок 13 – Элементы после первого объединения

В итоге может получиться такая структура:

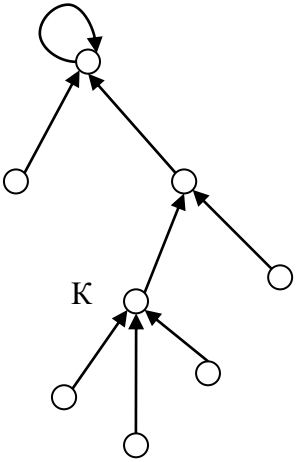


Рисунок 14 – Элементы после очередного объединения

Рассмотрим, чем будет заниматься *find.* Если вызвали *find* для вершины *K*, то *find* идёт по ссылкам на родителя находит корень, корень – это её представитель (представитель вершины *K*).

Выясним, как теперь делать объединение. Представьте, что теперь имеется какая-то более сложная ситуация после объединений.

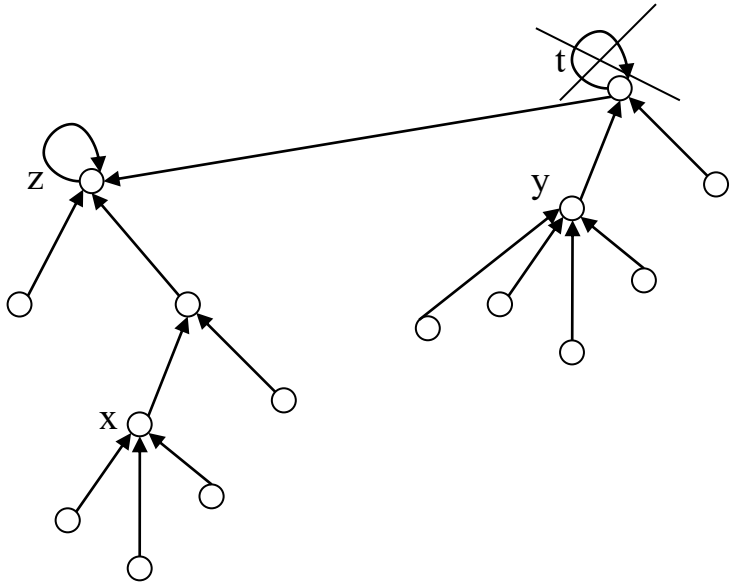


Рисунок 15 – Элементы после последнего объединения

И теперь надо объединить *x* и *y*, имеющие представителей *z* и *t*. Сначала напишем псевдокод *make set()* и *find()*.

```
make set(x)
x → par = x // родитель x - это сам x
x → rank = 0  // ранг x - это ноль
```

Время работы O(1).

```
find(x)
//запомним x в промежуточную переменную
temp=x
// пока temp ≠ своему родителю
while (temp ≠ temp → par)
идём наверх
(temp ≠ temp → par)
return temp
```

Время работы $O(\log n)$.

```
  union(x,y)
// найдём представителей
z=find(x)
t=find(y)
// делаем что-то, только если z ≠ t
if(z ≠ t)  // то перекинуть ссылку из t на z, так как глубина
дерева – определяющий фактор, поскольку find работает про-
порционально глубине дерева.
```

2.7 Задача коммивояжера (*TSP – travelling salesman problem*)

Пусть есть некоторое количество избирательных участков. Нужно их все посетить, пройдя минимальное расстояние. То есть нужно найти такую перестановку вершин V_{p1}, V_{p2}, .., V_{pn}, чтобы сумма ребер $\sum \left| V_{pi} - V_{pi-1} \right|$ была минимальна. Решать будем для графа, в котором выполнено неравенство треугольника, то есть для *i, j, k*, длина ребра $l_{ik} \le l_{ij} \ne l_{jk}$ (*with triangle Inequality*).

Это фактически *NP*-полная задача: есть ли в графе *G* путь коммивояжера длиной меньше *P*. А оптимизационная задача: найти минимальный путь коммивояжера. Решаем задачу приближенно, то есть, находим решение, которое отличается от оптимального в разумное количество раз.

Алгоритм

1 шаг. Найдем в этом графе МОД.

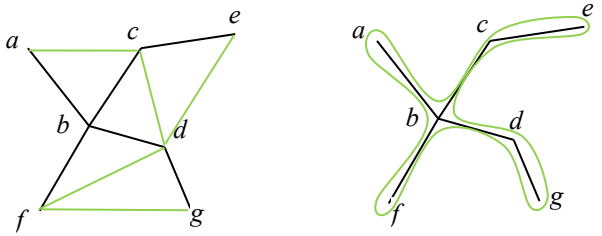


Рисунок 16 – Пример графа

2 шаг. Руководствуемся правилом правой руки выхода из лабиринта. Выбираем исходную вершину и обойдем МОД. Но существует неприятность: прошли через вершины по 2 раза. Формально, то, что сделано, называется Эйлеров обход дерева (ЭОД).

Суть ЭОД: пусть есть дерево, которое «подвешено за корень»

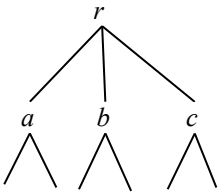


Рисунок 17 – Пример дерева, «подвешенного за корень»

Тогда ЭОД определяется так: корень; ЭОД поддерева *a*; корень; ЭОД поддерева *b*; корень; ЭОД поддерева *c*; корень. Пример:

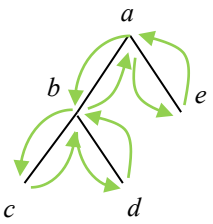


Рисунок 18 – ЭОД дерева

У ЭОД есть визуализация на плоскости: правило левой руки. Получился обход, в котором многие вершины повторяются. Но необходимо без повторений. Идем по рисунку без повторений (по правилу левой руки): *a*, *b*, *f*, теперь *b* не выписываем, *d*, *g*, теперь *d* и *b* не выписываем, *c*, *l*, теперь *c* не выписываем, замыкаем *a*.

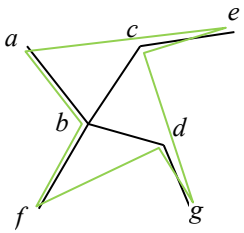


Рисунок 19 – Правило левой руки

Это конечно не идеальный ответ, выясним насколько этот ответ плох. Пусть оптимальный ответ имеет длину *opt*, он как-то устроен. Удалим произвольное ребро и полученный путь назовем *p*; его длина *p<opt*. А путь является деревом, причем содержащим все вершины графа, более того – это МОД.

Пусть *P* – некоторое остовное дерево, а *MST* – минимальное, следовательно *opt>p>=MST*. Рассмотрим черные ребра на рисунке 6 – это было МОД, по которому ходим по правилу левой руки, один раз в одну сторону, другой раз – в другую – это есть двойная длина МОД. Благодаря неравенству треугольника, когда хитрым образом сокращали путь (см. рисунок 7) от *f* к *d*, минуя *b*, суммарную длину не ухудшали *fd<fb+bd* относительно черного пути. Значит длина найденного пути *<= 2 * MST*, то есть по сравнению с оптимальным не хуже, чем в 2 раза.

2.8 Найти потерявшееся число

Дан массив, состоящий из 999 различных целых чисел в интервале [1; 1000]. Узнать, какого числа не хватает.

Начнем с неоптимального решения и постепенно придем к самому оптимальному.

Решение 1. Наивное решение: проверить, есть ли число 1, затем 2, и т.д.

```
for i = 1 to 1000
    проверь, что i принадлежит массиву, если нет, верни i
endfor
```

Время работы $\theta(n^2)$. Память $\theta(1)$.

Решение 2. Чтобы улучшить решение 1, можно отсортировать исходный массив, а затем применить решение 1. Сортировать можно за время $n\log(n)$. Применение решения 1 будем искать двоичным поиском за время $n\log(n)$. Значит, время работы решения 2 равно $n\log(n)$.

Решение 3. Когда имеются все числа от 1 до 1000, то это означает, что имеем дело с арифметической прогрессией. Её сумма равна $S_n = (1+1000)\cdot\frac{1000}{2}$. Предположим, что в последовательности не хватает какого-то числа *x*. Тогда сумма равна $1001\cdot\frac{1000}{2} - x$. Значит, надо найти $\sum_{i=1}^{999} array_i$, затем $1001\cdot\frac{1000}{2} - \sum_{i=1}^{999} array_i = x$, задача решена за *n* шагов. Выясним можно ли решить эту задачу за меньшее время, за сублинейное время. Разберемся, а почему взята именно сумма в решении 3. Что будет, если возьмем произведение от 1 до 1000? Получим 1000! Если не хватает одного числа, то получим $\frac{1000!}{x}$. Тогда, чтобы найти искомое число, надо посчитать $\frac{1000!}{\prod_{i=1}^{999} array_i}$. Но 1000! – это очень большое число. Воспользовавшись формулой приближения Стирлинга, получим $\left(\frac{1000}{e}\right)^{1000} = 330^{1000}$. Это число так велико, что не помещается в память компьютера. Чтобы считать отдельно, числитель и знаменатель дроби $\frac{1000!}{\prod_{i=1}^{999} array_i}$, надо произвести много вычислений. Лучше будет, если изменить порядок: $\left(\left(\frac{1}{a_1}\cdot\frac{2}{a_2}\right)\cdot\frac{3}{a_3}\right).....\frac{999}{a_{999}}\cdot 1000$. Таким образом, решение 3 самое удачное.

2.9 Найти два потерявшихся числа

Дан массив, состоящий из 999 различных целых чисел в интервале [1; 1000]. Необходимо найти два потерявшихся числа.

В предыдущей задаче составили уравнение, из которого нашли потерявшееся число. По аналогии составляются два уравнения с двумя неизвестными и отыскиваются потерявшиеся числа. Просуммируем числа $1+...+1000=\frac{1000\cdot 1001}{2}$

$$\begin{cases} \frac{1000\cdot 1001}{2}-\sum_{i=1}^{999} array_i = x+y \\ \frac{1000!}{\prod_{i=1}^{999} array_i} = x\cdot y \end{cases}$$

Время работы $\theta(n)$. Однако полученная система – не единственная.

Например: $1^2+2^2+...+1000^2$.

2.10 Найти лишнее число

Пусть дан массив $a_1, a_2, ..., a_n$, где каждый элемент $\in [1, 2, ..., n-1]$. Все элементы массива – целые числа и находятся в интервале от 1 до *n*-1. Согласно принципу Фибоначчи, существуют хотя бы два элемента с одинаковым значением. Разработаем алгоритм, который находит повторяющееся значение.

Решение 1. Посчитаем, сколько раз встречается в массиве число 1, сколько раз число 2, и т.д. до (*n*-1) числа, и вернем те числа, которые встречаются более одного раза. Время работы $\theta(n^2)$. Память $\theta(c)$.

Решение 2. Применяем сортировку подсчетом: заводим дополнительный массив, в который помещаем количество повторений каждого элемента массива. Время работы линейное $\theta(n)$. Память $\theta(n)$.

Решение 3. Написать алгоритм со временем работы $\theta(n\log(n))$, памятью $\theta(n\log(n))$. Этим ограничениям отвечает *QuickSort*. После работы *QS* получим массив $b_1 \le b_2 \le b_3 \le ... \le b_n$. При *QS* рекурсии может понадобиться $\log(n)$ памяти.

Решение 4. Только одно число встречается два раза. Сумма *n* членов арифметической прогрессии равна $\frac{n(n+1)}{2}$. Если из этой суммы отнять найденную сумму имеющихся в массиве элементов, то получим искомое число. Время работы $\theta(n)$, память $\theta(\tilde{n})$.

Решение 5. Можно посмотреть на элементы массива как на адреса в памяти, т.е. как на указатели. Это значит, что если в массиве есть одинаковые элементы, то если посмотреть на массив, как на связанный список, то увидим, что один элемент указывает, например, на пятый, и какой-то другой тоже указывает на пятый. Это значит, что имеется цикл. Т.е. если построим алгоритм, который, получив связанный список, ответит на вопрос, «есть ли в списке цикл?» и будет работать $\theta(n)$ времени, сможем всегда ответить на вопрос, «есть ли в массиве два одинаковых числа?» и что это за число.

Там, где начинается цикл – это и есть искомый повторяющийся элемент. Решить эту задачу лучше с помощью способа «двух бегунов» – указателей, которые перемещаются по списку с разной скоростью.

Но здесь есть тонкость: предположим, строим список так – берем первый элемент массива, его содержимое указывает по какому адресу хранится второй элемент списка, переходим по этому адресу и т.д. Но с таким решением возможны проблемы. Предположим, что в какой-то момент времени мы вернулись, например, так:

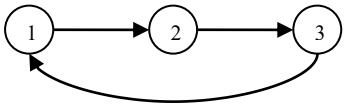


Рисунок 20 – Возврат в списке

Здесь нет двух элементов с одинаковыми значениями, а цикл получен, потому, что начали с единицы. Стоит начать с элемента a_n. Т.к. элементы массива принимают значения от 1 до *n*-1. А это значит, что никогда не будет адреса, который вернет обратно в *n* и не получим петлю на a_n. Затем запустим два «бегуна»: первый с a_n, а второй с a_{n-k}.

2.11 Задача 100 дверей

Какие двери будут открыты через сто шагов, если имеется 100 пронумерованных двери, которые закрыты, причем:

1 шаг. Каждая дверь меняет свое состояние (закрытая открывается и наоборот).

2 шаг. Каждая вторая дверь меняет свое состояние, т.е. дверь с четным номером.

3 шаг. Каждая третья дверь меняет свое состояние ($i\%3 == 0$).

…

100 шаг. Каждая сотая дверь меняет свое состояние, т.е. только дверь номер 100

Вместо 100 дверей возьмем 10.

Таблица 2 – Состояние дверей

i	*Д1*	*Д2*	*Д3*	*Д4*	*Д5*	*Д6*	*Д7*	*Д8*	*Д9*	*Д10*
1	.	.	.	.	.	.	.	.	.	.
2		.		.		.		.		.
3			.			.			.	
4				.				.		
5					.					.
6						.				
7							.			
8								.		
9									.	
10										.

Вначале дверь закрыта. Если над дверью произведено четное количество операций, то она закрыта, то есть состояние не изменилось. Если нечетное – открыта, т.е. состояние изменилось. Так из таблицы видно, что двери с номерами 1, 4, 9 открыты.

Может все двери, номера которых являются квадратами некоторого натурального числа, будут открыты? Смотрим на таблицу: меняем состояние двери, если остаток от деления её номера на номер итерации равен нулю.

Например: дверь №6 меняла своё состояние на 1, 2, 3 и 6 итерации, т.е. на итерации, номер которой является делителем номера двери. Дверь №8: на 1, 2, 4, и 8 итерации. Зная, что дверь закрыта при четном количестве операций. Это замечание приводит к выводу, что дверь не изменит своего состояния (будет закрыта), если количество делителей будет чётным.

Дверь открыта, когда количество делителей нечетное. Найдем у каких из чисел от 1 до 100 нечётное число делителей.

В теории числе есть функции, связанные с количеством делителей числа, например, функция Эйлера. Т.е. можно взять эту функцию, дать её компьютеру и получить количество делителей. Но лучше сделать более быстрый способ. Замечено, что четное количество делителей имели числа 1, 4, 9. Гипотеза: может это числа, являющиеся квадратами?

Пусть имеется число *x*. Если *a* делитель *x*, то получим такое целое натуральное число *b*, что $x = a \cdot b$. Т.е., если *a* делит *x*, то и *b* делит *x*. Значит всегда, когда есть делитель *a*, то есть и делитель *b*, т.е. есть два делителя. Это правило нарушается, когда $a = b$. И тогда делитель один: $x = a \cdot a$ т.е. *x* – это квадрат натурального числа. Значит, открыты двери с номерами – квадратами: 1, 4, 9, 16, 25, 36, 49, 64, 81, 100. Эту задачу можно обобщить.

2.12 Определение цикла в списке

Дан однонаправленный список *LinkedList.* Найти, есть ли в этом списке цикл.

Если в однонаправленном списке есть цикл, то никогда не будет найден конец списка, т.к. он стал несвязным. Кроме того, если написать цикл, и обойти от начала до конца списка, то такой цикл никогда не закончится!

Наша цель написать алгоритм, который получает указатель на начало списка, а возвращает, есть ли в этом списке цикл.

Решение 1. Начнем с первого элемента и будем идти до тех пор, пока не встретим указатель на *NULL*, если же *NULL* не найден, то есть цикл. Этот алгоритм зациклится, если в списке будет цикл.

Решение 2. Начнем идти по списку и каждый посещенный элемент будем помечать. Можно завести структуру данных (дополнительно) или изменять сами элементы списка, добавив дополнительное поле. Но не всегда это можно сделать, во-первых; а во-вторых, придется обойти весь список, чтобы добавить это поле. Но если будет цикл, не сможем обойти весь список. Предположим, что мы с этой проблемой справились – смогли обойти весь список, помечая в дополнительном поле факт посещения узла списка. Если дошли до NULL и ни разу не встретили элемент с закрашенным полем, значит в списке нет циклов. Если встретили элемент с закрашенным полем, то цикл есть. Но это решение не верное, т.к. при наличии цикла невозможно обойти ВЕСЬ список.

Решение 3. Пусть нельзя менять структуру списка. Возьмем массив, в который будем записывать адрес очередного узла. При дальнейшем анализе массива, если найдутся одинаковые элементы, то цикл есть. Т.е. в массив заносим адрес, и проверяем, есть ли такой адрес в массиве. Если да, то цикл найден. Если дошли до конца списка, то цикла нет. Если есть n узлов в списке, то обход списка требует $\theta(n)$ времени. Если массив не отсортирован, то поиск требует $\theta(n)$. Итого $\theta(n^2)$ времени надо алгоритму. Память $\theta(n)$. Это решение можно улучшить, если в процессе заполнения сортировать массив, но это тоже занимает время. Кроме этого массив – не лучшая структура данных, т.к. если в процессе работы оказывается, что длины массива не хватает, то придется делать реаллокацию.

Решение 4. Аналогично решению 3, но вместо массива адресов можно взять сбалансированное бинарное древо (АВЛ, красно-черное, 2-4-дерево…). Каждый поиск и каждая вставка занимали бы по $n\log(n)$ времени. Тогда общее время $\theta(n\log(n))$.

Решение 5. Вместо массива адресов можно использовать таблицы, тогда поиск занимает константное время. Тогда время $\theta(n)$. Память $\theta(n)$.

Решение 6. У каждого элемента списка есть адреса, которые в общем случае находятся в некотором адресном пространстве, есть минимальный адрес и есть максимальный адрес. Не обязательно, что это первый и последний элементы. Всегда можно найти разность между *ними*. Тогда обходя список от начала до конца (если в списке нет цикла), количество элементов, которые увидим, будет меньше найденной разности. Если цикл есть, то, обходя список, никогда не придем в конец, и количество элементов в какой-то момент станет больше найденной разности. Памяти необходимо 3 ячейки: *min, max, diff*, то есть память $\theta(\text{const})$. Время $\theta(\textit{размер адресного пространства})$.

Решение 7.

Пусть есть круг, по которому бегут два спортсмена. Если они бегут с разной скоростью, то они будут встречаться бесконечное количество раз. Если они бегут с одинаковой скоростью в одном направлении, то они не встретятся никогда. Если они бегут по прямой, и $V_1 > V_2$, то они не встретятся никогда в противоположность бегу по кругу. Именно эта идея и будет реализована при решении задачи. Возьмем два указателя, и один будет «бежать» немного быстрее, чем другой и в итоге они помогут найти цикл.

Алгоритм: *pointer1* указывает на *node1* и его скорость $V_1 = 1$, *pointer2* на *node2,* и его скорость $V_2 = 2$.

```
while (не конец списка)
 if (pointer1==pointer2)
       stop, есть цикл
 else
       pointer1 += v1
       pointer2 += v2
```

Если цикла нет, то *pointer2* придет к *NULL* раньше и его никогда не догонит *pointer1*. Если цикл есть, необходимо доказать, что в какой-то момент времени *pointer1* станет >= *pointer2.*

Для двух спортсменов решение очевидно, так как круг – непрерывное множество точек и поэтому утверждение верно. Список – дискретное множество, *p1* и *p2* бегут по конкретным номерам. Осталось доказать, что если цикл есть, то алгоритм работает, т.е. $\exists t : p_1(t) = p_2(t)$

Пусть длина цикла *l*. В какой-то момент времени *p1* будет указывать на начало цикла, а *p2* находиться на расстоянии *y* от *p1*. Как описывается движение *p1*? Нанесем координаты на список. Запишем уравнение, которое позволит найти координату *p2* и *p1* в момент времени *t*.

$p_1(t) = t \cdot \mathrm{mod}(l)$ т.к. цикл длиной *l*, но в момент времени *l+1* координата $p_1 \neq l-1$, а равна единице.

$p_2(t) = Y + 2t \cdot \mathrm{mod}(l)$, т.к. координата увеличивается на 2. *y* – начальная координата.

Решаем систему:

$$\begin{cases} \mathrm{p}_1 = t \bmod(l) = x \\ p_2 = Y + 2t \bmod(l) = z \end{cases} \Leftrightarrow \begin{cases} x - t = k \cdot l \\ z - y - 2t = m \cdot l \end{cases} \Leftrightarrow \begin{cases} x = t + k \cdot l \\ z = y + 2t + m \cdot l \end{cases}$$

Вычитаем из первого уравнения второе. Учитывая, что правые части равны.

$kl - y - t - ml = 0$

$y + t = (k - m)l$

Значит *(y+t)* должно делиться на *l*.

$y + t = \xi \cdot l$

$y = \xi \cdot l - y$

Время $\theta(n)$. Память $\theta(const)$.

Решение 9. Проверим: есть ли цикл, который начинается с первого элемента, если есть, то выводим сообщение, если нет, то проверяем, есть ли цикл, который начинается со второго элемента и т.д., пока не пройдем весь список.

Шаг 1. Проверить наличие цикла, начинающегося с первого элемента.

Шаг 2. Проверить наличие цикла, начинающегося со второго элемента.

…

Шаг *n*. Проверить наличие цикла, начинающегося с *n*-го элемента.

Проблема этого решения: на шаге 1 берем второй элемент, проверяем, равен ли он первому, затем третий и т.д. Но неизвестно,

когда остановимся, т.к. не знаем размер списка. Это решение хорошее, но имеется проблема. Надо изменить алгоритм:

1) вместо того, чтобы проверить, существует ли цикл, который начинается с первого элемента, проверим, есть ли цикл длиной 1, начинающийся с первого элемента;

2) проверим, есть ли цикл длиной 2, начинающийся с первого элемента или цикл длиной 1, начинающийся со второго элемента;

3) проверим, есть ли цикл длиной 3, начинающийся с первого элемента, или длиной 2 со второго элемента или длиной 1 с третьего элемента и т.д.

Этот алгоритм корректен, он обойдет весь список и гарантированно найдет цикл. Время работы $\theta(length(list))$.

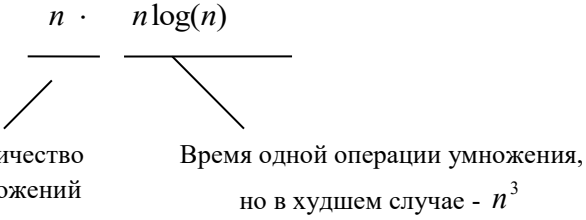
2.13 Задача о количестве нулей в конце факториала

Вход: целое число m. Выход: количество нулей в конце числа $m!$

Например:

1) для 2! программа вернет 0;
2) для 5! программа вернет 1 (5! = 120);
3) для 10! программа вернет 2;

Решение 1. Вычислить факториал $n!$ и подсчитать нули. Но сложность по времени этого решения велика



Итого $\theta(n^4)$. Теперь посчитаем нули числа, которое пропорционально n^n, а всего цифр $\log(n^n) \approx n\log(n)$. Итого $\theta(n^4)$.

Сложность по памяти велика, т.к. $n!$ *содержит* $n\log(n)$ цифр. И *int* и *long* быстро исчерпают диапазон.

Решение 2. Известно, что:

$1 \cdot 2 = 2$

$1 \cdot 2 \cdot 3 = 6$

$1 \cdot 2 \cdot 3 \cdot 4 = 24$

$1 \cdot 2 \cdot 3 \cdot 4 \cdot \underline{5} = 12\underline{0}$

...

$1 \cdot 2 \cdot 3 \cdot 4 \cdot \underline{5} \cdot \ldots \cdot \underline{10} = \ldots\underline{00}$

Гипотеза: количество нулей в конце $n!$ равно количеству чисел среди 1...n, которые делятся на 5.

25! = 5! · 5!

Алгоритм:

```
для каждого x из чисел от 1 до n
  t += количество чисел x, таких, что x%5= =0
return t
```

Временная сложность:

1) n операций;

2) $\log(n)$ раз можно поделить числа на 5.

Итого $\theta(n\log(n))$.

Пространственная сложность: надо хранить количество нулей, но не больше $n\log(n)$, т.е. не больше количества цифр, но для хранения в бинарной форме надо $\log_2(n\log(n))$, т.е. намного лучше, чем в предыдущем случае.

Решение 3. Чтобы узнать, сколько чисел из $[1;n]$ делятся 5 без остатка достаточно $\left\lfloor \frac{n}{5} \right\rfloor$. Значит кратных пяти $\left\lfloor \frac{n}{5} \right\rfloor$, кратных 25 $\left\lfloor \frac{n}{25} \right\rfloor$, кратных 125 $\left\lfloor \frac{n}{125} \right\rfloor$

Чтобы узнать количество нулей в конце надо

$$\left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{25} \right\rfloor + \left\lfloor \frac{n}{125} \right\rfloor + \ldots + \text{пока есть на что делить}$$

Алгоритм:

до тех пор пока $\left\lfloor \frac{n}{5} \right\rfloor > 0$

$$t+=\left\lfloor \frac{n}{5} \right\rfloor$$

$$n=\frac{n}{5}$$

Количество шагов цикла равно количеству раз, сколько можно поделить на 5. Это равно $\log(n)$. Значит, время $\theta(\log_2(n))$. Память $\log(n)$.

2.14 Операции с битами

Написать алгоритм, определяющий количество единиц в бинарном представлении числа.

Решение 1. Проверить каждый бит, если он равен 1, то нарастить счетчик на 1.

```
for i = 1 to n
 if( бит i равен 1)
       увеличить счетчик на 1
end
```

Время работы $\theta(n)$, n – количество бит в числе.

Но как проверить, что *i*-й бит равен 1?

Например: последний бит можно проверить так: *if (n and 00000001== n),* то последний бит 1; предпоследний бит: *if (n and 00000010 == n),*то последний бит равен 1, и т.д. до первого бита.

Решение 2. Если сдвигать число вправо на 1 бит, каждый раз проверяя, а не равно ли оставшееся число нулю, можно остановить сдвиг, как только найдется последняя единица.

Пример:

00000111 & 00000101; c = 1. Сдвиг числа вправо.

0000011 & 0000010; c = 2. Сдвиг числа вправо.

000001 & 000001; c = 3. Сдвиг числа вправо.

```
while (x != 0)
   if ((x & 00000001) == 1)
          count ++;
return count;
```

Время $\theta(n)$, так как количество итераций равно номеру последней единицы в данном числе, значит, в худшем случае имеем *n* операций.

Решение 3. А если число большое, причем единицы идут в начале числа, тогда время работы довольно большое. Можно идти с начала числа, двигать его влево. Пусть даны два числа.

Таблица 3 – Исходные данные для задачи операции с битами

								№ 2	№ 3
0	0	0	0	1	1	1	1	4 операции	4 операции
1	1	1	1	0	0	0	0	8 шагов	4 шага

Как видно, № 3 работает не хуже, чем № 2.

Пусть дано число 01010000.

Обнулить первую же единицу с правого края числа можно, выполнив операцию *x and (x-1).* Если сделать операцию:

$$01010000 \;\&\; (-1 + 01010000 - 1) = 01010000 \;\&\; 01001111 = 01000000.$$

Убрали единицу в седьмом разряде.

Повторим:

$$01000000 \;\&\; (-1 + 01000000) = 01000000 \;\&\; 00111111 = 00000000$$

Напишем псевдокод:

```
while( x!=0)
       x and x-1
              count++;
```

Здесь количество итераций равно количеству единиц.

2.15 Поиск строки в файле

Формулировка: дан файл, состоящий из N строк различной длины. Написать алгоритм, возвращающий случайную строку файла с вероятностью $p = \frac{1}{N}$.

1) N известно.

Решение 1. Генерируем случайное число

```
i = RAND(1…N);//1
return S[i];
```

Время работы Θ (длины файла *l*).

2) N неизвестно.

Решение 2. Пройтись по файлу. Найти количество строк *N//*Θ*(l)* и воспользоваться решением 1. На это потребуется время Θ *(l)*. Общее время Θ *(2l)*.

3) *N неизвестно.* По файлу можно пройти только один раз.

Решение 3.

Шаг 1: пройти по файлу, вычислить *N* и записать во вспомогательный массив

```
a[1] offset начала 1-й строки……..a[N]…….
```

Время Θ *(l)*, память Θ *(N)*.

Шаг 2: `x = rand(1..N)`

Шаг 3: `return file + a[x]`

file – указатель на начало файла, *a[x]* – *offset*, который хранится в ячейке с *№ x*.

4) *N неизвестно. Дополнительный массив не заводить.* Можно пройти по файлу 1 раз.

Генерируем случайное число из [1; 2].

```
x=Rand[1,2];
if(x == 1)
  winner = offset a[1];
else
  winner = offset a[2];
```

```
x = Rand[1,2];
if(x == 1)
  winner = winner;
else
  winner=a[3];
……….
      до шага N
      return S[winner];
```

Посчитаем вероятность того, что алгоритм вернет:

$$P(1^{\text{ю}}\text{ строку})=\underbrace{\frac{1}{2}\cdot\frac{1}{2}\cdots\frac{1}{2}}_{\text{N-1 раз}}=\frac{1}{2}^{(N-1)}\neq\frac{1}{N}$$

……..

$$P(\text{N}^{\text{ю}}\text{ строку})=\frac{1}{2}\neq\frac{1}{N}$$

$$P(\text{алгоритм вернет i-ю строку})=\underbrace{\frac{1}{2}\cdot\frac{1}{2}\cdots\frac{1}{2}}_{\text{N-i раз}}=\frac{1}{2}^{(N-1)}\neq\frac{1}{N}$$

Если бы можно было подобрать вероятности так, чтобы они были = $\frac{1}{N}$, то задача будет решена, причем решена верно.

1) `winner = offset a[1] с вероятностью` $\frac{1}{2}$

`else winner = offset a[2]` // если $P\neq\frac{1}{2}$

2) `winner = winner с вероятностью` $\frac{2}{3}$ // если строк всего 3, то чтобы вероятность каждой строки $=\frac{1}{3}$, надо $\frac{1}{2}\cdot p=\frac{1}{3}\Rightarrow p=\frac{2}{3}$; т.е. на шаге 1 $p_1=\frac{1}{2}$; на шаге 2 $p_2=\frac{2}{3}$.

`else winner=offset a[3]` $p=\frac{1}{3}$

3) `winner = winner c` $p = \frac{3}{4}$

`winner = offset a[4] c` $p = \frac{1}{4}$

и т.д.

Пример: $p(1^{\text{я}}$ строка победит везде$) = \frac{1}{2} \cdot \frac{2}{3} \cdot \frac{3}{4} \cdots \frac{N-1}{N} = \frac{1}{N}$

……..

$P(N^{\text{я}}$ строка победит$) = \frac{1}{N}$.

Время *Θ(l)*. *OEIS.org* – энциклопедия целочисленных последовательностей.

2.16 Продолжи ряд

Дан ряд: 1, 2, 3, 5, 7, 13, 21,… Необходимо его продолжить.

I способ: дописать любое число, например π.

Можно построить полином, который в точке 1 равен 1, в 2 – 2, в 3 –3, в 4 –5, в 5 –7, в 6 – 13, 7 – 21, 8 – π,…..

Интерполяция имеет следующее графическое представление:

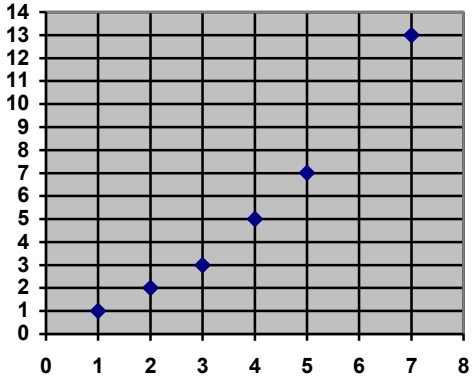


Рисунок 21 – Интерполяция ряда

II способ. Запишем в общем виде ряд $a_1, a_2, \ldots, a_n$. Здесь a_n – это количество практических чисел, меньших 2^n. Число n называется *практическим*, если для $\forall x \in N, x \le n$; $x = k_1 + k_2 + \ldots$, где k_i – различные делители числа n.

Тогда можно продолжить: 1, 2, 3, 5, 7, 21, 36, 61. Пример: 12; делители числа 12: 1, 2, 3, 4, 6. Любое число меньше 12 можно представить как сумму этих делителей. Например, 1 = 1; 2 = 2; 3 = 3; 4 = 4; 5 = 3 + 2; 6 = 6; 7 = 3 + 4; 8 = 2 + 6; 9 = 3 + 6; 10 = 4 + 6; 11 = 6 + 4 + 1.

2.17 Вероятность существования треугольника

На отрезке длиной 1, например, [0; 1] случайно и равновероятно выбирают две точки x и y. Или в другой формулировке: есть шнур длиной 1 и его разрезают случайным образом в двух местах. Какова вероятность, что из полученных трех кусков можно получить треугольник.

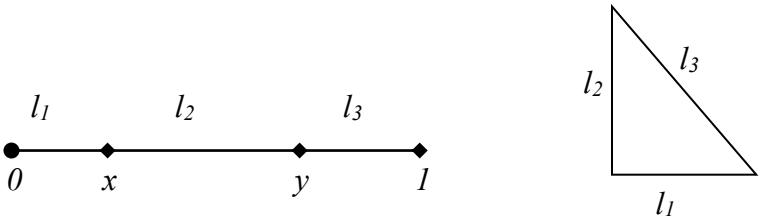


Рисунок 22 – Демонстрация возможности построить треугольник

Пусть x – меньшее из чисел, а y – большее. Чтобы треугольник существовал, необходимо чтобы сумма любых двух сторон была больше третьей.

Вероятность получить треугольник, если $x > 0{,}5$ равна нулю. $P(\Delta \mid x > 0{,}5) = 0$

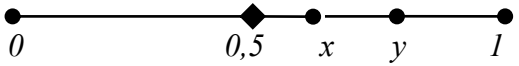


Рисунок 23 – Вероятность получить треугольник

Оценим вероятность *P(x > 0,5)*, т.е. вероятность, что треугольник составить нельзя. Меньшее из двух чисел больше, чем 0,5, когда каждое из двух чисел больше чем 0,5.

$P(x > 0,5) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. 1/2 для *x* и 1/2 для *y*.

Получим: вероятность, что треугольник составить нельзя, если обе точки *x* и *y* попадут:

1) справа от 0,5 равна $\frac{1}{4}$.

2) слева от 0,5 также $\frac{1}{4}$.

Значит с $P = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$ треугольник составить нельзя. Т.е. вероятность, что треугольник составить можно равна $\frac{1}{2}$.

Решим задачу формально: рассмотрим 2 случая.

Пусть $x < y$.

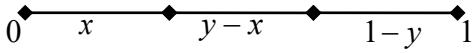


Рисунок 23 – Вероятность получить треугольник на отрезке [0; 1]

Чтобы треугольник существовал, необходимо:

$$\begin{cases} x < y - x + 1 - y \\ y - x < x + 1 - y \\ 1 - y < x + y - x \end{cases} \Leftrightarrow \begin{cases} x < 1 - x \\ 2y - 2x < 1 \\ -2y < -1 \end{cases} \Leftrightarrow \begin{cases} x < 0,5 \\ y - x < 0,5 \\ y > 0,5 \end{cases} \Leftrightarrow$$

$$\Leftrightarrow \begin{cases} x < 0,5 \\ y - x < 0,5 \\ y > 0,5 \end{cases} \Leftrightarrow \begin{cases} x < 0,5 \\ y - x < 0,5 \\ y > 0,5 \end{cases}$$

Воспользуемся геометрической интерпретацией

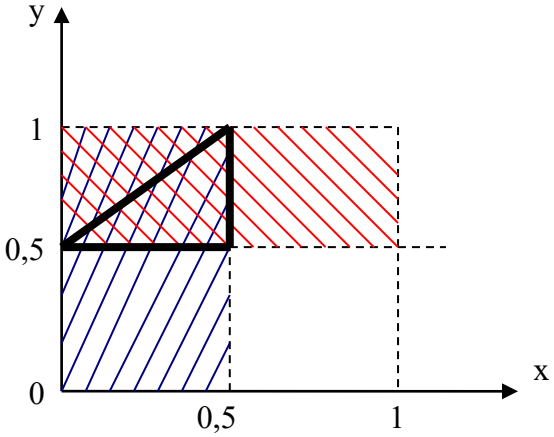


Рисунок 24 – геометрическая интерпретация

I область: $x < 0,5$ (синяя штриховка)

II область: $y > 0,5$ (красная штриховка)

III область: $y-x < 0,5;\ y < 0,5+x$ (черное выделение).

Находим пересечение всех областей. Видно, что его площадь = $\frac{1}{8}$. $P(\Delta \mid x < y) = \frac{1}{8}$.

Если $x > y$. Тогда $P(\Delta \mid x > y) = \frac{1}{8}$. Значит $P = \frac{1}{4}$.

2.18 Задача «Лестница Фибоначчи»

Найти количество способов подняться наверх по лестнице длиной *n* ступеней, если можно подниматься на 1 или 2 ступени.

Способ 1. Если есть лестница длиной 1 ступень, то имеется 1 способ; длиной 2 ступени – 2 способа; 3 ступени – 3 способа; 4 ступени – 5 способов.

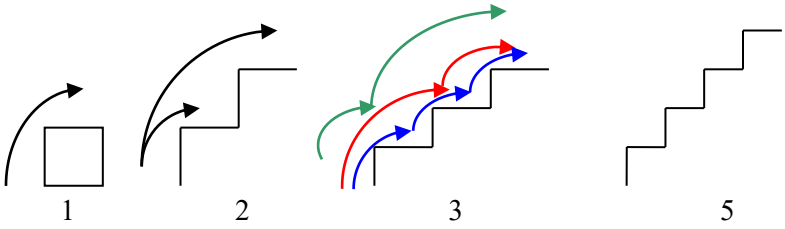


Рисунок 25 – Количество способов подняться наверх по лестнице

Заметили, что количество способов образуют последовательность Фибоначчи.

Способ 2: Динамическое программирование. Обозначим через a_n – количество способов подняться на лестницу высотой n ступеней. На $n_ю$ ступень могли прийти с $(n-1)$ или с $(n-2)$ ступени, причем эти способы взаимоисключаются. Значит $a_n = a_{n-1} + a_{n-2}$; $a_0 = a_1 = 1$.

Т.е. количество способов подняться на лестницу высотой n равно количеству способов подняться на лестницу высотой $(n-1)$ плюс количество способов подняться на лестницу высотой *(n - 2)* ступеней.

Алгоритм

```
a0=a1=1
for i=1 to n
   an=an-1+an-2
return an;
```

Время работы $\Theta(n)$.

Способ 3: Оценка a_n. Оценим a_n сверху и снизу. Разобьем всю лестницу на пары ступеней. На каждой паре принимаем как минимум два решения: подняться на 1 ступеньку или на 2. Итак, есть $\frac{n}{2}$ пар и каждая пара дает минимум 2 решения. Значит $2^{n/2} \le a^n$.

Оценим a_n сверху. Найдем лишние варианты. На каждой ступеньке можем принять не больше, чем 2 решения, но не факт, что на каждой ступеньке побываем (если с первой прыгнули на третью,

то на второй не побываем). Но в любом случае, как бы не шли, на каждой ступеньке можем побывать 1 раз и принять максимум 2 решения, т.е. $a_n \le 2^n$. Итак: $2^{n/2} \le a^n \le 2^n$.

Но разница $(2^n - 2^{\frac{n}{2}})$ велика. Рассмотрим примеры для двух значений *n*.

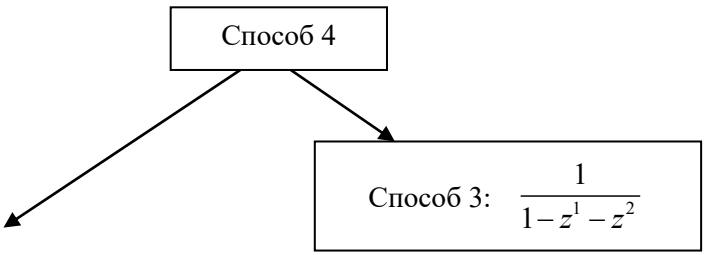
n = 10: $2^{10} = 1024$; $2^5 = 32$; $2^{10} - 2^5 = 992$.

n =20: $2^{20} \approx 1$ млн.; $2^{10} = 1024$; $2^{20} - 2^{10} \approx 1$ млн.

Способ 4 (магический). Пусть подъему на 1 ступень соответствует z^1, а на 2 – z^2. Значит, подъему на 1 или 2 соответствует (z^1+z^2) и так для каждой ступени. Производящая функция для сочетания (z^1+z^2) равна $\frac{1}{1-z^1-z^2}$.

Это и есть решение задачи, т.е. искомое количество способов. Но надо найти a_n. Для этого надо разложить эту функцию в ряд Тейлора и взять a_n как коэффициент разложения при x^n.

Способ 5. Получим формулу для вычисления a_n.



– напрямую взять уравнение $a_n = a_{n-1} + a_{n-2}$;

– перейти к производящим функциям $F_n = F_{n-1} + F_{n-2}$, отсюда получим ту же производящую функцию, что и в 3.

Далее разложим производящую функцию в ряд Тейлора. Получим $a_n = \left(\frac{\sqrt{5}-1}{2}\right)^n + \left(\frac{\sqrt{5}+1}{2}\right)^n$; $\frac{\sqrt{5}+1}{2} > 1$; $\frac{\sqrt{5}-1}{2} < 1$.

Зная, что *n* – целое, можно отбросить $\left(\frac{\sqrt{5}-1}{2}\right)^n$. Останется $a_n = \frac{\sqrt{5}+1}{2}$.

2.19 Задача «Альбом наклеек»

Первая формулировка В альбоме $n = 100$ наклеек. В конвертах наклейки распределены случайно равновероятно. Т.е. покупая конверт с наклейкой, мы можем в нем обнаружить 1 наклейку с любым номером от 1 до 100, с вероятностью $\frac{1}{N}$.

Цель: заполнить весь альбом и найти какое минимальное количество конвертов x надо купить, чтобы заполнить альбом (это значит надо найти математическое ожидание x).

Вторая формулировка: есть *CD*-диск, на котором записано 100 музыкальных произведений. Вероятность выбора каждого произведения равна 1/100. Сколько в среднем надо прослушать мелодий, чтобы услышать все мелодии хотя бы по 1 разу.

Эта задача называется *coupon collector*. Она из учебника «50 занимательных задач из теории вероятностей», автор Вастселлер.

x может принимать – минимум 100 значений, т.е. $x \geq 100$.

$\mathrm{E}x = \sum_{x=100}^{\infty} x \cdot P_r(X = x)$, т.е. матожидание равно сумме произведений каждого x на вероятность x.

Ex может быть найдено различными способами:

1) оценка матожидания;
2) другие способы вычисления матожидания.

Путь конверты приобретаются пока не заполнится весь альбом. Допустим, куплен конверт с наклейкой №1; а ее в альбоме не было. Значит *success*, кладем наклейку в альбом. Продолжаем покупать конверты. И в какой-то момент опять попадается наклейка, которой в альбоме нет. И т.д. В какой-то момент времени попадается сотая наклейка, которой нет. И здесь необходимо остановиться. Разберемся в чем же состоит хитрость этой задачи.

Пусть x_1 – количество конвертов, которые необходимо купить, чтобы получить первую новую наклейку;

x_2 – количество конвертов, которые надо купить необходимо получить вторую новую наклейку после первой;

x_3 – количество конвертов, которые необходимо купить чтобы получить третью новую наклейку после второй;

и т.д.

x_{100} - количество конвертов, которые необходимо купить чтобы получить сотую новую наклейку.

$x = x_1 + x_2 + x_3 + ... + x_{100}$.

Но цель найти матожидание. Значит

$\mathrm{E}X = \mathrm{E}(x_1 + x_2 + x_3 + ... + x_{100})$

$\mathrm{E}X = \mathrm{E}X_1 + \mathrm{E}X_2 + \mathrm{E}X_3 + ... + \mathrm{E}X_{100}$

$\mathrm{E}X_1 = \frac{100}{100} = 1$, т.к. наш альбом пуст.

$\mathrm{E}X_2 = \frac{100}{99};\ \mathrm{E}X_3 = \frac{100}{98};\ \mathrm{E}X_4 = \frac{100}{94};\ ...; \mathrm{E}X_{100}\ \frac{100}{1};$

Получили: $\mathrm{E}X = 100 \cdot (1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{100})$.

В скобках имеем усеченный гармонический ряд (УГР). А сумма УГР равна *ln(n)* + константа Эйлера, т.е. $\mathrm{E}X = 100(\ln 100 + \xi)$.

Т.е. если в альбоме n номеров, то надо купить $n \cdot \ln n$ конвертов.

А теперь оценим E*X*. Пусть n – количество номеров в альбоме. Оценим вероятность того, что купим n^2 конвертов и не заполним альбом; т.е. что не попалась ни разу I наклейка, II наклейка,…, сотая наклейка.

$\Pr(\ ) = \Pr(A_1 \cup A_2 \cup ... \cup A_n) \le \Pr(A_1) + \Pr(A_2) + ... + \Pr(A_n) = n \cdot \Pr(A_1)$,

т.к. вероятности равны.

Найдем какова вероятность того, что, купив n^2, ни разу не встретим первую наклейку $\Pr(A_1)$. Т.к. конверты независимы, то это значит, что в первом конверте будет не первая наклейка с вероятностью $1 - \frac{1}{n}$, во втором … $1 - \frac{1}{n}$ и т.д. n^2 раз. Т.е. $n \cdot \Pr(A_1) = n \cdot (1 - \frac{1}{n})^{n^2} = n \cdot (1 - \frac{1}{n})^{n \cdot n}$; величина $(1 - \frac{1}{n})^n = e^{-1}$ для больших n, т.е. $n \cdot \Pr(A_1) = n \cdot e^{-n} \to 0$ (при $n \to \infty$). Т.е. вероятность что, купив n^2 конвертов, не заполним альбом, быстро стремится к нулю. Но мы взяли n^2. Но могли бы взять меньше конвертов, например $10n \cdot \log_{10} n$.

$$. (1 - \frac{1}{n})^{n \cdot 10n \cdot \log_{10} n} = n \cdot e^{-\lg n^{10}} = \left[\left| e^{-\lg n} \right| \right] = n \cdot \frac{1}{10n} \le \frac{1}{10} .$$

2.20 Задача имитации симметричной монеты с помощью несимметричной монеты

Дана монета *B(p)*, где p – вероятность, причем неизвестная. Например вероятность выпадения «орла» больше, чем «решки». Необходимо сгенерировать случайную величину x, которая имеет распределение Бернулли с вероятностью $\frac{1}{2}$, т.е. такую x, которая подходит для реализации симметричной монеты.

Могут быть 3 решения:

1) алгоритм, который вернет точную симметричную монету и работает конечное время (это идеальное решение);

2) алгоритм имитирует симметричную монету с вероятностью $\frac{1}{2} \pm \varepsilon$ и будет работать конечное время;

3) $p = \frac{1}{2}$, но время неограниченно.

Предлагаем читателю реализовать предложенные алгоритмы самостоятельно.

2.21 Найти $\sqrt{x}$

Разработать алгоритм, который находит $\sqrt{x}$, где x – 32-битное целое число. Одно из решений – построить таблицу.

Таблица 4 – Таблица для задачи извлечения корня

$\sqrt{x_1}$	x_1
$\sqrt{x_2}$	x_2
. . .	

Если x – 32-битное целое число, то $\sqrt{x}$ - 16-битное целое число.

Таблица 5 – Диапазон для задачи извлечения корня

0	$0^2 = 0$
1	$1^2 = 1$
2	$2^2 = 4$
…	…
$2^{16} - 1$	$(2^{16} - 1)^2$

На препроцесс нам понадобится время $2^{16} \cdot time(x^2)$, где $time(x^2)$ – время на возведение в квадрат одного числа, но всего 2^{16} чисел.

Когда алгоритм будет работать с этой таблицей, имеются две сложности: если $x = 4$, то 4 находится по индексу 2 в этой таблице и ответ найден. Алгоритм поиска не обговариваем пока.

Если $x = 5$, то возникает вопрос: 5 не в таблице, 5 находится между 4 и 9.

В этом случае можно вернуть *min, max*, среднее (сделав интерполяцию).

Алгоритм

`input` $x \in 32bit$ `integer`
`output` $y \in 16bit$ `integer`, такое что $y^2 \approx x$

Например $y^2 \le x \le (y+1)^2$.

В таблице имеем массив упорядоченных чисел. Чтобы найти x, линейный поиск не оптимален, двоичный поиск работает $\log n$ времени; в нашем случае $\log n \approx 16$.

2.22 Сортировка

Пусть дан отсортированный по возрастанию массив целых чисел $A[1], A[2], ...A[n]$. Необходимо найти i, такой что $i = A[i]$.

Таблица 6 – Отсортированный по возрастанию массив

Элемент массива	-1	0	**3**	10	100
Номер элемента массива	1	2	**3**	4	5

Решение 1. Можно преобразовать исходный массив и получить новый массив, в котором и искать ответ с помощью *BinarySearch*.

Создадим массив *B[1..n]*. Каждый элемент $B[i] = A[i] - i$.

Массив B будет возрастающий, возможно не строго. Докажите это самостоятельно.

В массиве B с помощью *BinarySearch* будем искать нули, т.к. нуль получается только, если A[*i*] = *i*.

Построение массива $B : \theta(n)$ времени и $\theta(n)$ памяти. Т.е. столько же времени как в последовательном поиске. Значит, применим *BinarySearch*, но для *A* и искать будем $A[i] - i = 0$. Т.е. можно не тратить время на формирование массива. Время работы алгоритма равно времени *BinarySearch*, т.е. $\log n$, причём дополнительной памяти не надо.

3 РЕШЕНИЕ НЕСТАНДАРТНЫХ ЗАДАЧ

3.1 «Заключенные»

В одной стране царь решил дать шанс спастись ста заключенным. Заключенные могут по 1 заходить в комнату, в которой находится 100 коробок, внутри которых написан номер заключенного. Заключенному можно открыть 50 коробок. Если он открыл коробку со своим номером, то это хорошо, т.к. царь сказал, что все заключенные будут спасены тогда и только тогда, когда каждый обнаружит свое имя. Если хотя бы 1 свое имя не найдет, то все 100 будут казнены. Но царь был, добр и он разрешил перед тем, как зайти в комнату, заключенным совместно выработать стратегию. Но после того, как заключенный вышел из комнаты ничего говорить нельзя, помечать или ломать коробки нельзя. Какой алгоритм позволит увеличить шанс на спасение?

Решение1. Неэффективное решение. Каждый заключенный, заходя в комнату, открывает первую коробку, но в ней номер только одного заключенного, остальные 99 в этой коробке своего номера не найдут. Вероятность успеха *P*=0.

Решение 2. Пусть каждый заключенный случайно выберет 50 коробок из 100. Тогда вероятность, что спасется

$№1:\ P_1 = \frac{1}{2};\ №2:\ P_2 = \frac{1}{2};\ \ldots\ P_{100} = \frac{1}{2}.$

Т.е. вероятность что спасутся все $P = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \ldots \cdot \frac{1}{2} = (\frac{1}{2})^{100}$. Она мала.

Решение 3. Попробуем найти решение с вероятностью P>0,3. Это решение нетривиальное. Допустим заходит №1 и открывает коробку №1. Если там написано №1, то он спасен, а если что-то другое, например, №5, тогда он идет и открывает коробку с №5, если там написано 44, то он открывает коробку №44. Предположим там написано 1. Тогда заключенный спасен. Этой же стратегии придерживаются все заключенные.

Заходит заключенный №2. Открывает коробку №2, если находит там 2, то выходит, иначе открывает коробку с найденным номером и т.д. пока не найдет коробку со своим номером или не исчерпается 50 коробок.

Оценим вероятность спасения P(*), если каждый из заключенных воспользуется предложенной стратегией.

Если посмотреть сразу и на номера коробок и на надписи внутри коробок:

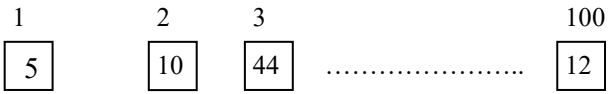


Рисунок 26 – Исходные данные для задачи 3.1

то увидим, что внутри коробок имеется случайная перестановка номеров коробок. Эта перестановка обладает некоторым свойством.

Предположим заключенный №1 открыл коробки.

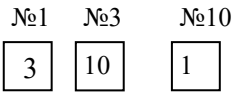


Рисунок 27 – Данные после открывания коробки

Этот объект называется циклом перестановок длиной 3.

Цикл длиной 1(неподвижной точкой) называется №1.

Следующее замечание тривиально: «заключенный №1 спасется тогда и только тогда, если его номер находится в цикле длиной не больше 50».

Итак, вероятность спасения $P(*) > P$ (в случайной перестановке на ста элементах нет циклов длиной больше 50). А это значит, что нет циклов длиной 51, длиной 52, длиной 53 и так до 100. Это есть объединение событий. Но эти события не совместны, т.е. не может быть одновременно, что есть цикл длиной 51, есть цикл длиной 52, …. Поэтому напишем не вероятность спасения, а вероятность гибели. Она равна

$$P_{гибели} = P(\frac{есть}{51}) + P(\frac{есть}{52}) + ... + P(\frac{есть}{100}).$$

Покажем, что $P(51) = \frac{1}{51}$; $P(52) = \frac{1}{52}$. .., $P(100) = \frac{1}{100}$,

т.е. $P_{гибели} = \sum_{i=51}^{100} \frac{1}{i} =$ [заменим сумму с учетом ошибки $\int \frac{1}{i}] \approx$.

$$\approx \ln 100 - \ln 51 \approx \ln 100 - \ln \frac{100}{2} = \ln 100 - \ln 100 + \ln 2 \approx 0,7,$$

$$P_{гибели} \approx 0,3 \Rightarrow P_{спасения} = 0,3.$$

Чем меньше людей, тем больше вероятность. Для n = 20, P = 0,5 и коробок тоже 20. Покажем, что вероятность того, что есть цикл длиной 51: $P(\frac{есть\_цикл}{51}) = \frac{1}{51}$.

Среди 100 элементов не могут быть два цикла длиной 51, цикл может быть только 1. Этот цикл будет на каких-то 51 элементах. Их надо выбрать $\binom{100}{51}$. Осталось 49 элементов. Элементы надо упорядочить каким-либо способом. Количество этих способов 49!, а количество способов упорядочить 100 элементов равно 100!, а для выбора 51 элемента из 100, которые дадут цикл длиной 51, количество способов равно 50!(не 51, т.к. есть точка отсчета, поэтому 51-1=50).

$$P(\frac{есть}{51}) = \frac{(C_{51}^{100}) \cdot 49! \cdot 50!}{100!} = \frac{100! \cdot 49! \cdot 50!}{51! \cdot 49! \cdot 100!} = \frac{50!}{51!} = \frac{1}{51},$$

где $C_{51}^{100} = \frac{100!}{51! \cdot 49!}$

Аналогично $P(\frac{есть}{52}) = \frac{1}{52}$, ... Такие вычисления не верны для малых *n*, так как имеются другие вычисления.

3.2 Сумасшедший пассажир

В самолете места пронумерованы от 1 до 100. В очереди стоят 100 пассажиров. Каждый пассажир заходит в салон самолета и садится на свое место. Но пассажир с билетом №1 не совсем нормальный, вместо того, чтобы сесть на свое место, он садится на любое место. И вероятность того, что он сядет на *i*-е место, равна $\frac{1}{100}$. Все остальные пассажиры хотят сесть на свое место. Но один из пассажиров не сможет этого сделать, т.к. его место займет сумасшедший пассажир. Поэтому он садится на любое место из оставшихся свободных мест и т.д. процесс продолжается пока все пассажиры не усядутся. Какова вероятность того, что сотый пассажир сядет на свое место: $P(X_{100} = 1) - ?$

1) пусть имеется всего 2 пассажира, тогда вероятность $P = \frac{1}{2}$;

2) для $N = 3$: если I пассажир сядет на свое место, то третий сядет на свое место. Но если I пассажир сядет на №2, тогда III пассажир сядет на свое место тогда и только тогда, когда II-й не сядет на №3. Но свободных мест 2, значит

$$P = \frac{1}{3} + \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{2}$$

3) для $N = 4$; $P = \frac{1}{2}$.

Для двух случаев: когда сумасшедший пассажир садится №1 или №100 сразу ясно будет ли сотый пассажир сидеть на своем месте.

А если I пассажир сядет на место №*i*? Тогда все пассажиры от 2 до *i*-1 сядут на свои места. Тогда *i*-й пассажир должен найти себе место. Если он выбрал №1 или №100, то опять все понятно. А если он выбрал №*k*? И т.д.

Всегда имеется 2 пути: сотый пассажир будет сидеть на №100 или нет, т.е. $P(X_{100}=1)=P(X_{100}=0)=\frac{1}{2}$. Это было неформальное решение.

Формально: распишем вероятность:

$$N_{100}=\frac{1}{100}+\frac{1}{100}[\quad]+\frac{1}{100}[\quad]+\ldots\ldots\ldots+\frac{1}{100}\cdot 0;$$

I пас сядет на №1	I пас сядет на №2	I пас сядет на №3	I пас сядет на №100
	Вероятность того, что последний пас. сядет на свое место при условии, что I пас сел на №2.	Вер-ть того, что последний пас. сядет на свое место при условии, что I пас сел на №3.	Т.к. 100 пас. не сможет сесть на №100

$$N_{100}=\frac{1}{100}+\frac{1}{100}\cdot N_{99}+\frac{1}{100}\cdot N_{98}+\frac{1}{100}\cdot N_{97}+\ldots$$

$$N_{100}=\frac{1}{100}+\frac{1}{100}\cdot\sum_{i=99}^{2}N_i$$

Осталось доказать, что $N_i=\frac{1}{2}$ для $\forall i>1$. $N_2=N_3=\frac{1}{2}$ уже доказано ранее.

Остальное докажите самостоятельно по индукции.

3.3 Неслучайные случайности

Дан интервал целых чисел [*A, B*] и список чисел из этого интервала $a_1, a_2, ...a_m$. Написать программу, которая вернет случайное равновероятное число *x* из [*A; B*], но не принадлежит списку $a_1, a_2, ...a_m$.

Пример: *A* = 0, *B* = 10; список: 0, 2, 4, 6, 8, 10.

Ответ: $1 \vee 3 \vee 5 \vee 7 \vee 9$. $P = \frac{1}{5}$.

Возможно, от того как соотносятся *m* и *n,* будет зависеть оптимальность решения. Если *n* много меньше *m*, то может быть один алгоритм, *n* много больше *m* – другой алгоритм.

Алгоритм может быть вероятностным или детерминированным.

Рассмотрим четыре решения этой задачи:

1) неверное;
2) вероятностно неоптимальное;
3) вероятностно оптимальное;
4) детерминированное.

Вместо того, чтобы рассмотреть [*A, B*] может быть рассмотрено [0…*n*-1]. Здесь *n* чисел. Попробуем визуализировать задачу.

Решение 1. Сгенерируем $x \in [0; n-1]$. Проверим $x \in a_1, a_2, ..., a_m$?

```
если x∈списку исключений
   вернем следующий x∈списку
иначе
        return x.
```

Но *x* будет возвращено не с одинаковой вероятностью.

Решение 2. Пусть *x* – случайное число, $\in [0, n-1]$;

```
if x∈списку исключений
        вернуться к генерации случайного числа
else
        return x;
```

Замечания:

1) как проверить принадлежность *X*:

а) бинарный поиск, если *list* отсортирован;

б) последовательный поиск, если *list* не отсортирован.

2) *x* равновероятно выбран из [0; *n*-1];

3) время работы: количество генераций случайного числа вероятностное.

Время работы равно $\frac{n}{n-m}$ - количество генераций.

Если $m<<n$, т.е. m мало, т.е. мал размер списка исключений, то количество генераций ≈ 1. Время работы стремится к 1.

Если $m \approx n$. Например $m = 0{,}9 \cdot n$, то генерировать придется 10 раз. Это хуже. Оценим время для случаев *а* и *б* пункта 1.

а

с сортировкой

$$m \cdot \log m + \frac{n}{n-m} \cdot \log m$$

сортир-ка	генерация чисел	проверка $\in$-ти

б

без сортировки

$$\frac{n}{n-m} \cdot m$$

генерация чисел	количество сравнений

Чтобы ответить на вопрос: когда какой способ лучше – необходимо решить неравенство $m \cdot \log m + \frac{n}{n-m} \cdot \log m < \frac{n}{n-m} \cdot m$

Решение 3 (наивное). Можно сгенерировать массив из тех чисел, которые нужны, а затем из него выбрать случайно и равновероятно одно число. По памяти: нужен дополнительный массив, поэтому $\theta(n+m)$. По времени: необходимо построить массив $\theta(n+m)$.

Решение 4. Подходящих чисел (*n*-*m*) штук. Т.е. можно сгенерировать число $x \in [0; n-m]$. Допустим, *x* попало в шестую клетку, т.к. исключили все неправильные числа, но они в исходном массиве есть, то нам необходимо будет пройти слева направо, посчитать, сколько чисел запрещены и перейти к *x*, которое сгенерировали. После этого может получиться так, что опять появятся запрещенные числа и т.д.

Рассмотрим это на примере: $x = Rand[0...5]$. Пусть *x=4*. Пройдемся только по тем числам, которые запрещены *n* < 4: 1, 2, 3 (всего 3 числа). Значит, к 4 надо прибавить 3, это 7. Но пока дойдем до 7, встретим еще одно число, которое запрещено (5). Т.е. к 7+1=8. Вернем 8 – случайное равновероятное число.

3.4 Муравьи на палке

По палке бегают несколько идентичных муравьев с одинаковой постоянной скоростью. Когда муравей добегает до края палки, он падает с нее. Когда два муравья встречаются, они мгновенно разворачиваются и начинают бежать в противоположных направлениях. Через какое максимальное время все муравьи упадут с палки? (или, в конкретном варианте задачи: если всю палку один муравей пробегает за 1 минуту, докажите, что через минуту все муравьи упадут с палки).

Решение. Ключевой момент в решении этой задачи заключается в следующем: поскольку все муравьи идентичны, то момент их столкновения и разворота неотличим от ситуации, если бы они просто прошли сквозь друг друга, продолжив движение в своих первоначальных направлениях. Индивидуальность муравьев не имеет значения для конечного результата: необходимо найти только общее время, когда палка опустеет. Таким образом, задачу можно переформулировать: «Через какое максимальное время все муравьи упадут с палки, если они просто проходят сквозь друг друга?».

Максимальное время определяется тем, как долго может оставаться на палке муравей, который изначально находился дальше всех от ближайшего к нему края, и движется в сторону этого края. Если самый «неудачно» расположенный муравей (тот, которому

нужно пробежать всю длину палки, чтобы упасть) может пробежать всю палку за 1 минуту (как в примере из поиска), то максимальное время, через которое все муравьи упадут, составит 1 минуту.

Максимальное время, через которое все муравьи упадут с палки, равно времени, которое требуется одному муравью, чтобы пробежать всю длину палки. Это время не зависит от количества муравьев и их начального расположения.

3.5 Разборчивая невеста (Привередливый программист)

К девушке пришли свататься 5 человек, у каждого есть некоторые признаки и их «стоимость».

Таблица 7 – Элементы и их стоимость

x_1	x_2	x_3	……	x_n
c_1	c_2	c_3	……	c_n

Все стоимости разные. $\forall i \neq j,\ c_i \neq c_j$. Диапазон c не известен. Невеста хочет выйти замуж за лучшего принца. Принцы приходят по одному, а она должна или принять или отвергнуть. Если принцесса отвергла претендента, то он больше никогда к ней не приходит.

Решение. Если принцесса выберет *i*-го принца, то *stop*. А если *(i+1)*-й принц лучше *i*-го? То есть необходимо разработать стратегию, максимизировав шанс принцессы остановиться на лучшем кандидате.

Цель: алгоритм максимизации шанса принцессы выбрать лучшего принца.

Эту задачу можно решить с помощью динамического программирования. Ввести целевую функцию *F*, которая зависит от того, на каком шаге находимся, и где находится лучший вариант. Аргументы функции *F* – шаг и лучший кандидат. Однако найдем решение без динамического программирования.

Стратегия 1. Взять кандидата с №5. Вероятность $P = \frac{1}{n}$, где *n* - количество кандидатов.

Стратегия 2. Принцесса пропускает №1, а потом берет первого попавшегося со стоимостью $> C_1$.

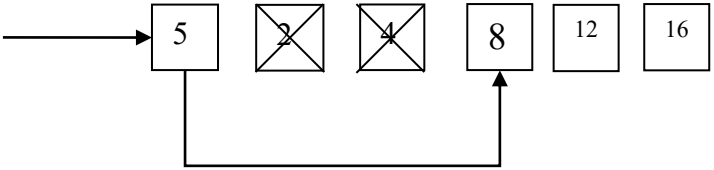


Рисунок 28 – Данные после того, как принцесса пропускает №1, а потом берет первого попавшегося

Пусть эта стратегия называется $\propto_1$. Найдем вероятность того, что в $\propto_1$ принцессе все же попадется лучший принц *P(A)*.

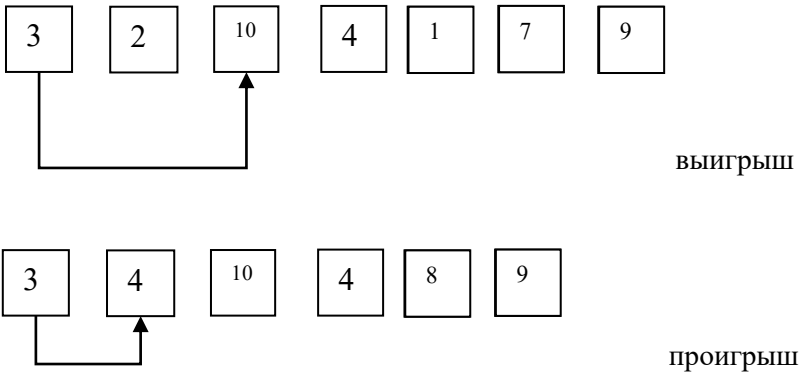


Рисунок 29 – Ситуация в задаче в случае выигрыша и проигрыша

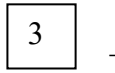
$P(A) = \sum_{i=1}^{n} P(A|x = i) \cdot P(x = i)$, при условии, что лучший принц находится на месте *i*

$P(x = i)$ – вероятность того, что *x=i,* т.е. что лучший принц находится на месте *i*

Если *i* = 1, то вероятность успеха $P = \frac{1}{n} \cdot 0 = 0.$

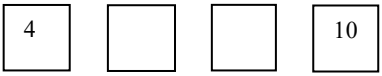
$i = 2$, вероятность выигрыша $P = \frac{1}{n} \cdot 1 = \frac{1}{n}$.

i=3:

Выбор принцесса выберет лишь в том случае, если принц, который стоит между 3 и 10, будет стоить меньше 3, а если больше, то принцесса его выберет, но он не лучший. Это значит, что из двух чисел 

принцесса выиграет, если второе число меньше 3. Но вероятность этого $\frac{1}{2}$, следовательно, для $i = 3$ вероятность выигрыша равна $P = \frac{1}{n} \cdot \frac{1}{2}$.

$i = 4$:



1 2 3 4

Когда принцесса остановится на 10? только если стоимость 2, 3 принцев будет меньше 4. Вероятность того, что стоимость первого принца будет меньше стоимости и 2 и 3 равна $\frac{1}{3}$.

Всего 3 кандидата. Вероятность того, что третий будет максимальной стоимости равна $\frac{1}{3}$. Т.е. при $i = 4$ вероятность выигрыша равна $P = \frac{1}{n} \cdot \frac{1}{3}$. Значит:

$$P(A) = \frac{1}{n} \cdot 0 + \frac{1}{n} \cdot 1 + \frac{1}{n} \cdot \frac{1}{2} + \frac{1}{n} \cdot \frac{1}{3} + \cdots$$

$P(A) = \sum_{i=1}^{n} \frac{1}{i}$ – это сумма геометрической прогрессии, она $\approx \frac{\lg n}{n}$

Если n = 1000, то $P(A) = \frac{\log(1000)}{1000} = \frac{7}{1000}$, что в 7 раз больше стратегии 1, где $P = \frac{1}{1000}$.

Стратегия 3. α_2

Пропустим первого кандидата, выберем следующего со стоимостью больше выбранного на первом шаге. $P(A) = \sum_{i=1}^{n} P(A|x = i) \cdot P(x = i)$. Если лучший принц будет на №1 или №2, то вероятность выигрыша равна 0.

i = 1: $P = \frac{2}{n} \cdot 0$

i = 2: $P = \frac{2}{n} \cdot 0$

i = 3: вероятность выигрыша $\frac{1}{n} \cdot 1$

i = 4:

Если $\max(c_1, c_2) > c_3$, то выберем №4. То есть $P = \frac{1}{n} \cdot \frac{2}{3}$.

i = 5: $P = \frac{1}{n} \cdot \frac{2}{4}$

Т.е. получим: $P = \frac{2}{n} \cdot 0 + \frac{1}{n} \cdot 1 + \frac{1}{n} \cdot \frac{2}{3} + \frac{1}{n} \cdot \frac{2}{4} + \ldots \approx \frac{1}{n} \cdot 2 \cdot \log n$.

Для α_2: $P = 2 \cdot \frac{\log n}{n}$, то есть $P\ \alpha_2 > P\ \alpha_1$ в 2 раза. То есть вероятности стратегий имеют вид:

α_1	α_2	α_3	α_4	…	α_n
$\frac{\log n}{n}$	$2 \cdot \frac{\log n}{n}$			…	$\frac{1}{n}$

Найдем i, такое что $arg\ max\ \alpha_k$, будет максимальным.

Таблица 8 – Стратегии и их вероятности

α_1	α_2	α_3	α_4
0,2	0,3	0,5	0,2

Найдем i, при котором вероятность максимальна. Найдем α_i в общем виде.

$$\alpha_i = P(A_i) = \sum_{j=1}^{n} \mathrm{P}(\mathrm{A_i} \mid X = j) \cdot \mathrm{P}(X = j)$$

$\mathrm{P}(X = j) = 1/n$, где j – место, где находится лучший принц. Если $j < i$, то $\mathrm{P}(\mathrm{A_i} \mid X = j) = 0$. Пусть $j > i$, тогда первых i кандидатов принцесса пропускает. Принцесса останавливается на j-м принце, если максимальный из первых $(j\text{-}1)$ принцев находится среди первых i принцев.

Вероятность успеха равна вероятности того, что максимум будет находиться в первой части и она равна $\frac{i}{j-1}$. Итак:

$$P = \sum_{j=i+1}^{n} \frac{i}{j-1} \cdot \frac{1}{n} = \frac{1}{n} \cdot i \cdot \sum_{j=i+1}^{n} \frac{i}{j-1} = \frac{1}{n} \cdot (\lg(n) - \lg(i))$$

Теперь надо найти i, который дает максимум функции $lg\ i$. Найдем f'_i, приравняем к нулю, и найдем: $i = \frac{n}{e}$. Т.е. надо пропустить $\frac{n}{e}$ кандидатов, а $\left\lceil i = \frac{n}{e} \right\rceil$ – ответ.

4 ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ

4.1 Классические задачи

Ниже приведены формулировки задач, которые можно решить с помощью жадных алгоритмов.

1. Задача о выборе заявок (*Activity Selection Problem*). Дано множество мероприятий с временем начала и окончания. Необходимо выбрать максимальное количество непересекающихся по времени мероприятий, которые можно посетить.

2. Задача о выборе мероприятий. Аналогично предыдущей, но максимизируется общая «ценность» (вес) выбранных мероприятий, если у каждой заявки есть вес.

3. Задача о рюкзаке (*Fractional Knapsack Problem*). Дано множество предметов с весом и стоимостью. Необходимо заполнить рюкзак максимальной стоимости, если предметы можно делить (брать их части).

4. Задача о покрытии множества (*Set Cover Problem*). Дано универсальное множество и набор подмножеств. Необходимо найти минимальное количество подмножеств, объединение которых равно универсальному множеству (жадный алгоритм дает приближенное решение).

5. Задача о минимальном покрывающем дереве (*Minimum Spanning Tree*). Найти дерево с минимальной суммой весов ребер, которое соединяет все вершины графа (Алгоритмы Прима и Краскала).

6. Задача о кратчайшем пути от одной вершины до всех остальных с неотрицательными весами (*Dijkstra's Algorithm*). Найти кратчайшие пути от заданной вершины до всех остальных в графе с неотрицательными весами ребер.

7. Задача о размене монет (*Coin Change Problem*). Дана система денежных купюр/монет (обычно стандартная, например, 1, 5, 10, 50, 100). Необходимо выдать заданную сумму минимальным количеством монет.

8. Задача планирования заданий с крайним сроком (*Job Sequencing with Deadlines*). Даны задания, каждое с крайним сроком и прибылью. Необходимо выбрать подмножество заданий, которые можно выполнить до срока, чтобы максимизировать общую прибыль.

9. Задача о выборе банкоматов (*Facility Location Problem* – упрощенная). Выбрать минимальное количество мест для установки банкоматов, чтобы каждый житель был в пределах определенного расстояния от банкомата.

10. Кодирование Хаффмана (Huffman Coding). Построить оптимальное префиксное кодирование для набора символов с заданными частотами появления, чтобы минимизировать среднюю длину кода.

11. Задача о заполнении контейнеров. Есть контейнер ограниченной вместимости и набор предметов разного веса. Уложить максимальное количество предметов в контейнер.

12. Задача о выборе минимального числа контейнеров. Даны предметы разного веса и контейнеры фиксированной грузоподъемности.

13. Задача о выборе съемочных групп (*Scheduling tasks with deadlines and profits on multiple machines)*. Распределить съемочные группы по локациям для максимизации общего числа отснятых сцен с учетом временных окон доступности.

14. Задача о ледоколе. Распределить ледоколы для прокладывания маршрутов с наименьшими затратами.

15. Задача об оптимальном хранении на магнитной ленте. Набор программ разного размера нужно записать на ленту так, чтобы минимизировать среднее время доступа к программам (упорядочивание по размеру).

4.2 Задачи по работе с интервалами и сортировкой

1. Объединение интервалов. Дано множество интервалов. Найти минимальное количество интервалов, которые нужно объединить, чтобы покрыть исходные.

2. Удаление интервалов. Дано множество интервалов. Найти минимальное количество интервалов, которые нужно удалить, чтобы оставшиеся не пересекались.

3. Сортировка задач по времени завершения (*Earliest Finish Time First*). При планировании на одном процессоре.

4. Минимизация опозданий (*Minimizing Lateness*). Планирование заданий на одном процессоре для минимизации максимального опоздания.

4.3 Задачи на массивах и строках

1. Задача о заправках (*Gas Station Problem*). Минимизация количества остановок для заправки по пути с ограниченным запасом топлива.

2. Задача о расстановке заправочных станций. На маршруте известны расстояния между городами. Найти минимальное количество заправок, чтобы можно было проехать весь маршрут, учитывая запас хода автомобиля.

3. Максимальная сумма подмассива (*Kadane's Algorithm*). Поиск непрерывного подмассива с наибольшей суммой (использует жадный подход).

4. Сумма двух чисел из массива, ближайшая к нулю.

5. Выбор наибольшего числа. Дана строка цифр и число *k*. Удалить *k* цифр, чтобы оставшееся число было наибольшим.

6. Задача о соединении отрезков веревки/кабеля. Даны длины отрезков, объединять их парами с минимальными затратами на каждом шаге.

7. Балансировка нагрузки (*Load Balancing* – упрощенная). Распределение задач по нескольким процессорам для минимизации максимальной загрузки.

8. *Merge k sorted lists* (слияние *k* отсортированных списков). Используя min-heap (приоритетную очередь).

9. Поиск наибольшего общего префикса в массиве строк.

4.4 Прочие прикладные задачи

1. Создание наибольшего/наименьшего числа из данного набора цифр.

2. Задача о расстановке охраны (*Guarding an Art Gallery – 1D*). Найти минимальное количество охранников для покрытия всех участков одномерной галереи.

3. Задача о минимальном количестве стрел для уничтожения шаров. Если шары представлены горизонтальными интервалами на оси.

4. Поиск медианы в потоке данных (*Median maintenance*). Используя две кучи (приоритетные очереди).

5. Задача о максимальном количестве платформ для поездов. Найти минимальное количество платформ, необходимых для обслуживания всех поездов с данными временами прибытия и отправления.

6. Задача о покрытии точками на прямой. Даны точки на прямой, нужно найти минимальное количество отрезков заданной длины, чтобы покрыть все точки.

7. Построение пермутации для максимальной/минимальной суммы.

8. Нахождение минимальной разницы между любыми двумя элементами в отсортированном массиве.

9. Создание расписания экзаменов без накладок (*Approximation*). Жадное раскрашивание графа.

10. Задача о минимальном количестве нажатий кнопок для получения числа на калькуляторе (если операции позволяют жадный подход).

11. Максимальное количество платформ для поездов. На железнодорожной станции задано время прибытия и отправления поездов. Найти минимальное количество платформ, необходимых для их обслуживания.

12. Задача о прыжках кузнечика. Найти минимальное количество прыжков, чтобы добраться до конца ряда клеток с разной длиной возможных прыжков.

13. Задача о покупке билетов (минимальное ожидание). Даны люди в очереди, каждый из которых тратит разное время на обслуживание. Как их расставить, чтобы минимизировать суммарное время ожидания?

14. Задача о конференц-залах: Максимизация числа встреч в одном конференц-зале (аналог задачи о выборе заявок).

15. Поиск максимального подмассива (алгоритм Кадана). Найти непрерывный подмассив с максимальной суммой в одномерном массиве.

16. Задача о рюкзаке 0/1 (приближенно). Для некоторых частных случаев (например, если веса и ценности коррелируют). В общем виде не решается жадно.

17. Задача остовного дерева с ограничениями. Взвешенный граф, найти остовное дерево, удовлетворяющее дополнительным критериям (иногда возможно жадное приближение).

18. Задача о наименьшем количестве стрел для уничтожения шаров. На горизонтальной оси даны диаметры и координаты шаров. Каждая стрела летит вертикально вверх и может пробить все шары, которые находятся на ее пути. Найти минимальное количество стрел.

19. Назначение задач исполнителям. Распределить задания между исполнителями для минимизации общего времени завершения всех задач (в определенных конфигурациях).

20. Задача о дележе конфет. Распределить конфеты среди детей так, чтобы каждый получил хотя бы одну, а суммарный «уровень счастья» был максимальным (при определенных условиях).

21. Задача о выборе рекламных баннеров. Максимизировать кликабельность, выбирая баннеры с наибольшим текущим рейтингом на каждом шаге.

22. Задача о составлении расписания: Оптимизация расписания занятий для минимизации конфликтов.

23. Покрытие точками. На прямой заданы отрезки, нужно выбрать минимальное число точек, чтобы каждый отрезок содержал хотя бы одну точку.

24. Задача о минимальном количестве камер наблюдения. На прямой расположены дома, нужно расставить камеры так, чтобы все дома были под наблюдением с минимальным числом камер.

25. Задача о паре с максимальной суммой, не превышающей K. Найти два элемента в массиве, сумма которых максимальна и при этом меньше или равна заданному числу K.

26. Задача о максимальной сумме подпоследовательности (с ограничениями). Выбор элементов из последовательности для максимизации суммы с определенными ограничениями.

27. Выбор точек для покрытия окружностью. Найти минимальное количество окружностей заданного радиуса, чтобы покрыть все заданные точки на плоскости.

28. Минимальное количество прыжков для достижения цели в массиве. В массиве указано, как далеко можно прыгнуть из каждой позиции.

29. Задача о дождевателях. На поле даны координаты дождевателей и их радиусы, нужно выбрать минимальное число дождевателей, чтобы полить все поле.

ЗАКЛЮЧЕНИЕ

Представленный практикум по решению задач по дисциплине «Алгоритмы обработки данных» охватывает аспекты разработки, анализа и применения фундаментальных алгоритмических подходов. Основная цель практикума заключалась в закреплении теоретических знаний, полученных в ходе изучения дисциплины, и формировании устойчивых практических навыков их применения для решения реальных вычислительных задач.

В ходе выполнения заданий студенты имеют возможность освоить жадную стратегию разработки алгоритмов. Были рассмотрены и практически реализованы большое количество задач с применением структур данных, таких как массивы, списки, стеки, очереди, деревья и графы.

Решение разнообразных задач позволит развить системное мышление и способность критически оценивать применимость конкретного алгоритма для достижения поставленной цели. Особое внимание было уделено эффективности и оптимизации реализованных решений.

Практикум позволит не только глубоко понять принципы работы алгоритмов обработки данных, но и подготовить студентов к решению инженерных и исследовательских задач в области информационных технологий. Навыки, приобретенные в рамках данного курса, являются фундаментом для дальнейшего профессионального роста и успешной работы над проектами, требующими высокой производительности и надежности программного обеспечения.

ЛИТЕРАТУРА

1. Бойков В.А. О применении жадных алгоритмов в некоторых задачах дискретной математики // Программные продукты и системы. 2019. Т. 32. № 1. С. 55–62.

2. Галатенко В. В., Лившиц Е. Д. Обобщенные приближенные слабые жадные алгоритмы //Матем. заметки, 78:2 (2005), 186–201 URL: https://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=mzm&paperid=2581&option_lang=rus

3. Дасгупта С., Пападимитриу Х., Вазирани У. Алгоритмы; [пер. с англ.; под ред. А. Шеня]. М.: МЦНМО, 2014. 320 с.

4. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. – 3-е изд. – М.: Вильямс, 2013. – 1328 с.

5. Лившиц Е. Д. О нижних оценках скорости сходимости жадных алгоритмов, Изв. РАН. Сер. матем., 2009, том 73, выпуск 6, 125–144 URL: https://www.mathnet.ru/im2742

6. Макконнелл Дж. Основы современных алгоритмов. 2-е изд., доп. – М.: Техносфера, 2004. – 368 с.

7. Новиков Ф.А., Поздняков С. Н. Жадные алгоритмы // Компьютерные инструменты в образовании. 2005. № 2. URL: https://cyberleninka.ru/article/n/zhadnye-algoritmy (дата обращения: 01.11.2025).

8. Рафгарден Т. Совершенный алгоритм. Жадные алгоритмы и динамическое программирование. – СПб.: Питер, 2020. – 253 с.

9. Трауб Дж., Вожняковски Х. Общая теория оптимальных алгоритмов. – М.: Мир, 1983. – 384 с.

10. Шень А. Программирование: теоремы и задачи. – 7-е изд., дополненное.– М.: МЦНМО, 2021. – 320 с.

Учебное издание

Кирсанова Алла Владимировна

ПРАКТИКУМ ПО РЕШЕНИЮ ЗАДАЧ ПО ДИСЦИПЛИНЕ
«АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ»

Учебно-практическое пособие

Компьютерная верстка *А. Н. Федоренко*

ИЛ № 06150. Сер. АЮ от 21.02.02.
Подписано в печать 13.02.26. Формат 60 × 90/16.
Усл. печ. л. 4,375. Электронное издание. Заказ № 805.

Изд-во Приднестр. ун-та. 3300, г. Тирасполь, ул. Мира, 18.