МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»

А.А. Максутов

Методические материалы по программе «Современные методы решения задач в программировании»

Москва 2016

Максутов А.А. Методические материалы по программе «Современные методы решения задач в программировании». – М.: НИЯУ МИФИ, 2016. – 40 с.

В настоящих методических материалах рассмотрены основные методы решения задач различной сложности и даны рекомендации по применению каждого из рассмотренных методов на примере решения задач единого государственного экзамена (ЕГЭ) уровней В и С. Методические материалы предназначены как для учителей школ, осуществляющих подготовку выпускников к сдаче ЕГЭ по информатике, так и для школьников в качестве справочного пособия при самостоятельной подготовке.

СОДЕРЖАНИЕ

Введение

Несмотря на обилие инструментов, представленных в виде разнообразных языков программирования, современных сред разработки и нескончаемых баз знаний, поиск оптимальных решений нетривиальных задач всегда остается основой работы разработчика ПО.

При этом, некоторые задачи, если не требуют её глубокого математического анализа, то вынуждают применить несколько методов решения задач, прежде чем будет найден оптимальный вариант. За всё время существования кибернетики, как науки, таких методов было разработано достаточное количество, чтобы, овладев ими, решить практически любую прикладную задачу за разумные сроки. Далее мы рассмотрим теоретические основы, необходимые для освоения и сравнения эффективности применения методов, а также рассмотрим применение этих методов на практике.

Сложность алгоритмов

Многие алгоритмы предлагают выбор между объёмом памяти и скоростью. Задачу можно решить быстро, используя большой объём памяти, или медленнее, занимая меньший объём.

Типичным примером в данном случае служит алгоритм поиска кратчайшего пути. Представив карту города в виде сети, можно написать алгоритм для определения кратчайшего расстояния между двумя любыми точками этой сети. Чтобы не вычислять эти расстояния всякий раз, когда они нам нужны, мы можем вывести кратчайшие расстояния между всеми точками и сохранить результаты в таблице. Когда нам понадобится узнать кратчайшее расстояние между двумя заданными точками, мы можем просто взять готовое расстояние из таблицы.

Результат будет получен мгновенно, но это потребует огромного объёма памяти. Карта большого города может содержать десятки тысяч точек. Тогда, описанная выше таблица, должна содержать более 10 млрд. ячеек. Т.е. для

того, чтобы повысить быстродействие алгоритма, необходимо использовать дополнительные 10 Гб памяти.

Из этой зависимости проистекает идея объёмно-временной сложности. При таком подходе алгоритм оценивается, как с точки зрении скорости выполнения, так и с точки зрения потреблённой памяти.

Мы будем уделять основное внимание временной сложности, но, тем не менее, обязательно будем оговаривать и объём потребляемой памяти.

Оценка порядка

Время выполнения того или иного алгоритма складывается из целого множества факторов: время инициализации переменных, ввод и подготовка входных данных, количество и организация циклов, а также количество итераций каждого из них. При этом 2 разных по длине массива данных при использовании одного алгоритма обработки могут выполняться 1 и 10 секунд соответственно, а при использовании другого – 2 и 5 секунд соответственно. Таким образом, при оценке сложности выполнения того или иного алгоритма во внимание берется зависимость количества операций от объема входных данных.

Особенно хорошо подобная ситуация прослеживается при написании алгоритмов параллельной обработки информации, когда подготовка данных к параллельному вычислению может занимать б**о**льшую часть времени работы программы.

В общем случае наиболее сложными частями программы, обычно, является выполнение циклов и процедур. При этом сложность алгоритма можно оценить по порядку величины. Алгоритм имеет сложность O(f(n)), если при увеличении размерности входных данных N, время выполнения алгоритма возрастает с той же скоростью, что и функция f(N). Рассмотрим код, который находит максимальный элемент в каждой строке матрицы A[NxN]:

```
1   for i in xrange(N):
2       max = 0
3       for j in xrange(N):
4           if max < A[i][j]:
5               max = A[i][j]
6       print max
```

В этом алгоритме переменная i меняется от 1 до N. При каждом изменении i, переменная j тоже меняется от 1 до N. Во время каждой из N итераций внешнего цикла, внутренний цикл тоже выполняется N раз. Общее количество итераций внутреннего цикла равно N*N. Это определяет сложность алгоритма $O(N^2)$.

Вызов процедур или функций не может интерпретироваться однозначно, когда мы говорим о сложности алгоритма. Если процедура выполняет какое-либо определенное число действий, то её сложность оценивается, как O(1). Однако, если в процедуре содержится цикл, количество итераций которого зависит от некоторого параметра n, то и сложность в данном случае оценивается, как O(f(n)). При этом, если вызов процедуры происходит внутри цикла, то сложность процедуры умножается на сложность, которая формируется циклом. Если же процедура вызывается вне цикла, то её сложность складывается со сложность остальной части программы/процедуры. Рассмотрим на примере:

```
 1  def fast(n):
 2      for i in xrange(n):
 3          for j in xrange(n):
 4              do_something()
 5
 6  def slow(n):
 7      for i in xrange(n):
 8          for j in xrange(n):
 9              for k in xrange(n):
10                  do_something()
11
12  fast(N)
13  slow(N)
```

Для начала определим сложности функций fast и slow. Очевидно, что сложность функции fast $O(n^2)$, а slow – $O(n^3)$. Эти функции вызываются

последовательно(строки 12-13). При этом их сложности складываются, образуя общую сложность $O(n^3) + O(n^2) = O(n^3)$. Заметьте, что сложность функции slow на порядок больше, чем сложность функции fast, поэтому сложностью последней при сложении мы можем пренебречь. Рассмотрим пример с вложенным вызовом функции:

```
 1  def fast(n):
 2      for i in xrange(n):
 3          for j in xrange(n):
 4              slow(n)
 5
 6  def slow(n):
 7      for i in xrange(n):
 8          for j in xrange(n):
 9              for k in xrange(n):
10                  do_something()
11
12  fast(N)
```

В данном случае на каждой итерации цикла по переменной j функции fast вызывается функция slow. При этом сложности этих функций перемножаются $O(n^3) * O(n^2) = O(n^5)$.

Задача обработки потоков данных.

Рассмотрим следующую задачу:

В физической лаборатории проводится долговременный эксперимент по изучению гравитационного поля Земли. По каналу связи каждую минуту в лабораторию передаётся положительное целое число – текущее показание прибора «Сигма 2015». Количество передаваемых чисел в серии известно и не превышает 10 000. Все числа не превышают 1000. Временем, в течение которого происходит передача, можно пренебречь.

Необходимо вычислить «бета-значение» серии показаний прибора – минимальное чётное произведение двух показаний, между моментами передачи которых прошло не менее 6 минут. Если получить такое произведение не удаётся, ответ считается равным –1.

Напишите программу для решения поставленной задачи, которая будет эффективна как по времени, так и по памяти (или хотя бы по одной из этих характеристик).

Программа считается эффективной по времени, если время работы программы пропорционально количеству полученных показаний прибора N, т.е. при увеличении N в k раз время работы программы должно увеличиваться не более чем в k раз.

Программа считается эффективной по памяти, если размер памяти, использованной в программе для хранения данных, не зависит от числа N и не превышает 1 килобайта.

Перед программой укажите версию языка программирования и кратко опишите использованный алгоритм.

Входные данные представлены следующим образом. В первой строке задаётся число N – общее количество показаний прибора. Гарантируется, что $N > 6$. В каждой из следующих N строк задаётся одно положительное целое число – очередное показание прибора.

Пример входных данных:

```
11
12
45
5
3
17
23
21
20
19
18
17
```

Программа должна вывести одно число – описанное в условии произведение либо –1, если получить такое произведение не удаётся.

Пример выходных данных для приведённого выше примера входных данных:

54

Разбор задачи обработки потоков данных

Первая идея, которая приходит в голову при решении этой задачи – решение полным перебором с предварительной записью элементов в массив. И действительно, это решение очень просто реализовать:

```
 1 N = int(input())
 2 a=[]
 3 for i in xrange(N):
 4     a.append(int(input()))
 5
 6 m = a[0]*a[6]
 7 for i in xrange(N-6):
 8     for j in xrange(i+6, N):
 9         if a[i]*a[j] < m and a[i]*a[j]%2 == 0:
10             m = a[i]*a[j]
11
12 if m%2!=0:
13     print -1
14 else:
15     print m
```

Однако, такое решение не является оптимальным ни по времени(O(n*log(n))), ни по используемой памяти(O(n)). Для оптимального решения этой задачи необходим инструмент, который бы позволил реализовать движение “кадра”, которое реализовано корректировкой диапазонов изменения счетчиков i и j.

Очередь и стек

Очередь — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, First In — First Out). Добавление элемента (принято обозначать словом enqueue — поставить в очередь) возможно лишь в конец очереди, выборка — только из начала

очереди (что принято называть словом dequeue — убрать из очереди), при этом выбранный элемент из очереди удаляется.

Очередь в программировании используется, как и в реальной жизни, когда нужно совершить какие-то действия в порядке их поступления, выполнив их последовательно. Примером может служить организация событий в Windows. Когда пользователь оказывает какое-то действие на приложение, то в приложении не вызывается соответствующая процедура (ведь в этот момент приложение может совершать другие действия), а ему присылается сообщение, содержащее информацию о совершенном действии, это сообщение ставится в очередь, и только когда будут обработаны сообщения, пришедшие ранее, приложение выполнит необходимое действие.

Клавиатурный буфер BIOS организован в виде кольцевого массива, обычно длиной в 16 машинных слов, и двух указателей: на следующий элемент в нём и на первый незанятый элемент.

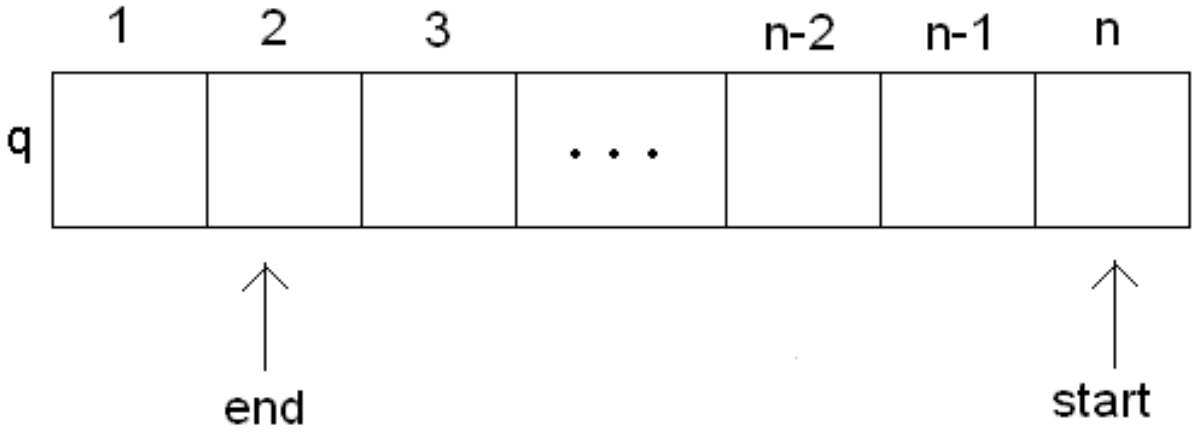


Рисунок 1 - Организация очереди на массиве

Стек (англ. stack — стопка; читается стэк) — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

В цифровом вычислительном комплексе стек называется магазином — по аналогии с магазином в огнестрельном оружии (стрельба начнётся с патрона, заряженного последним).

В 1946 Алан Тьюринг ввёл понятие стека. А в 1957 году немцы Клаус Самельсон и Фридрих Л. Бауэр запатентовали идею Тьюринга.

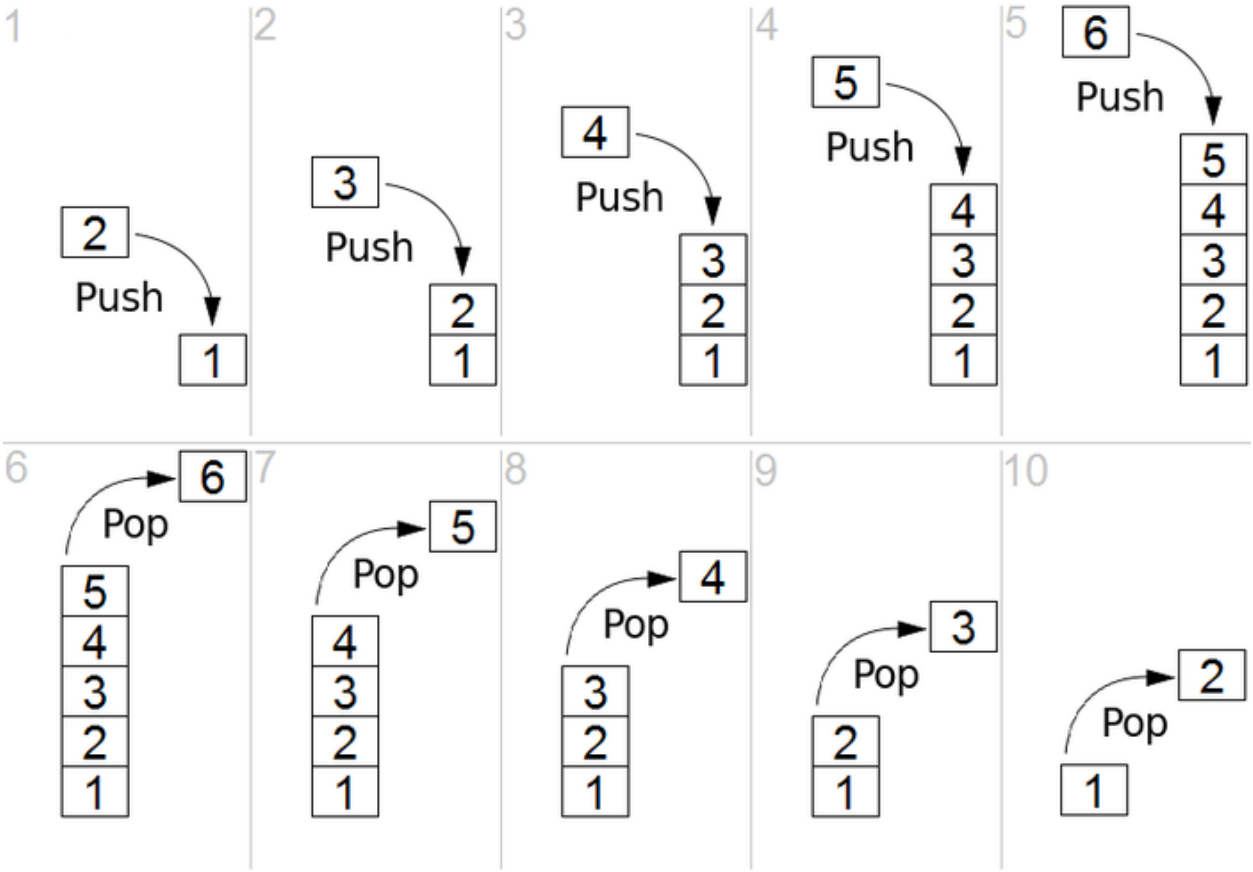


Рисунок 2 - организация стека с использованием одномерного массива

Предложенные структуры данных очень распространены в программировании и часто являются основой многих решений.

Альтернативное решение задачи обработки потока данных

Для обеспечения условия разницы во времени мы будем использовать очередь из 6-ти элементов. Очередь будет организована на основе статического массива. Запись в массив будет осуществляться циклически при помощи индекса, построенного на счетчике итераций. Для получения четного числа минимум один из множителей должен быть четным. Поэтому для получения минимального произведения необходимо будет хранить в памяти

минимальный четный и нечетный элемент последовательности. В общем случае алгоритм будет выглядеть следующим образом:

1. Инициализация массива первыми 6ю элементами последовательности
2. Цикл n-6 итераций:
    a. Извлечение i%6 элемента из очереди
    b. Сравнение с минимальным четным/нечетным элементом последовательности и, при необходимости, обновление соответствующего значения
    c. Запись нового элемента входного потока в очередь
    d. Определение четности и расчет минимального произведения с использованием нового элемента
    e. Обновление минимального произведения, если найден новый минимум
3. Вывод полученного результата.

Далее приведен исходный код, разработанный по данному алгоритму:

```
 1 N = int(input())
 2 a=[]
 3 for i in xrange(6):
 4     a.append(int(input()))
 5
 6 m0=0
 7 m1=0
 8 m=0
 9 for i in xrange(N-6):
10     if a[i%6]%2:
11         if m1:
12             if a[i%6] < m1:
13                 m1 = a[i%6]
14         else:
15             m1 = a[i%6]
16     else:
17         if m0:
18             if a[i%6] < m0:
19                 m0 = a[i%6]
20         else:
21             m0 = a[i%6]
22     a[i%6] = int(input())
23     if m==0:
```

```
24          m = m0 * a[i%6]
25      if m == 0 and a[i%6] % 2 == 0:
26          m = m1 * a[i%6]
27      if m0*a[i%6] < m:
28          m = m0*a[i%6]
29
30      if m1*a[i%6] < m and (m1*a[i%6])%2==0:
31          m = m1*a[i%6]
32
33
34  if m%2 != 0 or m == 0:
35      print -1
36  else:
37      print m
```

Сложность этой реализации по времени оценивается, как O(n), по памяти – O(1).

Структурное программирование

Первоначально идея структурного программирования появилась на свет в связи с оператором goto и сомнениями в целесообразности его применения. Впервые подобные сомнения высказал Хайнц Земанек (Heinz Zemanek) на совещании по языку Алгол в начале 1959 года в Копенгагене. Однако это выступление не привлекло к себе внимания и не имело последствий. Эдсгер Дейкстра (Edsger Dijkstra) вспоминает: «До некоторой степени я виню себя за то, что в то время не смог оценить значимость этой идеи».

Ситуация коренным образом изменилась через десять лет, когда в марте 1968 года Дейкстра опубликовал своё знаменитое письмо «Оператор Go To считается вредным» (Go To Statement Considered Harmful). Это поистине исторический документ, оказавший заметное влияние на дальнейшее развитие программирования.

Судьба самого документа очень интересна. Дело в том, что Дейкстра дал статье совсем другое название: «Доводы против оператора GO TO» (A Case against the GO TO Statement).

Однако в момент публикации произошло нечто непонятное — статья почему-то загадочным образом превратилась в «Письмо к редактору», причем прежнее название столь же загадочно исчезло. Что произошло на самом деле?

Дейкстра объяснил таинственное превращение статьи в письмо лишь много лет спустя, в 2001 году, за год до смерти: «Как все это случилось? Я отправил статью под названием «Доводы против оператора GO TO». Чтобы ускорить публикацию, редактор превратил мою статью в «Письмо к редактору». При этом он придумал для статьи новое название, которое изобрел сам. Редактором был Никлаус Вирт».

Цель структурного программирования состоит в упрощении чтения программного кода путем разбиения его на структурные единицы(блоки), соединенные управляющими конструкциями. Блоки могут содержать отдельные инструкции и подпрограммы. Подпрограммы в современных языках программирования представлены в виде процедур и/или функций. В рамках парадигмы структурного программирования существует 7 основных принципов:

1. Отказ от использования оператора goto т.к. это усложняет чтение программ, формирует так называемый спагетти-код.
2. Любая программа строится из трех базовых управляющих конструкций: последовательность, ветвление, цикл.
3. Базовые конструкции могут быть произвольным образом вложены друг в друга.
4. Повторяющиеся фрагменты программы целесообразно оформить в виде подпрограммы.
5. Логически законченная группа инструкций образует собой блок.
6. Все перечисленные конструкции должны иметь один вход и один выход.
7. Разработка программы ведется итеративно, используя метод нисходящего проектирования.

Нисходящее и восходящее проектирование.

Нисходящее проектирование – один из методов разработки проектов, систем, программ, при котором разработка производится сверху вниз. Суть метода заключается в итеративной декомпозиции задач на более мелкие подзадачи до

тех пор, пока каждая из них не будет представлять собой простой функциональный элемент.

Как правило, при основном нисходящем проектировании, проектирование компонент происходит по восходящему проектированию, когда компонент собирается из уже существующих функциональных элементов.

Разработка программного обеспечения полностью по принципу восходящего проектирования возможна лишь для сравнительно небольших групп программ, ограниченных по количеству несколькими модулями, когда разработчики способны оценивать в любое время структуру комплекса программ в целом и структуру и функции отдельных модулей на всех уровнях иерархии.

Поэтому при разработке больших программных изделий, содержащих сотни модулей, наиболее рациональным принципом является нисходящее проектирование.

Часто оба метода применяются одновременно:

- Нисходящее проектирование — при объединении в единое целое;
- Восходящее проектирование — при разработке общих хорошо отлаженных блоков.

Рассмотрим данный подход на примере задачи проверки валидности IP адреса. IP адрес состоит из 4 октетов, значения которых разделены точками.

На первом этапе проектирования перед нами стоит задача проверить валидность IP адреса. Эта задача может быть декомпозирована на 3 подзадачи:

1. Выделение октетов из потока
2. Проверка каждого октета на принадлежность допустимому диапазону
3. Вывод сообщения о результате проверки

Второй пункт этой последовательности шагов представляет собой простой цикл с проверкой условия. 1 и 3 пункты обладают рядом особенностей.

Организация ввода-вывода

Важно отметить, что задача, рассмотренная ранее, была разбита на 3 подзадачи, которые можно выделить для подавляющего большинства задач в программировании:

1. Ввод и первичная обработка входных данных
2. Обсчет введенных данных
3. Приведение и вывод результатов

Так же, как правило, в разных задачах операции ввода вывода представляют собой одну и ту же последовательность инструкций, различаясь, возможно, только некоторыми условиями, специфичными в рамках решения той или иной задачи. Поэтому целесообразно иметь под рукой или в памяти стандартный набор функций, реализующий ввод из входного потока переменных определенных типов данных: чисел, строк, отдельных слов.

В некоторых языках высокого уровня для подобных целей реализованы специальные библиотеки, что в разы упрощает данную задачу.

Множества решений

При разработке алгоритма также немаловажным является определение множеств возможных полученных результатов. Как правило, множества можно на $(n+1)^m$ кластеров, где m – количество параметров, на которое накладываются условия, а n – количество условий, которым должны удовлетворять параметры. Рассмотрим на примерах.

В случае, когда множество результатов X должно лежать в некотором диапазоне [m,n], множество возможных результатов делится на 3 кластера:

1. x<m
2. x>=m && x<=n
3. x>n.

Если же результат задается парой параметров (x,y), и на них накладываются условия принадлежности диапазону, то множество решений будет разделено на 9 кластера:

x<m y>n	x>=m && x<=n y>n	x>m y>n
x<m y>=m && y<=n	x>=m && x<=n y>=m && y<=n	x>m y>=m && y<=n
x<m y<m	x>=m && x<=n y<m	x>m y<m

Решение некоторых задач становится проще, когда во внимание принимаются множества возможных решений. Рассмотрим это на примере.

Задача проверки контрольного значения.

Рассмотрим решение одной из задач ЕГЭ:

По каналу связи передаётся последовательность положительных целых чисел, все числа не превышают 1000. Количество чисел известно, но может быть очень велико. Затем передаётся контрольное значение последовательности — наибольшее число R, удовлетворяющее следующим условиям:

1) R — произведение двух различных переданных элементов последовательности («различные» означает, что не рассматриваются квадраты переданных чисел, произведения различных элементов последовательности, равных по величине, допускаются);
2) R делится на 14.

Если такого числа R нет, то контрольное значение полагается равным 0. В результате помех при передаче как сами числа, так и контрольное значение могут быть искажены.

Напишите эффективную, в том числе по используемой памяти, программу (укажите используемую версию языка программирования, например, Borland Pascal 7.0), которая будет проверять правильность контрольного значения. Программа должна напечатать отчёт по следующей форме:

Вычисленное контрольное значение: ...

Контроль пройден (или — Контроль не пройден)

Перед текстом программы кратко опишите используемый Вами алгоритм решения.

На вход программе в первой строке подаётся количество чисел N. В каждой из последующих N строк записано одно натуральное число, не превышающее 1000. В последней строке записано контрольное значение.

Пример входных данных:

6

77

14

7

9

499

100

7700

Пример выходных данных для приведённого выше примера входных данных:

Вычисленное контрольное значение: 7700

Контроль пройден

Разбор задачи проверки контрольного значения.

Для начала надо определить каким образом формируется множество возможных контрольных значений, если не брать во внимание условие, что оно должно быть максимальным.

Контрольное значение должно делиться на 14. Путем факторизации (разложения на простые множители) числа 14, получаем, что существует 2 способа получить контрольное значение:

1. $CV = (7 * n) * (2 * m)$

2. $CV = (14 * n) * m$

Таким образом задача сводится к поиску 4 значений из потока входных данных. Далее представлен программный код, реализующий этот поиск.

```
N = int(input())

m2 = 0
m7 = 0
m14 = 0
mx = 0
for i in xrange(N):
    a = int(input())
    if a % 2 == 0 and a % 7 == 0:
        if m2 > m7:
            if m7 < a:
                m7 = a
        else:
            if m2 < a:
                m2 = a
    elif a % 2 == 0:
        if a > m2:
            m2 = a
    elif a % 7 == 0:
        if a > m7:
            m7 = a

    if a % 14 == 0 and a > m14:
        m14 = a
    elif a > mx:
        mx = a

if int(input())==max(m14*mx, m2*m7):
    print "Yes"
else:
    print "No"
```

Задача нахождения максимальной суммы из элементов пар.

Рассмотрим решение одной из задач ЕГЭ с применением метода нисходящего проектирования. Условие задачи выглядит следующим образом:

Задание: Имеется набор данных, состоящий из n пар положительных целых чисел. Необходимо выбрать из каждой пары ровно одно число так, чтобы сумма всех выбранных чисел не делилась на 3 и при этом была максимально возможной. Если получить требуемую сумму невозможно, в качестве ответа нужно выдать 0.

Напишите программу для решения этой задачи. В этом варианте задания оценивается только правильность программы, время работы и размер использованной памяти не имеют значения.

Постарайтесь сделать программу эффективной по времени и используемой памяти (или хотя бы по одной из этих характеристик).

Программа считается эффективной по времени, если время работы программы пропорционально количеству пар чисел N, т.е. при увеличении N в k раз время работы программы должно увеличиваться не более чем в k раз. Программа считается эффективной по памяти, если размер памяти, использованной в программе для хранения данных, не зависит от числа N и не превышает 1 килобайта.

Перед текстом программы кратко опишите Ваш алгоритм решения, укажите использованный язык программирования и его версию (например, Free Pascal 2.6.4).

Входные данные:

На вход программе в первой строке подаётся количество пар N ($1 \leq N \leq 100\,000$). Каждая из следующих N строк натуральных числа, не превышающих 10 000.

Пример входных данных для варианта Б:

```
6
1 3
5 12
6 9
5 4
3 3
1 1
```

Пример выходных данных для приведённых выше примеров входных данных:

```
32
```

Разбор задачи нахождения максимальной суммы из элементов пар.

Для начала рассмотрим множества всех возможных решений данной задачи. Из условия становится ясно, что они делятся на 3 больших кластера: те, что дают при делении на 3 остаток 1, 2 и те, что делятся без остатка. Для решения задачи мы можем зафиксировать максимальное значение в каждом из этих множеств, итеративно складывая значения пар с предыдущими полученными значениями из этих множеств.

Таким образом задача нахождения суммы сводится к поиску 3х максимальных сумм, каждая из которых может получиться 6 разными способами. При использовании метода нисходящего проектирования, задача примет следующий вид:

1. Поиск максимальной суммы
    1. Инициализация исходных данных
    2. Расчет сумм
        1. Расчет суммы кратной 3
            1. Сложение каждой суммы, полученной на предыдущей итерации с элементами пары
            2. Проверка на кратность
        2. Расчет суммы, дающей остаток от деления равный 1
            1. Сложение каждой суммы, полученной на предыдущей итерации с элементами пары
            2. Проверка на кратность
        3. Расчет суммы, дающей остаток от деления равный 2
            1. Сложение каждой суммы, полученной на предыдущей итерации с элементами пары
            2. Проверка на кратность
    3. Вывод максимального результата

Далее представлен код, реализующий данный подход. Код в рамках данного решения может быть оптимизирован, но для наглядности оптимизация не проводилась. Обрабатываемая пара чисел хранится в

переменных a1, a2. Суммы, полученные после каждой итерации хранятся в переменных s0, s1, s2. Для хранения сумм полученных в ходе итерации путем сложения каждой из сумм s0, s1, s2 с каждым значением текущей введенной пары a1, a2 используются переменные t01, t02, t11, t12, t21, t22.

```
 1  N = int(input())
 2  s0 = 0
 3  s1 = 0
 4  s2 = 0
 5  a1, a2 = [int(x) for x in raw_input().split()]
 6  if a1 > a2:
 7      a1, a2 = a2, a1
 8  if a1 % 3 == 0:
 9      s0 = a1
10  if a1 % 3 == 1:
11      s1 = a1
12  if a1 % 3 == 2:
13      s2 = a1
14  if a2 % 3 == 0:
15      s0 = a2
16  if a2 % 3 == 1:
17      s1 = a2
18  if a2 % 3 == 2:
19      s2 = a2
20  for i in xrange(N-1):
21      a1, a2 = [int(x) for x in raw_input().split()]
22      t01 = 0
23      t02 = 0
24      if s0 != 0:
25          t01 = s0+a1
26          t02 = s0+a2
27      t11 = 0
28      t12 = 0
29      if s1 != 0:
30          t11 = s1+a1
31          t12 = s1+a2
32      t21 = 0
33      t22 = 0
34      if s2 != 0:
35          t21 = s2+a1
36          t22 = s2+a2
37      if t01 % 3 == 0 and t01 > s0:
38          s0 = t01
39      if t02 % 3 == 0 and t02 > s0:
40          s0 = t02
41      if t01 % 3 == 1 and t01 > s1:
42          s1 = t01
43      if t02 % 3 == 1 and t02 > s1:
```

```
44              s1 = t02
45          if t01 % 3 == 2 and t01 > s2:
46              s2 = t01
47          if t02 % 3 == 2 and t02 > s2:
48              s2 = t02
49
50          if t11 % 3 == 0 and t11 > s0:
51              s0 = t11
52          if t12 % 3 == 0 and t12 > s0:
53              s0 = t12
54          if t11 % 3 == 1 and t11 > s1:
55              s1 = t11
56          if t12 % 3 == 1 and t12 > s1:
57              s1 = t12
58          if t11 % 3 == 2 and t11 > s2:
59              s2 = t11
60          if t12 % 3 == 2 and t12 > s2:
61              s2 = t12
62
63          if t21 % 3 == 0 and t21 > s0:
64              s0 = t21
65          if t22 % 3 == 0 and t22 > s0:
66              s0 = t22
67          if t21 % 3 == 1 and t21 > s1:
68              s1 = t21
69          if t22 % 3 == 1 and t22 > s1:
70              s1 = t22
71          if t21 % 3 == 2 and t21 > s2:
72              s2 = t21
73          if t22 % 3 == 2 and t22 > s2:
74              s2 = t22
75
76      if s1 > s2:
77          print s1
78      else:
79          print s2
```

Количество памяти, которое требуется для работы данной программы никак не зависит от объёма входных данных, а сами данные обрабатываются за один проход. Таким образом сложность алгоритма по времени составляет O(n), а сложность по используемой памяти O(1). Данное решение удовлетворяет всем требованиям задачи. Однако само решение получилось громоздким.

Функциональное программирование

Одним из способов найти более оптимальное решение может стать попытка рассмотреть задачу с точки зрения функционального программирования.

Функциональное программирование — раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний (в значении, подобном таковому в теории автоматов). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например, как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

На практике отличие математической функции от понятия «функции» в императивном программировании заключается в том, что императивные функции могут опираться не только на аргументы, но и на состояние внешних по отношению к функции переменных, а также иметь побочные эффекты и менять состояние внешних переменных. Таким образом, в императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, можно получить разные данные на выходе из-за влияния на функцию состояния переменных. А в функциональном языке при вызове функции с одними и теми же

аргументами мы всегда получим одинаковый результат: выходные данные зависят только от входных. Это позволяет средам выполнения программ на функциональных языках кешировать результаты функций и вызывать их в порядке, не определяемом алгоритмом и распараллеливать их без каких-либо дополнительных действий со стороны программиста (что обеспечивают функции без побочных эффектов — чистые функции).

Основной особенностью функционального программирования, определяющей как преимущества, так и недостатки данной парадигмы, является то, что в ней реализуется модель вычислений без состояний. Если императивная программа на любом этапе исполнения имеет состояние, то есть совокупность значений всех переменных, и производит побочные эффекты, то чисто функциональная программа ни целиком, ни частями состояния не имеет и побочных эффектов не производит. То, что в императивных языках делается путём присваивания значений переменным, в функциональных достигается путём передачи выражений в параметры функций. Непосредственным следствием становится то, что чисто функциональная программа не может изменять уже имеющиеся у неё данные, а может лишь порождать новые путём копирования и/или расширения старых. Следствием того же является отказ от циклов в пользу рекурсии.

Привлекательная сторона вычислений без состояний — повышение надёжности кода за счёт чёткой структуризации и отсутствия необходимости отслеживания побочных эффектов. Любая функция работает только с локальными данными и работает с ними всегда одинаково, независимо от того, где, как и при каких обстоятельствах она вызывается. Невозможность мутации данных при пользовании ими в разных местах программы исключает появление труднообнаруживаемых ошибок (таких, например, как случайное присваивание неверного значения глобальной переменной в императивной программе).

Поскольку функция в функциональном программировании не может порождать побочные эффекты, менять объекты нельзя как внутри области

видимости, так и снаружи (в отличие от императивных программ, где одна функция может установить какую-нибудь внешнюю переменную, считываемую второй функцией). Единственным эффектом от вычисления функции является возвращаемый ей результат, и единственный фактор, оказывающий влияние на результат — это значения аргументов.

Таким образом, имеется возможность протестировать каждую функцию в программе, просто вычислив её от различных наборов значений аргументов. При этом можно не беспокоиться ни о вызове функций в правильном порядке, ни о правильном формировании внешнего состояния. Если любая функция в программе проходит модульные тесты, то можно быть уверенным в качестве всей программы. В императивных программах проверка возвращаемого значения функции недостаточна: функция может модифицировать внешнее состояние, которое тоже нужно проверять, чего не нужно делать в функциональных программах.

Традиционно упоминаемой положительной особенностью функционального программирования является то, что оно позволяет описывать программу в так называемом «декларативном» виде, когда жесткая последовательность выполнения многих операций, необходимых для вычисления результата, в явном виде не задаётся, а формируется автоматически в процессе вычисления функций. Это обстоятельство, а также отсутствие состояний даёт возможность применять к функциональным программам достаточно сложные методы автоматической оптимизации.

Ещё одним преимуществом функциональных программ является то, что они предоставляют широчайшие возможности для автоматического распараллеливания вычислений. Поскольку отсутствие побочных эффектов гарантировано, в любом вызове функции всегда допустимо параллельное вычисление двух различных параметров — порядок их вычисления не может оказать влияния на результат вызова.

Недостатки функционального программирования вытекают из тех же самых его особенностей. Отсутствие присваиваний и замена их на порождение

новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным компонентом становится высокоэффективный сборщик мусора. Нестрогая модель вычислений приводит к непредсказуемому порядку вызова функций, что создает проблемы при вводе-выводе, где порядок выполнения операций важен. Кроме того, очевидно, функции ввода в своем естественном виде (например, getchar из стандартной библиотеки языка С) не являются чистыми, поскольку способны возвращать различные значения для одних и тех же аргументов, и для устранения этого требуются определенные ухищрения.

Для преодоления недостатков функциональных программ уже первые языки функционального программирования включали не только чисто функциональные средства, но и механизмы императивного программирования (присваивание, цикл, «неявный PROGN» были уже в Лиспе). Использование таких средств позволяет решить некоторые практические проблемы, но означает отход от идей (и преимуществ) функционального программирования и написание императивных программ на функциональных языках. В чистых функциональных языках эти проблемы решаются другими средствами, например, в языке Haskell ввод-вывод реализован при помощи монад — нетривиальной концепции, позаимствованной из теории категорий.

Альтернативное решение задачи

Рассмотрим задачу, как последовательность вычисляемых функций. Первой функцией в этой последовательности станет вычисление максимально возможной суммы из элементов пар:

$$sum = \sum \max(a1, a2)$$

В случае, если эта сумма не удовлетворяет условию задачи(делится без остатка на 3), необходимо произвести корректировку результата. Императивно это будет выглядеть, как замена одного слагаемого суммы на

другое. При этом изменение суммы должно быть минимальным и не кратным 3. Можем выразить это в виде формулы:

$$result = sum - delta\ ,$$

где

$$delta = \begin{cases} 0, \text{при } sum\ mod\ 3\ \neq 0 \\ \min\left(\{d \mid d = |a1 - a2| : |a1 - a2|\ mod\ 3\ \neq 0\}\right) \end{cases}$$

На основе этих формул можно составить код. Код написан в Императивном стиле для простоты восприятия.

```
 1  N = int(input())
 2
 3  sum = 0
 4  delta = 0
 5  for i in xrange(N):
 6      a1, a2 = [int(x) for x in raw_input().split()]
 7      sum += max(a1, a2)
 8      if delta % 3 == 0:
 9          delta = abs(a1-a2)
10      elif abs(a1-a2) % 3 > 0:
11          delta = min(delta, abs(a1-a2))
12
13  print sum if sum % 3 != 0 else sum - delta \
14      if sum - delta % 3 != 0 else 0
```

Заметно, что при использовании такого подхода решение задачи становится проще и лаконичнее. Входные данные так же обрабатываются за один проход и объем используемой памяти не зависит от объема входных данных. Таким образом решение так же удовлетворяет всем требованиям задания.

Введение в динамическую теорию игр

Чтобы мотивировать дальнейшее изложение рассмотрим следующую ситуацию.

«Выбор консоли». Двое приятелей одновременно и независимо друг от друга идут покупать игровые консоли. При этом они решают, какого типа консоли им купить. Первый предпочитает XBOX, второй – PlayStation. Обладание консолью любимого типа первый оценивает в a (a >> 1) некоторых

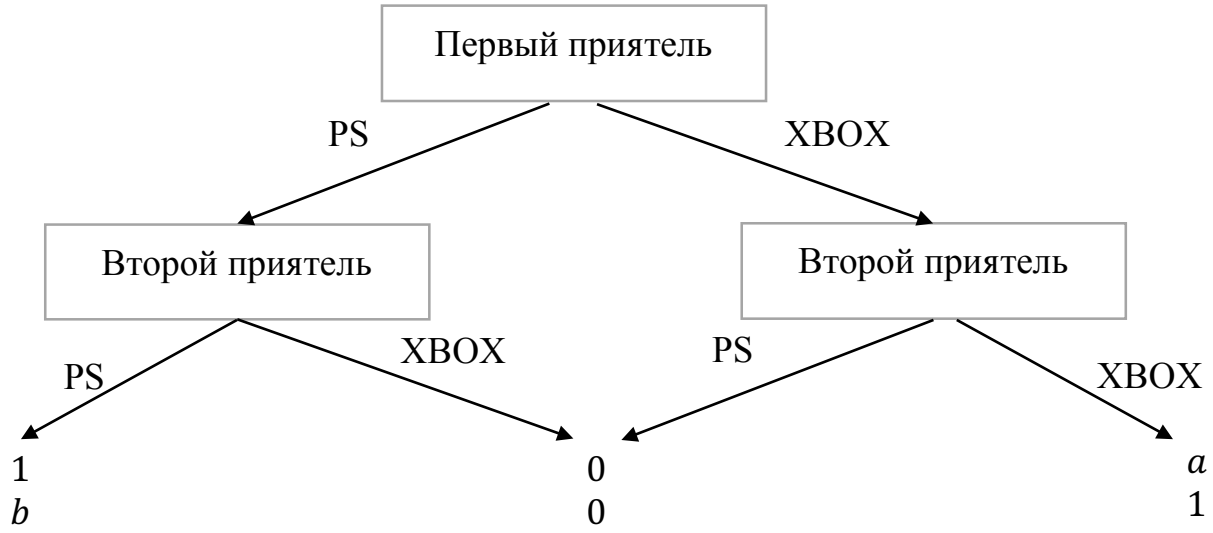
условных единиц (у.е.), а второй – в b (b >> 1) у.е. Ценность консоли другого типа для обоих будем считать равной единице. Каждый приятель теряет выгоду (a = b = 0), если они выберут разные консоли, поскольку в таком случае они не смогут играть вместе. Наша задача – понять, какой выбор сделают приятели при различных значениях величин a и b.

Подобные проблемы – предмет исследований теории игр, точнее, теория игр изучает, как люди принимают решения в ситуациях, когда на результат этих решений влияют решения других людей. Такие люди называются игроками, а ситуации выбора – играми. Задача теории игр – по данному описанию условий игры предсказать, какие решения примут игроки, и каким будет исход игры. В ситуации «Выбор компьютера» два приятеля оказываются игроками, причем на оценку выбора одного игрока влияет решение, принимаемое другим (другими игроками).

Многие ситуации, включающие взаимодействие индивидуумов, являются по своему смыслу динамическими. Люди взаимодействуют друг с другом во времени и действуют, реагируя на те решения, которые ранее приняли другие. Другими словами, принимая решения, каждый игрок что-то знает о решениях, уже принимавшихся другими игроками, что предполагает очередность принятия решений (ходов).

Динамической будем называть такую игру, в которой каждый игрок может сделать несколько ходов и, по крайней мере, один из игроков, делая ход, знает, какой ход сделал другой игрок (возможно, он сам). В этой ситуации он стоит перед свершившимися фактами (уже сделанными ранее и известными ему ходами) и должен учитывать их при выборе своих действий.

Подобные игры удобно представить в виде дерева игры:



Решение игры можно найти в предположении, что игроки рациональны и что рациональность и структура игры являются всем известными фактами, что обычно предполагается при анализе ситуаций методами теории игр. При этом естественно начать анализ с конца игры, т.е. воспользоваться методом обратной индукции.

Рассмотрим поведение второго приятеля. Несложно проследить, что ему выгоднее купить ту же консоль, что купил и первый приятель, ведь 1 и b больше, чем 0. Исходя из этих соображений, дерево игры можно перерисовать следующим образом:

В результате перед первым приятелем, при условии, что второй приятель чисто из вредности не купит другую консоль, стоит задача при желании сравнить значения a и b и сделать вывод о том, кто хочет консоль определенного типа больше… Или наплевать на желание друга и купить себе XBOX!

Задача «Игра Паши и Вали»

Рассмотрим следующую задачу:

Два игрока, Паша и Валя, играют в следующую игру. Перед игроками лежит куча камней. Игроки ходят по очереди, первый ход делает Паша. За один ход игрок может добавить в кучу **один** камень или увеличить количество камней в куче **в два раза**. Например, имея кучу из 15 камней, за один ход можно получить кучу из 16 или 30 камней. У каждого игрока, чтобы делать ходы, есть неограниченное количество камней.

Игра завершается в тот момент, когда количество камней в куче становится не менее 20. Если при этом в куче оказалось не более 30 камней, то победителем считается игрок, сделавший последний ход. В противном случае победителем становится его противник. Например, если в куче было 17 камней и Паша удвоит количество камней в куче, то игра закончится, и победителем будет Валя. В начальный момент в куче было S камней, $1 \leq S \leq 19$.

Будем говорить, что игрок имеет выигрышную стратегию, если он может выиграть при любых ходах противника. Описать стратегию игрока – значит описать, какой ход он должен сделать в любой ситуации, которая ему может встретиться при различной игре противника.

Выполните следующие задания.

1. а) При каких значениях числа S Паша может выиграть в один ход? Укажите все такие значения и соответствующие ходы Паши.

   б) У кого из игроков есть выигрышная стратегия при S = 18, 17, 16? Опишите выигрышные стратегии для этих случаев.
2. У кого из игроков есть выигрышная стратегия при S = 9, 8? Опишите соответствующие выигрышные стратегии.
3. У кого из игроков есть выигрышная стратегия при S = 7? Постройте дерево всех партий, возможных при этой выигрышной стратегии (в виде рисунка или таблицы). На рёбрах дерева указывайте, кто делает ход; в узлах – количество камней в позиции.

Разбор задачи «Игра Паши и Вали»

Рассмотрим пункт 1а. Для того, чтобы определить множество S, при котором Паша выигрывает в 1 ход, необходимо рассмотреть множество результатов W, которые являются выигрышными. По условию задачи, чтобы Паша выиграл, он должен сделать так, чтоб на столе оказалось от 20 до 30 камней. При этом он может выполнять всего две операции:

1. Положить 1 камень на стол

$$s = s + 1$$

2. Удвоить количество камней на столе

$$s = s * 2$$

Для того, чтобы получить множество S, необходимо применить к множеству выигрышных результатов W применить функции, обратные возможным ходам Паши.

$$S' = \{W' | \, w' = w - 1 : w \in W \} \cup \{W'' | \, w'' = w/2 : w \in W \}$$

Таким образом множество S’ состоит из элементов {[10,15], [19,29]}. Однако, надо понимать, что во множестве S не могут находиться элементы множества W, т.к. это подразумевает выигрыш Вали на момент хода Паши. Таким образом множество S иметь следующий вид:

$$S = S' - W = \{[10,15], 19\}$$

Рассмотрим задание 1б. При S = 16, 17 или 18 удваивать количество камней не имеет смысла, так как после такого хода выигрывает противник. Поэтому можно считать, что единственный возможный ход – это добавление в кучу одного камня. При S = 18 после такого хода Паши в куче станет 19 камней. В этой позиции ходящий (т.е. Валя) выигрывает (см. п. 1а): при S = 18 Паша (игрок, который должен ходить первым) проигрывает. Выигрышная стратегия есть у Вали.

При S = 17, после того как Паша своим первым ходом добавит один камень, в куче станет 18 камней. В этой позиции ходящий (т.е. Валя) проигрывает (см. выше): при S = 17 Паша (игрок, который должен ходить первым) выигрывает. Выигрышная стратегия есть у Паши.

При S = 16 выигрышная стратегия есть у Вали. Действительно, если Паша первым ходом удваивает количество камней, то в куче становится 32 камня, и игра сразу заканчивается выигрышем Вали. Если Паша добавляет один камень, то в куче становится 17 камней. Как мы уже знаем, в этой позиции игрок, который должен ходить (т.е. Валя), выигрывает.

Во всех случаях выигрыш достигается тем, что при своём ходе игрок, имеющий выигрышную стратегию, должен добавить в кучу один камень.

Рассмотрим пункт 2 и 3. Легко заметить, что они полностью или частично сводятся к пункту 1. При S = 9 или 8 выигрышная стратегия есть у Паши. Она состоит в том, чтобы удвоить количество камней в куче и получить кучу, в которой будет соответственно 18 или 16 камней. В обоих случаях игрок, который будет делать ход (теперь это Валя), проигрывает (п. 1б).

При S = 7 выигрышная стратегия есть у Вали. После первого хода Паши в куче может стать либо 8, либо 14 камней. В обеих этих позициях выигрывает игрок, который будет делать ход (теперь это Валя). Случай S = 8 рассмотрен в п. 2, случай S = 14 рассмотрен в п. 1а.

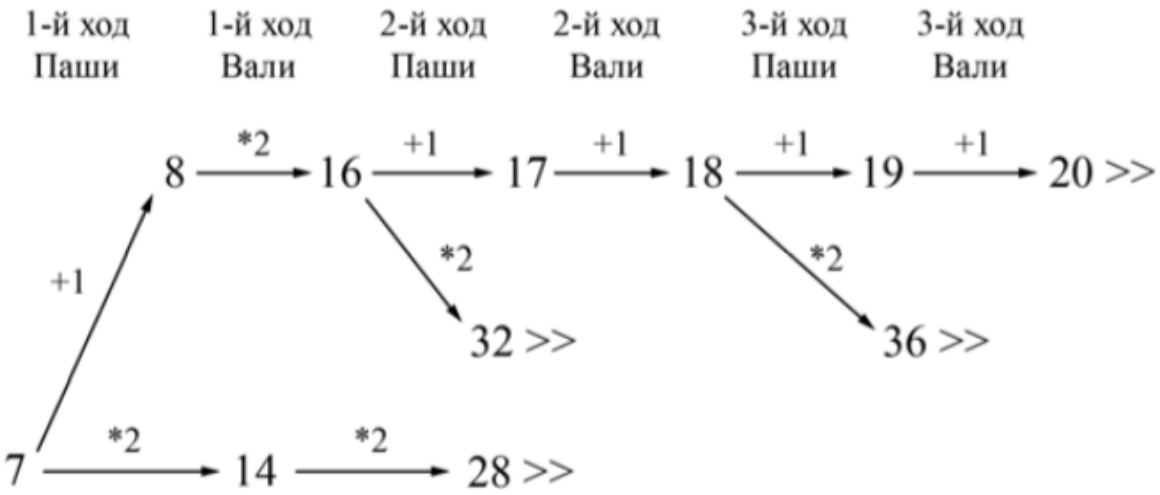


Рисунок 3 - Дерево игры при s=7

Решение систем логических уравнений

Умение мыслить последовательно, рассуждать доказательно, строить гипотезы, опровергать негативные выводы, не приходит само по себе, это умение развивает наука логика. Логика – это наука, изучающая методы

установления истинности или ложности одних высказываний на основе истинности или ложности других высказываний.

Овладение азами этой науки невозможно без решения логических задач. Проверка сформированности умений применять свои знания в новой ситуации осуществляется за счет сдачи. В частности, это умение решать логические задачи. Задания В15 в ЕГЭ, являются заданиями повышенной сложности, так как они содержат системы логических уравнений. Можно выделить различные способы решения систем логических уравнений. Это сведение к одному уравнению, построение таблицы истинности, декомпозиция, последовательное решение уравнений и т.д.

Задача: Решить систему логических уравнений:

$$\begin{cases} \overline{A} \to B = 0 \\ A + C = 1 \end{cases}$$

Рассмотрим метод сведения к одному уравнению. Данный метод предполагает преобразование логических уравнений, таким образом, чтобы правые их части были равны истинностному значению (то есть 1). Для этого применяют операцию логического отрицания. Затем, если в уравнениях есть сложные логические операции, заменяем их базовыми: «И», «ИЛИ», «НЕ». Следующим шагом объединяем уравнения в одно, равносильное системе, с помощью логической операции «И». После этого, следует сделать преобразования полученного уравнения на основе законов алгебры логики и получить конкретное решение системы.

Решение 1: Применяем инверсию к обеим частям первого уравнения:

$$\begin{cases} \overline{\overline{A} \to B} = 1 \\ A + C = 1 \end{cases}$$

Представим импликацию через базовые операции «ИЛИ», «НЕ»:

$$\begin{cases} \overline{A + B} = 1 \\ A + C = 1 \end{cases}$$

Поскольку левые части уравнений равны 1, можно объединить их с помощью операции “И” в одно уравнение, равносильное исходной системе:

$$\overline{(A+B)} * (A+C) = 1$$

Раскрываем первую скобку по закону де Моргана и преобразовываем полученный результат:

$$\left(\overline{A} * \overline{B}\right) * (A+C) = 1 \Leftrightarrow \overline{A} * \overline{B} * C = 1$$

Полученное уравнение, имеет одно решение: A=0, B=0 и C=1.

Следующий способ – построение таблиц истинности. Поскольку логические величины имеют только два значения, можно просто перебрать все варианты и найти среди них те, при которых выполняется данная система уравнений. То есть, мы строим одну общую таблицу истинности для всех уравнений системы и находим строку с нужными значениями.

Решение 2: Составим таблицу истинности для системы:

A	B	C	$\overline{A}$	$\overline{A} \to B$	$A + C$
1	1	1	0	1	1
1	1	0	0	1	1
1	0	1	0	1	1
1	0	0	0	1	1
0	1	1	1	1	1
0	1	0	1	1	0
0	**0**	**1**	**1**	**0**	**1**
0	0	0	1	0	0

Полужирным выделена строчка, для которой выполняются условия задачи. Таким образом, A=0, B=0 и C=1.

Способ декомпозиции. Идея состоит в том, чтобы зафиксировать значение одной из переменных (положить ее равной 0 или 1) и за счет этого упростить уравнения. Затем можно зафиксировать значение второй переменной и т.д.

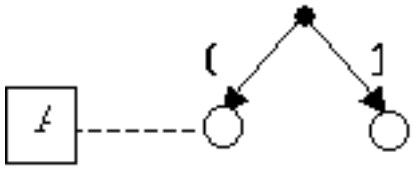
Решение 3: Пусть A = 0, тогда:

$$\begin{cases} \overline{0} \to B = 0 \\ 0 + C = 1 \end{cases}$$

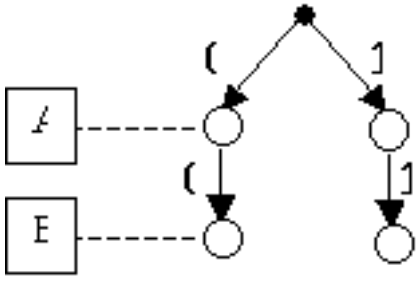
Из первого уравнения получаем B=0, а из второго – C=1. Решение системы: A = 0, B = 0 и C = 1.

Так же можно воспользоваться методом последовательного решения уравнений, на каждом шаге добавляя по одной переменной в рассматриваемый набор. Для этого необходимо преобразовать уравнения таким образом, что бы переменные вводились в алфавитном порядке. Далее строим дерево решений, последовательно добавляя в него переменные.

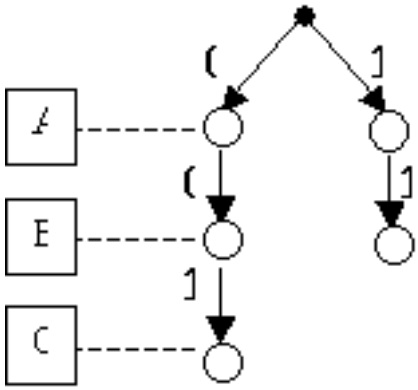
Первое уравнение системы зависит только от A и B, а второе уравнение от A и C. Переменная A может принимать 2 значения 0 и 1:



Из первого уравнения следует, что , поэтому при A = 0 получаем B = 0, а при A = 1 имеем B = 1. Итак, первое уравнение имеет два решения относительно переменных A и B.



Изобразим второе уравнение, из которого определим значения C для каждого варианта. При A=1 импликация не может быть ложной, то есть вторая ветка дерева не имеет решения. При A=0 получаем единственное решение C = 1:



Таким образом, получили решение системы: A = 0, B = 0 и C = 1.

В ЕГЭ по информатике очень часто требуется определить количество решений системы логических уравнений, без нахождения самих решений, для этого тоже существуют определенные методы. Основной способ нахождения количества решений системы логических уравнений – замена переменных. Сначала необходимо максимально упростить каждое из уравнений на основе законов алгебры логики, а затем заменить сложные части уравнений новыми переменными и определить количество решений новой системы. Далее вернуться к замене и определить для нее количество решений.

Задача: Сколько решений имеет уравнение (A→B) + (C→D) = 1? Где A, B, C, D – логические переменные.

Решение: Введем новые переменные: X = A→B и Y = C→D. С учетом новых переменных уравнение запишется в виде: X + Y = 1.

Дизъюнкция верна в трех случаях: (0;1), (1;0) и (1;1), при этом X и Y является импликацией, то есть является истинной в трех случаях и ложной – в одном. Поэтому случай (0;1) будет соответствовать трем возможным сочетаниям параметров. Случай (1;1) – будет соответствовать девяти возможным сочетаниям параметров исходного уравнения. Значит, всего возможных решений данного уравнения 3+9=15.

Следующий способ определения количества решений системы логических уравнений – бинарное дерево. Рассмотрим данный метод на примере.

Задача: Сколько различных решений имеет система логических уравнений:

$$\begin{cases} \overline{x_1} + x_2 = 1 \\ \overline{x_2} + x_3 = 1 \\ \cdots \\ \overline{x_{m-1}} + x_m = 1 \end{cases}$$

Приведенная система уравнений равносильна уравнению:

(x1 → x2)*(x2 → x3)*…*(xm-1 → xm) = 1.

Предположим, что x1 – истинно, тогда из первого уравнения получаем, что x2 также истинно, из второго - x3=1, и так далее до xm = 1. Значит набор (1; 1; …; 1) из m единиц является решением системы. Пусть теперь x1=0, тогда из первого уравнения имеем x2 =0 или x2 =1.

Когда x2 истинно получаем, что остальные переменные также истинны, то есть набор (0; 1; …; 1) является решением системы. При x2=0 получаем, что x3=0 или x3=, и так далее. Продолжая до последней переменной, получаем, что решениями уравнения являются следующие наборы переменных (m+1 решение, в каждом решении по m значений переменных):

(1; 1; 1; …; 1)

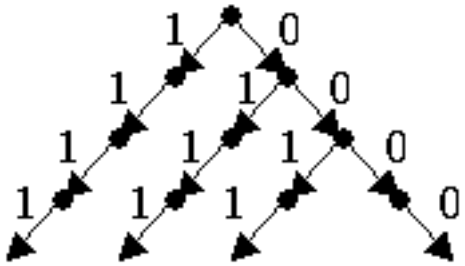
(0; 1; 1; …; 1)

…

(0; 0; 0; …; 0)

Такой подход хорошо иллюстрируется с помощью построения бинарного дерева. Количество возможных решений – количество различных ветвей построенного дерева. Легко заметить, что оно равно m+1.

Переменные	Дерево	Количество решений
x_1		
x_2		3
x_3		4
…		…



В случае трудностей в рассуждениях и построении дерева решений можно искать решение с использованием таблиц истинности, для одного – двух уравнений.

Перепишем систему уравнений в виде:

$$\begin{cases} x_1 \to x_2 = 1 \\ x_2 \to x_3 = 1 \\ \dots \\ x_{m-1} \to x_m = 1 \end{cases}$$

И составим таблицу истинности отдельно для одного уравнения:

x_1	x_2	$(x_1 \to x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

Составим таблицу истинности для двух уравнений:

x_1	x_2	x_3	$x_1 \to x_2$	$x_2 \to x_3$	$(x_1 \to x_2) * (x_2 \to x_3)$
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	0	1	0
1	1	0	1	0	0
1	1	1	1	1	1

Далее можно увидеть, что одно уравнение истинно в следующих трех случаях: (0; 0), (0; 1), (1; 1). Система двух уравнений истина в четырех случаях (0; 0; 0), (0; 0; 1), (0; 1; 1), (1; 1; 1). При этом сразу видно, что существует решение, состоящее из одних нулей и еще m решений, в которых добавляется по одной единице, начиная с последней позиции до заполнения всех возможных мест. Можно предположить, что общее решение будет иметь такой

же вид, но чтобы такой подход стал решением, требуется доказательство, что предположение верно.

Подводя итог всему вышесказанному, хочется обратить внимание, на то, что не все рассмотренные методы являются универсальными. При решении каждой системы логических уравнений следует учитывать ее особенности, на основе которых и выбирать метод решения.