Статья опубликована в диске к журналу «Мир ПК». 2005. № 9.

О решении олимпиадных задач по программированию формата ACM ICPC

С. А. Оршанский
Санкт-Петербургский государственный университет информационных технологий, механики и оптики
Чемпион мира ACM ICPC 2004 года

Интересно читать мнение любых людей, добившихся выдающихся результатов в своей области. Статья С.А. Оршанского не является исключением.

А.А.Шалыто, докт.техн.наук, профессор

1. Про олимпиады

1.1. Соревнования по информатике и программированию

Олимпиады по информатике, как и олимпиады по математике, широко распространены и имеют достаточно долгую историю. Командный студенческий чемпионат мира по программированию ACM ICPC (*Association for Computing Machinery International Collegiate Programming Contest*) [1–4] проводится с 1977 года. Международная олимпиада школьников по информатике IOI (*International Olympiad in Informatics*) проводится с 1989 года. Эти олимпиады позволяют выявлять способности, как в математике, так и в программировании, а также умение работать под стрессом в сжатых временных рамках. Указанные соревнования студентов традиционно являются командными, а школьников – личными. В России более долгую историю имеют олимпиады школьников по информатике. В книге [5] собраны все задачи Московских олимпиад по программированию, которые прошли с 1980 по 1988 гг. Материалы для подготовки к школьным олимпиадам можно найти также в книгах [6 – 8].

Популярность соревнований по информатике и программированию стремительно растет. Их спонсорами выступают такие крупные корпорации, как *AT&T*, *Microsoft*, *IBM*, *Google*. Естественно, появились исследования о том, как эффективно участвовать в соревнованиях, готовиться к ним, многочисленные советы и рассказы очевидцев [9]. К этой категории относится и настоящая статья. Автор имеет большой опыт участия в соревнованиях по информатике и программированию, преимущественно – командных студенческих. В настоящей статье речь пойдет о решении задач чемпионата мира ACM ICPC или аналогичных соревнований.

Цель статьи – попытаться ответить на вопрос «Как решить задачу?» при условии, что она одна и решить ее надо достаточно быстро. При этом необходимо решить задачу наверняка, а не с 50% вероятностью. Аспекты командной борьбы,

стратегия и тактика не рассматриваются, но некоторые соображения по этим вопросам будут приведены.

В работе излагаются некоторые общие принципы решения задач. Затем эти принципы иллюстрируются на примере задачи, не очень простой, но, по нынешним понятиям, достаточно стандартной. В приложении приведен результат – решение задачи на языке *Pascal* (*Borland Delphi*). В работе также затронут вопрос о минимальном круге идей и методов, которыми целесообразно владеть каждому участнику соревнований. Эти идеи и методы являются базовыми не только для подготовленных участников, но и для составителей задач. За это олимпиады иногда подвергаются критике. Однако в этом соревнования по программированию мало чем отличаются от других сфер человеческой деятельности. Если бы на разных этапах соревнования давали принципиально различные задачи, то отборочные соревнования потеряли бы смысл. Принципиально изменять характер задач из года в год на всех этапах – четвертьфиналах, полуфиналах и в финале, – нереально. В этом и нет необходимости, поскольку неясно, кого в таком случае будет выявлять чемпионат. На сегодняшний день чемпионат отбирает лучших в командном решении задач формата ACM ICPC. На этих соревнованиях команда состоит из трех человек, ей предоставляется один компьютер на пять часов для решения 8 – 12 задач.

1.2. Об обмене опытом

В книге [4] можно прочитать легендарный текст Леонида Волкова и Никиты Шамгунова «Как стать чемпионом Урала по программированию». Многие участники олимпиад знают наизусть фразу из этого текста: «Я не знаю, как решать задачи. Я знаю только, что после того, как решишь их много, начинаешь делать это лучше, начинаешь лучше видеть возможные подходы к решению задач, начинаешь лучше их чувствовать».

Вероятно, многолетний опыт участия в командных соревнованиях по программированию позволяет мне высказать ряд конструктивных соображений по вопросу «Как решать задачи?». Не отвечая на вопрос «Как стать чемпионом мира по программированию?», я постараюсь сформулировать и объяснить, чем я руководствовался при решении задач. Видимо, такие принципы все-таки существуют, хотя они, к сожалению, и не работают без интуиции [10].

Сама постановка вопроса «Как решать задачи?» может показаться кощунственной, поскольку решение задач – процесс творческий [11]. Однако описание некоторых методов и приемов, помогающих лучше решать задачи, вполне может служить ответом на поставленный вопрос, не предполагая, тем не менее, погружения в недра сознания. Подобным образом занятия физкультурой являются лишь механизмом, стимулирующим возможности тела, но совсем не обязательно раскрывающим сущность этих возможностей. Еще одно подтверждение вышесказанному – работы Г.С. Альтшуллера по теории решения изобретательских задач (ТРИЗ) [12]. Мало кто усомнится, что решение изобретательских задач – процесс творческий, однако ТРИЗ активно применяется на практике в самых разных сферах человеческой деятельности. Надо сказать, что многие идеи Г.С. Альтшуллера применимы не только для решения инженерных, но и научных и олимпиадных задач.

В заключение раздела отметим, что обычно в литературе приводятся задачи, а иногда – задачи с решениями [13–16]. Однако методика решения олимпиадных задач описана недостаточно подробно и, в основном, передается от предыдущего поколения олимпиадников к следующему в устной форме. В настоящей работе делается попытка хотя бы частично заполнить этот пробел.

1.3. Существующие наработки

Начнем с обзора существующих материалов, помогающих научиться решать задачи. Наиболее важную роль играет наличие собственно задач, на которых можно тренироваться. Каждый год в мире происходит огромное количество соревнований по программированию различного уровня. Стали появляться сайты с интерактивными архивами задач. В них не просто доступен текст задачи, как в библиотеках, но и имеется возможность послать решение на проверку и немедленно, почти как на настоящих соревнованиях, получить результат. На таких сайтах ведутся рейтинги участников по числу решенных задач, проводятся онлайн-соревнования, обычно носящие любительский или тренировочный характер.
Два крупнейших российских архива задач:

1. *Saratov State University :: Online Contester*. Онлайн-система тестирования олимпиадных задач Саратовского государственного университета. http://acm.sgu.ru/
2. *Ural State University Problem Set Archive with Online Judge System*. Онлайн-система тестирования олимпиадных задач Уральского государственного университета. http://acm.timus.ru/

Как отмечалось выше, существует ряд публикаций [13–16] с разбором конкретных задач всероссийских и международных олимпиад. Также есть публикации, например [17], в которых излагаются конкретные методики, применяемые при решении задач, небольшие хитрости и приемы.

Идеи настоящего текста адресованы читателю, участвовавшему в нескольких соревнованиях, хотя бы тренировочных. Поэтому предполагается, что читатель представляет, что такое олимпиадные задачи рассматриваемого формата, знает, что они проверяются на тестах, что засчитывается только программа, прошедшая все тесты и т. д. Информацию об этом можно найти в книге [2].

Решением задачи является программа, читающая входные данные и выводящая соответствующие выходные данные. Задача может быть либо решена полностью, либо не решена совсем, тогда как на школьных олимпиадах важно уметь писать программы, которые работают почти всегда или почти на всех наборах входных данных. Кроме того, в последнее время появились новые типы задач, например интерактивные задачи. На официальных командных соревнованиях такие задачи пока практически не встречаются, поэтому их рассматривать не будем.

На сайтах http://acm.sgu.ru/, http://acm.timus.ru/ и http://neerc.ifmo.ru/school/ можно найти ссылки на другие архивы задач и иные материалы схожей тематики.

2. Общая схема решения задачи

Перед вами условие задачи. Эта задача как-бы единственная, другой не дано. Поэтому требуется не оценивать ее, а решать, и именно ее. Это предположение делается для простоты, на практике все сложнее. Рассмотрим семь этапов, через которые проходит решение задачи. Конечно, некоторые из них могут пропускаться, смешиваться, распараллеливаться между членами команды и т. д.

Основное правило: можно пропускать не более одного этапа. Так переход к третьему этапу («построение общей схемы решения») должен происходить только после полного завершения первого этапа («чтение условия»), а, например, переход к седьмому этапу («посылка решения в жюри») – только после окончательного завершения пятого этапа («реализация»). То же верно для второго и четвертого, третьего и пятого, четвертого и шестого этапов решения задачи. При этом соседние этапы частично пересекаются. Обязательно обдумайте это правило, прочитав про все семь этапов – из него следует много важных выводов.

2.1. Чтение условия

На этом этапе необходимо внимательно прочесть условие, не пропуская ни одной фразы. Типичные проблемы:

- в обстановке соревнований сложно быть внимательным. Отведите достаточно времени на спокойное чтение условия. Отдохните полминуты, чтобы сконцентрироваться, но не спешите читать условие «наискосок». Неверное понимание условия может привести к тому, что вы будете решать совершенно другую задачу, а не ту, что сформулирована в условии;
- обычно в условии есть так называемое литературное введение, придающее задаче сюжет («background»). Чтение такого описания обычно утомляет, отвлекает, а также расслабляет, так как авторам задач обычно не чуждо чувство юмора. Однако будьте осторожны: во введении может, прямо или косвенно, содержаться важная информация, касающаяся условия. Если «background» не вынесен в отдельный раздел, то его придется прочитать. Обычно это делают не очень внимательно, выискивая начало содержательного текста, который, скорее всего, пойдет потом без перерыва;
- незнание или плохое знание английского языка может помешать правильному пониманию условий задач, которые даже на российских олимпиадах формулируются на английском языке. Это делается для тренировки перед финалами олимпиад. Практически на всех официальных соревнованиях разрешено использовать словарь. Не пренебрегайте этой возможностью и переводите те слова, которые критичны для понимания смысла. Если условие плохо понятно в целом, переведите по словарю и те слова, которые, на первый взгляд, не важны для понимания смысла. Изучайте английский язык на досуге – пригодится;
- ключевое условие может быть спрятано, например, в формате выходных данных. Без этого условия задача может быть совершенно другой, но тоже вполне корректной. Конечно, на серьезных

соревнованиях такого обычно не бывает, но все-таки будьте внимательны. Ошибки при чтении условия дорого обходятся.

2.2. Построение математической модели

На этом этапе необходимо понять, в чем заключается задача – построить ее математическую модель «в голове». Не думайте, что невыполнение этого этапа означает неправильное понимание. Можно внимательно прочитать текст, но не построить никакой математической модели. Попытка формализовать прочитанное часто выявляет множество нестыковок, возникших из-за важной фразы, пропущенной при чтении или неверно понятой. Хорошая проверка – внимательно рассмотреть приведенный пример входных и выходных данных и понять, почему выход соответствует входу.

Что значит «построить математическую модель»? Это означает достаточно формально и математически строго понять условие. С моей точки зрения, понять условие – это, как правило, научиться вручную, с помощью ручки и листа бумаги, находить ответ для простых наборов входных данных (тестов). Кстати, полезно не только научиться, но и проделать это для нескольких таких тестов. При этом улучшится понимание условия и, возможно, вскроется неправильное прочтение. Кроме того, могут прийти новые идеи, и в любом случае будут готовы тесты для дальнейшей проверки. Таким образом для простых задач вы разберетесь, понимаете ли вы их условия, тогда как для сложных задач простых тестов может оказаться недостаточно.

Возможна и другая трактовка того, что такое построить математическую модель. В этом случае построить математическую модель означает придумать решение, которое будет работать на абстрактной виртуальной, математической машине при неограниченной памяти, неограниченном времени, неограниченном диапазоне переменных и отсутствии потерь точности в вещественной арифметике.

При этом требуется разработать более-менее формальный алгоритм решения задачи. Это не означает даже решения задачи на уровне идеи. Неэффективное решение задачи оказывается практически равносильным пониманию условия. Например, дана строка и требуется найти подстроку, оптимальную по некоторому параметру. Решение вытекает из способа ручного поиска ответа на маленьких примерах: перебрать все подстроки и выбрать оптимальную. Скорее всего, вы не будете реализовывать это решение, но следует его придумать или наметить, или хотя бы почувствовать. Это и означает построить математическую модель в олимпиадном смысле – понять и осмыслить условие задачи.

Иногда условие включает неизвестное понятие. Даже если в условии это понятие определяется, от вас могут потребоваться значительные усилия по пониманию, что же требуется сделать. Соответственно, чем больше идей и методов вы освоили при подготовке, тем меньше шансов встретить в условии малознакомое понятие. Более того, привыкая к основным идеям, отрабатывая различные приемы, вы будете все чаще встречать знакомые задачи, полностью или с небольшими вариациями совпадающие с задачами, которые вы видели раньше.

2.3. Построение общей схемы решения

Теперь следует перейти от понимания того, что необходимо сделать, к пониманию того, как это сделать. В этом разделе намечается эффективный алгоритм решения задачи и пути его реализации. Это наименее формализуемая часть всего процесса решения. В ней может заключаться вся суть задачи, но она может быть и тривиальной при достаточно сложной задаче в целом. В чем же заключается этот этап?

У каждого участника существует некоторое понятие о «кирпичиках», элементарных структурных единицах создаваемой программы – в нашем случае об алгоритмических единицах. Каждая такая единица характеризуется функциональностью, эффективностью, сложностью написания (количеством кода) и т. д. Различные «кирпичики» имеют разные возможности по модификации.

На этом этапе необходимо построить решение из «кирпичиков». Участник, решающий задачу, не обязательно должен хорошо понимать и осознавать данный конкретный «кирпичик». Например, он может знать, что такое *Венгерский алгоритм*, какова его эффективность и насколько долго его писать. Однако он может в данный момент и не помнить деталей, а всего лишь быть уверенным в том, что он эти детали вспомнит, если потребуется, или попросит помощи у своего товарища по команде.

Итак, на этом этапе строится схема решения из «кирпичиков». При этом возникают следующие трудности:

- плохая стыковка. Интуитивная комбинация нескольких сложных блоков, внутренняя структура которых тяжело осознается человеком, может на первый взгляд решать задачу, однако при внимательном рассмотрении будут возникать проблемы. Может выясниться, что, например, у задачи, решаемой динамическим программированием, отсутствует свойство субоптимальности, или что маловероятный на первый взгляд «худший случай» возникает на любом достаточно большом наборе входных данных, или что-нибудь в том же духе. Эти проблемы сродни неправильному решению математической задачи;
- эффективность. Придуманная схема может оказаться неэффективной. При этом необходимо учитывать не только асимптотические оценки, но и игнорируемые в этих оценках константы. Самый надежный выход из этой ситуации – создание принципиально более эффективного решения всей задачи или какой-то подзадачи. Возможная альтернатива: «добивание» данного решения различными алгоритмическими и программистскими оптимизациями, что рискованно – можно потратить кучу времени, но так и не сдать задачу;
- другой тип задачи. Не все задачи решаются построением из «кирпичиков». Решение задачи может базироваться на математических идеях, которые необходимо просто придумать. В этом случае «кирпичиками» могут служить типичные приемы для подобных задач: отсортировать, начать с максимального/минимального элемента, использовать динамическое программирование и т. д.

При решении сложной задачи, к которой никак не подступиться, имеет смысл некоторое время «подолбить» ее стандартными методами. Это может не привести к решению, а может привести и к неправильному решению, внешне похожему на правильное. Однако такой подход оправдан хотя бы потому, что большинство задач только на первый взгляд – сложные. Практически невозможно придумывать к каждому

соревнованию 8-12 задач, совершенно новых и непохожих на задания прошлых лет. Если же такое случается, то многие задачи остаются нерешенными никем. Поэтому в задачах полным-полно вариаций на одни и те же темы;

- наличие нескольких решений. Синтез из «кирпичиков» может помочь быстро придумать решение, которое потом долго и нудно реализуется, тогда как для данной задачи может быть лучше немного подумать, и все существенно упростится.

2.4. Стыковка

Под стыковкой понимается уточнение решений, принятых на предыдущем этапе. Необходимо достаточно медленно и тщательно проговорить, из каких частей будет состоять программа, какие массивы и структуры будут выделены и т. д.

На этом этапе часто всплывают различные проблемы. Наиболее распространенная схема стыковки – один человек придумал общую схему решения задачи и рассказывает ее второму. Это неплохой метод, часто экономящий силы и время, но при этом возникают проблемы. Первая проблема: второй человек должен был сам, заранее и внимательно прочитать условие задачи. Вторая – он должен реально слушать первого, а не только делать вид, как это происходит в половине случаев.

Исключительно полезно писать решение задачи на бумаге. Лучше, чтобы это были не наброски, а большие законченные фрагменты или даже вся программа целиком, включая объявление всех переменных. Полезность написания кода на бумаге не в том, что перепечатывать код с бумаги на компьютер быстрее, чем писать «из головы», хотя это, конечно, и так. В написанном коде проще обнаружить, все ли необходимые переменные и процедуру используются. После этого не понадобится десять раз прокручивать программу на экране в поисках нужного места. Это экономит время. Но это также вторично.

Основная идея написания кода на бумаге в том, что этим форсируется завершение стыковки, происходит упорядочение мыслей. Записать код – значит четко сформулировать решение. Впрочем, мне встречались случаи, когда довольно смутные мысли оформлялись в виде кода, «примерно передающего идею», а потом практически подгонялись под ответ вариациями с индексами массива – заменами *i* на *i+1* и т.п. Иногда это приводит к успеху, но чаще угадать не удается, и требуется сконцентрироваться, додумать и сразу написать верный код. В большинстве случаев приходится «переобдумать» и переписать значительную часть программы – написание недообдуманной программы помогает, уменьшая нагрузку на мозг, но приводит к очень большой потере времени. Впрочем, большой опыт и высокая техника иногда позволяют «на лету» перекроить программу, делающую не то и не так, в правильно и быстро работающую. Фаза стыковки в этом случае может быть исключена вовсе. Описанная ситуация все-таки является «высшим пилотажем», требующим внимания всех трех участников команды, или хотя бы двух из них, а также изрядной доли везения. Несмотря на то, что я неоднократно участвовал в решении задач указанным образом, обычно лучше все-таки заранее подумать и разложить по полочкам все детали, чем потом спешно сшивать разрозненные куски в подобие решения, почему-то выдающее верные ответы.

Обычно пропуск стыковки и переход сразу к реализации возникал либо по причине усталости и нежелания сосредоточиться и тщательно продумать решение, либо из-за желания немедленно начать что-нибудь писать, чтобы не терять время и не повышать нервозность обстановки в команде из-за простоя компьютера.

Часто стыковка выявляет ошибки в решении, неэффективность решения и т. д. Она же наводит на мысль, как придумать другое, существенно более простое решение. Это еще один аргумент в пользу того, что десять минут размышления могут в дальнейшем сэкономить полчаса. На этапе стыковки необходимо «раскрыть кирпичики», вспомнить, как же пишутся эти якобы известные стандартные алгоритмы.

Вариант при нехватке времени: один из участников пишет основную часть на компьютере, а второй – стандартный алгоритм на бумаге, а потом «вбивает» его в компьютер. При этом необходимо тщательно согласовать интерфейс – не только для ускорения и упрощения «вбивания», но, и, в первую очередь, для взаимопонимания того, что же собственно требуется. Второй участник, которого просят написать алгоритм на бумаге, в 80% случаев должен спросить: «А зачем в решении этот алгоритм?» Тут первый участник, скорее всего, начнет мяться и запинаться, и выяснится, что он придумал что-то сложное и громоздкое, а этот алгоритм нужен как подзадача чего-то другого, что решается само по себе значительно проще.

У разных участников соревнований, в том числе и успешных, различное отношение к этапу стыковки. Я считаю этот этап очень важным и настаиваю на его выполнении. Этот этап сложен, требует большой концентрации, но немного времени. Им часто пренебрегают, что, при недостаточном опыте и интуиции, может привести к печальным результатам. Пренебрегают обычно из-за усталости в конце соревнований или экономии сил в начале. Однако, если только вы не экономите время, переключаясь на другую задачу, я рекомендую уделить внимание стыковке.

Противоположная концепция – сразу начать писать. При достаточном опыте, интуиции, уверенности, что «задача простая, раз ее все сдают» или критическом недостатке времени в конце, когда необходимо хоть как-то попытаться ее решить, стыковку можно опустить, точнее, производить одновременно с написанием решения. Этот подход применяется повсеместно, но вряд ли его можно рекомендовать.

2.5. Реализация

На этом этапе собственно пишется программа. Иногда предпочтительнее программирование «сверху вниз», иногда – «снизу вверх», или их комбинация.

Первый подход используется, когда существует общее видение программы – тогда пишется основная часть, а функции и процедуры не реализуются, только согласовывается их интерфейс. Это делается достаточно быстро, если разбиение на подпрограммы достаточно удачно, и позволяет окончательно уложить мысли в голове. Также упрощается отладка, понимание и дописывание другими участниками.

Подход «снизу вверх» используется, когда требуется что-то писать, не очень понятно, что именно, а время уходит. Тогда можно сначала написать то, что потребуется в любом случае, например – ввод входных данных, функции для

работы с геометрией или арифметику повышенной точности. Хорошо, если в команде устоялись реализации стандартных алгоритмов, конвенции о названиях переменных и функций и т. д. При необходимости потратьте полминуты и допишите комментарии о том, что за значения хранятся в каждой из переменных, что возвращает та или иная функция и т. д.

На практике, конечно, используются смешанные подходы.

2.6. Тестирование и отладка

Добившись того, чтобы программа компилировалась, необходимо убедиться в ее правильности. Проблемы могут быть в мелких ошибках, допущенных в процессе написания: перепутанные имена переменных, неверный знак в формуле и т. д. Решение может быть принципиально неправильным или неэффективным. Размер массивов может быть недостаточным или, напротив, чрезмерным, что будет вызывать ошибку «превышен предел памяти».

Поэтому программу необходимо тестировать, если, конечно, речь идет не о последней минуте соревнований. Программу, дописанную за три минуты до конца, не следует тестировать только при условии, что тестирующая система работает очень нестабильно, и на посылку решения в жюри одной минуты мало.

Первое правило тестирования – проверяйте задачу на тесте (наборе входных данных) из примера. Какой бы правильной ни казалась ваша программа, каким бы простым ни был тест из примера, все равно в половине случаев тест из примера не пройдет. Все-таки решение пишется в обстановке нервного напряжения и на скорость. Далее, не ленитесь придумывать свои тесты. Вводите много «маленьких» тестов. Старайтесь не стирать тест, однажды введенный в компьютер. Если вы хотите слегка изменить его, предварительно скопируйте – пусть лучше будет два теста.

Второе правило – внимательно проверяйте, что программа выдала на тесте. Очень часто, когда программа правильно работала на девяти тестах, придуманных командой, но выдает неправильный ответ на десятом, команда этого не замечает, поскольку запускает программу на десятом тесте только для «очистки совести».

Отметим, что в достаточно сложных задачах помогает встраивание в программу элементов автоматической проверки. Очень полезна процедура `assert` или ее аналоги. Например, после сортировки можно проверить, действительно ли массив отсортирован. Если это не так, пусть программа завершается с ошибкой. Если задача достаточно сложная и включает не только сортировку, такие предосторожности практически наверняка окупятся. Единственная проблема – не недостаток времени, а лень, ведь на тренировках все десять раз писали тот или иной алгоритм. К сожалению, никакая тренировка не гарантирует безошибочного написания на соревнованиях даже таких простых алгоритмов, как, например, алгоритм Евклида, алгоритм Дейкстры или двоичный поиск.

Кроме «маленьких» тестов, необходимо всегда проверять решение на так называемом «максимальном тесте». Для каждой задачи следует, если это возможно, сгенерировать «максимальный тест» – полностью случайный большой тест с максимальными ограничениями. Сгенерированный таким образом тест не во всех задачах будет худшим по времени работы программы. Далеко не во всех – и хорошо бы понимать, в каких. Но на практике вы будете поражены, увидев,

насколько часто такой тест приводит к ошибке времени выполнения. Конечно, ответ к такому тесту проверить нелегко, но часто по ответу легко понять, что он – неправильный. Добавление в программу многочисленных проверок, лучше всего – исчерпывающих (если задача большая и трудная), облегчает использование «максимальных тестов».

Вечная дилемма: что лучше – тестировать или проверять логику? Если в программе есть небольшая часть, вызывающая большие сомнения, лучше ее обдумать и сразу написать правильно. Потому что если в программе есть и другие ошибки, тестирование может оказаться долгим и утомительным. Если вы не тестируете программу, распечатайте решение и проверяйте его по распечатке, освободив компьютер для других участников команды.

Необходимо выдерживать баланс между этими подходами. Всегда следует проверять программу на нескольких «небольших» тестах. В случае неверного ответа снова имеется альтернатива: отлаживать программу или искать ошибки, внимательно читая ее код (как правило, опять же по распечатке). Помните, что для отладки часто достаточно трех-пяти минут, тогда как поиск ошибок на бумаге легко может затянуться на полчаса. Это занимает одного из членов команды, нервирует всех троих и к тому же увеличивает штрафное время. Выбирать следует по обстоятельствам.

По ходу соревнований приходится регулярно принимать тактические решения. Проверять решение дальше или отправить его в жюри? Тестировать, отлаживать программу, или искать ошибки по распечатке? Писать решение задачи в одиночку или вдвоем? Важно правильно расставлять приоритеты. Что важнее в последний час соревнований – надежное решение одной задачи или рискованная попытка решить еще две?

На практике сложно уделить тактике достаточно внимания. Опишу единственный метод, который оказывался полезным во всех без исключения случаях. В самом начала соревнования возьмите чистый лист бумаги и выпишите на нем в столбик буквы, соответствующие задачам. В дальнейшем вычеркивайте решенные задачи. Удобно также помечать задачи, по которым есть идеи, или решения к которым частично написаны. Обнаружив задачу, на решение которой заведомо не хватит времени, также вычеркните ее из списка.

2.7. Посылка на проверку в жюри

Не забывайте про отладочную информацию, включение/выключение оптимизации и проверок переполнение арифметики, стека, выхода за границы массива – если вы, конечно, не пишете на языке *Java*. Существуют сторонники макроса *DEBUG* в олимпиадном программировании, которым можно отметить, какие части программы, выполняются при отладке, а с какими программа посылается в жюри. Иногда его использование оправдано, но обычно очень уж лень им пользоваться. Впрочем, это приводит к многочисленным «забыл включить проверку переполнения», «забыл убрать отладочную информацию» и т. д. А когда из жюри приходит сообщение «Неверный ответ», отладочная информация возвращается обратно, и затем, после исправления ошибки, ее опять забывают убрать. Та же история происходит, когда требуется выводить информацию в стандартный вывод и читать ее из стандартного ввода, а для отладки используется файл. Чтобы не забывать о том, что перед посылкой следует что-то изменить,

можно в клиенте для посылки решения в строке с названием задачи каждый раз писать: «Не забыть убрать файлы» или что-нибудь в таком же духе. Помогает очень сильно – реально может спасти от трех-четырех лишних попыток за контест.

Не смотрите, как ваши товарищи по команде посылают программу в жюри, как они нажимают кнопки в клиенте и т. д. Это отнимает ваше время и внимание, а они нервничают, что вы не готовитесь и не обдумываете другую задачу. Впрочем, если уж вы наблюдаете за ними, проконтролируйте, чтобы они выбрали нужный файл, нужный язык программирования и нужную задачу, включили/отключили проверки и т. д.

Не смотрите просто так в монитор, ожидая ответа. Сервер, проверяющий решения участников, может быть перегружен. Он может быть временно в нерабочем состоянии – это типично для соревнований самого разного уровня. Тестируйте эту задачу, пишите другую, думайте над третьей и т. д. В конце соревнований участники часто вносят «случайные изменения» и посылают программу в жюри еще раз, вносят и посылают и т. д. Однако, если осталось хотя бы пять минут, лучше придумайте пару небольших содержательных тестов и проверьте программу на них. Если обнаружится ошибка – сконцентрируйтесь и исправьте ее, а не просто добейтесь верного ответа на данном конкретном примере. Помните, что в программе часто есть повторяющиеся места, поэтому одна и та же ошибка могла быть допущена многократно.

Довольно часто обнаруживается фрагмент написанной программы, который может быть улучшен. Приемлемость существующей реализации может зависеть от трактовки условия задачи, нетривиальных свойств использованных алгоритмов, даже от некоторой доли везения, поскольку иногда участники допускают столь необычные и редко проявляющиеся ошибки, что заготовленные тесты эти ошибки не выявляют. Вы можете улучшить этот сомнительный фрагмент и послать программу на проверку, однако подобные действия часто наоборот приводят к увеличению количества ошибок в программе. Поэтому перед внесением исправлений, в необходимости которых вы не уверены, сделайте резервную копию всего решения. Обнаружив впоследствии действительно серьезную ошибку, вы, вероятно, захотите к этой копии вернуться.

3. Применение предложенной схемы решения к конкретной задаче

Теперь настало время продемонстрировать описанную схему на примере решения конкретной задачи. Приведем условие задачи.

Задача «Непоглощающий детерминированный конечный автомат».
(Non Absorbing Deterministic Finite Automaton (DFA)).

Автор задачи: Андрей Станкевич.
Источник: Летние сборы команд-участниц чемпионата мира ACM по программированию. Петрозаводск, 2003.

Расположение: задача № 201 в архиве олимпиадных задач Саратовского государственного университета http://acm.sgu.ru/

Условие задачи сформулировано на английском языке. Перевод на русский выполнен мною. Аббревиатура «DFA» переводится как «ДКА» – детерминированный конечный автомат.

Ограничения и требования:

- ограничение по времени: 2с;
- ограничение по памяти: 64 Мб;
- входные данные: стандартный ввод;
- выходные данные: стандартный вывод.

В теории компиляторов и языков широко используются конечные автоматы. Детерминированный конечный автомат (ДКА) – это упорядоченный набор $<\Sigma, U, s, T, \varphi>$, где Σ – конечное множество, называемое входным алфавитом, U – конечный набор состояний, s из U – начальное состояние, T – множество терминальных состояний из U, и, наконец, $\varphi: U \times \Sigma \to U$ – функция переходов.

Входом автомата является строка α над алфавитом Σ. Первоначально автомат находится в состоянии s. На каждом шаге он читает первый символ входной строки и изменяет свое состояние на $\varphi(u, c)$, где u – текущее состояние. После этого первый символ входной строки удаляется, и шаг повторяется. Если к моменту исчерпания входной строки автомат будет находиться в терминальном состоянии, то говорят, что он допускает исходную строку α, в противном случае – отвергает ее.

Иногда для упрощения автомата вводится понятие непоглощающих ребер. Это значит, что в дополнение к функции переходов φ вводится также функция $\chi : U \times \Sigma \to \{0, 1\}$. Тогда при совершении перехода из состояния u по символу c, первый символ из входной строки удаляется, только если $\chi(u, c) = 0$. Если же $\chi(u, c) = 1$, то входная строка остается без изменений, и следующий переход производится из нового состояния, но по тому же символу c. В первом случае говорят, что произошел переход по поглощающему ребру, во втором – по непоглощающему.

По определению такой автомат допускает строку α, если после некоторого числа шагов он попадает в терминальное состояние, а строка оказывается пустой.

Задача: по данному ДКА с непоглощающими ребрами найти количество строк данной длины N, которые он допускает.

Формат входных данных

Первая строка входного файла содержит Σ – подмножество английского алфавита (несколько маленьких латинских букв).

Вторая строка содержит $K = |U|$ – количество состояний автомата $(1 \le K \le 1000)$. Состояния нумеруются от 1 до K.

Третья строка содержит S $(1 \le S \le K)$ – начальное состояние и $L=|T|$ – количество терминальных состояний, а затем L различных целых чисел со значениями от 1 до K каждое – номера терминальных состояний.

Следующие K строк содержат по $|\Sigma|$ целых чисел каждая и определяют функцию φ.

Затем располагаются K строк определяют функцию χ тем же способом. Последняя строка входного файла содержит N ($1 \le N \le 60$).

Формат выходных данных

Выведите единственное число – количество строк длины N над алфавитом Σ, допускаемых данным ДКА.

Приведенная ниже таблица содержит описание примера.

Пример входных данных	Пример выходных данных
ab 2 1 1 2 2 1 1 2 0 1 0 0 3	2

В этом примере автомат допускает две строки: “aaa” и “abb”.

3.1. Чтение условия

Условие этой задачи достаточно просто для прочтения. Во-первых, оно написано на русском языке. Во-вторых, оно практически полностью формализовано, и литературное введение состоит из одной фразы: «В теории компиляторов и языков широко используются конечные автоматы». Все. Больше вас ничего не отвлекает. Можете сказать спасибо автору задачи, избавившему вас от ненужных деталей, которые, впрочем, могли бы несколько оживить это сухое и формальное условие задачи, изложенное последовательно, связно и без подвохов. Понятие ДКА, которое может быть неизвестно читателю, определено в тексте.

Однако Вы, дорогой читатель, вероятно, прочитали это условие «наискосок», невнимательно и небрежно, опустив некоторые детали. Скорее всего, вы поняли, что есть какая-то модификация конечного автомата, и следует посчитать количество строк, которое она допускает. Если вы не знаете, что такое «конечный автомат» и «допускает», то тогда у вас, вероятно, сложилось еще более смутное представление об условии данной задачи.

Вы можете возразить, что прочитали условие невнимательно потому, что читаете этот текст, а вовсе не решаете задачу. Очень зря. Вы думаете, на соревновании, решая реальные задачи, вы сможете читать условие иначе, не так, как сейчас?

Теперь, пожалуйста, вернитесь к условию и прочитайте его еще раз, очень внимательно.

3.2. Построение математической модели

Этот этап является естественным продолжением предыдущего. Вы уже прочитали условие? Скорее всего, вы поняли в лучшем случае половину.

Перечитайте условие еще раз: медленно, по одному предложению, при необходимости используя ручку и бумагу. Не ленитесь, разберитесь, что требуется в задаче.

Контрольный вопрос: приведите пример входных данных, ответ для которого – единица. Не на уровне идеи, а в числах. Если он существует, запишите его вместе с ответом. Этот пример еще пригодится вам при тестировании. Встретив рассматриваемую задачу на настоящих соревнованиях, немногие участники действительно будут придумывать какой-либо набор входных данных сразу после прочтения условия, чтобы не напрягаться. Поэтому события часто развиваются по следующему сценарию. Один участник спрашивает другого: «Ты читал задачу F?" Другой отвечает: «Да, там какие-то конечные автоматы, какие-то допускаемые строки. Что-то непонятное и, наверное, нудное. Лучше почитаем другие задачи» – и они откладывают эту задачу в сторону, даже приблизительно не поняв ее условия. При этом один из участников потратил время на чтение этой задачи, и оба – на обсуждение.

Итак, строим математическую модель. Сначала необходимо понять, что такое «детерминированный конечный автомат».

3.2.1. Конечные автоматы

Для решения рассматриваемой задачи не требуется никаких предварительных знаний о конечных автоматах. Однако у большинства участников студенческих олимпиад по информатике наверняка есть интуитивное представление о конечных автоматах. Представление это, как правило, обрывочное, синтетическое и, к тому же, искаженное, что зачастую оказывается хуже, чем отсутствие каких бы то ни было знаний в рассматриваемой области.

Проблемы проистекают из того, что детерминированный конечный автомат – это математический объект, рассматриваемый вместе с регулярными языками и соответствующими им грамматики. Само же словосочетание «конечный автомат» применяется значительно более широко, вводя участника соревнований в заблуждение, что он встретил знакомое понятие. Также учтите, что у авторов задачи может быть другое представление о том, что такое ДКА, даже неверное и отличающееся от определения в классической книге [18]. Сейчас вы должны понимать, что такое ДКА, иначе вам следует вернуться к чтению условия.

Для создания ясного, хорошо сформированного представления о конечных автоматах следует порекомендовать учебник для ВУЗов [19]. В нем внятно проведено различие между конечным автоматом как техническим элементом и математическим объектом, распознающим или преобразующим некоторый язык. Хотя это различие относится скорее к терминологии и предметной области, чем к математической сущности автоматов, оно обычно порождает много путаницы и непонимания.

Классическая книга [18] посвящена теории автоматов и соответствующих формальных языков и грамматик. В частности, в ней описано понятие *детерминированный конечный автомат*, о котором и идет речь в задаче. Следует понимать, что здесь понятия «входное воздействие» и «входной символ» эквивалентны – входное воздействие берется из множества возможных входных воздействий, входной символ – из входного алфавита.

Конечный автомат можно рассматривать как ориентированный граф. Алгоритмы, применяемые к задачам на графах, часто полезны и для конечных автоматов. Говоря об олимпиадных задачах, в условии не обязательно должен непосредственно фигурировать конечный автомат – он может легко прослеживаться, а может даже требовать специального построения.

3.2.2. Окончательное понимание условия

Имеется ДКА с непоглощающими ребрами – ребрами, при совершении перехода по которым не считывается символ. В этом случае следующий переход происходит из нового состояния, но по тому же символу. Это будет продолжаться до тех пор, пока не произойдет переход по поглощающему ребру, либо пока процесс не зациклится. Важно, анализируя условие задачи, не упустить из виду возможность существования циклов из непоглощающих ребер.

ДКА с непоглощающими ребрами сводится к ДКА без них. Из каждого состояния по каждому символу происходит переход в некоторое другое состояние, возможно, после прохождения автоматом по цепочке непоглощающих ребер. Исключением является попаданием в цикл. Чтобы учесть эту возможность, добавим еще одно состояние «Недопуск», не являющееся терминальным. Поскольку после попадания в цикл из непоглощающих ребер очередной символ входной строки никогда не будет удален, то автомат по определению не допускает исходную строку. Соответственно, вместо перехода по непоглощающему ребру, приводящему в цикл, автомат может совершить переход по поглощающему ребру в состояние «Недопуск», и исходная строка не будет допущена. Все ребра из состояния «Недопуск» ведут в него же.

Чтобы найти количество строк данной длины, допускаемых данным ДКА, достаточно перебрать все строки указанной длины и последовательно подать их на вход автомату.

Обратимся к примеру входных данных. ДКА, описанный в нем, имеет два состояния. Первое состояние – начальное, второе – терминальное. Необходимо посчитать количество допускаемых строк из трех символов.

На рис.1 изображен данный ДКА с непоглощающим ребром, выделенным пунктиром. На этом рисунке начальное состояние подчеркнуто, а терминальное выделено двойным контуром.

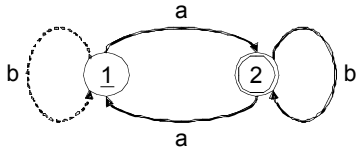

Рис.1. ДКА с непоглощающими ребрами

Преобразуем автомат для того, чтобы избавиться от непоглощающих ребер. На рис. 2 изображен ДКА, эквивалентный исходному. Состояние «Недопуск» помечено буквой «Н», а запятая позволяет не изображать кратные ребра.

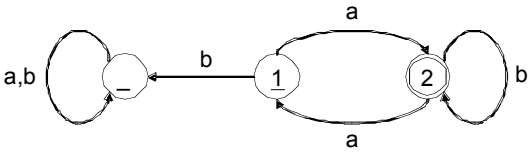

Рис.2. ДКА после удаления непоглощающих ребер

Из рассмотрения рисунков следует, что в первом состоянии на вход не должен подаваться символ “*b*”, иначе это приведет к зацикливанию. Соответственно, первый символ должен быть “*a*”. Учитывая, что терминальное состояние одно – второе, можно утверждать, что автомат допускает две строки: “*aaa*” и “*abb*”. Действительно, ответ – два.

3.3. Построение общей схемы решения

Рассматриваемая задача – достаточно стандартна, и ее решение хорошо поддается собиранию из «кирпичиков», описанному в разд. 2.3. Продемонстрируем ход мыслей при поиске решения.

В процессе осознания задачи было выявлено, что ДКА с непоглощающими ребрами можно преобразовать в ДКА без таких ребер. Пока неясно, насколько эффективно можно выполнить это преобразование. Теперь есть ДКА то ли с непоглощающими ребрами (если не удастся придумать эффективный алгоритм), то ли без них. Остается, собственно, найти ответ на задачу – количество допускаемых строк. Ясно, что это нельзя делать перебором и проверкой всех строк заданной длины, так как их может оказаться недопустимо много.

Как вообще можно посчитать количество строк, удовлетворяющих каким-то свойствам? Первый существующий подход – использование хитрых комбинаторных соображений и подсчет этих формул на компьютере. Второй подход (иногда оба подхода сочетаются): если система слишком сложна, необходимо найти некую рекуррентную формулу, вывести количество строк данной длины с какими-то свойствами через количество строк другой длины с другими свойствами. Затем применить динамическое программирование для последовательного вычисления всех этих значений. Таким образом, опыт подсказывает, что для подсчета количества строк, допускаемых данным ДКА, следует использовать динамическое программирование. Для подготовленного участника идея подсчета количества строк, обладающих некоторым свойством, с помощью динамического программирования сама по себе является «кирпичиком».

Практическое соображение: алфавит может состоять из `26` символов (`|A| = 26`). Строка может быть из `60` символов. Мало этого, общее представление о конечных автоматах свидетельствует о том, что все строки могут допускаться. Тогда ответ будет 26^{60}, что весьма много. Насколько много? При наличии компьютера или хорошего калькулятора, на этот вопрос ответить легко. Определим $\log_{10}(26^{60}) = 60\log_{10}(26) = 60(\ln(26) / \ln(10)) \approx 60 * 1.415 \approx 84.9$. Следовательно, в ответе может быть `85` цифр в десятичном представлении. Такое количество строк не поместится в стандартный тип (вида `Integer`). Поэтому при написании решения на языке *Pascal* придется вручную реализовать арифметику повышенной точности. Соответствующие функции также относятся к разряду «кирпичиков», входящих в обязательную подготовку.

Интуитивное представление о том, что количество строк, допускаемых данным ДКА, можно определить с помощью динамического программирования, необходимо развить. Сразу заметим: непохоже, чтобы эту систему можно было решить чисто комбинаторной формулой. Кроме этого замечания, проистекающего из опыта, к сожалению, ничего более конкретного сразу сказать нельзя. Хитрые сочетания формул, основанные на функции переходов, также, скорее всего, окажутся соотношениями динамического программирования. Поверим в это ощущение, проистекающее из опыта, и выделим подзадачи и рекуррентные соотношения.

Ключевая идея: преимущество модели конечного автомата в том, что вся история выражается одним числом – номером состояния, и этих состояний конечное количество. С точки зрения вопроса, допускается ли конкретная строка, имеет значение не только состояние автомата, но и количество поглощенных символов строки. Количество поглощенных символов совпадает с количеством прошедших шагов, тактов работы автомата – рассматривается уже ДКА без непоглощающих ребер. Рассмотрим пару `(state, k)` – текущее состояние – `state,` и прошло `k` шагов.

Рассмотрим строки длины `k`, обладающие следующим свойством: получив на вход такую строку, ДКА приходит из начального состояния в состояние `state`. Обозначим число этих строк через `f(state, k)`. Вычисление этого значения для различных пар `(state, k)` и является подзадачей. Для успешного применения динамического программирования необходимы рекуррентные соотношения, которые легко получаются, учитывая упомянутые свойства конечного автомата. Приведем эти соотношения:

$$f(state,0) = \begin{cases} 1, state = initial \\ 0, state \neq initial \end{cases}$$

$$f(state,k) = \sum_{\substack{st \in U, c \in A \\ \varphi(st,c)=state}} f(st,k-1),$$

где `initial` – начальное состояние ДКА.

Сформулируем схему решения, состоящего из двух этапов.

1. Превращаем ДКА с непоглощающими ребрами в ДКА без таковых. Наличие решения с помощью динамического программирования для ДКА без непоглощающих ребер укрепляет уверенность в том, что эффективное устранение непоглощающих ребер также возможно.
2. Применяем динамическое программирование, используя вышеприведенное рекуррентное соотношение. Целесообразно, хотя и не обязательно, использовать динамическое программирование «снизу вверх». Для нахождения ответа необходимо будет просуммировать $\sum_{t \in T} f(t,n)$ `f(terminal_state, n)` по всем терминальным состояниям `t`.

В заключение раздела отметим, что применение динамического программирования на конечном автомате, как и вообще на графе, является очень эффективным. Перекрывающиеся подзадачи автоматически выделяются: необходимо лишь отслеживать целевую функцию во всех состояниях автомата. В качестве примера можно привести задачу, при решении которой используется

техника, полностью аналогичная второй части решения настоящей задачи: задача «Currency Exchange» («Обмен валюты»). Автор: Николай Дуров. Источник: Четвертьфинал NEERC-2001, северный подрегион. Задача размещена на сайте http://acm.timus.ru/ под номером 1162. Фактически математическая модель этой задачи является конечным автоматом, где состояниями являются валюты, а переходами между ними – обменные пункты.

Следует указать еще одну известную задачу: «Censored!» («Цензура!»). Автор: Николай Дуров. Источник: Четвертьфинал NEERC-2001, северный подрегион. Задача размещена на сайте http://acm.timus.ru/ под номером 1158. Распространенный способ ее решения состоит из двух этапов: построение конечного автомата, распознающего набор строк (алгоритм построения называют алгоритмом Ахо-Корасика), и использование динамического программирования на полученном конечном автомате. Похожая задача без динамического программирования – задача «Obscene words filter» («Фильтр грубых слов»). Автор: Павел Атнашев. Источник: чемпионат УрГУ 25.10.2003. Задача размещена на сайте http://acm.timus.ru/ под номером 1269.

Мною были проанализированы все задачи в архивах олимпиадных задач http://acm.sgu.ru/ и http://acm.timus.ru/ на первое июня 2005 года. Кроме задачи, рассматриваемой в настоящей статье, и трех только что упомянутых задач, в остальных задачах использование конечных автоматов ограничивается алгоритмом Кнута-Морриса-Пратта.

3.4. Стыковка

Рассмотрим первый этап. Как его выполнить?

В разд. 3.2 («Построение математической модели») была установлена теоретическая возможность выполнения данного этапа. Как обсуждалось выше, результатом построения математической модели обычно является некий примитивный и неэффективный алгоритм решения – формально записанное понимание того, как из входных данных получаются выходные. Попробуем применить его на практике.

Последовательно переберем все пары (состояние, символ). Предположим, что ДКА находится в рассматриваемом состоянии, а выбранный символ является очередным на входной ленте. Совершаем переход по данному символу. Если переход произошел по непоглощающему ребру, совершаем следующий переход по тому же символу. Если переход снова произошел по непоглощающему ребру, повторяем тот же маневр. В результате либо символ будет поглощен, либо автомат войдет в цикл. В первом случае можно изменить переход из исходного состояния, из которого автомат прошел по цепочке непоглощающих ребер, сразу поставив переход в конечное состояние. Во втором – можно поставить ребро в некоторое фиктивное состояние – «Недопуск», которое не является терминальным. Оно замкнуто на себя при переходе по всем символам. Попадание в это состояние будет соответствовать входу в цикл из непоглощающих ребер в исходном автомате.

Непосредственное «проговаривание» решения привело к неплохому результату. Оценим эффективность решения. Как определить, что произошло попадание в цикл? Если за $|U|$ шагов автомат не перейдет по непоглощающему ребру, то это будет означать, что он вошел в цикл ($|U|$ – число состояний автомата). Верхняя оценка времени работы: $|U|*|U|*|A|$ – порядка $26 * 10^6$.

Много это или мало? По меркам 2003 года, в котором была предложена задача, за секунду можно было выполнить 10^7 совсем простых операций на языке высокого уровня, таком, как языки *Pascal* или *C++*. Соответственно, такой способ устранения непоглощающих ребер должен требовать не более секунды, что в первом приближении представляется допустимым – один из двух этапов требует половину времени, ведь ограничение на один тест составляет две секунды, как сказано в условии.

Приведем псевдокод для устранения непоглощающих ребер. В псевдокоде множество всех состояний обозначено через `State`.

```
for c in Alpha do      // По всем символам алфавита
  for i in State do    // По всем состояниям автомата
begin
  cur := i             // Текущее состояние
  z := n               // Количество состояний
  // Пока ребро-переход из состояния k по символу j –
  // поглощающее, и еще сделано не очень много переходов
  while (χ[ cur,c] = 1) and (z > 0) do
  begin
    cur := φ[ cur][ c]  // Перейти
    z := z - 1          // Уменьшить счетчик
  end
  // Если все еще очередное ребро – непоглощающее
  if (χ[ cur][ c] = 1) then
    φ[ i][ c] := 0      // Значит – цикл
  else
    φ[ i][ c] := φ[ cur][ c] // Иначе переставляем ребро
  // И теперь снимаем пометку «непоглощающее ребро»
  χ[ i][ j] := 0
end
```

Кроме рассмотренного, существует также решение, которое устраняет непоглощающие ребра за `O(|U|*A)` с помощью поиска в глубину. Не следует искать его на этом этапе – и в практическом программировании, и в олимпиадном. Необходимо рассмотреть решение второй части, а потом уже решать, необходимо ли более эффективно устранять непоглощающие ребра, или предложенный вариант приемлем. В данной работе ограничимся неэффективным способом устранения непоглощающих ребер.

Перейдем к рассмотрению второго этапа. Как его выполнить?

Теперь можно утверждать, что построили ДКА без поглощающих ребер – как будто уже держим его в руках.

Используем динамическое программирование «снизу вверх» для поиска ответа. В соответствии с теорией, выражаем ответ подзадачи через ответы, ранее найденные для других подзадач. Это осуществляется следующим образом. Рекуррентные соотношения, приведенные в разд. 3.3, выражают значение функции `f(state, k)` через значения функции `f(st, k-1)` для различных состояний `st` $\in$ `U`. Однако для данного состояния `state` неудобно определять, из какого состояния `st` мог произойти переход в это состояние на предыдущем шаге. Поэтому удобнее зафиксировать пару `(state, k)` и, перебирая все символы

алфавита, определять, в какие состояния s (в зависимости от очередного символа) будет происходить переход на следующем шаге. При этом каждый раз необходимо увеличивать значение функции f(s, k+1) на величину f(state, k).

Заведем для значений функции f одноименный массив f[state, k]. Запишем на псевдокоде решение для второго этапа с помощью динамического программирования, основанное на рекуррентных соотношений, приведенных в разд. 3.3. Учтите, что цикл по всем состояниям включает фиктивное состояние «Недопуск» под номером ноль, добавленное после удаления непоглощающих ребер. Поэтому вместо множества состояний State будет встречаться множество $State_0$, включающее еще и состояние «Недопуск».

```
// Один способ оказаться в начальном состоянии
f[ init,  0] := 1
// И ноль – во всех остальных состояниях
for i in State0 do
  if (i <> init) then
    f[ i,0] := 0

// Считаем количество допускаемых n-символьных строк
for k := 1 to n do
for st in State0 do   // По всем состояниям
for c in Alpha do     // По всем символам алфавита
   f[ φ[ st,c] ,k] := f[ φ[ st,c] ,k] + f[ st,k-1]

ans := 0
// Суммируем по всем терминальным состояниям
for st in TerminalState do ans := ans + f[ st,n]
```

Зачем заводить двумерный массив? Ведь в каждый момент используется только предыдущее значение. Достаточно одномерного массива sum[state] – количество способов оказаться в данном состоянии, количество прошедших шагов или, что то же, пропущенных символов, зафиксировано.

Изменим псевдокод, чтобы вместо двумерного массива использовался одномерный.

```
// Один способ оказаться в начальном состоянии
sum[ init] := 1
// И ноль – во всех остальных состояниях
for i in State0 do
  if (i <> init) then
    sum[ i] := 0

// Считаем количество допускаемых n-символьных строк
for k := 1 to n do
begin
  sum2 := sum
  for i in State0 do sum[ i] := 0
  for i in State do
    for c in Alpha do
      sum[ φ[ i][ c]] := sum[ φ[ i][ c]] + sum2[ i]
end
```

```
ans := 0
// Суммируем по всем терминальным состояниям
for st in TerminalState do
    ans := ans + sum[ st]
```

Оценим это решение. Длинные числа складываются `N * |U| * |A|`, что в худшем случае может достигать `60*1000*26` ≈ `1.5` миллионов раз. Второй секунды на это достаточно. И на копирования временного массива `sum2` в `sum` – тем более. Скорее всего, для первой части не потребуется искать более эффективное решение.

3.5. Реализация

В реализации рассмотренной задачи важно четко разделить две части решения: превращение ДКА с непоглощающими ребрами в ДКА без таких ребер и дальнейший подсчет с помощью динамического программирования. Заметим, что можно начать с реализации второй части, предполагая, что все ребра – поглощающие. Тогда эту часть можно сразу же протестировать, одновременно проверив ввод из файла, имена входных и выходных файлов и т. д. Кстати, на этапе реализации всегда следует внимательно перечитать описание формата входных и выходных данных.

Необходимо ли эффективно реализовывать первую часть, связанную с удалением непоглощающих ребер? Если вам практически все равно, реализовывать более или менее эффективный вариант, или более эффективный вариант уже написан на бумаге, то и используйте его. Всегда может оказаться, что другую часть программы вы реализовали не так эффективно, как предполагало жюри, и можно тем самым повысить вероятность того, что программа все-таки не превысит предел времени.

Приведенные выше соображения, применительно к рассматриваемой задаче, показывают, что в ней можно использовать неэффективную реализацию первого этапа решения, которая все-таки принципиально проще, и в ней сложнее ошибиться. Разумно написать ее, отладить программу и, лишь получив сообщение от жюри *time limit exceeded* (превышен предел времени), что маловероятно, переписать эту часть более эффективно. Простой вариант пишется быстро, и он облегчит поиск ошибок в других частях программы. Наша команда часто реализовывала существенно более сложные решения, чем требовалось в задаче, используя общие, отработанные, но технически сложные методы. К сожалению, на практике из-за специфики конкретной задачи более сложные решения иногда оказывались менее эффективными, что преодолевалось множеством локальных оптимизаций.

Псевдокод для второй части уже был приведен в разд. 3.4. Как уже было отмечено, ответ может иметь очень большое число разрядов. Поэтому потребуется арифметика повышенной точности («длинная арифметика»). Для написания решения на языке `Pascal` (*Borland Delphi*) длинную арифметику придется реализовывать самим. Для изучения арифметики повышенной точности и компьютерной арифметики в целом следует порекомендовать книгу Д. Кнута [20]. Практически вся четвертая глава («Арифметика») этой книги является очень

полезной для решения олимпиадных задач по программированию и должна быть внимательно прочитана с первой страницы до последней. Впрочем, изучения книги Д. Кнута для этого недостаточно.

Потребность в длинной арифметике возникает в олимпиадных задачах достаточно часто. Поэтому реализация соответствующих функций должна быть хорошо отработана и являться, в терминологии разд. 2.3, одним из «кирпичиков». В данной задаче для повышения эффективности разумно реализовать длинную арифметику по основанию 10^9 – хранить по девять десятичных цифр в одной ячейке массива, задающего длинное число.

Разработанная программа приведена в приложении. Исходный код и скомпилированная программа опубликованы на сайте http://is.ifmo.ru/, раздел «Статьи».

Для иллюстрации того, что решение построено из «кирпичиков», программа разделена на несколько фрагментов, примерно этим «кирпичикам» соответствующих.

3.6. Тестирование и отладка

Какие же тесты следует написать в данном случае? Начнем с максимально простых. Вводим:

```
a
1
1 1 1
1
0
1
```

Что может быть проще? Один символ, одно состояние – начальное и терминальное, одно поглощающее ребро, ведущее в то же состояние. Число строк из одного символа – одна. В графической форме этот тест представлен на рис.3.

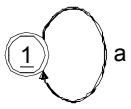

Рис. 3. Иллюстрация к первому тесту

Для этого теста ответ: `1`.

Если ответ, выданный вашей программой, разойдется с правильным на столь простом примере, вам будет легко искать ошибку. Не останавливайтесь на достигнутом! Поменяйте последнюю единицу на двойку. На тройку. На `60`. Используйте свой шанс обнаружить простой пример, на котором программа неверно работает. Не думайте, что если программа работает для двух, то она будет работать и для трех (имеется в виду последнее число во входных данных).

Теперь поменяйте ноль на единицу – сделайте ребро непоглощающим. Правильный ответ: ноль. Можете при этом попытаться заменить последнее число во входе также на ноль, хотя это и запрещено форматом входных данных, то есть

посчитать количество пустых строк, допускаемых таким автоматом – в ответе должна быть единица.

Следующая стадия: два состояния, причем только одно из них терминальное. Входные данные:

```
a
2
1 1 1
2
1
0
0
4
```

В графической форме этот тест представлен на рис. 4.

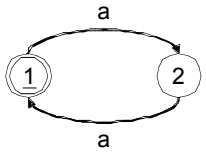

Рис. 4. Иллюстрация ко второму тесту

Для этого теста ответ: `1`.

Если поменять четверку на тройку, то в ответе будет ноль – из-за четности. Несмотря на всю примитивность этих тестов, их использование позволяет найти существенный процент ошибок. Причина в том, что программа вообще задумана для того, чтобы быть правильной. Поэтому сложные ошибки вполне могут проявляться на простых тестах. После этого обычно понятно, где ошибка, и ее можно исправить, не вдаваясь в детали, почему программа не работает на данном тесте. Впрочем, иногда лучше все-таки внимательно изучить, что же происходит в программе на этих входных данных, иначе можно исправить одну ошибку и пропустить другую, или даже добавить новую, которая, тем не менее, приведет к правильному ответу на данном тесте.

Заметьте, что в вышеприведенных примерах с двумя состояниями начальным и терминальным было первое состояние. Это нецелесообразно, поскольку некоторые ошибки могут остаться незамеченными. Поэтому варьируйте эти простые тесты, например, замените “*a*” в первой строчке на “*g*” и т. д. Проверьте случай, когда терминальных состояний нет вовсе.

Теперь необходимо проверить «длинную арифметику» и арифметику вообще, поскольку в предыдущих тестах проверялось только простое сложение. Пусть теперь в алфавите два символа, а в автомате – одно состояние.

```
sm
1
1 1 1
1 1
0 0
10
```

В графической форме этот тест представлен на рис. 5.

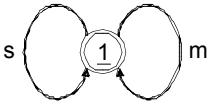

Рис.5. Иллюстрация к третьему тесту

Для этого теста ответ: $2^{10} = 1024$.

Поменяйте 10 на 20, 30, 60. Старайтесь каждый раз понять, похоже ли число в ответе на соответствующую степень двойки по первым и последним цифрам, количеству цифр. В принципе, полезно один раз выучить количество цифр и по несколько цифр в начале и конце у какого-нибудь числа вида 2^{100}, так как это может неоднократно пригодиться на соревнованиях.

Сделаем еще один хитрый тест. Все это, на самом деле, производится очень быстро, даже если тесты не были подготовлены заранее – на бумаге. Если вы готовите тесты на бумаге, проверяйте их и находите ответ заранее. Если ответ слишком длинный, продумайте, как по выходным данным программы проверить, «похожи» ли они на правильный ответ. Будьте в состоянии объяснить идею теста. Итак, следующий тест:

```
jnm
2
2 1 1
1 1 1
2 1 2
1 1 1
0 0 0
10
```

В графической форме этот тест представлен на рис.6.

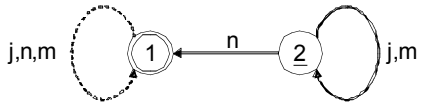

Рис.6. Иллюстрация к четвертому тесту

В чем смысл этого теста? Начальное состояние – второе. Конечное – первое. Все ребра из первого ведут в первое и непоглощающие. Следовательно, в первое состояние необходимо попадать последним шагом. Из второго состояния в первое ведет ребро, помеченное символом n. При переходах по остальным ребра состояния не изменяются. Все три ребра, исходящие из первой вершины – поглощающие.

Для этого теста ответ: $2^9 = 512$.

Старайтесь каждый раз придумывать хотя бы один содержательный тест – чуть-чуть содержательный. Такой тест обычно вскрывает очень много видов ошибок, хотя, конечно, может не повезти, и ошибка не будет обнаружена.

Следующая проверка – «максимальный тест». Для его генерации пишется специальная программа. На практике она может быть встроена в решение. Заметим, что генерируемый тест не является худшим для проверки удаления непоглощающих ребер, но эта проверка в нем проводится, причем довольно активно. Для каждого ребра случайным образом выбирается, является ли оно поглощаюшим. Генератор теста:

```
var i, j:integer;
    c : char;
begin
  for c := 'a' to 'z' do write(c);
  writeln;
  writeln(1000);
```

```
  write('1 1000 ');
  for i := 1 to 1000 do write(i,' ');
  writeln;
  for i := 1 to 1000 do
    for j:=1 to 26 do
      writeln(random(1000)+1);
  for i:=1 to 1000 do
    for j:=1 to 26 do
      writeln(random(2));
  writeln(60);
end.
```

Пожалуйста, внимательно изучите, что делает эта программа. Ответ на этот тест мы знаем: 26^{60}. В нем 85 цифр, как и в том, что выводит программа, но как полностью убедиться в правильности работы решения на этом тесте? Еще раз заметьте, что свойство ребра быть непоглощающим генерируется случайным образом. Не приведет ли это к ошибкам? Легко заметить, что число 26^{60} должно заканчиваться на цифру 6, тогда как вы это вряд ли получите, запустив программу на примере, сгенерированном вышеприведенным кодом. Причина именно в непоглощающих циклах, уменьшающих ответ. Не все строки длины 60 допускаются сгенерированным ДКА. Замените `random(2)` на ноль, перезапустите генератор и получите корректный тест, ответ на который будет 26^{60}, как и ожидалось.

3.7. Посылка на проверку в жюри

Здесь нет никаких специальных рекомендаций для данной конкретной задачи. Следуйте общим советам, описанным в разд. 2.7.

4. Заключение

В данной работе рассмотрены общие принципы решения задач в рамках командных студенческих соревнований по программированию формата ACM ICPC. Выполнена попытка формализовать и описать процесс решения задачи вместе с ключевыми аспектами для каждого этапа. Рассмотрен пример. Приведены комментарии и рекомендации.

Конечно, никакая инструкция не заменит реального опыта. Однако, на фортепиано почему-то принято учиться играть у преподавателя, и так начинали практически все великие композиторы. Соответственно, чтение этих рекомендаций должно быть не попыткой применить к себе чужие методы, а содержательным восприятием чужого опыта, изложенным в виде набора рекомендаций. В процессе моей олимпиадный карьеры я много обсуждал различные тактические и стратегические аспекты, точки зрения изменялись на диаметрально противоположные в течение дня. Могло быть и так, что в один сезон мы придерживались одной тактики, а в другой – другой. Тактика зависит от опыта, целей, соперников.

Важно тренироваться самостоятельно, особенно если вы чувствуете, что сильно отстаете хотя бы от одного из своих напарников. Важно быть заинтересованным.

Если вы систематически изучаете эту проблему, то, следовательно, вы всерьез нацелены на решение задач. Необходимо втянуться в этот процесс, жить решением задач, может быть, только им. Необходимо получать удовольствие от этого, чтобы можно было проснуться ночью и пойти решать задачи, быть готовым обсуждать задачи в любое время – во всяком случае, такое помешательство должно быть на начальном этапе карьеры. Конечно, придется потратить сотни часов и решить сотни задач, в том числе самостоятельно. Впрочем, это еще не гарантирует результата.

И благодарите создателей ACM ICPC за то, что в нем можно участвовать только в студенческие годы, причем в финале – не более двух раз. Съехавшую таким образом крышу будет очень непросто вернуть на место… Удачи Вам!

Я благодарен профессору СПбГУ ИТМО А.А.Шалыто, убедившему меня написать эту статью и внесшему множество предложений по ее улучшению.

Источники

1. *Асанов М.О., Парфенов В.Г.* Финальные соревнования чемпионата мира по программированию. Потрясающий успех петербургских команд // Компьютерные инструменты в образовании. 2001, № 2. http://ict.edu.ru/lib/ *Командный чемпионат мира по программированию ACM 2003/2004*. Северо-Восточный Европейский регион / Под ред. проф. В.Н. Васильева и проф. В.Г. Парфенова. СПб.: СПбГУ ИТМО. 2003.
2. *Богатырев Р.* К истории чемпионатов мира ACM по программированию // Мир ПК – Диск. 2004, № 6. http://is.ifmo.ru/belletristic/_acmhist.pdf
3. *Богатырев Р.* Нас не догонят?! // Мир ПК – Диск. 2005, №5. http://is.ifmo.ru/belletristic/_acm2005.pdf
4. *Брудно А.Л., Каплан Л.И.* Московские олимпиады по программированию. М.: Наука, 1990.
6. *Беров В.И., Лапунов А.В., Матюхин В.А., Пономарев А.Е.* Особенности национальных задач по информатике. Киров: Триада-С, 2000.
7. *Кирюхин В., Лапунов А., Окулов С.* Задачи по информатике. Международные олимпиады 1989-1996 гг. М.: ABF, 1996.
8. *Овсянников А., Овсянникова Т., Марченко А., Прохоров Р.* Избранные задачи олимпиад по информатике. М.: Тровант, 1997.
9. *Скиена С., Ревилла М.* Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям. М: Кудиц-Образ, 2005.
10. *Адамар Ж.* Исследование психологии процесса изобретения в области математики. М.: МЦНМО, 2001.
11. *Пуанкаре А.* О науке. М.: Наука, 1983.
12. *Сайт* «Генрих Саулович Альтшуллер, автор ТРИЗ, РТВ и ТРТЛ». http://www.altshuller.ru/
13. *Андреева Е.В.* Решение задач XIII междунардной олимпиады // Информатика. 2001. № 37, 40, 42–44.
14. *Окулов С.М., Шулятников Д.В.* Разбор задач международной олимпиады 2000 года // Информатика. 2001. № 12.

15. *Станкевич А.С.* Решение задач I Всероссийской командной олимпиады по программированию // Информатика. 2001. № 12.
16. *Станкевич А.С.* II Всероссийская командная олимпиада школьников по программированию // Информатика. 2002. № 12.
17. *Андреева Е.В.* Олимпиады по информатике. Путь к вершине // Информатика. 2001. № 38, 40, 42, 44, 46, 48; 2002. № 6, 8, 10, 12, 14, 16.
18. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.
19. *Карпов Ю.Г.* Теория автоматов. СПб.: Питер, 2002.
20. *Кнут Д.Э.* Искусство программирования. Т. 2. Получисленные алгоритмы. М.: «Вильямс», 2001.

Приложение. Полный исходный текст решения задачи «Непоглощающий детерминированный конечный автомат» на *Borland Delphi*

```
{$apptype console}   // Создать консольное приложение

// Включить проверки переполнения и отключить оптимизацию
{$o-,q+,r+}

uses Math, SysUtils; // Подключить модули Math и SysUtils
```

```
          {Длинная арифметика. Сложение и вывод числа}

const
  pow = 9;            // Длинная арифметика по 9 десятичных цифр
  base = round(1e9); // 10^pow
  m = 10;             // Требуемое количество таких «цифр»
type
  long = array[0..m] of integer; // Тип: длинное число

// Процедура сложения двух длинных чисел a и b
// Результат записывается в a
// Перед параметром b стоит ключевое слово var,
// так как передавать массив по значению слишком медленно,
// по ссылке гораздо быстрее
procedure add(var a : long; var b : long);
var i, c : integer;
begin
  c := 0;
  for i := 0 to m do
  begin
    c := c + a[i] + b[i];
    if c >= base then
    begin
      a[i] := c - base;
      c := 1;
    end else
    begin
      a[i] := c;
      c := 0;
    end;
  end;
// Если вдруг "цифр" в типе long окажется недостаточно,
// сигнализируется ошибка
  assert(c = 0);
end;

// Печать длинного числа
procedure print(var x : long);
var i, j : integer;
begin
  i := m;
  while (i > 0) and (x[i] = 0) do dec(i);
  write(x[i]);
  for j := i - 1 downto 0 do
    write(format('%.' + IntToStr(pow) + 'd', [x[j]]));
end;
```

```
                    { Константы и переменные}

const               // Две константы из условия задачи
  max_nst = 1000;  // Максимальное количество состояний
  max_na = 26;     // Максимальный размер алфавита

var
  alfa : string;  // Входной алфавит автомата
  na : integer;   // Количество символов во входном алфавите
  nst : integer;  // Количество состояний автомата
  ist : integer;  // Начальное состояние автомата
  ntst : integer; // Количество терминальных состояний автомата

  len : integer;  // Длина рассматриваемых строк
// Является ли состояние терминальным
  term : array [ 1..max_nst] of boolean;
// Функция переходов
  fi : array [ 1..max_nst, 1..max_na] of integer;
// Является ли ребро непоглощающим?
  ee : array [ 1..max_nst, 1..max_na] of boolean;

  sum, sum2 : array [ 0..max_nst] of long;
  i, j, k, z : integer; // Временные переменные
  ans : long;
```

```
                    { Ввод входных данных}

begin
  readln(alfa);      // Читаем алфавит
  na := length(alfa);// Имеет значение только размер алфавита
  read(nst);         // Читаем количество состояний
  read(ist);         // Читаем номер начального состояния
  read(ntst);        // Читаем количество терминальных состояний

// Читаем номера терминальных состояний
  for i := 1 to nst do
    term[ i] := false;
  for i := 1 to ntst do
  begin
    read(j);
    term[ j] := true;
  end;
// Читаем функцию переходов φ
  for i := 1 to nst do
  begin
    for j := 1 to na do read(fi[ i][ j] );
  end;
// Читаем функцию χ
  for i := 1 to nst do
  begin
    for j := 1 to na do
    begin
      read(k);
      ee[ i][ j] := k = 1;
    end;
  end;
// Читаем N – длину строк, количество которых необходимо найти
  read(len);
```

```
            {Этап 1. Устранение непоглощающих ребер}

  for j := 1 to na do          // По всем символам алфавита
  for i := 1 to nst do         // По всем состояниям автомата
  begin
    k := i;                    // Текущее состояние
    z := nst;                  // Количество состояний

// Пока ребро-переход из состояния k по символу j -
// поглощающее, и еще слелано не очень много переходов
    while (ee[ k][ j] ) and (z > 0) do
    begin
      k := fi[ k][ j];         // Перейти
      dec(z);                  // Уменьшить счетчик
    end;
// Если все еще очередное ребро - непоглощающее
    if ee[ k][ j]  then
    begin
         fi[ i][ j]  := 0;     // Значит - цикл
    end else
    begin
         fi[ i][ j]  := fi[ k][ j];// Иначе переставляем ребро
    end;
   // И теперь снимаем пометку «непоглощающее ребро»
       ee[ i][ j]  := false;
  end;
```

```
    {Этап 2. Подсчет количества строк, допускаемых автоматом,}
            {с помощью динамического программирования}

// Инициализация
  for i := 0 to nst do
    fillchar(sum[ i], sizeof(sum[ i]), 0);
  sum[ ist][ 0]  := 1;
```

```
// Последовательное вычисление
  for k := 1 to len do
  begin
    sum2 := sum;
    for i := 0 to nst do
      fillchar(sum[ i], sizeof(sum[ i]), 0);
    for i := 1 to nst do
      for j := 1 to na do
      begin
        add(sum[ fi[ i][ j]], sum2[ i]);
      end;
  end;
```

```
// Получение ответа
// Для этого суммируем количество способов оказаться
// в каждом терминальном состоянии
  fillchar(ans, sizeof(ans), 0);
  for i := 1 to nst do
    if term[ i]  then
      add(ans, sum[ i]);
```

```
                          {Вывод ответа}

  print(ans);
end.
```