Тестирование программ

Они писали код в тетради,
Задачи тестили в уме...
Спектакль преподавателей ЛКШ.2013.Июль

Тестирование программ, т.е. проверка уже написанных программ на наличие ошибок в них и поиск этих ошибок — это один из важнейших навыков программиста-олимпиадника, да и программиста-профессионала тоже. Я бы сказал, что он важнее любого конкретного отдельно взятого алгоритма, а то и группы алгоритмов.

В классических школьных олимпиадах вы весь тур решаете задачи, а проверяются они только после тура. (Или во время тура, но вам не сообщают результаты проверки — или сообщают, но только на тестах из условия — это все не очень существенно отличается.) В результате, если ваша программа не пройдёт часть тестов жюри, вы никак не сможете это исправить. Поэтому на классических олимпиадах тестирование — не просто важный, а сверхважный навык.

На современных школьных олимпиадах высокого уровня — на международной, всероссийской олимпиадах; также есть слухи (на сентябрь 2014 г.), что эта система может быть распространена на региональные (областные) олимпиады, — применяется система с «токенами», в которой участники могут проверить своё решение на полном наборе тестов во время тура олимпиады. На такую проверку налагаются определённые ограничения, но тем не менее это, конечно, существенно снижает важность умения тестировать. Тем не менее все равно навык тестирования остаётся важным; умея быстро найти тест, на котором ваша программа не работает, вы сможете все ошибки и исправить быстрее.

В олимпиадах, проводимых по стандарту ACM (это в первую очередь командные олимпиады), участники тоже могут сдавать решения на проверку на полном наборе тестов по ходу тура, но за каждую неудачную попытку начисляются штрафные минуты. В дополнение к этому, если решение не проходит хотя бы один тест, оно не засчитывается вообще. В результате умение быстро и надёжно тестировать здесь также является весьма существенным.

Итак, тестировать нужно уметь и уметь хорошо.

Когда вы решаете задачи — не важно, где — на учебных занятиях, на олимпиадах или ещё где-либо — всегда старайтесь все задачи решать с минимальным числом штрафных попыток. Старайтесь все свои решения проверять максимально тщательно, и воспринимайте любую неудачную попытку как показатель того, что тестируете вы все-таки неидеально. Радуйтесь всегда, когда сложная задача проходит все тесты с первой попытки, и огорчайтесь, если нет — даже если никаких штрафов за неверную попытку не предусмотрено.

Поэтому в этой теме мы обсудим, как же надо тестировать.

Часть I. Стратегия тестирования

Конечно, необходимость тестирования и необходимый объем сильно зависят от сложности задачи. Если вы пишите задачу, которая для вас очевидно является очень простой, и вы абсолютно уверены, что напишите её без ошибок, то её скорее всего можно и не очень тщательно тестировать. Часто бывает достаточно просто прогнать тест из условия и ещё пару тестов — и можно сдавать. Особенно это относится к ситуации, когда время ценно, а штраф за ошибочную попытку отсутствует — как в системе с токенами.

Аналогично, если до конца тура остаётся пара минут, то вы просто не успеете провести полный цикл тестирования. Но если задача не совсем элементарна, или если цена ошибки высока, то тестируйте задачу тщательно. Ни в коем случае не пренебрегайте этим; лучше вы потратите лишнее время на тестирование, но не получите штрафа и будете уверены, что задача работает.

Часть II. Какие тесты надо использовать

Главное правило тестирования следующее:

> Тестирование — это последовательный, систематический процесс.

В тестировании должна быть система, тестирование должно быть тщательным и аккуратным. Вы не должны тестировать задачу по принципу «подсуну несколько случайных тестов и посмотрю, разумные ли результаты выдаёт программа». Не следует придумывать какие попало тесты, не следует стараться понаворотить в один тест побольше всего интересного. Вместо этого надо чётко понимать, зачем вы делаете каждый очередной тест, понимать, какие тесты вы будете использовать дальше, и так далее.

Соображения, которые я излагаю ниже, конечно, применимы не к каждой задаче — но в каждой задаче смотрите, что из того, что указано ниже, применимо.

1. Минимальные тесты. Первым делом подсуньте своей программе *самый маленький тест*, какой только возможен. Это зачастую $N = 1$ или даже $N = 0$ или что-нибудь подобное. Часто бывает, что такой «самый маленький» тест вообще один в своём роде — например, существует только одна перестановка из 1 числа, или только один граф без петель с 1 вершиной. Если же нет — выберите любой из самых маленьких тестов. Можно даже попробовать два самых маленьких теста; например, если на вход подаётся массив из N чисел, то попробуйте два теста с $N = 1$: когда в массиве единственный элемент равен 1 и когда он равен, например, 137. (А вообще, см. ниже про разнообразие тестов — может быть, стоит проверить ещё и элемент, равный 0, -1 и -137.) Правда, в этом случае сначала убедитесь, что минимальным тестом не является вообще тест с $N = 0$!

(Вообще, многие любят начинать тестирование с теста из условия. На мой взгляд, в большинстве случаев это неправильный подход, почему — см. ниже.)

(Однажды ко мне обратился один школьник со словами, что его программа не работает в тестирующей системе, но у него она работает. Насколько я помню, там вводилось поле $N \times M$ из чисел. Я начал тестировать его задачу, введя сначала минимальный тест — $N = M = 1$, на поле одна единичка. Программа зависла.)

При тестировании на таком минимальном тесте зачастую вообще половина программы не будет выполняться — какие-нибудь циклы не выполняться ни разу и т.п. Ничего, в таком случае тем более минимальный тест — это хорошая проверка остальной части программы, а также проверка того, действительно ли эти циклы не должны выполняться в этом случае ни разу.

После этого чуть увеличьте размер теста (возьмите $N = 2$) и постарайтесь протестировать все возможные (!) такие тесты. Например, если у вас задана перестановка, проверьте обе возможные перестановки; если граф без петель — то все возможные графы с двумя вершинами. Как правило, таких тестов не так уж и много, и проверить все не составляет проблем. Если же таких тестов очень много, то постарайтесь проверить хотя бы все возможные категории таких тестов. Например, в задаче сортировки массива обязательно проверьте случай, когда первое число больше второго, когда оно равно второму и когда оно меньше второго. (Это все при $N = 2$!) Таких «категорий» обычно не так уж и много.

Далее переходите к следующему размеру теста и опять-таки постарайтесь проверить все возможные тесты или хотя бы все категории тестов. И так далее — до тех пор, пока при очередном размере тестов количество возможных тестов или категорий не станет уж очень велико (ну, грубо говоря, больше чем 5–7); часто это бывает уже при $N = 3$. В таком случае на этом N проверьте несколько тестов из существенно различных категорий, после чего уже плавно переходите к следующему разделу.

2. Простые маленькие тесты. Далее попробуйте несколько небольших тестов. Как правило, сюда же попадёт тест из условия. Старайтесь эти тесты делать различными, добавляя в них те или иные «фичи», которые могут оказаться важны, но которые вы толком не смогли протестировать раньше из-за того, что тесты были маленькие. Например, это могут быть мосты или циклы в графе, несвязные графы с ненулевым количеством рёбер в каждой компоненте связности; или сложные — сначала возрастающие, потом убывающие, потом опять возрастающие — перестановки типа 3 4 1 2, или массивы, в которые есть одинаковые числа, но они не соседние (2 1 3 2), и т.п.

3. Тест из условия. Отдельно скажу про тест (тесты) из условия. Его, безусловно, надо проверить. Но более того, его надо проверить очень тщательно. Если ваш ответ хотя бы на один символ отличается от приведённого в условии, то очень тщательно убедитесь, что ваш ответ правильный. Конечно, часто бывает так, что на тест из условия может быть несколько правильных ответов, и нередко жюри специально даёт в пример такой ответ, который вряд ли настоящее решение по задаче выведет — чтобы сбить вас с толку и не подсказать правильное решение. Поэтому это нормально, если ваш ответ отличается от ответа в примере. Но тем не менее, может также оказаться и то, что вы просто неправильно поняли задачу или неправильно поняли формат выходных данных, или не заметили фразы типа «выведите лексикографически наименьший ответ» или что-нибудь подобное.

Поэтому если ваш ответ отличается от ответа из примера, обязательно сделайте все следующие три пункта:

- Перечитайте задачу, убедитесь, что вы правильно понимаете условие и форматы входных/выходных данных. Убедитесь, что вы не пропустили ничего в условии, в том числе какие-нибудь примечания после примера и т.п.
- Убедитесь, что вы понимаете, почему ответ из примера верный.
- Убедитесь, что вы понимаете, что ваш ответ тоже верный и соответствует условию задачи.

Конечно, это полезно делать вообще всегда, но если ваш ответ на тест из примера отличается от ответа из условия, то это сделать просто необходимо.

4. Разные тесты. Часто бывает, что в задаче достаточно просто выделяются разные типы, разные варианты тестов. Это могут быть даже не «частные» случаи, про которые сказано ниже — эти типы могут быть явно не выделены ни в условии, ни в решении, но тем не менее надо подумать и понять, не бывает ли так, что решения задачи в разных ситуациях чем-то отличаются, нет ли в задаче каких-то возможностей, которые могут бы задействованы не в каждом тесте, и т.д. Соответственно, надо постараться сделать тесты, которые покроют все эти варианты.

Например, если в задаче есть несколько способов выполнять какое-то действие, и надо такие способы скомбинировать оптимальным образом, то проверьте по отдельности, что ваша программа умеет выполнять оба способа. Пример: задача: дано N резисторов, их можно соединять последовательно и параллельно, надо составить схему с заданным сопротивлением. Проверьте на простых тестах, что ваша программа может использовать и последовательное, и параллельное соединение — подсуньте два теста, в которых надо сделать только одно соединение, и это должно быть а) параллельное; б) последовательное. Ещё пример: игра, в которой есть два типа ходов, надо найти выигрышную стратегию. Проверьте, что ваша программа умеет делать оба типа ходов — просто подсунув тесты, где нужен только один ход — по тесту на каждый тип ходов.

Эти варианты могут быть слегка замаскированы. Например, задача (простая): проверить, может ли слон дойти за один ход с одной клетки доски до другой. Слон умеет ходить в четырёх направлениях — проверьте их все.

Ещё пример: если вы чувствуете, что решения слегка различаются для чётного и нечётного N, то, пусть даже у вас программа одна, и у вас там нет проверки на чётность N — все равно проверьте оба варианта. Например: посчитать количество правильных скобочных последовательностей, содержащих ровно N скобок (не *пар* скобок, а именно скобок). Если N нечётно, то ответ — ноль. В нормальном решении двумерной динамикой (по паре (количество скобок, баланс)) это получится автоматически, и в программе не будет никаких if'ов, проверяющих, чётно ли N. Тем не менее при тестировании не забудьте проверить как чётные N, так и нечётные.

5. «Подлые» тесты. Подумайте, какие в вашей задаче могут быть «подлые» тесты. Например, когда решения нет, или когда решение в каком-то смысле пограничное, или не такое, как во многих тестах и т.п. Если у вас есть в программе особые случаи, то подумайте и о них. Протестируйте все такие тесты, при этом старайтесь на каждый случай придумывать не очень большой пример. (Не обязательно совсем минимальный, но и не надо лишних наворотов.)

6. Пороговые тесты. Часть бывает так, что в вашей задаче есть тесты, где при небольшом изменении входные данных ответ — или по крайней мере логика его получения — меняются сильно. Например, задача «выведите наибольшую степень двойки, которая меньше или равна данному числу N». Ясно, что таким пороговым случаем является случай, когда N само является степенью двойки. Например, на всем отрезке $[32, 63]$ ответ один и тот же, а вот когда N становится равно 64, ответ резко меняется. Или, например, в вашей задаче есть ровно два «пути», и вам нужно выбрать минимальный — тогда пороговой оказывается ситуация, когда оба пути имеют одинаковую длину: тут при вариации входных данных минимальным будет становиться то один путь, то другой.

Обязательно протестируйте такие тесты, причём не только сам пороговый тест, но и ± 1 от него, а то и ± 2. Например, в задаче про степень двойки обязательно протестируйте числа 63, 64, 65, а может быть, ещё и 62 и 66 — чтобы убедиться, что переход на новый ответ или вариант его получения происходит в правильный момент.

7. Крайние случаи. Это фактически частный случай предыдущего пункта, но иногда удобнее думать о крайних случаях отдельно. Это, например, ситуации, когда решение только-только появилось, т.е. чуть изменить входные данные — и решение пропадёт; или когда, например, решение использует все данные элементы, и т.п.

Кроме того, сюда же можно отнести ситуации, когда только один из параметров входных данных является минимальным возможным. Например, если вам дана матрица, то полезно проверить, как ваша программа будет работать на размерах $N \times 1$ и $1 \times M$ (а ещё и с нулевым размером, если это допустимо)[1]; если у вас граф — то как программа будет работать, если вершин несколько, а вот ребро только одно; если граф с петлями, то, конечно, надо проверить разные графы с одной вершиной (хотя это можно считать ещё и «минимальными» тестами).

8. Частные случаи. Если задача имеет какие-то частные случаи, то, конечно, надо их проверить все.

А именно, во-первых, бывает, что в самом условии задачи указаны различные случаи. Например, бывают геометрические задачи, где требуется определить взаимное расположение объектов, типа «если прямые пересекаются, то выведите их точку пересечения; если они параллельны, то выведите то-то; если совпадают, то выведите то-то». Бывает, что и в формате входных данных присутствуют какие-то варианты. Конечно, надо проверить их все.

[1] См. реальный пример: `http://acm.sgu.ru/forum_action.php?id=837&all=1`

Но не менее важно проверить и все варианты, которые есть в вашей программе. Если случай, допустим, $b = 0$ вы рассматриваете особо, т.е. у вас в программе стоит `if b=0`, — конечно, надо этот случай тщательно протестировать. Причём этот случай тоже надо тестировать полноценно *систематически*. В простейшем случае — если b задано во входных данных, и если $b = 0$, то следует простой вывод — достаточно проверить один или несколько таких тестов. Но если в случае $b = 0$ у вас отдельная сложная логика, то, значит, этот случай надо тщательно протестировать — начиная с минимальных тестов (минимальных среди тех, у которых $b = 0$), небольших тестов, пороговых и т.д.

Аналогично, если значение b берётся не из входного файла, а вычисляется сложной логикой, то нужно постараться придумать полноценный набор тестов, начиная с минимальных и т.д., во всех в которых будет получаться $b = 0$, и их все проверить.

Конечно, это относится к любому частному случаю, фактически к любому `if`, который у вас есть в программе. Вы должны как минимум быть уверены, что в процессе вашего тестироваться *каждая строчка программы* исполнилась хотя бы один раз (а лучше — много раз). Более того, хорошо бы покомбинировать возможные пути исполнения, возможные частные случаи. Например, если у вас идёт сначала частный случай $a = 0$, а потом частный случай $b = 0$, и при этом возможны все четыре комбинации пар (a, b) (т.е. следующие комбинации: $(0, 0)$; $(0, \neq 0)$; $(\neq 0, 0)$; $(\neq 0, \neq 0)$), то все четыре и надо протестировать. Конечно, если `if`'ов у вас много и они могут проходить во множестве разных порядков, то число таких комбинаций может быть очень велико и все вы их не протестируете — но протестируйте основные.

9. Максимальные тесты. Почему-то многие люди, даже тщательно проверив всё изложенное выше, пренебрегают тестированием на максимальных тестах. Это неправильно. Протестировать несколько макстестов не так уж и сложно, зато позволяет избежать возможных ошибок.

Бывают задачи, в которых максимальный тест можно ввести вручную — например, если на вход подаётся только одно число. Тогда возьмите и введите этот максимальный тест.

Но в большинстве задач максимальный тест имеет большой размер — значит, вам придётся написать отдельную программу-генератор этого теста. Ничего сложного в этом нет; если в вашей задаче, например, на вход подаётся число N, а потом массив из N чисел, то напишите простую программу, которая выведет во *входной* файл максимально возможное N, а потом сколько надо единиц. Или возрастающую последовательность от 1 до N. Или N случайных чисел — но это немного хуже, почему — я напишу ниже.

Что надо проверять на макстесте? Во-первых, зачастую вы сможете даже проверить точный ответ на задачу. Например, в задаче сортировки массива вы можете подсунуть последовательность от N до 1, и проверить, что в выходной файл ваша основная программа вывела последовательность от 1 до N.

Во-вторых, вы можете по крайней мере проверить разумность ответа. Если вы суммируете в программе положительные числа, то отрицательный ответ должен вас заставить задуматься и пойти искать в вашей программе арифметическое переполнение. И не только отрицательный, но и слишком маленький тоже. Если вы сортируете массив, и подали на вход случайную последовательность чисел, то вывод, начинающийся с кучи нулей, вас тоже должен насторожить. (Ну, конечно, если вы уверены, что у вас не было столько нулей во входных данных.)

В-третьих, вы можете проверить свою программу на переполнение различных массивов. На паскале это делается автоматически, не забудьте только поставить ключ компилятора `{$r+}`. На си это существенно сложнее, но вы можете обнаружить какие-нибудь странные результаты; а если вы очень сильно вылезли за границы массива, то вы получите сообщение об ошибке типа «программа выполнила недопустимую операцию» (под Windows), или просто «Segmentation fault» (segfault) под Linux. Кстати, в ряде случаев эти ошибки (под Linux) можно обнаружить утилитой Valgrind, если она установлена у вас на компьютере, см. отдельный раздел ниже.

И в-четвёртых, вы проверите время работы вашей программы. Во многих задачах это не является проблемой, и вы заранее уверены, что программа уложится в ограничение времени, но так бывает не всегда, и проверить лишний раз не мешает.

Когда вы готовите макстест, надо иметь в виду ещё и вот что. Бывают просто максимальные тесты, которые являются максимальными просто по формальному тексту условия. А бывают тесты, которые максимальны именно для вашего решения — например, на которых ваше решение работает дольше всего, или на которых оно использует наибольшее количество памяти, и т.п.

Так вот, тестировать надо и те тесты, и другие. Например, если входные данные — это массив из N чисел ($N \leqslant 1000$), то все тесты с $N = 1000$ будут формально максимальными. Но при этом может оказаться, что ваша программа медленнее работает, если массив упорядочен, или если все числа в массиве одинаковы. Более того, может оказаться, что при $N = 1000$ ваша программа работает быстро и не требует много памяти, а вот при $N = 999$ — намного медленнее и требует больше памяти потому, что это нечётное число (по тем или иным причинам). Или, например, при $N = 997$ (наибольшее простое число до 1000). Конечно, в таком случае над тестировать и $N = 1000$, и $N = 999$ или 997.

10. Максимальные тесты для вещественных чисел. Отдельный особый случай, который надо отдельно проверять — это максимальные тесты в задачах, в который вы как бы то ни было работаете с

вещественными числами. В таком случае надо обязательно проверить, не падает ли ваше решение по точности. Зачастую в задаче имеет значение *абсолютная* погрешность — либо вы сравниваете вещественные числа с абсолютной ε (например: `if abs(a-b)<eps`), либо в выходных данные требуется вывести число с определенными количеством знаков *после запятой*.

Тогда одно дело, если числа, с которыми вы работаете, имеют значения порядка единицы, и другое дело, если значения могут быть порядка, например, 10 000. Пусть, для определенности, у вас $\varepsilon = 10^{-5}$. Тогда если у вас $a, b \approx 1$, погрешности вычислений, возможно, будут меньше ε, даже если вы используете тип `single`, и все будет работать с этим типом. Но как только значения a и b становятся порядка, например, 10 000, точности `single` перестает хватать. Аналогичная проблема будет, если вам требуется вывести ответ, например, с 5 знаками после запятой — точности `single` будет хватать только пока вы выводите не слишком большие числа.

Другая возможная тут проблема — если ваш язык программирования выводит не определенное количество знаков *после запятой*, а определенное количество знаков *всего* (как, например, `cout` в C++ по умолчанию). Тогда он может выводить числа порядка 1 так, как вам надо, но для чисел порядка 10 000 он будет выводить слишком мало знаков. (Конечно, здесь я говорю о числах порядка 10 000 только для определенности, в реальности значения чисел, при которых возникает проблема, будет зависеть от условия задачи и от ваших типов данных и вашего ε.)

Как же это тестировать? Если вы подсунете просто максимальный тест, то, скорее всего, вы не заметите ошибку. Поэтому подсуньте максимальный тест, в котором одно из чисел слегка уменьшено — и проверьте, что ваша программа отличает это от просто максимального теста.

Пример: задача: посчитать площадь треугольника, координаты вещественные по модулю до 100, ответ надо вывести с 3 знаками после запятой. Максимальный тест (максимальный в смысле величины ответа, конечно), в данном случае — например, $(-100, -100)$, $(-100, 100)$, $(100, -100)$ (может, это и не совсем максимальный — хотя вроде несложно доказать, что тут он максимальный, — но тут нам не надо абсолютно максимальный, достаточно, чтобы ответ был сравним с максимальный по порядку величины). Ответ здесь 20 000, и вряд ли на нем будут какие-нибудь проблемы. Но введите тест $(-100, -100)$, $(-100, 100)$, $(99.9999, -100)$ — и убедитесь, что ваша программа выводит 19 999.99, а не 20 000. Если вы в C++ используете `cout`, то по умолчанию у вас получится именно 20 000, т.к. он округлит ответ до шести значащих цифр. Можете еще добавить девяток в координату и проверить, что в ответе выводятся сколько надо знаков. Если входные данные должны быть целочисленны, то в этой задаче проблем не возникнет, но если бы координаты были бы до 10^6, то эти же проблемы возникли бы и уже при полностью целочисленных данных. Аналогичная проблема будет, если вы проверяете, например, лежат ли три точки на одной прямой — убедитесь, что точки $(-1000000, -1000000)$, $(0, 0)$ и $(1000000, 999999)$ не лежат.

11. Все возможные тесты. Иногда бывает так, что в принципе допустимых тестов по задаче не так уж и много (например, на вход подаётся одно число от 0 до 100) — что вы можете покрыть тестированием если не все возможные тесты, то по крайней мере существенную их часть. Тогда, безусловно, имеет смысл это сделать, пусть даже это и займёт немного больше времени.

Если тестов совсем мало (грубо говоря, не больше 20), то проверьте их все вручную. Если их больше (до 100–200), то проверьте существенную часть, при этом отдельные области «пространства допустимых тестов» постарайтесь покрыть максимально плотно — например, если на вход подаётся одно число от 0 до 100, то проверьте, например, все (!) числа от 0 до 20, все числа от 60 до 70, все числа от 91 до 100, а также ещё с десяток промежуточных вариантов.

Кроме того, если тестов не больше чем примерно 1000–100 000 000 (в зависимости от того, сколько времени ваша программа решает один тест), то можно организовать процесс, подобный стресс-тестированию (см. ниже), но только генерировать не случайные тесты, а последовательно автоматически сгенерировать и проверить все возможные тесты.

Часть III. Дополнительные комментарии по процессу тестирования

1. Тестируйте даже идею! Тестирование стоит начинать еще до того, как вы начали писать программу, особенно когда речь идет не про какой-нибудь стандартный алгоритм. Когда вам кажется, что вы придумали, как решать задачу — не бросайтесь сразу ее писать. Возьмите листок бумаги, ручку, и проверьте, работает ли ваша идея на паре простых тестов (хотя бы на тесте из условия).

Конечно, если задача «отсротируйте массив», то тут проверять нечего — надо брать и писать стандартный алгоритм. Есть еще много задач, когда тестировать идею может быть и бессмысленно.

Но есть и много задач, где идея нетривиальна — и ее надо проверить. Например, если задача решается формулой — проверьте эту формулу вручную на тесте из условия. Если вы придумали решение динамическим программированием — не поленитесь, вручную просчитайте матрицу ДП на каком-нибудь маленьком тесте.

Это не только позволит вам найти возможную ошибку в идее еще до написания программы, но — если тестирование идеи было удачным — обеспечит вам хороший «плацдарм» для последующего тестирования программы, когда вы ее напишите. Намного проще и приятнее тестировать программу, когда один пример

вы уже подробно разобрали и точно понимаете, что и где должно получаться в вашей программе при работе на этом примере.

2. Знайте ответ на тест заранее. Прежде чем запускать программу на некотором тесте, посчитайте ответ на этот тест вручную. Тестировать программу на тесте, на который вы не знаете ответ — наполовину, если не больше, бессмысленно. При этом ответ надо посчитать заранее, потому что если вы уже знаете ответ, выведенный программой, то вы будете пытаться не решить задачу самостоятельно, а объяснить ответ программы — и с высокой вероятностью «объясните», даже если ответ программы неверный.

В частности, это обозначает, что следует избегать тестов, на которые вы не можете вычислить ответ. Если, например, у вас в задаче граф, не следует рисовать какой попало граф с 10-20 вершинами, запускать на нем вашу программу, смотреть на ответ и думать: «Да, похоже на правду...» Лучше нарисуйте граф, на который вы сможете посчитать ответ, или все-таки потратьте время и найдите ответ на задачу на вашем графе ДО запуска программы.

Этот же принцип относится и к максимальным тестам. Может показаться, что на макстест вы не можете найти ответ «руками», но зачастую бывает, что на некоторые тесты определённой структуры вы найти ответ сможете. Например, если вам надо посчитать сумму N введённых чисел, то если вы будете тестировать на случайных числах, то вы не сможете вычислить ответ. А если вы будете тестировать на последовательности натуральных чисел от 1 до N, то сумма их находится легко — это сумма арифметической прогрессии. Аналогично, ответ на какую-нибудь задачу на случайном графе найти сложно, а на полном графе, или на графе, представляющем из себя просто цикл из N вершин, — может оказаться намного проще.

Поэтому я и не советую тестировать на случайных тестах. Тестируйте лучше на тестах с регулярной структурой, на которых вы можете аналитически найти ответ. (На случайных тестах тоже может быть полезно тестировать, но только как дополнение к тестам с регулярной структурой.)

3. Вспомните условие задачи. Когда вы тестируете программу, у вас в голове уже сидит ваш метод решения. В результате, когда вы в уме вычисляете ответ на тест, может оказаться, что вы уже на автомате применяете какую-то идею, какое-то соображение, которое использует и ваша программа, но которое неверно. (Например, вы можете думать, что ответ на тест всегда бывает только строго больше нуля, на это опирается ваша программа, и вы, когда в уме решаете тест, тоже ищете ответ только среди положительных чисел — а на самом деле ответ иногда может быть ровно ноль.)

Поэтому, особенно когда вы тестируете совсем маленькие тесты, полезно на мгновение вспомнить условие задачи, может быть, даже его перечитать, и отвлечься от вашего решения, посмотреть на тест и на ваш ответ «сверху», подумать, нет ли тут чего, что вы не поняли или не учли.

Пример. Задача: в школе есть три класса. В первом классе a учеников, во втором b, в третьем c. В этих классах решили поменять мебель на новую. Сколько надо купить новых парт, если за каждой партой могут сидеть два ученика? Задача довольно простая, но вы можете случайно ошибочно подумать, что надо посчитать общее число учеников $N = a + b + c$, и исходя из него, вычислить ответ (N div 2 или N div $2 + 1$, в зависимости от четности N). Вы будете тестировать вашу программу, но, если вы уже написали такое решение, то скорее всего ответы на ваши тесты в уме вы тоже будете вычислять по *этой же* формуле — и поэтому ошибок вы не найдете. Но вот вы вводите тест «1 1 1», получаете ответ «2», понимаете, что ответ соответствует формуле: и правда, трех учеников можно посадить за две парты, но не меньше... Но отвлекитесь на момент от решения, вспомните условие задачи — можно ли *в три класса* поставить в общей сложности две парты, чтобы в каждом классе за партой сидел один ученик? Нельзя.

4. Разнообразие тестов. Старайтесь вносить разнообразие в ваши тесты — старайтесь, чтобы используемые вами тесты не были слишком однотипны, чтобы в них было как можно меньше всего общего.

Конечно, вы и так внесёте серьёзное разнообразие за счёт варьирования N и рассмотрения различных приведённых выше случаев. Но старайтесь вносить разнообразие и далее. Например, если входные данные — это N и N различных чисел, то не следует тестировать *только* на тестах, на которых эти N чисел являются перестановкой. Например, если вы тестируете варианты с $N = 2$, то не следует удовлетворяться вариантами «1 2» и «2 1». Попробуйте ещё и «100 997» и «-8 -5» (если это допустимо, конечно).

Если у вас задан граф и вершина в нем («начальная вершина»), то не забывайте, что она может иметь номер, отличный от 1. Казалось бы, вашему алгоритму все равно, какая начальная вершина — вы считали её номер и дальше используете эту переменную — но тем не менее не помешает этот номер варьировать.

Если вам заданы N чисел, вводите их в разном порядке: возрастающем, убывающем, и т.п. — даже если вам кажется, что вашему решению порядок входных данных не важен. Если задана матрица, то не забывайте, что она может не быть квадратной, и т.д. Если задан круг, не забывайте, что его центр может быть и не $(0, 0)$. Если задано N, то не забывайте, что оно может быть как чётным, так и нечётным — даже если вроде в задаче это не имеет значения.

И так далее.

Чем это отличается от того, что я писал выше? Выше я призывал вас последовательно и максимально полно рассматривать всевозможные случаи. Рассматривать случай $N = 1$, рассматривать все возможные

тесты с $N = 2$, рассматривать крайние случаи, особые случаи и т.д. Но это все касалось тех ситуаций, когда довольно очевидно, что это важные варианты и что от них реально может что-то зависеть.

В этом же разделе я призываю вас думать также о тех вариантах, от которых, казалось бы, ничего не зависит. Конечно, не надо перебирать абсолютно все возможные такие варианты (не следует перебирать на каждом тесте все возможные порядки нумерации вершин графа, например). Но полезно от теста к тесту двигаясь по изложенному в предыдущих разделах алгоритму, не забывать варьировать те параметры, от которых, казалось бы, ничего не зависит (т.е. стоит в каждом новом тесте как-нибудь по-новому нумеровать вершины графа).

5. Ориентируйтесь не на формальность условия, а на смысл. Бывает так, что ваша программа может работать для тестов, выходящих за рамки формата входных данных. Например, в условии написано $N \geqslant 1$, но вы понимаете, что при $N = 0$ задача тоже имеет смысл и вроде ваша программа должна бы работать — так и протестируйте на $N = 0$. Аналогичная ситуация — если задана строка длины $\geqslant 1$, но ваш алгоритм должен бы работать и на пустой строке — проверьте пустую строку.

6. «Белый ящик» и «чёрный ящик». Есть два принципа тестирования программ — принцип «чёрного ящика» и «белого ящика». Первый подразумевает, что вы не знаете внутреннего устройства программы, и тестируете только исходя из знания задачи. Второй же подразумевает, что вы внутреннее устройство программы знаете.

Так вот, вы должны сочетать оба способа. Я про частные случаи этого правила уже писал выше: при тестировании частных случаев надо тестировать как те, которые очевидно следуют из условия, так и те, которые вам пришлось разобрать в коде. Аналогично, макстесты надо брать и те, которые максимальны с точки зрения условия, так и те, которые максимальны с точки зрения вашей программы.

Но это же относится и ко всему остальному тестированию. Например, крайние и пороговые случаи также бывают как с точки зрения условия, так и с точки зрения вашей программы.

7. Не теряйте тесты. Если вы уже протестировали вашу программу на многих, особенно не очень тривиальных, тестах, то не теряйте эти тесты. Возможно, вы найдёте в программе ошибку, и вам придётся все тестировать заново. Поэтому старайтесь тесты так или иначе сохранять.

Полезный приём — если задача не требует чтения входного файла «до конца файла», то в конце файла может быть произвольный мусор. Например, если формат входного файла «сначала задано число N, а далее идут N чисел», то скорее всего вы напишите программу так, что она будет полностью игнорировать все, что идёт после этих N чисел. Тогда во входном файле вы можете легко держать несколько тестов сразу — программа будет просто подцеплять первый из них. Когда вам надо протестировать на очередном тесте, вы его просто переносите в начало входного файла.

(А если бы формат входного файла был бы просто «входной файл содержит несколько чисел», то вам бы пришлось читать «до конца файла», и такой трюк не прошёл бы — или как минимум был бы существенно сложнее.)

8. Мультитест. Бывают задачи, в которых во входном файле находятся сразу несколько примеров, и ваша программа должна решить их все. Тогда при тестировании вы легко можете проверять сразу несколько примеров, и это весьма удобно. Если же в вашей задаче во входном файле задается только один пример, то можете, для удобства тестирования, все равно организовать мультитесь — например, решать не один пример, а решать, пока не кончится входной файл. Тестировать будет удобнее.

9. Использование команды `assert`. Практически во всех языках программирования есть функция `assert`. Она принимает как минимум один параметр — значение логического типа (`bool` или `boolean`) и делает очень простую вещь: если значение ложно, то она прерывает программу с ненулевым кодов возврата, как правило, выводя на экран дополнительную информацию. В паскале она принимает ещё и второй параметр — строку, которая будет выведена на экран в случае, если выражение ложно; в си этого второго параметра нет, зато команда выведет на экран имя входного файла и строку, на которой была вызвана `assert`.

Зачем нужна эта команда? Часто в программе у вас бывают моменты, когда вы уверены, что некоторое утверждение должно выполняться. Например, если вы сортировали массив, то вы уверены, что при нормальной работе программы массив будет отсортирован. Если вы искали точку пересечения двух прямых, то найденная точка должна принадлежать обеим прямым. Бывает так, что в некоторый момент программы вы уверены, что некоторая переменная больше нуля, или что одна переменная строго меньше другой. Зачастую это важно для дальнейшего хода программы, хотя это и не обязательно.

Соответственно, вы можете добавить в программу проверку такого условия. Это позволит вам быстро детектировать ситуации, когда это условие не выполнилось: например, если вы добавите проверку отсортированности массива, и если у вас сортировка работает плохо, то вы сразу будете знать, что виновата именно сортировка, а не другие части вашего кода. Без проверки вы были бы вынуждены долго отлаживать программу в поисках, откуда идёт неправильный ответ (а ещё вам могло бы повезти и ответ оказался бы правильным, и вы вообще не заметили бы ошибку).

Как добавить эту проверку? Конечно, можно все реализовать самостоятельно:

```
if not условие then begin
    writeln('Ошибка:...');
    halt(2); // завершить программу с ненулевым кодом возврата
end;
```

Но вот для этого и существует команда `assert`. Вы просто пишите `assert(условие);` в нужном месте кода — и эта проверка будет выполнена (точнее см. ниже).

Например: проверка того, что массив отсортировался по неубыванию:

```
for i:=2 to n do
    assert(a[i]>=a[i-1],'Array not sorted!');
```

(На C++ аналогично, только у команды `assert` нет второго аргумента, вы не можете передать собственное сообщение об ошибке.)

Проверка, что точка пересечения прямых найдена верно:

```
найти точку пересечения
assert(точка принадлежит первой прямой);
assert(точка принадлежит второй прямой);
```

Частный случай: проверка, что в это место кода программа не должна никогда попасть, делается командой `assert(false)`. Типичный пример такого использования — в ветке `else` в конце цепочки `if`'ов, примерно так:

```
if dir='north' then...
else if dir='south' then...
else if dir='east' then...
else if dir='west' then...
else assert(false, 'Unknown direction ' + dir);
```

Обратите внимание, что в аргументе команды `assert` пишется условие, которое должно быть выполнено при правильной работе, а не его отрицание. Фактически команду `assert` можно понимать как «проверь, что...». Ошибка будет продиагностирована, если условие *не* выполнено.

Используйте команду `assert` в своих программах. Везде в тот момент, когда вы думаете «а вот тут должно выполняться такое-то условие», добавьте `assert`. Это несложно, зато позволит вам обнаруживать ошибки прямо там, где они зарождаются. Если у вас есть выделенные части кода, результат которых легко проверить (сортировка, пересечение прямых и т.п.) — добавьте `assert` в конец этих частей, чтобы их и проверять.

Правда, при использовании `assert`'ов надо иметь в виду следующее. В зависимости от настроек, компилятор иногда может полностью игнорировать их — для ускорения производительности. Во Free Pascal `assert`'ы включаются ключом компилятора `{$ASSERTIONS ON}`. Как они включаются/отключаются в C++, я наизусть не знаю. Поэтому перед использованием `assert`'ов убедитесь, что они работают: добавьте простую команду типа `assert(false)` и проверьте, что программа падает. Если нет — посмотрите, что надо сделать, чтобы включить `assert`'ы. Если никак не получается их включить — ну напишите самостоятельно функцию `myassert`, которая будет делать то же самое.

Отдельный вопрос — что делать с `assert`'ами при сдаче программы на проверку. Конечно, в каждом случае надо отдельно думать, но вообще рецепт довольно простой: если вы сможете получить результат проверки и у вас будет возможность что-то исправить, то не отключайте `assert` — в протоколе проверки вы увидите «ошибку времени выполнения» или подобный вердикт, и сможете заподозрить `assert`; но вот если результат проверки вам не будет доступен, то лучше их отключить — вдруг повезёт?..

Часть IV. Пример: задача сортировки массива

Я без особенных комментариев приведу здесь, какие тесты и в каком порядке я бы вводил в программу сортировки массива. Будем считать, что ограничения — длина массива до 1000, элементы по модулю не превосходят 10 000.

Конечно, ни на какой задаче невозможно полностью проиллюстрировать все приведённые выше принципы, и ни в какой задаче вы не сможете им абсолютно строго следовать, поэтому относитесь к этому примеру именно как к иллюстрации, а не как к идеальному тестированию.

Итак, я бы делал следующие тесты (каждая строка ниже — это отдельный тест).

```
1
10
1 2
2 1
3 3
```

```
-4 3
3 -4
1 2 3
9 8 7
-1 6 -1
5 3 4
7 7 1
4 7 4 7
5 4 4 4
10000 10000 10000 -10000 -10000 -10000
1 2 3 4 5 6
6 5 4 3 2 1
4 4 4 4 4 4
4 4 4 4 3 4
7 7 7 7 7 7 1
1 2 3 4 ... 1000
1000 999 ... 1
10000 9980 ... -9980 // 1000 чисел; надеюсь, я не ошибся с расчётом
137 137 ... 137 // 1000 одинаковых чисел
10000 10000 ... 10000 // 1000 одинаковых чисел
```

Часть V. Стресс-тестирование

1. Общие принципы. Особый вид тестирования — это так называемое стресс-тестирование. В широком смысле — это любое тестирование на очень большом, массовом вводе; но в узком смысле олимпиадного программирование — это тестирование, организованное следующим образом.

Вы пишите по задаче два решения — одно основное, которое вы собственно и хотите протестировать; второе «тупое», которое работает медленно, но максимально надёжно. В «тупом» решении важно действительно все писать максимально тупо, стараясь использовать как можно меньше предположений и утверждений, которые вы можете предполагать по задаче; часто говорят, что «тупое» решение — это просто перевод условия задачи на язык программирования. Часто в качестве «тупого» решения подходит рекурсивный перебор. Старайтесь не заимствовать никакой код из «тупого» решения в основное и наоборот, чтобы не переносить баги Очень обидно будет, если и в тупом, и в основном решении будет один и тот же баг.

Кроме того, вы пишите генератор случайных небольших (чтобы «тупое» решение работало) тестов и программу («чекер»), сравнивающую ответы двух ваших решений, и все это в цикле запускаете: генератор—два решения—чекер—генератор—и т.д. Даёте этой связке поработать некоторое время — либо пока не найдётся тест, на котором два решения дадут разные результаты, либо пока не пройдёт какое-то время, будет проверено много тестов, и вы не решите, что хватит. Зачастую стресс-тестирование, особенно если оно работает больше минуты без ошибок, можно запускать в фоновом режиме, пока вы работаете над другими задачами.

Так за небольшое время вы можете проверить тысячи тестов. Если в вашем решении есть ошибки, которые встречаются сравнительно часто, то, скорее всего, вы их найдёте. (Правда, конечно, надо всегда помнить, что стресс-тестирование — не замена полноценному последовательному тестированию и может найти далеко не все ошибки.)

2. Организация стресс-тестирования. Как организовать такой запуск в цикле? Можно, конечно, написать скрипт (.bat/.cmd или .sh-файл), но можно и проще. Все делаете в одной программе. Пишите четыре процедуры (функции). Процедура-генератор генерирует очередной тест и (важно!) сохраняет его во входной файл. Две процедуры решения (основное и «тупое») каждая читает входной файл (!), решает задачу и выводит ответ: основное решение — в основной выходной файл, «тупое» — в какой-нибудь ещё файл. И процедура-чекер считывает оба файла и сравнивает результаты.

И в основной программе вы просто в цикле запускаете эти процедуры.

Важно то, что все общение между процедурами ведётся через файлы. (Например, можно было бы сделать так, чтобы генератор сразу заполнял массив в памяти, который потом использует решение, или чтобы чекер ответы брал прямо из соответствующих переменных — но так делать не надо.) Это позволяет вам, во-первых, протестировать не только собственно алгоритм, но и ввод-вывод. А во-вторых, это позволяет вам потом очень легко переделать программу на окончательное решение, которое вы будете сдавать — вы просто в основной программе цикл с вызовами кучи процедур замените на вызов процедуры основного решения; в процедуре основного решения ничего исправлять не надо будет!

Есть ещё один момент, который надо иметь в виду при стресс-тестировании. Если таки найдётся тест, на котором ваши программы дали разные результаты, то вам нужно суметь этот тест не потерять. Поэтому в любой реализации стресс-тестирования надо сделать так, чтобы, как только чекер определил,

что результаты различаются, он тут же прерывал бы тестирование. Во второй реализации (все в одной программе) это просто: чекер будет просто тут же завершать программу. В первой реализации (через скрипты) чекер будет должен возвращать ненулевой код возврата, а в скрипте вы должны будете это проверять. Главное — сделать все так, чтобы когда стресс-тестирование остановлено, у вас во входном файле как раз оказался бы найденный тест, а в выходных файлах — результаты работы обоих решений. Кстати, это ещё одна причина, почему генератор и чекер должны общаться с решениями через файлы, даже если вы реализуете все в одной программе — так вам намного проще посмотреть тест, на котором решения разошлись.

Важный момент еще. В генераторе вы можете делать `randomize`, т.е. инициализировать датчик случайных чисел текущим временем и т.п. Тогда при каждом новом запуске стресс-тестирования у вас будут получаться новые тесты. Но это же может оказаться и недостатком: вы теряете повторяемость результатов, которая иногда может быть полезна. Поэтому может иметь смысл ставить некоторый фиксированный `seed` (записывая `const SEED=89624; ... randseed:=SEED`, а значение для `SEED` уже выбирая вручную случайно) — тогда при каждом новом запуске стресс-тестирования тесты будут те же. Это далеко не всегда полезно (зачем вам опять тестировать те же тесты?), но иногда может быть полезно.

Близкая мысль — запоминать `seed`, используемый для генерации *каждого очередного теста*. Это делать можно примерно так:

```
procedure generate;
var seed:integer;
begin
seed:=random(2000000000);
writeln(seed);
randseed:=seed;
...
```

Теперь вы знаете, какой `seed` у вас использовался перед каждым новым тестом и, если захотите повторить какой-то конкретный тест, то вместо `random(2000000000)` просто вписываете это значение `seed`.

И еще один важный момент. Если ва нашли контр-пример, то не забудьте его отдельно сохранить — чтобы потом, когда вы найдете ошибку и исправите программу, проверить ее на этом примере — а исправили ли вы то, что хотели?

3. Стратегия стресс-тестирования. Стресс-тестирование — это, конечно, не самый быстрый процесс. В отличие от простого тестирования, описанного в предыдущих пунктах, подготовка и запуск стресс-тестирования занимает немало времени. Поэтому я считаю, что стресс-тестирование надо рассматривать как дополнительную возможность, которую использовать стоит не всегда. Если у вас есть возможность проверить свою программу на тестах жюри во время тура (система с токенами, или командная олимпиада), и вы уже тщательно её оттестировали вручную, то, возможно, стоит и отправить её на проверку — возможно, стресс и не понадобится. Если задача на тестах жюри не работает, и вы не можете найти ошибку — тогда и стоит написать стресс.

Если же отправить решение на тестирование во время тура возможности нет, то следует здраво оценить ситуацию, и решить, что стоит делать: писать стресс по этой задаче или работать над другими задачами. Здесь, конечно, универсального рецепта нет, все зависит от многих параметров, в том числе от того, насколько вы уверены в своём решении по этой задаче, насколько просто написать генератор—тупое решение—чекер по ней, и как вы оцениваете свои шансы успешно написать другую задачу (задачи) за оставшееся от тура время. Но, если таких «других задач» уже не осталось, т.е. если вы уже написали все задачи тура, а время ещё есть — то вот тут самое правильное — это как раз написать стресс-тестирование, сначала по одной задаче, потом — и по другим. Аналогично, если другие задания очень сложные, и вы считаете, что вы уже сделали по ним что могли (написали заглушки, перебор или какие-нибудь подзадачи), то стоит написать стресс-тестирование по простым задачам (да и по вашим частным решения сложных задач тоже).

В общем, если стратегия применения обычного тестирования достаточно проста, то над тем, когда и как применять стресс на реальных олимпиадах, надо думать в зависимости от ситуации.

И ещё раз подчеркну, что стресс-тестирование не способно найти все ошибки. Поэтому его следует рассматривать только как дополнение к основному тестированию, но не как его замену. Кроме того, это обозначает, что, если стресс-тестирование в течение некоторого времени не нашло ни одной ошибки, то может иметь смысл поменять параметры генератора — например, генерировать тесты с бóльшим N, или немного другой структуры и т.п.

4. Что должен из себя представлять чекер? Самый простой вариант — он должен просто сравнивать два решения, найденные двумя программами, и прерывать работу стресс-тестирования, если они различаются. Но это далеко не всегда возможно; простейший пример — если в задаче требуется найти оптимальное решение, и таких решений может быть несколько, то две ваших программы легко могут найти два разных решения и обе оказаться правы. В таком случае, конечно, идеальный вариант — чекер должен

проверять *корректность* решения основной программы, используя вывод тупого решения как подсказку. Например, чекер проверит, что стоимости обоих решений совпадают, и проверит, что решение основной программы корректно и имеет нужную стоимость. (Собственно, примерно так и работают чекеры, подготовленные жюри для финального тестирования, только вместо вывода тупого решения они используют вывод решения жюри.)

Но такой чекер написать бывает сложно. Тогда можно поступать и по-другому — написав чекер, который не до конца проверяет корректность ответа. Например, можно проверять только стоимость, не проверяя самого решения. Конечно, стресс-тестирование с таким чекером найдёт меньше ошибок, но все равно может быть полезным.

5. Стресс-тестирование без тупого решения и/или без чекера. В пределе такой подход приводит к тому, что в определённых случаях можно проводить стресс-тестирование и без тупого решения вообще. Например, вы можете просто проверять правильность ответа (например, что выведенный путь существует в графе), и что он соответствует той стоимости, которую вывело ваше решение, но не проверять, что этот ответ оптимальный. Или, например, если надо найти точку пересечения двух отрезков, вы можете проверять, что выведенная точка действительно принадлежит обоим отрезкам — тогда вы не сможете отловить ситуации, когда ваше решение выводит «отрезки не пересекаются», хотя они на самом деле пересекаются, но остальные ошибки отловить сможете.

Важный случай такого стресс-тестирования без тупого решения — это если надо не проверять корректность ответа, а проверять, завершается ли вообще ваше решение корректно. Например, укладывается ли оно в ограничение времени, не падает ли оно с ошибкой, или не нарушает ли оно формат выходных данных на определённых тестах. В первых двух случаях вам даже не нужен и чекер, в последнем случае чекер будет просто проверять формат выходных данных. Этот подход особенно полезен, если вы уже сдавали решение на проверку, и получили вердикт, что оно прошло все тесты, кроме некоторых, на которых, упало с ошибкой, или нарушило формат выходных данных, или превысило предел времени (правда, в последнем случае, конечно, надо сначала постараться просто потестировать на больших тестах).

При этом, если вы проверяете, укладывается ли ваше решение в ограничение времени, вы, конечно, можете встроить в код стресс-тестирования проверку на время работы, но можете — для получения грубых результатов — просто после завершения каждого теста выводить что-нибудь на экран, глазом смотреть, сколько времени занимает каждый тест, и просто прервать программу, если прошло несколько секунд, а тест ещё не решён. Кроме того, конечно, в этом случае вам надо, чтобы генератор создавал не маленькие тесты, а близкие к максимальным.

6. Стресс-тестирование с заранее известным ответом. Это — идея немного другая, чем в предыдущем параграфе. Вы можете написать генератор так, что он заранее будет знать ответ на каждый сгенерированный тест. Например, если задача — сортировка массива, то генератор может генерировать более-менее случайный отсортированный массив, а потом его перемешивать. Тогда вам не нужно тупое решение, а чекер становится довольно простым.

Часть VI. Отладка программы

1. Что делать, если вы нашли тест, на котором ваша программа не работает? Если вы наконец нашли тест, на котором ваша программа не работает, то многие тут же бросятся отлаживать свою программу на этом тесте, пытаться понять, почему она на нем не работает. Это — в корне неправильный подход.

Первое, что надо сделать, если вы нашли такой тест — это попробовать его уменьшить! Надо попробовать получить меньший тест, на котором программа все равно не будет работать. Чем меньше тест, тем проще на нем отлаживать программу, тем проще проверять корректность отдельных кусков программы, тем проще искать ошибку. Поэтому первым делом постарайтесь действительно максимально уменьшить этот тест.

Конечно, если у вас уже тест самый маленький, какой только возможно, то уменьшать некуда — тогда да, начинайте отладку. Если у вас тест, например, с $N = 2$, то скорее всего уменьшать тоже уже некуда — если вы следовали приведённому выше алгоритму, то все тесты с $N = 1$ вы уже проверили, и на них программа работает — поэтому отлаживайте программу на вашем тесте с $N = 2$.

Но если ваш тест пусть чуть-чуть больше, пусть даже $N = 4 \dots 5$, то скорее всего вы не проверяли все возможные тесты с меньшим N. Поэтому имеет смысл потестировать подробнее на тестах с $N = 3 \dots 4$, поискать там тест, на котором программа не будет работать.

Тем более это имеет смысл, если программа не работает на ещё бóльших тестах — т.е. если вы уже перешли к тем разделам тестирования, где вы берете тесты с не самыми минимальными N. В частности — и это важно — если вы нашли тест, на котором программа не работает, при помощи стресс-тестирования, то тоже сначала попробуйте его вручную уменьшить.

Как уменьшать тест, как искать тест поменьше, на котором программа тоже не работает? Есть два основных подхода. Во-первых, можно внимательно посмотреть на тест и попытаться понять, что же в нем такого особенного, почему тест не работает. Например, может быть, в графе есть мост, а раньше вы

тесты с мостами не тестировали? Может быть, вы заметите какую-нибудь подобную особенность — тогда попробуйте придумать тест поменьше с такой же особенностью, проверьте, работает ли ваша программа на нем.

Во-вторых, просто попробуйте поудалять элементы из теста. Если на вход подаётся массив — попробуйте по очереди поудалять каждый его элемент. Если граф — поудаляйте вершины и ребра. Если тест большой, то имеет смысл начать с удаления вообще половины теста. Останавливайтесь, когда убедитесь, что не получается никакими подобными ухищрениями уменьшить размер теста так, чтобы программа все ещё не работала бы на нем.

Особый случай — если у вас программа выдаёт неправильный ответ на макстесте. Там, прежде чем кидаться уменьшать тест, надо подумать. Есть ряд типичных причин, почему на больших тестах ваша программа не работает, в первую очередь — переполнения массивов и переполнения переменных. Проверьте сначала их. Если же ошибки не находите, то тогда уменьшайте тест. Возьмите тот же генератор, только уменьшите N в 10 раз. Если на полученном тесте программа все равно не работает, то уменьшите ещё в 10 раз; иначе, наоборот, возьмите N равное $1/3$ от максимального. И так далее, попробуйте делением пополам установить, начиная с какого N программа перестаёт работать, и поисследуйте внимательнее поведение программы на этом пороге.

Зачем все это надо? Ну, во-первых, как уже было сказано выше, чем меньше тест, тем проще на нем вести отладку программы. Но есть и ещё одна важная причина. В процессе уменьшения теста вы вполне можете обнаружить какую-нибудь закономерность, которой подчиняются тесты, на которых ваша программа не работает. Вы можете найти какую-нибудь особенность теста (те же мосты в графе), которая, видимо, связана с тем, что программа на этом тесте не работает. После этого, возможно, вам даже не понадобится отладка — достаточно просто будет слегка подумать, и вы, возможно, поймёте, где в программе вы могли допустить ошибку, которая приводила бы именно к наблюдаемому поведению. Вообще, после того, как вы так поисследовали программу, и поняли, на каких тестах она работает, на каких нет, бывает полезно на несколько минут отвлечься от компьютера, если есть возможность, то пройтись куда-нибудь, думая над тем, какая ошибка в коде может дать такой эффект. Но, конечно, это зависит от ситуации — часто проще бывает и пойти в отладку, про что я напишу чуть ниже.

Отмечу ещё, что подобная минимизация тестов особенно важна в серьёзном программном обеспечении. Если вы пользуетесь какой-нибудь программой и обнаружили, что, по вашему мнению, эта программа в каких-нибудь случаях работает некорректно (браузер неправильно отображает страницу, компилятор неправильно компилирует вашу программу, и т.п.); да даже если вы думаете, что это не ошибка в программе, а просто вы чего-то не понимаете — и вы хотите обратиться к авторам программы или на какой-нибудь форум, посвящённый этой программе — то первое, что надо сделать — это максимально минимизировать пример, на котором у вас что-то не работает.

Бессмысленно авторам компилятора писать, что ваша программа компилируется неверно и в качестве примера прикладывать программу на 1000 строчек (если только все 1000 не важны для примера, но так бывает крайне редко); аналогично бессмысленно спрашивать на каких-нибудь форумах, почему эта программа компилируется не так, как вы ожидаете. Сначала попробуйте уменьшить программу до минимального примера, который демонстрирует проблему — зачастую это всего 10–20 строчек, иногда до 50-100. Уберите из программы все, что не нужно для демонстрации проблемы — и только после этого обращайтесь за помощью. Полученная программа часто называется Minimal Working Example, или Short Self-Contained Correct Example — и является одним из основных правил хорошего тона при обращении за помощью. Конечно, это относится не только к программам, но и ко многим другим проблемам, которые вы можете обнаружить на компьютере.

2. Про тест из условия. Выше я уже писал, что не советую начинать тестирование сразу с теста из условия. Основная причина этого — именно в том, что тест из условия, как правило, не является самым маленьким. Если ваша программа не работает на тесте из условия, то, скорее всего, есть ещё много тестов, которые меньше, и на которых она тоже не работает. Поэтому логичнее начать именно с этих тестов.

Вообще, именно поэтому логично брать тесты в «возрастающем» порядке, начиная с самых маленьких тестов, и заканчивая самыми большими. Тогда все простые ошибки вы найдёте на самых маленьких тестах, и вам будет с ними проще работать; а если вы найдёте ошибку на очередном не самом маленьком тесте, то скорее всего вы уже будете знать, что на многих меньших тестах программа работает, а значит, вам не придётся сильно уменьшать тест, да и вообще, возможно, ошибка будет связана с какими-то особенностями этого конкретного теста — что может вам дать дополнительную подсказку.

3. Как отлаживать программу, когда тест уже известен. Итак, вы нашли тест, на котором ваша программа не работает, уменьшили его, насколько это возможно, и даже слегка подумали на тему «почему именно на этом тесте программа не работает, хотя она работает на других, чем таким этот тест отличается от остальных?» Теперь можно приступать к исследованию, что же именно ваша программа делает неправильно.

Только всегда помните, что такое исследование можно проводить разными способами. Во-первых, всегда есть пошаговое исполнение программы, оно же отладка или debugging (хотя два последних слова могу

употребляться и в широком смысле поиска ошибки). Вы можете заставить вашу среду программирования выполнять операции пошагово, останавливаясь после каждой строчки и показывая содержимое всех переменных. Этот способ бывает полезен, особенно, если тест действительно очень маленький, или если вы уже другими способами более-менее локализовали ошибку, и если «пристальное всматривание» в код (см. ниже) не помогает. Но тем не менее зачастую пошаговое выполнение — далеко не самый эффективный способ поиска ошибок.

Второй способ — «пристальное всматривание» в код программы. Если тест совсем маленький, то очень легко глазом пройтись по коду, представляя в уме, как он будет выполняться — часто вы так сможете найти ошибку. Аналогично, если другими способами вы локализовали ошибку в пределах нескольких строчек — вместо того, чтобы бросаться в пошаговую отладку, лучше внимательно посмотрите на эти строчки и представьте себе в уме, как они выполняются.

Полезно, кстати, еще чуть-чуть отвлечься от кода и подумать, а нет ли тут какой-нибудь тупой ошибки, которая могла бы повлиять? Например, не перепутали ли вы n и m, или не забыли ли вы где-нибудь написать $+1$ и т.п.

Но третий, важный, способ — это так называемый отладочный вывод, debug output. В ключевые места своей программы добавьте вывод на экран ключевых переменных, запустите программу, и посмотрите, начиная с какого места выводимые значения стали неправильными. Вы поймёте, с чего начались ошибки: какая переменная и в каком участке кода стала вычисляться неверно. Если вы сразу не найдёте ошибку, то добавьте более подробный вывод в этот участок кода, особенно в места, где вычисляется эта переменная. И так далее — вы сможете локализовать ошибку до 3–5 строчек, после чего уже легко найдёте ошибку.

Особенно это полезно, если у вас в программе циклы или сложная рекурсия. При пошаговом исполнении вам, как правило, довольно тяжело попасть на определённую итерацию цикла или на определённый уровень рекурсии: надо ставить сложные breakpoint'ы, или долго пошагово идти до этого места, — а при отладочном выводе вы вставляете вывод на каждой итерации (или только на нужных), запускаете программу и сразу все что надо видите.

Отладочный вывод бывает полезно добавлять на начало или конец итераций циклов, на вход или выход функций и т.п.; в дальнейшем будете добавлять ещё вывод, уже понимая, что именно надо ловить.

Отдельно отмечу про полезность отладочного вывода в задачах на динамическое программирование. В этих задачах обычно вы вычисляете какой-нибудь массив или таблицу значений, причём каждое очередное значение вычисляется по довольно простым формулам на основе ранее посчитанных значений. В таком случае, если вы нашли тест, на котором ваша программа не работает, очень полезно сделать следующее. Во-первых, на бумажке возьмите и вычислите эту таблицу сами, не опираясь на ваш код, а опираясь на ваши формулы, а также на понимание того, что каждое значение обозначает. (Вы ведь уже уменьшили тест, поэтому скорее всего таблица будет не такой уж и большой.) Проверьте, что ответ получается верный; если нет — то проблема не в вашей программе, а в вашей динамике вообще.

Если же «на бумажке» у вас все получилось, то добавьте в свою программу в конце вывод этой же таблицы на экран. Сравните с тем, что у вас на бумажке, посмотрите, в каком порядке у вас вычисляются значения, и найдите первое (по времени вычисления) значение, которое на бумажке отличается от того, что в программе. Раз это значение первое такое, то те величины, исходя из которых оно вычисляется, посчитаны в программе правильно — значит, ошибка именно в коде вычисления этого значения. Вот и разбирайтесь уже с этим кодом, это обычно уже просто.

Аналогичный подход полезен не только в задачах на ДП, но и в других задачах, где вы также вычисляете некоторый набор значений одно через другое.

Еще важная вещь — всегда помните, что ошибка может быть не только в программе, но и в тесте или ответе на него, в тупом решении, чекере (если вы используете стресс-тест), вашем понимании условия, и даже в компиляторе (но, конечно, последнее крайне маловероятно).

4. Что делать, когда вы уже нашли ошибку в коде. Прежде, чем её исправлять, надо подумать, действительно ли вы уверены в том исправлении, которое хотите реализовать, действительно ли оно верное и надёжное.

Во-первых, может оказаться так, что аналогичная ошибка есть ещё и в другом месте кода (особенно, если вы размножали код методом Ctrl-C/Ctrl-V). Не забудьте исправить все такие места.

Во-вторых, может оказаться так, что вы нашли лишь частный случай большой ошибки. Например, вы можете обнаружить, что какой-то код требует особого учёта случая $x = 2$ — не спешите добавлять проверку `if x=2` в код. Вдруг на самом деле он не работает при любых x, являющихся степенями двойки — просто кроме значения $x = 2$, никакие другие степени двойки в ваших тестах не попадались? Поэтому важно понять, почему ваш код ошибочен, в каких случаях, и только после этого аккуратно исправлять.

В-третьих, исправление — это не обязательно добавление новой проверки, нового `if`. Всегда старайтесь все сделать как можно проще — может быть, этот случай можно учесть, изменив начальное значение какой-нибудь переменной? Сделав так, чтобы какой-нибудь цикл начинался не с 1, а с 0? Добавив фиктивный столбец или строку в матрицу? И т.д.

Наконец, вот вы внесли исправление и убедились, что программа на этом тесте стала правильно работать. Теперь фактически программу надо тестировать заново — ведь легко может оказаться, что на каких-то других тестах программа перестала работать. Конечно, это зависит от ситуации, при некоторых исправлениях можно уже и не прогонять полный набор тестов, но все равно исправленный кода надо протестировать *последовательно* и *систематически*.

Часть VII. Дополнительные замечания

1. Думайте во время написания кода. Процесс написания кода — фактически, это единственное время, когда вы последовательно и очень подробно просматриваете всю вашу программу. Поэтому параллельно с написанием кода всегда думайте, не допускаете ли вы тут каких-нибудь ошибок, нет ли каких-нибудь ситуаций, когда тот код, который вы пишите, будет работать неверно.

Если вы пишите оператор присваивания, подумайте, всегда ли выражение в правой части есть именно то, что вы хотите присвоить этой переменной. Подумайте, нет ли ситуаций, когда присваивание вообще не нужно, или когда надо присвоить другое значение. Если вы пишите `if`, подумайте, правильно ли написано условие, покрывает ли оно все нужные случаи и только их, нет ли случаев, которые надо добавить, или, наоборот, исключить. Если вы пишите цикл, то, аналогично, подумайте, всегда ли он работает корректно. Нет ли каких-нибудь крайних случаев, нет ли с ним каких-нибудь проблем.

И так далее, над каждой строчкой, над каждой командой думайте, нельзя ли её как-нибудь обмануть, нет ли таких тестов, когда это будет работать неверно.

Заодно, кстати, во время написания кода продумывайте последующее тестирование. Если вы поняли, что есть какой-то особый случай — не важно, добавили вы `if` или как-то по-другому учли его — запомните его или запишите, чтобы впоследствии протестировать.

2. Упрощайте код. Старайтесь писать код как можно проще. Не усложняйте. Не вводите лишних `if`'ов, если можно обойтись без них, просто изменив начальное значение какой-нибудь переменной или добавив фиктивный элемент массива. Не пишите два похожих цикла, если можно обойтись одним и это существенно сократит программу — но и наоборот, не мешайте в одном цикле совершенно разные действия, например, ввод данных и вычисления. В последнем случае лучше напишите два отдельных цикла: сначала все считайте, потом только начинайте решать задачу.

Старайтесь разделять разные по смыслу действия в разных частях программы. Например, если вам надо выполнить нетривиальную операцию со строкой (например, удалить повторяющиеся пробелы и добавить недостающие пробелы после знаков препинания), а потом ещё и вывести её хитро — например, разбив на куски не длиннее 80 символов каждый — то не надо все это пихать в один сложный цикл. Лучше сделайте два цикла, первый из которых сначала отформатирует строку как надо, а второй её как надо выведет. (А в этом примере — лучше и вообще три цикла: первый удалит повторяющиеся пробелы, второй добавит недостающие пробелы, третий выведет как надо.) Старайтесь, чтобы в каждом конкретном цикле, каждой конкретной процедуре у вас было как можно меньше параметров, о которых надо думать. Например, в том же примере с форматированием строки, если вы делаете все в одном цикле, то там вы должны думать и о том, какой по счету пробел вы сейчас видите, и о том, был ли предыдущий символ знаком препинания, и о том, сколько символов вы уже вывели на текущую строку. А если вы разобьёте это на три цикла, то в каждом цикле будет важен только один из этих параметров.

Выносите отдельные по логике части в отдельные процедуры, пусть даже эта процедура будет вызываться только в одном месте кода. Например, в рекурсивном переборе выносите обработку найденного решения в отдельную процедуру, чтобы не загромождать основную процедуру перебора.

Отдельная тема — повторяющийся код. Старайтесь не допускать дублирования кода, когда в разных местах программы у вас написано одно и то же — сделайте лучше процедуру. Если у вас в разных местах программы похожий, но не идентичный код, подумайте, нельзя ли его убрать — например, использовав процедуру с параметрами, или просто завернув повторяющийся код в цикл. Повторяющийся код — один из очевидных признаков плохого стиля программы.

Но при этом помните, что простота кода не тождественна его краткости. В большинстве случаев, укорачивая код, вы делаете его проще, но это не следует доводить до предела, не надо укорачивать код в ущерб его понятности. Например, как уже говорилось выше, не бойтесь выделить логически отделённый кусок кода в отдельную процедуру, пусть код и станет длиннее.

Еще близкая тема — когда пишите код, помните, что его вы, скорее всего, будете тестировать и отлаживать. Упрощайте код и с точки зрения последующей отладки.

3. Не исправляйте код, если не нашли тест. Если вы в процессе исследования вашей программы вдруг осознали, что в каком-то месте программы есть, похоже, ошибка, то не бегите её сразу исправлять. Сначала попробуйте найти тест, на котором ваша программа будет некорректно работать из-за этой ошибки. Зачастую это несложно; если же это сложно, то, может быть, это и не ошибка никакая?

Поэтому подберите тест, который «эксплуатирует» эту ошибку, убедитесь, что на этом тесте программа не работает, после чего исправьте ошибку и убедитесь, что программа стала работать корректно.

Последний шаг очень важен. Собственно, это — основная причина, зачем вы искали тест до исправления ошибки. Если вы не нашли теста, то вы не сможете проверить, что правильно исправили ошибку, поэтому искать тест до исправления ошибки — это очень полезно.

4. Проверяйте типа и границы массивов. Не поленитесь и отдельным действием (!), даже если программа вроде работает и вы не можете найти контр-тест, подумайте, не может у вас что-нибудь переполниться — или какая-нибудь целочисленная переменная, или какой-нибудь массив. Вообще, это лучше делать еще в тот момент, когда вы начали писать программу (пишете `var a:integer;` — подумайте, а не может ли переполниться `integer`?). Если есть подозрения, что может — замените на `int64` или подумайте, что еще сделать.

Правда, еще лучше, если вы нашли у себя во «вроде работающей» программе такую подозрительную переменную или массив — попробуйте сначала придумать контр-тест (см. предыдущий параграф). Но даже если не придумаете — все равно поправьте тип переменной или размер массива.

5. Перечитывайте задачу. Если вы сдавали программу на проверку, она не прошла тесты, и вы никак не можете найти ошибку — перечитайте условие задачи. Возможно, вы её просто неправильно поняли. Подумайте, нет ли в условии неоднозначностей, мест, которые можно понять двояко.

Если вы не можете сдавать задачу на проверку во время тура, то тем более прочитайте условие после написания задачи — вдруг вы обнаружите что-нибудь новое, что вы не учли.

6. Перечитывайте программы. Если вы не можете сдавать задачу на проверку во время тура, то перочитайте в конце тура ваше решение, особенно если есть время. Последовательно, строчка за строчкой, думая, что этот код делает и почему. Как и при написании кода (как я советовал выше), думайте, нет ли какого-нибудь варианта, когда этот код может не сработать. Внимательно прочитайте код — и, возможно, вы найдете-таки случай, который у вас не учтен.

Если во время тура вы можете сдавать задачу на проверку, то такое перечитывание все равно может быть полезно. Либо перед отправкой на проверку, либо если вы никак не можете найти ошибку.

Часть VIII. Переполнение массивов в C++: Valgrind и AddressSanitizer

Этот раздел немного выпадает из основной темы, но тем не менее, думаю, он будет тут полезен.

Итак, пусть у вас есть программа на C/C++, и вы подозреваете в ней переполнение массива, но не можете понять, так ли оно, и если да, то где это происходит. Детектировать эти проблемы призваны два средства — Valgrind и AddressSanitizer. Оба могут быть недоступны на вашем компьютере (подробнее см. ниже), поэтому на олимпиадах это далеко не универсальный рецепт. Если же вы можете управлять тем, что установлено на вашем компьютере (в первую очередь если это ваш личный компьютер), то очень полезно установить Valgrind или нужную версию gcc для AddressSanitizer.

Итак, по порядку. Давайте напишем простую программу для примера:

```
#include <iostream>
#include <vector>
using namespace std;

vector<int>* foo(int n) {
  vector<int> a;
  a.resize(4);
  a[0] = n;
  for (int i=1; i<=n; i++)
     a[i] = a[i-1] + i;
  return &a;
}

int main() {
  int n;
  cin >> n;
  vector<int> *b = foo(n);
  cout << (*b)[n] << endl;
  return 0;
}
```

Здесь две ошибки: во-первых, при $n \geqslant 4$ будет происходить доступ к элементам за пределами массива a; во-вторых, из процедуры `foo` возвращается адрес локальной переменной, который будет невалидным.

На самом деле, про вторую ошибку gcc выдаёт соответствующий warning:

```
a.cpp: In function ‘std::vector<int>* foo(int)’:
a.cpp:6:15: warning: address of local variable ‘a’ returned [-Wreturn-local-addr]
   vector<int> a;
```

но мы сделаем вид, что не заметили его — в реальности могут быть более запутанные ситуации, которые на этапе компиляции не заметны, а появляются только на этапе выполнения. (Хотя этот вовсе не обозначает, что надо игнорировать warning'и — наоборот, их игнорировать ни в коем случае нельзя.)

Скомпилируем и запустим эту программу. Мы заметим, что она вполне разумно работает, и по крайней мере при маленьких n выдаёт вполне разумные результаты. На моем компьютере (Kubuntu Linux 14.04, gcc 4.8.2) она прекрасно работает вплоть до $n = 33789$, а уже при $n = 33790$ падает с ошибкой `Segmentation fault (core dumped)`.

В принципе, будь в задаче ограничение $n \leqslant 10000$, мы бы вообще не заметили бы ничего подозрительного.

Давайте посмотрим, как мы можем искать, где происходит segfault, или как мы можем найти эти ошибки, если даже о segfault и не подозреваем.

1. Valgrind. Это — отдельная утилита под Linux для поиска подобных ошибок. Под Windows она не доступна, под OS X, говорят, доступна, но я не проверял.

Скомпилируем нашу программу как обычно, а потом запустим из командной строки

```
valgrind ./a
```

здесь `./a` — команда, которую мы хотим выполнить, т.е. команда запуска нашей программы. Мы увидим следующий вывод:

```
==6480== Memcheck, a memory error detector
==6480== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6480== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==6480== Command: ./a
==6480==
```

На этом выполнение программы остановится: программа ждёт, когда мы введём n. Введём для начала 1 и увидим следующее (здесь и далее я разбил некоторые строчки, чтобы влезло по ширине):

```
==6579== Invalid read of size 4
==6579==    at 0x8048C08: std::vector<int, std::allocator<int> >::operator[](unsigned int)
    (in /home/petr/tmp/valgrind/a)
==6579==    by 0x8048A78: main (in /home/petr/tmp/valgrind/a)
==6579==  Address 0xbef13e14 is just below the stack ptr.  To suppress, use: --workaround-gcc296-bugs=yes
==6579==
==6579== Invalid read of size 4
==6579==    at 0x8048A79: main (in /home/petr/tmp/valgrind/a)
==6579==  Address 0x435a02c is 4 bytes inside a block of size 16 free'd
==6579==    at 0x402B838: operator delete(void*) (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==6579==    by 0x804924A: __gnu_cxx::new_allocator<int>::deallocate(int*, unsigned int)
    (in /home/petr/tmp/valgrind/a)
==6579==    by 0x8048DC0: std::_Vector_base<int, std::allocator<int> >::_M_deallocate(int*, unsigned int)
    (in /home/petr/tmp/valgrind/a)
==6579==    by 0x8048C70: std::_Vector_base<int, std::allocator<int> >::~_Vector_base()
    (in /home/petr/tmp/valgrind/a)
==6579==    by 0x8048B49: std::vector<int, std::allocator<int> >::~vector()
    (in /home/petr/tmp/valgrind/a)
==6579==    by 0x8048A16: foo(int) (in /home/petr/tmp/valgrind/a)
==6579==    by 0x8048A60: main (in /home/petr/tmp/valgrind/a)
==6579==
2
==6579==
==6579== HEAP SUMMARY:
==6579==     in use at exit: 0 bytes in 0 blocks
==6579==   total heap usage: 1 allocs, 1 frees, 16 bytes allocated
==6579==
==6579== All heap blocks were freed -- no leaks are possible
==6579==
==6579== For counts of detected and suppressed errors, rerun with: -v
==6579== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Здесь 2 — это вывод нашей программы, а все, что начинается со знаков равенства — это вывод Valgrind'a. Сообщения достаточно просто читаются: «ошибочное чтение данных размера 4». Под сообщением об ошибке указано, где она произошла. В первом случае это в функции `std::vector<int,...>::operator[]`, (т.е. в функции доступа к элементу вектора), вызванной из функции `main`. Во втором случае — это просто в функции `main`.

После этого идёт информация о том, что это за адрес. В первом случае это `Address 0xbef13e14 is just below the stack ptr` — видимо, это следует понимать, что этот адрес «над» указателем на вершину

стека функций — можно догадаться, что это локальная переменная функции, которая только что завершилась. Но вообще это уточнение действительно не очень помогает, приходится довольствоваться тем, что мы знаем, что ошибка в операторе `[]` в функции `main`.

Зато во второй ошибке уточнение намного более детальное. Оно начинается фразой `Address 0x435a02c is 4 bytes inside a block of size 16 free'd`, которая легко понимается как «Этот адрес (который мы ошибочно попытались прочитать) находится на расстоянии 4 байта от начала блока памяти размера 16, который был освобождён...», и далее указывается, где именно освобождён (освобождён — в смысле освобождения занимаемой памяти). Блок был освобождён в операторе `delete` в функции `deallocate` в ... в функции `std::vector<...>::~vector()` (т.е. в деструкторе вектора!) в функции `foo`. Ура, это очень понятно: в функции `foo` был удалён некоторый вектор, к которому мы теперь пытаемся получить доступ! Если повнимательнее посмотреть на код, то найти место ошибки несложно.

Ещё проще все будет, если скомпилировать программу с ключом `-g`, т.е. командой вида
`g++ -g a.cpp -o a`.
Этот ключ заставляет gcc сохранить информацию, которая будет полезна при отладке, в том числе соответствие между инструкциями программы и номерами строк исходного файла. В результате вывод Valgrind'a станет ещё понятнее (я опустил неинтересные куски):

```
==6690== Invalid read of size 4
==6690==    at 0x8048C08: std::vector<int, std::allocator<int> >::operator[](unsigned int) ...
==6690==    by 0x8048A78: main (a.cpp:18)
==6690==  Address 0xbea39e14 is just below the stack ptr.  ...
==6690==
==6690== Invalid read of size 4
==6690==    at 0x8048A79: main (a.cpp:18)
==6690==  Address 0x435a02c is 4 bytes inside a block of size 16 free'd
<...>
==6690==    by 0x8048A16: foo(int) (a.cpp:11)
==6690==    by 0x8048A60: main (a.cpp:17)
==6690==
```

Обратите внимание, что теперь выводятся номера строк в нашей программе (`a.cpp:18`), что позволяет совсем точно найти, где происходит ошибочное чтение.

Давайте теперь запустим нашу программу из-под Valgrind ещё раз, но на этот раз введём ей в качестве n число 5. Мы получим ещё ошибки следующего вида:

```
==6723== Invalid write of size 4
==6723==    at 0x80489FB: foo(int) (a.cpp:10)
==6723==    by 0x8048A60: main (a.cpp:17)
==6723==  Address 0x435a038 is 0 bytes after a block of size 16 alloc'd
==6723==    at 0x402A6DC: operator new(unsigned int) (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==6723==    by 0x804959B: __gnu_cxx::new_allocator<int>::allocate(unsigned int, void const*) (new_allocator.h:104)
==6723==    by 0x804942D: std::_Vector_base<int, std::allocator<int> >::_M_allocate(unsigned int)
    (in /home/petr/tmp/valgrind/a)
==6723==    by 0x8049035: std::vector<int, std::allocator<int> >::_M_fill_insert
        (__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, unsigned int, int const&)
        (vector.tcc:483)
==6723==    by 0x8048D1F: std::vector<int, std::allocator<int> >::insert
        (__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, unsigned int, int const&)
        (stl_vector.h:1024)
==6723==    by 0x8048BC9: std::vector<int, std::allocator<int> >::resize(unsigned int, int) (stl_vector.h:707)
==6723==    by 0x80489A9: foo(int) (a.cpp:7)
==6723==    by 0x8048A60: main (a.cpp:17)
```

(Это все с ключом компиляции `-g`.)

Здесь тоже чётко указано, что на десятой строчке программы мы попытались записать данные размера 4 по некорректному адресу. Этот адрес расположен в 0 байт позади (т.е. сразу после) блока из 16 байт, который был выделен (allocated) внутри ... команды `std::vector::resize()`, вызванной на седьмой строке нашей программы.

Согласитесь, что это весьма полезная информация, намного полезнее, чем простой segfault.

2. AddressSanitizer. Это — встроенная в компилятор возможность проверки подобных ошибок. Она доступна в новых версиях компилятора gcc (начиная с версии 4.8.0), под линуксом точно, но вроде как и под windows. Чтобы её задействовать, надо скомпилировать программу, с параметром `-fsanitize=address` (прямо так):

```
g++ -fsanitize=address a.cpp -o a
```

После этого просто запустим программу и введём ей в качестве параметра число 1. Увидим следующий вывод:

```
==6906== ERROR: AddressSanitizer: heap-use-after-free
    on address 0xb5c007f4 at pc 0x8048e77 ...
```

```
READ of size 4 at 0xb5c007f4 thread T0
    #0 0x8048e76 (/home/petr/tmp/valgrind/a+0x8048e76)
    #1 0xb5e42a82 (/lib/i386-linux-gnu/libc-2.19.so+0x19a82)
    #2 0x8048b40 (/home/petr/tmp/valgrind/a+0x8048b40)
0xb5c007f4 is located 4 bytes inside of 16-byte region [0xb5c007f0,0xb5c00800)
freed by thread T0 here:
    #0 0xb60f0824 (/usr/lib/i386-linux-gnu/libasan.so.0.0.0+0x11824)
    #1 0x8049fd8 (/home/petr/tmp/valgrind/a+0x8049fd8)
    #2 0x80494e0 (/home/petr/tmp/valgrind/a+0x80494e0)
    #3 0x8049210 (/home/petr/tmp/valgrind/a+0x8049210)
    #4 0x8049014 (/home/petr/tmp/valgrind/a+0x8049014)
    #5 0x8048d94 (/home/petr/tmp/valgrind/a+0x8048d94)
    #6 0x8048e28 (/home/petr/tmp/valgrind/a+0x8048e28)
    #7 0xb5e42a82 (/lib/i386-linux-gnu/libc-2.19.so+0x19a82)
previously allocated by thread T0 here:
    #0 0xb60f0624 (/usr/lib/i386-linux-gnu/libasan.so.0.0.0+0x11624)
    #1 0x804a4b6 (/home/petr/tmp/valgrind/a+0x804a4b6)
    #2 0x804a26f (/home/petr/tmp/valgrind/a+0x804a26f)
    #3 0x8049b42 (/home/petr/tmp/valgrind/a+0x8049b42)
    #4 0x804932f (/home/petr/tmp/valgrind/a+0x804932f)
    #5 0x8049095 (/home/petr/tmp/valgrind/a+0x8049095)
    #6 0x8048c7b (/home/petr/tmp/valgrind/a+0x8048c7b)
    #7 0x8048e28 (/home/petr/tmp/valgrind/a+0x8048e28)
    #8 0xb5e42a82 (/lib/i386-linux-gnu/libc-2.19.so+0x19a82)
...
```

К сожалению, здесь мало полезной информации, разве что сообщение о том, что ошибка чтения из 16-байтного региона, освобождённого раньше. Ключ `-g` не помогает; в интернете есть рекомендации использовать внешние скрипты — но можете это изучить сами.

При вводе $n = 5$ получаем другую ошибку:

```
WRITE of size 4 at 0xb5d00800 thread T0
    <...>
0xb5d00800 is located 0 bytes to the right of 16-byte region [0xb5d007f0,0xb5d00800)
```

что тоже даёт немного информации, хотя все же больше, чем ничего.

3. Общие замечания. Ни Valgrind, ни AddressSanitizer не способны поймать абсолютно все ошибки доступа к памяти. Они, фактически, только знают, какая память доступна вашей программе, а какая нет, и могут проверять только это. В частности, если бы мы использовали не `std::vector` (который выделяет память на куче), а обычный массив (`int a[4]`), то что Valgrind, что AddressSanitizer могли бы и не заметить ошибок. Точнее, Valgrind заметил бы ошибку возврата локальной переменной, а вот ошибку выхода за пределы массива не заметил бы. AddressSanitizer в таком случае замечает только выход за пределы массива, но не возврат локальной переменной.

Можете поэкспериментировать и посмотреть, кто, когда и какие ошибки может заметить.

Ещё имейте в виду, что Valgrind существенно затормаживает работу программы (до 10–20 раз). AddressSanitizer — не настолько, но я особенно не тестировал.

Ну и да, если вы пишите на паскале, то все намного проще: ключи компилятора `{$r+,q+,s+,i+}` вам помогут.