Заметки по алгоритмическому программированию

Автор: Петр Калинин, 2008–2015
Этот документ можно распространять по лицензии
@bookID, author = author, title = title, date = date, OPTeditor = editor, OPTeditora = editora, OPTeditorb = editorb, OPTeditorc = editorc, OPTtranslator = translator, OPTannotator = annotator, OPTcommentator = commentator, OPTintroduction = introduction, OPTforeword = foreword, OPTafterword = afterword, OPTsubtitle = subtitle, OPTtitleaddon = titleaddon, OPTmaintitle = maintitle, OPTmainsubtitle = mainsubtitle, OPTmaintitleaddon = maintitleaddon, OPTlanguage = language, OPToriglanguage = origlanguage, OPTvolume = volume, OPTpart = part, OPTedition = edition, OPTvolumes = volumes, OPTseries = series, OPTnumber = number, OPTnote = note, OPTpublisher = publisher, OPTlocation = location, OPTisbn = isbn, OPTchapter = chapter, OPTpages = pages, OPTpagetotal = pagetotal, OPTaddendum = addendum, OPTpubstate = pubstate, OPTdoi = doi, OPTeprint = eprint, OPTeprintclass = eprintclass, OPTeprinttype = eprinttype, OPTurl = url, OPTurldate = urldate,
GNU General Public License версии 3 или более поздней.
Последнюю версию документа, а также исходный код для системы LaTeX можно скачать с `https://github.com/petr-kalinin/progtexts`

О лицензии на эти заметки

...право формулировать задачу и объяснять ее решение
является неотчуждаемым естественным правом всякого,
кто на это способен.

А. Шень. Программирование: теоремы и задачи

Эти заметки вы можете бесплатно скачать на сайте `https://github.com/petr-kalinin/progtexts` и использовать любыми законными способами. Я не беру денег за их использование и распространение, но, соответственно, *требую*, чтобы и вы не ставили никаких ограничений на использование этого текста, если вы его куда-то дальше распространяете. Более того, с того же сайте вы можете свободно скачать и исходный код этих заметок для системы LaTeX, и вы можете в него вносить изменения и/или использовать в своих работах, но я требую, чтобы в таком случае вы сделали свободным и исходный код ваших исправлений или тех работ, где вы используете текст этих заметок.

А именно, я распространяю эти заметки на условиях лицензии GNU General Public License версии 3 (или, по вашему выбору, более старшей версии). Строгий текст лицензии вы можете прочитать на сайте Free Sofware Foundation по адресу `http://www.gnu.org/licenses/gpl-3.0.en.html` (неофициальный русский перевод: `http://rus-gpl.ru/rusgpl.html`), или в файле `COPYING.txt`, распространяемым вместе с LaTeX-кодами этих заметок. Ниже я вкратце объясняю, что это обозначает; если вы знаете GNU GPL, то можете пропустить эти объяснения до раздела «Дополнительные замечания». В частности, если вам интересно, почему я использую именно GNU GPL, а не CC BY-SA или другую CC лицензию, то читайте раздел «Дополнительные замечания».

Итак, вы можете свободно использовать эти заметки при условии, что вы их не изменяете (в частности, сохраняете указание на мое авторство и указание на лицензию). Вы можете их также распространять (выставлять на сайте, распространять в печатном виде и т.п.) куда угодно на этих же условиях (т.е. по этой же лицензии). Кроме того, вы можете модифицировать эти заметки, а также использовать текст заметок в своих работах, на следующих условиях. Во-первых, вы должны распространять полученный текст (измененную версию этих заметок или свою собственную работу) также на условиях GNU GPL, в том числе не ограничивая дальнейшее распространение. Во-вторых, вы должны сделать свободно доступным на условиях GNU GPL «*исходный код*» измененных заметок или вашей работы, т.е. тот формат, в котором вы сами вносили изменения и/или создавали свою работу. А именно, если вы правите исходный код LaTeX, то вы должны сделать исправленный исходный код LaTeX свободно доступным. Вы не можете распространять только полученный PDF или вообще только печатную версию, а исходный LaTeX-код сделать закрытым. Или, если вы, к примеру, используете Photoshop для правки уже сгенерированного PDF, то вы должны распространять также и «сырой» файл PSD. Этот «*исходный код*» вы должны или распространять сразу вместе с окончательным вариантом (PDF, печатной версией и т.п.), или в PDF или печатную версию должна быть включена информация о том, как получить этот «*исходный код*». Более строгие определения см. в полном тексте лицензии, ссылки на который приведены выше.

Дополнительные замечания

Вообще, часто для подобных «творческих работ» используются лицензии Creative Commons (CC). Но я распространяю эти заметки на условиях именно GNU GPL потому, что CC-лицензии не содержат понятия исходного кода и не требуют его раскрытия. Если бы я распространял заметки на условиях лицензий CC, то кто угодно мог бы внести изменения в LaTeX-исходники, скомпилировать их в PDF и дальше распространять только PDF, а LaTeX-исходники засекретить. Это довольно непродуктивно, т.к. скомпилированный PDF править намного сложнее, чем LaTeX-исходники. Я хочу, чтобы все измененные версии этих заметок сопровождались исходниками, чтобы их мог править кто угодно, да и чтобы я сам мог подходящие изменения включить в оригинальный текст заметок. Соответственно, GNU GPL имеет понятие *исходного кода*, причем с достаточно широким определением: «the preferred form of the work for making modifications to it» («предпочитаемая форма произведения для создания его модификаций»). Для заметок, набранных в LaTeX, исходный код — это очевидно исходный код LaTeX, поэтому GPL позволяет достичь именно той цели, которую я сформулировал выше: она требует, чтобы всякий, кто будет распространять измененную версию заметок, распространял бы и соответствующий исходный код LaTeX.

Кстати, исходный код этих заметок лежит в репозитории на GitHub. Если уж вы изменяете заметки, я буду очень рад, если вы пришлете pull request, чтобы я мог при желании легко включить ваши изменения в мою версию.

Еще обратите внимание, что лицензия, конечно, относится только к конкретному тексту этих заметок. Авторское право вообще защищает не идеи, а конкретные воплощения этих идей, поэтому, конечно, я не претендую и не могу претендовать на какие-либо авторские права в отношении алгоритмов, задач и т.п., описанных в этих текстах. Авторские права распространяются только на конкретную текстовую формулировку этих алгоритмов и задач. Кроме того, конечно, я разрешаю свободное использование без каких-либо требований лицензии разумно небольших фрагментов этих заметок. Например, вы, конечно, можете использовать фрагменты кода (но не целые программы), приведенные в этих заметках, в своих программах без каких-либо ссылок на меня, или формулировки отдельных задач и т.п.

Раньше я распространял некоторые фрагменты этих заметок по лицензии CC-BY-SA. Конечно, на них по-прежнему распространяется эта лицензия, но на этот текст и на последующие версии распространяется только лицензия GPL.

Оглавление

Часть 1.

Начало работы в Free Pascal

1.1. Установка

Free Pascal (сокращённо FP) — это свободное кросс-платформенное программное обеспечение, поэтому его можно легко скачать с официального сайта, можно свободно распространять, и можно установить на все современные операционные системы.

Чтобы установить FP под Windows, скачайте программу установки с официального сайта `http://freepascal.org` (что не очень тривиально), или из другого надежного источника (участники моего курса на informatics могут скачать установщик со странички курса). Установите FP с помощью этой программы, ничего сложного в установщике нет. Я точно не помню, создаёт ли установщик ярлык на рабочем столе; если нет, то найдите исполняемый файл FP. Путь к нему имеет вид `C:\FPC\bin\i386-win32\fp.exe` (начальная часть может отличаться в зависимости от того, куда установлен FP, у меня он установлен в `C:\FPC`). Удобнее всего вынести ярлык к FP на рабочий стол или ещё куда-нибудь, чтобы было легко запускать.

Если вы работаете в другой операционной системе, то разберитесь, как установить FP, самостоятельно. В Linux, например, FP есть в репозиториях всех ведущих дистрибутивов.

1.2. Первая программа

Запустите Free Pascal. Появится окошко, похожее на показанное на рис. 1.1а или 1.1б. Если появилось окошко, показанное на рис. 1.1а, то в FP, в меню File выберите пункт New — окошко примет вид, показанный на рис. 1.1б. В этом окне наберите следующий текст (рис. 1.2):

```
begin
writeln('This is test! ',2*2);
end.
```

(Здесь ' — это символ, называемый апостроф. Он находится в латинской раскладке клавиатуры на той же клавише, что и русская буква э. В разных шрифтах он выглядит по-разному, но обычно как половина символа кавычек.)

Убедитесь, что опечаток нет. Сохраните программу: нажмите F2 или выберите пункт меню File — Save. FP предложит выбрать имя файла для сохранения, для первой программы можно выбрать любое имя.

После этого запустите программу, нажав Ctrl-F9 или в меню Run выбрав пункт Run. (Если вы не сохранили программу заранее, то FP предложит вам сохранить её перед запуском. Сохраните.) Окно FP на мгновение пропадёт, может быть, вы успеете увидеть за ним другое, чёрное, окошко, после чего окошко FP с набранной программой (рис. 1.2) появится обратно. Это значит, что программа успешно выполнилась. Если вместо этого вы видите картинку, похожую на рис. 1.4а, это значит, что в программе есть ошибки. Про это ниже (раздел 1.3), а пока будем считать, что ошибок нет и программа успешно выполнилась.

Нажмите Alt-F5 или в меню Debug выберите пункт User screen. Синее окошко редактора пропадёт, а появится то самое чёрное окошко, которое вы могли заметить, когда запускалась программа. Оно показано на рис. 1.3а. В нем, во-первых, виден заголовок FP — строчки, начинающиеся с квадратика. Их FP выводит один раз при каждом запуске редактора. Далее идёт строка "`Running "c:\fpc\bin\i386-win32\1.exe "`" (путь может быть другой, в зависимости от того, куда сохранена ваша программа). Это строка была выведена в тот момент, когда FP начал запускать вашу программу. И, наконец, есть строка "`This is test! 4`", которую и напечатала наша программа. Почему она напечатала именно это, обсудим чуть ниже.

На чёрном экране нажмите любую клавишу — вы вернётесь обратно в редактор. Позапускайте программу (Ctrl-F9) ещё несколько раз и посмотрите на результаты (Alt-F5). Вы увидите картину, показанную на рис. 1.3б: FP каждый раз печатает строку "`Running...`" перед запуском программы, потом программа печатает свою строку. Вывод программы перемешивается с выводом FP — ничего страшного, это нормально.

a) a)

Рис. 1.1: Основное окно FP

Рис. 1.2: Простейшая программа

а) б)

Рис. 1.3: Вывод программы

а) б)

Рис. 1.4: Ошибка компиляции

1.3. Ошибки компиляции

Компиляцией называется (это не совсем точное определение) процесс перевода программы из текстового вида ("исходного кода") в такой вид, в котором её можно запустить на компьютере ("исполняемый файл"). Когда вы нажимаете Ctrl-F9, происходит сначала компиляция, а потом полученный исполняемый файл запускается. Соответственно, система, которая выполняет компиляцию, называется компилятором; это — основная часть Free Pascal.

В вашей программе могут быть серьёзные ошибки — такие, что компилятор даже не может скомпилировать программу, и уж тем более её невозможно запустить (а могут быть и не столь серьёзные — программа запустится, но выдаст неверный результат). В таком случае компилятор выдаст сообщение, похожее на показанное на рис. 1.4а. В этом случае следует поступать следующим образом. Не вчитывайтесь в то, что показано в этом окошке. Вместо этого нажмите два раза Enter — FP поставит вас курсор на то место, где он обнаружил ошибку, и напишет красную строчку с сообщением об ошибке, см. рис 1.4б.

Пока для вас важным будет то, куда компилятор поставил курсор — примерно в том месте и ошибка. Важен также текст ("сообщение об ошибке"), выданный на красной строчке ("`Syntax error, ...`"). Поначалу сообщения об ошибке сложно понимать, но со временем вы выучите наиболее часто встречающиеся и будете сразу понимать, что не так. В данном случае, если вы знаете английский, то сообщение об ошибке понятно: «Синтаксическая ошибка, "." ожидалась, но "конец файла" найден». Очевидно, мы забыли точку после end. Если вы не так хорошо знаете английский, то вам придётся внимательно всматриваться в программу и искать ошибку, хотя это, конечно, не очень сложно. На самом деле, если даже вы неплохо знаете английский, вы все равно далеко не все сообщения об ошибках сможете понимать — поначалу смысл многих сообщений будет загадочен, и вам придётся все равно просто внимательно искать опечатки и ошибки. Со временем, независимо от того, хорошо ли вы знаете английский, список наиболее частых ошибок вы выучите.

Обратите ещё внимание, что компилятор поставил курсор не туда, где мы забыли точку, а дальше. Дело в том, что, строго говоря, между `end` и точкой можно было ещё поставить сколько угодно пробелов и переводов строк. Так оказалось (точнее, это я специально сделал), что после `end` в этом примере у меня шли ещё несколько пробелов и переводов строк. Их компилятор честно пропустил, ожидая найти точку, но файл кончился, а точки так и не нашлось. Поэтому компилятор сообщил об ошибке, но уже на конце файла, а не сразу после `end`.

Вывод из этой ситуации такой: компилятор не телепат и не может точно определить, где вы допустили ошибку. Он устанавливает курсор туда, где текст программы впервые разошёлся с правилами языка. Поэтому бывает, что на самом деле ваша ошибка чуть выше, чем курсор (а иногда — и намного выше). Но тем не менее место, куда компилятор ставит курсор, обычно бывает полезно при поиске ошибки.

Попробуйте в своей программе поделать разные ошибки и посмотрите, как на них отреагирует FP. (Особенно внимательные смогут заметить, что эту программу можно слегка изменить определённым образом так, что будет казаться, что появилась ошибка, но компилятор не будет ругаться. Если вы найдёте, как это сделать, то знайте, что это правда, это действительно не ошибка, но так писать просто не принято.)

1.4. Как работает эта программа

Давайте разберём, как эта программа работает. Напомню её текст:

```
begin
writeln('This is test! ',2*2);
end.
```

Вообще, любая программа — это, в первую очередь, последовательность команд, которые программист даёт компьютеру, а компьютер будет последовательно их выполнять. В языке программирования паскаль начало этой последовательности команд (т.е. начало программы) обозначается словом `begin`, а конец — словом `end.` (с точкой на конце!). Это — обязательные элементы любой программы, поэтому они есть и в нашей программе.

Между ними в нашей программе одна команда — `writeln('This is test! ',2*2);`. Команда `writeln` обозначает "вывести на экран" (английское слово 'write' обозначает писать, а ln — это сокращение от line — строка). В скобках после слова `writeln` указываются, как говорят, *аргументы* команды. Они разделяются запятыми, в данном случае у команды два аргумента: первый — `'This is test! '`, и второй — `2*2`. После команды ставится точка с запятой; вообще, это общее правила паскаля — после каждой команды должна идти точка с запятой (есть некоторые исключения, но пока они нам не важны).

Если аргументом команды является некоторая строка, заключённая в апострофы (символы '), то команда `writeln` выводит эту строку на экран как есть (без апострофов). Поэтому первым делом наша команда выводит на экран текст "`This is test! `" (включая пробел на конце!).

Вторым аргументом команды `writeln` в нашем примере является арифметическое выражение `2*2`. Если аргументом команды (любой команды, не обязательно именно `writeln`, просто других мы пока не знаем) является арифметические выражение, то компьютер сначала вычислит его, а потом передаст команде. Поэтому в данном случае сначала компьютер вычислит 2·2, получит 4, а потом передаст результат команде `writeln`, которая выведет его на экран.

В итоге получается, что наша программа выводит `This is test! 4`.

1.5. Использование паскаля как калькулятора

Таким образом можно использовать паскаль как калькулятор. Например, если надо посчитать значение выражения $7 + 3 \cdot (8 - 2)$, то можно написать команду `writeln(7+3*(8-2));`, не забыть `begin/end`, после чего запустить программу — и на экран будет выведен результат. Обратите внимание, что скобки учтутся корректно и порядок действий будет правильный. Две скобки в конце команды — это одна является частью выражения, а вторая заканчивает список аргументов команды `writeln`.

В выражениях можно использовать следующие операторы:

- `+-` — сложение и вычитание (в том числе то, что называется *унарный* минус для записи отрицательных чисел: чтобы написать $2 \cdot (-4)$, надо написать `2*(-4)`);
- `*` — умножение;
- `/` — деление, но с ним все не так просто, см. следующий раздел;
- `div` и `mod` — это деление с остатком. Вспомните младшие классы и деление с остатком: 16 разделить на 3 будет 5 ("неполное частное") и в остатке 1. Вот `div` вычисляет неполное частное, а `mod` — остаток. Пишется так: `16 div 3` и `16 mod 3`, как будто `div` и `mod` — это один большой символ, а-ля плюс или звёздочка. (Пробелы вокруг `div` и `mod` надо ставить, чтобы все не сливалось в одну строку);
- Скобки (только круглые) работают для группировки операций, можно использовать вложенные скобки, например, `2*(3-(4+6))`.

Кроме того, есть так называемые *функции*:

- Запись `abs(-3)` обозначает взятие числа по модулю: $|-3|$. Обратите внимание: пишется сначала *имя функции* (в данном случае `abs`), а потом в скобках — от чего взять эту функцию (от чего взять модуль в данном случае). То, что в скобках, аналогично командам называется *аргументом функции*.
- Аналогично, запись `sqrt(4)` обозначает взятие квадратного корня (если не знаете, что это такое, то пока пропустите этот пункт), но с ним те же сложности, что и с делением, см. ниже.

Все эти операции можно комбинировать. Например, команда `writeln( (20*3) + sqrt( 2+abs(5-7) ) );` выведет на экран значение выражения $20 \cdot 3 + \sqrt{2 + |5 - 7|}$. Пробелы в команде поставлены, чтобы проще было читать; вообще, в паскале пробелы можно ставить в любом разумном месте (внутри названий команд и чисел нельзя, но около скобок, знаков препинания и прочих символов можно).

В одной программе можно вычислять несколько выражений. Например, программа

```
begin
writeln(2*2,' ',2+2);
writeln(3*3);
end.
```

вычисляет три выражения. Первая команда `writeln` выводит на экран две четвёрки, разделённых пробелом. (Обратите внимание, что пробел мы выводим специально, указав вторым аргументом команды `writeln` строку `' '`, состоящую из одного пробела. Если бы мы не написали этот аргумент, то программа вывела бы две четвёрки подряд, и они слились бы в одно число 44. Компьютер не будет делать ничего, если вы его специально не попросите — в том числе не будет выводить пробелы между числами.)

Вторая команда просто выводит одно число 9. Оно будет выведено на отдельной строке, т.к. каждая команда `writeln` выводит одну строку.

1.6. Проблемы с делением

Проблема с делением, а также с извлечением квадратного корня состоит в том, что результат этой операции не всегда является целым числом. Компилятор не знает заранее, получится в результате целое число или нет, поэтому он всегда действует так, как будто получается вещественное число (вещественные числа — это и целые числа, и дробные). А вещественные числа паскаль по умолчанию выводит в форме, которая, возможно, является довольно непривычной для вас.

Напишем, например, следующую команду: `writeln(16/3);`. По этой команде на экран будет выведено `5.3333333333333333E+0000`. Это — так называемая *экспоненциальная*, или *научная* форма записи вещественных чисел, или форма записи *с плавающей точкой*.

Её читают следующим образом. Она всегда содержит ровно одну букву E (латинскую, большую или маленькую). Перед буквой E записывается обычная число — десятичная дробь, в данном случае 5.3333333333333333 (если вы ещё не знаете, то знайте, что в компьютерах используется не десятичная запятая, а десятичная точка). После буквы E идёт целое число, в данном случае ноль. Оно обозначает, надо ли и на сколько позиций сдвинуть десятичную точку. Если после E идёт положительное число, то десятичную точку надо сдвинуть на столько позиций вправо, если отрицательное — то влево. В нашем случае после E идёт число 0, поэтому точку двигать не надо, и 5.3333333333333333 и есть ответ. И правда, $16/3 \approx 5.3333333333333333$; знак приближенного равенства стоит потому, что на самом деле дробь бесконечная, но компьютер не может работать с бесконечными дробями, поэтому он отбросил хвост дроби, оставив только начало.

Если бы мы написали `160/3`, то увидели бы результат `5.3333333333333333E+0001`. Это значит, что надо взять число 5.3333333333333333 и сдвинуть точку на одну позицию вправо, получив 53.333333333333333. И это действительно правильный ответ.

Аналогично, `1/3` выводит на экран `3.3333333333333333E-0001`, т.е. надо взять число 3.3333333333333333 и сдвинуть точку на одну позицию влево (приписав ведущие нули), получится 0.33333333333333333.

Ещё примеры: $250 = 2.5e2$, $0.00123 = 1.23e{-}3 = 12.3e{-}4 = 123e{-}5$. Буква E, как я уже говорил, может быть и большой, и маленькой, хотя при выводе на экран паскаль всегда использует большую букву. В последнем примере видно, как точка может перемещаться по числу, поэтому запись и называется записью с плавающей точкой. (Хотя при выводе на экран паскаль всегда делает так, чтобы перед точкой была ровно одна ненулевая цифра.) Ведущие нули после E также, конечно, можно не писать, хотя паскаль при выводе всегда пишет число после E из четырёх цифр.

(На самом деле запись имеет следующую простую трактовку: буква E обозначает "умножить на 10 в такой-то степени". Например, записать $2.5e2$ обозначает "2.5 умножить на 10 в степени 2": $2.5 \cdot 10^2$. Несложно видеть, что это равносильно тому, что написано выше.)

Эта запись на самом деле широко используется, и, даже если вы с ней ещё не сталкивались, то будете с ней ещё много сталкиваться. Особенно широко она используется в науке (и потому называется научной), поэтому что позволяет легко записывать большие и маленькие числа. Например, свет в вакууме проходит примерно 300000000 метров за одну секунду. Если каждый раз писать это число, то очень легко запутаться в нулях. Если же написать $3e8$, то ошибку допустить намного сложнее. Аналогично, например, масса одного атома кислорода равна примерно $2.65e{-}23$ грамма, т.е. 0.00000000000000000000000265 грамма. Ясно, что первая запись проще. (Правда, в тексте часто пишут не $2.65e{-}23$, а $2.65 \cdot 10^{-23}$, но суть та же. В программах пишут через E, т.к. писать верхний индекс сложно.)

Поэтому это вполне законная запись; её понимают все люди и все нормальные программы. Поэтому паскаль и выводит в таком виде — чтобы уверенно выводить как очень большие, так и очень маленькие числа, он ведь не знает заранее, какой результат получится. Такой вывод абсолютно нормален; в частности, везде в нашем курсе, когда вам надо будет выводить вещественное число, вывод такого формата будет считаться корректным. Если вы будете сдавать задачи на нашем сайте, то программа, выводящая ответ в таком виде не будет иметь никаких проблем.

Аналогично, компьютер сам понимает такую запись. Вы можете в программе записывать числа с плавающей точкой, и программа будет прекрасно работать. Например, если вы хотите узнать, сколько весят $2.68e22$ атомов кислорода (примерно столько атомов кислорода содержатся в одном литре кислорода при атмосферном давлении и комнатной температуре), просто напишите в паскале `writeln(2.68e22 * 2.65e-23);`.

В общем, не бойтесь этой записи, и умейте её читать. Но если вы хотите, чтобы паскаль выводил вещественные числа обычным образом, без E, то воспользуйтесь следующим трюком. После арифметического выражения в аргументе команды `writeln` напишите магическую строчку `:20:20`, например `writeln( 1/3 :20:20);` (пробелы здесь стоят для удобства чтения, на самом деле они не обязательны). Мы пока не будем обсуждать, что это значит, просто запомните. Потом узнаете подробнее, как и почему это работает.

1.7. Простейший ввод и вывод. Переменные

Но не очень интересно писать программы, которые всегда выводят одно и то же. Хочется, чтобы программа что-нибудь запрашивала у пользователя, и работала с учётом того, что пользователь ввёл. Давайте, например, напишем программу, которая будет спрашивать у пользователя два числа и выводить на экран их сумму.

Но для этого нам придётся научиться ещё одной важной вещи. Когда пользователь вводит два числа, программе надо их как-то запомнить, чтобы потом сложить между собой и результат вывести на экран. Для этого у компьютера есть память (оперативная память). Программа может попросить себе два кусочка этой памяти и положить туда числа, введённые пользователем. А потом посмотреть, что там лежит, сложить эти два числа, и вывести на экран.

Чтобы попросить себе место в памяти, есть специальная конструкция:

```
var a,b:integer;
```

Здесь `var` — это специальное слово, обозначающее "сейчас я буду просить память". Слово `integer` (в переводе с английского — целое число) обозначает, что нам нужна память для хранения целых чисел (а не вещественных, не строк, и т.п.) А буквы `a` и `b` — это названия, которые мы дадим тем двум кускам памяти, которые нам дадут — ведь надо же нам будет дальше в программе как-то эти кусочки памяти обозначать.

Полная программа будет выглядеть так:

```
var a,b:integer;
begin
read(a,b);
writeln(a+b);
end.
```

(Обратите внимание, что `var` идёт до `begin`.)

Построчно программу можно расшифровать так:

`var a,b:integer;` — хочу два места в памяти, в которых буду хранить целые числа. Одно место я буду называть `a`, а другое `b`.

`begin` — начинаются основные действия программы.

`read(a,b);` — новая команда, которая нам ещё не встречалась. Она обозначает: подожди, пока пользователь введёт с клавиатуры два числа, и сохрани их в память: одно — на место `a`, второе — на место `b`. Команде `read` можно передавать любое количество аргументов, но все они должны быть именами таких "кусочков памяти": например, если бы нам надо было считать только `a`, но не `b`, то можно было бы написать `read(a);`. Если бы у нас были бы четыре "кусочка памяти" `a`, `b`, `c` и `e`, то мы могли бы написать `read(a, b, c, e);`.

`writeln(a+b);` — здесь в арифметическом выражении встретились опять наши буквы. Это значит, что прежде чем вычислять арифметическое выражение, надо посмотреть, какие числа лежат на соответствующих местах в памяти, и подставить эти числа в это выражение, и только после этого проводить вычисления.

`end.` — программа закончилась.

Наберите эту программу и запустите её. Вы увидите, что, в отличие от прошлых программ, чёрное окошко не просто мелькнуло, а осталось активным на экране. Это значит, что программа ждёт, пока вы что-нибудь введёте. Наберите на клавиатуре, например, `2 3` и нажмите Enter. На экран вернётся обратно синее окошко редактора паскаля — это значит, что программа завершила работу. Нажмите Alt-F5 — и вы увидите, что программа перед этим вывела на экран 5 — т.е. сумму 2 и 3.

Таким образом, программа отработала.

Такие места в памяти, как мы сделали, называются *переменные*, т.е. у нас в программе две переменных: `a` и `b`. В качестве имён можно было выбрать другую строку, не обязательно только из одной буквы, — например, `first` и `second`. Переменных можно делать и больше, и меньше; например, если написать `var q,w,e,r,t:integer;`, то у вас будет пять переменных. Не бойтесь создавать переменные; переменные — это основная вещь, с которой работают программы.

Ещё несколько замечаний по нашей программе. Во-первых, программа не вывела на экран никаких "приглашений" типа "Введите a и b". Как уже было сказано выше, паскаль ничего за вас делать не будет; если вы хотите, чтобы программа вывела это на экран, то так и сделайте: `writeln('Введите a и b'); read(a,b);`. (Разбивать программу на строки, как и расставлять пробелы, можно тоже любым разумным образом, но обычно пишут по одной команде на строку.) Но мы не будет выводить такие приглашения в наших программах, мы будем считать, что пользователь сам знает, что от него требуется. В задачах, которые вы будете решать, будет чётко написано, что надо вывести на экран — и ничего лишнего выводиться не должно.

Второе замечание. В некоторых книжках вы можете прочитать, что ещё бывает команда `readln`. Не надо ей пользоваться (во всяком случае пока вы не работаете с переменными типа string). В книжках могут написать, что `read` нужно использовать, когда числа находятся на одной строке, а `readln` — когда на разных, но это неправда. Запустите нашу программу с `read`, и введите 2 Enter 3 Enter. Программа отработает корректно и выведет на экран 5, несмотря на то, что два числа были на разных строках. Дело в том, что `read` прекрасно работает и когда числа на одной строке, и когда на разных. Поэтому никакой необходимости использовать `readln` нет.

Третье замечание. Книжки могут советовать всякие премудрости типа `clrsrc;` (если не знаете, что это, то не страшно) или поставить `readln;` в конец программы — все это тоже не требуется; как видите, можно прекрасно работать и без этих команд.

И наконец четвёртое замечание. На самом деле, у компилятора Free Pascal есть несколько разных режимов работы, и эти режимы несколько различаются (например, в одном режиме в переменную типа `integer` нельзя записывать числа больше чем примерно 32000, а в другом — нельзя записывать числа больше чем примерно 2000000000 — два миллиарда). Мы будет изучать режим, называемый `delphi` (в нем в `integer` можно записывать числа до двух миллиардов, и есть ещё ряд отличий от других режимов). Чтобы быть уверенным, что ваша программа всегда будет компилироваться именно в этом режиме, напишите в начале программу магическую строку `{$mode delphi}`. В итоге программа будет выглядеть так:

```
{$mode delphi}
var a,b:integer;
begin
read(a,b);
writeln(a+b);
end.
```

Пока вы не знаете, какие бывают режимы компилятора, пишите всегда `{$mode delphi}` в начале программы. Потом, когда будете знать разные режимы, сможете выбирать, какой вам нужен, но вообще `delphi` — пожалуй, самый удобный.

1.8. Организация работы в отдельном каталоге

Вы будете писать много программ на Free Pascal, поэтому стоит на компьютере сделать отдельную папку, куда вы их будете складывать, и где вообще вы будете работать. Создайте такую папку и запомните путь к ней. После этого, запустите FP (или перезапустите его, если он у вас уже запущен), и в меню File выберите пункт Change dir. В открывшемся окне в дереве каталогов найдите свою папку (это может быть не так просто, но вы разберётесь), и нажмите кнопку Chdir. После этого команды открытия (File — Open) и сохранения (File — Save) файлов будут работать именно с указанной директорией (слова директория, каталог и папка — это синонимы). Вы сможете сохранять свои программы в неё, открывать ранее написанные программы, и т.д. Когда вы будете в будущем писать программы, читающие данные из файлов, то эти файлы вы тоже будете размещать в этой директории, и т.д. (Конечно, с программами и файлами в других директориях вы тоже сможете работать, но несколько сложнее.)

Выполняйте команду File — Change dir каждый раз при запуске FP. Это надо делать когда у вас не открыто ни одного файла (лучше сразу после запуска FP), иначе в диалогах открытия и сохранения файла вам, возможно, придётся ещё раз переходить в нужный каталог.

Или настройте ярлык для FP так, чтобы ваш каталог являлся "рабочей папкой" для этого ярлыка (см. следующий раздел). Когда садитесь работать за другой компьютер, в первую очередь выясняйте, в каком каталоге вы можете сохранять свои решения, и переходите (File — Change dir) в этот каталог.

Если вы никуда не переходили, что FP будет сохранять ваши программы туда же, где находится сам Free Pascal. Сохранять свои программы туда — признак дурного тона, т.к. мешать свои программы и программы, относящиеся к FP, не стоит.

Следующие три абзаца относятся к Windows; но если у вас другая операционная система, то там могут возникнуть аналогичные проблемы, поэтому прочитайте их все равно.

Итак, после того, как вы сделали каталог для своих программ, сохраните в нем какую-нибудь программу и запустите её. После этого перейдите в этот каталог через "Мой компьютер" или любым другим штатным средством просмотра каталогов. Вы обнаружите, что в каталоге есть несколько файлов с именем, совпадающим с именем вашей программы. Файлы отличаются расширением: файл с расширением `.pas` (например, `myprogram.pas`) — это и есть исходный код вашей программы, файл с расширением `.exe` — это полученный из программы исполняемый файл; могут быть и другие файлы, например, `.bak` — это запасная копия вашего файла (созданная перед последним сохранением).

Возможно, вы не увидите расширений файлов — возможно, вы просто увидите кучу файлов с одним и тем же именем (например, `myprogram`), только с разными значками. Тогда наведите курсор на любой из этих файлов. Появится всплывающая подсказка, в которой, в том числе, возможно, будет указан тип файла (например, Файл "BAK"). К сожалению, Windows не для всех файлов пишет тип, но для .pas-файлов, скорее всего, напишет Файл "PAS" или что-то подобное; также может быть "Delphi Source File", если у вас на компьютере ещё и установлена Delphi.

Чтобы с этим не путаться, в просмотре папки зайдите в меню Сервис — Параметры папок (в разных версиях Windows название пункта меню может отличаться). На закладке "Вид" в огромном списке среди кучи галочек и переключателей найдите галочку "Скрывать расширения для зарегистрированных типов файлов". Отключите эту галочку и нажмите OK. Теперь у каждого файла вы будете видеть расширение и уже не перепутаете исполняемый файл (`.exe`) и файл с исходным текстом (`.pas`).

Зачем это нужно? Затем, что если вы куда-то хотите отправить свою программу, и хотите, чтобы получатель мог разобраться в том, что у вас в ней написано, то, конечно, вам надо отправлять именно `.pas`, а не `.exe`. В частности, когда вы будете сдавать свои решения в тестирующую систему на сайте, вам надо будет отправлять именно `.pas`; если вы захотите отправить свою программу мне, чтобы я вам с ней помог, то тоже отправляйте `.pas`. Отправить `.exe` вместо `.pas` — это частая ошибка, не допустите её!

1.9. Работа с буфером обмена Windows

Одним из недостатков среды Free Pascal является сложная работа FP с буфером обмена Windows. Привычные вам команды Ctrl-C/Ctrl-V во Free Pascal не работают вообще. Вместо них есть команды Ctrl-Ins/Shift-Ins (на самом деле, и во многих других Windows-приложениях они являются синонимами Ctrl-C/Ctrl-V соответственно), но в FP эти команды работают с внутренним буфером обмена FP, никак не связанным с буфером обмена Windows. Это значит, что скопировать текст из одного места программы в другое место вы можете, а вот скопировать текст из FP в другое Windows-приложение и наоборот вы не можете. Но на самом деле это возможно: для этого в меню Edit есть пункты Copy to Windows и Paste from Windows — они и работают с буфером обмена Windows. Ими и надо пользоваться, чтобы скопировать текст из FP в Windows или наоборот.

1.10. Антивирус

Есть ещё одна проблема, которая вас может подстерегать, это антивирус. Многие антивирусы очень подозрительно относятся к работе Free Pascal, да и вообще любой среды для программирования, любого компилятора. Ещё бы — откуда ни возьмись, появляются новые программы, постоянно меняются (когда вы их перекомпилируете), а ещё сам FP (или среда для программирования) может пытаться залезать в память к написанным вами программам (особенно позже, когда вы будете заниматься *отладкой*). В общем, с точки зрения антивируса, когда вы программируете, на компьютере происходят странные вещи, и он может пытаться вам помешать. Например, известны случаи, когда при определённых условиях антивирус молча удаляет только что скомпилированную программу, и работать становится невозможно.

В общем, если происходит что-то странное, попробуйте отключить антивирус. Если поможет, то добавьте каталог, где вы работаете, в список исключений антивируса, или как-нибудь ещё настройте его, чтобы он вам не мешал.

1.11. Настройка окна FP

Не очень удобно, что окошко FP по умолчанию достаточно маленькое. Это — довольно неприятный недостаток FP, его интерфейс текстовый, а не графический. Но есть способ сделать его таким, как вам удобнее. Инструкции

дальше в этом разделе относятся к операционной системе Windows; если у вас другая операционная система, то попробуйте сделать аналогичную вещь средствами вашей ОС. Это может быть даже проще; например, в Linux FP автоматически принимает размер и шрифт той консоли, в которой его запустили.

Итак, ниже я описываю алгоритм настройки окна FP в Windows. Возможно, вы обнаружите, что какие-то действия здесь лишние, или, наоборот, вам придётся некоторые действия повторить два раза — я настолько подробно не проверял, как этот алгоритм работает.

Во-первых, необходимо создать ярлык к исполняемому файлу FP. Если вы ещё не создали ярлык на рабочем столе или ещё где-нибудь, то создайте его. Щёлкните по ярлыку правой кнопкой мыши и выберите пункт меню "Свойства". Вы увидите диалоговое окно с множеством вкладок. На основной вкладке (вкладке "Ярлык") есть, в частности, поле "Рабочая папка" — именно здесь надо указать тот каталог, где вы хотите хранить ваши программы, чтобы каждый раз не делать File — Change dir. (После этого при первом запуске из новой директории FP предложит вам создать файл настроек в этой директории — согласитесь, и в следующем диалоге выберите вариант "Copy existing").

Далее, в свойствах ярлыка нам важны будет вкладки "Шрифт" и "Расположение". Зайдите на вкладку "Шрифт", поиграйтесь с вариантами, и выберите какой вам удобнее. Я предпочитаю "Точечные шрифты" размера "10x18". Далее, перейдите на вкладку "Расположение" и в двух разделах "Размер буфера экрана" и "Размер окна" введите один и тот же размер, например, ширина 100, высота 40.

Нажмите в диалоговом окне OK и запустите FP с ярлыка. Вы не увидите особых изменений, кроме изменённого шрифта. Но теперь в меню Options выберите пункт Environment и там пункт Preferences. Вы увидите окошко настроек, в его левом верхнем углу будет выпадающее меню "Video mode". Выберите в нем последний пункт — он, скорее всего, имеет вид "100x40", т.е. содержит те самые размеры, которые вы указали ранее в свойствах ярлыка. Нажмите в диалоговом окне OK и вы увидите, что окошко FP стало существенно больше. Не выходя из FP (!), в меню Options выберите пункт Save. Вы сохранили настройки FP, и теперь он каждый раз будет запускаться с этим размером окна.

Более того, теперь если вы будете изменять размеры окна в свойствах ярлыка (только не забывайте, что там надо менять одновременно в разделе "Размер буфера экрана" и "Размер окна"), то окошко FP будет также меняться. В частности, вы можете сделать два ярлыка с разными размерами окна — и с одного ярлыка FP будет запускаться с одним размером окна, а с другого — с другим (например, это полезно, если вы работаете на ноутбуке и иногда подключаете к нему внешний монитор, на котором умещается бóльшее по размеру окно).

Часть 2.

Перебор с возвратом

Перебор — довольно своеобразный метод решения задач. На серьезных олимпиадах задачи на перебор встречаются довольно редко, но все равно во многих задачах, в которых есть «правильное», эффективное решение, перебор (хорошо написанный, с крутыми отсечениями и эвристиками) может помочь набрать если не полный балл, то хотя бы половину (а то и больше).

[Конечно, перебором в некотором смысле также можно называть решения, которые просто что-то перебирают, но мы перебором будем называть в первую очередь перебор некоторых, скажем так, комбинаторных объектов, написанный специфическим методом, который я и буду излагать ниже.]

Основная цель перебора — перебрать все объекты из некоторого множества, дабы что-то сделать с каждым. Наиболее часто, вроде, встречаются три варианта:

- либо надо найти объект (любой), удовлетворяющий некоторому условию,
- либо посчитать количество таких объектов,
- либо найти в некотором смысле оптимальный объект (дающий минимальную стоимость и т.п.)

Конечно, перебор можно писать по-разному. Например, можно написать процедуру, которая по номеру объекта построит сам объект, а потом в основном коде просто написать цикл по номерам объектов; объекты нумеровать можно, например, средствами динамического программирования (когда буду писать про ДП, напишу и про нумерацию объектов). Можно написать код, который по объекту будет создавать в некотором смысле следующий. Но наиболее общим и в большинстве случаев не многим более сложным в реализации (а чаще — намного более простым и не требующим дополнительных размышлений) является рекурсивный перебор, также (насколько я понимаю) называемый перебором с возвратом, backtracking. Помимо более простой реализации, он обладает рядом других достоинств: например, в нем возможно очень легко реализовывать различные отсечения и эвристики.

В тексте будут встречаться задания. К большинству из них в конце приведен ответ, а к некоторым — еще и подсказка. Кроме того, я думаю, что стоит написать и потестить все программы, которые я тут привожу. Только учтите, что перебор всегда работает очень долго, поэтому большие значения n, k и других параметров подсовывать, как правило, не стоит.

2.1. Элементарные примеры

Я тут долго думал, стоит сначала объяснять общую идею или все-таки сначала пример разбирать. Буду сначала пример, так, наверное, понятнее.

2.1.1. Перебор всех 2^k двоичных чисел из k разрядов

Примечание: Это, пожалуй, единственный пример, когда в некоторых случаях имеет смысл писать прямой перебор. Действительно, перебрать все k-значные двоичные числа можно легко: `for i:=0 to (1 shl k) - 1 do` — и i пробежит все k-значные двоичные числа. Это бывает полезно, например, в динамике по подмножествам или динамике по профилю, но нередко бывает полезнее рекурсивный перебор, который мы и будем тут разбирать.

Пусть нам надо, например, вывести на экран все k-значные двоичные числа (их всего, очевидно, 2^k). Напишем следующую программу

```
var a:array...

procedure check;
var i:integer;
begin
for i:=1 to k do
    write(a[i]);
writeln;
end;

procedure find(i:integer);
begin
if i>k then begin
   check;
   exit;
end;
a[i]:=0;
find(i+1);
a[i]:=1;
find(i+1);
end;

begin
readln(k);
```

и посмотрим, как она работает. Есть массив a, в котором будет накапливаться наше двоичное число, по одной цифре в элементе массива.

Процедура *check* делает то, что надо сделать с очередным найденным числом: выводит его на экран. Если бы надо было делать что-то ещё, она бы делала что-то ещё; более подробно я напишу ниже.

Основная часть программы — процедура $find$. Её «*основная идея*» — пусть у нас в массиве a первые $i-1$ элементов уже заполнены некоторыми двоичными цифрами, т.е. сформировано некоторое начала k-битового числа. Тогда результатом вызова $find(i)$ будет перебор всех возможных 2^{k-i+1} «концов» числа, т.е. вызов процедуры *check* для всех 2^{k-i+1} двоичных чисел с заданным началом. (У нас заполнены первые $i-1$ элементов, т.е. осталось заполнить $k-i+1$, потому и 2^{k-i+1} вариантов; i обозначает номер первой незаполненной позиции — позиции, откуда надо начинать заполнять массив a.)

2.1.2. Почему это работает?

Попробуем понять, как она работает (т.е. почему сказанное выше про неё верно). Можно понять это с нескольких сторон :)

Посмотрим с конца.

Вопервых, как работает $find(k+1)$. Видно, что она просто запускает *check* и выходит. Но именно это она и должна сделать в соответствии с «основной идеей». Действительно, вызов $find(k+1)$ обозначает, что первые $(k+1)-1$ элементов массива уже заполнены, т.е. заполнены *все* k элементов, т.е. уже сформировано очередное решение — осталось только вывести его на экран.

Рассмотрим теперь работу $find(k)$. Она сделает следующее: поставит в $a[k]$ цифру 0 и вызовет $find(k+1)$, которая (как мы только что видели) выведет массив на экран. Потом она поставит в $a[k]$ цифру 1 и опять вызовет $find(k+1)$, которая опять просто выведет решение на экран.

Это полностью соответствует «основной идее». Действительно, первые $k-1$ элементов массива уже должны быть заполнены, поэтому осталось только перебрать два варианта заполнения последней цифры и вывести оба на экран. Именно это она и делает.

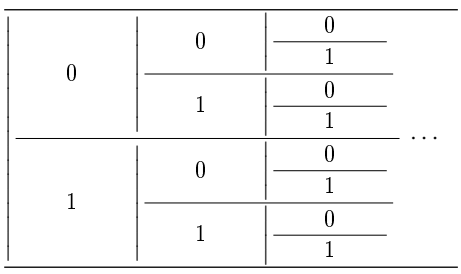
Посмотрим теперь $find(k-1)$. К её вызову первые $k-2$ элемента массива уже должны быть заполнены, поэтому осталось перебрать все варианты заполнения оставшихся двух элементов. Что сделает $find(k-1)$. Она поставит в $a[k-1]$ ноль и вызовет $find(k)$, которая, как мы видели, переберёт все возможные 1-циферные окончания и выведет их на экран. Далее, когда $find(k)$ отработает, произойдёт возврат в $find(k-1)$, она поставит в $a[k-1]$ единицу и опять запустит $find(k)$, которая переберёт все возможные 1-циферные окончания такого начала. Видно, что таким образом $find(k-1)$ переберёт все возможные 2-циферные окончания решения, сформированного перед её вызовом, т.е. отработает в соответствии с «основной идеей».

Теперь $find(k-2)$. Ей надо перебрать все возможные 3-циферные окончания. Она поставит ноль в $a[k-2]$ и вызовет $find(k-1)$, которая, как мы только что видели, действительно перебирает все 2-циферные окончания. После этого поставит единицу в $a[k-2]$ и опять вызовет $find(k-1)$. Поскольку все 3-циферные окончания — это либо ноль и 2-циферное окончание, либо единица и 2-циферное окончание, то вызов $find(k-2)$ действительно перебирает все 3-циферные окончания.

И так далее [на самом деле выше я просто тремя несколько различными способами описывал фактически одно и то же. Работа $find(k-2)$ ничем принципиально не отличается от $find(k-1)$ и т.п.].

Вообще, $find(i)$ надо перебрать все окончания от i-ой цифры до конца массива. Но все такие окончания — это либо ноль и окончание от $i+1$-ой до конца, либо единица и окончание от $i+1$-ой до конца. В соответствии с этим $find(i)$ и ставит в $a[i]$ ноль и вызывает $find(i+1)$, тем самым перебирая все окончания от $i+1$-ой цифры до конца, потом ставит в $a[i]$ единицу и опять вызывает $find(i+1)$.

Теперь ясно, что $find(1)$ перебирает все k-значные окончания пустого числа (т.е. числа, в котором ноль цифр), т.е. действительно решает задачу.



Посмотрим теперь на ту же работу с начала (в смысле, не с конца) [на самом деле то, что я тут пишу — это в некотором смысле тавтология. Я одно и тоже переписываю в разных вариантах, надеясь, что хотя бы одним вы проникнетесь :)].

Все двоичные числа можно представить в виде таблицы, приведённой слева: в первую очередь разделяем числа по первой цифре, во вторую очередь по второй и т.д. В соответствии с этим и работают процедуры $find$. Можно себе представить ось времени направленную вертикально вниз, с верхней границей таблицы — моментом запуска $find(1)$, нижней границей — концом запуска $find(1)$. Самая левая вертикальная черта отражает работу $find(1)$: она работает все время. Следующая вертикальная черта состоит из двух частей: они отражают работу $find(2)$. Процедура $find(2)$ будет запущена дважды ($find(1)$ запустит её дважды), потому две черты. Каждый запуск $find(2)$ запустит $find(3)$ два раза — итого четыре запуска $find(3)$, отражаемые четырьмя кусочками третьей вертикальной прямой. (все четыре копии будут работать одна за другой, а не одновременно, ведь вертикальная ось — это ось времени). Видно, что делает каждая процедура $find$: она ставит в соответствующую ячейку массива a ноль, потом один (цифры справа от вертикальной черты, соответствующей запуску процедуры), и для каждой цифры запускать процедуру $find$ «следующего уровня» (две вертикальные черты ещё правее). Видно и как будет в итоге меняться массив a: вначале в нем все нули, потом, начиная с правых цифр, в нем меняются нули на единицы и т.д., в конце — все единицы.

Наконец, ещё один вариант представления того, что происходит. Он, может, не так ясно разъясняет работу, но весьма полезен для понимания идей перебора вообще.

2.1.3. Дерево решений

Все множество решений (в нашем случае решения — это все k-битовые двоичные числа) можно представить в виде дерева, делая сначала разделение решений по первому биту, потом по второму и т.д.:

С этой точки зрения работа процедуры $find$ очень похожа на поиск в глубину по этому дереву (на самом деле, она и есть поиск в глубину). Мы сначала обходим левое поддерево корня, проходя ребро и рекурсивно запускаясь от левого сына корня, после окончания обхода обходим правое поддерево, проходя соответствующее ребро и рекурсивно запускаясь от правого сына. Представление о дереве решений нам будет очень полезно в дальнейшем.

Я надеюсь, что в этом месте вполне понятно, как работает процедура $find$.

2.1.4. О процедуре *check*

Обратите внимание, что на самом деле, как видно, нам совсем не важно, что делает процедура *check*. Эта процедура делает то, что нужно в данной конкретной задаче сделать с найденным решением (в нашем случае — с найденным k-битным числом): надо его вывести на экран — выведем, надо в файл сохранить — сохраним, надо проверку какую-нибудь сделать — сделаем и т.д. Для написания собственно *перебора* не важно, что она будет делать; основная задача перебора — поставлять процедуре *check* одно за другим решения. Но именно процедура *check* будет делать то, зачем мы делали перебор: считать такие объекты, или проверять, подходит ли объект под условие, или искать объект минимальной стоимости...

2.1.5. Общая идеология поиска

Итак, нам надо перебрать объекты из некоторого множества. Более конкретно — вызвать процедуру *check* для каждого объекта. Таким образом, основная задача перебора будет состоять в том, чтобы вызвать процедуру *check* для всех объектов из нашего множества.

Обычно объекты из множества можно задавать некоторым массивом, элементы которого принимают те или иные значения. В приведённом выше примере это был массив a — массив двоичных цифр; везде ниже я аналогичные массивы тоже буду обозначать a. Обычно перебрать все подходящие значения очередного элемента массива a легко; в приведённом выше примере каждый элемент массива a мог принимать два значения: ноль и один.

Тогда перебрать все объекты можно с помощью следующей процедуры:

```
procedure find(i:integer);
begin
if (выбраны все элементы, т.е. сформировано некоторое решение) then begin
   check;
   exit;
end;
Для каждого возможного значения a[i] begin
    a[i]:=это значение;
    find(i+1);
end;
end;
```

Комментарии:

1. Проверка на то, что решение сформировано. В простейшем случае это будет просто if $i > k$, как выше, но могут быть и более сложные варианты (например, если число элементов не фиксировано).
2. Цикл по возможным значениям $a[i]$. Опять-таки, в каждом конкретном случае, конечно, свой. Как правило, это будет цикл for, нередко с вложенным if, например,

```
for j:=1 to n do if (j может быть значением a[i]) then begin
    a[i]:=j;
    find(i+1);
end;
```

примеры будут ниже.

Эта процедура $find$ работает аналогично приведённому выше примеру (и вообще, все процедуры $find$ в переборе работают аналогично друг другу): считая, что начало из $i-1$ элемента фиксировано, перебирает все возможные окончания. Она смотрит, какой может быть i-й элемент, перебирает все его значения, и для каждого запускает рекурсивно $find(i+1)$, которая переберёт все окончания, считая первые i элементов фиксированными.

Процедура *check* делает то, что надо сделать с решением. В большинстве случаев это проверка, удовлетворяет ли найденное решение каким-либо требованиям (примеры см. ниже), поэтом так и названа. Как я уже много раз говорил, конкретный вид процедуры *check* нам не важен.

2.1.6. Перебор всех k-значных чисел в n-ичной системе счисления

(Всего таких чисел n^k)

(Зачем я все время привожу, сколько таких объектов: просто для того, чтобы вы могли лишний раз проверить, что вы понимаете, о каких объектах идёт речь: посчитайте сами в уме количество таких объектов и сравните; никакой больше нагрузки это не несёт.)

```
procedure find(i:integer);
var j:integer;
begin
if i>k then begin
   check;
   exit;
end;
for j:=0 to n-1 do begin
    a[i]:=j;
    find(i+1);
end;
end;
```

Я надеюсь, что работа этой процедуры если и не очевидна после всего вышеизложенного, то за несколько секунд становится понятной. Единственное отличие от примера 1 — то, что надо перебирать не 2 цифры, а n, и потому перебор делаем циклом.

2.1.7. Разложение числа N в степени двойки

Несколько притянутый за уши пример: по данному числу N определить, можно ли его представить в виде суммы k степеней двойки, не обязательно различных.

Примечание: Конечно, эту задачу, как и многие другие, которые мы тут будем обсуждать, вполне можно решать другими, более разумными, быстрыми и правильными методами, чем перебором, но мы тут будем обсуждать именно переборные решения в качестве иллюстрации общих концепций.

Будем перебирать все возможные наборы из k степеней двойки; соответственно, в массив a будем записывать последовательно эти степени.

```
procedure check;
var j:integer;
    s:longint;
begin
s:=0;
for j:=1 to k do
    s:=s+a[j];
if s=n then begin
   for j:=1 to k do
       write(a[j],' ');
   writeln;
end;
end;
```

```
procedure find(i:integer);
var j:integer;
begin
if i>k then begin
   check;
   exit;
end;
for j:=0 to 30 do begin
    a[i]:=1 shl j;
    find(i+1);
end;
end;
```

Во-первых, я ещё раз привожу текст процедуры *check*, чтобы вы видели, что она будет делать здесь (а она проверяет, подходит ли нам такое решение, и, если да, то выводит его на экран).

Во-вторых, обратите внимание на перебор всех степеней двойки циклом по j. Можно, конечно, этот перебор написать и по-другому, например так:

```
a[i]:=1;
while a[i]<1 shl 30 do begin
      find(i+1);
      a[i]:=a[i] shl 1;
end;
```

или типа того: не суть важно, как написать перебор, главное, правильно написать, не забыв ни одного варианта; в частности, обратите внимание, что этот вариант кода по сравнению с приведённым в процедуре *find* выше перебирает на одну степень двойки меньше. (**Контрольный вопрос 2.1:** *Видите, почему?*)

Я надеюсь, что в остальном идея работы процедуры понятна.

Задание 2.2: *Напишите эту программу (собственно, я надеюсь, что и предыдущие программы вы написали). Потестите её (обратите внимание, что тут время работы от n не зависит, только от k, потому имеет смысл брать и большие n). Найдите в ней баг и придумайте, как его исправить. Кроме того, заметьте, что одно и то же решение выводится несколько раз, отличаясь перестановкой слагаемых. Придумайте, как это исправить (может быть, вам поможет сначала почитать следующий пример, но лучше подумайте сначала, не читая примера дальше).*

2.1.8. Перебор всех сочетаний из n по k (т.е. всех C_n^k)

Хочется также в массив a записывать выбранные элементы. Но тут возникнут две проблемы: во-первых, надо, чтобы все элементы были различными, во-вторых, чтобы сочетания не повторялись из-за изменения порядка элементов (ведь $\{1,3\}$ и $\{3,1\}$ — это одно и то же сочетание).

Задание 2.3: *Можно, конечно, это проверять в процедуре check. Т.е. процедура find будет фактически работать по предыдущему примеру, а процедура check будет отбирать то, что нужно. Напишите такую программу. Обратите внимание на то, чтобы не брать одно и то же сочетание несколько раз.*

Обе проблемы решаются одной идеей: будем требовать, чтобы в массиве a элементы шли строго по возрастанию. Тогда получаем следующую процедуру $find$ (считаем, что элементы, из которых мы собираем сочетание, занумерованы от 0 до $n-1$):

```
procedure find(i:integer);
var j:integer;
begin
if i>k then begin
   check;
   exit;
end;
for j:=0 to n-1 do if j>a[i-1] then begin
    a[i]:=j;
    find(i+1);
end;
end;
```

Обратите внимание на нетривиальный for. Проверка гарантирует, что все элементы будут идти по возрастанию. На самом деле, очевидно, что весь for можно заменить на

```
for j:=a[i-1]+1 to n-1 do
```

без всяких if; так и надо писать, пример выше приведён скорее для того, чтобы вы поняли, как иногда бывает надо проверять дополнительные условия.

Кроме того, заметьте, что теперь не все ветви перебора заканчиваются формированием решения. Действительно, если, например, $k=3$, а мы на первом же уровне перебора (т.е. в $find(1)$) возьмём $a[1]=n-1$, то видно, что на втором уровне (т.е. в $find(2)$) нам будет нечего делать. Аналогично, если $k=3$, а на первом уровне берём $a[1]=n-2$, то на втором придётся взять $a[2]=n-1$ и на третьем делать нечего.

Задание 2.4: *а) Напишите эту программу. Обратите внимание на подготовку вызова $find(1)$; проверьте, что перебираются действительно* **все** *сочетания (например, выводя их в файл и проверяя при маленьких n и k).*

б) Добавьте в программу код, который выводит (на экран или в файл) «лог» работы рекурсии (например, выводя при присвоении $a[i]:=j$; на экран строку '`a[i]=j`', сдвинутую на i пробелов от левого края строки: вам этот вывод покажет, что́ на самом деле делает программа и пояснит предыдущий абзац); этот «лог» лучше выводить вперемешку с найденными решениями, чтобы видеть, какая ветка рекурсии чем закончилась. Подумайте над тем, как исправить то, что описано в предыдущем абзаце, т.е. как сделать так, чтобы каждая ветка рекурсии заканчивалась нахождением решения.

Замечу ещё, что в этой задаче можно написать процедуру $find$ немного по-другому. А именно, будем ей теперь передавать два параметра, i и x. Смысл параметра i тот же, что и раньше, а x обозначает, начиная с какого числа надо перебирать очередной элемент:

```
procedure find(i:integer;x:integer);
var j:integer;
begin
if i>k then begin
   check;
   exit;
end;
for j:=x to n-1 do
    a[i]:=j;
    find(i+1,j+1);
end;
end;
```

На самом деле тут x будет всегда равен $a[i-1]+1$, просто, может быть, такую процедуру проще понять.

Смысл процедуры $find$ теперь такой: перебрать все возможные окончания нашего сочетания, в которых все элементы не меньше, чем x.

Вообще, иногда и в других задачах имеет смысл передавать процедуре $find$ дополнительные параметры, которые так или иначе ограничивают область перебора очередного элемента, точнее, подсказывают, какие значения элемента стоит перебирать. Как правило, их (параметры) всегда можно выразить через уже сформированную часть решения, но иногда проще их передавать, чем каждый раз пересчитывать.

2.1.9. Перебор всех $n!$ перестановок из n чисел (от 1 до n)

Здесь из проблем, перечисленных в начале предыдущего примера, осталась одна: надо, чтобы все элементы перестановки были различными. Порядок же, наоборот, как раз таки важен, и поэтому такой приём, как в прошлом примере, здесь не пойдёт.

Поэтому применим другой приём, который весьма полезен бывает во многих задачах на перебор. А именно, введём второй глобальный массив, массив was, в котором будем фиксировать, использовали ли мы каждое число. Т.е. очередным элементом в перестановку будем ставить только те числа, которые ещё не были использованы. (Естественно, в массиве a будем хранить получающуюся перестановку).

```
var was:array...

procedure find(i:integer);
var j:integer;
begin
```

```
if i>n then begin
   check;
   exit;
end;
for j:=1 to n do if was[j]=0 then begin
    a[i]:=j;
    was[j]:=1;
    find(i+1);
    was[j]:=0;
end;
end;
```

Во-первых, тут у нас количество элементов в объекте, которое раньше было k, теперь равно n — общему количеству элементов, поэтому такое условие выхода из рекурсии.

Во-вторых, как собственно работает процедура $find(i)$. Она перебирает, какой элемент надо поставить на i-е место. Этот элемент не должен быть использован ранее (т.е. не должен уже стоять в массиве a), потому и проверка `if was[j]=0`. Далее, она ставит этот элемент в массив a, помечает, что он теперь использован и запускает $find(i+1)$ для перебора всех «хвостов» текущей перестановки. При этом переборе элемент j использован уже не будет, т.к. в $was[j]$ помечено, что он уже взят. Надеюсь, что работа процедуры понятна.

Задание (элементарное) 2.5: *Напишите программу перебора всех A_n^k — всех размещений из n по k (в них, в отличии от C_n^k, порядок важен).*

А теперь обратите особое внимание на строчку

```
    was[j]:=0;
```

в приведённом выше тексте. Обсуждению её мы посвятим почти всё оставшееся в текущей части время. Она является примером очень важной идеи, пожалуй, самого важного правила, которое есть при написании переборных программ. Именно несоблюдение этого правила (а точнее, забывание про него), на мой взгляд, является одним из основных источников ошибок в переборе, поэтому ВСЕГДА, КОГДА ПИШЕТЕ ПЕРЕБОР, ПОМНИТЕ ПРО НЕГО:

ПРОЦЕДУРА $find$ ДОЛЖНА ВСЕГДА ВОЗВРАЩАТЬ НАЗАД ВСЕ ИЗМЕНЕНИЯ, КОТОРЫЕ ОНА ПРОИЗВОДИТ (за небольшими исключениями, когда вы ЧЁТКО ОСОЗНАЕТЕ, почему некоторое изменение можно не возвращать назад), причём лучше всего возвращать назад изменения сразу после вызова $find(i+1)$.

Здесь процедура $find$ пометила, что элемент j использован. Строка

```
    was[j]:=0;
```

отыгрывает назад это изменение, что вполне логично, т.к. процедура $find(i+1)$ переберёт все окончания, у которых на i-м месте стоит j, и после этого мы будем перебирать другие варианты, в которых элемент j больше (пока) не используется. Очевидно, что, если бы этой строки не было, это привело бы к глобальным ошибкам в работе программы. Если вам это не очевидно, то тщательно продумайте этот момент; это важно и на самом деле это показывает, насколько хорошо вы понимаете работу перебора. Если никак не можете понять, в чем дело, вспомните аргументацию раздела 2.1.2, и промоделируйте аналогично работу в этом случае.

Другие программы могут делать изменения в других (глобальных) переменных; примеры будут потом. И всегда надо тщательно проверить, что откат назад происходит. В простых случаях поможет просто вручную изменять значения назад, как в примере выше. В более сложных случаях может быть не так просто отыграть все изменения. В таком случае может помочь сохранение старых переменных в стеке процедуры и восстановление их целиком, например

```
type tWas=array...
var was:tWas;

procedure find(i:integer);
var j:integer;
    oWas:tWas;  {old was}
begin
if i>n then begin
   check;
   exit;
end;
oWas:=was; {сохраняем старый массив}
for j:=1 to n do if was[j]=0 then begin
    a[i]:=j;
    was[j]:=1;
    find(i+1);
    was:=oWas; {восстанавливаем его}
end;
end;
```

Один из минусов этого подхода — то, что довольно активно расходуется память в стеке, но зато не надо тщательно следить за всеми изменениями, которые делает $find$, и не надо думать, какой же командой надо откатить изменения (здесь это было очевидно, но могут быть более сложные случаи).

Обратите внимание вот ещё на что: кажется, что эту же процедуру можно написать по-другому, так, чтобы она восстанавливала массив was *до* работы:

```
procedure find(i:integer);
var j:integer;
    oWas:tWas; {old was}
begin
if i>n then begin
   check;
   exit;
end;
oWas:=was; {сохраняем старый массив}
for j:=1 to n do begin
    was:=oWas; {восстанавливаем}
    if was[j]=0 then begin
       a[i]:=j;
       was[j]:=1;
       find(i+1);
    end;
end;
end;
```

Но не очевидно, что этот вариант будет работать, т.к. последнее изменение не будет «откачено», и после окончания процедуры *find* массив was будет не таким, каким он был раньше (на самом деле его тут же исправит восстановление массива на уровень выше, но как минимум не очевидно, что это будет работать, надо думать). Поэтому старайтесь восстанавливать все изменения как можно раньше.

Кстати, ещё обратите внимание: *все* программы, которые мы до сих пор писали, изменяют массив a и *не* откатывают изменения. Поймите, почему *это* не страшно.

И, наконец, последнее замечание в этой части. В Borland Pascal есть своеобразный баг при отладке рекурсивных процедур. А именно, как известно, в BP есть следующие четыре основных клавиши, управляющие работой программ при отладке:

- F9: выполнять программу до конца или до ближайшего breakpoint,
- F8: выполнить текущую строку,
- F7: если в текущей строке нет вызовов функций и процедур, кроме стандартных, то выполнить текущую строку (то же, что и F8), иначе войти в отладку вызова функции, присутствующего в данной строке.
- F4: Run to cursor: выполнять программу, пока она не дойдёт до выполнения строки, на которой стоит курсор.

Так вот, в BP клавиша F8 действует на самом деле примерно как F4 по следующей строке: она не выполняет текущую строку *полностью*, а выполняет программу до тех пор, пока впервые не станет выполняться следующая строка кода. Если в программе нет рекурсивных вызовов, эти два варианта равносильны, а вот если рекурсия есть, то все хуже. Пример: наша любимая процедура *find*

```
procedure find(i:integer);
begin
...
  ...
  find(i+1);
  ...
end;
```

Если на строке $find(i+1)$; вы нажмёте F8, то программа остановится не тогда, когда эта $find(i+1)$ отработает полностью, а когда *какой-нибудь* вызов *find* с этой же строки отработает. Как правило, это будет глубоко в рекурсии. Например, вы нажали F8 на этой строке при $i = 1$ — программа остановится на следующей строке, но не при $i = 1$, а (скорее всего) типа при $i = k$ и т.п. (т.е. когда она *впервые* дойдёт до строки, следующей за $find(i+1)$;). Это может оказаться очень неожиданно, т.к. у вас сразу меняются значения i и всех остальных переменных, причём нет так, как вы ожидали, но, если помнить об этой особенности, то ничего неожиданного нет. Но в таком случае отладка рекурсивных программ становится весьма нетривиальной. Чтобы сделать «настоящее» F8, т.е. отработать этот вызов полностью, приходится на следующей строке ставить breakpoint с условием $i =<$ тому, что надо $>$, и жать F9.

Ещё раз замечу, что это относится не только к перебору, но к любым рекурсивным процедурам.

В Delphi (и FP?) этого бага вроде нет.

2.1.10. Совсем общая концепция перебора

Все задачи до сих пор у нас в основном крутились вокруг некоторого массива a, который мы последовательно заполняли. Действительно, очень многие задачи, решаемые рекурсивным перебором, можно представить именно так — как задачу перебора возможных заполнений некоторого массива a.

Но перебор, на самом деле, намного более мощная идея. Пусть у нас есть задача, в которой нам надо перебрать набор решений, а каждое решение образуется некоторой последовательностью «элементарных» шагов. То есть пусть мы можем говорить о каких-то «состояниях», «позициях» в этой задаче, из каждого состояния/позиции есть набор «ходов» в другие позиции, и нам надо найти последовательность ходов, приводящую к требуемой «конечной» позиции (или посчитать, сколько таких последовательностей есть, или найти оптимальную из них и т.д.) При этом будем считать, что у нас нет зацикливаний: мы не можем из одной позиции сделать несколько ходов и вернуться в нее же.

Тогда эта задача несложно решается перебором. Процедура `find` будет работать так: она будет считать, что у нас уже сформирована некоторая позиция. Процедура будет перебирать все возможные ходы из этой позиции, и рекурсивно запускать себя из полученных позиций.

Простейший пример — карточный пасьянс типа косынки. У нас есть текущая позиция (не будем сейчас обсуждать, как ее представить в программе; будем также считать, что мы знаем все закрытые карты, иначе ответ не определен). Мы хотим определить, сойдется ли пасьянс, т.е. есть ли такая последовательность наших действий, при которой пасьянс сходится.

Если бы в каждый момент у нас был бы лишь один возможный ход, то задача была бы простой: мы просто делали бы эти ходы и посмотрели бы на результат.

Но в «косынке» из каждой позиции у нас может быть несколько ходов. Поэтому процедура `find` будет работать так: по данной позиции она будет перебирать все возможные ходы и рекурсивно запускаться для поиска дальнейшего решения.

```
procedure find;
begin
if ходов нет then
    check; // процедура проверит, сошелся ли пасьянс
    exit;
end;
for все возможные ходы do begin
    сделать ход
    find
    откатить ход назад (!)
end;
end;
```

Еще пример — крестики-нолики на поле 3×3. Пусть нам надо написать программу, которая будет искать оптимальный в некотором смысле ход из данной позиции. Для простоты оптимальность определим так: оптимальным будем называть такой ход, после которого мы точно сможем выиграть независимо от ходов противника. Если таких ходов несколько, выберем любой из них. Если таких ходов нет, но есть ходы, гарантирующие нам ничью, то выберем любой из ничейных ходов. Если же все ходы ведут к нашему проигрышу (при условии идеального соперника), то сообщим об этом.

(Отмечу, что «оптимальность» хода можно было бы определить и сложнее, например, попытаться как-то учесть возможность противнику ошибиться. Но мы так усложнять не будем.)

Для этого просто переберем все возможные способы развития партии, начиная с некоторой позиции. Теперь у нас будет не процедура `find`, а функция. Она будет принимать в качестве параметра, кто (крестики или нолики) ходят сейчас и будет возвращать, кто выигрывает при идеальной игре обоих соперников. Код будет примерно такой:

```
function find(player:integer):integer; // player=-1 -- нолики, player=1 --- крестики
begin
проверить, не окончена ли игра (т.е. выиграл ли уже кто-то и не заполнено ли поле)
if игра окончена then begin
   if крестики выиграли then result:=1
   else if нолики выиграли then result:=-1
   else result:=0; // ничья
   exit;
end;
// переменная optimal хранит номер выигрывающего игрока (-1, 0 или 1)
// изначально худший для нас вариант --- выигрывает противник
optimal:=-player; // -player как раз дает противника
for i:=1 to 3 do
    for j:=1 to 3 do if клетка (i,j) свободна then begin
        сходим в клетку (i,j)
        winner:=find(-player); // рекурсивно переберем дальнейшие варианты
                               // и узнаем, кто выигрывает
        if player=1 then
            if winner>optimal then // для крестиков мы хотим
                optimal:=winner;  // номер выигрывающего игрока как можно больше
                  // т.е. крестики лучше ничьей, а ничья лучше ноликов
        else // player=-1 --- нолики
            if winner<optimal then // для ноликов мы хотим
                optimal:=winner;  // номер выигрывающего игрока как можно меньше
                  // т.е. нолики лучше ничьей, а ничья лучше крестиков

        отменим ход в (i,j) // откатимся!!
    end;
// теперь optimal --- выигрывающий игрок при самом лучшем нашем ходе
result:=optimal;
end;
```

Задание 2.6: *(Сложное) Напишите эту программу полностью и доведите ее до такого состояния, чтобы можно было играть с компьютером в крестики-нолики.*

Так можно решать практически любую игру, в которой не бывает зацикливаний. (А на самом деле если зацикливания возможны, то первый вопрос — а что происходит в реальной игре в таким случае? Ведь вряд ли игра на самом деле будет продолжаться до бесконечности? В шахматах, например, при трехкратном повторении позиции объявляется ничья, поэтому зацикливания невозможны, просто надо хранить все позиции, которые уже встречались.)

Правда, конечно, есть проблема — если игра сложная, с множеством ходов и длинными партиями, то времени перебрать все возможные партии не хватит. Например, в шахматах таким перебором решаются разве что малофигурные эндшпили.

Еще пример такой задачи:

Задание 2.7: *(Задача 159 с* `informatics.mccme.ru`*) Радиолюбитель Петя решил собрать детекторный приемник. Для этого ему понадобился конденсатор емкостью C мкФ. В распоряжении Пети есть набор из n конденсаторов, емкости которых равны c_1, c_2, ..., c_n, соответственно. Петя помнит, как вычисляется емкость параллельного соединения двух конденсаторов ($C_{new} = C_1 + C_2$) и последовательного соединения двух конденсаторов ($C_{new} = C1 \cdot C2/(C1 + C2)$). Петя хочет спаять некоторую последовательно-параллельную схему из имеющегося набора конденсаторов, такую, что ее емкость ближе всего к искомой (то есть абсолютная величина разности значений минимальна). Разумеется, Петя не обязан использовать для изготовления схемы все конденсаторы.*

Напомним определение последовательно-параллельной схемы. Схема, составленная из одного конденсатора, – последовательно-параллельная схема. Любая схема, полученная последовательным соединением двух последовательно-параллельных схем, – последовательно-параллельная, а также любая схема, полученная параллельным соединением двух последовательно-параллельных схем, – последовательно-параллельная. Обратите внимание, что это определение не допускает произвольные схемы, а только полученные именно последовательностью параллельных или последовательных соединений.

2.1.11. Задачи к части 2.1

Я надеюсь, что вы решите одну-две задачи и хотя бы *серьёзно* (хотя бы день) подумаете над остальными (или решите их), прежде чем переходить к части 2.2. Часть из (нормальных переборных) решений этих задачи использует идеи, про которые я буду рассказывать в части 2.2, но будет неплохо, если вы додумаетесь до них сами :) хотя бы в этих задачах (а я буду стараться рассказывать для общего случая), или если напишете что-то хоть и корявое, но работающее.

До того, как переходить к части 2.2, порешайте эти задачи. Точнее, сначала убедитесь, что материал части 2.1 у вас «осел» в голове, и что вы часть 2.1 понимаете (а для этого прорешайте задачи из текста части 2.1), потом решайте задачи. Если не решите (подумав над задачами хотя бы некоторое время, день-два), смотрите подсказки. Попробуйте учесть их и подумать над задачами ещё. Потом разберите решения. Может быть, последние три задачи вам покажутся нетривиальными — ну хотя бы попробуйте их решать...

Задание 2.8: *Напишите программу перебора всех последовательностей из 0 и 1 без k нулей подряд, в которых всего n символов. (Например, при $k = 2$ и $n = 3$ это будут последовательности 010, 011, 101, 110 и 111). Основной задачей программы будет посчитать, сколько таких последовательностей всего, но имеет смысл выводить их на экран (или в файл) для проверки.*

а) Напишите эту программу, модифицировав пример 1, т.е перебирая ВСЕ *последовательности из 0 и 1 длины n, и проверяя, что последовательность «правильная», только в процедуре check.*

б) Напишите программу, которая будет перебирать ТОЛЬКО *такие последовательности, т.е. чтобы* КАЖДАЯ *ветка перебора заканчивалась нахождением решения, и в процедуре check проверки не были бы нужны.*

в) (дополнительный пункт, не имеющий отношения к перебору) Если вы раньше не сталкивались с такой задачей, то попробуйте найти закономерность ответов при фиксированном k (т.е. сначала посмотрите на ответы на задачу при $k = 2$ и найдите в них закономерность, потом поищите закономерность при $k = 3$, потом при $k = 4$ и т.д.) Кстати, не забудьте, что тестить имеет смысл и очевидный случай $k = 1$:)

Задание 2.9: *Паросочетание в произвольном графе. Рассмотрим граф с $2N$ (т.е. чётным) количеством вершин. Паросочетанием в нем назовём набор из N рёбер, у которых все концы различны (т.е. каждая вершина соединена ровно с одной другой: разбиение вершин на пары). [В олимпиадном программировании обычно рассматривается только паросочетание в* ДВУДОЛЬНОМ *графе, т.к. там есть простой эффективный алгоритм. Но у нас граф будет произвольным и мы будем решать задачу перебором]. [Т.е. смысл этой задачи на самом деле — чтобы вы умели перебирать все разбиения на пары]*

а) Напишите программу, которая будет перебирать все разбиения вершин на пары и проверять, является ли такое разбиение паросочетанием (т.е. все ли нужные ребра присутствуют в нашем графе).

б) Считая, что граф полный и взвешенный, напишите программу, которая найдёт паросочетание наименьшего веса.

Задание 2.10: *Напишите программу перебора всех разложений числа N на натуральные слагаемые.*

Вариант 1: ровно на k слагаемых

а) считая, что слагаемые могут повторяться и что порядок слагаемых важен (т.е. что $2+1$ и $1+2$ — это разные решения);

б) считая, что порядок слагаемых не важен, т.е. выводя только одно разложение из разложений $2 + 1$ и $1 + 2$, при этом допуская одинаковые слагаемые;

в) считая, что все слагаемые должны быть различны, при этом порядок слагаемых не важен.

Вариант 2: на сколько угодно слагаемых в тех же трёх подвариантах (а, б и в)

Написав программы, прежде чем тестировать их, ответьте в уме на такой вопрос: ваша (уже написанная!) программа в варианте а) будет при $n = 3$ выводить решения $1+2$ и $2+1$. А при $n = 2$ она будет выводить $1+1$ один раз или два раза (во второй раз как будто переставив единички)?

Задание 2.11: *Задача «Числа». Дана последовательность из N чисел. За одно действие разрешается удалить любое число (кроме крайних), заплатив за это штраф, равный произведению этого числа на сумму двух его соседей. Требуется удалить все числа (кроме двух крайних) с минимальным суммарным штрафом.*

У этой задачи есть (не самое тривиальное) динамическое решение, но напишите переборное решение. Тут надо перебрать все варианты удаления чисел и выбирать из них тот, который даст минимальный штраф.

Задание 2.12: *(Какая-то довольно искусственная задача, но хорошо подходит для иллюстрации одной из идей далее). Посчитать количество последовательностей из m нулей и n единиц, удовлетворяющих следующих условиям. Первое условие: никакие две единицы не должны стоять рядом. Таким образом единицы делят последовательность на несколько групп из подряд идущих нулей. Второе условие: количество нулей в последовательных группах должно неубывать, и при этом в соседних группах должно отличаться не более чем на 1. Эта задача имеет динамическое решение, но напишите перебор.*

2.2. Отсечения

Как правило, переборное решение, написанное тупо, делает кучу лишней работы. А именно:

- если нам надо найти объект, который удовлетворяет определённым требованиям, то перебор находит ещё кучу объектов, которые этому требованию не удовлетворяют (например, в задаче о разложении N на k степеней двойки находятся куча наборов из k степеней двойки, которые в сумме не дают N)
- если надо найти объект с минимальной стоимостью и т.п., то находятся куча объектов, у которых сумма явно не минимальная.
- если надо посчитать некоторые объекты, то тоже возможно, что много веток рекурсии не будут приводить к решению.

Очевидно, что, заставив перебор не делать хотя бы часть этой работы, можно его резко ускорить. Т.е. мы хотим заставить перебор не перебирать те ветки дерева решений, которые нам точно не интересны.

2.2.1. Разложение числа N в сумму k степеней двойки

У нас была такая процедура:

```
procedure find(i:integer);
var j:integer;
begin
if i>k then begin
   check;
   exit;
end;
for j:=0 to 30 do begin
    a[i]:=1 shl j;
    find(i+1);
end;
end;
```

Попробуем какие-нибудь ветки дерева решений поотсекать. Т.е. фактически надо, чтобы $find$ не рассматривало некоторые варианты $a[i]$, которые нам точно не интересны.

Во-первых, очевидно, что не стоит рассматривать $a[i]$, если оно больше n. Более того, если ввести глобальную переменную s, в которой хранить текущую сумму, то можно рассматривать только $a[i]$, которые не превосходят $n-s$:

```
var s:...
procedure find(i:integer);
var j:integer;
begin
if ....
end;
for j:=0 to 30 do if 1 shl j<=n-s then begin
    a[i]:=1 shl j;
    s:=s+a[i];
    find(i+1);
    s:=s-a[i]; {Откатываем назад!!!}
end;
end;
```

Очевидно, что это тоже будет работать корректно (кстати, это ещё и исправляет баг с переполнением в процедуре *check*, который обсуждался в части I).

Заметьте, что можно это написать и по-другому:

```
var s:...
procedure find(i:integer);
var j:integer;
begin
if i>k...
end;
if s>=n then
   exit;
for j:=0 to 30 do  begin
    a[i]:=1 shl j;
    s:=s+a[i];
    find(i+1);
    s:=s-a[i]; {Откатываем назад!!!}
end;
end;
```

т.е. вынести проверку в начало процедуры $find$. Теперь, если в некоторый момент мы взяли $a[i] > n - s$, то мы попробуем такой вариант, войдём в $find(i+1)$, но тут же выйдем назад, т.к. $s \geqslant n$. Обратите внимание, что проверка стала нестрогой ($\geqslant$, а не $>$), т.к. я её переместил после проверки `if i>k`.

(Этот вариант, в отличии от предыдущего, видимо, не будет решать бага с переполнением, но баг с переполнением — это особая фича именно этой задачи; в других задачах таких приколов может не быть).

Задание 2.13: *Переделайте эту программу, чтобы не хранить глобальную переменную s, а передавать s (или $n - s$, т.е. оставшееся число) через параметр процедуры.*

Вообще мне кажется более естественным проводить отсечения в начале процедуры $find$ (как правило, после проверки на выход из рекурсии), как в последнем примере. Типа процедура $find$ сначала оценит, сто́ит ли вообще возится с дальнейшим перебором: если не стоит (в данном случае если $s \geqslant n$), то $exit$, иначе перебираем все подряд,

Попробуем поотсекать дальше. Например, очевидно, что в начале $find(i)$ у нас ещё $k-i+1$ мест не заполнены. На них встанут как минимум единицы (т.е. 2^0), поэтому, если $s+k-i+1 > n$, то тоже можно отсекать. Вообще, обычно отсечения можно проводить почти что до бесконечности :) придумывая все более и более точные критерии того, что решения не найдётся, и, если на олимпиаде делать нечего, то можно над этим и думать. Главное, нигде не наглючить, т.к. сложность программы с увеличением количества отсечений возрастает, и соответственно возрастает вероятность ошибки, а вот скорость работы программы может и не сильно увеличиваться.

2.2.2. Виды отсечений

Какие обычно бывают отсечения:

- Если задача программы — найти оптимальный объект (объект с минимальной стоимостью и т.п.), то обычно можно оценить, какая минимальная стоимость будет у объекта, когда мы его достроим, исходя из текущего начала объекта (например, часто она больше текущей стоимости). Если эта «оценка снизу» на стоимость достроенного объекта уже больше лучшей из полученных на данный момент стоимостей, то дальше искать бессмысленно (см. пример дальше).
- Если задача программы — получить объект с определёнными свойствами, то если очевидно, что это свойство точно уже не выполнить при данном начале, то дальше искать бессмысленно (как в примере выше: если $s > n$, то искать дальше точно бессмысленно).
- Если же задача программы — посчитать количество таких объектов, то здесь все сложнее. Смысл отсечения тут разве что в том, чтобы каждая ветка рекурсии заканчивалась нахождением решения. Пример будет ниже.
- Сразу замечу про ещё одно важное отсечение: отсечение по времени. Про него тоже скажу ниже.

Пример по второму пункту мы разобрали; разберём примеры по двум оставшимся пунктам.

2.2.3. Пример на отсечения при подсчёте количества объектов

Пусть цель программы — посчитать объекты. Рассмотрим в качестве примера задание 2.12 (сначала попробуйте сами его порешать!) Конечно, как и предлагалось в подсказке, будем перебирать разбиения нулей на группы. Будем решать задачу: сколькими способами можно разбить m нулей на nn групп так, чтобы в последовательных группах количества нулей или совпадали, или увеличивались на единицу.

Во-первых, тут $i = 1$ — особый случай (см. ещё ниже). Когда мы выбираем, сколько нулей у нас будет в первой группе, нет никаких ограничений. А когда выбираем, сколько нулей в дальнейших группах, у нас только два варианта: столько же, как и в прошлой группе, и на единицу больше. Будем передавать в процедуру количество нулей, которые осталось разбить по группам, чтобы было удобнее. Получаем следующий код (дополнительные комментарии см. ниже):

```
procedure find(i,k:integer);
var j:integer;
    prev:integer;
begin
if i>nn then begin
   check;
   exit;
end;
if k<=0 then
   exit;
if i=1 then begin
   for j:=1 to k do begin
       a[1]:=j;
       find(2,k-j);
   end;
end else begin
    prev:=a[i-1];
    a[i]:=prev;
    find(i+1,k-a[i]);
    a[i]:=prev+1;
    find(i+1,k-a[i]);
end;
end;
```

Подумаем, как можно отсечь. По сути, наша цель — чтобы каждая ветка заканчивалась нахождением решения. Подумаем, как может получиться так, что решение не найдётся. Могут быть два варианта: либо у нас нулей слишком мало осталось, либо слишком много. Что значит слишком мало — значит, что, даже если расходовать их в группы по минимуму, нулей не хватит. Групп у нас остаётся ещё $nn - i + 1$, в каждую надо как минимум $prev$ нулей, поэтому, если $k < prev \cdot (nn - i + 1)$, то решений точно не найдётся. Аналогично, что значит, что нулей слишком много? В первую группу мы можем поставить максимум $prev + 1$ нуль, во вторую — $prev + 2$ и т.д. В $nn - i + 1$-ую — $prev + nn - i + 1$, тогда общее количество нулей (сумма арифметической прогрессии) получится $(prev + 1 + prev + nn - i + 1) \cdot (nn - i + 1)/2$. Поэтому, если $k > (prev + 1 + prev + nn - i + 1) \cdot (nn - i + 1)/2$, то решений тоже точно не найдётся. Поэтому получаем отсечение

```
if (k<prev*(nn-i+1))or (k>(prev+1+prev+nn-i+1)*(nn-i+1) div 2) then
   exit;
```

и окончательное решение (привожу на этот раз полную программу):

```
{$A+,B-,D+,E+,F-,G-,I+,L+,N+,O-,P-,Q+,R+,S+,T-,V-,X+,Y+}
{$M 65520,0,655360}
var a:array[1..100] of integer;
    n,m:integer;
    nn:integer;
    ans:longint;
    res:longint;

procedure check;
var i:integer;
    s:integer;
begin
s:=0;
for i:=1 to nn do
    s:=s+a[i];
if s<>m then
   exit;
for i:=2 to nn do
    {на всякий случай проверим, что все правильно.
    На самом деле это никогда не должно сработать}
    if (a[i]<>a[i-1])and(a[i]<>a[i-1]+1) then begin
       writeln('!!2');
       halt;
    end;
inc(ans);
end;

procedure find(i,k:integer);
      {k --- сколько нулей осталось разбить по группам}
var j:integer;
    prev:integer;
begin
if i>nn then begin
   check;
   exit;
end;
if k<=0 then begin
       {если нулей не осталось,
        то бессмысленно что-либо перебирать.
        Обратите внимание, что это
        написано после предыдущего if'а.}
   exit;
end;
if i=1 then begin
      {i=1 здесь особый случай, т.к. у него
      нет предыдущей группы.
      Как это сделать поэлегантнее, я не придумал}
   for j:=1 to k do begin
       a[1]:=j;
       find(2,k-j);
   end;
end else begin
    prev:=a[i-1];
    if (k<prev*(nn-i+1))or
         (k>(prev+1+prev+nn-i+1)*(nn-i+1) div 2) then
       exit;
    a[i]:=prev;
    find(i+1,k-a[i]);  {k-a[i] нулей осталось разбить}
    a[i]:=prev+1;
    find(i+1,k-a[i]);
end;
end;

function count(n,m:integer):longint;
begin
ans:=0;
nn:=n;
if n>0 then
   find(1,m);
count:=ans;
end;

begin
read(n,m);
res:=count(n-1,m)+2*count(n,m)+count(n+1,m);
{если n=1, то в count передастся n-1=0.
Для этого и стоит проверка if n>0 в функции count}
writeln(res);
end.
```

Можете потестить это решение. На тесте, на котором самый большой ответ при ограничениях $n, m \leqslant 100$ (видимо, $n = 27$, $m = 100$) оно у меня работает секунды три, при том, что динамика там же работает немногим быстрее. Если же отсечение убрать, то тормозит много сильнее.

А теперь самое важное тут: обратите внимание, что теперь ЛЮБАЯ ВЕТКА ПЕРЕБОРА (за исключением, возможно, небольшого их числа, у которых отсечение произошло бы на последнем шаге) ЗАКАНЧИВАЕТСЯ НАХОЖДЕНИЕМ РЕШЕНИЯ. Следовательно, мы можем оценить, сколько всего веток перебора будет: у дерева решений листьев будет примерно столько же, сколько мы найдём решений, т.е. равно ответу на задачу. Очевидно, что, поскольку мы тут не умеем считать объекты пачками, т.е. мы каждый объект (разбиение) считаем отдельно, то быстрее работать вряд ли получится: на каждый объект нужен лист дерева решений, т.е. листьев должно быть не меньше, чем ответ на задачу (с другой стороны то же самое: процедура *check* делает $inc(ans)$, следовательно, она должны будет быть вызвана как минимум столько раз, каков ответ на задачу. Могло оказаться, что она вызвана будет намного большее количество раз, но в нашей программе это не так: мы специально сделали, чтобы каждый вызов *check* делал $inc(ans)$; ладно, почти каждый. Ещё меньше вызовов *check* сделать в рамках решений, которые делают только $inc(ans)$, невозможно).

Количество листьев приблизительно характеризует время работы программы: понятно, что, чем их больше, тем дольше работает программа. Поэтому всегда надо стараться уменьшить число листьев. Но тут мы уменьшили их до минимума: меньше, чем ответ на задачу, не получится. Таким образом, быстрее написать перебор, видимо, тут не получится. Мы оптимизировали дерево решений как могли. (Не *программу*, а *дерево решений*. Решение, может быть, можно оптимизировать ещё. Например, придумать, как убрать цикл из процедуры check; может быть, избавиться от деления на 2 в отсечении, и т.п. Но *дерево решений* все равно уже не изменится).

Кроме того, можно время, которое работает наша программа, теперь можно оценить по ответу на тест (ведь процедура *check* будет именно столько раз вызываться — ну, почти столько); если ответ небольшой, то можно рассчитывать, что наша программа тормозить не будет. Идея переборного решения этой задачи родилась у меня, когда я узнал, что максимальный ответ в тестах был пятизначным. Стало ясно, что такой перебор тормозить не может.

Задание 2.14: *(Творческое) Попробуйте придумать, как бы написать программу, чтобы не нужно было выделять $i = 1$ в особый случай. Это не очень тривиально, и есть несколько вариантов, как это сделать.*

Общепрограммистский (т.е. не относящийся именно к перебору) комментарий: Всегда старайтесь все делать как можно проще. Особые случаи — это то, что очень сильно усложняет программу, поэтому всегда старайтесь придумать, как бы обойтись без особых случаев. Кроме того, особые случаи — это первый признак того, что решение у вас неправильное. Т.е. если у вас в программе появляется особый случай, то задайтесь вопросом: чем этот

случай действительно особый? Почему его не получается обработать общим случаем? Нет ли ещё аналогичных особых случаев? (собственно, именно наличие уверенного и обоснованного ответа на последний вопрос и обозначает, что вы поняли, почему этот случай особый) Может быть, есть другие особые случаи, причём их много — проще говоря, надо искать другой алгоритм для *общего* случая, т.е. ваше текущее решение неправильное? (На самом деле имеет смысл задавать вообще ещё более общий вопрос: зачем написана *каждая* строчка кода, каждый цикл, каждый if? Почему без них нельзя обойтись?) И даже если вы поняли, чем этот случай действительно особый, попробуйте его все-таки свести к общему случаю (см. пример в следующем абзаце); правда, не переусердствуйте; эта задача — плохой пример, здесь сведение слишком сложное и, может быть, проще оставить все как было. Если в результате сведения все получается только сложнее, то, может быть, и не надо избавляться от особого случая.

В данной задаче вроде ясно, почему случай $i = 1$ особый: в первую группу мы можем поставить сколько угодно нулей, в то время как во вторую и дальнейшие — либо столько же, сколько и в предыдущей, либо на единицу больше. Но это не оправдание до конца. Во многих задачах удаётся и в такой ситуации свести частный случай к общему, например, введением нулевых элементов (сравните с осуществлением требования, чтобы в переборе всех C_n^k и т.п. все элементы возрастали: случай $i = 1$ там обрабатывается в общем порядке, несмотря на то, что у него нет предыдущего элемента. Ну и что, а мы сделали ему предыдущий элемент, $a[0]$, который всегда меньше всего, что может быть). Но в этой задаче я не смог придумать, как бы поэлегантнее избежать выделения $i = 1$ в особый случай. Конечно, можно убрать этот особый случай из процедуры $find$, но придётся наворачивать кучу кода в других местах... В общем, видимо, проще программу сделать у меня не получается. Но вдруг у вас получится? Или хотя бы попробуйте сделать, чтобы действительно понять, в чем тут трудности.

2.2.4. Отсечения в задачах на оптимизацию

Пусть цель программы — найти оптимальный объект. Рассмотрим в качестве примера задание 2.11 про удаление чисел со штрафом.

(Ещё раз замечу, что на самом деле многие задачи, которые мы тут обсуждаем, решаются более крутыми способами. Например, эта задача решается динамикой. Но тут мы обсуждаем, как их можно было решать перебором.)

Итак, наша программа будет перебирать все возможные последовательности удаления чисел, и для каждой считать штраф. Процедура *check* будет проверять, верно ли, что штраф меньше оптимального, и, если да, то запоминать текущее решение. Напишем программу следующим образом:

```
var a,b,ans:array...
    nn:integer;
    cur,best:longint;

procedure insert(i:integer;v:integer);
var j:integer;
begin
for j:=nn downto i do
    b[j+1]:=b[j];
b[i]:=v;
inc(nn);
end;

function delete(i:integer):integer;
var j:integer;
begin
delete:=b[i];
for j:=i+1 to nn do
    b[i-1]:=b[i];
dec(nn);
end;

procedure check;
begin
if cur<best then begin
   best:=cur;
   ans:=a;
end;
end;

procedure find(i:integer);
var j:integer;
    x:integer;
begin
if nn=2 then begin
   check;
   exit;
end;
for j:=2 to nn-1 do begin
    a[i]:=j;
    cur:=cur+b[j]*(b[j-1]+b[j+1]);
    x:=delete(j);
    find(i+1);
    insert(j,x);
    cur:=cur-b[j]*(b[j-1]+b[j+1]);
end;
end;
```

Поясню. У нас есть массив a, в котором, как всегда, мы храним текущее решение. В данном случае — последовательность номеров удаляемых чисел. Ans — наилучшее найденное на данный момент решение. Cur — текущий штраф (за те удаления, которые мы уже сделали), $best$ — штраф в наилучшем найденном к данному моменту решении. В массиве b мы храним текущие числа, nn — их количество.

Процедура *delete* удаляет число из массива b, корректируя nn, и возвращает удалённое число. Процедура *insert* отыгрывает удаление числа: вставляет его назад. На самом деле лучше было бы работать со связными списками, где удалить и вставить число можно намного быстрее (т.к. циклы в *insert* и *delete* сильно тормозят программу), но, чтобы не загромождать основные идеи, в этом примере я буду писать так. Процедура *check* просто проверяет, лучшее это решение, или нет.

Процедура $find$ — основная процедура перебора. Работает она так. Во-первых, если в массиве осталось всего 2 элемента ($nn = 2$; на самом деле, очевидно, это условие равносильно $i = n - 2$), то делать больше ничего не надо (удалять надо все числа, кроме крайних), поэтому *check* и *exit*.

В противном случае посмотрим, какое число будем удалять. Запомним его номер в массиве a, скорректируем текущий штраф cur и текущий массив b (последнее — вызовом *delete*), и пойдём перебирать дальше: $find(i + 1)$. После этого не забудем вернуть все назад!

Надеюсь, что вам понятно, как работает эта программа. Пара замечаний:

- Здесь процесс «отката» изменений весьма нетривиален. Можно было сохранить cur и b в стеке:

```
procedure find(i:integer);
var j:integer;
    ocur:... {old cur}
    ob:.. {old b}
begin
if nn=2 ...
end;
ocur:=cur;
ob:=b;
for j:=2 to nn-1 do begin
    a[i]:=j;
    cur:=cur+b[j]*(b[j-1]+b[j+1]);
    delete(j);
    find(i+1);
    b:=ob;
    cur:=ocur;
end;
end;
```

обратите внимание, что теперь переменная x не нужна. Но в результате может не хватить стека, т.к. каждая копия процедуры *find* будет хранить свой массив *ob*.

- На самом деле нас теперь массив a нужен только для того, чтобы выводить ответ. Если сам ответ выводить не надо, то можно не хранить массив a (и, соответственно, массив ans) вообще. Тогда нам не нужна будет и переменная i; процедура *find* теперь не будет принимать никаких параметров (!), она теперь будет перебирать один шаг удаления и запускаться рекурсивно для дальнейшего перебора (а текущая глубина перебора пока неявно присутствует в переменной nn).

Задание 2.15: *Напишите программу без массива a и переменной i. Попробуйте её осознать «с нуля».*

Задание 2.16: *Напишите эту программу, работая со связными списками.*

Подумаем тут над тем, какие можно придумать отсечения. Во-первых, если все числа положительны, то очевидно, что если $cur \geqslant best$, то решения лучше чем текущее, мы точно не найдём. Поэтому первое отсечение — `if cur>=best then exit`. Это (как я уже говорил), фактически, стандартное отсечение в подобных задачах.

Обратите внимание: условие нестрогое. `if cur>=best`, а не `if cur>best`. Действительно, нам несколько раз получать оптимальное решение не надо. Вот если бы надо было вывести *все* оптимальные решения, тогда пришлось бы писать $cur > best$.

Можно пытаться придумывать другие отсечения. Например, за каждое удаление мы получаем штраф, как минимум равный удвоенному удаляемому числу (считаем, что у нас числа натуральные, и значит, сумма соседей $\geqslant 2$). Поэтому за удаление всех чисел мы получим штраф как минимум равный удвоенной сумме всех чисел. Поэтому, если знать сумму всех чисел (кроме крайних) s, то можно отсекать по условию $cur + 2 \cdot s \geqslant best$. Сумму можно поддерживать во время работы программы или каждый раз считать заново. (Поддерживать значит хранить в отдельно переменной и быстро пересчитывать при каждом изменении массива. Мы это уже много раз делали).

Задание 2.17: *Напишите оба этих варианта программы: с хранением суммы в отдельной переменной или с вычислением каждый раз заново.*

На самом деле, как я уже говорил, отсечения можно придумывать до бесконечности. Можно учесть, какой минимальный элемент у нас остался, и умножать не на 2, а на удвоенный этот элемент и т.д. Главное, не запутаться в этих отсечениях, нигде не ошибиться, не опоздать решить другую задачу :) и не писать отсечения, которые будут все равно неэффективны. Т.е. главное — не переборщить.

Задание 2.18: *Придумайте отсечения к задаче о паросочетании в произвольном графе (задание 2.9, в обоих вариантах: а и б). Напишите полную программу.*

2.3. Эвристики

Обсудим более подробно задачи на оптимизацию. Как несложно видеть, эффективность всех отсечений в них далеко не в последнюю очередь определяется тем, какое решение мы уже нашли, т.е. чему у нас равно значение *best*. Чем лучше у нас *best*, т.е. чем оптимальнее найденное нами на данный момент решение, тем больше веток будет отсекаться.

Поэтому имеет смысл как можно быстрее получать хорошие решения. Я вижу, в общем-то, три способа это делать:

- Во-первых, можно ещё до запуска перебора найти какие-нибудь решения,
- во-вторых, можно перебирать решения в некотором особом порядке, стараясь, чтобы хорошие решения были бы пораньше,
- и в-третьих, можно пытаться, найдя некоторое решение, попробовать его как-нибудь улучшить.

Рассмотрим это по порядку.

2.3.1. Эвристики до перебора

Можно ещё до запуска перебора попробовать найти какое-нибудь решение, причём постаравшись, чтобы оно было не слишком плохое. Т.е. специально перед запуском перебора запустить какой-нибудь быстрый алгоритм, который найдёт какое-нибудь решение. Возможно (даже скорее всего), оно будет неоптимальным (если бы оно всегда было оптимальным, то перебор нам не был бы нужен), но все равно неплохим, позволяя отсекать плохие ветки перебора.

Например, в большинстве задач имеет смысл попробовать найти решение с помощью жадного алгоритма. Жадным называется любой алгоритм, который на каждом шагу пытается сделать локально оптимальное действие, не заботясь от том, что будет на следующем шаге.

Например, в нашей любимой задаче про удаление чисел можно предложить следующий алгоритм решения: в начальной ситуации выполняем то удаление, которое даёт наименьший штраф, в получившейся ситуации опять делаем удаление, которое даёт наименьший штраф и т.д., пока не останутся два числа:

```
best:=0;
ob:=b;
for i:=1 to n-2 do begin
    {найдем, какое удаление сейчас дешевле всего}
    min:=inf;       {число, которое больше любого возможного штрафа}
    for j:=2 to n-2 do
        if b[j]*(b[j-1]+b[j+1])<min then begin
           minj:=j;
           min:=b[j]*(b[j-1]+b[j+1]);
        end;
    {и удалим, записав результат сразу в ans}
    best:=best+min;
    ans[i]:=j;
    delete(minj);
end;
b:=ob;
```

Это пишется в главной программе, перед вызовом $find(1)$, т.е. перед началом перебора (или ещё лучше это сделать отдельной процедурой и вызвать её до вызова $find(1)$). Я сразу сохраняю решение в $best$ и ans, т.к. это будет то «текущее оптимальное» решение, с которым мы начнём перебор (а раньше в начале перебора у нас не было никакого «текущего оптимального» решения, а $best$ равнялось какому-нибудь очень большому числу, inf).

Конечно, нет никакой гарантии, что этот метод даст оптимальное решение; более того, в данной конкретной задаче даже вообще не очевидно, что он даст решение, близкое к оптимуму. Но это в любом случае лучше, чем начинать перебор с $best = inf$.

Задание 2.19: *Приведите пример, когда этот алгоритм находит неоптимальное решение.*

Задание 2.20: *Напишите эту программу полностью. Сравните её эффективность с программой без предварительного жадного поиска решения.*

В общем случае можно придумать много разных алгоритмов (не обязательно жадных, кстати), которые, может быть, дают неплохой результат. Как правило, никаких доказательств их эффективности (оптимальности) нет, есть только надежда, что они дадут неплохой результат. Поэтому (насколько я понимаю) такие алгоритмы называются *эвристиками*.

(Это не значит, что жадный алгоритм никогда не является *точным* решением задачи; бывают задачи, которые *точно* решаются жадностью. Но переборные задачи обычно *точно* жадностью не решаются).

В одной задаче можно придумывать очень много эвристик. Например, здесь же можно пытаться удалять числа в порядке возрастания. Не знаю, будет это хуже или лучше, но попробовать не мешает. Можно наоборот придумать антижадный алгоритм: выбирать на каждом шагу удаление с наибольшим штрафом, или удалять числа в порядке убывания. Для каждого из этих алгоритмов можно попытаться объяснить, почему он правдоподобен (например, зачем удалять числа в порядке убывания: если большое число удалять в конце, то оно, во-первых, на себя штрафа много потребует, а во-вторых, потребует много штрафа ещё несколько раз, когда оно будет оказываться соседом с удаляемым числом. Если же удалить в начале, то оно не будет «мешаться» позже, т.е. не будет оказываться соседом с удаляемым числом). Но это все будут лишь оправдания; я сомневаюсь, что можно придумать строгие *доказательства* этих алгоритмов: скорее всего, в общем случае они неверны. Тем не менее они могут дать неплохое начальное приближение.

Если делать на олимпиаде нечего, можно заниматься придумыванием кучи эвристик, реализовать их все (!) и программно выбирать, какая лучше. В итоге ваша программа будет делать следующее: запускает первую эвристику, смотрит ответ на неё. Запускает вторую, смотрит её ответ, если он лучше, то заменяет «текущий оптимальный» ответ $best$ и текущее решение ans на ответ второй эвристики. Потом запускает третью и т.д. Тем самым в начале перебора $best$ будет лучшим из всего, что на этом тесте смогли сделать эвристики.

Задание 2.21: *Напишите задачу про удаление чисел со всеми четырьмя обсуждавшимися тут эвристиками. Может быть, вы придумаете ещё эвристики к ней?*

Как правило, эвристики работают намного быстрее перебора, и поэтому обычно *можно* написать много эвристик, и это по крайней мере не ухудшит программу. Кроме того, и при написании эвристик не стоит очень оптимизировать их; например, удалять элементы в порядке убывания можно, выбирая на каждом шагу минимальный из оставшихся элементов заново (фактически, сортируя выбором главного элемента), а не сортируя их предварительно qsort'ом и т.п. — все равно, если N большое, то у вас нет шансов пройти тест, потому что перебор не успеет отработать, а на маленьких N все равно, какую сортировку применить.

Другое дело, что увеличивать количество эвристик опасно, т.к. (как всегда) есть риск где-нибудь ошибиться. Поэтому всегда надо делать разумное количество эвристик и разумно распределять своё время: может быть, стоит все-таки придумать нормальное решение или, если уж и не успеете решить задачу по-нормальному, то хотя бы проверьте, что все работает! что вы нигде не наглючили, в т.ч. не забыли ничего откатить в процедуре перебора...

2.3.2. Эвристики во время перебора

Во время перебора можно жонглировать порядком, в котором выбираются значения для каждого элемента. Если есть основания думать, что оптимальное решение скорее достигнется по одной ветке перебора, а не по другой, то имеет смысл сначала пойти по ней.

Например, в нашей любимой задаче про удаление чисел можно в процедуре $find(i)$ перебирать, какое число мы будем удалять, не просто слева-направо (`for j:=2 to nn-1`), как было во всех примерах выше, а, например, в порядке убывания. Т.е.: удалить самое большое число. запустить $find(i+1)$. Восстановить самое большое число, удалить число поменьше, запустить $find(i+1)$. Восстановить это число и т.д.

Это можно реализовать, заведя массив *was* и отмечая в нем, какие числа мы уже пытались на этом уровне рекурсии удалять:

```
procedure find(i:integer);
var j,k:integer;
    x:integer;
    was:array...
    maxj:integer;
    max:integer;
begin
if nn=2 then begin
   check;
   exit;
end;
fillchar(was,sizeof(was),0);
for k:=2 to nn-1 do begin
      {найдем наибольший из элементов, которые
      еще не пробовали удалять на этом уровне рекурсии}
    max:=0;
    for j:=2 to nn-1 do
        if (was[j]=0)and(b[j]>max) then begin
           max:=b[j];
           maxj:=j;
        end;
      {и попробуем удалить его}
    was[maxj]:=1;
    a[i]:=maxj;
    cur:=cur+b[maxj]*(b[maxj-1]+b[maxj+1]);
    x:=delete(maxj);
      {переберем, что может получиться в этом варианте}
    find(i+1);
      {после этого откатим изменения}
    insert(maxj,x);
    cur:=cur-b[maxj]*(b[maxj-1]+b[maxj+1]);
end;
end;
```

Обратите внимание, что массив *was* выделен в стеке, а не глобальной переменной. Понятно, почему: потому что у каждой *find* свой массив *was*. Когда работает $find(5)$ (т.е. были вызовы $find(1)$, $find(2)$, ..., $find(5)$, и все пять процедур находятся в стеке), то она должна отдельно хранить, кого она использовала; $find(4)$ (которая только что вызвала $find(5)$) — отдельно и т.д. Надеюсь, понятно.

В принципе, аналогично можно написать и так, чтобы удалять в порядке увеличения (или уменьшения) штрафа за удаление (т.е. чтобы первым рассмотреть жадный ход, потом (перебрав все решения, начинавшиеся на жадный ход) — рассмотреть следующий вариант и т.д.).

Задание 2.22: *Напишите такую программу.*

Эти идеи — тоже по сути эвристики, в том смысле, что они тоже никак строго не обосновываются. Есть просто надежда, что они помогут, но не ясно, почему, и не ясно, насколько сильно помогут.

Обратите внимание, что здесь не получится применить несколько эвристик одновременно, поэтому придётся вам выбирать, какой эвристике вы больше доверяете :)

И ещё. Рассмотрим программу, написанную чуть выше, которая в первую очередь удаляет самое большое число. Каким будет решение, для которого она первый раз вызовет процедуру *check*? Это будет решение, в котором первым ходом было удалено самое большое число, вторым — самое большое из оставшихся и т.д., т.е. в точности решение, которое нашлось бы одной из рассмотренных в разделе 2.3.1 эвристик-до-перебора. В задании 2.21 вы писали эту задачу с четырьмя эвристиками, но теперь *первое же* найденное перебором решение будет совпадать с решением, найденных одной из них, поэтому эту эвристику можно и не запускать. Если ещё не понятно, почему, то попробуйте понять.

2.3.3. Локальная оптимизация

Эту идею я сам ни разу не применял, пример можете посмотреть в ОНЗИ[1]. Идея состоит в следующем: пусть мы нашли какое-то решение. Попробуем его *немного* поизменять, вдруг получится лучше. Например, вспомним задачу о паросочетании минимального веса в полном графе. Пусть перебор нашёл некоторое решение. Попробуем, например, всеми возможными способами поменять ребра «крест-накрест». Т.е. перебираем все $n(n-1)/2$ пар рёбер и для каждой пары рёбер $(u—v)$ и $(u'—v')$, входящих в решение, рассматриваем решение, которое отличается от найденного заменой этой пары рёбер на $(u—v')$ и $(u'—v)$, или что-то типа того: (храним в массиве a сами ребра)

```
for i:=1 to n do
    for j:=i+1 to do begin
        {начало первого ребра меняем с началом второго}
        t:=a[i].a;
        a[i].a:=a[j].a;
        a[j].a:=t;
        проверить получившееся решение
        вернуть a назад.
    end;
```

Может быть, это имеет смысл применять не для каждого найденного решения, а только для решений, которые становятся текущими-лучшими.

Я не уверен, что это имеет смысл делать здесь. Ещё раз: я сам никогда этого не применял. Поэтому подробности смотрите в ОНЗИ, там это довольно подробно описано. Но, как и со всеми эвристиками, тут нет строгих рассуждений, что лучше, что хуже. Что вам кажется лучше, то и делайте.

Задание 2.23: *Придумайте эвристики до перебора и во время перебора к задаче о паросочетании в произвольном графе (задание 2.9, в обоих вариантах: а и б). Напишите полную программу.*

[1] Виталий Беров, Антон Лапунов, Виктор Матюхин, Анатолий Пономарев. *Особенности национальных задач по информатике.*

2.4. Дополнительные идеи

2.4.1. Отсечение по времени

В реальных олимпиадных задачах у вас всегда есть ограничение времени, превышать которое не стоит. Если ваша программа проработает больше отведённого времени, то вы получится 0 баллов за тест — ничего хуже в этом тесте быть не может :) Поэтому имеет смысл избегать такого результата всеми силами, в надежде, что, может быть, повезёт и получится что-то лучше. А именно:

- В задачах на оптимизацию (т.е. когда надо найти объект с оптимальными параметрами) можно, когда вы видите, что время подходит к концу, просто вывести лучший найденный на данный момент объект и завершить работу. Если повезёт и он на самом деле будет оптимальным, то тест будет зачтён, иначе хуже, чем TL, все равно не будет.
- В задачах на поиск объекта ничего не остаётся, как вывести, что решения не существует (естественно, ваша программа должна также завершаться сразу, как только нашла решение). Кстати, это далеко не всегда бывает неправильно, даже наоборот: в хорошем наборе тестов обязательно должны быть большие тесты с ответом «решения не существует». Если же вам требовалось (редкий случай) вывести *все* решения, то ничего не остаётся, как их все найденные на данный момент и вывести, и завершить работу (про вывод *всех* решений см. ниже).
- В задачах на подсчёт числа объектов ничего не остаётся, как вывести, сколько вы уже насчитали. Конечно, скорее всего это будет неправильно (ваша программа работала секунду, насчитала, допустим 1432 объекта, и хочет работать ещё секунду... неужто она не найдёт ни одного объекта больше? :) а если найдёт, то, значит, 1432 — неверный ответ), но работать больше нельзя и ничего лучше тут, видимо, не придумаешь.

Примечание: Тестирование программы с отсечением по времени на компьютере жюри выглядит весьма эффектно, особенно если вы смотрите непосредственно в экран тестирующего компьютера, на котором пишется, сколько времени осталось: время приближается к TL, и все уже готовы увидеть TL, но нет — за доли секунды до TL ваша программа завершается, запускается чекер и вы видите WA — что ж, не повезло — или OK — ура, повезло.

Как технически делать отсечение по времени? В Borland Pascal (как и в любых DOS-программах) есть очень удобная вещь: по адресу 0:$46C (т.е. в ячейке памяти с абсолютным номером $46C_{16}$) лежат четыре байта, которые образуют переменную типа *longint*. Её значение автоматически (!) увеличивается на единицу примерно 18 раз в секунду (за счёт прерывания таймера). Поэтому отсечение по времени можно писать так:

```
var t:longint absolute 0:$46c;
      {такая конструкция с absolute позволяет указать абсолютный адрес,
      где в памяти будет храниться значение переменной}
    t0:longint;

procedure find(...)
begin
if i>k...
end;
if t>t0+18 then begin {считая, что ограничение времени 1 секунда}
   out; {процедура out выводит текущее найденное решение в выходной файл}
   halt;
end;
...
end;

begin {основная программа}
t0:=t; {сохраним начальный момент времени в t0}
...
end.
```

может, стоит ставить 17, а не 18, на всякий случай, и т.п.

В Windows-программах все не так просто, в частности потому, что нужно учитывать процессорное время, потраченное вашей программой. Насколько я понимаю, есть функция *getTickCount*, но считается, что она подтормаживает и *каждый* раз в функции *find* её вызывать — это очень медленно. Тогда может иметь смысл завести ещё глобальную переменную *nn*, которая будет считать, сколько раз вы вошли в *find*, и только, например, каждый 1024-ый раз проверять. Типа того:

```
var t:longint;
    nn:longint;

procedure find...
begin
if i>k...
end;
if (n and 1023=0)and(gettickcount>t0+1000) then begin
   ...
end;
inc(nn);
...
```

```
...

begin
t0:=gettickcount;
nn:=0;
...
end.
```

(`n0 and 1023`, а не `n0 mod 1024`, поскольку первое работает намного быстрее; по той же причине проверяю каждые 1024, а не каждые 1000 раз). На само деле, конечно, необходимая частота проверок сильно зависит от задачи: иногда может понадобиться и каждый 65 536-ой раз проверять и т.п.; каждый раз стоит подбирать константу заново, чтобы проверки были достаточно частыми, но не слишком частыми.

И ещё, конечно, можно это организовать и по-другому:

```
if (n=1000) then begin
   n:=0;
   if (gettickcount>t0+1000) then begin
      ...
   end;
end;
inc(nn);
```

т.е. при проверке сбрасывать счётчик в ноль. Теперь взятие остатка по модулю не нужно вообще, и можно работать с любым модулем. Но подобная проверка все равно не сильно быстрее проверки по модулю «два в степени k», поэтому как вам больше нравится, так и пишите. Первый способ позволяет вам также узнать в конце программы, сколько же раз на самом деле вызывалась ваша функция.

2.4.2. Перебор двумерного массива

Иногда объекты, которые мы перебираем, проще представлять в виде двумерного массива (а не одномерного, как было всегда раньше). Пусть, например, надо перебрать все способы заполнения *матрицы* $N \times N$ нулями и единицами. Можно это написать так:

```
procedure find(i,j:integer); {i,j --- координаты клетки, которую перебираем}
begin
if i>n then begin {если кончилась вся матрица}
   check;
   exit;
end;
if j>n then begin {если кончилась текущая строка}
   find(i+1,1);   {то перейти к следующей}
   exit;
end;
a[i,j]:=0;
find(i,j+1);
a[i,j]:=1;
find(i,j+1);
end;
```

Осознайте этот пример.

2.4.3. Вариации порядка выбора элементов

(Это не то, что обсуждалось в разделе про эвристики.) Иногда имеет смысл заполнять элементы ответа не в том порядке, в котором приходит в голову, а продумать, в каком. Например, пусть наша задача — дано N^2 чисел, проверить, можно ли из них составить магический квадрат (т.е. квадрат, в котором сумма каждой строки равна сумме каждого столбца). Можно, конечно, перебирать так, как написано в предыдущем пункте: т.е. выбирать значения для первой строки, потом для второй и т.д... Но можно поступить так: в $find(1)$ перебираем значение клетки $(1,1)$, в $find(2) - (1,2)$, ... $find(n) - (1,n)$, $find(n+1) - (2,1)$ и внимание! $find(n+2) - (3,1)$, $find(n+3) - (4,1)$ и т.д., потом остаток второй строки, потом остаток второго столбца и т.д., в таблице справа следующего абзаца для $N = 5$ приведены номера, какая клетка какой по счету будет.

1	2	3	4	5
6	10	11	12	13
7	14	17	18	19
8	15	20	22	23
9	16	21	24	25

Смысл в том, что в этой задаче есть естественное отсечение: если мы заполнили очередную строку или столбец, то стоит сразу проверить, что его сумма равна сумме всех чисел, делённой на N (очевидно, что именно такая должна быть сумма каждой строки и каждого столбца). Поэтому стоит заполнять таблицу в таком порядке, чтобы проверять можно быть как можно быстрее. Если заполнять построчно, то проверять можно будет после первой строки (при глубине рекурсии N), после второй ($2N$), после третьей ($3N$), и т.д., зато в конце — на всей последней строке будем проверять суммы столбцов.

А если делать заполнять по очереди строки и столбцы (как описано два абзаца назад и показано в примере справа), то отсечения будут: после первой строки (на глубине N), после первого столбца (на глубине $2N-1$, а не $2N$ (!)), после второй строки ($3N-2$, а не $3N$) и т.д. — т.е. отсечения будут раньше и программа будет работать быстрее.

Аналогичные идеи могут быть и в других задачах, хотя, наверное, весьма редко.

2.4.4. Вывод всех оптимальных решений

Пусть надо вывести *все* оптимальные решения. Можно, конечно, завести большой массив, куда их записывать, но имхо проще поступить так: при нахождении очередного оптимального решения просто выводить его сразу в файл. Находится ещё одно столь же хорошее решение — его тоже выводим туда же. Если же находится решение, которое ещё оптимальнее, чем все, что было раньше, то делаем *rewrite* — и все решения, которые были выведены раньше, сотрутся. Это все делается в процедуре *check*, конечно.

Пример: пусть в задаче про удаление чисел надо было бы вывести *все* оптимальные решения. Тогда пишем

```
procedure check;
var i:integer;
begin
```

```
if cur<best then begin
   best:=cur;
   rewrite(f);
   writeln(f,cur);
end;
if cur=best then  {это выполнится и в случае, когда только что нашлось еще более хорошее решение}
   for i:=1 to n-2 do write(f,a[i],' ');
end;
```

Т.е. если нашлось решение ещё лучше, то *rewrite* — потираем все решения, что были найдены раньше, выводим новую оптимальную сумму (если, конечно, требуется по условию), и делаем $best := cur$.

Далее, если $cur = best$, а это теперь будет и если мы только что нашли ещё более хорошее решение (т.е. если только что сделали *rewrite* и т.д.), и если мы просто нашли ещё одно столь же хорошее решение, что и раньше, то выводим его.

Заметьте, что теперь массив *ans* не нужен.

Не забудьте, что в таком случае уже нельзя делать отсечение по нестрогому условию (т.е. $\geqslant$), а только по строгому ($>$).

Кстати, ещё мысль. Аналогично можно поступить и если выводить надо только одно решение. Можно его не сохранять в *ans*, а сразу выводить

```
procedure check;
var i:integer;
begin
if cur<best then begin
   best:=cur;
   rewrite(f);
   writeln(f,cur);
   for i:=1 to n-2 do write(f,a[i],' ');
   close(f);
end;
end;
```

За счёт $close(f)$ при отсечении по времени можно будет сразу делать *halt* — в каждый момент времени лучшее на данный момент времени решение у нас уже лежит в файле, и при отсечении по времени вам ничего вообще делать не надо, просто *halt*.

Приложение А.

Задачи и ответы

А.1. Условия всех задач

...упражнений, которые настоятельно и, как всегда, безуспешно, рекомендую делать...

А.1.1. Начало работы в Free Pascal

А.1.2. Перебор с возвратом

Контрольный вопрос 2.1 (стр. 16): Видите, почему?

Задание 2.2 (стр. 16): Напишите эту программу (собственно, я надеюсь, что и предыдущие программы вы написали). Потестите её (обратите внимание, что тут время работы от n не зависит, только от k, потому имеет смысл брать и большие n). Найдите в ней баг и придумайте, как его исправить. Кроме того, заметьте, что одно и то же решение выводится несколько раз, отличаясь перестановкой слагаемых. Придумайте, как это исправить (может быть, вам поможет сначала почитать следующий пример, но лучше подумайте сначала, не читая примера дальше).

Задание 2.3 (стр. 16): Можно, конечно, это проверять в процедуре *check*. Т.е. процедура $find$ будет фактически работать по предыдущему примеру, а процедура *check* будет отбирать то, что нужно. Напишите такую программу. Обратите внимание на то, чтобы не брать одно и то же сочетание несколько раз.

Задание 2.4 (стр. 17): а) Напишите эту программу. Обратите внимание на подготовку вызова $find(1)$; проверьте, что перебираются действительно **все** сочетания (например, выводя их в файл и проверяя при маленьких n и k).

б) Добавьте в программу код, который выводит (на экран или в файл) «лог» работы рекурсии (например, выводя при присвоении $a[i] := j$; на экран строку '`a[i]=j`', сдвинутую на i пробелов от левого края строки: вам этот вывод покажет, что́ на самом деле делает программа и пояснит предыдущий абзац); этот «лог» лучше выводить вперемешку с найденными решениями, чтобы видеть, какая ветка рекурсии чем закончилась. Подумайте над тем, как исправить то, что описано в предыдущем абзаце, т.е. как сделать так, чтобы каждая ветка рекурсии заканчивалась нахождением решения.

Задание (элементарное) 2.5 (стр. 18): Напишите программу перебора всех A_n^k — всех размещений из n по k (в них, в отличии от C_n^k, порядок важен).

Задание 2.6 (стр. 20): (Сложное) Напишите эту программу полностью и доведите ее до такого состояния, чтобы можно было играть с компьютером в крестики-нолики.

Задание 2.7 (стр. 21): (Задача 159 с `informatics.mccme.ru`) Радиолюбитель Петя решил собрать детекторный приемник. Для этого ему понадобился конденсатор емкостью C мкФ. В распоряжении Пети есть набор из n конденсаторов, емкости которых равны c_1, c_2, ..., c_n, соответственно. Петя помнит, как вычисляется емкость параллельного соединения двух конденсаторов ($C_{new} = C_1 + C_2$) и последовательного соединения двух конденсаторов ($C_{new} = C1 \cdot C2/(C1 + C2)$). Петя хочет спаять некоторую последовательно-параллельную схему из имеющегося набора конденсаторов, такую, что ее емкость ближе всего к искомой (то есть абсолютная величина разности значений минимальна). Разумеется, Петя не обязан использовать для изготовления схемы все конденсаторы.

Напомним определение последовательно-параллельной схемы. Схема, составленная из одного конденсатора, – последовательно-параллельная схема. Любая схема, полученная последовательным соединением двух последовательно-параллельных схем, – последовательно-параллельная, а также любая схема, полученная параллельным соединением двух последовательно-параллельных схем, – последовательно-параллельная. Обратите внимание, что это определение не допускает произвольные схемы, а только полученные именно последовательностью параллельных или последовательных соединений.

Задание 2.8 (стр. 21): Напишите программу перебора всех последовательностей из 0 и 1 без k нулей подряд, в которых всего n символов. (Например, при $k = 2$ и $n = 3$ это будут последовательности 010, 011, 101, 110 и 111). Основной задачей программы будет посчитать, сколько таких последовательностей всего, но имеет смысл выводить их на экран (или в файл) для проверки.

а) Напишите эту программу, модифицировав пример 1, т.е перебирая все последовательности из 0 и 1 длины n, и проверяя, что последовательность «правильная», только в процедуре *check*.

б) Напишите программу, которая будет перебирать только такие последовательности, т.е. чтобы каждая ветка перебора заканчивалась нахождением решения, и в процедуре *check* проверки не были бы нужны.

в) (дополнительный пункт, не имеющий отношения к перебору) Если вы раньше не сталкивались с такой задачей, то попробуйте найти закономерность ответов при фиксированном k (т.е. сначала посмотрите на ответы на задачу при $k = 2$ и найдите в них закономерность, потом поищите закономерность при $k = 3$, потом при $k = 4$ и т.д.) Кстати, не забудьте, что тестить имеет смысл и очевидный случай $k = 1$:)

Задание 2.9 (стр. 21): Паросочетание в произвольном графе. Рассмотрим граф с $2N$ (т.е. чётным) количеством вершин. Паросочетанием в нем назовём набор из N рёбер, у которых все концы различны (т.е. каждая вершина соединена ровно с одной другой: разбиение вершин на пары). [В олимпиадном программировании обычно рассматривается только паросочетание в двудольном графе, т.к. там есть простой эффективный алгоритм. Но у нас граф будет произвольным и мы будем решать задачу перебором]. [Т.е. смысл этой задачи на самом деле — чтобы вы умели перебирать все разбиения на пары]

а) Напишите программу, которая будет перебирать все разбиения вершин на пары и проверять, является ли такое разбиение паросочетанием (т.е. все ли нужные ребра присутствуют в нашем графе).

б) Считая, что граф полный и взвешенный, напишите программу, которая найдёт паросочетание наименьшего веса.

Задание 2.10 (стр. 21): Напишите программу перебора всех разложений числа N на натуральные слагаемые.

Вариант 1: ровно на k слагаемых

а) считая, что слагаемые могут повторяться и что порядок слагаемых важен (т.е. что $2+1$ и $1+2$ — это разные решения);

б) считая, что порядок слагаемых не важен, т.е. выводя только одно разложение из разложений $2 + 1$ и $1 + 2$, при этом допуская одинаковые слагаемые;

в) считая, что все слагаемые должны быть различны, при этом порядок слагаемых не важен.

Вариант 2: на сколько угодно слагаемых в тех же трёх подвариантах (а, б и в)

Написав программы, прежде чем тестировать их, ответьте в уме на такой вопрос: ваша (уже написанная!) программа в варианте а) будет при $n = 3$ выводить решения $1 + 2$ и $2 + 1$. А при $n = 2$ она будет выводить $1 + 1$ один раз или два раза (во второй раз как будто переставив единички)?

Задание 2.11 (стр. 21): Задача «Числа». Дана последовательность из N чисел. За одно действие разрешается удалить любое число (кроме крайних), заплатив за это штраф, равный произведению этого числа на сумму двух его соседей. Требуется удалить все числа (кроме двух крайних) с минимальным суммарным штрафом.

У этой задачи есть (не самое тривиальное) динамическое решение, но напишите переборное решение. Тут надо перебрать все варианты удаления чисел и выбирать из них тот, который даст минимальный штраф.

Задание 2.12 (стр. 22): (Какая-то довольно искусственная задача, но хорошо подходит для иллюстрации одной из идей далее). Посчитать количество последовательностей из m нулей и n единиц, удовлетворяющих следующих условиям. Первое условие: никакие две единицы не должны стоять рядом. Таким образом единицы делят последовательность на несколько групп из подряд идущих нулей. Второе условие: количество нулей в последовательных группах должно неубывать, и при этом в соседних группах должно отличаться не более чем на 1. Эта задача имеет динамическое решение, но напишите перебор.

Задание 2.13 (стр. 23): Переделайте эту программу, чтобы не хранить глобальную переменную s, а передавать s (или $n - s$, т.е. оставшееся число) через параметр процедуры.

Задание 2.14 (стр. 24): (Творческое) Попробуйте придумать, как бы написать программу, чтобы не нужно было выделять $i = 1$ в особый случай. Это не очень тривиально, и есть несколько вариантов, как это сделать.

Задание 2.15 (стр. 26): Напишите программу без массива a и переменной i. Попробуйте её осознать «с нуля».

Задание 2.16 (стр. 26): Напишите эту программу, работая со связными списками.

Задание 2.17 (стр. 26): Напишите оба этих варианта программы: с хранением суммы в отдельной переменной или с вычислением каждый раз заново.

Задание 2.18 (стр. 26): Придумайте отсечения к задаче о паросочетании в произвольном графе (задание 2.9, в обоих вариантах: а и б). Напишите полную программу.

Задание 2.19 (стр. 27): Приведите пример, когда этот алгоритм находит неоптимальное решение.

Задание 2.20 (стр. 27): Напишите эту программу полностью. Сравните её эффективность с программой без предварительного жадного поиска решения.

Задание 2.21 (стр. 27): Напишите задачу про удаление чисел со всеми четырьмя обсуждавшимися тут эвристиками. Может быть, вы придумаете ещё эвристики к ней?

Задание 2.22 (стр. 28): Напишите такую программу.

Задание 2.23 (стр. 28): Придумайте эвристики до перебора и во время перебора к задаче о паросочетании в произвольном графе (задание 2.9, в обоих вариантах: а и б). Напишите полную программу.

... Вы не ужасайтесь... реально все тривиально...

A.2. Подсказки

A.2.1. Начало работы в Free Pascal

A.2.2. Перебор с возвратом

Контрольный вопрос 2.1 (стр. 16): Посмотрите, как будет заканчиваться цикл while.

Задание 2.2 (стр. 16): Для поиска бага попробуйте включить ключи компилятора.

Задание 2.4 (стр. 17): а) Включите ключи компилятора; б) Подумайте, почему некоторые ветки не находят решения и как это исправить.

Задание 2.8 (стр. 21): б) Можно дописывать ноль, только если текущая последовательность заканчивается меньше, чем на $k-1$ нулей. Можно каждый раз считать заново, на сколько нулей заканчивается текущая последовательность, а можно передавать в $find$ дополнительный параметр — сколько нулей стоят в конце текущей последовательности. Попробуйте написать оба способа.

Задание 2.9 (стр. 21): На самом деле вариант а) отличается от варианта б) только процедурой *check* и возможными отсечениями (см. раздел II). Основное в процедуре $find$ у них одно и то же: перебор всех разбиений N объектов на пары. Пожалуй, основной нетривиальностью, над которой придётся подумать, тут будет то, что в $find(i)$ может оказаться, что i-я вершина уже с кем-нибудь спарена. Можно предложить два варианта решения проблемы:

1. Можно в массиве хранить *список выбранных рёбер* (!): он тогда будет

```
array of record a,b:integer; end;
```

переменная i в $find$ будет указывать, какое ребро мы хотим выбрать (в смысле, $i=1$ значит, что мы ещё не выбрали ни одного ребра, $i=2$ — что выбрали одно и т.д.).

В процедуре $find$ теперь ищем первую вершину, которая ещё не «спарена», т.е. не является концом ни одного из взятых ещё рёбер, её обязательно берём, и перебираем ей пару. Для того, чтобы не тратить время на проверку, «спарена» ли вершина, можно завести массив was, в котором отмечать, спарены ли вершины (и не забывать откатывать!)

Это решение довольно прямо идёт по естественной идеологии перебора: нам надо выбрать N рёбер — так и будем их последовательно выбирать, записывая номера выбранных в массив a.

2. Но можно делать и, как мне кажется, проще. Можно в массиве a хранить номер «парной» вершины к данной вершине: т.е. $a[i]$ — номер вершины, парной к i, или ноль, если вершина пока ещё не спарена. В частности, для уже спаренных вершин обязательно должно быть $a[a[i]]=i$. Процедура $find(i)$ будет перебирать пары к i-ой вершине. А именно, если она уже с кем-то спарена, то перебирать нечего, иначе перебираем все свободные вершины в качестве пары. Массив was тут не нужен, т.к. «спаренность» вершины можно проверять, проверяя $a[i]=0$. Обратите ОСОБОЕ ВНИМАНИЕ на то, что здесь придётся откатывать ИЗМЕНЕНИЯ В МАССИВЕ a! — это довольно редкий случай, но вот вам пример, когда это действительно нужно.

Задание 2.10 (стр. 21): Можно ввести дополнительную глобальную переменную, в которой хранить текущую сумму слагаемых, в процедуре $find$ увеличивать её на то слагаемое, которое поставили, и не забывать потом вернуть назад. Можно поступить по-другому: передавать в процедуру $find$ дополнительный параметр, который обозначает, сколько ещё осталось разложить (т.е. $N-$ *(сумма уже выбранных слагаемых)*). При этом тогда очередное слагаемое будет ограничено сверху значением этого параметра. Вариант 2: подумайте, какое должно быть условие выхода из рекурсии.

Задание 2.11 (стр. 21): В массиве a будем хранить последовательность удалений (на самом деле, тут нам массив a практически не будет нужен). Стоит (в добавок к массиву a) хранить ещё один глобальный массив, в котором будет храниться текущее состояние чисел, и ещё глобальную переменную — текущий штраф. При удаление очередного числа надо откорректировать глобальный массив, удалив из него это число (и сдвинув другие числа), а также изменить текущий штраф. Не забудьте все отыгрывать назад.

Но более продвинутый вариант — хранить текущее состояние чисел связным списком, а не массивом, тогда удалять и добавлять элементы просто.

Задание 2.12 (стр. 22): Простую программу перебора написать несложно, только лучше перебирать не последовательности из нулей и единиц, а сразу способы разбиения m нулей на данное количество групп. Т.е. написать функцию $count(g,m)$, которая будет считать число способов разбиения m нулей на g групп с учётом второго условия, а в ответ выводить $count(n-1,m)+2\cdot count(n,m)+count(n+1,m)$, поскольку возможны четыре варианта:

1. Первый и последний символы искомой последовательности — единицы, тогда групп нулей у нас $n-1$ и потому таких последовательностей $count(n-1,m)$.

2. Первый символ — единица, последний — ноль, тогда групп нулей n и таких последовательностей $count(n,m)$.

3. Первый символ — ноль, последний — единица, тогда групп нулей n и таких последовательностей опять-таки $count(n,m)$.

4. Первый и последний символы — ноли, тогда групп нулей $n+1$ и таких последовательностей $count(n+1,m)$.

Функция же $count$ будет инициализировать и запускать перебор нужных разбиений (т.е. и будет той «главной программой», откуда мы запускаем $find(1)$). Массив a будет хранить количество нулей в очередной группе. Можно проверять отличие соседних групп на 1 только в *check*, а можно и по ходу перебора.

Но интереснее постараться сделать так, чтобы (почти) все ветки заканчивались нахождением решения. Для этого надо, во-первых, сразу в переборе перебирать только те разбиения, где количества нулей в соседних группах или равно, или отличается на единицу, а во-вторых, проверять, хватит ли нам нулей на оставшиеся группы; это мы ещё будем обсуждать в разделе 2.2.3.

Задание 2.14 (стр. 24): Я вижу как минимум два варианта; в обоих для вычисления ответа при данных nn и m придётся запускать процедуру $find$ несколько раз. Во-первых, можно в массиве a устанавливать нулевой элемент, типа того: в процедуре *count*:

```
ans:=0;
nn:=n;
if n>0 then begin
   for i:=1 to nn do begin
       a[0]:=i;
       find(1,m);
   end;
count:=ans;
```

Можно передавать в процедуру $find$ параметр, указывающий предыдущее число; в *count* опять потребуется цикл.

В обоих случаях появляется ещё проблема с тем, что некоторые решения будут считаться дважды (решения, начинающиеся на 3, будут считаться при $a[0] = 2$ и $a[0] = 3$). Можете подумать, что с этим делать.

В общем, ответа на это задание я не привожу, если вы напишите что-нибудь, и оно будет правильно проходить тест 27 100 (на него ответ 94762), то круты :)

Задание 2.16 (стр. 26): На самом деле это задание не на перебор, а на работу со связными списками. Процедура $find$ останется той же, только теперь мы будем хранить текущую последовательность чисел не в массиве, а в списке (на массивах или в динамической памяти), т.к. так проще вставлять и удалять элементы.

Задание 2.18 (стр. 26): В а) проверяйте наличие ребра — и больше сложно что-то придумать; в б) можно написать стандартное отсечение для задач оптимизации: сравнить текущий ответ с наилучшим. Придумайте в б) что-нибудь ещё! Например, к текущей сумме можно добавлять вес минимального оставшегося ребра, умноженный на количество рёбер, которые ещё надо добавить.

Задание 2.19 (стр. 27): Проблема всех жадных решений в том, что, возможно, оптимальный первый шаг вынудит нас слишком много заплатить за последующие, при том, что неоптимальный первый шаг, возможно, позволил бы платить меньше. Соответственно, надо придумать тест, когда, выбрав минимальный штраф на первом ходу, на втором нам приходится платить очень много

Задание 2.23 (стр. 28): Ну, конечно, можно написать жадную эвристику: берём кратчайшее ребро, добавляем его в паросочетание. Берём следующее кратчайшее ребро, которое можно взять и добавляем и т.д. Попробуйте что-нибудь ещё придумать.

A.3. Ответы

A.3.1. Начало работы в Free Pascal

A.3.2. Перебор с возвратом

Контрольный вопрос 2.1 (стр. 16): На последней итерации цикла `a[i]` станет `1 shl 29`, оно обработается, потом удваивается, становится равным `1 shl 30`, и происходит окончание цикла. Значение `1 shl 30` не обрабатывается.

Задание 2.2 (стр. 16): Баг в том, что при вычислении суммы чисел в *check* может быть переполнение. Можно, например написать

```
procedure check;
var j:integer;
    s:longint;
begin
s:=0;
for j:=1 to k do if s<=n-a[j]  then
    s:=s+a[j]
else exit;
if s=n then begin
   for j:=1 to k do
       write(a[j],' ');
   writeln;
end;
end;
```

Должно бы вроде работать.

Исключить повторный вывод одного и того же решения можно, потребовав, чтобы слагаемые неубывали.

```
for j:=0 to 30 do if 1 shl j>=a[i-1] begin
```

Задание 2.3 (стр. 16): Проверить неповторяемость можно, проверяя, что элементы в массиве идут в неубывающем порядке — т.е. идея та же, что и ниже в основном тексте

Задание 2.4 (стр. 17): а) $find(1)$ обращается к $a[0]$. Чтобы все работало, надо перед вызовом $find(1)$ установить $a[0] = -1$ или ещё меньше :), иначе сочетания не смогут начинаться с нуля и т.п. (Именно потому я и предложил считать, что элементы у нас занумерованы от 0 до $n-1$, а не от 1 до n: в последнем случае достаточно было поставить $a[0] = 0$ и это было бы легче не заметить :)).

б) Понятно, что в $find(i)$ бессмысленно ставить $a[i] = n - 1$, если только i не равно k. Вообще, ясно, что не имеет смысла ставить $a[i] > n - (k - i) - 1$ (вроде так, может быть ± 1, подумайте), т.к. элементов на оставшиеся места не хватит. Поэтому стоит делать цикл от $a[i-1] + 1$ до $n - (k - i) - 1$.

Задание (элементарное) 2.5 (стр. 18): То же, что и для перестановок, только проверка на выход из рекурсии будет `if i>k`, а не `if i>n`.

Задание 2.7 (стр. 21): TODO

Задание 2.8 (стр. 21): б) Ну понятно: будем ставить ноль только при условии, что среди предыдущих $k-1$ символов есть единицы. Для $k = 2$ это написать просто:

```
procedure find(i:integer);
begin
if....
end;
a[i]:=1;
find(i+1);
if a[i-1]=1 then begin {ставим ноль, только если предыдущий символ --- 1}
   a[i]:=0;
   find(i+1);
end;
end;
```

только тут надо будет убедиться, что $a[0] = 1$, чтобы последовательности могли *начинаться* с нуля.

Для бóльших k можно писать цикл, который будет считать, на сколько нулей заканчивается текущая последовательность (только аккуратно с $a[0]$, $a[-1]$ и т.д., чтобы последовательности могли начинаться с нулей) — попробуйте это написать!, — а можно это не считать каждый раз заново, а передавать в $find$ дополнительным параметром:

```
procedure find(i,l:integer);
begin
if...
end;
a[i]:=1;
find(i,0);  {на конце текущей последовательности единица, т.е. ноль нулей :) }
if l<k-1 then begin  {можно дописать еще один ноль}
   a[i]:=0;
   find(i+1,l+1);  {стало на один ноль больше}
end;
end;
```

в главной программе тогда надо вызывать $find(1, 0)$ и никаких проблем с $a[0]$ и т.п.

в) Закономерность обсудим в теме "Динамическое программирование".

Задание 2.9 (стр. 21): Общий текст для пунктов а) и б) (как уже было отмечено в подсказках, процедура *find* почти не отличается для них); в части II вам будет предложено придумать отсечения здесь.

В соответствии с вариантом 1 из подсказок:

```
var a:array... of record a,b:integer; end;
    was:array...

procedure find(i:integer); {выбираем i-е ребро паросочетания}
var j,k:integer;
begin
if i>n then begin
   check; {процедура check разная в вариантах а и б}
   exit;
end;
{найдем первую свободную вершину}
for j:=1 to 2*n do {в графе 2*n вершин!}
    if was[j]=0 then break;
{теперь j --- номер первой свободной (не входящей в паросочетание) вершины.
Добавим ее в паросочетание и будем искать парную к ней}
was[j]:=1;
for k:=1 to 2*n do {можно k:=j+1 to 2*n, т.к. до j-ой все точно спарены}
    if was[k]=0 then begin {тут хочется проверить наличие ребра, но пока мы считаем, что это делаем в check}
       was[k]:=1;
       a[i].a:=j;
       a[i].b:=k;
       find(i+1);
       was[k]:=0;
    end;
was[j]:=0;
{обратите внимание, что именно здесь!
или надо was[j]:=0 внутри цикла, но и was[j]:=1 тоже!}
end;
```

В соответствии с вариантом 2 из подсказок:

```
var a:array... of integer;

procedure find(i:integer); {выбираем парную к i-й вершине}
var j:integer;
begin
if i>2*n then begin {количество вершин в графе --- 2*n, а не n!}
   check;
   exit;
end;
if a[i]<>0 then {парная вершина уже выбрана, перебирать нечего}
   find(i+1)
else {надо перебрать все варианты}
     for j:=i+1 to 2*n do if a[j]=0 then begin {i+1 --- т.к. все до i-ой уже точно спарены}
         {спарим i-ую и j-ую вершины}
         a[i]:=j;
         a[j]:=i;
         find(i+1);
         a[i]:=0;
         a[j]:=0;{!!!обязательно, т.к. иначе i-я и j-я будут считаться еще спаренными!}
     end;
end;
```

Задание 2.10 (стр. 21): Варианты а, б, в различаются только тем, что в в) достаточно потребовать, чтобы слагаемые строго возрастали, в б) — неубывали, а в а) это все не имеет значения.

Разберём вариант 1в): заведём глобальную переменную s, в которой храним текущую сумму.

```
var s:...

procedure find(i:integer);
var j:integer;
begin
if i>k then....
end;
for j:=a[i-1]+1 to n-s do begin {слагаемое должно быть больше предыдущего, но явно не больше, чем n-s}
    a[i]:=j;
    s:=s+j;
    find(i+1);
    s:=s-j; {откатываем изменения !!!}
end;
end;
```

Обратите внимание, что в процедуре *check* придётся проверять, что $s = n$.

Варианты 1б и 1в отличаются только нижней границей цикла: $a[i-1]$ и 1 соответственно.

Вариант 2 отличается, в первую очередь, условием выхода из рекурсии. Тут несложно видеть, что условие выхода из рекурсии будет `if s=n`, и в *check* проверять ничего не придётся.

Можно писать и по-другому, не вводя переменную s, а в процедуру *check* передавая оставшуюся сумму rem; теперь процедура *find* будет иметь смысл «разложить число rem на слагаемые» с какими-нибудь условиями. Например для 2а:

```
procedure find(i:integer;rem:integer);
var j:integer;
begin
if rem=0 then begin {если rem=0, то мы разложили уже всё N, т.е. нашли решение}
   check;
   exit;
end;
for j:=a[i-1]+1 to rem do begin
    a[i]:=j;
    find(i+1,rem-a[i]); {осталось разложить rem-a[i]}
end;
end;
```

Можно в *find* передавать и текущую сумму, и т.д. — как вам приятнее.

Задание 2.11 (стр. 21): Разберём в разделе 2.2.4.

Задание 2.12 (стр. 22): Обсудим в разделе 2.2.3.

Задание 2.13 (стр. 23): Элементарно :) передаём s

```
procedure find(i:integer;s:integer);
var j:integer;
begin
if ....
end;
if s>=n then
   exit;
for j:=0 to 30 do  begin
    a[i]:=1 shl j;
    find(i+1,s+a[i]);
end;
end;
```

соответственно в главной программе вызываем $find(1, 0)$;

или передаём $nn = n - s$

```
procedure find(i:integer;nn:integer);
var j:integer;
begin
if ....
end;
if nn<=0 then
   exit;
for j:=0 to 30 do begin
    a[i]:=1 shl j;
    find(i+1,nn-a[i]);
end;
end;
```

Обратите внимание, что здесь *find* в главной программе вызываем $find(1, n)$.

Задание 2.15 (стр. 26): Ну, собственно, все в тексте было сказано.

```
procedure find;
var j:integer;
    ocur:...
    ob:..
begin
if nn=2 then begin
   check;
   exit;
end;
ocur:=cur;
ob:=b;
for j:=2 to nn-1 do begin
    cur:=cur+b[j]*(b[j-1]+b[j+1]);
    delete(j);
    find;
    b:=ob;
    cur:=ocur;
end;
end;
```

например (или через *insert* и не хранить *ob* и *ocur*).

Суть в осознании программы «с нуля». Действительно, что делает эта программа. Здесь процедура *find* по заданной в массиве b последовательности просто перебирает *все* варианты удаления этих чисел. Как она это делает? Если удалять нечего, то просто проверяем. Иначе перебираем, какое число будем удалять первым, удаляем его и вызовом *find* переберём все варианты удаления оставшихся чисел. Процедура *find* тут почти никак не учитывает предысторию, она просто смотрит на текущую позицию и думает, что бы с нею сделать... Может быть, это осознать даже проще, чем все мои рассуждения в I части про перебор окончаний решений и т.д.

Задание 2.16 (стр. 26): Я предпочитаю хранить списки в динамической памяти; может быть, вам приятнее хранить их в массиве record'ов или в нескольких массивах.

```
type pnode=^tnode;
     tnode=record prev,next:pnode; val:integer; end;
     {предыдущий и следующий элементы и значение}
var head:pnode {голова списка}

procedure find(i:integer);
var j:integer;
    ocur:...
    p:pnode;
begin
if nn=2 then begin
   check;
   exit;
end;
ocur:=cur;
p:=head^.next; {начинаем со второго элемента в списке}
j:=2;
while p^.next<>nil do begin
       {пока не дошли до конца списка;
       обратите внимание, что последний
       элемент списка не рассматриваем!}
    a[i]:=j;
    cur:=cur+p^.val*(p^.prev^.val+p^.next^.val);
    p^.prev^.next:=p^.next;
    p^.next^.prev:=p^.prev;  {удалили элемент p из списка}
    find(i+1);
    cur:=ocur;
    p^.prev^.next:=p;
    p^.next^.prev:=p;  {вставили его назад}
    p:=p^.next; {перешли к следующему}
    inc(j);
end;
end;
```

Обратите внимание на то, что элемент удаляется из списка, но не из памяти. Его адрес остаётся в переменной p, благодаря чему мы можем его восстановить назад (в этом смысле роль переменной p в некотором смысле похоже на роль переменной x в первоначальной версии реализации). Обычно вставка элемента в список делается немного хитрее, здесь же мы воспользовались тем, что этот элемент там только что был и мы его просто возвращаем на место.

За счёт связных списков мы смогли избежать циклов в процедурах *insert* и *delete*, что должно сильно (порядка в N раз) ускорить программу. Но, как уже говорилось, это не имеет отношения к перебору, а только к тому, что, если вы хотите вставлять/удалять элементы в произвольное место, то лучше использовать список, а не массив.

Кроме того, обратите внимание на переменную j. Ею мы просто считаем элементы списка, чтобы знать, что записать в массив a.

Задание 2.17 (стр. 26): Собственно, все просто.

Если поддерживать сумму чисел:

```
var s:...
procedure find(i:integer);
var j:integer;
    ocur:...
    ob:..
begin
if nn=2 then begin
   check;
   exit;
end;
if cur+2*s>=best then exit;
ocur:=cur;
ob:=b;
for j:=2 to nn-1 do begin
    a[i]:=j;
    cur:=cur+b[j]*(b[j-1]+b[j+1]);
    s:=s-b[j];
    delete(j);
    find(i+1);
    b:=ob;
    cur:=ocur;
    s:=s+b[j];{Не забываем ее восстанавливать}
end;
end;
```

Если вычислять каждый раз заново:

```
procedure find(i:integer);
var j:integer;
    ocur:...
    ob:..
    s:...
begin
if nn=2 then begin
   check;
   exit;
end;
s:=0;
for j:=2 to nn-1 do
    s:=s+b[j];
if cur+2*s>=best then exit;
ocur:=cur;
ob:=b;
for j:=2 to nn-1 do begin
    a[i]:=j;
    cur:=cur+b[j]*(b[j-1]+b[j+1]);
    s:=s-b[j];
    delete(j);
    find(i+1);
    b:=ob;
    cur:=ocur;
    s:=s+b[j];{Не забываем ее восстанавливать,
       т.к. она нам понадобится при следующем j}
end;
end;
```

Задание 2.18 (стр. 26): Я думаю, напи́шите, ничего тут сложного нет.

Задание 2.19 (стр. 27): Например, следующий тест:

2 1 100 1

Наш жадный алгоритм сначала удалит единицу со штрафом $1 \cdot (2 + 100) = 102$, а потом будет вынужден удалять сотню со штрафом $100 \cdot (2 + 1) = 300$. На самом же деле ясно, что сотню надо удалять, пока с ней соседствуют единицы, т.е. можно *сначала* удалить сотню со штрафом $100 \cdot (1 + 1) = 200$, а потом единицу со штрафом $1 \cdot (2 + 1) = 3$, получив намного меньший суммарный штраф.

Кстати, этот пример указывает на то, что наш алгоритм очень тупой. Ниже в основном тексте я говорю о других идеях «а-ля жадных» алгоритмов.

Задание 2.20 (стр. 27): Ответ писать не буду, надо просто сделать то, что сказано в основном тексте: взять код перебора, который был раньше, и запустить жадность перед перебором.

Задание 2.21 (стр. 27): Ответа тоже не будет.

Задание 2.22 (стр. 28): Элементарно аналогично приведённому в тексте коду :)

```
procedure find(i:integer);
var j,k:integer;
    x:integer;
    was:array...
    minj:integer;
    min:integer;
begin
if nn=2 then begin
   check;
   exit;
end;
fillchar(was,sizeof(was),0);
for k:=2 to nn-1 do begin
    min:=inf;  {бесконечность}
    for j:=2 to nn-1 do
        if (was[j]=0)and(b[j]*(b[j-1]+b[j+1])<min) then begin
           min:=b[j]*(b[j-1]+b[j+1]);
           minj:=j;
        end;
    was[minj]:=1;
    a[i]:=minj;
    cur:=cur+b[minj]*(b[minj-1]+b[minj+1]);
    x:=delete(minj);
    find(i+1);
    insert(minj,x);
    cur:=cur-b[minj]*(b[minj-1]+b[minj+1]);
end;
end;
```

Задание 2.23 (стр. 28): Программу писать не буду, пишите сами.