Наумов Лев Александрович

МЕТОДЫ РАЗБИЕНИЯ ЗАДАЧ НА ПОДЗАДАЧИ. РЕКУРСИЯ, «РАЗДЕЛЯЙ И ВЛАСТВУЙ» ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Какие задачи решать труднее всего? Глупый вопрос, конечно же – сложные. За этим вопросом напрашивается другой, не менее глупый: решать то их, конечно, сложно, но, все-таки, как? На него ответить вообще нельзя. Все зависит от конкретной задачи. Однако мы попытаемся здесь описать один наиболее общий метод: их можно разбивать на «меньшие» подзадачи, которые уже решаются легче. В начале необходимо оговорить, что под словом «сложные» в данном случае мы будем понимать не сложные для программиста, а сложные для компьютера, трудоемкие задачи.

Мы предложим различные методы разбиения задачи на подзадачи. Причем это разбиение должно, очевидно, характеризоваться следующим свойством: решив все получившиеся подзадачи, мы должны получить решение исходной. Как же этого достичь? Основной принцип заключается в явном выделении рекуррентных соотношений, зависимостей от уже полученных результатов. А может быть, просто, удастся выделить подзадачи, которые работают лишь с подмножеством исходных данных, а не со всем набором. Вариантов много. Однако основное внимание будет уделено одному из таких приемов – методу динамического программирования.

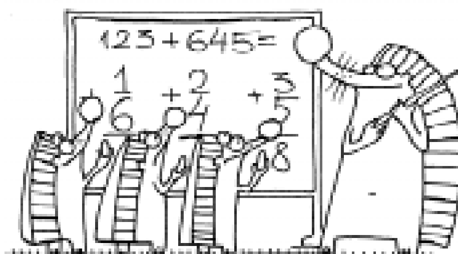
Прежде, чем перейти к основной части повествования, введем обозначение. Если в тексте Вы встретите такую запись: < k > (где k – натуральное число), то это будет означать, что Вам был изложен весь материал, достаточный для решения задачи k. Список задач приведен в крнце статьи.

...прямо или косвенно обращается сама к себе.

Начнем, пожалуй, с самого простого приема, с рекурсии. Скорее всего, Вы с ним уже знакомы, но так как мы в дальнейшем будем активно его использовать, то позвольте все-таки о нем рассказать.

Этот механизм основан на использовании так называемой рекурсивной функции. Она обладает тем свойством, что прямо или косвенно обращается сама к себе. Рекурсией полезно пользоваться, если задачу можно разбить на подзадачи, решаемые за «разумное время», и если при этом суммарный размер подзадач – «небольшой». В предыдущем предложении так много кавычек из-за того, что общего определения «разумного времени» и «небольшого размера» дать нельзя. Значения, вкладываемые в эти понятия, зависят от задачи, целей, которые преследуются, приоритетов при построении решения и многого другого.

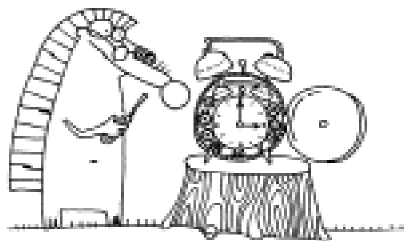
Когда и как использовать этот метод? Как правило, поступают так: рекурсивная функция принимает ряд параметров, описывающих те действия, которые ей надлежит выполнить. Ее «рекурсивность» обеспечивается тем, что из нее вызывается она же, но с другими значениями пара-



...различные методы разбиения задачи на подзадачи.

метров. Главное – не забыть проконтролировать, чтобы процесс когда-нибудь закончился. Для этого нужно, чтобы при некоторых значениях параметров функция больше не вызывала себя. Эти значения называются терминальными.

Продемонстрируем работу рекурсивного метода. В качестве примера рассмотрим вычисление суммы всех целых степеней *x* от 0 до некоторого *n*, сосчитаем значение функции $f(x,n) = x^n + x^{n-1} + \ldots + x + 1$. Эта функция обладает определенный свойством, позволяющим нам применить рекурсию. Оно выражается в следующем равенстве: $f(x,n) = x \cdot (x^{n-1} + x^{n-2} \ldots + x + 1) + 1$. Вы убедитесь в этом, если просто вынесете *x* за скобку из всех слагаемых, кроме последнего. Фактически, оно означает, что $f(x,n) = x \cdot f(x,n-1) + 1$, а это и есть рекуррентное соотношение, позволяющее перейти от решения задачи для $n-1$ к решению для *n*.



...рекуррентное соотношение, позволяющее перейти от решения задачи для n –1 к решению для n.

Остается один вопрос: а какое значение *n* считать терминальным? Ну, подумайте сами, результат возведения в какую степень нам всегда известен заранее, независимо от числа *x*? Конечно – в нулевую. Так пусть терминальным значением и будет $n = 0$.

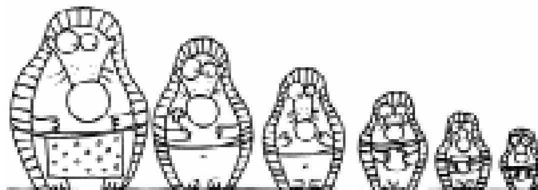
Соответствующая рекурсивная процедура будет иметь вид:

```
function f(x:real; n:integer):real;
begin
     if n=0 then begin
          f:=1;
          exit;
     end;
     f:=x*f(x, n-1)+1;
end;
```

На самом деле Вас «обманули», и мы надеемся, что Вы это уже заметили сами. Мотивация выбора нуля, как терминального значения параметра, совсем иная. Действительно, мы всегда знаем, что $f(x,0) = 1$. Но, посудите сами, разве нам не известно, что $f(x,1) = x + 1$?, что $f(x,2) = x^2 + x + 1$? Конечно, известно! Просто если бы мы выбрали, например, двойку как терминальное значение, то подсчитать $f(x, 0)$ и $f(x, 1)$ мы бы уже не смогли. В этой задаче стоило выбирать наименьшее *n*, для которого может потребоваться вычисление значения функции.

Помимо того, что терминальные значения должны быть предусмотрены, они должны еще и достигаться при рекурсивном вычислении. В приведенном выше примере это, очевидно, выполняется. Ведь начав с любого *n*, мы, постоянно вычитая из него единицу, рано или поздно, доберемся до нуля. Тем самым, «глубина рекурсии» у нас ограничена и равна $n + 1$. Вызов $f(n)$ приведет к вычислению значения требуемой функции посредством вызова *f* самой из себя, *n* раз. То есть для всех значений второго параметра от $n - 1$ до 0. При этом глубина рекурсии равна $n + 1$, так как надо посчитать еще и сам первый вызов $f(n)$. ◂ 1, 2 ▸

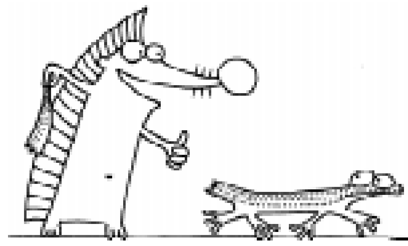
А вообще говоря, почему глубина рекурсии должна быть ограничена? Дело в том, что при программировании нужно исходить из того, что чрезмерная глубина может вызвать ошибку. Если функция будет бесконечно вызывать и вызывать себя же, то на определенном этапе это приведет к переполнению стека. При этом, выполнение программы, несомненно, не сможет продолжаться. О подробностях природы этого эффекта мы здесь говорить не будем. Однако, наверное, будет полезно упомянуть, что, например, в среде программирования *Borland*



...почему глубина рекурсии должна быть ограничена?

Pascal 7.0 есть средства, позволяющие следить и предотвратить возникновение этой ошибки. Открыв при отладке программы окно *Call stack* (это можно сделать через главное меню *Debug* | *Call stack* или сочетанием клавиш <*Ctrl*> + <*F3*>) Вы сможете наблюдать за последовательностью вызовов функций и изменением их параметров. Если при определенных обстоятельствах возникает ошибка *«Stack overflow error»*, значит, Вам есть еще над чем поработать в реализуемом рекурсивном алгоритме. Или, как минимум, нужно наложить некоторые ограничения на область значений параметров. ◀ 3 ▶

Следующий метод называется «разделяй и властвуй». Он заключается также в разбиении задачи на части, каждая из которых решается отдельно. Затем решения объединяются, и получается решение всей задачи. Это – очень общее определение механизма «разделяй и властвуй». Отметим один частный случай: один из вариантов довольно эффективного использования данной методики, а именно, ее применение к задачам, подзадачи которой являются ее же «уменьшенными версиями», причем такими, что их, в свою очередь, можно решать рекурсивно. Под «уменьшенными версиями», как говорилось выше, имеются в виду задачи, аналогичные исходной, только обрабатывающие не все множество входных данных, а лишь некоторое его подмножество. Принципиально она не меняется, но за счет этого «уменьшения» она может упроститься и алгоритмически.



Он заключается также в разбиении задачи на части...

Рассмотрим сказанное выше поподробнее. Например, пусть какая-то задача оперирует с набором из n элементов. Было бы здорово разбить его на q наборов из меньшего числа элементов. Причем так, чтобы ни один из элементов не принадлежал более чем одному набору.

Итак, предположим, что набор разделяется на q непустых поднаборов, содержащих $n_1, n_2, \dots, n_q$ элементов, соответственно. Так как $n_1 + n_2 + \dots + n_q = n$, то $n_i < n$ для любого i от 1 до q. Если исходную задачу переформулировать таким образом в q подзадач (по одной для каждого поднабора), решить каждую из них отдельно и объединить решения, то получится результат, построенный методом «разделяй и властвуй».

Упомянутый выше частный случай достигается, когда каждую из сформулированных задач для n_i ($i = 1, 2, \dots q$) можно также решить разбиением соответствующих поднаборов на более мелкие. Проделывая такую операцию рекурсивно до определенного предела (терминального значения длины поднабора), Вы можете получить решения подзадач для наборов из нескольких элементов. Затем уже легко построить решение всей задачи как некое объединение этих решений.

Рассмотрим пример применения данного механизма. Обсудим, так называемую сортировку слиянием. Пусть, например, нам дан набор из восьми чисел $T = \{7, 6, 3, 7, 9, 2, 1, 4\}$. Наша задача заключается в том, чтобы отсортировать их, скажем, по возрастанию. Согласно методу «разделяй и властвуй», разделим его пополам, получим два поднабора $T_1 = \{7, 6, 3, 7\}$ и $T_2 = \{9, 2, 1, 4\}$ и будем над ними властвовать! А точнее, будем сортировать уже их, каждый отдельно. Как мы будем это делать? Конечно же, опять поделив и тот и другой наборы пополам (таким образом, мы получим $T_{11} = \{7, 6\}$, $T_{12} = \{3, 7\}$, $T_{21} = \{9, 2\}$ и $T_{22} = \{1, 4\}$). Деление продолжим рекурсивно до того момента, пока не получим наборы из одного единственного элемента ($T_{111} = \{7\}$, $T_{112} = \{6\}$, $T_{121} = \{3\}$, $T_{122} = \{7\}$, $T_{211} = \{9\}$, $T_{212} = \{2\}$, $T_{221} = \{1\}$ и $T_{222} = \{4\}$). Естественно, такие наборы можно сразу считать отсортированными. Тогда объединим два упорядоченных набора. Причем так, чтобы их объединение опять было упорядоченным ◀ 4 ▶. Проделаем это рекурсивно до тех пор, пока снова не получим набор, равный по длине исходному. На-

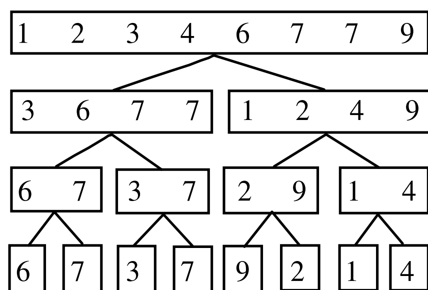
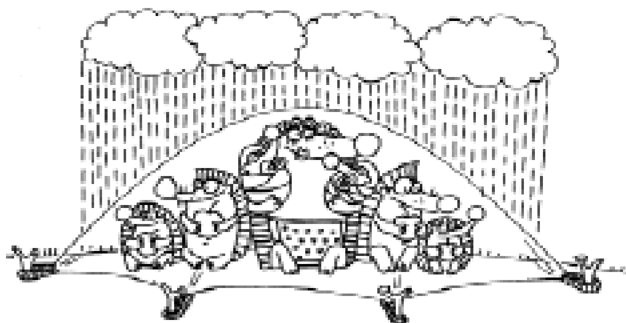


Рисунок 1



...о выпуклой оболочке на плоскости.

деемся, рисунок 1 поможет Вам понять, как именно должно осуществляться объединение наборов.

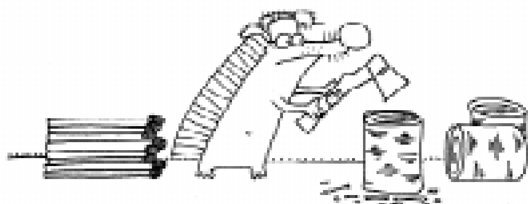
Механизм слияния наборов мы рассматривать не будем, так как он не имеет непосредственного отношения к обсуждаемой теме. ◂ 5 ▸

Выше мы рассказывали о том, что при решении задач рекурсивным методом существует опасность вызвать переполнение стека. Поэтому нужно стараться минимизировать глубину рекурсии. Читатель, наверное, согласится с тем, что разбивать исходный набор вплоть до поднаборов длины 1 вряд ли имеет смысл. Ведь набор из двух элементов уже можно упорядочить, проверив одно условие. Конечно, можно проверкой некоторого количества разных условий отсортировать и набор из четырех элементов. Однако это, пожалуй, уже лишнее. Так можно «вручную» сортировать наборы из любого наперед заданного числа элементов. ◂ 6 ▸ Механизм сортировки наборов с длиной, меньшей терминального значения, тоже можно предусмотреть.

А вот другой пример применения принципа «разделяй и властвуй». Рассмотрим часто встречаемую в программировании задачу о выпуклой оболочке на плоскости. Для конечного множества точек Q выпуклой оболочкой называется наименьший по площади выпуклый многоугольник, содержащий все точки из Q. Фактически, задача заключается в отыскании некоего подмножества Q (обозначим его S), все точки которого являются вершинами упомянутого многоугольника. Поясним сказанное выше примером. Пусть $Q = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ (здесь числа – номера, индексы, идентификаторы точек, не более того). Точки располагаются так, как указано на рисунке 2. Их координаты задают их расположение. Вряд ли стоит приводить здесь некие конкретные числа, так как наша задача заключается в том, чтобы объяснить принцип, алгоритм решения.

Итак, выпуклой оболочкой является многоугольник, вершинами которого являются точки множества S. В данном случае $S = \{1, 4, 5, 6, 7, 9\}$. Соответствующий многоугольник показан на рисунке 3. Все оставшиеся точки (то есть те, которые принадлежат Q, но не принадлежат S) располагаются внутри этого многоугольника.

Для построения выпуклой оболочки существует множество методов: просмотр



...разбивать исходный набор ... до поднаборов ...

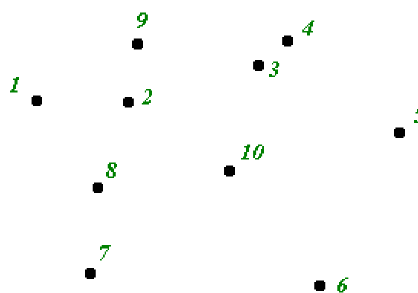


Рисунок 2

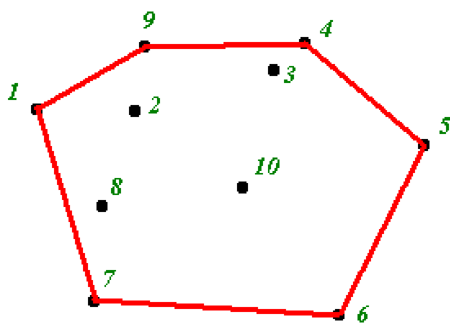


Рисунок 3

...разделить имеющееся множество ... на два подмножества, приблизительно равных...

Грэхема, проход Джарвиса, метод добавления точек, «метод стрижки и поиска». Если Вас это заинтересовало, то о них Вы можно узнать из книги [1]. Мы же остановимся лишь на решении задачи приемом «разделяй и властвуй». Все происходит абсолютно так же, как при сортировке слиянием. Сначала необходимо разделить имеющееся множество точек на два подмножества, приблизительно равных по мощности (числу элементов). Причем сделать это нужно так, чтобы ни один из многоугольников, построенных на неких точках одной половинки как на вершинах, не пересекался ни с каким многоугольником, построенным на точках другой половинки. Достичь этого можно, разделив плоскость вертикальной прямой на две зоны.

Затем следует получить решение для обеих половинок. Результат представлен на рисунке 4.

Естественно, что задачу для каждой зоны нужно решать так же, разделяя ее на две подзоны. Разделение следует продолжать рекурсивно до тех пор, пока в одной зоне не окажется лишь 3 точки, для которых выпуклая оболочка получается простым их соединением. Конечно, если в зоне было, например, всего 5 точек, то отделив 3, Вы получите зону в которой – 2 точки. Что же делать? Прочитайте описание излагаемого метода до конца и подумайте, является ли это проблемой.

Теперь остается объединить решения, что на самом деле не очень просто. Для двух зон, изображенных на рисунке 4, получим рисунок 5.

Здесь результирующий прямоугольник обведен сплошными отрезками. Трудность

...задачу для каждой зоны нужно решать так же, разделяя ее на две подзоны.

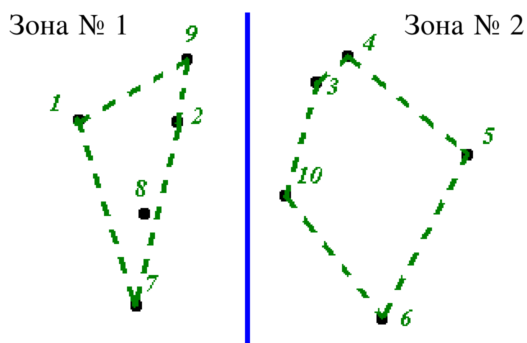


Рисунок 4

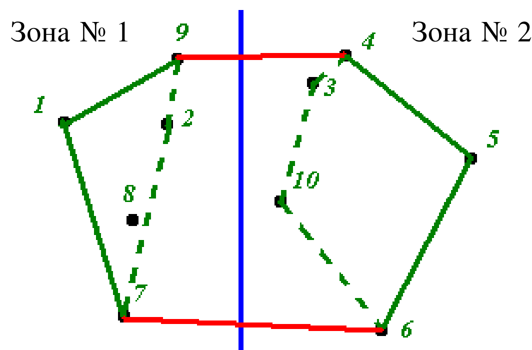
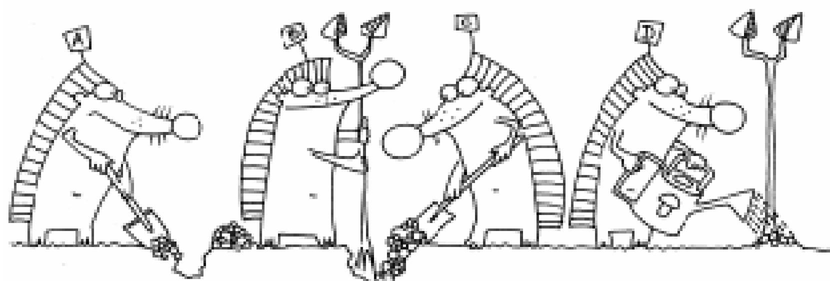


Рисунок 5



...задачу можно разбить на подзадачи, решаемые за «разумное время»...

при объединении заключается в выборе соединяемых точек. Почему мы предпочли отрезок 9–4 отрезкам 9–3, 9–10 и остальным? Дело в том, что нужно производить соединения так, чтобы все точки объединяемых многоугольников лежали по одну сторону от проводимой прямой, и чтобы получающийся многоугольник был минимален. Над этим правилом Вам предстоит подумать в задаче 7.

Ранее говорилось, что задача о выпуклой оболочке «часто встречается в программировании». Не надо из этого делать вывод, что программисты нередко решают задачи планиметрии. Просто многие проблемы из самых разных областей сводятся к построению выпуклой оболочки для некоторого множества, природа которого далека от геометрической, и понятие координат его элементов заменено на что-то специфические для данной задачи. ◂ 7, 8, 9, 10 ▸

Нельзя не отметить, что применительно к задаче 8 этот метод дает существенный выигрыш в эффективности. А именно, он требует $1{,}5n - 2$ сравнений, в то время как обычный способ вычисления в цикле требует $2n - 3$ ($n - 1$ на поиск минимума и $n - 2$ на поиск максимума; минимум в рассмотрении уже не участвует).

Метод «разделяй и властвуй» зачастую повышает эффективность на порядки. Чтобы познакомится с примерами, можно обратиться к книге [2]. Мы же переходим к основной теме – динамическому программированию.

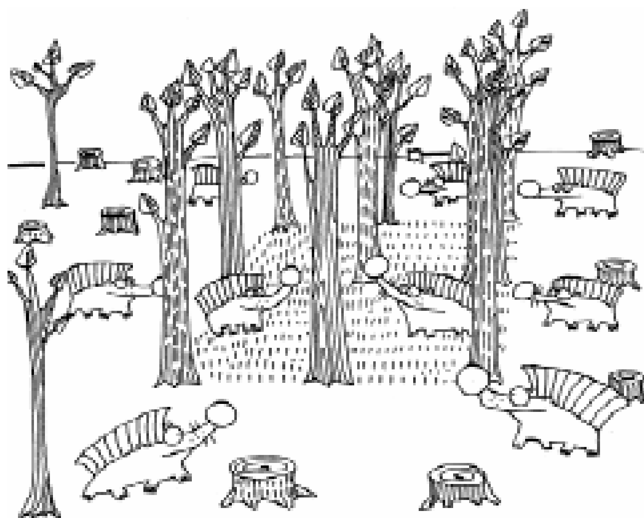
Рекурсия, как уже говорилось, полезна, когда задачу можно разбить на подзадачи, решаемые за «разумное время», причем так, чтобы суммарный размер этих подзадач был «небольшим». Но что делать, если разбиения задачи размера n приводит к n задачам, размера $n - 1$? В таких ситуациях зачастую можно получить более эффективные алгоритмы методом динамического программирования.

Как и техника «разделяй и властвуй», динамическое программирование позволяет построить решение общей задачи, разбивая ее на подзадачи и объединяя их решения. Однако, как было продемонстрировано выше, при использовании метода «разделяй и властвуй» нужно производить разбиение на системы независимых подзадач. А при динамическом программировании необходимо разделение на взаимосвязанные подзадачи. Дробить следует так, чтобы у подзадач были общие части, общие «подподзадачи», если угодно. Если бы в такой ситуации мы использовали метод «разделяй и властвуй», то эти самые «подподзадачи» решались бы по нескольку раз. Динамический метод, решив ее однажды, запоминает результат в специальном хранилище, как правило, – таблице. Когда этот результат понадобится, будет достаточно лишь обратиться к соответствующей ее ячейке.

Давайте вместе рассмотрим следующую задачу. Пусть нам дана таблица F, состоящая из n строк и m столбцов. В каждой ячейке таблицы может содержаться либо «0», либо «1». От нас требуется найти сторону наибольшего квадрата из единиц, который есть в этой таблице.

Поясним сказанное примером F для n и m равных 6.

0	1	1	1	1	0
0	1	1	1	1	0
1	1	1	1	0	1
0	0	1	1	1	0
1	0	1	0	0	1
0	0	0	1	0	0



...требуется найти сторону наибольшего квадрата из единиц...

Если бы на вход алгоритма, который мы должны построить, подали бы такую таблицу, то для нее нужно было бы получить ответ 3. Искомый квадрат выделен в таблице.

Как же ее решать? Очевидно, полный перебор всевозможных вариантов – дело не очень перспективное, особенно при больших n и m.

Предлагается следующее решение: заведем таблицу D, равную по размеру F. В каждую ее клетку $[i, j]$ поместим величину стороны квадрата из единиц, имеющего нижний правый угол в клетке $F[i, j]$. Ясно, что там, где в F были нули, там и в D будут нули.

Для приведенной выше исходной таблицы вспомогательная таблица D должна выглядеть так:

0	1	1	1	1	0
0	1	2	2	2	0
1	1	2	**3**	0	1
0	0	1	2	1	0
1	0	1	0	0	1
0	0	0	1	0	0

Очевидно, наша задача свелась к вычислению наибольшего значения в таблице D. Очень хорошо! Но для начала не мешало бы выяснить, как эта таблица строится. Это делается очень просто. Надеемся, тот факт, что первая строка и первый столбец просто переносятся из F в D, – очевиден. Ведь на них мы можем обнаружить либо квадрат со стороной равной единице (то есть просто – «1»), либо ноль, который вообще не может являться «нижним правым углом квадрата из единиц».

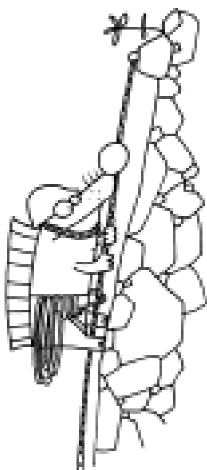
Как же заполнять D дальше, при $2 \le i \le n$ и $2 \le j \le m$? Нули, как уже неоднократно говорилось, просто переписываются. А что делать, если $F[i, j]$=1? Ясно, что $D[i, j]$ отлична от единицы, только если ее левый ($D[i, j-1]$), верхний ($D[i-1, j]$) и левый верхний ($D[i-1, j-1]$) соседи содержат не ноль. Причем содержимое клетки $F[i, j]$ может продлить лишь наименьший из квадратов, упирающихся в его соседей. Формально, это означает, что $D[i, j] = min(D[i-1, j], D[i, j-1], D[i-1, j-1]) + 1$. Это и понятно, так как к квадрату, которому соответствует $D[i-1, j]$, могла добавиться, как максимум, только одна строка. К квадрату, учтенному в $D[i, j-1]$, – только один столбец. А к $D[i-1, j-1]$ – одна строка и один столбец. Таким образом, размер результирующего квадрата не может превышать $D[i-1, j]+1$, $D[i, j-1]+1$ и $D[i-1, j-1]+1$. Это, пожалуй, вполне исчерпывающе объясняет написанную формулу.

Теперь сведем имеющиеся результаты относительно D воедино. Получим:

$$D[i, j] = \begin{cases} F[i, j], \text{ при } i = 1 \text{ или } j = 1 \\ \begin{cases} 0, \text{ если } F[i, j] = 0 \\ \min(D[i-1, j], D[i, j-1], D[i-1, j-1]) + 1, \\ \qquad \text{если } F[i, j] = 1 \end{cases} \end{cases}$$

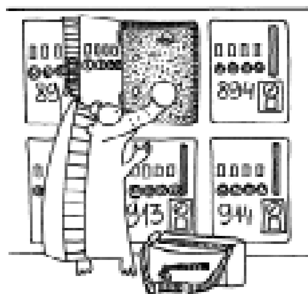
при $2 \le i \le n$ и $2 \le j \le m$.

Как реализовать вычисление D на компьютере? Сначала – заполнить первую строку и столбец. А затем – вычислять в двойном цикле по i и j от меньших индексов – к большим. При таком подходе Вы защищены от того, что может потребоваться значение из ячейки, которое еще не вычислялось. ◂ 11 ▸



«снизу вверх»

Продемонстрированный только что метод решения задачи, иллюстрировал прием динамического программирования «снизу вверх» (явное указание на это заключено во фразе «от меньших индексов – к большим»). Если есть такой способ, то должен быть и способ «сверху вниз»? Совершенно верно. Его отличие состоит в методе работы с таблицей полученных результатов подзадач. А именно: изначально все клетки таблицы инициализируются неким значением, сигнализирующем о том, что информация для данной клетки еще не вычислялась. Условно обозначим его как ∞. По ходу решения задачи, естественно, происходит обращение к определенным ячейкам таблицы. Если значение в требуемой ячейке равно ∞, то вызывается рекурсивная функция, вычисляющая это значение. В противном случае просто берется соответствующая информация из определенной клетки таблицы. Такой метод больше тяготеет к названию «рекурсия с сохранением результатов», чем «динамическое программирование сверху вниз», но с классической терминологией спорить не будем.



...обращение к определенным ячейкам таблицы.

Как правило, метод «снизу вверх» более или, по крайней мере, не менее эффективен, чем «сверху вниз», если каждая из подзадач должна быть решена несколько раз. Просто метод «сверху вниз» требует много дополнительного времени на разнообразные проверки наличия информации в таблице и тому подобное. К тому же, он увеличивает глубину рекурсии. Ведь на любом ее уровне может внезапно потребоваться еще более глубокое «погружение» для дополнения таблицы требуемой информацией. В то время как метод «снизу вверх» характеризуется тем, что мы гарантированно на каждом шаге имеем всю необходимую для дальнейших вычислений информацию.

Тем не менее, если метод «сверху вниз» существует, значит он кому-нибудь нужен. Дело в том, что получаемый алгоритм будет несколько понятнее. Он позволит однозначно ответить на вопрос «откуда что берется». Какие значения, на каком этапе выполнения вычисляются. Это особенно актуально, если последовательность вычисления значений сложна.

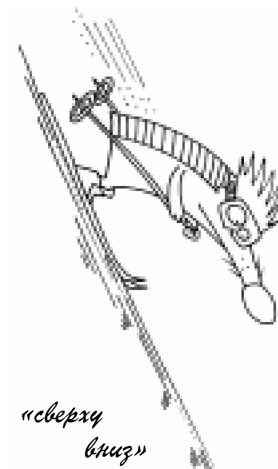
Сказанное выше не означает, что этот метод не стоит использовать. Наоборот, предыдущий абзац, скорее, – лишний аргумент в пользу его применения. К тому же, он полезен, когда не все подзадачи решаются несколько раз. ◂ 12 ▸

На практике очень часто встречается задача о выделении наибольшей общей подпоследовательности. Давайте рассмотрим и ее решение методом динамического программирования. Основное свойство этой задачи, которое мы будем при этом использовать, заключается в следующем:

Для нахождения наибольшей общей подпоследовательности последовательностей X и Y нужно:

1) Если последний элемент последовательности X совпадает с последним элементом последовательности Y, то решать задачу для этих последовательностей без последних элементов, а потом приписать этот элемент к результату.

2) Если последний элемент последовательности X не совпадает с последним элементом последовательности Y, решать две задачи. Первая – для последовательности X без последнего элемента



«сверху вниз»

и последовательности Y. Вторая – для последовательности Y без последнего элемента и последовательности X. Наибольший из результатов и будет наибольшей общей подпоследовательностью.

Таким образом, получилось очевидное перекрытие подзадач: ведь наибольшую общую подпоследовательность для X и Y без последних элементов придется искать, как в случае 1, так, вероятно, и в случае 2. Очевидно, что в этой ситуации имеет смысл сохранять результаты вычислений в таблице. Обозначим за $c\,[i, j]$ длину наибольшей общей подпоследовательности для последовательности X, состоящей из i элементов, и последовательности Y – из j элементов. Тогда приходим к соотношению:

$$c[i,j] = \begin{cases} 0, \text{ при } i = j = 0, \\ c[i-1, j-1] + 1, \ \text{ если } i, j > 0 \\ \quad \text{и последние элементы совпадают,} \\ \max(c[i, j-1], c[i-1, j]), \ \text{ если } i, j > 0 \\ \quad \text{и последние элементы не совпадают.} \end{cases}$$

Как Вы, должно быть, уже догадались, $c\,[i, j]$ и составляет содержание одноименной таблицы, необходимой для корректной работы метода динамического программирования. Здесь $i = 0, 1, \dots, m$, а $j = 0, 1, \dots, n$.

Кроме того, потребуется вторая таблица b, в которой $b\,[i, j]$ хранит «происхождение» $c\,[i, j]$. Ведь возможны 3 варианта появления конкретного значения $c\,[i, j]$: от $c[i-1, j-1]$, от $c[i, j-1]$ или от $c[i-1, j]$. Очень красивым и экономичным решением для организации b было бы такое, при котором на каждую ячейку отводилось бы 2 бита. Таким образом, одна ячейка могла бы хранить значения от 0 до 3. Этого было бы вполне достаточно (даже одно – лишнее), а интерпретировать их можно было бы так:

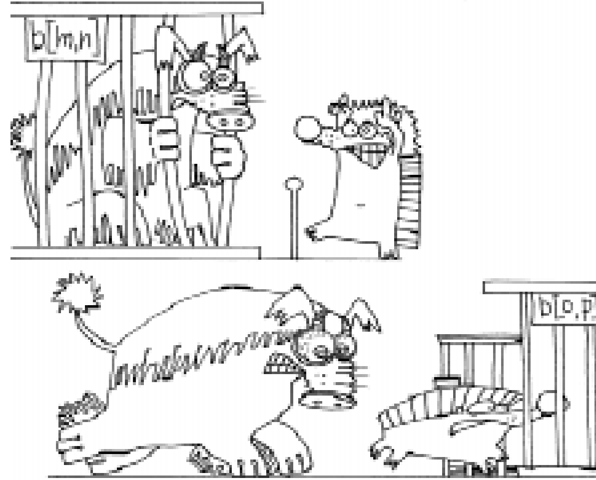
0 – значение для инициализации ячеек b. На самом деле, конечно, оно не нужно, но что ему без дела пропадать ☺.

1 – $c\,[i, j]$ «произошло» от $c\,[i-1, j-1]$.

2 – $c\,[i, j]$ «произошло» от $c\,[i, j-1]$.

3 – $c\,[i, j]$ «произошло» от $c\,[i-1, j]$.

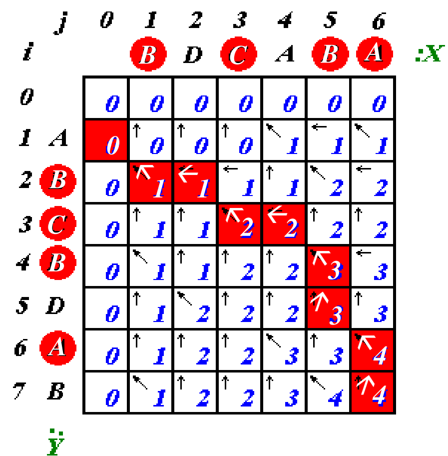
Теперь, пусть алгоритм выполнен и таблицы b и c заполнены. Как получить Z? Нужно из клетки $b\,[m, n]$ проделать путь до



Нужно из клетки b [m, n] проделать путь до некоторой клетки b [0, p].

некоторой клетки $b\,[0, p]$ или $b\,[r, 0]$ (для какого-нибудь p или r), выписывая одинаковые члены последовательностей X и Y. Звучит угрожающе, не правда ли? Посмотрите на рисунок 6, и, надеемся, Вам станет понятнее.

На рисунке представлено некое совмещение информации таблиц b и c для $X = \{A, B, C, B, D, A, B\}$ и $Y = \{B, D, C, A, B, A\}$. Число в каждой клетке соответствует значению $c\,[i, j]$ для нее. Стрелочка – некое наглядное представление информации ячейки $b\,[i, j]$. Она указывает на ту ячейку, от которой «произошло» $c\,[i, j]$. Такое представление очень удобно для построения пути из $b\,[m, n]$. Оно просто заключается в прохождении по этим стрелкам до тех пор, пока не будет достигнута граница матрицы. Клет-



i \ j		0	1	2	3	4	5	6
			B	D	C	A	B	A :X
0		0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4
	:Y							

Рисунок 6

Оно ... заключается в прохождении по этим стрелкам до тех пор, пока не будет достигнута граница матрицы.

ки, входящие в этот путь, на рисунке выделены. Также выделены члены X и Y, которые войдут в Z. Это – такие члены, которые одинаковы для данной клетки пути возвращения из $b\,[m,\ n]$. В результате операций, производимых на этом рисунке, делается вывод, что $Z = \{B, C, B, A\}$. ◂ 13 ▸

Основное отличие задачи о квадрате из единиц, рассмотренной выше, от задачи о НОП заключается в том, что последняя требует восстановления решения по составленной матрице. Такие задачи возникают довольно часто и иногда требуют некоего дополнительного хранилища информации для «описания пути», по которому пришлось пройти при построении решения. Однако зачастую, если хорошо продумать алгоритм, можно найти путь, на котором можно обойтись без этого хранилища. ◂ 14 ▸

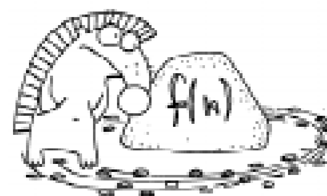
На этом, пожалуй, и закончим. Обратим внимание, что здесь не было противопоставления методов рекурсии, «разделяй и властвуй» и динамического программирования. Задача была в том, чтобы показать насколько велики спектры покрываемых ими задач, каковы критерии выбора того или иного метода, а также как они сочетаются между собой.

В общем, мы постарались рассказать, как решать «сложные» задачи. Предложенные приемы – сильное оружие, и поэтому они могут иметь как позитивный, так и негативный эффект от использования.

Большинство материалов, примеров и определений были взяты из книг [1] и [2].

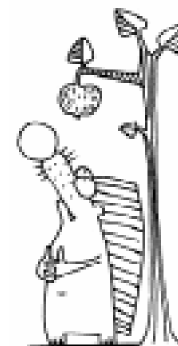
Задача 1.

Уровень 1. Оцените, каково преимущество метода рекурсии при вычислении $f\,(n)$ указанным выше способом перед обыкновенным последовательным подсчетом в цикле. Если Вы полагаете, что никакого преимущества нет, то аргументируйте Вашу точку зрения.



Задача 2.

Уровень 1. Придумайте, как рекурсивным методом сосчитать факториал числа. Факториалом числа n называется произведение всех натуральных чисел от 1 до n. Он обозначается $n!$. По определению, 0! считается равным единице.



Существенен ли выигрыш при решении этой задачи рекурсивным методом по сравнению с решением «в лоб». А может быть, последнее решение оказывается эффективнее первого?

Какие ограничения на n накладывает придуманный Вами алгоритм? Какое значение Вы выбрали терминальным?

Уровень 2. Напишите программу, которая его реализует. Организуйте запрос числа n, скажем, из файла, а результат выводите в другой файл.

Входной файл:

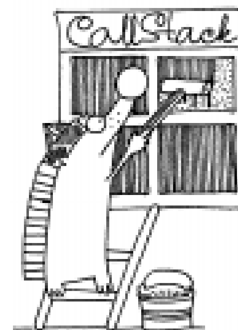
7

Выходной файл:

5040

Задача 3.

Уровень 2. Исследуйте написанное Вами решение задачи 2 с помощью окна *Call Stack*. Оно есть не только в *Borland Pascal*, а практически во всех средах программирования на языках Pascal, C и тому подобных.



Какую глубину рекурсии имеет Ваш алгоритм?

Задача 4.

Уровень 1. Придумайте наиболее эффективный алгоритм слияния двух упорядоченных наборов, причем так, чтобы их объединение опять было упорядоченным. Результатом этого задания Вам предстоит пользоваться в дальнейшем, так что подумайте хорошо. Оцените трудоемкость предложенного Вами алгоритма.

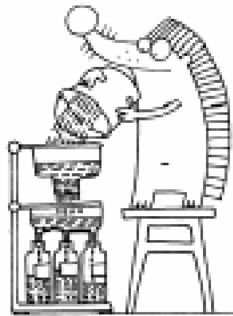
Задача 5.

Уровень 1. Сформулируйте алгоритм сортировки слиянием для случая, когда число элементов во входном наборе не является степенью двойки. Оцените трудоемкость алгоритма.

Уровень 2. Напишите программу, реализующую этот алгоритм. Число элементов и содержимое входного набора прочитайте из входного файла. Результирующий отсортированный набор поместите в выходной файл.

Входной файл:

9
7
6
3
7
9
2
1
4
1



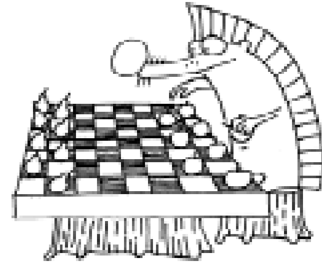
Выходной файл:

1
1
2
3
4
6
7
7
9

Задача 6.

Уровень 1. Как только при помощи проверки определенных условий упорядочить набор из двух элементов Вы, надеемся, догадываетесь. На всякий случай, чтобы Вы вполне поняли задание, приводим программу, которая это реализует:

```
if a<b then begin
     t:=a;
     a:=b;
     b:=t;
end;
```



Здесь мы изначально считали, что элементы набора содержатся в переменных *a* и *b*, а *t* – временная переменная. После выполнения программы наибольший из элементов будет в *a*.

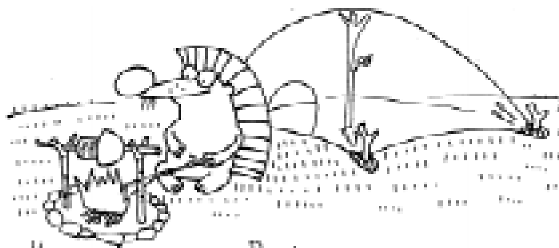
А как сделать то же самое для трех элементов? А для четырех – сможете? Вообще, можете что-нибудь сказать о том, как принципиально нужно решать задачу для *k* переменных?

Если у Вас достаточно опыта, попробуйте написать программу, которая бы для данного *k* строила бы систему «if'ов», необходимую для упорядочивания соответствующего набора. В результате должен получаться компилируемый файл исходного кода на языке Pascal, C или другом языке программирования.

Задача 7.

Уровень 1. Четко сформулируйте алгоритм решения задачи о выпуклой оболочке методом «разделяй и властвуй». Каким образом можно улучшить описанный в тексте метод?

Уровень 2. Реализуйте сформулированный алгоритм. Во входном файле на первой строчке – натуральное число *n*, количество элементов в *Q*. Далее – *n* строк с координатами точек по возрастанию их номеров. То есть во второй строке – координаты первой точки, в третьей – второй и так далее. Ко-



ординаты записываются так: абсцисса, пробел, ордината. В выходной файл поместите список точек множества *S*, отсортированный по возрастанию номеров точек.

Входной файл:

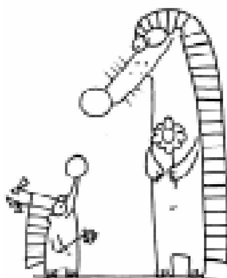
10
–6 3
–3 3
1 4.2
2 5.2
5.5 1.6
3 –4.5
–4.3 –4.1
–0.5 –4.05
–2.9 5.5
0 0

Выходной файл:

1
4
5
6
7
9

Задача 8.

Уровень 1. Придумайте, как можно методом «разделяй и властвуй» решить задачу о нахождении минимума и максимума чисел некого набора. Подсказка: разбивайте набор на поднаборы до тех пор, пока их длина не станет равна 2. Полезность этой подсказки весьма сомнительна, так как Вам еще осталось придумать, каким образом организовать эти разбиения. Оцените трудоемкость полученного метода.



Уровень 2. Напишите программу, которая его реализует. Данные считывайте из входного файла следующего вида: в первой строке – *n*, затем – *n* строк, содержащих по одному числу каждая. Именно эти числа Вам и нужно исследовать. В выходной файл в первую строку поместите значение минимума, а во вторую – максимума.

Входной файл:

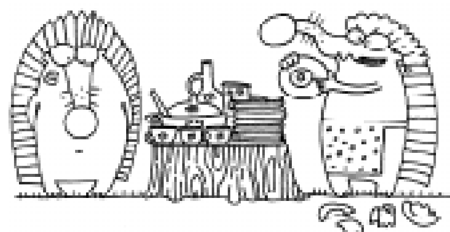
7
1
3
5
–1
–7
7
5

Выходной файл:

–7
7

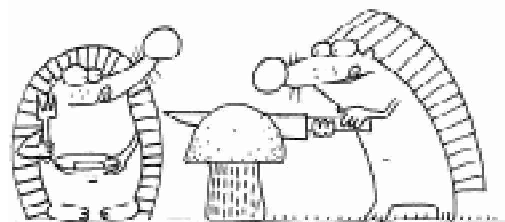
Задача 9.

Уровень 1. Подумайте, по каким законам стоит производить разбиение набора в предыдущей задаче, а по каким нет. Есть ли какие-либо принципиально разные способы, которые повлекут за собой разное число сравнений при решении задачи. Ответ аргументируйте.



Задача 10.

Уровень 1. Подумайте, стоит ли при решении задачи 8 производить деление вплоть до наборов длины большей 2. Например, рассматривать по 3, 4, 5, ... элементов и решать подзадачи для именно такого их количества. Ответ аргументируйте. Если Вы уже решили задачу 6, то Вам будет существенно легче ответить на этот вопрос.



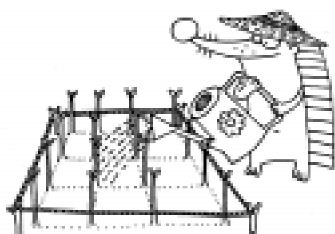
Задача 11.

Уровень 2. Напишите программу, которая бы решила задачу о квадрате из единиц в таблице. Саму таблицу берите из входного файла, в котором в первой строке – *n*

и m. Далее – n строк, по m символов «0» или «1» в каждой. Ответ (длину ребра квадрата) поместите в выходной файл.

Входной файл:

6 6
011110
011110
111101
001110
101001
000100



Выходной файл:

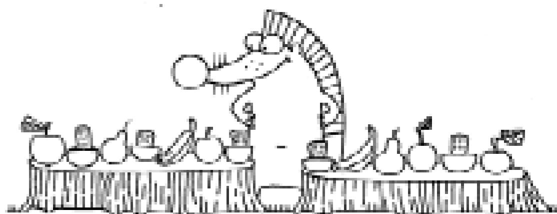
3

Задача 12.

Уровень 2. Напишите программу, которая бы решала предыдущую задачу, но методом динамического программирования «сверху вниз» (просто цикл итерируйте от больших индексов к меньшим). Все вопросы ввода/вывода данных решайте аналогично. Какова глубина рекурсии при таком решении? Применение этого метода к задаче о квадрате из единиц может вызвать у Вас недоумение: а причем он здесь? Да, его использование, конечно, надуманно, но если у Вас уже есть решение «снизу вверх», то изменения, которые необходимо внести, – невелики, однако это поможет Вам почувствовать специфику обоих подходов.

Задача 13.

Уровень 2. Напишите программу, которая реализует алгоритм нахождения НОП двух последовательностей. Допустим, возможные члены последовательностей – заглавные буквы латиницы. Входной файл состоит из двух строк, представляющих собой X и Y. В выходной файл поместите одно из возможных значений Z. Как можно организовать вывод всевозможных Z, когда вариантов – несколько?



Входной файл:

ABCBDAB
BDCABA

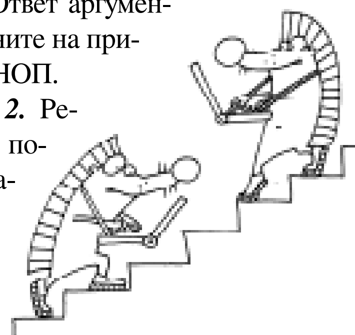
Выходной файл:

BCBA

Задача 14.

Уровень 1. Подумайте, может ли метод динамического программирования «сверху вниз» оказаться полезнее, чем «снизу вверх» при решении задач, требующих восстановления решения. Ответ аргументируйте и поясните на примере задачи о НОП.

Уровень 2. Решите задачу 14 посредством динамического программирования «сверху вниз».



Литература.

1. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построения и анализ. М.: МЦНМО, 2000, с. 288–312, 822–829.

2. *Ахо А., Хопкрофт Д., Ульман Д.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979, с. 70–81, 83–85.

Наши авторы, 2002.
Our authors, 2002.

Наумов Лев Александрович,
студент СПбГИТМО.