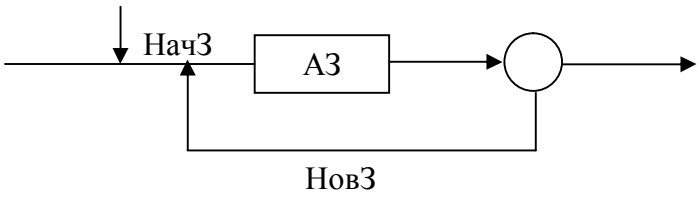
Способы построения эффективных алгоритмов.

Итерация.

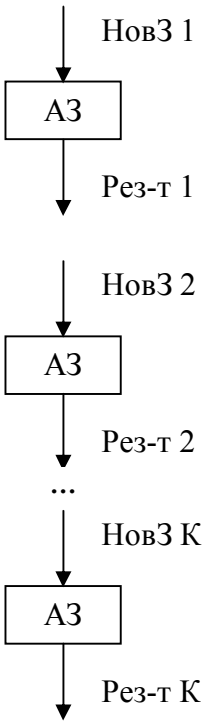
Данный способ построения алгоритма позволяет более наглядно и сжато описывать вычислительный процесс. Выделяют итерацию по значениям и итерацию по аналитическим зависимостям.



Итерацию по значениям можно представить следующим образом:

Примером являются циклы в программе. Этот процесс использует одни и те же аналитические зависимости, и значение величины, полученное на предыдущем шаге, служит для получения следующего значения этой величины.

Итерация по аналитической зависимости.



Одинаковые операции производятся над различными исходными данными. Такие итерации реализуются в программе в виде стандартных подпрограмм или функций.

	Временная сложность	*Емкостная сложность*
Итерация по значениям	Временная сложность возрастает за счёт организации циклов	Емкостная сложность уменьшается пропорционально числу циклов
Итерация по аналитической зависимости	Временная сложность возрастает за счёт организации циклов и передачи данных	Емкостная сложность уменьшается пропорционально числу циклов и числу совпадающих аналитических зависимостей.

Этот метод разработки алгоритмов известен как **метод подъёма**. Алгоритм подъёма начинается с принятия начального предположения или вычисления начального решения задачи. Затем начинается насколько возможно быстрое движение "вверх" от начального решения по направлению к лучшим решениям.

Когда алгоритм достигнет такую точку, из которой больше не возможно двигаться наверх, алгоритм останавливается. К сожалению, не возможно гарантировать, что окончательное решение, полученное с помощью алгоритма подъёма, будет оптимальным. Этот недостаток часто ограничивает применение метода подъёма.

Алгоритмы подъёма часто применяют, когда необходимо быстро получить приближённое решение.

Рекурсия.

Иногда оказывается удобным свести задачу к другой, более простой задаче, а иногда - свести задачу к ней же самой, упростив исходные данные. Способы сведения задачи к ней же самой, но с изменёнными исходными данными, называется **рекурсией**. Алгоритм решения задачи, который содержит обращение к себе, называется **рекурсивным алгоритмом**. Применение рекурсии позволяет получать более эффективные и краткие описания алгоритмов, чем это было бы возможно без рекурсии.

Рассмотрим алгоритм прохождения двоичного дерева во внутреннем порядке. Этот алгоритм присваивает узлам номера в соответствии с внутренним порядком. Он рекурсивно обращается к себе для нумерации поддерева.

Алгоритм 1. Нумерация узлов двоичного дерева в соответствии с внутренним порядком.

Дано двоичное дерево, представлено массивами ЛЕВЫЙ_СЫН и ПРАВЫЙ_СЫН. В результате получим массив НОМЕР, такой, что НОМЕР [i] - номер узла i во внутреннем порядке. Кроме массивов ЛЕВЫЙ_СЫН, ПРАВЫЙ_СЫН и НОМЕР, алгоритм использует глобальную переменную СЧЁТ, значение которой - номер очередного узла в соответствии с внутренним порядком. Начальное значение переменной СЧЁТ=1. Параметр УЗЕЛ вначале равен 0. Процедура применяется рекурсивно.

```
procedure ВНУТРЕННИЙ_ПОРЯДОК (УЗЕЛ):
begin
    if ЛЕВЫЙ_СЫН [УЗЕЛ]≠0 then
        ВНУТРЕНИЙ_ПОРЯДОК(ЛЕВЫЙ_СЫН[УЗЕЛ]);
    НОМЕР[УЗЕЛ] ← СЧЁТ;
```

```
        СЧЁТ ← СЧЁТ + 1;
        if ПРАВЫЙ_СЫН[УЗЕЛ] ≠0 then
            ВНУТРЕННИЙ_ПОРЯДОК(ПРАВЫЙ_СЫН[УЗЕЛ])
end.
```

Эта процедура используется в алгоритме:

```
begin
    СЧЁТ ← 1;
    ВНУТРЕННИЙ_ПОРЯДОК(КОРЕНЬ)
end.
```

Алгоритмы с использованием рекурсии являются более простыми, чем без рекурсии. Если бы приведённый выше алгоритм не был записан рекурсивно, то необходимо использовать механизм, позволяющий вернуться к исходной точке (предку), т.е. стек.

Для оценки временной и емкостной сложности алгоритма необходимо представить его в виде РАМ-программы и оценить сложность. Время, требуемое для вызова рекурсивной процедуры, пропорционально времени, требуемому для вычисления значений фактических параметров и запоминания указателей их значений в стеке. Время возвращения не превосходит этого времени (т.е. времени вызова рекурсивной процедуры).

Время работы алгоритма пропорционально времени вычисления процедур, к которым происходит обращение на каждом шаге работы алгоритма.

Обозначим T_i (n) - время вычисления i-ой процедур, как функцию одного параметра n - размера рассматриваемого входа. Тогда можно установить зависимость T_i (n) = f (T_{i-1} (n)) и получить рекуррентное уравнение для определения T(n) при заданном n. Тем самым получим оценку временной сложности алгоритма.

Использование рекурсии в общем случае, как правило, приводит к сокращению емкостной сложности, и увеличению временной сложности алгоритма.

Декомпозиция: анализ и синтез.

Этот метод связан со сведением трудной задачи к последовательности более простых задач, т.е. выполняется анализ первоначальной задачи о возможности разбиения этой задачи размерности n на k задач с размерностями m_1, m_2, …, m_k, таких, что

$m_1 + m_2 + \ldots + m_k \leq n$

После того как найдены решения k задач, решение первоначальной задачи может быть получено из решений этих более простых задач, т.е. осуществляется синтез частных решений, позволяющий получить решение первоначальной задачи.

Этот метод также называют **методом частных целей** или **"разделяй и властвуй".**

Рассмотрим задачу о нахождении наибольшего и наименьшего элементов множества S, содержащего n элементов.

Алгоритм 1. В начале найдём наибольший элемент множества S. Алгоритм имеет вид:

begin

MAX ← произвольный элемент из S

for все другие элементы X из S **do**

if X>MAX **then** MAX ← X

end.

Эта процедура находит наибольший элемент множества S, произведя (n-1) сравнение его элементов. Аналогично можно найти наименьший из остальных (n-1) элементов, произведя (n-2) сравнения.

Т.О. для нахождения наибольшего и наименьшего элементов при $n \geq 2$ потребуется (n-1) + (n-2) = 2n-3 сравнений.

Алгоритм 2. Рассмотрим применение декомпозиции.

Множество S, состоящее из n элементов, разобьем на два подмножества S_1 и S_2 по n/2 элементов в каждом, тогда с помощью описанного выше алгоритма 1 можно найти наименьший и наибольший элементы в каждой из двух половин с помощью рекурсии. А максимальный и минимальный элементы множества S можно было бы определить произведя ещё 2 сравнения.

Рассмотрим работу алгоритма. Для простоты будем считать, что n - степень числа 2, $n \geq 2$ и мощность множества S $||S|| = 2^k$, $k \geq 1$.

procedure MAXMIN (S):

1. **if** ||S|| = 2 **then**

begin

2. пусть S = {a, b}

```
3.                  return (MAX (a, b), MIN (a, b))
                end
          else
                begin
4.                  разбить S на два равных подмножества S1  и
5.          S2
6.                  (max1, min1) ← MAXMIN (S1)
7.                  (max2, min2) ← MAXMIN (S2)
                    return (MAX(max1, max2), MIN (min1, min2)
            )
                end
      end.
```

К множеству S применяется рекурсивная процедура MAXMIN. Она вырабатывает пару (a, b), где a - максимальный, a b - минимальный элементы в S. Заметим, что сравнения элементов происходят на шаге 3 (сравниваются два элемента множества S, из которых оно состоит) и на шаге 7, где сравниваются максимальные и минимальные элементы множества S_1 и S_2.

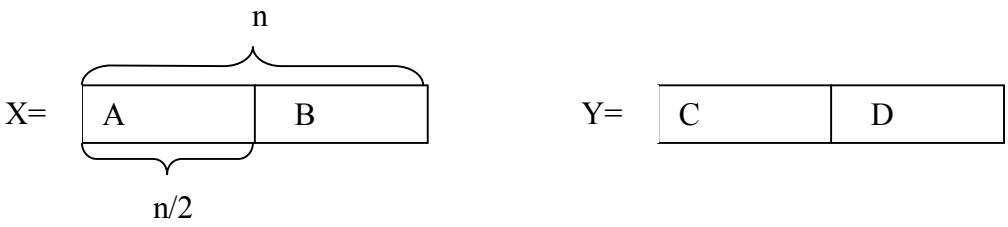
Пусть T(n) - число сравнений элементов множества S, которые надо произвести в процедуре MAXMIN. При n = 2 число сравнений равно 1. Если n>2, то число сравнений T(n)=2T(n/2)+2. Решением будет служить функция T(n)=3/2 n - 2. Т.о. временная сложность рассмотренного алгоритма O(n).

Рассмотрим ещё один пример, в котором применение метода декомпозиции позволяет уменьшить так же порядок роста сложности алгоритма.

Рассмотрим умножение двух n-разрядных двоичных чисел. Традиционный метод требует $O(n^2)$ битовых операций. В рассмотренном ниже примере, достаточно уже порядка $n^{\log 3} \approx n^{1,59}$ битовых операций (логарифм по основанию 2).

Пример 2. Пусть X и Y - два n-разрядных двоичных числа, и n - степень числа 2 (для простоты).

Разобьём X и Y на две равные части:



Если рассматривать каждую из этих частей как (n/2)-разрядное число, то произведение чисел X и Y можно представить в виде:

$$X * Y = (A*2^{n/2} + B)(C*2^{n/2} + D) = AC*2^{n} + (AD + BC)*2^{n/2} + BD$$

Это равенство даёт способ вычисления произведения X и Y с помощью 4-х умножений (n/2)-разрядных чисел (т.е. умножений A, B, C, D), несколько сложений и сдвигов (умножений на степень числа 2).

Произведение Z=X*Y можно вычислить по алгоритму:

begin

u ← (a+b)*(c+d);

v ← a*c;

w ← b*d;

$z \leftarrow v*2^{n} + (u-v-w)*2^{n/2} + w$

end.

Оценим временную сложность алгоритма. Для умножения здесь необходимо выполнить три умножения (n/2)-разрядных, несколько сложений и сдвигов. Для вычисления u, v, w можно применять эту программу рекурсивно. Сложение и сдвиг занимают O(n) времени.

Т.о., временная сложность умножения двух n-разрядных чисел ограничена сверху функцией

$$T(n) = \begin{cases} k & при \quad n = 1 \\ 3T(n/2) + kn & при \quad n > 1 \end{cases}$$

где k - постоянная, отражающая сложения и сдвиг в выражениях, входящих в алгоритм.

Решение рекуррентных уравнений ограничено сверху функцией $T(n)=3kn^{\log 3}-2kn \approx 3kn^{1,59}-2kn$.

Рассмотренный алгоритм умножения двух целых чисел можно применять не только к двоичным числам, но и к десятичным. Такой способ умножения приводит к асимптотической эффективности, т.к. для достаточно

больших n, любое фиксированное число n-разрядных сложений требует меньше времени, чем n-разрядное умножение.

Если алгоритм применять рекурсивно, то в результате получим асимптотическое улучшение временной сложности.

Балансировка

Равновесия или выполнение принципа балансировки - один из путей построения эффективных алгоритмов.

Балансировка - это разделение операций преобразования данных так, чтобы уравнивались показатели эффективности при решении задачи в любых условиях.

Рассмотрим задачу расположения целых чисел в порядке неубывания. Простейший способ сделать это (метод пузырька): 1) найти наименьший элемент, исследуя всю последовательность; 2) поменять местами найденный минимальный элемент с первым элементом; 3) процесс повторить для остальных (n-1)-го элемента, второй наименьший элемент оказывается на втором месте; и так далее.

Временная сложность такого алгоритма

$$T(n) = \begin{cases} 0 & при \quad n = 1 \\ T(n-1) + n - 1 & при \quad n > 1 \end{cases}$$

Решением рекуррентных уравнений является T(n)= n(n-1)/2 , т.е. $O(n^2)$. Хоть алгоритм можно применять рекурсивно, он неэффективен для больших n.

Для разработки асимптотически эффективного алгоритма сортировки воспользуемся принципом сбалансированности.

Разобьем задачу размера n на две подзадачи размера ≈n/2. Затем упорядочивают элементы каждого подмножества и выполняют объединение упорядоченных подмножеств. Этот метод называется “сортировкой слиянием”.

Пусть дана последовательность чисел $x_1, x_2, \ldots, x_n$, где n – степень числа 2.

В результате необходимо получить последовательность $y_1, y_2, \ldots, y_n$, члены которой расположены в порядке возрастания.

Для решения этой задачи используются две процедуры: процедура СЛИЯНИЕ(A, T), где A и T – две упорядоченные последовательности. Работа этой процедуры состоит в следующем: выбирается большее из наибольших

остающихся в A и T и выбранный элемент удаляется из последовательности. В случае совпадения можно отдавать предпочтение последовательности A.

Так как подмножества A и T упорядочены, то для работы процедуры потребуется не более чем $\|A\| + \|T\| + 1$ сравнений.

Кроме того, применяется процедура COPT(i, j), сортирующая последовательность $x_i, x_{i+1},\ldots,x_j$, которая имеет длину 2^k, где $k \geq 0$. Для сортировки начальной последовательности $x_1, x_2,\ldots,x_n$ вызывается процедура COPT(1, n).

```
procedure COPT(i, j):
    if i = j then return x_i
        else
            begin
                m ← (i + j - 1)/2;
                return СЛИЯНИЕ(COPT(i, m), COPT(m + 1, j))
            end
```

Рекуррентные уравнения для подсчёта числа выполняемых сравнений имеют вид:

$$T(n) = \begin{cases} \mathbf{0} & при \quad n = \mathbf{1} \\ \mathbf{2}T(n/\mathbf{2}) + n - \mathbf{1} & при \quad n > \mathbf{1} \end{cases}$$

Решением рекуррентного уравнения является: T(n)=O(n log n).

* * *

Далее мы рассмотрим два важных метода построения и анализа эффективных алгоритмов: динамическое программирование и жадные алгоритмы.

Динамическое программирование обычно применяется к задачам, в которых искомый ответ состоит из частей, каждая из которых в свою очередь даёт оптимальное решение некоторой подзадачи. Динамическое программирование полезно, если про разных путях многократно встречаются одни и те же подзадачи; основной технический приём – запоминать решения встречающихся подзадач на случай, если та же подзадача встретится вновь.

Жадные алгоритмы, как и динамическое программирование, применяются в тех случаях, когда искомый объект строится по частям. Жадный алгоритм делает на каждом шаге «локально оптимальный» выбор. Простой пример: стараясь набрать данную сумму денег минимальным

числом монет, можно последовательно брать монеты наибольшего возможного достоинства (не превосходящего той суммы, которую осталось набрать).

Амортизационный анализ – это средство анализа алгоритмов, производящих последовательность однотипных операций. Вместо того, чтобы оценивать время работы для каждой из этих операций по отдельности, амортизационный анализ оценивает среднее время работы в расчёте на одну операцию. Разница может оказаться существенной, если трудоёмкие операции не могут идти подряд. На самом деле амортизационный анализ – не только средство анализа алгоритмов, но ещё и подход к разработке алгоритмов: ведь разработка алгоритма и анализ скорости его работы тесно связаны.

Динамическое программирование

Подобно методу «разделяй и властвуй», динамическое программирование решает задачу, разбивая её на подзадачи и объединяя их решения. Алгоритмы типа «разделяй и властвуй» делят задачу на независимые подзадачи, эти подзадачи – на более мелкие подзадачи и так далее, а затем собирают решение основной задачи «снизу вверх». Динамическое программирование применимо тогда, когда подзадачи не являются независимыми, иными словами, когда у подзадач есть общие «подподзадачи». В этом случае алгоритмы типа «разделяй и властвуй» будет делать лишнюю работу, решая одни и те же подподзадачи по несколько раз. Алгоритм, основанный на динамическом программировании, решает каждую из подзадач единожды и запоминает ответы в специальной таблице. Это позволяет не вычислять заново ответ к уже встречавшейся подзадаче.

В типичном случае динамическое программирование применяется к задачам оптимизации. У такой задачи может быть много возможных решений; их «качество» определяется значением какого-то параметра, и требуется выбрать оптимальное решение, при котором значение параметра будет минимальным или максимальным (в зависимости от постановки задачи). Вообще говоря, оптимум может достигаться для нескольких разных решений.

Последовательность построения алгоритма, основанного на динамическом программировании:

1. описать строение оптимальных решений,
2. выписать рекуррентные соотношения, связывающее оптимальные значения параметра для подзадач,

3. двигаясь снизу вверх, вычислить оптимальное значение параметра для подзадач,

4. пользуясь полученной информацией, построить оптимальное решение.

Основную часть работы составляют шаги 1-3. Если нас интересует только оптимальное значение параметра, шаг 4 не нужен. Если же шаг 4 необходим, для построения оптимального решения иногда приходится получать и хранить дополнительную информацию в процессе выполнения шага 3.

Рассмотрим решение некоторых оптимизационных задач с помощью динамического программирования.

Перемножение нескольких матриц

Пусть мы хотим найти произведение

$A_1A_2...A_n$

последовательности n матриц $<A_1 , A_2 , ... , A_n>$. Мы будем пользоваться стандартным алгоритмом перемножения двух матриц в качестве подпрограммы. Но прежде надо расставить скобки, чтобы указать порядок умножений. Будем говорить, что в произведении матриц полностью расставлены скобки, если это произведение либо состоит из одной-единственной матрицы, либо является заключённым в скобки произведением двух произведений с полностью расставленными скобками. Поскольку произведение матриц ассоциативно, конечный результат вычислений не зависит от расстановки скобок. Например, в произведении $A_1A_2A_3A_4$ можно полностью расставить скобки пятью разными способами:

$(A_1 (A_2 (A_3A_4)))$, $(A_1 ((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1 (A_2A_3))A_4)$, $(((A_1A_2)A_3)A_4)$

во всех случаях ответ буде один и тот же.

Вот стандартный алгоритм перемножения матриц (здесь КОЛ_СТРОК и КОЛ_СТОЛБЦОВ обозначают количество строк и столбцов матрицы соответственно):

УМНОЖЕНИЕ_МАТРИЦ(A, B)

если КОЛ_СТРОК[B] ≠ КОЛ_СТОЛБЦОВ[A]

тогда вывести «умножить нельзя»

иначе для i ← 1 **до** КОЛ_СТРОК[A]

делать для j ← 1 **до** КОЛ_СТОЛБЦОВ[B]

делать C[i, j] ← 0

для к ← 1 **до** КОЛ_СТОЛБЦОВ[A]

делать C[i, j] ← C[i, j] + A[i, к] * B[к, j]

вернуть C

При выполнении этого алгоритма делается КОЛ_СТРОК[A]*КОЛ_СТОЛБЦОВ[A]*КОЛ_СТОЛБЦОВ[B] умножений и столько же сложений. Для простоты будем учитывать только умножения.

Чтобы увидеть, как расстановка скобок может влиять на стоимость, рассмотрим последовательность из трёх матриц $<A_1 , A_2 , A_3 >$ размерностью 10×100, 100×5 и 5×50 соответственно. При вычислении $((A_1 * A_2) A_3)$ нужно 10*100*5+10*5*50=7500 умножений, а при $(A_1 (A_2 * A_3))$ придётся сделать 100*5*50+10*100*50=75000 умножений, что на целый порядок больше первого варианта.

Задача об умножении последовательности матриц может быть сформулирована следующим образом: дана последовательность из n матриц $<A_1 , A_2 , ... , A_n>$ заданных размеров (матрица A имеет размер $p_{i-1}×p_i$); требуется найти такую (полную) расстановку скобок в произведении $A_1 * A_2 * ... * A_n$, чтобы вычисление произведения требовало наименьшего числа умножений.

Простой перебор всех возможных расстановок скобок не даст эффективного алгоритма.

Шаг 1: строение оптимальной расстановки скобок

Обозначим для удобства через $A_{i..j}$ матрицу, являющуюся произведением $A_i * A_{i+1} * ... * A_j$. При вычислении произведения, диктуемой расстановкой скобок, мы сначала вычисляем произведения $A_{1..к}$ и $A_{к+1..n}$, а затем перемножаем их и получаем ответ $A_{1..n}$. стоимость этой оптимальной расстановки равна стоимости вычисления матрицы $A_{1..к}$ плюс стоимость вычисления матрицы $A_{к+1..n}$ плюс стоимость перемножения этих двух матриц.

Чем меньше нам потребуется умножений для вычисления $A_{1..к}$ и $A_{к+1..n}$, тем меньше будет общее число умножений. Стало быть, оптимальное решение задачи о перемножении последовательности матриц содержит оптимальные решения задач о перемножении её частей.

Шаг 2: рекуррентное соотношение

Теперь необходимо выразить стоимость оптимального решения задачи через стоимость оптимальных решений её подзадач. Такими подзадачами будут задачи об оптимальной расстановке скобок в произведениях $A_{i..j}= A_i * A_{i+1} * ... * A_j$ для $1 \le i \le j \le n$. Обозначим через m[i, j] минимальное

количество умножений, необходимое для вычисления всего произведения $A_{1..n}$ есть m[1, n].

Числа m[i, j] можно вычислить так. Если i =j, то последовательность состоит из одной матрицы $A_{i..i} = A_i$ и умножение вообще не нужно. Значит m[i, i] = 0 для i = 1, 2, ... , n. Для случая когда i < j минимальное количество умножений можно вычислить по формуле, полученной на первом шаге:

$$m[i,j] = m[i,] + m[к+1,j] + p_{i-1}*p_к*p_j.$$

Число к может принимать всего лишь j-i различных значений: i,i+1,...,j-1. Поскольку одно из них оптимально, достаточно перебрать эти значения к и выбрать наилучшее.

Пусть s[i,j] равно числу к, для которого

$$m[i,j] = m[i,] + m[к+1,j] + p_{i-1}*p_к*p_j.$$

Шаг 3: вычисление оптимальной стоимости

Пользуясь выводами предыдущего шага легко описать алгоритм, определяющий минимальную стоимость вычисления произведения $A_1 * A_2 * ... * A_n$ (т.е. число m[1,n]). Сложность этого алгоритма $O(n^3)$.

Пусть $r_0, r_1 \ldots r_n$ есть входная последовательность, в которой r_{i-1} и r_i – размер строк и столбцов в матрице M_i. m_{ij} – минимальная сложность вычисления произведения цепочки матриц $M_i*M_{i+1}*\ldots*M_j$ и $1 \le i \le j \le n$.

$$m_{ij} = \begin{cases} 0 & ,если\ i = j \\ \underset{i \le k \le j}{MIN}(m_{ik} + m_{k+1,j} + r_{i-1} r_k r_j), & если\ j > i \end{cases} \quad (*)$$

Здесь m_{ik} – минимальная сложность вычисления произведения матриц $M`=M_i*M_{i+1}*\ldots*M_k$

$m_{k+1,j}$ - минимальная сложность вычисления произведения матриц $M``= M_{k+1}*M_{k+2}*\ldots*M_j$

Третье слагаемое $r_{i-1}*r_k*r_j$- сложность умножения M` на M``, т.к. матрица M` размером $(r_{i-1}*r_k)$, а матрица M`` размера (r_k*r_j).

Алгоритм. Алгоритм динамического программирования для вычисления порядка, минимизирующего сложность умножения цепочки из n матриц $M_1*M_2*\ldots*M_n$.

```
begin
for i←1 to n do mij←0;
for l←1 to n-1 do
        for i←1 to n-1 do
                begin
                j←i+l;
```

$m_{ij} \leftarrow MIN_{i \le k < j} (m_{ik} + m_{k+1,j} + r_{i-1} * r_k * r_j)$
end;
write m_{1n}
end.

Например, пусть необходимо определить порядок перемножения следующих матриц:

M = M_1 * M_2 * M_3 * M_4,
[10x20] [20x50] [50x1] [1x100]

В результате применения алгоритма получим значения $m_{ij,}$ которые приведём в таблице:

m_{11}=0	m_{22}=0	m_{33}=0	m_{44}=0
m_{12}=10000	m_{23}=1000	m_{34}=5000	
m_{13}=1200	m_{24}=3000		
m_{14}=2200			

Т.о., минимальное число операций , требуемых для вычисления этого произведения равно 2200. Порядок, в котором можно произвести эти умножения, легко определить, приписав каждой клетке таблицы то значение k, на котором достигается минимум.

Когда применимо динамическое программирование

Оптимальность для подзадач

При решении оптимизационной задачи с помощью динамического программирования необходимо сначала описать структуру оптимального решения. Будем говорить, что задача обладает свойством оптимальности для подзадач, если оптимальное решение задачи содержит оптимальные решения её подзадач. Чтобы убедиться, что задача обладает этим свойством, надо показать, что, улучшая решение подзадачи, мы улучшим и решение исходной задачи. Если задача обладает таким свойством, то динамическое программирование может оказаться полезным для её решения.

Перекрывающиеся подзадачи

Второе свойство задач существенное при использовании динамического программирования, - небольшое число различных подзадач. Благодаря этому при рекурсивном решении задачи мы многократно выходим на одни и те же подзадачи. В таком случае говорят, что у оптимизационной задачи имеются перекрывающиеся подзадачи. В типичных случаях количество подзадач полиномиально зависит от размера исходных данных.

В задачах, решаемых методом «разделяй и властвуй», так не бывает: для них рекурсивный алгоритм, как правило, на каждом шаге порождает совершенно новые подзадачи. Алгоритмы, основанные на динамическом программировании, используют перекрытие подзадач следующим образом: каждая из подзадач решается только один раз, и ответ заносится в специальную таблицу; когда эта же подзадача встречается снова, программа не тратит время на её решение, а берёт готовый ответ из таблицы.

Динамическое программирование «сверху вниз»

Если каждая из подзадач должна быть решена хоть один раз, метод динамического программирования («снизу вверх») обычно эффективнее, чем рекурсия с запоминанием ответов, поскольку реализация рекурсии (а также проверка, есть в таблице ответ или ещё нет) требует дополнительного времени. Но если для нахождения оптимума не обязательно решать все подзадачи, подход «сверху вниз» имеет то преимущество, что решаются лишь те подзадачи, которые действительно нужны.

Жадные алгоритмы

Для многих оптимизационных задач есть более простые и быстрые алгоритмы, чем динамическое программирование. Речь идёт о жадных алгоритмах. Такой алгоритм делает на каждом шаге локально оптимальный выбор, в надежде, что итоговое решение также окажется оптимальным. Это не всегда так, – но для многих задач такие алгоритмы дают оптимум.

Когда применим жадный алгоритм?

Общих рецептов нет, но существуют две особенности, характерные для задач, решаемых жадными алгоритмами. Это принцип жадного выбора и свойство оптимальности для подзадач.

Принцип жадного выбора

Говорят, что к оптимизационной задаче применим принцип жадного алгоритма, если последовательность локально оптимальных (жадных) выборов даёт глобально оптимальное решение. Различие между жадными алгоритмами и динамическим программированием можно пояснить так: на каждом шаге жадный алгоритм берёт «самый жирный кусок», а потом уже пытается сделать наилучший выбор среди оставшихся, каковы бы они ни были; алгоритм динамического программирования принимает решение, просчитав заранее последствия для всех вариантов.

Оптимальность для подзадач

Решаемые с помощью жадных алгоритмов задачи обладают свойством оптимальности для подзадач: оптимальное решение всей задачи содержит в себе оптимальные решения подзадач.

Жадный алгоритм или динамическое программирование?

И жадный алгоритм, и динамическое программирование основываются на свойстве оптимальности подзадач, поэтому может возникнуть искушение применить динамическое программирование в ситуации, где хватило бы жадного алгоритма, или, напротив, применить жадный алгоритм к задаче, в которой он не даст оптимума.

Применение жадного алгоритма рассмотрим на задаче сжатия информации методом Хаффмана.

Сжатие по алгоритму Хаффмана

Huffman - Сначала кажется что создание файла меньших размеров из исходного без кодировки последовательностей или исключения повтора байтов будет невозможной задачей. Но давайте мы заставим себя сделать несколько умственных усилий и понять алгоритм Хаффмана (Huffman). Потеряв не так много времени мы приобретем знания и дополнительное место на дисках.

Сжимая файл по алгоритму Хаффмана первое что мы должны сделать - это необходимо прочитать файл полностью и подсчитать сколько раз встречается каждый символ из расширенного набора ASCII. Если мы будем учитывать все 256 символов, то для нас не будет разницы в сжатии текстового и EXE файла.

После подсчета частоты вхождения каждого символа, необходимо просмотреть таблицу кодов ASCII и сформировать мнимую компоновку между кодами по убыванию. То есть не меняя местонахождение каждого символа из таблицы в памяти отсортировать таблицу ссылок на них по убыванию. Каждую ссылку из последней таблицы назовем "узлом". В дальнейшем (в дереве) мы будем позже размещать указатели которые будут указывает на этот "узел". Для ясности давайте рассмотрим пример:

Мы имеем файл длинной в 100 байт и имеющий 6 различных символов в себе. Мы подсчитали вхождение каждого из символов в файл и получили следующее:

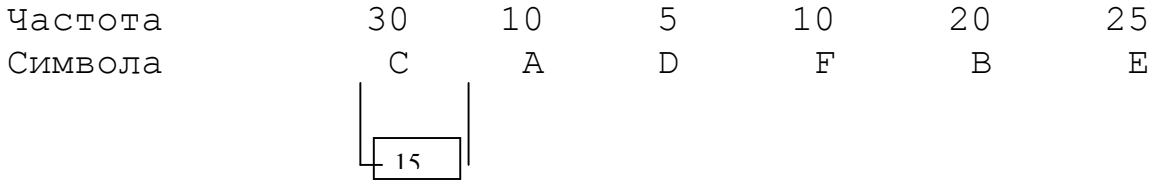
символ	A	B	C	D	E	F
число вхождений	10	20	30	5	25	10

Теперь мы берем эти числа и будем называть их частотой вхождения для каждого символа. Разместим таблицу как ниже.

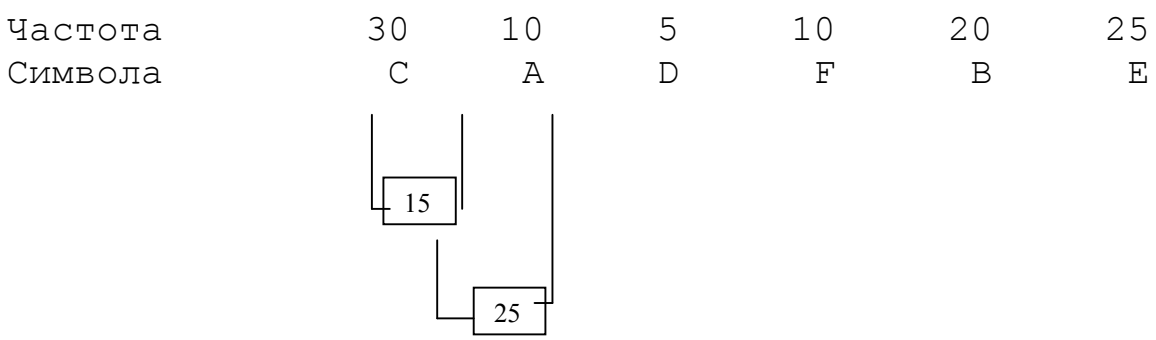
символ	C	E	B	F	A	D
число вхождений	30	25	20	10	10	5

Мы возьмем из последней таблицы символы с наименьшей частотой. В нашем случае это D (5) и какой либо символ из F или A (10), можно взять любой из них например A.

Сформируем из "узлов" D и A новый "узел", частота вхождения для которого будет равна сумме частот D и A :

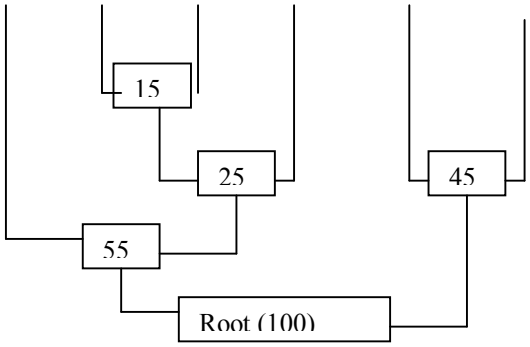


Номер в рамке - сумма частот символов D и A. Теперь мы снова ищем два символа с самыми низкими частотами вхождения. Исключая из просмотра D и A и рассматривая вместо них новый "узел" с суммарной частотой вхождения. Самая низкая частота теперь у F и нового "узла". Снова сделаем операцию слияния узлов :



Рассматриваем таблицу снова для следующих двух символов (B и E). Мы продолжаем в этот режим пока все "дерево" не сформировано, т.е. пока все не сведется к одному узлу.

Частота	30	10	5	10	20	25
Символа	C	A	D	F	B	E



Теперь когда наше дерево создано, мы можем кодировать файл. Мы должны всенда начинать из корня (Root). Кодируя первый символ (лист дерева C) Мы прослеживаем вверх по дереву все повороты ветвей и если мы делаем левый поворот, то запоминаем 0-й бит, и аналогично 1-й бит для правого поворота. Так для C, мы будем идти влево к 55 (и запомним 0), затем снова влево (0) к самому символу. Код Хаффмана для нашего символа C - 00. Для следующего символа (A) у нас получается - лево,право,лево,лево , что выливается в последовательность 0100. Выполнив выше сказанное для всех символов получим

```
C = 00   ( 2 бита )
A = 0100 ( 4 бита )
D = 0101 ( 4 бита )
F = 011  ( 3 бита )
B = 10   ( 2 бита )
E = 11   ( 2 бита )
```

Каждый символ изначально представлялся 8-ю битами (один байт), и так как мы уменьшили число битов необходимых для представления каждого символа, мы следовательно уменьшили размер выходного файла. Сжатие складывется следующим образом :

Частота	первоначально	уплотненные биты	уменьшено на
C 30	30 x 8 = 240	30 x 2 = 60	180
A 10	10 x 8 = 80	10 x 3 = 30	50
D 5	5 x 8 = 40	5 x 4 = 20	20

F 10	10 x 8 = 80	10 x 4 = 40	40
B 20	20 x 8 = 160	20 x 2 = 40	120
E 25	25 x 8 = 200	25 x 2 = 50	150

Первоначальный размер файла : 100 байт - 800 бит;
Размер сжатого файла : 30 байт - 240 бит;

240 - 30% из 800 , так что мы сжали этот файл на 70%.

Все это довольно хорошо, но неприятность находится в том факте, что для восстановления первоначального файла, мы должны иметь декодирующее дерево, так как деревья будут различны для разных файлов. Следовательно мы должны сохранять дерево вместе с файлом. Это превращается в итоге в увеличение размеров выходного файла.

В нашей методике сжатия и каждом узле находятся 4 байта указателя, по этому, полная таблица для 256 байт будет приблизительно 1 Кбайт длинной.

Таблица в нашем примере имеет 5 узлов плюс 6 вершин (где и находятся наши символы), всего 11. 4 байта 11 раз - 44. Если мы добавим после небольшое количество байтов для сохранения места узла и некоторую другую статистику - наша таблица будет приблизительно 50 байтов длинны.

Добавив к 30 байтам сжатой информации, 50 байтов таблицы получаем, что общая длинна архивного файла вырастет до 80 байт. Учитывая, что первоначальная длинна файла в рассматриваемом примере была 100 байт - мы получили 20% сжатие информации.

Не плохо. То что мы действительно выполнили - трансляция символьного ASCII набора в наш новый набор требующий меньшее количество знаков по сравнению с стандартным.

Что мы можем получить на этом пути ?

Рассмотрим максимум, который мы можем получить для различных разрядных комбинаций в оптимальном дереве, которое является несимметричным.

Мы получим что можно иметь только :

4 - 2 разрядных кода;
8 - 3 разрядных кодов;

```
  16 - 4 разрядных кодов;
  32 - 5 разрядных кодов;
  64 - 6 разрядных кодов;
 128 - 7 разрядных кодов;
```

Необходимо еще два 8 разрядных кода.

```
   4 - 2 разрядных кода;
   8 - 3 разрядных кодов;
  16 - 4 разрядных кодов;
  32 - 5 разрядных кодов;
  64 - 6 разрядных кодов;
 128 - 7 разрядных кодов;
--------
 254
```

Итак мы имеем итог из 256 различных комбинаций которыми можно кодировать байт. Из этих комбинаций лишь 2 по длине равны 8 битам. Если мы сложим число битов которые это представляет, то в итоге получим 1554 бит или 195 байтов. Так в максимуме , мы сжали 256 байт к 195 или 33%, таким образом максимально идеализированный Huffman может достигать сжатия в 33% когда используется на уровне байта.

Все эти подсчеты производились для не префиксных кодов Хаффмана т.е. кодов, которые нельзя идентифицировать однозначно. Например код A - 01011 и код B - 0101. Если мы будем получать эти коды побитно, то получив биты 0101 мы не сможем сказать какой код мы получили A или B , так как следующий бит может быть как началом следующего кода, так и продолжением предыдущего.

Необходимо добавить, что ключом к построению префиксных кодов служит обычное бинарное дерево и если внимательно рассмотреть предыдущий пример с построением дерева , можно убедится , что все получаемые коды там префиксные.

Одно последнее примечание - алгоритм Хаффмана требует читать входной файл дважды, один раз, считая частоты вхождения символов, другой раз производя непосредственно кодирование.

Сложность полученного алгоритма равна O(n log n).

Переборные алгоритмы

Данная статья открывает цикл работ, посвященных не особенностям какого-то отдельного языка программирования (например Паскаля), а общим ИДЕЯМ и МЕТОДАМ разработки алгоритмов. Тем не менее, опираться мы все равно будем на теория, числа, оптимальные алгоритмы, вычислительная, определение, который вы уже знаете. Первоначальный вариант любого алгоритма мы будем записывать на псевдокоде - языке, который занимает промежуточное положение между нашим обычным языком и языками программирования. Он не имеет каких-то жестких правил и требований, т.к. предназначен прежде всего для человека, а не компьютера. Это позволит нам избавиться от излишней детализации алгоритма на раннем этапе разработки и сразу выразить его основную идею. Превратить этот псевдокод в программу на Паскале задача совсем несложная - как это делать вы быстро поймете.
Основные идеи первого задания - ПЕРЕБОР, РЕКУРСИЯ, ПЕРЕБОР С ОТХОДОМ НАЗАД. Этими понятиями должен хорошо владеть каждый программист. Кроме того, переборные задачи составляют значительную долю всех школьных олимпиад по информатике.

1. Порождение и перебор комбинаторных объектов

Во многих прикладных задачах требуется найти оптимальное решение среди очень большого (но конечного!) числа вариантов. Иногда удается построить это решение сразу, но в большинстве случаев единственный способ его отыскать состоит в переборе ВСЕХ возможных вариантов и сравнении их между собой. Поэтому так важно для нас научиться строить алгоритмы ПЕРЕБОРА различных комбинаторных объектов - последовательностей, перестановок, подмножеств и т.д.
Схема перебора всегда будет одинакова:
- во-первых, надо установить ПОРЯДОК на элементах, подлежащих перечислению (в частности, определить, какой из них будет первым, а какой последним);

- во-вторых, научиться переходить от произвольного элемента к НЕПОСРЕДСТВЕННО СЛЕДУЮЩЕМУ за ним (т.е. для заданного элемента x1 строить такой элемент x2, что x1<x2 и между x1 и x2 нет других элементов).
Наиболее естественным способом упорядочения составных объектов является ЛЕКСИКОГРАФИЧЕСКИЙ порядок, принятый в любом словаре (сначала сравниваются первые буквы слов, потом вторые и т.д.) - именно его

мы и будем чаще всего использовать. А вот процедуру получения следующего элемента придется каждый раз изобретать за- ново. Пока запишем схему перебора в таком виде:

```
X:=First;
    while X<>Last do Next(X);
```

где First - первый элемент; Last - последний элемент; Next - процедура получения следующего элемента.

1.1. Последовательности

Напечатать все последовательности длины N из чисел 1,2,...,M.
First = (1,1,...,1) Last = (M,M,...,M)
Всего таких последовательностей будет M^N (докажите!). Чтобы понять. как должна действовать процедура Next, начнем с примеров. Пусть N=4,M=3. Тогда:
Next(1,1,1,1) -> (1,1,1,2) Next(1,1,1,3) -> (1,1,2,1) Next(3,1,3,3) -> (3,2,1,1)
Теперь можно написать общую процедуру Next:

```
procedure Next;
  begin
    {найти i: X[i]<M,X[i+1]=M,...,X[N]=M};
    X[i]:=X[i]+1;
    X[i+1]:=...:=X[N]:=1
  end;
```

Если такого i найти не удается, то следующей последовательности нет - мы добрались до последней (M,M,...,M). Заметим также, что если бы членами последовательности были числа не от 1 до M, а от 0 до M-1, то переход к следующей означал бы прибавление 1 в M-ичной системе счисления. Полная программа на Паскале выглядит так:

```
program Sequences;
  type Sequence=array [byte] of byte;
  var M,N,i:byte;
      X:Sequence;
      Yes:boolean;
  procedure Next(var X:Sequence;var Yes:boolean);
    var i:byte;
  begin
    i:=N;
    {поиск i}
    while (i>0)and(X[i]=M) do begin X[i]:=1;dec(i) end;
    if i>0 then begin inc(X[i]);Yes:=true end
    else Yes:=false
  end;
```

```
begin
  write('M,N=');readln(M,N);
  for i:=1 to N do X[i]:=1;
  repeat
     for i:=1 to N do write(X[i]);writeln;
     Next(X,Yes)
  until not Yes
end.
```

1.2. Перестановки

Напечатать все перестановки чисел 1..N (то есть последовательности длины N, в которые каждое из чисел 1..N входит ровно по одному разу).
First = (1,2,...,N) Last = (N,N-1,...,1)
Всего таких перестановок будет N!=N*(N-1)*...*2*1 (докажите!). Для составления алгоритма Next зададимся вопросом: в каком случае i-ый член перестановки можно увеличить, не меняя предыдущих? Ответ: если он меньше какого-либо из следующих членов (членов с номерами больше i).
Мы должны найти наибольшее i, при котором это так, т.е. такое i, что X[i]<X[i+1]>...>X[N] (если такого i нет, то перестановка последняя). После этого X[i] нужно увеличить минимально возможным способом, т.е. найти среди X[i+1],...,X[N] наименьшее число, большее его. Поменяв X[i] с ним, остается расположить числа с номерами i+1,...,N так, чтобы перестановка была наименьшей, то есть в возрастающем порядке. Это облегчается тем, что они уже расположены в убывающем порядке:

```
procedure Next;
begin
  {найти i: X[i]<X[i+1]>X[i+2]>...>X[N]};
  {найти j: X[j]>X[i]>X[j+1]>...>X[N]};
  {обменять X[i] и X[j]};
  {X[i+1]>X[i+2]>...>X[N]};
  {перевернуть X[i+1],X[i+2],...,X[N]};
end;
```

Теперь можно написать программу:

```
program Perestanovki;
  type Pere=array [byte] of byte;
  var N,i,j:byte;
      X:Pere;
      Yes:boolean;
  procedure Next(var X:Pere;var Yes:boolean);
    var i:byte;
```

```
    procedure Swap(var a,b:byte);  {обмен переменных}
      var c:byte;
    begin c:=a;a:=b;b:=c end;
  begin
    i:=N-1;
    {поиск i}
    while (i>0)and(X[i]>X[i+1]) do dec(i);
    if i>0 then
      begin
        j:=i+1;
        {поиск j}
        while (j<N)and(X[j+1]>X[i]) do inc(j);
        Swap(X[i],X[j]);
        for j:=i+1 to (N+i) div 2 do Swap(X[j],X[N-j+i+1]);
        Yes:=true
      end
    else Yes:=false
  end;
begin
  write('N=');readln(N);
  for i:=1 to N do X[i]:=i;
  repeat
    for i:=1 to N do write(X[i]);writeln;
    Next(X,Yes)
  until not Yes
end.
```

1.3. Разбиения

Перечислить все разбиения целого положительного числа N на целые положительные слагаемые (разбиения, отличающиеся лишь порядком слагаемых, считаются за одно).

Пример: N=4, разбиения: 1+1+1+1, 2+1+1, 2+2, 3+1, 4.

First = (1,1,...,1) - N единиц Last = (N)

Чтобы разбиения не повторялись, договоримся перечислять слагаемые в невозрастающем порядке. Сказать, сколько их будет всего, не так-то просто (см.следующий пункт). Для составления алгоритма Next зададимся тем же вопросом: в каком случае i-ый член разбиения можно увеличить, не меняя предыдущих?

Во-первых, должно быть X[i-1]>X[i] или i=1. Во-вторых, i должно быть не последним эле ментом (увеличение i надо компенсировать уменьшением следующих). Если такого i нет, то данное разбиение последнее. Увеличив i,

все следующие элементы надо взять минимально возможными, т.е. равными единице:

```
  procedure Next;
    begin
        {найти i: (i<L) and ( (X[i-1]>X[i]) or (i=1) )}
        X[i]:=X[i]+1;
        { L:= i + X[i+1]+...+X[L] - 1 }
        X[i+1]:=...:=X[L]:=1
    end;
```

Через L мы обозначили количество слагаемых в текущем разбиении (понятно, что 1<=L<=N). Программа будет выглядеть так:

```
 program Razbieniya;
   type Razb=array [byte] of byte;
   var N,i,L:byte;
       X:Razb;
   procedure Next(var X:Razb;var L:byte);
     var i,j:byte;
         s:word;
   begin
     i:=L-1;s:=X[L];
     {поиск i}
     while (i>1)and(X[i-1]<=X[i]) do begin s:=s+X[i];dec(i) end;
     inc(X[i]);
     L:=i+s-1;
     for j:=i+1 to L do X[j]:=1
   end;
 begin
   write('N=');readln(N);
   L:=N; for i:=1 to L do X[i]:=1;
   for i:=1 to L do write(X[i]);writeln;
   repeat
     Next(X,L);
     for i:=1 to L do write(X[i]);writeln
   until L=1
 end.
```

1.4. Подсчет количеств

Иногда можно найти количество объектов с тем или иным свойством, не перечисляя их. Классический пример: C(n,k) - число всех k-элементных подмножеств n-элементного множества - можно найти, заполняя таблицу значений функции C по формулам:

C(n,0) = C(n,n) = 1 (n>=1) C(n,k) = C(n-1,k-1) + C(n-1,k) (n>1, 0<k<n);
или по формуле n!/(k!*(n-k)!) (первый способ эффективнее, если надо вычислить много значений C(n,k)).
Попробуем посчитать таким способом количество разбиений из пункта 1.3. Обозначим через R(N,k) (при N>=0, k>=0) число разбие- ний N на целые положительные слагаемые, не превосходящие k (при этом R(0,k) считаем равным 1 для всех k>=0). Очевидно, что число R(N,N) и будет искомым. Все разбиения N на слагаемые, не превосходящие k, разобьем на группы в зависимости от максимального слагаемого (обозначим его i).
Число R(N,k) равно сумме (по всем i от 1 до k) количеств разбиений со слагаемыми не больше k и максимальным слагаемым, равным i. А разбиения N на слагаемые не более k с первым слагаемым, равным i, по существу представляют собой разбиения n-i на слагаемые, не превосходящие i (при i<=k). Так что
R(N,k) = R(N-1,1)+R(N-2,2)+...+R(N-k,k).
Остальное вы сделаете сами в домашнем задании.

2. Рекурсия

Вы уже знаете, что рекурсивными называются процедуры и функции, которые вызывают сами себя. Рекурсия позволяет очень просто (без использования циклов) программировать вычисление функций, заданных рекуррентно, например факториала f(n)=n!:
f(0)=1 f(n)=n*f(n-1).
Оказывается, рекурсивные процедуры являются удобным способом порождения многих комбинаторных объектов. Мы заново решим здесь несколько задач предыдущей главы и вы убедитесь, что запись многих алгоритмов значительно упростится благодаря использованию рекурсии.

2.1. Факториал

Еще раз напомним рекурсивный алгоритм вычисления факториала:

```
program Factorial;
  var N:word;
  function F(n:word):longint;
  begin
    if n=0 then F:=1 else F:=n*F(n-1)
  end;
begin
  write('N=');readln(N);
  writeln('N!=',F(N))
end.
```

2.2. Ханойская башня

Игра "Ханойская башня" состоит в следующем. Есть три стержня. На первый из них надета пирамидка из N колец (большие кольца снизу, меньшие сверху). Требуется переместить кольца на другой стержень. Разрешается перекладывать кольца со стержня на стержень, но класть большее кольцо поверх меньшего нельзя. Составить программу, указывающую требуемые действия.

Напишем рекурсивную процедуру перемещения M верхних колец с A-го стержня на B-ый в предположении, что остальные кольца больше по размеру и лежат на стержнях без движения:

```
procedure Move(M,A,B:integer);
  var C:integer;
begin
  if M=1 then begin writeln ('сделать ход ',A,'->',B) end
  else
     begin
       C:=6-A-B; {C - третий стержень: сумма номеров равна 6}
       Move(M-1,A,C);
       Move(1,A,B);
       Move(M-1,C,B)
     end
end;
```

Сначала переносится пирамидка из M-1 колец на третий стержень C. После этого M-ое кольцо освобождается, и его можно перенести на B. Остается перенести пирамиду из N-1 кольца с C на B. Чем это проще первоначальной задачи? Тем, что количество колец стало на единицу меньше. Теперь основную программу можно записать в несколько строк:

```
program Hanoi;
  var N:integer;
  procedure Move(M,A,B:integer);
          .............
begin
  write('N=');readln(N);
  Move(N,1,2)
end.
```

Если вы владеете основами компьютерной графики, можете попробовать "нарисовать" каждый ход на экране.

Таким образом, ОСНОВНАЯ ИДЕЯ любого рекурсивного решения - свести задачу к точно такой же, но с меньшим значением параметра. При этом какое-то минимальное значение параметра (например, 1 или 0) должно

давать решение без рекурсивного вызова - иначе программа "зациклится" (последовательность рекурсивных вызовов будет бесконечной). Это напоминает метод математической индукции в математике. В некоторых задачах удобно наоборот, увеличивать значение параметра при рекурсивном вызове. Тогда, естественно, "безрекурсивное" решение должно предусматриваться для некоторого максимального значения параметра. Попробуем использовать эту идею для перебора комбинаторных объектов.

2.3. Последовательности (рекурсивный алгоритм)

Задача та же, что в пункте 1.1. Опишем рекурсивную процедуру Generate(k), предъявляющую все последовательности длины N из чисел 1,...,M, у которых фиксировано начало X[1],X[2],...,X[k]. Понятно, что при k=N мы имеем тривиальное решение: есть только одна такая последовательность - это она сама. При k<N будем сводить задачу к k+1:

```
procedure Generate(k:byte);
  var i,j:byte;
begin
  if k=N then
    begin for i:=1 to N do write(X[i]);writeln end
  else
    for j:=1 to M do
       begin X[k+1]:=j; Generate(k+1) end
end;
```

Основная программа теперь выглядит очень просто:

```
program SequencesRecursion;
  type Sequence=array [byte] of byte;
  var M,N:byte;
     X:Sequence;
  procedure Generate(k:byte);
       .............
begin
  write('M,N=');readln(M,N);
  Generate(0)
end.
```

2.4. Перестановки (рекурсивный алгоритм)

Задача та же, что в пункте 1.2. Опишем рекурсивную процедуру Generate(k), предъявляющую все перестановки чисел 1,...,N, у которых фиксировано начало X[1],X[2],...,X[k]. После выхода из процедуры массив X будут иметь то же значение, что перед входом (это существенно!). Понятно,

что при k=N мы снова имеем только тривиальное решение - саму перестановку. При k<N будем сводить задачу к k+1:

```
procedure Generate(k:byte);
  var i,j:byte;
  procedure Swap(var a,b:byte);
    var c:byte;
  begin c:=a;a:=b;b:=c end;
begin
  if k=N then
    begin for i:=1 to N do write(X[i]);writeln end
  else
    for j:=k+1 to N do
       begin
         Swap(X[k+1],X[j]);
         Generate(k+1);
         Swap(X[k+1],X[j])
       end
end;
```

Основная программа:

```
program PerestanovkiRecursion;
  type Pere=array [byte] of byte;
  var N,i,j:byte;
     X:Pere;
  procedure Generate(k:byte);
     ..............
begin
  write('N=');readln(N);
  for i:=1 to N do X[i]:=i;
  Generate(0)
end.
```

Чтобы до конца разобраться в этой непростой программе, советуем выполнить ее на бумаге при N=3. Обратите внимание, что порядок вывода перестановок не будет лексикографическим!

3. Перебор с отходом назад

Как вы уже поняли, перебор комбинаторных объектов - задача весьма трудоемкая даже для компьютера. Например, перестановок из восьми чисел будет 8! = 40320 - число немаленькое. Поэтому в любой переборной задаче главная цель состоит в СОКРАЩЕНИИ ПЕРЕБОРА, т.е. в исключении тех объектов, которые заведомо не могут стать решением задачи. Предположим, что нам требуется рассмотреть только те перестановки, для которых сумма |X[i]-i| равна 8. Понятно, что их будет гораздо меньше: например, все

перестановки, начинающиеся на 8,7,... рассматривать не нужно! Как можно модифицировать наш переборный алгоритм в этом случае? Если на каком-то этапе сумма
|X[1]-1| + |X[2]-2| + ... + |X[k]-k|
уже больше 8, то рассматривать все перестановки, начинающиеся на X[1],...,X[k] уже не нужно - следует вернуться к X[k] и изменить его значение ("отойти назад" - отсюда название метода).
Для такой ситуации мы рассмотрим один общий метод, который почти всегда позволяет значительно сократить перебор. Пусть искомое решение находится среди последовательностей вида
X[1],...,X[N],
где каждое X[i] выбирается из некоторого множества вариантов A[i]. Предположим мы уже построили начало этой последовательности X[1],...,X[k] (k<N) и хотим продолжить его до решения.
Предположим также, что у нас есть некоторый простой метод P(X[1],...,X[k]), который позволяет получить ответ на вопрос: можно продолжить X[1],...,X[k] до решения (true) или нет (false). Заметим, что значение true еще НЕ ГАРАНТИРУЕТ существование такого продолжения, но зато значение false ГАРАНТИРУЕТ непродолжаемость ("не стоит дальше и пробовать"). Получаем простую рекурсивную процедуру ПЕРЕБОРА С ОТХОДОМ НАЗАД:

```
procedure Backtracking(k);
begin
   for (y in A[k]) do
    if P(X[1],...,X[k-1],y) then
      begin
        X[k]:=y;
        if k=N then {X[1],...,X[N] -решение}
        Backtracking(k+1)
      end
end;
```

Перечислить все расстановки 8-ми ферзей на шахматной доске, при которых они не бьют друг друга

Классической задачей, которая решается методом перебора с отходом назад считается задача о восьми ферзях: требуется перечислить все способы расстановки 8-ми ферзей на шахматной доске 8 на 8, при которых они не бьют друг друга. Эту задачу решил больше 200 лет тому назад великий математик Леонард Эйлер. Заметьте, что у него не было компьютера, но тем не менее он абсолютно верно нашел все 92 таких расстановки!
Очевидно, на каждой из 8 вертикалей должно стоять по ферзю. Каждую такую расстановку можно закодировать одномерным массивом

X[1],...,X[8],
где X[i] - номер горизонтали для i-го ферзя. Поскольку никакие два ферзя не могут стоять на одной горизонтали (тогда они бьют друг друга), то все X[i] различны, т.е. образуют перестановку из чисел 1..8. Можно, конечно, перебрать все 8! таких перестановок и выбрать среди них те 92, которые нас интересуют. Но число 8!=40320 довольно большое.
Поэтому мы воспользуемся алгоритмом перебора с отходом назад, который позволит значительно сократить перебор и даст ответ намного быстрее:

```
program Queens;
  const N=8;
  type Index=1..N;
       Rasstanovka=array [Index] of 0..N;
  var X:Rasstanovka;
     Count:word;
  function P(var X:Rasstanovka;k,y:Index):boolean;
    var i:Index;
  begin
    i:=1;
    while (i<k)and(y<>X[i])and(abs(k-i)<>abs(y-X[i])) do inc(i);
    P:=i=k
  end;
  procedure Backtracking(k:Index);
    var i,y:Index;
  begin
    for y:=1 to N do
      if P(X,k,y) then
         begin
           X[k]:=y;
           if k=N then
             begin
               for i:=1 to N do write(X[i]);writeln;inc(Count)
             end;
           Backtracking(k+1)
         end
  end;
begin
  Count:=0;
  writeln('Расстановки ',N,' ферзей:');
  Backtracking(1);
  writeln('Всего ',Count,' расстановок')
end.
```