Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Вятский государственный гуманитарный университет»

Дополнительная подготовка школьников по дисциплине «Информатика и информационные технологии»

Учебный модуль
Перебор с возвратом

О. А. Пестов

Киров
2011

СОДЕРЖАНИЕ

1. Общая схема рекурсивного перебора 3
2. Перечисление комбинаторных объектов. 9
  2.1. Задача перечисления всех перестановок 9
  2.2. Сочетания 11
3. Оптимизация перебора 13
  3.1. Метод ветвей и границ 13
  3.2. Представление числа в виде суммы заданных слагаемых 16
4. Задания для самостоятельного решения 20
Литература 21

1. Общая схема рекурсивного перебора

Есть такие задачи, для решения которых приходится организовывать полный перебор возможных вариантов. *Перебор с возвратом* (*backtracking*) – это общий метод упорядоченного перебора. Перебор с возвратом особенно удобен для решения задач, требующих проверки потенциально большого, но конечного числа решений. В настоящем разделе описывается схема рекурсивного перебора с возвратом.

В самом общем случае мы полагаем, что решение можно записать как вектор (массив переменной длины) $B = (b_1,b_2,\ldots,b_n)$, удовлетворяющий заданным условиям и ограничениям, или как множество таких векторов. При этом в одних задачах размерность решения (число n) может быть известна заранее, а в других заранее не определена.

Метод перебора с возвратом основан на том, что при поиске решения многократно делается попытка расширить текущее частичное решение, то есть его продолжить. Если расширение невозможно, то происходит возврат к предыдущему более короткому частичному решению, и делается попытка его расширить другим возможным способом.

В качестве исходного частичного решения мы выбираем пустой вектор, который будем обозначать ( ). На основе заданных ограничений выясняем, какие элементы являются кандидатами в b_1; обозначим это подмножество через S_1. В качестве b_1 выбираем первый элемент из S_1 и получаем частичное решение (b_1). В общем случае, по частичному решению $(b_1,b_2,\ldots,b_{k-1})$ на основе ограничений задачи строится S_k, из которого выбираются кандидаты для расширения частичного решения $(b_1,b_2,\ldots,b_{k-1})$ до $(b_1,b_2,\ldots,b_{k-1},b_k)$. Если $S_k=\{\}$, то есть частичное решение $(b_1,b_2,\ldots,b_{k-1})$ не может быть расширено, мы возвращаемся и выбираем новый элемент b_{k-1}. Если новый элемент b_{k-1} выбрать нельзя, мы возвращаемся еще дальше и выбираем новый элемент b_{k-2} и т. д.

Проиллюстрируем идею перебора с возвратом на поиске решения головоломки «Судоку». Напомним, что в данной головоломке игровое поле представляет собой квадрат размером 9×9, разделенный на меньшие квадраты со стороной в 3 клетки. В начале игры в некоторых клетках уже записаны числа от 1 до 9.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Рис. 1. Начальная позиция

В «Судоку» есть всего одно правило. Необходимо заполнить свободные клетки цифрами от 1 до 9 так, чтобы в каждой строке, в каждом столбце и в каждом малом квадрате 3×3 каждая цифра встречалась бы только один раз.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Рис. 2. Решение головоломки. Его можно записать в виде вектора B=(5,3,4,6,7,8,9,1,2,6,7,2,1,9,5,3,4,8,1,9,8,3,4,2,5,6,7,…,3,4,5,2,8,6,1,7,9)

В ходе перебора мы на очередном шаге выбираем, какое число поставить в клетку. Если ни одно из чисел не подходит для текущей клетки, то нужно вернуться на одну клетку назад. Первое правило говорит о том, как расширить текущее решение (если это возможно), второе правило – как выходить из тупика. В этом и состоит сущность перебора с возвратом: продолжать расширение исследуемого решения до тех пор, пока это возможно, и когда решение нельзя расширить, возвращаться по нему и пытаться сделать другой выбор на самом близком шаге, где имеется такая возможность.

Рассмотрим подробнее, как можно найти решение головоломки, изображенной на рис. 3, применяя перебор с возвратом.

11	12	13	14	15 7	16	17 9	18 1	19 2
21	22 7	23 2	24 1	25 9	26 5	27 3	28 4	29 8
31 1	32 9	33 8	34 3	35 4	36 2	37 5	38 6	39 7
41 8	42 5	43 9	44 7	45 6	46 1	47 4	48 2	49 3
51	52 2	53	54 8	55 5	56 3	57 7	58 9	59 1
61 7	62 1	63 3	64 9	65 2	66 4	67 8	68 5	69 6
71 9	72 6	73 1	74 5	75 3	76 7	77 2	78 8	79 4
81 2	82 8	83 7	84 4	85 1	86 9	87 6	88 3	89 5
91	92 4	93	94 2	95 8	96 6	97 1	98 7	99 9

Рис. 3. Пример головоломки. Клетки пронумерованы в соответствии с номером строки и столбца

Обозначим через $M = \{1,2,3,4,5,6,7,8,9\}$ множество возможных значений в клетке. Под решением будем понимать вектор *B*, представляющий собой последовательность элементов из множества *M*. Каждый элемент соответствует числу, записанному в одной клетке. Количество элементов вектора *B* равно количеству клеток (9·9=81). Правда, некоторые клетки известны заранее и значения в них можно не перебирать. В нашем случае, можно считать, что решение – это вектор из 10-и элементов (по одному элементу на каждую неизвестную клетку). Первый элемент соответствует клетке 11, второй клетке 22 и т.д. (13, 14, 16, 21, 51, 53, 91, 93).

На i-м шаге будем определять множество S_i возможных значений для клетки следующим образом: включим значение $m \in M$ в S_i в том и только том случае, если оно еще не содержится в текущей строке, столбце или малом квадрате. Например, S_1={3,4,5,6} так, как ни одно из этих чисел на данный момент не содержится в первой строке, первом столбце и первом квадрате. Договоримся, что если множество S_i содержит более одного элемента, сохраним в нем тот же порядок элементов, что и во множестве M.

Начнем поиск решения по шагам. Текущий частичный вектор будем обозначать через B.

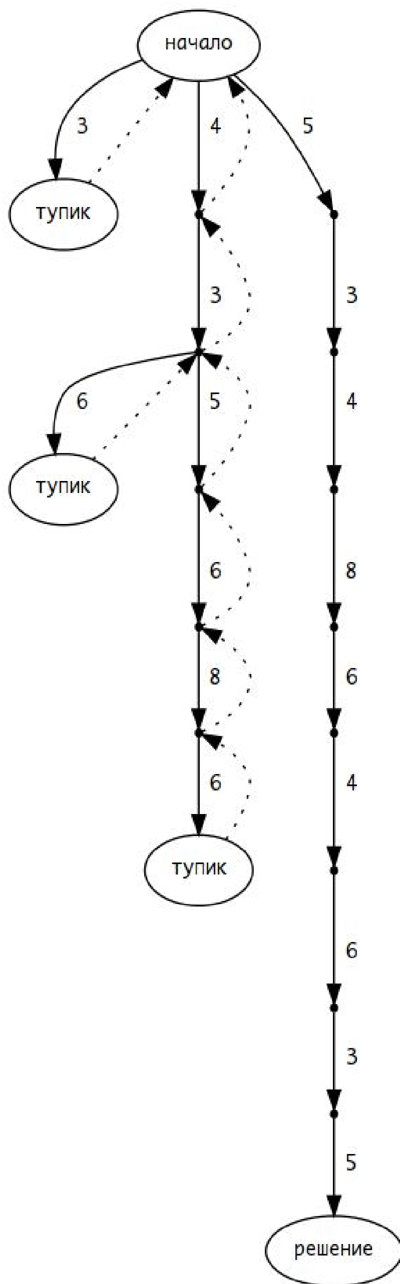

Рис. 4. Дерево поиска

Шаг 1. Полагаем B=( ).

Шаг 2 [клетка (1,1)]. Вычисляем S_1={3,4,5,6}. Выбираем 3 для расширения частичного решения, т.е. B=(3). Переходим в клетку (1,2).

Шаг 3 [клетка (1,2)]. Во втором столбце известны 8 из 9-и клеток, оставшаяся должна иметь значение 3, но в первой строке уже есть тройка (поставленная на шаге 2), поэтому S_2={}. Возвращаемся назад.

Шаг 4 [клетка (1,1)]. Расширяем частичное решение следующим вариантом из S_1={3,4,5,6}. Теперь B=(4).

Шаг 5 [клетка (1,2)]. S_2={3}, B=(4,3).

Шаг 6 [клетка (1,3)]. S_3={5,6}, B=(4,3,5).

Шаг 7 [клетка (1,4)]. S_4={6}, B=(4,3,5,6),

Шаг 8 [клетка (1,6)]. S_5={8}, B=(4,3,5,6,8). Текущая версия решения представлена на рис. 5.

Шаг 9 [клетка (2,1)]. S_6={6}, B=(4,3,5,8,6), переходим в клетку (5,1).

Шаг 10 [клетка (5,1)]. S_7={}, возвращаемся в предыдущую клетку (2,1). Все варианты из S_6 проверены, поэтому возвращаемся в клетку (1,6) и т.д. возвращаемся до клетки, где есть еще нерассмотренные варианты [клетка (1,3)].

Шаг 11 [клетка (1,3)]. Выбираем следующее нерассмотренное расширение из S_3={5,6}: B=(4,3,6).

Шаг 12 [клетка (1,4)]. S_4={}, возвращаемся в предыдущую клетку.

Продолжаем процесс до тех пор, пока не найдем решения или не переберем все варианты.

11 4	12 3	13 5	14 6	15 7	16 8	17 9	18 1	19 2
21	22 7	23 2	24 1	25 9	26 5	27 3	28 4	29 8
31 1	32 9	33 8	34 3	35 4	36 2	37 5	38 6	39 7
41 8	42 5	43 9	44 7	45 6	46 1	47 4	48 2	49 3
51	52 2	53	54 8	55 5	56 3	57 7	58 9	59 1
61 7	62 1	63 3	64 9	65 2	66 4	67 8	68 5	69 6
71 9	72 6	73 1	74 5	75 3	76 7	77 2	78 8	79 4
81 2	82 8	83 7	84 4	85 1	86 9	87 6	88 3	89 5
91	92 4	93	94 2	95 8	96 6	97 1	98 7	99 9

Рис. 5. Частичное решение после шага 8

Процесс получения решения для нашей задачи можно изобразить графически, построив дерево поиска с возвратом (рис. 4). Построение дерева начинается с начальной вершины (корня дерева), которой соответствует пустое частичное решение. Ребра дерева помечаются элементами множества и соответствуют выбору расширения частичного решения. В дереве поиска отмечена вершина, соответствующая решению. Само решение складывается из последовательности значений, приписанных ребрам пути при движении от корня к отмеченной вершине.

Реализация перебора с возвратом обычно приводит к экспоненциальным алгоритмам, что связано с экспоненциальным числом исследуемых вершин в дереве поиска с возвратом.

Мы привели пример задачи, в которой требуется найти любое решение, если оно есть. В этом случае алгоритм останавливается, как только найдет первое решение. Если бы надо было перечислить все решения, мы продолжили бы поиск. В этом случае перебор закончился бы в начальной вершине дерева поиска, при этом все варианты продолжения для корня

дерева были использованы для поиска. Может оказаться, что решения у головоломки нет. Тогда также пришлось бы исследовать все возможные варианты.

Процесс решения задачи перебором с возвратом подразделяется на отдельные подзадачи, которые удобно описывать с использованием рекурсии.

При использовании рекурсии отпадает необходимость непосредственно организовывать возвраты и отслеживать правильность их выполнения. Они становятся встроенной частью механизма выполнения рекурсивных вызовов.

Рекурсивный алгоритм начинает работу с пустого вектора, который будем обозначать ( ). Ниже приведена рекурсивная реализация перебора с возвратом на псевдокоде:

```
procedure backtracking(vector, i)
 if vector не имеет смысла расширять then exit
 if vector является решением then записать vector
 построить Si
 for s ∈ Si do
     backtracking(vector+s, i+1)
```

Для запуска процедуры необходимо ее вызвать:

```
backtracking((), 1)
```

2. Перечисление комбинаторных объектов.

В настоящем разделе рассматриваются примеры использования метода перебора с возвратом в задачах перечисления элементарных комбинаторных объектов.

2.1. Задача перечисления всех перестановок

Пусть $X = \{x_1, x_2, \ldots, x_n\}$ – конечное множество различных элементов. Под перестановкой элементов множества X понимается произвольная последовательность $\left(x_{i_1}, x_{i_2}, \ldots, x_{i_n}\right)$ длины n, в которой каждый элемент из X встречается ровно один раз.

В этом разделе рассматривается задача перечисления всех перестановок для множеств X вида $\{1,2,\ldots,n\}$. Например, если $X=\{1,2,3\}$, то получаются перестановки: 123, 132, 213, 231, 312, 321.

Общая схема решения основана на рекурсии. Будем генерировать все перестановки в массиве `B`. В каждый момент времени мы находимся в некоторой вершине дерева поиска на глубине i. Нам известны первые i–1 элементов перестановки (частичное решение) и выбирается новый элемент на позицию i (расширение частичного решения):

```
{выбираем элемент на позицию i}
procedure gen(i)
 {если n элементов известны, то перестановка найдена}
 if i > n then вывести решение
 { Si это еще не использованные числа}
 построить Si
 for s ∈ Si do
      b[i] = s {расширяем решение}
      gen(i+1)
      b[i] = 0 {восстанавливаем решение}
```

Чтобы построить S_i, необходимо перебрать числа, использованные в текущем частном решении, и вычеркнуть их из списка возможных вариантов. Делать это на каждом шаге не эффективно. Лучше параллельно с глобальным массивом `B` хранить глобальный массив `u` (англ. *used* – использованный) такой, что `u[j]=true` если и только если число j занято в текущей перестановке. Поддерживать состояние массива `u` в актуальном состоянии несложно. При этом значительно упрощается построение S_i – нам

фактически необходимо перебрать все *j* такие, что `u[j]=false`. Приходим к следующему решению:

```
procedure print;
var j : integer;
begin
  for j := 1 to n do
    write(b[j], ' ');
  writeln;
end;

procedure gen(i : integer);
var j : integer;
begin
  if i > n then begin print; exit; end;
  for j := 1 to n do
    if not u[j] then
    begin
      b[i] := j; {расширяем текущее решение числом j}
      u[j] := true; {говорим, что число j занято}
      gen(i+1);
      b[i] := j; {возвращаемся к предыдущему решению}
      u[j] := false; {освобождаем число j}
    end;
end;
```

Пример дерева вызовов процедуры `gen` для генерации перестановок длины 3 приведен на рис. 6.

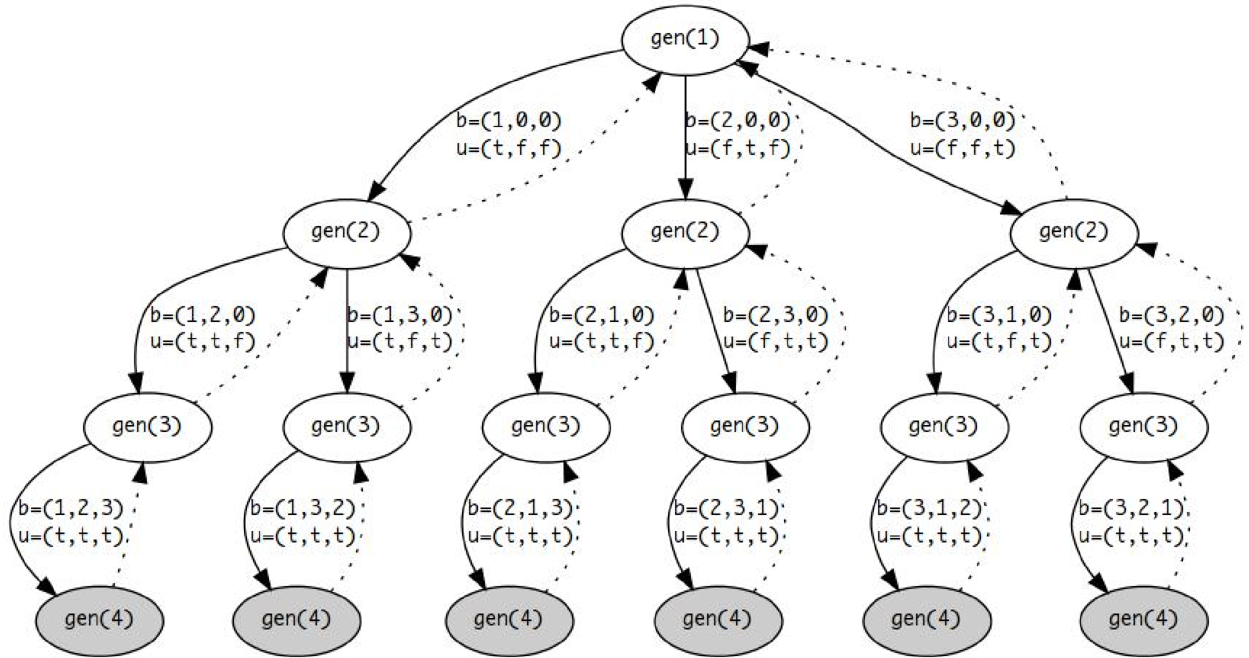

Рис. 6. Дерево поиска для перечисления всех перестановок из трех элементов. Здесь *t* означает *true*, а *f* – *false*

Заметим, что явное применение метода перебора с возвратом к перечислению всех перестановок порождает их в лексикографическом (словарном) порядке. В лексикографическом порядке, к примеру, перечисляются обычно фамилии учеников в классном журнале. Для случая перестановок, если $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ и $\tau = (\tau_1, \tau_2, \ldots, \tau_n)$ – перестановки, то мы говорим, что σ лексикографически меньше τ, если и только если для некоторого $k \geq 1$ верно, что $\sigma_j = \tau_j$ для всех $j < k$ и $\sigma_k < \tau_k$. Другими словами, найдется такая позиция *k*, что в *k*-й позиции перестановки σ стоит меньшее число, чем в перестановке τ, а во всех предшествующих позициях числа в перестановках совпадают. К примеру, перестановка из трех элементов 213 лексикографически меньше перестановки 231, так как в первой позиции числа совпадают, а во второй позиции в перестановке 213 стоит меньшее число, чем в перестановке 231 (1 меньше 3).

2.2. Сочетания

Пусть $X=\{x_1,x_2,\ldots,x_n\}$ – конечное множество различных элементов. Сочетанием мощности *k* называется *k*-элементное подмножество множества *X* или, другими словами, выборка *k* элементов из множества *X*.

В этом разделе мы рассмотрим задачу порождения всех сочетаний в лексикографическом порядке для множества X вида $\{1,2,\ldots,n\}$. Заметим, что

одно и то же множество {1,2,3} можно записать разными способами: {1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2} и {3,2,1}. При перечислении будем использовать первый способ записи (в порядке возрастания элементов). Так, например, сочетания из шести элементов по три (т.е. трехэлементные подмножества множества {1,2,3,4,5,6}) записываются в лексикографическом порядке следующим образом: 123, 124, 125, 126, 134, 135, 136, 145, 146, 156, 234, 235, 236, 145, 246, 256, 345, 346, 356, 456.

Общая идея решения идентична той, которая использовалась для перечисления перестановок. Будем генерировать сочетания в массиве *B*. В каждый момент времени мы находимся в некоторой вершине дерева поиска на глубине *i*. Нам известны первые i-1 элементов сочетания (частичное решение) и выбирается новый элемент на позицию *i* (расширение частичного решения). Отличие заключается в более простом способе построения множества S_i – множества возможных расширений текущего решения. В случае сочетаний, если мы знаем что $b_{i-1}=v$, то $S_i=\{v+1,v+2, \ldots,n\}$. Например, если n=6, i=3 и текущее частичное решение b=(1,3), то для его расширения подходят числа S_3={4,5,6}. Это верно так, как элементы в сочетании идут в порядке возрастания. Реализация получается чуть более простой, чем в случае с перестановками:

```
procedure print;
var j : integer;
begin
  for j := 1 to k do
    write(b[j], ' ');
  writeln;
end;

procedure gen(i : integer);
var j : integer;
begin
  if i > k then begin print; exit; end;
  for j := b[i-1]+1 to n do
    begin
      b[i] := j; {расширяем текущее решение числом j}
      gen(i+1);
      b[i] := j; {возвращаемся к предыдущему решению}
    end;
end;
```

3. Оптимизация перебора

3.1. Метод ветвей и границ

Применение метода перебора с возвратом к решению задач приводит к алгоритмам с очень большой временн*о*й трудоемкостью. Для ускорения вычислений стараются организовать перебор так, чтобы как можно раньше найти решение и в процессе поиска исключать заведомо неподходящие варианты, т. е. сокращать дерево поиска.

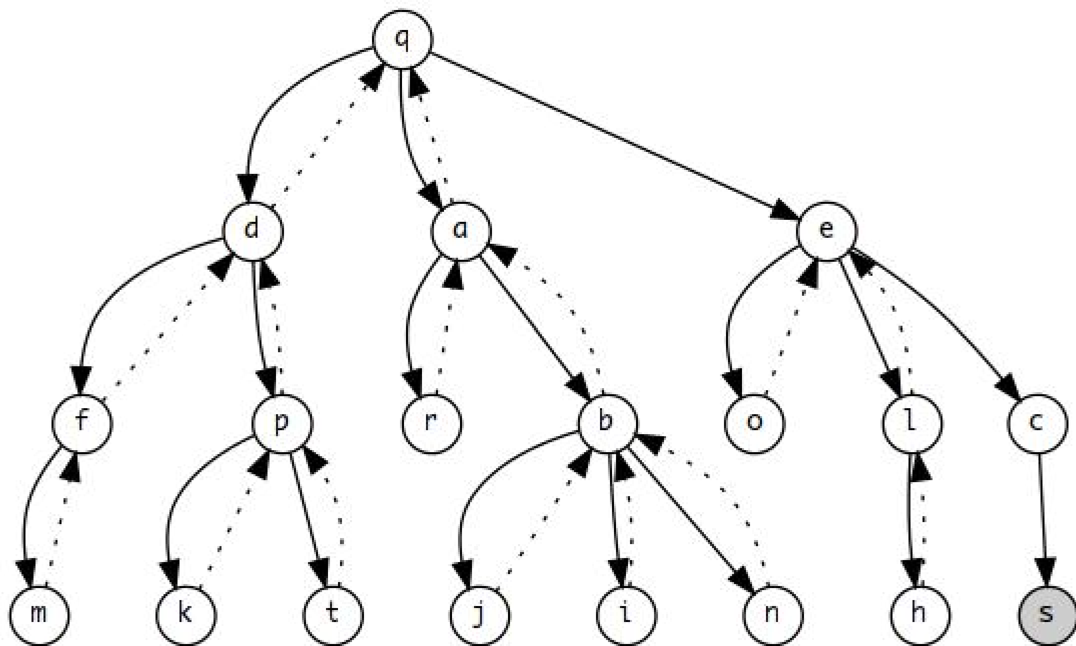

Рис. 7. Пример дерева поиска. Для удобства всем вершинам присвоены случайные идентификаторы. Вершина *s*, соответствует решению.

Заметьте, что для примера на рис. 7, прежде чем найти решение (вершина *s*), было рассмотрено много других вариантов. Уменьшить это количество можно изменив порядок расширения частичного решения. Например, на рис. 7 показано, что частичное решение, соответствующее вершине *q*, расширяется до частичных решений *d*, *a*, *e* (именно в этом порядке). А что если изменить этот порядок? Например, в вершине *q* использовать порядок *d*, *e*, *a*. А в вершине *e* порядок *c*, *o*, *l*. Тогда решение будет найдено раньше (рис. 8).

В общем случае расширения выбирают в порядке уменьшения вероятности нахождения решения. То есть первым выбирается расширение, которое с большей вероятностью приведет к решению. Каким образом оценивается вероятность, зависит от конкретной задачи. Также надо

понимать, что изменив порядок расширения можно и ухудшить время работы (например, если вместо порядка на рис. 8, использовать порядок на рис. 7). Если решения не существует, то изменение порядка не приведет к ускорению, поскольку в этом случае все равно придется перебрать все варианты (пусть и в другом порядке).

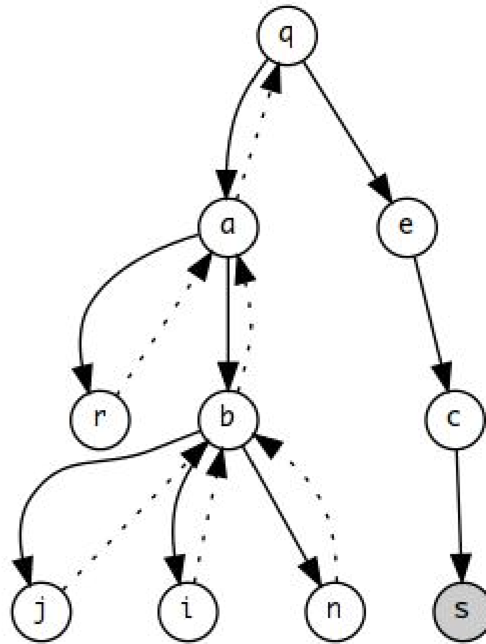

Рис. 8. Дерево поиска с измененным порядком расширения

Другой способ сокращения перебора – как можно более раннее исключение вариантов, которые точно не приведут к решению. Пусть для примера на рис. 7 мы каким-то образом смогли узнать, что расширения частичных решений *a* и *p* не приведут к полным решениям. Это позволит уменьшить общее количество рассматриваемых вариантов (рис. 9).

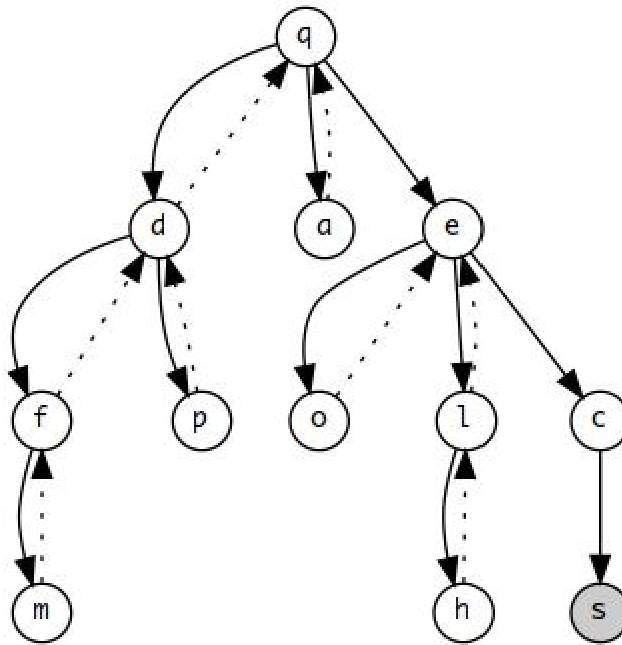

Рис. 9. Дерево поиска с учетом исключений

Наибольший эффект два рассмотренных приема (изменение порядка расширения и исключение неподходящих вариантов) приносят, если их применять одновременно (рис. 10).

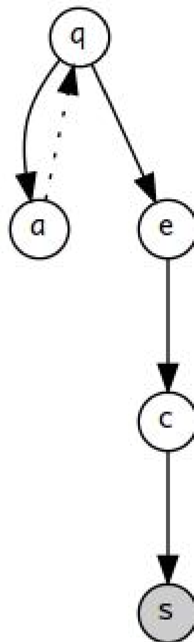

Рис. 10. Дерево поиска с измененным порядком расширения и с учетом исключений

Данный способ оптимизации перебора обычно реализуется с помощью *метода ветвей и границ*. Метод основан на упорядоченном переборе решений и рассмотрении только тех из них, которые являются по

определенным признакам перспективными, и отбрасывании сразу целых множеств решений, которые бесперспективны.

Для применения метода ветвей и границ должна быть определена стоимость частичных решений; кроме того, для всех частичных решений $(b_1, b_2, \ldots, b_{k-1})$ и для всех расширений $(b_1, b_2, \ldots, b_{k-1}, b_k)$ должно выполняться условие $\mathrm{cost}(b_1, b_2, \ldots, b_{k-1}) \le \mathrm{cost}(b_1, b_2, \ldots, b_{k-1}, b_k)$. Здесь через $\mathrm{cost}(b_1, b_2, \ldots, b_k)$ обозначена стоимость частичного решения $(b_1, b_2, \ldots, b_k)$.

Условия задачи часто позволяют устанавливать границы (верхние и/или нижние) для стоимости решения. Мы можем отбросить частичное решение $(b_1, b_2, \ldots, b_{k-1}, b_k)$, если его стоимость превышает верхнюю границу. Мы можем также отбросить частичное решение, если по его стоимости можем определить, что ни при каком расширении этого частичного решения не будет достигнута нижняя граница.

3.2. Представление числа в виде суммы заданных слагаемых

Задано множество $X = \{x_1, x_2, \ldots, x_n\}$ целых положительных чисел и число S. Требуется узнать, может ли число S быть представлено как сумма некоторых из чисел множества X, если каждое число можно использовать не более чем по одному разу. Например, если X={2,3,4,5,6,8} и S=21, то ответ будет следующий: 21=2+5+6+8=3+4+6+8.

Решение будем представлять в виде вектора $B = (b_1, b_2, \ldots, b_n)$, где $b_i = 1$, если i-ое число входит в разложение и $b_i = 0$ иначе. Так для рассмотренного примера существует два решения (1,0,0,1,1,1) и (0,1,1,0,1,1).

Можно написать переборное решение, которое будет перечислять все возможные вектора B, а в конце проверять, что получившаяся сумма равна S.

```
var S, n, i : integer;
    x, b : array[1..20] of integer;

procedure print;
var j, z : integer;
begin
  z := 0;
  for j := 1 to n do
    z := z+x[j]*b[j]; {считаем сумму}
  if z = S then {если сумма совпала, то выводим ответ}
  begin
    for j := 1 to n do
      if (b[j] = 1) then
```

```
        write(x[j], ' ');
      writeln;
    end;
  end;

  procedure dfs(k : integer);
  begin
    if k > n then begin print; exit; end;
    b[k] := 1; {используем число k в разложении}
    dfs(k+1);
    b[k] := 0; {пропускаем число k в разложении}
    dfs(k+1);
  end;

  begin
    read(S);      {сначала читаем число S}
    read(n);      {читаем количество элементов множества X}
    for i := 1 to n do
      read(x[i]); {читаем сами элементы}
    dfs(1);       {запускаем перебор}
  end.
```

Всего существует 2^n векторов длины n из 0 и 1. Общее количество вершин в дереве поиска равно $2^{n+1}-1$ (если n=6, то вершин 127). Попробуем сократить перебор, используя метод ветвей и границ.

Пусть $B = (b_1, b_2, \ldots, b_k)$ – частичное решение. Определим стоимость cost(B), как сумму чисел из множества X, которым соответствует значение 1 в B. Отметим два момента:

а) стоимость любого частичного вектора B, расширение которого может привести к решению задачи, не может быть больше S, т.е. $cost(B) \le S$. Таким образом, мы определили верхнюю границу для стоимости частичных решений. В процессе перебора частичных решений будем проверять полученное неравенство и не строить расширения для B в случае, когда $cost(B) > S$;

б) для любого частичного вектора B сумма еще нерассмотренных элементов не может быть меньше, чем разность $S - cost(B)$ (нельзя отбросить слишком много, иначе нам не хватит оставшихся элементов для получения S).

Оба условия можно записать в виде формулы:

$$S - \sum_{i=k+1}^{n} x_i \le \mathrm{cost}(B) \le S.$$

Чтобы эффективно проверять верхнюю и нижнюю границы во время перебора, придется изменить функцию dfs. Кроме номера шага, который определяет глубину текущей вершины в дереве поиска, она будет также получать:

- *cost* – сумму элементов, использованных в разложении на данный момент;
- *left* – сумму оставшихся элементов.

Изначально *cost*=0, $left = \sum_{i=1}^{n} x_i$.

```
var S, L, n, i : integer;
    x, b : array[1..20] of integer;

procedure print;
var j : integer;
begin
  for j := 1 to n do
    if (b[j] = 1) then
      write(x[j], ' ');
  writeln;
end;

procedure dfs(k, cost, left : integer);
begin
  if (cost > S) or (cost < S-left) then exit;
  if (k > n) then begin
    if cost = S then
      print;
    exit;
  end;
  b[k] := 1; {используем число k в разложении}
  dfs(k+1, cost+x[k], left-x[k]);
  b[k] := 0; {пропускаем число k в разложении}
  dfs(k+1, cost, left-x[k]);
end;
```

```
begin
  L := 0;
  read(S); {сначала читаем число S}
  read(n); {читаем количество элементов множества X}
  for i := 1 to n do
  begin
    read(x[i]); {читаем сами элементы}
    L := L+x[i];
  end;
  dfs(1, 0, L); {запускаем перебор}
end.
```

В случае если X={2,3,4,5,6,8} и S=21, то дерево поиска содержит 67 вершин (сравните со 127 в неоптимизированном алгоритме).

Изменим теперь порядок расширения частичных решений. А именно, предположим, что числа в X упорядочены в порядке убывания (если это не так, то отсортируем их предварительно). Таким образом, на первом шаге перебирается самое большое число. Это приводит к тому, что ограничение на нижнюю и верхнюю границу нарушается раньше и соответственно раньше происходит исключение ненужных вариантов. Так для рассматриваемого примера дерево поиска будет содержать 27 вершин.

Табл. 1. Процесс работы алгоритма, основанного на методе ветвей и границ (до нахождения первого решения) для S=21, X={8,6,5,4,3,2}. Серая строка соответствует найденному решению. Строки, выделенные жирным шрифтом, представляют частичные решения, в которых нарушается ограничение на границы.

Шаг	Решение	*k*	*cost*	*left*	Нижняя граница	Верхняя граница
1	( )	1	0	28	–7	21
2	(1)	2	8	20	1	21
3	(11)	3	14	14	7	21
4	(111)	4	19	9	12	21
5	**(1111)**	**5**	**23**	**5**	**16**	**21**
6	(1110)	5	19	9	12	21
7	**(11101)**	**6**	**22**	**6**	**15**	**21**
8	(11100)	6	19	9	12	21
9	(111001)	7	21	7	14	21

4. Задания для самостоятельного решения

1. Для всех приведенных функций написать код запускающих программ, предусматривающий ввод и вывод данных. Скомпилировать и запустить программы на ряде примеров.

2. Напишите программу, которая выводит все перестановки длины N в обратном лексикографическом порядке. Например, если N=3, то она должна вывести 321, 312, 231, 213, 132, 123.

3. Напишите программу, которая выводит все перестановки длины N такие, что любые два соседних элемента отличаются как минимум на два. Например, если N=4, то это 1324 и 2413.

4. Напишите программу, которая выводит все последовательности длины N из 0 и 1.

5. Напишите программу, которая выводит все последовательности длины N из 0 и 1 без 2-х единиц подряд. Например, если N=3, то это 000, 001, 010, 100, 101.

6. Напишите программу, которая выводит все сочетания из N по K в обратном лексикографическом порядке. Например, если N=5 и K=2, то она должна вывести 54, 53, 52, 51, 43, 42, 41, 32, 31, 21.

7. Напишите программу, которая выводит все разложения заданного числа N на слагаемые. Например, если N=4, то это 1+1+1+1, 1+1+2, 1+3, 2+2 и 4.

8. Напишите программу для решения задачи о рюкзаке: пусть имеется рюкзак заданной грузоподъемности; также имеется некоторое множество предметов различного веса и различной стоимости; требуется упаковать рюкзак так, чтобы сумма стоимостей упакованных предметов была бы максимальной. Программа должны считывать вместимость рюкзака S, количество предметов N и затем описания предметов. Каждый предмет задается двумя числами – весом и стоимостью. Реализуйте два варианта: перебор всех вариантов и метод ветвей и границ.

Литература

1. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы. М.: Вильямс, 2007.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979.
3. Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К. Алгоритмы: Построение и анализ. – 2-е изд. – М.: Издательский дом «Вильямс», 2005.
4. Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. – М.: Мир, 1980.
5. Шень А. Программирование: теоремы и задачи. – 3-е изд. – М.: МЦНМО, 2007.