# Алгоритмы комбинаторики

В.А. Костин

1997

# Содержание



# 1. Генерация перестановок

## 1.1. Генерация перестановок в лексикографическом порядке

При решении некоторых задач возникает необходимость генерирования перестановок $n$-го порядка. Чаще всего генерирование перестановок связано с задачами перебора, в которых решение данной представляет собой некоторую перестановку, обладающую конкретным заданным свойством. Для поиска искомой перестановки мы перебираем все возможные перестановки и проверяем для каждой из них выполнение этого конкретного свойства. Последовательное генерирование перестановок определяет на множестве всех перестановок некоторый порядок, а именно: пусть $f$ и $g$ перестановки, тогда $f < g$, если в этой генерации перестановка $f$ появляется раньше перестановки $g$.

Например, предположим, мы разобрали кубик Рубика на отдельные части и собрали его случайным образом. После этого мы хотим проверить, можно ли его преобразовать таким образом, чтобы все его грани были одного цвета. Моделирование подобной задачи на компьютере явно потребует генерации перестановок.

С другой стороны, при проектировании решения задачи обычно имеются вполне определённые аргументы в пользу выбора той или иной упорядоченности перестановок при генерации. Фактически, эта упорядоченность определяет алгоритм генерации перестановок. Чаще всего эти алгоритмы строятся по схеме, когда каждая последующая перестановка вычисляется как некоторая функция от предыдущей.

*Замечание.* Интересен вопрос, какого порядка перестановки можно генерировать, в разумное время, на современных ЭВМ? Вследствие того, что общее число перестановок $n$-го

порядка равно $n!$, современные ЭВМ позволяют генерировать перестановки не более чем 16-го порядка (обоснуйте это!).

Вначале мы рассмотрим алгоритмы генерации всех перестановок в лексикографическом порядке. Этот порядок свойственен расположению слов в различных словарях, поэтому его часто называют словарным. Он характеризуется тем, что буквы алфавита считаются упорядоченным множеством, а слова в словаре располагаются вначале с меньших (ранее перечисленных в алфавите) букв, а затем с больших. Слова, начинающиеся с одинаковых букв, располагаются в зависимости от упорядоченности вторых букв в слове, и так далее. Для перестановок множества $\{1, 2, \ldots, n\}$ числа считаются упорядоченными естественным образом. Формально можно дать следующее определение лексикографического порядка для перестановок.

**Определение 1.** Пусть $f = \langle a_1, \ldots, a_n \rangle$, $g = \langle b_1, \ldots, b_n \rangle$, будем говорить, что $f < g$ в лексикографическом порядке, если существует $k \geq 1$ такое, что $a_k < b_k$ и $a_q = b_q$ для $q < k$.

**Пример.** При $n = 4$ в лексикографическом порядке перестановки располагаются так:

| | | | |
|---|---|---|---|
| 1. $\langle 1,2,3,4 \rangle$ | 7. $\langle 2,1,3,4 \rangle$ | 13. $\langle 3,1,2,4 \rangle$ | 19. $\langle 4,1,2,3 \rangle$ |
| 2. $\langle 1,2,4,3 \rangle$ | 8. $\langle 2,1,4,3 \rangle$ | 14. $\langle 3,1,4,2 \rangle$ | 20. $\langle 4,1,3,2 \rangle$ |
| 3. $\langle 1,3,2,4 \rangle$ | 9. $\langle 2,3,1,4 \rangle$ | 15. $\langle 3,2,1,4 \rangle$ | 21. $\langle 4,2,1,3 \rangle$ |
| 4. $\langle 1,3,4,2 \rangle$ | 10. $\langle 2,3,2,4 \rangle$ | 16. $\langle 3,2,4,1 \rangle$ | 22. $\langle 4,2,3,1 \rangle$ |
| 5. $\langle 1,4,2,3 \rangle$ | 11. $\langle 2,4,1,3 \rangle$ | 17. $\langle 3,4,1,2 \rangle$ | 23. $\langle 4,3,1,2 \rangle$ |
| 6. $\langle 1,4,3,2 \rangle$ | 12. $\langle 2,4,3,1 \rangle$ | 18. $\langle 3,4,2,1 \rangle$ | 24. $\langle 4,3,2,1 \rangle$ |

Лексикографический порядок может быть интерпретирован так. Пусть каждая перестановка интерпретируется как целое число, записанное в $n$-ричной позиционной системе (с цифрами «0», «1», …, «$n-1$», «$n$»). Тогда генерация их в лексикографическом порядке — это перечисление в порядке возрастания чисел, состоящих из $n$ разных цифр. Наша цель заключается в построении алгоритма генерации всех перестановок в лексикографическом порядке для произвольного $n$. Для этого мы выясним, как должна выглядеть следующая перестановка в генерации, если мы знаем текущую, при условии, что текущая перестановка не является последней в генерации.

Рассмотрим подробнее свойства лексикографического порядка генерации перестановок:

**Л1** В первой перестановке элементы располагаются в возрастающей последовательности, в последней — в убывающей (докажите это свойство для произвольного $n$).

**Л2** Последовательность всех перестановок можно разбить на $n$ блоков длины $(n-1)!$, соответствующих возрастающим значениям элемента в первой позиции. Остальные $n-1$ позиций блока, содержащего элемент $p$ в первой позиции, определяют последовательность перестановок множества $\{1, \ldots, n\} \setminus \{p\}$ в лексикографическом порядке.

Это свойство легко иллюстрируется приведенным примером генерации перестановок 4-порядка. Нетрудно видеть, что все перестановки 4-порядка разбиты на четыре столбца, при этом у перестановок первого столбца на 1-ой позиции расположен элемент 1, у второго — элемент 2 и так далее. Кроме того, в каждом столбце элементы, расположенные в перестановках со 2-ой по 4-ую позиции, образуют перестановки этих элементов в лексикографическом порядке. Для первого столбца перестановки элементов $\{2, 3, 4\}$; для второго — $\{1, 3, 4\}$; третьего — $\{1, 2, 4\}$; для четвертого $\{1, 2, 3\}$. Отметим также, что в каждом столбце элементы, расположенные со 2-ой по 4-ую позиции в первой перестановке, образуют возрастающую последовательность, а в последней перестановке эти же элементы расположены в убывающей последовательности (свойство Л1 лексикографического порядка).

Таким образом, если мы рассмотрим перестановки каждого столбца, то для элементов, расположенных со 2-ой по 4-у позиции, полностью выполняются свойства Л1 и Л2. Это замечание приводит к следующему обобщению свойства Л2 для перестановок произвольного порядка:

**Л3** Последовательность всех перестановок можно разбить на $n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$ блоков выбором значений $p_1, \ldots, p_k$ элементов, расположенных на первых $k$ позициях. При этом блок $p_1, \ldots, p_k$ предшествует блоку $q_1, \ldots, q_k$, если $p_1, \ldots, p_k$ меньше $q_1, \ldots, q_k$ в лексикографическом порядке. Кроме того, для перестановок каждого такого обобщённого блока элементы, расположенные с $k+1$-ой по $n$-ую позиции, представляют собой генерацию перестановок этих элементов в лексикографическом порядке.

Теперь мы готовы сформулировать самое важное свойство лексикографического порядка, на основе которого легко преобразовать текущую перестановку в следующую. Это свойство можно записать так:

**Л4** Любая текущая перестановка является заключительной для некоторого обобщённого блока. Этот блок определяется элементами текущей перестановки, расположенными на позициях в конце перестановки и представляющими собой максимально возможную убывающую последовательность значений элементов перестановки. Справедливость последнего замечания следует из свойства Л1 лексикографического порядка.

В дальнейшем максимально возможную убывающую последовательность значений элементов перестановки, расположенную на позициях в конце перестановки, будем называть «хвостом» перестановки.

**Пример.** Рассмотрим приведенную выше генерацию перестановок 4-го порядка, тогда:

- Перестановка $\langle 2,1,4,3 \rangle$ является заключительной для блока, состоящего из перестановок 7. $\langle 2,1,3,4 \rangle$ и 8. $\langle 2,1,4,3 \rangle$, элементы $4,3$ образуют её хвост
- Перестановка $\langle 3,1,2,4 \rangle$ является заключительной для блока, состоящего из одной перестановки 13. $\langle 3,1,2,4 \rangle$, её хвост состоит из одного элемента — $4$
- Перестановка $\langle 2,4,3,1 \rangle$ является заключительной для второго столбца приведенной генерации, её хвост — $4,3,1$
- Перестановка $\langle 4,3,2,1 \rangle$ является заключительной для всей генерации перестановок 4-го порядка, её хвост совпадает со всей перестановкой.

Как же воспользоваться этим свойством, чтобы преобразовать текущую перестановку в следующую? Это можно сделать по следующему алгоритму:

1. Выделяем хвост текущей перестановки
2. Если он не совпадает со всей перестановкой, то ищем в хвосте первый с конца перестановки элемент, больший элемента перестановки, расположенного непосредственно перед её хвостом (если перестановка совпадает со своим хвостом, то она является заключительной во всей генерации)
3. Меняем местами элемент, найденный в предыдущем пункте, с элементом, расположенным непосредственно перед хвостом перестановки
4. Располагаем все элементы, преобразованного в пункте 3 хвоста перестановки, в обратном порядке (инвертирование преобразованного хвоста перестановки)

**Пример.** Перестановка $\langle 2,1,4,3 \rangle$ преобразуется по вышеприведенному алгоритму в перестановку $\langle 2,3,1,4 \rangle$, а перестановка $\langle 3,1,2,4 \rangle$ в $\langle 3,1,4,2 \rangle$.

Рассмотрим перестановку 15-порядка $\langle 15,2,4,3,1,13,7,10,14,12,11,9,8,6,5 \rangle$, она преобразуется в перестановку $\langle 15,2,4,3,1,13,7,11,5,6,8,9,10,12,14 \rangle$

**Упражнение.** В какие перестановки преобразуются следующие перестановки:

1. $n = 3$, $\langle 2, 3, 1 \rangle$
2. $n = 5$, $\langle 2, 5, 4, 3, 1 \rangle$
3. $n = 7$, $\langle 4, 5, 2, 3, 1, 6, 7 \rangle$
4. $n = 8$, $\langle 2, 4, 3, 6, 8, 7, 5, 1 \rangle$

*Замечание.* Можно провести формальное доказательство того факта, что преобразованная по данному алгоритму перестановка действительно совпадает со следующей после текущей перестановки в лексикографическом порядке. Однако мы такое доказательство опустим.

*Замечание.* Приведенный выше алгоритм преобразования текущей перестановки в следующую формально можно записать так:

Пусть $p = \langle p_1, \ldots, p_k, \ldots, p_j, \ldots, p_n \rangle$, где $1 \leq k < n$, $p_k < p_{k+1}$ и для $q$: $k < q < n$ следует $p_q > p_{q+1}$ (если $p = \langle n, n-1, \ldots, 1 \rangle$, то $k = 0$), $j > k$ и $p_j > p_k$ и для $q$: $j < q \leq n$ следует $p_q < p_k$; тогда следующая за $p$ перестановка, при $k \neq 0$, имеет вид $\langle p_1, \ldots, p_{k-1}, p_j, p_n, p_{n-1}, \ldots, p_{j+1}, p_k, p_{j-1}, \ldots, p_{k+1} \rangle$.

Теперь мы легко можем написать на языке Паскаль алгоритм генерации всех перестановок в лексикографическом порядке. Он основан на поиске $k$ и $j$ в текущей перестановке, транспозиции элементов $p_k$ и $p_j$, и инвертировании ее «хвоста»:

```
program LEX;
    const n = ...; {порядок перестановок}
    var p: array [0..n] of 0..n; { текущая перестановка}
    k: 0..n;
    j, r, m: 1..n;
begin
    for k := 0 to n do
        p[k] := k; {задание начальной перестановки}
    k := 1;
    while k <> 0 do
    begin
        for k := 1 to n do
            write(p[k]);
        writeln; {вывод перестановки}
        k := n-1;
        while p[k] > p[k+1] do
            k := k - 1; {поиск k}
        j := n;
        while p[k] > p[j] do
            j := j - 1; {поиск j}
        r := p[k];
        p[k] := p[j];
        p[j] := r; {транспозиция p_k и p_j }
        j := n;
        m := k + 1;
        while j > m do {инвертирование хвоста перестановки}
        begin
            r := p[j];
            p[j] := p[m];
            p[m] := r;
            j := j - 1;
            m := m + 1
```

```
        end
    end
end.
```

*Комментарий.* Нулевой элемент включен в массив $p$ для того, чтобы обеспечить конец цикла `{поиск k}` после генерации последней перестановки.

**Упражнение.** Протестируйте приведенную выше программу при $n = 3$.

Оценим временную вычислительную сложность приведенной программы. Обычно временная вычислительная сложность программ, представленных на языке высокого уровня, оценивается как порядок роста числа исполняемых операторов программы в зависимости от некоторого параметра исходных данных [3]. Однако, в алгоритмах генерации перестановок такой подход малоэффективен, так как в процессе работы любого алгоритма генерации всех перестановок порождается $n!$ перестановок, т.е. временная вычислительная сложность всегда будет, по крайней мере, $O(n!)$ — величина слишком быстро растущая. Любая «экономия» в реализации будет сказываться только на коэффициенте пропорциональности при $n!$. Поэтому для того, чтобы удобнее было сравнивать различные алгоритмы генерации перестановок, обычно вводят другие критерии оценки вычислительной сложности. Здесь разумно ввести два критерия — количество транспозиций элементов перестановки, выполняемых в среднем при генерации одной перестановки, и аналогичное среднее числа сравнений элементов перестановки в операторах `{поиск k}` и `{поиск j}`.

Оценим их число. Пусть $T_k$ — число транспозиций, выполняемых при вызове оператора `LEC(n-k+1)`, т.е. $T_k$ — число транспозиций, которые необходимо выполнить при генерации перестановок $k$-го порядка. Имеем

$$T_k = k \cdot T_{k-1} + (k-1) \cdot (1 + \lfloor \frac{k-1}{2} \rfloor) = k \cdot T_{k-1} + (k-1) \cdot \lfloor \frac{k+1}{2} \rfloor \quad ,$$

где $\lfloor \alpha \rfloor$ = «целой части числа $\alpha$».

Первое слагаемое определяет число транспозиций при вызовах оператора `LEC(n-k)`, а второе — число транспозиций, выполняемых в операторах `{3}` и `{4}`. Заметим, что $T_1 = 0$.

Для решения этого рекуррентного уравнения сделаем замену переменной. Пусть $S_k = T_k + \lfloor \frac{k+1}{2} \rfloor$, тогда

$$S_1 = 1, S_k = k \cdot (S_{k-1} + \delta_{k-1}) \quad ,$$

где $\delta_k = 0$, если $k$ нечетно, и $\delta_k = 1$, если $k$ четно.

Решение этого рекуррентного соотношения легко получается по индукции:

$$S_k = k! \sum_{j=0}^{\lfloor (k-1)/2 \rfloor} \frac{1}{(2j)!},$$

т.е.

$$T_k = k! \sum_{j=0}^{\lfloor (k-1)/2 \rfloor} \frac{1}{(2j)!} - \lfloor \frac{k+1}{2} \rfloor$$

Учитывая, что $\sum_{j=0}^{m} \frac{1}{(2j)!} \approx \mathrm{ch}(1) \approx 1.543$ и $\lfloor (k+1)/2 \rfloor = o(k!)$, получаем

$$T_k \approx k! \cdot \mathrm{ch}(1) \quad ,$$

т.е. на генерирование одной перестановки в лексикографическом порядке требуется в среднем $\mathrm{ch}(1) \approx 1.543$ транспозиций.

Перейдем теперь к оценке числа сравнений элементов перестановки в операторах `{поиск k}` и `{поиск j}`; обозначим это число $C_n$.

Определим $C_n$ как функцию от $C_{n-1}$. Отметим, что при генерации каждого из $n$ блоков, определенных в Л2, требуется $C_{n-1}$ сравнений, а таких блоков $n$. Далее, при переходе от одного блока к другому оператор `{поиск k}` выполняется $n-1$ раз, а оператор `{поиск j}` при переходе от блока $p$ к блоку $p+1$ $(1 \le p < n)$ — $p$ раз, т.е.

$$C_n = n \cdot C_{n-1} + (n-1) \cdot (n-1) + 1 + \ldots + (n-1)\,, \quad C_1 = 0$$

или

$$C_n = n \cdot C_{n-1} + (n-1) \cdot (3n-2)/2\,, \quad C_1 = 0$$

Пусть $D_n = C_n + (3n+1)/2$, тогда $D_1 = 2$, $D_n = n \cdot D_{n-1} + 3/2$, и по индукции получаем

$$D_n = n! \cdot (\frac{1}{2} + \sum_{i=1}^{n} \frac{3/2}{i!})$$

или учитывая, что $e = \sum_{i=0}^{\infty} \frac{1}{i!}$, получаем $D_n \approx n! \cdot (\frac{3}{2}e - 1)$ Тогда, $C_n = n! \cdot (\frac{3}{2}e - 1) - (3n+1)/2$, учитывая, что $(3n+1)/2 = o(n!)$, получаем $\frac{C_n}{n!} = \frac{3}{2}e - 1$. Таким образом, на генерирование одной перестановки алгоритмом LEX в среднем выполняется $\frac{3}{2}e - 1 \approx 3.077$ сравнений.

*Замечание.* Применение методов рекурсивного программирования не требует выяснения того факта, как выглядит следующая перестановка после текущей в лексикографическом порядке. Легко построить соответствующую рекурсивную программу непосредственно на основе свойств Л1–Л3. Эта рекурсивная программа может быть написана так:

```
program LEX1 (output);
    const n = ...; {n порядок перестановок}
    var p: array [1..n] of 1..n;
    i, r: 1..n;
procedure INVERT(m: integer); {инвертирование p[m]...p[n] }
var i, j: 1..n;
begin
    i := m;
    j := n;
    while i < j do
    begin
        r := p[i];
        p[i] := p[j];
        p[j] := r;
        i := i + 1;
        j := j - 1
    end
end {INVERT};
procedure Lec(k:integer);
var i: 1..n;
begin
    if k = n then
{1} begin
        for i := 1 to n do
            write(p[i]);
        writeln
    end
    else
        for i := n downto k do
{2}     begin
```

```
            LEC(k+1);
            if i > k then
            begin
                r := p[i];
                p[i] := p[k];
                p[k] := r;
{3}             INVERT(k+1)
{4}         end
        end
end {LEC};
begin
    for i := 1 to n do
        p[i] := i;
    LEC(1)
end.
```

*Комментарий.* Процедура `INVERT` служит для восстановления первоначальной перестановки (свойство Л1) после генерации всех перестановок данного обобщенного блока. Процедура `LEC` осуществляет либо печать перестановки (строка `{1}`), если все $n$ позиций уже сформированы, либо (по свойству Л2) генерирует перестановки $n-k+1$ порядка как последовательность $n-k+1$ блоков перестановок $n-k$ порядка с возрастающим по значению элементом на $k$ позиции.

Тест $n = 3$

| | | | |
|---|---|---|---|
| {2} | k=1 | i=3 | |
| {2} | k=2 | i=3 | |
| {1} | k=3 | | вывод $\langle 1\,2\,3\rangle$ |
| {3} | k=2 | i=3 | $p = \langle 1\,3\,2\rangle$ |
| {4} | k=2 | | $p = \langle 1\,3\,2\rangle$ |
| {2} | k=2 | i=2 | |
| {1} | k=3 | | вывод $\langle 1\,3\,2\rangle$ |
| {3} | k=1 | i=3 | $p = \langle 2\,3\,1\rangle$ |
| {4} | k=1 | | $p = \langle 2\,1\,3\rangle$ |
| {2} | k=1 | i=2 | |
| {2} | k=2 | i=3 | |
| {1} | k=3 | | вывод $\langle 2\,1\,3\rangle$ |
| {3} | k=2 | i=3 | $p = \langle 2\,3\,1\rangle$ |
| {4} | k=2 | | $p = \langle 2\,3\,1\rangle$ |
| {2} | k=2 | i=2 | |
| {1} | k=3 | | вывод $\langle 2\,3\,1\rangle$ |
| {3} | k=1 | i=2 | $p = \langle 3\,2\,1\rangle$ |
| {4} | k=1 | | $p = \langle 3\,1\,2\rangle$ |
| {2} | k=1 | i=3 | |
| {3} | k=2 | i=3 | |
| {1} | k=3 | i=3 | вывод $\langle 3\,1\,2\rangle$ |
| {3} | k=2 | i=3 | $p = \langle 3\,2\,1\rangle$ |
| {4} | | | $p = \langle 3\,2\,1\rangle$ |
| {2} | k=3 | i=2 | |
| {3} | k=2 | | вывод $\langle 3\,2\,1\rangle$ |

**Упражнение.** Проведите формальное доказательство правильности алгоритма процедуры `LEC`.

*Указание.* Методом математической индукции докажите, что если $p[k] < \ldots < p[n]$, то вызов `LEC(k)` приводит к генерированию всех перестановок множества $p[k], \ldots, p[n]$ в лексикографическом порядке при неизменных значениях $p[1], \ldots, p[k-1]$.

Для оценки сложности приведенной рекурсивной программы наряду со средним числом количества транспозиций элементов перестановки, нам необходимо определить среднее число вызовов процедуры `LEC`, как функции от $n$ — порядка перестановок.

Пусть $B_n$ — число вызовов процедуры `LEC` при генерации всех перестановок $n$-порядка программой `LEX1`. Для $B_n$ справедливо следующее рекуррентное соотношение:

$$B_1 = 1, B_n = n \cdot B_{n-1} + 1$$

Его решением является последовательность $B_n = n! \cdot \sum_{i=1}^{n} \frac{1}{i!}$. Это приводит к тому, что в среднем на одну перестановку приходится $e - 1$ вызовов процедуры `LEC`. Этот результат

позволяет сравнить алгоритмы `LEX` и `LEX1`. Фактически сравнение сводится к оценке того, что эффективнее реализуется на конкретной ЭВМ: $e-1$ вызовов рекурсивной процедуры или 3.077 сравнений целых чисел.

Наряду с лексикографическим порядком достаточно часто встречается генерирование перестановок в антилексикографическом порядке.

**Определение 2.** Пусть $f = \langle a_1, \ldots, a_n \rangle$, $g = \langle b_1, \ldots, b_n \rangle$, будем говорить, что $f < g$ в антилексикографическом порядке, если существует $k \leq n$ такое, что $a_k > b_k$ и $a_q = b_q$ для $q > k$.

**Пример.** При $n = 4$ в антилексикографическом порядке перестановки располагаются так:

| | | | |
|---|---|---|---|
| 1. $\langle 1,2,3,4 \rangle$ | 7. $\langle 1,2,4,3 \rangle$ | 13. $\langle 1,3,4,2 \rangle$ | 19. $\langle 2,3,4,1 \rangle$ |
| 2. $\langle 2,1,3,4 \rangle$ | 8. $\langle 2,1,4,3 \rangle$ | 14. $\langle 3,1,4,2 \rangle$ | 20. $\langle 3,2,4,1 \rangle$ |
| 3. $\langle 1,3,2,4 \rangle$ | 9. $\langle 1,4,2,3 \rangle$ | 15. $\langle 1,4,3,2 \rangle$ | 21. $\langle 2,4,3,1 \rangle$ |
| 4. $\langle 3,1,2,4 \rangle$ | 10. $\langle 4,1,2,3 \rangle$ | 16. $\langle 4,1,3,2 \rangle$ | 22. $\langle 4,2,3,1 \rangle$ |
| 5. $\langle 2,3,1,4 \rangle$ | 11. $\langle 2,4,1,3 \rangle$ | 17. $\langle 3,4,1,2 \rangle$ | 23. $\langle 3,4,2,1 \rangle$ |
| 6. $\langle 3,2,1,4 \rangle$ | 12. $\langle 4,2,1,3 \rangle$ | 18. $\langle 4,3,1,2 \rangle$ | 24. $\langle 4,3,2,1 \rangle$ |

**Упражнение.**

1. Сформулируйте свойства А1-А3 антилексикографического порядка, аналогичные свойствам Л1–Л3 для лексикографического порядка.

2. Определите соотношения между некоторой перестановкой и непосредственно следующей за ней в антилексикографическом порядке.

3. Постройте нерекурсивный алгоритм ANTILEX, порождающий все перестановки в антилексикографическом порядке (сравните с `[1]`).

4. Постройте рекурсивный алгоритм ANTILEX1, порождающий все перестановки в антилексикографическом порядке.

## 1.2. Генерация перестановок за минимальное число транспозиций

При конструировании алгоритмов с использованием генерации перестановок, рассмотренные алгоритмы генерации в лексикографическом или антилексикографическом порядке иногда мало эффективны.

**Пример.** Пусть задана квадратная матрица вещественных чисел $\|a_{ij}\|$ порядка $n$. Вычислить

$$\sum a[1,i_1] \cdot a[2,i_2] \cdot \ldots \cdot a[n,i_n] \tag{1}$$

где суммирование проводится по всем перестановкам $\langle i_1\, i_2\, \ldots\, i_n \rangle$ $n$-го порядка.

Выражение (1) называется перманентом [1] матрицы $||a_{ij}||$ и весьма похоже на ее определитель. Заметим, что вычисление перманента или определителя непосредственно по определению имеет сложность $O(n!)$, так как требуется генерация всех перестановок $n$-го порядка. Поэтому машинные алгоритмы вычисления определителей никогда не строятся непосредственно на его определении, а используют ряд его свойств, уменьшающих вычислительную сложность. Перманенты, однако, подобными свойствами не обладают, поэтому для их вычисления вполне приемлем подход, основанный на их определении. Остановимся на таком алгоритме подробнее.

Как уже отмечалось, подобный алгоритм требует генерирования всех перестановок $n$-го порядка. Однако генерация перестановок в лексикографическом или антилексикографическом порядке нецелесообразна. Здесь более желателен такой алгоритм генерирования, при

[1] Перманенты квадратных матриц является частным случаем перманентов прямоугольных матриц. Теория перманентов рассматривается в комбинаторной математике. Она имеет приложение к решению теоретико-вероятностных, комбинаторных и физических задач [3].

котором каждая последующая перестановка отличалась бы от предыдущей на транспозицию только двух ее элементов.

Действительно, пусть последующая перестановка отличается от предыдущей на транспозиции $[i_k, i_j]$, $k \neq j$; тогда для того, чтобы вычислить произведение, соответствующее последующей перестановке, не надо выполнять все $n$ умножений, определяемые формулой (1), а достаточно произведение, соответствующее, предыдущей перестановке, умножить на $\dfrac{a[j,i_j]}{a[k,i_k]}$, т.е. можно получить только за одно деление и умножение (считаем, что $a[k,i_k] \neq 0$).

Для полного решения поставленной в примере задачи покажем возможность алгоритма генерирования перестановок, в котором каждая последующая перестановка отличается от предыдущей на одну транспозицию. Опишем только такие из них, которые могут быть представлены по следующей рекурсивной схеме [2].

Пусть генерируемые элементы перестановки содержатся в массиве $p[1], \ldots, p[n]$ и имеется некоторая процедура `PERM(m)`, генерирующая все перестановки элементов, хранящиеся в $p[1], \ldots, p[m]$, $1 \leq m \leq n$, таким образом, что каждая последующая перестановка отличается от предыдущей одной транспозицией. Тогда генерирование перестановок $n$-го порядка может быть представлено как $n$ обращений `PERM(n-1)`, при условии, что при каждом очередном обращении элемент перестановки, хранящийся в $p[n]$, заменяется на другой элемент из $p[1], \ldots, p[n-1]$ так, что элементы в $p[n]$ не повторяются. В общем виде процедура `PERM(m)` может быть описана так:

```
procedure PERM(m: integer); {1 ≤ m ≤ n}
    var i,k: integer;
    {p, b - глобальные массивы}
begin
    if m = 1 then
        {p[1], ... , p[n] содержит новую перестановку}
        for i := 1 to n do
            write(p[i])
    else
        for i := 1 to m do
        begin
            PERM(m-1);
            if i < m then
            begin
                k := p[m];
{*}             p[m] := p[b[m,i]];
                p[b[m,i]] := k
            end
        end
end;
```

Из текста процедуры видно, что элемент `p[m]` после $i$-го вызова процедуры `PERM(m-1)` заменяется элементом `p[b[m,i]]`. Для того чтобы эта схема работала правильно, мы должны определить массив $b_m$ так, чтобы гарантировать, что каждая транспозиция `{*}` вводит новый элемент в $p[m]$. Как корректно построить матрицу $b_m$?

Это может быть сделано рекурсивно. Предположим, что уже построена матрица

$$\begin{Vmatrix} b[1,1] & \ldots & b[1,m] \\ \vdots & \ddots & \vdots \\ b[m,1] & \ldots & b[m,m] \end{Vmatrix}$$

удовлетворяющая требуемым условиям. Заметим, что значения элементов $b[j,i]$, $i \le j \le n$, произвольны (их в дальнейшем будем обозначать $\bullet$). Для построения матрицы $b_{m+1}$ достаточно определить элементы $b[m+1,1], b[m+1,2], \ldots, b[m+1,m]$. Обозначим $\varphi_m$ следующую перестановку чисел $1,\ldots,m$: $\varphi_m(i) =$ индекс $j$, такой, что $p[j]$ содержит начальное значение переменной $p[i]$ после выполнения `PERM(m)`. Тогда в качестве $b[m+1,1]$ можно взять любое значение из $1,\ldots,m$, пусть это будет $k_1$. В качестве $b[m+1,2]$ — любое значение из $1,\ldots,m$, отличное от $\varphi_m(k_1)$; пусть это будет $k_2$. В качестве $b[m+1,3]$ — любое значение из $1,\ldots,m$, отличное от $\varphi_m(\varphi_m(k_1)) = \varphi_m^2(k_1)$ и $\varphi_m(k_2)$; пусть это будет $k_3$, и т.д. В качестве $b[m+1,m]$ окажется единственное значение из $1,\ldots,m$, отличное от $\varphi_m^m(k_1), \varphi_m^{m-1}(k_2), \ldots, \varphi_m(k_{m-1})$.

**Пример.**

$$b_1 = \left\|\bullet\right\|, \quad b_2 = \left\|\begin{matrix}\bullet & \bullet\\ 1 & \bullet\end{matrix}\right\|, \quad b_3 = \left\|\begin{matrix}\bullet & \bullet & \bullet\\ 1 & \bullet & \bullet\\ 1 & 1 & \bullet\end{matrix}\right\|, \quad b_4 = \left\|\begin{matrix}\bullet & \bullet & \bullet & \bullet\\ 1 & \bullet & \bullet & \bullet\\ 1 & 1 & \bullet & \bullet\\ 1 & 2 & 3 & \bullet\end{matrix}\right\|$$

Если в начальный момент в массиве $p$ записана перестановка $\langle 1\,2\,3\,4\rangle$, то после обращения к процедуре `PERM(4)` с матрицей $b_4$ перестановки генерируются в следующем порядке:

| | | | |
|---|---|---|---|
| 1. $\langle 1,2,3,4\rangle$ | 7. $\langle 4,2,1,3\rangle$ | 13. $\langle 1,3,4,2\rangle$ | 19. $\langle 4,3,2,1\rangle$ |
| 2. $\langle 2,1,3,4\rangle$ | 8. $\langle 2,4,1,3\rangle$ | 14. $\langle 3,1,4,2\rangle$ | 20. $\langle 3,4,2,1\rangle$ |
| 3. $\langle 3,1,2,4\rangle$ | 9. $\langle 1,4,2,3\rangle$ | 15. $\langle 4,1,3,2\rangle$ | 21. $\langle 2,4,3,1\rangle$ |
| 4. $\langle 1,3,2,4\rangle$ | 10. $\langle 4,1,2,3\rangle$ | 16. $\langle 1,4,3,2\rangle$ | 22. $\langle 4,2,3,1\rangle$ |
| 5. $\langle 2,3,1,4\rangle$ | 11. $\langle 2,1,4,3\rangle$ | 17. $\langle 3,4,1,2\rangle$ | 23. $\langle 3,2,4,1\rangle$ |
| 6. $\langle 3,2,1,4\rangle$ | 12. $\langle 1,2,4,3\rangle$ | 18. $\langle 4,3,1,2\rangle$ | 24. $\langle 2,3,4,1\rangle$ |

**Упражнение.** Исходя из матрицы $b_3$ постройте матрицу $b_4$ таким образом, чтобы первой генерировалась перестановка $\langle 1\,2\,3\,4\rangle$, а последней, 24-ой, перестановка $\langle 4\,1\,2\,3\rangle$

**Определение 3.**

1. Будем говорить, что матрицы генерирования перестановок $b_n$ и $b_n'$ эквивалентны, если у них соответственно совпадают элементы, лежащие ниже главной диагонали
2. Будем говорить, что две процедуры генерирования перестановок эквивалентны, если исходя из начальной перестановки $\langle 1\,2\,\ldots\,n\rangle$, они генерируют все другие перестановки в одном и том же порядке

**Упражнение.**

1. Доказать, что процедуры `PERM` с двумя неэквивалентными матрицами генерирования не эквивалентны.
2. Доказать, что число $T_n$ неэквивалентных матриц генерирования порядка $n$ равно $\prod_{i=1}^{n-1} i!$

Из упражнения 2 следует, что рекурсивными схемами могут быть описаны $T_n$ способов генерирования перестановок из единичной перестановки $n$-го порядка при условии, что каждая последующая перестановка отличается от предыдущей ровно на одну транспозицию.

*Замечание.* При описании матрицы $b_m$ в виде двухмерного массива больше половины ее элементов (т.е. элементов $b_{ij}$, для которых $j \ge i$) не используются процедурой `PERM`. Поэтому для экономии памяти вместо двухмерной матрицы $b_m$ можно использовать одномерный массив $b_1$ размером $m(m-1)/2$.

**Упражнение.** Модернизируйте процедуру perm таким образом, чтобы вместо двухмерного массива $b_m$ использовался бы одномерный массив $b1$, в котором необходимые для генерации элементы располагались в порядке $b_{21}, b_{31}, b_{32}, \ldots, b_{n1}, b_{n2}, \ldots, bnn-1$.

Наряду с представлением $b_m$ в виде массива часто бывает удобно задавать элементы $b_m$ динамически как значения соответствующей функции от $m$ и $i$.

**Пример.**

```
function b(m, i:integer): integer; {m>i}
begin
    if (m mod 2 = 0) and m > 2 then
        if i < m-1 then
            b := 1
        else
            b := m-2
    else
        b := m-1
end;
```

**Упражнение.** Построить матрицу $b_5$, соответствующую функции $b$, и доказать, что процедура PERM с использованием этой матрицы работает корректно. Обобщить полученный результат для произвольного $n$ (Подробно такое доказательство рассмотрено в [2], там же приведены другие функциональные представления $b(m, i)$.)

Алгоритмы генерирования, построенные по схеме процедуры perm, удобно применять также для генерации перестановок, удовлетворяющих заданным свойствам.

**Упражнение.** Модернизируйте процедуру perm, чтобы она генерировала все возможные «беспорядки» чисел $1, \ldots, n$, т.е. такие перестановки, в которых для любого $i$, $1 \le i \le n$, $p[i] \ne i$.

Возникает естественный вопрос, существует ли генерации перестановок, в которых каждая последующая отличается от предыдущей на транспозицию, и которые не описываются схемой процедуры perm? Ответ на этот вопрос утвердителен.

**Пример.** $n = 4$.

| | | | |
|---|---|---|---|
| 1. $\langle 1234 \rangle$ | 7. $\langle 1342 \rangle$ | 13. $\langle 4321 \rangle$ | 19.$\langle 2431 \rangle$ |
| 2. $\langle 1243 \rangle$ | 8. $\langle 1324 \rangle$ | 14. $\langle 3421 \rangle$ | 20. $\langle 4231 \rangle$ |
| 3. $\langle 1423 \rangle$ | 9. $\langle 3124 \rangle$ | 15. $\langle 3241 \rangle$ | 21. $\langle 4213 \rangle$ |
| 4. $\langle 4123 \rangle$ | 10. $\langle 3142 \rangle$ | 16. $\langle 3214 \rangle$ | 22. $\langle 2413 \rangle$ |
| 5. $\langle 4132 \rangle$ | 11. $\langle 3412 \rangle$ | 17. $\langle 2314 \rangle$ | 23. $\langle 2143 \rangle$ |
| 6. $\langle 1432 \rangle$ | 12. $\langle 4312 \rangle$ | 18. $\langle 2341 \rangle$ | 24. $\langle 2134 \rangle$ |

Нетрудно видеть, что представленная генерация обладает тем свойством, что каждая последующая перестановка в ней отличается от предыдущей на одну транспозицию соседних элементов. Кроме того, приведенная генерация легко обобщается на случай произвольного $n$. А именно, каждая перестановка $(n-1)$-го порядка имеет $n$ позиций, куда может быть помещен элемент $n$, чтобы получить перестановку $n$-го порядка ($n-1$ позиция перед каждым элементом заданной перестановки и 1 — после последнего). Предположим, что построена последовательность перестановок элементов $1, 2, \ldots, n-1$, аналогичная приведенной. Тогда требуемую последовательность перестановок элементов $1, 2, \ldots, n$ можно получить, вставляя элемент $n$ всеми возможными способами в каждую перестановку элементов $1, 2, \ldots, n-1$. При этом $n$ перемещается между последней и первой позициями попеременно назад и вперед $(n-1)!$ раз.

**Упражнение.** Показать, что перестановка $\langle 4\,1\,5\,6\,3\,2 \rangle$ непосредственно следует за перестановкой $\langle 4\,1\,5\,3\,6\,2 \rangle$. Как выглядят две последние перестановки в генерации $n$-го порядка?

Покажем, что такая генерация не может быть получена никаким вариантом процедуры `perm`. Для этого заметим, что при генерации процедурой `perm` каждое конкретное значение хранится в $p[n]$ в $(n-1)!$ последовательных перестановках, а в приведенной генерации каждое конкретное значение сохраняется в $p[n]$ не более чем $(n-1)$-й последовательной перестановке.

**Упражнение.** Доказать, что для произвольного $n$ не существует варианта процедуры `perm`, в котором генерация каждой последующей перестановки получается одной транспозицией соседних элементов в предыдущей.

Перейдем теперь к разработке программы генерации перестановок с помощью одной транспозиции соседних элементов. Приведенная выше генерация получена по рекурсивной схеме, однако непосредственная реализация этой схемы имеет существенный недостаток: для генерации всех перестановок $n$-го порядка требуется хранение всех перестановок $(n-1)$-го порядка. То есть подобный алгоритм имеет емкостную вычислительную сложность $O((n-1)!)$. Для того чтобы уменьшить емкостные характеристики этой схемы, воспользуемся аналогичным приемом, который был применен для построения нерекурсивного варианта генерации перестановок в лексикографическом порядке.

Предположим, что уже сгенерирована перестановка $\langle a_1 \ldots a_n \rangle$, удовлетворяющая необходимым требованиям; тогда для того чтобы построить следующую перестановку, нужно определить элемент $a_k$, $1 \le k \le n$, который перемещается на новую позицию, и направление перемещения. Направление перемещения можно контролировать с помощью специального вектора $d$ длины $n$, $i$-я координата которого принимает значение 1, если элемент $i$ перемещается вперед (от позиций с меньшими номерами к большим), и -1, если элемент $i$ перемещается назад. Изменение значения координаты происходит тогда, когда элемент $i$ достигает крайних позиций.

Нетрудно видеть, что перемещаемым является элемент с наибольшим номером, который еще не достиг своего крайнего положения в направлении своего перемещения; т.е. если мы удалим из перестановки все элементы со значениями больше $a_k$, то в оставшейся перестановке $a_k$-го порядка возможно перемещение этого элемента в соответствующем ему направлении. Для удобства реализации этого условия вместо перестановки $\langle a_1 \ldots a_n \rangle$ будем рассматривать вектор длины $n+2$, следующего вида $\langle n+1, a_1, \ldots, a_n, n+1 \rangle$. В этом случае перемещаемым в перестановке $\langle a_1 \ldots a_n \rangle$ является элемент с наибольшим номером, перед которым в направлении перемещения не стоит элемент с большим значением. Для быстрого поиска месторасположения в перестановке элемента с конкретным значением введем в программу генерации также перестановку $r$, обратную к $\langle a_1 \ldots a_n \rangle$. Тогда программа генерации выглядит так:

```
program gen(input, output);
    const n = ; n1 = ; {n1 = n + 1}
    var
        p: array [0..n1] of 1..n1;
        {p[1],...,p[n] - генерируемая перестановка}
        r: array [1..n] of 1..n;
        {r - перестановка, обратная к p[1],...,p[n] }
        d: array [1..n] of -1..1; {d - вектор направлений}
        i, j, k, t: integer;
begin
    for i := 1 to n do
    begin
        p[i] := i;
        r[i] := i;
        d[i] := -1
```

```
    end;
    d[1] := 0;
    p[0] := n1;
    p[n1] := n1;
    i := n;
    while i <> 1 do
    begin
        for j := 1 to n do
            write(p[j]);
        writeln;
        i := n;
        while p[ r[i]+d[i] ] > i do
        begin
{*}         d[i] := -d[i]; i := i-1 {поиск перемещаемого элемента}
        end;
        k := r[i];
        t := k + d[i];
        j := p[t];
        p[k] := j; {транспозиция элементов}
        p[t] := i;
        r[i] := r[j]; {корректировка обратной перестановки}
        r[j] := k
    end
end.
```

**Упражнение.**

1. Доказать, что условием окончания генерирования всех перестановок является то, что перемещаемым становится элемент 1.

2. Показать, что операторы строки `{*}` выполняются $\sum_{i=1}^{n} i!$ раз. Определить временную вычислительную сложность алгоритма. Какова его емкостная вычислительная сложность?

3. Написать программу, которая генерирует перестановки с помощью одной транспозиции соседних элементов и при $n = 4$ выдает следующую последовательность:

| | | | |
|---|---|---|---|
| 1. $\langle 1234 \rangle$ | 7. $\langle 3124 \rangle$ | 13. $\langle 4321 \rangle$ | 19. $\langle 4213 \rangle$ |
| 2. $\langle 2134 \rangle$ | 8. $\langle 1324 \rangle$ | 14. $\langle 4312 \rangle$ | 20. $\langle 4231 \rangle$ |
| 3. $\langle 2314 \rangle$ | 9. $\langle 1342 \rangle$ | 15. $\langle 4132 \rangle$ | 21. $\langle 2431 \rangle$ |
| 4. $\langle 2341 \rangle$ | 10. $\langle 3142 \rangle$ | 16. $\langle 1432 \rangle$ | 22. $\langle 2413 \rangle$ |
| 5. $\langle 3241 \rangle$ | 11. $\langle 3412 \rangle$ | 17. $\langle 1423 \rangle$ | 23. $\langle 2143 \rangle$ |
| 6. $\langle 3214 \rangle$ | 12. $\langle 3421 \rangle$ | 18. $\langle 4123 \rangle$ | 24. $\langle 1243 \rangle$ |

Кроме приведенных алгоритмов генерации перестановок, возможны алгоритмы, основанные на других характеристических свойствах упорядочивания. Например, перестановки могут быть упорядочены с учетом числа циклов или инверсий и т.п. Такие алгоритмы достаточно специфичны и здесь не рассматриваются. Классы перестановок с фиксированным числом циклов или инверсий представляют большой интерес в перечислительной комбинаторике и при анализе алгоритмов.

## 2. Генерация подмножеств множества

При программировании на языке Паскаль возможны два подхода к представлению подмножеств конечного множества. В первом, основанном на использовании структурного типа

`SET OF`, базовый тип (элементы множества) задается некоторым простым типом. При этом пользователь может давать имена элементам множества, определять их упорядоченность и применять операции, допустимые над множествами языка Паскаль. Второй подход основан на том, что пользователь интерпретирует множества с помощью других типов, чаще всего массивов. В этом случае, элементы множества трактуются как элементы массива, они могут считаться как упорядоченные, например, по упорядоченности индекса, а выполняемые над ними операции определяются применяемыми операторами или процедурами. Возможна также интерпретация множеств в виде списочных структур. Отметим, что второй подход естественен, когда по условию программируемой задачи элементы сами по себе имеют сложную структуру.

Следует отметить, что описания множеств с использованием типа `SET OF` фактически сводится к представлению множеств в виде упакованных булевских массивов и реализация соответствующих операций над ними. Но для удобства пользователя все это выполняется в процессе трансляции программы.

Для того чтобы более точно описать проблемы, возникающие в нижеописанных алгоритмах, будем считать, что базовое множество содержит $n$ элементов, а каждое конкретное его подмножество представляется в виде массива. При этом $i$-й элемент массива принимает значение 1, если i-й базовый элемент ($1 \leq i \leq n$) принадлежит множеству, и нулю в противном случае (сравните с понятием характеристической функции множества, используемым в математике). Кроме того, считаем, что упорядоченность элементов множества соответствует упорядоченности индексов массива. Будем изображать множества в виде кортежей из $n$ нулей и единиц.

**Пример.** $X = (1, 0, 0, 0, 1, 1)$ означает, что базовое множество содержит шесть упорядоченных элементов, а в множество $X$ включены первый, пятый и шестой по порядку элементы. Пустое множество в этом случае представляется $(0, 0, 0, 0, 0, 0)$, а множество всех элементов $(1, 1, 1, 1, 1, 1)$.

Первой рассмотрим задачу генерации всех подмножеств данного множества. Здесь, как и с перестановками, наиболее распространенными являются генерирование подмножеств в лексикографическом порядке и генерирование, при котором каждое последующее подмножество отличается от предыдущего вставкой или удалением одного элемента.

## 2.1. Генерация подмножеств в лексикографическом порядке

**Определение 4.** Пусть $A = (x_1, \ldots, x_n)$ и $B = (y_1, \ldots, y_n)$, $x_i, y_i \in \{0, 1\}$, $1 \leq i \leq n$. Будем говорить, что множество $A$ предшествует множеству $B$ лексикографическом порядке ,если существует $i$, $1 \leq i \leq n$, $x_i < y_i$ и $x_j = y_j$ для любого $j$, $1 \leq j < i$; обозначим это $A < B$.

**Упражнение.** Пусть $0 \leq a < b < 2^n$, $a$, $b$ — натуральные; представим $a$ и $b$ в двоичной системе с выписыванием всех $n$ двоичных разрядов и пусть $a = \sum_{i=1}^{n} x_i \cdot 2^{n-i}$ и $b = \sum_{i=1}^{n} y_i \cdot 2^{n-i}$. Доказать, что тогда $(x_1, \ldots, x_n) < (y_1, \ldots, y_n)$.

Основываясь на результате упражнения, можно написать следующую программу генерации всех подмножеств данного множества в лексикографическом порядке.

```
program SET1(output);
    const n = ;
          n1 = ; {n1 = n + 1}
    var s: array [1..n1] of 0..1;
        i, j: integer;
begin
    for i := 1 to n1 do
        s[i] := 0;
```

```
    while s[n1] = 0 do
    begin
        for i := 1 to n do
            write(s[i]);
        writeln;
        i := 1;
{**}    while s[i] = 1 do
        begin
            s[i] := 0;
            i := i + 1
        end;
{*}     s[i] := 1
    end
end.
```

*Комментарий.* Пусть множеству $S$ соответствует число $s$, тогда множеству, следующему за $S$, соответствует число $s+1$.

**Упражнение.**

1. Доказать корректность алгоритма `SET1`.
2. Показать, что условие цикла `{**}` проверяется $2^{n+1}-1$ раз.
3. Определить вычислительную сложность алгоритма.
4. Написать программу генерации всех подмножеств в лексикографическом порядке, если $s$ описано как `SET OF 1..n`.

*Замечание.*

1. Следует отметить, что в системе команд любой вычислительной машины имеются команды, выполняющие арифметическое сложение двоичных последовательностей определенной длин, таких, как 8 бит — 1 байт, 16 — 2 байта и тому подобное. Кроме того, часто имеются специальные программные конструкции (например, макрокоманды, выполняющие сложение более длинных двоичных последовательностей). Непосредственное применение таких команд в программе `SET1` может существенно улучшить ее временные характеристики.
2. Для построения других алгоритмов генерации подмножеств, представляет интерес свойства последовательности значений переменной $i$ перед исполнением оператора `{*}`. Заметим, что этот оператор `{*}` исполняется $2^n$ раз, при этом последнее значение $i=n+1$ приводит к окончанию генерации всех подмножеств для множества из $n$-элементов. Пусть $I_n$ обозначает эту последовательность значений переменной $i$ за исключением последнего. $I_n$ можно трактовать как последовательность номеров позиций младшей единицы в двоичном разложении чисел $1, 2, \ldots, 2^{n-1}$. По построению двоичной позиционной системы последовательность $I_n$ удовлетворяет следующему рекурсивному определению $I_1 = 1$, $I_n = I_{n-1}, n, I_{n-1}$ (первый раз единица в $n$-ой позиции появляется для числа $2^{n-1}$, при этом во всех других позициях с 1-ой по $(n-1)$-ую значения становятся нулевыми, затем для чисел $2^{n-1}+1, \ldots, 2^n-1$ повторяется процесс заполнения единицами позиций с номерами меньшими $n$).

**Упражнение.** Доказать, что если $I_{n-1} = i_1, i_2, \ldots, i_{2^{n-1}}$ , $n > 1$, то $I_n = 1, i_1+1, 1, i_2+1, 1, \ldots, i_{2^{n-1}}+1, 1$.

## 2.2. Генерация подмножеств за счет их минимального изменения

Кроме генерации подмножеств данного множества в лексикографическом порядке весьма интересны схемы генерации при которых каждое последующее подмножество отличается от предыдущего вставкой или удалением одного элемента. Первым, естественно, рассмотрим вопрос, связанный с существованием подобных схем. Для этого вначале разберем частные генерации в зависимости от мощности базового множества, затем обобщим их. Считаем, что конкретные подмножества представлены в виде характеристических векторов.

Пусть $n$ - мощность базового множества.

$n = 1$

1. (0), либо 1. (1),
2. (1). 2. (0).

$n = 2$

| | |
|---|---|
| 1. (0,0), либо ему симметричный | 1. (0,0), |
| 2. (1,0), | 2. (0,1), |
| 3. (1,1), | 3. (1,1), |
| 4. (0,1). | 4. (1,0). |

Других вариантов, начинающихся с пустого множества, нет!

$n = 3$

1. (0,0,0)
2. (1,0,0)
3. (1,1,0)
4. (0,1,0)
5. (0,1,1)
6. (1,1,1)
7. (1,0,1)
8. (0,0,1)

**Упражнение.** Построить другие варианты генерации подмножеств для $n = 3$, начинающиеся с представления пустого множества.

$n = 4$

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 |

Обобщим приведенные примеры:

Пусть $C_1, C_2, \ldots, C_{k-1}, C_k$ содержит все $k = 2^n$ двоичных представлений подмножеств множества из $n$ элементов, причем каждое последующее подмножество отличается от предыдущего вставкой или удалением одного элемента. Тогда последовательность, генерирующая все подмножества множества из $n + 1$ элемента может быть получена, например, так:

$$C_1 0, C_2 0; \ldots, C_{k-1} 0, C_k 0, C_k 1, C_{k-1} 1, \ldots, C_2 1, C_1 1 \qquad (2)$$

Так построенные последовательности двоичных слов являются симметрично отраженными относительно $n$-ой позиции.

Рассмотрим последовательность номеров изменяемых разрядов при переходе от одного двоичного слово к другому. Обозначим ее $P_n$. Тогда эта последовательность удовлетворяет следующему рекурсивному определению

$$P_1 = 1, P_n = P_{n-1}, n, P_{n-1}^{\text{перевернутая}}, n > 1$$

По индукции легко доказать, что для $n > 0$ $P_n = P_n^{\text{перевернутая}}$.

Таким образом последовательность $P_n$ совпадает с ранее определенной последовательностью $I_n$.

**Пример.** $n = 4$, $P_4 = I_4 = 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1$.

Заметим, что значение $m$, $1 \le m \le n$, первый раз встречается как $2^{m-1}$-ый член $P_n$, затем повторяется через каждые $2^{m-1}$ членов последовательности.

Учитывая сказанное, можно построить на языке Паскаль следующий рекурсивный алгоритм:

```
program SET2(output);
    const n = ;
    var s: array [1..n] of 0..1;
        i: integer;
procedure GRAY (m: integer);
begin
    if m = 0 then
    begin
        for i := 1 to n do
{1}         write(s[i]);
        writeln;
    end
    else
    begin
{2}     GRAY(m-1);
{3}     s[m] := 1 - s[m];
        GRAY(m-1)
    end
end;
begin
    for i := 1 to n do
        s[i] := 0;
    GRAY(n)
end.
```

**Упражнение.**

1. Проверить, что при $n = 4$ программа действительно генерирует требуемую последовательность.
2. Показать, что в процессе исполнения оператора строки `{2}` вызов процедуры `GRAY` происходит $2^{m-1}$ раз.
3. Доказать корректность программы `SET2`.
4. Определить вычислительную сложность программы `SET2`.

Упорядоченную последовательность двоичных $n$-разрядных наборов обычно называют кодами Грея $n$-го порядка (происхождение этого названия см. [4]), если каждый набор в этой последовательности отличается от предыдущего изменением только одного разряда.

Пусть задана некоторая перестановка $\langle a_1, \ldots, a_n \rangle$. Нетрудно видеть, что если мы в строке `{1}` программы `SET2` заменим оператор печати на `write(s[a[i]])`, то модернизируемая программа будет также строить коды Грея. Их обычно называют симметрично-отраженными. Можно показать, что существует $n!$ различных симметрично-отраженных кодов Грея, начинающихся с нулевого кодового слова.

**Упражнение.** Для какого наименьшего $n$ существует несимметрично отраженный код Грея, начинающийся с нулевого кодового слова?

Перейдем к построению итеративного варианта алгоритма SET2. Здесь достаточно учесть равенство $I_n = P_n$:

```
program SET3;
    const n = ;
    var s: array [1..n] of 0..1;
        i, j, k, p: integer;
begin
{0} for k := 1 to n do
        s[k] := 0;
    i := 0; {i определяет число сгенерированных подмножеств}
    repeat
        for k := 1 to n do
            write(s[k]);
        writeln;
{1}     i := i + 1;
        p := 1;
        j := i;
{2}     while j mod 2 = 0 do
        begin { j*2^(p-1) = i }
            j := j div 2;
            p := p + 1
        end; {p определяет номер изменяемого разряда}
        if p <= n then
{3}         s[p] := 1-s[p]
{4} until p > n
end.
```

*Комментарий.* Пусть после выполнения оператора {1} $i$ имеет двоичное разложение $b_m \dots b_p 0 \dots 0$, где $b_p = 1$ (или до выполнения оператора {1} значение $i$ в двоичной системе выглядело как $b_m \dots b_{p+1} 01 \dots 1$ (сравните с программой SET1)). Для определения $p$ достаточно выполнить оператор {2}. Условие {4} означает, что уже сгенерировано $2^n$ кодовых слов.

**Упражнение.** Определить вычислительную сложность программы SET3.

Улучшить временные характеристики приведенной программы можно за счет замены цикла {2} более совершенной конструкцией. Здесь возможны два подхода. Первый основан на непосредственном применении машинных команд, а второй — на использовании определенной закономерности в последовательности номеров изменяемых разрядов в текущем кодовом слове.

Перейдем к изложению первого способа. Предположим, что порядок генерируемых кодовых слов меньше чем длина машинного слова в арифметических и логических командах ЭВМ. Пусть знак $\equiv$ обозначает команду поразрядного сравнения двух слов (т.е. в результате получается новое слово, в котором единицы стоят только в тех разрядах, которые различны в заданных словах). Знак $\&$ обозначает поразрядную конъюнкцию двух слов. Применив эти команды, можно улучшить временные характеристики программы SET3. В самом деле, $s$ можно хранить в упакованном виде в одном слове. Пусть в i1 хранится значение $i$ до выполнения оператора {1}, т.е. `i1 = i - 1`. Тогда оператор {0} эквивалентен `s := 0;` оператор {2} — `p := (i≡i1) & i;` оператор {3} — `s := s ≡ p;` условие {4} записывается как $i = 2^n$.

**Пример.** Пусть длина машинного слова равна 16. После оператора {1} $i = 3984$. Тогда

машинное представление:

$$\begin{aligned} i &= 0000\ 1111\ 1001\ 0000 \\ i1 = i-1 &= 0000\ 1111\ 1000\ 1111 \\ i \equiv i1 &= 0000\ 0000\ 0001\ 1111 \\ p = (i \equiv i1)\&i &= 0000\ 0000\ 0001\ 0000 \end{aligned}$$

Случай, когда $n$ совпадает с длиной машинного слова, поддается подобной модификации, но требует учета «переполнения» слов.

**Упражнение.** Используя равенство $I_n = P_n$ показать, что если $0 \le i < n$ и $b_n b_{n-1} \dots b_0$ — двоичное представление $i$, записанное в $n+1$ позиции и $G_i = g_1 g_2 \dots g_n$ $i$-ый по порядку код Грея, генерируемый программой `SET3`, то $g_k = b_{n-k} + b_{n-k+1}$, $1 \le k \le n$. Построить программу генерации кодов Грея на основе последнего равенства.

Второй способ [4] строится на хранении в памяти последовательности значений номеров изменяемых разрядов в текущем кодовом слове, т.е. на хранении значений последовательности $P_n$. Вследствие рекурсивного определения $P_n$ естественно организовать хранение ее элементов в виде стека. В этом случае генерация ее элементов может быть описана по следующей итеративной схеме:

1. {инициализация стека} Вначале стек содержит элементы $n, n-1, \dots, 1$ (с 1 в вершине).
2. Алгоритм выталкивает верхний элемент $i$ и помещает его в последовательность.
3. В стек добавляются элементы $i-1, i-2, \dots, 1$.
4. Повторяются выполнения с шага 2 до тех пор, пока стек не опустошится.

**Упражнение.**

1. Доказать корректность приведенного алгоритма.
2. Написать программу генерации $P_n$ по этому алгоритму.
3. Оценить временную и емкостную сложность полученной программы.

Отметим, что занесение значений в стек фактически совпадает с занесением значений параметра $m$ в стек исполняемой программы при вызовах процедуры `GRAY` из `SET2`, и поэтому не дает существенного выигрыша по сравнению с алгоритмами `SET2` и `SET3`. Однако если организовать стек в виде списочной структуры особого типа, при которой действия по включению на шаге 3 элементов $i-1, i-2, \dots, 1$ в стек выполняются за постоянное число операций, не зависящих от i, то можно заметно улучшить временные характеристики алгоритма.

Пусть стек хранится в массиве $(t_0, t_1, \dots, t_n)$, при этом $t_0$ указывает на верхний элемент в стеке, и для каждого $i > 0$ $t_i$ определяет значение, расположенное в стеке под элементом $i$, если $i$ находится в стеке. Заметим, что элементы $i-1, i-2, \dots, 1$ помещаются в стек после удаления элемента $i$ за счет изменения значения $t_0$ на 1. Если $i$ нет в стеке, то значение $t_i$ может быть, вообще говоря, любым, так как не оказывает никакого влияния на вычисления. Однако его значение разумно установить равным $i+1$, т.к. по свойству алгоритма, в случае когда в следующий раз элемент $i+1$ будет помещен в стек, элементом, находящимся над ним, будет $i$. Удаление из стека элемента $i$ в этом случае может быть осуществлено за счет пересылки $t_{i-1}$ в $t_i$. Алгоритм порождения последовательности $P_n$ на языке Паскаль может быть записан так:

```
program PN;
    const n = ; n1 = ; {n1 = n + 1}
    var t: array [0..n] of 1..n1; {стек}
        p: integer;
begin
    for p := 0 to n do
        t[p] := p + 1; {инициализация стека}
    p := 0;
    while p < n1 do
    begin
        p := t[0];          {1}
        if p <> n1 then
        begin
            writeln (p);    {2}
            t[p-1] := t[p]; {3}
            t[p] := p + 1;
            if p <> 1 then
                t[0]:=1     {4}
        end
    end
end.
```

**Упражнение.**

1. Протестируйте программу `PN` при $n = 3$.
2. Докажите корректность алгоритма генерации последовательности $P_n$ для произвольного $n$.

*Замечание.* Если оператор `{1}` поместить после оператора `{2}`, то оператор `{4}` можно исключить, т.к. в этом случае неверные значения `t[0]` будут исправляться оператором `{3}`.

Учитывая последнее замечание, алгоритм генерации кодов Грея на основе порождения последовательности $P_n$ по алгоритму `PN` может быть записан так:

```
program SET4;
    const n = ; n1 = ; n2 = ; {n1 = n+1; n2 = n+2}
    var t: array [0..n1] of 1..n2;
        s: array [1..n1] of 0..1;
        i, p: integer;
begin
    for i := 1 to n1 do
    begin
        s[i]:=0;
        t[i]:=i+1 {инициализация стека}
    end;
    p := 0;
    t[0] := 1;
{1} while p < n1 do
    begin
        for i := 1 to n do
            write(s[i]);
```

```
        writeln;
        p := t[0];
        s[p] := 1-s[p];
        t[0] := 1;
        t[p-1] := t[p];
        t[p] := p+1
    end
end.
```

*Комментарий.* Добавление в массивы $t$ и $s$ $n+1$-х элементов сделано для того, чтобы сократить число проверок внутри цикла `{1}`.

## 2.3. Генерация мультимножеств

Для представления данных некоторых задач более естественными, чем множества, являются множества с повторениями элементов, или мультимножества. При записи мультимножеств с каждым элементом удобно связывать неотрицательное целое число, указывающее количество повторений этого элемента.

**Пример.** Мультимножество $(x, x, x, y, y, z, z, z, u)$ удобно записывать как $(3\cdot x, 2\cdot y, 3\cdot z, 1\cdot u)$.

Применение коэффициентов кратности удобно также для представления мультимножеств при построении алгоритмов. Пусть $x < y < z < u$, тогда множество $(3\cdot x, 2\cdot y, 3\cdot z, 1\cdot u)$ представимо как $(3, 2, 3, 1)$.

Таким образом, если для представления подмножеств некоторого базового множества из $n$ элементов были использованы $n$-последовательности нулей и единиц, то для представления мультимножеств, которые можно построить на базе этого множества, удобно использовать $n$-кортежи неотрицательных целых чисел.

**Упражнение.**

1. Пусть задано мультимножество $A = (m_1, \ldots, m_n)$, $m_i \geq 0$, $1 \leq i \leq n$. Доказать, что A имеет $\prod_{i=1}^{m}(m_i + 1)$ мультиподмножеств.

2. Пусть $A = (m, \ldots, m)$, где базовое множество содержит $n$ элементов. Написать программу генерации всех мультиподмножеств $A$ в лексикографическом порядке. Заметим, что подобная генерация фактически совпадает с выводом всех чисел от 0 до $m^n - 1$ в $m$-ричной позиционной системе.

3. Написать программу генерации всех мультиподмножеств $A$ из предыдущего упражнения, при котором каждое последующее мультиподмножество отличается от предыдущего вставкой или удалением одного элемента. Генерация начинается с представления пустого множества. (Вначале постройте рекурсивный алгоритм генерации, например, воспользовавшись последовательностью

   $$C_1 0; C_2 0; \ldots; C_P 0; C_P 1; \ldots; C_1 1; C 1_2; \ldots; C_P 2; C_P 3; \ldots, \quad \text{где} p = m^n,$$

   которая определяет генерацию мультимножеств $(n+1)$-го порядка. Затем постройте итеративный алгоритм, подобный `SET3`.)

# 3. Генерация K-подмножеств

Подмножество, содержащее $k$ элементов заданного множества $A$ из $n$ элементов ($k \leq n$), обычно называют $k$-подмножествами просто $k$-множествами. Для представления $k$-подмножеств, наряду с представлением в виде $n$-последовательностей нулей и единиц, которое обычно применяют только для хранения в упакованном виде, часто используется представление в виде $k$-кортежей натуральных чисел. При этом каждый кортеж считается упорядоченным, например, в порядке возрастания. Это связано с тем, что чаще $k \leq n/2$. (Так как $k$-множество и его дополнение однозначно связаны, и поставленную перед пользователем задачу удается переформулировать, используя либо $k$-множество, либо его дополнение.) Поэтому при хранении $k$-множеств в виде $n$-последовательностей нулей и единиц не менее половины из них являются нулями. При хранении $k$-кортежами могут, например, указываться порядковые номера элементов, входящих в $k$-множество.

**Пример.** Множеству $(0,0,0,0,1,1,0,1,1,0,0)$ соответствует кортеж $(5,6,8,9)$.

Перейдем теперь к алгоритму генерирования $k$-подмножеств в лексикографическом порядке. Прежде всего дадим определение лексикографической упорядоченности в терминах $K$-кортежей.

**Определение 5.** Множество $(a_1,\ldots,a_k)$, $a_1 < \ldots < a_k$, предшествует множеству $(b_1,\ldots,b_k)$, $b_1 < \ldots < b_k$, если существует $i$, $1 \leq i \leq k$, такое, что для любого $j$, $j < i$, $a_j = b_j$, $a_i < b_i$.

**Упражнение.** Доказать, что это определение эквивалентно определению лексикографической упорядоченности из п. 2.1.

Первым в лексикографическом порядке является кортеж $(1,\ldots,k)$, а последним — $(n-k+1,\ldots,n)$. Пусть $(a_1,\ldots,a_k)$ не совпадает с последним кортежем генерации, тогда непосредственно следующим за ним является $(b_1,\ldots,b_k) = (a_1,\ldots,a_{p-1},a_p+1,a_p+2,\ldots,a_p+k-p+1)$, где $p = \max(i|\, a_i < n-k+i)$. При этом, если $(b_1,\ldots,b_k)$ не последний, то следующий за ним выглядит так: $(b_1,\ldots,b_{r-1},b_r+1,b_r+2,\ldots,b_r+k-r+1)$, где $r = p-1$, если $b_k = n$, и $k$, если $b_k < n$.

**Пример.** Генерации $k$-подмножеств в лексикографическом порядке. $n=5$, $k=2$

1. (1,2), $p=2$
2. (1,3), $p=2$
3. (1,4), $p=2$
4. (1,5), $p=1$
5. (2,3), $p=2$
6. (2,4), $p=2$
7. (2,5), $p=1$
8. (3,4), $p=2$
9. (3,5), $p=1$
10. (4,5)

Это приводит к следующему алгоритму на языке Паскаль:

```
program SUBSET1;
    const n = ; k= ; {k ≤ n}
    var s: array[1..k] of 1..n;
        i, p: integer;
begin
    for i := 1 to k do
{1}     s[i] := i; {задание первого K-множества}
    p := k;
    while p >= 1 do
    begin
        for i := 1 to k do {вывод очередного K-множества}
            write(s[i]);
        writeln;
        if s[k] = n then
```

```
            p := p-1
        else
            p := k;
        if p >= 1 then
        begin
            s[p] := s[p]+1;
            for i := p+1 to k do
{3}             s[i] := s[i-1] + 1
        end
    end
end.
```

Оценим вычислительную сложность программы. Заметим, что оператор строки `{1}` исполняется $k$ раз; строки `{2}` — $C_k^n$; а строки `{3}` — $C_k^n - 1 + C_{k-1}^{n-1} - 1 + \ldots + C_1^{n-k+1} - 1$ раз, равное $C_k^{n+1} - k - 1$. То есть, вычислительная сложность программы $O(C_k^{n+1})$. Учитывая, что $C_k^{n+1} = C_k^n \cdot \dfrac{n+1}{n-k+1}$ и $C_{k-1}^n \cdot \dfrac{k}{n-k+1}$ получаем, что алгоритм линеен, за исключением случаев, когда $k = n - o(n)$. В этом случае алгоритм можно применить для порождения $(n-k)$-подмножеств, с последующим переходом к их дополнению.

**Упражнение.** Написать вариант программы `SUBSET1` для $k$-подмножеств, представленных $n$-последовательностями нулей и единиц. Оценить вычислительную сложность построенной программы.

## 3.1. Генерация k-подмножеств заменой одного элемента

Для того чтобы убедиться в том, что подобные генерации существуют, рассмотрим некоторые свойства симметрично-отраженных кодов Грея.

**Определение 6.** Пусть $\alpha$ — некоторая цепочка над алфавитом $X$. Тогда $\alpha^m$, где $m$ — целое неотрицательное число, обозначает конкатенацию $m$ раз цепочки $\alpha$.

**Пример.** $X = \{a, b, c\}$ $(ab)^3 = ababab$ $ab^3 = abbb$.

В дальнейшем симметрично-отраженный код Грея $n$-го порядка, генерируемый программой `SET2`, будем обозначать $G(n)$. Естественно, что любое кодовое слово из $G(n)$ является цепочкой длины $n$ над $\{0, 1\}$. Последовательности кодовых слов из $G(n)$, содержащие $k$ единиц и упорядоченные соответственно с их генерацией программой `SET2`, будем обозначать $G(n, k)$. Последовательности, в которых кодовые слова располагаются в обратном порядке по отношению к $G(n)$ или $G(n, k)$, будем обозначать $GR(n)$ и $GR(n, k)$ соответственно.

***Теорема* 1.** *Первым кодовым словом в $G(n)$ с ровно $k$ единицами будет $1^k0^{n-k}$, а последним — $1^{k-1}0^{n-k}1$ и $0^n$, если $k = 0$.*

*Доказательство.* Доказательство проводится индукцией по $n$, при $n = 1$ — очевидно. Пусть теорема справедлива для $i \le n$ и любых $k$, покажем, что теорема справедлива для $n+1$ и всех $k$. В терминах $G(n)$ соотношение (2) из п. 2.2 может быть записано как $G(n+1) = G(n)0; GR(n)1$, поэтому первое кодовое слово с $k \ge 0$ единицами (кроме $k = n+1$) есть первое кодовое слово в $G(n)$ с приписанным справа нулем. Иными словами, $1^k0^{n+1-k}$, как и требовалось. Аналогично последнее кодовое слово с $k \ge 1$ единицами в $G(n)$ с приписанной справа единицей — это $1^{k-1}0^{n-k+1}1$, как и требовалось. В случае $k = n+1$ единственным словом в $G(n+1)$, содержащим $n+1$ единицу, является $1^{n+1}$. □

***Теорема* 2.** *Соседние кодовые слова в $G(n, k)$ различаются ровно в двух разрядах.*

*Доказательство.* Доказательство проводится индукцией по $n$. Теорема, очевидно, справедлива для $n = 1$, поскольку $k = 0$, либо $k = 1$. Предположим, что теорема верна для $n$ и всех $k$, $0 \le k \le n$. Покажем, что она верна для $n+1$ и любого $k$, $0 \le k \le n+1$. Если $k = 0$ или $k = n+1$, она справедлива, так как $G(n+1)$ содержит только одно соответствующее кодовое слово. Для $1 \le k \le n$ теорема выполняется по индукционному предположению, если кодовые слова оба либо в $G(n)0$ либо $GR(n)1$. Таким образом, требуется только рассмотреть, на сколько разрядов отличается последнее кодовое слово с $k$ единицами в $G(n)0$ от первого кодового слова с $k$ единицами в $GR(n)1$. Из предыдущей теоремы следует, что эти слова имеют вид: при $k = 1$ $0^{n-1}10$ и $0^n1$, при $k > 1$ $1^{k-2}10^{n-k}10$ и $1^{k-2}00^{n-k}11$ соответственно. □

***Следствие.*** *Последовательность кодовых слов $G(n,k)$, $n \le k \le 0$, можно определить следующей рекурсией:*

$$G(n,k) = G(n-1,k)0;\ GR(n-1,k-1)1\ u\, G(n,0) = 0^n, G(n,n) = 1^n \qquad (3)$$

В самом деле, это определение действительно задает последовательность, совпадающую с той, которая получается удалением из $G(n)$ всех кодовых слов с числом единиц, не равным $k$. Рекурсивное определение (3) приводит к следующей программе генерации кодовых слов $G(n,k)$ на языке Паскаль.

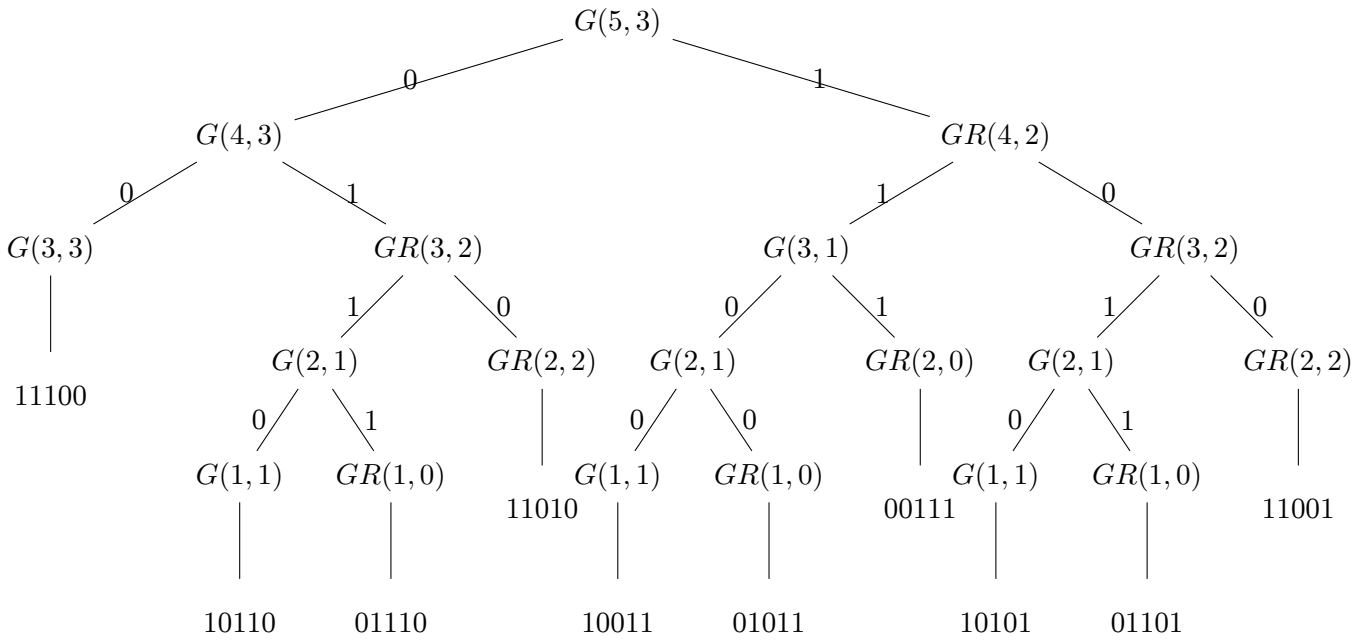
```
program subset2;
    const n = ; k= ; {0<k<=n}
    var i: integer;
        s: array[1..n] of 0..1;
procedure print(m, h: integer);
begin
    if h <> 0 then
        h := 1;
    for i := 1 to m do
        s[i] := h;
    for i := 1 to n do
        write(s[i]);
    writeln
end;
procedure GR(m, h: integer); forward;
procedure G(m, h: integer);
begin
    if (h = 0) or (m = h) then
        print(m, h)
    else
    begin
        s[m] := 0;
        G(m-1, h);
        s[m] := 1;
        GR(m-1, h-1)
    end
end;
procedure GR(m, h: integer);
begin
    if (h=0) or (m=h) then
        print(m, h)
    else
    begin
```

```
        s[m] := 1;
        G(m-1, h-1);
        s[m] := 0;
        GR(m-1, h)
    end
end;
begin {SUBSET2}
    G(n,k)
end.
```

*Комментарий.* Процедура `GR(m,h)` соответствует $GR(m,h)$. Процедура `print` служит для вывода генерируемых кодовых слов.

Для дальнейшего анализа алгоритма остановимся подробнее на дереве вызова процедур `G` и `GR`.

**Пример.** $n = 5$, $k = 3$.




Из текста программы видно, что рекурсивная структура $G(n,k)$ представляется в виде бинарного дерева. Каждую дугу, ведущую в $G(m-1,h)$ или $GR(m-1,h)$, пометим значением, которое присваивается $s[m]$ перед вызовом этой процедуры. Обозначать мы будем соответственно $G(m-1,h)s[m]$ или $GR(m-1,h)s[m]$. Каждое кодовое слово может быть составлено из значения, вырабатываемого листом, и пометок дуг, составляющих ветвь от этого листа к корню дерева. Очевидно, что два соседних кодовых слова в $G(n,k)$ могут отличаться только в тех разрядах, которые соответствуют несовпадающим частях их ветвей. Каждая вершина бинарного дерева характеризуется ее уровнем-значением $m$. При этом является однозначноопределенной часть текущего кодового слова — позиции $s[m],\ldots,s[n]$. Кроме того, число $h$ определяет количество единиц, находящиеся в позициях $s[1],\ldots,s[m-1]$.

**Упражнение.**

1. Доказать, что листья дерева могут быть помечены ($m \neq 0$) $G(m,0)1$, $GR(m,0)1$ или $G(m,m)0$, $GR(m,m)0$, но не могут с $G(m,0)0$, $GR(m,0)0$ или с $G(m,m)1$, $GR(m,m)1$.

2. Определить число вызовов процедур `G` и `GR` как функцию от $n$ и $k$. (*Указание.* Определить число кодовых слов, начало которых состоит из $1, 2, \ldots, k$ единиц, и число слов, начало которых состоит из $1, 2, \ldots, n-k$ нулей.)

3. Определить вычислительную сложность программы `SUBSET2`.

4. Доказать корректность приведенной программы.

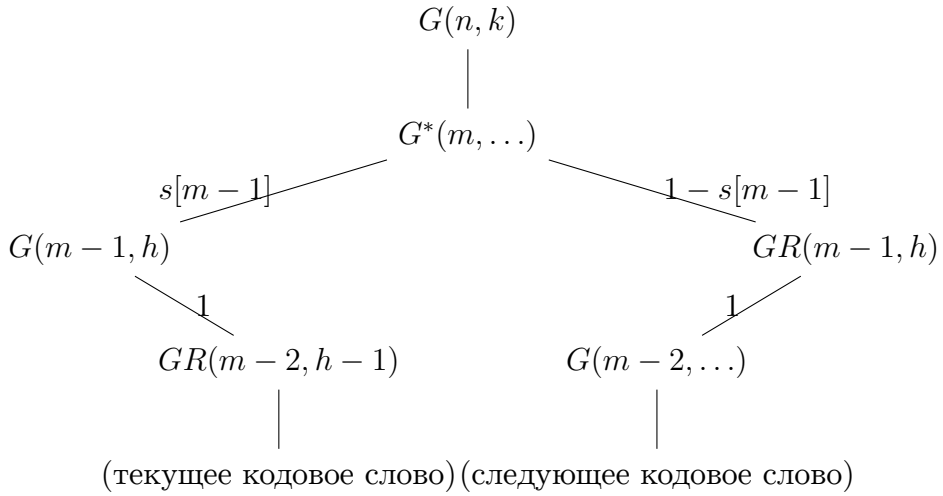
Перейдем к построению итеративного алгоритма генерирования $G(n, k)$. Такой алгоритм должен итеративно моделировать левосторонний [2] обход дерева вызова процедур `G` и `GR`. В этом случае моделирование осуществляется за счет перехода от текущего кодового слова к следующему за счет информации о еще не полностью обработанных узлов дерева вызова, лежащих на ветви от листа, соответствующему текущему кодовому слову, до корня дерева — вершины $G(n, k)$. Естественно, информация о таких узлах хранится в стеке обхода дерева. Рассмотрим какую информацию необходимо хранить о каждом из таких узлов в стеке. Прежде всего заметим, что по свойству программы `SUBSET2`, для каждого узла дерева обхода левой его ветви может соответствовать только вызов процедуры `G`, а правой — вызов процедуры `GR`. Поэтому еще не полностью обработанными являются узлы, соответствующие вызову процедуры `G`. Для каждого такого узла в стеке достаточно хранить значение его уровня — $m$. Так как мы знаем, что последующая обработка этого узла будет связана с преобразованием части кодового слова — разряды $s[1], \ldots, s[m]$, при этом значение $h$ легко вычисляется как число единиц, расположенных в разрядах $s[1], \ldots, s[m-1]$.

Пусть $m$ — значение, лежащее на вершине стека обхода дерева вызова. Тогда это значение соответствует некоторому вызову процедуры `G(m-1,h)`, который в свою очередь однозначно определяет узел дерева обхода, лежащий на пути из листа, соответствующего текущему кодовому слову в корень дерева, при этом узел $G(m-1, h)$ является первым узлом типа $G$, лежащим на этом пути. Непосредственного предка узла $G(m-1, h)$ обозначим $G^*(m, \ldots)$. Узел $G^*(m, \ldots)$ может соответствовать вызову как процедуры `G`, так и процедуры `GR`, при этом для процедуры `G` $s[m]$ в текущем кодовом слове равно 0, для процедуры `GR` — 1. Пусть узел $GR(m-1, \ldots)$ непосредственный правый потомок вершины $G^*(m, \ldots)$. Тогда путь из корня дерева обхода к листу, соответствующему следующему кодовому слову за текущим в $G(n, k)$, может быть описан как

$$G(n, k), \ldots, G^*(m, \ldots), GR(m-1, \ldots), G(m-2, \ldots), \ldots, G(p, \ldots),$$

где $p$ — уровень узла следующего кодового слова. Т.е. путь из корня дерева обхода к листу, соответствующему следующему кодовому слову, совпадает на участке $G(n, k), \ldots, G^*(m, \ldots)$ с путем к текущему кодовому слову, затем осуществляется переход по правой ветви узла $G^*(m, \ldots)$, после чего путь проходит только по левым ветвям узлов дерева.

[2] Обработка информации, хранящейся в узлах бинарного дерева, по левостороннему обходу характеризуется тем, при обработки каждой вершины вначале совершается обход левой ветви, выходящей из этой вершины; затем обработка информация хранящейся непосредственно в этой вершине; после этого, обход правой ветви, исходящей из вершины.




Таким образом, для перехода к следующему кодовому слову мы должны преобразовать текущее кодовое слово в разрядах $1,\ldots,m$, а также занести в стек значения уровней вызова процедуры `G` — $m-2,\ldots,p+1$. В преобразовании кодового слова одним из изменяемых разрядов является $s[m]$, а другой может быть вычислен на основе теоремы 1 по следующей схеме:

**случай 1.** $s[m]=0$, $h>1$, это означает, что первые $(m-1)$ разрядов текущего кодового слова являются последним кодовым словом в $G(m-1,h)$, а начало следующего кодового слова представляет собой первое кодовое слово в $GR(m-1,h-1)$, т.е.

$$1^{h-2}10^{m-h-1}10\ldots \text{переходит в} 1^{h-2}00^{m-h-1}11\ldots$$

если $s[m]=0$, $h=1$, тогда

$$0^{m-2}10\ldots \text{переходит в} 0^{m-2}01\ldots$$

**случай 2.** $s[m]=1$, $h>0$, это означает, что первые $(m-1)$ разрядов текущего кодового слова являются последним кодовым словом в $G(m-1,h)$, а начало следующего кодового слова представляет собой первое кодовое слово в $GR(m-1,h)$, т е.

$$1^{h-1}00^{m-h-2}11\ldots \text{переходит в} 1^{h-1}10^{m-h-2}10\ldots$$

если $s[m]=1$, $h=0$, тогда

$$0^{m-2}01\ldots \text{переходит в} 0^{m-2}10\ldots$$

Таким образом, вторым изменяемым разрядом является

$$\begin{array}{ll} s[h-1], & \text{если } s[m]=0 \text{ и } h>1 \\ s[m-1], & \text{если } s[m]=0 \text{ и } h=1 \\ s[h], & \text{если } s[m]=1 \text{ и } h>1 \\ s[m-1], & \text{если } s[m]=1 \text{ и } h=0 \end{array}$$

Нам осталось установить только значение параметра $p$, от которого зависит число новых узлов дерева с не пройденными правыми ветвями.

Если $h=0$ или $h=m-1$, то в стек никакие новые узлы не добавляются, поскольку узел $G(m-1,h)$ является листом, однако для последующих преобразований $h$ должно быть

изменено на $h+1$. С другой стороны, в случае $h \neq 0$ и $h \neq m-1$ мы должны включить в стек узлы $m-1, m-2, \ldots, h+1$, за исключением того случая, когда $s[m-2] = \ldots = s[1] = 0$; в этом же случае в стек включается только $m-1$. Значение $h$ должно быть вычислено заново, и, так как среди $s[m-1], s[m-2], \ldots, s[h+1]$ единицей может быть только $s[m-1]$, мы можем $h$ вычислить как $h-s[m-1]$. Если при этом $h$ становиться равным 0, мы должны иметь $s[m-2] = \ldots = s[1] = 0$.

На основе этих рассуждений строится следующая программа:

```
program subset3;
    const n = ; k = ; n1 = ; n2 = ; {n1=n+1, n2=n+2}
    var i, m, h: integer;
    s: array[1..n1] of 0..1; {генерируемое кодовое слово}
    t: array[1..n1] of 1..n2; { стек узлов}
begin
    for i := 1 to k do
    begin
        s[i] := 1;
        t[i] := i+1
    end;
    for i := k + 1 to n1 do
    begin
        s[i] := 0;
        t[i] := i+1
    end;
    h := k;
    t[1] := k+1; {t[1] указывает на вершину стека}
    m := 0;
    while m <> n1 do
    begin
        for i := 1 to n do
            write(s[i]);
        writeln; {вывод очередного кодового слова}
        m := t[1];
        t[1] := t[m];
        t[m] := m+1; {выбор сына узла ветвления}
        if s[m] = 1 then
        begin
            if h <> 0 then
                s[h] := 1-s[h]
            else
                s[m-1] := 1-s[m-1];
            h := h+1
        end
        else
        begin
            if h <> 1 then
                s[h-1] := 1-s[h-1]
            else
                s[m-1] := 1-s[m-1];
            h := h-1
        end;
        s[m] := 1-s[m]; {конец корректировки кодового слова}
        if (h = m-1) or (h = 0) then
```

```
            h := h+1
        else
        begin
            h := h - s[m-1]; {корректировка h}
{1}         t[m-1] := t[1];
            if h = 0 then
                t[1] := m-1
            else
{2}             t[1] := h+1 {корректировка стека}
        end
    end
end.
```

*Комментарий.* Стек реализуется таким же способом, как в `SET4`. `t[1]` указывает на узел в вершине стека, и каждый элемент `t[j]` в стеке немедленно получает новое значение, как только он исключается из стека. Присваивания `{1}` и `{2}` служат для добавления в стек $m-1, \ldots, h+1$.

**Упражнение.**

1. Доказать корректность программы `SUBSET3`.
2. Откорректировать программу так, чтобы она правильно функционировала при $k = 0$.

# Литература


1. Ю. В. Матисевич. Десятая проблема Гильберта. М.: Физматлит, 1993
2. В. Липский. Комбинаторика для программистов. Мир, М. 1988.
3. Х Минк. Перманенты. Мир, М. 1982.
4. Э. Рейнгольд, Ю. Нивергельт, Н. Део. Комбинаторные алгоритмы. Теория и практика. Мир. М. 1980.


# 4. Процедурные типы

В Turbo Pascal процедуры и функции можно рассматривать как некоторые параметры и можно использовать переменные, принимающие значение процедуры или функции. С этой целью вводятся процедурные типы, которые указывают, какой вид подпрограммы (процедуру или функцию) можно использовать в качестве параметра и с какими параметрами должны быть эти подпрограммы.

Объявление процедурного типа похоже на заголовок подпрограммы: пишется слово `procedure` или `function`, за которым в круглых скобках записывается список формальных параметров; для функции после списка формальных параметров через двоеточие указывается тип функции. В отличие от заголовка подпрограммы, здесь не указывается имя подпрограммы.

**Пример.**

```
type
    proc1 = procedure;
            {процедура без параметров}
    proc2 = procedure (var x, y: integer);
            {процедура с двумя параметрами-переменными типа integer}
    func1 = function (x: real): real;
            {функция типа real с одним параметром типа real}
```

Далее можно ввести переменные этих типов:

```
var
    p1: proc1;
    p2: proc2;
    f1: func1;
```

После этого процедурным переменным можно присваивать значения конкретных процедур и функций. Как и в во всех других случаях, процедурная переменная и подпрограмма должны быть совместимы для присваивания.

Существует ряд правил использования подпрограмм в качестве процедурной переменной:

- они должны компилироваться с ключом компилятора `{$F+}` или иметь директиву `far` для получения полного адреса подпрограмм
- они не должны быть стандартными процедурами и функциями
- они не должны объявляться внутри других процедур и функций
- они не должны быть типа `inline` или `interrupt`.

**Пример.**

```
{$f+}
procedure swap(var a, b: integer);
    var t: integer;
begin
    t := a;
    a := b;
    b:= t
end;
function tan(angle: real): real
begin
    tan := sin(angle)/cos( angle)
    {проверка, что cos( angle)<>0, осуществляется в самой программе}
end;
{$f-}
```

В этом случае процедурным переменным, введенным ранее, можно присвоить значения:

```
p2 := swap;
f1 := tan;
```

а вызовы `p2(i,j)` и `f1(x)` эквивалентны соответственно `swap(i,j)` и `tan(x)`.

Так же как и указатель, процедурная переменная занимает 4 байта памяти, в которых помещается полный адрес подпрограммы (поэтому подпрограммы необходимо компилировать с ключом `{$f-}`).

Процедурные переменные можно использовать так же, как и переменные других типов: в выражениях (если эта переменная — функция), в виде оператора (если эта переменная — процедура), как компонента другой более сложной переменной, как передаваемый в подпрограмму параметр. Идея единства данных и подпрограмм получила дальнейшее развитие в объектно-ориентированном программировании.

**Пример.** Вычисление интеграла по формуле Симпсона.

Для аппроксимации интеграла

$$s = \int_a^b f(x)dx$$

используется сумма конечного числа узловых значений $f_i$:

$$s_k = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \ldots + 4f_{n-3} + 2f_{n-2} + 4f_{n-1} + f_n),$$

где $f_i = f(a + i \cdot h)$, $h = (b-a)/n$ и $n = 2^k$. Число узловых точек равно $n+1$, а $h$ — расстояние между двумя соседними узловыми точками. Значение интеграла $s$ аппроксимируется последовательностью $s_1, s_2, \ldots$, которая сходится, если функция ведет себя достаточно хорошо (гладкая) и если арифметические операции выполняются точно.

На каждом шаге число узловых точек удваивается. Конечно, хорошая программа не будет вычислять $f(x)$ $2^k$ раз на каждом $k$-м шаге, поскольку можно использовать значения $f_i$, полученные на предыдущих шагах. В сумме $s_k$ три члена

$$s_k = s_k^{(1)} + s_k^{(2)} + s_k^{(4)},$$

которые обозначают суммы в узловых точках с весами 1, 2, 4. Их можно определить с помощью рекуррентных соотношений для $k > 1$:

$$\begin{aligned}
s_k^{(1)} &= \frac{1}{2}s_{k-1}^{(2)} \\
s_k^{(2)} &= \frac{1}{2}s_{k-1}^{(2)} + \frac{1}{4}s_{k-1}^{(4)} \\
s_k^{(4)} &= \frac{4h}{3}(f(a+h) + f(a+3h) + \ldots + f(a+(n-1)h))
\end{aligned}$$

и начальных значений:

$$\begin{aligned}
s_1^{(1)} &= \frac{h}{3}(f(a) + f(b)) \\
s_1^{(2)} &= 0 \\
s_1^{(4)} &= \frac{4h}{3}f(\frac{a+b}{2})
\end{aligned}$$

```
program integral;
    const pi = 3.141592;
    type func1 = function (x: real): real;
    var ep: real;
{$f+}
```

```
function bil(x: real): real;
    var a1, a2: real;
        i:integer;
begin
    a1: = sin(10*x);
    a2: = sin(x);
    if a2 = 0 then
        a1: = 1
    else
        a1 := a1/a2;
    bil := a1*a1*a1*a1*a1*a1;
end;
function bi(x: real):real;
begin
    bi := sin(x)
end;
{$f-}
function simpson(a, b: real; f: func1): real;
    var i ,n: integer;
        s, ss, s1, s2, s4, h: real;
begin
    n := 2;
    h := (b-a)*0.5;
    s1 := h*(f(a)+f(b));
    s2 := 0;
    s4 := 4*h*f(a+h);
    s := s1 + s2 + s4;
    repeat
        ss := s;
        n := 2*n;
        h := h/2;
        s1 := 0.5*s1;
        s2 := 0.5*s2 + 0.25*s4;
        s4 := 0;
        i := 1;
        repeat
            s4 := s4 + f(a+i*h);
            i := i+2
        until i>n;
        s4 := 4*h*s4;
        s := s1+s2+s4
    until abs(s-ss)<ep;
    simpson := s/3
end; {simpson}
begin
    ep := 0.001;
    writeln((1/(2*pi))*simpson(0,2*pi,bil));
    writeln(simpson(0,pi/2,bi))
end.
```

# 5. Описание и обработка последовательных файлов

*Файлом* в языке Паскаль называется объект определенного типа — типа файла. Файл состоит из последовательности компонент одного и того же типа и может в разное время содержать различное число компонент (возможно, и ни одной).

Тип компонент файла может быть любым, но *не должен содержать внутри себя других файлов или указателей.* (Последнее ограничение связано с тем, что файлы хранятся, как правило, на внешних устройствах, при этом указатели, которые указывают на объекты в основной памяти, теряют смысл после окончания работы программы, в то время как файл может быть сохранен и дольше.)

Определение типа файла выглядит так

```
file of T
```

где `T` — определение типа компонент файла.

С помощью этого определения можно описывать переменные, значения которых будут файлы.

С особенностью размещения файлов связано еще одно ограничение, накладываемое на работу с переменной типа файла: *нельзя присваивать такой переменной новое значение с помощью оператора присваивания и передавать значение такой переменной в процедуру или функцию.* Это ограничение не относится к передаче параметров-переменных в процедуру.

Изменение значений переменных типа файла происходит путем добавления или удаления компонент файла с помощью стандартных процедур обработки файлов.

Файлы могут быть внешними и внутренними (локальными). Локальные айлы создаются при входе в ту подпрограмм, где они описаны, и уничтожаются ри выходе из нее. Такие файлы могут физически располагаться как во внешней, так и в оперативной памяти.

**Пример.** Заголовок программы с описанием файлов.

```
program p (output, f1, f2);
    type элемент = record
                       ключ: integer;
                       строка: string
                   end;
    var f, f1, f2: file of элемент;
```

`f1`, `f2` — внешние файлы, `f` — локальный. `output` — стандартный для ввода.

*Замечание.* Поскольку работа любой программы состоит в конечном итоге в том, что изменяется содержимое файлов, являющихся параметрами программы, то свойство любой программы можно выразить, описав, какие преобразования над файлами она выполняет.

Например, НОД$(a, b)$. Вначале input $=< a, b >$, $b \neq 0$. После output $=< g >$, где $g =$ НОД$(a, b)$.

В Турбо Паскале существуют три класса файлов: типизированные, текстовые и нетипизированные. Для типизированных указан тип компонент файла; в текстовых компоненты имеют стандартный тип `TEXT`; в нетипизированных компоненты представляют собой конечные совокупности символов или байтов, при этом представление `char` или `byte` не играют никакой роли, а важно лишь с точки зрения объема занимаемых данных.

Файловая система Турбо Паскаля ориентирована на возможности операционной системы DOS. Каждому файлу в языке ставится в соответствие файловая переменная определенного типа. Поэтому перед началом работы с файлом необходимо установить соответствие. Для этого в языке используется процедура:

```
assign(f, name: string),
```

где `f` — переменная любого файлового типа, а строковое выражение `name` содержит полное имя файла, удовлетворяющее требованием операционной системы.

**Пример.**

```
assign(f1,'c:\kostin\tree.text')
```

Чтобы начать работу с файлом, необходимо его открыть. Работа с неоткрытым файлом считается ошибкой. Открытие осуществляется двумя способами в зависимости от характера будущей работы с файлом. Если файл открывается для чтения, его содержимое изменяться не может. Если файл открывается для записи, то это означает, что файл будет создаваться заново, при этом содержащаяся в нем информация ранее уничтожается, а новые компоненты присоединяются к концу файла, начиная с первой.

Открытие файла осуществляется стандартными процедурами: `reset(f)` — для чтения; `rewrite(f)` — для записи.

В дальнейшем в любой момент времени файл можно «переоткрывать», т.е. файл, первоначально открытый для чтения, может быть в последствие открыт для записи, и наоборот.

## 5.1. Организация ввода-вывода. Стандартные процедуры и функции для всех файлов

### Процедуры

`Assign (F, Name)` — связь файловой переменной с внешним файлом, имеющим имя `Name`. `Name` — переменная или константа типа `string` (или совместимого для присваивания с ним типа) или типа `PChar`. Имя типа должно быть написано в соответствии с правилами MS DOS, может включать путь и не должно превышать 79 символов. Если строка имени пустая, осуществляется связь со стандартными файлами ввода или вывода.

`Reset(F[,Size])` — открытие существующего файла. Открывается существующий файл, с которым связана файловая переменная `F`, и указатель текущей компоненты файла настраивается на начало файла. Необязательный параметр целого типа `Size` используется только с файлами без типа и задает размер пересылаемого элемента информации в байтах. По умолчанию этот размер равен 128.

`Rewrite(F[,Size])` — открытие нового файла. Открывается новый пустой файл, и ему присваивается имя, заданное процедурой `Assign`. Если файл с таким именем уже существует, то он уничтожается. Необязательный параметр `Size` имеет тот же смысл, что и в процедуре `Reset`.

`Close(F)` — закрытие внешнего файла. Закрывает внешний файл, с которым связана файловая переменная `F`. При этом в случае необходимости в содержимое файла вносятся все произведенные изменения.

### Функции

`Eof(F)` — конец файла. Принимает значение `true`, если указатель текущей компоненты находится за последней компонентой файла (за последним символом, если файл текстовой), и `false` — в противном случае.

`IOResult` — результат последней операции ввода-вывода. Возвращает значение 0, если операция ввода-вывода завершилось успешно, и другое число — в противном случае. После применения этой функции параметр состояния последней операции ввода-вывода сбрасывается в 0.

## 5.2. Стандартные процедуры и функции для текстовых файлов

Текстовой файл представляет собой совокупность символов, разделенных на строки, причем в конце каждой строки стоит признак конца строки.

Особенностью работы с текстовыми файлами является то, что значения которые вводятся и выводятся с помощью процедур `read` или `write`, могут быть не обязательно типа `Char` или `String`, а также и других стандартных простых типов. Эти процедуры могут также работать и с ASCIIZ-строками.

Имеется две стандартные файловые переменные для текстового файла: `input` (для ввода с клавиатуры) и `output` (для вывода на экран дисплея).

Файл типа `Text` может быть открыт либо для чтения процедурой `reset`, либо для записи процедурой `rewrite` или `append`.

### Процедуры

`Append(f)` — открытие файла для добавления в конец информации. Открывается существующий файл, с которым связана файловая переменная `F`, и указатель текущей компоненты файла настраивается на конец файла.

`Read(F,<список ввода)` — чтение информации из файла. Из файла, с которым связана файловая переменная `F`, читаются значения для одной или нескольких переменных списка ввода.

`ReadLn(F,<список ввода)` — чтение информации из файла. То же, что и процедура `read`, но непрочитанная часть строки, включая признак конца строки, пропускается.

`Write(F,<список вывода>)` — запись информации в файл. В файл, с которым связана файловая переменная `F`, записываются значение выражений списка вывода.

`WriteLn(F,<список вывода>)` — запись информации в файл. То же, что и процедура `write`, но выводимая информация завешается признаком конца строки.

### Функции

`Eoln(F)` — конец строки файла. Принимает значение `true`, если текущей компонентой файла является признак конца строки или функция `eof(F)` принимает значение `true`. В остальных случаях функция принимает значение `false`.

## 5.3. Стандартные процедуры и функции для типизированных файлов

### Процедуры

`Read(F,<список ввода)` — чтение информации из файла. То же, что и процедура `read` для текстовых файлов, но переменные, в которые читается информация, должны быть того же типа, что и компонента файла.

`Write(F,<список вывода>)` — запись информации в файл. То же, что и процедура `write` для текстовых файлов, но список вывода представляет собой переменные того же типа, что и компонента файла.

`Seek(F,Num)` — настройка на требуемую компоненту файла. Осуществляется настройка на компоненту файла, с которым связана файловая переменная `F`. Компонента файла определяется номером `Num`, причем нумерация начинается с нуля.

`Truncate(F)` — удаление части файла, начиная с текущей позиции. Удаляется часть файла, начиная с текущей позиции и до его конца.

**Функции**

`FilePos(F)` — номер текущей компоненты файла. Функция возвращает номер текущей компоненты файла, с которым связана файловая переменная `F`. Нумерация начинается с нуля.

`FileSize(F)` — текущий размер файла. Функция возвращает текущий размер файла, с которым связана файловая переменная `F`, в компонентах этого файла.

**Задача.** Задан файл, каждая компонента которого состоит из целого числа (ключа) и 80-ти символов текста (данных). Требуется переписать содержимое этого файла в другой файл, изменив содержимое одной его компоненты с заданным ключом. Ключ и новое содержимое компоненты задаются в качестве исходных данных во входном потоке. Если компоненты с заданным ключом нет в исходном файле, то новая компонента добавляется в конец файла. Если в исходном файле есть несколько компонент с заданным ключом, то обновляется первая из таких компонент.

**Решение:** Считаем `input` — содержит исходный данные; `output` — для вывода сообщений о ходе работы программы; `a` — исходный файл; `b` — формируемый файл.

Работа программы будет состоять из двух фаз. На первой фазе содержимое файла `a` переписывается в файл `b`, пока не встретится компонента с заданным ключом. На второй фазе остаток файла `a` дописывается в файл `b`. Процесс заканчивается, когда будет исчерпано содержимое файла `a`.

Логическая переменная `naid` в программе имеет значение `true` только после того как ключ с заданным значением обнаружен.

```
program ff1;
    type rec = record
                   kl: integer;
                   ct: string[80]
               end;
    var r, nov: rec;
        a, b: file of rec;
        i: integer;
        naid: boolean;
begin
    with nov do
    begin
        readln(kl);
        readln(ct)
    end;
    assign(a, 'c:\pas\pasv\f1.txt');
    assign(b, 'c:\pas\pasv\f2.txt');
    reset(a);
    rewrite(b);
    naid := false;
    while not(naid or eof(a)) do
    begin
        read(a, r);
        if nov.kl = r.kl then
            naid:=true
        else
            write(b,r)
    end;
```

```
    write(b, nov);
    while not eof(a) do
    begin
        read(a, r);
        write(b, r)
    end;
    writeln('перепись завершена');
    close(a);
    close(b)
end.
```

Файл a можно создать следующей программой:

```
program ff2;
    type rec = record
                   kl: integer;
                   ct: string[80]
               end;
    var r: rec;
        a: file of rec;
        i: integer;
begin
    assign(a, 'c:\pas\pasv\f1.txt');
    rewrite(a);
    for i := 1 to 5 do
        with r do
        begin
            readln(kl);
            readln(ct);
            write(a, r)
        end;
    close(a)
end.
```

**Упражнение.** Рассмотреть задачу упорядочивания по ключам компонент файла (метод слияния).

# 6. Линейные списки

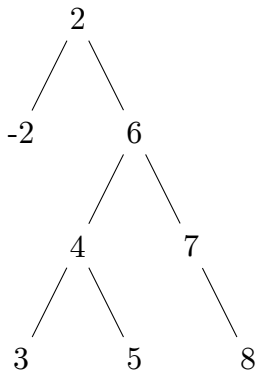
```
program ref(input, output);
    type YK = ^EL;
         EL = record
                  VAL: integer;
                  REF: YK
              end;
    var X: YK;
procedure SN1(var Z: YK);
    var A: integer;
        Y: YK;
begin
    Z := nil;
    read(A);
```

```
    while A<>0 do
    begin
        new(Y);
        Y^.VAL := A;
        Y^.REF := Z;
        Z := Y;
        read (A);
    end
end;
procedure S1N(var Z: YK);
    var A: integer;
    Y: YK;
begin
    Z := nil;
    read(A);
    if A<>0 then
    begin
        new(Y);
        Z := Y;
        Y^.VAL := A;
        read(A);
        while A<>0 do
        begin
            new(Y^.REF);
            Y := Y^.REF;
            Y^.VAL := A;
            read(A);
        end;
        Y^.REF := nil;
    end
end {S1N};
procedure PS(X: YK);
begin
    writeln;
    while X<>nil do
    begin
        write(X^.VAL, ' ');
        X := X^.REF
    end;
    writeln
end;
procedure S1NR(var X: YK);
    var A: integer;
begin
    read(A);
    if A<>0 then
    begin
        new(X);
        X^.VAL := A;
        S1NR(X^.REF)
    end
    else
```

```
        X := nil
end;
procedure YS(var X: YK);
    var A: integer;
        Z, Y, U: YK;
        T: boolean;
begin
    read(A);
    if A<>0 then
    begin
        new (X);
        X^.REF := nil;
        X^.VAL := A;
        read(A)
    end;
    while A<>0 do
    begin
        new(Y);
        Y^.VAL := A;
        if X^.VAL>A then
        begin
            Y^.REF := X;
            X := Y
        end
        else
        begin
            Z := X;
            U := Z;
            T := true;
            while (Z<>nil) and T do
            begin
                T := Z^.VAL<A;
                U := Z;
                Z := Z^.REF
            end;
            Y^.REF := Z;
            U^.REF := Y;
        end;
        read(A)
    end
end;
begin {MAIN PROGRAM}
    YS(X);
    PS(X);
    { S1NR(X);
    PS(X)}
end.
```

# 7. Дерево поиска (сортировки)

Пусть задан массив чисел: 2 6 4 3 7 -2 8 5, его дерево поиска выглядит так:




Оно строится следующим образом:

Последовательно для каждого значения элемента массива двигаемся по дереву, начиная с корня, до точки роста, к которой привешиваем этот элемент. При этом двигаемся по левой ветви, если значение подвешиваемого элемента меньше значения помещенного в этом узле элемента и по правой в противном случае. На языке Паскаль этот алгоритм может реализован процедурой pd, приведенной ниже. В этой реализации мы считаем, что признаком конца массива, расположенного во входном потоке, является появление нулевого значения.

```
program dersort;
    type
        ref = ^yz;
        yz = record
                i: integer;
                l, r: ref
             end;
    var x: ref;
procedure pd(var x: ref);
    var z, y, y1: ref;
        a: integer;
begin
    read(a);
    if a<>0 then
    begin
        new(x);
        x^.i := a;
        x^.l := nil;
        x^.r := nil;
        read(a);
        while a<>0 do
        begin
            y:=x;
            new(z);
            z^.i := a;
            z^.l := nil;
            z^.r := nil;
            while y<>nil do
            begin
                y1 := y;
                if a < y^.i then
                    y := y^.l
                else
```

```
                    y := y^.r
                end;
                if a < y1^.i then
                    y1^.l := z
                else
                    y1^.r := z;
                read(a)
            end
        end
    end;
procedure cod(var x: ref);
begin
    if x<>nil then
    begin
        cod(x^.l);
        write(x^.i:3);
        cod(x^.r)
    end
end;
begin
    pd(x);
    cod(x)
end.
```

Построенное дерево обладает следующим интересным свойством:

При симметричном обходе дерева поиска элементы массива перечисляются в порядке возрастания значений элементов. Это свойство дерева поиска может быть положено в основу построения алгоритма сортировки элементов массива. Однако более интересной является обобщение этой задачи, которую можно сформулировать так:

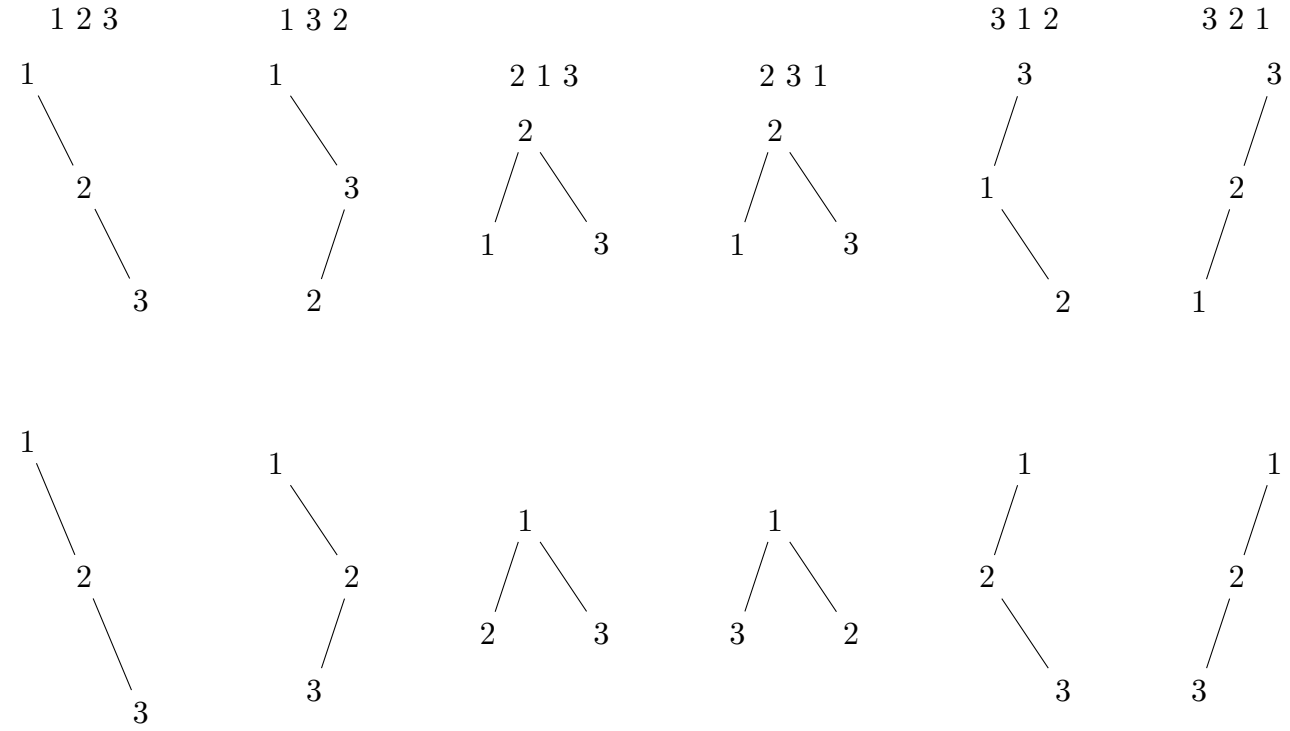
Необходимость сортировки массива связана, в основном, с тем, что поиск конкретных значений элементов в упорядоченном массиве может быть осуществлен в среднем за $O(\log_2 n)$, где $n$ — число элементов в массиве, а в неупорядоченном — за $O(n)$. Предположим, что мы отказываемся от сортировки массивов, заменяя ее построением дерева поиска и поиском конкретных значений элементов в построенном дереве. Давайте оценим эффективность подобной замены.

Ограничимся рассмотрением случая, когда все элементы массива различны. Рассмотрим различные перестановки элементов массива и для каждой такой перестановки построим соответствующее ей дерево поиска. Оценим необходимое число сравнений различных элементов массива для поиска конкретного значения элемента в дереве поиска в среднем по всем возможным перестановкам элементов массива и среднее число сравнений для построения каждого такого дерева.

Пусть $T_n$ — мультимножество всех деревьев, соответствующее всем перестановкам элементов массива.

*Замечание.* Совокупность $T_n$ действительно представляет мультимножество, однако, если узлы дерева помечать номерами мест откуда взято конкретное значение элемента массива то эта совокупность будет множеством.

**Пример.** $n = 3$, элементы массива: 1 2 3, тогда возможны следующие деревья поиска:




Обозначим общее число вершин $i$-го уровня во всей совокупности $T_n$ через $m(i,n)$, а число точек роста, расположенных на $i$-том уровне, через $q(i,n)$. Установим некоторые соотношение между $m(i,n)$ и $q(i,n)$.

Каждая вершина $i$-го уровня, при $i > 1$, расположена в конце дуги, исходящей из вершины $(i-1)$-го уровня, а число таких дуг равно удвоенному числу вершин $(i-1)$-го уровня за вычетом числа точек роста того же уровня, так что

$$m(i,n) = 2 \cdot m(i-1,n) - q(i-1,n), i > 1$$

Естественно принять, что $m(i,n) = 0$, при $i > n$.

Рассмотрим теперь, что происходит при переходе от совокупности $T_n$ к совокупности $T_{n+1}$. Каждое дерево из $T_n$ порождает $n+1$ дерево совокупности $T_{n+1}$ соответственно его $n+1$ точке роста. В каждом из этих деревьев на уровне $i$ сохраняются все вершины $i$-го уровня исходного дерева и, кроме того, в этой частичной совокупности из $n+1$ дерева к ним добавится столько новых вершин $i$-го уровня, сколько точек роста в исходном дереве на $i-1$, т.е.

$$m(i,n+1) = (n+1) \cdot m(i,n) + q(i-1,n), i > 1$$

Складывая полученные равенства, получаем

$$m(i,n+1) = n \cdot m(i-1,n) + 2m(i-1,n), i > 1$$

Число вершин первого уровня совокупности $T_n$ равно числу деревьев:

$$m(1,n) = n!$$

Оценим теперь вычислительную сложность поиска конкретного значения элемента в дереве поиска. Прежде всего заметим, что если элемент находится в дереве на $i$-м уровне, то он обнаруживается после $i$ сравнений.

Пусть $t \in T_n$, $t$ — конкретное дерево, обозначим для $t$ число вершин $i$-го уровня через $m^t(i)$, $i = 1, \ldots, n$. Тогда среднее число сравнений, необходимое для поиска в дереве $t$ будет равно

$$k^t = \sum_{i=1}^{n} i \cdot m^t(i)$$

Но нас интересует среднее значение числа сравнений по всем $n!$ деревьям $n$-го порядка, образующих совокупность $T_n$. Оно равно

$$K(n) = \frac{1}{n!}\sum_{t\in T_n} k^t = \frac{1}{n!}\sum_{t\in T_n}\frac{1}{n}\sum_{i=0}^{n} i\cdot m^t(i) = \frac{1}{n\cdot n!}\sum_{i=1}^{n}(i\cdot\sum_{t\in T_n} m^t(i))$$

Но, по определению, $\sum_{t\in T_n} m^t(i) = m(i,n)$, поэтому

$$K(n) = \frac{1}{n\cdot n!}\sum_{i=1}^{n} i\cdot m(i,n)$$

Опираясь на рекуррентное соотношение для $m(i,n)$, можно получить аналогичное соотношение для $k(n)$

$$K(n) = \frac{(n-1)\cdot(n+1)}{n^2}k(n-1) + \frac{2n-1}{n^2},$$

а с его помощью доказать (по индукции) справедливость выражения

$$K(n) = \frac{2(n+1)}{n}\sum_{i=1}^{n}\frac{1}{n} - 3, n \geq 1$$

Но $\sum_{i=1}^{n}\frac{1}{n} = \ln n + O(1)$, т.е. $K(n) = 2\cdot\ln n + O(1) = 2\cdot\ln 2\log_2 n + O(1)$. Так как $2\cdot\ln 2 \approx 1.39$, среднее время поиска конкретного значения элемента в дереве поиска превышает минимально необходимое на 40%.

Перейдем к задаче определение средней сложности построения дерева поиска.

Пусть некоторому массиву соответствует дерево $t$, то элемент, размещающийся в вершине уровня $i$, займет свое место после $i-1$ сравнения. Отсюда число сравнений, затрачиваемых на организацию этого массива, равно

$$p^t = \sum_{i=1}^{n}(i-1)m^t(i) = \sum_{i=1}^{n} i\cdot m^t(i) - \sum_{i=1}^{n} m^t(i)$$

Первая сумма в этом выражении равна $n\cdot k^t$ , а вторая дает $n$ — число вершин дерева $t$, так что

$$p^t = n\cdot k^t - n$$

Усредняя по всем деревьям из $T_n$, получаем выражение для среднего числа $p(t)$ сравнений, требуемых для построения дерева:

$$p(t) = \frac{1}{n!}\sum_{t\in T_n} p^t = \frac{n}{n!}\sum_{t\in T_n} k^t - n = n * k(n) - n$$

Используя выражение для $k(n)$, получаем

$$p(n) = 2(n+1)\sum_{i=1}^{n}\frac{1}{i} - 4n$$

Для больших $n$ справедлива асимптотическая формула

$$p(n) = 2n\ln n + O(n) = 2\cdot n\cdot\ln 2\cdot\log_2 n + O(n)$$

Так что и здесь вычислительная сложность оказывается больше минимально необходимой в тоже число раз, что для поиска конкретного значения элемента в дереве поиска.

**Упражнение.** Определить дисперсию числа сравнений, необходимых для поиска конкретного значения элемента в дереве поиска. Эта дисперсия определяется выражением

$$d(n) = \frac{1}{n\cdot n!}\sum(i-k(n))^2 m(i,n)$$

# 8. Деревья

**Определение 7.** Связный граф без циклов называется *деревом*. Связный граф без циклов с выделенной вершиной называется *корневым деревом*. Выделенную вершину называют *корнем* дерева.

Легко определяются понятия: потомка, сына, предка, отца для данной вершины; понятие листа и внутренней вершины.

В программировании большой интерес представляют деревья, вершины которого помечены метками, отличающие одну вершину от другой. В дальнейшем будем считать, что в дереве с $n$ вершинами в качестве меток используются числа $1, \ldots, n$.

Заметим, что одно и тоже непомеченное дерево может быть помечено различными способами, т.е. двум различным помеченным деревьям может соответствовать одно и тоже непомеченное дерево.

**Пример.** $n = 3$.

$1 \text{ —— } 2 \text{ —— } 3 \qquad 2 \text{ —— } 1 \text{ —— } 3$

***Теорема 3*** (Кэли, 1897)**.** *Число помеченных деревьев с $n$ вершинами равно $n^{n-2}$.*

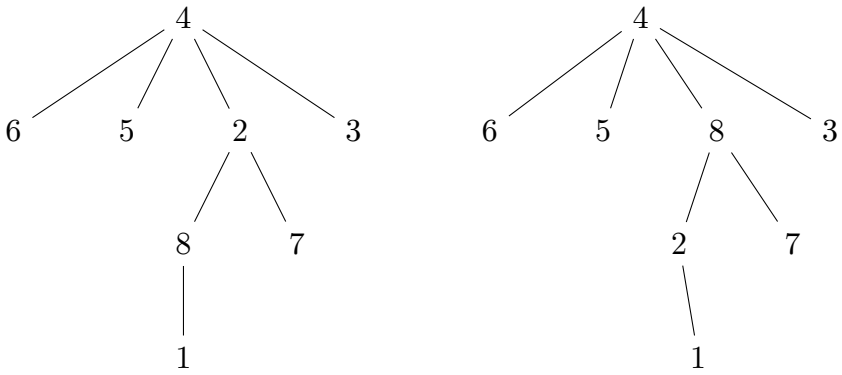
*Доказательство (Прюфер, 1917).* Пусть дано дерево с $n$ вершинами, помеченными числами $1, \ldots, n$. Свяжем с этим деревом последовательность натуральных чисел $i_1, \ldots, i_{n-2}$, построенную следующим образом:

1. положим $j = 0$
2. повторим следующий процесс $n - 2$ раза:

   увеличим значение $j$ на единицу; найдем в дереве лист, помеченный натуральным числом с наименьшим значением. Пусть это значение $k_j$, и пусть отцом листа $k_j$ является вершина, помеченная числом $i_j$. Выберем значение $i_j$ в качестве $j$-ого элемента последовательности. Удалим в дереве ребро $(i_j, k_j)$.

После исполнения этого алгоритма начальное дерево преобразуется в дерево, состоящее из одного ребра: либо $(i_{n-2}, n)$, либо, в случае $i_{n-2} = n$, — из ребра $(n, n-1)$. □

**Упражнение.** Выполните приведенный алгоритм для деревьв




Заметим, что в приведенном алгоритме, построенный им код определен однозначно.

Рассмотрим алгоритм восстановления дерева по его коду Прюфера $i_1, \ldots, i_{n-2}$. Для этого

1. Восстановим заключительное звено дерева. Как было отмечено, им является ребра либо $(i_{n-2}, n)$, в случае $i_{n-2} \neq n$, либо ребро $(n, n-1)$, в случае $i_{n-2} = n$.
2. Для восстановления других ребер дерева выполним следующее

(a) Пусть $j = n - 2$.

(b) Повторим $n - 2$ раза

Если вершина $i_{j-1}$ (исключая $j = 1$) еще не включена в дерево, то строим в нём ребро $(i_j, i_{j-1})$; в противном случае строим ребро $(i_j, m)$, где $m$ — наибольший номер вершины, еще не включенной в дерево.

**Упражнение.** Восстановите деревья по кодам Прюфера, полученным в предыдущем упражнении.

*Замечание.* Пусть задана произвольная последовательность натуральных чисел $i_1, \ldots, i_{n-2}$, каждое из которых из промежутка $1..n$. Тогда, по приведенному алгоритму, для этой последовательности может быть построено помеченное дерево, при этом двум разным последовательностям соответствуют два разных дерева.

Таким образом, теорема Кэли следует из установленной биекции.

Рассмотрим теперь задачу построения приведенных алгоритмов на языке Паскаль. Вначале разберем задачу построения кода Прюфера по заданному дереву. Будем считать, что дерево представлено в виде таблицы смежности, а его код Прюфера мы получаем в целочисленном массиве из $n - 2$ элементов, где $n$ — число вершин помеченного дерева.

*Замечание.* При решении многих задач, связанных с деревьями, выбранное для них представление не является наиболее удобным. В дальнейшем будут рассмотрены другие представления деревьев. В виде списка смежности, вообще говоря, может быть представлен граф самого общего вида, однако при решении данной задачи мы будем исходить из того, представленный в списке смежности граф действительно является деревом. Алгоритм проверки этого свойства представленного графа будет также рассмотрен ниже.

Для решения поставленной задачи мы будем исходить из того, что нам задан следующий глобальный контекст:

```
program pruff;
    const n = 7;
    type v = ^t;
         t = record
                 i: 1..n;
                 r: v
             end;
         pru = array[0..n-2] of integer; {нулевой элемент - 'корень', равный n}
    var k, i, j, a: integer; {n-1 - для основной программы }
        sp: array[1..n] of v;
        x, y: v;
```

Тогда процедура построения кода Прюфера выглядит так:

```
procedure inpruff(var pr: pru); {строит код Прюфера по списку смежности}
    var rab: boolean;
begin
    for j := 1 to n-2 do
    begin
        k:=1;
        repeat
            rab := false;
            if sp[k] = nil then
                k := k+1
            else
```

```
                if sp[k]^.r <> nil then
                    k := k+1
                else
                    rab := true
        until rab;
        a : =sp[k]^.i;
        pr[j] := a;
        sp[k] := nil;
        x := sp[a];
        while (x^.i<>k) do begin
            y := x;
            x := x^.r
        end;
        if sp[a] = x then
            sp[a] := x^.r
        else
            y^.r := x^.r
    end;
end; {inpruff}
```

Процедура восстановления дерева по коду Прюфера выглядит так:

```
procedure outpruff; {преобразует код Прюфера в список смежности}
    procedure bk(var a, b: integer); {включить вершины a, b в список смежности}
    begin
        new(x);
        new(y);
        x^.i := a;
        x^.r := sp[b];
        sp[b] := x;
        y^.i := b;
        y^.r := sp[a];
        sp[a] := y
    end; {bk}
begin {outpruff}
    for i := 1 to n do
        sp[i] := nil;
    pr[0] := n;
    a := pr[n-2];
    if a = n then
        i := n-1
    else
        i := n;
    bk(a, i);
    for j := n-2 downto 1 do
{1}     if sp[pr[j-1]] = nil then
            bk(pr[j-1], pr[j])
        else
        begin
            while sp[i] <> nil do
                i := i - 1;
            bk(i, pr[j])
```

```
        end
end; {outpruff}
```

*Комментарий.* Нулевой элемент в описании типа `pru` введен для корректного вычисления, в случае $j = 1$, логического выражения в условии оператора `{1}`. В массиве `pr`, используемого процедурой `outpruff` в качестве значения кода Прюфера, нулевому элементу присвоено значение равное $n$.

*Замечание.* Код Прюфера является оптимальным по памяти кодирования деревьев. В самом деле, $W = \bigcup_{n=1}^{\infty} W_n$, где $W_n$ — конечные множества. Предположим,что при каком-то способе кодирования элементов из $W$ для запоминания одного элемента из $W_n$ используется самое большое $f(n)$ бит памяти. Кодирование $f(n)$ называется оптимальным, если $\lim_{n\to\infty} \dfrac{f(n)}{h(n)} = 1$, где $h(n) = \log_2 |W_n|$.

Для оценки энтропии $h(n)$ для множества деревьев с $n$ вершинами по теореме Кэли имеем $h(n) = (n-2)\cdot\log_2 n$. Отсюда следует, что для кодирования деревьев кодом Прюфера необходимо $f(n) = (n-2)\cdot\log_2 n$ бит памяти, и поэтому $\lim_{n\to\infty} \dfrac{f(n)}{h(n)} = 1$.

При задании дерева списками смежности имеем $\lim_{n\to\infty} \dfrac{f(n)}{h(n)}$ для неориентированных графов, и $\lim_{n\to\infty} \dfrac{f(n)}{h(n)} = 2$ для ориентированных.

В.А. Евстигнеев. Применение теории графов в программировании. Наука. 1985.

# 9. Транспортная задача

```
program transport;
    uses CRT; {1 - туда 2 - обратно 3 - оба }
    const n = 8; mr = 1.0E10;
    type yk = ^el;
         el = record
                  nom: integer;
                  r: real;
                  tp: 1..3;
                  next: yk
              end;
    var sp: array[1..n] of record
                               b: real;
                               ref: yk
                           end;
        f: Text;
        i, j: integer;
        a: real;
        x, y: yk;
        t: 1..3;
procedure WWOD;
begin
    assign(f, 'gro.txt');
    reset(f);
    while not EOF(f) do
    begin
        read(f, i);
```

```
        sp[i].b := mr;
        if eoln(f) then
            sp[i].ref := NIL
        else
        begin
            write(i);
            read(f, j, a, t);
            new(x);
            x^.nom := j;
            x^.tp := t;
            x^.r := a;
            x^.next := nil;
            sp[i].ref := x;
            while not eoln(f) do
            begin
                read(f, j, a, t);
                new(y);
                y^.nom := j;
                y^.r: = a;
                y^.tp := t;
                y^.next := nil;
                x^.next := y;
                x := y
            end
        end
    end
end;
procedure VYVOD;
    var p: integer;
begin
    writeln;
    for p := 1 to n do
    begin
        write(p:3, ' ', sp[p].b:9:1, ' ');
        x := sp[p].ref;
        while x <> nil do
        begin
            write(x^.nom:2, x^.r:5:1, x^.tp:2, ' ');
            x := x^.next
        end;
        writeln;
    end;
end;
procedure WAVE (i: integer; rr: real); {в ширину}
    var pp, k: integer;
        p: array[1..n] of integer;
begin
    x := sp[i].ref;
    pp := 0;
    while x <> nil do
    begin
        k := x^.nom;
```

```
        a := rr + x^.r;
        if ((x^.tp = 1) or (x^.tp = 3)) and (sp[k].b > a) then
        begin
            pp := pp + 1;
            sp[k].b := a;
            p[pp] := k
        end;
        x := x^.next
    end;
    for k := 1 to pp do
        WAVE(p[k], sp[p[k]].b)
end;
{procedure WAVE (i: integer; rr: real); {в глубину}
    var x: yk;
begin
    x := sp[i].ref;
    while x <> nil do begin
        k := x^.nom;
        a := rr + x^.r;
        if ((x^.tp = 1) or (x^.tp = 3)) and (sp[k].b > a) then
        begin
            sp[k].b := a;
            WAVE(k,a)
        end;
        x := x^.next
    end;
end;}
procedure REVERSE (j: integer);
    var a: real;
        k: integer;
begin
    if j <> i then
    begin
        x := sp[j].ref;
        k := j;
        a := sp[j].b;
        while x <> nil do
        begin
            if (x^.tp > 1) and (sp[x^.nom].b < a) then
            begin
                k := x^.nom;
                a := sp[x^.nom].b
            end;
            x := x^.next;
        end;
        REVERSE(k);
        write('-->', j)
    end
    else
        write(j)
end;
begin
```

```
    ClrScr;
    WWOD;
    VYVOD;
    Writeln('введите начальную и конечную вершины');
    read(i, j);
    sp[i].b := 0;
    WAVE(i,0);
    VYVOD;
    if sp[j].b >= mr then
        writeln('вершины не связаны путем')
    else
        REVERSE(j)
end.
```