**Алгоритм Шеннона — Фано**

**Алгоритм Шеннона — Фано** — один из первых алгоритмов сжатия, который впервые сформулировали американские учёные [Шеннон](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D0%BD%D0%BD%D0%BE%D0%BD,_%D0%9A%D0%BB%D0%BE%D0%B4_%D0%AD%D0%BB%D0%B2%D1%83%D0%B4) и [Роберт Фано](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%BD%D0%BE,_%D0%A0%D0%BE%D0%B1%D0%B5%D1%80%D1%82). Данный метод сжатия имеет большое сходство с [алгоритмом Хаффмана](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A5%D0%B0%D1%84%D1%84%D0%BC%D0%B0%D0%BD%D0%B0), который появился на несколько лет позже и является логическим продолжением [алгоритма Шеннона](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A8%D0%B5%D0%BD%D0%BD%D0%BE%D0%BD%D0%B0). Алгоритм использует коды переменной длины: часто встречающийся символ кодируется кодом меньшей длины, редко встречающийся — кодом большей длины. Коды Шеннона — Фано — префиксные, то есть никакое кодовое слово не является [префиксом](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D1%84%D0%B8%D0%BA%D1%81) любого другого. Это свойство позволяет однозначно декодировать любую последовательность кодовых слов.

Алгоритм был независимо друг от друга разработан Шенноном (публикация «Математическая теория связи», 1948 год) и, позже, Фано (опубликовано как технический отчёт).

**Условие Фано**: для того, чтобы сообщение, записанное с помощью неравномерного по длине кода, однозначно раскодировалось, достаточно, чтобы **никакой код не был началом другого (более длинного) кода**.

Более «математическая» формулировка:

Если в код входит слово *a*, то для любой непустой строки *b* слова *ab* в коде не существует.

**Обратное условие Фано** также является достаточным условием однозначного декодирования неравномерного кода. В нём требуется, чтобы **никакой код не был окончанием другого (более длинного) кода**.

Для возможности однозначного декодирования достаточно выполнения одного из условий — или прямого, или обратного.

## Алгоритм вычисления кодов Шеннона — Фано.

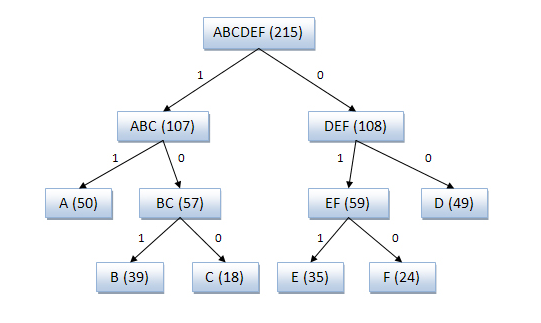
Код Шеннона — Фано строится с помощью дерева. Построение этого дерева начинается от корня. Всё множество кодируемых элементов соответствует корню дерева (вершине первого уровня). Оно разбивается на два подмножества с примерно одинаковыми суммарными вероятностями. Эти подмножества соответствуют двум вершинам второго уровня, которые соединяются с корнем. Далее каждое из этих подмножеств разбивается на два подмножества с примерно одинаковыми суммарными вероятностями. Им соответствуют вершины третьего уровня. Если подмножество содержит единственный элемент, то ему соответствует концевая вершина кодового дерева; такое подмножество разбиению не подлежит. Подобным образом поступаем до тех пор, пока не получим все концевые вершины. Ветви кодового дерева размечаем символами 1 и 0, как в случае кода Хаффмана.

При построении кода Шеннона — Фано разбиение множества элементов может быть произведено, вообще говоря, несколькими способами. Выбор разбиения на уровне n может ухудшить варианты разбиения на следующем уровне (n + 1) и привести к неоптимальности кода в целом. Другими словами, оптимальное поведение на каждом шаге пути ещё не гарантирует оптимальности всей совокупности действий. Поэтому код Шеннона — Фано не является оптимальным в общем смысле, хотя и дает оптимальные результаты при некоторых распределениях вероятностей. Для одного и того же распределения вероятностей можно построить, вообще говоря, несколько кодов Шеннона — Фано, и все они могут дать различные результаты. Если построить все возможные коды Шеннона — Фано для данного распределения вероятностей, то среди них будут находиться и все коды Хаффмана, то есть оптимальные коды.

### Пример кодового дерева

Исходные символы:

* A (частота встречаемости 50)
* B (частота встречаемости 39)
* C (частота встречаемости 18)
* D (частота встречаемости 49)
* E (частота встречаемости 35)
* F (частота встречаемости 24)

[](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%A8%D0%B5%D0%BD%D0%BD%D0%BE%D0%BD%D0%B0.PNG)

Полученный код: A — 11, B — 101, C — 100, D — 00, E — 011, F — 010.

Кодирование Шеннона — Фано является достаточно старым методом сжатия, и на сегодняшний день оно не представляет особого практического интереса. В большинстве случаев длина последовательности, сжатой по данному методу, равна длине сжатой последовательности с использованием кодирования Хаффмана. Но на некоторых последовательностях могут сформироваться неоптимальные коды Шеннона — Фано, поэтому более эффективным считается сжатие методом Хаффмана.

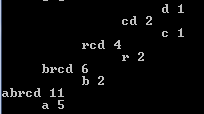
# Постановка задачи.

Вход программы: некоторый текст.

Результаты работы программы:

1) таблица частот

2) построенное дерево;



3) код каждого символа;



4) закодированный текст;

5) раскодировать последовательность кодов, которая вводится, используя построенное дерево.

Помощь к заданию.

*Выбор структур данных*

1. alphabet- строка с алфавитом

Для нашего примера alphabet=’ABCDEF’

2. alwes= new Array() – массив для хранения частот символа,

например alwes[A]=50.

3. Для представления двоичных деревьев выберем массив, элементами которого является запись. Для каждого узла дерева в записи сохраним информацию о его левом, правом сыне и родителе, вес узла, начальная и конечная позиция подстроки из алфавита.. Количество элементов в массиве соответствует количеству узлов дерева.

function tree (lson, rson, parent, wes, start\_pos,end\_pos) {

…

}

T=new Array()

Результирующее состояние массива для дерева из рассмотренного выше примера:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| lson | 1 | 4 | 8 | 0 | 6 | 0 | 0 | 10 | 0 | 0 | 0 |
| rson | 2 | 5 | 9 | 0 | 7 | 0 | 0 | 11 | 0 | 0 | 0 |
| parent | 0 | 1 | 1 | 2 | 2 | 5 | 5 | 3 | 3 | 8 | 8 |
| wes | 215 | 107 | 108 | 50 | 57 | 39 | 18 | 59 | 49 | 35 | 24 |
| start\_pos | 0 | 0 | 3 | 0 | 1 | 1 | 2 | 3 | 5 | 3 | 4 |
| end\_pos | 5 | 2 | 5 | 0 | 2 | 1 | 2 | 4 | 5 | 3 | 4 |

4. Процедура построения дерева

lastusel-последний созданный узел.

function ShTree(usel){

//Критерий выхода: если позиции символов совпали, то это конец

if (T[usel].start\_pos <>T[usel].end\_pos){

//поиск середины

s = 0;

i = T[usel].start\_pos;

while ((s+ alwes [alphabet [i]]<T[usel].wes/2) и (i<T[usel].end\_pos))

{

s = s + alwes [alphabet [i]]

i++

}

if i> T[usel].start\_pos

i=i-1

else

s = alwes [alphabet [i]]

T[usel].lson=lastusel+1

T[usel].rson=lastusel+2

T[lastusel+1]=new tree(0,0,usel,s,T[usel].start\_pos,i)

T [lastusel+2]=new tree(0,0,usel,T[usel].wes-s,i+1,T[usel].end\_pos)

lastusel=lastusel+2

//Рекурсия левая ветка дерева }

ShTree(T[usel].lson);

// Правая ветка дерева }

ShTree(T[usel].rson);

}

}

5. Вывод построенного дерева на экран

Идея следующая: организовывается рекурсивная функция, параметрами которой является указатель и строка для вывода на консоль.   
  
Сначала в нее подается корень и пустая строка.   
Если есть левое поддерево (левый сын), вызывается эта же функция для левого поддерева и в строку ей дописывается один пробел,   
затем печатается текущая вершина с использованием текущей строки отступов,   
если есть правое поддерево (правый сын), функция вызывается для правого поддерева и в строку ей снова дописывается один пробел.  
Получится дерево, уложенное на бок.

Алгоритмы просмотра дерева

Самой интересной особенностью обработки бинарных деревьев является та, что при изменении порядка просмотра дерева, не изменяя его структуры, можно обеспечить разные последовательности содержащейся в нем информации. В принципе возможны всего четыре варианта просмотра: слева-направо, справа-налево, сверху-вниз и снизу-вверх. Прежде чем увидеть, к каким результатам это может привести, приведем их.

а. Просмотр дерева слева – направо

**procedure** ViewLR(Root:PNode); {LR -> Left – Right }

**begin**

**if** Root<>Nil **then**

**begin**

ViewLR(Root^. left); {просмотр левого поддерева}

{Операция обработки корневого элемента –

вывод на печать, в файл и др.}

ViewLR(Root^.right); { просмотр правого поддерева }

**end**;

**end**;

б. Просмотр справа налево

**procedure** ViewRL(Root:PNode); {LR -> Right – Left}

**begin**

**if** Root<>Nil **then**

**begin**

ViewRL(Root^.right); {просмотр правого поддерева}

{Операция обработки корневого элемента –

вывод на печать, в файл и др.}

ViewRL(Root^.left); { просмотр левого поддерева }

**end**;

**end**;

в. Просмотр сверху – вниз

**procedure** ViewTD(Root:PNode); {TD –> Top-Down}

**begin**

**if** Root<>Nil **then**

**begin**

{Операция обработки корневого элемента –

вывод на печать, в файл и др.}

ViewTD(Root^.left); {просмотр левого поддерева}

ViewTD(Root^.right); { просмотр правого поддерева }

**end**;

**end**;

г. Просмотр снизу-вверх

**procedure** ViewDT(Root:PNode); {DT –> Down - Top}

**begin**

**if** Root<>Nil **then**

**begin**

ViewDT(Root^.left); {просмотр левого поддерева}

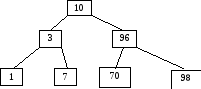
ViewDT(Root^.right); { просмотр правого поддерева }

{Операция обработки корневого элемента –

вывод на печать, в файл и др.}

**end**;

**end;**

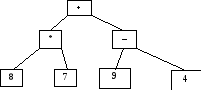


*Пример 1*. Рассмотрим результаты просмотра для приведенных алгоритмов, при условии, что обработка корневого элемента сводится к выводу значения его информационного поля, а дерево в этот момент имеет следующие узлы:

Результаты просмотра:

|  |  |
| --- | --- |
| Алгоритм «Слева направо» | 1, 3, 7, 10, 70, 96, 98 |
| Алгоритм «Справа налево» | 98, 96, 70, 10, 7, 3, 1 |
| Алгоритм «Сверху вниз» | 10, 3, 1, 7, 96, 70, 98 |

Из приведенной таблицы видно, что, просто изменяя порядок просмотра дерева (слева-направо и справа-налево), можно получить отсортированные по возрастанию или по убыванию числа.

*Пример 2*. Пусть в узлах дерева расположены элементы арифметического выражения:

Результаты просмотра:

|  |  |  |
| --- | --- | --- |
| «Слева направо» | 8 \* 7 + 9 – 4 | инфиксная форма записи выражения |
| «Сверху вниз» | **+** \* 8 7 – 9 4 | префиксная форма записи выражения |
| «Снизу вверх» | 8 7 \* 9 4 – **+** | постфиксная форма записи выражения |