Моделирование случайной величины

В лабораторных работах потребуется моделировать случайную величину на ЭВМ. Для этого можно воспользоваться генератором псевдослучайной последовательности. Например, использовать линейно-конгруэнтный генератор.

Искомая последовательность псевдослучайных чисел получается из соотношения

$$X_{n+1} = \left(aX_n + c\right) \bmod m, \quad n \geq 0.$$

Программа проверки соответствия коэффициентов *a* и *c* линейно-конгруэнтного генератора коэффициентам генератора от Borland по результирующей псевдослучайной последовательности.

```
#include <fstream.h>
#include <stdlib.h>
void main(int mn,char* nm[])
{
if(mn!=4) cerr<<"proba.exe output.qfb N begin\n",exit(1);
ofstream ou(nm[1],ios::binary);
if(!ou) cerr<<"Выходной файл "<<nm[1]<<" не создан!\n",exit(1);
ofstream oi("rnd_borl.qfb",ios::binary);
if(!oi) cerr<<"Выходной файл \"rnd_borl.qfb\" не создан!\n",exit(1);
float r;
long x=0x015A4E35;
long i,k=atol(nm[2]);
long beg=atol(nm[3]);
srand(beg);
for(i=0;i<k;i++)
 {
 r=rand(), oi.write((char*)&r,4);
 beg=x*beg+1, r=(beg>>16)&0x7fff, ou.write((char*)&r,4);
 }
ou.close(); oi.close();
```

Сущность и цели операции кодирования

Под *кодированием* (в узком смысле или дискретном кодировании) понимается процедура сопоставления дискретному сообщению $a_i(i=0, 1, ..., K-1)$ определённой последовательности элементов $b_i=\{b_k\}(i=0, 1, ..., K-1; k=0, 1, ..., m-1;)$. Обычно $m<K$ (например, a_i — буквы русского алфавита и $K=32$, а b_k — двоичные символы 0 и 1, и $m=2$).

С помощью кодирования решатся следующие *задачи*:

1. Согласование алфавита источника сообщений с алфавитом канала.
2. «Сжатие» информации (уменьшение или полное устранение избыточности, содержащейся в сообщении).
3. Обнаружение или исправление ошибок, возникающих в канале связи из-за помех и искажений сигнала.

Если кодирование решает только первую задачу, код называют *примитивным*. Вторую задачу решает *экономный код*. Третью задачу решает *помехоустойчивый код*. Помехоустойчивые коды также называют *корректирующими* (они способны обнаруживать и исправлять возникающие в канале ошибки и таким образом повышают верность приёма). Корректирующие коды работают за счёт внесения определённой избыточности в передаваемую кодовую последовательность.

В современных цифровых информационных системах часто встречаются все три вида кодирования, применяемых последовательно: сначала первичное сообщение представляется в двоичной форме (примитивный код), затем оно сжимается для устранения избыточности (экономный код), затем кодируется помехоустойчивым кодом для повышения верности передачи.

Классификация кодов и их представление

Основной признак классификации: *основание кода* m — число различных используемых в коде символов. Наиболее простыми и наиболее распространёнными являются *двоичные* (*бинарные*) коды, у которых $m = 2$. Коды с $m > 2$ называют *многопозиционными*.

Различают *блочные* и *непрерывные* коды. *Блочными* называют коды, в которых каждый элемент («символ») сообщения a_i преобразуется в определённую последовательность (блок) кодовых символов b_i, называемую *кодовой комбинацией*. *Непрерывные* коды образуют последовательность кодовых символов b_i, в которой невозможно выделить отдельные кодовые комбинации.

Коды подразделят также на *равномерные* и *неравномерные*. В *равномерных* кодах все кодовые комбинации содержат одинаковое количество символов (*разрядов*). В *неравномерных* кодах число символов (разрядов) в кодовых комбинациях, соответствующих различным исходным символам источника может быть различно.

К любому коду обязательно предъявляется требование *однозначности декодирования*. Это означает, что всякой последовательности или группе последовательностей кодовых символов должна однозначно соответствовать последовательность элементов передаваемого сообщения, то есть потери информации при кодировании и декодировании должны равняться нулю.

Если объём алфавита источника сообщений равен K, то каждую букву можно закодировать с помощью n-разрядного m-позиционного кода при условии, что $m^n \geq K$.

Если имеет место равенство, то есть при кодировании используются все возможные кодовые комбинации, то код называется *безызбыточным* или *примитивным*. Если же число возможных кодовых комбинаций больше числа различных символов источника и, следовательно, часть кодовых комбинаций остаётся «неиспользованной», то такой код называют *избыточным* или *корректирующим*.

Любой код характеризуется *коэффициентом избыточности*:

$$\rho_к = 1 - \frac{H(B)}{\log m} = 1 - \frac{\upsilon_к H(B)}{\upsilon_к \log m} = 1 - \frac{H'(B)}{\upsilon_к \log m},$$

где $\upsilon_к$ — скорость выдачи символов кодером, $H'(B)$ — производительность кодера. Поскольку при кодировании потери информации отсутствуют, производительность кодера равняется производительности источника: $H'(B) = H'(A)$. Тогда

$$\rho_к = 1 - \frac{H'(A)}{\upsilon_к \log m} = 1 - \frac{\upsilon_и H(A)}{\upsilon_к \log m} = 1 - \frac{H(A)}{\overline{n} \log m}$$

где $\upsilon_и$ — скорость выдачи символов источником, $\overline{n} = \frac{\upsilon_к}{\upsilon_и}$ — среднее число кодовых символов, приходящееся на один символ источника.

Для источника без избыточности и равномерного кода $\rho_к = 1 - \frac{\log K}{n \log m}$.

При кодировании совокупность символов источника сообщений a_i удобно представить последовательностью чисел: $0, 1, 2, ..., K-1$ (то есть пронумеровать символы источника).

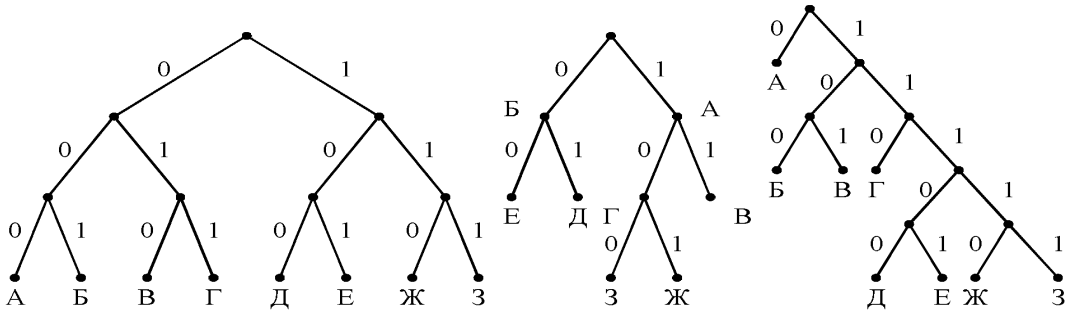
С другой стороны, любое число *M* можно записать в заданной *m*-ичной *системе счисления* следующим образом: $M = b_{n-1}m^{n-1} + b_{n-2}m^{n-2} + ... + b_k m^k + ... + b_1 m^1 + b_0 m^0$,

где коэффициенты b_k принимают значения от 0 до $m-1$.

Такой же принцип применяется и при кодировании символов источника безызбыточным *m*-разрядным *n*-позиционным кодом. В общем виде минимально необходимое число разрядов можно найти из соотношения $m^n \geq K \Rightarrow n = \log_m K$.

Всякий блочный код можно представить в виде таблицы (например, ASCII), в которой каждой букве алфавита источника поставлена в соответствие определённая кодовая комбинация. Помимо таблицы код часто представляют в виде «*кодового дерева*».

Рассмотрим пример построения кодового дерева на примере двоичного кода. Для этого, начиная с определённой точки («*корня*»), будем проводить отрезки прямой («*ветви*»), наклонённые влево (если кодовый символ равен 0) или вправо (если кодовый символ равен 1). На рисунке приведены примеры кодовых деревьев для равномерного (а); неравномерного приводимого (б); неравномерного неприводимого (в).



а) *б)* *в)*

А) Каждой букве источника соответствует своя вершина кодового дерева. С помощью такого дерева можно однозначно декодировать любую последовательность символов, если только она принята с самого начала. Это справедливо для любого *равномерного* кода.

Б) Символы располагаются не только в вершинах, но и в узлах дерева. Декодирование такого кода будет неоднозначным. Например, символ 0 на входе декодера может означать символ «Б», а может оказаться началом кодовых комбинаций, соответствующих символов «Д» и «Е». Такой код называют *приводимым*. В этом случае для однозначного декодирования необходимо использовать специальные символы-разделители. Примером неравномерного приводимого кода может служить код Морзе.

В) Это тоже неравномерный код. При этом однозначное декодирование возможно без разделителей. Здесь все буквы располагаются только в вершинах кодового дерева. Ни одна кодовая комбинация не будет являться началом другой. Этот код является *неприводимым* или *префиксным*. Если последовательность кодовых символов принята сначала, то можно образовать законченную кодовую комбинацию только дойдя до какой-либо вершины. Возвращаясь к корню дерева можно декодировать следующую кодовую комбинацию. Такие коды широко применяются в современных архиваторах.

Неприводимость кода является достаточным, но не необходимым условием однозначности декодирования. Теоретически возможно составить коды, не обладающие этим свойством, но допускающие однозначное декодирование без применения дополнительных разделителей при помощи более сложной логики.

}Сжатие информации (Экономное кодирование)

Экономное кодирование, называемое также *сжатием* данных, используется для уменьшения времени передачи информации или объёма памяти, необходимой для её хранения. Суть экономного кодирования заключается в том, что избыточность на выходе кодера уменьшается по сравнению с избыточностью на его входе (избыточностью первичного источника). Сжатие данных не может быть бóльшим некоторого теоретического предела.

Среднее количество бит, приходящихся на одно кодируемое значение дискретной случайной величины, не может быть меньшим, чем энтропия этой дискретной случайной величины, то есть $ML(X) \geq HX$ для любой дискретной случайной величины и её кода.

Учёными доказано, что существует такое кодирование (Шеннона–Фэно), что $HX \geq ML(X) - 1.$

В итоге имеем двойное неравенство $ML(X) \geq HX \geq ML(X) - 1.$

Для *n*-мерной дискретной случайной величины $\vec{X} = (X_1, X_2, ..., X_n)$ при одинаково распределённых и независимых дискретных случайных величин, входящих в неё, справедливо $H\vec{X} = nHX_1$.

Пусть количество бит кода на единицу сообщения $\vec{X}$:

$L_1(\vec{X}) = \frac{L(\vec{X})}{n}$, тогда $ML_1(\vec{X})$ — среднее количество бит кода на единицу сообщения при передаче бесконечного множества сообщений $\vec{X}$. Из $ML(X) - 1 \leq HX \leq ML(X)$ для кода Шеннона–Фэно для $\vec{X}$ следует $ML_1(X) - \frac{1}{n} \leq HX_1 \geq ML_1(X)$ для этого же кода.

Таким образом, с ростом длины *n* сообщения, при кодировании методом Шеннона–Фэно всего сообщения целиком среднее количество бит на единицу длины сообщения будет сколь угодно мало отличаться от энтропии единицы сообщения.

Такое кодирование практически нереализуемо из-за роста трудоёмкости при росте длины сообщения. Невозможна отправка сообщения по частям Требуется отправка кода вместе с его исходной длиной — снижается эффект от сжатия.

Для повышения степени сжатия используют метод блокирования. По выбранному значению $\varepsilon > 0$ можно выбрать такое *s*, что если разбить всё сообщение на блоки длиной *s* (всего $\frac{n}{s}$ блоков), то кодированием Шеннона-Фэно таких блоков, рассматриваемых как единицы сообщения, можно сделать среднее количество бит на единицу сообщения бóльшим энтропии менее, чем на ε.

Пример. ДСВ $X_1, X_2, ..., X_n$ для $i = 1, ..., n$

$$P(X_i = 0) = p = \frac{3}{4} \qquad P(X_i = 1) = q = \frac{1}{4}$$

тогда $HX_i = -\frac{3}{4}\log_2\frac{3}{4} - \frac{1}{4}\log_2\frac{1}{4} = 2 - \frac{3}{4}\log_2 3 \approx 0{,}811 \frac{\text{бит}}{\text{симв.}}$

Пример. 2-мерной ДСВ $ML_1(\vec{X}) = \frac{27}{32} = 0{,}84375 \frac{\text{бит}}{\text{симв.}}$, 3-мерной ДСВ $ML_1(\vec{X}) \approx 0{,}823 \frac{\text{бит}}{\text{симв.}}$, 4-мерной ДСВ $ML_1(\vec{X}) \approx 0{,}818 \frac{\text{бит}}{\text{симв.}}$ и т. п.

Таким образом, показано, как находить пределы для средней длины кода на единицу сообщений, передаваемых много раз.

Пример программы, подсчитывающей количество бит в среднем на единицу сообщения, для блоков разной длины (распределение задаётся сразу для *n*-мерной величины)

```
float f(float& m,int* p,int l,int* a=NULL,int x=0,int s=0,int u=0)
{ if(!a) a=new int[l];
for(int i=1;i<=x*2+!x;i++)
 if(s+i<l) f(m,p,l,a,a[u]=i,s+i,u+1);
 else
  {
  if(s+i==l)
   { int j,S,k,W; float ML;
   for(a[u]=i,i=0;i<=u;cout<<a[i++]<<" "); cout<<endl;
   for(cout<<"L: ",i=1;i<=u;cout<<(a[i-1]<<1)-a[i]<<" x "<<i<<", ",i++);
   cout<<(a[i-1]<<1)<<" x "<<i<<endl;
   for(W=k=S=i=0;i<=l;S+=p[i++]); cout<<"S="<<S<<endl;
   for(i=1;i<=u;i++) for(j=0;j<(a[i-1]<<1)-a[i];j++,W+=p[k++]*i);
   for(j=0;j<(a[i-1]<<1);j++,W+=p[k++]*i);
   ML=W/float(S)/(log(double(l+1))/log(2.)); if(ML<m) m=ML;
   cout<<"ML="<<ML<<endl;
   }
  break;
  }
if(!u) delete[] a;
return m;
}

int main()
{ int i,N; float m;
//int p[]={3,1};
//int p[]={9,3,3,1};
int p[]={27,9,9,9,3,3,3,1};
//int p[]={81,27,27,27,27,9,9,9,9,9,9,3,3,3,3,1};
m=N=sizeof p/sizeof p[0]; cout<<"N="<<N<<endl;
cout<<"ML1(X)="<<f(m,p,N-1)<<endl;
return 0;
}
```

Минимально возможное среднее количество кодовых символов на один символ источника (предел Шеннона) можно найти из условия

$$\rho_{и} = 1 - \frac{H(A)}{\overline{n}_{\min} \log m} = 0 \Rightarrow \overline{n}_{\min} = \frac{H(A)}{\log m}.$$

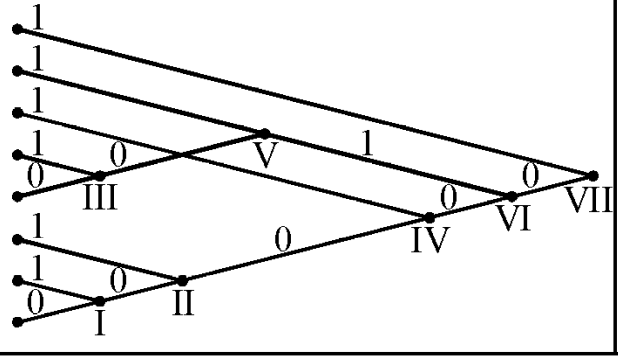
Так как примитивный код не может обеспечить эффективное согласование источника с избыточностью с каналом связи, то для канала без помех эффективно согласование для такого источника может быть достигнуто при использовании неравномерного кода, длительность комбинации которого должна соответствовать вероятности выбора источником отдельных символов (букв). Такое кодирование называют *статистическим*.

Неравномерный код при статистическом кодировании выбирается так, чтобы более вероятные символы передавались более короткими кодовыми комбинациями, а менее вероятные — более длинными. В итоге средняя длина кодовой комбинации уменьшается по сравнению с равномерным кодированием.

Код Хаффмана

Построение этого кода рассмотрим на конкретном примере источника с объёмом алфавита $K = 8$.

Символы (буквы) алфавита располагают в порядке убывающей вероятности, затем выбирают пару букв с наименьшими вероятностями (0,02 и 0,03), от них проводят прямые (ветви на кодовом дереве) до узла I — точки условного символа с суммарной вероятностью $0{,}02 + 0{,}03 = 0{,}05$. При этом ветви, проведённой сверху вниз, приписывают символ 1, а ветви, проведённой снизу вверх, — символ 0. Среди символов a_1, a_6 и I снова находят пару символов с наименьшими вероятностями (0,08 и 0,05) и от них проводят прямые до точки II — точки условного символа с суммарной вероятностью $0{,}08 + 0{,}05 = 0{,}13$. Этот процесс продолжается дальше, пока построение не замыкается к вершине, то есть формирование кодовой комбинации идёт слева направо. Мы получили кодовое дерево, которое позволяет нам написать кодовые комбинации для всех символов, начиная построение с вершины дерева. Полученный неравномерный код является префиксным.

Символ источника	Вероятность $p(a_k)$	Кодовое дерево	Код	n_k	$n_k p(a_k)$
a_1	0,40		1	1	0,40
a_2	0,13		011	3	0,39
a_3	0,12		001	3	0,36
a_4	0,11		0101	4	0,44
a_5	0,11		0100	4	0,44
a_6	0,08		0001	4	0,32
a_7	0,03		00001	5	0,15
a_8	0,02		00000	5	0,10

Средняя длина $\overline{n}\ \dfrac{\text{бит}}{\text{символ}}$ кодовой комбинации в нашем примере

$$\overline{n} = \sum_{k=0}^{7} n_k p_k = 2{,}6,$$

в то время как при примитивном кодировании двоичным кодом пришлось бы для всех кодовых комбинаций использовать три символа (разряда).

Предел Шеннона при двоичном коде для среднего числа символов на букву

$$\overline{n}_{\min} = \frac{H(A)}{\log_2 2} = H(A).$$

В нашем примере энтропия источника (считаем, что отсутствуют вероятностные связи между символами)

$$H(A) = -\sum_{k=1}^{8} p_k \log_2 p_k = 2{,}535\ \frac{\text{бит}}{\text{символ}}.$$

То есть код Хаффмана позволил получить близкое к теоретическому пределу значение средней длины кодового слова.

Программа кодера по Хаффману

```
struct haff
{ unsigned s; int i; int p; char c; char k;int j;
void print() {cout<<"c="<<c<<" i="<<i<<" j="<<j<<" p="<<p<<" s="<<s<<" k="<<k<<endl;}
};
#define Q 0 // 1 - по вероятности, 0 - согласно ссылке на сортировку
int main(int mn,char* nm[])
{ int N,size,i,j,k,min,jmin1,jmin2,max,imax,Max=-2u/2,s,v;
int b[256]={0};
haff* h; short* c; unsigned char *p,g;
ExeFile(nm[0]); if(mn!=3) cerr<<nm[0]<<" in.ext out.arh\n",exit(1);
ifstream in(nm[1],ios::binary); if(!in) cerr<<"file \""<<nm[1]<<"\" not open!\n",exit(1);
in.seekg(0,ios::end);size=in.tellg();in.seekg(0,ios::beg);
p=new unsigned char[size];if(!p) cerr<<"Error memory!\n",exit(1);
in.read((char*)p,size);in.close();
for(i=0;i<size;i++) b[p[i]]++;
for(N=i=0;i<256;i++) N+=!!b[i];
cout<<"N="<<N<<endl;
h=new haff[2*N+1];if(!h) cerr<<"Error memory!\n",exit(1);
for(j=0;j<2*N-1;j++) h[j].c=h[j].s=h[j].i=h[j].k=h[j].p=h[j].j=0;
for(j=i=0;i<256;i++) if(b[i]) h[j].s=b[i],h[j++].c=i;
vector<vector<char>*> V(N);
for(j=-N;j<0;imax>=0?h[imax].i=h[imax].j=j++:j++) for(imax=-1,max=k=0;k<N;k++) if(!h[k].i) if(h[k].s>max) max=h[k].s,imax=k;
for(s=j=0;j<N;s+=h[j++].s);
cout<<"s="<<s<<endl;
for(i=0;i<N-1;i++)
 { for(min=Max,jmin1=-1,j=0;j<N+i;j++) if(!h[j].p) if(min>=h[j].s) min=h[j].s,jmin1=j;
 h[jmin1].p=N+i;
 for(min=Max,jmin2=-1,j=0;j<N+i;j++) if(!h[j].p) if(min>=h[j].s) min=h[j].s,jmin2=j;
 h[jmin2].p=N+i;
 if(Q) h[jmin1].k=0x30,h[jmin2].k=0x31; // установка цифрового кода (0 или 1) по вероятности
 else if(h[jmin1].j<h[jmin2].j) h[N+i].j=h[jmin1].j,h[jmin1].k=0x31,h[jmin2].k=0x30; else h[N+i].j=h[jmin2].j,h[jmin1].k=0x30,h[jmin2].k=0x31;
 h[N+i].s=h[jmin1].s+h[jmin2].s;
 h[N+i].i=i+1;
 }
for(j=0;j<N;j++)
 { for(i=0,k=j;h[k].p;k=h[k].p,i++);
 V[j]=new vector<char>(i); if(!V[j]) cerr<<"Error memory!\n",exit(1);
 for(i=0,k=j;h[k].p;k=h[k].p,i++) (*V[j])[V[j]->size()-1-i]=h[k].k;
 }
for(j=0;j<N;j++,cout<<endl)
 for(k=(int)(unsigned char)h[j].c-32,k<0?cout<<"\'\\"<<k+32<<"\': ":cout<<"\'"<<char(k+32)<<"\': ",i=0;i<V[j]->size();i++) cout<<(*V[j])[i];
ofstream ou(nm[2],ios::binary); if(!ou) cerr<<"Error create the file \""<<nm[1]<<"\"\n",exit(1);
g=N-1; ou.put(g);
for(j=0;j<N;j++)
 for(ou.put(h[j].c),g=V[j]->size(),ou.put(g),i=0;i<V[j]->size();i+=8, ou.put(g) ) for(g=k=0;k<8;++k) g|=((*V[j])[(i+k)%V[j]->size()]&1)<<(7-k);
for(i=0;i<256;b[i++]=-1); for(i=0;i<N;b[(unsigned char)h[i].c]=i++);
for(g=k=j=0;j<s;j++) for(i=0;i<V[b[p[j]]]->size();g|=((*V[b[p[j]]])[i++]&1)<<(7-k%8),++k%8?1:(ou.put(g),g=0));
if(k%8) ou.put(g); g=k%8,ou.put(g); cout<<"Кодированная часть файла в битах: "<<k<<endl;
ou.close(); for(j=0;j<N;j++) delete V[j]; delete [] p; delete [] h; return 0;
}
```

Адаптивное кодирование по Хаффману

Предлагается однопроходный алгоритм сжатия без передачи таблицы кодов.

При каждом сопоставлении символу кода, в следующем ходе вычислений этому же символу может быть сопоставлен другой код, то есть происходит адаптация алгоритма к поступающим для кодирования символам.

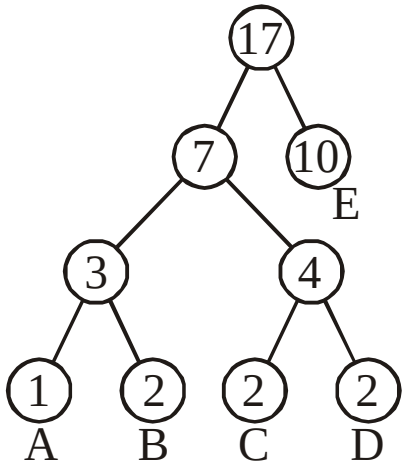
Дерево кодирования содержит вначале только один специальный символ, имеющий частоту 0. Он необходим для занесения в дерево новых символов.

Символы кодируются 8-битым кодом из расширенной таблицы ASCII.

При построении кода упорядочивается структура дерева. Листья дерева располагаются в порядке возрастания частот и затем в порядке возрастания стандартных кодов символов. Левые ветви помечаются 0, а правые — 1.

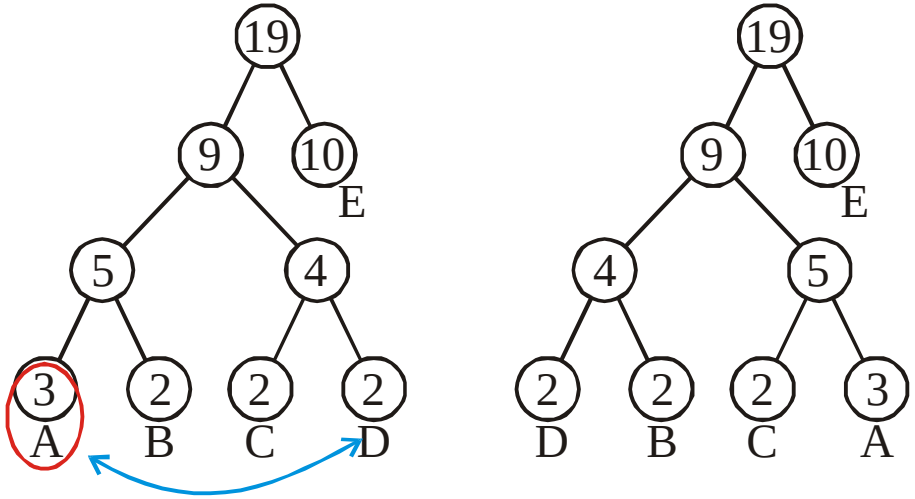
Бинарное дерево называется упорядоченным, если его узлы могут быть перечислены в порядке неубывания веса и узлы, имеющие общего родителя, должны находится рядом, на одном ярусе. Перечисление идёт по ярусам снизу-вверх и слева-направо в каждом ярусе.

Упорядоченное дерево

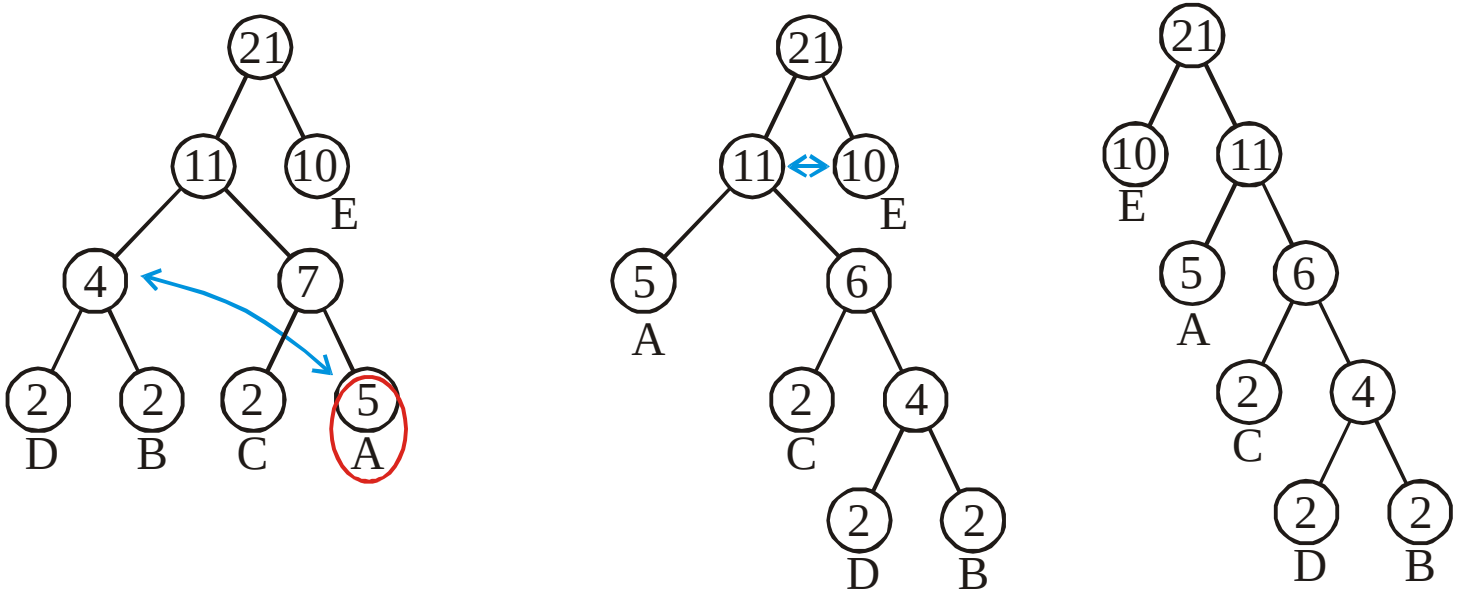


Если дерево кодирования упорядоченно, то при изменении веса существующего узла дерево не нужно целиком перестраивать — в нём достаточно лишь поменять местами два узла: узел, вес которого нарушил упорядоченность, и последний из следующих за ним узлов меньшего веса. После перемены мест узлов, необходимо пересчитать веса всех узлов-предков.

Если на дереве добавить ещё две буквы A, то узлы A и D меняются местами

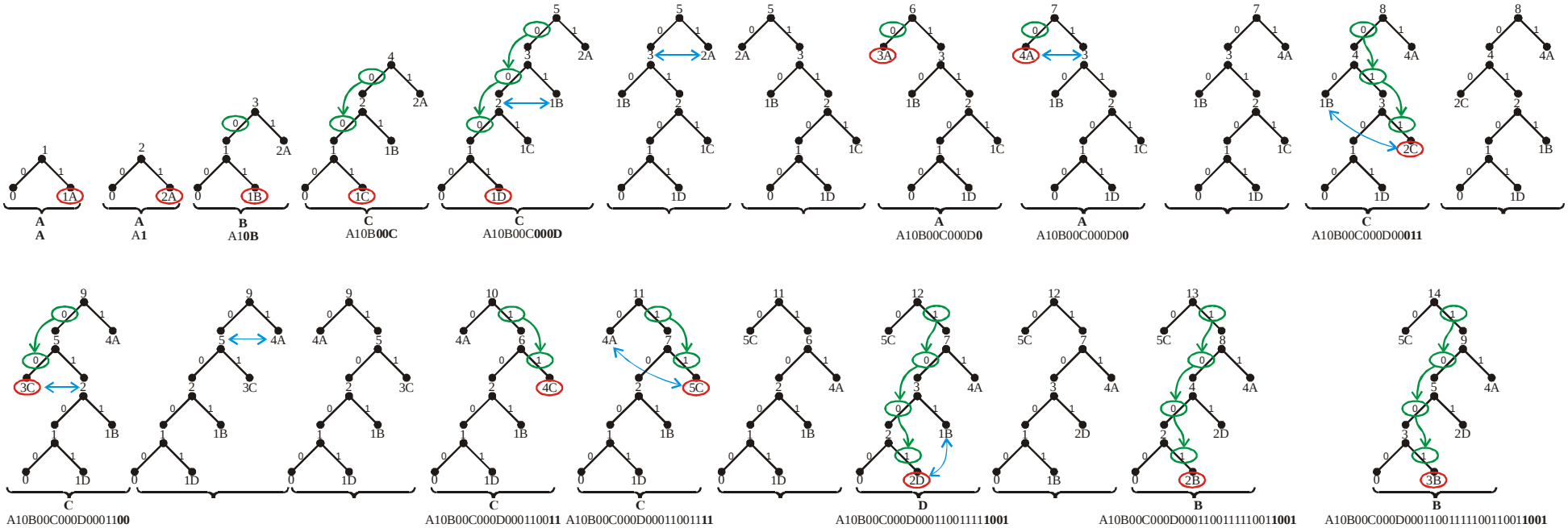


Если добавить ещё две буквы A, то меняются местами сначала узел A и узел, родительский узлов D и B, а затем узел E и узел-брат E.



Кодирование по адаптивному алгоритму Хаффмана

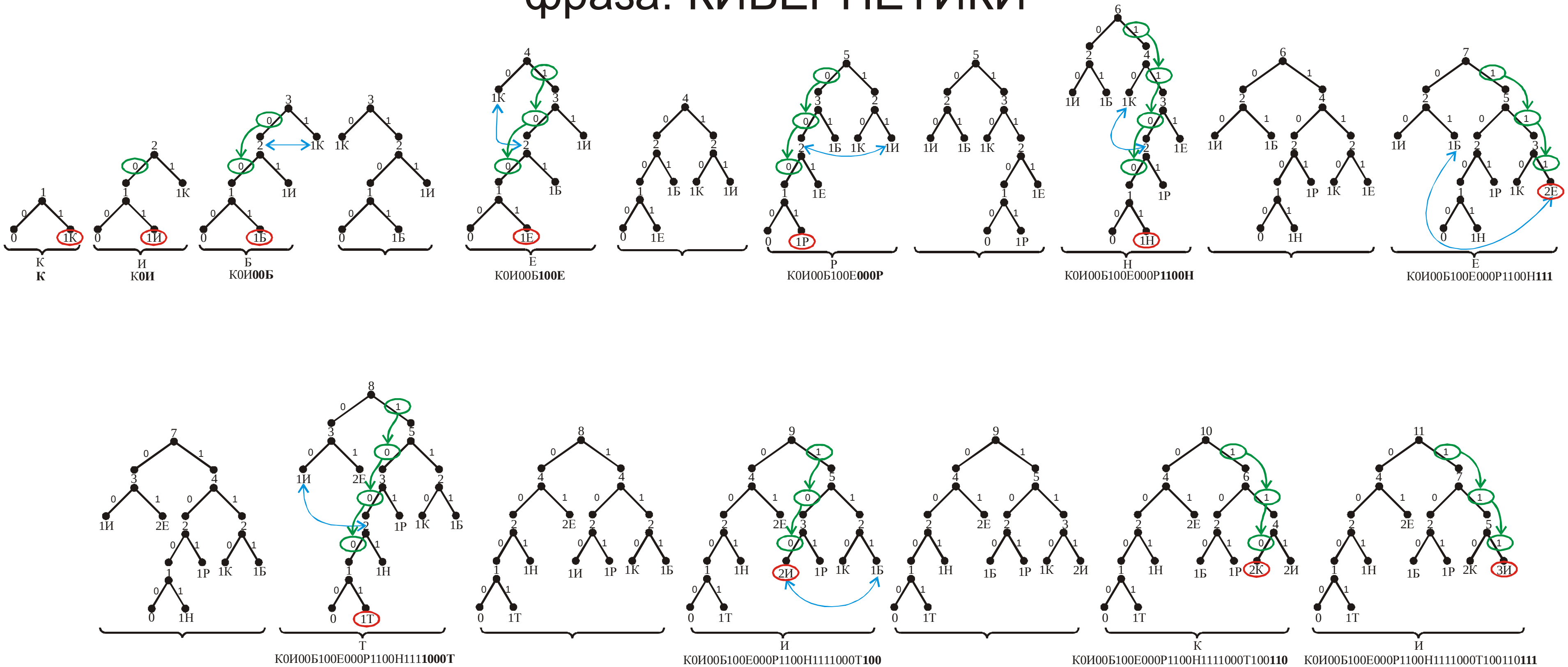
фраза: AABCDAACCCCDBB



code(AABCDAACCCCDBB)=A10B00C000D000110011111001100110011=01000001100100001000001000001100001000100000110011111001100110011001

L(code(AABCDAACCCCDBB))=62 бита

Кодирование по адаптивному алгоритму Хаффмана
фраза: КИБЕРНЕТИКИ

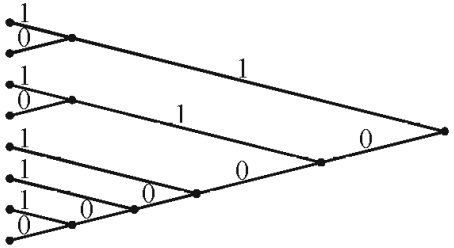


code(КИБЕРНЕТИКИ)=К0И00Б100Е000Р1100Н1111000Т100110111=100010100100010000010000001100100001010001001000011001000110111110001001001010011011$_{cp866}$=
=110010100110010000011000001100110001010001101000011001100110111110001101001010011011$_{cp1251}$

L(code(КИБЕРНЕТИКИ))=85 **бит**

Код Шеннона Фэно

Процедура сводится к тому, что поэтапно, начиная от вершины, массив символов источника делится на две группы с примерно равными суммарными вероятностями. Символам первой группы приписывается символ 1, символам второй группы приписывается кодовый символ 0. Формирование кодовой комбинации идёт справа налево В нашем примере $\overline{n} = 2{,}65$, что на много больше, чем для кода Хаффмана.

Символ источника	Вероятность $p(a_k)$	Кодовое дерево	Код	n_k	$n_k p(a_k)$
a_1	0,40		11	2	0,80
a_2	0,13		10	2	0,26
a_3	0,12		011	3	0,36
a_4	0,11		010	3	0,33
a_5	0,11		001	3	0,33
a_6	0,08		0001	4	0,32
a_7	0,03		00001	5	0,15
a_8	0,02		00000	5	0,10

Если помимо неравномерного распределения символов первичного источника между ними также присутствует статистическая связь (соседние символы или группы символов зависимы), то рассмотренные коды Хаффмана и Шеннона-Фэно окажутся менее эффективными. Так, например, для русского языка избыточность, обусловленная неравновероятностью символов $\rho_{\text{вер}} = 0{,}13$, а статистической связью — $\rho_{\text{св}} = 0{,}73$. В этом случае более эффективным окажется подход, при котором описанными выше методами кодируются не отдельные символы источника, а целые последовательности символов. Такое кодирование называют *кодированием с укрупнением алфавита*. При глубоких статистических связях длины кодируемых цепочек значительно увеличиваются и возрастает сложность алгоритмов кодирования и декодирования, а также время, затрачиваемое на обработку.

Программа кодера по Шеннона Фэно

```
void haf(int a,int b,char k,int u,int& z,haff* h)
{ if(a==b) return;
int r,s,j,w; float t;
h[--z].p=u; h[z].k=k+0x30;
for(s=0,j=a;j<=b;j++) h[j].p=z,s+=h[j].s;
for(t=s/2.,w=0,j=a-1;w<t;w+=h[++j].s);
r=j-1+(t-w+h[j].s>w-t);
for(j=a;j<=b;h[j].k=(j<=r)+0x30,j++);
u=z; haf(a,r,1,u,z,h); haf(r+1,b,0,u,z,h);
}

int main(int mn,char* nm[])
{ int b[256]={0},N,size,i,j,k,max,imax,s,r,z,w,v;
float t; haff* h; unsigned char *p,g;
ExeFile(nm[0]); if(mn!=3) cerr<<nm[0]<<" in.ext out.arh\n",exit(1);
ifstream in(nm[1],ios::binary); if(!in) cerr<<"file \""<<nm[1]<<"\" not open!\n",exit(1);
in.seekg(0,ios::end);size=in.tellg();in.seekg(0,ios::beg);
p=new unsigned char[size];if(!p) cerr<<"Error memory!\n",exit(1);
in.read((char*)p,size);in.close();
for(i=0;i<size;i++) b[p[i]]++;
for(N=i=0;i<256;i++) N+=!!b[i];
cout<<"Размер таблицы равен "<<N<<" символам."<<endl;
h=new haff[2*N+1];if(!h) cerr<<"Error memory!\n",exit(1);
for(j=0;j<2*N-1;j++) h[j].c=h[j].s=h[j].i=h[j].k=h[j].p=h[j].j=0;
for(j=i=0;i<256;i++) if(b[i]) h[j].s=b[i],h[j++].c=i;
vector<vector<char>*> V(N);
for(j=-N;j<0;imax>=0?h[imax].i=h[imax].j=j++:j++) for(imax=-1,max=k=0;k<N;k++) if(!h[k].i) if(h[k].s>max) max=h[k].s,imax=k;
for(s=j=0;j<N;s+=h[j++].s);
cout<<"Количество символов в исходном файле равно "<<s<<".\n";
z=2*N-2;
for(s=0,j=0;j<N;j++) h[j].p=z,s+=h[j].s;
for(t=s/2.,w=0,j=-1;w<t;w+=h[++j].s);
r=j-1+(t-w+h[j].s>w-t);
for(j=0;j<N;h[j++].k=(j<=r)+0x30);
haf(0,r,1,2*N-2,z,h); haf(r+1,N-1,0,2*N-2,z,h);
for(j=0;j<N;j++)
 { for(i=0,k=j;h[k].p;k=h[k].p,i++);
 V[j]=new vector<char>(i); if(!V[j]) cerr<<"Error memory!\n",exit(1);
 for(i=0,k=j;h[k].p;k=h[k].p,i++) (*V[j])[V[j]->size()-1-i]=h[k].k;
 }
for(j=0;j<N;j++,cout<<endl)
 for(k=(int)(unsigned char)h[j].c-32,k<0?cout<<"\'\\"<<k+32<<"\': ":cout<<"\'"<<char(k+32)<<"\': ",i=0;i<V[j]->size();i++) cout<<(*V[j])[i];
ofstream ou(nm[2],ios::binary); if(!ou) cerr<<"Error create the file \""<<nm[1]<<"\"\n",exit(1);
g=N-1; ou.put(g);
for(j=0;j<N;j++)
 for(ou.put(h[j].c),g=V[j]->size(),ou.put(g),i=0;i<V[j]->size();i+=8, ou.put(g) ) for(g=k=0;k<8;++k) g|=((*V[j])[(i+k)%V[j]->size()]&1)<<(7-k);
for(i=0;i<256;b[i++]=-1); for(i=0;i<N;b[(unsigned char)h[i].c]=i++);
for(g=k=j=0;j<s;j++) for(i=0;i<V[b[p[j]]]->size();g|=((*V[b[p[j]]])[i++]&1)<<(7-k%8),++k%8?1:(ou.put(g),g=0));
if(k%8) ou.put(g); g=k%8,ou.put(g); cout<<"Кодированная часть файла в битах: "<<k<<endl;
ou.close(); for(j=0;j<N;j++) delete V[j]; delete [] p; delete [] h; return 0;
}
```

Программа декодера файлов, закодированных по Хаффману или Шеннона Фэно

```
int main(int mn,char* nm[])
{ int o=0,N,size,i,j,k,l=0,r,e,t,ii;
unsigned char *p,q; short *c,v; char* a;
ExeFile(nm[0]); if(mn!=3) cerr<<nm[0]<<" in.arh out.ext\n",exit(1);
ifstream in(nm[1],ios::binary); if(!in) cerr<<"file \""<<nm[1]<<"\" not open!\n",exit(1);
in.seekg(0,ios::end);size=in.tellg();in.seekg(0,ios::beg);
p=new unsigned char[size];if(!p) cerr<<"Error memory!\n",exit(1);
in.read((char*)p,size);in.close();
N=(unsigned char)p[l++]+1;
a=new char[N]; c=new short[2*N-2];
for(i=0;i<2*N-2;c[i++]=-1);
cout<<"Размер таблицы равен "<<N<<" символам."<<endl;
for(j=0;j<N;j++,cout<<endl)
 {
 a[j]=p[l++],r=(unsigned char)p[l++];
 k=(unsigned char)a[j]-32,k<0?cout<<"\'\\"<<k+32<<"\': ":cout<<"\'"<<char(k+32)<<"\': ";
 k=2*N-2; // начинаем с последней ячейки
 for(q=i=0;i<r;i++,q>>=1)
  {
  if(!q) t=(unsigned char)p[l++],q=128;
  v=!!(t&q),cout<<v;
  if(c[(k-N)*2+v]<0)
   if(i==r-1)
    c[(k-N)*2+v]=j;
   else
    {
    for(e=k-1;e>=N;--e) if(c[(e-N)*2+v]<0 && c[(e-N)*2+!v]<0) break;
    if(c[(e-N)*2+!v]<0)
     k=c[(k-N)*2+v]=e;
    else
     {
     for(;e>=N;--e) if(c[(e-N)*2+!v]<0 && c[(e-N)*2+v]<0) break;
     k=c[(k-N)*2+v]=e;
     }
    }
  else k=c[(k-N)*2+v];//,cout<<"!!!->!!!"<<endl;
  }
 }
cout<<"Заголовок "<<l<<" байт"<<endl;
size=(size-l-1-!!p[size-1])*8+p[size-1];
cout<<"Сжатый текст содержал "<<size<<" бит."<<endl;
ofstream ou(nm[2],ios::binary); if(!ou) cerr<<"Error create the file \""<<nm[1]<<"\"\n",exit(1);
for(q=128,v=N-2,j=0;j<size;j++)
 { if(!q) q=128,l++;
 k=!!(p[l]&q),q>>=1;
 if(c[2*v+k]<N) ou.put(a[c[2*v+k]]),v=N-2;
 else v=c[2*v+k]-N;
 }
ou.close(); delete [] p, delete [] c, delete [] a; return 0;
                                                                                    }
```

Арифметическое кодирование

Алгоритм кодирования Хаффмана не может передавать на каждый символ сообщения менее одного бита информации. Схема кодирования, при которой некоторые символы кодируются менее, чем одним битом является арифметическое кодирование.

По исходному распределению вероятностей при выбранной для кодирования дискретной случайной величине строится таблица, состоящая из пересекающихся только в граничных точках отрезков для каждого из значений этой дискретной случайной величины.; объединение этих отрезков должно образовывать отрезок $[0, 1]$, а их длины должны быть пропорциональными вероятностям соответствующих значений дискретных случайных величин. Алгоритм кодирования заключается в построении отрезка, однозначно определяющего данную последовательность значений дискретных случайных величин. Затем для построенного отрезка находится число, принадлежащее его внутренней части и равное целому числу, делённому на минимально возможную положительную целую степень двойки. Это число и будет кодом для рассматриваемой последовательности. Все возможные конкретные коды — это числа строго большие нуля и строго меньшие одного, поэтому можно отбрасывать лидирующий ноль и десятичную запятую, но нужен ещё специальный код-маркер, сигнализирующий о конце сообщения. Отрезки строятся так. Если имеется отрезок для сообщения длины $n-1$, то для построения отрезка для сообщения длины n, разбиваем его на столько же частей, сколько значений имеет рассматриваемая дискретная случайная величина. Это разбиение делается совершенно также как и самое первое (с сохранением порядка). Затем выбирается из полученных отрезков тот, который соответствует заданной конкретной последовательности длины n.

Принципиальное отличие этого кодирования от рассмотренных ранее методов в его непрерывности, то есть в ненужности блокирования. Эффективность арифметического кодирования растёт с ростом длины сжимаемого сообщения (для кодирования Хаффмана и Шеннона-Фано этого не происходит). Недостатком арифметического кодирования является большие требования к вычислительным ресурсам.

При сжатии заданных данных, например, из файла все рассмотренные методы требуют двух проходов. Первый для сбора частот символов, используемых как приближённые значения вероятностей символов, и второй для собственно сжатия.

Пример. Пусть дискретная случайная величина X может принимать только два значения 0 и 1 с вероятностями $\frac{2}{3}$ и $\frac{1}{3}$ соответственно. Сопоставим значению 0 отрезок $\left[0,\frac{2}{3}\right]$, а 1 — $\left[\frac{2}{3},1\right]$. Тогда $\dim\left(\overrightarrow{X}\right)=3$, $H(X)=H\left(\overrightarrow{X}\right)/3=\log_2 3-\frac{2}{3}\approx 0{,}9183\,\frac{\text{бит}}{\text{симв.}}$. Таблица построения кодов:

Интервалы и коды			Вероятность	Код Хаффмена
		$111[\frac{26}{27},1] \ni \frac{31}{32}=0.11111$	$^1/_{27}$	0000
	$11[\frac{8}{9},1]$	$110[\frac{8}{9},\frac{26}{27}] \ni \frac{15}{16}=0.1111$	$^2/_{27}$	0001
		$101[\frac{22}{27},\frac{8}{9}] \ni \frac{7}{8}=0.111$	$^2/_{27}$	010
$1[\frac{2}{3},1]$	$10[\frac{2}{3},\frac{8}{9}]$	$100[\frac{2}{3},\frac{22}{27}] \ni \frac{3}{4}=0.11$	$^4/_{27}$	001
		$011[\frac{16}{27},\frac{2}{3}] \ni \frac{5}{8}=0.101$	$^2/_{27}$	011
	$01[\frac{4}{9},\frac{2}{3}]$	$010[\frac{4}{9},\frac{16}{27}] \ni \frac{1}{2}=0.1$	$^4/_{27}$	100
		$001[\frac{8}{27},\frac{4}{9}] \ni \frac{3}{8}=0.011$	$^4/_{27}$	101
$0[0,\frac{2}{3}]$	$00[0,\frac{4}{9}]$	$000[0,\frac{8}{27}] \ni \frac{1}{4}=0.01$	$^8/_{27}$	11.

$$M\left(L_1\left(\overrightarrow{X}\right)\right)=\frac{65}{81}\approx 0,8025\,\frac{\text{бит}}{\text{симв.}}\text{ (арифметическое),}$$

$$M\left(L_1\left(\overrightarrow{X}\right)\right)=\frac{76}{81}\approx 0,9383\,\frac{\text{бит}}{\text{симв.}}\text{ (блочный Хаффмана),}$$

$$M\left(L_1\left(\overrightarrow{X}\right)\right)=M\left(L(X)\right)=1\,\frac{\text{бит}}{\text{симв.}}\text{ (Хаффмана).}$$

Среднее количество бит на единицу сообщения для арифметического кодирования получилось меньше, чем энтропия. Это связано с тем, что в рассмотренной простейшей схеме кодирования, не описан код-маркер конца сообщения, введение которого неминуемо сделает это среднее количество бит большим энтропии.

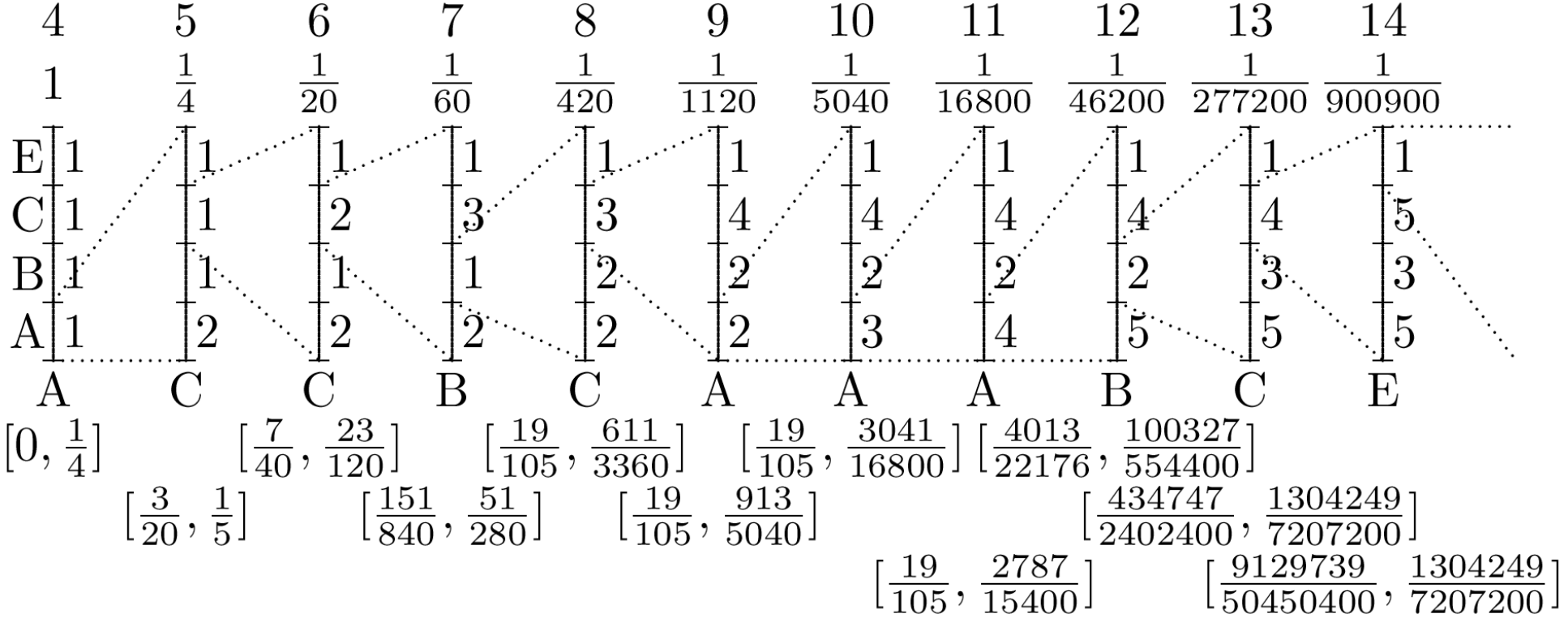
Получение исходного сообщения из его арифметического происходит по следующему алгоритму.

Шаг 1. В таблице для кодирования значений дискретной случайной величины определяется интервал, содержащий текущий код, — по этому интервалу однозначно определяется один символ исходного сообщения. Если этот символ — маркер конца сообщения, то конец.

Шаг 2. Из текущего кода вычитается нижняя граница содержащего его интервала. Полученное число считывается новым текущим значением кода. Переход к шагу 1.

Адаптивное арифметическое кодирование

Каждому символу сопоставляется его вес: вначале он для всех равен 1. Все символы располагаются в естественном порядке, например, по возрастанию. Вероятность каждого символа устанавливается равной его весу, делённому на суммарный вес всех символов. После получения очередного символа и постройки интервала для него, вес этого символа увеличивается на 1.



Программа арифметического кодера (без описания функций работы со строками)

```
int main(int mn,char* nm[])
{ int i,j,k,size,b[256],y[256],sz,s; unsigned char *p,*r; if(mn==2||mn==3); else cerr<<"drob.exe in.ext <tabl_kol.txt>\n",exit(1);
ifstream in(nm[1],ios::binary); if(!in) cerr<<"Not open \""<<nm[1]<<"\" file!\n",exit(1);
in.seekg(0,ios::end); size=in.tellg(); in.seekg(0,ios::beg); p=new unsigned char[size]; if(!p) cerr<<"No memory!\n",exit(1);
in.read((char*)p,size); in.close(); memset(b,0,sizeof(int)*256); memset(y,-1,sizeof(int)*256);
if(mn==3) { char c; ifstream in(nm[2]); for(sz=0;!in.eof();sz++) in>>c>>i; in.close(); }
else { for(i=0;i<size;i++) b[p[i]]++; for(sz=i=0;i<256;sz+=!!b[i++]); }
cout<<sz<<endl; r=new unsigned char[sz];
if(mn==3)
 { int y; ifstream in(nm[2]); for(s=i=0;i<sz;i++) in>>r[i]>>y,b[r[i]]=y; in.close(); for(i=0;i<sz;i++) cout<<unsigned(r[i])<<": "<<b[r[i]]<<endl; }
else
 { for(j=i=0;i<256;i++) if(b[i]) r[j++]=i; for(i=0;i<sz;i++) cout<<unsigned(r[i])<<": "<<b[r[i]]<<endl;
 for(s=i=0;i<sz;i++) s+=b[r[i]]; cout<<"s="<<s<<" size="<<size<<endl; }
vector<string> w(sz+1);
string z,lg,rg,zn,lg2,qq; { stringstream out; out<<size; z=out.str(); }
for(w[0]="0",s=0,i=1;i<=sz;i++) { s+=b[r[i-1]]; stringstream out;  out<<s; w[i]=out.str(); }
if(mn==3) { stringstream out;  out<<w[sz]; z=out.str(); }
for(i=0;i<sz;i++) y[r[i]]=i;
for(i=0;i<sz;i++) (r[i]<32?cout<<"\'\\"<<int(r[i]):cout<<"\'"<<r[i]),cout<<"\': "<<w[i]<<"/"<<z<<"..."<<w[i+1]<<"/"<<z<<endl;
lg=w[y[p[0]]]; rg=w[y[p[0]]+1]; zn=z;
r[y[p[0]]]<32?cout<<"\'\\"<<int(r[y[p[0]]]):cout<<"\'"<<r[y[p[0]]]; cout<<"\': "<<lg<<"/"<<zn<<"..."<<rg<<"/"<<zn<<endl;
for(i=1;i<size;i++)
 { lg2=add(mulp(lg,z),mulp(w[y[p[i]]],sub(rg,lg))); zn=mulp(zn,z); rg=add(mulp(lg,z),mulp(w[y[p[i]]+1],sub(rg,lg))); lg=lg2;
 r[y[p[i]]]<32?cout<<"\'\\"<<int(r[y[p[i]]]):cout<<"\'"<<r[y[p[i]]]; cout<<"\': "<<lg<<"/"<<zn<<"..."<<rg<<"/"<<zn<<endl;
 }
string t,tlg,trg,tzn,tch,ch,zz,s1,s2;
for(i=0;i<size;i++) cout<<p[i]; cout<<endl; for(i=0;i<sz;i++) cout<<r[i]; cout<<endl;
for(i=0;i<sz;i++) (r[i]<32?cout<<"\'\\"<<int(r[i]):cout<<"\'"<<r[i]),cout<<"\': "<<w[i]<<"/"<<z<<"..."<<w[i+1]<<"/"<<z<<endl;
vector<int> W(256,-1); for(i=0;i<sz;i++) W[r[i]]=1; { stringstream out; out<<sz; z=out.str(); }
for(lg2=z,t="1",tlg="1",k=0;k<W.size();k++) if(W[k]>0) k<32?cout<<"\'\\"<<int(k):cout<<"\'"<<char(k),cout<<"\': "<<W[k]<<" ";
for(s=i=0;r[i]!=p[0];s+=W[r[i++]]); { stringstream out; out<<s; lg=out.str(); }
rg=addp(lg,mulp(tlg,t));
(p[0]<32?cout<<"\'\\"<<int(p[0]):cout<<"\'"<<p[0]),cout<<"\'=> "<<lg<<"/"<<z<<"..."<<rg<<"/"<<z<<endl;
for(j=1;j<size;j++)
 { lg2=addp(lg2,"1"); W[p[j-1]]++;
 for(k=0;k<W.size();k++) if(W[k]>0) k<32?cout<<"\'\\"<<int(k):cout<<"\'"<<char(k),cout<<"\': "<<W[k]<<" ";
 for(s=i=0;r[i]!=p[j];s+=W[r[i++]]);
 lg=mulp(lg,lg2); rg=mulp(rg,lg2); z=mulp(z,lg2); { stringstream out; out<<s; s1=out.str(); }
 s+=W[r[i]]; { stringstream out; out<<s; s2=out.str(); }
 tlg=addp(lg,mulp(s1,div(subp(rg,lg),lg2,ch))); trg=addp(lg,mulp(s2,div(subp(rg,lg),lg2,ch)));
 lg=tlg; rg=trg; (p[j]<32?cout<<"\'\\"<<int(p[j]):cout<<"\'"<<p[j]),cout<<"\'=> "<<lg<<"/"<<z<<"..."<<rg<<"/"<<z<<endl;
 }
for(zn=z,ch="0",zz="2";;zz=mulp(zz,"2"))
 { tlg=mulp(lg,zz);trg=mulp(rg,zz);tzn=mulp(zn,zz);
 ch=divp(tlg,zn,t); tch=mulp(ch,zn);
 cout<<lg<<"/"<<zn<<" < "<<ch<<"/"<<zz<<" < "<<rg<<"/"<<zn<<" - test"<<endl;
 cout<<tlg<<"/"<<tzn<<" < "<<tch<<"/"<<tzn<<" < "<<trg<<"/"<<tzn<<" - общий знаменатель"<<endl;
 for(;ltp(tch,trg);ch=addp(ch,"1"),tch=mulp(ch,zn)  );
 ch=subp(ch,"1"),tch=mulp(ch,zn); cout<<tlg<<"/"<<tzn<<" < "<<tch<<"/"<<tzn<<" < "<<trg<<"/"<<tzn<<" - итог test"<<endl;
 cout<<lg<<"/"<<zn<<" < "<<ch<<"/"<<zz<<" < "<<rg<<"/"<<zn<<" - итог test"<<endl; if(!ltp(tch,tlg)&&!eq(tch,tlg)) break;
 }
cout<<"ch/zz="<<ch<<"/"<<zz<<endl; for(t="",lg2="";!eq(zz,"1");) zz=divp(zz,"2",t),ch=divp(ch,"2",t),lg2=t+lg2;
cout<<"-------------------------------------=>"<<lg2<<endl; delete [] p; delete [] r;return 0;
}
```

Подстановочные или словарно-ориентированные алгоритмы сжатия информации.

Методы Лемпела-Зива

Методы Шеннона-Фано, Хаффмана и арифметического кодирования называют статистическими. Далее рассматриваются словарные алгоритмы, которые носят менее математически обоснованный, но более практичный характер.

Алгоритм LZ77

Алгоритм LZ77 был опубликован в 1977 г. Разработан израильскими математиками Якобом Зивом и Авраамом Лемпелом.

Основная идея LZ77 состоит в том, что второе и последующие вхождения некоторой строки символов в сообщении заменяются ссылками на её первое вхождение.

Алгоритм LZ77 использует уже просмотренную часть сообщения как словарь. Чтобы добиться сжатия, он пытается заменить очередной фрагмент сообщения на указатель в содержимое словаря.

Алгоритм LZ77 использует «скользящее» по сообщению окно, разделённое на две неравные части. Первая, большая по размеру, включает уже просмотренную часть сообщения. Вторая, намного меньшая, является буфером, содержащим ещё незакодированные символы входного потока.

Обычно размер окна составляет несколько килобайт, а размер буфера — не более 100 байт. Алгоритм пытается найти в словаре (большей части окна) фрагмент, совпадающий с содержимым буфера.

Алгоритм LZ77 выдаёт коды, состоящие из трёх элементов:

– смещение в словаре относительно его начала подстроки, совпадающей с началом содержимого буфера;

– длина этой подстроки;

– первый символ буфера, следующий за подстрокой.

Пример. Размер окна — 20 символов, словаря — 12 символов, а буфера — 8. Кодируется сообщение «ПРОГРАММНЫЕ ПРОДУКТЫ ФИРМЫ MICROSOFT». Пусть словарь уже заполнен. Тогда он содержит строку «ПРОГРАММНЫЕ », а буфер — строку «ПРОДУКТЫ». Просматривая словарь, алгоритм обнаружит, что совпадающей подстрокой будет «ПРО», в словаре она расположена со смещением 0 и имеет длину 3 символа, а следующим символом в буфере является «Д». Таким образом, выходным кодом будет тройка $\langle 0,3,'Д'\rangle$. После этого алгоритм сдвигает влево всё содержимое окна на длину совпадающей подстроки +1 и одновременно считывает столько же символов из входного потока в буфер. Получаем в словаре строку «РАММНЫЕ ПРОД», в буфере — «УКТЫ ФИР». В данной ситуации совпадающей подстроки обнаружить не удаться и алгоритм выдаст код $\langle 0,0,'У'\rangle$, после чего сдвинет окно влево на один символ. Затем словарь будет содержать «АММНЫЕ ПРОДУ», а буфер «КТЫ ФИРМ». И т. д.

Декодирование кодов LZ77 проще их получения, так как не нужно осуществлять поиск в словаре.

Недостатки LZ77:

1) с ростом размеров словаря скорость работы алгоритма –кодера пропорционально замедляется;

2) кодирование одиночных символов очень неэффективно.

Пример. Закодировать по алгоритму LZ77 строку «КРАСНАЯ КРАСКА».

СЛОВАРЬ(8)	БУФЕР(5)	**КОД**
«........»	«КРАСН»	<0,0‘К’>
«.......К»	«РАСНА»	<0,0‘Р’>
«......КР»	«АСНАЯ»	<0,0‘А’>
«.....КРА»	«СНАЯ »	<0,0‘С’>
«....КРАС»	«НАЯ К»	<0,0‘Н’>
«...КРАСН»	«АЯ КР»	<5,1‘Я’>
«.КРАСНАЯ»	« КРАС»	<0,0‘ ’>
«КРАСНАЯ »	«КРАСК»	<0,4‘К’>
«АЯ КРАСК»	«А....»	<0,0‘А’>

В последней строчке, буква «А» берётся не из словаря, так как она последняя.

Длина кода вычисляется следующим образом: длина подстроки не может быть больше размера буфера, а смещение не может быть больше размера словаря минус 1. Следовательно, длина двоичного кода смещения будет округлённым в большую сторону $\log_2(\text{размер словаря})$, а длина двоичного кода дл длины подстроки будет округлённым в большую сторону $\log_2(\text{размер буфера}+1)$. А символ кодируется 8 битами (например ASCII+)

В последнем примере длина полученного кода равна $9\cdot(3+3+8)=126$ бит, против 112 бит исходной длины строки.

Алгоритм LZSS

В 1982 году Сторером (Storer) и Шиманским (Szimanski) на базе LZ77 был разработан алгоритм LZSS.

Код, выдаваемый алгоритмом LZSS, начинается с однобитового префикса, различающего собственно код от незакодированного символа. Код состоит из пары: смещение и длина, такими же как и для LZ77. В LZSS окно сдвигается ровно на длину найденной подстроки или на 1, если не найдено вхождение подстроки из буфера в словарь. Длина подстроки в LZSS всегда больше нуля, поэтому длина двоичного кода для длины подстроки — это округлённый до большего целого двоичный логарифм от длины буфера.

Пример. Закодировать по алгоритму LZSS строку «КРАСНАЯ КРАСКА».

СЛОВАРЬ(8)	БУФЕР(5)	КОД	ДЛИНА КОДА
«........»	«КРАСН»	0‘К’	9
«.......К»	«РАСНА»	0‘Р’	9
«......КР»	«АСНАЯ»	0‘А’	9
«.....КРА»	«СНАЯ »	0‘С’	9
«....КРАС»	«НАЯ К»	0‘Н’	9
«...КРАСН»	«АЯ КР»	1<5,1>	7
«..КРАСНА»	«Я КРА»	0‘Я’	9
«.КРАСНАЯ»	« КРАС»	0‘ ’	9
«КРАСНАЯ »	«КРАСК»	1<0,4>	7
«НАЯ КРАС»	«КА...»	1<4,1>	7
«АЯ КРАСК»	«А....»	1<0,1>	7

Здесь длина полученного кода равна $7 \cdot 9 + 4 \cdot 7 = 91$ бит.

LS77 и LZSS обладают следующими очевидными недостатками:

1) невозможность кодирования подстрок, отстоящих друг от друга на расстоянии, большем длины словаря;

2) длина подстроки, которую можно закодировать, ограничена размером буфера.

Если механически чрезмерно увеличивать размеры словаря и буфера, то это приведёт к снижению эффективности кодирования, так как с ростом этих величин будут расти и длины кодов для смещения и длины, что сделает коды для коротких подстрок недопустимо большими. Кроме того, резко увеличится время работы алгоритма-кодера.

Алгоритм LZ78

В 1978 году авторами LZ77 был разработан алгоритм LZ78, лишённый названных недостатков.

Алгоритм LZ78 не использует «скользящее» окно, он хранит словарь из уже просмотренных фраз. При старте алгоритма этот словарь содержит только одну пустую строку (строку длины нуль). Алгоритм считывает символы сообщения до тех пор, пока накапливаемая подстрока входит целиком в одну из фраз словаря. Как только эта строка перестанет соответствовать хотя бы одной фразе словаря, алгоритм генерирует код, состоящий из индекса строки в словаре, которая до последнего введённого символа содержала входную строку, и символа, нарушившего совпадение. Затем в словарь добавляется введённая подстрока. Если словарь уже заполнен, то из него предварительно удаляют менее всех используемую в сравнениях фразу.

Ключевым для размера получаемых кодов является размер словаря во фразах, потому что каждый код при кодировании по методу LZ78 содержит номер фразы в словаре. Из последнего следует, что эти коды имеют постоянную длину, равную округлённому в большую сторону двоичному логарифму размера словаря +8 (это количество бит в байт-коде расширенного ASCII).

Пример. Закодировать по алгоритму LZ78 строку «КРАСНАЯ КРАСКА», используя словарь длиной 16 фраз.

ВХОДНАЯ ФРАЗА (В СЛОВАРЬ)	КОД	ПОЗИЦИЯ СЛОВАРЯ
«»		0
«К»	<0‘К’>	1
«Р»	<0‘Р’>	2
«А»	<0‘А’>	3
«С»	<0‘С’>	4
«Н»	<0‘Н’>	5
«АЯ»	<3‘Я’>	6
« »	<0‘ ’>	7
«КР»	<1‘Р’>	8
«АС»	<3‘С’>	9
«КА»	<1‘А’>	10

Указатель на любую фразу такого словаря — это число от 0 до 15, для его кодирования достаточно четырёх бит.

В последнем примере длина полученного кода равна $10 \cdot (4+8) = 120$ битам.

Алгоритм Лемпела-Зива для двоичных символов

В технике экономного кодирования более типичной является ситуация, когда о некотором избыточном источнике известен лишь его алфавит, но не известна его статистика (распределение вероятностей последовательностей символов). Так, например, может стоять задача разработки некоторого универсального сжимающего двоичного префиксного кода для передачи текста на различных языках, заданных каждый своим алфавитом, или для сжатия данных различного рода.

Основная идея этого алгоритма заключается в том, что последовательность символов источника разбивается на максимально короткие различимые цепочки, которые не встречались раньше, а затем эти цепочки кодируются равномерным кодом. Например, если источник выдаёт двоичную последовательность из 18 символов: 101100110001111000, то она будет разбита на такие цепочки: 1, 0, 11, 00, 110, 001, 111, 000. При таком разбиении все префиксы конкретной цепочки могут находиться лишь слева, и две цепочки с одинаковым префиксом всегда отличаются в последнем символе (бите). В каждой цепочке кодируется её префикс равномерным двоичным кодом и один бит используется для кодирования последнего (справа) символа цепочки. Если $C(n)$ означает число различных цепочек разбиения для данной последовательности из n символов (в нашем примере $C(n)=8$), то требуется $\log_2 C(n)$ бит, чтобы закодировать номер префикса к данной цепочке (в нашем примере 3 бита) и ещё один бит для описания последнего элемента этой цепочки. Для приведённой выше цепочки и её разбиения получаем следующий равномерный код: $(000,1)$, $(000,0)$, $(001,1)$, $(010,0)$, $(011,0)$, $(100,1)$, $(011,1)$, $(100,0)$. Декодирование такого кода однозначное. В нашем примере кодовые комбинации содержат 32 символа (4×8) вместо исходных 18 символов, то есть получилось растяжение вместо сжатия. Однако для длинных последовательностей сообщений эффективность алгоритма растёт, и при $n\to\infty$ сжатие сообщений приближается к своему пределу, поскольку доказано, что для любого стационарного источника сообщений при использовании двоичного кода Зива-Лемпела среднее число кодовых символов на один символ источника стремится к величине энтропии.

Алгоритм LZW

В 1984 году Уэлчем (Welch) был создан алгоритм LZW путём модификации LZ78.

Пошаговое описание алгоритма-кодера.

Шаг 1. Инициализация словаря всеми возможными односимвольными фразами (обычно 256 символами расширенного ASCII). Инициализация входной фразы w первым символом сообщения.

Шаг 2. Считать очередной символ K из кодируемого сообщения.

Шаг 3. Если КОНЕЦ_СООБЩЕНИЯ

Выдать код для w

Конец

Если фраза wK уже есть в словаре

Присвоить входной фразе значение wK

Перейти к Шагу 2

Иначе

Выдать код w

Добавить wK в словарь

Присвоить входной фразе значение K

Перейти к Шагу 2

Как и в случае с LZ78 для LZW ключевым для размера получаемых кодов является размер словаря во фразах: LZW-коды имеют постоянную длину, равную округлённую в большую сторону двоичному логарифму размера словаря.

Пример. Закодировать строку «КРАСНАЯ КРАСКА». Размер словаря — 500 фраз.

ВХОДНАЯ ФРАЗА, wK (В СЛОВАРЬ)	КОД для W	**ПОЗИЦИЯ СЛОВАРЯ**
ASCII+		**0-255**
«КР»	**0‘К’**	**256**
«РА»	**0‘Р’**	**257**
«АС»	**0‘А’**	**258**
«СН»	**0‘С’**	**259**
«НА»	**0‘Н’**	**260**
«АЯ»	**0‘А’**	**261**
«Я »	**0‘Я’**	**262**
« К»	**0‘ ’**	**263**
«КРА»	**<256>**	**264**
«АСК»	**<258>**	**265**
«КА»	**0‘К’**	**266**
«А»	**0‘А’**	

В этом примере длина полученного кода равна $12 \cdot 9 = 108$ битам.

При переполнении словаря, то есть когда необходимо внести новую фразу в полностью заполненный словарь, из него удаляют либо наиболее редко используемую фразу, либо все фразы, отличающиеся от одиночного символа. Для LZ78 возможна и полна очистка словаря.

Эффективность кодирования источника (сжатия данных) оценивается через его избыточность следующим соотношением: $\eta_и = 1 - \rho_и$.

Для русского языка $\rho_и \approx 0{,}75$ и, следовательно, максимальная эффективность экономного кодирования $\eta_и \approx 0{,}25$.

Сжатие информации с потерями

Когда сжимается информация, используемая лишь для качественной оценки
Используется в основном для трёх видов данных: полноцветная графика, звук и видеоинформация.
Два этапа: исходная информация приводится (с потерями) к виду в котором её можно эффективному сжимать алгоритмами сжатия без потерь.
Идея сжатия графической информации: глаз человека воспринимает полностью только информацию о яркости и в меньшей степени о цвете и насыщенности, что позволяет отбрасывать часть информации о двух последних атрибутах.
Сжатие видеоинформации основано на небольшом изменении изображении при переходе к соседнему кадру. То есть сжатая информация представляет собой запись некоторых базовых кадров и последовательности изменений в них. Далее информация может быть сжата другим методом.

Аудиоинформация может быть сжата за счёт частотных изменений в спектре сигнала или, например, за счёт предсказания поведения аудиосигнала.