**Генерация булеана конечного множества.**

Определение 1. Пусть дано некоторое множество **A**. Булеаном множества **А** называется множество, состоящее из всех подмножеств **А**.

Как получить булеан множества **А**? Пусть множество **А** состоит из n элементов. Тогда каждое его подмножество **B** можно задать **характеристической последовательностью** из n нулей и единиц. Единица на i-м месте последовательности означает, что i-й элемент множества **А** входит в подмножество **В**, ноль – что не входит. Таким образом, вместо генерации всех подмножеств данного множества можно генерировать всевозможные последовательности из нулей и единиц длины n.

**Генерация всех последовательностей длины n из нулей и единиц в лексикографическом порядке.**

Определим лексикографический порядок на множестве последовательностей. Будем считать, что последовательность **a** предшествует последовательности **b**, если первые i членов у этих последовательностей одинаковы, а i+1 – й член у последовательности **a** меньше.

Последовательности из нулей и единиц можно считать двоичными числами. Тогда последовательность **a** предшествует **b**, если ее двоичное число меньше. Сгенерировать все n – разрядные двоичные числа очень просто. Нужно начать с числа 000…0 и прибавлять по единице до тех пор, пока не получим число 111…1. Для того, чтобы прибавить к двоичному числу единицу, необходимо перебирать цифры данного числа справа налево до тех пор, пока в каком-нибудь разряде не встретим 0. Затем этот разряд нужно сделать равным 1, а все разряды, лежащие справа от него, сделать нулями.

В качестве примера приведем программу, которая генерирует все последовательности из нулей и единиц длины 4.

#include<stdio.h>

#include<conio.h>

#define n 4

int x[n];

main(){

int i,k;

for(i=0;i<n;i++) // Формируем начальную последовательность.

x[i]=0;

while(1){

for(i=0;i<n;i++) // Печатаем очередную последовательность

printf(“%d “,x[i]);

printf(“\n”);

for(k=n-1;k>=0 && x[k]==1;k--) // Ищем первый справа ноль

x[k]=0; // Попутно единицы справа от первого нуля

//превращаем в нули

if(k==-1) break; // Если ни одного нуля не нашли,

// то все последовательности уже сгенерированы.

else x[k]=1; // Иначе на место найденного нуля ставим 1.

}

getch();

return 0;

}

**Генерация всех последовательностей длины n из чисел 0,1…k-1 в лексикографическом порядке.**

Аналогично можно генерировать все последовательности длины n из чисел 0,1,…k-1. Начинаем с последовательности 000…0. Печатаем очередную последовательность. Строим по ней следующую. Для этого, двигаясь с конца последовательности, находим элемент, который меньше k-1. Увеличиваем его на 1, а все элементы, расположенные правее его, делаем нулями. Печатаем следующую последовательность и т.д. Процесс продолжается до тех пор, пока все элементы последовательности не станут равны k-1.

Упр. Видоизменить программу из предыдущего параграфа, чтобы она печатала всепоследовательности длины n из чисел 0,1…k-1.

**Коды Грея длины n.**

Часто возникает необходимость генерировать все подмножества данного множества таким образом, чтобы следующее подмножество получалось из предыдущего добавлением или удалением одного элемента. В терминах характеристических последовательностей это означает, что каждая следующая характеристическая последовательность отличается от предыдущей в одном разряде. Набор всех последовательностей из нулей и единиц длины n, удовлетворяющих этому свойству, называется бинарным кодом Грея. Наиболее часто применяют рефлексивный бинарный код Грея.

Приведем рефлексивный бинарный код Грея для n=1,2,3.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N=1 | | | n=2 | | | n=3 | | |
| № | Код Грея | у | № | Код Грея | у | № | Код Грея, | у |
| 1 | 0 | 0 | 1 | 00 | 0 | 1 | 000 | 0 |
| 2 | 1 |  | 2 | 01 | 1 | 2 | 001 | 1 |
|  |  |  | 3 | 11 | 0 | 3 | 011 | 0 |
|  |  |  | 4 | 10 |  | 4 | 010 | 2 |
|  |  |  |  |  |  | 5 | 110 | 0 |
|  |  |  |  |  |  | 6 | 111 | 1 |
|  |  |  |  |  |  | 7 | 101 | 0 |
|  |  |  |  |  |  | 8 | 100 |  |

Если отбросить старший разряд, станет понятно, почему данный код называется рефлексивным (т.е. отраженным). Дело в том, что вторая половина значений в оставшейся части кода Грея эквивалентна первой половине, только в обратном порядке. В примере для n=2 и n=3 первая и вторая половины выделены желтым и синим цветом. Если же разделить каждую половину ещё раз пополам, свойство будет сохраняться для каждой из половин половины и т. Д.

Будем начинать генерацию с последовательности 000…0. Для продолжения процесса генерации нам необходимо на каждом шаге знать номер разряда, который будет на этом шаге изменяться. Будем хранить эти номера в массиве **y**. Очевидно, что в массиве должно быть 2n – 1 элементов. Элемент **yi** будет обозначать номер разряда, который изменяется при переходе от i – той последовательности к следующей. В примере справа от кодов Грея записаны элементы массива **y** для n=1,2,3. Если мы знаем массив **у** кода Грея для i элементов, нетрудно получить значения элементов массива **y** для i+1- го элемента. Для этого нужно переписать старый массив **y**, затем записать в массив i, а затем снова переписать старый массив **y.** Это наблюдение позволяет написать несложную программу генерации кода Грея длины n.

#include<stdio.h>

#include<conio.h>

#define n 5

#define nn 32

int x[n], y[nn];

main(){

int i,j,k=1;

for(i=1;i<n;i++){ //Формируем массив **y** для i+1-го элемента

//Левая половина была сформирована на предыдущем шаге

y[k]=i; //В середину записываем i

for(j=0;j<k;j++) //Формируем правую половину

y[k+j+1]=y[j];

k=2\*k+1;

}

for(i=0;i<n;i++) //Создаем начальную последовательность

x[i]=0;

for(i=0;i<=k;i++){

for(j=n-1;j>=0;j--) //Печатаем очередную последовательность

printf(“%d “,x[j]);

printf(“\n”);

x[y[i]]=1-x[y[i]]; //Изменяем в ней нужный разряд

}

getch();

return 0;

}

Если сравнить номера элементов последовательности кодов Грея и элементы массива **y**, можно заметить, что **yi** – это количество нулей, которыми заканчивается число i+1 в двоичной записи. Чтобы получить **yi** , нужно найти максимальную степень двойки, на которую делится i+1. Это позволяет написать следующую версию программы генерации кода Грея:

#include<stdio.h>

#include<conio.h>

#define n 5

int x[n];

main(){

int i,j,y,k=1,del;

for(i=0;i<n;i++){

x[i]=0; //Формируем начальную последовательность

k=k\*2; //k=2n– количество последовательностей

}

for(i=0;i<k;i++){

for(j=n-1;j>=0;j--) //Печатаем очередную последовательность

printf(“%d “,x[j]);

printf(“\n”);

y=0; del=i+1; //Считаем максимальную степень двойки,

while(del%2==0){ //на которую делится i+1.

del/=2;

y++;

}

x[y]=1-x[y]; //Меняем нужный разряд

}

getch();

return 0;

}

**Генерация всех последовательностей длины n из чисел 0,1…k-1 с минимальными изменениями.**

Требуется перечислить все последовательности длины n из чисел 0,..k-1 в таком порядке, чтобы каждая следующая отличалась от предыдущей в единственной цифре, причем не более, чем на 1.

Идея решения взята из [1]. Рассмотрим прямоугольную доску из n вертикалей и k горизонталей. Пронумеруем горизонтали от 0 до k-1. На каждой вертикали поставим шашку. Таким образом, положения шашек соответствуют последовательностям из чисел 0,..k-1 длины n (s-ый член последовательности соответствует высоте шашки на s-ой горизонтали). На каждой шашке нарисуем стрелочку, которая может быть направлена вверх или вниз. Вначале все шашки поставим на нижнюю (нулевую) горизонталь стрелочкой вверх. Далее двигаем шашки по такому правилу: найдя самую правую шашку, которую можно подвинуть в направлении нарисованной на ней стрелки, двигаем ее на одну клетку в этом направлении, а у всех стоящих правее ее шашек (они уперлись в край) меняем направление стрелки на противоположное. Процесс заканчивается, когда ни одна шашка не может сделать ход.

Очевидно, что при таком движении каждая следующая последовательность отличается от предыдущей в одном разряде, причем ровно на единицу.

Упр. Доказать, что будут получены все последовательности.

Реализуем данный алгоритм. Направления стрелок будем хранить в отдельном массиве d. Стрелке вверх соответствует значение 1, стрелке вниз –значение -1.

#include<stdio.h>

#include<conio.h>

#define n 4

#define k 3

int x[n],d[n];

int nelzia(int i){ //Функция возвращает 1, если i-я шашка не может

if((x[i]==k-1 && d[i]==1) || //сделать ход, и 0, если может.

(x[i]==0 && d[i]==-1)) return 1;

else return 0;

}

main(){

int i,j;

for(i=0;i<n;i++){

x[i]=0; //Сначала все шашки на нижней горизонтали.

d[i]=1; //Сначала все стрелки вверх.

}

while(1){ .

for(i=0;i<n;i++) //Печатаем очередную последовательность.

printf("%d ",x[i]);

printf("\n");

for(i=n-1;i>=0 && nelzia(i);i--) //Ищем самую правую шашку,

; //которая может сделать ход.

if(i==-1) break; //Если не нашли, то генерация закончена..

x[i]=x[i]+d[i]; //Если нашли, делаем ход.

for(j=i+1;j<n;j++) //У шашек правее нашей

d[j]=-d[j]; //Меняем направление стрелок.

}

getch();

return 0;

}

**Генерация сочетаний в лексикографическом порядке.**

Рассмотрим множество Z = {1,2,3,…n}. Требуется сгенерировать все его подмножества, состоящие из k элементов.

Будем считать, что элементы нашего подмножества располагаются в порядке возрастания. Начинаем с подмножества {1,2,3,…k}. По предыдущему подмножеству {x1,x2,…,xk} будем строить следующее таким образом: просматривая наше подмножество справа налево, находим самый первый элемент, не достигший максимального значения. Очевидно, что максимальное значение для xk равно n, для xk-1 – n-1 и т. д. Найденный элемент xi увеличиваем на 1, а все элементы справа от него делаем самыми маленькими из возможных: xi+1 = xi +1, xi+2 = xi+1 +1 и.т.д.

Получим следующую программу:

#include<stdio.h>

#include<conio.h>

#define n 6

#define k 3

int x[k];

main(){

int i,j;

for(i=0;i<k;i++) //Формируем начальную последовательность

x[i]=i+1;

while(1){

for(i=0;i<k;i++) //Печатаем очередную последовательность

printf("%d ",x[i]);

printf("\n");

for(i=k-1;i>=0 && x[i]==n+i-k+1;i--) //Ищем первый справа элемент, не достигший

; //максимального значения

if(i==-1) break; //Если не нашли, то заканчиваем работу.

x[i]++; //Если нашли, то увеличиваем его на 1

for(j=i+1;j<k;j++) //и заполняем правую часть

x[j]=x[j-1]+1; //минимально возможными значениями.

}

getch();

return 0;

}

**Генерация перестановок из n элементов в лексикографическом порядке.**

Рассмотрим множество Z = {1,2,3,…n}. Требуется сгенерировать все перестановки элементов множества Z.

Начинаем с перестановки {1,2,3,…n}. По предыдущей перестановке {x1,x2,…,xn} будем строить следующую за ней в лексикографическом порядке. У нашей перестановки можно увеличить i-й член, не меняя предыдущих, если он меньше какого-либо из следующих членов (членов с номерами больше i). Таким образом: просматривая нашу перестановку справа налево, находим самую первую позицию i, т.ч. xi<xi+1. Если такой позиции нет, то заканчиваем работу.

После этого xi нужно увеличить наименьшим возможным способом. Для этого, просматривая элементы перестановки от xi слева направо, ищем наименьший из элементов xj, т.ч. xi<xj.

Меняем местами xi с xj. Теперь элементы справа от xi образуют убывающую последовательность. Переписываем элементы от xi+1 до xn в обратном порядке.

Следующая программа реализует данный алгоритм:

#include<stdio.h>

#include<conio.h>

#define n 4

int x[n];

main(){

int i,j,tmp,min,nmi;

for(i=0;i<n;i++) //Формируем начальную перестановку

x[i]=i+1;

while(1){

for(i=0;i<n;i++) //Печатаем очередную перестановку

printf("%d ",x[i]);

printf("\n");

for(i=n-2;i>=0 && x[i+1]<x[i];i--) //Ищем первый справа xi, т.ч. xi < xi+1

;

if(i==-1) break; //Если не нашли, заканчиваем работу.

min=100;nmi=i; //Ищем справа от xi наименьший xj, т.ч. xi < xj.

for(j=i+1;j<n;j++)

if(x[j]>x[i] && x[j]<min){

min=x[j];nmi=j;

}

x[nmi]=x[i];x[i]=min; //Меняем xi с xj.

for(j=i+1;j<=(i+n)/2;j++){ //Переписываем элементы справа от xi

tmp=x[j];x[j]=x[i+n-j];x[i+n-j]=tmp;//в обратном порядке.

}

}

getch();

return 0;

}

**Генерация перестановок из n элементов с минимальными изменениями.**

Рассмотрим множество Z = {1,2,3,…n}. Требуется сгенерировать все перестановки элементов множества Z. При этом необходимо, чтобы каждая следующая перестановка получалась из предыдущей при помощи одной транспозиции (перестановки) двух соседних элементов. Например, при n = 3 допустим такой порядок: 3 2 1 -> 2 3 1 -> 2 1 3 -> 1 2 3 -> 1 3 2 -> 3 1 2.

Идея решения взята из [1]. Рассмотрим множество последовательностей y0, y1,…, yn-1 целых неотрицательных чисел, у которых y0<=0, y1<=1,…, yn-1<=n-1. Установим взаимно однозначное соответствие между множеством перестановок из n элементов и множеством таких последовательностей. Именно, каждой перестановке поставим в соответствие последовательность y0, y1,…, yn-1, где yi - количество чисел, меньших i+1 и стоящих левее i+1 в этой перестановке. В качестве примера рассмотрим случай n=3.

|  |  |  |  |
| --- | --- | --- | --- |
| Перестановка | Последовательность | | |
|  | y0 | y1 | y2 |
| 321 | 0 | 0 | 0 |
| 231 | 0 | 0 | 1 |
| 213 | 0 | 0 | 2 |
| 123 | 0 | 1 | 2 |
| 132 | 0 | 1 | 1 |
| 312 | 0 | 1 | 0 |

Из анализа данной таблицы видно, что изменение на единицу одного из членов последовательности **y** соответствует транспозиции двух соседних элементов перестановки. Именно, увеличение yi на 1 соответствует транспозиции числа i+1 с его правым соседом, а уменьшение - с левым.

Теперь вспомним решение задачи о генерации всех последовательностей длины n из чисел 0,1…k-1 с минимальными изменениями. Заменим прямоугольную доску доской в форме лестницы (пронумеруем вертикали от 0 до n-1, высота i-ой вертикали равна i). Двигая шашки по тем же правилам, мы сгенерируем все возможные последовательности **y**. Параллельно с изменением **y** будем корректировать перестановку. Для этого можно искать в ней число i+1, но это требует лишних действий. Поэтому мы будем для получения i+1 в массиве **inv\_x** хранить перестановку, обратную нашей, изменяя ее по мере необходимости.

Первой будет сгенерирована перестановка n,n-1,…1. Ей соответствует последовательность **y** из одних нулей.

Программа, реализующая данный алгоритм, выглядит так:

#include<stdio.h>

#include<conio.h>

#define n 4

int y[n], x[n+1], inv\_x[n+1], d[n];

int nelzia(int i){

if((y[i]==i && d[i]==1) || (y[i]==0 && d[i]==-1)) //Функция возвращает 1,если i – й

return 1; //шашкой нельзя пойти,

else return 0; //и 0, если можно.

}

void printfX(void){ //Функция печатает очередную

for(int i=1;i<=n;i++) //перестановку x.

printf("%d ",x[i]);

printf("\n");

}

main(){

int i,j,pos1,pos2,val1,val2,tmp;

for(i=0;i<n;i++){ //Заносим начальные значения.

x[i+1]=n-i;

inv\_x[i+1]=n-i;

d[i]=1;

y[i]=0;

}

while(1){

printfX();

for(i=n-1;i>=0 && nelzia(i);i--) //Ищем самую правую шашку,

; //у которой есть ход.

if(i==-1) break; //Если не нашли, то конец.

y[i]=y[i]+d[i]; //Генерируем новый **y**.

pos1 = inv\_x[i+1]; //pos1 – номер элемента i+1 в перестановке

val1 = i+1; //pos2 – номер соседа, с которым будем

pos2 = pos1 + d[i]; //его менять.

val2 = x[pos2];

tmp = x[pos1]; //Меняем i+1 с соседом

x[pos1] = x[pos2]; //в перестановке x и в обратной.

x[pos2] = tmp;

tmp = inv\_x[val1];

inv\_x[val1] = inv\_x[val2];

inv\_x[val2] = tmp;

for(j=i+1;j<n;j++) //Меняем направление движения

d[j]=-d[j]; //шашек, которые уперлись в край доски

} //правее нашей шашки.

getch();

return 0;

}