Über FP mit Hamming-Codes Problem sprechen

Dieser Beitrag wurde ursprünglich auf Chinesisch geschrieben und auf CSDN veröffentlicht.

Problem Link

Das Problem besteht darin, die lexikografisch kleinste `n` Zahlen zu finden, sodass der Hamming-Abstand zwischen je zwei Zahlen mindestens `d` beträgt.

Der Hamming-Abstand kann mit XOR berechnet werden. `1^0=1`, `0^1=1`, `0^0=0`, `1^1=0`. Das XOREN zweier Zahlen ergibt eine Zahl, bei der die gesetzten Bits die unterschiedlichen Bits darstellen. Man kann dann die Anzahl der gesetzten Bits im Ergebnis zählen.

Ich habe einmal einen Fehler gemacht, weil die Ausgabe 10 Zahlen pro Zeile erfordert, wobei die letzte Zeile möglicherweise weniger als 10 Zahlen enthält. Meine anfängliche Ausgabe hatte einen Leerraum nach der letzten Zahl in der letzten Zeile, gefolgt von einem Zeilenumbruch.

Ich denke, dies ist eine ziemlich gute funktionale Programmierstil-Code. Der Vorteil ist, dass er strukturierter ist, sodass `main` wie ein Top-Level in Lisp oder anderen funktionalen Sprachen funktioniert.

Auf diese Weise muss ich keine neue cpp-Datei erstellen, um unvertraute Funktionen zu testen oder einzelne Funktionen zu debuggen. Ich kann einfach `deal()` auskommentieren und `main` als Top-Level REPL (read-print-eval-loop) verwenden.

Lisp hat mich auch gelehrt, so funktional wie möglich zu programmieren, FP! Auf diese Weise kann jede Funktion extrahiert und separat debuggt werden. Die Semantik ist auch klarer. Zum Beispiel:

`hamming(0, 7, 2)` bedeutet, zu überprüfen, ob sich die binären Darstellungen von 0 und 7 um mindestens 2 Bits unterscheiden. 7 ist `111`, daher unterscheiden sie sich um 3 Bits und die Funktion gibt `true` zurück.

Ich kann also `deal()` auskommentieren und `hamming(0, 7, 2)` hinzufügen, um diese Funktion unabhängig zu testen.

AC Code:

```
/*
{
ID: lzwjava1
PROG: hamming
LANG: C++
}
*/
#include<cstdio>
#include<cstring>
#include<math.h>
```

```
#include<stdlib.h>
#include<algorithm>
#include<ctime>
using namespace std;
const int maxn=1000;

bool hamming(int a,int b,int d)
{
    int c=a^b;
    int cnt=0;
    for(int i=0;i<=30;i++)
    {
        if((1<<i) & c)
        {
            cnt++;
            if(cnt>=d) return true;
        }
    }
    return false;
}

void printArr(int *A,int n)
{
    for(int i=0;i<n;i++)
    {
        printf("%d",A[i]);
        if((i+1)%10==0 || (i==n-1)) printf("\n");
        else printf(" ");
    }
}

bool atLesat(int *A,int cur,int i,int d)
{
    for(int j=0;j<cur;j++)
        if(!hamming(A[j],i,d))
            return false;
    return true;
}

void dfs(int *A,int cur,int n,int d)
```

```
{
    if(cur==n)
    {
        printArr(A,n);
        return;
    }
    int st=(cur==0? 0: A[cur-1]+1);
    for(int i=st;;i++)
    {
        if(atLesat(A,cur,i,d))
        {
            A[cur]=i;
            dfs(A,cur+1,n,d);
            return;
        }
    }
}

void deal()
{
    int n,b,d;
    scanf("%d%d%d",&n,&b,&d);
    int A[n];
    dfs(A,0,n,d);
}

int main()
{
  freopen("hamming.in","r",stdin);
  freopen("hamming.out","w",stdout);
    deal();
    //printf("%.2lf\n",(double)clock()/CLOCKS_PER_SEC);
  return 0;
}

/*
*/
```