Hablando sobre FP con el problema de códigos de Hamming

Este post fue originalmente escrito en chino y publicado en CSDN.

Enlace del problema

El problema pide encontrar los `n` números lexicográficamente más pequeños tales que la distancia de Hamming entre cualquier par de números sea al menos `d`.

La distancia de Hamming se puede calcular usando XOR. `1^0=1`, `0^1=1`, `0^0=0`, `1^1=0`. Entonces, XOR de dos números dará un número donde los bits establecidos representan los bits diferentes. Luego, se puede contar el número de bits establecidos en el resultado.

Hice un error una vez porque la salida requiere 10 números por línea, con la última línea que puede tener menos de 10. Mi salida inicial tenía un espacio en blanco después del último número en la última línea, seguido de un salto de línea.

Creo que este es un buen estilo de código de Programación Funcional. El beneficio es que está más estructurado, haciendo que `main` actúe como un nivel superior en Lisp u otros lenguajes funcionales.

De esta manera, no necesito crear un nuevo archivo cpp para probar funciones poco familiares o depurar funciones individuales. Simplemente puedo comentar `deal()` y usar `main` como un nivel superior REPL (read-print-eval-loop).

Lisp también me enseñó a programar de la manera más funcional posible, ¡FP! De esta manera, cada función se puede extraer y depurar por separado. Las semánticas también son más claras. Por ejemplo:

`hamming(0, 7, 2)` significa verificar si las representaciones binarias de 0 y 7 difieren en al menos 2 bits. 7 es `111`, por lo que difieren en 3 bits, y la función devuelve verdadero.

Entonces, puedo comentar `deal()` y agregar `hamming(0, 7, 2)` para probar esta función independientemente.

Código AC:

```
/*
{
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;
}

/*

'''
```