Parler de FP avec le problème des codes de Hamming

Ce message a été initialement écrit en chinois et publié sur CSDN.

Lien vers le problème

Le problème consiste à trouver les `n` nombres les plus petits au sens lexicographique tels que la distance de Hamming entre deux nombres quelconques soit d'au moins `d`.

La distance de Hamming peut être calculée en utilisant le XOR. `1^0=1`, `0^1=1`, `0^0=0`, `1^1=0`. Ainsi, le XOR de deux nombres donnera un nombre où les bits définis représentent les bits différents. On peut ensuite compter le nombre de bits définis dans le résultat.

J'ai fait une erreur une fois parce que la sortie nécessite 10 nombres par ligne, la dernière ligne pouvant en avoir moins de 10. Ma sortie initiale avait un espace de fin après le dernier nombre de la dernière ligne, suivi d'un saut de ligne.

Je pense que c'est un bon code de style Programmation Fonctionnelle. Le bénéfice est qu'il est plus structuré, ce qui fait que `main` agit comme un niveau supérieur dans Lisp ou d'autres langages fonctionnels.

Ainsi, je n'ai pas besoin de créer un nouveau fichier cpp pour tester des fonctions inconnues ou déboguer des fonctions individuelles. Je peux simplement commenter `deal()` et utiliser `main` comme un REPL (boucle lecture-évaluation-affichage) de niveau supérieur.

Lisp m'a également appris à programmer de la manière la plus fonctionnelle possible, FP! De cette façon, chaque fonction peut être extraite et déboguée séparément. Les sémantiques sont également plus claires. Par exemple:

`hamming(0, 7, 2)` signifie vérifier si les représentations binaires de 0 et 7 diffèrent d'au moins 2 bits. 7 est `111`, donc elles diffèrent de 3 bits, et la fonction retourne true.

Ainsi, je peux commenter `deal()` et ajouter `hamming(0, 7, 2)` pour tester cette fonction indépendamment.

Code 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;
}

/*
```