# 计算系统基础第二次上机作业题解

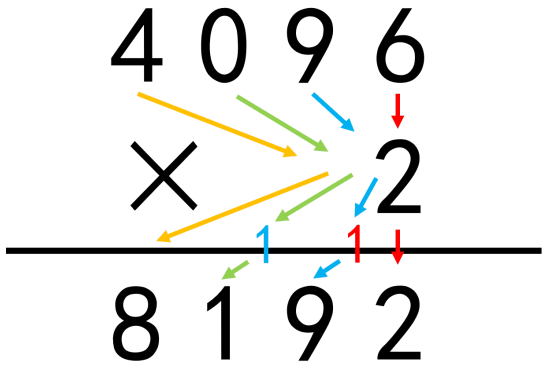
## 211850016 Sakiyary 写于21.11.13晚

## A. powerof2.c 高精度2的幂次

这题是一道高精度计算（高精度计算，也被称为大整数计算，即运用一些算法结构来支持超过数据类型大小的整数间的运算），基本思路就是用数组模拟竖式计算（即将小学老师教的的竖式计算呈现进代码）。

（有一些进位细节与数组长度/边界问题需要注意！）

基本思路就如下图：




习惯上，下标最小的位置存放的是数字的**最低位**，即存储**反转的数组**。这么做的原因在于，数字的长度可能发生变化，但我们希望同样权值位始终保持对齐（例如，希望所有的个位都在下标`[0]`，所有的十位都在下标`[1]`……），以此在进行竖式计算时更加方便。

下面呈现一下源码与注释：

```
#include <stdio.h>
#include <string.h>

int n;
int two[10005];//乘数(如4096)
int ans[10005];//积(如8192)
int twolen = 1;//乘数的位数(即有效数组长度)
int anslen;//积的位数

//至于为什么将数组开到10005，只是怕越界且不知道2^10000是多少位，才这样保险地开

int main() {
    scanf("%d", &n);
```

```
    two[0] = 1;//初始化乘数为1
    //以下为主体循环
    for (int i = 0; i < n; i++) {
        //以下统计乘数的位数
        for (int j = 10000; j > 0; j--) {
            if (two[j - 1] != 0) {
                twolen = j;
                break;
            }
        }
        //由于积为乘数x2,故积的位数最多比乘数的位数多1
        anslen = twolen + 1;
        //由个位(最小位)开始逐位竖式乘法
        for (int j = 0; j < anslen; j++) {
            ans[j] += two[j] * 2;
            //以下为进位!!!
            if (ans[j] >= 10) {
                ans[j + 1] += 1;
                ans[j] %= 10;
            }
        }
        //让积成为下一次循环的乘数
        for (int j = 0; j <= anslen; j++){
            two[j] = ans[j];
        }
        //清除积中的数值
        memset(ans, 0, sizeof(ans));
    }
    //以下为输出模块
    int i;
    for (i = anslen; i > 0; i--){
        if (two[i] != 0){
            break;//去掉积的高位可能存在的"0"
        }
    }
    for (; i >= 0; i--){
        printf("%d", two[i]);//输出
    }
    return 0;
}
```

如果你现在已经理解了这道题的基础思路，那么你就可以挑战一下C-PL课程OJ平台上的附加题：[a + b problem]和[超级计算器]

加油！你可以写出来的！

# B. ai.c 数字转换英文

~~(我觉得这道题更适合当签到题)~~

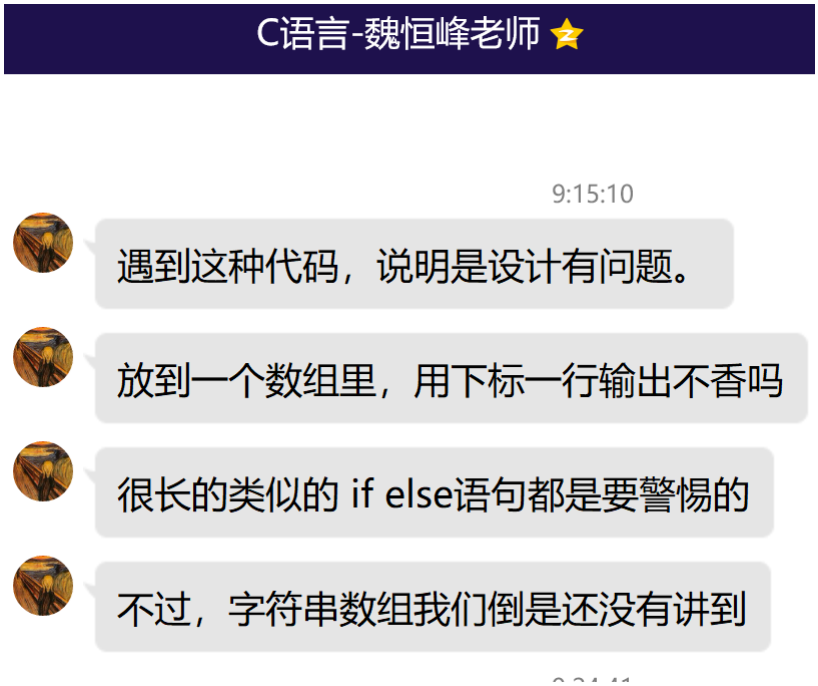
这道题可以用全屏`if-else`或`switch-case`顺利通关，只要考虑完所有的数与特例。

先判断**"大三位"**，即**十亿位**`Billion`、**百万位**`Million`、**千位**`Thousand`（数据上限是2,147,483,647也就是21亿，所以这些就够了）。

然后判断每个大三位内的**"小三位"**。数字`1~19`都需要特判，`20~99`就可以借用前面写的`1~9`，然后再判定有无**百位**，若有**百位**就继续借用`1~9`并加上`Hundred`。

至此，每个大三位和小三位就都处理完了。

**但是！！！**




~~（做一波预言，蚂蚁老师会在21.11.15周一的课上将这件事（本题解写于13日晚））~~

所以我们得想一个办法避免掉全屏的判断语句，像蚂蚁老师说的那样存进一个字符串里，用下标来输出。

~~这个思路其实和第一次上机作业题解中第三题的**标记法**有一点相似~~

直接看一下源码与注释：

```
#include <stdio.h>

//函数声明
int onenum2char(int a, int b);//判定小三位中的特例
int twonum2char(int a, int b);//判定小三位中的20~99
void threenum2char(int a);//判定大三位

long n;
int num[32] = {0};

int main() {
    scanf("%d", &n);
    //以下为数的拆分，将每一位都存进数组
    //len即数组长度、数的位数
    int len = 0;
    while (n != 0) {
        len++;
```

```
        num[len] = n % 10;
        n /= 10;
    }
    //以下为主体判定与输出
    //输出部分均在外置函数中
    for (int i = (len - 1) / 3; i >= 0; i--) {
        //i即大三位的位数，注意看(len-1)/3才是i的初始值
        //flag标记该大三位内有无非零值
        int flag = 0;
        //如果小三位的百位非零，则含Hundred
        if (onenum2char(0, num[i * 3 + 3])) {
            printf("Hundred ");
            flag = 1;
        }
        //如果小三位的百位十位个位均非零，则判定大三位
        if (twonum2char(num[i * 3 + 2], num[i * 3 + 1]) || flag) {
            threenum2char(i);
        }
        //注意看下面的函数，理解一下为什么我将函数写在if条件中也可以输出结果！
    }
    return 0;
}

int onenum2char(int a, int b) {
    if (a == 0 && b == 0)return 0;//和主循环中的flag同理，判定是否全为零，全为0则返回值为0
    //以下用蚂蚁老师的方法来输出，"Zero"只是为了占位子，没有实际用处
    static char *onenum[] = {
            "Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine",
            "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"
    };
    printf("%s ", onenum[a * 10 + b]);
    return 1;//非零则返回值为1
}

int twonum2char(int a, int b) {
    if (a < 2) {
        return onenum2char(a, b);
    }//判定是否要用特例函数
    //以下用蚂蚁老师的方法来输出，"0"和"1"只是为了占位子，没有实际用处
    static char *twonum[] = {
            "0", "1", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"
    };
    printf("%s ", twonum[a]);
    return onenum2char(0, b);//既可以判定是否非零，也可以同时运行一遍该函数，将onenum2char的返回值作为该函数返回值
}
```

```
void threenum2char(int a) {
    //以下用蚂蚁老师的方法来输出大三位，由于i的取值问题，故用"One"来占位，没有实际用处
    static char *threenum[] = {
            "One", "Thousand", "Million", "Billion"
    };
    printf("%s ", threenum[a]);
}
//（其实上述函数中这些占位都可以优化掉，但是我懒）
```

# C. antwar.c 蚂蚁大战（数列消除）

~~这题的错数据直接让我最初的方法卡了几个小时没过呜呜呜，不过21.11.12晚已经修复！~~

这道题我们直接用**最朴素**的想法吧！

如果**第i只蚂蚁的值>0**，**第i+1只蚂蚁的值<0**，那我们就让他们打一架，将败者（或者同归于尽）**删去**，后面的蚂蚁**整体前移**补上空位，以此类推。

姑且称该方法为**删除法**。~~（删除法是能在数据错误的情况下得出OJ认为正确的结果的！）~~

因为我想讲两种**朴素**的方法，所以我们先写个`main`函数出来：

```
int n;
int ant[10005] = {0};
int main(){
    scanf("%d",&n);
    //此处应有一些初始化
    //以下为读入模块
    for(int i=1;i<=n;i++){
        scanf("%d",&ant[i]);
    }
    //以下为大战模块
    while(/*某个条件*/){
        Fight();//Fight即为主要的蚂蚁大战算法
    }
    //以下为输出模块
    for(int i=1;i<=/*删除后的长度*/;i++){
        printf("%d ",ant[i]);
    }
    return 0;
}
```

写完了**main函数**，我们就开始想**删除法**的主体**Fight过程**：

```
int left =n;//left即为删除后的字符长度（指剩下的蚂蚁们

void Fight() {
    //以下为从1到left让蚂蚁们对一波线
    for (int i = 1; i <= left; i++) {
        if (ant[i] > 0 && ant[i + 1] < 0) {
            int j;
```

```
            if (ant[i] == -ant[i + 1]) {
                for (j = i; j <= n; j++){
                    ant[j] = ant[j + 2];
                }
                left -= 2;
            }
            else if (ant[i] > -ant[i + 1]) {
                for (j = i + 1; j <= n; j++){
                    ant[j] = ant[j + 1];
                }
                left -= 1;
            }
            else if (ant[i] < -ant[i + 1]) {
                for (j = i; j <= n; j++){
                    ant[j] = ant[j + 1];
                }
                left -= 1;
            }
        }
    }
}
```

那么我们写完了`Fight()`过程和`main`函数，现在缺一个将两者连接在一起的条件，即`while()`中的终止条件，要对线到什么时候停呢?

这里我们再用一个`flag`标记变量，要是能进行`Fight()`过程，就`flag++`，要是到最后走一遍`Fight()`过程，`flag`还是0，就可以跳出循环了。

整体代码呈现如下（注释也一并复制过来了）：

```
#include <stdio.h>

int n;
int flag = 1;
int left;
int ant[10005] = {0};

void Fight();

int main(){
    scanf("%d",&n);
    left = n;
    //以下为读入模块
    for(int i=1;i<=n;i++){
        scanf("%d",&ant[i]);
    }
    //以下为大战模块
    while(flag != 0){
        Fight();
    }
    //以下为输出模块
```

```
    for(int i=1; i <= left; i++){
        printf("%d ",ant[i]);
    }
    return 0;
}

void Fight() {
    flag = 0;
    //以下为从1到left让蚂蚁们对一波线
    for (int i = 1; i <= left; i++) {
        if (ant[i] > 0 && ant[i + 1] < 0) {
            int j;
            if (ant[i] == -ant[i + 1]) {
                for (j = i; j <= n; j++){
                    ant[j] = ant[j + 2];
                }
                flag++;
                left -= 2;
            }
            else if (ant[i] > -ant[i + 1]) {
                for (j = i + 1; j <= n; j++){
                    ant[j] = ant[j + 1];
                }
                flag++;
                left -= 1;
            }
            else if (ant[i] < -ant[i + 1]) {
                for (j = i; j <= n; j++){
                    ant[j] = ant[j + 1];
                }
                flag++;
                left -= 1;
            }
        }
    }
}
```

## 下面为无意义的优化环节，不想看的直接跳转D题！！！

这个方法，循环的次数还是有点多了，我们可以进行一些简单的次数优化。

我们注意到，当蚂蚁数列删除到形如`-1000 -500 -100 …… 200 600 1000`，在`Fight()`的时候已经不需要再管最左边的负数和最右边的正数了，那么我们就可以把`Fight()`的范围适当缩小。

那我们就来写一个`Narrow()`过程：

```
int start = 1;
```

```
int end = left;
void Narrow(){
    for(int i=start;i<=end;i++){
        start=i;
        if(ant[i]>0){
            break;
        }
    }
    for(int i=end;i>=start;i--){
        end=i;
        if(ant[i]<0){
            break;
        }
    }
}
```

接下来就是将优化`Narrow()`过程整合进我们的主代码，同时注意，我们不再需要`flag`这个标记变量，因为当`(start == end)`时，我们就能跳出循环了！

```
#include <stdio.h>

int n;
int left;
int start, end;
int ant[10005] = {0};

void Fight();

void Narrow();

int main() {
    scanf("%d", &n);

    left = n;
    start = 1;
    end = left;

    for (int i = 1; i <= n; i++) {
        scanf("%d", &ant[i]);
    }

    while (start != end) {
        Narrow();
        Fight();
    }

    for (int i = 1; i <= left; i++) {
        printf("%d ", ant[i]);
    }
    return 0;
```

```
}

void Fight() {
    for (int i = start; i <= end; i++) {
        if (ant[i] > 0 && ant[i + 1] < 0) {
            int j;
            if (ant[i] == -ant[i + 1]) {
                for (j = i; j <= n; j++) {
                    ant[j] = ant[j + 2];
                }
                left -= 2;
            }
            else if (ant[i] > -ant[i + 1]) {
                for (j = i + 1; j <= n; j++) {
                    ant[j] = ant[j + 1];
                }
                left -= 1;
            }
            else if (ant[i] < -ant[i + 1]) {
                for (j = i; j <= n; j++) {
                    ant[j] = ant[j + 1];
                }
                left -= 1;
            }
        }
    }
}

void Narrow() {
    for (int i = start; i <= end; i++) {
        start = i;
        if (ant[i] > 0) {
            break;
        }
    }
    for (int i = end; i >= start; i--) {
        end = i;
        if (ant[i] < 0) {
            break;
        }
    }
}
```

可由于删除法本身局限（即每次都要后方整体补位），所以并没能有效地减少大部分循环次数。

所以我给出我一开始的方法~~（也就是被错数据折磨几个小时的方法）~~，我称之为**设0法**，顾名思义，被杀的位置不是删除，而是设为0。

也没什么好解释的，反正比删除法少循环很多次就是了，直接呈现搭配`Narrow()`过程的**设0法**源码：

```
#include <stdio.h>
```

```c
int n;
int start, end;
int ant[10005] = {0};
int copyant[10005] = {0};

void Fight();

void Narrow();

int main() {
    scanf("%d", &n);

    start = 1;
    end = n;

    for (int i = 1; i <= n; i++) {
        scanf("%d", &ant[i]);
        copyant[i] = ant[i];
    }

    while (start != end) {
        Narrow();
        Fight();
    }

    for (int i = 1; i <= n; i++) {
        if (ant[i] == copyant[i]) {
            printf("%d ", ant[i]);
        }
    }
    return 0;
}

void Fight() {
    for (int i = start; i <= end; i++) {
        if (ant[i] > 0) {
            int j;
            for (j = i + 1; j <= end; j++) {
                if (ant[j] != 0) {
                    break;
                }
            }
            if (ant[j] < 0) {
                if (ant[i] == -ant[j]) {
                    ant[i] = 0;
                    ant[j] = 0;
                } else if (ant[i] > -ant[j]) {
                    ant[j] = 0;
                } else if (-ant[j] > ant[i]) {
                    ant[i] = 0;
```

```
                }
            }
        }
    }
}

void Narrow() {
    for (int i = start; i <= end; i++) {
        start = i;
        if (ant[i] > 0) {
            break;
        }
    }
    for (int i = end; i >= start; i--) {
        end = i;
        if (ant[i] < 0) {
            break;
        }
    }
}
```

我觉得都挺朴素的！嗯！确信！~~（听说要是两种方法混用且不加Narrow优化就会TLE）~~

# D. diversestring.c 判定连续不重复字符串

~~别管***英文题面啦！！！~~

助教哥哥：翻译一下题面，如果一个字符串由无重复的连续字符构成，输出Yes，否则输出No(连续是指符合a-z的字典序)。

连续字符的定义的解释：`badce`是一串连续字符，`abde`就不是。

那么，这道题继续用的是第一次上机作业题解中第二题的**标记法**，如下：

```
    cnt[str[i] - 'a']++;
```

在对每个字符标记之后，那么判重就非常简单了。至于连续，当我们在循环**标记数组**的时候，寻找到第一串连续出现的字母后，再打个标记，如果之后还能找到落单的字母，那这个字符串就是不合法的了。

下面呈现一下源码和注释：

```
#include <stdio.h>
#include <string.h>

int n;
int cnt[27];//标记数组
int len;//字符串长度
int ans = 0;//总标记
int flag = 0;//连续标记
int i, j;
char str[60];//初始字符串
```

```c
int main() {
    scanf("%d", &n);
    //以下为主体循环
    for (i = 0; i < n; i++) {
        scanf("%s", str);
        len = strlen(str);
        //strlen()是一个返回字符串长度的函数，包含在第二个头文件中
        //注意strlen()的返回值是unsigned long long，这边直接赋值给int型是偷懒的行为！
        for (int j = 0; j < len; j++) {
            cnt[str[j] - 'a']++;//打标记！！！接下来可以舍弃初始字符串
        }
        //以下为判定重复
        for (j = 0; j <= 26; j++){
            if (cnt[j] > 1){
                ans++;//用总标记来查重
            }
        }
        //以下为判定连续
        for (j = 0; j <= 26; j++) {
            int k;
            if (flag == 0 && cnt[j] == 1) {
                flag = 1;//标记已经出现过一次连续字符串了
                for (k = j; k <= 26; k++) {
                    j = k;
                    if (cnt[k] == 0) {
                        break;
                    }
                }
            }
            if (flag == 1 && cnt[j] > 0){
                ans++;//如果再往后寻找还能找到字符，那总标记就不再是0，即该字符串非法
            }
        }

        //以下为输出模块，看总标记是否为0
        if (ans == 0){
            printf("Yes\n");
        }
        else {
            printf("No\n");
        }
        //以下清空所有标记
        memset(cnt, 0, sizeof(cnt));
        ans = 0;
        flag = 0;
    }
    return 0;
}
```

# Extra. permutation.c 全排列（用车轮造小车车）

这道附加题会让我们想起C-PL课程OJ平台上的5-function中的下一个排列（next-permutation.c）。那么“下一个排列”这道题的代码就是这道附加题的**轮子**啦！

思路很简单，先将数列**按升序排序**（冒泡，选择，桶，随便你用哪个，快速排序也可以），然后不断循环你的**“轮子”**代码~~（直到出错）~~就可以啦！

那这个也没什么好呈现的，应该都能造出**“小车车”**吧~（就是终止条件要想一下哦！）

如果是会用C++的佬，那这题直接用C++秒杀吧！

`<algorithm>`这个库里有一个`sort()`函数（即快速排序），还有一个`next_permutation()`函数（即下一个排序），直接用就完事啦！