高精度

TEACHING COURSEWARE POWPOINT

授课时间：2025.08.20

讲评

【答案】

题号	(1)	(2)	(3)	(4)
答案	√	√	C	A

【分析】

(1)x==0||x==1 等价于 x<=1,(7 & 3)等价于 3。

(2)函数前的 inline 去掉也可以正常运行。

(3)递归求解,可以转换为递推求解。

```
f(0)=f(1)=3;        f(2)=2-f(0)=-1;      f(3)=3-f(1)=0;
f(4)=4-f(2)=5;      f(5)=5-f(3)=5;       f(6)=6-f(4)=1;
f(7)=7-f(5)=2;      f(8)=8-f(6)=7;       f(9)=9-f(7)=7。
```

(4)递推即可,f(10)=10-f(8)=3。

讲评

●判断题

(1)09 行的 ack(3, 4)改为 ack(-1,-1),程序能正常输出结果。(　　)

(2)ack 函数增长很慢。(　　)

●选择题

(3)输出为(　　)。

A. 125　　B. 250　　C. 114　　D. 514

(4)将 09 行 ack(3, 4)修改为 ack(2, 5),输出将为(　　)。

A. 125　　B. 13　　C. 130　　D. 26

【答案】

题号	(1)	(2)	(3)	(4)
答案	×	×	A	B

【分析】ack 函数的意义为参数对应的阿克曼函数值。

(1)程序无限递归下去,不会终止。

(2)阿克曼函数增长的非常快,ack(4, 3)已经无法计算。

(3)经典的阿克曼函数的递归求解,可以手工完成,因为涉及到两个参数,建议画出一张二维表。

(4)经典的阿克曼函数的递归求解,可以手工完成,因为涉及到两个参数,建议画出一张二维表。

讲评

我们从递归定义出发推导 $A(1, n)$ ：

已知 $m = 1$ ，根据递归规则，当 $m > 0$ 且 $n > 0$ 时， $A(1, n) = A(0, A(1, n - 1))$ 。

又因为基础规则 1（ $m = 0$ 时 $A(0, x) = x + 1$ ），所以 $A(0, A(1, n - 1)) = A(1, n - 1) + 1$ ，即：

$$A(1, n) = A(1, n - 1) + 1$$

这是一个递推关系，我们可以通过展开递推式求解通项：

当 $n = 1$ 时，先求 $A(1, 0)$ ：根据基础规则 2（ $m > 0$ 且 $n = 0$ ）， $A(1, 0) = A(0, 1)$ ；再由基础规则 1， $A(0, 1) = 1 + 1 = 2$ ，所以 $A(1, 0) = 2$ 。

当 $n = 1$ 时， $A(1, 1) = A(1, 0) + 1 = 2 + 1 = 3$ ；

当 $n = 2$ 时， $A(1, 2) = A(1, 1) + 1 = 3 + 1 = 4$ ；

以此类推，可归纳出 $A(1, n) = A(1, 0) + n$ 。

由于 $A(1, 0) = 2$ ，所以：

$$A(1, n) = n + 2$$

讲评

已知 $m = 2$ ，根据递归规则，当 $m > 0$ 且 $n > 0$ 时， $A(2, n) = A(1, A(2, n - 1))$ 。
又因为已推导出 $A(1, x) = x + 2$ （ $A(1, n)$ 的通项公式 ）
所以 $A(1, A(2, n - 1)) = A(2, n - 1) + 2$ ，即：
$A(2, n) = A(2, n - 1) + 2$

同样通过递推关系求解通项：
先求 $A(2, 0)$ ：根据基础规则 2（ $m > 0$ 且 $n = 0$ ）， $A(2, 0) = A(1, 1)$ ；由 $A(1, n)$ 的通项公式， $A(1, 1) = 1 + 2 = 3$ ，所以 $A(2, 0) = 3$ 。
当 $n = 1$ 时， $A(2, 1) = A(2, 0) + 2 = 3 + 2 = 5$ ；
当 $n = 2$ 时， $A(2, 2) = A(2, 1) + 2 = 5 + 2 = 7$ ；
以此类推，可归纳出 $A(2, n) = A(2, 0) + 2n$ 。
由于 $A(2, 0) = 3$ ，所以：
$A(2, n) = 2n + 3$

讲评

函数表达式	依赖的前置结果	计算过程	最终结果
A(3, 0)	A(2, 1)	2 * 1 + 3	5
A(3, 1)	A(3, 0)	2 * 5 + 3	13
A(3, 2)	A(3, 1)	2 *13 + 3	29
A(3, 3)	A(3, 2)	2 * 29 + 3	61
A(3, 4)	A(3, 3)	2 * 61 + 3	125

传引用

1. 值传递：函数用变量名做参数，修改的是副本，不影响原值
2. 引用传递：函数用&变量名做参数，修改的是原值本身

一、值传递（不会修改原值）

可以想象成 "复印文件"：

你有一份作业（原变量）

同学想参考，你复印了一份给他（函数参数是副本）

同学在复印件上修改（函数内部操作）

你的原件不会有任何变化

二、引用传递（会修改原值）

可以想象成 "直接给别人你的作业本"：

你把自己的作业直接给同学（函数参数是原变量的别名）

同学在你的作业本上修改（函数内部操作）

你的原件会跟着改变

传引用


```cpp
#include <iostream>
using namespace std;
// 函数参数是普通变量（值传递）
void change(int x) {
    x = 100;  // 只修改了副本
    cout << "函数里的x: " << x << endl;
}

int main() {
    int a = 10;
    cout << "调用前的a: " << a << endl;

    change(a);  // 传递的是a的副本

    cout << "调用后的a: " << a << endl;  // a的值还是10
    return 0;
}
```

传引用


```cpp
#include <iostream>
using namespace std;

// 函数参数是引用（用&表示）
void change(int &x) {  // x是原变量的"别名"
    x = 100;  // 直接修改原变量
    cout << "函数里的x: " << x << endl;
}

int main() {
    int a = 10;
    cout << "调用前的a: " << a << endl;

    change(a);  // 传递的是a本身

    cout << "调用后的a: " << a << endl;  // a的值变成了100
    return 0;
}
```

目录

01

‘低’精度

TEACHING
COURSEWARE

TEACH

‘低’ 精度

数据类型	位宽（二进制位数）	符号位（是否区分正负）	十进制范围（近似值）
int	32 位	有（最高位为符号位）	-2^31 ~ 2^31 -1（-21 亿 ~ 21 亿）
unsigned int	32 位	无（所有位都存数值）	0 ~ 2^32 -1（0 ~ 42 亿）
long long	64 位	有（最高位为符号位）	-2^63 ~ 2^63 -1（-9e18 ~ 9e18）

02

高精度

TEACHING
COURSEWARE

TEACH

高精度

这就是 “高精度计算” 的必要性 —— 当数字超过unsigned long long的最大范围（1.8e19），或者需要处理几百位的大数字时，必须用 “数组 / 字符串按位存储” 的方式，模拟竖式运算来计算，这就是我们接下来要学的 “高精度”。

03

高精度加减

TEACHING
COURSEWARE

TEACH

高精度加减

模拟竖式加法，逐位算 + 处理进位

代码完全复刻竖式加法的逻辑

从低位（数组下标0）到高位（数组下标max(n1,n2)-1）逐位计算，同时处理每一步的进位：

初始化关键变量：jin表示进位，初始化为0（每一位相加前无进位）；

len取两个加数的位数最大值（max(n1,n2)），确保遍历能覆盖所有位。

逐位计算总和：循环中，当前位总和sum = a[i] + b[i] + jin

需注意，当i超过某数组长度时（如计算 “123+45”，i=2时b[2]未赋值，默认为0），未赋值的数组元素视为0，避免漏算高位。

拆分当前位与进位：sum%10取总和的个位，作为结果数组c当前位的值（如sum=15，个位5存入c[i]）；sum/10取总和的十位及以上部分，作为下一位的进位（如sum=15，进位1）。

特殊处理：收尾进位，避免结果漏位

当遍历完所有位后，若仍存在未处理的进位（如 “999+1”，遍历结束后jin=1），需将进位追加到结果的最高位，否则会导致结果少一位：

代码通过if (jin != 0)判断：若进位不为0，则将进位存入结果数组c的len下标（此时len是原加数的最大位数，对应结果的新高位），同时将len加1，更新结果的总位数。

高精度加减


```cpp
#include<bits/stdc++.h>
using namespace std;
// 定义数组存储数字，大小设为100006（满足两个1e5位数字相加，结果最多1e5+1位，避免越界）
int a[100006];  // 存储第一个加数（下标0对应个位，逆序存储）
int b[100006];  // 存储第二个加数（下标0对应个位，逆序存储）
int c[100006];  // 存储加法结果（下标0对应个位，逆序存储）
string s1, s2;  // 用字符串接收输入的超大数字（避免直接存数字溢出）
int main() {
    // 1. 输入两个超大数字（以字符串形式）
    cin >> s1 >> s2;
    int n1 = s1.size();  // 获取第一个数字的位数
    int n2 = s2.size();  // 获取第二个数字的位数
    // 2. 将字符串逆序存入数组（关键：让数组下标0对应数字个位，方便从低位到高位计算）
    // 遍历第一个字符串，s1[n1-i-1]表示从字符串末尾（个位）开始取字符
    for (int i = 0; i < n1; i++) {
        a[i] = s1[n1 - i - 1] - '0';  // 字符转数字（如'5'-'0'=5）
    }
    // 同理，处理第二个字符串
    for (int i = 0; i < n2; i++) {
        b[i] = s2[n2 - i - 1] - '0';
    }
```

高精度加减

```
    // 3. 模拟竖式加法，逐位计算并处理进位
    int jin = 0;                    // 进位初始化为0（每一位相加前，进位从0开始）
    int len = max(n1, n2);          // 遍历的最大长度：取两个数字的位数最大值，确保覆盖所有位
    for (int i = 0; i < len; i++) {
        // 计算当前位总和：a[i]（第一个数当前位）+ b[i]（第二个数当前位）+ 上一位进位
        // 注意：当i超过某数组长度时，该数组对应位为0（如123+45，i=2时b[2]=0）
        int sum = a[i] + b[i] + jin;
        c[i] = sum % 10;            // 取总和的个位，作为结果当前位（如sum=15，个位5存入c[i]）
        jin = sum / 10;             // 取总和的十位及以上部分，作为下一位的进位（如sum=15，进位1）
    }

    // 4. 处理最后剩余的进位（如999+1，遍历结束后jin=1，需追加到结果末尾）
    if (jin != 0) {
        c[len] = jin;  // 将进位存入结果数组的下一位（此时len是原最大长度，对应结果高位）
        len++;         // 结果总位数加1（因多了一位进位）
    }

    // 5. 输出结果（结果数组是逆序存储，需从最后一位往前遍历，即从高位到低位输出）
    for (int i = len - 1; i >= 0; i--) {
        cout << c[i];
    }

    return 0;
}
```

高精度加减

核心运算：模拟竖式减法，逐位减 + 借位传递

代码复刻竖式减法的 “从低位到高位逐位计算” 逻辑

关键在于处理 “当前位不够减” 时的借位：

1.初始化变量

jie表示借位，初始化为0（每一位相减前无借位）；

len取两个数字的位数最大值（max(n1,n2)），确保遍历覆盖所有位。

2.逐位计算差值

循环中，当前位的被减数为 “原被减数位 - 上一位借位”，即：

sub = a[i] - jie - b[i]

若sub ≥ 0：表示够减，直接将sub存入结果数组c，借位jie重置为0；

若sub < 0：表示不够减，需向高位借 1 当 10（sub += 10），同时标记借位jie = 1（下一位计算时需减去借位）。

示例：计算 “1000 - 1” 时，个位0 - 0 - 1不够减，借位后变为10 - 1 = 9，借位jie=1；十位原本是0，减去借位后变为-1，继续借位…… 最终得到结果999。

高精度加减

去除前导 0，还原正确格式

减法结果可能因借位产生前导 0（如 “1000 - 999 = 0001”），需通过以下步骤整理：

去除前导 0

从结果数组的最高位（len-1）往前遍历，跳过连续的 0：

条件len > 1确保当结果为 “0000” 时，保留最后一个 0（正确输出 “0”）。

逆序输出结果

结果数组c是逆序存储（下标 0 为个位），需从len-1到0遍历输出，还原 “高位到低位” 的正确顺序。

高精度加减


```cpp
#include<bits/stdc++.h>
using namespace std;
int a[100006];  // 存储被减数（大数，下标0对应个位，逆序存储）
int b[100006];  // 存储减数（小数，下标0对应个位，逆序存储）
int c[100006];  // 存储减法结果（下标0对应个位，逆序存储）
string s1, s2;  // 字符串接收输入的超大数字
// 辅助函数：判断s1是否大于等于s2（确保是大数减小数，避免结果为负数）
bool compare(string s1, string s2) {
    if (s1.size() != s2.size()) {
        // 位数不同：位数多的数字更大
        return s1.size() > s2.size();
    } else {
        // 位数相同：逐位比较（从高位到低位）
        return s1 >= s2;
    }
}
int main() {
    // 1. 输入两个超大数字，先判断大小，确保被减数≥减数
    cin >> s1 >> s2;
    if (!compare(s1, s2)) {
        // 若s1<s2，交换两者（保证后续计算是大数减小数），最后输出负号
        swap(s1, s2);
        cout << "-";  // 标记结果为负数
    }
```

高精度加减


```
// 2. 将字符串逆序存入数组（下标0对应个位，方便从低位到高位计算）
for (int i = 0; i < n1; i++) {
    a[i] = s1[n1 - i - 1] - '0';
}
for (int i = 0; i < n2; i++) {
    b[i] = s2[n2 - i - 1] - '0';
}
// 3. 模拟竖式减法，逐位计算并处理借位
int jie = 0;                    // 借位初始化为0（每一位相减前，借位从0开始）
int len = max(n1, n2);          // 遍历最大长度：取两个数字的位数最大值
for (int i = 0; i < len; i++) {
    // 当前位被减数：a[i]（原被减数当前位） - 上一位借位（若上一位借了1，当前位需减1）
    int sub = a[i] - jie - b[i];
    if (sub < 0) {
        // 不够减，需向高位借1当10（如当前位是2，借位后变为12）
        sub += 10;
        jie = 1;    // 标记下一位需要还借位（下一位计算时要减1）
    } else {
        jie = 0;    // 够减，无借位
    }
    c[i] = sub;  // 将计算出的差值存入结果数组
}
```

高精度加减


```
    // 4. 去除结果的前导0（如结果是00123，需改为123；若结果是0000，保留1个0）
    // 从结果数组最后一位（高位）往前找，直到找到非0数字，确定有效长度
    while (len > 1 && c[len - 1] == 0) {
        len--;  // 有效长度减1（跳过前导0）
    }
    // 5. 输出结果（逆序数组需从后往前遍历，即从高位到低位输出）
    for (int i = len - 1; i >= 0; i--) {
        cout << c[i];
    }
    return 0;
}
```

04

高精度乘

TEACHING
COURSEWARE

TEACH

高精度乘

核心运算：交叉相乘 + 进位处理（O（n^2）核心）

代码完全复刻竖式乘法的 “交叉相乘” 逻辑，通过双重循环遍历两个乘数的每一位，累加乘积到结果数组对应位置，再统一处理进位，这是 O（n^2）复杂度的关键：

1. 交叉相乘：逐位遍历，累加乘积到对应位

双重循环逻辑：外层循环遍历第一个乘数a的每一位（i从0到n1-1，对应个位到n1-1位），内层循环遍历第二个乘数b的每一位（j从0到n2-1，对应个位到n2-1位）。

乘积位置规则：根据竖式乘法原理，a[i]（第i位）与b[j]（第j位）的乘积，会贡献到结果的第i+j位（如a[0]（个位）×b[1]（十位）=结果的第1位（十位）），因此代码通过c[i+j] += a[i] * b[j]，将每一组乘积累加到结果数组的对应位置。

高精度乘

2. 统一处理进位：逐位拆分，传递高位

交叉相乘后，结果数组c的每一位可能是两位数（如c[0]=15、c[1]=22）
需按 “个位存当前位，高位作进位” 的规则整理：
初始化jin=0（进位初始为 0），遍历结果数组的最大可能位数（n1+n2）：
当前位总和 = 数组值 + 上一位进位（total = c[k] + jin）；
取总和的个位存入当前位（c[k] = total % 10，如total=15则c[k]=5）；
取总和的高位作为下一位进位（jin = total / 10，如total=15则jin=1）。
此步骤确保结果数组的每一位最终都是 0-9 的单个数字，符合数字表示规则。

高精度乘

O（n^2）复杂度的高精度乘法，本质是**完全模拟竖式乘法的“交叉相乘” 规则**：设大数 A 有 n 位，大数 B 有 m 位，A 的第 i 位（从 0 开始，对应个位）与 B 的第 j 位相乘，结果会贡献到最终结果的第 i+j 位（需叠加所有交叉相乘结果，再处理进位），最终遍历所有 i 和 j 的组合（共 n×m 次计算），因此时间复杂度为 O（nm），当 n≈m 时即为 O（n^2）。

高精度乘


```cpp
#include<bits/stdc++.h>
using namespace std;
int a[100005];  // 存储第一个大数（下标0存个位，逆序）
int b[100005];  // 存储第二个大数（下标0存个位，逆序）
int c[200005];// 存储结果（两个n位数字相乘，结果最多2n位，需双倍长度）
string s1, s2;
int main() {
    cin >> s1 >> s2;
    int n1 = s1.size();  // 第一个大数的位数
    int n2 = s2.size();  // 第二个大数的位数
    // 2. 将字符串逆序存入数组（下标0对应个位，方便按位计算）
    for (int i = 0; i < n1; i++) {
        a[i] = s1[n1 - i - 1] - '0';  // 字符转数字（如'7'-'0'=7）
    }
    for (int i = 0; i < n2; i++) {
        b[i] = s2[n2 - i - 1] - '0';
    }
```

高精度乘


```
// 3. 核心：交叉相乘（O(n1× n2)复杂度，即O(n^2)）
// 遍历第一个数的每一位（i：0~n1-1，对应个位到n1-1位）
for (int i = 0; i < n1; i++) {
    // 遍历第二个数的每一位（j：0~n2-1，对应个位到n2-1位）
    for (int j = 0; j < n2; j++) {
        // A的i位 ×  B的j位，结果累加到C的i+j位（竖式乘法的交叉位规则）
        c[i + j] += a[i] * b[j];
    }
}
// 4. 处理进位（遍历结果数组，逐位整理进位）
int max_len = n1 + n2;  // 两个数相乘的最大可能位数（n1位× n2位=最多n1+n2位）
int jin = 0;            // 进位初始化为0
for (int k = 0; k < max_len; k++) {
    int total = c[k] + jin;  // 当前位总和 = 交叉相乘累加值 + 上一位进位
    c[k] = total % 10;       // 取个位作为当前位结果
    jin = total / 10;        // 取高位作为下一位进位
}
```

高精度乘


```
    // 5. 去除结果的前导0（如结果是00012345，需改为12345；全0则保留1个0）
    // 从结果最大可能位数往前找，直到找到非0数字
    while (max_len > 1 && c[max_len - 1] == 0) {
        max_len--;  // 有效长度减1，跳过前导0
    }
    // 6. 输出结果（结果数组逆序存储，从后往前遍历即高位到低位）
    for (int k = max_len - 1; k >= 0; k--) {
        cout << c[k];
    }

    return 0;
}
```