算法基础（一）:排序，二分，高精度

如何提高能力?

1. 上课理解思想
2. 默写，最主要是思想
    1. 看主要思想
    2. 模板背一遍
    3. 默写，以题为主，模板题
    4. 提高熟练度
        1. 删掉重写，重复三到五次
    5. 需要注意的是，理解一个算法以后也会忘记的，要反反复复默写背过

排序

快速排序

基本思想

1. 找到一个划分的标准元素x，可以是最左边的元素，可以是最右边的元素，也可以是随机的元素，也可以是中间的元素
2. 进行划分，将数组变为左边的元素都<=x, 右边的元素都>=x

    1. 方法一：可以再分配两个数组，扫描后进行合并，共扫描两次（如果忘记方法二的话，直接暴力用这种方法，时间复杂度也是线性的）
    2. 方法二：交换式，两个下标依次向前移动，然后交换
3. 递归处理左右两段

代码实现

```
void quick_sort(int q[], int l, int r)
{
    //递归的终止情况
    if(l >= r) return;
    //第一步：分成子问题
    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    while(i < j)
    {
        do i++; while(q[i] < x);
        do j--; while(q[j] > x);
        if(i < j) swap(q[i], q[j]);
    }
    //第二步：递归处理子问题
    quick_sort(q, l, j), quick_sort(q, j + 1, r);
    //第三步：子问题合并.快排这一步不需要操作，但归并排序的核心在这一步骤
}
```

边界分析

1. 以`j`为划分递归时，`x`不能选择`q[r]`，否则递归会无限划分无法退出，比如数组`1, 2`，若以`j`划分，则开始时`l = 0, r = 1, i = -1,  j = 2`，循环一次后`i = j = 1, l = 0`，此时进入递归`quick_sort(q, l, j)`，`l`仍为0，`r`仍为1，进入无限递归。同理，当以`i`进行递归划分时，`x`不能取`q[l]`
2. 在`while`循环中，不能加上等号`q[i] <= x`因为如果这个数组的`q[l....r]`所有元素都相等的话会导致数组下标越界，看似没有问题，但是如果后面的元素一直`<=x`则最后下标会一直递增，一直到`Memory Limit Exceeded`
3. 第一个`while`循环不能用`i<=j`因为如果数组为`1 2`然后`x = 1`则完成`while`循环后`j = -1, i = 1`，然后再进入第二个递归又是`l = 0, r = 1`进入无限递归，但是若`i < j`则完成`while`循环后`i = j = 0 1`，不会进入无限递归
4. `while`循环中的`if(i < j)`可以改成`if(i <= j)`加上等号也就是再交换一次，没有影响，下一步就直接跳出循环了
5. 递归中若以`j`为划分递归的标准，则不能用`quick_sort(q, l, j - 1), quick_sort(q, j, r);`因为：
    1. 以数组`2 1 2 1 1`为例，划分元素为`q[l] = 2`，则最后得到的数组为`1 1 1 2 2`，`j =2`指向`1`，`i = 3`指向`2`，此时第二个`quick_sort(q, j, r)`递归的数组为`1 2 2`不满足快排的思想：右边的数组必须大于等于`x`
    2. 且`1 2 2`进入第二个递归`quick_sort(q, j, r)`时，始终有`l = j（第一个数的下标，不变），r = n-1`进入无限递归

3. 也可以这样理解，下证`j`的取值范围为`[l...r-1]`
    1. 若`j = r`
        1. 说明外循环只进行了一次就退出了，否则`j`至少会自减两次，且此时`q[j] = q[r] <= x`
        2. 由于只进行了一次外循环，所以`q[l...i-1] < x, q[i] >= x,q[j+1...r] > x, q[j] <= x, j <= i`
        3. 此时`j = r`又由于`while循环结束`，可得`i >= r`，`i`此时不可能为`r+1`（很显然），所以`i = r`，于是由于`do-while语句`可以得到：`p[l...i-1] < x`且`p[i] = p[r] >= x`
        4. 所以此时必有`p[r] = x`但很显然这个命题并不能一直成立，当我们取`x = q[l + r >>1]`的时候就不成立
        5. 所以`j`不会超过r
    2. 若`j < l`
        1. 则由于`do - while`语句，有`p[l...r] > x`显然不成立
    3. 所以`j`的取值范围为`[l..r-1]`
    4. 所以若递归划分为`quick_sort(q, l, j - 1), quick_sort(q, j, r)`时，第二个递归`j`可以取`l`从而进入无限递归，而采用模板中的方法第一个递归传递的`j`始终会减小，第二个`j + 1`始终增大，所以不会进入无限递归

归并排序

基本思想

1. 确定分界点：`mid = (1+r) >> 2`
2. 先进行递归排序
3. 利用双指针算法归并合二为一

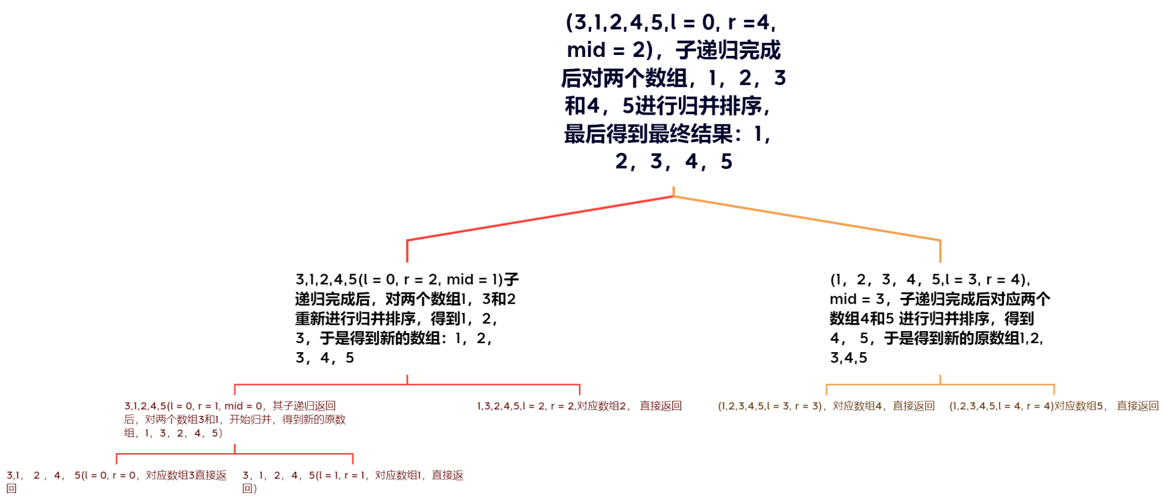
代码实现

```
int q[N], tmp[N];
//给定需要排序的数组以及左右边界
void merge_sort(int q[], int l, int r){
    if(l >= r) return;
    int mid = l + r >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);
    //i, j分别是划分出的两个数组的指针
    //k是临时数组的下标
    //由于归并的时候只是归并数组q[N]的一部分，所以i,j是从l以及mid+1开始的
    int k = 0, i = l, j = mid + 1;
    //进行比较以及归并
    while(i <= mid && j <= r){
        if(q[i] <= q[j]){
            tmp[k++] = q[i++];
        }else{
```

```
                tmp[k++] = q[j++];
            }
        }
        while(i <= mid) tmp[k++] = q[i++];
        while(j <= r) tmp[k++] = q[j++];
        //将归并后的数组写入原来的数组中
        for(i = l, j = 0; i <= r; i++, j++) q[i] = tmp[j];
    }
```

递归过程分析

以`3 1 2 4 5为例`，先看括号内的参数，再从叶子结点自底向上，自左向右分析



二分

整数二分

基本思想

整数二分的目的是：

取中间值`mid`，然后检查`mid`的性质，来找到一组数据中具有不同性质的两组数据的临界点

以二分查找为例：

取的中间值就是`mid`，我们要找的数是`x`那么这个数组中就具有`<x`以及`>=x`两种性质，我们通过检查`mid`的性质：其其为下标的数组元素是否`<x`或者是否`>=x`,来找到这两种性质的分界点`x`，最后返回的结果是分界点的下标

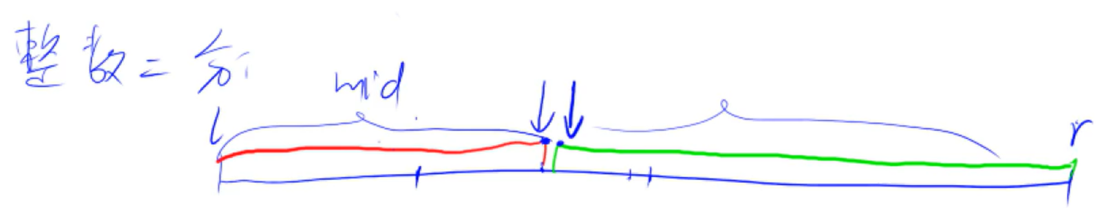
代码实现

```
bool check(int x){
    /*....*/
}//检查x是否满足某种性质
//区间[l, r]被划分为<x 以及 >=x 两种性质的时候，返回的结果一定是左边界最右边的那个值
//check(mid)检查的时候就是检查是否满足左边的性质了
```

```
int bearch_1(int l, int r){
    while(l < r){
        int mid = l + r + 1 >>1;
        if(check(mid)){
            l = mid;
        }else{
            r = mid -1;
        }
    }
    return l;
}

//区间[l, r]被划分为<x 以及 >=x 两种性质的时候，返回的结果一定是右边界最左边的那个值
//即返回1 2 3 3 4 5返回第一个3的下标，也就是左边性质的边界
int bearch_2(int l, int r){
    while(l < r){
        int mid = l + r >> 1;
        //check(mid) 为是否 >= x是为了看看是否满足右边性质返回第一个3的下标
        //check(mid) 其实就规定了划分的性质，这个模板是找到右边性质最左边的值
        //check(mid)若为是否>x，则说明区间倍划分成了<=x, 与>x两个部分，则返回的值就是>x的最左边边界返回4的下标
        //
        if(check(mid)) r = mid;
        else l = mid + 1;
    }
    return l;
}
```

代码理解



红色和绿色分别代表两种性质，在二分查找中， 红色代表小于x，绿色代表大于等于x

注意：

1. 走出循环的时候一定是`l = r`，因为是一点点减一的
2. 这一堆数据中的不同性质的临界点必为两个，我们要根据情况来使用不同的模板来查找，比如在二分查找中，`<x`这种性质的分界点是小于`x`的最大数，`>=x`这种性质的分界点是`>=x`的最小数，所以我们使用二分查找的模板就是去寻找右边性质的临界点
3. 找左边的临界点的时候是用模板二，找右边临界点的时候是用模板一
4. 使用模板一的时候，也就是找左边临界点的时候，由于c语言的出发舍入规则，必须把`mid`设置为`l + r + 1>>2`，比如就两个数，下标为`0, 1`，`l = 0, r = 1, mid = 0`当进入循环的时候，若此时下标`mid`对应的数满足性质则变为`l = 0, r = 1,mid = 0`进入死循环

5. 二分模板是一定可以找到结果的，比如二分查找，哪怕具有`>=x`的性质中没有数x，我们的模板仍然会找到这个性质的临界点: 大于等于x的最小值，至于这个值正不正确是由题目决定的，与我们的模板无关
6. 做题的时候先写`check(mid)`然后再看自己找的是做临界点还是右临界点，若是左临界点，则`mid = l + r + 1 >> 2`
7. 哪怕在右边的性质中的边界点是两个，比如右边的性质是大于等于x，但是数组中有两个x，但是最后的结果仍然是找到边界点，比如数组`1 2 2 3 3 4`，右边的性质是大于等于3，我们用二分来找右边的临界点，找到的数的下标是3，是最左边3

浮点数二分

基本思想

与整数二分类似，不过由于浮点数是连续的，所以边界点就只有一个，所以也就一个模板

代码实现

```cpp
bool check(double x){
    /*.....*/
    //检查x是否满足某种条件
}

double bsearch_3(double l, double r){
    const double eps = 1e-6;//eps表示精度，取决于题目对精度的要求
    while(r - l > eps){
        double mid = (r + l) / 2;
        if(check(mid)){
            r = mid;
        }else{
            l = mid;
        }
    }
    return l;
}
```

代码理解

1. 浮点数二分只有一个边界点，每次取`mid`的时候都是准确取值，精准二分
2. `check(mid)`若满足红色性质，则`r = mid`，若满足绿色性质则`l = mid`

高精度

基本思想

1. 一般而言，大整数的位数是10^6级别
2. 大整数的存储，数的每一位存到数组里面去，且数组的第0位存数个位，因为如果结果有进位的话，直接在高位，也就是数组末尾增加一位要比在数组的开始增加一位方便

加法

基本思想

1. 从个位开始往前算
2. 对于每一位，两个数对应的位相加并加上上一位的进位，若大于十则进一，否则不进一，`Ai + Bi + t`

代码实现

```cpp
#include<iostream>
#include<vector>

using namespace std;

const int N = 1e6 + 10;

vector<int> add(vector<int> &A, vector<int> &B){
    vector<int> C;
    int t = 0;//作为进位,以及中间结果
    //简化代码，不用区分A和B谁的位数大,注意相加有进位的情况，AB，位数相加结束，但是不能退出循环，要处理进位
    for(int i = 0; i < A.size() || i <B.size() || t != 0; i++){
        if(i < A.size()) t += A[i];//t作为中间结果
        if(i < B.size()) t += B[i];

        C.push_back(t % 10);
        t /= 10;//t作为进位
    }
}

int main(){
    string a, b;
    vector<int> A, B;
    //用字符串读入两个整数
    cin >> a >> b;
    //假如a=123456
    //将a存入数组A中
    for(int i = a.size() - 1; i >= 0; i--){
        A.push_back(a[i] - '0');//A中是654321
}
    for(int i = b.size() - 1; i >= 0; i--){
        B.push_back(b[i] - '0');
    }
    //auto自动判断C的类型
    auto C = add(A, B);

    for(int i = C.size(); i >= 0; i--){
        cout<<C[i];
    }
    return 0;

```

```
}
```

减法

基本思想

1. 对于每一位先计算`Ai - Bi - t`，t是上一位的借位，若大于等于0，则结果不变；若小于0，则结果为`Ai - Bi - t + 10`，并向上一位借一位
2. 保证大整数A大于等于B，若A小于B，则交换总是算大数减小数，保证模板的最高位不会向前借位，然后加上负号
3. 这里保证A与B都是正数

代码实现

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

const int 100010;

//判断A与B的大小
bool cmp(vector<int> &A,  vector<int> &B){
    //A与B的位数不相等的情况
    if(A.size() != B.size()){
        return A.size() > B.size();
    }
    //A与B的位数相等的情况，从最高位开始比较
    for(int i = A.size() - 1; i >= 0; i--){
        if(A[i] != B[i]){
            return A[i] > B[i];
        }
    }
    //A = B
    return true;
}

vector<int> sub(vector<int> &A, vector<int> &B){
    vector<int> C;
    //t作为中间变量以及向前的借位
    //保证A一定大于B
    for(int i = 0,t = 0; i < A.size(); i++){
        t = A[i] - t;
        //如果B还有位可以减则减去B，否则不减
        if(i < B.size()){
            t -= B[i];
        }
        //(t + 10) % 10包含了两种情况，若是负数则需要加10，若是正数则结果必不大于10,加10再对十取余刚好把10去掉
        C.push_back((t + 10) % 10)
        //此时t再作为向前的借位
```

```
        if(t < 0){
            t = 1;
        }else{
            t = 0;
        }
    }
    //去掉前导0，由于是减法，高位相减结果为0的时候也会导入C中（比如111-110，结果是001，此时需要去掉前面的两个0
    //C是单个0的时候不去掉这个0，当C的位数大于1并且C的最高位为0的时候就要将这个0去掉
    while(C.size() > 1 && C.back() == 0){
        C.pop_back();
    }
}


int main(){
    string a, b;
    vector<int> A, B;
    cin >> a >> b;

    for(int i = a.size()-1; i >= 0; i--){
        //注意存进去的是数字
        A.push_back(a[i]-'0');
    }
    for(int i = b.size()-1 i >= 0; i--){
        B.push_back(b[i]-'0');
    }

    if(cmp(A,B)){
        auto C = sub(A, B);

        for(int i = C.size()-1; i >= 0; i--){
            printf("%d", C[i]);
        }
    }else{
        printf("-");
        for(int i = C.size()-1; i >= 0; i--){
            printf("%d", C[i]);
        }

    }
    return 0;
}
```

乘法

基本思想

1. 运算的对象是一个大整数A与一个正常的整数b
2. 从个位出发，$C_i = (A_i * b + t_i)$ ，$t_{i+1} = (A_i * b + t_i)/10$，$t_i$是上一位向这一位的进位，$t_{i+1}$是这一位向下一位的进位

代码实现

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

vector<int> mul(vector<int> &A, int &b){
    vector<int> C;
    int t = 0;
    //注意从个位开始计算
    for(int i = 0; i <= A.size() ; i++){
        t = A[i]*b + t;
        C.push_back(t % 10);
        t /= 10;
    }
    //当最高位进位不为0的时候仍然需要处理t
    while(t){
        C.push_back(t % 10);
        t /= 10;
    }
    //去掉前导0，1111*0的情况
    while(C.size() > 1 && C.back() == 0){
        C.pop_back();
    }
    return C;
}

int main(){
    string a;
    int b;
    cin >> a >> b;
    vector<int> A;
    for(int i = a.size() - 1; i >= 0; i--){
        A.push_back(a[i] - '0');
    }

    auto C = mul(A, b);
    for(int i = C.size() - 1; i >= 0; i--){
        printf("%d", C[i]);
    }

    return 0;
}
```

除法

基本思想

1. 除法仍然是一个大整数除以一个正常整数，求得得商是C，余数是r
2. 除法是从最高位开始算，但是仍然是从最低位开始存
3. 对于A得第`i`位`Ai`，先是前面得到得余数r*10，然后加上`Ai`得到中间值，这个中间值再除以`b`得到的商作为结果的商的一位，得到的余数作为下一位的余数，一直到`Ai`处理完，得到的商其实是正常存放在结果中的，所以需要`reverse`一下，不需要反序输出，但需要处理前导0
4. 最开始的时候r = 0, A的第一位A1，中间值为r * 10 + A1，然后除以b的商作为结果的商的第一位，余数作为下一位的余数

代码实现

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

vector<int> div(vector<int> &A, int b, int &r){
    r = 0;
    //r同时作为余数和中间结果
    for(int i = A.size() - 1; i >= 0; i--){
        r = r*10 + A[i];//r作为中间结果
        C.push_back(r / b);
        r %= b;//r再作为下一位的余数
    }
    //reverse函数将C反转一下
    reverse(C.begin(), C.end());
    //去掉前导0
    while(C.size() > 1 && C.back() == 0){
        C.pop_back();
    }
    return C;
}



int main(){
    string a;
    int b;
    cin >> a >> b;
    vector<int> A;
    for(int i = a.size() - 1; i >= 0; i--){
        A.push_back(a[i]-'0');
    }
    int r;
    auto C = div(A, b, r);
```

```
    for(int i = C.size() - 1; i >= 0; i--){
        printf("%d", C[i]);
    }
    cout << endl;
    cout << r;
    return 0;
}
```