代码的效率问题

首先依然是注意，测试请用 linux 下的 g++或者 WIN 下的 VS。

1、 字符串的输入，string 与 字符数组

应该有很多同学都遇到过这个问题，读取字符串的时候用 string 就超时，用字符数组就过了：

```
string s;
cin >> s;
```

```
char s[1001];
cin >> s;
```

这两种方式的效率差别有那么大么？接下来我们做一个实验：

由于通过键盘手动输入会影响时间，所以我们采用从文件读入：

先用程序来生成一个文件，等下我们就用这个文件来测试读取效率。为了看出时间差别，这个文件最好大一点，我这里生成了一个 10000 行的文件 input.txt

```
char s[1001];
for (int i=0; i<1000; i++)
    s[i] = 'a' + rand() % 26;

ofstream outFile("input.txt");
if (!outFile)
{
    cout << "cannot open the file: input.txt" << endl;
    return -1;
}

for (int i=0; i<10000; i++)
{
    int len = rand() % 1000 + 1;
    for (int i=0; i<len; i++)
        outFile << s[i];
    outFile << endl;
}

outFile.close();
```

接下来我们来测试一下分别用 string 跟字符数组来读取它需要的时间：

```
    ifstream inFile("input.txt");
    if (!inFile)
    {
        cout << "cannot open the file: input.txt" << endl;
        return -1;
    }

    clock_t startTime = clock();

    string s;                   // 测试字符数组就把这行改成char s[1001];
    while (inFile >> s) ;

    cout << (double)(clock() - startTime) / CLOCKS_PER_SEC << "s" << endl;

    inFile.close();
```

分别运行两种情况，然后对比一下运行时间。

接下来该分析为什么会有这种差异了：
我们先来看一下 string 里面怎么处理读入字符串读入的，也就是 string 里面是如何实现>>操作符的重载的，我知道你不一定能找到源码，所以这里我直接告诉你它的思路，完整源码你可以去看 string 里面的这个函数：

```
template<class _Elem,
    class _Traits,
    class _Alloc> inline
    basic_istream<_Elem, _Traits>& operator>>(
        basic_istream<_Elem, _Traits>&& _Istr,
        basic_string<_Elem, _Traits, _Alloc>& _Str)
```

从函数名就可以看出它的功能是将输入流_Istr 的内容读取到字符串_Str 中，大概步骤如下：

1、 判断输入流的状态是否正常，如果正常则进入下一步开始读入
2、 清空字符串_Str 的内容（就跟你们自己实现的 string 一样，它并没有删除它占有的空间，只有在析构的时候才删除）
3、 从输入流缓冲区中读取每个字符（每次读取一个，直到读取结束）：
    a) 如果该字符是 EOF 标识符，则记录当前状态为 EOF 状态
    b) 如果该字符是空白符（比如空格、制表符之类的），则跳出循环，当前字符串读取结束
    c) 如果既不是 EOF 又不是空白符，就是可以存入字符串的字符了，这时候_Str 调用它的 append 方法将该字符添加到字符串里面
4、 接下来就是一些标记状态的处理了，最后返回这个已经处理过的输入流，所以才能连续 cin 两个字符串：cin >> s1 >> s2;

看到这里，如果你还记得曾经你的 string 是如何实现的，那么我猜你应该能够猜到

效率为什么差别那么大了。
还不懂？那我再提示一下，string 里面的 append 方法你是如何实现的？
想到了你就基本上找到答案了，等我讲完字符数组的方式再贴答案。

接下来我们看如果是字符数组，cin 操作会是怎样的：
这个符号重载是在 istream 里面的，函数声明如下：

```
template<class _Elem,
    class _Traits> inline
    basic_istream<_Elem, _Traits>& operator>>(
        basic_istream<_Elem, _Traits>& _Istr, _Elem *_Str)
```

这里的处理就很简单了：
1、 判断输入流的状态是否正常，如果正常则进入下一步开始读入
2、 从输入流缓冲区中读取每个字符（每次读取一个，直到读取结束）：
   a) 如果该字符是 EOF 标识符，则记录当前状态为 EOF 状态
   b) 如果该字符是空白符（比如空格、制表符之类的），则跳出循环，当前字符串读取结束
   c) 如果既不是 EOF 又不是空白符，就是可以存入字符串的字符了，这时候直接把字符接到当前 _Str 的后面

这里要注意两点，传进来的字符串 _Str 的大小，也就是字符数组的大小是未知的，我们根本就没有传入这个参数。
那么你可能要问，假如我是 char s[10]; 然后用它来读入 15 个字符的字符串，可行么？按照这里写的代码，它不就是数组越界了么？
对的，你可以试试，最好用 debug 模式，用 release 模式运行时可能不会报错。

好的，该揭晓我们的答案了：
这里的差别就在于，字符数组的字符是直接塞在那个字符数组指针后面的，而 string 那边是调用 append 函数，在 append 函数里面呢，我们需要做一些判断，比如有没有超出当前 string 里面的字符数组的容量啊，如果超出了就要增加分配空间...

你要说，哦！我懂了，重新分配空间造成了这效率差距么？
那你就大错特错了...不信？自己写个程序测试一下嘛...
还是刚刚那个输入文件，还是从文件读入，这时候我们先把字符串的 size 设置为大于 1000 的数值，string 里面有个 resize 函数：s.resize(1005);
在前面那个代码里面的 string s; 后面加多这行，然后你可以先输出 s.capacity();
看看它是不是真的大于 1000 了。
然后接下来的程序主体不变，这时候再运行一下，看程序的运行时间。
看完之后我猜你就知道结果了...

那么问题到底在哪里？
其实问题很简单，用 string 比用字符数组多了一些判断，它读入一个字符就要判断一次是否超出数组的容量（当然实际上 string 里面的实现可不仅仅是一次判断哦，它还要做一些额外的判断，有兴趣可以自己看看 string 里面是怎么实现的）。
这里我们就当成是只多了一次判断而已，那么想想刚刚我们生成的文件，总共 1W

行，每行是 1~1000 之间的随机数，平均下来就是每行 500 个字符，也就是说总共有 500W 个字符，每个字符一次判断，总共就多了 500W 次判断...

问题又来了...500W 次判断就需要那么久么？
当然是不可能的...你自己写一个程序试试不就好了，肯定不超过 0.1s...
那么问题的关键究竟在哪里？
其实关键在于函数调用，从源码中，我们可以看到它读取一个字符至少多了这么多次函数调用（缩进表示调用层次）：

```
_Str.append(1, _Traits::to_char_type(_Meta));
    _Chassign(this->_Mysize, _Count, _Ch);
        _Traits::assign(*(this->_Myptr() + _Off), _Ch);
            return (this->_BUF_SIZE <= this->_Myres ? _STD addressof(*this->_Bx._Ptr)
                : this->_Bx._Buf);
    _Eos(_Num);
        _Traits::assign(this->_Myptr()[this->_Mysize = _Newsize], _Elem());
            return (this->_BUF_SIZE <= this->_Myres ? _STD addressof(*this->_Bx._Ptr)
                : this->_Bx._Buf);
```

好吧，算算至少都有 11 个了...
你就写个程序测试一下 11 个函数调用...500W 次需要多久...
而且还不知道里面还有多少层函数调用...我也懒得去看太多...
好了，大概原因就是这样，想深究的请自己看源码去吧...

不要问什么为什么 string 这种效率人家还要用...这种问题真的很 low...人家没事要写那么多额外操作作甚...肯定是为了安全跟方便复用啊...你用字符数组然后每次都去很注意内存？然后再一不小心程序就崩掉了？

2、 代码的不必要操作

这个问题很好说了，不必要的重复操作什么的都放循环外面去就好了，自己写代码的时候注意一下就搞定了...
这里我就挑两份代码出来给大家解释一下大概是怎么样的问题：

```
int main() {
    string la, bi;
    while (cin >> la) {
        cin >> bi;
        int len1 = la.length();
        int len2 = bi.length();
        int **map = new int*[len1];
        for (int i = 0; i < len1; i++) map[i] = new int[len2];
        // 处理过程就省略了
        cout << map[len1-1][len2-1] << endl;
    }
  return 0;
}
```

代码主体我们就不看了，管它对不对，反正肯定是过不了的...
我们只看我标绿底的那两行，它们是在循环里面的，也就是说，每一个用例我都得去为它分配空间。
这里有两个问题：

a. 自己 new 的空间没有 delete，动态内存分配需要自己回收内存，否则...内存泄露...
b. 需要每个测试用例都重新分配内存空间么？上一个用例输出之后，它的空间是不是就没用了，是不是可以继续给下一个测试用例使用？
   改进方式：直接用 int map[1001][1001]来分配内存，或者自己动态分配一次，记得自己 delete 掉（如果是静态分配，就只能作为全局变量或者静态变量，具体请自己百度以下关键词：C++ 大数组 全局变量）。

接下来我们看第二份代码：
它用的也是 string，然后两重循环来求矩阵的数值是这样写的，其中 fir 表示第一个字符串，sec 表示第二个字符串：

```
for (i = 0; i < fir.length(); i++) {
            for (j = 0; j < sec.length(); j++) {
```

好吧，这里就是刚刚强调过的函数调用问题...你这样写就是每次循环判断条件都要去调用一下那个 sec.length()函数...
不信？可以自己单步调试看看嘛...
还不信？可以自己写个程序测试一下先把两个 length 保存到一个变量里面再循环，看看时间差多少...

最后送大家一句话...写代码的习惯直接关系到代码的效率，而不仅仅是所谓的算法...光去注重什么算法，写不出好代码也是没用的...（请注意我没有说算法不重要...要是不重要就不会有那么多大神玩算法了...算法肯定是重要的啊！）

别问什么 TA 为什么你知道这么多...

string 那个我本来也是不知道的...不过你们遇到这个问题的时候我就猜到大概是这种原因了（再敢问为什么我会猜到这种原因上去...非要逼我用万能的智商来堵住你的问题么）...但是最开始又懒得自己写，于是就去找谷歌度娘...可是他们都不成全我...最后我就只能自己看着源码自己写了...（怎么看源码，单步调试跳进去然后你就可以看个够了...）

接下来附上我在找谷歌度娘的时候碰到的一些额外的收获，有兴趣的同学可以自己去看看了解一下：

探寻 C++最快的读取文件的方案
https://www.byvoid.com/blog/fast-readfile
关于 C++中的流缓冲
http://blog.csdn.net/u012333003/article/details/27181817

让你们多去找谷歌度娘是因为你自己查找资料的时候就有可能找到一些额外的收获...当然花的时间肯定是比别人直接告诉你答案要多得多咯...但是暂且不说人家知不知道其他额外的东西，就算人家知道，你又没问，我怎么知道你会不会...难道我还追着你问，这个你会不会啊，那个你会不会啊...再退一步...就算人家想追着问...估计也没想到那个点啊...