求二叉树的最大深度，也即其高度。

递归版本比较容易理解。利用层次遍历非递归求二叉树高度主要的思想是：**一层一层地出队列** — 在我们每次访问完毕一层时，这时队列中存储的刚好是下一层的所有元素。所以在下一次循环开始时，首先记录该层的元素个数，一次性访问完这一层的所有元素。



比如求树的宽度，也就是元素个数最多的那一层元素个数。记录所有层间的最大值即可。这题是求二叉树的最小高度，与最大高度类似，不同之处在于层次遍历时，如果在该层遇到有叶子节点则立即返回，在进行层次遍历时需要进行叶子节点的判断。



给定一个数组,输出除了本位置数字以外其余数的乘积

1. 一个长度为n的整数数组result，满足result[i]=除input[i]之外所有数的乘积（不溢出），比如
2. \* 输入input={2,3,4,5};输出 result={60,40,30,24};

：判断有0的情况，如果有0则其他都为0.如果没0,可使用先求全部乘积，再除以自身。

找出数组中第k大小的数，输出数所在的位置。

快速排序，快排中的每一步，都是将待排数据分做两组，其中一组的数据的任何一个数都比另一组中的任何一个大，然后再对两组分别做类似的操作，然后继续下去……

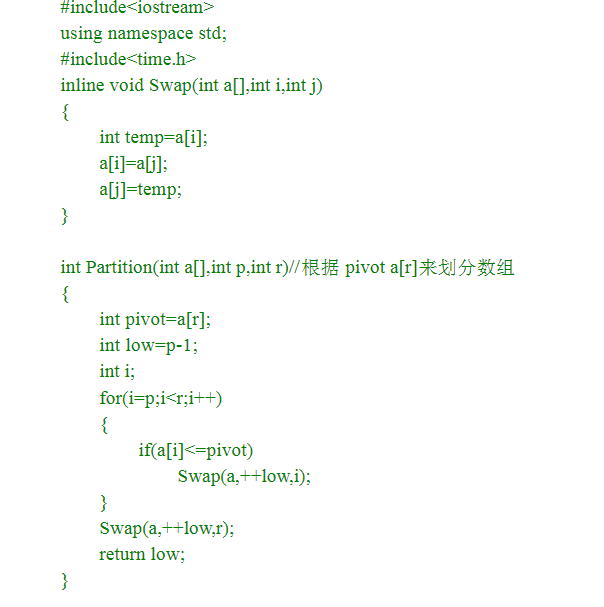
在本问题中，假设 N 个数存储在数组 S 中，我们从数组 S 中随机找出一个元素 X，把数组分为两部分 Sa 和 Sb。Sa 中的元素大于等于 X，Sb 中元素小于 X。

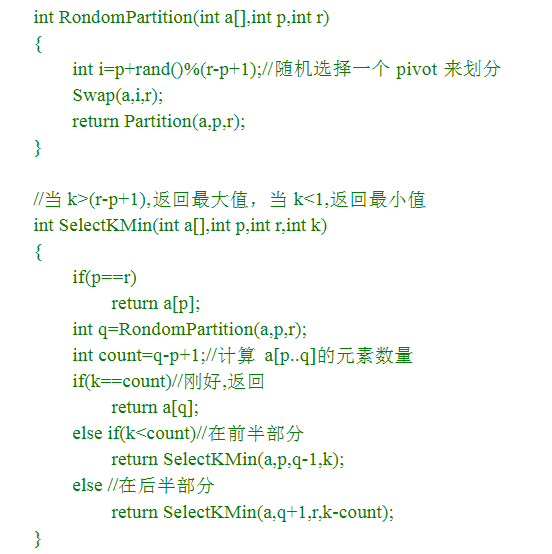
这时，有两种可能性：

1. Sa中元素的个数小于K，Sa中所有的数和Sb中最大的K-|Sa|个元素（|Sa|指Sa中元素的个数）就是数组S中最大的K个数。

2. Sa中元素的个数大于或等于K，则需要返回Sa中最大的K个元素。

这样递归下去，不断把问题分解成更小的问题，平均时间复杂度 O（N \*log2K）。伪代码如下：





****最长递增子序列****

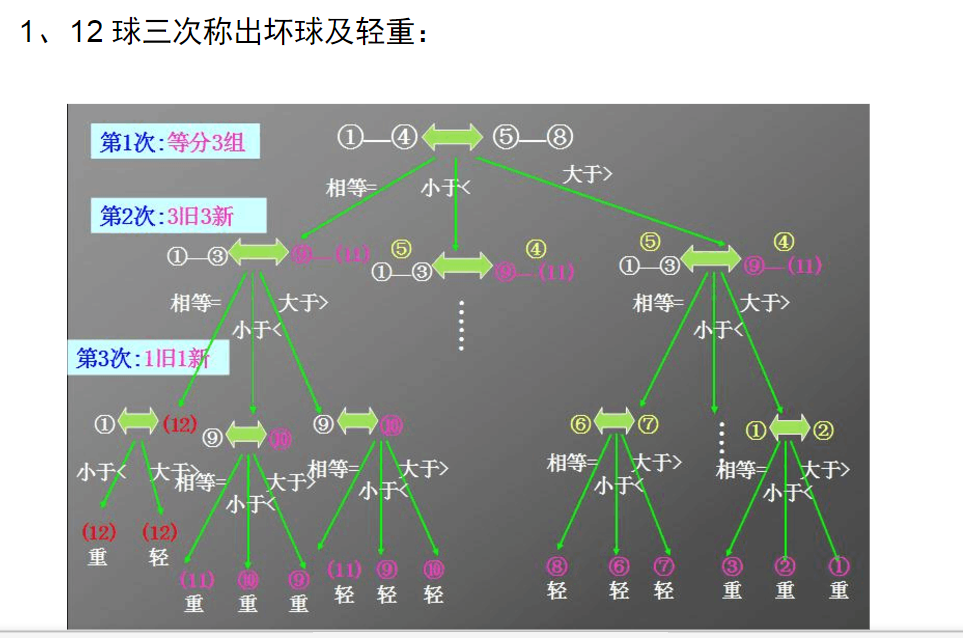
排序+LCS求解

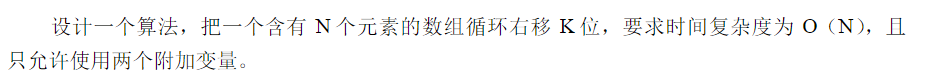
这个方法也是很直观的，对原数组a进行排序得到一个有序的数组b， 这样出现在数组a的最长递增子序列也一定是数组b的子序列,转化为最长公共子序列问题

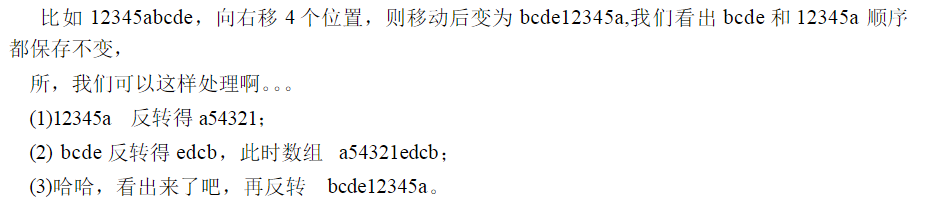
1. **int** LCS(**int** \* arr, **int** \* arrcopy, **int** len)
2. {
3. **for**(**int** i = 1; i <= len; ++i)
4. {
5. **for**(**int** j = 1; j <= len; ++j)
6. {
7. **if**(arr[i-1] == arrcopy[j-1])
8. {
9. dp[i][j] = dp[i-1][j-1] + 1;
10. }**else** **if**(dp[i-1][j] > dp[i][j-1])
11. {
12. dp[i][j] = dp[i-1][j];
13. }**else**
14. {
15. dp[i][j] = dp[i][j-1];
16. }
17. }
18. }
19. **return** dp[len][len];
20. }

归并排序

1. //将有二个有序数列a[first...mid]和a[mid...last]合并。
2. **void** mergearray(**int** a[], **int** first, **int** mid, **int** last, **int** temp[])
3. {
4. **int** i = first, j = mid + 1;
5. **int** m = mid,   n = last;
6. **int** k = 0;
8. **while** (i <= m && j <= n)
9. {
10. **if** (a[i] <= a[j])
11. temp[k++] = a[i++];
12. **else**
13. temp[k++] = a[j++];
14. }
16. **while** (i <= m)
17. temp[k++] = a[i++];
19. **while** (j <= n)
20. temp[k++] = a[j++];
22. **for** (i = 0; i < k; i++)
23. a[first + i] = temp[i];
24. }
25. **void** mergesort(**int** a[], **int** first, **int** last, **int** temp[])
26. {
27. **if** (first < last)
28. {
29. **int** mid = (first + last) / 2;
30. mergesort(a, first, mid, temp);    //左边有序
31. mergesort(a, mid + 1, last, temp); //右边有序
32. mergearray(a, first, mid, last, temp); //再将二个有序数列合并
33. }
34. }
36. **bool** MergeSort(**int** a[], **int** n)
37. {
38. **int** \*p = **new** **int**[n];
39. **if** (p == NULL)
40. **return** **false**;
41. mergesort(a, 0, n - 1, p);
42. **delete**[] p;
43. **return** **true**;
44. }







## **输入两个整数序列。其中一个序列表示栈的push顺序,判断另一个序列有没有可能是对应的pop顺序。**

思路：

1.首先新建一个栈模拟入栈入栈，都是在push序列中进行。

2.将push序列依次开始入栈，直到栈顶元素等于pop序列的第一个元素。

3.push的栈顶元素出栈，pop序列移到第二元素。

4.如果push栈顶继续等于pop序列现在的元素，则继续出栈并pop后移。

5.如果push已经全部入栈但是pop序列未遍历结束，且栈顶元素不等于现在所指元素则返回false。

bool isPopSeries(int push[],int pop[],int length)//网上流传的经典答案

{

if(!push||!pop||length<=0)

return false;

stack<int> temp;

int pushNum=0;

int i = 0;

while(i < length)

{

while(temp.empty()||temp.top()!=pop[i])

{

if(pushNum < length)

temp.push(push[pushNum++]);

else

return false;

}

if(!temp.empty()&&temp.top()==pop[i])

{

temp.pop();

i++;

}

}

return true;

}

两个非降序链表的并集,1->2->3 和 2->3->5 并为 1->2->3->5

1. **int** OutCombine(Node\* p1, Node\* p2)
2. {
3. **while**(p1!=NULL && p2!=NULL)
4. {
5. **if**(p1->value>p2->value)
6. {
7. cout<<p2->value<<" ";
8. p2=p2->pNext;
9. }
10. **else** **if**(p1->value<p2->value)
11. {
12. cout<<p1->value<<" ";
13. p1=p1->pNext;
14. }
15. **else**
16. {
17. cout<<p1->value<<" ";
18. p1=p1->pNext;
19. p2=p2->pNext;
20. }
21. }
22. **if**(p1!=NULL)//剩余的部分直接添加到末尾。
23. {
24. **while**(p1!=NULL)
25. {
26. cout<<p1->value<<" ";
27. p1=p1->pNext;
28. }
29. cout<<endl;
30. }
31. **else** **if**(p2!=NULL)
32. {
33. **while**(p2!=NULL)
34. {
35. cout<<p2->value<<" ";
36. p2=p2->pNext;
37. }
38. cout<<endl;
39. }
40. **return** 0;
41. }

**题目描述：**

如何对n个数进行排序，要求时间复杂度O(n)，空间复杂度O(1)

**解析：**

利用计数排序法，设置一大小为65536的int数组，范围a[0]~a[65535]，并初始为0，然后遍历n个数，假设这n个数在数组array[0...n-1]中，则i取值从0到n-1同时执行a[array[i]]++，最后再依照顺序读数组a，遇到不为0时，将对应的下标读回数组array，计数是几次就读几次，覆盖原有数，这样得出的array即为排序所求

因为空间复杂度大小已知，为65536，执行循环次数约为n+65536 ，所以其空间复杂度为O(n)，空间复杂度O(1)

1. 输入一个正数 n，输出所有和为 n 连续正数序列   
           例如输入 15，由于 1+2+3+4+5=4+5+6=7+8=15，所以输出 3 个连续序列 1-5、4-6 和 7-8。   
           分析：我们用两个数small和big分别表示序列的最小值和最大值。首先把small初始化为 1，big初始化为 2。如果从small到big的序列的和大于n的话，我们向右移动small，相当于从序列中去掉较小的数字。如果从small到big的序列的和小于n的话，我们向右移动big，相当于向序列中添加big的下一个数字。一直到small等于(1+n)/2，因为序列至少要有两个数字。
2. void PrintSequence(int small,int big)
3. {
4. for (int i=small;i**<big**;i++)
5. {
6. cout**<<i<<**"-";
7. }
8. cout**<<big<<endl**;
9. }
11. //寻找和为n的连续序列
12. void FindContinuesSequence(int n)
13. {
14. int small=1;
15. int big=2;
16. int middle=(1+n)/2;
17. int sum=small+big;
18. while(small**<middle**)
19. {
20. //sum==n，直接输出序列
21. if(sum==n)
22. PrintSequence(small,big);
23. //sum**>**n，small向右移，直至sum**<**=n
24. while(sum**>**n)
25. {
26. sum-=small;
27. small++;
28. if(sum==n)
29. PrintSequence(small,big);
30. }
31. //sum**<n**，big向右移
32. big++;
33. sum+=big;
34. }
35. }

2、输入两个整数 n 和 m，从数列1，2，3.......n 中 随意取几个数,使其和等于 m ,要求将其中所有的可能组合列出来.。

使用回溯算法，代码如下：

1. #define N 1000
2. int a[N];  //辅助数组，用来存储满足条件的组合
3. int c=0, n=20, m=30;
5. void work(int sum, int cc)
6. {
7. //和等于m，递归终止，输出组合
8. if(sum == m)
9. {
10. for(int i = 0; i **<** **c**; ++i)
11. printf("%d ", a[i]);
12. printf("\n");
13. return;
14. }
16. //数列从1到n遍历一遍
17. for(int i = cc; i **<**= n; ++i)
18. {
19. //和大于m，此次递归终止
20. if(sum + i **>** m)
21. return;
23. //和小于m，继续递归
24. if(sum + i **<**= m)
25. {
26. a[c++] = i;
27. work(sum + i, i + 1);
28. --c;   //回溯控制
29. }
30. }
31. }
33. int main()
34. {
35. work(0, 1); //初始条件（和初始值为0，数列起始值为1）
37. return 0;
38. }

输入两个字符串，从第一字符串中删除第二个字符串中所有的字符。

例如，输入”They are students.”和”aeiou”，则删除之后的第一个字符串变成”Thy r stdnts.”。

思路：不可避免的是遍历第一个字符串，如果遍历一个字符，都需要去第二个字符串中查找其存不存在，那么复杂度会是O(nm)，当然由于字符数有限，所以m是个常量。关于查找速度最快的当然是hash表，对于8位字符，size=2^8足矣。

关于删除字符，后面的字符要往前移，如果每删除一个就移一次，O(n^2)这复杂度实在太高，仅仅用快慢指针就可以搞定，这个方法非常有用，比如求解循环链表。

初始化：快慢指针指向第一个字符

循环：如果快指针指的是不需要的字符，将值赋给慢指针后，快慢指针同时++;如果快指针指向待删除字符，那么直接++；

终止：快指针指向'\0'

1. #include <iostream>
3. #define NUMBER 256
5. **using** **namespace** std;
7. **char** firstArray[] = "They are students.";
8. **char** secondArray[] = "aeiou";
9. **const** **int** firstSize = **sizeof** firstArray / **sizeof** \*firstArray;
10. **const** **int** secondSize = **sizeof** secondArray / **sizeof** \*secondArray;
12. **bool** flag[NUMBER];
14. **void** deleteArray(**char** \*firstArray, **char** \*secondArray)
15. {
16. **if**(firstArray == NULL || secondArray == NULL)
17. **return**;
19. **for**(**int** i = 0; i < NUMBER; i++) {
20. flag[i] = **false**;
21. }
23. **for**(**int** i = 0; i < secondSize; i++) {
24. **int** pos = **static\_cast**<**int**>(secondArray[i]);
25. flag[pos] = **true**;
26. }
28. **char** \*fast = firstArray, \*slow = firstArray;
30. **while**(\*fast != '\0') {
31. **if**(flag[\*fast] == **false**) {
32. \*slow = \*fast;
33. slow++;
34. }
35. fast++;
36. }
37. \*slow = 0;
38. }

用递归颠倒一个栈。例如输入栈{1, 2, 3, 4, 5}，1 在栈顶。颠倒之后的栈为{5, 4, 3, 2, 1}，5 处在栈顶

分析：乍一看到这道题目，第一反应是把栈里的所有元素逐一pop出来，放到一个数组里，然后在数组里颠倒所有元素，最后把数组中的所有元素逐一push进入栈。这时栈也就颠倒过来了。颠倒一个数组是一件很容易的事情。不过这种思路需要显示分配一个长度为O(n)的数组，而且也没有充分利用递归的特性。

我们再来考虑怎么递归。我们把栈{1, 2, 3, 4, 5}看成由两部分组成：栈顶元素1和剩下的部分{2, 3, 4, 5}。如果我们能把{2, 3, 4, 5}颠倒过来，变成{5, 4, 3, 2}，然后在把原来的栈顶元素1放到底部，那么就整个栈就颠倒过来了，变成{5, 4, 3, 2, 1}。

接下来我们需要考虑两件事情：一是如何把{2, 3, 4, 5}颠倒过来变成{5, 4, 3, 2}。我们只要把{2, 3, 4, 5}看成由两部分组成：栈顶元素2和剩下的部分{3, 4, 5}。我们只要把{3, 4, 5}先颠倒过来变成{5, 4, 3}，然后再把之前的栈顶元素2放到最底部，也就变成了{5, 4, 3, 2}。

至于怎么把{3, 4, 5}颠倒过来……很多读者可能都想到这就是递归。也就是每一次试图颠倒一个栈的时候，现在栈顶元素pop出来，再颠倒剩下的元素组成的栈，最后把之前的栈顶元素放到剩下元素组成的栈的底部。递归结束的条件是剩下的栈已经空了。这种思路的代码如下：

// Reverse a stack recursively in three steps:// 1. Pop the top element// 2. Reverse the remaining stack// 3. Add the top element to the bottom of the remaining stack

template<typename T> void ReverseStack(std::stack<T>& stack)

{

if(!stack.empty()) {

T top = stack.top();

stack.pop();

ReverseStack(stack);

AddToStackBottom(stack, top);

}

}

我们需要考虑的另外一件事情是如何把一个元素e放到一个栈的底部，也就是如何实现AddToStackBottom。这件事情不难，只需要把栈里原有的元素逐一pop出来。当栈为空的时候，push元素e进栈，此时它就位于栈的底部了。然后再把栈里原有的元素按照pop相反的顺序逐一push进栈。

注意到我们在push元素e之前，我们已经把栈里原有的所有元素都pop出来了，我们需要把它们保存起来，以便之后能把他们再push回去。我们当然可以开辟一个数组来做，但这没有必要。由于我们可以用递归来做这件事情，而递归本身就是一个栈结构。我们可以用递归的栈来保存这些元素。

基于如上分析，我们可以写出AddToStackBottom的代码：

// Add an element to the bottom of a stack:

template<typename T> void AddToStackBottom(std::stack<T>& stack, T t)

{

if(stack.empty()) {

stack.push(t);

}

else {

T top = stack.top();

stack.pop();

AddToStackBottom(stack, t);

stack.push(top);

}

}