## Reorder List：(快慢指针+反转链表) // 回文链表题目

Given a singly linked list L: L0→L1→…→Ln-1→Ln,

reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→…

* 主要思路：快慢指针找到中间节点，将后面的链表反转（前插法），合并链表

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

//reorder list

class Solution{

public:

ListNode\* findMiddle(ListNode\* head)

{

ListNode\* slow=head;

ListNode\* fast = head->next;

while ( fast && fast->next)

{

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

ListNode\* reverse\_list(ListNode\*head)

{

ListNode\* pre = head;

ListNode\* next = head->next;

ListNode\* cur = next;

pre->next = NULL;

while (cur)

{

next = cur->next;

cur->next = pre;

pre = cur;

cur = next;

}

return pre;

}

void reorderList(ListNode \*head) { //void

if (head==NULL||head->next==NULL||head->next->next==NULL)

{

return;

}

//快慢指针找到中间节点，将后面的链表反转（前插法），合并链表

//另：题目要求是就地解决，应该是不能用辅助栈之类的

ListNode\* middel = findMiddle(head);

//反转链表

ListNode\* last = reverse\_list(middel->next);

middel->next = NULL;

ListNode\* temp = NULL;

ListNode\* cur = head;

while (last) //防止形成环 middel->next = NULL;

{

temp = last->next;

last->next = cur->next;

cur->next = last;

last = temp;

cur = cur->next->next;

}

return;

}

};

## 2. Implement strStr()

class Solution\_28 {

public:

// find\_first\_of()在源串中从位置pos起往后查找，只要在源串中遇到一个字符，该字符与目标串中任意一个字符相同，就停止查找，返回该字符在源串中的位置；若匹配失败，返回npos。

// string查找find()函数，都有唯一的返回类型，那就是size\_type，即一个无符号整数（按打印出来的算）。若查找成功，返回按查找规则找到的第一个字符或子串的位置；若查找失败，返回npos，即-1（打印出来为4294967295）

int strStr(string haystack, string needle) { // 字符串匹配

int ret = haystack.find(needle);

return ret;

}

char \*strStr1(char \*haystack, char \*needle) {

int len1 = strlen(haystack);

int len2 = strlen(needle);

if (len1 < len2)

{

return NULL;

}

if (len2 == 0)

{

return haystack;

}

int i = 0;

for (; i < len1 - len2 + 1; i++)

{

int j = 0;

while (haystack[i+j]==needle[j])

{

if (j == len2 - 1)

{

return haystack + i;

}

j++;

}

}

return NULL;

}

void getNextval(char\*p,vector<int>& next)

{

int len = strlen(p);

next[0] = -1;

int k = -1; //前缀序列

int j = 0;

while (j < len)

{

if (k==-1||p[j]==p[k])

{

j++; k++;

if (p[j]!=p[k])

{

next[j] = k;

}

else

{

next[j] = next[k];

}

}

else

{

k = next[k];

}

}

}

char \*strStr(char \*haystack, char \*needle)

{

int len1 = strlen(haystack);

int len2 = strlen(needle);

int i = 0, j = 0;

vector<int> next(128,0);

getNextval(needle, next);

while (i<len1&&j<len2)

{

if (j==-1||haystack[i]==needle[j])

{

i++; j++;

}

else

{

j = next[j];

}

}

if (j==len2)

{

return haystack + i-j;

}

return NULL;

}

};

};

## 3.树的遍历

//先序遍历

void PreOrderTraversal(BinTree BT)

{

if (BT)

{

printf("%5d", BT->Data);

PreOrderTraversal(BT->Left);

PreOrderTraversal(BT->Right);

}

return;

}

//先序非递归实现

void PreOrderTraversal\_(BinTree BT)

{

BinTree T = BT;

stack<BinTree> s; //创建并初始化堆栈

while (T || !s.empty()) //树不为空或者栈不为空，继续循环

{

while (T) //一直向左并将沿途节点压入堆栈中

{

s.push(T);

printf("%5d", T->Data);//第一次遇见节点的时候访问

T = T->Left;

}

if (!s.empty())

{

T = s.top(); //先记录，在出栈

//printf("%d", T->Data);

s.pop();

T = T->Right; //转向右子树

}

}

return;

}

//中序遍历

void InOrderTraversal(BinTree BT)

{

if (BT)

{

InOrderTraversal(BT->Left);

printf("%5d", BT->Data);

InOrderTraversal(BT->Right);

}

return;

}

//中序非递归实现

void InOrderTraversal\_(BinTree BT)

{

BinTree T = BT;

stack<BinTree> s; //创建并初始化堆栈

while (T || !s.empty()) //树不为空或者栈不为空，继续循环

{

while (T) //一直向左并将沿途节点压入堆栈中

{

s.push(T);

T = T->Left;

}

if (!s.empty())

{

T = s.top(); //先记录，在出栈

printf("%5d", T->Data); //第二次遇见节点的时候访问 //第三次呢？？？

s.pop();

T = T->Right; //转向右子树

}

}

return;

}

//后序遍历

void PostOrderTraversal(BinTree BT)

{

if (BT)

{

PostOrderTraversal(BT->Left);

PostOrderTraversal(BT->Right);

printf("%5d", BT->Data);

}

return;

}

//要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点P，先将其入栈。如果P不存在左孩子和右孩子，

//则可以直接访问它；或者P存在左孩子或者右孩子，但是其左孩子和右孩子都已被访问过了，则同样可以直接访问该结点。

//若非上述两种情况，则将P的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子前面被访问，

//左孩子和右孩子都在根结点前面被访问。

void PostOrderTraversal\_2(BinTree BT)

{

stack<BinTree> S;

BinTree cur;

BinTree pre = NULL;

S.push(BT);

while (!S.empty())

{

cur = S.top();

if ((cur->Left == NULL&&cur->Right == NULL) || (pre != NULL && (pre == cur->Left||pre==cur->Right)))

{

printf("%5d", cur->Data); //如果当前结点没有孩子结点或者孩子节点都已被访问过

S.pop();

pre = cur;

}

else

{

if (cur->Right != NULL) //将P的右孩子和左孩子依次入栈

{

S.push(cur->Right);

}

if (cur->Left!=NULL)

{

S.push(cur->Left);

}

}

}

}

//postorder traversal  
//\* 核心思想是用栈做辅助空间，先从根往左一直入栈，直到为空，然后判断栈顶元素的右孩子，如果不为空且未被访问过，  
//\* 则从它开始重复左孩子入栈的过程；否则说明此时栈顶为要访问的节点（因为左右孩子都是要么为空要么已访问过了），  
//\* 出栈然后访问即可，接下来再判断栈顶元素的右孩子...直到栈空。

vector<int> postorderTraversal(TreeNode\* root) {

vector<int> nodes;

stack<TreeNode\*> toVisit;

TreeNode\* curNode = root;

TreeNode\* lastNode = NULL;

while (curNode || !toVisit.empty()) {

if (curNode) { //一直压左孩子

toVisit.push(curNode);

curNode = curNode -> left;

}

else {

TreeNode\* topNode = toVisit.top();

if (topNode -> right && lastNode != topNode -> right) //右孩子非空且未被访问过

curNode = topNode -> right;

else {

nodes.push\_back(topNode -> val);

lastNode = topNode;

toVisit.pop();

}

}

}

return nodes;

}

//队列实现： 遍历从根结点开始，首先将根结点入队，然后开始执行循环：结点出队、访问该结点、其左右儿子入队

//层序基本过程：先根结点入队，然后

// 从队列中取出一个元素；

// 访问该元素所指结点；

// 若该元素所指结点的左、右孩子结点非空，则将其左、右孩子的指针顺序入队。

void LevelOrderTraversal(BinTree BT) //层次输出叶子节点可借助此思路

{

queue<BinTree> Q;

BinTree T;

if (!BT)

{

return;

}

Q.push(BT);

while (!Q.empty())

{

T = Q.front();

Q.pop();

printf("%5d", T->Data);

if (T->Left)

{

Q.push(T->Left);

}

if (T->Right)

{

Q.push(T->Right);

}

}

}

//遍历二叉树的应用

//输出二叉树中的叶子节点,前中后序遍历改编都可以

void PreOrderPrintLeaves(BinTree BT)

{

if (BT)

{

if (!BT->Left&&!BT->Right)

{

printf("%5d", BT->Data);

}

PreOrderPrintLeaves(BT->Left);

PreOrderPrintLeaves(BT->Right);

}

}

//二叉树的高度，要知道左右子树高度才行，所以有后序遍历方法

int PostOrderGetHeight(BinTree BT)

{

int LH, RH, MaxH;

if (BT)

{

LH = PostOrderGetHeight(BT->Left);

RH = PostOrderGetHeight(BT->Right);

MaxH = (LH > RH) ? LH + 1 : RH + 1;

return MaxH;

}

else

{

return 0;

}

}

4.平衡树+生成树+Dijkstra(ppt)

## 5.排序算法

[归并，快排，堆排，希尔排序O(N\*log(N))）](http://www.cnblogs.com/ranjiewen/p/5930560.html)

//冒泡排序 O(n^2)

class BubbleSort {

public:

int\* bubbleSort(int\* A, int n) {

// write code here

for (int i = 0; i<n; i++)

{

for (int j = 0; j<n - i - 1; j++)

{

if (A[j]>A[j + 1])

{

int temp = A[j];

A[j] = A[j + 1];

A[j + 1] = temp;

}

}

}

return A;

}

};

//请编写一个选择排序算法 O(n^2)

class SelectionSort {

public:

int\* selectionSort(int\* A, int n) {

// write code here

int k = 0;

for (int i = 0; i < n-1; i++)

{

k = i;

for (int j = i; j < n; j++)

{

if (A[k]>A[j])

{

k = j;

}

}

if (k!=i)

{

int temp = A[i];

A[i] = A[k];

A[k] = temp;

}

}

return A;

}

};

//请编写一个插入算法 O(n^2)

class InsertionSort

{

public:

int\* insertionSort(int\* A, int n)

{

for (int i = 1; i < n; i++)

{

int temp = A[i];

int j = i - 1;

for (; j >= 0;j--) //j前面的已经排好序，从后面往前比较，当没有比当前值大的时候bereak;

{

if (A[j]>temp)

{

A[j + 1] = A[j];

}

else

{

break;

}

}

A[j + 1] = temp;

}

return A;

}

};

//归并排序 O(N\*log(N))

class MergeSort {

public:

int\* mergeSort(int\* A, int n) {

// write code here

mergeSort(A, 0, n - 1);

return A;

}

void mergeSort(int\* A, int beg, int end)

{

if (beg < end)

{

int mid = beg + (end - beg) / 2;

mergeSort(A, beg, mid);

mergeSort(A, mid + 1, end);

merge(A,beg,mid,end);

}

return;

}

void merge(int\* A, int beg\_, int mid\_, int end\_)

{

int \*B = new int[end\_ - beg\_ + 1];

int index1 = beg\_;

int index2 = mid\_ + 1;

int i = 0;

while (index1<=mid\_&&index2<=end\_)

{

if (A[index1]<=A[index2])

{

B[i++] = A[index1++];

}

else

{

B[i++] = A[index2++];

}

}

while (index1 <= mid\_)

{

B[i++] = A[index1++];

}

while (index2<=end\_)

{

B[i++] = A[index2++];

}

//memcpy(A,B,end\_-beg\_+1);

for (int i = 0; i < end\_ - beg\_ + 1;i++)

{

A[beg\_+i] = B[i]; //A[beg\_++] 不能写，改变了输入参数

}

delete[] B;

}

};

//快速排序 O(N\*log(N)) //非递归实现

#include <math.h>

class QuickSort {

public:

int\* quickSort(int\* A, int n) {

// write code here

quickSort(A, 0, n - 1);

return A;

}

void quickSort(int\* A, int low, int high)

{

if (low <= high)

{

int part = partition(A, low, high);

quickSort(A, low, part - 1);

quickSort(A, part + 1, high);

}

return;

}

int partition(int\* A, int low, int high)

{

int privotKey = A[low]; //基准元素

while (low < high)

{ //从表的两端交替地向中间扫描

while (low < high && A[high] >= privotKey)

--high; //从high 所指位置向前搜索，至多到low+1 位置。将比基准元素小的交换到低端

swap(&A[low], &A[high]);

while (low < high && A[low] <= privotKey)

++low;

swap(&A[low], &A[high]);

}

return low;

}

};

class QuickSort2 {

public:

int\* quickSort(int\* A, int n) {

// write code here

quickSort(A, 0, n - 1);

return A;

}

void quickSort(int\* A, int low, int high)

{

if (low <= high)

{

int randn = low + rand() % (high - low + 1); //随机选择关键字的下标

swap(&A[randn], &A[high]); //void swap(int\* A,int index1,int index2) //最好都操作下标

int part = partition(A, low, high);

quickSort(A, low, part - 1);

quickSort(A, part + 1, high);

}

return;

}

int partition(int\* A, int low, int high) //O(N)

{

//int pivot = A[low];//很多随机选择放在这里面，而且是以值的形式确定，而非下标标记为关键字

int pivot = low-1; //关键字的位置

for (int i = low ; i <= high; i++)

{

if (A[i] <= A[high])

{

swap(&A[i], &A[++pivot]); //感觉这样会把A数组前面的值覆盖？-->其实没有交换的效果就是把前面的交换到后面

}

}

return pivot;

}

};

快速排序的非递归实现：主要是将第一层每个子序列的左右边界用栈存储起来，然后在一层取出边界，再做partition,再存储下一层的边界

void quickSort2(double\* a, int left, int right)

{

stack<int> t;

if(left<right)

{

int p = partition(a, left, right);

if (p-1>left)

{

t.push(left);

t.push(p-1);

}

if (p+1<right)

{

t.push(p+1);

t.push(right);

}

while(!t.empty())

{

int r = t.top();

t.pop();

int l = t.top();

t.pop();

p = partition(a, l, r);

if (p-1>l)

{

t.push(l);

t.push(p-1);

}

if (p+1<r)

{

t.push(p+1);

t.push(r);

}

}

}

}

[用非递归、不用栈的方法，实现原位（in-place）的快速排序](http://www.cnblogs.com/fastcam/p/5919431.html)

//推排序 O(N\*log(N))

class HeapSort {

public:

int\* heapSort(int\* A, int n) {

// write code here

buildHeap(A, n); //初始时构建堆

//从最后一个元素开始对序列进行调整

for (int i = n - 1; i >= 0;i--)

{

swap(&A[0], &A[i]);

heapAdjust(A,0,i);

}

return A;

}

void buildHeap(int\* A, int size\_A)

{

for (int i = (size\_A)/ 2-1; i >= 0; i--)

{

heapAdjust(A,i,size\_A);

}

}

void heapAdjust(int\* A, int root, int size\_A) //大顶堆

{

int leftchild = 2 \* root + 1;

if (leftchild<size\_A) //递归形式

{

int rightchild = leftchild + 1;

if (rightchild<size\_A)

{

if (A[leftchild]<A[rightchild])

{

leftchild = rightchild;

}

}

//leftchild为左右子节点中较大的结点

if (A[root]<A[leftchild])

{

int temp = A[root];

A[root] = A[leftchild]; //将较大结点值上移到根节点

A[leftchild] = temp; //完成交换，子节点变为以前的根节点

heapAdjust(A, leftchild, size\_A);

}

}

return;

}

};

class HeapSort2 {

public:

int\* heapSort(int\* A, int n) {

// write code here

buildHeap(A, n); //初始时构建堆

//从最后一个元素开始对序列进行调整

for (int i = n - 1; i >= 0; i--)

{

swap(&A[0], &A[i]);

heapAdjust(A, 0, i);

}

return A;

}

void buildHeap(int\* A, int size\_A)

{

for (int i = (size\_A - 1) / 2; i >= 0; i--)

{

heapAdjust(A, i, size\_A);

}

}

void heapAdjust(int\* A, int root, int size\_A) //调整为大顶堆

{

int temp = A[root];

int leftchild = 2 \* root + 1;

while (leftchild < size\_A) //非递归形式

{

int rightchild = leftchild + 1;

if (rightchild < size\_A)

{

if (A[leftchild] < A[rightchild])

{

leftchild = rightchild;

}

}

//leftchild为左右子节点中较大的结点

if (A[root] < A[leftchild])

{

A[root] = A[leftchild]; //将较大结点值上移到根节点

root = leftchild; //更新新的根节点

leftchild = 2 \* root + 1;

}

else //当前结点大于左右子节点则不需要调整

{

break;

}

A[root] = temp; //完成交换，子节点变为以前的根节点

}

return;

}

};

//希尔排序 O(N\*log(N)) ---不稳定

class ShellSort {

public:

int\* shellSort(int\* A, int n) {

// write code here

int dk = n / 2;

while (dk>=1)

{

shellSort2(A,n,dk);

dk /= 2;

}

return A;

}

void shellSort(int\* A, int n, int dk)

{

for (int i = dk; i < n;i++)

{

int index = i; //当前访问的位置

while (index>=dk)

{

if (A[index-dk]>A[index])

{

swap(&A[index-dk],&A[index]); //交换不算最优，找到插入位置才交换

index -= dk;

}

else

{

break;

}

}

}

}

void shellSort2(int\* A,int n,int dk)

{

for (int i = dk; i < n;i++)

{

if (A[i]<A[i-dk]) //找到插入位置

{

int x = A[i];//复制哨兵

A[i] = A[i - dk];

int j = i - dk; //从该位置向前查找

while (x<A[j]&&j>=0) //防止j越界

{

A[j] = A[j - dk];

j -= dk; //向前移动

}

A[j + dk] = x;// 插入到正确位置

}

}

}

};

## 6. 高级算法：dp,bfs,dfs,backtracking

(1)全排列：Given a collection of numbers, return all possible permutations.

For example,  
[1,2,3] have the following permutations:  
[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

class Solution\_46 {

public:

void help(int i,vector<int> &nums,vector<vector<int>> &vecs)

{

if (i==nums.size())

{

vecs.push\_back(nums);

return;

}

else

{

for (int j = i; j < nums.size();j++)

{

swap(nums[i],nums[j]);

help(i + 1, nums,vecs);

swap(nums[i],nums[j]);

}

}

return;

}

vector<vector<int>> permute(vector<int>& nums) {

vector<vector<int>> vecs;

if (nums.size()==0)

{

return vecs;

}

help(0, nums,vecs);

return vecs;

}

};

(2) 组合数Combination

// 对于求C(n, m)，从第一个字符开始扫描，每个字符有两种情况，要么被选中，要么不被选中，如果被选中，递归求解C(n-1, m-1)。

// 如果未被选中，递归求解C(n-1, m)。不管哪种方式，n的值都会减少，递归的终止条件n=0或m=0。

// 数组的全组合数

void combination(vector<int> src, int i, int m, vector<int> &res, vector<vector<int>> &vecs)

{

if (i >= src.size() && m != 0)

{

return;

}

if (m == 0) //递归终止条件

{

copy(res.begin(), res.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

return;

}

//选择该元素

res.push\_back(src[i]);

combination(src, i + 1, m - 1, res, vecs);

res.pop\_back();

//不选该元素

combination(src, i + 1, m, res, vecs);

return;

}

int main()

{

int m, n;

cin >> m >> n; //C(n,m)

vector<int> input;

for (int i = 0; i < n; i++)

{

input.push\_back(i + 1);

}

vector<vector<int>> vecs;

vector<int> vec;

//combination(input,0, m, vec, vecs); ////C(n,m)

for (int i = 1; i <= n; i++) //C(n, 1), C(n, 2),...C(n, n)的总和

{

combination(input, 0, i, vec, vecs); ////C(n,m)

}

return 0;

}

（3）[Combination Sum 组合之和](http://www.cnblogs.com/grandyang/p/4419259.html): For example, given candidate set [2, 3, 6, 7] and target 7,A solution set is:[[7],[2, 2, 3]]

class Solution\_39 {

public:

vector<vector<int> > combinationSum(vector<int> &candidates, int target) {

vector<vector<int> > res;

vector<int> out;

sort(candidates.begin(), candidates.end());

combinationSumDFS(candidates, target, 0, out, res);

return res;

}

void combinationSumDFS(vector<int> &candidates, int target, int start, vector<int> &out, vector<vector<int> > &res) {

if (target < 0) return;

else if (target == 0) res.push\_back(out);

else {

for (int i = start; i < candidates.size(); ++i) {

out.push\_back(candidates[i]);

combinationSumDFS(candidates, target - candidates[i], i, out, res);

out.pop\_back();

}

}

}

};

## 7.二分查找：

//由于数组有序，所以使用二分查找方法定位k的第一次出现位置和最后一次出现位置

class Solution\_37{

public:

// 方法一：借助stl库函数

int GetNumberOfK(vector<int> data, int k)

{

return count(data.begin(), data.end(), k);

}

// 方法二：O(n)遍历

// 方法三：二分查找

int GetNumberOfK\_2(vector<int> data, int k) {

int lower = getLower(data, k);

int upper = getUpper(data, k);

return upper - lower + 1;

}

// 获取k第一次出现的下标

int getLower(vector<int> data, int k){

int start = 0, end = data.size() - 1;

int mid = (start + end) / 2;

while (start <= end){

if (data[mid] < k){

start = mid + 1;

}

else{ // data[mid] >= k

end = mid - 1;

}

mid = (start + end) / 2;

}

return start;

}

// 获取k最后一次出现的下标

int getUpper(vector<int> data, int k){

int start = 0, end = data.size() - 1;

int mid = (start + end) / 2;

while (start <= end){

if (data[mid] <= k){ // data[mid] == k

start = mid + 1;

}

else{

end = mid - 1;

}

mid = (start + end) / 2;

}

return end;

}

int getFirstK(int\* data, int k, int start, int end){

while (start <= end){

int mid = start + ((end - start) >> 1);

if (data[mid] == k){

if ((mid > 0 && data[mid - 1] != k) || mid == 0)

return mid;

else

end = mid - 1;

}

else if (data[mid] > k)

end = mid - 1;

else

start = mid + 1;

}

return -1;

}

int getLastK(int\* data, int length, int k, int start, int end){

while (start <= end){

int mid = start + ((end - start) >> 1);

if (data[mid] == k){

if ((mid < length - 1 && data[mid + 1] != k) || mid == length - 1)

return mid;

else

start = mid + 1;

}

else if (data[mid] < k)

start = mid + 1;

else

end = mid - 1;

}

return -1;

}

//标准的二分查找

int binary\_search(int\*data, int n, int target)

{

int low = 0, high = n - 1;

while (low<=high) //必须有等号，避免边界查找不到

{

int mid = low + (high - low) / 2;

if (data[mid]==target)

{

return mid;

}

else if (data[mid]>target)

{

high = mid - 1;

}

else

{

low = mid + 1;

}

}

return -1;

}

int binary\_search\_test(int\*data, int n, int target)

{

int low = 0, high = n - 1;

int mid = low + (high - low) / 2;

while (low <= high) //必须有等号，避免边界查找不到

{

if (data[mid] == target)

{

return mid;

}

else if (data[mid] > target)

{

high = mid - 1;

}

else

{

low = mid + 1;

}

mid = low + (high - low) / 2;

}

return -1;

}

// 用二分法寻找上界

int binary\_search\_upperbound(int array[], int low, int high, int target)

{

//Array is empty or target is larger than any every element in array

if (low > high || target >= array[high]) //控制找不到的情况

return -1;

int mid = (low + high) / 2;

while (high > low)

{

if (array[mid] > target)

high = mid;

else

low = mid + 1;

mid = (low + high) / 2;

}

return mid;

}

int binary\_search\_lowerbound(int array[], int low, int high, int target)

{

//Array is empty or target is less than any every element in array

if (high < low || target <= array[low])

return -1;

int mid = (low + high + 1) / 2; //make mid lean to large side

while (low < high)

{

if (array[mid] < target)

low = mid;

else

high = mid - 1;

mid = (low + high + 1) / 2;

}

return mid;

}

int search\_rotation(vector<int>& nums, int target) { //旋转数组查找

int l = 0, r = nums.size() - 1;

while (l <= r) {

int mid = (l + r) / 2;

if (target == nums[mid])

return mid;

// there exists rotation; the middle element is in the left part of the array

if (nums[mid] > nums[r]) {

if (target < nums[mid] && target >= nums[l])

r = mid - 1;

else

l = mid + 1;

}

// there exists rotation; the middle element is in the right part of the array

else if (nums[mid] < nums[l]) {

if (target > nums[mid] && target <= nums[r])

l = mid + 1;

else

r = mid - 1;

}

// there is no rotation; just like normal binary search

else {

if (target < nums[mid])

r = mid - 1;

else

l = mid + 1;

}

}

return -1;

}

};

int main()

{

int array[] = { 2, 3, 7, 7, 7, 13, 17 };

int target = 7;

Solution\_37 su\_37;

int ret1=su\_37.GetNumberOfK\_2(vector<int>(array,array+7), target); //3

int ret2 = su\_37.binary\_search(array,7, target); //3

int ret3 = su\_37.binary\_search\_test(array, 7, target); //3

int ret4 = su\_37.getUpper(vector<int>(array, array + 7), target); //4

int ret5 = su\_37.binary\_search\_upperbound(array, 0, 6, target);//5

int ret6 = su\_37.getLower(vector<int>(array, array + 7), target);//2

int ret7 = su\_37.binary\_search\_lowerbound(array, 0, 6, target);//1

vector<int> res(array, array + 7);

int ret8=lower\_bound(res.begin(), res.end(), target)-res.begin(); //2

int ret9 = upper\_bound(array, array + 7, target)-array; //5

}

## 8.最大子序列

int maxSubSum(const vector<int> & arr,int &begin,int &end){

int maxSum=0;

int currSum=0;

int newbegin=0;

for(int i=0;i<arr.size();++i){

currSum+=arr[i];

if(currSum>maxSum){

maxSum=currSum;

begin=newbegin;

end=i;

}

if(currSum<0){

currSum=0;

newbegin=i+1;

}

}

return maxSum;

}

**最长递增子序列**

//动态规划法求最长递增子序列 LIS

int dp[101]; /\* 设数组长度不超过100，dp[i]记录到[0,i]数组的LIS \*/

int lis; /\* LIS 长度 \*/

int LIS(int \* arr, int size)

{

for (int i = 0; i < size; ++i)

{

dp[i] = 1;

for (int j = 0; j < i; ++j)

{

if (arr[i] > arr[j] && dp[i] < dp[j] + 1)

{

dp[i] = dp[j] + 1;

if (dp[i] > lis)

{

lis = dp[i];

}

}

}

}

return lis;

}



[**两个字符串的最长连续公共子串**](http://www.cnblogs.com/ranjiewen/p/5854085.html)**：**

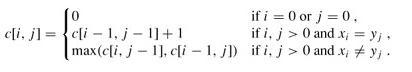
1.解法就是用一个矩阵来记录两个字符串中所有位置的两个字符之间的匹配情况，若是匹配则为1，否则为0。然后求出对角线最长的1序列，其对应的位置就是最长匹配子串的位置.

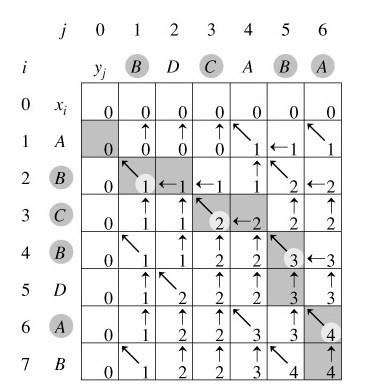
2.但是在0和1的矩阵中找最长的1对角线序列又要花去一定的时间。通过改进矩阵的生成方式和设置标记变量，可以省去这部分时间。当字符匹配的时候，我们并不是简单的给相应元素赋上1，而是赋上其左上角元素的值加一。我们用两个标记变量来标记矩阵中值最大的元素的位置，在矩阵生成的过程中来判断当前生成的元素的值是不是最大的，据此来改变标记变量的值，那么到矩阵完成的时候，最长匹配子串的位置和长度就已经出来了。

3.这样做速度比较快，但是花的空间太多。我们注意到在改进的矩阵生成方式当中，每生成一行，前面的那一行就已经没有用了。因此我们只需使用一维数组即可。

**最长公共子序列：**

最长公共子序列是一个十分实用的问题，它可以描述两段文字之间的"相似度"。设有二维[数组](http://baike.so.com/doc/5545345-5760453.html)f[i,j] 表示 X 的 i 位和 Y 的 j 位之前的最长公共子序列的长度。



记录路径的方法：递归的调用：

if(c[i][j]==(c[i-1][j-1]+1)){ //这样判断不需要额外空间记录

lcs(c,x,i-1,j-1);

//注意c的长度要比x大1

Cout<<x[i-1];

}else if(c[i][j]==c[i-1][j]){

lcs(c,x,i-1,j);

}else{

lcs(c,x,i,j-1);

}

int getMaxLengthSub(string s1, string s2)

{

int ret = 0;

vector<vector<int> > vecs(s1.size() + 1, vector<int>(s2.size() + 1, 0));

for (int i = 1; i <= s1.size(); i++)

{

for (int j = 1; j <= s2.size(); j++)

{

if (s1[i - 1] == s2[j - 1]) //下标

{

vecs[i][j] = vecs[i - 1][j - 1] + 1;

}

else

{

vecs[i][j] = max(vecs[i - 1][j], vecs[i][j - 1]);

}

}

}

ret = vecs[s1.size()][s2.size()];

return ret;

}

**最长不重复子串**

如果它与前面的最长不重复子串中的字符没有重复，那么就可以以它为结尾构成新的最长子串；如果有重复，那么就与某个稍短的子串构成新的子串或者单独成一个新子串。

对于每个当前的元素，我们“回头”去查询是否有与之重复的，如没有，则最长不重复子串长度+1，如有，则是与第一个重复的字符之后的串构成新的最长不重复子串，新串的长度便是当前元素下标与重复元素下标之差。

使用Hash： 要求子串中的字符不能重复，判重问题首先想到的就是hash，寻找满足要求的子串，最直接的方法就是遍历每个字符起始的子串。

/\* LNRS 动态规划+hash，时间复杂度O(n) 空间复杂度O(1)算法\*/

void LNRS\_dp\_hash\_ultimate(char \* arr, int size)

{

memset(visit, -1, sizeof visit);

int maxlen = maxindex = 0;

visit[arr[0]] = 0;

int curlen = 1;

for (int i = 1; i < size; ++i)

{

if (visit[arr[i]] == -1)

{

++curlen;

visit[arr[i]] = i; /\* 记录字符下标 \*/

}

else

{

curlen = i - visit[arr[i]];

visit[arr[i]] = i; /\* 更新字符下标 \*/

}

if (curlen > maxlen)

{

maxlen = curlen;

maxindex = i + 1 - maxlen;

}

}

output(arr);

}