**随机数列：**

给定一个正整数n，需要输出一个长度为n的数组，数组元素是随机数，范围为0 – n-1，且元素不能重复。

思路1：建一个 hashtable ，然后循环获取随机数，再到 hashtable 中找，如果hashtable 中没有这个数，则输出。

vector<int> GetRandomSequence2(int n)

{

vector<int> hash(n, 0);

vector<int> re(n);

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

{

int num = rand() % n;

while (hash[num])

num = rand() % (n + 1);

hash[num] = 1;

re[i] = num + 1;

}

return re;

}

思路2：

使用一个for循环遍历这个数组，每次循环都生成一个[0，52)之间的随机数RandNum，以RandNum为数组下标，把当前下标对应的值和RandNum对应位置的值交换，循环结束，每个牌都与某个位置交换了一次，这样一副牌就被打乱了。

void GetRandomSequence2(int n)

{

vector<int> poker(n);

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

poker[i] = i+1;

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

{

int RandNum = rand()%(i+1);

swap(poker[i], poker[RandNum]);

}

}

void shuffleArray(vector<int> num)

{

vector<int> res;

while(num.size()) {

int d = rand()%num.size();

res.push\_back(num[d]);

num.erase(num.begin()+d);

}

}

**大数相乘**

void reverseString(string &s)

{

if(s.size() < 2)

return;

int lo = 0;

int hi = s.size()-1;

while(lo < hi)

swap(s[lo++], s[hi--]);

}

void multiplyLargeNum(string s1, string s2)

{

int len1 = s1.size();

int len2 = s2.size();

reverseString(s1);

reverseString(s2);

int \*re = new int[len1+len2];

memset(re, 0, sizeof(int)\*(len1+len2));

for(int i = 0; i < len1; ++i)

for(int j = 0; j < len2; ++j)

re[i+j] += (s1[i]-'0')\*(s2[j]-'0');

for(int i = 0; i < len1+len2-1; ++i)

{

re[i+1] += re[i]/10;

re[i] %= 10;

}

for(int i = len1+len2-1; i >= 0; --i)

printf("%d", re[i]);

}

**排序**

//选堆快 希不稳：选择排序、堆排序、快排、希尔排序是不稳定的

//选堆归 基不变：选择排序、堆排序、归并排序、基数排序性能不变

//冒泡排序

void Bubble\_sort(vector<int> &A)

{

int len = A.size();

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

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

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

{

swap(A[j], A[j + 1]);

}

}

//选择排序

void Select\_sort(vector<int> &A)

{

int len = A.size();

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

{

int min = i;

for (int j = i + 1; j < len; ++j)

if (A[j] < A[min])

min = j;

swap(A[min], A[i]);

}

}

//插入排序

void Insert\_sort(vector<int> &A)

{

int len = A.size();

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

{

int key = A[i];

int j = i - 1;

while (j >= 0 && A[j] > key)

{

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

--j;

}

A[j + 1] = key;

}

}

//希尔排序

//希尔排序与插入排序很相似，插入排序相当于每次 d=1

void Shell\_sort(vector<int> &A)

{

int len = A.size();

int d = len / 2;

while (d > 0)

{

for (int i = d; i < len; ++i)

{

int key = A[i];

int j = i - d;

while (j >= 0 && A[j] > key)

{

A[j + d] = A[j];

j -= d;

}

A[j + d] = key;

}

d /= 2;

}

}

//归并排序

//合并两个有序数组A[left..mid]和A[mid+1..right],使A有序

void merge(vector<int> &A, int left, int mid, int right)

{

int index = 0;

int i = left; //第一个数组起点

int j = mid+1; //第二个数则起点

vector<int> temp(right - left + 1);

while (i <= mid && j <= right)

{

if (A[i] < A[j])

temp[index++] = A[i++];

else

temp[index++] = A[j++];

}

while (i <= mid)

temp[index++] = A[i++];

while (j <= right)

temp[index++] = A[j++];

for (i = left,j=0; i <= right; i++,j++)

A[i] = temp[j];

}

void mergesort(vector<int> &A, int left, int right)

{

if (left < right)

{

int mid = (left + right) / 2;

mergesort(A, left, mid);

mergesort(A, mid + 1, right);

merge(A, left, mid, right);

}

}

void Merge\_sort(vector<int> &A)

{

int len = A.size();

mergesort(A, 0, len-1);

}

//Lomuto

//int partition(vector<int> &A, int left, int right)

//{

// int key = A[right]; //选最后一个元素作为基准

// int pre = left - 1;

// for (int i = left; i < right; ++i)

// {

// if (A[i] <= key)

// {

// ++pre;

// swap(A[pre], A[i]);

// }

// }

// swap(A[pre + 1], A[right]);

// return pre+1;

//}

//Hoare

int partition(vector<int> &A, int left, int right)

{

int key = A[left]; //选第一个元素作为基准

while (left < right)

{

while (left < right && A[right] >= key)

--right;

swap(A[left], A[right]);

while (left < right && A[left] <= key)

++left;

swap(A[left], A[right]);

}

return left;

}

void Quick\_sort(vector<int> &A,int left, int right)

{

if (left < right)

{

int index = partition(A, left, right);

Quick\_sort(A, left, index-1);

Quick\_sort(A, index + 1, right);

}

}

//堆排序，数组0处存放根节点

void HeapAdjustDown(vector<int> &A, int start, int end) //start 向下调整，直到找到合适的位置

{

int cur = A[start]; //当前节点

int child = 2 \* start + 1; //当前节点的左孩子

while (child < end)

{

//找出左右孩子中最大的, child+1为右孩子

if (child + 1 < end && A[child + 1] > A[child])

child++;

//如果当前节点大于左右孩子节点，则不用调整

if (cur >= A[child])

break;

A[start] = A[child]; //把最大孩子节点值赋给当前节点

start = child; //继续调整子节点

child = 2 \* child + 1;

}

A[start] = cur;

}

void BuildHeap(vector<int> &A)

{

int len = A.size();

//从第一个非叶子节点len/2-1 开始调整堆

for (int i = len / 2 - 1; i >= 0; i--)

HeapAdjustDown(A, i, len);

}

void Heap\_sort(vector<int> &A)

{

int len = A.size();

BuildHeap(A);

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

{

//堆顶元素（最大值）和最后一个元素交换，然后调整剩下的元素（除最后一个元素外）

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

HeapAdjustDown(A, 0, i);

}

}

//基数排序

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*函数名称：GetNumInPos

\*参数说明：num 一个整形数据

\* pos 表示要获得的整形的第pos位数据

\*说明： 找到num的从低到高的第pos位的数据

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int GetNumInPos(int num, int pos)

{

int temp = 1;

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

temp \*= 10;

return (num / temp) % 10;

}

#define MAXPOS 10 //最高位的位数

void RadixSort(vector<int> &A)

{

int len = A.size();

vector<vector<int>> radixArray(10); //分为0~9的序列空间

for (int pos = 1; pos <= MAXPOS; pos++) //从个位开始到最高位数

{

for (int i = 0; i < len; i++) //分配过程

{

int num = GetNumInPos(A[i], pos);

radixArray[num].push\_back(A[i]);

}

for (int i = 0, j = 0; i < 10; i++) //收集

{

while (!radixArray[i].empty())

{

A[j++] = radixArray[i].front(); //取首部数据依次插入原数组

radixArray[i].erase(radixArray[i].begin()); //移除首部元素

}

}

}

}

//MSD

void RadixSort(vector<int> &A, int d)

{

int len = A.size();

vector<vector<int>> radixArray(10); //分为0~9的序列空间，用队列保存每个桶分配的数据

//位数大于0，且数组长度大于1

if (d >= 1 && len > 1)

{

for (int i = 0; i < len; i++) //分配过程

{

int num = GetNumInPos(A[i], d);

radixArray[num].push\_back(A[i]);

}

for (int i = 0, j = 0; i < 10; i++) //收集

{

RadixSort(radixArray[i], d-1); //递归

while (!radixArray[i].empty())

{

A[j++] = radixArray[i].front(); //取队列首部数据依次插入原数组

radixArray[i].erase(radixArray[i].begin());

}

}

}

}

**二分查找**

**二叉树遍历**

**递归**

前序

void preorder(TreeNode \*root){

    if(root){

        cout << root->val;

        preorder(root->left);

        preorder(root->right);

    }

} 

中序

void preorder(TreeNode \*root){

    if(root){

        preorder(root->left);

cout << root->val;

        preorder(root->right);

    }

} 

后序

void preorder(TreeNode \*root){

    if(root){

        preorder(root->left);

        preorder(root->right);

cout << root->val;

    }

} 

非递归

前序

void preorder\_dev(TreeNode \*root) {

if(!root) return;

stack<TreeNode \*> st;

while(root || !st.empty()) {

while(root) {

cout << root->val;

st.push(root);

root=root->left;

}

root = st.top();

st.pop();

root=root->right;

}

}

中序

void inorder\_dev(TreeNode \*root) {

if(!root) return;

stack<TreeNode \*> st;

while(root || !st.empty()) {

while(root) {

st.push(root);

root=root->left;

}

root = st.top();

st.pop();

cout << root->val;

root=root->right;

}

}

二叉树层次遍历

class Solution {

public:

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

vector<int> res;

if (root == NULL)

return res;

queue<TreeNode \*> q;

q.push(root);

while (q.size()) {

TreeNode \*t = q.front();

q.pop();

res.push\_back(t->val);

if (t->left)

q.push(t->left);

if (t->right)

q.push(t->right);

}

return res;

}

};

按行遍历二叉树

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

if (!root)

return{};

deque<TreeNode \*> q;

vector<vector<int>> res;

q.push\_back(root);

while (q.size())

{

vector<int> tmp;

int n = q.size();

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

{

TreeNode \*t = q.front();

q.pop\_front();

tmp.push\_back(t->val);

if (t->left)

q.push\_back(t->left);

if (t->right)

q.push\_back(t->right);

}

res.push\_back(tmp);

}

return res;

}

二叉树重建

前序和中序：

TreeNode\* buildTree(vector<int> &pre, vector<int> &in) {

if (pre.empty() || in.empty())

return NULL;

return buildTree(pre, in, 0, pre.size(), 0, in.size());

}

TreeNode\* buildTree(vector<int> &pre, vector<int> &in, int preS, int preE, int inS, int inE)

{

if (preS >= preE || inS >= inE)

return NULL;

TreeNode \*root = new TreeNode(pre[preS]);

int i = 0;

while (i < inE && in[inS + i] != pre[preS])

i++;

root->left = buildTree(pre, in, preS + 1, preS + 1 + i, inS, inS + i);

root->right = buildTree(pre, in, preS + 1 + i, preE, inS + i + 1, inE);

return root;

}

中序和后序

TreeNode\* buildTree(vector<int>& in, vector<int>& pre) {

if (in.empty() || pre.empty())

return NULL;

return buildTree(in, pre, 0, in.size(), 0, pre.size());

}

TreeNode\* buildTree(vector<int> &in, vector<int> &post, int inS, int inE, int postS, int postE)

{

if (inS >= inE || postS >= postE)

return NULL;

TreeNode \*root = new TreeNode(post[postE - 1]);

int i = 0;

while (i < inE && in[inS + i] != post[postE - 1])

i++;

root->left = buildTree(in, post, inS, inS + i, postS, postS + i);

root->right = buildTree(in, post, inS + 1 + i, inE, postS + i, postE - 1);

return root;

}

平衡二叉树的判断

bool IsBalanced\_Solution(TreeNode\* pRoot) {

int depth = 0;

return helper(pRoot, depth);

}

bool helper(TreeNode \*root, int &depth) {

if (root == NULL) {

depth = 0;

return true;

}

int left, right;

if (helper(root->left, left) && helper(root->right, right)) {

int dif = left - right;

if (dif <= 1 && dif >= -1) {

depth = 1 + (left>right ? left : right);

return true;

}

}

return false;

}

二叉搜索树的第K个节点（中序）

class Solution2 {

int count = 0;

public:

TreeNode\* KthNode(TreeNode\* pRoot, int k)

{

if (pRoot){

//中序遍历

TreeNode \*ret = KthNode(pRoot->left, k);

if (ret) return ret;

if (++count == k) return pRoot;

ret = KthNode(pRoot->right, k);

if (ret) return ret;

}

return NULL;

}

};

int kthSmallest(TreeNode\* root, int k) {

int count = countNodes(root->left);

if (k <= count)

return kthSmallest(root->left, k);

else if (k > count + 1)

return kthSmallest(root->right, k - 1 - count); // 1 is counted as current node

return root->val;

}

int countNodes(TreeNode \*root) {

if (root == NULL) return 0;

return 1 + countNodes(root->left) + countNodes(root->right);

}

**二叉树对称**

bool isSymmetric(TreeNode\* root) {

if (!root)

return true;

return isSymmetric(root->left, root->right);

}

bool isSymmetric(TreeNode\* root1, TreeNode \*root2)

{

if (!root1 || !root2)

return root1 == root2;

if (root1->val == root2->val)

return isSymmetric(root1->left, root2->right) && isSymmetric(root1->right, root2->left);

return false;

}

**二叉树中路径和最大**

任意一条路径，不必经过根节点

int maxPathSum(TreeNode\* root) {

int maxValue = INT\_MIN;

maxPathDown(root, maxValue);

return maxValue;

}

int maxPathDown(TreeNode \*node, int &maxValue)

{

if (node == NULL)

return 0;

int left = max(0, maxPathDown(node->left, maxValue));

int right = max(0, maxPathDown(node->right, maxValue));

maxValue = max(maxValue, left + right + node->val);

return max(left, right) + node->val;

}

**栈、队列的实现**

**两个栈实现队列**

class Solution

{

public:

void push(int node) {

stack1.push(node);

}

int pop() {

if (!stack2.empty())

{

int ret = stack2.top();

stack2.pop();

return ret;

}

else

{

while (!stack1.empty())

{

stack2.push(stack1.top());

stack1.pop();

}

return pop();

}

}

private:

stack<int> stack1;

stack<int> stack2;

};

**两个队列实现栈**

class Mystack{

public:

Mystack() :insert(0){}

void push(int x)

{

qus[insert].push(x);

}

void pop(){

while (qus[insert].size()>1)

{

qus[1 - insert].push(qus[insert].front());

qus[insert].pop();

}

if (qus[insert].size() > 0)

qus[insert].pop();

insert = 1 - insert;

}

private:

queue<int> qus[2];

int insert;

};

**是否是出栈顺序**

bool IsPopOrder(vector<int> pushV, vector<int> popV) {

if (pushV.size() != popV.size())

return false;

int n = pushV.size();

stack<int> st;

int i = 0, j = 0;

while (i < n || j < n) {

if (st.empty() || st.top() != popV[j])

st.push(pushV[i++]);

if (st.top() == popV[j]) {

st.pop();

j++;

}

}

return i == j ? true : false;

}

**实现一个栈，最大（小）操作时间复杂度O（1）**

class Solution {

public:

void push(int value) {

st.push(value);

if (minst.empty() || minst.top()>value)

minst.push(value);

else

minst.push(minst.top());

}

void pop() {

if (!st.empty())

{

st.pop();

minst.pop();

}

}

int top() {

if (!st.empty())

return st.top();

return -1;

}

int min() {

if (!st.empty())

return minst.top();

return -1;

}

private:

stack<int> st;

stack<int> minst;

};

**后缀表达式求值**

int evalRPN(vector<string>& tokens) {

int i = 0;

stack<int> st;

while (i < tokens.size())

{

string s = tokens[i];

if (s == "+" || s == "-" || s == "\*" || s == "/")

{

if (st.size() < 2)

return 0;

int num1 = st.top();

st.pop();

int num2 = st.top();

st.pop();

st.push(calculate(num1, num2, s));

}

else

st.push(atoi(s.c\_str()));

i++;

}

return st.top();

}

int calculate(int n1, int n2, string s)

{

if (s == "+")

return n2 + n1;

if (s == "-")

return n2 - n1;

if (s == "\*")

return n2 \* n1;

if (s == "/")

return n2 / n1;

}

**top K 问题**

1. **先建立大小为K的最小堆，然后比较堆顶元素和剩余元素。**

vector<int> findKLargest(vector<int>& nums, int k) {

if (k<1 || k>nums.size())

return 0;

//建立最小堆

for (int i = k / 2 - 1; i >= 0; i--)

HeapAdjustDown(nums, i, k);

//从k+1个元素开始于堆顶元素比较

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

{

//如果大于堆顶元素，交换，调整堆

if (nums[i]>nums[0])

{

swap(nums[i], nums[0]);

HeapAdjustDown(nums, 0, k);

}

}

return vector<int>(nums.begin(),nums.begin()+k);

}

void HeapAdjustDown(vector<int> &A, int start, int end) //start 向下调整，直到找到合适的位置

{

int cur = A[start]; //当前节点

int child = 2 \* start + 1; //当前节点的左孩子

while (child < end)

{

//找出左右孩子中最小的, child+1为右孩子

if (child + 1 < end && A[child + 1] < A[child])

child++;

//如果当前节点小于左右孩子节点，则不用调整

if (cur <= A[child])

break;

A[start] = A[child]; //把最小孩子节点值赋给当前节点

start = child; //继续调整子节点

child = 2 \* child + 1;

}

A[start] = cur;

}

1. **红黑数**

vector<int> findKthLargest(vector<int>& nums, int k) {

multiset<int> se;

for (auto it : nums)

{

if (se.size() < k)

se.insert(it);

else if (\*se.begin() < it) {

se.erase(se.begin());

se.insert(it);

}

}

return vector<int>(se.begin(),se.end());

}

//set默认升序，如果需要降序，自定义比较函数

//set<int,comp> se;

struct comp{

bool operator()(int v1, int v2){ return v1>v2; }

};

1. **partition 思想**

vector<int> findKthLargest(vector<int>& nums, int k) {

if(k<1 || k > nums.size()) return {};

int lo = 0, hi = nums.size() - 1;

int mid = partation(nums, lo, hi);

while (mid != k - 1)

{

if (mid > k - 1) hi = mid - 1;

else lo = mid + 1;

mid = partation(nums, lo, hi);

}

return vector<int>(nums.begin(),nums.begin()+k);

}

int partation(vector<int> &nums, int start, int end)

{

int key = nums[start];

while (start < end)

{

while (start < end && nums[end] <= key) end--;

nums[start] = nums[end];

while (start < end && nums[start] >= key) start++;

nums[end] = nums[start];

}

nums[start] = key;

return start;

}

**链表**

**链表倒数第K个节点**

ListNode\* FindKthToTail(ListNode\* pListHead, unsigned int k) {

if (pListHead == NULL || k < 1)

return NULL;

ListNode\* pre = pListHead, \*behind = pListHead;

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

pre = pre->next;

if (pre == NULL)

return NULL;

}

while (pre->next != NULL) {

pre = pre->next;

behind = behind->next;

}

return behind;

}

**删除链表倒数第K个节点**

ListNode\* removeNthFromEnd(ListNode\* head, int k) {

ListNode \*h1 = head, \*h2 = head;

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

h2 = h2->next;

if (h2 == NULL)

return head->next; // The head need to be removed, do it.

while (h2->next != NULL){

h1 = h1->next;

h2 = h2->next;

}

h1->next = h1->next->next; // the one after the h1 need to be removed

return head;

}

**交换单链表中相邻节点**

ListNode\* swapPairs(ListNode\* head) {

ListNode \*\*pp = &head, \*a, \*b;

while ((a = \*pp) && (b = a->next))

{

a->next = b->next;

b->next = a;

\*pp = b;

pp = &(a->next);

}

return head;

}

**单链表逆序**

class Solution {

public:

ListNode\* ReverseList(ListNode\* pHead) {

return ReverseList(pHead, NULL);

}

ListNode \*ReverseList(ListNode \*pHead, ListNode \*newHead) {

if (pHead == NULL)

return newHead;

ListNode \*p = pHead->next;

pHead->next = newHead;

return ReverseList(p, pHead);

}

};

**合并两个有序链表为有序**

ListNode\* Merge(ListNode\* pHead1, ListNode\* pHead2)

{

if (pHead1 == NULL)

return pHead2;

if (pHead2 == NULL)

return pHead1;

ListNode \*res;

if (pHead1->val<pHead2->val)

{

res = pHead1;

res->next = Merge(pHead1->next, pHead2);

}

else

{

res = pHead2;

res->next = Merge(pHead1, pHead2->next);

}

return res;

}

**有序链表转为平衡二叉搜索树**

TreeNode\* sortedListToBST(ListNode\* head) {

return sortedListToBST(head, NULL);

}

TreeNode\* sortedListToBST(ListNode\* start, ListNode \*end)

{

if (start == end)

return NULL;

ListNode \*fast = start;

ListNode \*slow = start;

while (fast != end && fast->next != end)

{

slow = slow->next;

fast = fast->next->next;

}

TreeNode \*root = new TreeNode(slow->val);

root->left = sortedListToBST(start, slow);

root->right = sortedListToBST(slow->next, end);

return root;

}

**两个链表的交点**

ListNode\* FindFirstCommonNode(ListNode\* pHead1, ListNode\* pHead2) {

ListNode \*p1 = pHead1, \*p2 = pHead2;

while (p1 != p2) {

p1 = p1 ? p1->next : pHead2;

p2 = p2 ? p2->next : pHead1;

}

return p1;

}

**链表环的入口**

ListNode\* EntryNodeOfLoop(ListNode\* pHead)

{

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

return NULL;

ListNode \*fast = pHead, \*slow = pHead;

while (fast && fast->next) {

fast = fast->next->next;

slow = slow->next;

if (fast == slow) //有环

break;

}

fast = pHead;

while (fast != slow){

fast = fast->next;

slow = slow->next;

}

return fast;

}

**删除有序链表中重复节点**

**1.保留一个重复节点**

ListNode\* deleteDuplicates(ListNode\* head) {

if (!head || !head->next)

return head;

ListNode \*p = head->next;

if (p->val != head->val)

{

head->next = deleteDuplicates(p);

return head;

}

else

return deleteDuplicates(p);

}

**2.不保留重复节点**

ListNode\* deleteDuplicates(ListNode\* head) {

if (!head || !head->next)

return head;

int val = head->val;

ListNode \*p = head->next;

if (p->val != val)

{

head->next = deleteDuplicates(p);

return head;

}

else

{

while(p && p->val == val)

p = p->next;

return deleteDuplicates(p);

}

}

**删除链表中指定值节点**

ListNode\* removeElements(ListNode\* head, int val) {

if (head == NULL)

return NULL;

if (head->val == val)

return removeElements(head->next, val);

else

{

head->next = removeElements(head->next, val);

return head;

}

}

**单链表排序**

1. **归并排序**

ListNode\* sortList(ListNode\* head) {

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

return head;

ListNode \*slow = head;

ListNode \*fast = head->next;

while (fast != NULL && fast->next != NULL)

{

slow = slow->next;

fast = fast->next->next;

}

fast = slow->next;

slow->next = NULL;

//合并两个有序链表

return merge(sortList(head), sortList(fast));

}

1. **快速排序**

ListNode\* sortList(ListNode\* head) {

quicksort(head, NULL);

return head;

}

void quicksort(ListNode\* head, ListNode \*end)

{

if (head == end || head == NULL)

return;

ListNode \*mid = partation(head, end);

quicksort(head, mid);

quicksort(mid->next, end);

}

ListNode \*partation(ListNode\* head, ListNode \*end)

{

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

return head;

ListNode \*pre = head;

ListNode \*p = head->next;

int key = head->val;

while (p != end)

{

if (p->val < key)

{

pre = pre->next;

swap(pre->val, p->val);

}

p = p->next;

}

swap(head->val, pre->val);

return pre;

}

**重置链表顺序**

**1-2-3-4 变成 1-4-2-3**

void reorderList(ListNode\* head) {

if (!head || !head->next) return;

// find the middle node: O(n)

ListNode \*p1 = head, \*p2 = head;

while (p2 && p2->next) {

p1 = p1->next;

p2 = p2->next->next;

}

p2 = reverseList(p1->next, NULL); //反转后半段链表

p1->next = NULL; //前半段链表尾部置为 NULL

p1 = head;

while (p1 && p2) {

auto t = p2->next;

p2->next = p1->next;

p1->next = p2;

p1 = p1->next->next;

p2 = t;

}

}

**两个链表相加**

ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {

return help(l1, l2, 0);

}

ListNode\* help(ListNode\* l1, ListNode\* l2, int carry)

{

if (l1 || l2 || carry)

{

int sum = (l1 ? l1->val : 0) + (l2 ? l2->val : 0) + carry;

carry = sum / 10;

ListNode \*p = new ListNode(sum % 10);

p->next = help(l1 ? l1->next : 0, l2 ? l2->next : 0, carry);

return p;

}

return NULL;

}

**哈希函数的构造**

1. **直接定址法**

如果关键字是连续的，且数目比较少的时候，可以直接用关键字本身作为哈希地址。例如1-100岁之间人口的数目，直接用年龄作为哈希地址。在处理字符的时候，ascii字符总共有256个，同样也可以直接用字符作为哈希表的下标。

1. **数字分析法**

当关键字的位数很多的时候，可以取关键字中某些取值比较均匀的数字作为哈希。比如在用一个区间的手机号码作为关键字的时候，因为范围已知，而且手机号码的前面很多位都是相同的，所有可以舍弃前几位，用后面分布比较均匀的位数作为哈希地址。

1. **除留余数法**

取一个数 p，关键字对这个数取模，余数作为哈希地址。

一般来说：p 取不大于哈希表表长，并且最接近表长的素数。

**哈希表解决冲突的方法**

1. **线性探测**

当某个关键字的哈希地址发生冲突，从该地址的下一个地址开始顺序试探，直到找到一个空的单元。

**优点：**只要哈希表没有满，一定可以找到一个不冲突的哈希地址。

**缺点**：容易产生二次聚集，存入的记录连成一片。删除元素很困难。

1. **再哈希法**

同时构造多个哈希函数，当用一个哈希函数计算的地址发生冲突，再用其他的哈希函数计算。

**优点：**不容易产生聚集。缺点：计算时间增加。

1. **链地址法**

思想：将所有哈希地址相同的元素构成一个单链表，单链表的头指针存放在哈希表的对应地址单元中。

**优点**：处理简单，不会产生聚集；同时可以动态增加，适合事先不知道表长度的情况；删除结点的操作也很容易实现。

**缺点**：指针需要额外的空间。

**字符串转换成整数**

int StrToInt(string str) {

if (str.empty())

return 0;

int i = 0;

while (isspace(str[i])) //跳过空格

i++;

int op = 1;

if (str[i] == '+' || str[i] == '-') {

if (str[i] == '-')

op = -1;

i++;

}

long long sum = 0;

while (i < str.size()) {

if (!isdigit(str[i]))

return 0;

sum = sum \* 10 + str[i] - '0';

if (sum\*op > INT\_MAX) return INT\_MAX;

if (sum\*op < INT\_MIN) return INT\_MIN;

i++;

}

return sum\*op;

}

**整数次方**

double Power(double base, int n) {

if (n == 0)

return 1;

if (n == 1)

return base;

if (base == 0)

return 0;

int d = n;

if (d < 0) {

d = 0 - d;

base = 1 / base;

}

double t = Power(base, d / 2);

if (d & 1)

return t \* t \* base;

return t \* t;

}

**1+2+3+…+n**

class T {

public:

T() { N++; sum += N; }

static int getSum() { return sum; }

private:

static int N;

static int sum;

};

int T::N = 0;

int T::sum = 0;

int Sum\_Solution(int n) {

T \*a = new T[n];

delete[]a;

a = NULL;

return T::getSum();

}

**不用加减乘除做加法**

int Add(int num1, int num2)

{

if (num2 == 0) return num1;

int sum = 0;

int carry = 0;

while (num2) {

sum = num1 ^ num2;

carry = (num1 & num2) << 1;

num1 = sum;

num2 = carry;

}

return num1;

}

**二进制中1的个数**

与位：

int countBits(uint32\_t n) {

int count = 0;

while (n) {

count++;

n = n&(n - 1);

}

return count;

}

递归：

int countBits(uint32\_t n) {

if (n == 0)

return 0;

if (n == 1)

return 1;

return countBits(n / 2) + n % 2;

}

**整数拆分后乘积最大**

**假设n>=2**

int integerBreak(int n) {

if (n == 2) return 1;

if (n == 3) return 2;

int ret = 1;

while (n>4)

{

ret \*= 3;

n -= 3;

}

return ret\*n;

}

**深度优先/回溯法**

**N皇后问题**

class Solution {

public:

vector<vector<string>> solveNQueens(int n) {

vector<vector<string>> re;

vector<int> col(n, 0); //保存每列是否已经放置

vector<int> queen(n, 0); //代表n行中放皇后的列数，queen[0]为第一行皇后所在列，...

backTrack(queen, 0, re, col);

return re;

}

void backTrack(vector<int> &queen, int row, vector<vector<string>> &re, vector<int> &col) {

if (row == queen.size()) {

vector<string> strs(queen.size(), string(queen.size(), '.'));

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

strs[i][queen[i]] = 'Q';

re.push\_back(strs);

return;

}

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

if (col[i])

continue;

col[i] = 1;

queen[row] = i; //尝试 row 行的皇后放在 i 列

if (check(queen, row))

backTrack(queen, row + 1, re, col);

col[i] = 0;

}

}

bool check(vector<int> queen, int row) { //检查前 row 行的放置是否正确

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

if (queen[i] == queen[row] || abs(queen[i] - queen[row]) == abs(i - row))

return false;

return true;

}

};

**岛的个数**

‘1’表示岛，‘0’表示水，一个岛被水包围（上下左右都为‘0’，边界外定义为‘0’）

int numIslands(vector<vector<char>>& grid) {

if (grid.empty())

return 0;

int count = 0;

int m = grid.size();

int n = grid[0].size();

for (int i = 0; i < m; i++) {

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

if (grid[i][j] == '1') {

count++;

dfs(grid, i, j); //找到一个岛，深度优先遍历，标记所有相邻的格子

}

}

return count;

}

void dfs(vector<vector<char>>& grid, int i, int j) {

if (i<0 || i >= grid.size() || j<0 || j >= grid[0].size() || grid[i][j] != '1')

return;

grid[i][j] = '2'; //用‘2’来表示已经被访问过

dfs(grid, i - 1, j);

dfs(grid, i, j - 1);

dfs(grid, i + 1, j);

dfs(grid, i, j + 1);

}

**动态规划**

**不同的路径**

**从start到finish，一次只能往下或往右走一步。**



对于每格，只有两条路径到达：左边和上边（如果存在），所以到达该格的路径数等于到达左边格子和到达上边格子路径数之和。

int uniquePaths(int m, int n) {

vector<vector<int>> dp(m, vector<int>(n, 0));

dp[0][0] = 1;

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

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

{

if (i>0) //上边存在

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

if (j>0) //左边存在

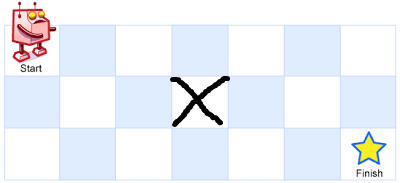
dp[i][j] += dp[i][j - 1];

}

return dp[m - 1][n - 1];

};

路径上有障碍（如下图），不能通过。



int uniquePathsWithObstacles(vector<vector<int>>& grid) {

if (grid[0][0] == 1)

return 0;

int m = grid.size();

int n = grid[0].size();

vector<vector<int>> dp(m, vector<int>(n, 0));

dp[0][0] = 1;

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

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

{

if (grid[i][j] == 0)

{

if (i>0)

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

if (j>0)

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

}

}

return dp[m - 1][n - 1];

}

**矩阵中和最小的路径**

从最左上角到最右下角，经过路径的和最小。每次只能往右或者往下走一步。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 2 | 4 | 3 | 7 | 6 |
| 2 | 5 | 1 | 2 | 6 | 5 |
| 4 | 3 | 2 | 4 | 5 | 2 |

对于每格，到达该格的只能从左边或上边（如果存在），最短的和=min（到达左边最短和，到达上边最短和）+本格数值。

int minPathSum(vector<vector<int>>& grid) {

int m = grid.size();

int n = grid[0].size();

vector<vector<int>> dp(m, vector<int>(n, 0));

dp[0][0] = grid[0][0];

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

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

{

if (i>0 && j>0) //从左边和上边选出路径和最小值

dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];

else if (i>0)

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

else if (j>0)

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

}

return dp[m - 1][n - 1];

}

**三角形中路径和最小**

从顶点到底边，每次只能向下行相邻格。

[ [2],

[3,4],

[6,5,7],

[4,1,8,3] ]

int minimumTotal(vector<vector<int>>& triangle) {

int n = triangle.size();

vector<int> cur(triangle[n - 1]);

for (int i = n - 2; i >= 0; i--)

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

cur[j] = min(cur[j], cur[j + 1]) + triangle[i][j];

return cur[0];

}

**产生N对括号对**

vector<string> generateParenthesis(int n) {

string s1("");

vector<string> re;

Permutation(s1, n, n, re);

return re;

}

void Permutation(string s1, int left, int right, vector<string> &re)

{

if (left == 0 && right == 0)

{

re.push\_back(s1);

return;

}

if (left != 0) Permutation(s1 + "(", left - 1, right, re);

if (right != 0 && left < right) Permutation(s1 + ")", left, right - 1, re);

}