**2021**年京东精选**50**面试题及答案

1. 求此N的最小公倍数。把每个数字分解质因数，算他们 每个质因数的贡献，然后乘起来。我的代码没写好（算质因

数不用这么慢的)。

#include<bits/stdc++. h>

using namespace std;

typedef long long 11;

#define maxn 100009

int fact [maxn];

bool prime [maxn];

11 mod = 987654321;

int cal (int *t,* int p) {

int ent = 0;

while(t % p == 0) { cnt++;

t /= P；

return ent;

void first () {

memset (prime, true, sizeof (prime)); prime[1] = false;

for(int i = 2; i <= 100000; i++： { int top = sqrt(i);

for(int j = 2; j <= top; j—) { if(i % j == 0){ prime[i] = false; break;

void solve(int Limit) { first ();

for (int i = 2; i <= Limit; i++' { int top = sqrt(i);

for (int j = 2; j <= top; ji) { if(prime[j] && i % j == 0) {

fact[j] = max (fact [j], cal (i, j));

I

if(prime[i])

fact [i] = max (fact [i], 1);

}

}

int main() {

11 n;

cin^^n;

solve (n);

11 ans = 1;

for(11 i = 1; i <= n; i++) {

for (11 j = 1； j <= fact[i]; j++) {

ans = ans \* i % mod;

}

}

cout<<ai^s<<endl;

return 0;

1. 去掉字符串构成回文。其实是经典的求回文子序列个数。

# inc lude <b i t s /s t dc++. h>

using namespace std;

typedef long long 11;

11 f[59] [59];

string str;

11 dfs(int i, int j) {

i£(i > j) {

return 0;

}

if(i == j) {

f[i] [j] = 1;

return f [i] [j];

}

[j] != 0){

return f [i] [j];f [i] [j] = dfs (i, j - 1) + dfs(i + 1, j) - dfs(i + 1, j - 1); if (str [i] == str[j])

f [i] [j] += dfs(i + 1, j - 1： + 1;

return f [i] [j];

int main。{

cin>>sh;

int len = str.length();

cout«dfs(0, len - l)«endl;

return 0;

1. 象棋的马走K步之后到(X,Y)的方案数。直接递推。

#include<bits/stdc++. h>

using namespace std;

typedef long long 11;

11 dp[10][10] [3];

11 mod = le9 + 7;

int

int

int

dx[8] = *{~2,* dy[8] = {-1, check(int x, if (x >= 0 &&

-1,

-2, int

L 2, 2,

-2, -1, y) {

x <= 8 && y return true;

L -L -2};

L 2, 2, 1}；

>=0 && y <= 8)

0；

< 8; i++) { dx[i]; dyti]；

i£ (check (tx.

ty)) {

return false;

void cal (int x, int y, int state) { dp [x] [y] [state]: for(int i = 0; i

int tx = x +

int ty = y +

dp[x] [y] [state] = (dp[x] [y] [state] + dp[tx] [ty] [state 1]) % mod;

int main() {

int K;

cin>> K;

int state = 0, nowstate;

dp[0] [0] [0] = 1;

while (K—) {

state = state " 1;

for(int i = 0; i <= 8; i++) {

for(int j = 0; j <= 8; j++) {  
cal (i, j, state);

int x, y;

cin^>x>>y;

cout«dp[x] [y] [state] «endl;

return 0;

1. 如何验证图的连通性？

#include<iostream>

#include<queue> ffinclude <stdio.h>

using namespace std;

ffdefine MAX\_VNUM 10

typedef struct

{

int weight;

}Adj,AdjMatrix[MAX\_VNUM][MAX\_VNUM];

typedef struct

{

AdjMatrix adjM;

i nt vNum;

}adjGraph;

〃创建一个图,节点从。幵始，注意传入引用 void CreateGraph(adjGraph &G)

{

cout<<"输入节点个数:"«endl; cin»G.vNum;

cout<<"输入图的邻接矩阵:"«endl;

for (int i=0;i<G.vNum;i++)

{

for (int j=0;j<G.vNum;j++)

{ cin»G. ad jM[i][j]. weight;

}

}

}

/瀚出一个图

void print(adjGraph G)

{

for(int i=0;i<G.vNum;i++)

{

for(int j =0;j<G.vNum;j ++)

{

cout<<G.adjM[i][j].weight<<"

}

cout«endl;//1寻换行流写入输出流，清空输出緩冲区

}

}

//war shall算法判断图的连通性

bool connectivityWarshall(adjGraph G)

{

adjGraph temp;//临时判断矩阵 temp.vNum = G.vNum;

〃初始化临时判断矩阵

for (int i =0;i<temp.vNum;i++)

{

for(int j =0;j<temp.vNum;j++)

{

If (G.adjM[l][J].weight)

temp.adjM[i][j].weight = 1; else

temp.adjM[i][j].weight = 0;

}

temp.adjM[i][i].weight = 1;

〃矩阵乘法算法Marshall, R(a)

for (int a =0;a<temp.vNum;a++)

{

for (int b=0;b<temp.vNum;b++)

{

if(temp.adjM[a][b].weight)

{

for (int c = 0;c<temp.vNum;c++) {

if (temp.adjM[c][a].weight) temp.adjM[c][b].weight = 1;

}

}

}

}

〃进行判断

for (int i=0;i<temp.vNum;i++；

{

for (int j=0;j<temp.vNum;j++)

{

if (!temp.adjM[i][j].weight)

return false;

}

}

return true;

〃广度优先搜索判断连通性

bool connectivityBFS(adjGraph G) {

queue〈int> q; //B月白队列用途？ bool visit [MAX\_VNUM]; 〃访问数组 int count = 0;

memset(vlslr,0,slzeof(visit)); q.push(0); 〃。节点入队列

while(!q.empty())

{

int v = q .front();

visit[v] = true;

q.pop();

count++;

〃与联通且没有被访问过节点人队列

for (int i =0;i<G.vNum;i++)

{

if (G.adjM[v][i].weight)

{

if(!visit[i])

{

q.push(i);

}

}

}

}

if (count == G.vNum)

return true;

else

return false;

}

〃深度优先搜索判断图的连通性,传逢数组会改变值,visit需初始化 void dfs\_visit(adjGraph G,int firstNode,bool visit[]) {

visit[firstNode] = 1;

for(int i=0; i<G.vNum;i++)

{

if(G.adjM[firstNode][i].weight & !visit[i]) dfs\_visit(G,i,visit);

bool connectivityDFS(adjGraph G)

{

bool visit [MAX\_VNUM]; 〃访问数组 memset(vlslr,0,slzeof(visit));

dfs\_visit(G,0,visit); 〃从。节点开始访问

for(int i=0;i<G.vNum;i++)

if (visit[i] == false) return false;

return true;

int main()

{

adjGraph G;

CreateGraph(G);

//print(G);

if (connectivityWarshall(G)) cout<<"连通"<<endl; else cout<〈"不连通"<<endl;

system("pause");

return 0;

1. git pull 和 git merge 区别?

你修改好了代码，先要提交

git commit "commit message”

然后有两种方法来把你的代码和远程仓庠中的代码合并：

1. git pull这样就直接把你本地仓库中的代码进行更新但问题是可能会有冲突 （conflicts）,个人不推荐。
2. 先git fetch origin （把远程仓库中origin最新代码取回）＞ 再git merge origin/master （把本地代码和已取得的远程仓库最新代码合并），如果你的改动和远 程仓库中最新代码有冲突，会提示，再去一个一个解决冲突，最后再从1开始。
3. 如果没有冲突，git push origin master,把你的改动推送到远程仓库中。
4. 手写快速排序代码.

public static int partition (int ari [], int int r) { //基准元素设为第一个

int v = arr [1];

//i指向基准的下一个元素，j指向最后一个元素

int i = 1+1, j = r;

while(true){

while(i <= r && axr [i] < v) i++;

while(j > 1 && axr[j] > v) j—;

//循环终止条件

if (i > j) break;

/发换皿[i]与皿[j]

int t = axr [i];

axr [i] = axr [j];

axr[j] = t;

i++；

/R寻基准元素与axr[j]交换 int t = axr [1];

axr [1] = axr [j];

axr[j] = t;

//返回基准元素所在位置 return j;

1. 内存分配方式有几种？

内存分配方式：

（1） 符号起始的区块（丄奖段）：通常有的是存放程序中未初始化或者初始化为。的 变量的和静态数据的区域。bss属于静态为存分配，程序结束后静态资源变量由系统自 动释放。

（2） 数据段：通常指存放程序中已初始化的全局变量的一块内存区域。也属于静态内 存分配。

（3） 代码段：有时也叫文本段，通常指的是用来存放程序执行代码（包含类成员函数 和全局函数及其他函数代码）*，*这部分区域的大小在程序运行前就已经确定，也有可能 包含一些只读的常数变量，例如字符串变量。

（4） 堆（heap）:用于存放进程运行中被动态分配的内存段，大小不固定。当进程调用 malloc或者new等函数时，新分配的内存就被动态添加到堆上（堆被扩张），当使用 free或者delete等函数释放内存时，被释放的内存从堆中被删除。需要注意的是，它 与数据结构中的堆是两回事，它的分配天式类似于链表。

（5） 栈（stack）:存放程序临时创建的局部变量，不包括static声明的变量，static 意味着在数据段中存放。除此之外，当幽!被调用时，其参数也会被压到栈中，并在调 用结束后，函数的返回值也会被放到栈中。栈由编译器自动释放。其操作方式类似于数 据结构中的柱。柱内有分配运算内置于处理器的指令集中，一般使用容存器来有取，效 率很高，但是分配的内存容量有限。

1. 在VC 6.0中定义一个数组a[1024][1024],能够运行吗？

不能，因为运行的时刻没有那么大的可分配内存块，栈内存不够，默认是1M的空间。

1. 请描述动态规划的基本思想？

分治法

将一个规模为n的问题分解为K个规模较小的子问题，这些子问题互相独立且与原问题 相同。逢归的解决这些问题，然后将各个子问题的解合并得到原问题的解

贪心法

当前的选择可能要依赖于已经做出的选择，但不依赖于有待于做出的选择和子问题。因 此贪心法是自顶向下，一步一步地做出貪心的选择

动态规划

动态规划的实质是分治思想和解决冗余，因此动态规划是一种将问题实例分析为更小 的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最优化问题的 算法策略

动态规划所针对的问题有一个显著的特征,即它对应的子问题树中的子问题呈现大量的 重复。动态规划的关键在于，对于重复的子问题，只在第一次遇到时求解，并把答案保 存起来，让以后再谒到时直接引用，不必要重新求解

1. 分布式服务接口请求的顺序性如何保证？
2. 首先，一般来说，从业务逻辑上最好设计系统不需要这种顺序的保证，因为一旦引入 顺序性保障，会导致系统复杂度的上升，效率会降低，对于热点数据会压力过大等问题。
3. 操作串行化。

首先使用一致性hash负载均衡策略，将同一个id的请求都分发到同一个机器上面去处 理，比如订单可以根据订单id。如果处理的机器上面是多线程处理的，可以引入内存 队列去处理，将相同id的请求通过hash到同一个队列当中，一个队列只对应一个处理 线程。

①最好能将多个操作合并成一个操作。

1. 编译时多态和运行时多态的区别？

编译时多态

主要是方法的重载，通过参数列表的不同来区分不同的方法。

运行时多态

也叫作动态绑定，一般是指在执行期间（非编译期间）判断引用对象的实际类型，根据 实际类型判断并调用相应的属性和方法。主要用于继承父类和实现接口时，父类引用指 向子类对象。

1. 常用的内存管理方法有哪几种？

段式

页式

段页式

1. 已知某二叉树的后序遍历序列是dabec,中序遍历序列 是deabc,它的前序遍历序列是什么？

cedba

1. 给定字符串(ASCII码0-255)数组，请在不开辟额外空 间的情况下删除开始和结尾处的空格，并将中间的多个连续 的空格合并成一个。例如：” i am a little boy. “，变成” iam a little boy”，C++语言实现，不要用伪代码作答，函数输入输 岀请参考如下的函数原型：

型：

void FoxmatString(char str [], int len) {

*}*

#include<stdio. h>

#include〈string. h>

void FormatString (char str [], int len)

[

if (str == NULL | | len <= 0) {

return;

int i = 0;

int j = 0;

if (str[i] ==‘‘){

while (str [i] == ' ') {

++i；

while (str[i] != ' ){

if (str [i] ==''&& str [i - 1] == ' ' | | str [i + 1] == ' \0‘){

++i；

continue;

}

str[j++] = str [i++];

}

str[j] = '\0‘；

}

int main() {

char a[] = i am a little boy. ”；

int len = strlen(a);

printf (°%d\n°, len);

Format String (a, len);

printf(°%d\n°, strlen(a));

printf(°%s\n°, a);

return 0;

}

16.给定一颗二叉树，以及其中的两个node (地址均非空)， 要求给岀这两个node的一个公共父节点，使得这个父节点 与两个节点的路径之和最小。描述你程序的最坏时间复杂 度，并实现具体函数，函数输入输岀请参考如下的函数原型； 型：

stmcy TreeMode {

TreeHode\* left; /月^向左子树

TreeHode\* right; /滞子树

TreeHode\* f ather; /潴向父亲节点

*I；*

TreeMode\* LowestCommonAncestor (TreeHode\* f irst, TreeMode\* second) {

*}*

int no deHeight(TreeNode\* node)

int height = 0; while(no de != NULL) height++;

node = no de~'father;

}

I

TreeNode\* LowestCommonAncestor(TreeHode\* first, TreeNode\* second)

[

int diff = no deHeight(first) - no deHeight(second);

if (diff > 0)

[

while(diff > 0)

[

first = first~>father;

diff—;

}

}

else

[

while(diff < 0)

[

second = sec ond~ ather.

diff++;

}

}

while(first != second)

[

first = first->father;

second = sec ond-ather;

}

return first;

}

17.计算第K个能表示(2Z \* 3勺\* 5M)的正整数(i,j,k为整 数)？其前7个满足此条件的数分别是1,234,5,6,8.

public class Main

{

public static void main( String [. args)

{

int [] a = new int [1501];

a[l] = 1；

TreeMap<Integer. Integer〉map = new TreeMap<Integer. Integer),();

Deque<Integer> Q2

new AxrayDeque^Integer^ ();

Deque<Integer> Q3

new AxrayDeque^Integer^ ();

Deque<Int eger> Q5

new AxrayDeque^Integer^ ();

map. put(2, 2);

map. put(3, 3);

map. put(5, 5);

for (int i = 2; i < 1501; i—) {

if (map. isEmpty ())

break;

Map. Ent ry< Int eg er. Integer〉*e* = map. pollFirstEntry (); int key = e. getKey ();

int val = e. getValue ();

if (val == 5) {

Q5. add (key \* 5);

map. put(Q5. pollFirs~ (), 5);

} else if (val == 3) {

Q5. add (key \* 5);

Q3. add (key \* 3);

map. put(Q3. pollFirsi (), 3);

} else {

Q5. add (key \* 5);

Q3. add (key \* 3);

Q2. add (key *\* 2);*

map. put(Q2. pollFirsi(), 2);

a[i] = key;

}

Scanner sc = new Scanner (System, in); while (sc.hasNextO) {

System, out. prmtln(a[sc.nextlnt ()]);

18. B-树和B+树的区别是什么？

B-树是一种多路搜索树（并不是二叉的。）*，*一颗m阶的B-树，或为空树，或者定义 任意非叶子结点最多只有M个儿子。

且M>2;根结点的儿子数为［2, M］。

除根结点以外的非叶子结点的儿子数为［H/2］。

每个结点存放至少M/2-1 （职上整）和至多M-1个关键字；（至少2个关键字）非叶子 结点的关键字个数=指向儿子的指针个数T ；

B+树，B+树是B-树的变体，也是一种多路搜索树：其定义基本与B-树同。

B-树是一种 多路搜索树（并不是二叉的。），一颗m阶的B-树，或为空树， 或 者定义任意非叶子结点最 多只 有M个儿子。

且M>2;根结点的儿子数为［2, M］。

除根结点以外的非叶子结点的儿子数对［M/2］。

每个结点存放至 少M/^l （职上整）和至多M-1个 关键 字；（至少2个关 键字）非叶子结点的关键字个数=指向儿子指针个数T；

B+树，B+树是B-树的变体，也是一种多路搜索树：其定义基本与B-树同。

19.分布式服务接口的幕等性如何设计（比如不能重复扣

款）？

所谓幕等性，就是说一个接口*，*多次发起同一个请求，你这个接口得保证结果是准确的, 比如不能多扣款，不能多插入一条数据，不能将统计值多加了 1。这就是幕等性，不给 大家来学术性词语了。

其实保证幕等性主要是三点：

（1） 对于每个请求必须有一个唯一的标识，举个例子：订单支付请求，肯定得包含订 单id,—个订单id最多支付一次，对吧

（2） 每次处理完请求之后，必须有一个记录标识这个请求处理过了，比如说常见的方 案是在吋四1中记录个状态嗜的，比如支付之前记录一条这个订单的支付流水，而且支 付流水采

（3） 每次接收请求需要进行判断之前是否处理过的逻辑处理，比如说，如果有一个订 单已经支付了，就已经有了一条支付流水，那么如果重复发送这个请求，则此时先插入 支付流水，orderld已经存在了，唯一键约束生效，报错插入不进去的。然后你就不用 再扣款了。

（4） 上面只是给大家举个例子，实际运作过程中，你要结合自己的业务来，比如说用 redis用皿derid作为唯一键。只有成功插入这个支付流水，才可以执行实际的支付扣



要求是支付一个订单，必须插入一条支何流水，order.id建一个唯一键，unique key 所以你在支付一个订单之前，先插入一条支付流水，ordez\_id就已经进去了 你就可以写一个标识到re dis里面去，set order\_id payed,下一次重复请求过来了， 先查：redis的order\_id对应的value,妇果是payed就说明已经支付过了，你就别重 复支付了

然后呢，你再重复支付这个订单的时候，你写尝试插入一条支付流水，数据库给你报错 了，说unique key冲突了，整个事务回滚就可以了

来保存一个是否处理过的标识也可以，服务的不同实例可以一起操作redis。

20. C和C++分配释放内存区别？

0.属性

new/delete是C++关键字，需要编译器支持。malloc/free是库函数，需要头文件支持。

1. 参教

使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计 算。而malloc则需要显式地指出所需内有的尺寸。

1. 返回类型

new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进 行类型转换，故“w是符合类型安全性的操作符。而malloc内存分配成功则是返回void *\*，*需要通过强制类型转换将void吋旨针转换成我们需要的类型。

1. 分配失败

new内存分配失败时，会抛出bac\_alloc异常。malloc分配内存失败时返回NULL。

1. 自定又类型

new会先调用operator new |^|数，申请足够的内存（通常底层使用malloc实现）。然 后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用 析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。 malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对 象构造和析构工作。

1. 重载

C++允许重载new/delete操作符，特别依，布局new的就不需要为对象分配内存，而是 指定了一个地址作为内存起始区域，new在这段内存上为对象调用构造函数完成初始化 工作，并返回此地址。而malloc不允许重载。

1. 内存区域

new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函数 从堆上动态分配内有。自由有储区是。淄于new操仲符的一个抽象概念，凡是通过rew 操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统 所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内 存，使用free释放已分配的对应内存。自由存储区不等于堆，如上所述，布局new就 可以不位于堆中。

21. 一个单词单词字母交换，可得另一个单词，如 army->mary,成为兄弟单词。提供一个单词，在字典中找到 它的兄弟。描述数据结构和查询过程。

解法一：

使用hash\_map和链表。

首先定义一个key,使得兄弟单词有相同的key,不是兄弟的单词有不同的key。例如, 将单词按字母从小到大重新排序后作为其key,比如bad的key为abd, good的key为 dgooo

使用链表将所有兄弟单词串在一起，hash\_map的key为单词的key, value为链表的起 始地址。

幵始时，先遍历字典，将每个单词都按照key加入到对应的链表当中。当需要找兄弟单 词时，只需求取这个单词的key,然后到hash-map中找到对应的链表即可。

这样创建hash-map时时间复杂度为0 (n),查找兄弟单词时时间复杂度是0 (1)。

解法二：

同样使用hash\_map和链表。

将每一个字母对应一个质数，然后让对应的质数相乘，将得到的值进行hash,这样兄 弟单词的值就是一样的了，并且不同单词的质数相乘积肯定不同。

使用链表将所有兄弟单词串在一起，hash-map的key为单词的质数相乘积,value为链 表的起始地址。

对于用户输入的单词进行计算，然后查找hash,将链表遍历输出就得到所有兄弟单词。 这样创建hash-map时时间复杂度为0 (n),查找兄弟单词时时间复杂度是*0*⑴。

#include<stdlib.h>

*it inc* lude<s tring. h>

Sdefine MAX\_SIZE 287

typedef struct hash^no de

{

char \*word;

struct hash\_node \*next;

}hash\_no de, \*hash\_map;

bin[!WLSIZE]= {MULL};

unsigned int get\_index(char \*pWord)//get hash index

[

int lerFstrlen(pWord);

int i;

unsigned int index=l;

for(i=0;i<len; i++)

index=index\* (pWord [i]' A’ +1；; //这里如果是大写字母的话就会使负值,所 以要根据情况而定

return index%I.tAX\_SIZE;

}

void ins er t\_wor d(char \*pWord) //insert word, if collision happens, use link list

[

unsigned int index=get\_index(pWord);

printf("%d\n— index);

hash\_node \*p;

for (p=bin[index] ;pJ=NULL;p=p->xiext)

if (strcmp (p~>word, pWord) ==0：

return;

p=(hash\_no de\*)malloc(sizeof(hash\_node));

p->word=(char\*)malloc(strlen(pWord) +1);

strcpy (p->word, pWord);

p->word[strlen(pWord)]=, \0‘;

p->next=b in [index];// 不断的插入到表头就好,this will be efficient bin[index]=p;

}

void search\_brother(char \*pWord) //search brother words

[

unsigned int index=get\_index(pWord);

hash\_node \*p;

for(p=bin[index];pJ=NULL;p=p->next)

if (strcmp (pWord, p->word) !=0；

printfp->word);

}

void main。

[

char \*string [] = {"marv— "army— 〃:ramy〃};

int lerFsizeof (string)/sizeof (char\*);

int i;

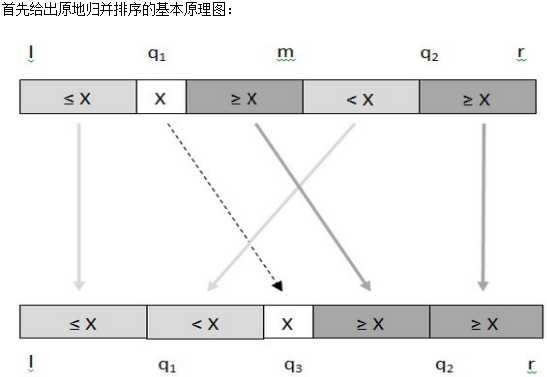
for(i=0;i<len; i++)

*ins* er t\_wor d (str ing [i]);

char wordf]=zzmaryz/;

search\_\_brother (word);

22.数组al[O,mid-l]和al[mid,num-l],都分别有序。将其 merge成有序数组al[O,num-1],要求空间复杂度0(1).



首先把给定的数组分成两部分,前一部分包括A m]中的元素，而后者则包括(m口]中的 元素。然后找到第一个数组中的中间值，也就是ql=(l侦)/2, ql位置的元素就是我们 要找的元素，大家可以举个例子自己算一下,当找到ql的时候,Q1前面的元素有Q1个, 这对接下来的编程很有影响，所以这点一定要弄清楚。这样第一个数组我们就分为了两 部分。接下来我们用Q1去划分(人工】这段元素，也是分成两部分。建议在取元素范围的 时候，(叫工】这段的前一部分个数为(q2-l-m)个，后一部分为(r - q2 +1)个,这样 约定后思路会清晰。接下来划分好了之后就是交换，有一种叫做block swapping的算 法，这个算法能在。(1)的空间复杂度下交换两个长度不相同的相邻存储区的数组。

#include<stdio. h>

*Siiic* lude^assert.

void swap (int \*a, int low, int high； {

while(low < high) {

int temp = \*(a + low);

\*(a + low) = \*(a + high);

\*(a + high) = temp;

low++;

high—;

void block\_exchange (int int low, int mid, int high) {

swap (a, low, mid);

swap (a, mid + 1$ high);

swap (a, low, high);

int b inar y\_s e ar ch (int value, int \*a, int low, int high) { /\*

如果数组中存在要查找的元素，那么返回high的位置的前后都有可能等于value 的值：

a： 1, *2,* 4,4, 7,9 value=4 mid=2返回的high位置的元素后有值等于value

b: 1, 2, 4,4, *7,*9,10 value=4 mid=3.返回的high位置的元素前有值等于value

c： 1,2, 7,9 value=4 high=3,返回的high值之前的元素都小于value,包括high 在内的以后得元素都大于value

\*/

assert(a != NULL);

while(low < high) {

int mid = low + (high - low： / 2;

if(value 仁 a[mid])

high = mid;

else

low = mid + 1;

}

return high;

void merge\_\_iri\_place (int \*a, int low, int mid, int high) {

int length! = mid 一 low +1;

int length2 = high - mid;

if(J(lengthl >= 0 && length2 >= 0))

return ;

if (lengthl >= length2) {

if (leng th2 <= 0)

return;

int ql = (low + mid) / 2;

int qZ = binary^sear ch (a [ql., a, mid 十 1 , high); int q3 = ql + (q2 - 1 - mid；;

ql, mid, q2 - 1); low, ql - 1, q3 - 1); q3 + 1, q2 - 1, high);

bio ck\_ex chang e (a, merge\_in\_place (a, merge\_in\_place (a.

} else {

if (leng thl <= 0)

return;

int ql = (mid + 1 int q2 = binary\_search(a[ql\_, a, low, mid);

+ high) /2;

int q3 = q2 + (ql - 1 - mid；; block\_exchange (a, q2, mid, ql);

merge\_in\_place (a, low, q2 - 1, q3 -1);

merge\_in\_place (a, q3 + 1, ql, high);

}

I

void main() {

int a[] = {爲 3, 5, 7,9, 2, 4,6,8,10};

int len = sizeof(a) /sizeof(int；;

int mid = len / 2 - 1;

merge\_in\_place (a, 0, mid, len - 1);

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

printf a[i]);

printfCXn^);

}

23. 一个url指向的页面里面有另一个url,最终有一个url指 向之前岀现过的url或空，这两种情形都定义为nulL这样构 成一个单链表。给两条这样单链表，判断里面是否存在同样 的url。url以亿级计，资源不足以hasho

本題可以抽象为有环和无环情况下的链表交叉问题：

情况一：两条单链表均无环

最简单的一种情况，由于两条链表如果交叉，他们的尾节点必然相等(Y字归并)， 所以只需要判断他们的尾节点是否相等即可。

情况二：两条单链表均有环

这种情况只需要拆开一条环路(注意需要保存被设置成null的节点)，然后判断 另一个单链表是否仍然存在环路，如果有在，说明无交叉，反之，则有交叉的情况。 情况三:两条单链表，一条有环路，一条无环路

这种情况显然他们是不可能有交叉的

附：如何判断一条单链表是否存在环路，以及找出环路的入口

快慢指针：在表头设置两个指针fast与slow, fast指针与slow指针同时向前移动， 但是f as t每次移动2个节点,slow每次移动1个节点，若fast指向null或者fast==slow 时停止，这时如果fast指向mil,则说明没有环路，若faSt==slow则说明有环路。 找环路入口：当fast==slow时，将fast重新指向表头。slow原地不动。然后fast和 slow在同时以每次一个节点的速度向前移动，当他们再次重合时，就是环路入口。证 明如下：

1.证明fast和slow肯定会重合 在slow和fast第一次相谒的时候，假定Slow走了 n步骤，环路的入口是在p步的时 候经过的*，*那么有slow走的路径：p+c = n; c为pl和p2相交点，距离环路入口 的距离；fast走的路径：p+c+k\*L =如；L为环路的周长，k是整数。显然，如 果从p+c点开始,pl再走n步骤的话，还可以回到p+c这个点同时p2从头开始走的话, 经过n步，也会达到p+c这点。

2. fast和slow在p+c点会重合,显然他们从环的入口点就开始重合

24.将单向链表reverse,如ABCD变成DCBA,只能搜索链表一 次.

#include <iostream>

using namespace std;

struct node

[

char data;

struct node \*next;

};

typedef struct node NODE;

void test\_exercise()

[

NODE \*head = new NODE; /健立附加头结点 head~>next = NULL;

/彳创建链表\*/

NODE \*curr ent, \*pr evi ous;

previous = head;

char irput;

cout « ^Irput your list table NODE data, end with

cin » irput;

while(irput J=)

{

current - new NODE;

current-〉data - irpu■:; current">next = NULL.

previous">next - curlent;

previous = pr evi ous"/ne x t;

cout « ^Irput your list table NODE data, end with cin » irput;

/欄出链表\*/ current - head">next; while(current != NULL)

[

cout << current-〉data << current - current~>next;

}

cout << endl;

湖转链表\*/ current - head">next;

NODE \*p = current~>next;

NODE \*q = p->next;

while(q != NULL)

[

p->next = current; current - p;

P = q;

q = q->next;

}

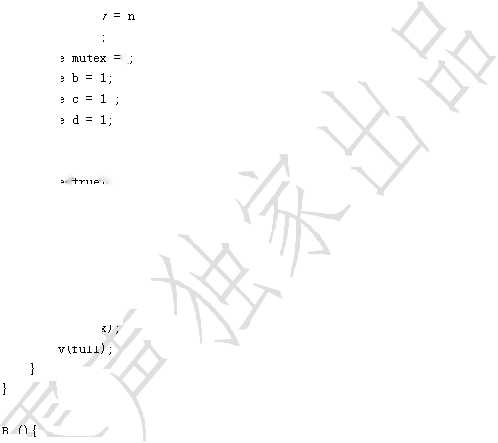
p->next = current;

current - p; head->next->next = NULL; head">next = current;

/欄出链表\*/

current = head~>next; while(current != NULL) {

cout << current-〉data << current - current~>next;



且B, C, D都读一次后，A才能再写。用P, V操作实现。

A (){

B Ut

empty full; mutex =1

25. A,B,C,D四个进程,A向Buf里面写数据,B,C,D向Buf里面读 数据，当A写完，且B,C,D都读一次后,A才能再写.用PV操作实 现.

响B,C, D四个进程，A向buf里面写数据，B, C, D向buf里面读数据, 当A写完，

semaphore semaphore semaphore semaphore semaphore semaphore

while(true){ P(empty);

P(b)；

Pic 丿；

P(d)； p(mutex); write ();

v(mutex): v(full)

while(true){ p(full); p(mutex); write ();

v(mutex); v(empty); v(b);

C(){

while(true){

零声学「謐理出品

2020年更多、更全、更新大厂面试资料Q群：762073882 p (full);

p(mutex);

write ();

v(mutex);

v(empty);

v(c);

}

I

D (){

while(true){

P (full);

p(mutex);

write ();

v(mutex);

v(empty);

v(d);

1. 进程和线程的区别介绍？

1、 首先是定义

进程：是执行中一段程序，即一旦程序被载入到内存中并准备执行，它就是一个进程。 进程是表示资源分配的的基本概念，又是调度运行的基本单位，是系统中的并发执行的 单位。

线程：单个进程中执行中每个任务就是一个线程。线程是进程中执行运算的最小单位。

2、 一个线程只能属于一个进程，但是一个进程可以拥有多个线程。多线程处理就是允 许一个进程中在同一时刻执行多个任务

3、 线程是一种轻量级的进程，与进程相比，线程给操作系统带来侧创建、维护、和管 理的负担要轻，意味着线程的代价或开销比较小。

4、 线程没有地址空间，线程包含在进程的地址空间中。线程上下文只包含一个堆栈、 一个寄存器、一个优先权，銭程文木包含在他的进程的文木片段中，进程拥有的所有 资源都属于线程。所有的线程共享进程的内存和资源。同一进程中的多个线程共享代 码段（代码和常量），数据段（全局变量和静态变量），扩展段（堆存储）。但是每个线程拥 有自己的栈段，寄存器的内容，栈段又叫运行时段，用来存放所有局部变量和临时变 里。

5、 父和子进程使用进程间通信机制，同一进程的线程通过读取和写入数据到进程变量 来通信。

6、 进程内的任何线程都被看做是同位体，且处于相同的级别。不管是哪个线程创建了 哪一个线程，进程内的任何线程都可以销毁、挂起、恢复和更改其它线程的优先权。线

程也要对进程施加控制，进程中任何线程都可以通过销毁主线程来销毁进程，销毁主线 程将导致该进程的销毁*，*对主线程的修改可能影响所有的线程。

7、子进程不对任何其他子进程施加控制，进程的线程可以对同一进程的其它线程施加 控制。子进程不能对父进程施加控制，正程中所有线程都可以对主线程施加控制。

相同点：

进程和线程都有ID脣存器组、状态和优先权、信息块，创建后都可更改自己的属性， 都可与父进程共享资源、都不鞭直接访问其他无关进程或线程的资源。

1. 请描述分布式的优势.

分布式结构就是将一个完整的系统，按照业务功能，拆分成一个个独立的子系统，在分 布式结构中，每个子系统就被称为“服务”。这些子系统能够独立运行在web容器中， 它们之间通过RPC方式通信。

举个例子，假设需要开发一个在线商城。按照微服务的思想，我们需要按照功能模块拆 分成多个独立的服务，如：用户服务、产品服务、订单服务、后台管理服务、数据分析 服务等等。这一个个服务都是一个个独立的项目，可以独立运行。如果服务之间有依赖 关系，那么通过RPC方式调用。

分布式的好处：

系统之间的耦合度大大降低，可以独立开发、独立部署、独立测试，系统与系统之间的 边界非常明确，排错也变得相当容易，开发效率大大提升。

系统之间的耦合度降低，从而系统更易于扩展。我们可以针对性地扩展某些服务。假设 这个商城要搞一次大促，下单量可能会大大提升，因此我们可以针对性地提升订单系统、 产品系统的节点数量，而对于后台管理系统、数据分析系统而言，节点数量维持原有水 平即可。

服务的复用性更高。比如，当我们将用户系统作为单独的服务后，该公司所有的产品都 可以使用该系统作为用户系统，无需重复开发。

1. 阅读下面代码，回答问题.

# include <stdio.h> nainO {

int su)n^ pad^ pAd;

Sum = pad = 5;

pAd = ++sum^ pA<R+^ ++pad; printf ( , pad);

果斟

6

1. 变量a是一个64位有符号的整，初始值用16进制表示 为:0x7FFFFFFFFFFFFFFF;变量b是一个64位有符号的整数，初 始值用16进制表示为0x8000000000000000.则a+b的结果 用10进制表示为多少？

-1

1. if [$2 -a $2 = "test"]中-a 是什么意思？

并且

1. 已知 int 占个字节,unsigned char 占 1 个字节,unsigned int number=Oxffaabcdd;下 种方式 可以将number的 值变为 Oxffaacddd?
2. \*( (unsigned char\*) (&number) + l)=0xcd;
3. number = (number & OxffffiJOff) | OxOOcdOO;
4. number = (number & 0xffee43dd) | Oxbbaacddd;
5. number = (number & Oxffccbcff) + 0x1100;

A,B,C

1. 模式串的长度是m,主串的长度是n(m<n),使用KMP算法 匹配的时间复杂度是？

0 (m+n)

KMP字符串匹配时间复杂度一定是0(N)銭性

33.求O-n之间二进制内没有连续三个1的数的个数.

#include <iostreajn>

using namespace std;

int v[100];

int dp [100] [3];

int dfs (int len, int flag, bool limit； {

if (flag == 3)return 0;

if (len == 0) return 1;

if (J limit&&dp [len] [flag] != T； return dp [len] [flag];

int maxx = limit ? v[len] : 1;

int ent = 0;

for (int i = 0; i <= maxx; i++)[

if (i == 0){

ent += dfs (lernl, 0, == v[len]):

}

else{

ent += dfs (lern 1, flag+1 , == v[len]);

}

}

return limit ? ent : dp[len] [flag] = ent;

}

int solve(long long x){

memset (v, 0, sizeof (v));

int k = 0;

while (x){

v[++k] = x % 2;

x »=1;

}

return df s (k, 0, true);

}

int main() {

memset(dp, -1, sizeof(dp));

long long a；

cin >> a;

cout«solve (a) «endl;

return 0;

34. C++ 智能指针 shared\_ptr、weak ptr 的实现.

Counter类

Counter 象的目地就是用来申请一个块内存来存引用计数。shareCount是SharedPtr 的引用计数，weakCount是弱引用计数。

当shareCount为0时，删除X对象。

当 weakCount 为 0 同时 shareCount 为 0 时，删除 Count er\*对象。

Counter实现如下：

class Counter

[

public：

int shareCount - 0;

int weakCount = 0;

};

SharedPtr类

主要的成员函数包括：

默认构造函数

参数为T\*的explicit单参数构造函数

参数为WeakPtr&的explicit单参数构造丞数

拷贝构造函数

拷贝赋值函数

析构函数

隐式类型转换操作符operator bool ()

operator -〉()

operator \* ()

ShazedPtr实现如下：

template<class T> class WeakPtr;

template<class T> class SharedPtr

{

public：

friend class WeakPtr<T>; /防便v/eak\_ptr与share」h设置弓囲计数和赋值。

SharedPtr ()

:m\_pResource(nullptr)

, m\_pCounter(new Count er())

[

m\_pCount er">shareCount = 1;

explicit SharedPtr(T\* pResource = nullptr)

:m\_pResource(pResource)

, m\_pCounter(new Count er())

m\_pCount er~>shareCount - 1;

SharedPtr (const WeakPtr<T>& other) // 供 WeakPtr 的 lock。使用 :m\_pResource (other. m\_pResource)

, m\_pCounter(other・ m\_pCounter)

[

if (0 -- m\_pCounter">shareCount) m\_pResource - nullptr;

SharedPtr(const SharedPtr<T>& o^her)

:m\_pResource (other">m\_pResource)

, m\_pCounter(other">m\_pCounter)

[

■H-(m\_pCounter->shareCount); // 増加引用计数

SharedPtr<T>& operator = (const SharedPtr<T>& other)

[

if (this == Mother) return xthis;

release ();

m\_pCounter = other・ m\_pC ounter; m\_pResource = other. m\_pResource;

++(m\_pCounter->shareCount); // 増加引用计数

return \*this;

}

"'SharedPtr ()

I

release ();

T& operator bool ()

[

return m\_pResource != nullpir;

// 如果nullptr == m\_pResource,抛出异常 return \*m\_pResource;

}

T\* operator -> ()

[

return m\_pResource;

}

private：

void release ()

[

// T\*肯定由 SharedPtr 释放，Count er\*如果没有 WeakPtr,也由 SharedPtr 释放

--m\_pCount er~>shareCount;

if (0 == m\_pCount er~>shareCount)

[

delete m\_pResource; m\_pResource - nullptr;

if (0 == m\_pCount er~>weakCount)

[

delete m\_pCounter;

m\_pCounter = NULL;

public：

T\* m\_pResource = nullptr;

Counter\* m\_pCounter - nullptr;



WeakPtr类

主要的成员函数包括：

默认构造函数

参数为SharedPh&的explicit单参数构造函数

拷贝构造函数

拷贝赋值函数

析构函数

lock。函数：取指向的SharePtr,如果未指向任何ShazePtr,或者已被析构，返回指 向 nullptr 的 SharePtr

expired ()©数：是否指向SharePtr,如果指向Share Ptr其是否已经析构

release ()© 数

WeakPtr实现如下：

template<class T> class WeakPtr

[

public：

friend class SharedPtr<T>; *If%*便 weak\_ptr 与 share\_ptr 设置引用计数和赋值。

WeakPtr ()

m\_pResource(nullptr)

, m\_pCount er (new Count er ())

[

m\_pCount er-^weakCount = 1;

}

WeakPtr(SharedPtr<T>& other)

:m\_pResource(other. m\_pResource)

,m\_pCount er (other. m\_pCounter)

[

++(m\_pCounter->weakCount);

*}*

WeakPtr(WeakPtr<T>& other)

:m\_pResource(other. m\_pResource)

, m\_pCounter(other. m\_pCounter)

{

++(m\_pC ount er-we akC ount);

}

WeakPtr<T>& operator = (WeakPtr<T>& other)

[

if (this == &other) return xthis;

release ();

m\_pCounter = other. m\_pCount er;

m\_pResource = other. m\_pResource;

++m\_pCount er-^weakCount;

return \*this;

release ();

m\_pCounter = other・ m\_pCounter;

m\_pResource - other. m\_pCount er;

+-hn\_pCount er->weakCount; // 増加弱引用计数 return \*this;

}

WeakPtr ()

[

release ();

*}*

SharedPtr<T> lock()

[

return SharedPtr<T>(\*this);

*}*

bool expired ()

[

if (m\_pCounter != nullptr && m\_pCountershareCount != 0) return false;

return true;

}

private：

void release()

{

--m\_pCount er">weakCount;

if (0 == m\_pCourrt er->weakCount *&& 0 ==* m\_pCount er->shareCount) // 必 须都为0才能删除

{

delete m\_pCounter;

m\_pCounter = NULL;

private：

T\* m\_pResource; //可能会成为悬挂指针

Counter\* m\_pCounter;

35.请问C++11有哪些新特性?

c卄11最常用的新特性如下：

aut。关键字：编译器可以根据初始值自动推导出类型。但是不能用于函数传参以及数 组类型的推导

nullptr关键字：mllptr是一种特殊类型的字面值，它可以被转换成任意其它的指针 类型；而NULL-般被宏定义为0,在谒到重载时可能会出现问题。

智能指针：C++11新増了 std： ：shared\_p-rk std： ： weak\_ptr等类型的智能指针，用于 解决内存管理的问题。

初始化列表：使用初始化列表来对类进行初始化

右值引用：基于右值引用可以实现移动语义和完美转发，消除两个对象交互时不必要的 对象拷贝，节省运算存储资源，提高效率

atomic原子操作用于多线程资源互斥操作

新増STL容器array以及tuple

1. 智能指针是线程安全的吗？哪些地方需要考虑线程安

全？

1. 智能指针对象中引用计数是多个智能指针对象共享的，两个线程中智能指针的引用计 数同时卄或者一，这个操作不是原子的，引用计数原来是1,卄了两次，可能还是2, 这样引用计数就乱了，有可能造成资源粹放或者程序崩溃的风险。所以说智能指针中 I或一的操作是需要加锁的*，*也就是说引用计数的操作是线程安全的
2. 智能指针的对象存放在堆上，两个线程同时去访问，就会造成线程安全问题.

std：: shaxed\_ptr 循环引用 struct ListNode

{

int \_data;

shared\_ptr<ListNode/r shared\_ptr〈ListNode, listNodeO { cout « )； int main。

-P^ev;

\_next;

"TistNode：)" « endl;

shared\_ptr<ListNode> nodel(new ListNode); shared\_ptr<ListNode> node2(new ListNode); cout « no del. use\_\_count () « endl; cout « node2. use\_\_count () « endl;

nodel->\_next = node2;

node2-^\_prev - no del;

cout « no del. use\_\_count () « endl;

cout « node2. use\_\_count () « endl;

return 0;

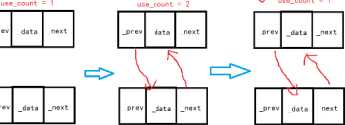
no del和no de2两个智能指针对象指向两个节点，引用计数变为1,我们不需要手动 delete nodel的\_next指向node2, node2的丄工邮指向no del,引用计数变成2

no del和no de2析构，引用计数减到1,但是\_mxt还指向下一个节点，\_prev指向上一 个节点

也就是说一next析构了，no de2释放了

也就是说\_prev析构了，no del释放了

但是一next属于node的成员,no del释放了，\_mxt才会析构，而rodel由\_prevW理, \_?工印属于1101^2成员，所以这就叫循环引用，谁都不会释放



n—1想餐桥皆.*戦安* 尢断中的富

如.

it■先桥构心3中第 Kiu.就这#峪入r死

解决方案

在引用计数的场景下，把sharedjtr换成weak\_ptr就可以了

原理就是，nodel->\_next = node2;和 node2->\_prev = no del;时 weak\_ptr 的 \_next 和口通卩不会増加nodel和node2的引用计数

struct ListNode

[

int \_data;

weak\_ptr<ListNode/r \_prev; weak\_ptr<ListNode^ \_next;

^ListNode () {

cout « "TistNodeO” « endl;

int main。

[

shared\_ptr<ListNode> nodel(new ListNode); shared\_ptr<ListNode> node2(new ListNode); cout « no del. use\_\_count () « endl; cout « node2. use\_\_count () « endl; nodel->\_next = node2;

node2~^\_prev - no del;

cout « no del. use\_\_count () « endl;

cout « node2. use\_\_count () « endl;

return 0;

}

如果不是new出来的空间如何用智能指针管理呢？

其实shared\_ptr设计了 一个删除器来解决这个问题

//仿函数的删除器

template'Cclass T>

struct FreeFunc {

void operator () (T\* ptr)

[

cout « "free：" « ptr « endl;

free (ptr);

*I*

}；

template'Cclass T>

struct DeleteAxrayFunc {

void operator () (T\* ptr)

[

cout « "delete]]" « ptr « endl;

delete[] ptr;

}

}；

int main()

[

FreeFunc<int*〉* freeFunc;

shared\_ptr<int> spl((int\*)malloc(4), freeFunc);

DeleteAxrayFunc<int> de1eteAxrayFunc;

shared\_ptr<int> sp2((int\*)malloc(4), deleteAxrayFunc); return 0;

}

1. 如何访问非法内存区？

1、在下列程序中，i和\*Pi都是未初始化的变量，它们的值都是不确定的。而Pi指向 的是未知位置，不属于程序所拥有的存儲单元，该指针变量称为野指针 #include<stdio. h> int main。

[

int i,\*pi；

\*pi=5；

printf (°%d\n°, i, \*pi);

return 0;

2、 使用已经释放过后的指针

堆空间用空闲链表法来组织，释放后的地址返回链表中，可能其他函数申请了该地址处 的空间。如果写了其他函数使用的空间，可能导致其他程序出错。malloc

3、 指针所指向的变量在指针之前被销毁

例如，指针指向了某个函数中的局部变量，当函数返回后，局部变量被销毁，如果栈空 间又被使用，再使用该指针可能就会出错。

1. 决策树是如何解决过拟合问题的？

产生过度拟合数据问题的原因

原因1：样本问题

（1） 样本里的噪音数据干扰过大，大到模型过分记住了噪音特征，反而忽略了真 实的输入输出间的关系5（什么是噪音数据？）

（2） 样本抽职错误，包括（但不限于）样本数量太少，抽样方法错误，抽样时没 有足够正确考虑业务场景或业务特点,等等导致抽出的样本数据不能有效足够代表业务 逻辑或业务场景；

（3） 建模时使用了样本中太多无关的输入变量。

原因2:构建决策树的方法问题

在决策树模型搭建中,我们使用的算法对于决策树的生长没有合理的限制和修萸的 话，决策树的自由生长有可能每片叶子里只包含单纯的事件数据或非事件数据，可以想 象，这种决策树当然可以完美匹配（拟合）训练数据，但是一旦应用到新的业务真实数 据时，效果是一塌糊'涂。

二.如何解决过度拟合数据问题

针对原因1的解决方法：

合理、有效地抽样，用相对能够反映业务逻辑的训练集去产生决策树；

针对原因2的解决方法（主荽）：

萸枝：提前停止树的増长或者对已经生成的树按照一定的规则进行后萸枝。 萸枝的方法

萸枝是一个简化过拟合决策树的过程。有两种常用的萸枝方法：

（1）先萸枝（prepruning）:通过提前停止树的构建而对树"萸枝”，一旦停止， 节点就成为树叶。该树叶可以持有子集元组中最频繁的类； 先萸枝的方法

有多种不同的方式可以让决策树停止生长，下面介绍几种停止决策树生长的方法: 限制决策树的高度和叶子结点处样本的数目

―1.定义一个高度，当决策树达到该高度时就可以停止决策树的生长，这是一种最为 简单的方法；

1. 达到某个结点的实例具有相同的特征向量，即使这些实例不属于同一类，也可以 停止决策树的生长。这种方法对于处理数据中的数据冲突问题非常有效：
2. 定义一个阚值,当达到某个结点的实例个数小于该阚值时就可以停止决策树的生 长；
3. 定义一个阚值，通过计算每次扩张对系统性能的増益，并比较増益值与该阚值的 大小来决定是否停止决策树的生长。

⑵后萸枝(postpruning):它首先构造完整的决策树，允许树过度拟合训练数据, 然后对那些置信度不够的结点子树用叶子结点来代替，该叶子的类标号用该结点子树中 最频繁的类标记。后萸枝的萸枝过程是删除一些子树，然后用其叶子节点代替，这个叶 子节点所标识的类别通过大多数原则(majority class criterion)确定。所谓大多数原 则，是指萸枝过程中，将一些子树删除而用叶节点代替，这个叶节点所标识的类别用这 棵子树中大多数训练样本所属的类别来标识，所标识的类称为majority class .相比于 先萸枝，这种方法更常用,正是因为在先萸枝方法中精确地估计何时停止树増长很困难。

后萸枝的方法

DEEP方法是一种比较简单的后萸枝的方法，在该方法中，可用的数据被分成两 个样例集合：一个训练集用来形成学习到的决策树，一个分离的验证集用来评估这个决 策树在后续数据上的精度，确切地说是用来评估修萸这个决策树的影响。这个方法的动 机是：即使学习器可能会被训练集中的随机错误和巧合规律所误导，但验证集合不大可 能表现出同样的随机'波动。所以验证集可以用来对过度拟合训练集中的虚假特征提供防 护检燈。

该萸枝方法考虑将书上的每个节点作为修萸的候选对象*，*决定是否修萸这个结点有 如下步骤组成：

1：删除以此结点为根的子树

2：使其成为叶子结点

3：赋予该结点关联的训练数据的最常见分类

4：当修萸后的树对于验证集合的性能不会比原来的树差时，才真正删除该结点

因为训练集合的过拟合，使得验证集合数据能够对其进行修正，反复进行上面的操 作，从底向上的处理结点，删除那些能够最大限度的提高验证集合的精度的结点，直到 进一步修萸有害为止(有害是指修萸会减低验证集合的精度)。

REP是最简单的后萸枝方法之一，不过由于使用独立的测试集，原始决策树相比， 修改后的决策树可能偏向于过度修萸。这至因为一些不会再测试集中出现的很稀少的训 练集实例所対应的分枝在荆枝辿如果训练集段小*，*通常不岑虑釆用KEP算法。

尽管REP有这个缺点，不过REP仍然作为一种基准来评价其它萸枝算法的性能。它 对于两阶段决策树学习方法的优点和缺点提供了了一个很好的学习思路。由于验证集合 没有参与决策树的创建，所以用REP萸枝后的决策树对于测试样例的偏差要好很多，能 够解决一定程度的过拟合问题。

2)PEP,悲观错误萸枝，悲观错误萸枝法是根据萸枝前后的错误率来判定子树的修 萸。该方法引入了统计学上连续修正的概念弥补REP中的缺陷，在评价子树的训练错误 公式中添加了一个常数*，*假定每个叶子结点都白动对实例的某个部分进行错误的分类。 它不需要像REP (错误率降低修萸)样*，*需要用部分样本作为测试数据*，*而是完全使用训 练数据来生成决策树，又用这些训练数据来完成萸枝。决策树生成和萸枝都使用训练集, 所以会产生错分。

把一棵子树（具有多个叶子节点）的分类用一个叶子节点来替代的话，在训练集上 的误判率肯定是上升的，但是在测试数据上不一定，我们需要把子树的误判计算加上一 个经验性的惩罚因子，用于估计它在测试数据上的误判率。对于一棵叶子节点，它覆盖 了 N个样本，其中有E个错误，那么该叶子节点的错误率为（E+0.5）/No这个0.5就 是惩罚因子，那么对于该棵子树，假设它有L个叶子节点，则该子树的误判率估计为：

(£&+0.5\*乙)/少,

萸枝后该子树内部节点变成了叶子节点，该叶子结点的误判个数J同样也需要 加上一个惩罚因子，变成Jb. 5。那么子树是否可以被萸枝就取决于萸枝后的错误5 在

工 *Ei* + 0.5 \* L

的标准误差内。对于样本的误差率e,我们可以根据经验把它估计成伯努利分布，那么 可以估计出该子树的误判次数均值和标准差

*E^subtree.err.count)* = N \* e

*var^subtree-err.count) = \/N* \* e \* (1 — e)

使用训练数据，子树总是比替换为一个叶节点后产生的误差小，但是使用校正的 误差计算方法却并非如此。萸枝的条件:当子树的误判个数大过对应叶节点的误判个数 —个标准差之后，就决定萸枝：

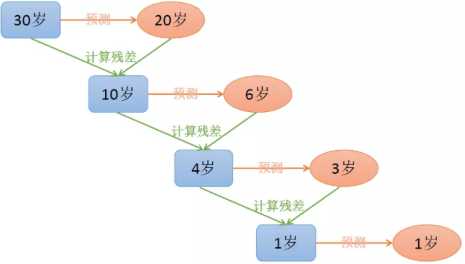
*E^subtree.err.count) — var(subtree-err.count) > E(leaf.err.count)*

这个条件就是萸枝的标准。当然并不一定非要大一个标准差，可以给定任意的 萤信区间，我们设定一定的显著性因子，就可以估算出误判次数的上下界。

1. 什么是GBDT算法？

GBDT （Gradient Boosting Decision Tree）梯度提升迭代决策树。GBDT 也是 Boosting算法的一种，但是和.WaBoost算法不同（AdaBoost算法上一篇文章已经介 绍）；区别如下：AdaBoost算法是利用前一轮的弱学习器的误差来更新样本权重值， 然后一轮一轮的迭代；GBDT也是迭代，但是GBDT要求弱学习器必须是CART模型， 而且GBDT在模型训练的时候，是要求模型预测的样本损失尽可能的小。

每一轮预测和实际值有残差，下一轮根据残差再进行预测，最后将所有预测相加，就是 结果。



GBDT模型可以表示为决策树的加法模型：

*M*

其中，T（x; 8m）表示决策树；8m为决策树的参数；M为树的个数。

采用前向分布算法，首先确定初始提升树fo （x） =0,第m步的模型是:

A（X）"0） + 7（"）

通过经验风险极小化确定下一棵树的参数：（其实就是让残差尽可能的小找到最优划分 点）

如=arg 曾+ 不;QJ）

-/=i

这里的L（）是损失函数，回归算法选择的损失函数一般是均方差（最小二乘）或者绝对 值误差;而在分类算法中一般的损失函数选择对数函数来表示.

GBDT既可以做回归也可以做分类，下面先描述一下做回归的算法流程：

已知一个训练数据集T = ｛（xL yl）, （x2沱），...，（xn, yn）｝,如果将训练集分为不同的 区域Rl,R2,...,m然后可以确定每个区域输出的常识c,c的计算是将每个区域的y 值相加再除以y的个数，其实就是求一个平均值。树可以表示为：

*J*华;。）=£勺［（\*郊）  
*j=i*

然后通过下图方式来确定具体分割点：

■ — min min Y （3 -c） +min £ *（yt-c2）2*

**, L q** 砂曷 *‘ Xier2 \_*

以上就是GBDT选择分割点的过程，如果特征有多个的话也是一样的道理，选择特征 和特征值使得误差最小的点，作为分割点。所以其实GBDT也可以用作特征选择，通过 GBDT可以将重要的特征选择出来，当特征非常多的时候可以用来做降维。然后再融合 类似逻辑回归这样的模型再进行训练。

40.写一个C++回调函数示例.

#include <stdio・h>

//函数指针

type def void(\*lpFunc) (void \*, char \*, int);

//调用回调函数的宿主函数，参数callback是原型名称为IpFunc的函数指针 void GetCallBack(void \*lpVoid, IpFunc callback, char\* name, int age){

//执行回调函数callback,其实是调用通过形参callback实际传过来的

//函数 fCallback

callback (ipVoid, name, age);

}

class A{

public：

AO {};

void outName(char szAlarm[], int age){

printf(^My name is %s, %d years old \n^, szAlarm, age);

}

/庞又一个类A的静态成员函数fCallback

static void fCallback(void \*lpVoid, char szAlarmt], int age){

//类A的成员函数中，使用类A定义一个对象指针p指向传进来的指针

//参数IpVoid,强制类型转换为：A\*

A \*p = (A\*) (ipVoid);

//A类型的对象指针P调用A类的成员寒素outName p~>outName(szAlarm, age);

//A类的成员函数Test

void Test () {

/座类A的成员函数Test中调用外部函数GetCallBack,将类A的静态 //成员函数fCallback名称传给第二个参数，实现fCallback函数的回调 GetCallBack (this, fCallback, "kevin^, 38);

int main(void)

[

A a;

a. TestO;

*}*

1. Linux日志文件统计某几个字符串，如何一条命令就能统 计岀来？

1-使用vim统计

用vim打开目标文件，在命令模式下，输入:%s/objStr//gn

1. 使用grep

grep -o 'objStrl\|objStr， filenane|wc ~1 #直接用\ | 链接起来即可

1. Linux大文件如何处理，如何分割？

使用split对文件进行切割，切割有两种方式

1. 根据行数切割，通过-1参数指定需要切割的行数

示例：指定文件名为split-line, -d参数以数字的方式显示

split -1 300 -d —verbose 文件名 split-line

1. 根据大小切割，通过九参数指定需要切割的大小

示例：指定-b参数指定文件大小进行切割，文件大小单位支持K, M, G, T, P, E, Z.

split ~b 30K ~d —verbose 文件名 split-size

43.你了解linux常用命令有哪些？

1、 Is命令

2、 cd切换

3、 pwd查看当前工作目录路径

4、 mkdir创建文件夹

5、 rm删除文件

7、 mv移动/修改文件名

8、 cp复制

9、 cat显示文件详情

14、which查看可执行文件的位置

16、 locate 命令

17、 find文件树中查找文件

⑤、grep文本搜索命令

18、 chmod访问权限

19、 tar压缩和解压

出、chown改为指定的用户或组

21、 df显示磁盘空间

22、 du查看使用空间

23、 In命令

24、 date显示时间

25*、* cal命令

27*、*wc命令 匆、ps查看进程

29、 top正执行的进程

30、 kill杀死进程

31、 free显示内存使用情况

32、 reboot开关机命令

33、 if conf ig 查看 ip 地址

34、 用户相关

|  |  |
| --- | --- |
| useradd oldboy | 整添加用户 |
| passwd redhat | 。设弟密码 |
| whoami | 女当前用户 |
| su - oldboy | /切换用户 |
| logout | 住退出用户登录 |
| 35、权限相关  r | Sread可读，可以用cat等命令查看 |
| w | Swrite写入，可以编辑或者删除这个文件 |
|  | JJ executable 可以执行 |

36、特殊字符重定向相关

» #追加重定向，把文字追加到文件的结尾

> #重定向符号，清空原文件所芍内容，然后把文字覆盖到文件末尾

< #输入重定向

« #将输入结果输入重定向

37、 iptables, firewall 防火墙

38、 vi和vim编辑文本

39、 PATH常见环境变量变量

40、 I管道命令

41、 alias起别名命令

44.解释下数据库ACID什么意思

ACID特性即数据库管理系统中事务(transaction)的四个特性：原子性(Atomicity)、 —致性(Consistency)、隔离性(Isolation)、持久性(Durability)

所谓事务，它是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分 割的工作单位。(执行单个逻辑功能的一组指令或操作称为事务)

1. 原子性

原子性是指事务是一个不可再分割的工作单元，事务中的操作要么都发生，要么都不发 生。

可采用“A向B转账”这个例子来说明解释

在DBMS中，默认情况下一条SQL就是一个单独事务，事务是自动提交的。只有显式的 使用start transaction开启一个事务，才能将一个代码块放在事务中执行。

1. 一致性

—致性是指在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。这是说 数据库事务不能破坏关系数据的完整性以及业务逻辑上的一致性。

如A给B转账，不论转账的事务操作是否成功，其两者的存款总额不变(这是业务逻辑 的一致性，至于数据库关系约束的完整性就更好理解了)。

保障机制(也从两方面着手)：数据摩层面会在一个事务执行之前和之后，数据会符合 你设置的约束(唯一约束，外键约束,check约束等)和触发器设置；此外，数据库的内 部数据结构(如B树索引或双向链表)部必须是正确的。业务的一致性一般由开发人 员进行保证，亦可砖移至数据库居面。

1. 隔离性

多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果。 在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空 间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务查看数据更 新时，数据所处的状态要么是另一事务修改它之前的状态，要么是另一事务修改它之后 的状态，事务不会查看到中间状态的数据。 事务最复杂问题都是由事务隔离性引起的。完全的隔离性是不现实的，完全的隔离性要 求数据摩同一时间只执行一条事务，这样会严重影响性能。

关于隔离性中的事务隔离等级（事务之间影响），参见相应博文

1. 持久性

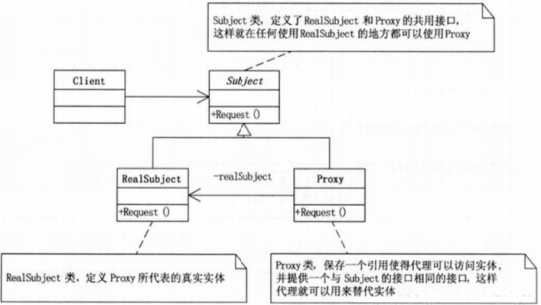
这是最好理解的一个特性：持久性，意味看在事务完成以后，该事务所对数据库所作的 更改便持久的保存在数据库之中，并不会被回滚。（完成的事务是系统永久的部分，对 系统的影响是永久性的，该修改即使出现致命的系统故障也将一直保持）

write ahead logging : SQL Server 中使用了 WAL （Write-Ahead Logging ）技术来保证 事务日志的ACID特性，在数据写入到数据库之前，先写入到日志，再将日志记录变更 到存储器中。

1. 说说代理设计模式，并画一下代理模式结构图.

所谓代理模式是指客户端并不直接调用实际的对象，而是通过调用代理，来间接的调用 实际的对象。

代理模式**（Proxy）**結构图



1. 什么是开闭原则.

幵闭原则（0CP）是面向对象设计中“可复用设计〃的基石，是面向对象设计中最重要 的原则之一，其它很多的设计原则都是实现开闭原则的一种手段。

幵闭原则中“开”，是指对于组件功能的扩展是开放的，是允许对其进行功能扩展的; 幵闭原则中“闭”，是指对于原有代码的修改是封闭的，即不应该修改原有的代码。

1. mysql联合索引的原理？

联合索引是指对表上的多个列合起来做一个索引。联合索引的创建方法与单个索引的创 建方法一样，不同之处仅在于有多个索引列，如下

create table t（

a int,

b int,

primary key（a）,

key idx\_a\_b （a, b）

）；

从本质上来说，联合索引就是一棵B+树，不同的是联合索引的键值得数量不是1,而

是＞=2。两个整型列组成的联合索引，假定两个键值得名称分别为a、b如图

I (2,4) £ I

(1.1) (1,2) (2,1) (2,4) (3.1) (3,2)

可以看到这与我们之前看到的单个键的B+树并没有什么不同，键值都是排序的，通过 叶子结点可以逻辑上顺序地读出所有数据，就上面的例子来说，即（L1）,（L2）,

（2,1） ,（2,4） ,（3,1） ,（3,2）；数据按（a,b）的顺序进行了存放。

因此，对于查询 select \* from table where a=xxx and b=xxx,显然是可以使用（a, b） 这个联合索引的，对于单个列a的查询select \* from table where a=xxx,也是可以 使用（a,b）这个索引的。

但对于b列的查询select \* from table where b=xxx,则不可以使用（a,b）索引， 其实你不难发现原因，叶子节点上b的值为1、2、1、4、1、2显然不是排序的，因此 对于b列的查询使用不到（a, b）索引.

联合索引的第二个好处是在第一个键相同的情况下，已经对第二个键进行了排序处理, 例如在很多情况下应用程序都需要查询某个用户的购物情况，并按照时间进行排序，最 后取出最近三次的购买记录，这时使用联合索引可以帮我们避免多一次的排序操作，因 为索引木身在叶子节点已经排序了 .

1. 数据库查询慢的原因？
2. 偶尔效率慢的情况

原因1：刷新“脏”页

1）什么是“脏”页

―当对数据库进行插入或者更新操作旺*，*数据库会立刻将刻存的数据页上的信息更 新，但是不会立刻将将更新的数据有到磁盘上，而是先保存到red。log中，等合 适的时机在将red。log的信息存储到磁盘上

针对这种内存中的数据页和磁盘上的数据不同的情况我们将内存中的数据页称为 “脏”页，而内存中和磁盘上数据相同的情况则称为“干净”页。

刷新“脏〃页时，系统会暂停其他的操作，全身心的将数据存到磁盘中，就会导致 平常正常执行的mysql语句变慢

2）什么时候会刷新“脏”页

redo log装满时，内存不够用时,mysql认为系统空闲时,mysql正常关闭时.会刷 新“脏"页

原因2:数据被锁住

可以用show processlist命令查看一下语句执行的状态，查看要查询的数据是否 被锁住

1. —直都存在效率慢的情况

原因1：查询的数据量太大

查看是否查询了不必要的行与列，谚免用select \* from table这样的语句 原因2:没有用到索引

当数据量很大时，若没有用索引采用全表索引是很耗费时间的。而这里没有用到索 引由可以分多钟情况

1） 没有建索引，

2） 索引失效

引起索引失效的可能原因

⑴在索引列上用了内置函数或善其他A\*/运算

⑵用通配符开头

⑶多列索引违背最佳最匹配原则

⑷皿操作符容器造成索引失效，除非皿的每个操作列都有索引

⑸字符串不加单引号

3） 系统选错索引

系统选错索引其实是索引失效的一种形式，但是由于涉及到的知识点较多，所以单 独莹出来分析。

系统选错索引导致索引失效时系统将全表扫描与用索引要扫描的行数进行比较，若 是觉得运用索引反而要复杂，则系统就会放弃索引采用全表扫描的方式。

1. 说说你了解的排序算法，如何优化一个算法，让其时间复 杂度降到nlogn.

快速排序

快速排序的基本思想是基于分治策略的，基本思想如下：

分解:先从序列中取出一个元素作为基准，以基准元素为标准将序列分解为两个子序列。 其中小于或等于基准的子序列在左侧，大于基准的子序列在右侧。

治理：对拆分之后的子序列进行快速排序

合并：将排序好的子序列合并在一起，从而得到整个序列的排序。

这像是我们常说的“大事化小，小事化了”，大的困难分解成一个个小的问题，逐个击 破。

分治法后面也会讲解

常见的基准元素选取方式有：选择第一个元素，最后一个元素，中位数等等。 代码

#include<iostream>

using namespace std;

/依取基准元素所在的位置

int GetMid(int arr [], int low, int high)

[

int i = low, j = high, pivot = arr [low];

while (i < j)/佼换后继续扫描

[

while (i<j&&axr[j]>pivot) j—；//右侧开始查找比基准元素更小的值 while (i < j&&axr[i] <= pivot) i++；//左侧查找比基准元素大于等于的值 if (i < j)

[

swap (azr[i++], azr[j-]：; //先交换后赋值

}

}

if (arr[i] > pivot)

[

swaptazrti - 1], azz [low]);/^±代码中提供的示例中，在谒到第一次扫描 寸交换i和j都指向37。如果不加最后的判断会出现错误。

return i ~ 1;

}

swap(arr [i], arr [low]);

return i;

}

//快速排序函数

void Quicksort (int axr [], int low, mt high)

[

int mid;

if (low < high)

[

/施归快排

mid = GetMid(axr, low, high；;

Quicksort(arr, mid + 1, high);〃右区间快排

Quicksort (axr, low, mid - 1：; /佐区间快排

int main()

[

int N = 9;

int axr [9] = { 30, 24, 5, 58, 37, 36,12, 42, 39 };

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

[

cout « arr[i] « *”宀*

*\*

cout<<endl;

Quicksort(axr, 0, NT);

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

[

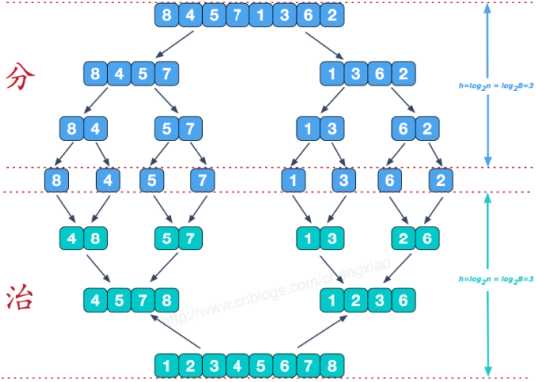
cout « azr[i] «

}

cout << endl;

合并排序

合并排序采用的就是分治策略，讲一个大的问题分成很多的小问题。先解决小问题，然 后通过小问题解决大问题。



通过将每一个分解的子序列排序,然后不所的逢归子序列合并。达到总序列的有序排列。 合久必分，分久必合就是合并排序的策略。

代码

#include<iostream>

using namespace std;

//最小单元序列的合并

void Merge (int arr [], int low, int mid, int high)

[

//l.申请与azr等长的数组B

int \*B = new int [high - low + 1：;

int i = low, j = mid + 1, k = 0.

//2.将azz数组一分为二，按照升序的规则将元素依次放到B数组中

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

[

if (azr[i] <= azr[j])

[

B [k++] = axr [i++];

}

else

[

B [k++] = axr [j++];

}

}

//3.将arz数组的升序未排序的部分复制到B,左侧和右侧

while (i <= mid) { B [k++] = axr [i++]; }

while (j <= high) { B [k++] = axr j++]; }

//4.将B数组复制到azr数组，释欣B

for (i = low, k = 0; i <= high; i++)

{

azr [i] = B [k++];

I

delete []b;

}

"合并排序

void MergeSort(int arr f], int low, int high)

//递归合并排序 if (low < high) int mid = (low + high) / 2; MergeSort (axr, low, mid); MergeSort (axr, mid + 1, high); Merge (axr, low, mid, high);

int main()

[

int N = 8;

int a [8] = { 42, 15, 23,6,8, 38, 50,12 };

KergeSort (a, 0, N~l);

for (size\* i = 0; i < N; i++)

[

cout « a[i] «

时间复杂度

其中快速排序的平均时间复杂度为。（nl。/）。合并排序的二叉树高度为log：,每一层 都是n个元素进行比较，所以总的时间复杂度为0 （nl of）。

50.如何划分子网？

—个IP地址一共有32位（二进制）,其中靠前的某些位表示网络号，后面的某些位表 示主机号，网络位数+ 主机位数=IP地址位数=32,简单来说，子网掩码就是网络号的位 数，不会理解的，我可以举个例子：192.168.0.0/24,这一看我们就知道小型公司常用 的网段，可用 IP 地址：192. 168.0.1-192.168.0. 254,子网掩码：255.255.255.0,斜 杠后面的24指的是网络号，那么显然可用的主机号就变成8位，那么可用的主机数就 是2的8次方-2=254。

计算子网掩码的方法就是：已知子网内IP数的多少，求出主机位的位数，用32减去主 机位数就等于网络位数，也就是子网掩码。举最简单的例子。一个C类网络，包括西6 个主机位置，256是2的8次方，所以主机位是8,那么网络位就是308=24,也就是 说子网掩码是24位，用二进制表示就是11111111.11111111.11111111.00000000,换 算成十进制就是255.255.255.0。再比如一个C类网络划分的子网，每个网络主机IP 数是32,而32是2的5次方，所以主机位是5,那么网络位就是32-5=27,也就是说 子网掩码是27位，用二进制表示就是11111111.11111111.11111111.11100000,换算 成十进制就是255. 255. 255. 224。再比如一个B类网络划分的子网，每个网络主机IP 数是1024,而1024是2的10次方，所以主机位是10,那么网络位就是32-10=22 ,也 就是说子网掩码是22位，用二进制表示就是11111111.11111111.11111100.00000000, 换算成十进制就是255. 255. 252. 0。

**2020**年更多、更全、更新大厂面试资料请力曄：762073882

202旌阿里曲 202衅百爵选 202帰嚇曲  
面涙题.pdf 题.pdf 面涙题.pdf

Q面试分享.mp4

H| TCPI呦议栈，一次课开启你的网络之门.mp4 碍&面

Q高校能服务器为什么房要内存池.mp4

Q手踌W线密&.mp4

Q reactorigtHQ线®现高并发艮务mp4

Q ngi n>®§—线密&的实现.m p4

Q MySQL的块應提作.mp4

Q高并发tcpip网塔io.mp4

D 去中心化，p2p，网络争SFiB定.mp4

Q 服家能職化\_糸璀遊.mp4

Q 区块链的底昙，去中心化网络的设计.mp4

Q 深入注出UDP传箍原理及踌分片方法.mp4

Q线程月陛事.mp4

B后台服务噂挂了怎么办.mp4

零声学「謐理出品

2020年更多、更全、更新大厂面试资料Q群：762073882