**2021**年头条精选**50**面试题及答案

1. C++智能指针如何解决内存泄露问题.
2. shazedjtr共享的智能指针

std：：shmed\_ph使用引用计数，每一个sharedjtr的拷贝都指向相同的内存。在最 后一个shared\_ptr析构的时候，内存才会被释放。

可以通过构造函数、std\_make\_shared辅助函数和reset方法来初始化shared\_ptr:

//构造函数初始化

std::shared\_ptrp ( new int(1));

std::shared\_ptrp2 = p ;

//对于一个未初始化的智能指针，可以通过reset方法来初始化。

std::shared\_ptrptr; ptr.reset ( new int (1));

if (ptr) {cout « "ph is not null.\n^ ; }

不能将一个原始指针直接赋值给一个智能指针：

std： ： shared\_ptrp = new int(l) ;//编译报错，不允许直接赋值

获取原始指针：

通过get方法来返回原始指针

std：:shared\_ptrptr ( new int(1));

int \* p =ptr. get ();

指针删除器：

智能指针初始化可以指定删除器

void DeletelntPtr ( int \* p ) {

delete p ;

}

std::shared\_ptrp ( new int , DeletelntPtr );

当P的引用技术为o时，自动调用删除器来释放对象的内存。删除器也可以是一个 lambda表达式，例如

std：:shared\_ptrp ( new int , [](int \* p){delete p});

注意事项：

1. .不要用一个原始指针初始化多个shared\_ptro
2. .不要再函数实参中创建Shared\_ptr,在调用函数之前先定义以及初始化它。

(。).不荽将lhi»指针作为 ^Cd\_plx返回出来。

0).要避免循环引用。

1. unique\_ptr独占的智能指针

unique\_ptr是一个独占的智能指针，他不允许其他的智能指针共享其内部的指针，不 允许通过赋值将一个unique\_ptr赋值给另外一个unique\_ptro

unique\_ptr不允许复制，但可以通过函数返回给其他的unique\_ptr,还可以通过 std：:move来转移到其他的unique\_ptr,这样它本身就不再拥有原来指针的所有权了*。* 如果希望只有一个智能指针管理资源或管理数组就用unique\_ptr,如果希望多个智能 指针管理同一个资源就用shared\_ptro

1. weak\_ptr弱引用的智能指针

弱引用的智能指针weak\_ptr是用来监视sharedjtr的，不会使引用计数加一，它 不管理shared\_ptr内部的指针，主要是对了监视shared\_ptr的生命周期，更像是 shared\_ptr的一个助手。

weak\_ptr没有重载运算符桥口因为它不共享指针，不能操作资源，主要是为了 通过Shared\_ptr获得资源的监测权，它的构造不会増加引用计数，它的析构不会减少 引用计数，纯粹只是作为一个旁观者来监视shmed\_ph中关离得资源是否存在。

weak\_ptr还可以用来返回this指针和解决循环引用的问题。

1. 常见web安全问题，SQL注入、XSS、CSRF,基本原理以 及如何防御
2. SQL注入

原理：

D.SQL命令可查询、插入、更新、删除等，命令的串接。而以分号字元为不同命 令的区别。(原本的作用是用于Subauery或作为查询、插入、更新、删除……等 的条件式)

1. • SQL命令对于传入的字符串参数是用单引号字元所包起来。(但连续2个单引 号字元，在SQL资料库中，则视为字串中的一个单引号字元)
2. . SQL命令中，可以注入注解

预防：

1. .在设计应用程序时，完全使用参数化查询(ParameterizedQuery)来设计数据 访问功能。
2. .在组合SQL字符串时，先针对所传入的参数作字元取代(将单引号字元取代为 连续个单引号字元)*。*
3. .如果使用PHP开发网页程序的话，亦可打幵PHP的腐术引号(Magic quote)功 能(自动将所有的网页传入参数，将单引号字元职代为连续2个单引号字元)*。*
4. .其他，使用其他更安全的方式连接SQL数据库。例如已修正过SQL注入问题的 数据库
5. .连接组件，例如 ASP.NET 的 SqlDataSource 对象或是 LINQ to SQL。 使用SQL防注入糸统。
6. XSS攻击

原理：

xss攻击可以分成两种类型：

1. 非持久型X”攻击

非持久型X”攻击是一次性的，仅对当次的页面访问产生影响。非持久型X”攻击 要求用户访问一个被攻击者篡改后的链接，用户访问该链接时，被植入的攻击脚本 被用户游览器执行，从而达到攻击目的。

1. 持久型XSS攻击

持久型XSS攻击会把攻击者的数据存储在服务器端，攻击行为将伴随着攻击数据 —直存在。下面来看一个利用持久型XSS攻击获取session id的实例。

防范：

1 .基于特征的防御

XSS漏洞和著名的SQL注入漏洞一样，都是利用了 Meb页面的编写不完善， 所以每一个漏洞所利用和针对的弱点都不尽相同。这就给XSS漏洞防御带来了困 难：不可能以单一特征来概括所有XSS攻击。

传统XSS防御多采用特征匹配万式，在所有提交的信息中都进行匹配检查。 对于这种类型的XSS攻击，采用的模式匹配方法一般会需要对“javascript”*这* 个关键字进行检索，一旦发现提交信息中包含“javascript” ,就认定为XSS攻 击。这种检测方法的缺陷显而易见：骇客可以通过插入字符或完全编码的方式躲避 检测：

1. .在javascript中加入多个tab键，得到

<IMG SRC="jav ascript:alert('XSS');">;

1. .在javascript中加入(空格)字符，得到

<IMG SRC="javascri pt:alert('XSS

1. .在javascript中加入(回车)字符，得到

<IMG SRC="javasc

ript:alert(,XSS,);,,>;

1. .在javascript中的每个字符间加入回车换行符，得到

<IMG SRC="javascrip\r\nt:alert('XSS');">

1. .对w javascript:alert( 'XSS')”采用完全编码，得到

<IMGSRC=javascrip?74:alert(1XSS')>

上述方法都可以很容易的躲避基于特征的检测。而除了会有大量的漏报外，基 于特征的

还存在大量的误报可能：在上面的例子中，对上述某网站这样一个地址，由于 包含了关键字“javascript”，也埒会触发报警。

1. 基于代码修改的防御

和SQL注入防御一样，XSS攻击也是利用了 Neb页面的编写疏忽，所以还有 一种方法就是从Web应用开发的角度来避免：

对所有用户提交内容进行可靠的输入验证，包括对URL、查询关键字、HTTP 头、POST数据等，仅接受指定长度范围内、采用适当格式、采用所预期的字符的 内容提交，对其他的「律过滤。

实现Session标记(session tokens) ^CAPTCHA系统或者HTTP引用头检查， 以防功能被第三方网站所执行。

确认接收的的内容被妥善的规范化，仅包含最小的、安全的Tag(没有 javascript),去掉任何对远程内容的引用(尤其是样式表和javascript),使用 HTTP only 的 cookie。

1. CSRF攻击

原理：

CSRF攻击原理比较简单，假设Web A为存在CSRF漏洞的网站，Web B为攻 击者构建的恶意网站，User C *h* Web A网站的合法用户。

1. .用户C打开浏览器，访问受信任网站A,输入用户名和密码请求登录网站A;
2. .在用户信息通过给证后，网站A产生Cookie信息并返回给浏览器，此时用 户登录网站A成功，可以正常发送请求到网站A;
3. 用户未退出网站A之前，在同一浏览器中，打开一个TAB页访问网站B;

4 .网站B接收到用户请求后，返回一些攻击性代码，并发出一个请求要求访 问第三方站点A;

5.浏览器在接收到这些攻击性代码后，根据网站B的请求，在用户不知情的 情况下携带Cookie信息，向网站A发出请求。网站A并不知道该请求其实是 由B发起的，所以会根据用户C的Cookie信息以C的权限处理该请求，导致 来自网站B的恶意代码被执行。

防范：

1. 检查Referer字段

HTTP头中有一个Referer字段，这个字段用以标明请求来源于哪个地址。在 处理敏感数据请求时，通常来说，Referer字段应和请求的地址位于同一域 名下。以上文银行操作为例，Referer字段地址通常应该是转账按钮所在的 网页地址，应该也位于www.examplebank.com之下。而如果是CSRF攻击传 来的请求，Referer字段会是包含恶意网址的地址，不会位于 [WWW.examplebank](http://WWW.examplebank). com之下,这时候服务器就能识别出恶意的访问。

2 .添加校验token

由于CSRF的本质在于攻击者欺骗用户去访问自己设置的地址，所以如果要求 在访问敏感数据请求时，要求用户浏览器提供不保存在cookie中，并且攻击 者无法伪造的数据作为校验，那么攻击者就无法再执行CSRF攻击。这种数据 通常是表单中的一个数据项。服务器将其生成并附加在表单中，其内容是一个 伪乱数。当客户端通过表単提交请求时,这个伪乱数也一并提交上去以供校燈。 正常的访问时，客户端浏览器能够正确得到并传回这个伪乱数，而通过CSRF 传来的欺骗性攻击中，攻击者无从事先得知这个伪乱数的值，服务器端就会因 为校验token的值为空或者错误，拒绝这个可疑请求。

1. linux中软连接和硬链接的区别.

原理上，硬链接和源文件的inode节点号相同，两者互为硬链接。软连接和源文件的 inode节点号不同，进而指向的block也不同，软连接block中存放了源文件的路径名*。* 实际上，硬链接和源文件是同一份文件，而软连接是独立的文件，类似于快捷方式，存 備看源文件的位置信息便于指冋。

使用限制上，不能对目录创建硬链接，不能对不同文件系统创建硬链接，不能对不存在 的文件创建硬链接；可以对目录创建软连接，可以跨文件系统创建软连接，可以 对不存在的文件创建软连接。

1. STL内存分配方式

在STL中考虑到小型区块所可能造成的内存碎片问题，SGI STL设计了双层级配置器， 第一级配置器直接使用malloc（）和free：）；第二级配置器则视情况采用不同的策略:当 配置区块超过128byteS时，则视之为足够大，便调用第一级配置器；当配置区块小于 128byteS时，则视之为过小，为了降低额外负担，便采用复杂的内存池的方式来整理, 而不再求助于第一级配置器。

每次配置器需要向系统要内存的时候，者所是按客户需求向系统申请的，而是一次性向 系统要了比需求更多的内存，放在内存池里，有一个free.start和free.end指示剩 余的空间（也就是说内存池剩余的空间都是连续的，因此每次重新向system heap要空 间的时候，都会把原先内存池里没用完的空间分配给合适的free list。）当heeTist 中没有可用区块了的时候，会首先从内存池里要内存，同样，也不是以按客户需求要多 少块的，而是一次可能会要上2。块，如果内存池内空间允许的话，可能会得到2。个 特定大小的内存，如果内存池给不了那么多，那么就只好尽力给出；如果连一个都给不 出，那么就要开始向系统即system heap要空间了。换算的标准是

bytes\_to\_get= 2\*total\_bytes+ROUND\_UP:heap\_size>>4）。这个时候使用的是 malloc, 如果没成功，就尝试着从大块一点的freelist那里要一个来还给内存池，如果还是不 彳亍，那么会调用第一级空间配置器的malloc： ： allocate,看看out-of-memory机制能 做点什么。

假设我们向系统要x大小的内存，

（1） x大于128byte,用第一级配置器直接向系统malloc,至于不成功的处理，过程 仿照new,通过set\_new\_handler来处理，直到成功返回相应大小的内存或者是抛出异 常或者是干脆结束运行；

（2） x小于128byte,用第二级配置器向内存池相应的free\_liSt要内存，如果该 freelist 面没有空闲块了，

（2.1） 从内存池里面要内存，缺省是获得20个节点，如果内存池中剩余的空间不能完 全满足需求量，但足够供应一个（含）以上的区块，则应尽力满足需求。

（2.2） 如果一个都不能够滴足的话，则从系统的heap里面要内存给到内存池，换算 的标准是 bytes\_to\_get= 2\*total\_bytes-ROUND\_UP （heap\_size»4）,申请的大小为需求 量的两倍加上一个祐加值，如果鬲存池中还有莉余的内荐，则将残余零头分配给适当 的free list,这时使用的是系统的malloc,如果要不到：

（2. 3）从比较大的freelist那里要内有到内存池，如果还是要不到：

（2.4）从系统的heap里面要内存给到内存池，换算标准跟2. 2 -样，但是这时候使 用的是第一级配置器的allocate,主要是看看能不能通过out\_of.memory机制得到一 点内存。所以，freelist总是从内存池里要内存的，而内存池可能从freelist也可能 从系统heap那里要内存。

SGI SIL的alloc的主要开销就在于管理这些小内存，管理这些小内存的主要开销就在 于，每次freelist ±的内存块用完了，需要重新要空间，然后建立起这个来。 freelist!的内存，会一直保留着直到廷序退出才还给系统。但这不会产生内存泄漏, —来是管理的都是小内存，二来是，占用的内存只会是整个程序运行过程中小内存占 用量最大的那一刻所占用的内存。

1. TCP的拥塞控制机制是什么？请简单说说

我们知道“P通过一个定时器（timer）采样了 RTT并计算皿，但是，如果网络上的 延时突然増加，那么，对这个事做出豹应对只有重传数据，然而重传会导致网络的 负担更重，于是会导致更大的延迟以及更多的丢包，这就导致了恶性循环，最终形成“网 络风暴” 一一 TCP的拥塞控制机制就是用于应对这种情况。

首先需要了解一个概念，为了在发送端调节所要发送的数据量，定义了一个“拥塞窗口” （Congestion Window）,在发送数据时，将拥塞窗口的大小与接收端ack的窗口大小 做比较，取较小者作为发送数据量的上限。

拥塞控制主要是四个算法：

1. 慢启动：意思是刚刚加入网络的连接，一点一点地提速，不要一上来就把路占满。 连接建好的幵始先初始化cwnd = 1,表明可以传一个MSS大小的数据。

每当收到一个ACK, cwnd卄；呈线性上升

每当过了一个RTT, cwnd = cwnd\*2;呈指数让升

阈值 ssthresh （ slow start threshold）,是一个上限，当 cwnd >= ssthresh 时，就 会进入“拥塞避免算法〃

1. 拥塞避免：当拥塞窗口 cwnd达到一个阚值时，窗口大小不再呈指数上升，而是以线 性上升，避免増长过快导致网络拥塞。

每当收到一个 ACK, cwnd = cwnd + 1/cwnd

每当过了—个 RTF, cwnd = cwnd + 1

拥塞发生：当发生丢包进行数据包重传旺，表示网络已经拥塞。分两种情况进行处理： 等到对超时，重传数据包

sshthresh = cwnd /2

cwnd重置为1

1. 进入慢启动过程

在收到3个duplicate ACK时就开启重传，而不用等到RTO超时

sshthresh = cwnd = cwnd /2

进入快速'恢复算法一一Fast Recovery

1. 快速恢复：至少收到了 3个Duplicated Acks,说明网络也不那么糟糕，可以快速恢 复。

cwnd = sshthresh + 3 \* MSS （ 3的意思是确认有3个数据包被收到了）

重传Duplicated ACKs指定的数据包

如果再收到 duplicated Acks,那么 cwnd = cwnd +1

如果收到了新的Ack,那么，cwnd = sshthresh ,然后就进入了拥塞避免的算法了。

1. Mysql连接语句如何实现快速查找？
2. like语句（和数据库做模糊匹配*）''蠕*可用于定义通配符

SELECT \* FKOM Persons WHERE City LIKE ' 1

1. 顶语句（表示某个字段是否在集合中）

SELECT \* FKOM Persons WHERE Las顼ame IN （' Adams'Carter'）

1. BETWEEN ••• AND会选取介于两个值之间的数据范围。这些值可以是数值、文本或者 日期。

SELECT \* FKOM Persons WHERE Las顼ame BETWEEN ' Adams' AND ’Carter'

1. Alias （给返回的字段或者查询的表设置别名）

SELECT po. Order ID, p. LastName, p. FirstName FKOM Persons AS p, Product\_Orders AS po WHERE p. LastNajne=‘Adams' AND p. FirstNajne=，JohnJ

1. join（匹配两个表数据）

SELECT Persons. LastName, Persons. FirstName, Orders. OrderNo FKOM Persons

INNER JOIN Orders ON Persons.Zd\_P = Orders. Id\_P

常用几种连接语句的区别

JOIN:如果表中有至少一个匹配，则返回行

LEFT JOIN:即使右表中没有匹配，也从左表返回所有的行

RIGHT JOIN:即使左表中没有匹配，也从右表返回所有的行

FULL JOIN:只要其中一个表中存在匹配，就返回行

UNION （注意跟join是不同的）（UNION環作符用于合并两个或多个SELECT语句的结 果集。）

（默认地，UNION操作符选职不同的值。如果允许重复的值，请使用UNION ALL。） 注意：请注意，UNION内部的SELECT语句必须拥有相同数量的列。列也必须拥有相似 的数据类型。同时，每条SELECT语句中的列的顺序必须相同。

SELECT column\_najne （s） FKOM table\_namel

UNION

SELECT column\_najne （s） FKOM table\_name2

SELECT INTO （语句从一个表中选取数据，然后把数据插入另一个表中。）

SELECT \* INTO Persons\_backup FROM Persons

SELECT \* INTO Persons IN 5 Backup, mdb' FKOM Persons

1. 输入一个矩阵，按照从外向里以顺时针的顺序依次打印岀 每一个数字，例如，如果输入如下4X4矩阵：1234567

8 9 10 11 12 13 14 15 16则依次打印岀数字

1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.

public static ArrayList<Integer> printMatrix（int [][] matrix） {

if （matrix, length == 0）{

return null;

ArrayList<Integer> list = new AirayListO （）;

//四个边界值

int top = 0;

int left = 0;

int right = matrix[0]. length;

int bottom = matrix, length;

//横着和竖着相加的值(1/-1)

int colFlage = 1;

int rowFlage = 1;

//现在是否是横着移动

boolean flag = true;

int i = 0;

int j = 0;

while (bottom \= top && right J= left) {

//横向移动

if (flag) {

for (; j < right && j >= left; j += rowFlage) { list, add (matrix [i] [j]);

}

//缩小墙壁

if (rowFlage > 0) {

top++;

} else {

bottom"-;

}

//指针恢复

i += rowFlage;

//切换下次横向移动方向

rowFlage = -rowFlage;

j += rowFlage;

广

//纵向移动

if (Jflag){

for (; i < bottom && i >= top; i += colFlage) { list, add (matrix [i] [j]);

}

//缩小墙壁

if (colFlage > 0) {

right—;

} else {

left++;

}

//切换下次纵向移动方向

colFlage = "colFlage;

//恢复指针位置

i += colFlage;

j += colFlage;

}

//切换横向和竖向移动 flag = Jflag;

}

return list;

}

1. 利用快速排序对单链表进行排序

#include^iostream>

#include< c t ime*〉*

using namespace std;

struct ListNode {

int val;

ListNode \*next;

ListNode(int x) : val(x), next (DULL) {}

};

class Solution {

public：

ListNode \*sortList(ListNode \*head) {

if (head == NULL)

return head;

ListNode\* tail=head;

ListNode\* end = tail;

while (tail->next != NULL)[

//if (tail->next~>next == NULL)

// end = tail;

tail = tail~>next;

}

qSortList (head, tail);

return head;

}

ListNode\* Partition(ListNode\* head, ListNode\* tail) [

ListNode\* newHead = new Lis顼ode(0); newHead->next = head;

ListNode\* ipt = newHead, \*jpt = head;

int ar = tail->val;

while (jpt != tail)

[

if (jpt~>val < ar)

[

ipt = ipt\_>next;

swap (ipt->val, jpt-/val);

}

jpt = jpt\_>next;

}

ipt = ipt\_>next;

swap(ipt->val, tail->val);

return ipt;

}

void qSortList(ListNode\*head, ListNode\*tail)

[

if (head == NULL)

return;

if (tail == NULL)

return;

if (tail->next J =NULL && tai»next == head)

return;

else if (tail->nex1*1*=NULL &&

tail->nex>next J =NULL &&

tail->nex >nex t== head)

return;

if (head == tail)

return;

if (head~>next == tail)

{

if (head~>val > tail~>val)

swap(head->vaL tail->val);

return;

}

ListNode\* mid = Partition (head, tail);

ListNode\*taill = head;

if (taillJ=mid)

while (taill->next != mid) taill = taill->next;

ListNode\*head2 = mid~>next;

qSortList (head, taill);

qSortList(head2, tail);

int main。

[

ListNode\* headO = new ListNode(200);

ListNode\* head = headO;

srand(time(NULL));

for (int i = 0; i < 10000; i++)

[

ListNode\* mNode = new ListNode(rand() % 4); headO->next = mNode;

headO = headO~>next;

I

Solution sin;

ListNode\*res=sln. sortList(head). while (res->next \= NULL)

[

cout « res->val « " ”；

res = res->next;

}

return 0;

1. Mysql如何实现快速查找？

々include〈iostrearn〉

*9* include<ctime>

using namespace std;

struct ListNode {

int val;

ListNode \*next;

ListNode(int x) : val(x), next (DULL) {} };

class Solution {

public：

ListNode \*sortList(ListNode \*head) {

if (head == NULL)

return head;

ListNode\* tail=head;

ListNode\* end = tail; while (tail->next != NULL)[

//if (tail->next->next == NULL)

// end = tail; tail = tail~>next;

I

qSortList (head, tail);

return head;

}

ListNode\* Partition(ListNode\* head, ListNode\* tail)

[

ListNode\* newHead = new Lis顼ode(0); newHead->next = head;

ListNode\* ipt = newHead, \*jpt = head;

int ar =

while (jpt != tail)

[

if (jpt~>val < ar)

[

ipt = ipL>next;

swap ;

}

jpt = jpt\_>next;

}

ipt = ipt\_>next;

swap (ipt->val, tail->val);

return ipt;

}

void qSortList(ListNode\*head, ListNode\*tail)

{

if (head == NULL)

return;

if (tail == NULL)

return;

if (tail->next! =NULL && tai»next == head) return;

else if (tail->nextJ=NULL &&tai1->nex*t~*>next!=NULL

&& tail->nex *t~*>nex-== head)

return;

if (head == tail)

return;

if (head->next == tail)

[

if (head~>val > tail~>val) swap(head->vaL tail->val);

return;

}

ListNode\* mid = Partition (head, tail);

ListNode\*taill = head;

if (taillJ=mid)

while (taill->next \= mid) taill = taill->next;

ListNode\*head2 = mid->next;

qSortList (head, taill);

qSortList(head2, tail);

}

}；

int main()

[

ListNode\* headO = new ListNode(200);

ListNode\* head = headO;

srand(time(NULL));

for (int i = 0; i < 10000; i++)

[

ListNode\* mNode = new ListNode(rand() % 4); head0->next = mNode;

headO = headO~>next;

}

Solution sin;

ListNode\*res=sln. sortList(head).

while (res->next != NULL)

{

cout « res->val « " ”;

res = res-^next;

return 0;

1. 将一个链表分为奇偶两个链表

#include<iostreajn>

# include <mal1o c. h>

using namespace std;

typedef struct node{

int data;

node \*next;

}LNode;

LNode \*L;

LNode\* create (LNode \*L, int axr [], mt n) {

L = (LNode\*)malloc(sizeof(LNode：);

L~>next = NULL;

LNode \*p = L;

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

LNode \*newNode = (LNode\*)malloc(sizeof(LNode)); newNode~>next = NULL;

newNode-^data = axr [i];

//头插法

p->next = newNode;

p - p->next;

return L;



//分离的方法

void spilt(LNode \*L, LNode \*B){

LNode \*p = L;

B = (LNode\*)malloc(sizeof(LNode：);

B~>next = NULL;

LNode \*q = B;

LNode \*r;

while(p->next != NULL){

if(p->next->data % 2 == 0){

”T面处理的是取下链表中的偶数节点插入到链表中

//偶数的时候取出来插入到链表B中

r - p->next;

p->next - p->next->next.

//q指针向下移动

q->next - r;

r->next = NULL;

q - q->next;

}else{

p = p\_>next;

}

I

"输出链表A的内容

LNode \*s = L;

while(s->next != NULL){

printf (”%d ”, s->next->data；;

s = s\_>next;

}

cout << endl;

"输出链表B的内容

s = B;

while(s->next != NULL){

printf (”％d ”, s->next->data；;

s = s->next;

int main(void){

int axr[12] = {12, 43, 2, 10, 100, 89, 11, 56, 45, 34, 77, 11};

LNode \*head = create (L, arr, 12；;

LNode \*B;

spilt (head, B);

return 0;

}

1. 有序数组二分查找，返回查找元素最后一次岀现的位置, 若不存在则返回-1

#include^iostream>

using namespace std;

void last (int [] a, int key) {

int min=0, max=a. length"l;

int mid=0;

while(min<=max){

mid=(max+min+1)/2;

if (key>=a[mid]) {

min=mid;

lelse {

max=mid"l;

if (max==min) {

break;

}

}

if (a[max]J=key) {//当查找元素不存在时'返回-1 return ~1;

lelse {

return max;

int main(void){

int[] a= {3, 5,10,10,10,13,13,19, 23};

last (a, 10);

return 0;

}

1. 在一个二维01矩阵中找到全为1的最大正方形，返回其 面积。

样例

10 10 0

10 111

11111

10 0 10

返回4

public int max Square (int [] [] matrix； {

// write your code here

int row = matrix, length; /#丁大小

int line = matrix [0]. length. //列大小

//一个与matrix相同大小的辅助数组

int [] [] tmp = new int [row] [line];

W matrix的第一行和第一列元素直接存放到

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

tmp [i] [0] = matrix [i] [0：;

for (int i=0;Kline;i++) {

tmp [0] [i] = matrix [0] [i：;

for(int i=l;i<row;i++){ for(int j=l;j<line;j++)[ if(matrix[i] [j] == 1){ tmp[i] [j]= Math, min (Math, mm ( tmp[i-l] [j], tmp[i] [j-1]：, tmp[i-l] [j-1]) + 1;

if(matrix[i] [j] == 0){  
tmp[i] [j] = 0;

int max=0; //记录tmp中最大元素的值(tmp中元素值表示正方形的边长) for(int i=0;i<row;i++){

for(int j=0;j<line;j++)[

if (tmp[i] [j] > max)[

max = tmp[i] [j].

return max\*max;

13.常见的HTTP状态码

1XX系列响应代码仅在与HTTP服务器沟通时使用。

100("Continue")

这是对HTTP LBYL (1 o ok-b ef or e-you-1 e sp )请求的一个可能的响应。该响应代码表明: 客户端应重新发送初始请求，并在请求中附上第一次请求时未提供的(可能很大或者包含敏 感信息的)表示。客户端这次发送的请求不会被拒绝。对LBYL请求的另一个可能的响应是 417("Expectation Failed")。

请求报头：要做一个LBYL请求，客户端必须把Expect请求报头设为字符串 〃100-continue气除此以外，客户端还需要设置其他一些报头，服务器将根据这些报头决定 是响应100还是417。

101("Switching Protocols")

当客户端通过在请求里使用Upgrade报头，以通知服务器它想改用除HTTP协议之外的 其他协议时，客户端将获得此响应代码。101响应代码表示“行，我现在改用另一个协议了”。 通常HTTP客户端会在收到服务器发来的101响应后关闭与服务器的TCP连接。101响应代 码意味着，该客户端不再是一个HTTP客户端，而将成为另一种客户端。

尽管可以通过Upgrade报头从HTTP切换到HTIPS,或者从HTIP1.1切换到某个未来的 版本，但实际使用Upgrade报头的情况比较少。Upgrade报头也可用于HTTP切换到一个完 全不同的协议(如IRC)上，但那需要在Web服务器切换为一个IRC服务器的同时，Web客 户端切换为一个IRC的客户端，因为服务器将立刻在同一个TCP连接上幵始使用新的协议。

请求报头:客户端把Upgrade报头设置为一组希望使用的协议。

响应报头：如果服务器同意切换协议，它就返回一个Upgrade报头，说明它将切换到那 个协议，并附上一个空白行。服务器不用关闭"P链接，而是直接在该"P连接上幵始使用 新的协议。

2XX:成功

2XX系列响应代码表明操作成功了。

200("0K")

—般来说，这是客户端希望看到的响应代码。它表示服务器成功执行了客户端所请求的 动作，并且在2XX系列里没有其他更适合的响应代码了。

实体主体:对于GET请求，服务器应返回客户端所请求资源的一个表示*。*对于其他请求, 服务器应返回当前所选资源的一个表示，或者刚刚执行的动作的一个描述。

-201 ("Created")

当服务器依照客户端的请求创建了一个新资源时，发送此响应代码。

响应报头：Location报头应包含指向新创建资源的规范URI。

实体主体：应该给出新创建资源的描述与链接。若已经在Location报头里给出了新资 源的URI,那么可以用新资源的一个表示作为实体主体。

-202("Accepted")

客户端的请求无法或将不被实时处理。请求稍后会被处理。请求看上去是合法的，但在 实际处理它时有出现问题的可能。

若一个请求触发了一个异步操作，或者T•需要现实世界参与的动作，或者一个需要很 长时间才能完成且没必要让Web客户端一直等待的动作时，这个相应代码是一个合适的选 择。

响应报头：应该把未处理完的请求暴露为一个资源，以便客户端稍后查询其状态。 Location报头可以包含指向该资源的URI。

实体主体：若无法让客户端稍后查询请求的状态，那么至少应该提供一个关于何时能处 理该请求的估计。

203(^Non"Authoritative Information^；

这个响应代码跟200 —样，只不过服务器想让客户端知道，有些响应报头并非来自该服 务器一他们可能是从客户端先前发送的一个请求里复制的，或者从第三方得到的。

响应报头：客户端应明白某些报头可能是不准确的，某些响应报头可能不是服务器自己 生成的，所以服务器也不知道其含义。

204("No Content")

若服务器拒绝对PUT、POST或者DELETE请求返回任何状态信息或表示，那么通常采用 此响应代码。服务器也可以对GET请求返回此响应代码，这表明“客户端请求的资源存在, 但其表示是空的”*。*注意与304 ("Not Modified")的区别。204常常用在Ajax应用里。服务 器通过这个响应代码告诉客户端：客户端的输入已被接受，但客户端不应该改变任何UI元 素。

实体主体：不允许。

205("Reset Content")

它与204类似，但与204不同的是，它表明客户端应重置数据源的视图或数据结构。假 如你在浏览器里提交一个表单，并得到响应代码204,那么表单里的各个字段值不变, 可以继续修改它们；但假如得到的响应代码2D5,那么表单里的各个字段将被重置为它们的 初始值。从数据录入方面讲：204适合对单条记录做一系列编辑，而205适于连续输入一组 记录。

206("Partial Content")

它跟200类似，但它用于对部分GET请求(即使用Range请求报头的GET请求)的响应。 部分GET请求常用于大型二进制文件的断点续传。

请求报头:客户端为Range请求报头设芝一个值。

响应报头：需要提供Date报头。ETag扌艮头与Content-Location报头的值应该跟正常 GET请求相同。

若实体主体是单个字节范围(byte range),那么HTTP响应里必须包含一个 Content-Range报头，以说明本响应返回的是表示的哪个部分，若实体主体是一个多部分实 体(multipart entity)(即该实体主体由多个字节范围构成)，那么每一个部分都要有自 己的 Content-Range 报头。

实体主体：不是整个表示，而是一个或者多个字节范围。

3XX重定向

3XX系列响应代码表明：客户端需要做些额外工作才能得到所需要的资源。它们通常用 于GET请求。他们通常告诉客户端需要向另一个URI发送GET请求，才能得到所需的表示*。* 那个URI就包含在Location响应报头里。

300("Multiple Choices")

若被请求的资源在服务器端存在多个表示，而服务器不知道客户端想要的是哪一个表示 时，发送这个响应代码。或者当客户端没有便用Accept-対艮头来指定一个表示，或者客户 端所请求的表示不存在时，也发送这个响应代码。在这种情况下，一种选择是，服务器返回 一个首选表示，并把响应代码设置为200,不过它也可以返回一个包含该资源各个表示的URI 列表，并把响应代码设为300 o

响应报头：如果服务器有首选表示，那么它可以在Location响应报头中给出这个首选 表示的URI。跟其他3XX响应代码一样，客户端可以自动跟随Location中的URI。

实体主体：一个包含该资源各个表示的UKI的列表。可以在表示中提供一些信息，以便 用户作出选择。

301("Moved Permanently")

服务器知道客户端试图访问的是哪个资源，但它不喜欢客户端用当前URI来请求该资 源。它希望客户端记住另一个听1,并在今后的请求中使用那个新的听1。你可以通过这个 响应代码来防止由于URI变更而导致老URI失效。

响应报头：服务器应当把规范URI放在Location响应报头里。

实体主体：服务器可以发送一个包含新反1的信息，不过这不是必需的。

302(“Found")

重要程度：应该了解，特别市编写客户端时。但我不推荐使用它。

这仓响应代码市造成大多数重定向方面的混乱的最根本原因。它应该是像307那样被处 理。实际上，在HTTP 1.0中，响应代码302的名称是” Moved Temporarily",不幸的是， 在实际生活中，绝大多数客户端篁它像303 —样处理。它的不同之处在于当服务器为客户端 的PUT, POST或者DELETE请求返回302响应代码时，客户端要怎么做。

为了消除这一混淆，在HTIP1.1中，该响应代码被重命名为"Found",并新加了一个响 应代码307。这个响应代码目前仍在广泛使用，但它的含义市混淆的，所以我建议你的服务 发送307或者303,而不要发送302.除非你知道正在与一个不能理解303或307的HTIP1.0 客户端交互。

响应报头:把客户端应重新请求的那个URI放在Location报头里。

实体主体：一个包含指向新听1的链接的超文本文档(就像301—样)。

303("See Other°)

请求已经被处理，但服务器不是直接返回一个响应文档，而是返回一个响应文档的URK 该响应文档可能是一个静态的状态信息，也可能是一个更有趣的资源。对于后一种情况，303 是一种令服务器可以“发送一个资源的表示，而不强迫客户端下载其所有数据"的方式。容 户端可以向Location报头里的URI发送GET请求，但它不是必须这么做。

303响应代码是一种规范化资源URI的好办法。一个资源可以有多个URIs,但每个资源 的规范URI只有一个，该资源的所有其他URIs都通过303指向该资源的规范URI,例如： 303 可以把一个对 http： //ww. example, com/software/current, tar. gz 的请求重定向到 http：//www. example.com/software/1.0.2.tar. gzo

响应报头：Location报头里包含资源的URI。

实体主体：一个包含指向新听1的链接的超文本文档。

304("Not Modified")

这个响应代码跟204 (〃N。Content")类似：响应实体主体都必须为空。但204用于没有 主体数据的情况，而304用于有主体数据，但客户端已拥有该数据，没必要重复发送的情况。 这个响应代码可用于条件HTTP请求(conditional HTTP request).如果客户端在发送GET 请求时附上了一个值为Sunday的If-Modified-Since报头，而客户端所请求的表示在服务 器端自星期日(Sunday)以来一直没有改变过，那么服务器可以返回一个304响应。服务器 也可以返回一个200响应，但由于客户端已拥有该表示，因此重复发送该表示只会白白浪费 宽带。

响应报头：需要提供Date报头。Etag与Content-Location报头的值，应该跟返回200

响应时的一样。若Expires, Cache-Control Vary报头的值自上次发送以来已经改变，那 么就要提供这些报头。

实体主体：不允许。

305("Use Proxy")

这个响应代码用于告诉客户端它需要再发一次请求，但这次要通过一个HTTP代理发送, 而不是直接发送给服务器。这个响应代码使用的不多，因为服务器很少在意客户端是否使用 某一特定代理。这个代码主要用于基于代理的镜像站点。现在，镜像站点(如 [http://ww.exajnple.com.mysite.com/)包含跟原始站点(如](http://ww.exajnple.com.mysite.com/)%e5%8c%85%e5%90%ab%e8%b7%9f%e5%8e%9f%e5%a7%8b%e7%ab%99%e7%82%b9(%e5%a6%82) <http://ww.exajnple.com/>) —样的内容，但具有不同的听1,原始站点可以通过307把客户端重新定向到镜像站点上。 假如有基于代理的镜像站点，那么你可以通E把[http://proxy.mysite.com/ig为代理，使](http://proxy.mysite.com/ig%e4%b8%ba%e4%bb%a3%e7%90%86%ef%bc%8c%e4%bd%bf) 用跟原始URI (http：//ww. example, com/)—样的URI来访问镜像站点。这里，原始站点 example, com可以通过305把客户端路由到f 地理上接近客户端的镜像代理。web浏览器 一般不能正确处理这个响应代码，这是导致335响应代码用的不多的另一个原因。

响应报头：Location报头里包含代理的URI。

306未使用

重要程度：无

306响应代码没有在HTTP标准中定义过。

307("Temporary Redirect")

请求还没有被处理，因为所请求的资源不在本地：它在另一个URI处。客户端应该向那 个听1重新发送请求。就GET请求来说，它只是请求得到一个表示，该响应代码跟303没有 区别。当服务器希望把客户端重新定向到一个镜像站点时，可以用307来响应GET请求。但 对于POST, PUT及DELETE请求，它们希望服务器执行一些操作，307和303有显著区别。 对POST, PUT或者DELETE请求响应303表明：操作已经成功执行，但响应实体将不随本响 应一起返回，若客户端想要获取响应实体主体，它需要向另一个听1发送GET请求。而307 表明：服务器尚未执行操作，客户端需要向Location报头里的那个URI重新提交整个请求。

响应报头:把客户端应重新请求的那个呱1放在Location报头里。

实体主体：一个包含指向新听1的链接的超文本文档。

4XX:客户端错误

这些响应代码表明客户端出现错误。不是认证信息有问题，就是表示格式或HTTP库本 身有问题。客户端需要自行改正。

400("Bad Request")

这是一个通用的客户端错误状态，当其他4XX响应代码不适用时，就采用400。此响应 代码通常用于“服务器收到客户端通过PUT或者POST请求提交的表示，表示的格式正确， 但服务器不懂它什么意思”的情况。

实体主体：可以包含一个错误的描述文档。

401("Unauthorized")

客户端试图对一个受保护的资源进行操作，却又没有提供正确的认证证书。客户端提供 了错误的证书，或者根本没有提供证书。这里的证书(credential)可以是一个用户名唇 码，也可以市一个API key,或者一个认证令牌。客户端常常通过向一个URI发送请求，并 查看收到401响应，以获知应该发送哪种证王，以及证书的格式。如果服务器不想让未授权 的用户获知某个资源的存在，那么它可以谎报一个404而不是401。这样做的缺点是：客户 端需要事先知道服务器接受哪种认证一这将导致HTTP摘要认证无法工作。

响应报头：WW-Authenticate报头描述服务器将接受哪种认证。

实体主体：一个错误的描述文档。假如疑务用户可通过“在网站上注册”的方式得到证 书，那么应提供一个指向该注册页面的链接。

402("Payment Required")

除了它的名字外，HTTP标准没有对该响应的其他方面作任何定义。因为目前还没有用 于HTTP的微支付系统，所以它被留作将来使用。尽管如此，若存在一个用于HTTP的微支付 系统，那么这些系统将首先出现在web服务领域。如果想按请求向用户收费，而且你与用户 之间的关系允许这么做的话，那么或许用得上这个响应代码。

注：该书印于2008年

403("Forbidden")

客户端请求的结构正确，但是服务器不想处理它。这跟证书不正确的情况不同一若证书 不正确，应该发送响应代码401。该响应代码常用于一个资源只允许在特定时间段内访问，

或者允许特定IP地址的用户访问的情况。403暗示了所请求的资源确实存在。跟401 —样，若服务器不想透露此信息，它可以谎报一个404。既然客户端请求的结构正确，那为 什么还要把本响应代码放在4XX系列(客户端错误)，而不是5XX系列(服务端错误)呢？ 因为服务器不是根据请求的结构，而是根据请求的其他方面(比如说发出请求的时间)作出 的决定的。

实体主体：一个描述拒绝原因的文档(耳选)。

404("Not Found")

这也许是最广为人知的HTTP响应代码了。404表明服务器无法把客户端请求的URI转 换为一个资源。相比之下，410更有用一些。web服务可以通过404响应告诉客户端所请求 的URI是空的，然后客户端就可以通过向该URI发送PUT请求来创建一个新资源了。但是 404也有可能是用来掩饰403或者401.

405("Method Not Allowd")

客户端试图使用一个本资源不支持的HTTP方法。例如：一个资源只支持GET方法，但 是客户端使用PUT方法访问。

响应报头：Allow报头列出本资源支持哪些HTTP方法，例如：Allow: GET, POST

406("Not Acceptable")

当客户端对表示有太多要求，以至于服务器无法提供满足要求的表示，服务器可以发送 这个响应代码。例如：客户端通过Accept头指定媒体类型为application/j son+hic,但是 服务器只支持application/jsono服务器的另一个选择是：忽略客户端挑剔的要求，返回 首选表示，并把响应代码设为200。

实体主体：一个可选表示的链接列表。

407(“Proxy Authentication Required”；

只有HTTP代理会发送这个响应代码。它跟401类似，唯一区别在于：这里不是无权访 问web服务，而是无权访问代理。跟401—样，可能是因为客户端没有提供证书，也可能是 客户端提供的证书不正确或不充分。

请求报头：客户端通过使用Prox厂Authorization报头(而不是Authorization)把证 书提供给代理。格式跟Authrization —样。

响应报头：代理通过Prox厂Authentic ate报头(而不是WW-Authentic ate)告诉客户 端它接受哪种认证。格式跟WW-Authenticate —样。

408("Reqeust Timeout")

假如HTTP客户端与服务器建立链接后，却不发送任何请求(或从不发送表明请求结束 的空白行)，那么服务器最终应该发送一个4。8响应代码，并关闭此连接。

409 ("Conflict")

此响应代码表明：你请求的操作会导致服务器的资源处于一种不可能或不一致的状态。 例如你试图修改某个用户的用户名，而修改后的用户名与其他存在的用户名冲突了。

响应报头:若冲突是因为某个其他资源的存在而引起的，那么应该在Location报头里 给出那个资源的URI。

实体主体：一个描述冲突的文档，以便容户端可以解决冲突。

410("Gone")

这个响应代码跟404类似，但它提供的有用信息更多一些。这个响应代码用于服务器知 道被请求的听1过去曾指向一个资源，但该资源现在不存在了的情况。服务器不知道

该资源的新听1,服务器要是知道该URI的话，它就发送响应代码301.410和310 一样, 都有暗示客户端不应该再请求该听1的意思，不同之处在于：410只是指出该资源不存在， 但没有给出该资源的新听1。RFC2616建议“对短期的推广服务，以及属于个人但不继续在 服务端运行的资源〃采用410.

411("Length Required")

若HTTP请求包含表示，它应该把Content-Length请求报头的值设为该表示的长度(以 字节为单位)。对客户端而言，有时这不太方但(例如，当表示是来自其他来源的字节流时)。

所以HTTP并不要求客户端在每个请求中部提供Content-Length报头。但HTTP服务器 可以要求客户端必须设置该报头。服务器可以中断任何没有提供Content-Length报头的请 求，并要求客户端重新提交包含Content-Length报头的请求。这个响应代码就是用于中断 未提供Content-Lenght报头的请求的。假如客户端提供错误的长度，或发送超过长度的表 示，服务器可以中断请求并关闭链接，并返回响应代码413。

412(^Precondition Failed")

客户端在请求报头里指定一些前提条件,并要求服务器只有在满足一定条件的情况下才 能处理本请求。若服务器不满足这些条件，就返回此响应代码。If-Unmodified-Since是一 个常见的前提条件。客户端可以通过PUT请求来修改一个资源，但它要求，仅在自客户端最 后一次获取该资源后该资源未被别人修改过才能执行修改操作。若没有这一前提条件，客户 端可能会无意识地覆盖别人做的修改，或者导致409的产生。

请求报头：若客户但设置了 If "Mat ch, If-None-Match 或 If-Urunodified-Since 报头， 那就有可能得到这个响应代码o If-None-Match稍微特别一些。若客户端在发送GET或HEAD 请求时指定了 If-None-Match,并且服务器不满足该前提条件的话，那么响应代码不是412 而是304,这是实现条件HTTP GET的基础。若客户端在发送PUT, POST或DELETE请求时指 定了 If-None-Match,并且服务器不满足该前捏条件的话，那么响应代码是412.另外，若容 户端指定了 If-Match或If-Unmodified-Since (无论采用什么HTTP方法),而服务器不滴足 该前提条件的话，响应代码也是412。

413("Request Entity Too Large")

这个响应代码跟411类似，服务器可以用它来中断客户端的请求并关闭连接，而不需要 等待请求完成。411用于客户端未指定长度的情况，而413用于客户端发送的表示太大，以 至于服务器无法处理。客户端可以先做一个LBYL (10ok-before-you-leap)请求，以免请 求被413中断。若LBYL请求获得响应代码为100,客户端再提交完整的表示。

响应报头：如果因为服务器方面临时谒到问题(比如资源不足)，而不是因为客户端方 面的问题而导致中断请求的话,服务器可以把Retry-After报头的值设为一个日期或一个间 隔时间，以秒为单位，以便客户端可以过段旺间重试。

414 ("RequesLURI Too Long")

HTTP标准并没有对URI长度作出官方限刽，但大部分现有的web服务器都对URI长度 有一个上限，而web服务可能也一样。导致UF.I超长的最常见的原因是：表示数据明明是该 放在实体主体里的，但客户端却把它放在了 U^I里。深度嵌套的数据结构也有可能引起URI 过长。

415(^Unsupported Media Type")

当客户端在发送表示时采用了一种服务畧无法理解的媒体类型,服务器发送此响应代 码。比如说，服务器期望的是XML格式，而容户端发送的确实JSON格式。

如果客户端采用的媒体类型正确，但格式有问题，这时最好返回更通用的400。

416("Requestd Range Not Satisfiable\*)

当客户端所请求的字节范围超出表示的实际大小时，服务器发送此响应代码。例如：你 请求一个表示的1-100字节，但该表示总共只用99字节大小。

请求报头：仅当原始请求里包含Range报头时，才有可能收到此响应代码。若原始请求 提供的是H-Range报头，则不会收到此响应代码。

响应报头：服务器应当通过Content-Range报头告诉客户端表示的实际大小。

417("Expectation Failed")

此响应代码跟100正好相反。当你用LBYL请求来考察服务器是否会接受你的表示时， 如果服务器确认会接受你的表示，那么你将获得响应代码100,否则你将获得417。

5XX服务端错误

这些响应代码表明服务器端出现错误。一般来说，这些代码意味着服务器处于不能执行 客户端请求的状态，此时客户端应稍后重试。有时，服务器能够估计客户端应在多久之后重 试。并把该信息放在Reh厂After响应报头里。

5XX系列响应代码在数量上不如4XX系列度，这不是因为服务器错误的几率小，而是因 为没有必要如此详细--对于服务器方面的问题*，*客户端是无能为力的。

500(“Internal Server Error

这是一个通用的服务器错误响应。对于大多数web框架，如果在执行请求处理代码时谒 到了异常，它们就发送此响应代码。

501("Not Implemented")

客户端试图使用一个服务器不支持的HTTP特性。

最常见的例子是：客户端试图做一个采用了拓展HTTP方法的请求，而普通web服务器 不支持此请求。它跟响应代码405比较相似，405表明客户端所用的方法是一个可识别的方 法，但该资源不支持，而501表明服务器根本不能识别该方法。

502("Bad Gateway")

只有HTTP代理会发送这个响应代码。它表明代理方面出现问题，或者代理与上行服务 器之间出现问题，而不是上行服务器本身有叵題。若代理根本无法访问上行服务器，响应代 码将是504。

503("Service Unavailable")

此响应代码表明HTTP服务器正常，只是F层web服务服务不能正常工作。最可能的原 因是资源不足：服务器突然收到太多请求，以至于无法全部处理。由于此问题多半由客户端 反复发送请求造成，因此HTTP服务器可以选择拒绝接受客户端请求而不是接受它，并发送 503响应代码。

响应报头：服务器可以通过Retry-After报头告知客户端何时可以重试。

504(“Gateway Timeout^)

跟502类似，只有HTTP代理会发送此响应代码。此响应代码表明代理无法连接上行服 务器。

505("HTTP Version Not Supported")

1. 25匹马，5个赛道，最少赛多少次找岀前三没有计时， 只有先后.

7次。理由如下：

1. 先分开赛5组(A-E), 5次，每组的最后两名肯定会被淘汰，(-10)o
2. 5组第一名赛一次,假设A1>B1>C1> D1>E1,那么Al肯定是总体第一名。则D, E全部被淘汰(-6).现在需要在剩下的里面职2个，那么C2,C3,B3也会被淘汰(-3)。
3. 那么就剩ITA2, A3, Bl, B2, C1 (,冉募一次,取前两名(-3)。

最多7次比呑，前5次总共淘汰10匹，箕6次淘汰9匹，第7次淘汰3匹。总共淘汰 22匹。

1. 计算机为什么能识别二进制机器码？

1、计算机的理论基础

布尔代数是计算机的理论基础，

Boolean （布尔运算）通过对两个以上的物体进行并集、差集、交集的运算，从而得到 新的物体形态。系统提供了 4种布尔运算方式：Union （并集）'Intersection （交集）和 Subtraction （差集，包括和B-A两种）.

1） 与逻辑和乘法

乘法原理中自变量是因变量成立的必要芸件，与逻辑的定义正好和乘法原理的描述一 致，所以与逻辑和乘法对应。

2） 或逻辑和加法

加法原理中自变量是因变量成立的充分芸件，或逻辑的定义正好和加法原理的描述一 致，所以或逻辑和加法对应。

乘法就是广义的与逻辑运算，加法就是广义的或逻辑运算。与逻辑运算可以看作是乘法 的特例。或逻辑运算可以看作是加法的特例。

总之，乘法原理、加法原理可以看作是与逻辑和或逻辑的定量表述；与逻辑和或逻辑可 以看作是乘法原理、加法原理的定性表述。

通俗来讲：这是一门运用”与“”或“” “非〃 “假〃 “真”来描述任意两个量（可以 是任何具体事物的或者抽象概念）的逻辑关系。

2、 逻辑代数与计算机电路

应用于逻辑中，解释。为假，1为真，△为与，V为或，的非。涉及变量和布尔运 算的表达式代表了陈述形式，两个这样的表达式可以使用上面的公理证实为等价的，当且仅 当对应的陈述形式是逻辑等价的。由于逻辑彳燉小的逻辑单元与二进制高度契合，再加上电 路最为简单的开和关恰好也对应。和1,于是就有了依据逻辑代数理论创建一系列的电路在 表达基础的逻辑理论，这就是计算机具有判断、计算能力的基础。

3、 二进制机器识别过程

根据前面两点可以知道，如果选用二进制原理作为计算机的判断计算依据，将会使得电 路制造的实现成为可能，但是自然界是不存在二进制的，为了处理这个问题，统一人为规定 将其他非二进制数据表示成二进制机器码，供计算机读取。然而。随着对数据的处理能力要 求越来越高，处理数据也越来越大，为了解决这个问题，汇编器出现，替代了将非二进制数 据转化为二进制数据，但是这远远不足，为了更好处理，直接将硬件与汇编器组合，单独发 展更高级汇编器（实质就是现在熟知的各类程序），这样，硬件与软件彻底分开。实质上就 是将数据转化与判断和数据的录入、存储、输出彻底分幵，使计算机的使用者可以完全不必 再关注计算机的具体运算。

也就是说，计算机为什么能够识别二进制机器码，是因为有以逻辑代数原理制造的数字 电路，为什么选用二进制已经解释过了。同旺，也应该明白，为何程序会出现假，1为真， A为与，V为或，呐非这五个元素，算法对何而来，指的就是优化数据之间的逻辑代数 关系。

1G.如何理解mac寻址？

曲C地址是网卡的物理地址，每块网卡都有一个属于自己的独有的MAC地址，如

00-0A-EB-97-5F-65。

IP地址就是我们常见的202.106.46.151 192.168.1.1这样的。

通过MC地址寻找主机是MC地址寻址，通过IP地址寻找主机叫IP地址寻址。它们适 用于不同的协议层。

1. 如何理解IO多路复用的三种机制Select, Poll, Epoll?
2. Select

首先先分析一下select函数

int select (

int maxfdpi,

fd\_set \*readset,

fd\_set \*wxiteset,

fd\_set \*exceptset,

const struct timeval \*timeout

);

【参数说明】

int maxfdpl指定待测试的文件描述字个数，它的值是待测试的最大描述字加1。

fd\_set \*readset , fd\_set \*writeset , fd\_set \*exceptset

fd\_set可以理解为一个集合，这个集合中存放的是文件描述符(file descriptor),即 文件句柄。中间的三个参数指定我们要让内核测试读、写和异常条件的文件描述符集合。 如果对某一个的条件不感兴趣，就可以把它设为空指针。

const struct timeval \*timeout timeout B知内核等待所指定文件描述符集合中的任 何一个就绪可花多少时间。其timeval结构用于指定这段时间的秒数和微秒数。

【返回值】

int若有就绪描述符返回其数目，若超旺则为。，若出错则为-1

select运行机制

select ()的机制中提供一种fd.set的数据结构，实际上是一个long类型的数组，每一 个数组元素都能与一打开的文件句柄(不管是Socket句柄,还是其他文件或命名管道或 设备句柄)建立联系，建立联系的工作由程序员完成，当调用select ()时，由内核根 据10状态修改fd.set的内容，由此来通知执行了 select ()的进程哪一 Socket或文件 可读。

从流程上来看，使用select函数进行10请求和同步阻塞模型没有太大的区别，甚至还 多了添加监视socket,以及调用select函数的额外操作，效率更差。但是，使用select 以后最大的优势是用户可以在一个线程内同时处理多个socket的10请求。用户可以注 册多个socket,然后不断地调用select读职被激活的socket,即可达到在同一个线程 内同时处理多个I。请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达 到这个目的。

select机制的问题

每次调用select,都需要把fd\_set集合从用户态拷贝到内核态，如果fd\_set集合很 大时，那这个开销也很大

同时每次调用select都需要在内核遍历传递进来的所有fd.set,如果fd.set集合很 大时，那这个开销也很大

为了减少数据拷贝带来的性能损坏*，*内核对被监控的f d\_set集合大小做了限制，并且 这个是通过宏控制的，大小不可改变(限制为1024).

1. Poll

poll的机制与select类似，与select任本质上没有多大差别，管理多个描述符也是进行轮询，根据描述符的状态进行处理，但是POU没有最大文件描述符数量的限制。 也就是说，Poll只解决了上面的问题3,并没有解决问题1, 2的性能开销问题。

下面是P11的函数原型：

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout); typedef struct pollfd {

//需要被检测或选择的文件描述符

int fd;

short events; short re vents;

//对文件描述符fd上感兴趣的事件

//文件描述符fd上当前实际发生的事件

} pollfd\_t;

poll改变了文件描述符集合的描述方式，使用了 pollfd结构而不是select的fd\_set 结构，使得Poll支持的文件描述符集合限制远大于select的1024

【参数说明】

struct pollfd \*fds fds是一个struct pollfd类型的数组，用于存放需要检测其状 态的socket描述符，并且调用poll函数之后fds数组不会被清空；一个pollfd结构 体表示一个被监视的文件描述符，通过传递fds指示pollO监视多个文件描述符。其 中，结构体的events域是监视该文件描述符的事件掩码，由用户来设置这个域，结构 体的revents域是文件描述符的操作结果事件掩码，内核在调用返回时设置这个域 nfds\_t nfds记录数组fds中描述符的总数量

【返回値】

int函数返回fds集合中就绪的读、写，或出错的描述符数量，返回。表示超时，返回

Epoll

epoll在Linux2.6内核正式提出，是基于事件駆动的I/O方式，相对于select来说， epoll没有描述符个数限制，使用一个文件描述符管理多个描述符，将用户关心的文件 描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一 次。

Linux中提供的epoll相关函数如下：

int epoll\_create(int size);

int epoll\_ctl(

int epf d,

int op, int fd, struct epoll\_event \*event

);

int epoll\_wait(

int epf d,

struct epoll\_event \* events, int maxevents,

int timeout

1. . epoll\_create函数创建一个epoll句柄，参数size表明内核要监听的描述符数量。 调用成功时返回一个epoll句柄描述符，失败时返回-1。
2. .epoll\_ctl函数注册要监听的事件类型。四个参数解释如下：

epfd表示epoll句柄

□P表示fd操作类型，有如下3种

EPOLL\_CTL\_ADD 注册新的 fd 到 epfd 中

EPOLL\_CTL\_MOD修改已注册的fd的监听事件

EPOLL\_CTL\_DEL 从 epfd 中删除一个 fd

fd是要监听的描述符

event表示要监听的事件

epoll\_event结构体定义如下：

struct epoll\_event {

uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

}；

typedef union epoll\_data {

void \*ptr;

int fd;

uint32\_t u32;

\_uint64\_t u64;

} epoll\_data\_t;

3）. epoll.wait函数等待事件的就绪，成功时返回就绪的事件数目，调用失败时返回 -1,等待超时返回0。

⑴epfd是epoll句柄

⑵events表示从内核得到的就绪事件集合

（3）maxevents告诉内核events的大小

⑷timeout表示等待的超时事件

epoll是Linux内核为处理大批量文件描述符而作了改进的poll,是Linux下多路复用 10接口 select/poll的増强版本，它能显著提高程序在大量并发连接中只有少量活跃 的情况下的系统CPU利用率。原因就是获职事件的时候，它无须遍历整个被侦听的描述 符集，只要遍历那些被内核10事件异步唤塑而加入Ready队列的描述符集合就行了。

epoll除了提供select/poll那种10事件的水平触发（Level Triggered）夕卜，还提供 了边缘触发（Edge Triggered）,这就使得用户空间程序有可能緩存10状态，减少 ep 01 l\_wai t /ep o 1 l\_pwai t的调用，提高应用程序效率。

⑴水平触发（LT）:默认工作模式，即当epoll-wait检测到某描述符事件就绪并 通知应用程序时，应用程序可以不立即处理该事件；下次调用epoll.wait时，会 再次通知此事件

⑵边缘触发（ET）:当epoll\_wait检测到某描述符事件就绪并通知应用程序时, 应用程序必须立即处理该事件。如果不处理，下次调用epoll.wait时，不会再次 通知此事件。（直到你做了某些操作导致该描述符变成未就绪状态了，也就是说边 缘触发只在状态由未就绪变为就绪旺只通知一次）。

LT和ET原本应该是用于脉冲信号的，可能用它来解释更加形象。Level和Edge指的就 是触发点，Level为只要处于水平，那么就一直触发，而Edge则为上升沿和下降沿的 时候触发。比如：0->1就是Edge, 1->1就是Levelo

ET模式很大程度上减少了 epoll事件的触发次数，因此效率比LT模式下高。

1. 给定一个数，算岀平均值是整数的算法。例如；5平均2 份，则结果为，2和支8平均为3份，则为2, 3, 3； 11平均4份，则为2, 3, 3, 3； 15平均3份，则为；5, 5, 5o

public class Divide {

public static int sum(int[] res)[

int sum=0;

for(int i：res) {

sum += i;

return sum;

public static void aveDivide(int n, int n) {

if (n <= 0){

return;

int res [] = new int[n];

int i = 0;

while(true) {

if(sum(res) == m) {

break;

} else {

res[i%n] x 1;

if

public static void main(String f] args) { aveDivide(15,10);

1. linux内核调度详细说一下

i.i、调度策略

定义位于 liniax/include/uapi/linux/sched. h 中

SCHED项ORMAL:普通的分时进程，使用的f air\_sched\_class调度类

SCHED\_FIFO:先进先出的实时进程。当调用程鳶把CPU分配给进程的时候，它把该进程 描述符保留在运行队列链表的当前位置。此调度策略的进程一旦使用CPU则一直运行。如果 没有其他可运行的更高优先级实时进程，进程就继续使用CPU,想用多久就用多久，即使还 有其他具有相同优先级的实时进程处于可运行状态。使用的是rt.sched.class调度类。

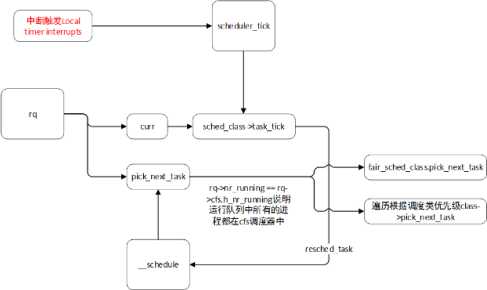
SCHED\_RR:时间片轮转的实时进程。当调度程序把CPU分配给进程的时候，它把该进程 的描述符放在运行队列链表的末尾。这种策略保证对所有具有相同优先级的SCHED\_RR实时 进程进行公平分配CPU时间，使用的rt.sched.class调度类

SCHED\_BATCH:是SCHED项0RMAL的分化版本。采用分时策略，根据动态优先级，分配 CPU资源。在有实时进程的时候，实时进程优先调度。但针对呑吐量优化，除了不能抢占外 与常规进程一样，允许任务运行更长时间，更!子使用高速緩存，适合于成批处理的工作，使 用的fair\_shed\_class调度类

SCHED\_IDLE:优先级最低，在系统空闲旺运行，使用的是idle\_sched\_class调度类， 给。号进程使用

SCHED\_DE.WLINE:新支持的实时进程调度策略，针对突发型计算，并且对延迟和完成时 间敏感的任务使用，基于EDF (earliest deadline first)，使用的是dl\_sched\_class调 度类。

1.2、调度触发



调度的触发主要有两种方式，一种是本地定时中断触发调用scheduler.tick ©数，然 后使用当前运行进程的调度类中的task\_tick,另外一种则是主动调用schedule,不管是哪 —种最终都会调用到—schedule函数，该函数调用pick\_netx\_task,通过rq->nr.running ==rq->cfS.h\_nz\_running判断出如果当前运行队列中的进程都在cfs调度器中,则直接调 用cfs的调度类(内核代码里面这一判断使用)likely说明大部分情况都是滴足该条件的)。 如果运行队列不都在cfs中，贝通过优先级 stop\_s ched\_c1as > dl\_s ched\_c1ass"t\_s che d\_c1ass"air\_s ched\_class~>idle\_s ched\_ class遍历选出下一个需要运行的进程。然后进程任务切换。

处于TASK\_RUNNIMG状态的进程才会被进程调度器选择，其他状态不会进入调度器。系 统发生调度的时机如下：

&调用 cond\_r e s che d ()时

&显式调用schedule ()时

&从中断上下文返回时

当内核开启抢占时，会多出几个调度时机如下：

d在系统调用或者中断上下文中调用preemJenableO时(多次调用系统只会在最后一 次调用时会调度)

d在中断上下文中，从中断处理函数返回到可抢占的上下文时

2、CFS调度

该部分代码位于 1 inux /k erne 1 /s che d/f air. c 中

定义了 const struct sched\_c 1 assf air\_sched\_c 1 ass,这个是 CFS 的调度类定义的对 象。其中基本包含了 CFS调度的所有实现。

CFS实现三个调度策略：

1> SCHED项ORMAL这个调度策略是被常规任务使用

2> SCHED\_BATCH这个策略不像常规的任务那样频繁的抢占,以牺牲交互性为代价 下，因而允许任务运行更长的时间以更好的利用緩存，这种策略适合批处理

3> SCHED.IDLE这是nice值甚至比19还弱，但是为了避免陷入优先级导致问题， 这个问题将会死锁这个调度器，因而这不是一个真正空闲定时调度器

CFS调度类：

n enqueue-task (-)当任务进入runnable状态,这个回调将把这个任务的调度实体 (entity)放入红黑树并且増加nz.running变量的值

n de queue\* ask (…)当任务不再是runnab le状态,这个回调将会把这个任务的调度 实体从红黑树中职出，并且减少nz.running 量的值

n yield\_task(-)除非compat\_yield sysctl是打开的，这个回调函数基本上就是 一个dequeue后跟一个enqueue,这那种情况下，他将任务的调度实体放入红黑树的最右端

n checkereempt\_cuzr (-)这个回调丞数是检查一个任务进入runnable状态是否

应该抢占当前运行的任务

n pick\_next\_taSk (-)这个回调函数选出下一个最合适运行的任务

n Set\_cuzr\_taSk (-)当任务改变他的调度类或者改变他的任务组，将调用该回调函 数

n taSk\_tick(-)这个回调函数大多数是被time tick调用。他可能引起进程切换。 这就駆动了运行时抢占

2.1、CFS 调度

Tcik中断，主要会更新调度信息，然后调整当前进程在红黑树中的位置。调整完成以 后如果当前进程不再是最左边的叶子，就标记为Need.resched标志，中断返回时就会调用 scheduler ()完成切换、否则当前进程继续占用CPU。从这里可以看出CFS抛弃了传统时间 片概念。Tick中断只需要更新红黑树。

红黑树键值即为runtime,该值通过调用update\_cuzr函数进行更新。这个值为 64位的变量，会一直逢増，\_enqueue\_entity中会将vxuntime作为键值将要入队的实体插 入到红黑树中。\_pick\_first\_entity会将红黑树中最左侧即vxuntime最小的实体取出。

21.如何从10亿数据中找到前1000大的数?

public class TopN {

//当前节点的父节点

private int parent(int n){ return (n~l)/2;

}

//当前节点的左子节点

private int left(int n){ return 2\*n+l;

}

//当前节点的右子节点

private int right(int n){ return 2\*n+2;

}

//构建堆

private void buildHeap (int n, int [] data) { for(int i=l;i<n;i++){



while(t!=0 && data[parent(t)]>data[t]) { int temp = data[parent(t)];

data [parent (t) ]=data[t]; data[t]=temp;

t=parent (t);

//调整堆，为小顶堆

private void adjust (int i, int n, int [] data) {

if (data[O]>=data[i]) {

return;

L

int temp = data[i];

data[i] = data[0];

data[0] = temp;

int t=0;

//调整时，堆顶比子节点大，从较小的子节点幵始调整

while((left(t)<n&&data[t]>data[left(t) ]) ||

(right (t) <n&&data[t： >data [right (t) ])) {

if((right(t) < n && data[right(t)] < data[left(t)])){ temp=data[t];

data [t] =data [right (-)];

data[right(t)]=temp.

t=right(t);

}else{

temp=data[t];

data[t]=data[left (t!];

data[left(t)]=temp;

//寻找topN数

public void f indTopN (int n, int.] data) { buildHeap(n, data);

for (int i=n; i<data. length; iT») { adjust (i, n, data);

//打印

public void print(int[] data){

for (int i=0; i<data. length; L) {

System, out. print (data [i. ");

1. 用UDP通讯如何保证对方百分百收到数据？

在UDP通讯中，当你的数据包发出去后，至于对方有没有正确收到数据，并不知道，那 么，如何保证你发出去的数据，对方一定能收到呢？？ ？我们可以借签"P协议的做法（回 复+ 重发+ 编号机制）

1） 接收方收到数据后，回复一个确认包，如果你不回复，那么发送端是不会知道接收 方是否成功收到数据的

比如A要发数据" {data}”到B,那B H姪U后，可以回复一个特定的确认包" {0K}” , 表示成功收到。

但是如果只做上面的回复处理，还是有I礪，比如B收到数据后回复给A的数据〃{0K}〃 的包，A没收到，怎么办呢？ ？ ？

2） 当A没有收到B的〃{0K}〃包后，要做定时重发数据，直到成功接收到确认包为止， 再发下面的数据，当然，重发了一定数量后还是没能收到确认包，可以执行一下姗的流程, 防止对方网卡更换或别的原因。

但是这样的话，B会收到很多重复的数据，假如每次都是B回复确认包A收不到的话。

3） 发送数据的包中加个标识符，比如A要发送的数据〃{标识符|data}〃到B, B收到后, 先回复“{0K} 〃确认包，冉根据丿泉有的标识符进行比较，如果标识符相同，则数据丢失，如 果不相同，则原有的标识符=接收标识符，且处理数据。

当A发送数据包后，没有收到确认包，贝拷隔x秒，把数据重发一次，直到收到确认包 后，更新一下标识符，再进行后一包的数据发送。

经过上面1）,2）, 3）点的做法，则可以保证数据百分百到达对方，当然，标识符用 ID号来代替更好。

1. 如何初始化一个指针数组？

例如:定义f指针瞄**\*a[5]**

让其各个指针头地**1** 畛另脂向

{1,1,1,1,1}, {2,2,2,2,21, {3,3,3,3,31, {4,4,4,4,41, {5,5,5,5,51

int a[5]= {....}

int\* pt [5];

for(i=0;i<5;i++)

[

pt[i] = &a[i]；

}

指针数组不可以直接赋值，因为指针本质上是一组地址。但可以通 过“指向”赋值

1. 你所知道的设计模式有哪些?

有23种设计模式，不需要所有的回答，但是其中常用的几种设计模式应该去掌握。 总体来说设计模式分为三大类：

创建型模式共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。 结构型模式共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合 模式、享元模式。

行为型模式共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模 式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

这里主要讲解简单工厂模式，代理模式，适配器模式，单例模式4中设计模式.

1、 简单工厂模式

主要特点是需要在工厂类中做判断，从而创造相应的产品，当増加新产品时，需要修改 工厂类。使用简单工厂模式，我们只需要知道具体的产品型号就可以创建一个产品。

缺点：工厂类集中了所有产品类的创建谗辑，如果产品量较大，会使得工厂类变的非常 腕肿。

2、 代理模式

代理模式：为其它对象提供一种代理以控制这个对象的访问。在某些情况下，一个对象 不适合或者不能直接引用另一个对象,而代理对象可以在客户端和目标对象之间起到中介作 用。

优点：

职责清晰。真实的角色只负责实现实际依业务逻辑，不用关心其它非本职责的事务，通 过后期的代理完成具体的任务。这样代码会简洁清晰。

代理对象可以在客户端和目标对象之间起到中介的作用，这样就保护了目标对象。

扩展性好。

3、 适配器模式

适配器模式可以将一个类的接口转换成容户端希望的另一个接口,使得原来由于接口不 兼容而不能在一起工作的那些类可以在一起工作。通俗的讲就是当我们已经有了一些类，而 这些类不能满足新的需求，此时就可以考虑是否能将现有的类适配成可以满足新需求的类。 适配器类需要继承或依赖已有的类，实现想要的目标接口。

缺点：过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调 用的是A接口，其实内部被适配成了 B接口的实现，一个系统如果太多出现这种情况，无 异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

4、单例模式

单例模式顾名思义，保证一个类仅可以有一个实例化对象，并且提供一个可以访问它的 全局接口。实现单例模式必须注意一下几点：

单例类只能由一个实例化对象。

单例类必须自己提供一个实例化对象。

单例类必须提供一个可以访问唯一实例化对象的接口。 单例模式分为懒汉和欲汉两种实现方式。

1. 如何计算struct占用的内存？
2. 每个成员按其类型大小和指定对齐参数n中较小的一个进行对齐
3. 确定的对齐参数必须能够整除起始地址(或偏移量)
4. 偏移地址和成员占用大小均需对齐
5. 结构体成员的对齐参数为其所有成员使用的对齐参数的最大值
6. 结构体总长度必须为所有对齐参数的整数倍

#include<stdio. h>

struct test

[

char a;

int b;

float c;

}；

int main(void)

{

printf(^char=%d\n°j sizeof(char):;

printf("int=%d\n”, sizeof(int));

printf ("f loat=%d\n”, sizeof (floa~));

printf(°struct test=%d\n°, sizeof(struct test));

return 0;

I

执行结果为1, 4, 4, 12

占用内存空间的计算过程：

对齐参数为4。假设结构体的起始地址为Ox。

a的类型为char,因此所占内存空间大小为1个字节，小于对齐参数4,所以选择1 为对齐数，而地址0x0能够被1整除，所以0x0为a的起始地址，占用空间大小为1个字节:

b的类型为int ,所占内存空间大小为4个字节，与对齐参数相同，因此4为对齐数， 0x1不能被4整除，因此不能作为b的起始地址，依次往下推，只能选用0x4作为b的起始 地址，因此中间会空出3个字节的空余空间；

c的类型为float,占用4个字节，因此4为对齐数，0x8能被4整除，所以c的起始 地址为0x8

因此整个结构体占用的内存大小为12字节。

**26.**根据要求编程

时间限制：**1**秒

空间限制：32768K

有**n**个字符串，每个字符串都是由**A-J**的大写字符构成。现在佛每个詡映射为 **T 0-9**版字，不够符映射为不同蹒字。这样每个字符串就可以看做f 疊，唯 —的要求是这些皈颁是正整数且它们的学符串不能有前导零。现在问你怎样映射字符 才能使得这些朝串表示值她和最大？

每组;**5^**照包含一每纟**SMle**第一个 **1E^ n ,**接下料 **n**行， 每行一度不超过**12**且仅包含大写字母**A-J**的字符串。**n**不大于**50,**且至少存在 个何？

51^ 述：

础一^，^>±左龄少。

献蚯**1：**

2

ABC

BCA

础阶**1：**

1875

#include <iostream>

#include <String>

#include <algorithm>

#include <cmath>

using namespace std;

struct weight{

long long num;

bool head;

}A[10];

int cmp (const weight const weigh•: &b)

[

return a. num<b. num;

}

int main() {

int n;

cin >> n;

string str;

long long JtlAX = 0;

while(n--) {

cin >> str;

for (int i=str. length ()T; i>=0; i—) {

A[str[i]-，A，].num += pov(10, str. length()-i-l);

if(i==0)

A[str[i]-,A，].head = true;

}

}

sort (A, A+10, cmp);

if(A[0]. head) {

int i;

for(i=l;i<10;i++){

if (JA[i].head) {

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

break;

}

}

sort (A+l, A+i+1, cmp);

}

for(int i=0; i<=9; i++) {

MAX += (A[i].num \* i);

}

cout « JtlAX;

}

27.试题描述：

小**a**和小**b**玩一个游戏,有**n**张卡牌,每张上面有两个正疊**x,y**。取T縛时,个人 积分增加**x,**团队积分增加**V**。

求小**a,**小**b**各取若干张牌，使得他们的个人积分相等。

5mA：

第一行f疊眼

接下来**n**行，每行两个疊**x,y,**用空格隔开。

础：

耘小**a**醐分和**d**、**b**醐分相跑候，团队脇樓大值。

10

其他：

对于 1例跡暖,0<n<=100, l<x<=le3, 0<y<=le6o

#include<iostreajn>

#include<cstring>

#include<cmath>

#include<cstdio>

using namespace std;

int n, m, dp[300][200000], x [100000], y ：100000];

int main()

[

scanf ("％d", &n);

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

[

scanf(°%d%d°, &x[i], &y[i]);

}

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

dp[O] [100000]=0;

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

[

for (int j=20000; j<=200000; j-i-) !

dp[i] [j]=max(dp[i~l] [j+x[i]], dp[i~l] [j-x[i]]); if (dp[i] [j] J=-l)dp[i] [j；+=y[i];

if (dp[i] [j]<dp[i-l] [j])dp[i] [j] =dp[i-l] [j];

}

}

printf(”％d”, dp[n][100000]);

28.试题描述;

f 球爾迷看台可痛S傾頌迷。官方想统计一共有多朝迷群体，最大的 球迷群体有多少人。球迷灘特性：朝帰体会瞄脚座位，不朝迷群体爵不相 邻的座位。（橄曲前后**t**脚、吊**t**脚、斜雜橄**6）** *3*

给定一^蛔牌必，有人，羅望队群体 书炽最大陋队群体入教**Q**。

第—行，2个教字，**M**、使臟文逗号隔开。 接下来**M**行，齣号研。

TT, 2数字，**P**和**Q**。

献样例：

10,10

0,0, 0,0, 0,0, 0,0, 0,0

0,0,0,1,1,0,1,0, 0,0

0,1, 0,0, 0,0, 0,1, 0,1

1,0, 0,0, 0,0, 0,0,1,1

0,0, 0,1,1,1, 0,0, 0,1

0,0, 0,0, 0,0,1,0,1,1

0,1,1,0, 0,0, 0,0, 0,0

0,0,0,1,0,1,0, 0,0,0

0,0,1,0,0,1,0, 0,0,0

0,1, 0,0, 0,0, 0,0, 0,0

6,8

#include<iostreajn>

#include<cstring>

#include<cstdio>

#include<cmath>

#include<stdio. h>

using namespace std;

int n, m, 1, k, sum, ans, ent;

char a[4000][4000], op;

bool b[4000] [4000]= {0};

int dfs(int x, int y)

{

if (a[x-l] [y]==, 1’&&b [x-1] [y]==0；

{

b[x-l] [y]=l;

dfs (x-1, y);

ans++;

}

if (a[x] [y+l]==‘ V&&b[x] [y+l]==0；

[

b[x] [y+l]=l;

dfs (x, y+1);

ans++;

if (a[x-l] [y+l]==‘ V&&b[x-1] [y+l>=0)

[

b[x-l] [y+l]=l;

dfs (x-1, y+1);

ans++;

}

if (a[x+l] [y]==‘ 1’ &&b [x+1] [y]==0；

[

b[x+l] [y]=l;

dfs (x+1, y);

ans++;

}

if (a[x][厂1]==‘l'&&b[x][厂1]==0；

[

b[x][厂 1]=1;

dfs (x,厂 1);

ans++;

}

if (a[x+l][厂 1]==' l'&&b[x+l][厂l：==0)

[

b[x+l][厂 1]=1;

dfs (x+1,厂 1);

ans++;

}

if (a[x+l] [y+l]==‘ l'&&b[x+l] [y+l：==0)

[

b[x+l] [y+l]=l;

dfs (x+1, y+1);

ans++;

I

if (a[x~l] [y-l]==, 1’&&b [xT][厂1：==0)

{

b[x-l][厂 1]=1;

dfs (x-1,厂 1);

ans++;

}

return ans;

}

int main。

[

scanf ("％d%c%d”, &n, &op, &m);

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

for(int j=l;j<=m;j++)

[

getchax ();

a[i] [j]=getchaz();

I

I

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

[

for(int j=l;j<=m;j++)

[

ans=0;

if(a[i] [j]==，O，)b[i] [j]=l; if(a[i][j]==‘ V&&b[i] [j]==0) [

sum++;

cnt=max (ent, df s (i, j；);

char p=',‘； printf (”%d”, sum); putchar(p); printf (”％d”, ent);

1. mysql为什么要使用B+树作为索引呢？

b树的特点：

一个M阶的b树具有如下几个特征：（如下图M=3）（下文的关键字可以理解为有效数 据,而不是单纯的索引）

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[恥，M],向上取整；（儿子数:[2, 3]）

非叶子结点的关键字个数=儿子数-1 *；*（关键字=2）

所有叶子结点位于同一层；

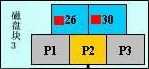
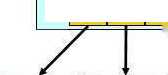
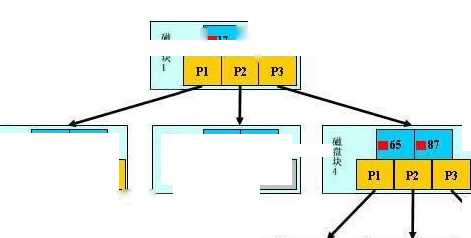
k个关键字把节点拆成k+1段，分别指向蚌1个儿子，同时满足查找树的大小关系*。*（k=2） 有关b树的一些特性，注意与后面的b+树区分：

关键字集合分布在整颗树中；

任何一个关键字出现且只出现在一个结点中；

搜索有可能在非叶子结点结束；

其搜索性能等价于在关键字全集内做一次二分查找：

b+树，是b树的一种变体，查询性能更好。m阶的b+树的特征:

\

*a*

**I；**

| 逾  **1\*.** | **|・8 ■** | | |
| --- | --- | --- | --- |
| 2 | **P1** | **P2** | **P3** |

\

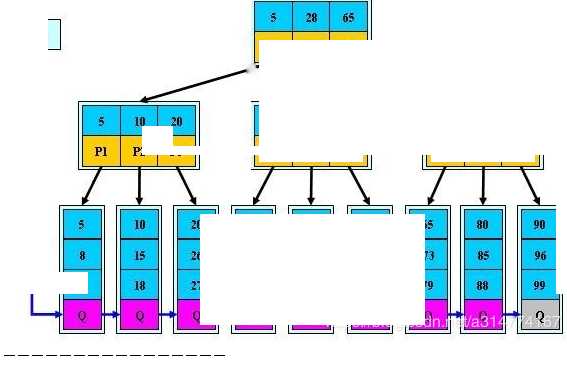
| **■3 |・5** | **■9 H10** | ■】**3 '15** | 皿|皿 |  |
| --- | --- | --- | --- | --- |
| 做**18**埃**5** |  | 磁带块**7** | 懺虞块**8** | 附数块。 |

晚块10

*fOSi^n*

有n棵子树的非叶子结点中含有n个关键字（b树是n-l个），这些关键字不保存数据, 只用来索引，所有数据都保存在叶子节点（b树是每个关键字都保存数据）。

所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子 结点本身依关键字的大小自小而大顺序链接。

所有的非叶子结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字*。* 通常在b+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。 同一个数字会在不同节点中重复出现，根节点的最大元素就是b+树的最大元素。

**DATA**

1U

21

**P2**

**P3**

选用B+树作为数据库的索引结构的原因有

| **P1** | **P2** |  |
| --- | --- | --- |
| **, |** | | |
| **28** |  | **56** |
| **P1** | **P2** | *P3* |

| **65** | **80** | **90** |
| --- | --- | --- |
| **P1** | **P2** | *P3* |

**Z0**

**35**

**26**

**27**

**50**

**27**

38

38

56

60

60

63

63

65

73

73

79

79

B+树的中间节点不保存数据，是纯索引，但是B树的中间节点是保存数据和索引的，相对 来说,B+树磁盘页能容纳更多节点元素，更“搂胖” 5

B+树查询必须查找到叶子节点，B树只要匹配到即可不用管元素位置，因此b+树查找更 稳定(并不慢)；

对于范围查找来说，B+树只需遍历叶子节点链表即可，B树却需要重复地中序遍历，在 项目中范围查找又很是常见的

増删文件(节点)时，效率更高，因为E+树的叶子节点包含所有关键字，并以有序的 链表结构存储，这样可很好提高増删效率。

1. 输入ping IP后敲回车，发包前会发生什么？

ping目标ip时，先查路由表，确定出接口

如果落在直连接口子网内，此时若为以太网等 多路访问网络 则先查询axp緩存，命 中则直接发出，否则在该接口上发axp询问目标ip的mac地址，职得后发出，若为PPP等 点 对点网络，则直接可以发出；

如果查表落在缺省路由上，此时若为以不网等多路访问网络则先查询网关axp緩 存，命中则直接发出，否则在该接口上发axp询问网关的mac地址，取得后发出，若为ppp 等点对点网络*，*则直接可以发出；

若查表未命中，则返回不可达。

1. 对于一颗二叉树，如何对此进行层次遍历，并且按行输

出。

解法1：

class Solution {

public：

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

BFS (root, 0);

return res;

}

public：

void BFS(Node\* node, int level)

[

if(node == nullptr) return;

if (level == res. sizeO)

res. push\_back (vector<in\*:> ());

res[level]. push\_back(node">Tal);

for(auto n：node~>children)

BFS(n, level+1);

private：

vector<vector<int>^ res;

};

解法2:

class Solution {

public：

vector<vector<int>> levelOrder(Hode\* root) { vector〈vectoroutput;

if (Jroot) return output;

queue<Node\*> treeTemp; treeTemp. push(root);

auto treeNum = treeTemp. size();

while (treeNum) {

vector<int> valTemp;

for (auto i=0; KtreeNun; ++i) { root = treeTemp. front(); valTemp. push\_back (root->val); treeTemp. pop();

for (auto childTemp：root">children) treeTemp. push(childTemp);

}

output. push\_back(valTemp); treeNum = treeTemp. size 3 ;

}

return output;

1. 如何实现一个高效的单向链表逆序输岀？

static class Node{

private int data;

private Node next;

Node(int d){

data = d;

next = null;

}

public int getDataO {

return data;

public void setData(int data) { this, data = data;

}

public Node getNext() {

return next;

public void setNext(Node next) { this, next = next;

public static void reverse(Node head)

{

if(null == head || null == head. getNext()){ return;

Node prev=null;

Node pcur = head. getNext();

Node next;

while (pcur *i*=null){

if (pcur.getNext()==null： {

pcur.setNext(prev);

break;

next=pcur.getNext();

pcur.setNext(prev);

prev=pcux;

pcur=next;

}

head. setNext(pcur);

Node tmp = head. getNext();

while(tmpJ= null){

System, out. print(tmp. ge\*Data()+^ “); tmp=tmp. getNext();

1. 从innodb的索引结构分析，为什么索引的key长度不 能太长？

key太长会导致一个页当中能够存放的key的数目变少，间接导致索引树的页数目变 多，索引层次増加，从而影响整体查询变更的效率。

1. MySQL的数据如何恢复到任意时间点？

恢复到任意时间点以定时的做全量备份，以及备份増量的binlog日志为前提。恢复到 任意时间点首先将全量备份恢复之后，再此基础上回放増加的binlog直至指定的时间点。

1. 请解释下为什么鹿啥发布恋情的时候，微博系统会崩溃, 如何解决？

鹿啥首先是一个明星，流量明星。粉丝量众多，所以，他已公布恋情，瞬间的流量很大。 但是我们要注意到，这里面有一个问题。就是这个瞬间流量増大，増的不仅是浏览量。如果 仅仅是阅读，我们只需把鹿啥的这条微博放入Redis緩存，以微博技术，不可能挂得了的 吧。

这个之所以微博挂掉，是因为这个时间段，转发+评论量非常的大，并不是只有阅读 量大。

另外针对明星的微博，会有一个消息推送功能。第一时间热点数据，只要你联的有网， 都能够收到推送。

最后总结如下：

L获取微博通过pull方式还是push方式

1. 发布微博的频率要远小于阅读微博
2. 流量明星的发微博，和普通博主要区分对待，比如在shading的时候，也要考虑 这个因素
3. 设计和实现一个LRU （最近最少使用）缓存数据结构，

使它应该支持一下操作：get和put。

get（key）-如果key存在于缓存中，则获取key的value （总是 正数），否则返回

class LRUCache{

public：

LRUCache(int capacity) {

cap = capacity;

}

int get(int key) {

auto it = m. find (key);

if (it == m. end()) return ~1;

1. splice (1. beginO, 1, it~>second); return it->second">second;

void set(int key, int value) { auto it = m. find (key);

if (it != m. end())

1. erase(it~>second);

1. push\_front(make\_pair(key, value));

m[key] = 1. beginO ;

if (m. size () > cap) {

int k = 1.rbegin()~>first;

1. pop\_\_back ();

m. erase (k);

1. 已知sqrt(2)约等于1.414,要求不用数学库，求sqrt⑵ 精确到小数点后10位。
2. 已知sqrt(2)约等于1.414,那么就可以在(1.4, 1.5)区间做二分查找，如： high=>l.5

low=>l.4

mid => (high+low)/2=1.45

1.45\*1.45>2 ? high=>1.45 : low => 1.45

循环到c)

1. 退出条件

前后两次的差值的绝对值<=0.0000000001,则可退出。

代码：

const double EPSINON = 0.0000000001,

double sqrt2()

[

double low = 1.4, high = 1. 5;

double mid = (low + high) / 2;

while (high - low > EPSINON)

high = mid;

}

else

!

low = mid;

}

mid = (high + low) / 2;

}

return mid;

}

1. 给定一个二叉搜索树(BST),找到树中第K小的节点。

/林

* Definition for a binary tree node.
* public class TreeNode {
* int val;
* TreeNode left;
* TreeNode right;
* TreeNode(int x) { val = x; }
* }

\*/

class Solution {

private class ResultType {

//是否找到

boolean found;

//节点数目

int val;

ResultType(boolean found, int val) {

this, found - found;

this, val = val;

}

}

public int kthSmallest(TreeNode root, int k) { return kthSmallestHelper(root, k). val;

private ResultType k thSmalles tHeIp er(TreeNode root, int k) { if (root == null) {

return new ResultType(false, 0);

ResultType left = kthSmallestHelper(root.left, k);

//左子树找到，直接返回

if (left, found) { return new ResultType(tiue, left, val);

}

//左子树的节点数目=K-l,结果为功。t的值

if (k - left, val == 1) { return new ResultType(tiue, root, val);

}

//右子树寻找

ResultType right = kthSmallestHelper(root, right, k - left, val - 1); if (right, found) {

return new ResultType (tiue, right, val);

*}*

//没找到，返回节点总数

return new ResultType(false, left, val + 1 + right, val);

1. 如何用socket编程实现ftp协议？

以linux为例，使用socket编程中的read。函数和write。函数已可实现文件的发送 接收，为啥还要专门建立ftp协议呢？单单使用:read。函数和write。函数，数据接口和命 令接口未分开，效率低。而ftp将数据接口和命令接口分开，提高了文件传输效率和安全性。

ftp协议的实现仍是使用socket编程，首先是实现tcp连接。

Socket客户端编程主要步骤如下：

socket ()创建一个 Socket

connect ()与服务器连接

write()和readO进行会话

close ()关闭 Socket

Socket服务器端編程主要步骏如下：

socket ()创建一个 Socket

bind()

listenO 监听

accept ()接收连接的请求

write()和readO进行会话

close ()关闭 Socket

建立tcp连接代码简示如下：

SOCKET control\_sock;

struct hostent \*hp;

struct sockaddr\_in server;

memset(^server, 0, sizeof(struct sockaddr\_in))

/\* 初始化 socket \*/

control\_sock = socket(AF\_INET, SOCK\_STREAJ.f, 0);

hp = gethostbyname(server\_name);

memcpy(&server・ s in\_addr, hp->h\_addr, hp->h\_length);

server. sin\_family - AF\_INET;

server. sin\_port = htons(port);

/\*连接到服务器端\*/

connect(control\_sock, (struct sockaddr \*)^server, sizeof(server));

/\*客户端接收服务器端的一些欢迎信息\*/

read (control\_sock, read\_\_buf, read\_\_len);

ftp客户端与服务器建立起tcp连接后，然后向服务器发送命令。(这一步建立的是命 令接口的tcp连接，数据接口连接尚未建立。)

通常第一步发送USER和PASS命令给证账号和密码后登陆服务器。若是匿名服务器则另 当别论。

登陆服务器代码简示如下：

/\* 命令”USER us er name \*/

sprintf(send\_buf, "USER %s\r\n°, username);

/\*客户端发送用户名到服务器端\*/

write(control\_sock, send\_buf, strlen(send\_buf));

/\*客户端接收服务器的响应码和信息，正常为

”331 User name okay, need password.”

\*/

read(control\_sock, read\_buf, read\_len);

/\* 命令 ” PASS password\r\n,? \*/

sprintf (send\_buf, "PASS passvord);

/\*客户端发送密码到服务器端\*/

write(control\_sock, send\_buf, strlen(send\_buf));

/\*客户端接收服务器的响应码和信息，正常为"230 User logged in, proceed.” \*/ read(control\_sock, read\_buf, read\_len);

接下来是选择发送PORT命令选择主动模弍还是发送PASV命令选择被动模式。

主动模式还是被动模式是相对服务器来说的。被动模式即命令接口连接和数据接口连接 都由客户端主动建立；主动模式是数据接口蟻由服务器主动建立。这里仅简述被动模式的 建立，主动模式可依理建立。

被动模式建立连接代码：

/\* 命令"PASV\r\n” \*/

sprintf(send\_buf, "PASV\r\n");

/\*客户端告浦服务器用被动模式\*/

write (control\_sock, send\_\_buf, str 1 en (send\_\_buf));

/\*客户端接收服务器的响应码和新开的端口号，\*正常为

"227 Entering passive mode (<hl, h2,h3, h4, pl, p2>)”

\*/

read(control\_sock, read\_buf, read\_len);

客户端发送PASV命令让服务器进入被动模式。服务器会打开数据端口并监听。并返回 响应码227和数据连接的端口号。

接下来通过数据端口下载上传查看文件就不整述了。

/\*连接服务器新开的数据端口 \*/

connect(data\_sock, (struct sockaddr x)&server, sizeof(server));

/\* 命令 ” CWD dirnameVr'n” \*/

sprintf(send\_buf, "CWD %s\r\n^, dirname);

/\*客户端发送命令改变工作目录\*/

write (control\_sock, send\_\_buf, str 1 en (send\_\_buf));

/\*客户端接收服务器的响应码和信息，正常为” 250 Command okay.” \*/

read(control\_sock, read\_buf, read\_len);

/\* 命令"SIZE filename\:r\n” \*/

sprintf (send\_buf, "SIZE *%s\x\n\* filename);

/\*客户端发送命令从服务器端得到下载文件的大小\*/

write (control\_sock, send\_\_buf, str 1 en (send\_\_buf));

/\*客户端接收服务器的响应码和信息，正常为” 213 "ize>” \*/

read(control\_sock, read\_buf, read\_len);

/\* 命令 ” RETR filenajne\r\n,? \*/

sprintf (send\_buf, "RETR *%s\x\n\* filename);

/\*客户端发送命令从服务器端下载文件\*/

write (control\_sock, send\_\_buf, str 1 en (send\_\_buf));

/\*客户端接收服务器的响应码和信息，正常为” 150 Opening data connection. ” \*/ read (control\_sock, read\_\_buf, read\_\_len);

/\*客户端创建文件\*/

file\_handle = open(disk\_name, CRFLAGS, RWXALL);

&(；；){

/\*客户端通过数据连接从服务器接收文件内容\*/

read(data\_sock, r*e*ad\_\_buf, r*e*ad二en);

/\*客户端写文件\*/

write (f ile\_handl*e,* read\_\_buf, r*e*ad\_len);

}

/\*客户端关闭文件\*/

rc = close (file\_handle);

下载完成后客户端退出服务器，关闭连接。

/\*客户端关闭数据连接\*/

close(data\_sock);

/\*客户端接收服务器的响应码和信息，正常为” 226 Transfer complete." \*/

read(control\_sock, read\_buf, read\_len);

/\* 命令 ” QUIT\r\n” \*/

sprintf (send\_buf, "QUIT\:r\n");

/\*客户端将商开与服务器端的连接\*/

write (control\_sock, send\_\_buf, str 1 en (send\_\_buf));

/\*客户端接收服务器的响应码，正常为” 200 Closes connection. " \*/

read (control\_sock, read\_\_buf, read\_\_len);

/\*客户端关闭控制连接\*/

close(control\_sock);

断点续传的实现

由于网络不稳定，在传输文件的过程中，可能会发生连接断开的情况，这时候需要客户 端支持断点续传的功能，下次能够从上次终止的地方开始接着传送。需要使用命令REST。 如果在断开连接前，一个文件已经传输了 512个字节。则断点续传开始的位置为512,服 务器会跳过传输文件的前512字节。

1. 给定一个数组，它的第i个元素是一支给定股票第i天的 价格。设计一个算法来计算你所能获取的最大利润。你可以 尽可能地完成更多的交易(多次买卖一支股票)。注意：你 不能同时参与多笔交易(你必须在再次购买前出售掉之前的 股票)。

package test;

import java. util. Scanner;

public class Many {

public static void main(String[. args) {

Scanner sc = new Scanner(System, in);

String str =”“；

str = sc. nextLine (); //读入数组中所有元素；

String arr[] = str. split C 0； // 用空格分割字符串

int n = arr. length;

int arra[] = new int [n];

for (int i = 0; i < arra.length; i++) {

arra[i] - Integer.parselnt(arr[i]);

// System, out. print(arra[i] + ";

} //将字符串转化为整数数组

int max = 0; //盈利最大

for (int i = 0; i < axra.length - 1; i++) {

int t = axra[i+l] - axra[i];

if(t > 0) max += t;

}

System, out. printIn(max);

1. 描述实时系统的基本特性

实时系统是指在系统工作时，能在特定啊寸间内完成特定的任务，其各种资源可以根据 需要进行动态的分配，因此其处理事务的能力强，速度快。

1） 高精度计时系统

计时精度是影响实时性的一个重要因素。在实时应用系统中，经常需要精确确定实时地 操作某个设备或执行某个任务，或精确的计算一个时间函数。这些不仅依赖于一些硬件提供 的时钟精度，也依赖于实时操作系统实现的高精度计时功能。

2） 多级中断机制

—个实时应用系统通常需要处理多种外部信息或事件，但处理的紧迫程度有轻重緩急之 分。有的必须立即作出反应，有的则可以延后处理。因此，需要建立多级中断嵌套处理机制, 以确保对紧迫程度较高的实时事件进行及时响应和处理。

3） 实时调度机制

实时操作系统不仅要及时响应实时事件中断，同时也要及时调度运行实时任务。但是, 处理机调度并不能随心所欲的进行，因为涉及到两个进程之间的切换，只能在确保“安全切 换”的时间点上进行，实时调度机制包括两个方面，一是在调度策略和算法上保证优先调度 实时任务；二是建立更多“安全切换”时间点，保证及时调度实时任务。

1. 求任意一颗二叉树最长路径长度

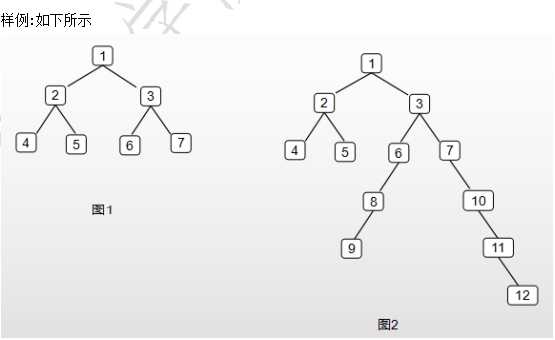


图1树的最长路径长度为4,图2的最长路径长度为7,图1最长路径经过根节点，顶 点为1,图2不经过，顶点为3

思路：

有中任意两个节点之间，连接起来的路径最长。方法就是求出每个节点的左子树和右子 树的高度，两者相加就是当前节点的最长路径，然后比较每个节点的最长路径，最大的就是 结果

实现方法：

定义一个静态变量MaxLength记录每一步最大长度，采职前序遍历来遍历每一个节点, 在遍历过程中，对当前节点的最长路径进行比较，对于每一个节点最长路径求法，先求出它 左子树和右子树的高度(节点数最多的路径)，然后相加即为当前节点最长路径

static Integer MaxLength=0;//记录最长路径

//遍历整棵树，得到最长路径

public void g e tL eng th(Tr e eNo de t){

if(tJ=null){

KaxLength=Math. max(LengthTree(t), MaxLength);

getLength(t. 1 chi Id); getLength(t. r chi Id);

}

}

//得到当前节点的最长路径

public int LengthTree(TreeNode t){

if (t==null)

return 0;

int left=heighTree(t. 1chiId);

int right=heighTree(t. rchiId);

int CurMax= 1 ef t-hr i ght;

return CurMax;

//求二叉树最大高度

public int heighTree(TreeNode t){

if (t==null)

return 0;

else

return Math, max(heighTree(t.Ichild), heighTree(t. rchiId))+1;

}

1. 手写冒泡排序算法,计算冒泡排序算法的时间复杂度？

public void swap(int[] array, int i, int j) {

array[i] = array[i] + array[j];

array[j] - array[i] 一 array[j];

array[i] - array[i] 一 array[j];

public void sort(int[] array) {

int n = array, length;

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

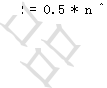
/依这里j不需要遍历到nT 了，因为n-l-i"n-l之间的元素 //已经排好序了，不需要再比较

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

/R寻最大元素移动到数组末尾

if (axraytj] > array[j - 1]) {

swap (array, j, j + 1)；

从代码中可以看出一共遍历了 n-1 + n-2 + ― + 2 + 1 = n \* (n~l) / 2

2 - 0. 5 \* n,

那么时间复杂度是0(『2)。

1. redis中的网络IO有了解过吗，它是单线程的还是多线程

的，为什么要用单线程。

redis采用网络10多路复用技术来保证在多连接的时候，系统的高呑吐量。

多路-指的是多个socket连接，复用-指的是复用一个线程。多路复用主要有三种技术: select, poll, epollo epoll是最新的也是目前最好的多路复用技术。

这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路I/O 复用技术可以让单个线程高效的处理多个连接请求(尽量减少网络I。的时间消耗)， 且Redis在内存中操作数据的速度非常快(内存内的操作不会成为这里的性能瓶颈)，主要 以上两点造就了 Redis具有很高的呑吐量。

**Figure 6.3. I/O multiplexing model.**

jppllolion

kernel

select

no datagram rvady

pivetg bkx:k> in vail <u select, waiting for ow of possibly many wekets to become rv.Hl.tble

proce\*\* bkvk^ tvhile data copied into applies I ion buffer

return readable

wait for cLila

reevfren

datagram tv^dy copy datagram

copy data from kcnwl to itovr

return OK

pnvfss  
datagram

因为Redis是基于内存的操作，CPU不是Re dis的瓶颈，Re dis的瓶颈最有可能是机器 内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，所以采用单线程



零声学「謐理出品

更多、更全大厂面试一料加Q群：762073882 的方案。

1. Learning to Rank 了解吗，三种模式说一下(pairwise、point

wise、list wise)

在机器学习的 ranking 技术 learning2rank＞包括 pointwisexpairwisexlistwise

三大类型。

1、 Pointwise Approach

1.1特点

Pointwise类方法，其L2R框架具有以下特征：

输入空间中样本是单个doc （和对应query）构成的特征向量；

输出空间中样本是单个doc （和对应query）的相关度；

假设空间中样本是打分函数；

损失函数评估单个doc的预测得分頭真实得分之间差异。

这里讨论下，关于人工标注标签怎么转换到pointwise类方法的输出空间： 如果标注直接是相关度s\_j,则doc x\_j的真实标签定义为y\_j=s\_j

如果标注是pairwise preference *s\_{u,v],*贝lj doc x\_j的真实标签可以利用该 doc击败了其他docs的频次

如果标注是整体排序兀，则docx二的真实标签可以利用映射函数，如将doc的 排序位置序号当作真实标签

1.2根据使用的ML方法不同，pointvise类可以进一步分成三类：基于回归的算 法、基于分类的算法，基于有序回归的算法。

（1） 基于回归的算法

此时，输出空间包含的是实值相关度得分。采用也中传统的回归方法即可。

（2） 基于分类的算法

此时，输出空间包含的是无序类别。对于二分类，S㈣、LR等均可；对于多分 类，提升树等均可。

（3） 基于有序回归的算法

此时，输出空间包含的是有序类别。通常是找到一个打分函数，然后用一系列 阈值对得分进行分割，得到有序类别。采用PRanking、基于margin的方法 都可以。

1.3缺陷

回顾概述中提到的评估指标应该基于query和position,

ranking追求的是排序结果，并不要求精确打分，只要有相对打分即可o pointwise 类方法并没有考虑同一个query对应的docs间的内部依赖性。一方面，导致输 入空间内的样本不是IID的，违反了 ML的基本假设，另一方面，没有充分利用 这种样本间的结构性。其次，当不同query对应不同数量的docs时，整体loss 将会被对应docs数量大的query组所支配，前面说过应该每组query都是等价 的。

损失函数也没有model到预测排序中的位置信息。因此，损失函数可能无意的过 多强调那些不重要的docs,即那些排序在后面对用户体验影响小的doco

1.4改进

如在loss中引入基于query的正则化因子的RankCosine方法。

2、 Pairwise Approach

2.1特点

Pairwise类方法，其L2R框架具有以下特征：

输入空间中样本是（同一query对应的）两个doc （和对应query）构成的两个 特征向量；

输出空间中样本是pairwise preference;

假设空间中样本是二变量函数；

损失函数评估doc pair的预测preference和真实preference之间差异。 2.2基于二分类的算法

Pairwise类方法基本就是使用二分类算法即可。

经典的算法有基于NN的SortNet ,基于NN的RankNet,基于fidelity loss 的 FRank,基于 AdaBoost 的 RankEoost,基于 SVM 的 RankingSW,基于提升树 的 GBRanko

2.3缺陷

虽然pairwise类相较pointwise类model到一些doc pair间的相对顺序信 息，但还是存在不少问题，回顾概述中提到的评估指标应该基于query和 position^

如果人工标注给定的是第一种和第三种，即已包含多有序类别，那么转化成 pairwise preference时必定会损失掉一些更细粒度的相关度标注信息。

doc pair的数量将是doc数量的二次，从而point wise类方法就存在的query 间doc数量的不平衡性将在pairw^e类方法中进一步放大。

pairwise类方法相对pointwise类方法对噪声标注更敏感，即一个错误标注会引 起多个doc pair标注错误。

pairwise类方法仅考虑了 doc pair的相对位置,损失函数还是没有model到预 测排序中的位置信息。

pairwise类方法也没有考虑同一个query对应的doc pair间的内部依赖性，即 输入空间内的样本并不是IID的，语反了 ML的基本假设，并且也没有充分利用 这种样本间的结构性。

2.4改进

pairwise类方法也有一些尝试，去一定程度解决上述缺陷，比如：

Multiple hyp er plane ranker,主要针对前述第一个缺陷 magnitude-preserving ranking,主要针对前述第一个缺陷

IRSW,主要针对前述第二个缺陷

采用Sigmoid进行改进的pairwise方法，主要针对前述第三个缺陷

P-norm push,主要针对前述第四个缺陷

Ordered weighted average ranking,主要针对前述第四个缺陷

LambdaRank,主要针对前述第四个缺陷

Sparse ranker,主要针对前述第四个缺陷

3、 Listwise Approach

3.1特点

Listwise类方法，其L2R框架具有以下特征：

输入空间中样本是（同一query对应的）所有doc （与对应的query）构成的多 个特征向量（列表）*；* 输出空间中样本是这些doc （和对应query）的相关度排序列表或者排列； 假设空间中样本是多变量函数，对于docs得到其排列，实践中，通常是一个打分 函数，根据打分函数对所有docs的打分进行排序得到docs相关度的排列； 损失函数分成两类，一类是直接和评价指标相关的，还有一类不是直接相关的。具 体后面介绍。

3.2根据损失函数构造方式的不同,listwise类可以分成两类直接基于评价指标的算 法，间接基于评价指标的算法。

（1） 直接基于评价指标的算法

直接取优化ranking的评价指标，也算是listwise中最直观的方法。但这并不 简单，因为前面说过评价指标都是富散不可微的，具体处理方式有这么几种： 优化基于评价指标的ranking error的连续可微的近似，这种方法就可以直接应 用已有的优化方法，如 SoftRank, ApproximateRank, SmoothRank

优化基于评价指标的ranking error的连续可微的上界，如SVW頌，SW-NDCG, PermuRank

使用可以优化非平滑目标函数的优化技术，如AdaRank, RankGP

上述方法的优化目标都是直接和racing的评价指标有关。现在来考虑一个概念, informativeness o通常认为一个更有信息量的指标，可以产生更有效的排序模型。 而多层评价指标（NDCG）相较二元评价（AP）指标通常更富信息量。因此，有时虽 然使用信息量更少的指标来评估模型，但仍然可以使用更富信息量的指标来作为 loss进行模型训练。

（2） 非直接基于评价指标的算法

这里，不再使用和评价指标相关的loss来优化模型，而是设计能衡量模型输出与 真实排列之间差异的1。奖，如此获得的模型在评价指标上也能获得不错的性能。

经典的如 , ListNet, ListMLE, StiuctRank, BoltzRanko

3.3缺陷

listwise 类相较 pointwise \* pairwise 对 ranking 的 model 更自然，解决了 ranking应该基于query和position问题。listwise类存在的主要缺陷是：一 些ranking算法需要基于排列来汁算1。”，从而使得训练复杂度较高，如 ListNet和BoltzRanko此外，位置信息并没有在loss中得到充分利用，可以考 虑在ListNet和ListMLE的loss中引入位置折扣因子。

3.4改进

pairwise类方法也有一些尝试，去一定程度解决上述缺陷，比如：

Multiple hyp er plane ranker,主要针对前述第一个缺陷 magnitude-preserving ranking,主要针对前述第一个缺陷 IRSW,主要针对前述第二个缺陷

采用Sigmoid进行改进的pairwise方法，主要针对前述第三个缺陷

P-norm push,主要针对前述第四个缺陷

Ordered weighted average ranking,主要针对前述第四个缺陷

LambdaRank,主要针对前述第四个缺陷

Sparse ranker,主要针对前述第四个缺陷

以上，这三大类方法主要区别在于损失函数。不同的损失函数决定了不同的模型学 习过程和输入输出空间。

1. 用户输入M,N值，从1至N开始顺序循环数数，每数到

M输岀该数值，直至全部输出。写岀C程序。

#include <stdio・h>

#include <stdlib. h>

typedef struct node{

int data;

node\* next;

Jnode;

void CreatList(node\*& head, node\*& -ail, int n){

if(n<l){

head = tail = NULL;

return;

head = new node ();

head~>data = 1;

head->next = NULL;

node\* p = head;

for(int i=2; i<n+l; i++){

p->next =new node ();

p = p->next;

p->data = i;

p->next = NULL;

tail = p;

tail -〉 next = head;

void Print(node\*& head){

node\* p = head;

while(p && p~>next != head){

printf (p~ > dat a);

p = p\_>next;

}

if(p) {

printf (p~>data);

void CountPrint (no de\*& head, no de\*& tail, int m) {

node\* pre = tail;

node\* cur - head;

int ent = m;

while(cur && cur">next != cur){

if(ent != 1){

ent一一;

pre = cur;

cur = cur">next;

!else{

printf (”%d”, cux->data).

pre->next =cux->next;

delete cur;

cur = pre-^next;

ent = m;

}

}

if (cur) {

printf (”%d”, cux->data);

delete cur;

head = tail =NULL;

int main() {

node\* head;

node\* tail;

int m;

int n;

s c anf\_s ("昵d", &n);

scanf\_s (”％d", &m);

CreatList (head, tail, n);

Print(head);

CountPrint (head, tail, m);

system("pause");

return 0;

}

1. zset的底层是用什么数据结构实现的。

zset底层的存储结构包括ziplist或skiplist,在同时满足以下两个条件的时候使用 ziplist,其他时候使用skiplist,两个条件如下：

有序集合保存的元素数量小于128个

有序集合保存的所有元素的长度小于64字节

当ziplist作为zset的底层存储结构时候，每个集合元素使用两个紧挨在一起的压缩

列表节点来保存，第一个节点保存元素的成员，第二个元素保存元素的分值。

当skiplist作为zset的底层存储结构的时候，使用skiplist按序保存元素及分值, 使用diet来保存元素和分值的映射关系。

ziplist数据结构

ziplist作为zset的存储结构时，格式如下图，细节就不多说了，我估计大家都看得 懂，紧挨着的是元素memeber和分值socore,整体数据是有序格式。

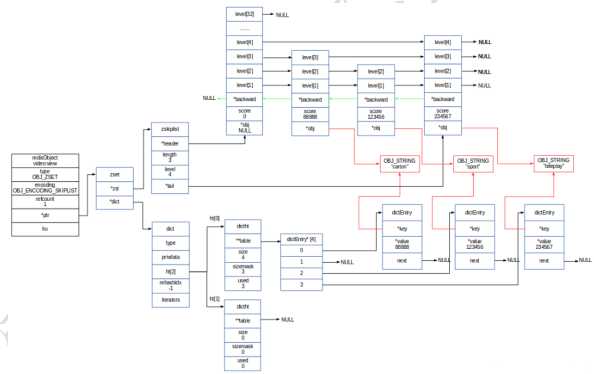
ziplist header

zset element

zl bytes zltail zllen value 0 score 0 ... zlend

skiplist数据结构

skiplist作为zset的存储结构，整体存储结构如下图，核心点主要是包括一个diet对象 和一个skiplist对象。diet保存key/value, key为元素，value为分值；skiplist保存的有序的 元素列表，每个元素包括元素和分值。两种数据结构下的元素指向相同的位置。



zset存储过程

zset的添加过程我们以zadd的操作作为例子进行分析，整个过程如下：

解析参数得到每个元素及其对应的分值

查找key对应的zset是否存在不存在则创建

如果存储格式是ziplist,那么在执行添加的过程中我们需要区分元素存在和不存在两种 情况，存在情况下先删除后添加；不存在情况F则添加并且需要考虑元素的长度是否超出限 制或实际已有的元素个数是否超过最大限制逹而决定是否转为skiplist对象。

如果存储格式是skiplist,那么在执行添加的过程中我们需要区分元素存在和不存在两种 情况，存在的情况下先删除后添加，不存在情况下那么就直接添加，在skiplist当中添加完

以后我们同时需要更新diet的对象。

1. 如果你要对班里的学生根据分数进行排名，你觉得用

redis里的哪个数据结构比较好。

1. 在第一象限内，有一些(n个)离散的点(x,y均为自然数, 且某一行、列都只有一个点)

代码需要输岀一些“最大点”(设待考察点为x0,y0,如果 存在x,y,满足x>x0且y>y0,则x0,y0不是最大点),输出顺 序为x值递增顺序。

Z)；寸于所有输入的点，按照X值逢増排序。一一因为X值没有重复，且已知X的范围, 所以使用位图法排序，复杂度0 (n)*。*直接sort就当做是nlog (n)咯！

1. 从最右侧的点(xn, yn)开始研究，他的右边没有点了，题干的条件被破坏，那么

(xn, yn)是最大点；同时用max来表示当前研究点右侧(包含当前)点的最大y值。

1. 研究(x n-l,y n-1)，因为x n~l的右侧已经有点x n,那么(x n-l,y n~l)是不

是最大点的关键就在于，y n-l<max?如果小于，那x nT不是最大点，否则他是最大点。 更新max值

1. 循环第三步，直到待考察的点为空。一一复杂度。3)
2. 给定一个序列，如【6,2,1】，要求输出；在某一个区间 上的最小值min与这个区间上所有值的和sum的乘积，最 终只输岀最大的乘积值。

思路；肯定是用两个index来模拟区间范围，然后在区间里寻找min和sum 了。 那么可以这么做：

int max="l;

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

int sum - 0;

int min = array[i];

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

零声学「謐理出品

更多、更全大厂面试斐料加Q群：762073882 if (array [j]〈min)min=array [j.; 新 min sum+=array[j];/德新 sum if ( min\*sum>max) max=min\*sun;

零声学「謐理出品

更多、更全大厂面试姿料加**Q**群：**762073882**整、更全大厂面试资料加**Q**群：**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®现高并灘騁.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**，网络承§一^**1**走**.mp4**

**Q** 服家能職化\_糸璀遊**.mp4**

**Q** 区块链的底昙，去中心化网塔的设计**.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

**Q**后台服务进暨了怎么办**.mp4**