

2022 年京东精选 50 面试题及答案

1. 求 1~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 ll;
#define maxn 100009
int fact[maxn];
bool prime[maxn];
ll mod = 987654321;
int cal(int t, int p) {
    int cnt = 0;
    while(t % p == 0) {
        cnt++;
        t /= p;
    }
    return cnt;
}
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; j++) {
            if(prime[j] && i % j == 0) {
                fact[j] = max(fact[j], cal(i, j));
            }
        }
        if(prime[i])
            fact[i] = max(fact[i], 1);
    }
}

int main() {
    ll n;
    cin>>n;
    solve(n);
    ll ans = 1;
    for(ll i = 1; i <= n; i++) {
        for(ll j = 1; j <= fact[i]; j++) {
            ans = ans * i % mod;
        }
    }
    cout<<ans<<endl;
    return 0;
}
```

2. 去掉字符串构成回文。其实是经典的求回文子序列个数。

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
ll f[59][59];
string str;
ll dfs(int i, int j) {
    if(i > j) {
        return 0;
    }
    if(i == j) {
        f[i][j] = 1;
        return f[i][j];
    }
    if(f[i][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 >> str;
        int len = str.length();
        cout << dfs(0, len - 1) << endl;
        return 0;
    }
```

3. 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 ll;
ll dp[10][10][3];
ll mod = 1e9 + 7;
int dx[8] = {-2, -1, 1, 2, 2, 1, -1, -2};
int dy[8] = {-1, -2, -2, -1, 1, 2, 2, 1};
int check(int x, int y) {
    if(x >= 0 && x <= 8 && y >= 0 && y <= 8)
        return true;
    return false;
}
void cal(int x, int y, int state) {
    dp[x][y][state] = 0;
    for(int i = 0; i < 8; i++) {
        int tx = x + dx[i];
        int ty = y + dy[i];
        if(check(tx, ty)) {
            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;
}
```

4. 如何验证图的连通性?

```
#include<iostream>
#include<queue>
#include <stdio.h>

using namespace std;
#define MAX_VNUM 10

typedef struct
{
    int weight;
}Adj,AdjMatrix[MAX_VNUM][MAX_VNUM];

typedef struct
{
    AdjMatrix adjM;
    int vNum;
}adjGraph;

//创建一个图,节点从 0 开始,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.adjM[i][j].weight;
    }
}

//输出一个图
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;//将换行流写入输出流，清空输出缓冲区
    }
}

//warshall 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[i][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 Warshall,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; //明白队列用途?
    bool visit[MAX_VNUM]; //访问数组
    int count = 0;
    memset(visit, 0, sizeof(visit));
    q.push(0); //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入队列
        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,传递数组会改变值,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(visit,0,sizeof(visit));
    dfs_visit(G,0,visit); //从 0 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;
}
```

5. git pull 和 git merge 区别?

你修改好了代码，先要提交

```
git commit -am "commit message"
```

然后有两种方法来把你的代码和远程仓库中的代码合并:

- 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。
- 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。
- 如果没有冲突，git push origin master，把你的改动推送到远程仓库中。

6. 手写快速排序代码.

```
public static int partition2(int arr[], int l, int r) {
    //基准元素设为第一个
    int v = arr[l];
    //i 指向基准的下一个元素，j 指向最后一个元素
    int i = l+1, j = r;
    while(true) {
        while(i <= r && arr[i] < v) i++;
        while(j > l && arr[j] > v) j--;
        //循环终止条件
        if(i > j) break;
        //交换 arr[i] 与 arr[j]
        int t = arr[i];
        arr[i] = arr[j];
        arr[j] = t;
        i++;
        j--;
    }
    //将基准元素与 arr[j] 交换
    int t = arr[l];
    arr[l] = arr[j];
    arr[j] = t;
}
```



```
//返回基准元素所在位置  
return j;  
}
```

7. 内存分配方式有几种?

内存分配方式:

(1) 符号起始的区块 (.bss 段): 通常指的是存放程序中未初始化或者初始化为 0 的变量的和静态数据的区域。bss 属于静态内存分配, 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, 它的分配方式类似于链表。

(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, 当函数被调用时, 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栈。栈内存分配运算内置于处理器的指令集中, 一般使用寄存器来存取, 效率很高, 但是分配的内存容量有限。

8. 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。

9. 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。递归的解决这些问题, 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, 一步一步地做出贪心的选择

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，让以后再遇到时直接引用，不必要重新求解

10. 分布式服务接口请求的顺序性如何保证？

①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。

②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。

③最好能将多个操作合并成一个操作。

12. 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。

13. 常用的内存管理方法有哪几种？

段式

页式

段页式

14. 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

15. 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”i am a little boy” ,C++语言实现，不要用伪代码作答，函数输入输出请参考如下的函数原型：

C++函数原型：

```
void FormatString(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] != '\0') {
        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);
    FormatString(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），要求给出这两个 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，函数输入输出请参考如下的函数原型：

C++函数原型：

```
structy TreeNode{
    TreeNode* left; //指向左子树
    TreeNode* right; //指向右子树
    TreeNode* father; //指向父亲节点
};

TreeNode* LowestCommonAncestor(TreeNode* first, TreeNode* second) {
}

int nodeHeight(TreeNode* node)
{
    int height = 0;
    while(node != NULL)
    {
        height++;
        node = node->father;
    }
}

TreeNode* LowestCommonAncestor(TreeNode* first, TreeNode* second)
{
    int diff = nodeHeight(first) - nodeHeight(second);
    if(diff > 0)
```

```
{
    while(diff > 0)
    {
        first = first->father;
        diff--;
    }
}
else
{
    while(diff < 0)
    {
        second = second->father;
        diff++;
    }
}
while(first != second)
{
    first = first->father;
    second = second->father;
}
return first;
}
```

17. 计算第 K 个能表示($2^i * 3^j * 5^k$)的正整数 (i,j,k 为整数)？其前 7 个满足此条件的数分别是 1,2,3,4,5,6,8.

```
public class Main
{
    public static void main(String[] args)
    {
        int[] a = new int[1501];
        a[1] = 1;
        TreeMap<Integer, Integer> map = new TreeMap<Integer, Integer>();
        Deque<Integer> Q2 = new ArrayDeque<Integer>();
        Deque<Integer> Q3 = new ArrayDeque<Integer>();
        Deque<Integer> Q5 = new ArrayDeque<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.Entry<Integer, Integer> e = map.pollFirstEntry();
int key = e.getKey();
int val = e.getValue();

if (val == 5) {
    Q5.add(key * 5);
    map.put(Q5.pollFirst(), 5);
} else if (val == 3) {
    Q5.add(key * 5);
    Q3.add(key * 3);
    map.put(Q3.pollFirst(), 3);
} else {
    Q5.add(key * 5);
    Q3.add(key * 3);
    Q2.add(key * 2);
    map.put(Q2.pollFirst(), 2);
}

a[i] = key;
}

Scanner sc = new Scanner(System.in);
while (sc.hasNext()) {
    System.out.println(a[sc.nextInt()]);
}
}
```

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 $[M/2]$ 。

每个结点存放至少 $M/2 - 1$ （取上整）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 - 1；

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为 $[M/2]$ 。

每个结点存放至少 $M/2-1$ （取上整）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 - 1；
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. 分布式服务接口的幂等性如何设计（比如不能重复扣款）？

所谓幂等性，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。这就是幂等性，不给大家来学术性词语了。

其实保证幂等性主要是三点：

- （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，一个订单 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 mysql 中记录个状态啥的，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，orderId 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 orderId 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款。

要求是支付一个订单，必须插入一条支付流水，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，order_id 就已经进去了
你就可以写一个标识到 redis 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，如果是 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. 参数

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存的尺寸。

2. 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 new 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* 指针转换成我们需要的类型。

3. 分配失败

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

4. 自定义类型

new 会先调用 operator new 函数, 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。

5. 重载

C++ 允许重载 new/delete 操作符, 特别的, 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。

6. 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存。自由存储区是 C++ 基于 new 操作符的一个抽象概念, 凡是通过 new 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 dgoo。

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。

开始时, 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 $O(n)$, 查找兄弟单词时时间复杂度是 $O(1)$ 。

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 $O(n)$ ，查找兄弟单词时时间复杂度是 $O(1)$ 。

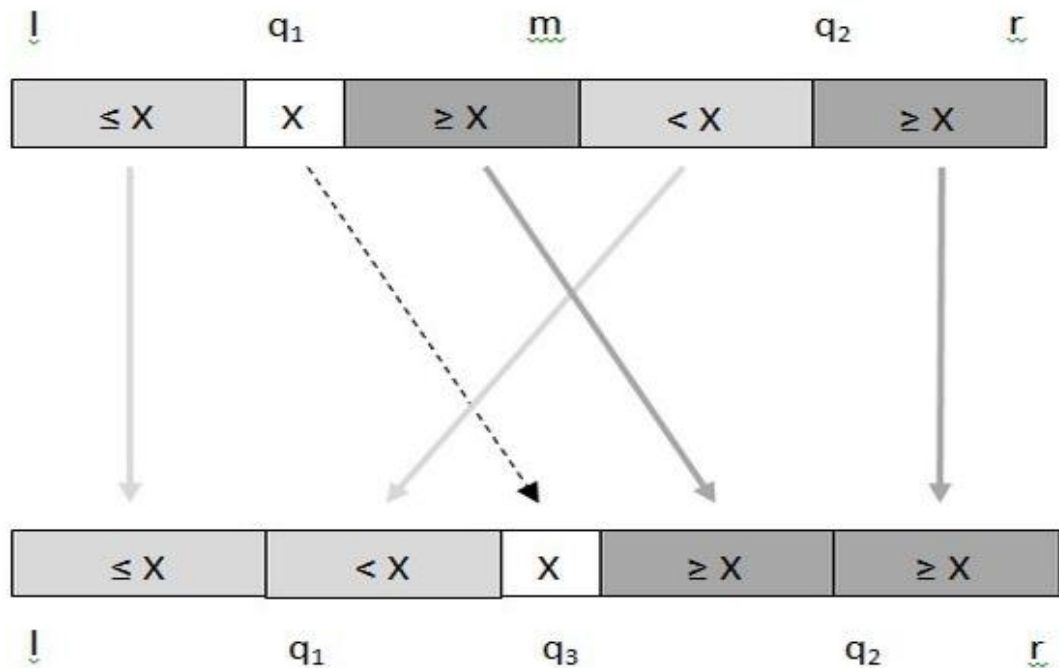
```
#include<stdlib.h>
#include<string.h>
#define MAX_SIZE 287
typedef struct hash_node
{
    char *word;
    struct hash_node *next;
}hash_node,*hash_map;
hash_map bin[MAX_SIZE]={NULL};
unsigned int get_index(char *pWord)//get hash index
{
    int len=strlen(pWord);
    int i;
    unsigned int index=1;
    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%MAX_SIZE;
}
void insert_word(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];p!=NULL;p=p->next)
        if(strcmp(p->word,pWord)==0)
            return;
    p=(hash_node*)malloc(sizeof(hash_node));
    p->word=(char*)malloc(strlen(pWord)+1);
    strcpy(p->word,pWord);
    p->word[strlen(pWord)]='\0';
    p->next=bin[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];p!=NULL;p=p->next)
    if(strcmp(pWord,p->word)!=0)
        printf("%s\t",p->word);
}
void main()
{
    char *string[]={"mary","army","ramy"};
    int len=sizeof(string)/sizeof(char*);
    int i;
    for(i=0;i<len;i++)
        insert_word(string[i]);
    char word[]="mary";
    search_brother(word);
}
```

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

首先给出原地归并排序的基本原理图：



首先把给定的数组分成两部分，前一部分包括 $[l, m]$ 中的元素，而后者则包括 $(m, r]$ 中的元素。然后找到第一个数组中的中间值，也就是 $q_1 = (l+m)/2$ ， q_1 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 $q1$ 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 $(m, r]$ 这段元素,也是分成两部分。建议在取元素范围的时候, $(m, r]$ 这段的前一部分个数为 $(q2-1-m)$ 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 $O(1)$ 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>
#include<assert.h>
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 *a, int low, int mid, int high) {
    swap(a, low, mid);
    swap(a, mid + 1, high);
    swap(a, low, high);
}

int binary_search(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 <= a[mid])
            high = mid;
        else
            low = mid + 1;
    }
    return high;
}

void merge_in_place(int *a, int low, int mid, int high) {
    int length1 = mid - low + 1;
```

```
int length2 = high - mid;
if(!(length1 >= 0 && length2 >= 0))
    return ;
if(length1 >= length2) {
    if(length2 <= 0)
        return;
    int q1 = (low + mid) / 2;
    int q2 = binary_search(a[q1], a, mid + 1, high);
    int q3 = q1 + (q2 - 1 - mid);
    block_exchange(a, q1, mid, q2 - 1);
    merge_in_place(a, low, q1 - 1, q3 - 1);
    merge_in_place(a, q3 + 1, q2 - 1, high);
} else {
    if(length1 <= 0)
        return;
    int q1 = (mid + 1 + high) / 2;
    int q2 = binary_search(a[q1], a, low, mid);
    int q3 = q2 + (q1 - 1 - mid);
    block_exchange(a, q2, mid, q1);
    merge_in_place(a, low, q2 - 1, q3 - 1);
    merge_in_place(a, q3 + 1, q1, high);
}
}
void main() {
    int a[]={1, 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("%d\t", a[i]);
    printf("\n");
}
```

23. 一个 url 指向的页面里面有另一个 url,最终有一个 url 指向之前出现过的 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 hash。

本题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, 如果存在, 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 fast 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 null, 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 第一次相遇的时候, 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 p1 和 p2 相交点, 距离环路入口的距离; fast 走的路径: $p+c+k*L = 2*n$; L 为环路的周长, k 是整数。显然, 如果从 p+c 点开始, p1 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; //建立附加头结点
    head->next = NULL;

    /*创建链表*/
    NODE *current, *previous;
```

```
previous = head;
char input;
cout << "Input your list table NODE data,end with '#' :";
cin >> input;
while(input != '#')
{
    current = new NODE;
    current->data = input;
    current->next = NULL;
    previous->next = current;
    previous = previous->next;
    cout << "Input your list table NODE data,end with '#' :";
    cin >> input;
}

/*输出链表*/
current = head->next;
while(current != NULL)
{
    cout << current->data << " ";
    current = current->next;
}
cout << endl;

/*倒转链表*/
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;

/*输出链表*/
current = head->next;
while(current != NULL)
{
```

```
        cout << current->data << " ";  
        current = current->next;  
    }  
}
```

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 P,V 操作实现.

mA, 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 P, V 操作实现。

```
semaphore empty = n  
semaphore full;  
semaphore mutex =1;  
semaphore b = 1;  
semaphore c = 1 ;  
semaphore d = 1;
```

```
A () {  
    while(true) {  
        p(empty);  
        p(b);  
        p(c);  
        p(d);  
        p(mutex);  
        write();  
        v(mutex);  
        v(full);  
    }  
}
```

```
B () {  
    while(true) {  
        p(full);  
        p(mutex);  
        write();  
        v(mutex);  
        v(empty);  
        v(b);  
    }  
}
```

```
    }  
}  
  
C() {  
    while(true) {  
        p(full);  
        p(mutex);  
        write();  
        v(mutex);  
        v(empty);  
        v(c);  
    }  
}  
  
D () {  
    while(true) {  
        p(full);  
        p(mutex);  
        write();  
        v(mutex);  
        v(empty);  
        v(d);  
    }  
}
```

26. 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，线程文本包含在他的进程的文本片段中，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量。

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，进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/寄存器组、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、都不直接访问其他无关进程或线程的资源。

27. 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。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。

28. 阅读下面代码,回答问题.

```
#include <stdio.h>
main() {
    int sum, pad, pAd;
    Sum = pad = 5;
    pAd = ++sum, pAd++, ++pad;
```

```
printf( "%d\n", pad);  
}  
输出结果是?
```

6

29. 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

30. if [\$? -a \$? = "test"]中-a 是什么意思?

并且

31. 已知 int 占 4 个字节,unsigned char 占 1 个字节,unsigned int number=0xffaabcdd; 下种方式可以将 number 的值变为 0xffaacddd?

- A. *((unsigned char*)(&number)+ 1)=0xcd;
- B. number =(number & 0xffff00ff) | 0x00cd00;
- C. number = (number & 0xffee43dd) | 0xbbaacddd;
- D. number = (number & 0xffccbcff) + 0x1100;

A, B, C

32. 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?

$O(m+n)$

KMP 字符串匹配时间复杂度一定是 $O(N)$ 线性

33. 求 0-n 之间二进制内没有连续三个 1 的数的个数.

```
#include <iostream>
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 (!limit && dp[len][flag] != -1) return dp[len][flag];
    int maxx = limit ? v[len] : 1;
    int cnt = 0;
    for (int i = 0; i <= maxx; i++){
        if (i == 0){
            cnt += dfs(len-1, 0, limit && i == v[len]);
        }
        else{
            cnt += dfs(len-1, flag+1, limit && i == v[len]);
        }
    }
    return limit ? cnt : dp[len][flag] = cnt;
}
int solve(long long x){
    memset(v, 0, sizeof(v));
    int k = 0;
    while (x){
        v[++k] = x % 2;
        x >>= 1;
    }
    return dfs(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 对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 T*对象。

当 weakCount 为 0 同时 shareCount 为 0 时，删除 Counter*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 单参数构造函数

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 * ()

SharedPtr 实现如下：

```
template<class T> class WeakPtr;
template<class T> class SharedPtr
```

```
{
public:
    friend class WeakPtr<T>; //方便 weak_ptr 与 share_ptr 设置引用计数和赋值。
```

```
    SharedPtr()
        : m_pResource(nullptr)
        , m_pCounter(new Counter())
    {
        m_pCounter->shareCount = 1;
    }
```

```
    explicit SharedPtr(T* pResource = nullptr)
        : m_pResource(pResource)
```

```
, m_pCounter(new Counter()))
{
    m_pCounter->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>& other)
    : m_pResource(other->m_pResource)
    , m_pCounter(other->m_pCounter)
{
    ++(m_pCounter->shareCount); // 增加引用计数
}

SharedPtr<T>& operator = (const SharedPtr<T>& other)
{
    if (this == &other) return *this;

    release();
    m_pCounter = other.m_pCounter;
    m_pResource = other.m_pResource;
    ++(m_pCounter->shareCount); // 增加引用计数

    return *this;
}

~SharedPtr()
{
    release();
}

T& operator bool()
{
    return m_pResource != nullptr;
}

T& operator * ()
{
    // 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, Counter*如果没有 WeakPtr, 也由 SharedPtr
        释放
        --m_pCounter->shareCount;

        if (0 == m_pCounter->shareCount)
        {
            delete m_pResource;
            m_pResource = nullptr;

            if (0 == m_pCounter->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 SharedPtr&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 SharePtr, 或者已被析构, 返回指向 nullptr 的 SharePtr

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

release() 函数

WeakPtr 实现如下：

```
template<class T> class WeakPtr
{
public:
    friend class SharedPtr<T>; //方便 weak_ptr 与 share_ptr 设置引用计数和赋值。

    WeakPtr()
        : m_pResource(nullptr)
        , m_pCounter(new Counter())
    {
        m_pCounter->weakCount = 1;
    }

    WeakPtr(SharedPtr<T>& other)
        : m_pResource(other.m_pResource)
        , m_pCounter(other.m_pCounter)
    {
        ++(m_pCounter->weakCount);
    }

    WeakPtr(WeakPtr<T>& other)
        : m_pResource(other.m_pResource)
        , m_pCounter(other.m_pCounter)
    {
        ++(m_pCounter->weakCount);
    }

    WeakPtr<T>& operator = (WeakPtr<T>& other)
    {
        if (this == &other) return *this;
        release();
        m_pCounter = other.m_pCounter;
        m_pResource = other.m_pResource;
        ++m_pCounter->weakCount;
        return *this;
    }

    WeakPtr<T>& operator =(SharedPtr<T>& other)
    {
        release();
        m_pCounter = other.m_pCounter;
        m_pResource = other.m_pCounter;
        ++m_pCounter->weakCount; // 增加弱引用计数
    }
};
```

```
        return *this;
    }

    ~WeakPtr()
    {
        release();
    }

    SharedPtr<T> lock()
    {
        return SharedPtr<T>(*this);
    }

    bool expired()
    {
        if (m_pCounter != nullptr && m_pCounter->shareCount != 0)
            return false;

        return true;
    }

private:
    void release()
    {
        --m_pCounter->weakCount;
        if (0 == m_pCounter->weakCount && 0 == m_pCounter->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++11 最常用的新特性如下：

auto 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: nullptr 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 一般被宏定义为 0, 在遇到重载时可能会出现问题。

智能指针: C++11 新增了 std::shared_ptr、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

新增 STL 容器 array 以及 tuple

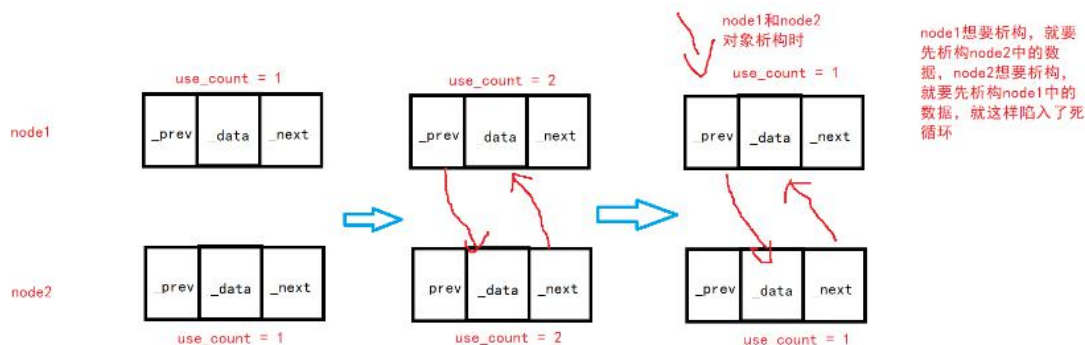
36. 智能指针是线程安全的吗? 哪些地方需要考虑线程安全?

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++或者--, 这个操作不是原子的, 引用计数原来是 1, ++了两次, 可能还是 2, 这样引用计数就乱了, 有可能造成资源未释放或者程序崩溃的风险。所以说智能指针中++或--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::shared_ptr 循环引用
struct ListNode
{
    int _data;
    shared_ptr<ListNode> _prev;
    shared_ptr<ListNode> _next;
    ~ListNode() { cout << "~ListNode()" << endl; }
};
int main()
{
    shared_ptr<ListNode> node1(new ListNode);
    shared_ptr<ListNode> node2(new ListNode);
    cout << node1.use_count() << endl;
    cout << node2.use_count() << endl;
    node1->_next = node2;
    node2->_prev = node1;
    cout << node1.use_count() << endl;
    cout << node2.use_count() << endl;
    return 0;
}
```

node1 和 node2 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

node1 的 _next 指向 node2, node2 的 _prev 指向 node1, 引用计数变成 2
node1 和 node2 析构, 引用计数减到 1, 但是 _next 还指向下一个节点, _prev 指向上一个节点
也就是说 _next 析构了, node2 释放了
也就是说 _prev 析构了, node1 释放了
但是 _next 属于 node 的成员, node1 释放了, _next 才会析构, 而 node1 由 _prev 管理, _prev 属于 node2 成员, 所以这就叫循环引用, 谁都不会释放



解决方案

在引用计数的场景下, 把 shared_ptr 换成 weak_ptr 就可以了
原理就是, node1->_next = node2; 和 node2->_prev = node1; 时 weak_ptr 的 _next 和 _prev 不会增加 node1 和 node2 的引用计数

```
struct ListNode
{
    int _data;
    weak_ptr<ListNode> _prev;
    weak_ptr<ListNode> _next;
    ~ListNode() {
        cout << "~ListNode()" << endl;
    }
};

int main()
{
    shared_ptr<ListNode> node1(new ListNode);
    shared_ptr<ListNode> node2(new ListNode);
    cout << node1.use_count() << endl;
    cout << node2.use_count() << endl;
    node1->_next = node2;
    node2->_prev = node1;
    cout << node1.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<class T>
struct FreeFunc {
    void operator() (T* ptr)
    {
        cout << "free:" << ptr << endl;
        free(ptr);
    }
};

template<class T>
struct DeleteArrayFunc {
    void operator() (T* ptr)
    {
        cout << "delete[]" << ptr << endl;
        delete[] ptr;
    }
};

int main()
{
    FreeFunc<int> freeFunc;
    shared_ptr<int> sp1((int*)malloc(4), freeFunc);
    DeleteArrayFunc<int> deleteArrayFunc;
    shared_ptr<int> sp2((int*)malloc(4), deleteArrayFunc);
    return 0;
}
```

37. 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储单元，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。

38. 决策树是如何解决过拟合问题的？

一. 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；（什么是噪音数据？）

(2) 样本抽取错误，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剪的话，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（主要）：

剪枝：提前停止树的生长或者对已经生成的树按照一定的规则进行后剪枝。

剪枝的方法

剪枝是一个简化过拟合决策树的过程。有两种常用的剪枝方法：

(1) 先剪枝 (prepruning)：通过提前停止树的构建而对树“剪枝”，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；

先剪枝的方法

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；

2. 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；

3. 定义一个阈值，当达到某个结点的实例个数小于该阈值时就可以停止决策树的生长；

4. 定义一个阈值，通过计算每次扩张对系统性能的增益，并比较增益值与该阈值的大小来决定是否停止决策树的生长。

(2) 后剪枝 (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

最频繁的分类标记。后剪枝的剪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，是指剪枝过程中，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。相比于先剪枝，这种方法更常用，正是因为先剪枝方法中精确地估计何时停止树增长很困难。

后剪枝的方法

1) REP 方法是一种比较简单后剪枝的方法，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剪这个决策树的影响。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验。

该剪枝方法考虑将树上的每个节点作为修剪的候选对象，决定是否修剪这个节点有如下步骤组成：

- 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: 当修剪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度的结点，直到进一步修剪有害为止 (有害是指修剪会减低验证集合的精度)。

REP 是最简单的后剪枝方法之一，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剪。这是因为一些不会再测试集中出现的很稀少的训练集实例所对应的分枝在剪枝过如果训练集较小，通常不考虑采用 REP 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剪枝算法的性能。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 剪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, 悲观错误剪枝, 悲观错误剪枝法是根据剪枝前后的错误率来判定子树的修剪。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自动对实例的某个部分进行错误的分类。它不需要像 REP (错误率降低修剪) 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剪枝。决策树生成和剪枝都使用训练集，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 $(E+0.5)/N$ 。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：

$$(\sum E_i + 0.5 * L) / \sum N_i$$

剪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 $J+0.5$ 。那么子树是否可以被剪枝就取决于剪枝后的错误 $J+0.5$ 在

$$\sum E_i + 0.5 * L$$

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

$$E(\text{subtree_err_count}) = N * e$$
$$\text{var}(\text{subtree_err_count}) = \sqrt{N * e * (1 - e)}$$

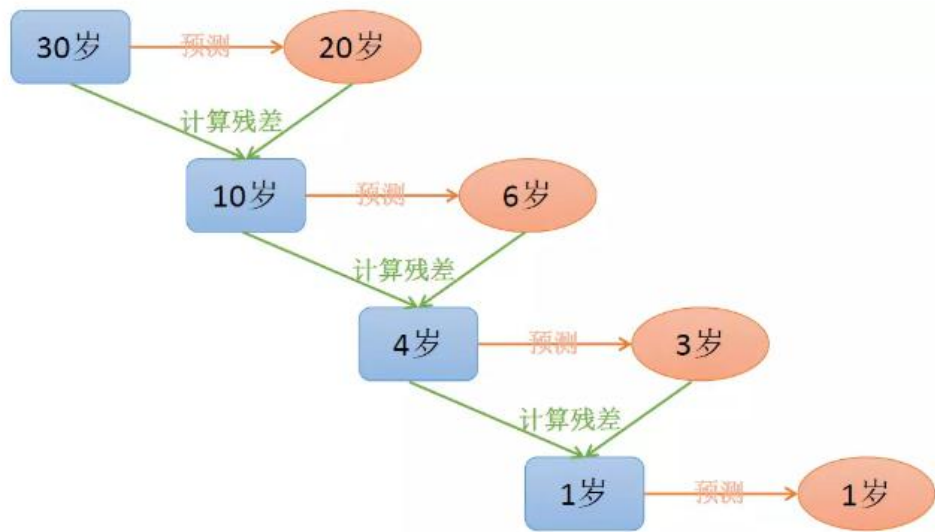
使用训练数据, 子树总是比替换为一个叶节点后产生的误差小, 但是使用校正的误差计算方法却并非如此。剪枝的条件: 当子树的误判个数大过对应叶节点的误判个数一个标准差之后, 就决定剪枝:

$$E(\text{subtree_err_count}) - \text{var}(\text{subtree_err_count}) > E(\text{leaf_err_count})$$

这个条件就是剪枝的标准。当然并不一定要大一个标准差, 可以给定任意的置信区间, 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。

39. 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 AdaBoost 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：

$$f_M(x) = \sum_{m=1}^M T(x; \Theta_m)$$

其中， $T(x; \Theta_m)$ 表示决策树； Θ_m 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 $f_0(x) = 0$ ，第 m 步的模型是：

$$f_m(x) = f_{m-1}(x) + T(x; \Theta_m)$$

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）

$$\hat{\Theta}_m = \arg \min_{\Theta_m} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m))$$

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 $T = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ ，如果将训练集分为不同的区域 R_1, R_2, \dots, R_n ，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：

$$T(x; \Theta) = \sum_{j=1}^J c_j I(x \in R_j)$$

然后通过下图方式来确定具体分割点:

$$\min_s \left[\min_{c_1} \sum_{x_i \in R_1} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2} (y_i - c_2)^2 \right]$$

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
typedef 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 lpFunc 的函数指针
void GetCallBack(void *lpVoid, lpFunc 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(lpVoid, name, age);
}

class A {
public:
    A() {} ;
    void outName(char szAlarm[], int age) {
        printf("My name is %s, %d years old \n", szAlarm, age);
    }
    //定义一个类 A 的静态成员函数 fCallback
    static void fCallback(void *lpVoid, char szAlarm[], 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 lpVoid, 强制类型转换为: A*
        A *p = (A*) (lpVoid);
```



```
//A 类型的对象指针 p 调用 A 类的成员函数 outName
p->outName(szAlarm, age);
}
//A 类的成员函数 Test
void Test() {
    //在类 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.Test();
}
```

41. Linux 日志文件统计某几个字符串，如何一条命令就能统计出来？

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`
2. 使用 grep
`grep -o 'objStr1\|objStr2' filename|wc -l` #直接用\| 链接起来即可

42. 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 `-l` 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 -l 300 -d --verbose 文件名 split-line`

2. 根据大小切割，通过 `-b` 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、ls 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
- 26、grep 文本搜索命令
- 18、chmod 访问权限
- 19、tar 压缩和解压
- 20、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、ln 命令
- 24、date 显示时间
- 25、cal 命令
- 27、wc 命令
- 28、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、ifconfig 查看 ip 地址
- 34、用户相关
 - useradd oldboy #添加用户
 - passwd redhat #设置密码
 - whoami #当前用户
 - su - oldboy #切换用户
 - logout #退出用户登录
- 35、权限相关
 - r #read 可读, 可以用 cat 等命令查看
 - w #write 写入, 可以编辑或者删除这个文件
 - x #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, 清空原文件所有内容, 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

40、| 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)、一致性 (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。

2. 一致性

一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）：数据库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）都必须是正确的。业务的一致性一般由开发人员进行保证，亦可转移至数据库层面。

3. 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库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

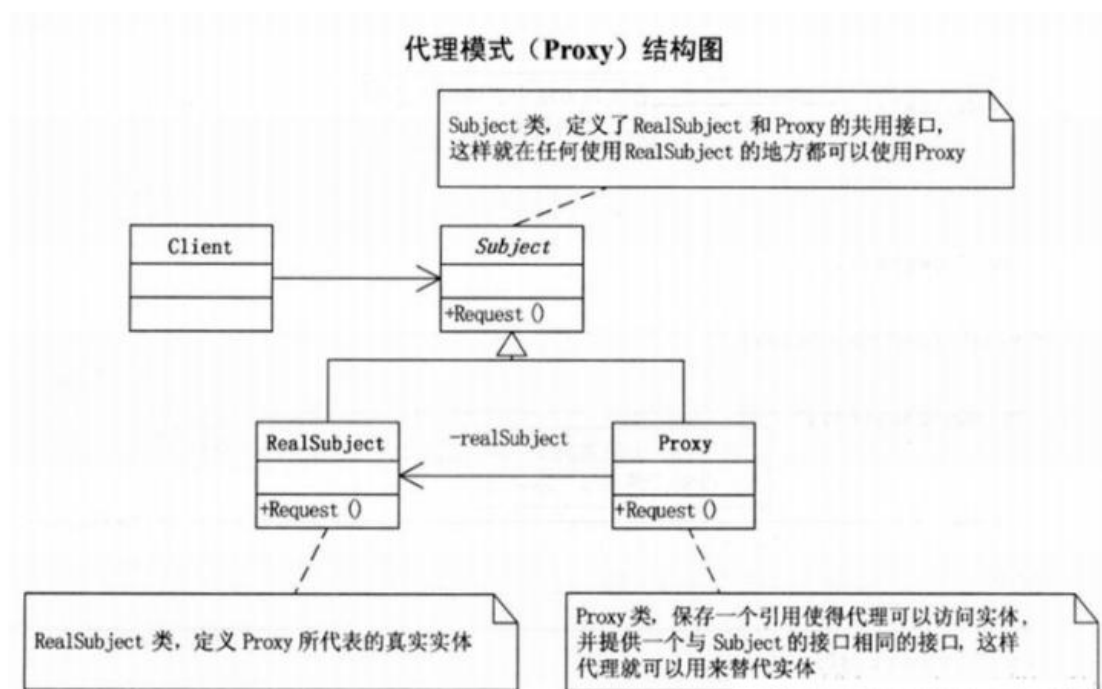
4. 持久性

这是最好理解的一个特性：持久性，意味着在事务完成以后，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。

45. 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。



46. 什么是开闭原则.

开闭原则 (OCP) 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。

开闭原则中“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；

开闭原则中“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。

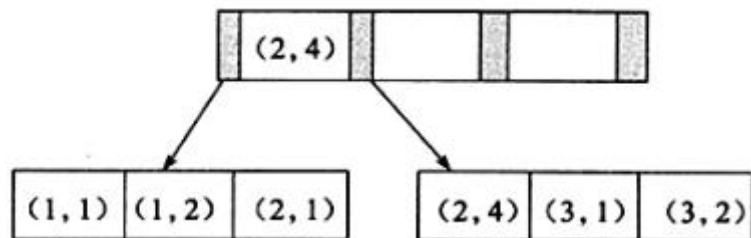
47. mysql 联合索引的原理?

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



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 (1, 1), (1, 2), (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, 因为索引本身在叶子节点已经排序了。

48. 数据库查询慢的原因?

1. 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时, 数据库会立刻将内存的数据页上的信息更新, 但是不会立刻将更新的数据存到磁盘上, 而是先保存到 redo log 中, 等合适的时机在将 redo 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

2. 一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, 避免用 `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种情况

1) 没有建索引,

2) 索引失效

引起索引失效的可能原因

(1) 在索引列上用了内置函数或者其他 `++*/` 运算

(2) 用通配符开头

(3) 多列索引违背最佳最匹配原则

(4) `or` 操作符容器造成索引失效, 除非 `or` 的每个操作列都有索引

(5) 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, 所以单独拿出来分析。

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。

49. 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 $n \log n$.

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;
```

```
//获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) // 交换后继续扫描
    {
        while (i < j && arr[j] > pivot) j--; // 右侧开始查找比基准元素更小的值
        while (i < j && arr[i] <= pivot) i++; // 左侧查找比基准元素大于等于的值
        if (i < j)
        {
            swap(arr[i++], arr[j--]); // 先交换后赋值
        }
    }
    if (arr[i] > pivot)
    {
        swap(arr[i - 1], arr[low]); // 在代码中提供的示例中，在遇到第一次扫描
        // 时交换 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;
}

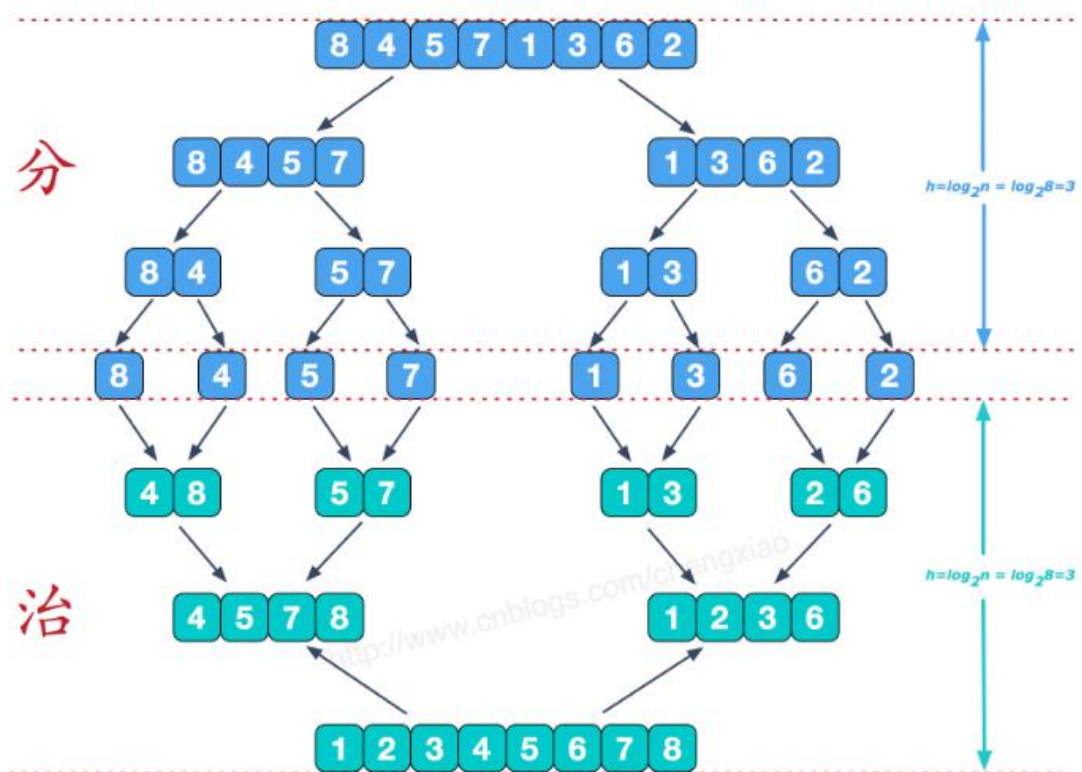
// 快速排序函数
void QuickSort(int arr[], int low, int high)
{
    int mid;
    if (low < high)
    {
        // 递归快排
        mid = GetMid(arr, low, high);
        QuickSort(arr, mid + 1, high); // 右区间快排
        QuickSort(arr, low, mid - 1); // 左区间快排
    }
}

int main()
{
    int N = 9;
    int arr[9] = { 30, 24, 5, 58, 37, 36, 12, 42, 39 };
    for (int i = 0; i < N; i++)
    {
        cout << arr[i] << ", ";
    }
    cout << endl;
    QuickSort(arr, 0, N-1);
    for (int i = 0; i < N; i++)
```

```
{  
    cout << arr[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,然后不断的递归子序列合并。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)
```

```
{
```

```
    //1. 申请与 arr 等长的数组 B
```

```
    int *B = new int[high - low + 1];
```

```
    int i = low, j = mid + 1, k = 0;
```

```
    //2. 将 arr 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 (arr[i] <= arr[j])
    {
        B[k++] = arr[i++];
    }
    else
    {
        B[k++] = arr[j++];
    }
}
//3. 将 arr 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++] = arr[i++]; }
while (j <= high){ B[k++] = arr[j++]; }
//4. 将 B 数组复制到 arr 数组, 释放 B
for (i = low, k = 0; i <= high; i++)
{
    arr[i] = B[k++];
}
delete []B;
}
```

```
//合并排序
void MergeSort(int arr[],int low,int high)
{
    //递归合并排序
    if (low < high)
    {
        int mid = (low + high) / 2;
        MergeSort(arr, low, mid);
        MergeSort(arr, mid + 1, high);
        Merge(arr, low, mid, high);
    }
}
```

```
int main()
{
    int N = 8;
    int a[8] = { 42, 15, 20, 6, 8, 38, 50, 12 };
    MergeSort(a, 0, N-1);
    for (size_t 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 $O(n\log^n)$ 。合并排序的二叉树高度为 \log_2^n , 每一层都是 n 个元素进行比较, 所以总的时间复杂度为 $O(n\log^n)$ 。


50. 如何划分子网?

一个 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 256 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 32-8=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加群: 762073882





 面试分享.mp4


 TCPIP协议栈，一次课开启你的网络之门.mp4




 高性能服务器为什么需要内存池.mp4


 手把手写线程池.mp4


 reactor设计和线程池实现高并发服务.mp4


 nginx源码—线程池的实现.mp4


 MySQL的块数据操作.mp4


 高并发 tcpip 网络io.mp4


 去中心化，p2p，网络穿透一起搞定.mp4

 服务器性能优化 — 异步的效率.mp4

 区块链的底层，去中心化网络的设计.mp4

 深入浅出UDP传输原理及数据分片方法.mp4

 线程那些事.mp4

 后台服务进程挂了怎么办.mp4

