

# 目錄

说明	1.1
C/C++ 问答题汇总	1.2
编程题汇总	1.3
腾讯 2017 暑期实习生编程题	1.3.1
构造回文	1.3.1.1
算法基础-字符移位	1.3.1.2
有趣的数字	1.3.1.3
剑指 Offer 编程题练习	1.3.2
二维数组中的查找	1.3.2.1
从尾到头打印链表	1.3.2.2
旋转数组的最小数字	1.3.2.3
替换空格	1.3.2.4
用两个栈实现队列	1.3.2.5
重建二叉树	1.3.2.6
网易 2017 校招笔试编程题	1.3.3
二进制权重	1.3.3.1
平方串	1.3.3.2
数位和	1.3.3.3
软件测试题汇总	1.4
计算机知识题汇总	1.5
前 75 题	1.5.1
75 题以后	1.5.2
Python 语言特性	1.6
1 Python 的函数参数传递	1.6.1
2 @staticmethod 和 @classmethod	1.6.2
3 类变量和实例变量	1.6.3
4 Python 中单下划线和双下划线	1.6.4

5 args and *kwargs	1.6.5
6 面向切面编程AOP和装饰器	1.6.6
7 单例模式	1.6.7
8 Python 函数式编程	1.6.8
9 Python 里的拷贝	1.6.9
stackoverflow-about-Python	1.7
Python中关键字yield有什么作用?	1.7.1
Python中的元类(metaclass)是什么?	1.7.2
Python中如何在一个函数中加入多个装饰器?	1.7.3
如何判断一个文件是否存在?	1.7.4
如何调用外部命令?	1.7.5
枚举类型的使用?	1.7.6
在Windows下安装pip?	1.7.7
字典合并问题	1.7.8
在Android上运行Python	1.7.9
根据字典值进行排序	1.7.10
在函数里使用全局变量	1.7.11
变化的默认参数	1.7.12
装饰器classmethod和staticmethod的区别	1.7.13
检查列表为空的最好办法	1.7.14
怎么用引用来改变一个变量	1.7.15
检查文件夹是否存在的操作	1.7.16
name=main的作用	1.7.17
super与init方法	1.7.18
str与repr的区别	1.7.19
循环中获取索引	1.7.20
类里的静态变量	1.7.21
怎么在终端里输出颜色?	1.7.22
为什么用pip比用easy_install的好?	1.7.23
join的问题	1.7.24

---

把列表分割成同样大小的块	1.7.25
为什么代码在一个函数里运行的更快	1.7.26
如何快速合并列表中的列表？	1.7.27
如何知道一个对象有一个特定的属性？	1.7.28
如何通过函数名的字符串来调用这个函数？	1.7.29
单下划线和双下划线的含义？	1.7.30
<b>Python 面试编程题</b>	<b>1.8</b>
1 台阶问题 斐波那契	1.8.1
2 变态台阶问题	1.8.2
3 矩形覆盖	1.8.3
4 杨氏矩阵查找	1.8.4
5 去除列表中的重复元素	1.8.5
6 链表成对调换	1.8.6
7 创建字典的方法	1.8.7
8 合并两个有序列表	1.8.8
9 二分查找	1.8.9
10 快排	1.8.10
11 找零问题	1.8.11
12 二叉树相关	1.8.12
13 单链表逆置	1.8.13

---

# Python && C/C++ 面试

整理自己准备 Python && C/C++ 面试时的相关资料

[GitBook 链接](#)

【PS: 突然发现 Python Cookbook 很适合拿来当考题, 以后我要是面试别人就用这个出题, ♪(=。=)嘿咻】 [Python CookBook](#)

## 资料来源

- [牛客网](#)
- [GitHub Python 面试题](#)
- [GitHub stackoverflow-about-Python](#)
- [Python CookBook](#)

## 题目

### 题目 1

如果友元函数重载一个运算符时，其参数表中没有任何参数则说明该运算符是：

- 一元运算符
- 二元运算符
- 选项A) 和选项B) 都可能
- 重载错误

### 正确答案及解析

答案为: **D**

友元函数重载时，参数列表为 1，说明是 1 元，为 2 说明是 2 元

成员函数重载时，参数列表为空，是一元，参数列表是 1，为 2 元

### 题目 2

关于 do 循环体 while(条件表达式)，以下叙述中正确的是？

- 条件表达式的执行次数总是比循环体的执行次数多一次
- 循环体的执行次数总是比条件表达式执行次数多一次
- 条件表达式的执行次数与循环体的执行次数一样
- 条件表达式的执行次数与循环体的执行次数无关

### 正确答案及解析

答案为: **D**

可能一次都没执行条件，可能执行相同的次数(参考 break/continue 等其他跳出循环的语句)

## 题目 3

下面有关 `static` 类变量和实例变量的描述，正确的有？

- `static` 类变量又叫静态成员变量，它不需要创建对象就可以已经在内存中存在了
- 在创建实例对象的时候，内存中会为每一个实例对象的每一个非静态成员变量开辟一段内存空间，用来存储这个对象所有的非静态成员变量值
- `static` 类变量是所有对象共有，其中一个对象将它值改变，其他对象得到的就是改变后的结果
- 实例变量则属对象私有，某一个对象将其值改变，不影响其他对象

## 正确答案及解析

答案为: **ABCD**

A : `static` 变量在未初始化时存储在 BSS 段，初始化后存储在 data section 数据段，A 正确

B : 静态成员则不会开辟空间，B 正确

C : `static` 变量是类变量，可理解为只有一份，C 正确

D : 可理解为：对实例对象，每个实例均有各自的一份变量，改变其值只是改变了自己的那一份，D 正确

## 题目 4

符号 `-`、`*`、`$` 分别代表减法、乘法和指数运算，且 a) 三个运算符优先级顺序为：`-` 最高，`*` 其次，`$` 最低； b) 运算符运算时为左结合，则 `5-3*2$2*4-3$2` 的结果为。

## 正确答案及解析

答案为: **256**

S1: 因为减号的优先级高，先算减法：

$$\begin{aligned} & (5 - 3) * 2 \$ 2 * (4 - 3) \$ 2 \\ &= 2 * 2 \$ 2 * 1 \$ 2 \end{aligned}$$

S2：下一步算乘法：

$$\begin{aligned} & (2 * 2) \$ (2 * 1) \$ 2 \\ &= 4 \$ 2 \$ 2 \end{aligned}$$

S3：最后求幂

$$\begin{aligned} & 4 \$ 2 \$ 2 \\ &= 256 \end{aligned}$$

## 题目 5

空格处应填写（）。

```
#include <iostream>
using namespace std;

class A
{
public:
    int m;
    int* p;
};

int main()
{
    A s;
    s.m = 10;
    cout<<s.m<<endl; //10
    s.p = &s.m;
    () = 5;
    cout<<s.m<<endl; //5
    return 0;
}
```

## 正确答案及解析

$*(s.p) = 5$

## 题目 6

下面程序的执行结果：

```
class A{
public:
    long a;
};

class B : public A {
public:
    long b;
};

void seta(A* data, int idx) {
    data[idx].a = 2;
}

int main(int argc, char *argv[]) {
    B data[4];
    for(int i=0; i<4; ++i){
        data[i].a = 1;
        data[i].b = 1;
        seta(data, i);
    }
    for(int i=0; i<4; ++i){
        std::cout << data[i].a << data[i].b;
    }
    return 0;
}
```

## 正确答案及解析

22221111

这道题应该注意 指针类型加减时步长的问题。

A 大小为 4

B 大小为 8

那么 : void seta(A\* data, int idx) { data[idx].a = 2; }

由于传入的实参为 B 类型，大小为 8，而形参为 A 类型，大小为 4

data[idx] 取 data + idx 处的元素，这时指针 data 加 1 的长度不是一个 B 长度，而是一个 A 长度，或者说是 1/2 个 B 长度。

这时该函数中 data[0~3] 指向的是原 data[0].a,data[0].b,data[1].a,data[1].b,

由于隐式类型转换的缘故，data[0].a, data[0].b, data[1].a, data[1].b 处的值全部由于 data[idx].a = 2; 操作变为 2。

这道题如果改为 void seta(B\* data, int idx)，那么形参中 data 指针加1步长为 8，结果就是 21212121。但是由于步长为 4，所以结果就是 22221111。

## 题目 7

下列代码试图打印数字1-9的全排列组合。

```
#include "stdio.h"
#define N 9
int x[N];
int count = 0;

void dump() {
    int i = 0;
    for (i = 0; i < N; i++) {
        printf("%d", x[i]);
    }
    printf("\n");
}

void swap(int a, int b) {
    int t = x[a];
    x[a] = x[b];
    x[b] = t;
}

void run(int n) {
    int i;
```

```

if (N - 1 == n) {
    dump();
    count++;
    return;
}
for (i = ____; i < N; i++) {
    swap(___, i);
    run(n + 1);
    swap(___, i);
}
}

int main() {
    int i;
    for (i = 0; i < N; i++) {
        x[i] = i + 1;
    }
    run(0);
    printf("* Total: %d\n", count);
}

```

其中 `run` 函数中缺失的部分应该依次为：

## 正确答案及解析

n, n, n

这是一道分治算法题。这种循环套递归的题目是很难一下子理解的，因此可以把问题的规模减小。

先试 3 个元素，然后我们发现，在循环里面第一句话是那当前的某个数和后面的某个数交换（包括和自己交换，也就是不交换），交换完了之后，后面那个递归就是分治了，也就是从待交换的数的下一个开始直到末尾的一段调用递归用分治搞定。

分治一直调用到最后无法交换的时候，也就是末尾两个指针相遇的时候程序就输出一种排列。所以在递归退出之后，根据程序的逻辑，在当前层循环里面应该只负责当前数和后面的数的交换，而当前数不能变，所以要把现场恢复，因此还需要调用一次 `swap` 再交换回来就可以了。所以根据程序逻辑，应该选择 C。自己画一个三个数的排列就可以看明白了，9 个数太复杂，搞明白关系就可以了。

## 题目 8

下面有关java和c++的描述，错误的是？

- java 是一次编写多处运行，c++ 是一次编写多处编译
- c++ 和 java 支持多重继承
- Java 不支持操作符重载，操作符重载被认为是 c++ 的突出特征
- java 没有函数指针机制，c++ 支持函数指针

## 正确答案及解析

B

JAVA没有指针的概念，被封装起来了，而C++有；

JAVA不支持类的多继承，但支持接口多继承，C++支持类的多继承；

C++支持操作符重载，JAVA不支持；

JAVA的内存管理比C++方便，而且错误处理也比较好；C++的速度比JAVA快。

C++更适用于有运行效率要求的情况，JAVA适用于效率要求不高，但维护性要好的情况。

## 题目 9

下列选项中，会导致用户进程从用户态切换到内核的操作是？

1. 整数除以零
2. sin( )函数调用
3. read系统调用

## 正确答案及解析

仅 I、III

用户态切换到内核态的 3 种方式

- 系统调用
- 异常

- 外围设备的中断

## I. 异常

## III. 系统调用

## 题目 10

```
#include<iostream>
using namespace std;
class MyClass
{
public:
    MyClass(int i = 0)
    {
        cout << i;
    }
    MyClass(const MyClass &x)
    {
        cout << 2;
    }
    MyClass &operator=(const MyClass &x)
    {
        cout << 3;
        return *this;
    }
    ~MyClass()
    {
        cout << 4;
    }
};
int main()
{
    MyClass obj1(1), obj2(2);
    MyClass obj3 = obj1;
    return 0;
}
```

运行时的输出结果是（）

## 正确答案及解析

### 122444

C MyClass obj3 = obj1;

obj3还不存在，所以调用拷贝构造函数输出2，

如果obj3存在，obj3=obj， 则调用复制运算符重载函数，输出3

首先程序中存在三个MyClass对象。

前两个对象构造时分别输出1,2

第三个对象是这样构造的MyClass obj3 = obj1;这里会调用拷贝构造函数，输出2

然后三个对象依次析构，输出444

所以最终输出122444

## 题目 11

In C++, what does "explicit" mean? what does "protected" mean?

- explicit-keyword enforces only explicit casts to be valid
- Protected members are accessible in the class that defines them and in classes that inherit from that class.
- Protected members only accessible within the class defining them.
- All the above are wrong

## 正确答案及解析

AB

普通构造函数能够被隐式调用. 而 explicit 构造函数只能被显式调用

## 题目 12

struct 和 class 的区别?

- struct 的成员默认是公有的

- struct 的成员默认是私有的
- 类的成员默认是私有的
- 类的成员默认是公有的

## 正确答案及解析

AC

## 题目 13

下面描述中，正确的是

- 虚函数是没有实现的函数
- 纯虚函数的实现是在派生类中
- 抽象类是没有纯虚函数的类
- 抽象类指针可以指向不同的派生类

## 正确答案及解析

BD

用关键字virtual修饰的成员函数叫做虚函数，虚函数是为了实现多态而存在的，必须有函数体

纯虚函数的声明，是在虚函数声明的结尾加=0，没有函数体。在派生类中没有重新定义虚函数之前是不能调用的

如果一个类中至少含有一个纯虚函数，此时称之为抽象类。所以抽象类一定有纯虚函数

基类类型的指针可以指向任何基类对象或派生类对象

## 已整理清单

- 腾讯 2017 暑期实习生编程题
  - 构造回文
  - 算法基础-字符移位
  - 有趣的数字
- 剑指 Offer 编程题练习
  - 二维数组中的查找
  - 从尾到头打印链表
  - 旋转数组的最小数字
  - 替换空格
  - 用两个栈实现队列
  - 重建二叉树
- 网易 2017 校招笔试编程题
  - 二进制权重
  - 平方串
  - 数位和

# 腾讯 2017 暑期实习生编程题

- 构造回文
- 算法基础-字符移位

# 构造回文

## 题目描述

给定一个字符串  $s$ ，你可以从中删除一些字符，使得剩下的串是一个回文串。如何删除才能使得回文串最长呢？输出需要删除的字符个数。

输入描述：输入数据有多组，每组包含一个字符串  $s$ ，且保证  $1 \leq s.length \leq 1000$ 。

输出描述：对于每组数据，输出一个整数，代表最少需要删除的字符个数。

输入例子：

abcdagoole

输出例子：

22

## 参考资料

[删除最少字符使字符串成为回文串](#)

## 分析

对于这题来说，插入字符和删除字符使其成为回文串，答案是一样的。首先求  $s$  的反串  $rs$ ，然后对  $s$  和  $rs$  求最长公共子序列，要删除的字符个数就是 LCS。

## Python 解答

自己用 Python 写了，虽然结果正确，但是超时了。思路是用动态规划。

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""

# 题目描述
给定一个字符串 s，你可以从中删除一些字符，使得剩下的串是一个回文串。如何删除才
```

能使得回文串最长呢？输出需要删除的字符个数。

## # 参考资料

[http://www voidcn com/blog/Do\\_Know/article/p-5978576.html](http://www voidcn com/blog/Do_Know/article/p-5978576.html)

## # 分析

对于这题来说，插入字符和删除字符使其成为回文串，答案是一样的。

首先求s的反串rs，然后对s和rs求最长公共子序列，要删除的字符个数就是 LCS.

"""

```
__author__ = '__L1n_w@tch'

MAX_LENGTH = 1010 # 1 <= s <= 1000
```

## # 答案用动态规划解决的

```
def dynamic_solve(s):
    """
```

动态规划求取构造回文要删除的字符个数  
 $dp[i][j]$  表示  $s_1$  的前  $i$  个字符串， $s_2$  的后  $j$  个字符串中的最大公共字符串

动态方程：

```
 $dp[i + 1][j + 1] = dp[i][j] + 1 \text{ if } s_1[i] == s_2[j] \text{ else } \max(dp[i][j + 1], dp[i + 1][j])$ 
:param s: 原始字符串
:return: 需要删除的字符个数
"""
```

```
# 先求反串
length = len(s)
reversed_s = s[::-1]
```

```
# 求最长公共子序列
dp = [[0 for i in range(MAX_LENGTH)] for i in range(MAX_LENGTH)] # 初始化
```

```
for i in range(length):
    for j in range(length):
        if s[i] == reversed_s[j]:
```

```

        dp[i + 1][j + 1] = dp[i][j] + 1
    else:
        dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j])

    return length - dp[length][length] # 长度减去最长公共子序列即为
    所求

if __name__ == "__main__":
    tests = ["", "a", "ab", "abcda", "google",
        "zgtklhfzomzjckwmluvivvcmhjrkwkvcjrxojobpdedpamds
        hcwwsetfbacvonecrdvugeibglvhxuymjvoryqjwullvzglqazxr
        dmczyvbgakjagttrezmvrlptiwoqkrtxuroeqmryzsgokopxxdpbej
        mtwvpnaqrgqladdszhdwxfckmehdvihgvacueqhvwvjxoitlpfr
        ckxkuksaqzjpwgoldyhugsacf1cdqhf1doaphgdbhaciixouav
        qxwlghadmfortqacbfffqzocinvuqpjthgekunjsstukeiffji
        pzzabkuiueqnjgkuiojwbjzfynafnlcaryyqjqfixaoeowh
        kxkbsnpsvnbxuywfxbnuoemxynbtgkqtjvzqikbafjnpbeirxx
        rohhnjqrbbqqzercqcrcswojyylunuevt
        tdhamlkzqnjrzibwckbkiygyusuaxpjrgjmurrohkhvjmpwmmt
        pcszpihcntyivrjplhyrqftghglkvqeidyhtmr1cljngeyaef
        fxnywpfsualufjwnffyqnpitgkkyrbwccqggycrvoocbw
        sdbftkigrkcbojuwwctknzzmvhbhbfzrqwzllulbabztqnznkq
        dyoqnrxhwavqhzyzvmmmphzxbikpharseywpfsqyybkynwbdrgf
        saxduxojcdqcjuaaywzbvdjgjqtoffasiuhvxcaockebkuxpi
        omqmrvsqhnyxfjceqevqvnapbk"]

    for each_test in tests:
        print(dynamic_solve(each_test))

```

## C 语言版本 1

自己用 C 又重写了一遍，期望能通过，结果发现提示说答案错，同样的数据，在网上测试的时候结果错了，在本机测试的时候自己出来的答案是对的。说明不是代码问题，而是题目所给字符串的输入问题，尝试用文件读入数据解决结果发现还是没错，所以就不知道了。以下是代码：

```

// 
// Created by w@tch on 16/7/21.
// 

#include <stdio.h>

```

```

#include <string.h>

#define MAX_LENGTH 1001

char *strrev(char *s) {
    if (s == NULL || s[0] == '\0')
        return s;

    for (char t, *p = s, *q = s + strlen(s) - 1; p < q; p++, q--)
    {
        t = *p;
        *p = *q;
        *q = t;
    }

    return s;
}

int dynamic_solve(char *s) {
    // 先求反串
    char *reverse_s = new char(strlen(s) * sizeof(char));
    strcpy(reverse_s, s);
    strrev(reverse_s);

    // 初始化
    int length = strlen(s);
    int i, j;
    int dp[MAX_LENGTH][MAX_LENGTH] = {0};

    // 求最长公共子序列
    for (i = 0; i < length; ++i) {
        for (j = 0; j < length; ++j) {
            if (s[i] == reverse_s[j]) {
                dp[i + 1][j + 1] = dp[i][j] + 1;
            }
            else {
                dp[i + 1][j + 1] = dp[i][j + 1] > dp[i + 1][j] ?
                    dp[i][j + 1] : dp[i + 1][j];
            }
        }
    }
}

```

```

    }

    return length - dp[length][length]; // 长度减去最长公共子序列即
为所求
}

int main(void) {
//    FILE *f = fopen("/Users/L1n/Desktop/C_C++ Projects/C_C_Plus_Plus_Knowledge/data.txt", "r");
FILE *f = fopen("data.txt", "r");
if (f != NULL) {
    char s[MAX_LENGTH] = {0};
    while (fscanf(f, "%s", s) != EOF) {
        printf("%d\n", dynamic_solve(s));
    }
}

return 0;
}

/* 提交用
int main() {
char s[MAX_LENGTH] = {0};
while (scanf("%s", s) != EOF) {
printf("%d\n", dynamic_solve(s));
}
}
*/

```

## C 语言版本 2

为了通过，照着网上的答案又重新敲了一遍。这回提交正确了，最终代码如下：

```

// 构造回文，解决方案 2，思路一样都是动态规划，不过用的库函数不一样
// Created by w@tch on 16/7/23.
#include <string>
#include <iostream>
#include <algorithm> // reverse, max 函数均位于该头文件中

```

```

#include <vector>

using namespace std;

int main(void) {
    string s;
    string rs;
    while (cin >> s) {
        rs = s; // 复制
        reverse(rs.begin(), rs.end()); // 逆置

        // 动态规划初始化
        int length = s.size();
        vector<vector<int>> lcs(length + 1, vector<int>(length +
1, 0));

        /* 思路解释
         * 两个字符串 BDCABA 和 ABCBDAB, 字符串 BCBA 和 BDAB 都是它们的最长公共子序列
         * lcs[i][j] 表示 s1 长度为 i, s2 长度为 j 时的最大公共子串长度
         * 现在已知 lcs[i - 1][j] 即 s1 长度为 i - 1, s2 长度为 j 时的最大公共子串长度
         * 而且已知 lcs[i][j - 1] 即 s1 长度为 i, s2 长度为 j - 1 时的最大公共子串长度
         * 当 s1 与 s2 同时增加一个字符之后, 判断 lcs[i][j] 的最大公共子串长度
         * 动态方程, 如果 s1 和 s2 增加的字符是相同的, 则最大公共子串长度可以 + 1, 否则最大公共子串长度为 lcs[i-1][j] 或 lcs[i][j-1]
         */
        for (int i = 1; i <= length; i++) {
            for (int j = 1; j <= length; j++) {
                if (s[i - 1] == rs[j - 1]) {
                    lcs[i][j] = lcs[i - 1][j - 1] + 1;
                }
                else {
                    lcs[i][j] = max(lcs[i][j - 1], lcs[i - 1][j]);
                }
            }
        }
    }
}

```

```
    }

    cout << length - lcs[length][length] << endl;
}

}

[◀] [▶]
```

# 算法基础-字符移位

## 题目描述

小Q最近遇到了一个难题：把一个字符串的大写字母放到字符串的后面，各个字符的相对位置不变，且不能申请额外的空间。你能帮帮小Q吗？

输入描述：

输入数据有多组，每组包含一个字符串s，且保证 $1 \leq s.length \leq 1000$ .

输出描述：

对于每组数据，输出移位后的字符串。

输入例子：

AkIeBiCeilD

输出例子：

kleieiABCD

## 分析

几个知识点：

- 不需要额外空间的两个数交换操作
- 思路是从后面开始遍历，遇到小写字母就将其与首位交换位置，其余位置依次移位

## Python 代码实现

最终提交如下：

```
def no_extra_space_swap(a, b):
    """
    不能构建额外空间，那么交换移动元素使用位操作的那个版本 swap()
    原理是异或了 2 次相当于没有异或
    """
```

```

:param a: 'a'
:param b: 'b'
:return: 'b', 'a'
"""
a, b = ord(a), ord(b)
a ^= b
b ^= a
a ^= b
return chr(a), chr(b)

def post_order_move(s):
    """
    把一个字符串的大写字母放到字符串的后面，各个字符的相对位置不变，且不能申请额外的空间。
    :param s: "AkleBiCeilD"
    :return: "kleieeilABCD"
    """
    s = list(s)
    last = len(s) - 1
    for i in range(last, -1, -1):
        if s[i].islower():
            continue
        else:
            s[i], s[last] = no_extra_space_swap(s[i], s[last])
            for j in range(i, last - 1):
                s[j], s[j + 1] = no_extra_space_swap(s[j], s[j + 1])
    last -= 1
    return ''.join(s)

import sys
for line in sys.stdin:
    a = line.strip()
    print post_order_move(a)

```



# 有趣的数字

## 题目描述

小Q今天在上厕所时想到了这个问题：有n个数，两两组成二元组，差的绝对值最小的有多少对呢？差的绝对值最大的呢？

输入描述：

输入包含多组测试数据。

对于每组测试数据：

N - 本组测试数据有n个数

a<sub>1</sub>,a<sub>2</sub>...a<sub>n</sub> - 需要计算的数据

保证：

$1 \leq N \leq 100000, 0 \leq a_i \leq \text{INT\_MAX}$ .

输出描述：

对于每组数据，输出两个数，第一个数表示差的绝对值最小的对数，第二个数表示差的绝对值最大的对数。

输入例子：

6

45 12 45 32 5 6

输出例子：

1 2

## 说明

自己在网上找了 Python 的正确答案，然后写单元测试不断测试自己的答案，详见对应文件夹下的 `unittest_solve`

## 最终提交

虽然写得很丑...漂亮版本的参考单元测试里面的 `right_answer` 吧

```
from collections import OrderedDict

__author__ = '__L1n_w@tch'

def solve(n, data):
    """
    求解出差的绝对值最小的对数以及差的绝对值最大的对数
    :param n: 待输入数据的个数, such as 6
    :param data: 待输入的数据, such as [45, 12, 45, 32, 5, 6]
    :return: 差的绝对值最小的对数, 差的绝对值最大的对数, 比如 (1, 2)
    """
    data = sorted(list(map(int, data.split())))

    # 读取时创建字典, 判断是否有重复数字
    od = OrderedDict()
    has_same_number = False
    min_abs_sub_pairs, max_abs_sub_pairs = 0, 0

    if n == 1:
        return "{} {}".format(0, 0)

    for each_number in data:
        value = od.get(each_number, 0)
        if value > 0:
            has_same_number = True
        od[each_number] = value + 1

    # if 分支, 有重复数字则遍历求取, 每个有重复数字能提供  $n * (n - 1) / 2$  对
    if has_same_number:
        for each_number in od:
            min_abs_sub_pairs += od[each_number] * (od[each_number] - 1) // 2
    else: # 无重复数字则依次遍历求取
        temp_od = od.copy() # 拷贝一份以便后面处理
```

```
pre_items = temp_od.popitem(last=False) # 获取最小项
min_abs_sub = -1 # 初始化

# 遍历每一项
for next_items in temp_od.items():
    if min_abs_sub == -1 or abs(pre_items[0] - next_items[0]) < min_abs_sub:
        min_abs_sub = abs(pre_items[0] - next_items[0])
        min_abs_sub_pairs = pre_items[1] * next_items[1]
    elif min_abs_sub == abs(pre_items[0] - next_items[0]):
        min_abs_sub_pairs += pre_items[1] * next_items[1]
    else:
        min_abs_sub_pairs += pre_items[1] * next_items[1]

    pre_items = next_items

# 计算绝对值最大的对数，最高和最低个数乘积即可
max_abs_sub_pairs = od.popitem(last=True)[1] * od.popitem(last=False)[1]

return "{} {}".format(min_abs_sub_pairs, max_abs_sub_pairs)

if __name__ == "__main__":
    while True:
        try:
            length = int(input())
            data = raw_input()
            print solve(length, data)
        except EOFError:
            break
```

## 说明

本文件保存练习剑指 Offer 时的相关代码及笔记

## 题目练习链接

[牛客网](#)

## 题目描述

在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

## 思路解释

从右上角开始查找，如果大于右上角的数，说明当前列不可能存在这个数；如果小于右上角的数，说明当前行不可能存在这个数。

## 条件限制

时间限制：1秒

空间限制：32768K

## Python 版本最终提交

```
# -*- coding:utf-8 -*-
class Solution:
    # array 二维列表
    def Find(self, array, target):
        # write code here
        row = len(array)
        column = len(array[0])

        if row > 0 and column > 0:
            x, y = 0, column - 1 # 从右上角第一个数字开始找

            while 0 <= x < row and 0 <= y < column:
                if array[x][y] == target:
                    return True
                elif array[x][y] > target: # 说明当前列不会有这个数了
                    y -= 1
                elif array[x][y] < target: # 说明当前行不会有这个数了
                    x += 1
            return False
```

## C/C++ 版本最终提交

```
class Solution {
public:
    bool Find(vector<vector<int>> array, int target) {
        int row = array.size();
        int column = array[0].size();

        if (row > 0 && column > 0) {
            int x, y;
            x = 0, y = column - 1;
            while (x < row && y >= 0) {
                if (array[x][y] == target) {
                    return true;
                }
                else if (array[x][y] > target) {
                    --y;
                }
                else {
                    ++x;
                }
            }
        }

        return false;
    }
};
```

## 题目描述

输入一个链表，从尾到头打印链表每个节点的值。

输入描述:

输入为链表的表头

输出描述:

输出为需要打印的“新链表”的表头

## Python 版本提交

```
class Solution:  
    # 返回从尾部到头部的列表值序列，例如[1,2,3]  
    def printListFromTailToHead(self, listNode):  
        # write code here  
        res = list()  
  
        if listNode:  
            res.insert(0, listNode.val) # 注意插入的是数值而不是整个  
            对象  
            while listNode.next:  
                res.insert(0, listNode.next.val)  
                listNode = listNode.next  
  
        return res
```

## C/C++ 版本提交

思路跟 Python 版本一样

```
class Solution {
public:
    vector<int> printListFromTailToHead(struct ListNode *head)
    {
        vector<int> result;

        if (head != NULL)
        {
            result.insert(result.begin(), head->val);
            while (head->next != NULL)
            {
                result.insert(result.begin(), head->next->val);
                head = head->next;
            }
        }
        return result;
    }
};
```

## 题目描述

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。

输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。

例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。

NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

## Python 版本最终提交

看别人的 C/C++ 答案那么长，结果 Python 就一行，果然还是得自己用 C 重写

```
class Solution:  
    def minNumberInRotateArray(self, rotateArray):  
        # write code here  
        return min(rotateArray)
```

## C/C++ 版本最终提交

根据二分查找改编的，思路是参考剑指 Offer 里给的

```
class Solution {  
public:  
    int minNumberInRotateArray(vector<int> rotateArray)  
    {  
        if (rotateArray.size() > 0)  
        {  
            int low, high, middle;  
            low = 0;  
            high = rotateArray.size() - 1;  
            middle = low;  
  
            // 结束条件
```

```
        while (rotateArray[low] >= rotateArray[high])
        {
            // 结束条件
            if (high - low == 1)
            {
                middle = high;
                break;
            }

            // 求取中间的索引值
            middle = (low + high) / 2;

            // 判断中间值的大小，知道它是前面的递增序列还是后面的递增
            if (rotateArray[middle] <= rotateArray[high])
            {
                high = middle;
            }
            else if (rotateArray[middle] >= rotateArray[low])
            {
                low = middle;
            }
        }

        return rotateArray[middle];
    }
    return 0;
}
};
```

## 题目描述

请实现一个函数，将一个字符串中的空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

## 思路

- Python: 直接 `replace()` 方法就行了.....
- C/C++: 剑指 Offer 给出了两种方法，一种是  $O(n^2)$ , 从头遍历，遇到空格就进行替换操作，同时把其余字符往后移动；另一种方法是先遍历一遍，知道有几个空格，接着准备两个指针及一个数组，指针分别指向原始字符串以及数组的末尾，然后从后头开始遍历，直至两个指针相遇

## Python 版本最终提交

感觉跟作弊一样的 Python:

```
# -*- coding:utf-8 -*-
class Solution:
    # s 源字符串
    def replaceSpace(self, s):
        # write code here
        return s.replace(" ", "%20")
```

## C/C++ 版本最终提交

注意对 `length` 参数的理解，还有这里返回结果是直接修改 `str` 而不是重新申请一段内存空间

```
//length为牛客系统规定字符串输出的最大长度，固定为一个常数
class Solution {
public:
    void replaceSpace(char *str, int length)
```

```

{
    // 这其实可以忽略
    if (str == NULL || length < 0)
    {
        return;
    }

    // 遍历一遍数一下有多少个空格
    int space_numbers = 0;
    int i = 0;
    while (str[i] != '\0') // 注意不能数 length, 否则会越界
    {
        if (str[i++] == ' ')
        {
            ++space_numbers;
        }
    }

    // 计算替换后的长度
    int raw_length = strlen(str);
    int after_replace_length = raw_length + space_numbers * 2
;

    // 定义指针
    int point_to_raw, point_to_new;
    point_to_new = after_replace_length;
    point_to_raw = raw_length;

    // while 循环遍历, 条件是两个索引值相等
    // while 结构体实现替换操作
    while (point_to_new != point_to_raw)
    {
        if (str[point_to_raw] != ' ')
        {
            str[point_to_new] = str[point_to_raw];
            --point_to_new;
        }
        else
        {
            str[point_to_new] = '0';
        }
    }
}

```

```
        str[point_to_new - 1] = '2';
        str[point_to_new - 2] = '%';
        point_to_new -= 3;
    }
    --point_to_raw;
}

};


```

## 题目描述

用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。队列中的元素为 int 类型。

## Python 版本最终提交

```
class Solution:
    def __init__(self):
        self.first_stack = list()
        self.second_stack = list()

    def push(self, node):
        # write code here
        self.first_stack.append(node)

    def pop(self):
        # return xx
        if len(self.second_stack) <= 0:
            while len(self.first_stack) > 0:
                self.second_stack.append(self.first_stack.pop())
        return self.second_stack.pop()
```

## C/C++ 版本最终提交

突然发现 C/C++ 的 pop() 方法并不返回值啊...

```
class Solution {
public:
    void push(int node)
    {
        stack1.push(node);
    }

    int pop()
    {
        if (stack2.size() <= 0)
        {
            while (stack1.size() > 0)
            {
                stack2.push(stack1.top());
                stack1.pop();
            }
        }
        int result = stack2.top();
        stack2.pop();
        return result;
    }

private:
    stack<int> stack1;
    stack<int> stack2;
};
```

## 题目描述

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列 {1,2,4,7,3,5,6,8} 和中序遍历序列 {4,7,2,1,5,3,8,6}，则重建二叉树并返回。

## Python 最终提交

```

class Solution:
    # 返回构造的TreeNode根节点
    def reConstructBinaryTree(self, pre, tin):
        # write code here
        if pre and tin:
            # 找到根节点
            root = TreeNode(pre[0])

            # 计算根节点在中序中的索引
            root_position_in_order = tin.index(root.val)

            # 计算左子树长度
            left_length = root_position_in_order
            right_length = len(tin) - root_position_in_order - 1

            # 比该索引值小的为左子树
            if left_length > 0:
                root.left = Solution().reConstructBinaryTree(pre[
                    1:root_position_in_order + 1],
                                                               tin
                                                               [:root_position_in_order])

            # 计算右子树长度
            if right_length > 0:
                # 比该索引值大的为右子树
                root.right = Solution().reConstructBinaryTree(pr
                    e[root_position_in_order + 1:],
                                                               ti
                                                               n[root_position_in_order + 1:])

        return root

```

## C/C++ 最终提交

思路差不多，不过换了种实现法

```
class Solution {
public:
    struct TreeNode *reConstructBinaryTree(vector<int> pre, vector<int> in)
    {
        if (pre.size() > 0 && in.size() > 0)
        {
            TreeNode * root = new TreeNode(pre[0]);

            vector<int>::iterator root_position_in_order = find(
                in.begin(), in.end(), root->val);

            int left_length = root_position_in_order - in.begin();

            root->left = Solution().reConstructBinaryTree(
                vector<int>(pre.begin() + 1, pre.begin() + 1
eft_length + 1),
                vector<int>(in.begin(), root_position_in_order));

            root->right = Solution().reConstructBinaryTree(
                vector<int>(pre.begin() + left_length + 1, p
re.end()),
                vector<int>(root_position_in_order + 1, in.e
nd()));

            return root;
        }
        return NULL;
    }
};
```

## 网易 2017 校招笔试编程题

总共记录三道题：

- 二进制权重
- 平方串
- 数位和

## 题目描述

一个数的二进制权重被定义为一个十进制数表示为二进制数中 '1' 的个数。例如：

1 的二进制权重就是 1

1717 表示为二进制数为 11010110101，所以二进制权重为 7

现在给出一个正整数 N，返回最小的一个大于 N 并且二进制权重跟 N 相同的数。

输入描述：

输入一个数 N ( $1 \leq N \leq 1,000,000,000$ )

输出描述：

输出一个数，即为最小的一个大于 N 并且二进制权重跟 N 相同的数

输入例子：

1717

输出例子：

1718

## 自己的解答

### 正确但是超时的解答

```
def get_weight(number):
    return bin(number).count("1")

n = int(sys.stdin.readline().strip())
weight_n = get_weight(n)
for i in range(n + 1, 1000000000):
    if get_weight(i) == weight_n:
        print(i)
```

## 尝试优化的代码

由于自己当时笔试的时候写出来的版本只能通过 50% 的例子，所以下面先写个测试代码，再来改进自己的优化算法：

```
# 测试代码
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""

这里的版本是一个超时但是正确的版本，用这个版本来进行测试
"""

import unittest
from binary_weight import solve

__author__ = '__L1n_w@tch'

class TestBinaryWeightSolve(unittest.TestCase):
    def test_solve(self):
        for i in range(1, 1000000000):
            test_answer = solve(i)
            right_answer = TestBinaryWeightSolve.__right_answer__(i)
            log_message = "i = {}, right: {}, my_answer: {}".format(i, right_answer, test_answer)
            self.assertEqual(right_answer, test_answer, log_message)

    @staticmethod
    def __right_answer__(n):
        weight_n = TestBinaryWeightSolve.__get_weight__(n)
        for i in range(n + 1, 1000000000):
            if TestBinaryWeightSolve.__get_weight__(i) == weight_n:
                return i
        raise RuntimeError("[*] 自己的算法有问题了")

    @staticmethod
```

```
def __get_weight(number):
    return bin(number).count("1")

if __name__ == "__main__":
    pass
```

通过测试的代码，不知道效率如何

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""

20160924 发现原来只是 index == len(reverse_temp) - 1: 那里写错了，改
过来之后能够通过测试了

20160924 这是之前只能通过 50% 例子的版本，现在利用测试找出算法缺陷
"""

__author__ = '__L1n_w@tch'

def get_weight(number):
    return bin(number).count("1")

def get_start_number(number):
    temp = bin(number)[2:]
    reverse_temp = temp[::-1]
    flag = False
    for index in range(len(reverse_temp)):
        # 查找到 0
        if reverse_temp[index] == "0" and flag == False:
            continue
        # 找到所要找的数字了
        if reverse_temp[index] == "1":
            flag = True
        # 找到最后一个数字了
        if index == len(reverse_temp) - 1:
```

```
        result = "1" + "0" * (index + 1)
        return int(result, 2)
    if reverse_temp[index] == "0" and flag == True:
        result = "0" * index + "1" + reverse_temp[index + 1:
]
    result = result[::-1]
    return int(result, 2)

def solve(number):
    start_number = get_start_number(number)
    weight_n = get_weight(number)
    for i in range(start_number, 10000000000):
        if get_weight(i) == weight_n:
            return i

if __name__ == "__main__":
    solve(1)
```

## 题目描述

如果一个字符串由完全相同的两段字符组成，我们称其为平方串。例如：

'aa'，'ABAB'，'abcabc' 是平方串。

'aaa'，'ABCabc'，'abcab' 不是平方串。

现在给出一个字符串求它所有的连续子串中有多少种平方串。

例如：

'aaabccabccCC'，我们会发现 'aa'，'abccabcc'，'cc'，and 'CC' 这四种平方串。其中 'aa'，'cc' 都出现了 2 次，但是我们只统计一次种数。

输入描述：

输入为一个字符串，长度  $length(0 \leq length \leq 50)$ 。只包含大小写字母。

输出描述：

输出一个整数，即为所求的种数

输入例子：

aaabccabccCC

输出例子：

4

## 自己的解答

以下这个解答只能通过示例，其他例子还没试，当时还没来得及提交就结束考试了。

```
def is_pingfang(text):
    if len(text) == 1:
        return False
    elif len(text) % 2 != 0:
        return False
    else:
        return text[len(text) // 2:] == text[:len(text) // 2]

text = "aaabccabccCC"
result_set = list()

for i in range(len(text)):
    for j in range(1, len(text) - i + 1):
        test = text[i:i + j]
        if is_pingfang(test):
            result_set.append(test)

# print(result_set)
print(len(set(result_set)))
```

## 题目描述

一个数字的数位和定义为这个数字所有位置的数值的总和.例如: 1234 的数位和为:

$1 + 2 + 3 + 4 = 10$  5463 的数位和为:  $5 + 4 + 6 + 3 = 18$  现在有 3 个数 A、B、C, 需要你求出在在 A、B 范围内 (包括 A、B) 的一个数 X, 让 X 的数位和与 C 的数位和差值的绝对值最小。

输入描述：

输入为一行，一行有 3 个数 A、B、C，使用空格隔开。

(  $1 \leq A, B, C \leq 1,000,000,000$  )

(  $0 \leq B - A \leq 100,000$  )

输出描述：

输出一个数，即为所求得 X，如果有多解输出最小的那个解。

输入例子：

19 10

输出例子：

1

## 自己的解答

```
import sys

def get_sum(number):
    number = str(number)
    result = 0
    for each_num in number:
        result += int(each_num)
    return result

def solve(A, B, C_sum):
    min_result = 100000
    min_x = B + 1
    for x in range(A, B + 1):
        temp = abs(get_sum(x) - C_sum)
        if temp < min_result:
            min_result = temp
            min_x = x
    return min_x

line = sys.stdin.readline().strip()
values = map(int, line.split())
A, B, C = values[0], values[1], values[2]
c_value = get_sum(C)
print(solve(A, B, c_value))
```

## 题目 1

既可以用于黑盒测试，也可以用于白盒测试的方法的是（）

- 逻辑覆盖法
- 边界值法
- 基本路径法
- 正交试验设计法

## 正确答案及解析

选B，边界值法。

边界值法既可以用于黑盒测试用例，也可以用于白盒测试用例。

基本路径法用于白盒测试。

正交试验设计法用于黑盒测试。

逻辑覆盖法用于白盒测试用例设计。

所以答案为B，边界值法。

## 逻辑覆盖法

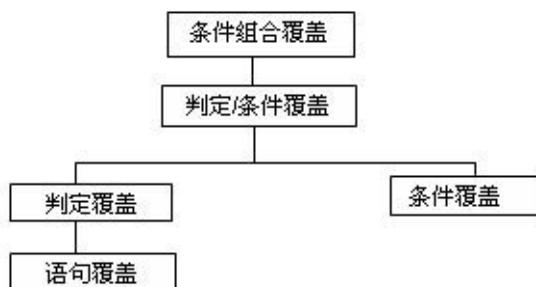
### 参考

由于覆盖测试的目标和程度不同，逻辑覆盖又分为：语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖和条件组合覆盖。

- 语句覆盖：测试用例能使被测程序的每条执行语句至少执行一次。
- 判定覆盖：测试用例能使被测程序中的每个判定至少取得一次“真”和一次“假”。又称分支覆盖。
- 条件覆盖：测试用例能使被测程序中每个判定的每个条件至少取得一次“真”和一次“假”。如果判定中只有一个条件，则条件覆盖便满足判定覆盖，否则，不一定。

- 判定/条件覆盖：测试用例既满足判定覆盖，又满足条件覆盖。
- 条件组合覆盖：测试用例使每个判定中所有可能的条件取值组合至少执行一次。

上述几种覆盖，其覆盖程度的强弱如图所示。



图中，连线上方的覆盖与连线下方的覆盖相比，其覆盖程度要强。例如，条件组合覆盖比判定/条件覆盖的覆盖程度强，也即满足条件组合覆盖的测试用例一定满足判定/条件覆盖，反之不成立。又例如，判定覆盖和条件覆盖分别在两个不同分支上，表示它们之间的覆盖程度不能比较强弱，也就是说满足判定覆盖的测试用例不一定能满足条件覆盖，反之亦然。也正因这样才有判定/条件覆盖。

## 正交试验设计法

### 参考

針對有多重輸入的情況設計測試案例，我們可用「窮舉法」來將所有的情況全部都考慮到，但由於窮舉法的組合非常地多，需要執行的時間也要很久，因此就有了「正交試驗設計法」來改善窮舉法的缺點。「正交試驗設計法」是一種針對多重輸入值的情況設計測試案例的方法，利用輸入的值做組合，挑選出有最有代表性的組合，來減少測試案例，用少數的測試案例，達到同樣的測試效果。

## 基本路径法

基本路径测试法是在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例的方法。

# 题目 2

下面不属于软件测试步骤的是

- 集成测试
- 回归测试
- 确认测试
- 单元测试

## 正确答案及解析

答案：回归测试

回归测试是指修改了旧代码后，重新进行测试以确认修改没有引入新的错误或导致其他代码产生错误。它不是软件测试的步骤

测试过程按4个步骤进行，即单元测试、集成测试、确认测试和系统测试及发版测试。

- 开始是单元测试，集中对用源代码实现的每一个程序单元进行测试，检查各个程序模块是否正确地实现了规定的功能。
- 集成测试把已测试过的模块组装起来，主要对与设计相关的软件体系结构的构造进行测试。
- 确认测试则是要检查已实现的软件是否满足了需求规格说明中确定了的各种需求，以及软件配置是否完全、正确。
- 系统测试把已经经过确认的软件纳入实际运行环境中，与其它系统成份组合在一起进行测试。

## 题目 3

单元测试能发现约80%的软件缺陷。请判断这句话的正确与否。

## 答案及解析

答案：对的

因为缺陷放大理论，在单元测试阶段发现的 bug 会在系统测试阶段被放大，放大倍数完全符合 80/20 理论

## 题目 4

下面属于白盒测试方法的是

- 等价划分方法
- 逻辑覆盖
- 边界值分析
- 错误推测法

## 答案及解析

答案：逻辑覆盖

其余为黑盒测试法。

## 题目 5

验收测试是由最终用户来实施的。请判断这句话的正确与否。

## 正确答案及解析

答案：错的

验收测试有 alpha 和 beta 两种，且都由用户来实施，区别在于是否为最终用户

alpha 测试是由一个用户在开发环境下进行的测试，也可以是公司内部用户在模拟实际操作环境进行的受控测试

beta 测试是由软件的多个用户在一个或多个实际使用环境下进行的测试

## 题目 6

软件测试的对象包括（ ）

- 目标程序和相关文档
- 源程序、目标程序、数据及相关文档
- 目标程序、操作系统和平台软件

- 源程序和目标程序

## 正确答案及解析

答案：B

软件测试的对象包括：程序、数据、文档。目标程序和源程序都属于程序。

## 题目 7

测试的关键问题是()

- 如何组织对软件的评审
- 如何验证程序的正确性
- 如何采用综合策略
- 如何选择测试用例

## 正确答案及解析

答案：D

测试用例是测试程序正确性与否的关键。一个覆盖完全的测试用例可以测试出程序是否正确运行，是否有 bug 等等，是最重要的。

## 题目 8

集成测试计划在需求分析阶段末提交。请判断这句话的正确与否。

## 正确答案及解析

答案：错的

集成测试计划在概要设计阶段末提交~

## 题目 9

负载测试是验证要检验的系统的能力最高能达到什么程度。请判断这句话的正确与否。

### 正确答案及解析

答案：对的

负载测试的目标是确定并确保系统在超出最大预期工作量的情况下仍能正常运行

## 题目 10

关于测试驱动开发，描述错误的是

- 测试驱动开发式是一种敏捷开发方法
- TDD需求开发人员学习测试相关知识
- 测试驱动开发不适合使用 CMM/CMMI 方法
- 测试驱动开发可以和结对编程结合使用

### 正确答案及解析

答案：C

CMM 是指“能力成熟度模型”，CMMI，能力成熟度模型集成；这两种方法属于测试驱动开发的方式。

结对编程技术是指两位程序员坐在同一工作台前开发软件。与两位程序员各自独立工作相比，结对编程能编写出质量更高的代码。

## 题目 11

下图用基本路径法测试需要覆盖几条路径？（ ）

例：有下面的C函数，用基本路径测试法进行测试。

```
void Sort(int iRecordNum,int iType){  
1. {  
2.     int x=0;  
3.     int y=0;  
4.     while (iRecordNum-- > 0)  
5.     {  
6.         if(0==iType)  
7.             { x=y+2; break; }  
8.         else  
9.             if(1==iType)  
10.                x=y+10;  
11.            else  
12.                x=y+20;  
13.     }  
14. }
```

## 正确答案及解析

4 条

- 1) (iRecordNum-->0) False --->只有1条路径
- 2) (iRecordNum-->0) True --->共有3条路径
  - 2.1) 0==iType --->1条
  - 2.2) 1==iType --->1条
  - 2.3) 其他的状况 --->1条

共 4 条

## 题目 12

下面有关白盒测试和黑盒测试说法错误的有？

- 白盒测试也称结构测试或逻辑驱动测试，是指基于一个应用代码的内部逻辑知识，即基于覆盖全部代码、分支、路径、条件的测试。
- 黑盒测试也称功能测试或数据驱动测试，它是在已知产品所应具有的功能，通过测试来检测每个功能是否都能正常使用
- 黑盒测试，也称作黑盒分析，是基于对程序内部细节有限认知上的软件调试方法

- 黑盒测试与白盒测试相比，更需要关心模块与模块之间的交互

## 正确答案及解析

答案：C

对于 C，度娘说“灰盒测试，也称作灰盒分析，是基于对程序内部细节有限认知上的软件调试方法”

对于 D，灰盒测试中，测试员可能知道系统组件之间是如何互相作用的，但缺乏对内部程序功能和运作的详细了解。对于内部过程，灰盒测试把程序看作一个必须从外面进行分析的黑盒。因此，假设测试对象有很多模块组成，1. 灰盒测试与黑盒测试相比，更关心模块与模块之间的交互 2. 灰盒测试与白盒测试相比，灰盒测试无需关心内部模块实现细节，而白盒测试仍需更深入了解内部模块细节。

## 题目 13

用边界值分析法，假定 $1 < X < 10$ ，那么X在测试中应该取的边界值是()

- $X=1, X=2, X=9, X=10$
- $X=2, X=9$
- $X=1, X=10$
- $X=1, X=5, X=6, X=10$

## 正确答案及解析

答案：A

边界值分析的基本思想是使用在最小值、略高于最小值、正常值、略低于最大值和最大值处取输入变量值，记为： $\text{min}$ 、 $\text{min+}$ 、 $\text{nom}$ 、 $\text{max-}$ 、 $\text{max}$  考虑到健壮性测试，还可以加一个略大于最大值  $\text{max+}$ ，以及一个略小于最小值  $\text{min-}$  的值。

## 题目 14

如果我们可以覆盖率检测来判断我们是否对所有的路径都进行了测试，但是仍然可能存在未被检测出来的缺陷，原因是()

- 全部选项
- 程序可能因为缺某些路径而存在问题
- 穷举路径的测试可能不好暴露数据敏感的错误
- 就算穷举路径测试也不能保证程序符合需求

## 正确答案及解析

答案：A

## 题目 15

计算一个任意三角形的面积， $S=\sqrt{(p(p-a)(p-b)(p-c))}$ , $p=(a+b+c)/2$ ,以下等价类测试用例中，不属于无效等价类的是（）

- a=5 , b=3 , c=6;
- a=2 , b=3 , c=5;
- a=7 , b=3 , c=3;
- a=2 , b=6 , c=3;

## 正确答案及解析

答案：A

选项 A 满足三角形两边之和大于第三边，两边之差小于第三边，是有效的

选项 BCD 都不满足两边之和大于第三边，属于等级无效的

## 题目 16

在下面说列举的逻辑测试覆盖中，测试覆盖最强的是？

- 条件覆盖
- 条件组合覆盖
- 语句覆盖
- 判定/条件覆盖

## 正确答案及解析

答案：B

条件覆盖 CC (Condition Coverage) , 设计足够多的测试用例，运行被测程序，使得每一判定语句中每个逻辑条件的可能取值至少满足一次。条件覆盖率的公式： 条件覆盖率=被评价到的条件取值的数量/条件取值的总数×100%

条件覆盖的缺点：只考虑到每个判定语句中的每个表达式，没有考虑到各个条件分支（或者涉及不到全部分支），即不能够满足判定覆盖。

条件组合覆盖，也称多条件覆盖 MCC (Multiple Condition Coverage) , 设计足够多的测试用例，使得每个判定中条件的各种可能组合都至少出现一次（以数轴形式划分区域，提取交集，建立最少的测试用例）。这种方法包含了“分支覆盖”和“条件覆盖”的各种要求。满足条件组合覆盖一定满足判定覆盖、条件覆盖、判定条件覆盖。条件组合覆盖率的公式： 条件组合覆盖率=被评价到的条件取值组合的数量/条件取值组合的总数。条件组合覆盖的缺点：判定语句较多时，条件组合值比较多。

语句覆盖 SC (Statement Coverage) , 就是设计若干个测试用例，运行被测程序，使得程序中每一可执行语句至少执行一次。这里的“若干个”，意味着使用测试用例越少越好。语句覆盖在测试中主要发现缺陷或错误语句。

判定条件覆盖 CDC (Condition/ Decision Coverage) , 设计足够多的测试用例，使得判定中的每个条件的所有可能（真/假）至少出现一次，并且每个判定本身的判定结果也至少出现一次。判定条件覆盖率的公式： 条件判定覆盖率=被评价到的条件取值和判定分支的数量/（条件取值总数+判定分支总数）。

判定条件覆盖的缺点：没有考虑单个判定对整体结果的影响，无法发现逻辑错误。

## 题目 17

如果某测试用例集实现了某软件的路径覆盖，那么它一定同时实现了该软件的

- 条件覆盖
- 判定/条件覆盖
- 组合覆盖
- 判定覆盖

## 正确答案及解析

答案：D

路径覆盖一定包含判定覆盖，与条件并没有直接关系

## 题目 18

设计系统测试计划需要参考的项目文档有哪些？

- 软件测试计划
- 可行性研究报告
- 软件需求规范
- 迭代计划

## 正确答案及解析

答案：ACD

【软件需求】是软件开发之前做好的，软件开发是根据这个做的，那么软件测试自然也需要参考该文件

【迭代计划】是软件的某个周期的计划，自然也需要参考

【可行性】是软件开发前做好，用于证明该计划可行的，没有必要参考

## 题目 19

下面哪些属于网游的测试内容？

- 客户端性能
- 服务器端性能
- 从运行完 game.exe 打开游戏界面后可进行的各种操作、玩法
- 界面

## 正确答案及解析

答案：ABCD

## 题目 20

下面属于白盒测试方法的有哪些？

- 语句覆盖
- 等价类划分
- 边界值分析
- 判定条件覆盖

## 正确答案及解析

答案：AD

具体的黑盒测试用例设计方法包括等价类划分法、边界值分析法、错误推测法、因果图法、判定表驱动法、正交试验设计法、功能图法、场景法等。

白盒测试的测试方法有代码检查法、静态结构分析法、静态质量度量法、逻辑覆盖法、基本路径测试法、域测试、符号测试、路径覆盖和程序变异

白盒测试法的覆盖标准有逻辑覆盖、循环覆盖和基本路径测试。其中逻辑覆盖包括语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖和路径覆盖。

## 题目 21

下面描述测试工具的功能正确的有？

- JMeter: 基于JAVA的压力测试工具，Badboy用来进行脚本的录制
- Junit: 白盒测试工具：针对代码测试
- LoadRunner: 负载压力测试
- TestLink: 用例管理工具

## 正确答案及解析

答案：ABCD

LoadRunner- 负载压力测试：预测系统性能。

JMeter+Badboy：基于JAVA的压力测试工具，Badboy用来进行脚本的录制 功能测试：通过自动录制、检测和回放用户的应用操作。将输出记录同预先给定的记录比较。

Junit：白盒测试工具：针对代码测试

测试管理工具：对测试需求、计划、用例、实施进行管理

测试辅助工具：本身不执行，可以生成测试数据，为测试提供数据准备

负载压力测试：LoadRunner:预测系统行为和性能的工业标准级负载测试工具。模拟上千万用户同时实施并发操作，来实时监控可能发生的问题。

功能测试： QTP(quicktest professional):自动测试工具

白盒测试：C++ TEST（做 C 和 C++ 的白盒测试） 、JUnit （Java 白盒测试）

缺陷管理工具：Mantis、BugFree、QC、TD

用例管理工具：TestLink、QC

测试辅助工具：SVN

## 题目 22

软件验收测试的合格通过准则是：

- 软件需求分析说明书中定义的所有功能已全部实现，性能指标全部达到要求。
- 所有测试项没有残余一级、二级和三级错误。
- 立项审批表、需求分析文档、设计文档和编码实现一致。
- 验收测试工件齐全。

## 正确答案及解析

答案：ABCD

验收测试工件齐全（测试计划，测试用例，测试日志，测试通知单，测试分析报告）

## 题目 23

软件验收测试包括哪些？

- 正式验收测试
- 白盒测试
- alpha 测试
- beta 测试

### 正确答案及解析

答案：ACD

软件验收测试分为三类：

正式验收测试；

非正式验收测试其中包括

- $\alpha$  测试（由用户、测试人员、开发人员共同参与的内部测试。）
- $\beta$  测试（内测后的公测，即完全交给最终用户测试。）

## 题目 24

软件测试计划评审会需要哪些人员参加？

- 项目经理
- SQA 负责人
- 配置负责人
- 测试组

### 正确答案及解析

答案：ABCD

软件测试计划评审会需要有项目经理、客户(可选)、配置管理员、测试经理、开发组长等人的参加。

## 题目 25

系统测试的策略有哪些？

- 负载测试
- 易用性测试
- 强度测试
- 安全测试

## 正确答案及解析

答案：ABCD

系统测试的16个测试策略：

功能测试、性能测试、压力测试、容量测试、安全性测试、GUI 测试、可用性测试、安装测试、配置测试、异常测试、备份测试、健壮性测试、文档测试、在线帮助测试、网络测试、稳定性测试。

## 题目 26

下面哪些测试属于黑盒测试方法（）

- 路径测试
- 等价类划分
- 边界值分析
- 条件判断
- 循环测试
- 因果图分析
- 正交分析法

## 正确答案及解析

答案：BCFG

具体的黑盒测试用例设计方法包括等价类划分法、边界值分析法、错误推测法、因果图法、判定表驱动法、正交试验设计法、功能图法、场景法等。白盒测试的测试方法有代码检查法、静态结构分析法、静态质量度量法、逻辑覆盖法、基本路径测试法、域测试、符号测试、路径覆盖和程序变异。

## 题目 27

有关"测试驱动开发"，下列说法正确的有？

- TDD 的原理是在开发功能代码之前，先编写单元测试用例代码，测试代码确定需要编写什么产品代码
- TDD 的基本思路就是通过测试来推动整个开发得进行，但测试驱动开发并不只是单纯的测试工作，而是把需求分析，设计，质量控制量化的过程。
- TDD 的重要目的不仅仅是测试软件，测试工作保证代码质量仅仅是其中一部分，而且是在开发过程中帮助客户和程序员去除模棱两可的需求。
- TDD 首先考虑使用需求（对象、功能、过程、接口等），主要是编写测试用例框架对功能的过程和接口进行设计，而测试框架可以持续进行验证。

## 正确答案及解析

答案：ABCD

## 题目 28

下列关于 alpha 测试的描述中正确的是哪些？

- alpha 测试需要用户代表参加
- alpha 测试不需要用户代表参加
- alpha 测试是系统测试的一种
- alpha 测试是验收测试的一种

## 正确答案及解析

答案：AD

alpha 测试都不是太正规的一种测试，它属于用户体验性测试，alpha 测试是测试环境尽量真实，由软件公司内部人员模拟各类用户对即将面世的软件产品进行测试，测试人员在一旁记录发现的问题和缺陷；对于软件项目来说，在系统测试后，有验收测试（有用户参与）；对于软件产品来讲，在系统测试后，有 alpha 和 beta 测试。

## 题目 29

下述有关负载测试，容量测试和强度测试的描述正确的有？

- 负载测试：在一定的工作负荷下，系统的负荷及响应时间。
- 强度测试：在一定的负荷条件下，在较长时间跨度内的系统连续运行给系统性能所造成的影响。
- 容量测试：容量测试目的是通过测试预先分析出反映软件系统应用特征的某项指标的极限值（如最大并发用户数、数据库记录数等），系统在其极限值状态下没有出现任何软件故障或还能保持主要功能正常运行。
- 容量测试是面向数据的，并且它的目的是显示系统可以处理目标内确定的数据容量。

## 正确答案及解析

答案：ABCD

## 题目 30

测试方法可以分成哪几种？

- 个人复查
- 抽查和会审
- 黑盒测试
- 白盒测试

## 正确答案及解析

答案：ABCD

软件测试可以是人工测试：如个人复查，抽查和会审等

也可以是机器自动测试，又有不同的分类：

按照否关软件内部结构具体实现角度划分

- A.白盒测试 B.黑盒测试 C.灰盒测试

按照软件发程按阶段划分

- A.单元测试 B.集成测试 C.确认测试 D.系统测试 E.验收测试

## 题目 31

测试人员要坚持原则，缺陷未修复完坚决不予通过。请判断这句话的正确与否。

### 正确答案及解析

答案：错的

缺陷分两种：

1、完全影响软件的正常运行或者影响客户的正常体验。

这种当然不能予以通过

2、不影响产品运行及客户正常体验且此软件急于使用。

以公司利益为出发，应予以通过。但在时间不紧急的情况下应不予以通过。

## 题目 32

代码评审员一般由测试员担任。请判断这句话的正确与否。

### 正确答案及解析

答案：错的

一般都是开发人员评审

## 题目 33

软件测试的目的是尽可能多的找出软件的缺陷。请判断这句话的正确与否。

### 正确答案及解析

答案：对的

1. 测试的目的是为了发现尽可能多的缺陷，不是为了说明软件中没有缺陷。
2. 成功的测试在于发现了迄今尚未发现的缺陷。所以测试人员的职责是设计这样的测试用例，它能有效地揭示潜伏在软件里的缺陷。

## 题目 34

判断对错。系统测试计划属于项目阶段性关键文档，因此需要同行评审。

### 正确答案及解析

答案：对的

同行评审目的:发现小规模工作产品的错误,系统测试计划属于项目阶段性关键文档，同行评审是必须的

## 题目 35

自底向上集成需要测试员编写驱动程序。请判断这句话的正确与否。

### 正确答案及解析

答案：对的

自底向上测试是从“原子”模块（即软件结构最低层的模块）开始组装测试，因测试到较高层模块时，所需的下层模块功能均已具备，所以不再需要桩模块。

自底向上集成方法不用桩模块，测试用例的设计亦相对简单，但缺点是程序最后一个模块加入时才具有整体形象，需要开发驱动模块。

## 题目 36

项目立项前测试人员不需要提交任何工件。请判断这句话的正确与否。

### 正确答案及解析

答案：对的

工件是加工过程中的生产对象。项目立项前，测试人员是不需要提供任何工件的。

## 题目 37

软件测试类型按开发阶段划分是？

- 需求测试、单元测试、集成测试、验证测试
- 单元测试、集成测试、确认测试、系统测试、验收测试
- 单元测试、集成测试、验证测试、确认测试、验收测试
- 调试、单元测试、集成测试、用户测试

### 正确答案及解析

答案：B

(1) 单元测试：单元测试又称为模块测试，是针对软件设计的最小单位程序模块进行正确性检查的测试工作，单元测试需要从程序内部结构出发设计测试用例，多个模块可以平行地独立进行单元测试。

(2) 集成测试 又称为组装测试或联合测试，在单元测试的基础上，需要将所有模块按照概要设计说明书和详细设计说明书的要求进行组装。

(3) 确认测试 确认测试的目标是验证软件的功能和性能以及其他特性是否与用户的要求一致。确认测试一般包括有效性测试和软件配置复查。一般有第三方测试机构进行。

(4) 系统测试 软件作为计算机系统的一部分，与硬件、网络、外设、支撑软件、数据以及人员结合在一起，在实际或模拟环境下，对计算机系统进行测试，目的在于与系统需求比较，发现问题

(5) 验收测试 以用户为主的测试，软件开发人员和质量保证人员参加，由用户设计测试用例。不是对系统进行全覆盖测试，而是对核心业务流程进行测试。

## 题目 38

alpha测试与beta的区别，描述错误的是？

- alpha测试是在用户组织模拟软件系统的运行环境下的一种验收测试，由用户或第三方测试公司进行的测试，模拟各类用户行为对即将面市的软件产品进行测试，试图发现并修改错误。
- Beta测试是用户公司组织各方面的典型终端用户在日常工作中实际使用beta版本，并要求用户报告异常情况，提出批评意见。
- beta测试的环境是不受开发方控制的，谁也不知道用户如何折磨软件，用户数量相对比较多，时间不集中。
- beta测试先于alpha测试执行

## 正确答案及解析

答案：D

Alpha测试由用户在开发者的场所进行，并且在开发者对用户的“指导”下进行测试。Alpha测试是在受控的环境中进行的。

Beta测试由软件的最终用户们在一个或多个客房场所进行。与Alpha测试不同，开发者通常在Beta测试的现场，因Beta测试是软件在开发者不能控制的环境中的“真实”应用。

## 题目 39

对手机软件的压力测试通常可以包括【多选】 ( )

- 存储压力
- 响应能力压力
- 网络流量压力
- 并发压力

## 正确答案及解析

答案：ABC

针对手机应用软件的系统测试，我们通常从如下几个角度开展：功能模块测试，交叉事件测试，压力测试，容量测试，兼容性测试，易用性/用户体验测试等。

对手机可以施加的压力测试类型主要有：存储压力、边界压力、响应能力压力、网络流量压力

并发压力是针对服务器的，因为每次并发是一个客户端

## 题目 40

以下对桩（stub）函数的描述正确的是？

- 在单元测试中调用其它模块
- 在单元测试中被其它模块调用
- 在自顶向下的集成过程中尤其有效
- 在自底向上的集成过程中尤其有效

## 正确答案及解析

答案：BC

桩模块（Stub）是指模拟被测试的模块所调用的模块，而不是软件产品的组成部分。在自顶向下的集成过程中尤其有效。

对桩（stub）函数的描述：在单元测试中被其它模块调用；在自顶向下的集成过程中尤其有效

## 题目 41

测试设计员的职责有哪些？

- 制定测试计划
- 设计测试用例
- 设计测试过程、脚本

- 评估测试活动

## 正确答案及解析

答案：BC

制定测试计划应是测试经理来做的，评估测试活动测试经理组织召集开发和测试的相关人员来做

## 题目 42

下面关于软件测试，描述正确的是？

- 软件测试是使用人工操作或者软件自动运行的方式来检验它是否满足规定的需  
求或弄清预期结果与实际结果之间的差别的过程。
- 软件测试的测试目标是发现一些可以通过测试避免的开发风险。
- 软件测试的原则之一是测试应该尽早进行，最好在需求阶段就开始介入。
- 软件测试主要工作内容是验证（verification）和确认（validation）

## 正确答案及解析

答案：ABCD

## 题目 43

做好文档测试需要注意的点有哪些？

- 仔细阅读，跟随每个步骤，检查每个图形，尝试每个示例
- 检查文档的编写是否满足文档编写的目的
- 内容是否齐全，正确，完善
- 标记是否正确

## 正确答案及解析

答案：ABCD

## 题目 45

集成测试的过程包括有以下哪些？

- 构建的确认过程
- 系统集成测试测试组提交过程
- 测试用例设计过程
- Bug的报告过程

### 正确答案及解析

答案：ABCD

系统集成测试主要包括以下过程：

1. 构建的确认过程。
2. 补丁的确认过程。
3. 系统集成测试测试组提交过程。
4. 测试用例设计过程。
5. 测试代码编写过程。
6. Bug的报告过程。
7. 每周/每两周的构建过程。
8. 点对点的测试过程。
9. 组内培训过程。

## 题目分布说明

- 1 ~ 75 题在 `readme1.md` 中
- 75 题以后在 `readme2.md` 中

## 题目 1

下列数据结构不是多型数据类型的是( )

- 堆
- 栈
- 字符串
- 有向图

## 正确答案

答案：C

多型就是数据元素的类型不确定，字符串的每个元素始终都是字符(char)，而不会是别的类型

## 题目 2

稀疏矩阵压缩存储后，必会失去随机存取功能()

## 正确答案及解析

答案：对的

稀疏矩阵在采用压缩存储后将会失去随机存储的功能。因为在这种矩阵中，非零元素的分布是没有规律的，为了压缩存储，就将每一个非零元素的值和它所在的行、列号做为一个结点存放在一起，这样的结点组成的线性表中叫三元组表，它已不是简单的向量，所以无法用下标直接存取矩阵中的元素。

## 题目 3

某二叉树的前序遍历序列为 `-+a*b-cd/ef`，后序遍历序列为 `abcd-*+ef/-`，问其中序遍历序列是()。

## 正确答案及解析

答案：  $a+b*c-d-e/f$

可以根据后缀表达式求出中缀表达式，中缀表达式是直观的表达式。

这里要注意理解，不能理解为树只给前序遍历和后序遍历就无法确定树的结构。

## 题目 4

一棵3阶B-树中含有2047个关键字，包含叶结点层，该树的最大深度为()

## 正确答案及解析

答案：12

$$\log_{[m/2]} \left( \frac{n+1}{2} \right) + 2$$

M阶B-树中含有N个关键字，最大深度为

## 题目 5

设某种二叉树有如下特点：每个结点要么是叶子结点，要么有2棵子树。假如一棵这样的二叉树中有m ( $m > 0$ ) 个叶子结点，那么该二叉树上的结点总数为（）。

- $2m+1$
- $2m-1$
- $2(m-1)$
- $2m$

## 正确答案及解析

答案：B

出度为0的结点为m

出度为2的结点 = 出度为0的结点 - 1 =  $m - 1$

题目中说：每个结点要么是叶子结点，要么有2棵子树

所以没有出度为1的结点

总结点数为： $2m - 1$

## 题目 6

若用数组  $S[0..n-1]$  做为两个栈  $S1$  和  $S2$  的共同存储结构，对任何一个栈，只有当  $S$  全满时才不能作入栈操作。为这两个栈分配空间的最佳方案是（ ）。

- $S1$  的栈底位置为 0， $S2$  的栈底位置为  $n-1$
- $S1$  的栈底位置为 0， $S2$  的栈底位置为  $n/2$
- $S1$  的栈底位置为 1， $S2$  的栈底位置为  $n/2$

## 正确答案及解析

答案：A

## 题目 7

下面关于二分查找的叙述中正确的是：

- 表必须有序，表可以顺序方式存储，也可以链表方式存储
- 表必须有序且表中数据必须是整型、实型或字符型
- 表必须有序，而且只能从小到大排列
- 表必须有序，且表只能以顺序方式存储

## 正确答案及解析

答案：D

折半查找方法适用于不经常变动而查找频繁的有序列表。

它的算法要求是必须是顺序存储结构，必须有序排列

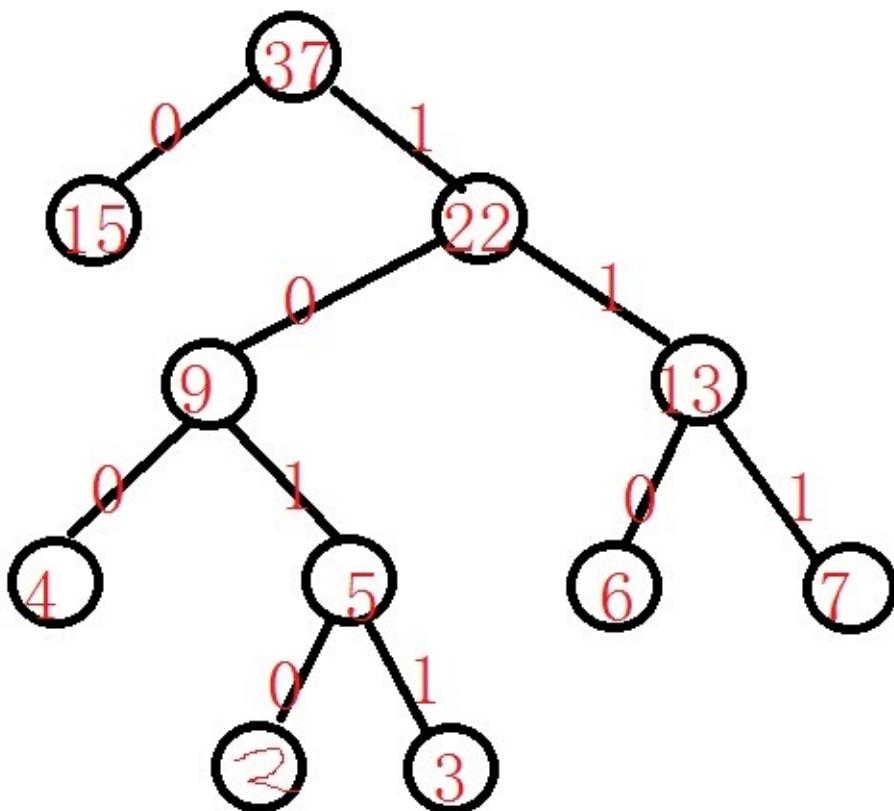
## 题目 8

假设某段通信电文仅由 6 个字母 ABCDEF 组成，字母在电文中出现的频率分别为 2, 3, 7, 15, 4, 6。根据这些频率作为权值构造哈夫曼编码，最终构造出的哈夫曼树带权路径长度与字母 B 的哈夫曼编码分别为 \_\_\_\_\_。(这里假定左节点的值小于右节点的值)

## 正确答案及解析

答案：86、1011

长度计算为  $(2+3) * 4 + (4+6+7) * 3 + 15 * 1 = 86$



## 题目 9

一个长度为 99 的循环链表，指针 A 和指针 B 都指向了链表中的同一个节点，A 以步长为 1 向前移动，B 以步长为 3 向前移动，一共需要同时移动多少步 A 和 B 才能再次指向同一个节点 \_\_\_\_\_。

## 正确答案及解析

答案：99

设需要走x步A、B才能重新相遇，那么  $x \% 99 = (3x) \% 99$ ；

令  $x \% 99 = r$ ，则可得  $x = 99k_1 + r$ ， $3x = 99k_2 + r$ ，整理两个式子可得  $x = 99 * (k_2 - k_1)/2$ ，因为均为正整数，所以x最小为99

## 题目 10

有一个数组（53,83,18,59,38,35），依次将其存储在hash表中，其中哈希函数为  $h(k)=k \% 7$ ，如采用线性探测（每次向后查找1位）的方式解决冲突，则该hash表上查找38,35,53访问hash表的表项次数分别为？, ?, ?。

### 正确答案及解析

5, 2, 1

表的顺序：

索引值	表项
0	38
1	35
2	
3	59
4	53
5	18
6	83

附带求模结果：

$53 \% 7 = 4$ ,  $83 \% 7 = 6$ ,  $18 \% 7 = 4$

$59 \% 7 = 3$ ,  $38 \% 7 = 3$ ,  $35 \% 7 = 0$

## 题目 11

对于某个函数调用，可以不给出被调用函数的原形的情况是（ ）。

- 被调用函数式无参函数
- 被调用函数式无返回值的函数
- 函数的定义在调用处之前
- 函数的定义在别的程序文件中

## 正确答案及解析

如果被调用函数的定义出现在主调函数之前，可以不必加以声明。因为编译系统已经事先知道了已定义的函数类型，会根据函数首部提供的信息对函数的调用作正确性检查。

## 题目 12

To speed up data access , we build cache system. In one system , The L1 cache access time is 5 ns , the L2 cache access time is 50 ns and the memory access time is 400 ns. The L1 cache miss rate is 50% , the L2 cache miss rate is 10%. The average data access time of this system is:

## 正确答案及解析

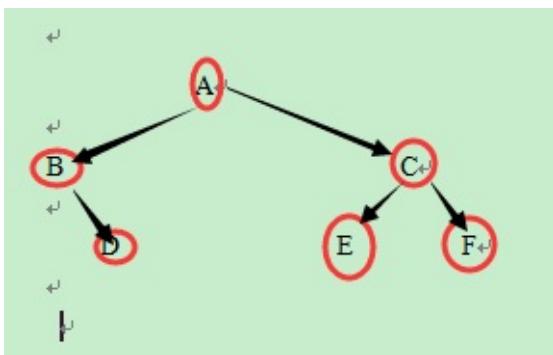
$$0.5 * 5 + 0.5 * 0.9 * (50 + 5) + 0.5 * 0.1 * (400 + 50 + 5) = 50$$

## 题目 13

某二叉树结点的中序序列为BDAECF,后序序列为DBEFCA,则该二叉树对应的森林包括()棵树

## 正确答案及解析

答案：3



构建后的二叉树的右下节点都是由对应的森林里不同的树相连而成的，可以想象二叉树右侧的先序遍历依次经过A-C-F到达树的底端，这三个结点在对应的森林中都必定分属不同的树。所以一共有三棵树。

## 题目 14

某表达式的前缀形式为 "+-\*^ABCD/E/F+GH" ,它的中缀形式为()

- $A^B * C - D + E / F / G + H$
- $A^B * (C - D) + (E / F) / G + H$
- $A^B * C - D + E / (F / (G + H))$
- $A^B * (C - D) + E / (F / (G + H))$

## 正确答案及解析

答案：C

前缀表达式的计算机求值特点：从右至左扫描表达式，遇到数字时，将数字压入堆栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算（栈顶元素  $op$  次顶元素），并将结果入栈；重复上述过程直到表达式最左端，最后运算得出的值即为表达式的结果。

根据从右至左扫描计算过程如下：

题目中的前缀形式为： $+ - * ^ A B C D / E / F + G H$

- 1) 首先扫描  $H$ , 是数字 入栈 , 栈中为:  $H$
  - 2) 扫描  $G$  为数字 入栈 , 栈中为:  $G, H$
  - 3) 扫描  $+$  为运算符 , 依次弹出  $G, H$  , 得到  $G+H$  的结果 入栈 , 栈中为：  $G+H$ (在这里方便讲解 标记为  $G+H$ )
  - 4) 扫描  $F$  为数字 , 入栈 , 栈中为  $F, G+H$
  - 5) 扫描  $/$  为运算符, 依次弹出  $F, G+H$  , 计算  $F/(G+H)$  的结果入栈 , 栈中为  $F/(G+H)$
  - 6) 扫描  $E$  为数字, 入栈 , 栈中为  $E, F/(G+H)$
  - 7) 扫描  $/$  为运算符, 依次弹出  $E, F/(G+H)$  , 计算  $E/(F/(G+H))$
  - 8) 扫描  $D$  为数字, 入栈 栈中为:  $D, E/(F/(G+H))$
  - 9) 扫描  $C$  为数字, 入栈 栈中为:  $C, D, E/(F/(G+H))$
  - 10) 扫描  $B$  为数字, 入栈 栈中为:  $B, C, D, E/(F/(G+H))$
  - 11) 扫描  $A$  为数字, 入栈 栈中为:  $A, B, C, D, E/(F/(G+H))$
  - 12) 扫描  $^$  为数字, 依次弹出  $A, B$  计算  $A^B$ 的结果入栈, 栈中为:  $A^B, C, D, E/(F/(G+H))$
  - 13) 扫描  $*$  为数字, 依次弹出  $A^B, C$  计算  $A^B * C$ 的结果入栈, 栈中为:  $A^B * C, D, E/(F/(G+H))$
  - 14) 扫描  $-$  为数字, 依次弹出  $A^B * C, D$  计算  $A^B * C - D$ 的结果入栈, 栈中为:  $A^B * C - D, E/(F/(G+H))$
  - 15) 扫描  $+$  为数字, 依次弹出  $A^B * C - D, E/(F/(G+H))$  计算  $A^B * C - D + E/(F/(G+H))$  的到结果
- 最后得到的表达式为：  $A^B * C - D + E/(F/(G+H))$

## 题目 15

下面有关 ibatis 中的#与\$的区别，描述错误的是？

- # 将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号
- \$ 方式能够很大程度防止 sql 注入。
- \$ 方式一般用于传入数据库对象，例如传入表名
- \$ 将传入的数据直接显示生成在 sql 中

## 正确答案及解析

答案：B

1. # 是把传入的数据当作字符串，如#user\_id\_list#传入的是1,2，则sql语句生成是这样，`in ('1,2')`，
2. \$ 传入的数据直接生成在sql里，如\$user\_id\_list\$传入的是1,2，则sql语句生成是这样，`in(1,2)`。
3. # 方式能够很大程度防止sql注入。
4. \$ 方式无法方式sql注入。
5. \$ 方式一般用于传入数据库对象。例如传入表名。
6. 一般能用#的就别用\$。

举例：

```
#str# 出来的效果是 'str'  
$str$ 出来的效果是 str
```

## 题目 16

下面关于B和B+树的描述中，不正确的是()

- B树和B+树都是平衡的多叉树
- B树和B+树都可用于文件的索引结构
- B树和B+树都能有效的支持顺序检索
- B树和B+树都能有效的支持随机检索

## 正确答案及解析

答案：C

B树只适用于随机检索，不适用于顺序检索；而B+树把所有关键码都存在叶结点上，这就为顺序检索也提供了方便。

- B 树和 B+ 树用于组织文件的动态索引结构。
- B 树和 B+ 树都是平衡的多分支树。
- B 树只适用于随机检索，不适用于顺序检索，而 B+ 树适用于顺序检索和随机检索

## 题目 17

求解最短路径的Floyd算法的时间复杂度为()

- $O(n)$
- $O(n+c)$
- $O(n*n)$
- $O(n*n*n)$

## 正确答案及解析

答案：D

```
int n;//n为节点个数
for(int i=0; i<n; ++i )
{
    for (int j=0; j<n; ++j )
    {
        for ( int k=0; k<n; ++k )
        {
            if ( Dis[i][k] + Dis[k][j] < Dis[i][j] )
            {
                // 找到更短路径
                Dis[i][j] = Dis[i][k] + Dis[k][j];
            }
        }
    }
}
```

## 题目 18

关于进程和线程，下列说法正确的是 \_\_\_\_

- 线程是资源分配和拥有的单位
- 线程和进程都可并发执行
- 在 linux 系统中，线程是处理器调度的基本单位
- 线程的粒度小于进程，通常多线程比多进程并发性更高
- 不同的线程共享相同的栈空间

## 正确答案及解析

答案：BCD

- A：进程是资源分配和拥有的单位
- C：线程是处理机调度和分配的单位
- B：进程之间可以并发执行，同一个进程的多个线程之间也可并发执行
- E：每个线程拥有自己的堆栈，和自己的寄存器上下文

## 题目 19

每个进程在操作系统中用进程控制块（process control block，PCB）来表示，请找出以下不属于进程控制块中的信息 \_\_\_\_\_。

- 进程PID
- 进程优先级
- 进程间通信方式
- 进程的执行时间

## 正确答案及解析

答案：D

通常 PCB 应包含如下一些信息：

- 进程标识符 name
- 进程当前状态 status
- 进程相应的程序和数据地址，以便把 PCB 与其程序和数据联系起来
- 进程资源清单。列出所拥有的除 CPU 外的资源记录
- 进程优先级 priority
- CPU 现场保护区 cpustatus
- 进程同步与通信机制，用于实现进程间互斥、同步和通信所需的信号量等
- 进程所在队列 PCB 的链接字
- 与进程有关的其他信息，如进程记账信息，进程占用 CPU 的时间等

## 题目 20

设计实时操作系统时，首先应该考虑系统的优良性和分配性。

## 正确答案及解析

答案：错的

设计实时操作系统时，首先应该考虑系统的实时性和可靠性。

## 题目 21

Inter-process communication (IPC) is the transfer of data among processes.  
Which of the following is NOT a typical programming technique for IPC?

- mutex
- pipe
- socket
- message queue

## 正确答案及解析

答案：A

题目问哪一个不是进程间通信的方式。其中进程间通信的方式有管道（pipe）、共享存储器系统、消息传递系统（message queue）以及信号量。而mutex是互斥锁，在锁机制中通过原语保证资源状态的检查和修改作为一个整体来执行，以保证共享数据操作的完整性，并不能在两个进程间传递消息。网络上的两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个socket，也就是说socket也是两个进程间的通信方式。

## 题目 22

TCP 协议中的 ESTABLISHED 状态是什么状况？

- 服务器端处于监听状态
- 服务器接受 SYN 报文，建立 TCP 连接时的三次握手会话过程中的一个中间状态

- 连接已经建立状态
- 初始状态

## 正确答案及解析

答案：C

- **CLOSED**：表示初始状态。
- **LISTEN**：表示服务器端的某个 **SOCKET** 处于监听状态，可以接受连接。
- **SYN\_RCVD**：这个状态表示接到了 **SYN** 报文，在正常情况下，这个状态是服务器端的 **SOCKET** 在建立 **TCP** 连接时的三次握手会话过程中的一个中间状态，很短暂，基本上用 **netstat** 你是很难看到这种状态的，除非你特意写了一个客户端测试程序，故意将三次 **TCP** 握手过程中最后一个 **ACK** 报文不予发送。因此这种状态时，当收到客户端的 **ACK** 报文后，它会进入到 **ESTABLISHED** 状态。
- **SYN\_SENT**：这个状态与 **SYN\_RCVD** 遥相呼应，当客户端 **SOCKET** 执行 **CONNECT** 连接时，它首先发送 **SYN** 报文，因此也随即它会进入到 **SYN\_SENT** 状态，并等待服务端的发送三次握手中的第2个报文。**SYN\_SENT** 状态表示客户端已发送 **SYN** 报文。
- **ESTABLISHED**：表示连接已经建立。
- **FIN\_WAIT\_1**：**FIN\_WAIT\_1** 和 **FIN\_WAIT\_2** 状态的真正含义都是表示等待对方的 **FIN** 报文。而这两种状态的区别是：**FIN\_WAIT\_1** 状态实际上是当 **SOCKET** 在 **ESTABLISHED** 状态时，它想主动关闭连接，向对方发送了 **FIN** 报文，此时该 **SOCKET** 即进入到 **FIN\_WAIT\_1** 状态。而当对方回应 **ACK** 报文后，则进入到 **FIN\_WAIT\_2** 状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应 **ACK** 报文，所以 **FIN\_WAIT\_1** 状态一般是比较难见到的，而 **FIN\_WAIT\_2** 状态还有时常常可以用 **netstat** 看到。
- **FIN\_WAIT\_2**：**FIN\_WAIT\_2** 状态下的 **SOCKET**，表示半连接，也即有一方要求 **close** 连接，但另外还告诉对方，还有数据需要传送，稍后再关闭连接。
- **TIME\_WAIT**：表示收到了对方的 **FIN** 报文，并发送出了 **ACK** 报文，就等 **2MSL** 后即可回到 **CLOSED** 可用状态了。如果 **FIN\_WAIT\_1** 状态下，收到了对方同时带 **FIN** 标志和 **ACK** 标志的报文时，可以直接进入到 **TIME\_WAIT** 状态，而无须经过 **FIN\_WAIT\_2** 状态。
- **CLOSING**：属于一种比较罕见的例外状态。正常情况下，当你发送 **FIN** 报文后，按理来说是应该先收到（或同时收到）对方的 **ACK** 报文，再收到对方的

FIN 报文。但是 CLOSING 状态表示你发送 FIN 报文后，并没有收到对方的 ACK 报文，反而却也收到了对方的 FIN 报文。

- CLOSE\_WAIT：这种状态的含义表示在等待关闭。当对方 close 一个 SOCKET 后发送 FIN 报文，你系统毫无疑问地会回应一个 ACK 报文给对方，此时则进入到 CLOSE\_WAIT 状态。
- LAST\_ACK：它是被动关闭一方在发送 FIN 报文后，最后等待对方的 ACK 报文。当收到 ACK 报文后，也即可以进入到 CLOSED 可用状态了。

## 题目 23

在下列存储形式中，哪一个不是树的存储形式？()

- 双亲表示法
- 孩子链表表示法
- 孩子兄弟表示法
- 顺序存储表示法

### 正确答案及解析

答案：D

二叉树可以，不过要补充很多 ##### 这个东东，很浪费空间，二叉树已经很浪费了，如果是 N 叉树呢……所以 D 不行

- 双亲表示法

以一组连续的空间存储数的节点，同时在每个元素中，附带一个用于指示其双亲节点在数组中的下标的指示器。

- 孩子表示法

与双亲表示法相反，每个节点在保存数据的同时记录了其左右孩子的下标

- 孩子兄弟表示法

在存储结点信息的同时，附加两个分别指向该结点最左孩子和右邻兄弟的指针域leftmostchild和rightsibling，即可得树的孩子兄弟链表表示

## 题目 24

中断响应时间是指（ ）。

- 从中断处理开始到中断处理结束所用的时间
- 从发出中断请求到中断处理结束所用的时间
- 从发出中断请求到进入中断处理所用的时间
- 从中断处理结束到再次中断请求的时间

## 正确答案及解析

答案：C

## 题目 25

IP 地址中网络号的作用有哪些？

- 指定了主机所属的网络
- 指定了网络上主机的标识
- 指定了设备能够进行通信的网络
- 指定被寻址的网中的某个节点

## 正确答案及解析

答案：A

网络号代表主机所在的网络区域所以 A 对

但是有网络号并不能确定到某一台机器上，所以不是一个节点，是一片区域所以BD错，也并不是指能够通信的网络，要能够通信要达到很多条件才行。

## 题目 26

下面哪种内存管理方法有利于程序的动态链接？（）

- 分段存储管理
- 分页存储管理
- 可变分区分配
- 固定分区分配

## 正确答案及解析

答案：A

动态链接是指在作业运行之前，并不把几个目标程序段链接起来。要运行时，先将主程序所对应的 目标程序装入内存并启动运行，当运行过程中又需要调用某段时，才将该段(目标程序)调入内存并进行链接。可见，动态链接 也要求以段作为管理的单位。

## 题目 27

下列有关图的遍历说法中，不正确的是

- 有向图和无向图都可以进行遍历操作
- 基本遍历算法两种：深度遍历和广度遍历
- 图的遍历必须用递归实现
- 图的遍历算法可以执行在有回路的图中

## 正确答案及解析

答案：C

图的遍历分为递归和非递归实现，即为深度遍历和广度遍历

其实所有的递归都可以变成非递归，通过使用栈来实现。

## 题目 28

下面有关多核CPU和单核CPU的描述，说法错误的是？

- 双核2.4GHZ，那么其中每单个核心的频率也是2.4GHZ
- 多核CPU功耗低，体积小
- 多核cpu共用一组内存，数据共享
- 所有程序在多核CPU上运行速度都快

## 正确答案及解析

答案：D

- 多核cpu的主要优势在于处理多任务，处理多任务时性能才能充分发挥出来
- 如果程序是单进程单线程的，多核cpu的处理速度核单核cpu的处理速度理论上是相同的
- 在多核cpu上为了充分利用其性能，程序最好采用多进程多线程异步非阻塞的方式运行

## 题目 29

一台主机要实现通过局域网与另一个局域网通信，需要做的工作是？

- 配置域名服务器
- 定义一条本机指向所在网络的路由
- 定义一条本机指向所在网络网关的路由
- 定义一条本机指向目标网络网关的路由

## 正确答案及解析

答案：D

如果主机想访问本地局域网外的某一网络，需要做两件事：

- 1、设置本机的默认网关。
- 2、本地局域网默认网关上需要设置一条路由，用以完成本地局域网内的任一主机到目标局域网主机的路由工作。

## 题目 30

三个程序a,b,c,它们使用同一个设备进行I/O操作，并按a,b,c的优先级执行(a优先级最高，c最低).这三个程序的计算和I/O时间如下图所示。假设调度的时间可忽略。则在单道程序环境和多道程序环境下(假设内存中可同时装入这三个程序，系统采用不可抢占的调度策略).运行总时间分别为()

计算 I/O 计算

a 30 40 10

b 60 30 10

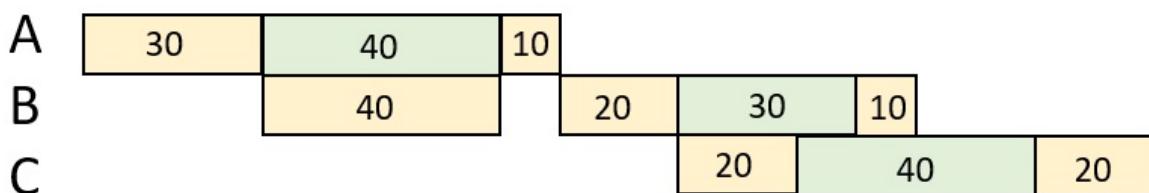
c 20 40 20

### 正确答案及解析

260， 180

单道时运行时间为  $80+100+80=260$

黄色是计算时间，绿色是 I/O 输出时间，多道程序环境下运行时间如图所示  
为  $30+40+10+20+80=180$



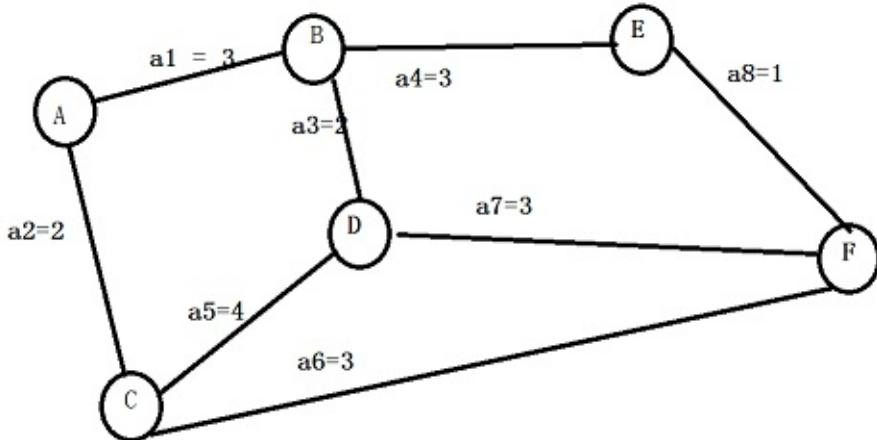
### 题目 31

下面关于求关键路径的说法不正确的是()

- 一个事件的最迟开始时间为以该事件为尾的弧的活动最迟开始时间与该活动的持续时间的差
- 求关键路径是以拓扑排序为基础的
- 一个事件的最早开始时间同以该事件为尾的弧的活动最早开始时间相同
- 关键活动一定位于关键路径上

### 正确答案及解析

答案 : A



- 节点代表事件，边代表活动
- 边，都是从左  $\rightarrow$  右，即  $a_3$  对应的是  $\langle B, D \rangle$ ,  $B \rightarrow D$  这个方向
- 该 AOE 网络的源点为 A, 汇点为 F
  - 事件 D 的最早发生时间：从源点 A 开始到达 D 的所有路径加和的最大值  

$$\max\{a_1, a_3, a_2, a_5\} = 6$$
  - 事件 D 的最迟发生时间：

首先求出汇点 F 的最早发生时间：

$$\max\{a_1, a_4, a_8, a_1, a_3, a_7, a_2, a_5, a_7, a_2, a_6\} = 8$$

汇点 F 的最早发生时间 -  $\max\{\text{汇点逆向到事件 } D \text{ 的路径累加之和}\}$  =

$$8 - \max\{a_7\} = 5$$

- 活动  $a_3 \langle B, D \rangle$ 
  - 最早发生时间：以  $a_3$  该活动为出发点的事件  $B \dots a_3 \dots \rightarrow D$ ，即 B 事件的最早发生时间；
  - 最迟发生时间：以  $a_3$  该活动对应的箭头所指向的事件的最迟发生时间 -  $a_3$  活动持续时间  $B \dots a_3 \dots \rightarrow D$ ，即，D 事件的最迟发生时间 -  $a_3$  活动的持续时间  $= 5 - 2 = 3$

综上所述，

最早发生时间，事件=活动，是从前往后计算

最迟发生时间，都是从后往前计算，

事件最迟发生时间与汇点和路径累加有关，

事件的时间计算要先于活动的时间计算

活动最迟发生时间与活动持续时间有关

## 题目 32

若某线性表最常用得操作是存取任一指定序号的元素和在最后进行插入和删除运算，则利用哪种存储方式最节省时间？

- 顺序表
- 双链表
- 带头结点的双循环链表
- 单循环链表

## 正确答案及解析

答案：A

线性表最常用得操作是存取任一指定序号的元素和在最后进行插入和删除运算；

进行任意位置存取，这个最省时省力的就是数组了，也就是顺序表。

而且元素是在最后的位置进行插入和删除运算，也就不涉及其他元素进行位移的额外操作，最多涉及的就是去扩展空间了。

所以答案选择 A 顺序表

## 题目 33

已知一棵二叉树，如果先序遍历的节点顺序是：ADCEFGHB，中序遍历是：CDFEGHAB，则后序遍历结果为：（）

## 正确答案及解析

CFHGEBDA

## 题目 34

设有数组  $A[i,j]$ ，数组的每个元素长度为 3 字节， $i$  的值为 1 到 8， $j$  的值为 1 到 10，数组从内存首地址 BA 开始顺序存放，当用以列为主存放时，元素  $A[5,8]$  的存储首地址为（）。

## 正确答案及解析

答案： BA + 180

注意是列为主，  $((7 * 8 + 5) - 1) * 3 = 180$

## 题目 35

对进程和线程的描述，一下哪个是正确的？

- 父进程的所有线程共享相同的地址空间，父进程的所有子进程共享相同的地址空间
- 改变进程里面主线程的状态会影响其他线程的行为，改变父进程的状态不会影响其他子进程
- 多线程会引发死锁，而多进程不会
- 以上都不对

## 正确答案及解析

答案：D

- A 错，进程拥有独立的地址空间
- B 错，主线程和子线程是并行关系的时候，并没有依赖关系。父进程和子进程中，子进程是父进程的一个副本，创建子进程后，子进程会有自己的空间，然后把父进程的数据拷贝到子进程的空间里。运行时，谁先运行是不确定的，这由系统决定
- C 错，多线程和多进程都会引起死锁，一般说的死锁指的是进程间的死锁

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

## 题目 36

下述有关hash冲突时候的解决方法的说法，错误的有？

- 通常有两类方法处理冲突：开放定址(Open Addressing)法和拉链(Chaining)法。
- 开放定址更适合于造表前无法确定表长的情况
- 在用拉链法构造的散列表中，删除结点的操作易于实现
- 拉链法的缺点是：指针需要额外的空间，故当结点规模较小时，开放定址法较为节省空间

## 正确答案及解析

答案：B

由于拉链法中各链表上的结点空间是动态申请的,故它更适合于造表前无法确定表长的情况

## 题目 37

如果关系模式 $R=(A,B,C,D,E)$ 中的函数依赖集 $F=\{A\rightarrow B, B\rightarrow C, CE\rightarrow D\}$ ,这是第几范式？

## 正确答案及解析

答案：第一范式

$F=\{A\rightarrow B, B\rightarrow C, CE\rightarrow D\}$ ，主键为 $(A,E)$ ，非主属性 $B,C$ 却并不是完全依赖于码 $(A,E)$ ，只依赖于主键的部分属性 $A$ ，因此不符合 $2NF$ , 只 $1NF$ 。

- \* 第一范式（1NF）：属性不可分；
- \* 第二范式（2NF）：符合 1NF，并且，非主属性完全依赖于主键，而不是依赖于部分主键属性；
- \* 第三范式（3NF）：符合 2NF，并且，消除传递依赖；
- \* BC 范式（BCNF）：符合 3NF，并且，主属性不依赖于主属性（若一个关系达到了第三范式，并且它只有一个候选码，或者它的每个候选码都是单属性，则该关系自然达到 BC 范式）；
- \* 第四范式：要求把同一表内的多对多关系删除；
- \* 第五范式：从最终结构重新建立原始结构。

## 题目 38

完全二叉树共有 700 结点，该二叉树有多少个叶子结点？

### 正确答案及解析

答案：350

因为  $12/2$  等于 6，等于父节点值，所以是最后一个带子节点的，拿总数减去 6，即为叶子节点数，同理，所以 700 作为最后一个节点，他的父节点是 350，所以序号 350 是最后一个非叶子节点，以下的都没有子节点， $700 - 350 = 350$  所以答案选 B

## 题目 39

下面 Transact-SQL 语句中可以用于创建主键的是（）

- alter table table1 with notcheck add constraint [PK\_table1] primary key nonclustered (column1) on primary;
- alter table table1 column1 primary key;
- alter table table1 column1;
- create table table1 (column1 char(13) not null primary, column2 int not) on primary;

### 正确答案及解析

答案：A

表中删除主键为：

```
alter table table_test drop primary key;
```

表中增加主键为：

```
alter table table_test add primary key(id);
```

## 题目 40

字符串 `www.qq.com` 所有非空子串（两个子串如果内容相同则只算一个）个数是（）

## 正确答案及解析

答案：50

要求的是子串，从左到右一次截取，10 个字符的子串，1 个；

9 个字符的子串，2 个；

8, 3 个；

7, 4 个；

.....

1, 10 个

共有： $1 + 2 + 3 + \dots + 10 = 10 * (10 + 1) / 2 = 55$

减去重复的：1 个字符时有 3 个 w, 2 个 q, 2 个 ., 2 个字符时有 2 个 ww, 故应减去： $(2+1+1+1)=5$

答案： $55 - 5 = 50$

## 题目 41

从浏览器打开 `http://www.nowcoder.com`，TCP/IP 协议族中不会被使用到的协议是（）

- SMTP
- HTTP
- TCP
- IP

## 正确答案及解析

答案：A

## 题目 42

一进程刚获得3个主存块的使用权，若该进程访问页面的次序是 {1,2,3,4,1,2,5,1,2,3,4,5}。当采用先进先出调度算法时，发生缺页次数是()次

### 正确答案及解析

答案：9

1被访问，第1次缺页，队列为1  
2被访问，第2次缺页，队列为1, 2  
3被访问，第3次缺页，队列为1, 2, 3  
4被访问，第4次缺页，队列为2, 3, 4。1出队列，4入队列。  
1被访问，第5次缺页，队列为3, 4, 1。2出队列，1入队列。  
2被访问，第6次缺页，队列为4, 1, 2。3出队列，2入队列。  
5被访问，第7次缺页，队列为1, 2, 5。4出队列，5入队列。  
1被访问，此时不缺页，队列不变，即队列为1, 2, 5。  
2被访问，此时不缺页，队列不变，即队列为1, 2, 5。  
3 被访问，第8次缺页，队列为 2, 5, 3。1出队列，3入队列。  
4 被访问，第9次缺页，队列为 5, 3, 4。2出队列，4入队列。  
5被访问，此时不缺页，队列不变，即队列为5, 3, 4。  
所以共9次缺页。

## 题目 42

以下对 CSMA/CD 描述正确的是？

- 在数据发送前对网络是否空闲进行检测
- 在数据发送时对网络是否空闲进行检测
- 在数据发送时对发送数据进行冲突检测
- 发生碰撞后 MAC 地址小的主机拥有发送优先权

### 正确答案及解析

答案：AC

CSMA/CD 协议即带冲突检测的载波监听多路访问协议。

- A，发送前空闲检测，只有信道空闲才发送数据
- B，发送时当前信号占据信道，信道必定不为空闲，检测空闲没意义
- C，发送过程中冲突检测，如果发生冲突，立即停止发送，随机避让。
- D，冲突发送后，两个发送端都随机避让一段时间，避让的时间是随机的，优先级相等，没有哪个优先权高的说法。

## 题目 43

内存管理中的 LRU 方法是用来管理什么的？

- 虚拟内存的分配
- 虚拟内存的释放
- 物理内存的分配
- 物理内存的释放

## 正确答案及解析

答案：AD

1. 页面调入：是给页面调入内存中，给它分配物理内存。
2. 页面置换，就是将内存中的页面置换出来，放到虚拟内存中，让物理内存空闲出来，让给需要使用的页面。
3. LRU：全称是：Least Recently Used（最近最久未使用）置换算法，所以这个算法涉及到了虚拟内存的分配和物理内存的释放。所以答案是 AD。

## 题目 44

http, telnet, ftp 的端口是 ?, ?, ?(FTP 写最常用的那一个端口就好)

## 正确答案及解析

80, 23, 21

FTP 端口号有两个 20 和 21，一个为控制端口，一个为数据端口

## 题目 45

VLAN的主要作用有？

- 保证网络安全
- 抑制广播风暴
- 简化网络管理
- 提高网络设计灵活性

## 正确答案及解析

答案：B

VLAN（virtual local area network）虚拟局域网，把大的局域网划分为几个单独的互不相通的虚拟局域网，隔离广播风暴。

## 题目 46

在公有派生情况下，有关派生类对象和基类对象的关系，下列叙述不正确的是()

- 派生类的对象可以赋给基类的对象
- 派生类的对象可以初始化基类的引用
- 派生类的对象可以直接访问基类中的成员
- 派生类的对象的地址可以赋给指向基类的指针

## 正确答案及解析

答案：C

A，B 和 D 说法是相同的，都是说可以用派生类对象初始化基类对象或者指针，是正确的

C，派生类对象只有在基类成员未被派生类覆盖的情况下才能访问基类中的成员

## 题目 47

国标规定交换机中具备CID功能的用户电路的配置比例暂定为

- 5%~10%
- 10%~20%
- 10%~30%
- 10%~40%

## 正确答案及解析

答案：C

## 题目 48

一棵非空的二叉树的前序序列和后序序列正好相反，则该二叉树一定满足()

- 其中任意一个结点均无左孩子
- 其中任意一个结点均无右孩子
- 其中只有一个叶结点
- 其中度为2的结点最多为一个

## 正确答案及解析

答案：C

一棵非空的二叉树的前序序列和后序序列正好相反，则该二叉树一定满足只有左子树或只有右子树。

A,B 明显不对，举最简单的例子，1为根节点，2分别为1的左右孩子，都满足前序序列和后序序列正好相反。

D 不可能出现一个度为2的节点。

## 题目 49

判断一个单向链表中是否存在环的最佳方法是( )

- 两重遍历
- 快慢指针
- 路径记录

- 哈希表辅助

## 正确答案及解析

答案：B

让快慢指针都从链表表头开始，快指针一次向前移动连个位置，慢指针每次向前移动一个位置。如果快指针到达了NULL，说明不存在环，但如果快指针追上了慢指针，说明存在环。

## 题目 50

把逻辑地址转变为内存的物理地址的过程称作（）。

- 编译
- 连接
- 运行
- 重定位或地址映射

## 正确答案及解析

答案：D

重定位对应的也是虚拟内存的地址，不是真正的物理地址

## 题目 51

对于IP地址130.63.160.2，MASK为255.255.255.0，子网号为（）

- 160.2
- 160
- 63.160
- 130.63.160

## 正确答案及解析

答案：B

130.63.160.2是B类IP地址

B类IP地址前16位（两个字节）为网络号，后16位是主机号

划分子网就是将主机号中的一部分拿出来当做子网号

这里子网掩码为255.255.255.0也就是把前三个字节当成了网络号

与B类IP默认的前两个字节作为网络号相比，第三个字节就是子网号，就是160

所以这个ip的网络号是130.63 子网号是 160 主机号是2

## 题目 52

关于进程的正确说法是()。

- 进程就是程序，或者说进程是程序的另一叫法
- 一个被创建了的进程，在它被消灭之前，处于进程的三种基本状态之一
- 多个不同的进程不可以包含相同的程序
- 一个处于等待队列中的进程，即使进入其他状态，仍然放在等待队列中

## 正确答案及解析

答案：B

进程的三种基本状态包括有就绪状态，执行状态，阻塞状态。

多个不同的进程可以包含相同的程序段，

## 题目 53

下列关于网络编程错误的是 \_\_\_\_\_。

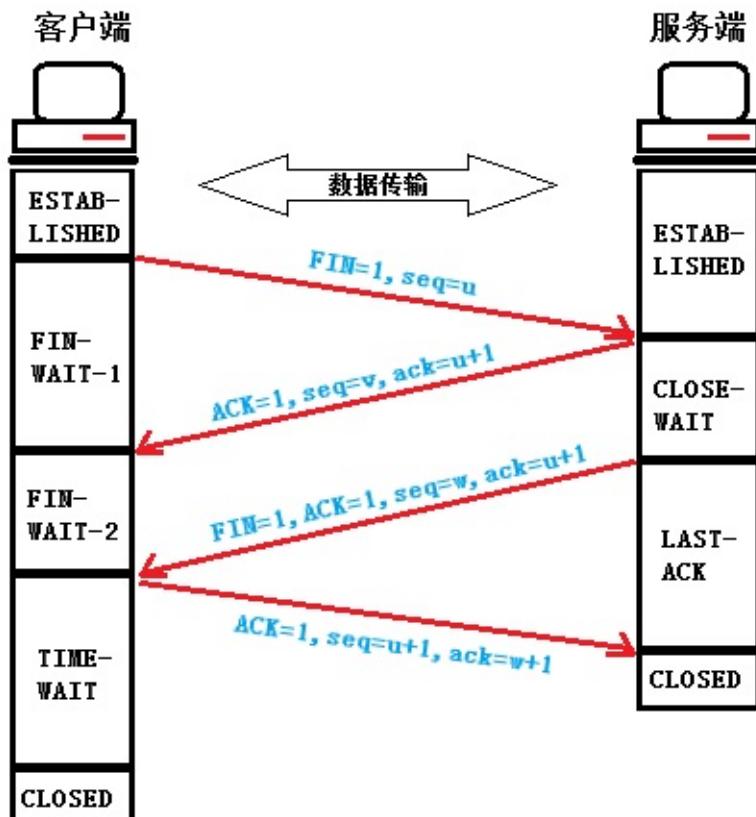
- UDP 是不可靠服务
- 主动关闭的一端会出现 TIME\_WAIT 状态
- 服务端编程会调用 listen(), 客户端也可以调用 bind()
- TCP 建立和关闭连接都只需要三次握手

- Linux 通过提供提供 socket 接口来进行网络编程
- 长连接相对短连接可以节省建立连接的时间

## 正确答案及解析

答案：D

TCP建立连接3次握手,断开链接需要4次



## 题目 54

单链表的每个结点中包括一个指针link，它指向该结点的后继结点。现要将指针q指向的新结点插入到指针p指向的单链表结点之后，下面的操作系列中哪一个是正确的？

- $q=p->link; p->link=q->link$
- $p=p->link=q->link; p->link$
- $q->link=p->link; p->link=q;$

- $p->link=1; q->link=p->link$

## 正确答案及解析

答案：C

首先应该让新结点q的link指向p的link也就是 $q->link = p->link;$

然后再让p的link指向q即可.也就是 $p->link = p;$

## 题目 55

有 $n(n>0)$ 个分支结点的满二叉树的深度是()

## 正确答案及解析

答案： $\log_2(n+1)+1$

假设树有K层，所有的分枝节点都在  $1 - (k - 1)$  层，每层都是满的，对有  $1 - (k - 1)$  层，有  $2^{k-1} - 1 = n$  变形后得：  $k = \log_2(n+1) + 1$ 。

## 题目 56

有订单表orders，包含字段用户信息userid，字段产品信息productid，以下语句能够返回至少被订购过两会的productid？

- select productid from orders where count (productid) >1
- select productid from orders where max (productid) >1
- select productid from orders where having count (productid) >1 group by productid
- select productid from orders group by productid having count (productid) >1

## 正确答案及解析

答案：D

顺序为：select, from, where, group by, having, order by, limit

## 题目 57

银行家算法中的数据结构包括有可利用资源向量 Available、最大需求矩阵 Max、分配矩阵 Allocation、需求矩阵 Need，下列选项中表述正确的是（ ）。

- $\text{Allocation}[i,j] = \text{Max}[i,j] + \text{Need}[i,j]$
- $\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$
- $\text{Max}[i,j] = \text{Allocation}[i,j] * \text{Need}[i,j]$
- $\text{Need}[i,j] = \text{Max}[i,j] + \text{Allocation}[i,j]$

## 正确答案及解析

答案：B

## 题目 58

进行数据库提交操作时使用事务（Transaction）是为了？

- 提高效率
- 保证数据一致性
- 网络安全
- 归档数据文件

## 正确答案及解析

答案：B

数据库事务(Database Transaction)，是指作为单个逻辑工作单元执行的一系列操作。事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。通过将一组相关操作组合为一个要么全部成功要么全部失败的单元，可以简化错误恢复并使应用程序更加可靠。一个逻辑工作单元要成为事务，必须满足所谓的ACID(原子性、一致性、隔离性和持久性)属性。

## 题目 59

以下代码的输出结果是？

```
#define a 10

void foo();
main(){

    printf("%d..",a);
    foo();
    printf("%d",a);
}

void foo(){
    #undef a
    #define a 50
}
```

## 正确答案及解析

答案：10..10

宏定义是在编译器预处理阶段中就进行替换了，替换成什么只与define和undefine的位置有关系，与它们在哪个函数中无关。

以本题为例：#define a 10 到 #undef

a之间的代码在预处理阶段就将a全部换为10，#define a 50后面的代码会将a替换为50。

如果没有#define a 50后面再使用a，编译器就会报错了。

## 题目 60

采用可重定位分区分配方式，（）。

- 使用用户程序占用若干不连续的内存空间
- 解决了碎片问题
- 为用户编写程序提供方便
- 扩充了内存容量，提供了虚拟存储器

## 正确答案及解析

答案：B

可重定位分区分配方式也是属于连续分配方式的，只是它在内存碎片很多而导致的程序不能放入内存时，进行“紧凑”（可能会移动原来的数据的，所以此时就需要重定位啦~紧凑完了，就能放进去啦~~），所以A是不对的哦

## 题目 61

mysql数据库有选课表 `learn(student_id int, course_id int)` ,字段分别表示学号和课程编号，现在想获取每个学生所选课程的个数信息，请问如下的sql语句正确的是

- `select student_id,sum(course_id)from learn`
- `select student_id,count(course_id)from learn group by student_id`
- `select student_id,count(course_id)from learn`
- `select student_id,sum(course_id)from learn group by student_id`

## 正确答案及解析

答案：B

`group by student_id` 是按学生号分组，每个编号的学生可能有多门课程，但不可能课程编号会重复，所以直接使用 `count(course_id)` ，否则就要使用 `count(distinct(course_id))`

## 题目 62

若度为m的哈夫曼树中,其叶结点个数为n,则非叶结点的个数为()

## 正确答案及解析

答案：  $[(n-1)/(m-1)]$

首先说明一点,我们平时一般所说的哈夫曼树是指最优二叉树,也叫做严格二叉树(注意不是完全二叉树),但是哈夫曼树完全不局限于二叉树,也存在于多叉树中,即度为m的哈夫曼树,也叫最优m叉树,严格m叉树(注意不是完全m叉树)

这题表示哈夫曼树的节点的度要么是0要么是m

设度不为0(即非叶结点)的个数为X

则总的结点数为:X+n

除根结点外,其余的每一个结点都有一个分支连向一个结点,对于度为m的每个结点都有m个分支,而度为0的结点是没有分支的,所以从分支的情况来看

总的结点数位: $X \cdot m + 1$  (这里的1为根结点)

两者相等,所以答案是  $(n-1) / (m-1)$

## 题目 63

计算斐波那契数列第n项的函数定义如下:

- 117
- 137
- 157
- 177

## 正确答案及解析

答案:D

若  $C(n)$  表示计算次数，则

```
C(0) = 1;  
C(1) = 1;  
C(n) = C(n-1) + C(n-2) + 1; n>=2;
```

故：

```
C(0) = 1;  
C(1) = 1;  
C(2) = 1 + 1 + 1 = 3;  
C(3) = 3 + 1 + 1 = 5;  
C(4) = 5 + 3 + 1 = 9;  
C(5) = 9 + 5 + 1 = 15;  
.....
```

## 题目 64

给定下列程序，那么执行 `printf("%d\n", foo(20, 13));` 的输出结果是

\_\_\_\_\_。

```
int foo(int x, int y){  
    if (x <= 0 || y <= 0)  
        return 1;  
    return 3 * foo( x-6, y/2 );  
}
```

## 正确答案及解析

答案：81

分析： $3^6 < 20 < 4^6$ ，递归 4 层。

$\log_{13} < \log_{16} = 4;$

所以结果为  $3^4 = 81$ 。

## 题目 65

设图 G 的相邻矩阵如下图:则 G 的顶点数和边数分别为()

```
01111  
10100  
11011  
10101  
10110
```

## 正确答案及解析

答案：5, 8

总共有5行，所以有5个顶点。

在无向图里，矩阵里任意两个1为两个顶点的连线，总共有8对。所以边数为8。

## 题目 66

( ) 操作不是P操作可完成的。

- 为进程分配处理机
- 使信号量的值变小
- 可用于进程的同步 \*使进程进入阻塞状态

## 正确答案及解析

答案：A

P操作分配的是我们申请的资源，并不是处理机

## 题目 67

关于下列操作哪个复杂度为O(1)?

- `vector<>` 中插入元素(动态数组)
- `set` 中查找元素
- `hasp_map` 中查找元素

- `dequeue` 尾部删除元素

## 正确答案及解析

答案：CD

双端队列，尾部插入一个应该是O(1)吧

## 题目 68

同一进程下的线程可以共享以下？

- stack
- data section
- register set
- file fd

## 正确答案及解析

答案：BD

线程共享的内容包括：

1. 进程代码段
2. 进程的公有数据(利用这些共享的数据，线程很容易的实现相互之间的通讯)
3. 进程打开的文件描述符
4. 信号的处理器
5. 进程的当前目录
6. 进程用户ID与进程组ID

线程独有的内容包括：

1. 线程ID
2. 寄存器组的值
3. 线程的堆栈
4. 错误返回码
5. 线程的信号屏蔽码

## 题目 69

下面描述中正确的为：

- 线性表的逻辑顺序与物理顺序总是一致的。
- 线性表的顺序存储表示优于链式存储表示。
- 线性表若采用链式存储表示时所有结点之间的存储单元地址可连续可不连续。
- 二维数组是其数组元素为线性表的线性表。

## 正确答案及解析

答案：CD

## 题目 70

在下列说法中，哪个是错误的（ ）

- 若进程A和进程B在临界段上互斥，那么当进程A处于该临界段时，它不能被进程B中断
- 虚拟存储管理中采用对换(swapping)策略后，用户进程可使用的存储空间似乎增加了
- 虚拟存储管理中的抖动(thrashing)现象是指页面置换(page replacement)时用于换页的时间远多于执行程序的时间
- 进程可以由程序、数据和进程控制块(PCB)描述

## 正确答案及解析

答案：AC

注意A选项，A进程是可以被B进程中断的，只是B不能进入临界区。

C选项中：是请求分页虚拟存储管理。

当需要执行访条的指令或使用某个数据而发现他们不再内存中时候，会产生缺页异常。

系统从磁盘中把此指令或数据所在的页面装入。缺页异常是由硬件所产生的一种特殊终端信号，其中当中断率较高时，整个系统的页面调度非常频繁造成大部分时间都花费在来回调度上，而不是执行任务，这种现象叫做“抖动”。——《操作系统》

## 题目 71

数组不适合作为任何二叉树的存储结构()

### 正确答案及解析

答案：错的

二叉树的顺序存储就是利用数组实现的。也就是用一组连续的存储单元存放二叉树中的结点。依据二叉树的性质，完全二叉树和满二叉树采用顺序存储比较合适，树中结点的序号可以唯一地反映出结点之间的逻辑关系，这样既能够最大可能地节省存储空间，又可以利用数组元素的下标值确定结点在二叉树中的位置，以及结点之间的关系。

## 题目 72

程序出错在什么阶段( )？

```
int main(void)
{
    http://www.taobao.com
    cout << "welcome to taobao" << endl;
    return 0;
}
```

- 预处理阶段出错
- 编译阶段出错
- 汇编阶段出错
- 链接阶段出错
- 运行阶段出错
- 程序运行正常

## 正确答案及解析

答案：F

"http:"是goto语句的标记，" // " 是注释，注释内容为"www.taobao.com"

## 题目 73

下列哪一个不属于关系数据库的特点？

- 数据冗余度小
- 数据独立性高
- 数据共享性好
- 多用户访问

## 正确答案及解析

答案：D

关系数据库的特点：

- 数据库存在的一个目的就是统一管理数据，减少数据冗余度，A正确；
- 数据独立性，指数据和其管理软件独立，以及数据及其结构的独立，B正确；
- 数据库就是为了方便用户之间共享数据，C正确；
- 数据库中存在锁机制，如果多用户访问可能导致数据不一致等，D不正确。

## 题目 74

通道能够完成（ ）之间数据的传输。

- CPU与外设
- 内存与外设
- CPU与主存
- 外设与外设

## 正确答案及解析

答案：B

CPU正在工作，突然想到要与外设通信，于是发命令给通道，然后接着做自己的工作。通道接到命令后，接通外设与内存，并在他们之间传递数据，等数据传递完成后，通知CPU进行处理。

## 题目 75

某IP地址192.168.48.10，掩码为255.255.255.128，其所在的子网为()，广播地址为()，有效的主机IP地址范围从()到().

正确答案及解析

答案：192.168.48.0/192.168.48.127/192.168.48.1 到 192.168.48.126

255转换为2进制是 11111111

128转换为2进制是 10000000

对地址 192.168.48.10 和掩码 255.255.255.128 进行 and 操作 得到 子网  
192.168.48.0

ip地址和掩码做and操作后，得到这个子网地址的都属于这个ip段， 192.168.48.0 ... 192.168.48.127 和 255.255.255.128 进行and操作后都是 192.168.48.0

其中， 192.168.48.127 为广播地址， 192.168.48.0 ... 192.168.48.126 为有效地址

## 题目 76

正确答案及解析

答案：

## 题目 77

正确答案及解析

答案：

## 题目 78

正确答案及解析

答案：

## 题目 79

正确答案及解析

答案：

## 题目 80

正确答案及解析

答案：

## 题目 81

正确答案及解析

答案：

## 题目 82

正确答案及解析

答案：

## 题目 83

正确答案及解析

答案：

## 题目 84

正确答案及解析

答案：

## 题目 85

正确答案及解析

答案：

## 题目 86

正确答案及解析

答案：

## 题目 87

正确答案及解析

答案：

## 题目 88

## 正确答案及解析

答案：

## 题目 89

### 正确答案及解析

答案：

## 题目 90

### 正确答案及解析

答案：

## 题目 91

### 正确答案及解析

答案：

## 题目 92

### 正确答案及解析

答案：

## 题目 93

### 正确答案及解析

答案：

## 题目 94

正确答案及解析

答案：

## 题目 95

正确答案及解析

答案：

## 题目 96

正确答案及解析

答案：

## 题目 97

正确答案及解析

答案：

## 题目 98

正确答案及解析

答案：

## 题目 99

### 正确答案及解析

答案：

## 题目 76

下面有关Cookie的说法，错误的是

- Cookie不是只有一个，而是一个网站一个
- Cookie总是保存在客户端中，按在客户端中的存储位置，可分为内存Cookie和硬盘Cookie
- 在HTTP请求中的Cookie是密文传递的
- 有一些Cookie在用户退出会话的时候就被删除了，这样可以有效保护个人隐私

## 正确答案及解析

答案：C

HTTP的cookie是明文传送的，HTTPS的cooike是密文传送的。

## 题目 77

关于linux的进程，下面说法不正确的是：

- 僵尸进程会被init进程接管，不会造成资源浪费；
- 孤儿进程的父进程在它之前退出，会被init进程接管，不会造成资源浪费；
- 进程是资源管理的最小单位，而线程是程序执行的最小单位。Linux下的线程本质上用进程实现；
- 子进程如果对资源只是进行读操作，那么完全和父进程共享物理地址空间。

## 正确答案及解析

答案：A

孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。

## 题目 78

拓扑结构指的是（）的拓扑结构。

- 资源网络
- 通信网络
- 线路网络
- 链路

## 正确答案及解析

答案：A

计算机网络拓扑结构是指网络中各个站点相互连接的形式，各个站点抽象来说都是网络资源。

计算机网络的最主要的拓扑结构有总线型拓扑、环型拓扑、树型拓扑、星型拓扑、混合型拓扑以及网状拓扑。其中环形拓扑、星形拓扑、总线拓扑是三个最基本的拓扑结构。在局域网中，使用最多的是星型结构。

## 题目 79

在下面的叙述中正确的是（）。

- 线程是比进程更小的能独立运行的基本单位
- 引入线程可提高程序并发执行的程度，可进一步提高系统效率
- 线程的引入增加了程序执行时时空开销
- 一个进程一定包含多个线程

## 正确答案及解析

答案：B

选B，A线程不能独立运行，线程需要进程所获得的资源。

C引入线程机制降低了时空的开销。

D一个进程至少包含一个主线程（线程数量大于等于1）。

## 题目 80

已知有一个关键字序列：(19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79) 散列存储在一个哈希表中，若散列函数为  $H(key) = key \% 7$ ，并采用链地址法来解决冲突，则在等概率情况下查找成功的平均查找长度为（）。

## 正确答案及解析

答案：1.5

这些关键字除以7取余后分别得到5, 0, 2, 1, 5, 6, 0, 6, 6, 4, 3, 2 存储结构如下

位置--存储

0-----14-84 //14查找1次，84需要查找2次，以下类似

1-----1

2-----23-79

3-----10

4-----11

5-----19-68

6-----20-27-55

总查找次数为  $1+2+1+1+2+1+1+1+2+1+2+3=18$

总共有12的关键字

平均查找次数为  $18/12=1.5$

## 题目 81

Longest Increasing Subsequence (LIS) means a sequence containing some elements in another sequence by the same order, and the values of elements keeps increasing. For example, LIS of {2, 1, 4, 2, 3, 7, 4, 6} is {1, 2, 3, 4, 6}, and its LIS length is 5. Considering an array with N elements, what is the average time and space complexity to get the length of LIS?

- Time:  $N^2$ , Space:  $N^2$
- Time:  $N^2$ , Space: N
- Time:  $N \log N$ , Space: N
- Time: N, Space: N
- Time: N, Space: C

## 正确答案及解析

答案：C

- 最长递增子序列，可以用动态规划（DP）算法来求解。
- 动态规划相当于递归的改进版，就是用辅助数组记录求解的中间过程来减少计算量。
- 需要一个长度为N的辅助数组记录当前子串的最长递增子序列的长度。
- 平均时间复杂度为 $N \log N$ ，空间复杂度为N

[最长递增子序列详解（longest increasing subsequence）](#)

## 题目 82

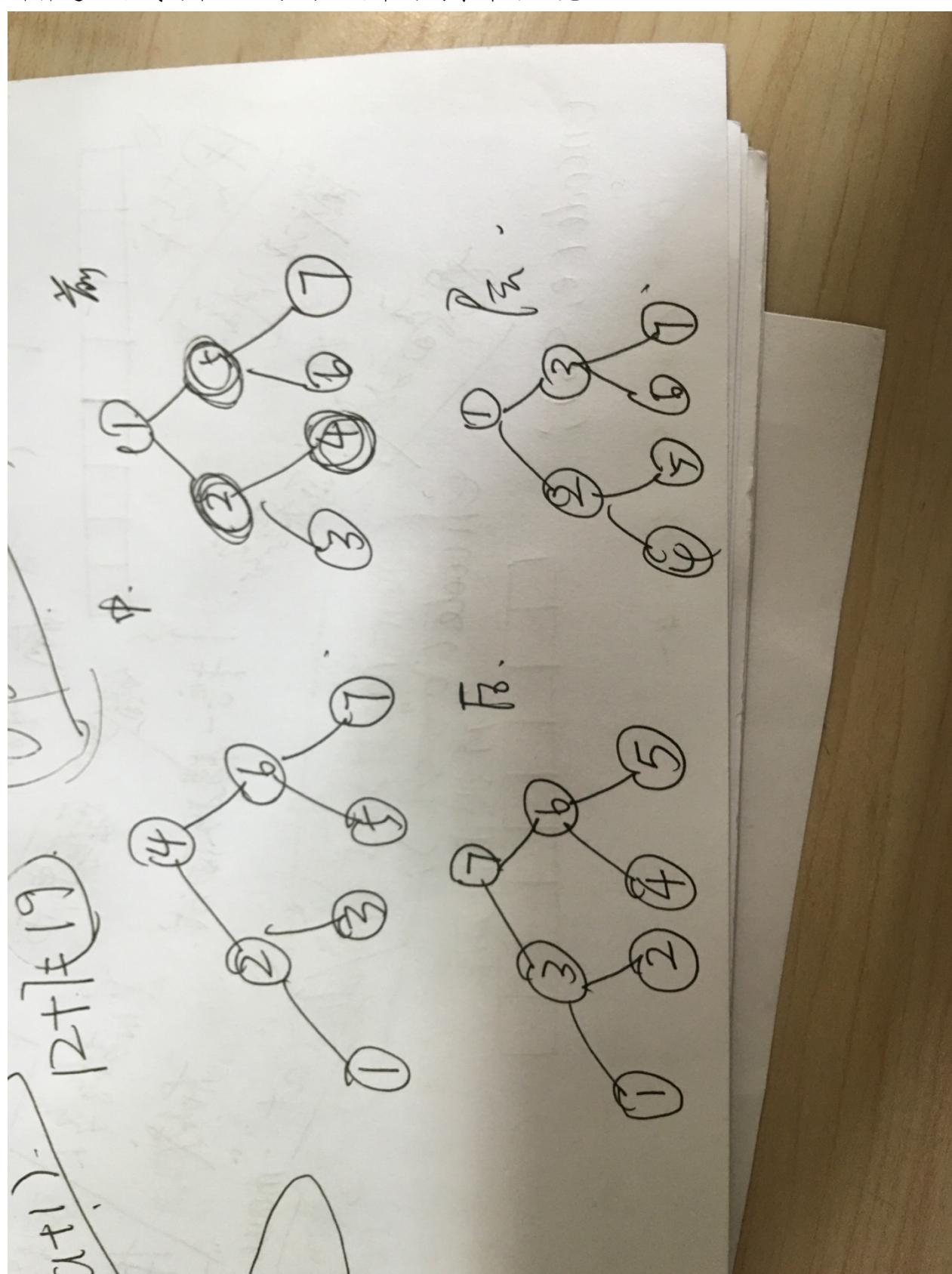
某二叉树T有n个结点,设按某种顺序对T中的每个结点进行编号,编号为1,2,...,n,且有如下性质:T中任一结点V,其编号等于左子树上的最小编号减1,而V的右子树的结点中,其最小编号等于V左子树上结点的最大编号加1。这时是按()编号的

- 中序遍历序列
- 前序遍历序列
- 后序遍历序列
- 层次顺序

## 正确答案及解析

答案：B

本题可以用假设法啊，直接假设7个节点，分别按照4种遍历方式给节点编号，看哪种遍历方式的节点编号满足题中的要求即可，见



## 题目 83

下面有关共享内存，说法不正确的是？

- 共享内存和使用信号量一样，属于进程间通信的一种方式。
- 使用shmget函数来创建共享内存
- 尽管每个进程都有自己的内存地址，不同的进程可以同时将同一个内存页面映射到自己的地址空间中，从而达到共享内存的目的
- 共享内存提供了同步机制，在第一个进程结束对共享内存的写操作之前，会有自动机制可以阻止第二个进程开始对它进行读取

## 正确答案及解析

答案：D

共享内存并没有提供同步机制。为了防止冲突，用信号量的方式来控制访问临界区的资源！

## 题目 84

有作业控制块JCB连成一串而形成的排队队列称为（）。

- 挂起队列
- 阻塞队列
- 就绪队列
- 后备队列

## 正确答案及解析

答案：D

三种调度模式：

- 高级调度（作业调度或长程调度JCB）：后备队列
- 中级调度（中程调度 pcb）：就绪队列，阻塞队列，
- 低级调度（进程调度或短程调度 pcb）：挂起队列 or 就绪队列 or 阻塞队列

## 题目 85

$$\int x \sec^2 x dx = \int x d(\quad)$$

- $\tan(x)$
- $\sec^2(x)$
- $-\tan(x)$
- $-\sec^2(x)$

## 正确答案及解析

答案：C

## 题目 86

如下代码输出结果是什么？

```
#include<stdio.h>
char *myString()
{
    char buffer[6] = {0};
    char *s = "Hello World!";
    for (int i = 0; i < sizeof(buffer) - 1; i++)
    {
        buffer[i] = *(s + i);
    }
    return buffer;
}
int main(int argc, char **argv)
{
    printf("%s\n", myString());
    return 0;
}
```

- Hello
- Hello World!
- Well
- 以上全部不正确

## 正确答案及解析

答案：D

函数 `char *myString()` 中没有使用 `new` 或者 `malloc` 分配内存，所有 `buffer` 数组的内存区域在栈区

随着 `char *myString()` 的结束，栈区内存释放，字符数组也就不存在了，所以会产生野指针，输出结果未知

## 题目 87

以下 GPU 缓冲区中哪个是深度缓冲区：()

- frame buffer
- z buffer
- color buffer
- stencil buffer

## 正确答案及解析

答案：B

## 题目 88

下列关于集中式总线解决方式的叙述中正确的是()

- 集中式串行链接,查询所有部件都用一条"总线请求"线
- 集中式定时查询,所有部件共用一条"总线忙"线
- 集中式独立请求,查询所有部件都用一条"总线请求"线
- 集中式定时查询,所有部件都用一条"总线请求"线

## 正确答案及解析

答案：ABD

集中式总线请求方案有三种，定时查询、串行连接和独立请求，定时查询和串行连接所有部件都用一条"总线请求"线，但是定时查询是由CPU去定时查询总线上的部件，而串行连接是部件去请求CPU。独立请求每个部件均有一条 "总线请求"线。

## 题目 89

二进制地址为011011110000，大小为（4）10和（16）10块的伙伴地址分别为：？，？。

### 正确答案及解析

答案：011011110100, 011011100000

均为十进制，大小分别为 4 和 16

在管理内存分配时提到了“伙伴块”的概念。

Linux伙伴地址的定义：

1. 两个块的大小相同；
2. 两个块相邻；
3. 两个块合并成一个更大的块时，首地址必须是块大小的整数倍。

我觉得这里题目这样描述会更好些：

“大小为4（十进制）和16（十进制）块的伙伴地址分别为”。**其实这里可以把块当做字节理解**，因为题目给的是二进制地址，况且块大小也没有说明。

假设图中一个方框为1字节的块。011011110000 = 0x6F0

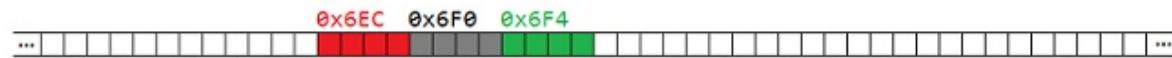
#### 1. 4字节块

与0x6F0相邻的2个4字节块首地址分别为：0x6EC、0x6F4。

若与前边的0x6EC块合并，则合并成的8字节块首地址为0x6EC，但它不是8的整数倍。(011011101100)

若与后边的0x6F4块合并，则合并成的8字节块首地址为0x6F0，它是8的整数倍。

因此0x6F0的4字节块的伙伴地址为：0x6F4，即011011110100



#### 2. 16字节块

同理，相邻两个16字节块首地址分别为：0x6E0、0x700。

若与前边的0x6E0块合并，则合并成的32字节块首地址0x6E0刚好是32的整数倍。(011011100000)

若与后边的0x700块合并，则合并成的32字节块首地址为0x6F0，但它不是32的整数倍。

因此0x6F0的4字节块的伙伴地址为：0x6E0，即011011100000



个人理解，欠妥的地方还望指正。

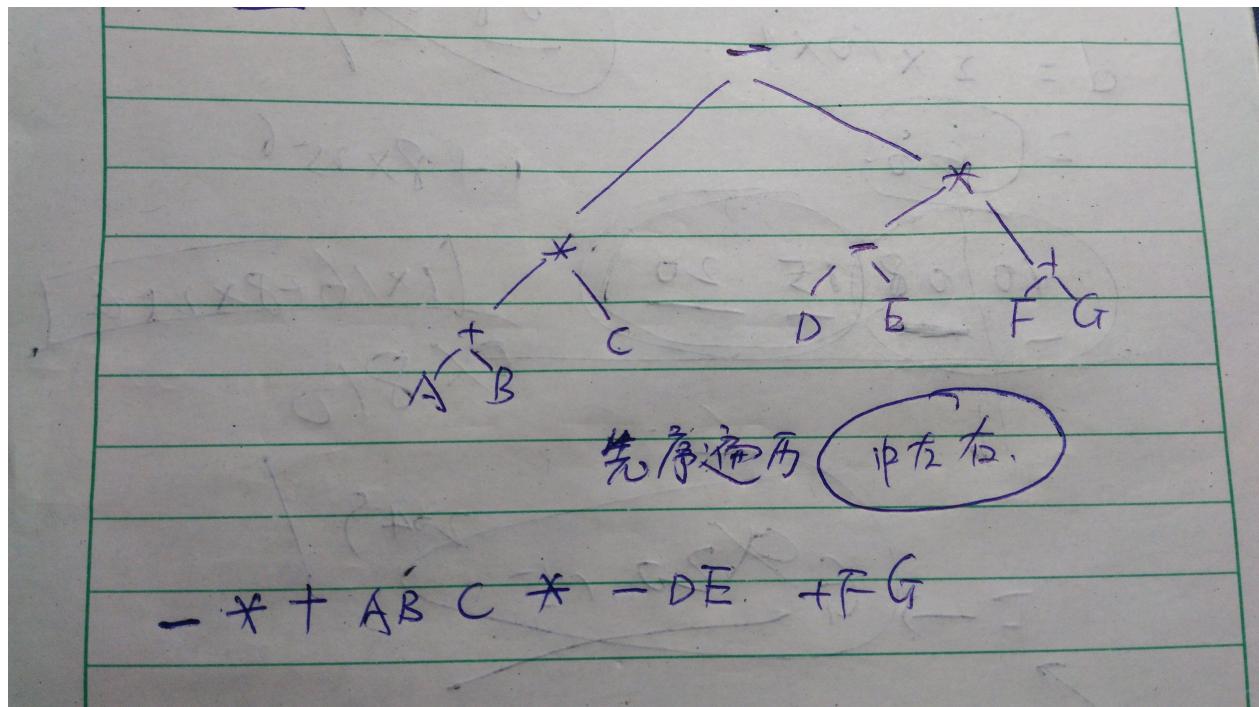
希望对同学有所帮助。

## 题目 90

写出表达式  $((A+B)*C-(D-E)*(F+G))$  的前缀表达式？。

正确答案及解析

答案：  $- * + A B C * - D E + F G$



## 题目 91

任一查找树(二叉分类树)的平均查找时间都小于用顺序查找法查找同样结点的线性表的平均查找时间()

正确答案及解析

答案：错的

只有左子树或者右子树的BST就是一样的了

## 题目 92

在重载运算符函数时，下面（）运算符必须重载为类成员函数形式（）

- +
- -
- ++
- ->

## 正确答案及解析

答案：D

只能使用成员函数重载的运算符有： =、()、[]、->、new、delete 。

## 题目 93

面向对象程序设计思想的主要特征不包括（）

- 封装性
- 多态性
- 继承性
- 模板

## 正确答案及解析

答案：D

面向对象程序语言的三大特征分别是：1.封装，2.继承，3.多态

## 题目 94

对任何数据结构链式存储结构一定优于顺序存储结构()

## 正确答案及解析

答案：错的

查找则用数组好，添加和删除则用链表。

## 题目 95

下列关于树的深度优先搜索算法描述错误的是？

- 按照某种条件往前试探搜索，如果前进中遭到失败，则退回头另选通路继续搜索，直到找到条件的目标为止。
- 先访问该节点所有的子节点，遍历完毕后选取它未访问过的子节点重复上述过程，直到找到条件的目标为止。
- 假设数的顶点数为V，则算法的复杂度为O(V)
- 深度优先算法非常适合使用递归来实现

### 正确答案及解析

答案：B

A选项讲的是深搜，B选项讲的是广搜啊

## 题目 96

下列各种操作的时间中，哪一个不属于活动头硬盘的存取访问时间？

- 寻道时间
- 旋转延迟时间
- 定位时间
- 传送时间

### 正确答案及解析

答案：C

硬盘的存取访问时间为三个部分：

寻道时间Ts，旋转延迟时间Tr和传送时间Tt

## 题目 97

设有一个10阶的对称矩阵A,采用压缩存储方式,以行序为主存储,a11为第一元素,其存储地址为1,每个元素占一个地址空间,则a85的地址为()

- 13
- 33
- 18
- 40

## 正确答案及解析

答案：B

数组下标从1开始,只存储其下三角形元素,在a8,5的前面有7行,第1行有1个元素,第2行有2个元素,...,第7行有7个元素,这7行共有 $(1+7) \times 7 / 2 = 28$ 个元素,在第8行中,a8,5的前面有4个元素,所以,a8,5前有 $28+4=32$ 个元素,其地址为33。

## 题目 98

假设二叉排序树的定义是：1、若它的左子树不为空，则左子树所有节点均小于它的根节点的值；2、若右子树不为空，则右子树所有节点的值均大于根节点的值；3、它的左右子树也分别为二叉排序树。下列哪种遍历之后得到一个递增有序数列()

- 前序遍历
- 中序遍历
- 后序遍历
- 广度遍历

## 正确答案及解析

答案：B

## 题目 99

某指令流水线由 5 段组成，各段所需要的时间分别是： $t$ 、 $3t$ 、 $t$ 、 $2t$  和  $t$ 。问如果连续执行 10 条指令，则吞吐率是多少？

## 正确答案及解析

答案：0.2857/t

参考博客

自己画出指令执行的时空图，可以看出执行第n条指令的时间是： $8t+(n-1)*3t$

而吞吐率=指令/时间 =  $10/(8t+9*3t)=10/35*t=0.2857t$

## 题目 100

通过文件名存取文件时，文件系统内部的操作过程是通过？

- 文件在目录中查找文件数据存取位置。
- 文件名直接找到文件的数据，进行存取操作。
- 文件名在目录中查找对应的i节点，通过i节点存取文件数据。
- 文件名在目录中查找对应的超级块，在超级块查找对应i节点，通过i节点存取文件数据

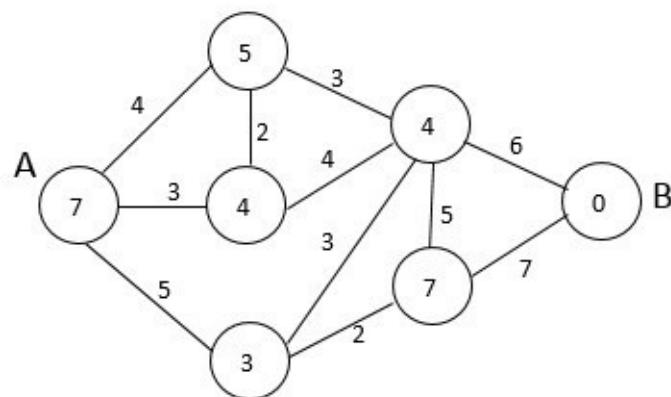
## 正确答案及解析

答案：C

如果一个文件有多个数据块，这些数据块很可能不是连续存放的，应该如何寻址到每个块呢？实际上，根目录的数据块是通过其inode中的索引项Blocks[0]找到的，事实上，这样的索引项一共有15个，从Blocks[0]到Blocks[14]，每个索引项占4字节。前12个索引项都表示块编号，例如上面的例子中Blocks[0]字段保存着24，就表示第24个块是该文件的数据块，如果块大小是1KB，这样可以表示从0字节到12KB的文件。如果剩下的三个索引项Blocks[12]到Blocks[14]也是这么用的，就只能表示最大15KB的文件了，这是远远不够的，事实上，剩下的三个索引项都是间接索引。

## 题目 101

图中每个圆圈是一个补给站，存储着一定数量的汽油（在圈中标识），每个圈之间的路上标识了这段路需要消耗的汽油量，一辆小车从A点出发，在图上随意行走，到达某个补给站后，可以获得这个补给站的所有汽油，则其到B点后最多剩余的汽油量是 \_\_\_\_\_。



### 正确答案及解析

答案：10

总共是  $7+1+3+1+0+5-7=10$

可以使用A\*算法获得

## 题目 102

给定如下C程序：

```
typedef struct node_s{  
    int item;  
    struct node_s* next;  
}node_t;  
node_t* reverse_list(node_t* head)  
{  
    node_t* n=head;  
    head=NULL;  
    while(n){  
        _____  
    }  
    return head;  
}
```

## 正确答案及解析

答案： node\_t\* m=n; n=n->next; m->next=head; head=m;

## 题目 103

若有定义int (\*pt) [3]；则下列说法正确的是：

- 定义了基类型为int的三个指针变量
- 定义了基类型为int的具有三个元素的指针数组pt
- 定义了一个名为\*pt、具有三个元素的整型数组
- 定义了一个名为pt的指针变量，它可以指向每行有三个整数元素的二维数组

## 正确答案及解析

答案：D

`int (*pt)[3]`，首先看括号内，`*pt`说明`pt`是一个指针，其指向的内容是`int[3]`，具有3个`int`元素的数组。

D选项说，可以指向每行有三个整数元素的二维数组，即`int[][][3]`

```
int(*pt)[3] = NULL;  
int arr[2][3] = {0};  
pt = arr;
```

## 题目 104

实现虚拟存储的目的是（）。

- 实现存储保护
- 事项程序浮动
- 扩充辅存容量
- 扩充主存容量

## 正确答案及解析

答案：D

## 题目 105

下面程序应该输出多少？

```

char *c[] = { "ENTER", "NEW", "POINT", "FIRST" };
char **cp[] = { c+3, c+2, c+1, c };
char ***cpp = cp;

int main(void)
{
    printf("%s", ***++cpp);
    printf("%s", *--*++cpp+3);
    printf("%s", *cpp[-2]+3);
    printf("%s\n", cpp[-1][-1]+1);
    return 0;
}

```

## 正确答案及解析

答案：POINTERSTEW

`***++cpp` : 首先 `cpp` 移动到 `cp+1` 的地方 然后取两次`*` 相当于 `**cp+1` 结果就是 `POINT` 移动 `cpp` 到 `cp+1` 的位置

`*--*++cpp+3` : 首先 `cpp` 移动到 `cp+2` 的地方 然后取`*` 相当于 `*--cp[2]+3` 然后 `--cp[2]` 的话相当于 `--(c+1)` 就等于 `c` 就变成了 `*c+3` 输出 `ER` 移动 `cpp` 到 `cp+2` `cp[2]` 变成 `c`

`*cpp[-2]+3` : 首先 `cpp[-2] = *(cpp-2)` [`cpp=cp+2`] = `**cp+3` 结果 `S` `T` 没有移动

`cpp[-1][-1]+1` : 首先 `cpp[-1] -> *(cp+1) -> c+2` 然后 `(c+2)[-1] -> (c+2)[-1]` `-> *(c+2-1) -> c[1]` 然后 `c[1]+1` 就变成了 `EW`

结果就是 `A`

这个题目中 中间 `cpp` 变动两次，`cp[2]` 变动一次 特别注意 `cp[2]` 是数组 里面 内容可以变动

## 题目 106

虚拟存储的容量受到下列哪一个因素的限制影响最大？

- 磁盘空间大小
- 物理内存大小
- 数据存放的实际地址
- 计算机地址位数

## 正确答案及解析

答案：B

本题问的是影响最大的因素，所以选内存大小。如果有影响的因素，则有内存大小，外存大小，计算机寻址位数。

## 题目 107

Which of the following is(are) true about providing security to database servers ?

Select all that apply

- Do not host a database server on the same server as your web server
- Do not host a database server on a server based system
- Do not use blank password for SA account
- Employ a centralized administration model

## 正确答案及解析

答案：ABCD

## 题目 108

Unix系统中，下列哪些可以用于进程间的通讯：（ ）

- socket
- 共享内存
- 消息队列
- 信号量

## 正确答案及解析

答案：ABCD

## 题目 109

以下数字在表示为double（8字节的双精度浮点数）时存在舍入误差的有（）。

- 2的平方根
- 10的30次方
- 0.1
- 0.5
- 100

## 正确答案及解析

答案：ABC

A毫无疑问不用说

8字节的共64位，按照标准的浮点数表示方法，应该是1位符号位，11位指数位，52位尾数位

对于B ( $2^{90} < B < 2^{100}$ )来说，指数位是够了，但是尾数位会不会够呢？  
 $B = 2^{30} \times 5^{30}$  也就是说将B表示成二进制后，其后30位全为0，从其第一个不为0到最后一个不为0的二进制表示位中，至少需要 $90 - 30 = 60$ 位来存储，而其尾数位只有52位，必然会产生舍入误差，所以B是的

对于C来说，将C表示成二进制便知  $10[0.1] = 2[0.00011001100110011\dots]$ ，亦为无限循环小数，所以将0.1表示成二进制的double型必然也会产生舍入误差

## 题目 110

在mysql中，与语句 `SELECT * FROM book b WHERE b.book_num NOT BETWEEN 200 AND 300;` 等价的有

## 正确答案及解析

答案： `SELECT * FROM book b WHERE b.book_num < 200 OR b.book_num > 300`

注意BETWEEN AND的范围，上下都取等于。

## 题目 111

随着装填因子 $a$ 的增大,用闭哈希法解决冲突,其平均搜索长度比用开哈希法解决冲突时的平均搜索长度增长得慢()

## 正确答案及解析

答案：错的

开哈希表-----链式地址法

闭哈希表-----开放地址法

装填因子增大，意味着哈希表的空间利用率在增大。

开哈希和闭哈希主要的区别在于，随着哈希表的密集度提高，使用闭哈希时，不仅会与相同哈希值的元素发生冲突，还容易与不同哈希值的元素发生冲突；而开哈希则不受哈希表疏密与否的影响，始终只会与相同哈希值的元素冲突而已。所以在密集度变大的哈希表中查找时，显然开哈希的平均搜索长度不会增长。

## 题目 112

进程从CPU退下时，将"现场"保存在系统栈内。

## 正确答案及解析

答案：错的

错，保存在任务栈中，系统栈要给下一个要运行的进程用

## 题目 113

采用深度优先搜索或拓扑排序算法可以判断出一个有向图中是否有环(回路)()

### 正确答案及解析

答案：对的

深度优先搜索只要在其中记录下搜索的节点数n，当n大于图中节点数时退出，并可以得出有回路

若有回路，则拓扑排序访问不到图中所有的节点，所以也可以得出回路

## 题目 114

对下列常见的各种网络术语，描述错误的是？

- DNS（域名系统）是一种用于TCP/IP应用程序的分布式数据库，因此它在TCP/IP体系结构中处于应用层。
- TFTP是一种文件传递应用程序，它使用的传输层协议是TCP
- Telnet是标准的提供远程登录功能的应用，可以在不同OS系统的主机之间运行
- Ping是对两个TCP/IP系统连通性进行测试的基本工具，它利用ICMP进行基本的请求和应答

### 正确答案及解析

答案：B

TFTP为简单文件传输协议，是基于UDP实现的，故传输层协议为UDP

## 题目 115

代码生成阶段的主要任务是（ ）

- 把高级语言翻译成汇编语言
- 把高级语言翻译成机器语言
- 把中间代码转换成依赖具体机器的目标代码

- 把汇编语言翻译成机器语言

## 正确答案及解析

答案：C

编译程序的工作过程一般划分为五个阶段：词法分析、语法分析、语义分析与中间代码产生、优化、目标代码生成。所以选C。

## 题目 116

求下面函数的返回值。

```
int func(x)
{
    int countx = 0;
    while (x)
    {
        countx++;
        x = x & (x - 1);
    }
    return countx;
}
```

假定  $x = 9999$ 。

## 正确答案及解析

答案：8

$x=x&(x-1)$ ； $\&$  是位运算符，需要将  $x$  转化成二进制计算，当  $\&$  两边都为1时，为1，否则为0，该题相当于求  $x$  的二进制式中含有1的个数，

$x&(x-1)$  求的是二进制中1的个数

$x|(x+1)$  求的是二进制中0的个数

## 题目 117

把校园中同一区域的两张不同比例尺的地图叠放在一起，并且使其中较小尺寸的地图完全在较大尺寸的地图的覆盖之下。每张地图上都有经纬度坐标，显然，这两个坐标系并不相同。我们把恰好重叠在一起的两个相同的坐标称之为重合点。下面关于重合点的说法中正确的是？

### 正确答案及解析

答案：必然有且只有一个重合点

假设地图有不止一个重合点，把任意两个重合点连成一条直线，则在大地图和小地图上的直线也是重合的，且长度相同。这导致两个地图的比例相等，与题设相矛盾。所以两个地图至多有一个重合点。

## 题目 118

通用多态是指

- 强制多态和包含多态
- 重载多态和强制多态
- 参数多态和重载多态
- 包含多态和参数多态

### 正确答案及解析

答案：D

```

//1.参数多态

//包括函数模板和类模板

//2.包含多态 virtual

class A{

    virtual void foo() { printf("A virtual void foo()"); }

};

class B : public A {

    void foo() { printf("B void foo()"); }

};

void test() {

    A *a = new B();

    a->foo(); // B void foo()

}

//3.重载多态

//重载多态是指函数名相同，但函数的参数个数或者类型不同的函数构成多态

void foo(int);

void foo(int, int);

//4.强制多态

//强制类型转换

```

## 题目 119

下面程序的输出结果是 \_\_\_\_\_。

```
#include <iostream.h>
#define SQR(A) A*A
void main() {
    int x=6,y=3,z=2;
    x/=SQR(y+z)/SQR(y+z);
    cout<<x<<endl;
}
```

## 正确答案及解析

答案：0

宏定义是一个很看重括号的东西

1. #define f(x) x\*x 这里 f(x+y) 就会被翻译成 x+y\*x+y 为什么，因为你没有添加括号啊宏定义只是简单的替换不会替你加括号

2. #define f(x) (x)\*(x) 这里 f(x+y) 就会翻译成 (x+y) \* (x+y) 就是这么回事

回到题上

上述式子等价为  $x = y + z * y + z / y + z * y + z$ ，再加上 /= 优先级最低，所以  $x = 3 + 6 + 2 / 3 + 6 + 2$  所以  $x = 0$

## 题目 120

程序动态链接的时刻是（ ）。

- 编译时
- 装入时
- 调用时
- 紧凑时

## 正确答案及解析

答案：B

B C都可以的 有装载时动态链接 也有运行时动态链接。。。

## 题目 121

一个具有8个顶点的连通无向图，最多有（）条边

正确答案及解析

答案：28

8个点中任选择两个，都可以有一条边，最多  $8 * 7 / 2 = 28$

## 题目 122

在分时操作系统中,进程调度经常采用( )算法

- 先来先服务
- 最高优先权
- 时间片轮转
- 随机

正确答案及解析

答案：C

时间片轮转。前提是进程资源是可抢占的。可是广泛采用的一种方式

其他选项说明：

先来先服务是最简单的进程调用方式，只需要维护一个队列即可。也可以说是最公平的调用方式。但是容易引起一个问题就是前面来的进程耗费较多的时间导致后面来的需要时间少的进程长时间得不到响应，会引起用户的不满

最高优先权。这个也容易引起低优先权的用户长时间得不到响应。想想军队里，战争的时候，越高级的长官越有必要占用较多的通信或者计算资源，也是合理的。

随机就不做评价了吧，跟抛硬币似的

## 题目 123

关于函数的描述正确的是 \_\_\_\_。

- 虚函数是一个**static**型的函数
- 派生类的虚函数与基类的虚函数具有不同的参数个数和类型
- 虚函数是一个非成员函数
- 基类中说明了虚函数后，派生类中起对应的函数可以不必说明为虚函数

## 正确答案及解析

答案：D

- 在派生类中重新定义该虚函数时，关键字**virtual**可以写也可以不写。
- 一个虚函数无论被公有继承多少次，它仍然保持其虚函数的特性。
- 虚函数必须是其所在类的成员函数，而不能是友元函数，也不能是静态成员函数（**static**）

## 题目 124

以下代码输出什么？

```
int a =1, b =32 ;  
printf("%d,%d", a<<b, 1<<32);
```

## 正确答案及解析

答案：1,1

执行 $a \ll b$ 时，编译器会先将 $b$ 与31进行and操作，以限制左移的次数小于等于31。 $b \& 31 = 0$ ，则 $a \ll b = 1$

执行 $1 \ll 32$ 时，编译器直接执行算术左移的操作。

来自 <http://blog.csdn.net/hgl868/article/details/7058909>

## 题目 125

下面数据结构能够支持随机的插入和删除操作、并具有较好的性能的是 \_\_\_\_。

- 数组和链表
- 链表和哈希表
- 哈希表和队列
- 队列和堆栈
- 堆栈和双向队列
- 双向队列和数组

## 正确答案及解析

答案：B

数组和队列不方便插入和删除，因为队列在中间不好插入和删除，队列只在端点插入和删除，所以队列要插入和删除要移位许多

## 题目 126

设一个系统中有5个进程，它们的到达时间和服务时间如下，A的到达时间为0，服务时间为3；B的到达时间为2，服务时间为6；C的到达时间为4，服务时间为4；D的到达时间为6，服务时间为5；E的 到达时间为8，服务时间为2，忽略I/O以及其他开销时间，若分别按先来先服务（FIFO）进行CPU调度，其平均周转时间为？

## 正确答案及解析

答案：8.6

先来先服务调度算法					
进程名	到达时间	服务时间	开始执行时间	完成时间	周转时间
A	0	3	0	3	3
B	2	6	3	9	7
C	4	4	9	13	9
D	6	5	13	18	12
E	8	2	18	20	12

周转时间 = 完成时间 - 到达时间  
 平均周转时间 = 所有进程周转时间 / 进程数 =  $(3+7+9+12+12) / 5 = 8.6$

## 题目 127

10.1.0.1/17的广播地址是（ ）

## 正确答案及解析

答案：10.1.127.255

## 题目 128

以下有关Http协议的描述中，正确的有？

- post请求一般用于修改服务器上的资源，对发送的消息数据量没有限制，通过表单方式提交
- HTTP返回码302表示永久重定向，需要重新URI
- 可以通过206返回码实现断点续传

- HTTP1.1实现了持久连接和管线化操作以及主动通知功能，相比http1.0有大福  
性能提升

## 正确答案及解析

答案：ACD

- HTTP /302 redirect: 302 代表暂时性转移(Temporarily Moved )。
- HTTP/206 “Partial Content”响应是在客户端表明自己只需要目标URL上的部分资源的时候返回的.这种情况经常发生在客户端继续请求一个未完成的下载的时候(通常是当客户端加载一个体积较大的嵌入文件,比如视屏或PDF文件 ),或者是客户端尝试实现带宽遏流的时候.

## 题目 129

如果系统现在需要在一个很大的表上创建一个索引，你会考虑那些因素，如何做以尽量减小对应用的影响？

- 增大 sort\_area\_size (8i) /pga\_aggregate\_target (Arrayi) 值
- 如果表有分区（一般大表都要用到分区的），按分区逐个建索引，如果是本地索引的话
- 系统空闲的时候建。
- 把日志文件放到另一个地方

## 正确答案及解析

答案：ABC

不选D，改日志位置需要down机，应用在这个时间用不了

## 题目 130

堆肯定是一棵平衡二叉树()

## 正确答案及解析

答案：错的

堆是二叉树，但不保证是平衡二叉树，因为堆中左右子树的高度差并不保证小于等于1。

## 题目 131

堆是满二叉树()

### 正确答案及解析

答案：错的

堆是完全二叉树，但不是满二叉树。

## 题目 132

在用堆排序算法排序时，如果要进行增序排序，则需要采用"大根堆"()

### 正确答案及解析

答案：对的

因为大根堆每次生成的根都会跟最右侧没有排序的叶子节点进行交换，从而使得越大的元素越放在后面。这个特性使用数组的结构能够很清晰的表现出来。最终得到了升序排列。

如果是小根堆，则每次拿到最小的跟最右侧未排过序的叶子节点进行交换，最终得到的序列是递减的。

## 题目 133

(101,88,46,70,34,39,45,58,66,10)是堆()

### 正确答案及解析

答案：对的

最小堆：直接父节点比两个子节点都小。

最大堆：直接父节点比两个子节点都大。

解析得到这是大堆，故是对的

## 题目 134

有 1000 个无序的整数，希望使用最快的方式找出前 50 个最大的，最佳的选择是（ ）

- 冒泡排序
- 基数排序
- 堆排序
- 快速排序

## 正确答案及解析

答案：C

1、快速排序：在最理想的情况下，即划分可以使得每次分到  $n/2$  的两个序列，复杂度为  $O(n \log n)$ 。

2、堆排序：无论什么情况都是  $O(n \log n)$ ，当然还有建堆的时间  $O(n)$ ，所以为  $n+n \log n$ ，但是，本题只是要前五十个，所以堆排序只需要执行 50 次就够了： $n+50 \log n$ 。

3、本题的条件是 1000 个无序整数， $1000+50 \log 1000 < 1000 \log 1000$ ，所以自然是堆排序最好。

p.s.：堆排序只需要一个  $n$  大小的二叉树就行了，快速排序除了  $n$  大小的数组，每次递归最少还得有  $\log n$  的栈，所以堆排序的空间复杂度也优于快速排序。

## 题目 135

在堆排序算法中我们用一个数组 A 来模拟二叉树 T，如果该 A[0] 存放的是 T 的根节点，那么  $A[K](K>0)$  的父亲节点是

- $(K-1)/2$

- $K/2$
- $(K+1)/2$
- 都不对

## 正确答案及解析

答案：A

当数组从0开始时，下标为k的结点的父结点下标为 $(k-1)/2$ ；

当数组从1开始时，下标为k的结点的父结点下标为 $k/2$ ；

## 题目 136

以下序列不是堆的是()

- (100,85,98,77,80,60,82,40,20,10,66)
- (100,98,85,82,80,77,66,60,40,20,10)
- (10,20,40,60,66,77,80,82,85,98,100)
- (100,85,40,77,80,60,66,98,82,10,20)

## 正确答案及解析

答案：D

最大（小）堆，所有节点都大（小）于其孩子节点， $i$  位置节点的孩子节点为  $2i$  和  $2i+1$ ， $i$  节点的父节点为  $i/2$  注意  $i$  是从1开始的。

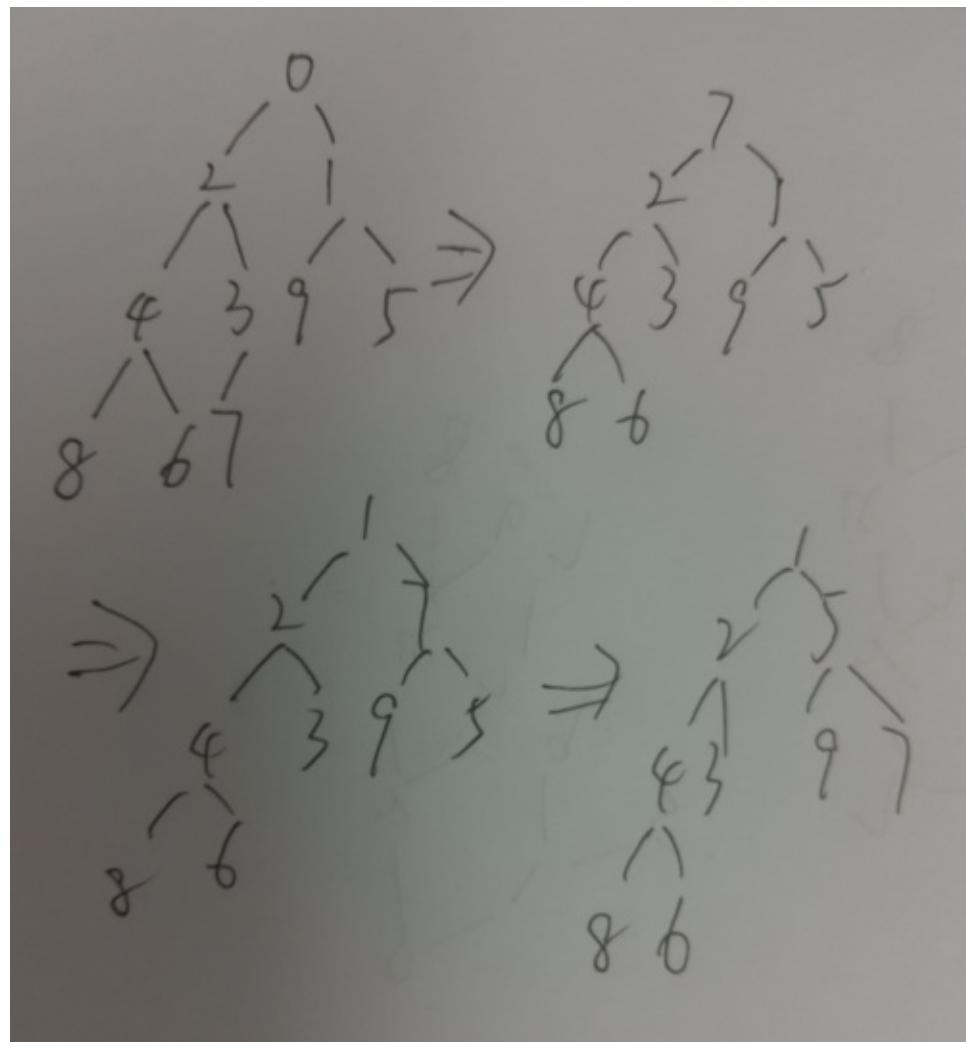
## 题目 137

【0、2、1、4、3、9、5、8、6、7】是以数组形式存储的最小堆，删除堆顶元素0后的结果是（）

- 【2、1、4、3、9、5、8、6、7】
- 【1、2、5、4、3、9、8、6、7】
- 【2、3、1、4、7、9、5、8、6】
- 【1、2、5、4、3、9、7、8、6】

## 正确答案及解析

答案：D



## 题目 138

就分类算法所用的辅助空间而言,堆分类、快速分类和归并分类的关系是()

## 正确答案及解析

答案： 堆分类<快速分类<归并分类

堆分类  $O(1)$  用于交换的时候临时变量

快速分类  $O(\log n)$  递归保存现场

归并分类  $O(n)$  用元素个数同长度空间保存归并结果

## 题目 139

下述二叉树中,哪一种满足性质:从任一结点出发到根的路径上所经过的结点序列按其关键字有序()

- 二叉排序树
- 哈夫曼树
- AVL树
- 堆

## 正确答案及解析

答案：D

- 一：二叉排序树的左子树节点值都小于根节点值，右子树节点值都大于根节点，因此假如根节点值为10，其左节点值为5，其左节点的右节点值为8，那么从右节点到跟节点的值依次为8 5 10，显然不是有序的
- 二：哈夫曼树是带权路径最小的二叉树，也不是
- 三：AVL树是二叉平衡树，只不过其左右子树的高度差有限制，在1之内，由一只非有序
- 四：堆是一种完全二叉树，其有大顶堆和小顶堆的分别，大顶堆是指其每个节点的值都大于其左右孩子的值（小顶堆反之），因此从任一节点到根节点是升序排列的（小顶堆反之）

## 题目 140

最坏情况下 insert sort, quick sort ,merge sort 的复杂度分别是多少？

- $O(n^2), O(n \log n), O(n^2)$
- $O(n^2), O(n^2), O(n \log n)$
- $O(n^2), O(n \log n), O(n \log n)$
- $O(n \log n), O(n \log n), O(n \log n)$

## 正确答案及解析

答案：B

1 : 简单选择	最好时间 $O(n^2)$	平均时间 $O(n^2)$	最坏时间 $O(n^2)$
2 : 直接插入	最好时间 $O(n)$	平均时间 $O(n^2)$	最坏时间 $O(n^2)$
3 : 冒泡排序	最好时间 $O(n)$	平均时间 $O(n^2)$	最坏时间 $O(n^2)$
4 : 希尔排序	最好时间 $O(n)$	平均时间 $O(\log n)$	最坏时间 $O(n^s)$
$1 < s < 2$			
5 : 快速排序	最好时间 $O(n \log n)$	平均时间 $O(n \log n)$	最坏时间 $O(n^2)$
6 : 堆排序	最好时间 $O(n \log n)$	平均时间 $O(n \log n)$	最坏时间 $O(n \log n)$
7 : 归并排序	最好时间 $O(n \log n)$	平均时间 $O(n \log n)$	最坏时间 $O(n \log n)$

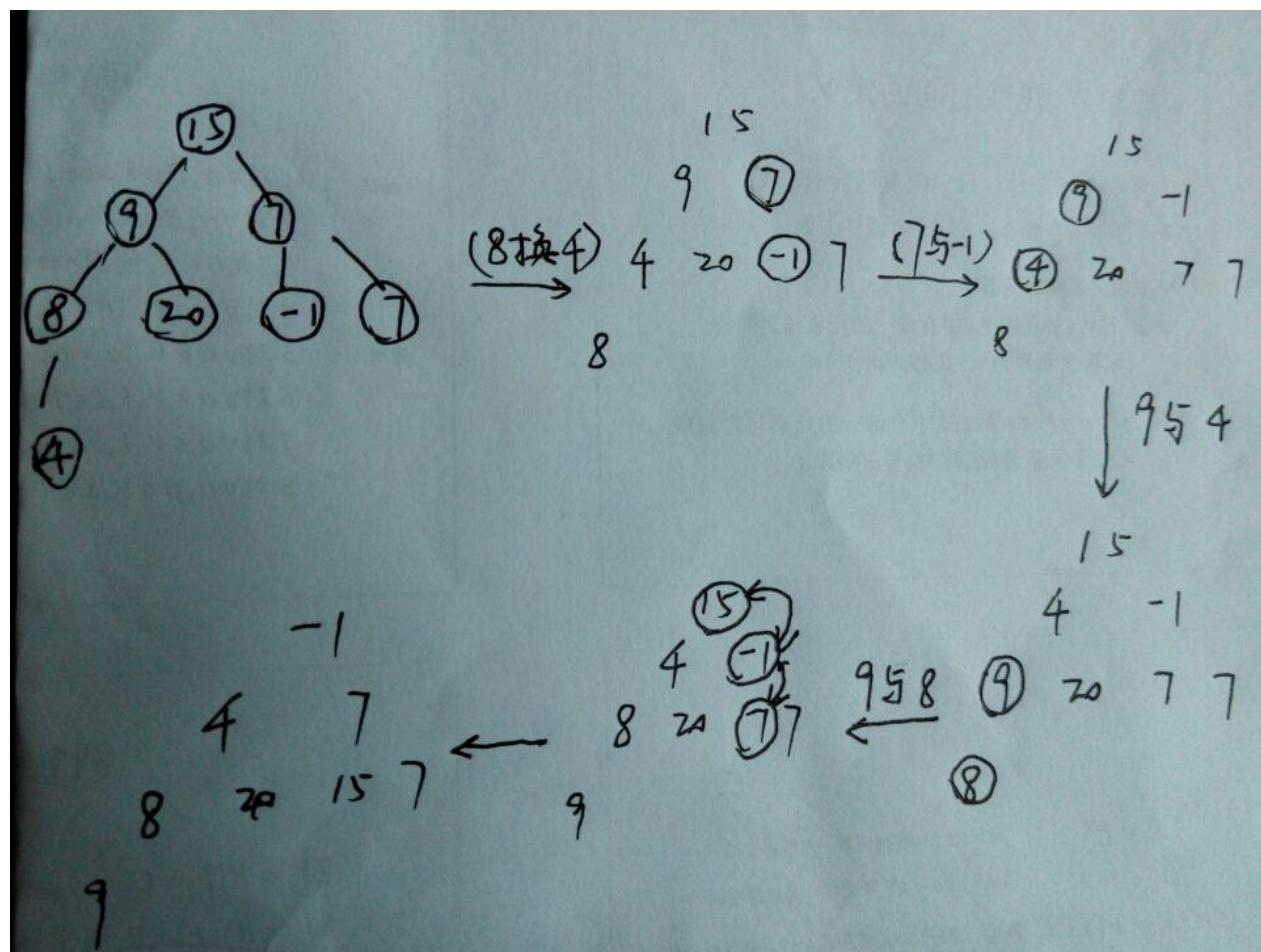
## 题目 141

有一组数据(15,9,7,8,20,-1,7,4),用堆排序的筛选方法建立的初始堆为()

- -1,4,8,9,20,7,15,7
- -1,7,15,7,4,8,20,9
- -1,4,7,8,20,15,7,9
- ABC均不对

### 正确答案及解析

答案：C



## 题目 142

关于序列 16 14 10 8 7 9 3 2 4 1 的说法下面哪一个正确 ( )

- 大顶堆
- 小顶堆
- 不是堆
- 二叉排序树

## 正确答案及解析

答案：A

大顶堆，在 $n$ 位置上的数要比在 $2n+1$ 和 $2n+2$ 位置上的数大( $n$ 从 0 开始)

## 题目 143

下面有关C++静态数据成员，说法正确的是？

- 不能在类内初始化
- 不能被类的对象调用
- 不能受private修饰符的作用
- 可以直接用类名调用

## 正确答案及解析

答案：D

通常静态数据成员在类声明中声明，在包含类方法的文件中初始化。

初始化时使用作用域操作符来指出静态成员所属的类。

但如果静态成员是整型或是枚举型const，则可以在类声明中初始化！！

如果改成有的静态数据成员是可以直接在类中初始化就对了

## 题目 144

对n个记录的文件进行堆排序，最坏情况下的执行时间是多少？()

- $O(\log 2n)$
- $O(n)$
- $O(n \log 2n)$
- $O(n^*n)$

## 正确答案及解析

答案：C

时间复杂度分析，第一步首先建堆需要用时 $O(n)$ ，第二步对大小为n的堆，取出元素放入数组尾部用时 $O(1)$ ，重新进行保持堆特性为 $O(lgn)$ ，因此 $O(n)+O(nlgn)$ ，总体时间复杂度为 $O(nlgn)$

## 题目 145

下标从1开始，在含有n个关键字的小根堆(堆顶元素最小)中，关键字最大的记录有可能存储在()位置上

- [n/2]
- [n/2]-1
- 1
- [n/2]+2

## 正确答案及解析

答案：D

小根堆中最大的数一定是放在叶子节点上，堆本身是个完全二叉树，完全二叉树的叶子节点的位置大于[n/2]

## 题目 146

下列哪一个关键码序列不符合堆的定义？

- A、C、D、G、H、M、P、Q、R、X
- A、C、M、D、H、P、X、G、O、R
- A、D、P、R、C、Q、X、M、H、G
- A、D、C、M、P、G、H、X、R、Q

## 正确答案及解析

答案：C

C选项中关键码序列用完全二叉树表示后很容易看出，结点值d大于了左子结点值c

## 题目 147

下列关于堆和栈的区别描述错误的有？

- 申请方式的不同，堆是系统自动分配，栈是自己申请
- 栈的大小是固定的，堆的大小受限于系统中有效的虚拟内存
- 栈的空间由系统决定何时释放，堆需要自己决定何时去释放

- 堆的使用容易产生碎片，但是用起来最方便

## 正确答案及解析

答案：A

1. 栈内存操作系统来分配，堆内存由程序员自己来分配。
2. 栈有系统自动分配，只要栈剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

## 题目 148

已知关键字序列5,8,12,19,28,20,15,22是最小堆，插入关键字3，调整后得到的最小堆是()

- 3,8,12,5,20,15,22,28,19
- 3,5,12,19,20,15,22,8,28
- 3,12,5,8,28,20,15,22,19
- 3,5,12,8,28,20,15,22,19

## 正确答案及解析

答案：D

## 题目 149

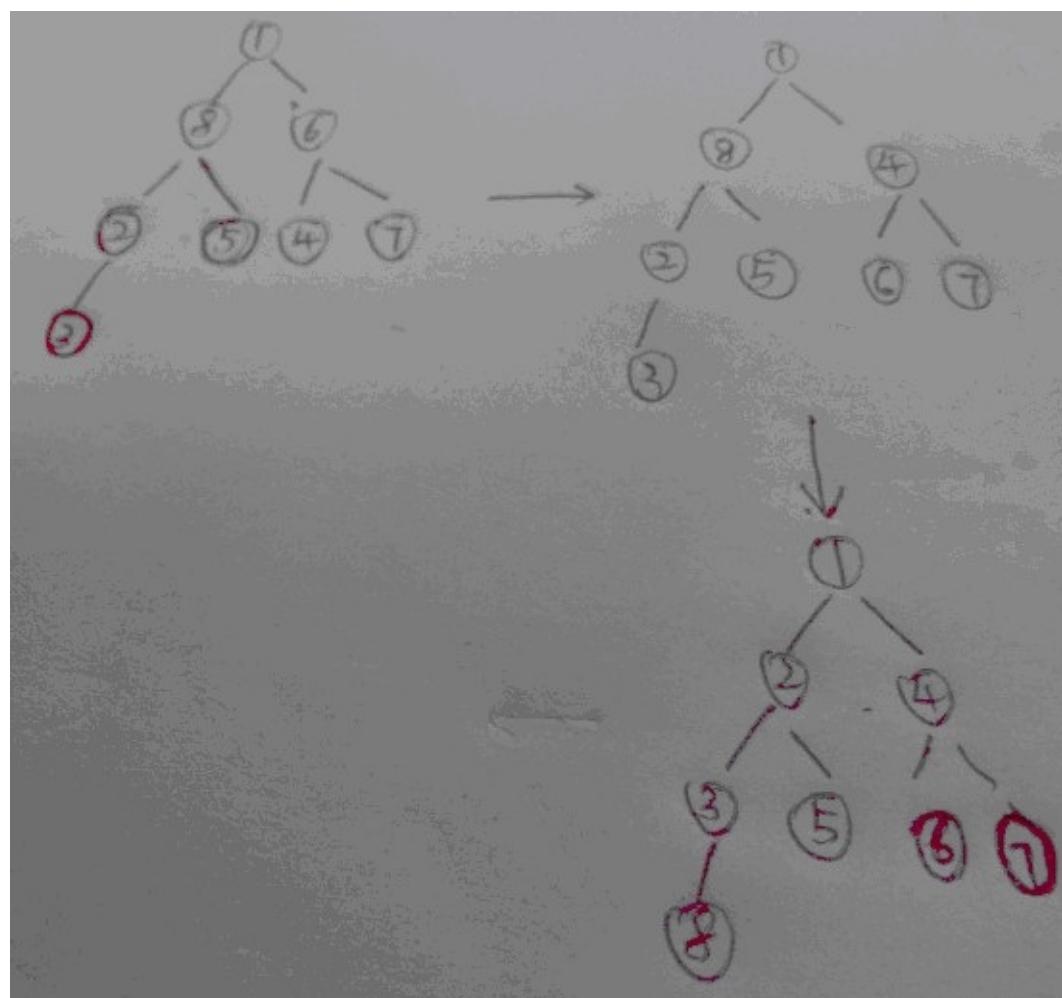
初始序列为1 8 6 2 5 4 7 3一组数采用堆排序，当建堆（小根堆）完毕时，堆所对应的二叉树中序遍历序列为：()

- 8 3 2 5 1 6 4 7
- 3 2 8 5 1 4 6 7
- 3 8 2 5 1 6 7 4
- 8 2 3 5 1 4 7 6

## 正确答案及解析

答案：A

最小堆：先以数组顺序构建一棵完全二叉树，再从第  $n/2 + 1$  个元素开始构建最小堆，再进行中序遍历。



首先从最后一个非端子结点开始(即2) 因为是建小根堆

所以交换子节点中小于自己的最小那个 3比2大——不换 看6与4换

接着8与2换且与3换 最后1不换 就形成了最终结果 然后再中序遍历之

# Python 语言特性

这一个节里主要是练习 Python 语言特性的相关代码。

## 参考资料

[interview\\_Python](#)

## 清单

1. Python的函数参数传递
2. @staticmethod和@classmethod
3. 类变量和实例变量
4. Python中单下划线和双下划线
5. \*args and \*\*kwargs
6. 面向切面编程AOP和装饰器
7. 单例模式
8. Python 函数式编程
9. Python 里的拷贝

## 零散知识点

### 元类(**metaclass**)

这个非常的不常用，但是像 ORM 这种复杂的结构还是会需要的，[详情](#)

## 自省

自省就是面向对象的语言所写的程序在运行时，所能知道对象的类型。

简单一句就是运行时能够获得对象的类型。

比如 `type()`, `dir()`, `getattr()`, `hasattr()`, `isinstance()` .

## 字典推导式

可能你见过列表推导时 #, 却没有见过字典推导式, 在 2.7 中才加入的: `d = {key: value for (key, value) in iterable}`

## 字符串格式化: % 和 .format

`.format` 在许多方面看起来更便利.

对于 % 最烦人的是它无法同时传递一个变量和元组. 你可能会想下面的代码不会有什问题:

`"hi there %s" % name` 但是, 如果 `name` 恰好是 (1,2,3), 它将会抛出一个 `TypeError` 异常.

为了保证它总是正确的, 你必须这样做:

`"hi there %s" % (name,)` # 提供一个单元素的数组而不是一个参数 但是有点丑.

`.format` 就没有这些问题. 你给的第二个问题也是这样, `.format` 好看多了.

你为什么不用它?

- 不知道它(在读这个之前)
- 为了和 Python2.5 兼容(譬如 `logging` 库建议使用 %)

## 迭代器和生成器

参考

## 鸭子类型

“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”我们并不关心对象是什么类型，到底是不是鸭子，只关心行为。比如在 Python 中，有很多 `file-like` 的东西，比如 `StringIO`, `GzipFile`, `socket`。它们有很多相同的方法，我们把它们当作文件使用。又比如 `list.extend()` 方法中，我们并不关心它的参数是不是 `list`, 只要它是可迭代的, 所以它的参数可以是 `list/tuple/dict/字符串/生成器等`. 鸭子类型在动态语言中经常使用，非常灵活，使得 Python 不像 java 那样专门去弄一大堆的设计模式。

## Python 中重载

函数重载主要是为了解决两个问题：

- 可变参数类型
- 可变参数个数

另外，一个基本的设计原则是，仅仅当两个函数除了参数类型和参数个数不同以外，其功能是完全相同的，此时才使用函数重载，如果两个函数的功能其实不同，那么不应当使用重载，而应当使用一个名字不同的函数。好吧，那么对于情况 1，函数功能相同，但是参数类型不同，Python 如何处理？答案是根本不需要处理，因为 Python 可以接受任何类型的参数，如果函数的功能相同，那么不同的参数类型在 Python 中很可能是相同的代码，没有必要做成两个不同函数。那么对于情况 2，函数功能相同，但参数个数不同，Python 如何处理？大家知道，答案就是缺省参数。对那些缺少的参数设定为缺省参数即可解决问题。因为你假设函数功能相同，那么那些缺少的参数终归是需要用的。好了，鉴于情况 1 跟 情况 2 都有了解决方案，Python 自然就不需要函数重载了。

## 新式类和旧式类

[这篇文章](#)很好的介绍了新式类的特性 新式类很早在 2.2 就出现了,所以旧式类完全是兼容的问题, Python3 里的类全部都是新式类. 这里有一个 MRO 问题可以了解下(新式类是广度优先,旧式类是深度优先),里讲的也很多.

### \_\_new\_\_ 和 \_\_init\_\_ 的区别

- \_\_new\_\_ 是一个静态方法, 而 \_\_init\_\_ 是一个实例方法.
- \_\_new\_\_ 方法会返回一个创建的实例, 而 \_\_init\_\_ 什么都不返回.
- 只有在 \_\_new\_\_ 返回一个 cls 的实例时后面的 \_\_init\_\_ 才能被调用.
- 当创建一个新实例时调用 \_\_new\_\_, 初始化一个实例时用 \_\_init\_\_.

PS: \_\_metaclass\_\_ 是创建类时起作用. 所以我们可以分别使用 \_\_metaclass\_\_, \_\_new\_\_ 和 \_\_init\_\_ 来分别在类创建, 实例创建和实例初始化的时候做一些小手脚.

## Python 中的作用域

Python 中，一个变量的作用域总是由在代码中被赋值的地方所决定的。当 Python 遇到一个变量的话他会按照这样的顺序进行搜索：`本地作用域 (Local) → 当前作用域被嵌入的本地作用域 (Enclosing locals) → 全局/模块作用域 (Global) → 内置作用域 (Built-in)`。

## GIL 线程全局锁

线程全局锁(Global Interpreter Lock)，即 Python 为了保证线程安全而采取的独立线程运行的限制，说白了就是一个核只能在同一时间运行一个线程。见 Python 最难的问题 解决办法就是多进程和下面的协程(协程也只是单 CPU,但是能减小切换代价提升性能)。

## 协程

简单点说协程是进程和线程的升级版，进程和线程都面临着内核态和用户态的切换问题而耗费许多切换时间，而协程就是用户自己控制切换的时机，不再需要陷入系统的内核态。Python 里最常见的 `yield` 就是协程的思想！

## 闭包

闭包(closure)是函数式编程的重要的语法结构。闭包也是一种组织代码的结构，它同样提高了代码的可重复使用性。当一个内嵌函数引用其外部作用域的变量，我们就会得到一个闭包。总结一下，创建一个闭包必须满足以下几点：

- 必须有一个内嵌函数
- 内嵌函数必须引用外部函数中的变量
- 外部函数的返回值必须是内嵌函数

感觉闭包还是有难度的，几句话是说不明白的，还是查查相关资料。

重点是函数运行后并不会被撤销，就像 16 题的 `instance` 字典一样，当函数运行完后，`instance` 并不被销毁，而是继续留在内存空间里。

这个功能类似类里的类变量，只不过迁移到了函数上。

闭包就像个空心球一样，你知道外面和里面，但你不知道中间是什么样。

## lambda 函数

其实就是一个匿名函数，为什么叫 lambda？因为和后面的函数式编程有关。

## Python 垃圾回收机制

Python GC 主要使用引用计数（reference counting）来跟踪和回收垃圾。在引用计数的基础上，通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用问题，通过“分代回收”（generation collection）以空间换时间的方法提高垃圾回收效率。

- 引用计数
  - PyObject 是每个对象必有的内容，其中 `ob_refcnt` 就是做为引用计数。当一个对象有新的引用时，它的 `ob_refcnt` 就会增加，当引用它的对象被删除，它的 `ob_refcnt` 就会减少。引用计数为 0 时，该对象生命就结束了。
  - 优点：
    - 简单
    - 实时性
  - 缺点：
    - 维护引用计数消耗资源
- 循环引用
- 标记-清除机制
  - 基本思路是先按需分配，等到没有空闲内存的时候从寄存器和程序栈上的引用出发，遍历以对象为节点、以引用为边构成的图，把所有可以访问到的对象打上标记，然后清扫一遍内存空间，把所有没标记的对象释放。
- 分代技术
  - 分代回收的整体思想是：将系统中的所有内存块根据其存活时间划分为不同的集合，每个集合就成为一个“代”，垃圾收集频率随着“代”的存活时间的增大而减小，存活时间通常利用经过几次垃圾回收来度量。
- Python 默认定义了三代对象集合，索引数越大，对象存活时间越长。
  - 举例：当某些内存块 M 经过了 3 次垃圾收集的清洗之后还存活时，我们就将内存块 M 划到一个集合 A 中去，而新分配的内存都划分到集合 B 中去。当垃圾收集开始工作时，大多数情况都只对集合 B 进行垃圾回收，而对集合 A 进行垃圾回收要隔相当长一段时间后才进行，这就使得垃圾收集机制需要处理的内存少了，效率自然就提高了。在这个过程中，集合 B 中的某些内存块由于存活时间长而会被转移到集合 A 中，当然，集合 A 中实

际上也存在一些垃圾，这些垃圾的回收会因为这种分代的机制而被延迟。

## Python 的 List

[推荐](#)

## Python 的 is

`is` 是对比地址，`==` 是对比值

## read, readline 和 readlines

- `read` 读取整个文件
- `readline` 读取下一行, 使用生成器方法
- `readlines` 读取整个文件到一个迭代器以供我们遍历

## Python 2 和 3 的区别

[Python 2.7.x 与 Python 3.x 的主要差异](#)

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
""" 参考网上资料，关于 Python 的函数参数传递问题
```

总结：

在 Python 中，strings, tuples, numbers 是不可更改的对象，而 list, dict 等则是可以修改的对象

```
"""
```

```
__author__ = '__L1n_w@tch'
```

```
def modify_fun_fail(value):
```

```
    """
```

当一个引用传递给函数的时候，函数自动复制一份引用，这个函数里的引用和外边的引用没有关系

所以这里函数把引用指向了一个不可变对象，当函数返回的时候，外面的引用不会被改变

```
:param value: int()
:return: None
"""
value += 50
print("调用函数 {}, 修改值为 {}".format("modify_fun_fail", value))
```

```
def modify_fun_success(a_list):
```

```
    """
```

函数内的引用指向的是可变对象，对它的操作就和定位了指针地址一样，在内存里进行修改

```
:param a_list: list()
:return: None
"""
a_list.append(1)
print("调用函数 {}, 修改值为 {}".format("modify_fun_success", a_list))
```

```
if __name__ == "__main__":
    number = 30
    print("接下来调用函数 {}, 当前 number 值为 {}".format("modify_fun_fail", number))
    modify_fun_fail(number)
    print("调用完成, number 值为 {}".format(number), end="\n\n")

    number_list = list()
    print("接下来调用函数 {}, 当前 number_list 值为 {}".format("modify_fun_success", number_list))
    modify_fun_success(number_list)
    print("调用完成, number_list 值为 {}".format(number_list))
```

```

#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
""" Python 有 4 个方法, 静态方法(staticmethod), 类方法(classmethod),
实例方法, 普通方法
"""

__author__ = '__L1n_w@tch'

def normal_function():
    print("I am just a 普通方法")

class FunctionTest:
    def method(self, value):
        """
        self 是对实例的绑定, 需要把实例自己传给函数, 调用的时候是这样的 f
        t.method(value), 其实是 method(ft, value)
        :param value: int()
        :return: None
        """
        print("I am just a 实例方法, {}; {}".format(self, value))

    @classmethod
    def class_method(cls, value):
        """
        # cls是对类的绑定
        :param value: int()
        :return: None
        """
        print("I am just a 类方法, {}; {}".format(cls, value))

    @staticmethod
    def static_method(value):
        print("I am just a 静态方法, 打印值 {}".format(value))

if __name__ == "__main__":

```

```
ft = FunctionTest()  
  
normal_function()  
  
ft.method(3.4)  
ft.static_method(3.4)  
ft.class_method(3.4)  
  
FunctionTest.static_method(3.4)  
FunctionTest.class_method(3.4)
```

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""
    关于类变量和实例变量的讨论
"""

__author__ = '__L1n_w@tch'

class VarTest:
    str_var = "I am 类变量"
    list_var = ["I am 类变量"]

if __name__ == "__main__":
    vt1 = VarTest()
    vt2 = VarTest()

    vt1.str_var = "I am 实例变量"
    print("vt1.str_var: {}".format(vt1.str_var))
    print("vt2.str_var: {}".format(vt2.str_var))
    print("VarTest.str_var: {}".format(VarTest.str_var), end="\n")

    # 在实例的作用域里把类变量的引用改变了，就变成了一个实例变量，self.list_var 不再引用 VarTest 的类变量 list_var 了
    vt1.list_var.append("vt1 到此一游")
    print("vt1.list_var: {}".format(vt1.list_var))
    print("vt2.list_var: {}".format(vt2.list_var))
    print("VarTest.list_var: {}".format(VarTest.list_var))
```

```

#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
""" 讨论关于单下划线和双下划线命名的问题

总结：
    __foo__:一种约定,Python内部的名字,用来区别其他用户自定义的命名,以防冲突.
    _foo:一种约定,用来指定变量私有.程序员用来指定私有变量的一种方式.
    __foo:这个有真正的意义:解析器用__className__foo来代替这个名字,以区别和其他类相同的命名.

"""
__author__ = '__L1n_w@tch'

class UnderlineTest:
    def __init__(self):
        self.__super_private = "超级私有"
        self._semi_private = "假私有"

if __name__ == "__main__":
    ut = UnderlineTest()
    try:
        print(ut.__super_private)
    except AttributeError as e:
        print("(!) 找不到超级私有")
        print(e)

    print("寻找假私有{}: {}".format(ut._semi_private, ut._semi_private))
    print("打印 __dict__: {}".format(ut.__dict__))

```

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
""" 讨论关于 args 和 kwargs 的相关知识点

总结：
    当你不确定你的函数里将要传递多少参数时你可以用 *args
    **kwargs 允许你使用没有事先定义的参数名
    可以混着用， 命名参数首先获得参数值然后所有的其他参数都传递给 *args 和
**kwargs

"""

__author__ = '__L1n_w@tch'

def test_args(*args):
    # 当你不确定你的函数里将要传递多少参数时你可以用 *args
    for count, value in enumerate(args):
        print("{}: {}".format(count, value))

def test_kwargs(**kwargs):
    # **kwargs 允许你使用没有事先定义的参数名
    for key, value in kwargs.items():
        print("Key:Value, {}:{}".format(key, value))

# 当调用函数时你也可以用 * 和 ** 语法
def star_operation(name, value, count):
    print("Name: {}, Value: {}, Count: {}".format(name, value, count))

if __name__ == "__main__":
    test_args("a", "1", "c", "b", "3", "2")
    test_kwargs(test1=1, test2=2, test3=3)

    # 它可以传递列表(或者元组)的每一项并把它们解包。注意必须与它们在函数里
```

的参数相吻合

```
a_list = ["名字", "值", "计数器"]  
star_operation(*a_list)
```

```

#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
""" 练习一下自己怎么写一个装饰器

完整资料参考: https://taizilongxu.gitbooks.io/stackoverflow-about-python/content/3/README.html
"""

__author__ = '__L1n_w@tch'

def no_argument_decorator(function):
    # 注意这是在脚本被解释期间(还没进入 main 代码就在跑的了)
    print("I am a 无参数修饰器")

    def wrapper():
        print("{} 装饰开始 {}".format("*" * 30, "*" * 30))
        function()
        print("{} 装饰结束 {}".format("*" * 30, "*" * 30))

    return wrapper


def decorator_maker_with_arguments(decorator_arg1, decorator_arg2):
    print("创建装饰器, 同时接收参数: {}, {}".format(decorator_arg1,
decorator_arg2))

    def my_decorator(func):
        print("装饰器, 得到参数: {}, {}".format(decorator_arg1, de
corator_arg2))

        # 不要忘了装饰器参数和函数参数!
        def wrapped(*args, **kwargs):
            print("装饰函数得到的参数: {}, {}, {}, {}".format(decor
ator_arg1, decorator_arg2,
                                                args, kwargs))
            return func(*args, **kwargs)

        return wrapped

    return my_decorator

```

```
        return func(*args, **kwargs)

    return wrapped

return my_decorator

@decorator_maker_with_arguments("a", "b")
# @no_argument_decorator
def no_argument_function():
    print("I am a 普通的无参数函数")

@decorator_maker_with_arguments("a", "b")
# @no_argument_decorator
def argument_function(*args):
    print("I am a 带多个参数的函数")

@decorator_maker_with_arguments("a", "b")
# @no_argument_decorator
def args_kwargs_function(*args, **kwargs):
    print("I am a 带 args & kwargs 参数的函数")

if __name__ == "__main__":
    no_argument_function()
    argument_function("c", "d")
    args_kwargs_function("c", "d")
```

## 文件 1

singleton\_for\_import.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""
作为被 import 的文件，实现单例
"""

__author__ = '__Lin_w@tch'

class MySingleton:
    def foo(self):
        pass

my_singleton = MySingleton()

if __name__ == "__main__":
    pass
```

## 文件 2

singleton.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""
单例模式的 4 种实现方法
1. 使用__new__方法
2. 共享属性(创建实例时把所有实例的 __dict__ 指向同一个字典,这样它们具有相同的属性和方法)
3. 装饰器版本

```

```

4. import方法
"""

from singleton_for_import import my_singleton

__author__ = '__Lin_w@tch'

# 使用 __new__ 方法
class Singleton:
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, "__instance"):
            orig = super(Singleton, cls)
            cls.__instance = orig.__new__(cls, *args, **kwargs)
        return cls.__instance

class TestSingleton1(Singleton):
    var = 1

"""

共享属性的方法，不知道为何出错了
class Singleton2:
    _state = {}

    def __new__(cls, *args, **kwargs):
        ob = super(Singleton2, cls).__new__(cls, *args, **kwargs)
        ob.__dict__ = cls._state
        return ob

class TestSingleton2(Singleton2):
    var = 2

"""

def singleton(cls, *args, **kwargs):
    instances = dict()

```

```
def get_instance():
    if cls not in instances:
        instances[cls] = cls(*args, **kwargs)
    return instances[cls]

return get_instance

@singleton
class TestSingleton3:
    var = 3

if __name__ == "__main__":
    ts1 = TestSingleton1()
    ts2 = TestSingleton1()
    assert ts1 == ts2 # 仅有一个实例

    # ts3 = TestSingleton2()
    # ts4 = TestSingleton2()
    # assert ts3 == ts4

    ts5 = TestSingleton3()
    ts6 = TestSingleton3()
    assert ts5 == ts6

my_singleton.foo() # import 到的本身就是个实例
```

## func\_code

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
""" 函数式编程相关知识
```

完整版参考资料：

<http://coolshell.cn/articles/10822.html>

Python 中函数式编程支持：

`filter` 函数的功能相当于过滤器，调用一个布尔函数 `bool_func` 来迭代遍历每个 `seq` 中的元素；  
返回一个使 `bool_seq` 返回值为 `true` 的元素的序列

`map` 函数是对一个序列的每个项依次执行函数

`reduce` 函数是对一个序列的每个项迭代调用函数

....

```
import functools
```

```
__author__ = '__L1n_w@tch'
```

```
def test_filter():
```

....

测试 `filter` 函数的功能

```
:return:
```

....

```
wait_to_filter = [i for i in range(10)]
```

```
after_filter = filter(lambda x: x > 5, wait_to_filter)
```

```
print("测试 filter: {}".format(list(after_filter)))
```

```
def test_map():
```

....

测试 `map` 函数的功能

```
:return:  
"""  
wait_to_map = [i for i in range(10)]  
after_map = map(lambda x: x * 2, wait_to_map)  
print("测试 map: {}".format(list(after_map)))  
  
def test_reduce():  
    """  
    测试 reduce 函数的功能, 看 doc 说 3.4 版本的 reduce 好像有所不同  
    :return:  
    """  
    wait_to_reduce = [i for i in range(1, 4)]  
    after_reduce = functools.reduce(lambda x, y: x * y, wait_to_reduce)  
    print("测试 reduce: {}".format(after_reduce))  
  
if __name__ == "__main__":  
    test_filter() # 过滤得到大于 5 的值  
    test_map() # 对一个序列每个项都乘以 2  
    test_reduce() # 求 3 的阶乘
```

## about\_copy.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""
关于 Python 的拷贝，有引用/浅复制/深复制的区别，(引用和 copy(), deepcopy()
的区别)
"""

import copy

__author__ = '__L1n_w@tch'

if __name__ == "__main__":
    List = [1, 2, 3, 4, ["a", "b"]]

    one_list = List # 赋值，传对象的引用
    second_list = copy.copy(List) # 对象拷贝，浅拷贝
    third_list = copy.deepcopy(List) # 对象拷贝，深拷贝

    List.append(5) # 修改对象 List
    List[4].append("c")

    print("List: {}".format(List))
    print("one_list: {}".format(one_list))
    print("second_list: {}".format(second_list))
    print("third_list: {}".format(third_list))
```

## 说明

记录一下学习 pdf: stackoverflow-about-Python 的笔记

资料链接: <https://taizilongxu.gitbooks.io/stackoverflow-about-python/content/>

# Python中关键字yield有什么作用？

为了理解 `yield` 有什么用，首先得理解 `generators`，再之前还要理解 `iterables`

## Iterables

可迭代的，比如创建了一个列表，可以一个一个读取它的每一项，这就叫迭代（iteration）：

```
>>> mylist = [1, 2, 3]
>>> for i in mylist:
...     print(i)
1
2
3
```

可以用在 `for...in...` 语句中的都是可迭代的，但是必须把它们的值放到内存里，当它们有很多值时就会消耗太多的内存

## Generators

生成器，也是迭代器的一种，但是只能迭代一次，因为它们不是全部存在内存里，只在要调用的时候在内存里生成：

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator:
...     print(i)
0
1
4
```

生成器和迭代器的区别就是用 `()` 代替 `[]`，还有不能用 `for i in mygenerator` 第二次调用生成器，因为每计算完一个值就会丢弃一个值

## Yield

`yield` 用法和 `return` 差不多，下面的函数将会返回一个生成器：

```
>>> def createGenerator():
...     mylist = range(3)
...     for i in mylist:
...         yield i*i
...
>>> mygenerator = createGenerator() # 创建生成器
>>> print(mygenerator) # mygenerator is an object!
<generator object createGenerator at 0xb7555c34>
>>> for i in mygenerator:
...     print(i)
0
1
4
```

如果函数要返回一个非常大的集合，而你只需要读取一次的话，用这个就很合适了。

这里当你调用函数时，函数里的代码并没有运行，仅仅返回生成器对象。每当 `for` 语句迭代生成器的时候代码才会运转。

当 `for` 语句第一次调用函数里的生成器对象，函数里的代码就开始运作，直到碰到 `yield`，然后会返回本次循环的第一个返回值。所以下一次调用也将运行一次循环然后返回下一个值，直到没有值可以返回。一旦函数运行没有碰到 `yield` 语句就认为生成器已经为空了

## 生成器的高级用法：控制迭代器的穷尽

生成器对于一些不断变化的值很有用，比如控制资源的访问：

```
>>> class Bank(): # 让我们建个银行, 生产许多ATM
...     crisis = False
...     def create_atm(self):
...         while not self.crisis:
...             yield "$100"
>>> hsbc = Bank() # 当一切就绪了你想要多少ATM就给你多少
>>> corner_street_atm = hsbc.create_atm()
>>> print([corner_street_atm.next() for cash in range(5)])
['$100', '$100', '$100', '$100', '$100']
>>> hsbc.crisis = True # 经济危机来了没有钱了!
>>> print(corner_street_atm.next())
<type 'exceptions.StopIteration'>
>>> wall_street_atm = hsbc.create_atm() # 对于其他ATM, 它还是True
>>> print(wall_street_atm.next())
<type 'exceptions.StopIteration'>
>>> hsbc.crisis = False # 麻烦的是, 尽管危机过去了, ATM还是空的
>>> print(corner_street_atm.next())
<type 'exceptions.StopIteration'>
>>> brand_new_atm = hsbc.create_atm() # 只能重新新建一个bank了
>>> for cash in brand_new_atm:
...     print cash
$100
$100
```

## Itertools 模块

该模块包含了一些特殊的函数可以操作可迭代对象，比如复制生成器，链接两个生成器等

```
>>> horses = [1, 2, 3, 4]
>>> races = itertools.permutations(horses)
>>> print(races)
>>> print(list(itertools.permutations(horses)))
[(1, 2, 3, 4),
 (1, 2, 4, 3),
 (1, 3, 2, 4),
 (1, 3, 4, 2),
 (1, 4, 2, 3),
 ...
]
```

## 理解迭代的内部机制

迭代是可迭代对象（对应 `__iter__()` 方法）和迭代器（对应 `__next__()` 方法）的一个过程。

# Python中如何在一个函数中加入多个装饰器？

示例：

```
def makebold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped

def makeitalic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped

@makebold
@makeitalic
def hello():
    return "hello world"

print hello() ## returns <b><i>hello world</i></b>
```

## 装饰器基础

- Python 中的函数都是对象
- 你可以在一个函数里定义另一个函数

这意味着函数可以返回另一个函数

## 自己动手实现装饰器

@decorator 就是下面的简写：

```
another_stand_alone_function = my_shiny_new_decorator(another_st
and_alone_function)
```

## 装饰器高级用法

### 在装饰器函数里传入参数

```
# 这不是什么黑魔法，你只需要让包装器传递参数：  
  
def a_decorator_passing_arguments(function_to_decorate):  
    def a_wrapper_accepting_arguments(arg1, arg2):  
        print "I got args! Look:", arg1, arg2  
        function_to_decorate(arg1, arg2)  
    return a_wrapper_accepting_arguments  
  
# 当你调用装饰器返回的函数时，也就调用了包装器，把参数传入包装器里，  
# 它将把参数传递给被装饰的函数里。  
  
@a_decorator_passing_arguments  
def print_full_name(first_name, last_name):  
    print "My name is", first_name, last_name  
  
print_full_name("Peter", "Venkman")  
# 输出：  
#I got args! Look: Peter Venkman  
#My name is Peter Venkman
```

### 装饰方法

在 Python 里面方法和函数几乎一模一样，唯一的区别就是方法的第一个参数是一个当前对象的 `self`。

```
def method_friendly_decorator(method_to_decorate):
    def wrapper(self, lie):
        lie = lie - 3 # 女性福音 :-)
        return method_to_decorate(self, lie)
    return wrapper

class Lucy(object):

    def __init__(self):
        self.age = 32

    @method_friendly_decorator
    def sayYourAge(self, lie):
        print "I am %s, what did you think?" % (self.age + lie)

l = Lucy()
l.sayYourAge(-3)
#输出: I am 26, what did you think?
```

如果想造一个更加通用的，可以同时满足方法和函数的装饰器，用

`*args, **kwargs` 即可

```
def a_decorator_passing_arbitrary_arguments(function_to_decorate):
    :
    # 包装器接受所有参数
    def a_wrapper_accepting_arbitrary_arguments(*args, **kwargs):

        print "Do I have args?:"
        print args
        print kwargs
        # 现在把*args, **kwargs解包
        # 如果你不明白什么是解包的话，请查阅：
        # http://www.saltycrane.com/blog/2008/01/how-to-use-args-
        # and-kwargs-in-python/
        function_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_arguments
```

## 把参数传递给装饰器

装饰器必须接收一个函数当做参数，所以不可以直接把被装饰函数的参数传递给装饰器

装饰器就是一个平常的函数，不用 `@` 也可以直接调用。当使用 `@my_decorator` 只是告诉 Python 去调用被变量 `my_decorator` 标记的函数  
一个 DEMO

```
def decorator_maker_with_arguments(decorator_arg1, decorator_arg2):

    print "I make decorators! And I accept arguments:", decorator_arg1, decorator_arg2

    def my_decorator(func):
        # 这里传递参数的能力是借鉴了 closures.
        # 如果对closures感到困惑可以看看下面这个:
        # http://stackoverflow.com/questions/13857/can-you-explain-closures-as-they-relate-to-python
        print "I am the decorator. Somehow you passed me arguments:", decorator_arg1, decorator_arg2

        # 不要忘了装饰器参数和函数参数!
        def wrapped(function_arg1, function_arg2) :
            print ("I am the wrapper around the decorated function.\n"
                   "I can access all the variables\n"
                   "\t- from the decorator: {0} {1}\n"
                   "\t- from the function call: {2} {3}\n"
                   "Then I can pass them to the decorated function"
                   .format(decorator_arg1, decorator_arg2,
                           function_arg1, function_arg2))
            return func(function_arg1, function_arg2)

        return wrapped

    return my_decorator

@decorator_maker_with_arguments("Leonard", "Sheldon")
def decorated_function_with_arguments(function_arg1, function_arg2):
    print ("I am the decorated function and only knows about my
arguments: {0}"
           " {1}".format(function_arg1, function_arg2))
```

需要记住的是，装饰器只能被调用一次，当 Python 载入脚本后，不可以动态地设置参数了。即运行了 `import x` 之后，函数已经被装饰了

于是弄一个通用的装饰器，只要装饰了这个装饰器，自定义的装饰器就可以接收任意的参数了

```
def decorator_with_args(decorator_to_enhance):
    """
    这个函数将被用来作为装饰器。
    它必须去装饰要成为装饰器的函数。
    休息一下。
    它将允许所有的装饰器可以接收任意数量的参数，所以以后你不必为每次都要做这
    个头疼了。
    saving you the headache to remember how to do that every tim
    e.
    """

    # 我们用传递参数的同样技巧。
    def decorator_maker(*args, **kwargs):
        # 我们动态的建立一个只接收一个函数的装饰器，
        # 但是他能接收来自maker的参数
        def decorator_wrapper(func):
            # 最后我们返回原始的装饰器，毕竟它只是'平常'的函数
            # 唯一的陷阱：装饰器必须有这个特殊的，否则将不会奏效。
            return decorator_to_enhance(func, *args, **kwargs)

        return decorator_wrapper
    return decorator_maker
```

删除多余注释：

```
def decorator_with_args(decorator_to_enhance):
    """
    这个函数将被用来作为装饰器，使得被它装饰的装饰器可以接收多个参数
    """

    def decorator_maker(*args, **kwargs):
        def decorator_wrapper(func):
            return decorator_to_enhance(func, *args, **kwargs)

        return decorator_wrapper

    return decorator_maker
```

使用方法：

```
@decorator_with_args
def decorated_decorator(func, *args, **kwargs):
    def wrapper(function_arg1, function_arg2):
        print "Decorated with", args, kwargs
        return func(function_arg1, function_arg2)
    return wrapper
```

之后调用这个自定义的装饰器，就可以传递任意参数了：

```
@decorated_decorator(42, 404, 1024)
def decorated_function(function_arg1, function_arg2):
    print "Hello", function_arg1, function_arg2

decorated_function("Universe and", "everything")
#输出：
#Decorated with (42, 404, 1024) {}
#Hello Universe and everything
```

自己成功使用的示例：

```
@decorator_with_args
def log_wrapper(func, *args, **kwargs):
    def wrapper(*func_args, **func_kwargs):
        print("[*] 测试装饰器")
        return func(*func_args, **func_kwargs)

    return wrapper
```

## 后续

- 装饰器是 Python2.4 里面引进的，所以确保 Python 解释器的版本  $\geq 2.4$
- 装饰器使函数调用变慢了
- 装饰器不能被取消
- 用装饰器装饰函数，可能会导致 DEBUG 难度变高

## functools 模块

`functools.wraps()` 函数，可以复制装饰器的名字、模块和文档给它的包装器  
【PS：`functools.wraps()` 本身就是一个装饰器】

```
#为了debug, 堆栈跟踪将会返回函数的 __name__
def foo():
    print "foo"

print foo.__name__
#输出: foo

# 如果加上装饰器, 将变得有点复杂
def bar(func):
    def wrapper():
        print "bar"
        return func()
    return wrapper

@bar
def foo():
    print "foo"
```

```
print foo.__name__  
#输出: wrapper  
  
# "functools" 将有所帮助  
  
import functools  
  
def bar(func):  
    # 我们所说的"wrapper", 正在包装 "func",  
    # 好戏开始了  
    @functools.wraps(func)  
    def wrapper():  
        print "bar"  
        return func()  
    return wrapper  
  
@bar  
def foo():  
    print "foo"  
  
print foo.__name__  
#输出: foo
```

## 装饰器的用途

传统的用法就是用它来为外部的库函数（你不能修改的）做扩展或者 DEBUG（你不想修改的）

Python 自身提供了几个装饰器，像 `property` 、 `staticmethod`

- Django 用装饰器管理缓存和视图的权限
- Twisted 用来修改异步函数的调用

## 如何判断一个文件是否存在？

不想用 `try` 语句，如何判断一个文件是否存在？

答案是：

```
import os.path  
os.path.isfile(file_name)
```

# 如何调用外部命令？

各种方法及各自的优缺点如下：

## os 库

### system 函数

对于运行简单的 Shell 命令而不去调用外部程序来说是非常好用的：

```
os.system("命令加参数")
```

把命令和参数传递给你系统的 Shell 中。

用这个命令的好处在于你可以一次运行好多命令，还可以设置管道来进行重定向等。但是，这要求你手动输入空格等 Shell 字符。

### popen 函数

```
stream = os.popen("命令和参数")
```

和 `os.stream` 差不多，但是它提供了一个链接标准输入/输出的管道，还有其他 3 个 `popen` 可以调用。可以传递字符串，也可以传递列表，列表就不用担心溢出字符了（escaping characters）

## subprocess 库

### Popen 管道

这个 `Popen` 是打算用来替代 `os.popen` 方法的，有点复杂：

```
subprocess.Popen("echo aaa", shell=True, stdout=PIPE).stdout.read()
```

而使用 `os.popen` :

```
os.popen("echo aaa").read()
```

它的最大优点就是一个类代替了原来 4 个不同的 `popen`

## call 方法

基本用法和 `Popen` 类参数一致，但是它会等待命令结束后才会返回程序

```
return_code = subprocess.call("echo aaa", shell=True)
```

## 总结

- OS 模块里也有 C 语言里的 `fork/exec/spawn` 方法，但是不建议直接使用
- 需要注意传递到 `shell` 命令一定要注意参数的安全性

```
subprocess.Popen("echo %s" % user_input, stdout=PIPE).stdout.read()
```

如果传入 `go die && rm -rf /` 会如何。。。

## 枚举类型的使用？

PEP435 标准已经把枚举添加到 Python3.4 版本，在 PyPi 中也可以向后支持，通过：

```
pip install enum34
```

如果下载 `enum` 没有数字将会是另一个版本

```
from enum import Enum
animal = Enum("Animal", "ant bee cat dog")
```

等价于：

```
class Animals(Enum):
    ant = 1
    bee = 2
    cat = 3
    dog = 4
```

在更早以前，有这些方法：

```
def enum(**enums):
    return type('Enum', (), enums)

>>> Numbers = enum(ONE=1, TWO=2, THREE='three')
>>> Numbers.ONE
1
>>> Numbers.TWO
2
>>> Numbers.THREE
'three'
```

或者

```
def enum(*sequential, **named):
    enums = dict(zip(sequential, range(len(sequential))), **named)
    return type('Enum', (), enums)

>>> Numbers = enum('ZERO', 'ONE', 'TWO')
>>> Numbers.ZERO
0
>>> Numbers.ONE
1
```

把值转换为名字：

```
def enum(*sequential, **named):
    enums = dict(zip(sequential, range(len(sequential))), **named)
    reverse = dict((value, key) for key, value in enums.items())
    enums['reverse_mapping'] = reverse
    return type('Enum', (), enums)
>>> Numbers.reverse_mapping['three']
'THREE'
```

这样会覆盖名字下的所有东西，但是对于枚举的输出很有用。如果转换的值不存在就会抛出 `KeyError` 异常

## 在Windows下安装pip

### Python3.4+

Python3.4 已经自带 pip 了，自带的包管理器中加入了 Ruby、Nodejs、Haskell、Perl、Go 等其他几乎所有的开源社区流行语言

### Python2.x 和 Python <=3.3

- 官方指南，下载 `get-pip.py`，然后运行
- 另一种方法：Python 包的安装器：`.msi`，可以在这个[网站](#)上下载

### 代理问题

可能需要 HTTP 代理，把环境变量设置为 `http_proxy` 和 `https_proxy`

```
http://proxy_url:port  
http://username:password@proxy_url:port
```

如果用的是微软的 NTLM 代码，还是换一个友好一点的吧

### 找不到 `vcvarsall.bat`

Python 中有的模块是用 C/C++ 编写的，pip 将尝试从源码进行编译。如果没有安装或设置过 C/C++ 编译器，将会看到这个错误

```
Error: Unable to find vcvarsall.bat
```

可以通过安装像 MinGw 或者 VC++ 这样的编译器来解决。微软实际上已经子弟啊了一个为 Python 准备的编译器：`vcpython27`

## 字典合并问题

利用 `update` 方法，合并后是修改原来的字典，而不是新建一个字典作为返回值，比如：

```
>>> x = {'a':1, 'b': 2}
>>> y = {'b':10, 'c': 11}
>>> z = x.update(y)
>>> print z
None
>>> x
{'a': 1, 'b': 10, 'c': 11}
```

## 解决方法一

```
z = dict(x.items() + y.items())
```

其中相同的键的值会被后一个字典的值覆盖

另外，如果是 Python3：

```
>>> z = dict(list(x.items()) + list(y.items()))
```

## 解决方法二

手动复制一遍再 `update`

```
z = x.copy()
z.update(y)
```

## 在 Android 上运行 Python

有一种方法，使用 Kivy。Kivy 是交互界面快速开发应用的 Python 开源库，类似于多点触控 APP

Kivy 运行在 Linux、Windows、macOS、Android 和 iOS。你也可以在所有的平台上运行 Python 代码

参考：[Kivy的应用](#)

## 根据字典值进行排序

对字典进行排序是不可能的，只有把字典转换成另一种方式才能排序。字典本身是无序的，但是像列表、元祖等其他类型是有序的。所以需要一个元祖列表来表示排序的字典

DEMO：

```
import operator
x = {1: 2, 3: 4, 4:3, 2:1, 0:0}
sorted_x = sorted(x.items(), key=operator.itemgetter(1))
```

也可以：

```
sorted(d.items(), key=lambda x: x[1])
```

## 在函数里使用全局变量

使用 `global` 关键字就可以了，例如：

```
globvar = 0

def set_globvar_to_one():
    global globvar      # 需要用global修饰一下globvar
    globvar = 1

def print_globvar():
    print globvar      # 如果要读globbar的值的话不需要用global修饰

set_globvar_to_one()
print_globvar()          # 输出 1
```

如果只进行读取全局变量的值，就不需要用 `global` 修饰

## 变化的默认参数

存在一个奇怪的现象：

```
def foo(a=[]):
    a.append(5)
    return a
>>> foo()
[5]
>>> foo()
[5, 5]
>>> foo()
[5, 5, 5]
>>> foo()
[5, 5, 5, 5]
>>> foo()
```

曾经被称为动态设计缺陷，然而这个问题应当有更深层次的解释。

Python 中的函数是最高等级的对象，而不仅仅是一小段代码。一个函数是一个被它自己定义而执行的对象，默认参数是一种“成员数据”，所以它们的状态和其他对象一样，会随着每一次调用而改变。

更详细的内容参考：[Python 中的默认参数](#)

## 装饰器**classmethod**和**staticmethod**的区别

一个对象实体调用方法，隐藏了传递的第一个参数 `self`：

```
a.foo(1)
```

如果方法是用 `classmethods` 装饰，那么被隐藏的第一个参数是对象的类而不是 `self`：

```
a.class_foo(1)
```

也可以用类调用，一般被装饰为 `classmethod` 是希望用类来调用，而不是实例来调用

用 `staticmethod` 装饰，不管传递给第一个参数是 `self` 还是 `cls`，表现都一样，静态方法被用来组织类之间有逻辑关系的函数

静态方法就是一个普通方法，一个不带参数绑定的方法

## 检查列表为空的最好办法

用隐藏的空列表的布尔值才是最 Pythonic 的方法：

```
if not a:  
    print("list a is empty")
```

虽然我自己一般用 `len(a) == 0`

## 怎么用引用来改变一个变量？

Python 文档里，参数传递的是值还是引用并没有明确说明，现在想要通过引用来改变变量，如何实现？

---

参数是通过 assignment 来传递的，原因：

- 传递的参数实际上是一个对象的引用（但是这个引用是通过值传递的）
- 一些数据类型是可变的，但有一些就不是

所以：

- 如果传递的是可变对象，那么就可以在方法里面改变，除非重新绑定了这个引用
- 如果传递的是一个不可变对象，那么就无法实现

### 是否可变类型？

- 列表-可变类型
- 字符串-不可变类型

## Solve

- 可以通过返回值实现
- 可以建一个存放值的类，然后把它传递给函数或者用一个已有的类，比如列表：

```
def use_a_wrapper_to_simulate_pass_by_reference(stuff_to_change):  
  
    new_string = something_to_do_with_the_old_string(stuff_to_change[0])  
    stuff_to_change[0] = new_string  
  
# 你可以像这样调用  
wrapper = [my_string]  
use_a_wrapper_to_simulate_pass_by_reference(wrapper)  
  
do_something_with(wrapper[0])
```

尽管看上去很笨重，但是实现需求了。。。

## 检查文件夹是否存在操作

方法一：

```
filename = "/my/directory/filename.txt"
dir = os.path.dirname(filename)

try:
    os.stat(dir)
except:
    os.mkdir(dir)

f = file(filename)
```

方法二：

```
def ensure_dir(f):
    d = os.path.dirname(f)
    if not os.path.exists(d):
        os.makedirs(d)
```

【注意】，如果在调用 `os.path.exists` 和 `os.makedirs` 之间被创建了，将会出现一个 `OSError`。然而捕获 `OSError` 并不能很好地解决这个问题，因为它将会忽略磁盘空间不足，没有足够权限等一些其他造成文件创建失败的因素

一个做法是捕获 `OSError` 异常并检查返回的错误代码

方法三：

```
os.makedirs(self.path_dir, exist_ok=True)
```

## name=main的作用

当 Python 解析器读取一个源文件时，它会执行所有的代码。在执行代码前，会定义一些特殊的变量。如果解析器运行的模块（源文件）作为主程序，它会把

`__name__` 变量设置成 `__main__`，如果只是引入其他的模块，`__name__` 变量将会设置成模块的名字

这么做的原因是有时你需要你写的模块既可以执行，还可以被当做模块导入到其他模块中去。通过检查是不是主函数，可以让你的代码只在它作为主程序运行时执行，而当其他人调用你的模块中的函数时不必执行

更多详情可以查看该[网址](#)

## super与init方法

super() 的好处是可以避免直接使用父类的名字，主要用于多重继承。

注意在 Python3 里面语法有所改变，可以用 super().\_\_init\_\_() 代替  
super(ChildB, self).\_\_init\_\_()

DEMO：

```
class Base(object):
    def __init__(self):
        print "Base created"

class ChildA(Base):
    def __init__(self):
        Base.__init__(self)

class ChildB(Base):
    def __init__(self):
        super(ChildB, self).__init__()

class ChildC(Base):
    def __init__(self):
        super().__init__()

print ChildA(), ChildB(), ChildC()
```

## str与repr的区别

有一点是可以肯定的：如果 `__repr__` 被定义而 `__str__` 没有，那么对象会自动设置 `__str__ = __repr__`

这意味着，几乎所有的对象你需要实现 `__repr__` 以便使用者更好地理解这个对象。然而实现 `__str__` 只是为了一个更漂亮的输出

实现 `__repr__` 是为了明确

实现 `__str__` 是为了可读性

## 如何在循环中获取索引

如果像 C 或者 PHP 那样加入一个状态变量那就太不 pythonic 了

最好的选择就是用内建函数 `enumerate()`

```
for idx, val in enumerate(ints):
    print(idx, val)
```

## 类里的静态变量

变量是在类定义时声明的，不是在类方法或静态变量中：

```
>>> class MyClass:  
...     i = 3  
...  
>>> MyClass.i  
3
```

注意变量 `i` 是类级别的，所以它和所有实例的 `i` 变量是不一样的，比如：

```
>>> m = MyClass()  
>>> m.i = 4  
>>> MyClass.i, m.i  
>>> (3, 4)
```

这与 C++ 以及 Java 不一样，但是和 C# 相同，就是静态成员不能被实例所引用

# 怎么在终端里输出颜色

## 方法一

Python `termcolor module` 模块

```
from termcolor import colored
print(colored("hello", "red"))
print(colored("world", "green"))
```

---

## 方法二

这依赖于你使用哪种操作系统，最常用的方法就是输出 ANSI 转义序列：

```
class bcolors:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
```

可以这么使用上面的代码：

```
print(bcolors.WARNING + "Warning: No active frommets remain. Con
tinue?" + bcolors.ENDC)
```

这种方法适合 macOS、Linux 和 Windows。还有一些其他的 ANSI 代码可以设置颜色，消除光标或者其他

如果想要应用更复杂的功能，应该看看 `curses` 模块，参考：[Python Curses HowTO](#)

## 怎么在终端里输出颜色?

---

如果不使用拓展的 ASCII，比如不是一个 PC，那么 # 或者 @ 可能是最好的选择

如果用的是 IBM扩展ascii字符设置，你还可以有更多的选择。176,177,178和219是"块字符"。

一些现代的基于文本的程序，像 *Swarf Fortress* 显示的文本使用图像模式，而用的字体也是传统的计算机字体的图像。可以在 [Dwarf Fortress Wiki](#) 找到你可以用的字符。

[Text Mode Demo Contest](#) 也有许多资料可供参考。

## 为什么用pip比用easy\_install的好？

pip 是对 easy\_install 以下方面进行了改进：

- 所有的包是在安装之前就下载了，所以不会出现只安装了一部分的情况
- 在终端上的输出更加友好
- 对于动作的原因会进行持续的跟踪
- 错误信息会非常有用
- 代码简洁精悍可以很好地编程
- 不必作为 egg 存档，能扁平化安装（仍然保存 egg 元数据）
- 原生的支持其他版本控制系统（Git、Mercurial、Bazaar）
- 卸载包
- 可以简单地定义修改一系列的安装依赖等

## join 的问题

为什么是 `string.join(list)` 而不是 `list.join(string)` ?

比如：

```
my_list = ["Hello", "world"]
print(my_list.join("-"))
# Produce: "Hello-world"
```

其实因为所有的可迭代对象都能被 `join` , 不仅仅是列表，但一般情况下我们要 `join` 的都是字符串

比如：

```
import urllib2
print '\n#####\n'.join(urllib2.urlopen('http://data.stack
exchange.com/users/7095'))
```

## 把列表分割成同样大小的块

希望有个方法，比如生成器，可以实现下面的效果：

```
l = range(1, 1000)
print(chunks(l, 10))
# [ [ 1..10 ], [ 11..20 ], ... , [ 991..999 ] ]
```

## Solve

```
def chunks(l, n):
    """ Yield successive n-sized chunks from l.
    """
    for i in xrange(0, len(l), n):
        yield l[i:i+n]
```

或者

```
tuple(l[i:i+n] for i in xrange(0, len(l), n))
```

## 打印效果

```
import pprint
pprint pprint(list(chunks(range(10, 75), 10)))
[[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
 [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
 [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
 [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
 [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
 [70, 71, 72, 73, 74]]
```

把列表分割成同样大小的块

---

## 为什么代码在一个函数里运行的更快？

比如同样的一段代码，放在函数里：

```
def main():
    for i in xrange(10**8):
        pass
main()
```

运行时间：

```
real    0m1.841s
user    0m1.828s
sys     0m0.012s
```

不放在函数里：

```
for i in xrange(10**8):
    pass
```

运行时间：

```
real    0m4.543s
user    0m4.524s
sys     0m0.012s
```

## 解答

这是因为存取一个本地变量比全局变量要快，这有关 CPython 的实现细节。

记住 CPython 解析器运行的是被编译过的字节编码（bytecode）。当一个函数被编译后，局部变量被存储在了固定大小的数组（不是一个 dict），而变量名赋值给了索引。于是你不能动态地为一个函数添加局部变量，检查一个局部变量就好像是一个指针去查找列表，对于在 PyObject 上的引用计数增长是微不足道的。

在查找全局变量（`LOAD_GLOBAL`）时，涉及到一个实实在在的`dict`的哈希查找。同时，这也是你想要一个全局变量时需要加上`global i`的原因：如果你在一个区域内指定变量，编译器就会建立一个`STORE_FAST`的入口，除非你不让它这么做。

全局查找速度其实也不慢，真正拖慢速度的是像`foo.bar`这样的属性查找

对比函数和全局的字节码：

```
# 函数
2      0 SETUP_LOOP          20 (to 23)
      3 LOAD_GLOBAL           0 (xrange)
      6 LOAD_CONST            3 (1000000000)
      9 CALL_FUNCTION         1
     12 GET_ITER
>> 13 FOR_ITER             6 (to 22)
     16 STORE_FAST            0 (i)

3      19 JUMP_ABSOLUTE       13
>> 22 POP_BLOCK
>> 23 LOAD_CONST           0 (None)
     26 RETURN_VALUE

# 全局
1      0 SETUP_LOOP          20 (to 23)
      3 LOAD_NAME              0 (xrange)
      6 LOAD_CONST            3 (1000000000)
      9 CALL_FUNCTION          1
     12 GET_ITER
>> 13 FOR_ITER             6 (to 22)
     16 STORE_NAME             1 (i)

2      19 JUMP_ABSOLUTE       13
>> 22 POP_BLOCK
>> 23 LOAD_CONST           2 (None)
     26 RETURN_VALUE
```

## 为什么代码在一个函数里运行的更快

---

区别在于 `STORE_FAST` 比 `STORE_NAME` 要快很多。这是因为在函数里 `i` 是一个局部变量而在全局区域它是一个全局变量

用 `dis` 模块可以看字节编码，可以直接解析函数。但是要解析全局代码必须用 `compile` 内建模块

## 如何快速合并列表中的列表？

除了循环，有没有更快的方法来实现一行合并列表中的列表？

比如这样的：

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]  
reduce(lambda x, y: x.extend(y), l)
```

当然上面这样会报错。。

## Solve

目前看来，最快的方法应该是：

```
[item for sublist in l for item in sublist]
```

可以用 `timeit` 模块进行验证：

```
$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' '[item  
for sublist in l for item in sublist]'  
10000 loops, best of 3: 143 usec per loop  
$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' 'sum(l,  
[])'  
1000 loops, best of 3: 969 usec per loop  
$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' 'reduce  
(lambda x,y: x+y,l)'  
1000 loops, best of 3: 1.1 msec per loop
```

## 解释

当有  $L$  个子串的时候用 `+`，即 `(sum)` 的时间复杂度是  $O(L^{**2})$ ，每次迭代的时候作为中间结果的列表的长度就会越来越长，而且前一个中间结果的所有项都会再拷贝一遍给下一个中间结果。所以当你的列表  $|$  含有  $L$  个子串， $|$  列表的第一

## 如何快速合并列表中的列表？

---

项需要拷贝  $L-1$  次，而第二项要拷贝  $L-2$  次，以此类推总数为： $I * (L ** 2) / 2$

列表推导式（`list comprehension`）只是生成一个列表，每次运行只拷贝一次（从开始的地方拷贝到最终结果）

## 如何知道一个对象有一个特定的属性？

使用 `hasattr()` :

```
if hasattr(a, 'property'):
    a.property
```

在大多数实际情况下，如果一个属性有很大可能存在，那么就直接调用它或者让它引发异常，或者用 `try/except` 捕获，这种方法比 `hasattr` 快。如果这个属性很多情况下不在，或者你不确定，那么用 `hasattr` 将会比触发异常更快。

## 如何通过函数名的字符串来调用这个函数？

假设我们有一个模块名字为 `foo`，其中有一个函数为 `bar`，但现在只给了字符串 `"bar"`，如何通过字符串 `"bar"` 来调用对应的函数 `foo.bar()` ？

### Solve

通过 `getattr` 方法：

```
import foo
methodToCall = getattr(foo, 'bar')
result = methodToCall()
```

可以简写为：

```
result = getattr(foo, 'bar')()
```

另外 `getattr` 可以用在实例绑定、模块级方法、类方法等

## 单下划线和双下划线的含义？

### 单下划线

在一个类中的方法或属性，用单下划线开头，就是告诉别的程序这个属性或方法是私有的。

引自 PEP-8：

单下划线：“内部使用”的弱指示器。比如，`from M import *` 将不会引进用单下划线开头的对象。

### 双下划线

引自 Python 文档：

任何 `__spam` 形式（至少两个下划线开头，最多一个下划线结尾）都是代替 `_classname__spam`，其中 `classname` 是当前类的名字。

### 约定

`__foo__`：Python 内部的名字，用来区别其他用户自定义的命名，以防冲突

`_foo`：用来指定变量私有

`__foo`：解析器将会用 `_classname__foo` 来代替这个名字，以区别和其他类相同的命名

# Python 面试编程题

这一节主要就是编程题部分。

## 参考资料

[interview\\_python](#)

## 清单

1. 台阶问题/斐波那契
2. 变态台阶问题
3. 矩形覆盖
4. 杨氏矩阵查找
5. 去除列表中的重复元素
6. 链表成对调换
7. 创建字典的方法
8. 合并两个有序列表
9. 二分查找
10. 快排
11. 找零问题
12. 二叉树相关
13. 单链表逆置

## 题目说明

经典的面试题：

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法？

斐波那契除了 GitHub 面试题给的那三种外，还有我之前实现过的一种，都整理在这个文件了，包括：

- 递归求斐波那契(函数形式)
- 循环求斐波那契(函数形式)
- 带记忆的递归求斐波那契(函数形式 + 装饰器)
- 循环求斐波那契(类形式)

## fibonacci.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

import timeit
from functools import wraps

__author__ = '__L1n_w@tch'

def recursion(n):
    """
    普通的递归求斐波那契
    :param n:
    :return:
    """
    if n <= 2:
        return n
```

```

else:
    return recursion(n - 1) + recursion(n - 2)

def memory(function):
    """
    作为修饰器存在的
    :param function:
    :return:
    """
    cache = {}

    @wraps(function) # 加这句主要是为了保留被修饰的函数的名字
    def wrap(args):
        if args not in cache:
            cache[args] = function(args)
        return cache[args]

    return wrap

@memory
def memory_recursion(n):
    """
    函数本身依旧是普通的递归求斐波那契，但是装有修饰器
    :param n:
    :return:
    """
    if n <= 2:
        return n
    else:
        return memory_recursion(n - 1) + memory_recursion(n - 2)

def circle(n):
    """
    循环求斐波那契
    :param n:
    :return:
    """

```

```

a, b = 0, 1
for i in range(n):
    a, b = b, a + b
return b

class Fibonacci:
    """
自己之前写的一个斐波那契数列
"""

def __init__(self, f0, f1):
    self.f0 = f0
    self.f1 = f1

def _fib(self, a, b):
    a = self.f0
    b = self.f1
    yield a
    yield b
    while True:
        a, b = b, a + b
        yield b

def fib_generator(self):
    """
生成器
:return:
"""
    return self._fib(self.f0, self.f1)

if __name__ == "__main__":
    print("方法一：匿名函数(其实就是递归)，注意这种写法被 PEP8 报警了...")
)
# fibonacci_1 = lambda n: n if n <= 2 else fibonacci_1(n - 1) + fibonacci_1(n - 2)
fibonacci_1 = recursion
time_cost = timeit.timeit("recursion(33)", setup="from fibonacci import recursion", number=1)

```

```

print("fibonacci_1(33) 耗时: {}".format(time_cost))

print("方法二: 依旧递归求解, 不过加了个记忆用的修饰器, 速率翻倍了都")
fibonacci_2 = memory_recursion
time_cost = timeit.timeit("memory_recursion(33)", setup="from fibonacci import memory_recursion")
print("fibonacci_2(33) 耗时: {}".format(time_cost))

print("方法三: 循环求解")
fibonacci_3 = circle
time_cost = timeit.timeit("circle(33)", setup="from fibonacci import circle", number=1)
print("fibonacci_3(33) 耗时: {}".format(time_cost))

print("方法四: 同样循环求解, 不过是通过生成器实现的")
fibonacci_4 = Fibonacci(1, 2).fib_generator()

```



## 单元测试

本文件是对同文件夹下的 fibonacci 几种写法进行测试

```

#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

import unittest
from fibonacci import recursion, memory_recursion, circle

__author__ = '__L1n_w@tch'

class FibonacciTest(unittest.TestCase):
    def setUp(self):
        self.wait_to_test = [recursion, memory_recursion, circle]
    # 待测试的函数

```

```
def normal_test(self):
    """
    简单的功能性测试罢了
    :return:
    """

    for function in self.wait_to_test:
        self.failUnless(function(1) == 1)
        self.failUnless(function(2) == 2)
        self.failUnless(function(3) == 3)
        self.failUnless(function(4) == 5)
        self.failUnless(function(5) == 8)
        self.failUnless(function(6) == 13)
        self.failUnless(function(7) == 21)
        self.failUnless(function(13) == 377)
        self.failUnless(function(15) == 987)
        self.failUnless(function(20) == 10946)
        self.failUnless(function(25) == 121393)
        self.failUnless(function(29) == 832040)
        self.failUnless(function(33) == 5702887)
        if function.__name__ == "recursion":
            print("递归 50 次时间太长，不测试了")
            continue
        self.failUnless(function(50) == 20365011074)
        print("函数: {}, 测试完毕".format(function.__name__))

if __name__ == "__main__":
    unittest.main()
```

## 题目说明

变态台阶问题，一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级，它也可以跳上 n 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法？

根据数学归纳法可证得答案为  $2^{n-1}$  (可参考剑指 Offer 面试题 9)

## similar\_frog\_jump.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

__author__ = '__L1n_w@tch'

def recursion(steps):
    if steps < 2:
        return steps
    else:
        return 2 * recursion(steps - 1)

if __name__ == "__main__":
    n = 233
    print("{} 级台阶: {}".format(n, recursion(n)))

    compute_jump = lambda steps: steps if steps < 2 else 2 * compute_jump(steps - 1)
    print("{} 级台阶: {}".format(n, compute_jump(n)))
```

## 题目说明

### 矩形覆盖

可以用  $2 * 1$  的小矩形横着或者竖着去覆盖更大的矩形。请问用 8 个  $2 * 1$  的小矩形无重叠地覆盖一个  $2 * 8$  的大矩形，总共有多少种方法？

同样可以参考剑指 Offer (面试题 9)

解题思路摘要：

我们先把  $2 * 8$  的覆盖方法记为  $f(8)$ 。用第一个  $1 * 2$  小矩形去覆盖大矩形的最左边时有两个选择，竖着放或者横着放。

当竖着放的时候，右边还剩下  $2 * 7$  的区域，这种情况下覆盖方法记为  $f(7)$ 。

接下来考虑横着放的情况。当  $1 * 2$  的小矩形横着放在左上角的时候，右下角必须和横着放一个  $1 * 2$  的小矩形，而在右边还剩下  $2 * 6$  的区域。

这种情形下的覆盖方法记为  $f(6)$ ，因此  $f(8) = f(7) + f(6)$ 。此时可以看出，这仍然是斐波那契数列。

## rectangle.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

__author__ = '__L1n__w@tch'

def circle_fibonacci(steps, fn_1, fn_2):
    """
    循环求解斐波那契数列, f(n) = f(n - 1) + f(n - 2)
    :param steps: 阶数
    :param fn_1: f(n - 1)
    :param fn_2: f(n - 2)
    :return: f(n)
    """
    for i in range(steps):
        fn_1, fn_2 = fn_2, fn_1 + fn_2
    return fn_2

if __name__ == "__main__":
    n = 3

    print("{} 级台阶: {}".format(n, circle_fibonacci(n, 0, 1)))
```

## 题 目

在一个  $m$  行  $n$  列二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

## 思 路

- \* 参考：剑指 Offer（面试题 3）
- \* 总结查找的过程，发现规律如下：
  - \* 首先选取数组中右上角的数字。如果该数字等于要查找的数字，查找过程结束；
  - \* 如果该数字大于要查找的数字，剔除这个数字所在的列；
  - \* 如果该数字小于要查找的数字，剔除这个数字所在的行。
  - \* 也就是说如果要查找的数字不在数组的右上角，则每一次都在数组的查找范围内剔除一行或者一列，这样每一步都可以缩小查找的范围，
  - \* 直到找到要查找的数字，或者查找范围为空。

## rectangle\_search.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

import timeit
import random

__author__ = '__L1n_w@tch'

def python_style_solve(a_list, number_to_find):
    """
    用 Python 的风格解决，思路是把二维数组转换为一维数组，然后执行查找
    :param a_list: 要查找的二维数组
    :param number_to_find: 待查找的数字
    """

    if not a_list or not a_list[0]:
        return False

    m, n = len(a_list), len(a_list[0])
    start, end = 0, m * n - 1
    while start <= end:
        mid = (start + end) // 2
        if a_list[mid // n][mid % n] == number_to_find:
            return True
        elif a_list[mid // n][mid % n] > number_to_find:
            end = mid - 1
        else:
            start = mid + 1
    return False
```

```

    :return: True or False
    """
    sum_list = list()
    for each in a_list:
        sum_list.extend(each)
    if number_to_find in sum_list:
        return True
    # print("Found!")

def c_style_solve(a_list, rows, columns, number_to_find):
    """
    用 C 的风格解决，思路参考剑指 Offer
    :param a_list: 要查找的二维数组
    :param rows: 行
    :param columns: 列
    :param number_to_find: 待查找的数字
    :return: True or False
    """
    found = False

    if a_list is not None and rows > 0 and columns > 0:
        row, column = 0, columns - 1
        while row < rows and column >= 0:
            if a_list[row][column] == number_to_find:
                found = True
                break
            elif a_list[row][column] > number_to_find:
                column -= 1
            else:
                row += 1
    return found

if __name__ == "__main__":
    python_time = list()
    c_time = list()

    for i in range(30):
        number = random.choice([i for i in range(9)])

```

```
print("第 {} 次, 查找数字: {}".format(i + 1, number))

    test_list = [[i for i in range(3)], [i for i in range(3,
6)], [i for i in range(6, 9)]]
    time_cost = timeit.timeit("python_style_solve(test_list,
{})".format(number),
                                setup="from __main__ import py
thon_style_solve, test_list")
    print("Python 风格执行时间: {} s".format(time_cost))
    python_time.append(time_cost)

    time_cost = timeit.timeit("c_style_solve(test_list, len(
test_list), len(test_list[0]), {})".format(number),
                                setup="from __main__ import c_
style_solve, test_list")
    print("C 风格执行时间: {} s".format(time_cost))
    c_time.append(time_cost)

    print("Python 平均时间: {}, C 平均时间: {}".format(sum(python_t
ime) / len(python_time), sum(c_time) / len(c_time)))
```

## 单元测试

```

#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
""" 对杨氏矩阵查找的几种解法进行单元测试
"""

import unittest
from rectangle_search import python_style_solve, c_style_solve

__author__ = '__L1n_w@tch'

class TestAnswer(unittest.TestCase):
    def setUp(self):
        self.repeat_list = [[1, 2, 8, 9], [2, 4, 9, 12], [4, 7,
10, 13], [6, 8, 11, 15]]
        self.no_repeat_list = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10,
11, 12], [13, 14, 15, 16], [17, 18, 19, 20]]

    def test_normal(self):
        # 测试数字在其中的情况
        wait_to_search = [1, 9, 2, 12, 4, 13, 6, 15, 11, 8, 10, 7
, 9, 4, 8, 2]
        print("二维数组: ".format(self.repeat_list))
        for each_number in wait_to_search:
            print("测试查找数字: {}".format(each_number))
            self.failUnless(python_style_solve(self.repeat_list,
each_number))
            self.failUnless(
                c_style_solve(self.repeat_list, len(self.repeat_
list), len(self.repeat_list[0]), each_number))

        # 测试数字不在其中的情况
        # self.failIf()

    if __name__ == "__main__":
        pass

```



## 题 目

去除重复元素的几种方法，包括

- 用集合去除重复元素
- 用字典去除重复元素
- 用字典去除并保持顺序
- 列表推导式

## 解 答

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

__author__ = '__L1n_w@tch'

def use_set(a_list):
    """
    用集合去除重复元素
    :param a_list: [1,2,1,2]
    :return: [1,2]
    """
    return list(set(a_list))

def use_dictionary(a_list):
    """
    用字典去除重复元素
    :param a_list: [1,2,1,2]
    :return: [1,2]
    """
    a_dict = dict().fromkeys(a_list).keys()
    return list(a_dict)
```

## 5 去除列表中的重复元素

```
def use_dict_keep_order(a_list):
    """
    用字典并保持顺序
    :param a_list: [1,2,1,2]
    :return: [1,2]
    """
    another_list = list(set(a_list))
    another_list.sort(key=a_list.index)
    return another_list

def use_list_derive(a_list):
    """
    列表推导式
    :param a_list: [1,2,1,2]
    :return: [1,2]
    """
    another_list = list()
    [another_list.append(i) for i in a_list if i not in another_list]
    return another_list

if __name__ == "__main__":
    List = [1, 2, 1, 2, 3, 3, 3, 4, 1, 0, 9, 9, 3, 11112221231]

    print(use_set(List))
    print(use_dictionary(List))
    print(use_dict_keep_order(List))
    print(use_list_derive(List))

    if use_set(List) == use_dictionary(List) == use_dict_keep_order(List) == use_list_derive(List):
        print("OK")
```

# 题 目

链表成对调换

1->2->3->4 转换成 2->1->4->3

通过递归实现

## swap\_pairs\_list\_node.py

```

#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

__author__ = '__L1n_w@tch'

def swap_pairs(head_node):
    """
    链表成对调换
    :param head_node:
    :return:
    """
    if head_node is not None and head_node.next is not None:
        next_node = head_node.next
        head_node.next = swap_pairs(next_node.next)
        next_node.next = head_node
        return next_node
    return head_node

def print_list_node(head_node):
    """
    顺序打印链表
    :param head_node: 头结点
    :return: None
    """

```

```
"""
for node in list_node:
    print("No.{0} List Node, Next: {1}".format(node.value, node.next.value if node.next is not None else None))
print()

class ListNode:
    def __init__(self, value, next_node=None):
        self.value = value
        self.next = next_node

if __name__ == "__main__":
    list_node = list()
    for i in range(4, 0, -1):
        list_node.append(ListNode(i, list_node[-1] if len(list_node) > 0 else None))
    list_node = list(reversed(list_node))

    print_list_node(list_node[0])

    swap_pairs(list_node[0])
    print_list_node(list_node[0])
```

## 题目

考怎么创建字典的?包括:

- 花括号创建
- dict 内置类
- dict().fromkeys 方法

## create\_dict.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

__author__ = '__L1n_w@tch'

if __name__ == "__main__":
    dictionary = {"key1": 1, "key2": 2}

    items = [("key1", 1), ("key2", 2)]
    dictionary_2 = dict(items)

    dictionary_3 = dict().fromkeys(("key1", "key2"), -1)
    dictionary_4 = dict(("key1", "key2"))

print(dictionary, dictionary_2, dictionary_3, dictionary_4)
```

## 题 目

考归并排序的两种实现方式, 一种递归一种循环

### merge\_sort.py

```

#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

__author__ = '__L1n_w@tch'

def recursion_merge_sort(list1, list2, a_list):
    def __recursion(list_wait_to_sort_1, list_wait_to_sort_2, li
st_after_sort):
        if len(list_wait_to_sort_1) == 0 or len(list_wait_to_sor
t_2) == 0:
            list_after_sort.extend(list_wait_to_sort_1)
            list_after_sort.extend(list_wait_to_sort_2)
            return list_after_sort
        elif list_wait_to_sort_1[0] < list_wait_to_sort_2[0]:
            list_after_sort.append(list_wait_to_sort_1[0])
            del list_wait_to_sort_1[0] # 为啥不用 list1.pop(0)...
        elif list_wait_to_sort_1[0] >= list_wait_to_sort_2[0]:
            list_after_sort.append(list_wait_to_sort_2[0])
            del list_wait_to_sort_2[0]
        return __recursion(list_wait_to_sort_1, list_wait_to_sor
t_2, list_after_sort)

    return __recursion(list1, list2, list())

def loop_merge_sort(list1, list2):
    after_sort = list()

```

```
while len(list1) > 0 and len(list2) > 0:
    if list1[0] < list2[0]:
        after_sort.append(list1[0])
        del list1[0]
    else:
        after_sort.append(list2[0])
        del list2[0]
after_sort.extend(list1)
after_sort.extend(list2)
return after_sort

if __name__ == "__main__":
    List1 = sorted([3, 4, 1, 2])
    List2 = sorted([2, 1, 1, 1])

    test_list = recursion_merge_sort(List1[::-1], List2[::-1], list())
    print(test_list)

    test_list = loop_merge_sort(List1, List2)
    print(test_list)
```



## 单元测试

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""
    测试归并排序是否正确
"""

import random
import unittest
from merge_sort import recursion_merge_sort, loop_merge_sort

__author__ = '__L1n_w@tch'

class TestMergeSort(unittest.TestCase):
    def test(self):
        for i in range(5):
            wait_to_sort = sorted([random.randint(-1000, 1000) for i in range(100)])
            wait_to_sort_2 = sorted([random.randint(-1000, 1000) for i in range(100)])

            after_sort = sorted(wait_to_sort + wait_to_sort_2)

            self.failUnless(after_sort == recursion_merge_sort(wait_to_sort[::1], wait_to_sort_2[::1], list())))
            self.failUnless(after_sort == loop_merge_sort(wait_to_sort[::1], wait_to_sort_2[::1]))


if __name__ == "__main__":
    pass
```

## 题目

考二分查找法的

### binary\_search.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

__author__ = '__L1n_w@tch'

def binary_search(a_list, number_to_find):
    """
    二分查找法
    :param a_list: 待查找列表
    :param number_to_find: 待查找数字
    :return: int() or False
    """
    start, end = 0, len(a_list) - 1
    while start < end:
        middle = (start + end) // 2
        if a_list[middle] > number_to_find:
            end = middle
        elif a_list[middle] < number_to_find:
            start = middle + 1
        else:
            return middle
    return start if a_list[start] == number_to_find else False

if __name__ == "__main__":
    List = [i for i in range(10)]
    print(binary_search(List, 2))
```

# 单元测试

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""
    单元测试二分查找
"""

import random
import unittest
from binary_search import binary_search

__author__ = '__L1n_w@tch'

class TestBinarySearch(unittest.TestCase):
    def test(self):
        list_wait_to_search = sorted([random.randint(-1000, 1000)
) for i in range(100)])
        number_wait_to_search = list_wait_to_search[random.randint(0, 100)]

        self.failUnless(number_wait_to_search ==
                        list_wait_to_search[binary_search(list_wait_to_search, number_wait_to_search)])
```

## 题目

快排实现

### qsort.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

__author__ = '__L1n_w@tch'

def qsort(a_list):
    if len(a_list) == 0:
        return a_list
    else:
        pivot = a_list[0]
        small = qsort([x for x in a_list[1:] if x < pivot])
        big = qsort([x for x in a_list[1:] if x >= pivot])
        return small + [pivot] + big

if __name__ == "__main__":
    List = [1, 3, 5, 1, 2]
    print(qsort(List))
```

## 单元测试

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""
    单元测试快排
"""

import random
import unittest
from qsort import qsort

__author__ = '__L1n_w@tch'

class TestQSort(unittest.TestCase):
    def test_qsort(self):
        for i in range(30):
            list_wait_to_sort = [random.randint(-1000, 1000) for
i in range(100)]
            self.failUnless(sorted(list_wait_to_sort) == qsort(list_wait_to_sort))
```

## 题 目

找零问题，给个金额让你找出对应的最小硬币数目

### **coin\_change.py**

```

#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

__author__ = '__L1n_w@tch'

def coin_change(values, money):
    """
    背包问题解法?
    :param values: [25, 21, 10, 5, 1]
    :param money: 63
    :return: {1:1, 2:2, 3:3, 4:4, 5:1, ..., 63:3}
    """
    coin_count = {i: 0 for i in range(money_wait_to_change + 1)}

    for cents in range(1, money + 1):
        min_coins = cents # 从第一个开始到money的所有情况初始
        for value in values:
            if value <= cents:
                temp = coin_count[cents - value] + 1 # 这不是经典的背包?
                if temp < min_coins:
                    min_coins = temp
        coin_count[cents] = min_coins
        print('面值为 :{0} 的最小硬币数目为 :{1}'.format(cents, coin_count[cents]))

    if __name__ == '__main__':
        pocket_monkey = [25, 21, 10, 5, 1]
        money_wait_to_change = 63
        coin_change(pocket_monkey, money_wait_to_change)

```

## 题目

实现二叉树的相关算法

### binary\_tree.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X
"""
    给定一个数组，构建二叉树，并且按层次打印这个二叉树，以及进行深度遍历
"""

import queue
from functools import wraps

__author__ = '__L1n_w@tch'

def decorator_with_argument(sentence):
    """
    带有一个参数的修饰器
    :param sentence: 要打印的句子
    :return: 修饰器
    """

    def decorator(function):
        @wraps(function)
        def wrap(*args):
            print(sentence)
            result = function(*args)
            print('')
            return result # 注意需要返回结果的

        return wrap

    return decorator
```

```

class Node:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

@decorator_with_argument("第一种层次遍历")
def layer_search(root):
    """
    层次遍历
    :param root: 根节点
    :return: None
    """
    stack = [root]
    while stack:
        current = stack.pop(0)
        print(current.data, end=" ")
        if current.left:
            stack.append(current.left)
        if current.right:
            stack.append(current.right)

@decorator_with_argument("第二种层次遍历")
def layer_order_traverse(root):
    """
    层次遍历第二种写法，其实就换了个数据结构
    :param root: 根节点
    :return: None
    """
    a_queue = queue.Queue()
    a_queue.put(root)
    while not a_queue.empty():
        node = a_queue.get()
        print(node.data, end=" ")
        if node.left:
            a_queue.put(node.left)
        if node.right:
            a_queue.put(node.right)

```

```

        a_queue.put(node.right)

@decorator_with_argument("先序遍历")
def pre_order_traverse(root):
    """
    深度遍历，先序遍历？

    PS: 之所以把递归封装起来是为了不重复调用装饰器
    :param root: 根节点
    :return: None
    """

    def __recursion(node):
        if not node:
            return None
        print(node.data, end=" ")
        __recursion(node.left)
        __recursion(node.right)

    return __recursion(root)

@decorator_with_argument("中序遍历")
def in_order_traverse(root):
    """
    中序遍历
    :param root: 根节点
    :return: None
    """

    def __recursion(node):
        if not node:
            return None
        if node.left:
            __recursion(node.left)
        print(node.data, end=" ")
        if node.right:
            __recursion(node.right)

    return __recursion(root)

```

```

@decorator_with_argument("后序遍历")
def post_order_traverse(root):
    """
    后序遍历
    :param root: 根节点
    :return: None
    """

    def __recursion(node):
        if not node:
            return None
        if node.left:
            __recursion(node.left)
        if node.right:
            __recursion(node.right)
        print(node.data, end=" ")

    return __recursion(root)

@decorator_with_argument("求最大树深")
def max_depth(root):
    """
    求树的最大深度
    :param root: 根节点
    :return: int()
    """

    def __recursion(node):
        if not node:
            return 0
        return max(__recursion(node.left), __recursion(node.right)) + 1

    return __recursion(root)

@decorator_with_argument("判断两棵树是否相同")

```

```

def is_same_tree(root1, root2):
    """
    递归判断两颗树是否相同
    :param root1: 根节点 1
    :param root2: 根节点 2
    :return: True or False
    """

def __recursion(node1, node2):
    if node1 is None and node2 is None:
        return True
    elif node1 and node2:
        return node1.data == node2.data \
               and __recursion(node1.left, node2.left) \
               and __recursion(node1.right, node2.right)
    else:
        return False

    return __recursion(root1, root2)

@decorator_with_argument("先序中序创建二叉树")
def create_binary_tree_by_pre_in_order(pre_order_list, in_order_list):
    """
    根据先序中序创建二叉树
    :param pre_order_list: [1, 3, 7, 0, 6, 2, 5, 4]
    :param in_order_list: [0, 7, 3, 6, 1, 5, 2, 4]
    :return: Node() or None
    """

def __recursion(pre_order, in_order):
    if not pre_order:
        return None
    current = Node(pre_order[0])
    index = in_order.index(pre_order[0])
    current.left = __recursion(pre_order[1:index + 1], in_order[:index])
    current.right = __recursion(pre_order[index + 1:], in_order[index + 1:])

```

```

        return current

    return __recursion(pre_order_list, in_order_list)

@decorator_with_argument("中序后序创建二叉树")
def create_binary_tree_by_post_in_order(post_order_list, in_order_list):
    """
    根据中序后序来创建二叉树
    :param post_order_list: [0, 7, 6, 3, 5, 4, 2, 1]
    :param in_order_list: [0, 7, 3, 6, 1, 5, 2, 4]
    :return: Node() or None
    """

    def __recursion(post_order, in_order):
        if not post_order:
            return None
        current = Node(post_order[-1])
        index = in_order.index(post_order[-1])
        current.left = __recursion(post_order[:index], in_order[:index])
        current.right = __recursion(post_order[index:-1], in_order[index + 1:])

        return current

    return __recursion(post_order_list, in_order_list)

if __name__ == "__main__":
    tree = Node(1, Node(3, Node(7, Node(0)), Node(6)), Node(2, Node(5), Node(4)))
    layer_search(tree)
    layer_order_traverse(tree)
    pre_order_traverse(tree)
    in_order_traverse(tree)
    post_order_traverse(tree)
    print(max_depth(tree))

```

```
tree2 = Node(11, Node(3, Node(7, Node(0)), Node(6)), Node(2,
Node(5), Node(4)))
print(is_same_tree(tree, tree2))

pre_order_visit = [1, 3, 7, 0, 6, 2, 5, 4]
in_order_visit = [0, 7, 3, 6, 1, 5, 2, 4]
tree = create_binary_tree_by_pre_in_order(pre_order_visit, i
n_order_visit)
post_order_traverse(tree)

post_order_visit = [0, 7, 6, 3, 5, 4, 2, 1]
tree = create_binary_tree_by_post_in_order(post_order_visit,
in_order_visit)
pre_order_traverse(tree)
```

## 题目

对单链表实现逆置操作, 即 `1 -> 2 -> 3` 变为 `3 -> 2 -> 1`

## single\_list\_reverse.py

```
#!/bin/env python3
# -*- coding: utf-8 -*-
# version: Python3.X

__author__ = '__L1n_w@tch'

class ListNode:
    def __init__(self, data=None, next_node=None):
        self.data = data
        self.next_node = next_node


def print_single_list(head_node):
    """
    打印单链表
    :param head_node: ListNode()
    :return: None
    """
    if head_node:
        print(head_node.data, end=" -> ")
        next_node = head_node.next_node
        while next_node:
            print(next_node.data, end=" -> ")
            next_node = next_node.next_node
        print("None")
    else:
        print("None")


def reverse_single_list(head_node):
```

```
"""
GitHub 上的思路是用 3 个指针，自己的思路是用栈来解决，看起来 GitHub
思路简单些

:param head_node: ListNode()
:return: None
"""

pre = head_node
current = head_node.next_node
pre.next_node = None

while current:
    temp = current.next_node
    current.next_node = pre
    pre = current
    current = temp
return pre


if __name__ == "__main__":
    single_list = ListNode(1, ListNode(2, ListNode(3, ListNode(4,
, ListNode(5, ListNode(6, ListNode(7, ListNode(8))))))))
    print_single_list(single_list)

    reversed_single_list = reverse_single_list(single_list)
    print_single_list(reversed_single_list)
```