PaperTest

Q&A

献给每一个在求职路上彷徨跋涉的你

目录

前 言

今天实验室吃了散伙饭，我在明光桥北的日子也进入了倒计时，有些时间竟然可以用散伙饭来作为度量单位的。回想我在这里打酱油似的的三年，终于还是有些后悔，虽然每天把科研学术玩笑似的挂在嘴边，却终究是不学无术，对于实验室的项目，不曾主动去探索过，从来都是为科研而科研，为工程而工程，All in all，我的研究生近三年，是有愧的，我没有实现过当初我来这个地方时所怀抱的那些憧憬，最后也带着那些遗憾离开这个地方。

我从去年三月开始陆陆续续投实习的岗位，但是自身太懒惰，一来本身就是学渣，二来准备的确实不充分，从来也不是一个主动出击的人，找来找去，竟然没有找到，最后自暴自弃似的也不再去找了，当全实验室都不在的时候，我仍然像个例外一样每天去实验室，当时很怕被师弟师妹问到实习怎么样啊之类的问题，因为我不知道怎么样去回答，后来心情也十分不好，因为除了自尊心很高之外没有其他特别高的东西了，于是把所有或明显或潜在或有意或无意的显摆行为腹黑的理解为秀 offer，但是，可以有一时的自怨自艾，最终的改变终究还是靠自己，正如你可以自嘲某一时刻的屌丝，但是不要沉浸在那样的坏情绪负能量里不能自拔。

暑假的时候慢慢的准备，看面试宝典，看数据结构，看 Python，搜之前的笔试题，被同学多次说都考过的题目你看干嘛，一次次的回答说我不会啊，然后把笔试准备或看到的问题集结下来，就有了这个文档，因为是通信背景，没有学过数据结构，也没有学过计算机的很多课程，所以之中，有很多在大牛看来很 Low 很基础的问题吧，大牛们就不必往下看了，这个文档是写给那些对未来不是很有信心或者被现实打击过后终于意识到自己的无知如我的那批人。

但是事实总是吊诡的，文档中所有准备的技术题目主要是为了互联网和外企，国企和银行的考试相对广而范，技术相对极少，有志于国企银行的同学也不必深究这个文档，我最后也签了一个比较奇葩的国企，因为他的笔试竟然完全没有行测，都是数据结构和代码程序，阴差阳错地走到了现在这一步。而互联网，百度一面挂掉了，阿里笔试没有通过，腾讯简历没有过，有道二面聊人生的时候挂掉了，去哪儿一面挂掉了，我总共投出了近 150 份网申或现场简历，涵盖北上广深宁济，通过了其中近 70 家的简历有了初试的机会，参加了其中的 65 家左右，收到了其中 35 家的进一步面试，收到其中 20 家的终面通知，参加了其中的 11 家的终面。

我最终收到了 7 个 offer,2 家运营商，1 家外企，4 家金融机构的 offer，同样涵盖北上广深。如果你看到这里要说我是晒 offer 的话，没错，但你猜对了一半，因为其实算不上晒 offer，因为牛逼的 offer 才算晒，我最终收到的都一般吧，写在这里，只是给诸位准备找工作的同学一个参考，虽然并非每一个人都如我这般的折腾，虽然未必每一个人都像我这样的漫无目的，但是，我想你想要的东西你适合的工作总是在一次次的被拒绝和被鄙视后与你的预期折中后的结果，你以为你适合的你未必适合，他人认为你适合的你也未必适合，你想要得到的你并不总是能够得到，你能做的只有珍惜你现有得到的，并努力追求更好的，得之汝幸，不得汝命。

本文档里的基础知识，多来自每次笔试后论坛上一些讨论的题目，由各网站查到的一些思路，主要来自维基百科，CSDN, CNBLOG 等等，大部分都表明了链接出处，有些未能及时标明，敬请见谅，多数代码由 CSDN 博客中获得，有些写的比较复杂的是我自己写的，有些问题我也不知道答案，仅仅列出了题目，有些题目很简单，亦不赘述，希望这些杂凑起来的片段能够给正在

或即将找工作的你一点点参考的价值。虽然所谓的“团队合作”如今几乎都没有任何隐蔽或者羞耻的色彩，demo，希望你能够获得属于你自己的判断。

从去年九月份开始正式参加校招，被各种笔试面试鄙视，不被人理解，连自己也不理解自己何以这么渣，有次参加某单位群面从头至尾竟然一句话也没有说，大多数时候穿着件薄薄的正装奔波在京城的各个角落，晚上回来的时候在北门买一份炒饭充饥，生日那天我还在金融街群面某单位，虽然后来我把她拒掉了。

今天喝的有点多，就写在这里吧，不管多么阴霾的今天，明天太阳照常升起。

陌生人，祝你有个灿烂的前程，祝你幸福。页脚是我的 csdn，没有什么东西，求关注。

2014-3-15

一． 操作系统及 LINUX

1. 进程与线程

1) 同步机制

临界区（Critical Section）、互斥量（Mutex）、信号量（Semaphore）、事件（Event）

(1) 临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。

(2) 互斥量：采用互斥对象机制。 只有拥有互斥对象的线程才有访问公共资源的权限，因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程访问。互斥不仅能实现同一应用程序的公共资源安全共享，还能实现不同应用程序的公共资源安全共享 .互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。

(3) 信号量：它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目 .信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的 PV 操作相同。它指出了同时访问共享资源的线程最大数目。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。

PV 操作及信号量的概念都是由荷兰科学家 E.W.Dijkstra 提出的。信号量 S 是一个整数，S 大于等于零时代表可供并发进程使用的资源实体数，但 S 小于零时则表示正在等待使用共享资源的进程数。

P 操作申请资源：

（1）S 减 1；

（2）若 S 减 1 后仍大于等于零，则进程继续执行；

（3）若 S 减 1 后小于零，则该进程被阻塞后进入与该信号相对应的队列中，然后转入进程调度。

V 操作释放资源：

（1）S 加 1；

（2）若相加结果大于零，则进程继续执行；

（3）若相加结果小于等于零，则从该信号的等待队列中唤醒一个等待进程，然后再返回原进程继续执行或转入进程调度。

(4) 事 件： 通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作 .

总结：

- 互斥量与临界区的作用非常相似，但互斥量是可以命名的，也就是说它可以跨越进程使用。所以创建互斥量需要的资源更多，所以如果只为了在进程内部是用的话使用临界区会带来速度上的优势并能够减少资源占用量。因为互斥量是跨进程的互斥量一旦被创建，就可以通过名字打开它。在 windows 编程中互斥器(Mutex)可以跨进程使用，而临界区(critical section)只能在进程内部各线程间使用。
- 互斥量（Mutex），信号量（Semaphore），事件（Event）都可以被跨越进程使用来进行同步数据操作，而其他的对象与数据同步操作无关，但对于进程和线程来讲，如果进程和线程在运行状态则为无信号状态，在退出后为有信号状态。所以可以使用 WaitForSingleObject 来等待进程和线程退出。
- 通过互斥量可以指定资源被独占的方式使用，但如果有下面一种情况通过互斥量就无法处理，比如现在一位用户购买了一份三个并发访问许可的数据库系统，可以根据用户购买的访问许可数量来决定有多少个线程/进程能同时进行数据库操作，这时候如果利用互斥量就没有办法完成这个要求，信号灯对象可以说是一种资源计数器。

2) 进程通信

http://en.wikipedia.org/wiki/Inter-process_communication

由于比较容易混淆，我们把进程间通信方法也列在这里做比较。

进程间通信就是在不同进程之间传播或交换信息，那么不同进程之间存在着什么双方都可以访问的介质呢？进程的用户空间是互相独立的，一般而言是不能互相访问的，唯一的例外是共享内存区。但是，系统空间却是“公共场所”，所以内核显然可以提供这样的条件。除此以外，那就是双方都可以访问的外设了。在这个意义上，两个进程当然也可以通过磁盘上的普通文件交换信息，或者通过“注册表”或其它数据库中的某些表项和记录交换信息。广义上这也是进程间通信的手段，但是一般都不把这算作“进程间通信”。因为那些通信手段的效率太低了，而人们对进程间通信的要求是要有一定的实时性。

进程间通信主要包括管道, 系统 IPC inter-process-communication (包括消息队列,信号量,共享存储), SOCKET.

(1) 管道分为有名管道和无名管道，无名管道只能用于父子进程之间的通信，而有名管道则可用于无亲属关系的进程之间。

(2) 消息队列用于运行于同一台机器上的进程间通信，与管道相似；

(3) 共享内存通常由一个进程创建，其余进程对这块内存区进行读写。得到共享内存有两种方式：映射/dev/mem 设备和内存映像文件。前一种方式不给系统带来额外的开销，但在现实中并不常用，因为它控制存取的是实际的物理内存；

(4) 信号量。本质上，信号量是一个计数器，它用来记录对某个资源（如共享内存）的存取状况。一般说来，为了获得共享资源，进程需要执行下列操作：

（1）测试控制该资源的信号量；

（2）若此信号量的值为正，则允许进行使用该资源，进程将进号量减 1；

（3）若此信号量为 0，则该资源目前不可用，进程进入睡眠状态，直至信号量值大于 0，进程被唤醒，转入步骤（1）；

（4）当进程不再使用一个信号量控制的资源时，信号量值加 1，如果此时有进程正在睡眠等待此信号量，则唤醒此进程。

(5) 套接字通信并不为 Linux 所专有，在所有提供了 TCP/IP 协议栈的操作系统中几乎都提供了 socket，而所有这样操作系统，对套接字的编程方法几乎是完全一样的

3) 同步与通信

很明显 2 者有类似，但是差别很大；同步主要是临界区、互斥、信号量、事件。
进程间通信是管道、内存共享、消息队列、信号量、socket
共通之处是，信号量和消息（事件）

(1) 进程(process)与线程(thread)的关系

1. Processes are typically independent, while threads exist as subsets of a process.
2. Processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources.
3. Processes have separate address spaces, whereas threads share their address space.
4. Processes interact only through system-provided inter-process communication mechanisms Context switching between threads in the same process is typically faster than context switching between processes.

4) 进程调度

进程的基本状态包括：等待态或称阻塞态(等待某个事件的完成)，就绪态(等待系统分配处理器以便运行)，运行态(占有处理器正在运行)。

运行态→等待态 往往是由于等待外设，主存等资源分配或等待人工干预而引起的。

等待态→就绪态 则是等待的条件已满足，只需分配到处理器后就能运行。

运行态→就绪态 不是由于自身原因，而是由外界原因使运行状态的进程让出处理器，这时候就变成就绪态。例如时间片用完，或有更高优先级的进程来抢占处理器等。

就绪态→运行态 系统按某种策略选中就绪队列中的进程占用处理器，变成运行态

不会发生: 阻塞态->运行态(运行态是从就绪队列中取值) 就绪态->等待态

进程调度是指按照一定的策略选择一个处于就绪状态的进程，使其获得处理机执行。

下面情况中需要进行进程调度：

(1)正在执行的进程执行完毕。如不选择新的就绪进程，将浪费处理机资源。

(2)执行中进程自己调用阻塞原语将自己阻塞起来进入睡眠等状态。

(3)执行中进程调用了 P 原语操作，从而因资源不足而被阻塞；或调用了 v 原语操作激活了等待资源的进程队列。

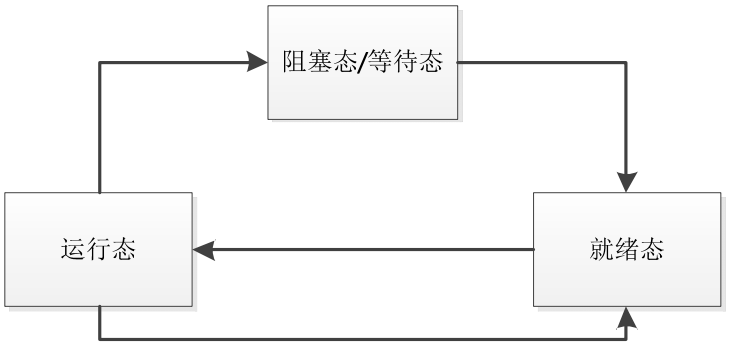
(4)执行中进程提出 I/O 请求后被阻塞。

(5)在分时系统中时间片已经用完。

(6)在执行完系统调用等系统程序后返回用户进程时，这时可看作系统进程执行完毕，从而可调度选择一新的用户进程执行。

(7)就绪队列中的某进程的优先级变得高于当前执行进程的优先级，从而也将引发进程调度。

进程调度算法包括先进先出算法，短进程优先算法，轮转法及多级反馈队列等。



5) 多进程与多线程的区别

进程是资源分配的最小单位，线程是 CPU 调度的最小单位；进程编程调戏简单可靠性高但是创建销毁开销大；线程正相反，开销小，切换速度快，但是编程调试相对复杂。

6) 死锁

死锁 (deallocks)： 是指两个或两个以上的进程（线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程（线程）在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象死锁。

产生死锁的四个必要条件:

(1) 互斥条件：一个资源每次只能被一个进程（线程）使用。

(2) 请求与保持条件：一个进程（线程）因请求资源而阻塞时，对已获得的资源保持不放。

(3) 不剥夺条件 ：此进程（线程）已获得的资源，在末使用完之前，不能强行剥夺。

(4) 循环等待条件 ：多个进程(线程)之间形成一种头尾相接的循环等待资源关系。

死锁的预防：

因为独占资源必须以互斥方式进行访问，所以要预防死锁只能从破坏后三个条件下手了。最简单的消除死锁的办法是重启系统。更好的办法是终止一个进程的运行。同样也可以把一个或多个进程回滚到先前的某个状态。如果一个进程被多次回滚，迟迟不能占用必需的系统资源，可能会导致进程饥饿。

1.破坏占有并等待条件：

要破坏这个条件，就要求每个进程必须一次性的请求它们所需要的所有资源，若无法全部获取就等待，直到满足为止，也可以采用事务机制，确保可以回滚，即把获取、释放资源做成原子性的。这个方法实现起来可能会比较困难，因为某些情况下，进程并不能事先直到自己需要哪些资源，也有时候并不需要分配到所有资源就可以运行。

2.破坏不可剥夺条件：

一个已占有资源的进程若要再申请新的资源，它必须先释放已占有的资源。若随后再需要这些资源，需要重新申请。

3.破坏循环等待条件：

将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作。

例：哲学家就餐问题(RR 2014 笔试)

详见：

https://zh.wikipedia.org/wiki/%E5%93%B2%E5%AD%A6%E5%AE%B6%E5%B0%B1%E9%A4%90%E9%97%AE%E9%A2%98

7) 进程与线程

进程可以有自己独立的栈等资源，而线程将继承或共享：进程的代码段、进程的公有数据(利用这些共享数据，线程很容易实现相互之间的通讯)、进程打开的文件描述符、信号的处理器、进程的当前目录和进程用户 ID 与进程组 ID.

进程是系统资源分配的最小单位;而线程是 CPU 调度的最小单位。

2. FORK

在 UNIX 里，除了进程 0（即 PID=0 的交换进程，Swapper Process）以外的所有进程都是由其他进程使用系统调用 fork 创建的，这里调用 fork 创建新进程的进程即为父进程，而相对应的为其创建出的进程则为子进程，因而除了进程 0 以外的进程都只有一个父进程，但一个进程可以有多个子进程。如果用户 fork 一个子进程后 exit，子进程的父进程将为 init.

进程在 linux 中呈树状结构，init 为根节点，其它进程均有父进程，某进程的父进程就是启动这个进程的进程，这个进程叫做父进程的子进程。

fork 的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程。fork 函数被调用一次但返回两次。两次返回的唯一区别是子进程中返回 0 值而父进程中返回子进程 ID。

进程可以看做程序的一次执行过程。在 linux 下，每个进程有唯一的 PID 标识进程。PID 是一个从 1 到 32768 的正整数，其中 1 一般是特殊进程 init，其它进程从 2 开始依次编号。当用完 32768 后，从 2 重新开始。

linux 中有一个叫进程表的结构用来存储当前正在运行的进程。可以使用“ps aux”命令查看所有正在运行的进程。

例: http://blog.csdn.net/xcxinghai/article/details/9093309

给出如下 C 程序，在 linux 下使用 gcc 编译：

```
1.  #include "stdio.h"
2.  #include "sys/types.h"
3.  #include "unistd.h"
4.   int main()
5.  {
6.      pid_t pid1;
7.      pid_t pid2;
8.      pid1 = fork();
```

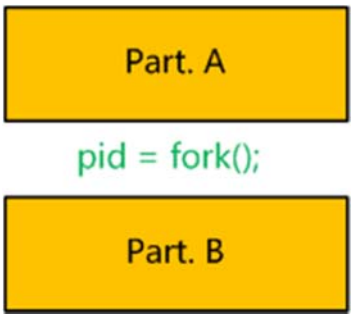
```
9.      pid2 = fork();
10.     printf("pid1:%d, pid2:%d\n", pid1, pid2);
11. }
```

要求如下：

已知从这个程序执行到这个程序的所有进程结束这个时间段内，没有其它新进程执行。 1、请说出执行这个程序后，将一共运行几个进程。

2、如果其中一个进程的输出结果是“pid1:1001, pid2:1002”，写出其他进程的输出结果（不考虑进程执行顺序）。

解答: 解题的关键就是要认识到 fork 将程序切成两段。看下图：



上图表示一个含有 fork 的程序，而 fork 语句可以看成将程序切为 A、B 两个部分。然后整个程序会如下运行：

step1、设由 shell 直接执行程序，生成了进程 P。P 执行完 Part. A 的所有代码。

step2、当执行到 pid = fork();时，P 启动一个进程 Q，Q 是 P 的子进程，和 P 是同一个程序的进程。Q 继承 P 的所有变量、环境变量、程序计数器的当前值。

step3、在 P 进程中，fork()将 Q 的 PID 返回给变量 pid，并继续执行 Part. B 的代码。

step4、在进程 Q 中，将 0 赋给 pid，并继续执行 Part. B 的代码。

这里有三个点非常关键:

1、P 执行了所有程序，而 Q 只执行了 Part. B，即 fork()后面的程序。（这是因为 Q 继承了 P 的 PC-程序计数器）

2、Q 继承了 fork()语句执行时当前的环境，而不是程序的初始环境。

3、P 中 fork()语句启动子进程 Q，并将 Q 的 PID 返回，而 Q 中的 fork()语句不启动新进程，仅将 0 返回。

解题:

下面利用上文阐述的知识进行解题。这里我把两个问题放在一起进行分析。

1、从 shell 中执行此程序，启动了一个进程，我们设这个进程为 P0，设其 PID 为 XXX（解题过程不需知道其 PID）。

2、当执行到 pid1 = fork();时，P0 启动一个子进程 P1，由题目知 P1 的 PID 为 1001。我们暂且不管 P1。

3、P0 中的 fork 返回 1001 给 pid1，继续执行到 pid2 = fork();，此时启动另一个新进程，设为 P2，由题目知 P2 的 PID 为 1002。同样暂且不管 P2。

4、P0 中的第二个 fork 返回 1002 给 pid2，继续执行完后续程序，结束。所以，P0 的结果为“pid1:1001, pid2:1002”。

5、再看 P2，P2 生成时，P0 中 pid1=1001，所以 P2 中 pid1 继承 P0 的 1001，而作为子进程 pid2=0。P2 从第二个 fork 后开始执行，结束后输出“pid1:1001, pid2:0”。

6、接着看 P1，P1 中第一条 fork 返回 0 给 pid1，然后接着执行后面的语句。而后面接着的语句是 pid2 = fork();执行到这里，P1 又产生了一个新进程，设为 P3。先不管 P3。

7、P1 中第二条 fork 将 P3 的 PID 返回给 pid2，由预备知识知 P3 的 PID 为 1003，所以 P1 的 pid2=1003。P1 继续执行后续程序，结束，输出“pid1:0, pid2:1003”。

8、P3 作为 P1 的子进程，继承 P1 中 pid1=0，并且第二条 fork 将 0 返回给 pid2，所以 P3 最后输出“pid1:0, pid2:0”。

9、至此，整个执行过程完毕。

所得答案：

1、一共执行了四个进程。（P0, P1, P2, P3）

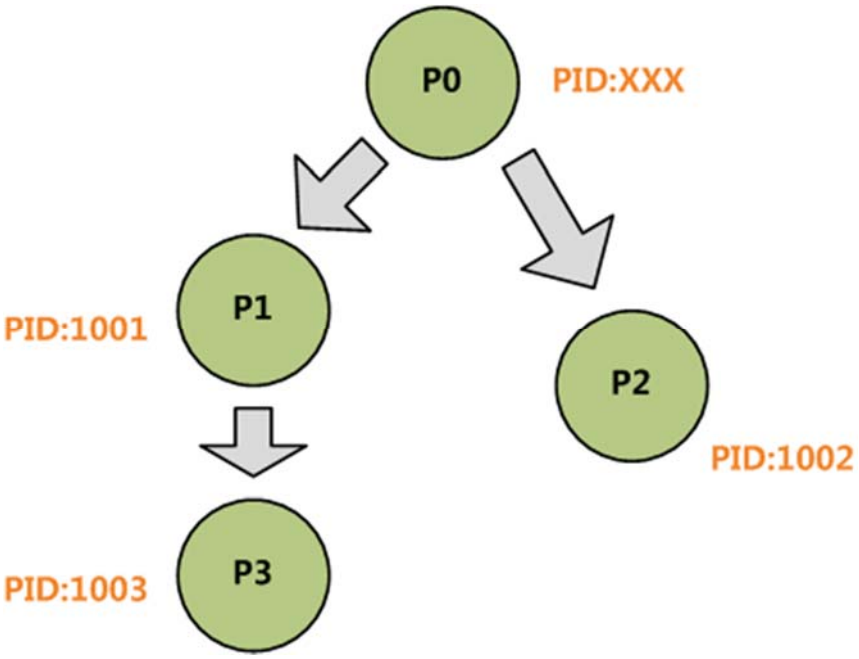
2、另外几个进程的输出分别为：

pid1:1001, pid2:0

pid1:0, pid2:1003

pid1:0, pid2:0

进一步可以给出一个以 P0 为根的进程树：



3. LINUX

(1) Linux 如何查看内存使用率(Ali 2013 面试)

可以使用 top 命令 按 q 键可退出该应用

判断一个系统的负载可以使用 top，uptime 等命令去查看，它分别记录了一分钟、五分钟、以及十五分钟的系统平均负载，系统平均负载被定义为在特定时间间隔内运行队列中(在 CPU 上运行或者等待运行多少进程)的平均进程数。例如：

$ uptime

09:50:21 up 200 days, 15:07, 1 user, load average: 0.27, 0.33, 0.37

见：http://heipark.iteye.com/blog/1340384

http://linux.chinaitlab.com/administer/809477.html

(2) linux 中文件权限共有十位，e.g.:-rw-rw-r--,最前面那个'-'代表的是类型(d 表示目录文件，-表示普通文件，b 表示块设备文件等),左边那三个 rw- 代表的是所有者（owners）,中间那三个 rw- 代表的是组群（group）,最后那三个 r-- 代表的是其他人（others),其中 r 表示文件可以被读（read）,w 表示文件可以被写（write）,x 表示文件可以被执行（如果它是程序的话）,- 表示相应的权限还没有被授予.其中表示成数字时 r:4,w:2,x:1,-:0.(Nokia 2013 笔试)

(3)

4. RAID

RAID is short for redundant arrays of independent disks 独立/廉价磁盘冗余阵列。

其基本思想就是把多个相对便宜的硬盘组合起来，成为一个硬盘阵列组，使性能达到甚至超过一个价格昂贵、容量巨大的硬盘。根据选择的版本不同，RAID 比单颗硬盘有以下一个或多个方面的好处：增强资料整合度，增强容错功能，增加处理量或容量。另外，磁碟阵列对于电脑来说， 看起来就像一个单独的硬盘或逻辑存储单元。

RAID 把多个硬盘组合成为一个逻辑磁区，因此，作业系统只会把它当作一个硬盘。RAID 目前有 0~7 及其中的若干两种组合等十余种不同的等级，不同的 RAID 方法各有其优缺点，

RAID 0: 将多个硬盘合为一个，一个损坏，即不可用，存取速度快但可靠性低

RAID 1: 镜像方式存储，利用率仅 1/2，但可靠性高

5. 测 试

1) 桩模块与驱动模块

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的部分。主模块作为驱动模块,与之直接相连的模块用桩模块代替。在集成测试前要为被测模块编制一些模拟其下级模块功能的“替身”模块，以代替被测模块的接口，接受或传递被测模块的数据，这些专供测试用的“假”模块称为被测模块的桩模块。

驱动模块是用来模拟被测试模块的上一级模块，相当于被测模块的主程序。它接收数据，将相关数据传送给被测模块，启用被测模块，并打印出相应的结果。

2) 脚本测试

手写脚本程序解决问题如下问题（一个文件中全是 ip 地址，读取这个文件，去掉其中的重复项，并输出最大的一个 ip。在纸上用脚本程序完成）（baidu 2014 面试）

cat xxx | sort | uniq -c |sort -nr |head -1

uniq 只是将邻近的重复行合并 uniq -c 是统计行的重复数 返回的第一列就是出现的次数 此脚本可以统计出现次数最多的 ip

3) 性能测试

测试简介：

https://zh.wikipedia.org/wiki/%E8%BD%AF%E4%BB%B6%E6%B5%8B%E8%AF%95

性能测试是通过自动化的测试工具模拟多种正常、峰值以及异常负载条件来对系统的各项性能指标进行测试。负载测试和压力测试都属于性能测试，两者可以结合进行。通过负载测试，确定在各种工作负载下系统的性能，目标是测试当负载逐渐增加时，系

统各项性能指标的变化情况。压力测试是通过确定一个系统的瓶颈或者不能接收的性能点，来获得系统能提供的最大服务级别的测试。

白盒逻辑测试 性能测试工具 等；

6. 堆栈数据代码区

一个由 c/C++编译的程序占用的内存分为以下几个部分:

在 Unix 中，从高地址到低地址依次为 stack heap data text

见：http://blog.chinaunix.net/uid-23860671-id-272257.html

(1) 栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

(2) 堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由 OS 回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。

(3) 全局区（静态区）（static）全局变量和静态变量的存储是放在一块的，**初始化的全局变量和静态变量在一块区域 data 段， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域 bss 段**。程序结束后有系统释放

(4) 文字常量区—常量字符串就是放在这里的。常量字符串不能修改，否则程序会在运行期崩溃.程序结束后由系统释放

(5) 程序代码区—存放函数体的二进制代码。

例程:

```
//main.cpp
int a = 0; 全局初始化区
char *p1; 全局未初始化区
main()
{
int b; 栈
char s[] = "abc"; 栈
char *p2; 栈
char *p3 = "123456"; 123456\0 在常量区，p3 在栈上。
static int c =0； 全局（静态）初始化区
p1 = (char *)malloc(10);
p2 = (char *)malloc(20);
分配得来得 10 和 20 字节的区域就在堆区。
strcpy(p1, "123456"); 123456\0 放在常量区，编译器可能会将它与 p3 所指向的"123456"优化成一个地方。
}
```

链接：http://fly-top.blog.163.com/blog/static/172755112201271541159547/

申请后系统的响应

栈：只要栈的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。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地

址处记录本次分配的大小，这样，代码中的 delete 语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

申请大小的限制

栈：在 Windows 下,栈是向低地址，从上到下生长的扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS 下，栈的大小是 2M（也有的说是 1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示 overflow。因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

申请效率的比较：

栈由系统自动分配，速度较快。但程序员是无法控制的。

堆是由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.另外，在 WINDOWS 下，最好的方式是用 VirtualAlloc 分配内存，他不是在堆，也不是在栈是直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活。

7. 文件读写

1) FCLOSE()

功能:关闭一个流。注意：使用 fclose()函数就可以把缓冲区内最后剩余的数据输出到磁盘文件中，并释放文件指针和有关的缓冲区。

如果流成功关闭，fclose 返回 0，否则返回 EOF（-1）。

如果流为 NULL，而且程序可以继续执行，fclose 设定 error number 给 EINVAL，并返回 EOF。

2) FOPEN()

函数功能：打开一个文件

函数原型：FILE * fopen(const char * path,const char * mode);

返回值：文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回 NULL，并把错误代码存在 errno 中。

mode 有下列几种形态字符串:

r 以只读方式打开文件，该文件必须存在

r+ 以可读写方式打开文件，该文件必须存在

rb+ 读写打开一个二进制文件，允许读写数据

rw+ 读写打开一个文本文件，允许读和写

w 打开只写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在则建立该文件。

w+ 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。

a 以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。（EOF 符保留）

a+ 以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留。（原来的 EOF 符不保留）

wb 只写打开或新建一个二进制文件；只允许写数据。

wb+ 读写打开或建立一个二进制文件，允许读和写。

ab+ 读写打开一个二进制文件，允许读或在文件末追加数据。

at+ 打开一个叫 string 的文件，a 表示 append,就是说写入处理的时候是接着原来文件已有内容写入，不是从头写入覆盖掉，t 表示打开文件的类型是文本文件，+号表示对文件既可以读也可以写。

上述的形态字符串都可以再加一个 b 字符，如 rb、w+b 或 ab+等组合，加入 b 字符用来告诉函数库以二进制模式打开文件。如果不加 b，表示默认加了 t，即 rt,wt,其中 t 表示以文本模式打开文件。由 fopen()所建立的新文件会具有 S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH(0666)权限，此文件权限也会参考 umask 值。

有些 C 编译系统可能不完全提供所有这些功能，有的 C 版本不用"r+","w+","a+",而用"rw","wr","ar"等，读者注意所用系统的规定。

3) FSEEK()

函数功能：重定位流(数据流/文件)上的文件内部位置指针

注意：不是定位文件指针，文件指针指向文件/流。位置指针指向文件内部的字节位置，随着文件的读取会移动，文件指针如果不重新赋值将不会改变指向别的文件。

函数原型：int fseek(FILE *stream, long offset, int fromwhere);

函数设置文件指针 stream 的位置。如果执行成功，stream 将指向以 fromwhere（偏移起始位置：文件头 0(SEEK_SET)，当前位置 1(SEEK_CUR)，文件尾 2(SEEK_END)）为基准，偏移 offset（指针偏移量，正数）个字节的位置。如果执行失败(比如 offset 超过文件自身大小)，则不改变 stream 指向的位置。成功，返回 0，失败返回-1，并设置 errno 的值，可以用 perror()函数输出错误。

4) FREAD()

函数原型：size_t fread (void *buffer,size_t size,size_t count,FILE *stream) ;

fread 是一个函数。从一个文件流中读数据，最多读取 count 个元素，每个元素 size 字节，如果调用成功返回实际读取到的元素个数，如果不成功返回 0。

5) FWRITE()

函数原型：size_t fwrite(const void* buffer, size_t size, size_t count, FILE* stream);

函数主要负责写入；具体写在什么地方，与文件的打开模式有关，如果是 w+，则是从 file pointer 指向的地址开始写，替换掉之后的内容，文件的长度可以不变，stream 的位置移动 count 个数；如果是 a+，则从文件的末尾开始添加，文件长度加大。

fseek 对此函数有作用，但是 fwrite 函数写到用户空间缓冲区，并未同步到文件中，所以修改后要将内存与文件同步可以用 fflush（FILE *fp）函数同步。

8. 硬链接与软链接

UNIX 文件系统提供了一种将不同文件链接至同一个文件的机制，我们称这种机制为链接。它可以使得单个程序对同一文件使用不同的名字。这样的好处是文件系统只存在一个文件的副本，系统简单地通过在目录中建立一个新的登记项来实现这种连接。

硬链接, Hard link: 该登记项具有一个新的文件名和要连接文件的 inode 号，文件的目录登记项就是所谓的文件硬链接（硬链接，目录登记项： 文件名-->文件的 inode；文件名可以有多个，但文件 inode 只是一个）。不论一个文件有多少硬链接，在磁盘上只有一个描述它的 inode， 只要该文件的链接数不为 0，该文件就保持存在。硬链接不能跨越文件系统，为避免无限递归，一般也不能在目录上建立 hard link。硬连接是直接建立在节点表上的(inode)，建立硬连接指向一个文件的时候，会更新节点表上面的计数值。举个例子，一个文件被连接了两次（硬连接），这个文件的计数值是 3，而无论通过 3 个文件名中的任何一个访问，效果都是完全一样的，但是如果删除其中任意一个，都只是把计数值减 1，不会删除实际的内容的，（任何存在的 文件本身就算是一个硬连接）只有计数值变成 0 也就是没有任何硬连接指向的时候才会真实的删除内容。对任何一个硬链接文件修改，都会对原文件进行修改。

软链接，symbolic link or soft link: 它是指向另一个文件的特殊文件，这种文件的数据部分仅包含它所要链接文件的路径名。软链接是为了克服硬链接的不足而引入的，软链接不直接使用 inode 号作为文件指针，而是使用文件路径名作为指针（软链接：文件名+数据部分--> 目标文件的路径名）。软件有自己的 inode，并在磁盘上有一小片空间存放路径名。因此，软链接能够跨文件系统，也可以和目录链接; 其二，软链接可以对一个不存在的文件名进行链接，但直到这个名字对应的文件被创建后，才能打开其链接。当软链接指向的文件删除重命名或移动后，软链接并不会发生更新，会维持原来的值不变。

二． C++与面向对象语言

1. C 语言基础问题

1) 关于 CONST 的问题

如果 const 关键字不涉及到指针，我们很好理解，下面是涉及到指针的情况：

```
int b = 500;
const int* a = &b;      [1]
int const *a = &b;      [2]
int* const a = &b;      [3]
const int* const a = &b; [4]
```

如果你能区分出上述四种情况，那么，恭喜你，你已经迈出了可喜的一步。不知道，也没关系，我们可以参考《Effective c++》Item21 上的做法，如果 const 位于星号的左侧，则 const 就是用来修饰指针所指向的变量，即指针指向为常量；如果 const 位于星号的右侧，const 就是修饰指针本身，即指针本身是常量。因此，[1]和[2]的情况相同，都是指针所指向的内容为常量（const 放在变量声明符的位置无关），这种情况下不允许对内容进行更改操作，如不能*a = 3 ；[3]为指针本身是常量，而指针所指向的内容不是常量，这种情况下不能对指针本身进行更改操作，如 a++是错误的；[4]为指针本身和指向的内容均为常量。

const 本身可以还可以修饰函数返回值或放在函数名后修饰函数；修饰函数返回值表示返回值为常量，不可以修改；如修饰函数 int ff(void)const;则表示函数不能够修改函数的成员变量；

2) 浅复制与深复制

(4) 浅复制:被复制对象的所有变量都含有与原来的对象相同的值,而其所有的对其他对象的引用都仍然指向原来的对象。浅复制在复制时，将这个对象的值字段和引用字段（均为非静态字段）全部复制过去，获得了这个对象的值和地址。即：当其中一个对象的引用字段所指向的地址中的变量变化时，所有浅复制对象中的该引用字段都会发生变化。（shallow copy）

(5) 深复制：被复制对象会将所有非引用类型的字段复制给新对象，同时将引用类型所指向地址中存的对象复制给新的对象。浅复制和深复制的区别仅在于对引用类型的对待上,一个是复制的地址,一个是复制的地址指向位置的数据.(deep copy)两者结合后成为 lazy copy.

3) 逆波兰表达式

逆波兰表示法（Reverse Polish notation，RPN，或逆波兰记法），是一种是由波兰数学家扬·武卡谢维奇 1920 年引入的数学表达式方式，在逆波兰记法中，所有操作符置于操作数的后面，因此也被称为后缀表示法。逆波兰记法不需要括号来标识操作符的优先级，以利用堆

栈结构和减少计算机内存访问。(正常表达式在二叉树中为中序遍历而逆波兰表示为后序遍历前序遍历为波兰表达式)

它的优势在于只用两种简单操作，入栈和出栈就可以搞定任何普通表达式的运算。其运算方式如下：如果当前字符为变量或者为数字，则压栈，如果是运算符，则将栈顶两个元素弹出作相应运算，结果再入栈，最后当表达式扫描完后，栈里的就是结果。

例子: a+b ---> a,b,+

a+(b-c) ---> a,b,c,-,+

a+(b-c)*d ---> a,b,c,-,d,*,+

a+d*(b-c)--->a,d,b,c,-,*,+

a=1+3 ---> a=1,3 +

4) C 语言变长参数

可变长参数：顾名思义，就是函数的参数长度（数量）是可变的。比如 C 语言的 printf 系列的（格式化输入输出等）函数，都是参数可变的。下面是 printf 函数的声明：

int printf (const char * format, ...);

可变参数函数声明方式都是类似的。

C 语言可变参数通过三个宏（va_start、va_end、va_arg）和一个类型（va_list）实现的，

void va_start (va_list ap, paramN);参数：ap: 可变参数列表地址 paramN: 确定的参数

功能：初始化可变参数列表（把函数在 paramN 之后的参数地址放到 ap 中）。

void va_end (va_list ap); 功能：关闭初始化列表（将 ap 置空）。

type va_arg (va_list ap, type); 功能：返回下一个参数的值。

va_list ：存储参数的类型信息。

好了，综合上面 3 个宏和一个类型可以猜出如何实现 C 语言可变长参数函数：用 va_start 获取参数列表（的地址）存储到 ap 中，用 va_arg 逐个获取值，最后用 va_arg 将 ap 置空。

http://www.cnblogs.com/chinazhangjie/archive/2012/08/18/2645475.html

5) 调用约定

(calling convention)在计算机科学中，调用约定是一种定义子过程从调用处接受参数以及返回结果的方法的约定。不同调用约定的区别在于：

a)参数和返回值放置的位置（在寄存器中；在调用栈中；两者混合）

b)参数传递的顺序（或者单个参数不同部分的顺序）

c)调用前设置和调用后清理的工作，在调用者和被调用者之间如何分配

d)被调用者可以直接使用哪一个寄存器有时也包括在内。(否则的话被当成 ABI 的细节)

e)哪一个寄存器被当作 volatile 的或者非 volatile 的，并且如果是 volatile 的，不需要被调用者恢复

调用约定	入栈参数清理	参数入栈顺序
cdecl	调用者处理	右->左
stdcall	函数自己处理	右 ->左
fastcall	函数自己处理	依赖于编译器

pascal 函数自己处理 左->右

由于_cdecl 调用方式的参数内存栈由调用者维护，所以变长参数 **Variable-length argument** 的函数能（也只能）使用这种调用约定。_cdecl 是 VC 中的默认调用方式。

_thiscall 是 C++ 语言特有的一种调用方式，用于类成员函数的调用约定。如果参数确定，this 指针存放于 ECX 寄存器，函数自身清理堆栈；如果参数不确定，this 指针在所有参数入栈后再入栈，调用者清理栈。__thiscall 不是关键字，程序员不能使用。参数按照从右至左的方式入栈。

VC 中默认调用是 __cdecl 方式，Windows API 使用 __stdcall 调用方式，在 DLL 导出函数中，为了跟 Windows API 保持一致，建议使用 __stdcall 方式。

6) 寄存器

EAX 与 EDX: 在 C 语言中，函数的返回值存放在寄存器 EAX 和 EDX 中，EAX 存放低 32 位，EDX 存放高 32 位，如果返回值不足 32 位，则只用到 EAX。(OP/WMSJ)

ESP 与 EBP: ESP-栈指针寄存器(extended stack pointer)，该指针永远指向系统栈最上面一个栈帧的栈顶。EBP：基址指针寄存器(extended base pointer)，该指针永远指向系统栈最上面一个栈帧的底部。

ESI 与 EDI: 寄存器 ESI 与 EDI 称为变址寄存器(Index Register)，它们主要用于存放存储单元在段内的偏移量，用它们可实现多种存储器操作数的寻址方式，为以不同的地址形式访问存储单元提供方便。

7) 关于内联函数 INLINE

内联函数在编译器嵌入适当位置。如果成员函数在类内直接定义，不需要加 inline 关键字，编译器直接内化为内联函数；
内联函数的定义对编译器而言必须是可见的，以便编译器能够在调用点内联展开该函数的代码。此时，仅有函数原型是不够的。

内联函数一般与宏相比较，宏采用的是直接代入的方式，不进行类型检查。

把内联函数的定义放在头文件中，可以确保在调用函数时所使用的定义是相同，并且保证在调用点该函数的定义对编译器可见。

如果 inline 函数在 cpp 内实现，则该函数仅能在该 cpp 内使用。尽量不要放在 cpp 中实现。

8) PACK

```
#pragma pack(n)
.....
#pragma pack(pop)
```

编译器中提供了#pragma pack(n)来设定变量以 n 字节对齐方式。n 字节对齐就是说变量存放的起始地址的偏移量有两种情况：第一、如果 n 大于等于该变量所占用的字节数，那么偏移量必须满足默认的对齐方式，第二、如果 n 小于该变量的类型所占用的字节数，那么偏移量为 n 的倍数，不用满足默认的对齐方式。(Perfect world)

9) 正则表达式

正则表达式(regular expression)，正则表达式使用单个字符串来描述、匹配一系列符合某个句法规则的字符串。在很多文本编辑器里，正则表达式通常被用来检索、替换那些符合某个模式的文本。正则表达式通常缩写成“regex”

基本语法：

(1) | 竖直分隔符代表选择。例如“gray|grey”可以匹配 grey 或 gray

(2) 常见数量限定符包括 +、?和 *（不加数量限定代表仅出现一次）：

+ 加号代表前面的字符必须至少出现一次。(1 次或多次)。例如，“goo+gle”可以匹配 google、gooogle、goooogle 等;

? 问号代表前面的字符最多只可以出现一次。(0 次或 1 次)。例如，“colou?r”可以匹配 color 或者 colour;

* 星号代表前面的字符可以不出现，也可以出现一次或者多次。(0 次或 1 次或多次)。例如，“0*42”可以匹配 42、042、0042、00042 等。

(3) 圆括号可以用来定义操作符的范围和优先度。例如，“gr(a|e)y”等价于“gray|grey”，“(grand)?father”匹配 father 和 grandfather。

(4) [xyz] 匹配所包含的任意一个字符。 [abc]可以匹配“plain”中的“a”。

[^xyz] 代表排除型字符。[^abc]可以匹配 plain 中的 plin

(5) [a-z]匹配指定范围内的任意字符。“[a-z]”可以匹配 a 到 z 内的小写字母。

[^a-z]排除型的字符范围。匹配任何不在指定范围内的任意字符。

(6) {n} o{2}不能匹配 Bob 中的 o，但是能匹配“food”中的两个 o；

{n,} 至少匹配 n 次。如 o{2,}可以匹配 foooood 中的所有 o

{n,m}m 和 n 非负，n<=m。最少匹配 n 次且最多匹配 m 次。o{1,3}可以匹配 Fooooood 中的前三个 o；

(7) ^ 匹配输入字符串的开始位置。

$ 匹配输入字符串的结束位置。

10)内存操作

```
char *GetMemory1(void)
{
 char p[] = "hello world";
 return p;
}
```

```
void Test(void)
{
 char *str = NULL;
 str = GetMemory1();
 printf(str);
}
```

以上程序可能**输出乱码**，因为 GetMemory 返回的是指向“栈内存”的指针，该指针的地址不是 NULL，但其原现的内容已经被清除，新内容不可知。

```
char *GetMemory1(void)
{
```

```
 char *p = "hello world";

 return p;
```

```
}
void Test(void)
{
    char *str = NULL;
    str = GetMemory1();
    printf(str);
}
```

此时将可以输出正确结果，因为 char *p 将声明一个全局的字符串，但是该字符串不能修改，如果需要修改可以将其改为 static char p[].

```
void GetMemory2(char *p)
{
 p = (char *)malloc(100);
}
```

```
char * GetMemory4()
{
 char *p = (char *)malloc(100);
 return p;}
```

```
void GetMemory5(char *&p)
{ p=(char*)malloc(100);}
```

以上左侧程序崩溃。因为 **GetMemory 并不能传递动态内存**，Test 函数中的 str 一直都是 NULL。strcpy(str, "hello world");将使程序崩溃。右侧程序可以返回内存空间使用。GetMemory5 也可以用于返回内存；

```
void GetMemory3(char **p, int num)
{
 *p = (char *)malloc(num);
}
```

```
void Test(void)
{
char *str = NULL;
GetMemory(&str, 100);
strcpy(str, "hello");
printf(str);
}
```

以上程序能够输出 hello 但是可能会发生内存泄漏。

```
void Test(void)
{
    char *str = (char *) malloc(100);
    strcpy(str, “hello”);
    free(str);
    if(str != NULL)
    {
        strcpy(str, “world”);
        printf(str);
    }
}
```

以上程序篡改动态内存区的内容，后果难以预料，非常危险。因为 free(str);之后，str 成为悬垂指针，if(str != NULL)语句不起作用。结果仍然可能输出 world.

11)四种强制类型转换

(1) const_cast 用于类型转换掉表达式的 const 或 volatileness 属性。

(2) dynamic_cast 它被用于安全地沿着类的继承关系向下进行类型转换。也即可以用 dynamic_cast 把指向基类的指针或引用转换成指向其派生类或其兄弟类的指针或引用，而且可以知道能否成功转换。失败的转换将返回空指针（当对指针进行类型转换时）或者抛出异常（当对引用进行类型转换时 bad_cast http://dwz.cn/aYGlG）。dynamic_casts 在帮助你浏览继承层次上是有限制的。它不能被用于缺乏虚函数的类型上，也不能用它来转换掉 constness。基类指针或引用 -> 派生类指针或引用

(3) static_cast 在功能上基本上与 C 风格的类型转换一样强大，含义也一样。但没有运行时类型检查来保证转换的安全性。可以用于用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换，进行上行转换（把派生类的指针或引用转换成基类表示）是安全的；进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的，此时需要使用 dynamic_cast；用于基本数据类型之间的转换，如把 int 转换成 char，把 int 转换成 enum。这种转换的安全性也要开发人员来保证；把空指针转换成目标类型的空指针；把任何类型的表达式转换成 void 类型。

(4) reinterpret_cast 使用这个操作符的类型转换，其的转换结果几乎都是执行期定义（implementation-defined）。因此，使用 reinterpret_casts 的代码很难移植。reinterpret_casts 的最普通的用途就是在函数指针类型之间进行转换。操作符修改了操作数类型,但仅仅是重新解释了给出的对象的比特模型而没有进行二进制转换。reinterpret_cast 是为了映射到一个完全不同类型的意思，这个关键词在我们需要把类型映射回原有类型时用到它。我们映射到的类型仅仅是为了故弄玄虚和其他目的，这是所有映射中最危险的。

12)SIZEOF

(1) 以下代码中的两个 sizeof 用法有问题吗？

```
void UpperCase( char str[] ) // 将 str 中的小写字母转换成大写字母
{for( size_t i=0; i<sizeof(str)/sizeof(str[0]); ++i )
      if( 'a'<=str[i] && str[i]<='z' ) str[i] -= ('a'-'A' );}
 char str[] = "aBcDe";
cout << "str 字符长度为: " << sizeof(str)/sizeof(str[0]) << endl;
UpperCase( str );
cout << str << endl;
```

答：函数内的 sizeof 有问题。根据语法，sizeof 如用于数组，只能测出静态数组的大小，无法检测动态分配的或外部数组大小。函数外的 str 是一个静态定义的数组，因此其大小为 6，函数内的 str 实际只是一个指向字符串的指针，没有任何额外的与数组相关的信息，因此 sizeof 作用于上只将其当指针看，一个指针为 4 个字节，因此返回 4。

(2) 64 位与 32 位的不同

在 32 位系统中：

Double 占 8 个字节　　float 4 个字节

long / int 占 4 个字节　指针 4 个字节　string 占四个字节

short 2 个字节　　char 1 个字节

没有成员变量的结构或类的大小为 1，因为必须保证结构或类的每一个实例在内存中都有唯一的地址。多重继承的空类其大小仍然为 1。

当数组作为参数传入函数时，退化为指针。

char *ss1 =”0123456789” sizof(ss1)=4

char ss1[]=”0123456789” sizeof(ss2)=11

而在 64 位系统条件下：与 32 位不同点在于，指针与 long 均变为 8 个字节，而在 32 位条件下均为 4 个字节，其他条件不变

13)动态库与静态库

静态链接库是.lib 格式的文件，一般在工程的设置界面加入工程中，程序编译时会把 lib 文件的代码加入你的程序中因此会增加代码大小，你的程序一运行 lib 代码强制被装入你程序的运行空间，不能手动移除 lib 代码。

动态链接库是程序运行时动态装入内存的模块，格式*.dll，在程序运行时可以随意加载和移除，节省内存空间。静态链接库与动态链接库都是共享代码的方式。

详见：http://blog.sina.com.cn/s/blog_61ba4898010153zu.html

14)压栈・优先级・位序・宏・UNION・指针

(1) C++程序中程序正确运行时会返回 0；

(2) printf 函数计算参数时是**从右向左**压栈的。 `printf( “%d,%d\n” , *ptr, *(++ptr));`

(3) `unsigned char b = ~a>>4;` “>>” 优先级比 ”~” 高。char 有 8 位。

(4) m = m & (m-1) 每运行一次会使 m 的 10 二进制数种减少一个 1.

(5) a ^ (a ^ b) = b

(6) 字符串与字符数组的区别在于，前者会在结尾自动添加 ’\0’

(7) 有 printf("%%d,%%d", a, b);输出？(Nokia)

如想试图输出'%',需要使用'%%'，故上述式子会输出'%d,%d';

(8) 符号数与无符号数相加时，会将有符号数隐式转换为无符号数。

(9) #include “file2.c” 则默认指用户当前目录中的文件。系统先在用户当前目录中寻找要包含的文件,若找不到,再按标准方式查找。如果程序中要包含的是用户自己编写的文件,宜用双撇号形式。

#include <iostream>系统到系统目录中寻找要包含的文件,如果找不到,编译系统就给出出错信息。

(10) Little endian eg:0x1234

Little Endian 就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

e.g:0x4000 34

0x4001 12

Big Endian 就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

e.g:0x4000 12

0x4001 34

检测是否大端序：

```
bool IsBigEndian()
```

```
{
    union
    {
        unsigned short a ;
        char b ;
    } c;

 c.a =0x0102 ;
if(c.b ==1)
    return true ;
 else
    return false ;
}
```

(11) ASCII 码，字符与码表对照，小写字母的 ASCII 码大

a ASCII 为 97

A ASCII 为 65

0~9 为 48~57

(12) 不用除法，实现 A/3

```
Test_divide3(int A)
{
    int n=0,A=1211;
    while(A>=3)
    {
        A-=3;
        n++;
    }
}
```

(13) 如下代码输出：

```
void test_const()
{
 const int a =1;
 int *p = const_cast<int*>(&a);
 *p = 2;
 const bool isequal = p==&a;
 cout<<endl;
 cout<<a<<"\t"<<*p<<"\t"<<isequal<<endl;
}
```

1 2 1

(14) 实现一个超大数据类并实现其乘法操作（WMSJ）

字符串

(15) 双层循环使用效率问题

双层循环，较长的循环放在内层效率要高，从而减少内层循环的频繁构造。

(16) 宏调用，宏在使用时只是将结果进行直接简单的替代。(Intel)

例如 #define calc(a,b) a*b/(a-b)

a=30,b=10;

calc(a+10,b-4) => **130** 30+10*10-4/(30+10-10-4)

(17) strcpy 拷贝的结束标志是查找字符串中的/0 因此如果字符串中没有遇到/0 的话 会一直复制，直到遇到/0.

(18) Union 中低位在低位 高位在高位

(19) 获得 struct 内部某个变量相对 struct 的偏移量

```
struct test
{
    int a;
    int b;
    char c;
    int d;
};
#define FIND(structTest,e) (size_t)&(((structTest*)0)->e)
size_t s = FIND(test,b);
```

(20) 函数指针

int (*fun)(int,int);函数指针 赋值：int (*p)(int,int)=&fun;

void* fun(int,int);返回指针的函数

int *ptr[] 或 int *(ptr[]) 声明了一个指针数组，里面存放的都是地址

int (*ptr)[] 声明了一个数组指针，他指向了一个数组

```
char ss[2][6]={"Honey","well"};
char *s2=(char*)ss;
printf("%s\n",&ss[0][0]+1);//oney
printf("%s\n",ss+1);//well
printf("%s\n",s2+1);//oney
```

(21) 句柄不是指针，一个句柄是指使用的一个唯一的整数值，即一个四字节长的数值，来标志应用程序中的不同对象和同类对象中的不同的实例，诸如，一个窗口，按钮，图标，滚动条，输出设备，控件或者文件等。应用程序能够通过句柄访问相应的对象的信息，但是句柄不是一个指针，程序不能利用句柄来直接阅读文件中的信息。如果句柄不用在 I/O 文件中，它是毫无用处的。 句柄是 windows 用来标志应用程序中建立的或是使用的唯一整数

句柄与普通指针的区别在于，指针包含的是引用对象的内存地址，而句柄则是由系统所管理的引用标识，该标识可以被系统重新定位到一个内存地址上。这种间接访问对象的模式增强了系统对引用对象的控制。

(22) 模板

template <typename T> 模板类：template <class T>

(23) ISR 终端服务子程序

ISR 中不允许有返回值，不允许传递参数，并且要求短而有效率，故尽量不要使用浮点数也不要使用 printf 等语句。

(24) 获得结构体内某元素的偏移量

#define FIND(structTest,e) (size_t)&(((structTest*)0)->e)

(25) 0xfacefeed 的十进制数最后一位是多少？

在十六进制中 A~F 分别代表 10~15，我们仅将个位相加即可，而 16 的各次方除 0 次方外都是 6，从而可以获得结果 0+0+2+4+0+4+4+3=7。注意各位乘 1，而其他位均乘 6.

(26) int a=(int)((unsigned int)0xffffffff+(unsigned int)0xffffffff);

cout<<a<<endl;

输出结果：-2;

(27) 逗号表达式的顺序是从左往右，最终表达式的值为最后一个表达式的值。

15)NEW & MALLOC

(1) new 错误时会抛出 bad_malloc 异常，而 malloc 失败时会返回 0。当然这与具体的编译器有关，而且我们也可以强制 new 不抛出异常，此时 new 也会返回 NULL.

(2) malloc 与 free 是 C++/C 语言的标准库函数，new/delete 是 C++的运算符。它们都可用于申请动态内存和释放内存。

(3) 对于非内部数据类型的对象而言，malloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。

因此 C++语言需要一个能完成动态内存分配和初始化工作的运算符 new，以一个能完成清理与释放内存工作的运算符 delete。注意 new/delete 不是库函数。

(4) new 可以认为是 malloc 加构造函数的执行。new 出来的指针是直接带类型信息的。而 malloc 返回的都是 void 指针，需要进行转换。

16)ENUM

```
enum colors
{
 red, yellow=0,blue
}aa=red ,bb=yellow;
```

对于枚举类型，除了已经定义好的元素外，其余都以 0 开始递增，所以 red=0;blue=1;所以 aa==bb 的结果将为 1.

2. 面向对象编程

1) 构造函数 虚函数 静态成员函数

(1) 因虚函数用于实现动态绑定，而构造函数在调用时是确定的不必动态绑定，故不可声明为动态绑定；

(2) 静态成员函数相当于全局函数，不可在其前加 const 或者 volatile，只能在文件所在的编译单位内使用，不能在其他编译单位内使用。我们把一个函数定义为静态函数就是为了在全局中进行调用进行标记或者其他的作用，如果定义为动态绑定进行覆盖，则与其

本身定义相违背。静态成员函数仅能访问静态的数据成员，不能访问非静态的数据成员，也不能访问非静态的成员函数，这是由于静态的成员函数没有 this 指针。

由于多态的存在析构函数应当声明为虚函数，这保证了不会出现由于析构函数未被调用而导致的内存泄露。

静态构造函数用于初始化任何静态数据，或用于执行仅需执行一次的特定操作。在创建第一个实例或引用任何静态成员之前，将自动调用静态构造函数。静态成员函数不可显式调用，没有参数。

(3) 使用纯虚函数或将构造函数声明为 private 可以阻止一个类被实例化。

(4) 在 C#中静态构造函数，只有在第一次实例化对象时才启动 satic 构造函数，以后再实例化时不再起作用。

静态构造函数是 C#的一个新特性，在编程过程中用处并不广，它的主要目的是用于初始化一些静态的变量。因为这个构造函数是属于类的，而不属于任何一个实例，所以这个构造函数只会被执行一次，而且是在创建此类的第一个实例或引用任何静态成员之前，由.NET 自动调用。

(5) 析构函数可以是私有的；如果将析构函数声明为私有的，可以保证只能在堆上 new 一个新的类对象。原因是 C++是一个静态绑定的语言。在编译过程中，所有的非虚函数调用都必须分析完成。即使是虚函数，也需检查可访问性。因些，当在栈上生成对象时，对象会自动析构，也就说析构函数必须可以访问。而堆上生成对象，由于析构时机由程序员控制，所以不一定需要析构函数。保证了不能在栈上生成对象后，需要证明能在堆上生成它。这里 OnlyHeapClass 与一般对象唯一的区别在于它的析构函数为私有。delete 操作会调用析构函数。所以不能编译。

那么如何释放它呢？答案也很简单，提供一个成员函数，完成 delete 操作。在成员函数中，析构函数是可以访问的。当然 detele 操作也是可以编译通过。

```
void OnlyHeapClass::Destroy() { delete this; }
```

构造函数私有化的类的设计可以保证只能用 new 命令在堆中来生成对象，只能动态的去创建对象，这样可以自由的控制对象的生命周期。但是，这样的类需要提供创建和撤销的公共接口。

另外重载 delete，new 为私有可以达到要求对象创建于栈上的目的，用 placement new 也可以创建在栈上。

2) COPY & ASSIGNMENT

当用于类类型对象时，初始化的复制形式和直接形式有所不同；直接初始化 int a(10); 直接调用与实参匹配的构造函数；

而复制初始化(使用 '=')总是调用复制构造函数 copy constructor。复制初始化首先使用指定构造函数创建一个临时对象，而后用复制构造函数将临时对象复制到正在创建的对象。特别地，当形参或返回值为类类型的值传递时，将调用复制构造函数。

如果不允许复制，应当显式的将复制构造函数声明为 private。

对于赋值操作符 assignment，如果类没有定义自己的赋值操作符，则编译器会自动生成一个；而对赋值操作符的重载与复制构造函数的操作类似。如第八部分第 10 节。

实际上，应当将两者看做一个单元，如果需要其中一个，几乎可以肯定需要另一个。

3) 列表初始化

const 与 reference 成员只能使用成员初始化列表进行初始化，不能进行赋值。

void Point(int x,int y): m_x(x),m_y(y){ };

使用成员初始化列表比使用赋值效率要高。

并且初始化列表初始化的顺序是根据成员变量的声明顺序来执行的。

对静态成员变量需要对其赋初值。

4) 多态

多态：是将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。多态可以简单的概括为“一种接口 多种实现”，在程序运行的过程中才决定调用的函数，即动态绑定 dynamic binding。

多态的作用主要是两个：1. 隐藏实现细节，使得代码能够模块化；扩展代码模块，实现代码重用；2. 接口重用：为了类在继承和派生的时候，保证使用家族中任一类的实例的某一属性时的正确调用。

```
class animal
{
    ...
    virtual void Move() //注意这是虚函数
    {
        cout<<"animal move"<<endl;
    }
};
class fish:public animal
{
    ...
    void Move()
    {
        cout<<"fish can swim"<<endl;
    }
};
class cat:public animal
{
...
    void Move()
    {
    cout<<"cat can run"<<endl;
    }
};
void main()
{
```

```
        animal* pAn;
        switch(用户选择)
        {
        case fish:
              pAn = new fish;
              break;
        case cat:
              pAn = new cat;
              break;
        ....
        }
        pAn->Move();
        delete pAn;
        }
}
```

5) 静态绑定与动态绑定

多态的精髓在于动态绑定，与之相对的是静态绑定。后者绑定的是对象的静态类型，某特性（比如函数）依赖于对象的静态类型，发生在编译期。后者绑定的是对象的动态类型，某特性（比如函数）依赖于对象的动态类型，发生在运行期。

```
1.  class B
2.  {
3.      void DoSomething();
4.      virtual void vfun();
5.  }
6.  class C : public B
7.  {
8.      void DoSomething();//首先说明一下，这个子类重新定义了父类的 no-virtual 函数，
    这是一个不好的设计，会导致名称遮掩；这里只是为了说明动态绑定和静态绑定才这样使用。
9.      virtual void vfun();
10. }
11. class D : public B
12. {
13.     void DoSomething();
14.     virtual void vfun();
15. }
16. D* pD = new D();
17. B* pB = pD;
```

pD->DoSomething()和 pB->DoSomething()调用的是同一个函数吗？

不是的，虽然 pD 和 pB 都指向同一个对象。因为函数 DoSomething 是一个 no-virtual 函数，它是静态绑定的，也就是编译器会在编译期根据对象的静态类型来选择函数。pD 的

静态类型是 D*，那么编译器在处理 pD->DoSomething()的时候会将它指向 D::DoSomething()。同理，pB 的静态类型是 B*，那 pB->DoSomething()调用的就是 B::DoSomething()。

pD->vfun()和 pB->vfun()调用的是同一个函数吗？

是的。因为 vfun 是一个虚函数，它动态绑定的，也就是说它绑定的是对象的动态类型，pB 和 pD 虽然静态类型不同，但是他们同时指向一个对象，他们的动态类型是相同的，都是 D*，所以，他们的调用的是同一个函数：D::vfun()。

另外当缺省参数和虚函数一起出现的时候情况有点复杂，极易出错。虚函数是动态绑定的，但是为了执行效率，缺省参数是静态绑定的。

6) EXPLICIT MUTABLE VOLATILE INTERNAL

(6) 在 C++中，类数据成员加上 mutable 后，修饰为 const 的成员函数就可以修改它了。

(7) 单个参数的构造函数如果不添加 explicit 关键字，会定义一个隐含的类型转换；

添加 explicit 关键字会消除这种变化。例如：

```
class String {
 String ( int n ); //本意是预先分配 n 个字节给字符串
 String ( const char* p ); //  用 C 风格的字符串 p 作为初始化值
 //...
 }
```

下面两种写法比较正常：

```
String s2 ( 10 );     //OK  分配 10 个字节的空字符串
String s3 = String ( 10 ); //OK  分配 10 个字节的空字符串
String s4 = 10; //编译通过，也是分配 10 个字节的空字符串
String s5 =  'a' ; //编译通过，分配 int（'a'）个字节的空字符串  均可编
```

译通过，但是如果添加 explicit 后，后两者将不能够使用。

(8) volatile 易变的，用以修饰被不同线程访问和修改的变量。意为“在编译器认识的范围内，这个数据可以被改变”。如果不加 volatile，要么无法编写多线程程序，要么编译器将失去大量优化机会。一般并行设备的寄存器、终端服务子程序中会访问到的非自动变量及多线程中被几个任务共享的变量都会应用到 volatile。甚至可以声明 const volatile 对象，例如只读的状态寄存器，此对象不能被程序员改变，但可以通过外面的工具改变。

(9) internal 关键字。Internal 为 C#中的关键字，是类型和类型的成员访问修饰符。只有在同一程序集或同一命名空间内的文件中，内部类型或成员才是可访问是一种访问控制权限。

7) 继承

无论公有保护还是私有继承，对于继承类内部来讲，都可以访问父类的公有和受保护成员，但是不可以访问私有成员。

而对继承类成员来讲，在公有继承情况下，可以访问父类的公有和受保护成员，但是不能修改受保护成员；在受保护继承条件下，可以访问父类的公有和受保护成员，但是所有成员都不可以修改；在私有继承下，所有成员均不可以访问和修改。

8) 堆栈溢出

```
void main()
{
    char str;
    char *s=&str;
    strcpy(s,"hello");
    printf(s);
}
```

上述程序会出现什么问题？

Run-Time Check Failure #2 - Stack around the variable 'str' was corrupted.

堆栈如前所述,是一段连续的空间,向低地址生长,ESP 和 EBP 为栈顶指针，分别指向栈顶及当前活动记录的顶部，上述程序运行时首先向堆栈中压入 函数返回地址 ret 及 EBP，而后因为将在 s 地址中存入了”hello”,所以 ESP-6 向上生长(低地址)六个字节；但是由于我们只申请了一个 char 的空间，容纳不下，只好向内存顶部继续写。从而覆盖掉了 EBP 及 ret。因此 main 函数返回时返回地址被篡改成了”ello”,从而导致错误。此即一次堆栈溢出。

详见：http://blog.csdn.net/cnctloveyu/article/details/4236212

由于字符串处理函数（gets，strcpy 等等）没有对数组越界加以监视和限制，我们利用字符数组写越界，覆盖堆栈中的老元素的值，就可以修改返回地址。

9) 重载操作符

大部分运算符可以重载，但是有几个运算符是不能够重载的，包括域操作符**’::’**，属性操作符**’.’**，**’.*’**以及条件选择运算符**’?:’**等。

不能通过连接现有合法操作符创建新的操作符，也不能为已有内置类型定义额外的新的运算符。

10) FINAL

在 C++11 中, final specifier Specifies that a virtual function can not be overridden in a derived class or that a class cannot be inherited.

见：http://msdn.microsoft.com/zh-cn/library/47ezsw2s(v=vs.90).aspx

final 修饰符用于指定类不能扩展或者方法或属性不能重写。它将防止其他类通过重写重要的函数来更改该类的行为。带有 final 修饰符的方法可以由派生类中的方法来隐藏或重载。类中的方法和属性以及类可以使用 final 修饰符来标记。接口、字段和接口的成员不能采用 final 修饰符。（SAP2014 校招）

不能将 final 修饰符与其他继承修饰符 (abstract) 组合。默认情况下，类成员既不是 abstract 也不是 final。继承修饰符不能与 static 修饰符组合。

override 修饰符用于重写基类中方法的方法。当基类不包含具有相同签名的成员时，不得将 override 修饰符用于方法。

类中的方法和属性可以使用 override 修饰符来标记。类、字段、接口和接口的成员不能采用 override 修饰符。

abstract 修饰符用于类中不具有实现的方法或属性或者用于包含这些方法的类。具有抽象成员的类不能使用 new 运算符来实例化。您可以从抽象基类派生抽象和非抽象的类。

类中的方法和属性以及类可以使用 abstract 修饰符来标记。如果一个类包含任何 abstract 成员，则必须标记为 abstract。接口和接口的成员为隐式抽象，它们不能采用 abstract 修饰符。字段不能为 abstract。

不能将 override 修饰符与其他版本安全修饰符 (hide) 组合。版本安全修饰符不能与 static 修饰符组合。默认情况下，除非基类方法具有 final 修饰符，否则方法将重写基类方法。不能重写 final 方法。

hide 修饰符用于隐藏基类中方法的方法。当基类不包含具有相同签名的成员时，不得将 hide 修饰符用于方法。

类中的方法和属性可以使用 hide 修饰符来标记。类、字段、接口和接口的成员不能采用 hide 修饰符。

11) C#

C#与 C++相比，有托管机制，不允许进行内存的操作，而是由固定的垃圾回收机制来完成；其次 C#与 Java 类似是运行在虚拟机上，前者运行在 .NET 后者运行在 Java 虚拟机上；最后 C#是完全面向对象的，所有都是类，不存在全局变量等，也没有多重继承等不易掌握的特点。

C#中声明一个数组时，方括号必须在类型的后面，例如 int [] mf1=new int[6];
String [] mf2={“C”,”C++”,”C#”};

3. 设计模式

1) UTF 编码协议

UTF-8
UTF-16

2) 创建型模式（CREATIONAL PATTERN）

在软件工程中，创建型模式是处理对象创建的设计模式，试图根据实际情况使用合适的方式创建对象。基本的对象创建方式可能会导致设计上的问题，或增加设计的复杂度。创建型模式通过以某种方式控制对象的创建来解决问题。

创建型模式由两个主导思想构成。一是将系统使用的具体类封装起来，二是隐藏这些具体类的实例创建和结合的方式。包含如下 pattern：

抽象工厂 Abstract Factory 模式，提供一个创建相关或依赖对象的接口，而不指定对象的具体类。

工厂方法 Factory Method 模式，允许一个类的实例化推迟到子类中进行。

生成器 Builder 模式，将一个复杂对象的创建与它的表示分离，使同样的创建过程可以创建不同的表示。

延迟初始化模式，将对象的创建，某个值的计算，或者其他代价较高的过程推迟到它第一次

需要时进行。

对象池模式，通过回收不再使用的对象，避免创建和销毁对象时代价高昂的获取和释放资源的过程。

原型 Prototype 模式，使用原型实例指定要创建的对象类型，通过复制原型创建新的对象。

单例 singleton 模式，保证一个类只有一个实例，并且提供对这个实例的全局访问方式。

3) 单例模式

单例模式拥有一个私有构造函数，确保用户无法无法通过 new 直接实例化它。

```
class CSingleton
{
//其他成员
public:
    static CSingleton* GetInstance()
    {
        if ( m_pInstance == NULL )  //判断是否第一次调用
        m_pInstance = new CSingleton();
        return m_pInstance;
    }
private:
    CSingleton(){};
    static CSingleton * m_pInstance;
    };
```

用户访问唯一实例的方法只有 GetInstance()成员函数。如果不通过这个函数，任何创建实例的尝试都将失败，因为类的构造函数是私有的。GetInstance()使用懒惰初始化，也就是说它的返回值是当这个函数首次被访问时被创建的。

4) 策略模式

创新工场 2014 校招. 在一个不断需要算法转换的情况下，适合哪种设计模式？

策略模式：定义一系列的算法,把每一个算法封装起来, 并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。也称为政策模式(Policy)。(Definea family of algorithms,encapsulate each one, andmake them interchangeable. Strategy lets the algorithmvary independently from clients that use it.)

当存在以下情况时使用 Strategy 模式

(1) 许多相关的类仅仅是行为有异。 “策略”提供了一种用多个行为中的一个行为来配置一个类的方法。即一个系统需要动态地在几种算法中选择一种。

(2) 需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间 /时间权衡的算法。当这些变体实现为一个算法的类层次时 ,可以使用策略模式。

(3) 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。

(4) 一个类定义了多种行为 , 并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的 Strategy 类中以代替这些条件语句。

各模式详见：http://blog.csdn.net/hguisu/article/details/7558249

5) MVC

MVC（Model View Controller）是一个框架模式，它强制性的使应用程序的输入、处理和输出分开。使用 MVC 应用程序被分成三个核心部件：模型、视图、控制器。它们各自处理自己的任务。模型（Model） “数据模型”（Model）用于封装与应用程序的业务逻辑相关的数据以及对数据的处理方法。MFC/ASP .NET 等都是 MVC 架构的实现。

“模型”有对数据直接访问的权力，例如对数据库的访问。“模型”不依赖“视图”和“控制器”，也就是说，模型不关心它会被如何显示或是如何被操作。但是模型中数据的变化一般会通过一种刷新机制被公布。为了实现这种机制，那些用于监视此模型的视图必须事先在此模型上注册，从而，视图可以了解在数据模型上发生的改变。

视图（View） 视图层能够实现数据有目的的显示（理论上，这不是必需的）。在视图中一般没有程序上的逻辑。为了实现视图上的刷新功能，视图需要访问它监视的数据模型（Model），因此应该事先在被它监视的数据那里注册。

控制器（Controller） 控制器起到不同层面间的组织作用，用于控制应用程序的流程。它处理事件并作出响应。“事件”包括用户的行为和数据模型上的改变。

The model consists of application data, business rules, logic, and functions. A view can be any output representation of data, such as a chart or a diagram. Multiple views of the same data are possible, such as a bar chart for management and a tabular view for accountants. The controller mediates input, converting it to commands for the model or view.

6) PIMPL

Private Implementation / pointer to implementation

直接的字面意思就是“实现私有化”，也如我们常常听到诸如“不要改动你的公有接口”这样的建议，Pimpl 机制，顾名思义，将实现私有化，力图使得头文件对改变不透明。主要作用是解开类的使用接口和实现的耦合。这种说法语义上更关注代码的实现方法，也就是一个指向实现的指针。如果不使用 pImpl 惯用手法，代码会像这样：

```
//c.hpp

    #include<x.hpp>
    class C
        {
         public:
                void f1();
         private:
                X x; //与 X 的强耦合
               };
```

像上面这样的代码，C 与它的实现就是强耦合的，从语义上说，x 成员数据是属于 C 的实现部分，不应该暴露给用户。从语言的本质上来说，在用户的代码中，每一次使用”new C”和”C c1”这样的语句，都会将 X 的大小硬编码到编译后的二进制代码段中（如果 X 有虚函数，则还不止这些）——这是因为，对于”new C”这样的语句，

其实相当于 operator new(sizeof(C))后面再跟上 C 的构造函数，而”C c1”则是在当前栈上腾出 sizeof(C)大小的空间，然后调用 C 的构造函数。因此，每次 X 类作了改动，使用 c.hpp 的源文件都必须重新编译一次，因为 X 的大小可能改变了。

在一个大型的项目中，这种耦合可能会对 build 时间产生相当大的影响。

pImpl 惯用手法可以将这种耦合消除，使用 pImpl 惯用手法的代码像这样：

```
//c.hpp

        class X;    //用前导声明取代 include
        class C
        {
         ...
         private:
            X* pImpl; //声明一个 X*的时候，class X 不用完全定义
        };
```

在一个既定平台上，任何指针的大小都是相同的。之所以分为 X*，Y*这些各种各样的指针，主要是提供一个高层的抽象语义，即该指针到底指向的是那个类的对象，并且，也给编译器一个指示，从而能够正确的对用户进行的操作（如调用 X 的成员函数）决议并检查。但是，如果从运行期的角度来说，每种指针都只不过是个 32 位的长整型（如果在 64 位机器上则是 64 位，根据当前硬件而定）。

正由于 pImpl 是个指针，所以这里 X 的二进制信息（sizeof(C)等）不会被耦合到 C 的使用接口上去，也就是说，当用户”new C”或”C c1”的时候，编译器生成的代码中不会掺杂 X 的任何信息，并且当用户使用 C 的时候，使用的是 C 的接口，也与 X 无关，从而 X 被这个指针彻底的与用户隔绝开来。只有 C 知道并能够操作 pImpl 成员指向的 X 对象。

7) RAII

RAII（Resource Acquisition Is Initialization）,也成为“资源获取就是初始化”，是 C++语言的一种管理资源、避免泄漏的惯用法。C++标准保证任何情况下，已构造的对象最终会销毁，即它的析构函数最终会被调用。

简单的说，RAII 的做法是使用一个对象，在其构造时获取资源，在对象生命期控制对资源的访问使之始终保持有效，最后在对象析构的时候释放资源。

根据 RAII 对资源的所有权可分为常性类型和变性类型，代表者分别是 boost：shared_ptr<>和 std：：auto_ptr<>；从所管资源的初始化位置上可分为外部初始化类型和内部初始化类型。

4. STL

1) VECTOR

(1) 以下代码有什么问题？

typedef vector IntArray;

```
IntArray array;
array.push_back( 1 );
array.push_back( 2 );
array.push_back( 2 );
array.push_back( 3 );// 删除 array 数组中所有的 2
for( IntArray::iterator itor=array.begin(); itor!=array.end(); ++itor )
{
    if( 2 == *itor ) array.erase( itor );
}
```

答：同样有缺少类型参数的问题。另外，每次调用“array.erase(itor);”，被删除元素之后的内容会自动往前移，导致迭代漏项，应在删除一项后使 **itor--**， 使之从已经前移的下一个元素起继续遍历。

2) UPPER_BOUND&LOWER_BOUND

iterator lower_bound(const key_type &key): 返回一个迭代器，指向键值>= key 的第一个元素。

iterator upper_bound(const key_type &key):返回一个迭代器，指向键值>key 的第一个元素。

3) MAP

形成映射关系，比如将 string 与 int 映射出来。

```
map<string,int> mm;
int tot=0;
int findindex(const string ss)
{
  if (mm.count(ss))
  {
      return mm[ss];
  }
  mm[ss]=tot++;
  return tot;
}
```

三．数据结构

1. 树

1) 基本知识

(1) 因为树本身的特点，使得对于树的大部分操作均可以通过递归的方式操作，而难点却在于相应的非递归方式的写法。在面试过程中也很可能会同时被问到递归与非递归方式的问题。在非递归式中一般会用到栈或队列，从而对各节点的访问顺序进行控制。

(2) 根据一棵树的先序/中序遍历，或者后序/中序遍历序列，都可以唯一地确定一棵树。具体算法如下：

首先根据前序遍历或后序遍历的第一个或最后一个为根节点

根据根节点可以在中序遍历中分出左子树部分与右子树部分

根据前序/后序遍历的其他节点及中序遍历的左右两部分重复上述两步骤即可。

如果树中只存在度为 0 和度为 2 的节点，则根据它的前序遍历和后序遍历序列，可以重构树的结构。否则不能唯一重构树。代码见 2).(6)

(3) 树的深度是从根节点开始算的，**根节点为第 1 层**，最大层次称为深度。

(4) 任何一棵和树对应的二叉树，其右子树必定为空。当森林转换为二叉树时，其第一棵树的子树森林转换成左子树，剩余树的森林转换为右子树。

2) 几个问题

(1) 获取树的深度与宽度

递归的方式：树的深度等于左右子树高层次+1。

```
int treeDepth(BSTreeNode *p)
{
    if (!p)
    {
        return 0;
    }
    int left = treeDepth(p->m_pLeft);
    int right = treeDepth(p->m_pRight);
    return left>right?left+1:right+1;
}
```

http://blog.csdn.net/linraise/article/details/9878859

http://blog.csdn.net/walkinginthewind/article/details/7518888

非递归的方式:利用栈逐层进行访问

```
int Depth(BSTreeNode *T)
{
    BSTreeNode * store[50];
```

```
    memset(store,0,50);
    int length=0;
    int depth=0;
    stack<BSTreeNode *> st;
    if (T)
    {
        st.push(T);
        while (!st.empty())
        {
            depth++;
            length=0;
            while (!st.empty())
            {
                BSTreeNode * p=st.top();
                st.pop();
                if (p->m_pLeft)
                    store[length++]=p->m_pLeft;
                if (p->m_pRight)
                    store[length++]=p->m_pRight;
            }
            for (int i=0;i<length;i++)
                st.push(store[i]);
        }
        return depth;
    }
    else
        return 0;
}
```

(2) 将树转变为双向链表

要求只改变指针，不创建新的结点。在中序遍历中按输出的结点序逐个更改其指针指向，使其转变为双向链表。

```
BSTreeNode *pHead=NULL;//指向循环队列头结点
BSTreeNode *pIndex=NULL;//指向前一个结点
void inOrderBSTree(BSTreeNode* pBSTree)
 {
    if (NULL==pBSTree)
        return;
    inOrderBSTree(pBSTree->m_pLeft);
    convertToDoubleList(pBSTree);
    inOrderBSTree(pBSTree->m_pRight);
}
```

```
void convertToDoubleList(BSTreeNode* pCurrent)
{
    pCurrent->m_pLeft=pIndex;//使当前结点的左指针指向双向链表中最后一个结点
    if (NULL==pIndex)//若最后一个元素不存在，此时双向链表尚未建立，因此将当前结点设为双向链表头结点
        pHead=pCurrent;
    else//使双向链表中最后一个结点的右指针指向当前结点
        pIndex->m_pRight=pCurrent;
    pIndex=pCurrent;//将当前结点设为双向链表中最后一个结点
}
```

(3) 判断一颗树是否为二叉排序树

```
bool isBST(BSTreeNode *p)
{
    if (p==NULL)
    {
        return true;
    }
    if (p->m_pLeft && p->m_pRight)
    {
        if (p->m_nValue<p->m_pRight->m_nValue &&
p->m_nValue>p->m_pLeft->m_nValue)
        {
            return isBST(p->m_pLeft)&&isBST(p->m_pRight);
        }
        else
        {
            return false;
        }
    }
    else if(p->m_pLeft&&!p->m_pRight)
    {
        if (p->m_nValue>p->m_pLeft->m_nValue)
        {
            return isBST(p->m_pLeft);
        }
        else
            return false;
    }
    else if (p->m_pRight&&!p->m_pLeft)
    {
        if (p->m_nValue<p->m_pRight->m_nValue)
        {
            return isBST(p->m_pRight);
        }
        else
            return false;
    }
    else
        return true;
}
```

(4) 树的销毁

树的销毁应从叶子节点开始逐个向上销毁。如采用非递归的方法，可以使用后序遍历逐个销毁结点，因后序遍历是先叶子结点后根节点的一种方法。

```
void destroyTree(treeNode *root)
{
    if (!root)
    {
        destroyTree(root->left);
        destroyTree(root->right);
        free(root);
    }
}
```

(5) 前序中序->后序或重构树

例如前序遍历为 DBACEGF 中序遍历为 ABCDEFG 输出后序遍历 ACBEFGD(暴风 2014)
由后序/前序和中序遍历，重构出原树。

```
1.  void buildTree(char* pre,char* mid,tree* &root){
2.      int len=strlen(pre);        //实际下先序序列和后序序列的长度是一样的
3.      //序列长度为 0,说明上一级已经是叶子节点,返回
4.      if(len==0)
5.          return;
6.      //返回先序的第一个元素在中序中出现的位置
7.      char* p=strchr(mid,pre[0]);
8.      int pos=(int)(p-mid);
9.      //建设子树的根节点
10.     //先序中第一个元素作为本子树的根节点
11.     root->data=pre[0];
12.     if(pos!=0){         //当前节点的左子树是存在的
13.         tree* left=(tree*)malloc(sizeof(tree));
14.         root->left=left;
15.         //左子树根节点上的元素就是先序中的第 2 个元素
16.         left->data=pre[1];
17.         char* left_pre=(char*)malloc((pos+1)*sizeof(char));
18.         char* left_mid=(char*)malloc((pos+1)*sizeof(char));
19.         //找到左子树的先序和中序
20.         strncpy(left_pre,pre+1,pos);
21.         left_pre[pos]='\0';
22.         strncpy(left_mid,mid,pos);
23.         left_mid[pos]='\0';
24.         //递归建立左子树
25.         buildTree(left_pre,left_mid,left);
26.     }
27.     else
28.         root->left=NULL;
29.
30.     if(pos!=len-1){     //当前节点的右子树是存在的
31.         tree* right=(tree*)malloc(sizeof(tree));
32.         root->right=right;
```

```
        //右子树根节点上的元素就是先序中的第 pos+2 个元素
        right->data=pre[pos+1];
        char* right_pre=(char*)malloc((len-pos)*sizeof(char));
        char* right_mid=(char*)malloc((len-pos)*sizeof(char));
        //找到右子树的先序和中序
        strncpy(right_pre,pre+pos+1,len-1-pos);
        right_pre[len-pos-1]='\0';
        strncpy(right_mid,mid+pos+1,len-1-pos);
        right_mid[len-pos-1]='\0';
        //递归建立右子树
        buildTree(right_pre,right_mid,right);
    }
    else
        root->right=NULL;
}
```

由后序/前序和中序遍历，推知后序/前序遍历

```
void postfrompremid(char *pre,char *mid,char*post)
{
    int length=strlen(pre);
    if (length==0)
    {
        return;
    }
    post[length-1]=pre[0];
    char *midnode=strchr(mid,pre[0]);
    if (!midnode)
    {
        cout<<"wrong input";
        return;
    }
    int midpos=(int)(midnode-mid);
    if (midpos!=0)//有左子树
    {
        char *leftpre=new char[midpos+1];
        char *leftmid=new char[midpos+1];
        strncpy(leftpre,pre+1,midpos);
        leftpre[midpos]='\0';
        strncpy(leftmid,mid,midpos);
        leftmid[midpos]='\0';
        postfrompremid(leftpre,leftmid,post);
        delete [] leftpre;
        delete [] leftmid;
    }
```

```
28.
29.     if (midpos!=length-1)//有右子树
30.     {
31.         char *rightpre = new char[length-midpos];
32.         char *rightmid = new char[length-midpos];
33.         strncpy(rightpre,pre+midpos+1,length-midpos-1);
34.         strncpy(rightmid,mid+midpos+1,length-midpos-1);
35.         rightpre[length-midpos-1]='\0';
36.         rightmid[length-midpos-1]='\0';
37.         postfrompremid(rightpre,rightmid,post+midpos);
38.         delete [] rightpre;
39.         delete [] rightmid;
40.     }
41. }
```

已知前序中序推知后序的程序可以依照上述程序依次写出，其中 strchr 用以查找字符出现位置，strncpy 用以复制指定数目的字符串。

(6) 树中相距最远的距离。

GD2014

```
int longest_dis(Node* root)
{
    int height1, height2;
    if( root==NULL)
        return 0;
    if( root->left == NULL ) && ( root->right == NULL )
        return 0;
    height1 = height(root->left);
    // height(Node* node) returns the height of a tree rooted at node
    height2 = height(root->right);
    if( root->left != NULL ) && ( root->right == NULL )
        return max(height1+1, longest_dis(root->left) );
    if( root->left == NULL ) && ( root->right != NULL )
        return max(height2+1, longest_dis(root->right) );
    return max(height1+height2+2, longest_dis(root->left), longestdis(root->right) );
}
```

(7) 树的镜像。

DQ2014，MT2014

(8) 树的”面积”

BD2014 输入一颗二叉树，输出其宽度×高度
高度即(1)中所述 height。宽度即(1)中所述的 length 取最大值输出即可。

(9) 树两个节点的高度差

HW2014.
树的高度定义为某节点其下树的层数。如 A 下面有两层，则 A 的高度为 3。

(10) 二叉树遍历

```
void preOrder1(BinTree *root)      //递归前序遍历
```

```
{
    if(root!=NULL)
    {
        cout<<root->data<<" ";
        preOrder1(root->lchild);
        preOrder1(root->rchild);
    }
}

void inOrder1(BinTree *root)      //递归中序遍历
{
    if(root!=NULL)
    {
        inOrder1(root->lchild);
        cout<<root->data<<" ";
        inOrder1(root->rchild);
    }
}

void postOrder1(BinTree *root)    //递归后序遍历
{
    if(root!=NULL)
    {
        postOrder1(root->lchild);
        postOrder1(root->rchild);
        cout<<root->data<<" ";
    }
}

void preOrder2(BinTree *root)     //非递归前序遍历
{
    stack<BinTree*> s;
    BinTree *p=root;
    while(p!=NULL||!s.empty())
    {
        while(p!=NULL)
        {
            cout<<p->data<<" ";
            s.push(p);
            p=p->lchild;
        }
        if(!s.empty())
        {
            p=s.top();
            s.pop();
            p=p->rchild;
        }
    }
}

void inOrder2(BinTree *root)      //非递归中序遍历
{
    stack<BinTree*> s;
    BinTree *p=root;
    while(p!=NULL||!s.empty())
    {
        while(p!=NULL)
        {
            s.push(p);
```

```
            p=p->lchild;
        }
        if(!s.empty())
        {
            p=s.top();
            cout<<p->data<<" ";
            s.pop();
            p=p->rchild;
        }
    }
}
```

后序遍历的非递归实现是三种遍历方式中最难的一种。

因为在后序遍历中，要保证左孩子和右孩子都已被访问并且左孩子在右孩子前访问才能访问根结点，这就为流程的控制带来了难题。下面介绍两种思路。

第一种思路：对于任一结点 P，将其入栈，然后沿其左子树一直往下搜索，直到搜索到没有左孩子的结点，此时该结点出现在栈顶，但是此时不能将其出栈并访问，因此其右孩子还为被访问。所以接下来按照相同的规则对其右子树进行相同的处理，当访问完其右孩子时，该结点又出现在栈顶，此时可以将其出栈并访问。这样就保证了正确的访问顺序。可以看出，在这个过程中，每个结点都两次出现在栈顶，只有在第二次出现在栈顶时，才能访问它。因此需要多设置一个变量标识该结点是否是第一次出现在栈顶。

```
void postOrder2(BinTree *root)    //非递归后序遍历
{
    stack<BTNode*> s;
    BinTree *p=root;
    BTNode *temp;
    while(p!=NULL||!s.empty())
    {
        while(p!=NULL)  //沿左子树一直往下搜索，直至出现没有左子树的结点
        {
            BTNode *btn=(BTNode *)malloc(sizeof(BTNode));
            btn->btnode=p;
            btn->isFirst=true;
            s.push(btn);
            p=p->lchild;
        }
        if(!s.empty())
        {
            temp=s.top();
            s.pop();
            if(temp->isFirst==true)     //表示是第一次出现在栈顶
            {
                temp->isFirst=false;
                s.push(temp);
                p=temp->btnode->rchild;
            }
            else                        //第二次出现在栈顶
            {
                cout<<temp->btnode->data<<" ";
                p=NULL;
            }
        }
    }
}
```

第二种思路：要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点 P，先将其入栈。如果 P 不存在左孩子和右孩子，则可以直接访问它；或者 P 存在左孩子或者右孩子，但是其左孩子和右孩子都已被访问过了，则同样可以直接访问

该结点。若非上述两种情况，则将 P 的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子前面被访问，左孩子和右孩子都在根结点前面被访问。

```
void postOrder3(BinTree *root)     //非递归后序遍历
{
    stack<BinTree*> s;
    BinTree *cur;                       //当前结点
    BinTree *pre=NULL;                  //前一次访问的结点
    s.push(root);
    while(!s.empty())
    {
        cur=s.top();
        if((cur->lchild==NULL&&cur->rchild==NULL)||
            (pre!=NULL&&(pre==cur->lchild||pre==cur->rchild)))
        {
            cout<<cur->data<<" "; //如果当前结点无孩子或者孩子都已访问过
            s.pop();
            pre=cur;
        }
        else
        {
            if(cur->rchild!=NULL)
                s.push(cur->rchild);
            if(cur->lchild!=NULL)
                s.push(cur->lchild);
        }
    }
}
```

(11) 求两个节点的最近公共祖先

可用递归的方式求解.依次检查各节点,如果两个节点位于当前节点的左右两侧,则当前节点为其最近的公共祖先;如果仅在一侧,则在该侧继续查找直到发现两个结点位于祖先两侧即可.

```
TreeNode* findcommonancestor(TreeNode *root, TreeNode *p, TreeNode *q)
    {
    if(root==NULL||p==NULL||q==NULL)
    return NULL;
    if(contain(root->left,p)&&contain(root->left,q))
    return findcommonancestor(root->left,p,q);
    if(contain(root->right,p)&&contain(root->right,q))
    return findcommonancestor(root->right,p,q);
    return root;
    }

    bool contain(TreeNode*root, TreeNode*p)
    {
      if(root==NULL)
       return false;
      if(root==p)
       return true;
      else
       return contain(root->left,p)||contain(root->right,p);
    }
```

3) 完全二叉树(COMPLETE BINARY TREE)

若设二叉树的深度为 h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在**最左边**，这就是完全二叉树。 完全二叉树是效率很高的数据结构，堆是一种完全二叉树，所以效率极高，像十分常用的排序算法、Dijkstra 算法、Prim 算法等都要用堆才能优化，几乎每次都要考到的二叉排序树的效率也要借助平衡性来提高，而平衡性基于完全二叉树。

题目：对于一个有 800 个结点的完全二叉树来说，他有多少个叶子结点？

一个 K 层满二叉树共有 2^K-1 个结点，故对完全二叉树来讲要求：2^K-1>=800;2^(k-1)-1<=800. K=10.故第 K 层有 800-511=289 个叶子结点，而在上一层中右边仍有若干叶子结点 256 -（289/2+1）= 111。故共有 289+111 = 400 个结点。

具有 n 个结点的完全二叉树的深度为[**log2n**]，对于结点 i 其左孩子为 2i，右孩子为 2i+1。

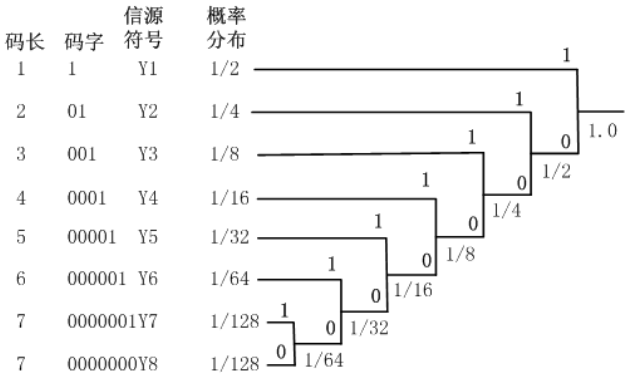
4) 次优查找树

在各个查找单元等概率的情况下，利用判定树可以获得二分查找的较好结果。但是当查找概率不等时，需要构建带权值的最优二叉树。而构建最优二叉树(即霍夫曼树)的代价过大，我们可以构建次优二叉树。

首先按照关键字排序，选择权值和靠近中间的作为根节点，两边的分别作为左子树和右子树，同理，将两边继续分直到形成叶子节点。由于在构建此树时，将某点左右两边的权值和进行比较，未考虑根节点的权值，有可能导致被选为根的关键字的权值比与它相邻的关键字的权值小，此时可做适当调整，选取临近权值较大的关键字作为次优二叉树的结点。

5) 最优二叉树 霍夫曼树

霍夫曼树又称最优二叉树，是一种带权值路径长度最小的树。信源编码中的霍夫曼编码即由此得出。



霍夫曼树的构建方法如下：

1. 把 n 个终端节点加入第一个伫列(依照权重大小排列，最小在前端)
2. 如果伫列内的节点数>1，则:

⑴ 从伫列前端移除两个最低权重的节点

⑵ 将(1)中移除的两个节点权重相加合成一个新节点

⑶ 加入第二个伫列

3. 最后在第一个伫列的节点为根节点

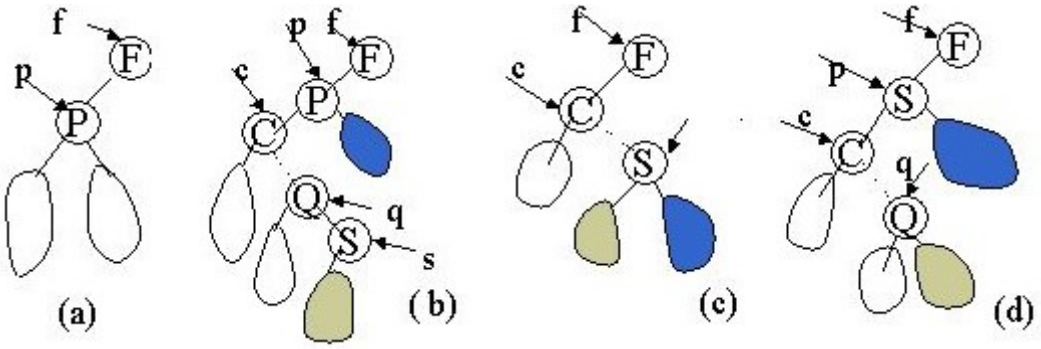
6) BST:SEARCH/INSERT/DELETE

二叉排序树 binary sort tree 或称二叉搜寻树是一种动态树表，中序遍历即可获得一个关键字的有序序列。一个无序序列可以通过构造一棵二叉排序树而变成有序系列。通过对其限制或扩展可以进一步构造 AVL 树及 B 树。

(1) 对二叉排序树的插入，都是在二叉树上新建新的叶子结点。对于不同的插入顺序，形成的二叉排序树是不同的。

(2) 利用二叉排序树查找，最好情况下与在折半查找中所形成的判定树查找效果相同，而最坏情况下为(n+1)/2，为顺序查找相同。期望 O(logn),最坏 O(n)；即可引入平衡树的概念。

(3) 而对其删除操作，若删除为叶子结点，则可将其直接删除，修改器双亲结点指针即可；若删除的结点仅有左子树或右子树，则直接另其双亲结点指向其左子树或右子树即可；若删除的结点左右子树均为非空，则首先找到删除结点 P 左子树 PL 的右子树中的最后一个点 S(PL 中的最大值)，可删除 P 后另 PL 代替 P，而 PR 连到 S 的右子树中图 c 或者将 S 代替 P，并将 SL 连到原来 S 的位置图 d。



删除：

```
treeNode* deleteNode(treeNode *root, int e)
{
 treeNode *p;
 treeNode *foundnode=root;
 enum direction{left,right};
 direction dir=left;
 treeNode *par=root;
 while(foundnode!=NULL&&foundnode->data!=e)
 {
        if (foundnode->data<e){
                par=foundnode;
                dir=right;
                foundnode = foundnode->right;}
        else{
                dir=left;
                par=foundnode;
```

```
                foundnode=foundnode->left;}
        }
        if (foundnode->data!=e) {
            cout<<"not found!";
            return root;}
        if (foundnode->left==NULL&&foundnode->right==NULL){
            p=foundnode;
            if (dir==left){par->left=NULL;}
            else{par->right=NULL;}
            free(p);}
        else if (foundnode->left==NULL&&foundnode->right!=NULL){
            p=foundnode;
            foundnode=foundnode->right;
            free(p);}
        else if (foundnode->right==NULL&&foundnode->left!=NULL){
            p=foundnode;
            foundnode=foundnode->left;
            free(p);}
        else {
            p=foundnode;
            treeNode *leftright=foundnode->left;
            while (leftright->right!=NULL)//获得左子树的最大值
                leftright=leftright->right;
            leftright->right=foundnode->right;
            foundnode=foundnode->left;
            free(p);}
        return root;
        }
```

7) 平衡二叉树与 AVL 树

由于二叉树的查找效率与树高有关系，所以引入平衡二叉树或称 AVL 树。

平衡二叉树具有如下的性质：在平衡二叉树中，对每一个结点而言，其左子树与右子树的深度之差绝对值都小于 1。

可以将二叉排序树构建成为平衡树，只是构建及删除过程中均需要调整两侧树高，以满足平衡条件。在平衡二叉树的表示中，需要添加一个新的表示值 平衡因子，表明当前点的左右深度之差，如左子树插入一个结点后，该值变增加 1，如果大于 1 时便需要进行相应的调整(对子树进行旋转)。在平衡树上查找的时间复杂度为 O(logn)。

AVL 树以一种自平衡的二叉查找树。

8) B 树与 B+树

见：http://www.cnblogs.com/oldhorse/archive/2009/11/16/1604009.html

B 树概括起来说是一个节点可以拥有多于 2 个子节点的多路平衡查找树。其存储排序数据并允许以 O(log n)的运行时间进行查找，顺序读取，插入和删除的数据结构。B-tree 算法减少定位记录时所经历的中间过程，从而加快存取速度。普遍运用在数据库和文件系统。

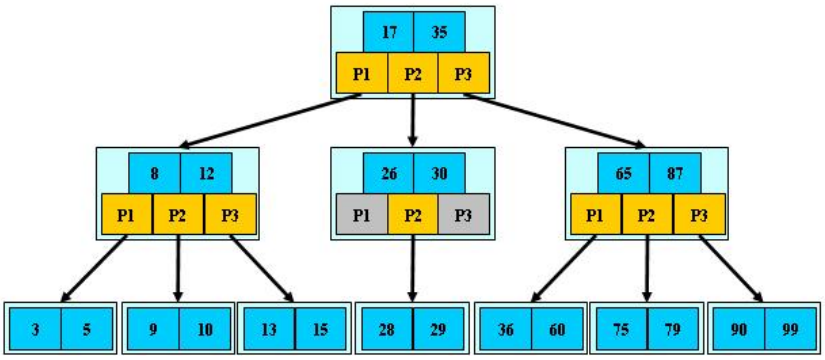
一棵 m 阶的 B-树或为空树或者满足：1.树中每个结点至多有 m 棵子树；2.若根节点不是叶子节点，则至少有两棵树；3.除根节点以外的其他非终端结点至少有 sup(m/2)棵子树，至多有 m-1 个；

B-树的生成是从空树起逐渐插入关键词而得。但由于 B-树结点中的关键字的个数必须≥sup[m/2]-1 根据其第三条性质；故而每次插入时，首先在最底层的某个非终端结点中添加一个关键字，如果关键字个数不超过 m-1（第一条性质），则插入完成，否则需要进行结点“分裂”，将多余结点逐层上移，直到满足条件。

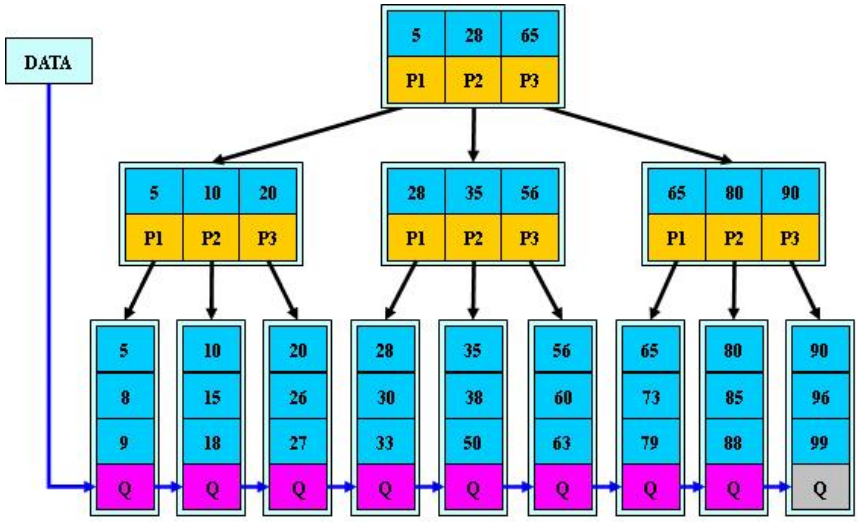
反之，当在 B-树中删除一个结点时首先应当找到该关键点的所在结点，并在其中删除之，若该节点为最下层的费终端结点，且其中的关键字数目不少于 sup(m/2)则删除完成，否则就要进行“合并”结点的操作。

B+ 树是一种树数据结构，通常用于数据库和操作系统的文件系统中。B+树与 B-树相比，在 B+树中所有的叶子节点包含了所有关键字的信息，所有的非终端信息可以看成索引部分，结点中仅含有其子树的最大或最小的关键字。B+树的随机查找插入删除均与 B-树类似，对于插入而言，若非终端节点等于给定值，并不终止，而是直到叶子结点，每次查找都会走一条从根到叶子节点的路径；其插入仅在叶子结点进行，当结点关键字个数超过 m-1 时将其分裂；删除也仅在叶子节点进行，当叶子结点中的最大关键字被删除时，其在费终端结点中的值可以作为一个分界关键字存在。

例 B 树：关键字集合分布在整颗树中；任何一个关键字出现且只出现在一个结点中；搜索有可能在非叶子结点结束；其搜索性能等价于在关键字全集内做一次二分查找；自动层次控制；



例 B+树，非叶子结点的子树指针与关键字个数相同；非叶子结点的子树指针 P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；为所有叶子结点增加一个链指针；所有关键字都在叶子结点出现；



9) 红黑树

红黑树是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组(STL 中的 set 与 map 底层均使用红黑树)。

红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。并添加一系列的约束条件，从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

因为每一个红黑树也是一个特化的二叉查找树，因此红黑树上的只读操作与普通二叉查找树上的只读操作相同。然而，在红黑树上进行插入操作和删除操作会导致不再符合红黑树的性质。恢复红黑树的属性需要少量(O(log n))的颜色变更(实际是非常快速的)和不超过三次树旋转(对于插入操作是两次)。虽然插入和删除很复杂，但操作时间仍可以保持为 O(log n) 次。

2. 栈

1) 括号配对

```
1.  typedef char SElemType;
2.  typedef int Status; // Status 是函数的类型,其值是函数结果状态代码，如 OK 等
3.  #define STACK_INIT_SIZE 10 // 存储空间初始分配量
4.  #define STACKINCREMENT 2 // 存储空间分配增量
5.  struct SqStack
6.  {
7.   SElemType *base; // 在栈构造之前和销毁之后，base 的值为 NULL
8.   SElemType *top; // 栈顶指针
9.   int stacksize; // 当前已分配的存储空间，以元素为单位
10.  }; // 顺序栈
11. Status InitStack(SqStack &S)//申请一个空栈
12. { S.base=(SElemType *)malloc(STACK_INIT_SIZE*sizeof(SElemType));
13.   S.top=S.base;
```

```
14.    if(!S.base) return ERROR;
15.    S.stacksize=STACK_INIT_SIZE;
16.    return OK;
17.   }
18. Status StackEmpty(SqStack S)//判断栈空
19. {if(S.base==S.top) return TRUE;
20.  else return FALSE;
21. }
22. Status Push(SqStack &S,SElemType e)//进栈
23. {
24.  if(S.top-S.base>=S.stacksize)
25.  {
26.        S.base=(SElemType
    *)realloc(S.base,(STACK_INIT_SIZE+S.stacksize)*sizeof(SElemType));
27.        if(!S.base) return ERROR;
28.        S.top=S.base+S.stacksize;
29.        S.stacksize=S.stacksize+STACK_INIT_SIZE;
30.  }
31.    *S.top=e;
32.    S.top++;
33.    return OK;
34.   }
35. Status Pop(SqStack &S,SElemType &e)//出栈
36. { if(S.base==S.top) return ERROR;
37.   e=*--S.top;
38.   return OK;
39.   }
40. void check()
41.   { // 对于输入的任意一个字符串，检验括号是否配对
42.     SqStack s;
43.     int i = 0;
44.     SElemType ch[80],*p,e;
45.     if(InitStack(s)) // 初始化栈成功
46.     {
47.       printf("输入表达式:");
48.       scanf("%s",ch);
49.       p=ch;
50.       while(*p) // 没到串尾
51.         switch(*p)
52.         {
53.           case '(':
54.           case '[':Push(s,*p);p++;i++;
55.                    break; // 左括号入栈，且 p++
56.           case ')':
57.           case ']':if(!StackEmpty(s)) // 栈不空
58.                    {
59.                      Pop(s,e); // 弹出栈顶元素
60.         if(*p==')'&&e!='('||*p==']'&&e!='[')   // 弹出的栈顶元素与*p 不配对
61.                    {
62.                       printf("括号不配对\n");
63.                       exit(ERROR);
64.                       }
65.                     else
66.                     {
67.                       p++;
68.                       break; // 跳出 switch 语句
69.                     }
70.                   }
```

```
                    else // 栈空
                    {
                      printf("没有一个左括号\n 配对不成功\n");
                      exit(ERROR);
                    }
            default: p++; // 其它字符不处理，指针向后移
          }
        if(StackEmpty(s)) // 字符串结束时栈空
         {if(i==0)
            printf("没有括号 不需要配对\n");
         else
            printf("配对成功\n");
          }
        else
           printf("没有一个右括号\n 配对不成功\n");
     }
  }
void main()
  {
    check();
  }
```

栈的基本操作有: Pop, Push, Check if empty,Initial;

3. 链表

1) 单向链表交点问题

如果存在交点，则两个链表必定程序 Y 字形，而不可能是 X 形；

如果存在交点，两个链表在交点及其之后的部分是一致的：长度和内容。

```
struct Node
{
   int data;
   struct Node * next;
};

 Node* FixIntersetion(Node* pHead1, Node* pHead2)
{
   Node* p1 = pHead1;
   Node* p2 = pHead2;
   int i = 1, j = 1, k = 0, diff = 0;
   //如果都是空链表，肯定没有交点
   if(pHead2 == NULL || pHead2 == NULL)
   {
      return NULL;
   }
   //获取链表长度
   while(p1->next != NULL)
   {
      p1 = p1->next;
      i++;
   }
   while(p2->next != NULL)
   {
      p2 = p2->next;
      j++;
```

```
27.    }
28.    //开始判断是否存在交点
29.    if(p1 != p2)
30.    {//根据有交点时，两个链表在交点及其之后的部分是公共的，因此，有交点的单链表的尾节点必定相同
31.       return NULL;       //如果尾节点不同，直接返回 NULL
32.    }
33.    else                  //否则寻找第一个相同的节点
34.    {
35.       p1=pHead1;
36.       p2=pHead2;
37.       //使得两者的起始比较位置离尾部的长度一致
38.       if(i>j)
39.       {
40.          diff=i-j;
41.          for(k=0; k<diff; k++)
42.          {
43.             p1 = p1->next;
44.          }
45.       }
46.       if(i<j)
47.       {
48.          diff=j-i;
49.          for(k=0; k<diff; k++)
50.          {
51.             p2 = p2->next;
52.          }
53.       }
54.       //开始比对，得出交点
55.       while(p1 != p2)
56.       {
57.          p1 = p1->next;
58.          p2 = p2->next;
59.       }
60.       return p1;
61.    }
62. }
```

2) 链表内环的存在问题

求环的长度，可以从碰撞点开始循环，当再次碰撞时经过的点数即环的长度；

求环的起点，碰撞点 p 到连接点的距离=头指针到连接点的距离，因此，分别从碰撞点、头指针开始走，相遇的那个点就是连接点。

```
1)  bool FixRing(Node * pHead)
2)  {
3)      Node * pSlow = pHead ;
4)      Node * pFast = pHead;
5)     while ( pFast && pFast -> next )  //如果存在环，不存在 p-next=NULL 的情况
6)      {
7)          pSlow = pSlow -> next;//前进一步
8)          pFast = pFast -> next -> next;//前进两步
9)          if ( pSlow == pFast )
10)             break ;
11)     }
12)     return   ! (pFast == NULL || pFast -> next == NULL);
13) }
```

如何判断一个单向链表是否存在环？哪种方法最好（Nokia）

A 双重循环 B 快慢指针 C 哈希查找

快慢指针方法前述方法即可，哈希方法如下：hash 表的方法，即设立一个数组，将链表结点中的值做数组下标，当赋值冲突时就是环的接入点

```
14) bool isloop(Llink *p)
15) {
16)  if(!p||!p->next)
17)   return true;
18)  int a[MAXSIZE],n=0;
19)  memset(a,0,sizeof(int)*MAXSIZE);
20)  p=p->next;
21)  while(p)
22)  {
23)   if(a[p->data]==-1)//存在环时，会发生冲突
24)   {
25)    cout<<"\nLoop node: "<<p->data<<endl
26)     <<"\nLen of node: "<<n<<endl;
27)    return true;
28)   }
29)   a[p->data]=-1;
30)   ++n;
31)   p=p->next;
32)  }
33)  return false;
34) }
```

3) 链表逆置反向存储

利用改变指针指向，依次将后面的指针指向前者。

```
struct Item
{
 char c;
 Item *next;
};

Item *Routine1(Item *x)
{
 Item *prev = NULL,
      *curr = x;
 while(curr)
 {
      Item *next = curr->next;
      curr->next = prev;
      prev = curr;
      curr = next;
 }
 return prev;
}
```

4) 将两个排序好的链表归并

递归方法：

```
ListNode * MergeList(ListNode*p1,ListNode*p2)
 {
        if (p1==NULL)
                return p2;
        if (p2==NULL)
                return p1;
        ListNode *head=NULL;
        if (p1->data<p2->data)
        {
                head = p1;
                head->next=MergeList(p1->next,p2);
        }
        else
        {
                head=p2;
                head->next=MergeList(p1,p2->next);
        }
        return head;
 }
```

非递归方法：

```
ListNode * MergeListNonRecursive(ListNode*p1,ListNode*p2)
{
        if (p1==NULL)
                return p2;
        if (p2==NULL)
                return p1;
        ListNode *head=NULL;
        if (p1->data<p2->data)
        {
                head=p1;
                p1=p1->next;
        }
        else
        {
                head = p2;
                p2=p2->next;
        }
        ListNode *temp=head;
        while(p1&&p2)
        {
                if (p1->data<p2->data)
```

```
            {
                temp->next=p1;
                p1=p1->next;
            }
            else
            {
                temp->next=p2;
                p2=p2->next;
            }
            temp=temp->next;
        }
        if (p1==NULL)
            temp->next=p2;
        else if (p2==NULL)
            temp->next=p1;
        return head;
}
```

4. 图

1) 基本知识

(1) n 个顶点 m 条边的全连通图，至少需要去掉几条边才能构成一棵树？

n 个顶点的树一定有 n-1 条边(但反之不成立)，所以需要去掉 m-(n-1)=m-n+1 条边

2) 图的表示

(1) 邻接表

邻接表是图的一种链式存储结构。邻接表中，对图中每个顶点建立一个单链表，第 i 个单链表中的结点表示依附于顶点 V_i 的边（对有向图是以顶点 V_i 为尾的弧）。

邻接表中的表结点和头结点结构：

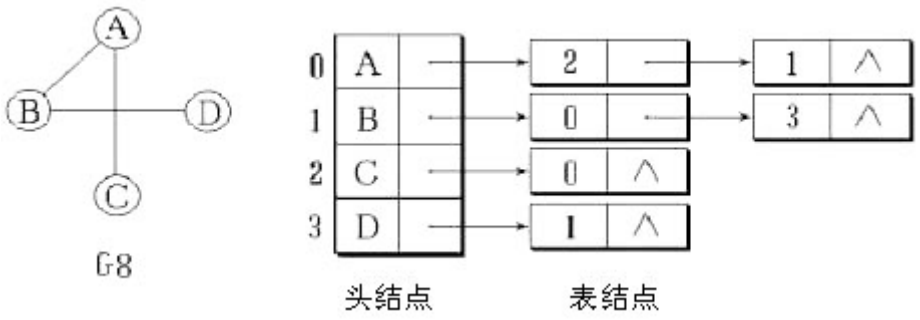
表结点

Adjvex	nextarc	info

头结点

Data	firstarc

无向图的邻接表

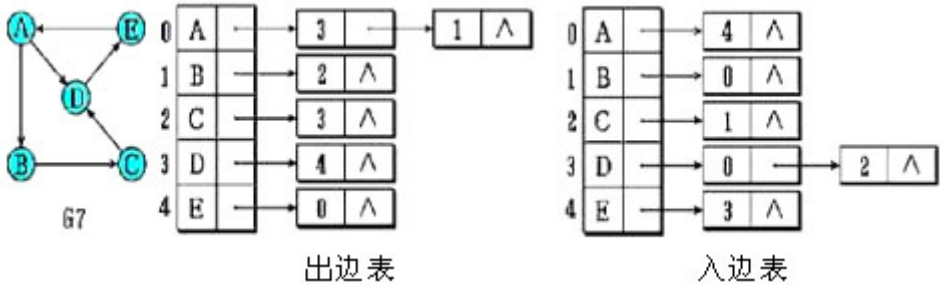


有向图的邻接表和逆邻接表

（一）在有向图的邻接表中，第 *i* 个单链表链接的边都是顶点 *i* 发出的边。

（二）为了求第 i 个顶点的入度，需要遍历整个邻接表。因此可以建立逆邻接表。

（三）在有向图的逆邻接表中，第 *i* 个单链表链接的边都是进入顶点 *i* 的边。



(a) 邻接表　　(b) 逆邻接表

邻接表小结

◆ 设图中有 n 个顶点，e 条边，则用邻接表表示无向图时，需要 n 个顶点结点，2e 个表结点；用邻接表表示有向图时，若不考虑逆邻接表，只需 n 个顶点结点，e 个边结点。

◆ 在无向图的邻接表中，顶点 v_i 的度恰为第 i 个链表中的结点数。

◆ 在有向图中，第 i 个链表中的结点个数只是顶点 v_i 的出度。在逆邻接表中的第 i 个链表中的结点个数为 v_i 的入度。

◆ 建立邻接表的时间复杂度为 O(n+e)。

(2) 邻接矩阵

邻接矩阵（Adjacency Matrix）的表示法，就是用一维数组存储图中顶点的信息，用矩阵表示图中各顶点之间的邻接关系。假设图 G＝（V，E）有 n 个确定的顶点，即 V＝{v0,v1,…,vn-1},则表示 G 中各顶点相邻关系为一个 n×n 的矩阵，矩阵的元素为：A[i][j]=wij

若 G 是网图，则邻接矩阵可定义为：A[i][j]=<vi,vj>

其中，wij 表示边(vi,vj)或<vi,vj>上的权值；∞表示一个计算机允许的、大于所有边上权值的数。

3) DFS&BFS

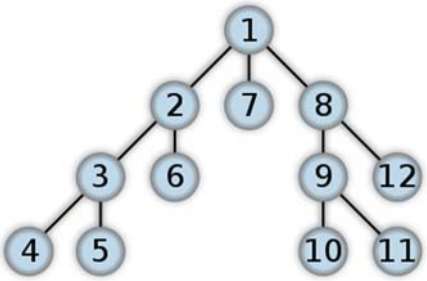
(1) 深度优先搜索算法（Depth-First-Search），是搜索算法的一种。是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点 v 的所有边都已被探寻过，搜

索将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。（类似**前序遍历**）参照之前的非递归遍历。

深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，可以方便的解决很多相关的图论问题，如最大路径问题等等。

```
std::stack<node*> visited, unvisited;
node* current, *root;
unvisited.push(root); //先把根节点放入 UNVISITED stack
while(!unvisited.empty()) //只有 UNVISITED 不空
{
    current=unvisited.top(); //当前应该访问的
    unvisited.pop();
    if(current->right!=NULL)
    unvisited.push(current->right); //把右边压入因右边的访问是在左边之后
    if(current->left!=NULL)
    unvisited.push(current->left);
    visited.push(current);
    cout<<current->value<<endl;
}
```

深度优先搜寻的读取数据顺序：



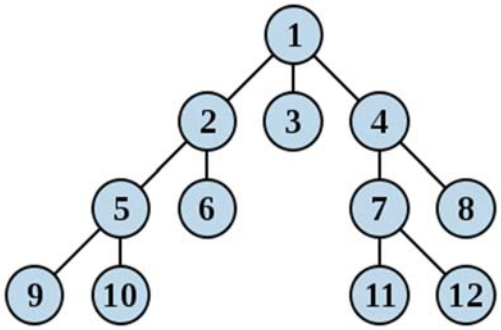
(2) BFS 是一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位置，彻底地搜索整张图，直到找到结果为止。BFS 并不使用经验法则演算法。BFS 是从根节点开始，沿着树的宽度遍历树的节点。如果所有节点均被访问，则算法中止。

从演算法的观点，所有因为展开节点而得到的子节点都会被加进一个**先进先出**的队列中。一般的实作里，其邻居节点尚未被检验过的节点会被放置在一个被称为 open 的容器中（例如伫列或是链表），而被检验过的节点则被放置在被称为 closed 的容器中。(open-closed 表)。

```
std::queue<node*> visited, unvisited;
node* current;
unvisited.push(root); //先把 root 放入 unvisited queue
while(!unvisited.empty()) //只有 unvisited 不空
 {
    current = (unvisited.front()); //目前应该遍历的
```

```
        if(current -> left != NULL)
            unvisited.push(current -> left); //把左边放入 queue 中
        if(current -> right != NULL) //右边压入。因为 QUEUE 是一个先进先出的结构，所以即使后面再压其他东西，依然会先访问这个。
            unvisited.push(current -> right);
        visited.push(current);
        cout << current -> value << endl;
        unvisited.pop();
    }
```

广度优先搜索的搜索顺序：



4) D&B&FW ALGORITHM

(1) **Dijkstra's algorithm** 使用了广度优先搜索算法。算法解决的是有向图中单个源点到其他顶点的最短路径问题。举例来说，如果图中的顶点表示城市，而边上的权重表示著城市间开车行经的距离，该演算法可以用来找到两个城市之间的最短路径。贪心算法。

该演算法的输入包含了一个有权重的有向图 G，以及 G 中的一个来源顶点 S。我们以 V 表示 G 中所有顶点的集合。每一个图中的边，都是两个顶点所形成的有序元素对。(u, v) 表示从顶点 u 到 v 有路径相连。我们以 E 所有边的集合，而边的权重则由权重函数 w: E → [0, ∞] 定义。因此，w(u, v) 就是从顶点 u 到顶点 v 的非负权重（weight）。边的权重可以想像成两个顶点之间的距离。任两点间路径的权重，就是该路径上所有边的权重总和。已知有 V 中有顶点 s 及 t，Dijkstra 演算法可以找到 s 到 t 的最低权重路径(例如，最短路径)。这个演算法也可以在一个图中，找到从一个顶点 s 到任何其他顶点的最短路径。

Demo: http://v.youku.com/v_show/id_XMjQyOTY1NDQw.html

(2) **Bellman-Ford** 求解单源最短路径问题的一种算法。它的原理是对图进行 V-1 次松弛操作，得到所有可能的最短路径。其优于迪科斯彻算法的方面是边的权值可以为负数、实现简单，缺点是时间复杂度过高，高达 O(VE)。但算法可以进行若干种优化，提高了效率。动态规划。

即进行不停地松弛，每次松弛把每条边都更新一下，若 n-1 次松弛后还能更新，则说明图中有负环，无法得出结果，否则就成功完成。Bellman-ford 算法有一个小

优化：每次松弛先设一个旗帜 flag，初值为 FALSE，若有边更新则赋值为 TRUE，最终如果还是 FALSE 则直接成功退出。Bellman-ford 算法浪费了许多时间做无必要的松弛，所以 SPFA 算法用队列进行了优化，效果十分显著。

(3) **Floyd-Warshall 算法**（Floyd-Warshall algorithm）是解决任意两点间的最短路径的一种算法，可以正确处理有向图或负权的最短路径问题，同时也被用于计算有向图的传递闭包。Floyd-Warshall 算法的时间复杂度为 O(N^3)，空间复杂度为 O(N^2)。Floyd-Warshall 算法的原理是动态规划。

5) 应用

Google 2014 Round A Test Problem E.

原题见：http://blog.csdn.net/wood_water/article/details/11982993

问题描述：有 N 个房间，每个房间都有对应的颜色标号（用小于等于两位的字母数字组合标志），相同颜色的房间可以利用 Teleporters 自由转换，另外还可以利用 Turbolifts，每个 Turbolifts 有特定的使用方式，可以从 A 房间到 B 房间，花费 T 时间。然后给定若干人及其初始位置和目的位置，输出到达的最小时间，如不可达，输出-1。

此题可以抽象为 N 个点的加权有向图之间的最短路径问题，如果两个点同色则直接双向连通，权值为零，可以直接抵达，可以将其看做同一个点。否则，若存在由 A 点到 B 点花费时间 T 的 Turbolifts，则存在 A 到 B 的权重为 T 的边，由此构建图，即可转化为求两点之间的最短路径，从而可利用第 4)部分算法进行求解。

如下代码即使用 D 算法求解，图的信息存储在 v 中，相当于邻接矩阵。

```
#define maxn 2010
#define maxm 80100
#define maxl 1000000000
#define mod 1000003
map<string,int> mm;
vector<pair<int,int> > v[maxn];

int find(const string &s){
    if(mm.count(s))return mm[s];//return s times
    mm[s]=++tot;
    v[tot].clear();
    return tot;
}
int d[maxn];
int dui[maxn];
bool visit[maxn];
int getDis(int x,int y){
    int head=0,tail=1,i,j,tmp;
    //there are tot different points
    for(i=1;i<=tot;++i){
        d[i]=maxl;
        visit[i]=false;
    }
    d[x]=0;
    visit[x]=true;
    dui[1]=x;
    while(head!=tail){
        if(++head==maxn)head-=maxn;
        x=dui[head];
        for(i=0;i<v[x].size();++i){
            j=v[x][i].first;
            tmp=d[x]+v[x][i].second;
            if(tmp<d[j]){
                d[j]=tmp;
                if(!visit[j]){
                    visit[j]=true;

if(++tail==maxn)tail-=maxn;
                    dui[tail]=j;
```

```
                }
            }
        }
        visit[x]=false;
    }
    if(d[y]<maxl)return d[y];else return -1;
}
string color2[maxm];
void solvepath(){
    int n,i,m,x,y,z,q;
    cin>>n;
    mm.clear();
    tot=0;
    for(i=1;i<=n;++i){
        cin>>color2[i];
    }//color 1:N
    cin>>m;
    for(i=1;i<=m;++i){
        cin>>x>>y>>z;//x->y in z time
        x=find(color2[x]);y=find(color2[y]);
        v[x].push_back(make_pair(y,z));
    }//
    cin>>q;// solider number
    for(i=1;i<=q;++i){
        cin>>x>>y;//pos
        x=find(color2[x]);y=find(color2[y]);
        cout<<getDis(x,y)<<endl;
    }
}
```

5. 排序

1) 基本知识

(1) 最坏情况下，合并两个大小为 n 的已排序数组所需要的比较次：2n-1.
以比较交换为基础的排序方式复杂度下界为 O(nlogn)

(2) 哪个排序的比较次数与初始情况无关？(C)
A Shell B 归并 C 选择 D 插入

(3) 排序的稳定的是指当有两个有相等关键的纪录 R 和 S，且在原本的串列中 R 出现在 S 之前，在排序过的串列中 R 也将会是在 S 之前。冒泡排序、插入排序、归并排序二叉树排序等是稳定的；而选择排序、快速排序、堆排序、希尔排序等均不稳定。

(4) 各种排序复杂度与比较及各种描述

名称	数据对象	稳定性	时间复杂度		空间复杂度	描述
			平均	最坏		
插入排序	数组、链表	✓	$O(n^2)$		$O(1)$	(有序区,无序区)。把无序区的第一个元素插入到有序区的合适的位置。对数组:少,换得多。
直接选择排序	数组	✗	$O(n^2)$		$O(1)$	(有序区,无序区)。在无序区里找一个最小的元素跟在有序区的后面。 对数组:多,换得少。
	链表	✓				
堆排序	数组	✗	$O(nlogn)$		$O(1)$	(最大堆,有序区)。从堆顶把根卸出来放在有序区之前,再恢复堆。
归并排序	数组、链表	✓	$O(nlogn)$		$O(n)+O(logn)$, 如果不是从下到上	把数据分为两段,从两段中逐个选最小的元素移入新数据段的末尾。可从上到下或上进行。
快速排序	数组	✗	$O(nlogn)$	$O(n^2)$	$O(logn),O(n)$	(小数,枢纽元,大数)。
Accum qsort	链表	✓	$O(nlogn)$	$O(n^2)$	$O(logn),O(n)$	(无序区,有序区)。把无序区分为(小数,枢纽元,大数),从后到前压入有序区。
决策树排序		✓	$O(logn!)$		$O(n!)$	O(n) <O(logn!) <O(nlogn)
计数排序	数组、链表	✓	$O(n)$		$O(n+m)$	统计小于等于该元素值的元素的个数 i,于是该元素就放在目标数组的索引 i位。(i≥0)
桶排序	数组、链表	✓	$O(n)$		$O(m)$	将值为 i 的元素放入i 号桶,最后依次把桶里的元素倒出来。
基数排序	数组、链表	✓	$O(k*n)$, 最坏: $O(n^2)$			一种多关键字的排序算法,可用桶排序实现。

- 均按从小到大排列
- k 代表数值中的"数位"个数
- n 代表数据规模
- m 代表数据的最大值减最小值

(5) 寻找一个数组中第二大的数。

```
int GetSecondMax(int *a,int len)
{
 if (len<2)
 {
      return -1;
 }
 int first=a[0];
 int second=a[1];
 for (int i=1;i<len; i++)
 {
      if (a[i]>first)
      {
           second=first;
           first=a[i];
      }
      else if(a[i]>second)
      {
           second=a[i];
      }
 }
 return second;
}
```

2) 快速排序

快速排序利用分治思想进行排序。其比较次数与具体数列相关。平均时间复杂度为O(nlogn),最坏情况为O(n2)。一般情况下,需要O(lgn)的辅助空间。快速排序是不稳定的。如下是用 C 和 python 分别实现的快速排序代码,实现由高到低排序。

```
int partition(int* array, int left, int
right)
{
 int index = left;
 int pivot = array[index];
 swap(array[index], array[right]);
 for (int i=left; i<right; i++)
 {
      if (array[i] > pivot)
      {
      swap(array[index++], array[i]);
      }// 降序
 }
 swap(array[right], array[index]);
 return index;
}
```

```
void qsort(int* array, int left, int right)
{
  if (left >= right)
        return;
  int index = partition(array, left, right);
  qsort(array, left, index - 1);//左部分
  qsort(array, index + 1, right);//右部分
}
```

PYTHON:

```
def quicksort2(arr,left,right):
    lp = left
    rp = right
    key = arr[right]
    if lp==rp:return
    while True:
        while (arr[lp]>= key) and (rp>lp):
            lp=lp+1
        while (arr[rp]<= key) and (rp>lp):
            rp=rp-1
        arr[lp],arr[rp] = arr[rp],arr[lp]
        if lp>=rp:break
    arr[rp],arr[right] = arr[right],arr[rp]
    if left<lp:
        quicksort2(arr,left,lp-1)
    quicksort2(arr,rp,right)
```

3) 插入排序

插入排序逐个插入已排序的数组中，比较次数会减少，但交换次数增多。
插入排序最好情况下为 O(n),平均及最坏情况下均为 O(n^2),是一种稳定的排序方式。需要的空间复杂度为 O(1)；
类似于冒泡排序，插入排序也是稳定的。

4) 希尔排序

希尔排序是对插入排序的一种改进，过将比较的全部元素分为几个区域来提升插入排序的性能。这样可以让一个元素可以一次性地朝最终位置前进一大步。然后算法再取越来越小的步长进行排序，算法的最后一步就是普通的插入排序，但是到了这步，需排序的数据几乎是已排好的了（此时插入排序较快）。步长选择对希尔排序的效果十分重要。在最好的情况下，时间复杂度最优为 O(n)，平均情况下为 O(n^3/2),最坏情况下为 O(n(lgn)^2);希尔排序是不稳定的。其余插入排序类似，比较次数也与初始数列有关。

插入排序：

```
void insertsort(int *data, int size){
    for (int i=1;i<size;i++)
    {
        int key=data[i];
        int j(i-1);
    for (;j>=0&&data[j]>key;j--)
            data[j+1]=data[j];
        data[j+1]=key;
    }
}
```

希尔排序：

```
void shellsort(int *data, size_t size){
  for (int gap = size / 2; gap > 0; gap /= 2)
      for (int i = gap; i < size; ++i){
          int key = data[i];
          int j = 0;
for( j = i -gap; j >= 0 && data[j] > key; j -=gap)
          data[j+gap] = data[j];
          data[j+gap] = key;}}
```

5) 选择排序

直接选择排序比较次数也不会发生变化任意两者均需比较。
时间复杂度平均最坏情况下为 O(n^2),一般情况下空间复杂度为 O(1)，不稳定。

6) 归并排序

归并排序同样利用了分治思想（n 个元素）：将序列每相邻两个数字进行归并操作，floor(n/2)形成个序列，排序后每个序列包含两个元素;将上述序列再次归并，形成 florr(n/4)个序列，每个序列包含四个元素;重复步骤 2，直到所有元素排序完毕。其比较次数与具体数列有关。需要 O(n)个辅助空间。归并排序最坏及平均时间复杂度均为 O(nlogn),而且是稳定的。经常用到的合并两个已排序的链表或者已排序的数组，其思路也均为归并排序最终的合并阶段。

```
//合并两个已排好序的数列到 b 中
void Merge(int a[], int b[], int low,
int mid, int high)
{
int k = low;
int begin1 = low;
int end1 = mid;
int begin2 = mid + 1;
int end2 = high;
while(k <= high )
{
    if(begin1 > end1)
      b[k++] = a[begin2++];
     //表示 begin1 已空
   else if(begin2 > end2)
      b[k++] = a[begin1++];
     //表示 begin2 已空
   else
      {
      if(a[begin1] <= a[begin2])
         b[k++] = a[begin1++];
      else
         b[k++] = a[begin2++];
         }
      }
}
```

```
//按照指定的序列分段进行排序
void MergePass(int a[], int b[], int seg, int
size)
{
      int seg_start_ind = 0;
      while(seg_start_ind <= size - 2 * seg)
      //size - 2 * seg 的意思是满足可两两归并
的最小值
      {
Merge(a, b, seg_start_ind, seg_start_ind + seg -
1, seg_start_ind + seg * 2 - 1);
            seg_start_ind += 2 * seg;
       }
//如果一段是正好可归并数量而另一段则少
于正好可归并的数量
if(seg_start_ind + seg < =size)
Merge(a, b, seg_start_ind, seg_start_ind + seg -
1, size - 1);
else
    for(int j = seg_start_ind; j < size; j++)
    //如果只剩下一段或者更少的数量
          b[j] = a[j];
}
```

```
void MergeSort(int a[], int size)
{
      int* temp = new int[size];
      int seg = 1;
      while(seg < size)
      {
            MergePass(a, temp, seg, size);
            seg += seg;
            MergePass(temp, a, seg, size);
            seg += seg;
      }
      delete [] temp;
}
```

原地归并：已知 A,B 两已排序数组，A 的缓存足够大，将 A 与 B 合并到 A 中形成一个有序数组。（SAP2014）其实逻辑思路很简单。但是现场手写要保证做对对学渣来说还是有些难度，需要多多练习啊。可以另开一个空间归并，如上，也可以插入。

```
void insertatk(int *a, int length,int
pos, int value)
{
  for (int i=length-1;i>=pos;i--)
  {
        a[i+1]=a[i];
  }
  a[pos]=value;
}
int* merge( int *a, int *b, int lengthA,
int lengthB)
{
  int *c=a;int j=0;
  int len = lengthA,count(0);
//判断是否已经到达 A 的尾部
  for (int i=0;i<lengthB;i++)
  {
        if (b[i]<=a[j])
        {
              insertatk(a,lengthA,j,b[i]);
              j++;
              lengthA++;
        }
        else
        {
          while(b[i]>a[j]&&count<len)
              {
                    j++;
                    count++;
              }
              if (count>=len)
              {
                    a[j++]=b[i];
                    continue;
              }
              insertatk(a,lengthA,j,b[i]);
              j++;
              lengthA++;
        }
  }
  return c;
}
```

7) 堆排序

堆排序通过建立堆，每次将堆的顶点剔除后余下的元素继续建堆，直到所有的元素均完成排列；故堆排序主要完成两个步骤：1.建立堆结构 2.删除堆根节点；堆排序的平均及最坏时间复杂度均为 O(nlgn),堆排序是就地排序,其空间复杂度为 O(1)，并且是不稳定的；堆通常是一个可以被看做一棵树的数组对象。堆总是满足下列性质：堆中某个节点的值总是不大于或不小于其父节点的值。根据 descending 还是 ascending 建立大根堆或小根堆。

建立堆的过程，是对完全二叉树进行调整的过程。首先将长为 n 的序列看作一个完全二叉树，然后从 n/2 结点开始逐渐调整，以建小根堆为例，如果该结点大于其子结点，则将该结点与较小的子结点交换，如此将 n/2 之前的结点也进行调整，从而得到结果。(暴风 2014 校招)

```
void sift(int d[], int ind, int len)//建堆
{
 int i = ind;//置 i 为要筛选的节点
 //c 中保存 i 节点的左孩子
 int c = i * 2 + 1;
//+1 的目的就是为了解决节点从 0 开始而他的左孩子一直为 0 的问题
 while(c < len)//#未筛选到叶子节点#%
 {
      //#如果要筛选的节点既有左孩子又有右孩子并且左孩子值小于右孩子#%
      //#从二者中选出较大的并记录#%
      if(c + 1 < len && d[c] < d[c + 1])
```

```
            c++;//选取较大子树
        //如果要筛选的节点中的值大于左右孩子的较大者则退出
        if(d[i] > d[c]) break;//父节点大于子节点则退出
        else
        {
            //#交换#%
            int t = d[c];
            d[c] = d[i];
            d[i] = t;
            //#重置要筛选的节点和要筛选的左孩子#%
            i = c;
            c = 2 * i + 1;
        }
    }
}
```

建堆示意过程可参见：http://dwz.cn/au2SM

```
void heap_sort(int d[], int n)
{
    //#初始化建堆, i 从最后一个非叶子节点开始#%
    for(int i = n / 2; i >= 0; i--)
        sift(d, i, n);
    for(int j = 0; j < n; j++)
    {
        //#交换#%每次找到最大的值置于该组后
        int t = d[0];
        d[0] = d[n - j - 1];
        d[n - j - 1] = t;
        //#筛选编号为 0 #%
        sift(d, 0, n - j - 1);
    }
}
```

8) 拓扑排序

拓扑排序 Topological Order 是一种针对有向无环图（Directed Acyclic Graph）的排序.

(1)从有向图中选择一个没有前驱(即入度为 0)的顶点并且输出它.

(2)从网中删去该顶点,并且删去从该顶点发出的全部有向边.

(3)重复上述两步,直到剩余的网中不再存在没有前趋的顶点为止.

拓扑排序常用来确定一个依赖关系集中，事物发生的顺序。例如，在日常工作中，可能会将项目拆分成 A、B、C、D 四个子部分来完成，但 A 依赖于 B 和 D，C 依赖于 D。为了计算这个项目进行的顺序，可对这个关系集进行拓扑排序，得出一个线性的序列，则排在前面的任务就是需要先完成的任务。

① 将图中顶点按拓扑次序排成一行，则图中所有的有向边均是从左指向右的。

② 若图中存在有向环，则不可能使顶点满足拓扑次序。
③ 一个 DAG 的拓扑序列通常表示某种方案切实可行。

9) 计数排序

当输入的元素是 n 个 0 到 k 之间的整数时，它的运行时间是 O(n + k)。计数排序不是比较排序，排序的速度快于任何比较排序算法。

由于用来计数的数组 C 的长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上 1），这使得计数排序对于数据范围很大的数组，需要大量时间和内存。例如：计数排序是用来排序 0 到 100 之间的数字的最好的算法如对年龄，但是它不适合按字母顺序排序人名。但是，计数排序可以用在基数排序中的算法来排序数据范围很大的数组。

算法的步骤如下：

- 找出待排序的数组中最大和最小的元素
- 统计数组中每个值为 i 的元素出现的次数，存入数组 C 的第 i 项
- 对所有的计数累加（从 C 中的第一个元素开始，每一项和前一项相加）
- 反向填充目标数组：将每个元素 i 放在新数组的第 C(i)项，每放一个元素就将 C(i)减去 1

```
void counting_sort(int *ini_arr, int *sorted_arr, int n)
{
        int *count_arr = (int *)malloc(sizeof(int) * 100);
        int i, j, k;
        for(k=0; k<100; k++){
                count_arr[k] = 0;}
        for(i=0; i<n; i++){
                count_arr[ini_arr[i]]++; }
        for(k=1; k<100; k++){
                count_arr[k] += count_arr[k-1]; }
        for(j=n; j>0; j--){
                sorted_arr[count_arr[ini_arr[j-1]]-1] = ini_arr[j-1];
                count_arr[ini_arr[j-1]]--; }
        free(count_arr);
}
```

6. HASH

Hash 表是一种数据结构，可以将键值 key 映射到我们需要其代表的位置 value。而其中间的对应关系是由散列函数实现的，散列函数有时候会出现将多个不同值存在同一个键的情况，这种情况成为碰撞。

一个同样大小的 hash 和数组之间各有何优缺点？（DeNA2014 笔试）

1) CONSISTENT HASHING

一致性 Hash 详见：http://blog.csdn.net/sparkliang/article/details/5279393

比如你有 N 个 cache 服务器（后面简称 cache ），那么如何将一个对象 object 映射到 N 个 cache 上呢，你很可能会采用类似下面的通用方法计算 object 的 hash 值，然后均匀的映射到到 N 个 cache ；

hash(object)%N

一切都运行正常，但是再考虑如下的两种情况；

1 一个 cache 服务器 m down 掉了（在实际应用中必须要考虑这种情况），这样所有映射到 cache m 的对象都会失效，怎么办，需要把 cache m 从 cache 中移除，这时候 cache 是 N-1 台，映射公式变成了 hash(object)%(N-1) ；

2 由于访问加重，需要添加 cache ，这时候 cache 是 N+1 台，映射公式变成了 hash(object)%(N+1) ；

1 和 2 意味着什么？这意味着突然之间几乎所有的 cache 都失效了。对于服务器而言，这是一场灾难，洪水般的访问都会直接冲向后台服务器；为此提出一致性 hashing 的概念。

单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

考量 Hash 算法的另一个指标是平衡性 (Balance) ，是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。

一致性 hash 的两个特点即在于环形 hash 空间及虚拟结点概念。

7. 查找元素

1) 一般二分查找

N 个数字已经从大到小排列，给定一个数 P，P 是否存在于这 N 个数中？在怎样的策略下，使得最坏的情况下，需要比较的次数最少？为什么？

顺序查找,平均需要比较（N+1）/2 次，最多需要比较 N 次。

如果二分查找，则最多需要比较 log(n)+1 次。

如果排列有规律，采用哈希表，则一次查表就可以找到结果。

2) 循环升序数组

二分查找的变体，二分查找的输入是已经排好序的序列，此时的输入方法见手写代码-二分查找，其一个变体为，如果输入为循环升序数组(如 678912345)时，应当如何调整？(百度面试)

```
int    binary_cycle_search(int *a, int ele,int low, int high)
 {
        if (low>=high)
        {
              return -1;
        }
        int mid =(low+high)/2;
        if (a[mid]==ele)
```

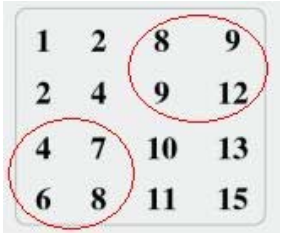
```
    {
        return mid;
    }

    if (a[mid]<=a[high])//右半部分已排序
    {
        if (a[mid]<ele&&ele<=a[high])
        {
            return binary_cycle_search(a,ele,mid+1,high);
        }
    else
        {
            return binary_cycle_search(a,ele,low,mid-1);
        }
    }
    else//左半部分已排序
    {
        if (ele>=a[0]&&ele<a[mid])
        {
            return binary_cycle_search(a,ele,low,mid-1);
        }
    else
        {
            return binary_cycle_search(a,ele,mid+1,high);
        }
    }
}
```

3) 杨氏矩阵

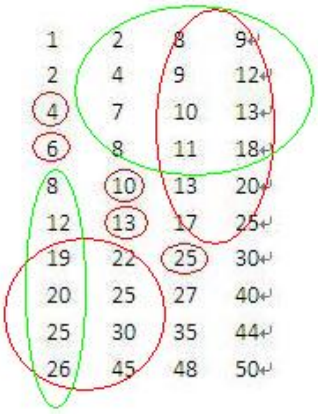
另外的一个变体即，对二维排序数组(行和列均排好序的数组)此即著名的“**杨氏矩阵**”进行查找某个元素(创新工场 2014bupt 考试)

此时可以首先将目标值与左上角或右下角值相比较，如果其小于左上角值或大于右上角值，则返回-1，否则开始在数组内部查找。因为二维排序数组，可以看成是一个沿对角线方向增长的结构，所以可以根据对角线进行查询。找到对角线上的一个元素，使得这个元素小于待查找的 key，并且下一元素大于待查找的 key，那么只要在这个元素的左下角矩阵和右上角矩阵递归继续对角线查找就可以了。



例如上图例子里查找 7，只要找到对角线的元素 4，然后递归查找红圈的矩阵就可以了，左上角矩阵最大值 4<7，右下角矩阵最小值 10>7，无需查找了，但是此题并没有告诉我们原始矩阵是 n*n 的，这是比较麻烦的地方，不过思路是一样的，无非不能用对角线查找这样简单的办法了，假设 m*n 的矩阵，对角线查找的办法改进为 i =

(m1+m2)/2，j = (n1+n2)/2 进行查找就可以了，(m1,n1)为矩阵最左上角元素下标，(m2,n2)为最右下角元素下标



假设查找 17，第一次比较 10，然后比较 25，然后比较 13，返回元素 13，这时候再递归查找 13 左下角的矩阵和右上角的矩阵就可以了（红色椭圆部分）；如果是查找 9，第一次比较 10，然后比较 4，然后比较 6，返回元素 6，这时候递归查找 6 左下角的矩阵和右上角矩阵（绿色椭圆部分）。

代码如下：

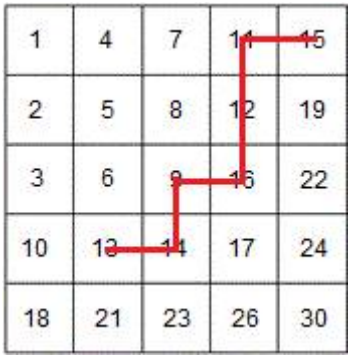
```
a 是二维数组首地址，(m1, n1)左上角坐标，(m2, n2)右下角坐标，参数 n 是矩阵一行的元素数
int binsearch(int value, int *a, int n, int m1, int n1, int m2, int n2)
{
    int begin_m1 = m1, begin_n1 = n1, end_m2 = m2, end_n2 = n2;
    int left_result = 0,    right_result = 0;
    int i = (m1+m2)/2, j = (n1+n2)/2;
    if (a == NULL)
        return 0;
    if (value < *(a+m1*n+n1) || value > *(a+m2*n+n2))
        return 0;
    else if (value == *(a+m1*n+n1) || value == *(a+m2*n+n2))
        return 1;

    while ((i!=m1 || j!=n1) && (i!=m2 || j!=n2)){
        if ( value == *(a+i*n+j) )
            return 1;
        else if ( value < *(a+i*n+j) ){
            m2 = i;
            n2 = j;
            i = (i+m1)/2;
            j = (j+n1)/2;
        }
        else{
            m1 = i;
```

```
                n1 = j;
                i = (i+m2)/2;
                j = (j+n2)/2;
            }
        }
        //search left & right
        if ( i<end_m2 )
            left_result = binsearch(value, a, n, i+1, begin_n1, end_m2, j);
        if ( j<end_n2 )
            right_result = binsearch(value, a, n, begin_m1, j+1, i, end_n2);
        if (left_result | right_result )
            return 1;
        else
            return 0;
}
```

第二种解法：Step-wise 线性搜索解法：

从右上角开始，每次将搜索值与右上角的值比较，如果大于右上角的值，则直接去除 1 行，否则，则去掉 1 列。如下图显示了查找 13 的轨迹。首先与右上角 15 比较，13<15，所以去掉最右 1 列，然后与 11 比较，这是 13>11，去掉最上面 1 行…以此类推，最后找到 13。算法复杂度 O(n)，最坏情况需要 2n 步，即从右上角开始查找，而要查找的目标值在左下角的时候。



代码如下：

```
1.  bool stepWise(int mat[][N_MAX], int N, int target,
2.                int &row, int &col) {
3.    if (target < mat[0][0] || target > mat[N-1][N-1]) return false;
4.    row = 0;
5.    col = N-1;
6.    while (row <= N-1 && col >= 0) {
7.      if (mat[row][col] < target)
8.        row++;
9.      else if (mat[row][col] > target)
10.       col--;
11.     else
12.       return true;
```

```
13.   }
14.   return false;
15. }
```

4) 跨行查找字符串

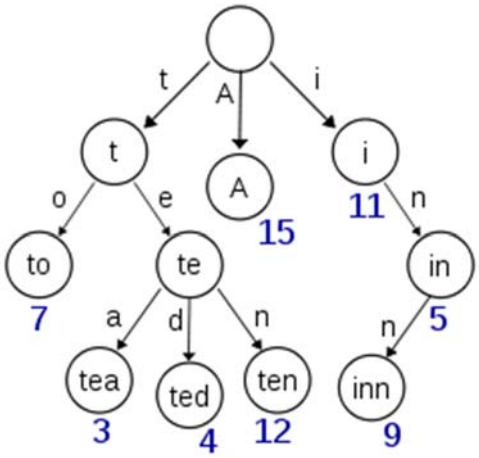
Yahoo 笔试。

5) TRIE 树

Trie 树又称单词查找树，Trie 树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来节约存储空间，最大限度地减少无谓的字符串比较，查询效率比哈希表高。

根节点不包含字符，除根节点外每一个节点都只包含一个字符； 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串； 每个节点的所有子节点包含的字符都不相同。

在计算机科学中，trie，又称前缀树或字典树，是一种有序树，用于保存关联数组，其中的键通常是字符串。与二叉查找树不同，键不是直接保存在节点中，而是由节点在树中的位置决定。一个节点的所有子孙都有相同的前缀，也就是这个节点对应的字符串，而根节点对应空字符串。一般情况下，不是所有的节点都有对应的值，只有叶子节点和部分内部节点所对应的键才有相关的值。



8. 其他

1) 主定理与复杂度

见算法导论主定理

定理 4.1(主定理) 设 $a \geqslant 1$ 和 $b > 1$ 为常数，设 $f(n)$为一函数，$T(n)$由递归式

$$T(n) = aT(n/b) + f(n)$$

对非负整数定义，其中 n/b 指 $\lfloor n/b \rfloor$ 或 $\lceil n/b \rceil$。那么 $T(n)$可能有如下的渐近界：

1)若对于某常数 $\varepsilon > 0$，有 $f(n) = O(n^{\log_b a - \varepsilon})$，则 $T(n) = \Theta(n^{\log_b a})$；

2)若 $f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a} \lg n)$；

3)若对某常数 $\varepsilon > 0$，有 $f(n) = \Omega(n^{\log_b a + \varepsilon})$，且对常数 $c < 1$ 与所有足够大的 n，有 $af(n/b) \leqslant cf(n)$，则 $T(n) = \Theta(f(n))$。 ■

73

例: T(n)=25T(n/5)+n^2 则其时间复杂度为?

属于第二种情况，故为 O(n^2lgn)

2) 静态存储与动态存储

变量的存储方式可分为：“静态存储”和“动态存储”两种。

静态存储变量通常是在变量定义时就分定存储单元并一直保持不变，直至整个程序结束。全局变量即属于此类存储方式。

动态存储变量是在程序执行过程中，使用它时才分配存储单元，使用完毕立即释放。典型的例子是函数的形式参数，在函数定义时并不给行参分配存储单元，只是在函数被调用时，才予以分配，调用函数完毕立即释放。如果一个函数被多次调用时，则反复地分配、释放形参变量的存储单元。

静态存储变量是一直存在的，而动态存储变量则时而存在时而消失。通常把由于变量存储方式不同而产生的特性称为变量的生存期。

3) 字符串匹配

(1) KMP 算法

KMP 算法是由提出该算法的三个人名缩写而成。是一种平均复杂度为 O(m+n)的字符串匹配算法,主要根据匹配字符串取得覆盖函数，在比对时根据模式串本身的特性跳过不匹配的字符，利用已获取的信息减少重复信息的匹配。

```
9.     int kmp_find(const string& target,const string& pattern)
10. {
11.  const int target_length = target.size();
12.  const int pattern_length = pattern.size();
13.  int * overlay_value = new int[pattern_length];
14.  overlay_value[0] = -1;//确定重合矩阵，表征模式串的重合效果
15.  int index = 0;
16.  for(int i=1;i<pattern_length;++i)
17.  {
18.       index = overlay_value[i-1];
19.       while(index>=0 && pattern[index+1]!=pattern[i])
20.       {
21.            index   = overlay_value[index];
22.       }
23.       if(pattern[index+1]==pattern[i])
24.       {
25.            overlay_value[i] = index +1;
26.       }
```

```
        else
        {
            overlay_value[i] = -1;
        }
    }

    //match algorithm start
    int pattern_index = 0;
    int target_index = 0;
    while(pattern_index<pattern_length&&target_index<target_length)
    {
        if(target[target_index]==pattern[pattern_index])
        {
            ++target_index;
            ++pattern_index;
        }
        else if(pattern_index==0)
        {
            ++target_index;
        }
        else
        {
            pattern_index = overlay_value[pattern_index-1]+1;
        }
    }
    if(pattern_index==pattern_length)
    {
        return target_index-pattern_index;
    }
    else
    {
        return -1;
    }
    delete [] overlay_value;
}
```

I	0	1	2	3	4	5	6
W[i]	A	B	C	D	A	B	D
T[i]	-1	0	0	0	0	1	2

(2) BM 算法

在文档实际应用当中,使用的是 BM 算法，BM 算法的速度一般比 kmp 算法快 3-5 倍。BM 算法在移动模式串的时候是从左到右，而进行比较的时候是从右到左的。BM 算法实际上包含两个并行的算法，坏字符算法和好后缀算法。这两种算法的目的就是让模式串每次向右移动尽可能大的距离。

(3) Not including the empty substring, the number of substrings of a string of length where symbols only occur once, is the number of ways to choose two distinct places between symbols to start/end the substring. Including the very beginning and very end of the string, there are such places. So there are non-empty substrings.(在一个由 n 个不同字符组成的字符串中，其子字符串的数目，由插空方法得出)

(4) In mathematics, a subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.（子序列）

(5) 已知一个栈，现在有 n 个按升序排列的数(1,2,3...n)，按照顺序入栈，出栈的顺序不确定，则有多少种可能的出栈顺序? C2nˆn - C2nˆ(n+1)

四．数据与计算机通信

1．OSI

OSI 七层协议由高到低包括应用层、表示层、会话层、传输层、网络层、数据链路层和物理层；

应用层能与应用界面沟通，以达到展示给用户的目的。 在此常见的协定有: HTTP，HTTPS，FTP，TELNET，SSH，SMTP，POP3 等。

表示层能为不同的用户端提供数据和信息的语法转换内码，使系统能解读成正确的数据。同时，也能提供压缩解压、加密解密。

会谈层用于为通讯双方制定通讯方式，并建立、注销会话（双方通讯）。

传输层用于控制数据流量，并且进行侦错及错误处理，以确保通讯顺利。而传送端的传输层会为封包加上序号，方便接收端把封包重组为有用的资料或档案。协议：TCP UDP

网络层的作用是决定如何将发送方的数据传到接收方。该层通过考虑网络拥塞程度、服务质量、发送优先权、每次路由的耗费来决定节点 X 到节点 Y 的最佳路径。router 路由器工作在这一层，通过不断的接收与传送数据使得网络变得相互联通。协议：IP ARP ICMP IGMP

数据链路层的功能在于管理第一层的位元资料，并且将正确的资料传送到没有传输错误的路线中。建立还有辨认资料开始以及结束的位置同时予以标记。另外，就是处理由资料受损、遗失甚至重复传输错误的问题，使后续的层级不会受到影响，所以它执行资料的侦错、重传或修正，还有决定设备何时进行传输。 设备有：Bridge 网桥(用于有线网与无线网之间)switch 交换器；

物理层定义了所有电子及物理设备的规范。其中特别定义了设备与物理媒介之间的关系，这包括了针脚、电压、线缆规范、集线器、中继器、网卡、主机适配器（在 SAN 中使用的主机适配器）以及其他的设备的设计定义。因为物理层传送的是原始的比特数据流，即设计的目的是为了保证当发送时的信号为二进制“1”时，对方接收到的也是二进制“1”而不是二进制“0”。因而就需要定义哪个设备有几个针脚，其中哪个针脚发送的多少电压代表二进制“1”或二进制“0”，还有例如一个 bit 需要持续几微秒，传输信号是否在双向上同时进行，最初的连接如何建立和最终如何终止等问题。HUB 集线器工作在此层。

为了更好理解物理层与数据链路层之间的区别，可以把物理层认为是主要的，是与某个单一设备与传输媒介之间的交互有关，而数据链路层则更多地关注使用同一个通讯媒介的多个设备（例如，至少两个设备）之间的互动。物理层的作用是告诉某个设备如何传送信号至一个通讯媒介，以及另外一个设备如何接收这个信号（大多数情况下它并不会告诉设备如何与通讯媒介相连接）。

UDP: IP 电话(VoIP)是典型的 UDP 应用。其他基于 UDP 的协议包括：域名系统(DNS(53))、简单网络管理协议(SNMP(161))、动态主机配置协议(DHCP)、路由信息协议(RIP)和某些影音串流服务；

TCP: 包括 HTTP(80)/HTTPS（万维网协议），SMTP(25)/POP3/IMAP（电子邮件协议）以及 FTP(21)（文件传输协议）Telnet(23)（远程登录协议）。

2．TCP 协议

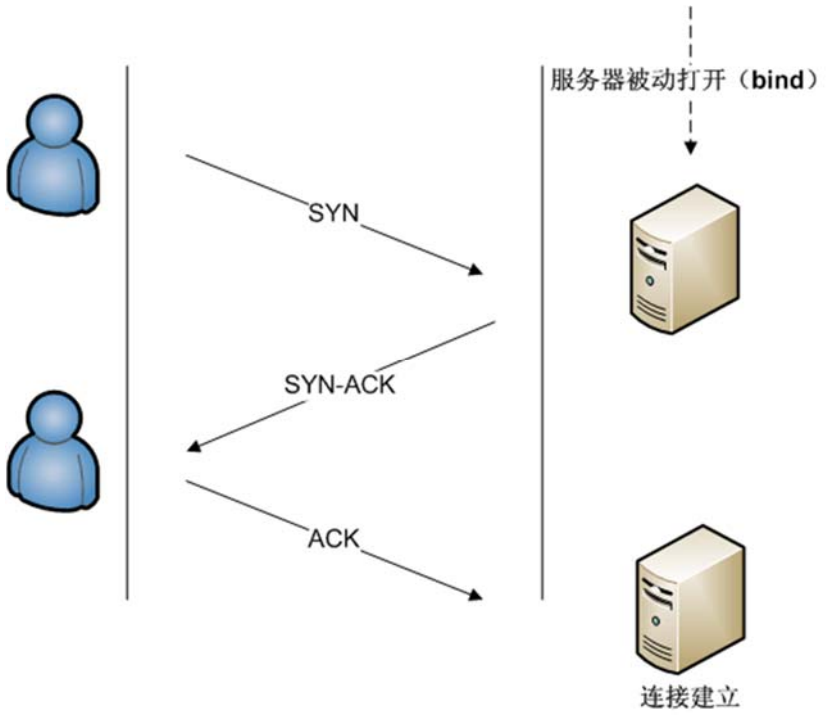
传输控制协议（Transmission Control Protocol, TCP）是一种面向连接的、可靠的、基于字节流的传输层（Transport layer）通信协议。在简化的计算机网络 OSI 模型中，它完成第四层传

输层所指定的功能，用户数据报协议（UDP）是同一层内另一个重要的传输协议。在因特网协议族（Internet protocol suite）中，TCP 层是位于 IP 层之上，应用层之下的中间层。不同主机的应用层之间经常需要可靠的、像管道一样的连接，但是 IP 层不提供这样的流机制，而是提供不可靠的包交换。

应用层向 TCP 层发送用于网间传输的、用 8 位字节表示的数据流，然后 TCP 把数据流分割成适当长度的报文段（通常受该计算机连接的网络的数据链路层的最大传输单元（MTU）的限制）。之后 TCP 把结果包传给 IP 层，由它来通过网络将包传送给接收端实体的 TCP 层。TCP 为了保证不发生丢包，就给每个包一个序号，同时序号也保证了传送到接收端实体的包的按序接收。然后接收端实体对已成功收到的包发回一个相应的确认（ACK）；如果发送端实体在合理的往返时延（RTT）内未收到确认，那么对应的数据包就被假设为已丢失将会被进行重传。TCP 用一个校验和（Checksum）函数来检验数据是否有错误；在发送和接收时都要计算校验和。

1. 通路的建立

TCP 用三路握手（three-way handshake）过程建立一个连接。在连接建立过程中，很多参数要被初始化，例如序号被初始化以保证按序传输和连接的强壮性。



一对终端同时初始化一个它们之间的连接是可能的。但通常是由一端打开一个接口（socket）然后监听来自另一方的连接，这就是通常所指的被动打开（passive open）。服务器端被被动打开以后，用户端就能开始建立主动打开（active open）。

(1)客户端通过向服务器端发送一个 SYN 来建立一个主动打开，作为三路握手的一部分。

(2)服务器端应当为一个合法的 SYN 回送一个 SYN/ACK。

(3)最后，客户端再发送一个 ACK。这样就完成了三路握手，并进入了连接建立状态。

2. 数据传输

在 TCP 的数据传送状态，很多重要的机制保证了 TCP 的可靠性和强壮性。它们包括：使用序号，对收到的 TCP 报文段进行排序以及检测重复的数据；使用校验和来检测报文段的错误；

使用确认和计时器来检测和纠正丢包或延时。如下图:

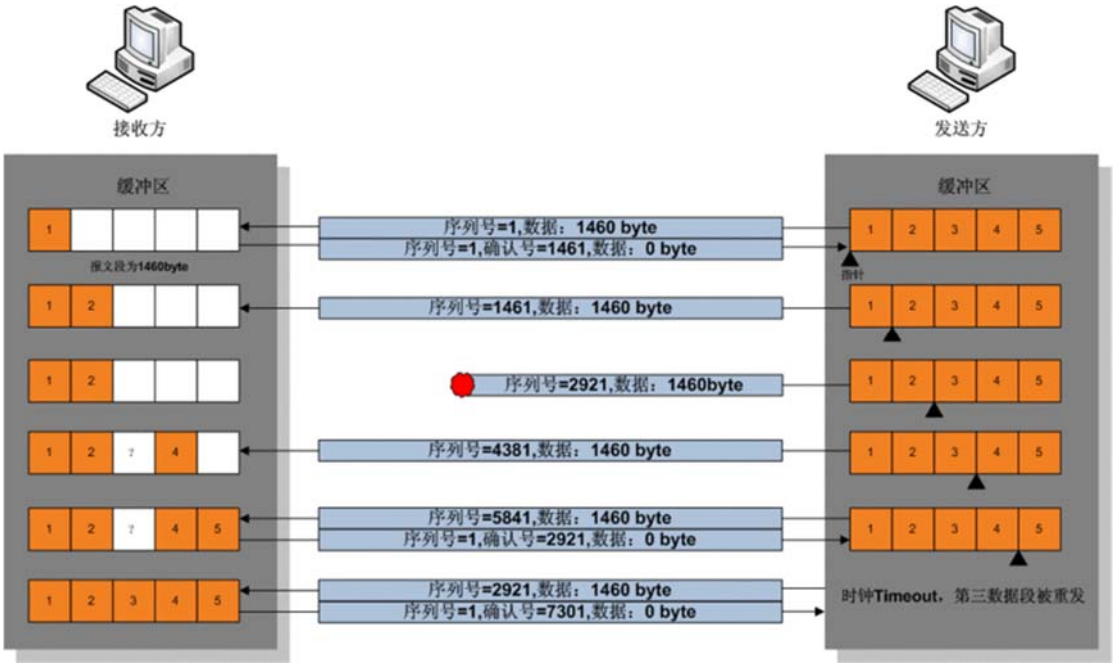
发送方首先发送第一个包含序列号为 1(可变化)和 1460 字节数据的 TCP 报文段给接收方。接收方以一个没有数据的 TCP 报文段来回复（只含报头），用确认号 1461 来表示已完全收到并请求下一个报文段。

发送方然后发送第二个包含序列号为 1461 和 1460 字节数据的 TCP 报文段给接收方。正常情况下，接收方以一个没有数据的 TCP 报文段来回复，用确认号 2921(1461+1460)来表示已完全收到并请求下一个报文段。发送接收这样继续下去.

然而当这些数据包都是相连的情况下，接收方没有必要每一次都回应。比如，他收到第 1 到 5 条 TCP 报文段，只需回应第五条就行了。在例子中第 3 条 TCP 报文段被丢失了，所以尽管他收到了第 4 和 5 条，然而他只能回应第 2 条。

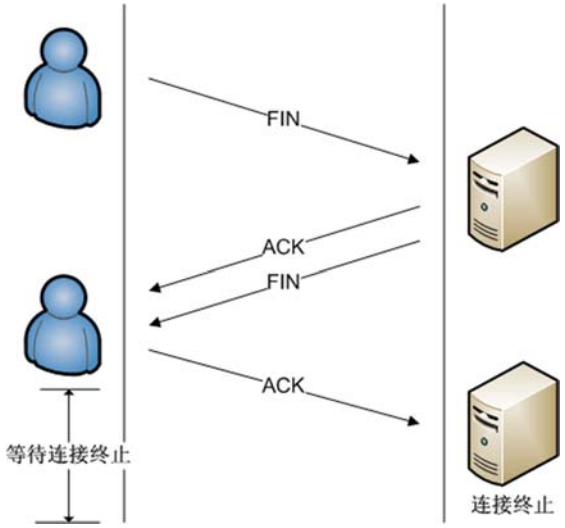
发送方在发送了第三条以后，没能收到回应，因此当时钟（timer）过时（expire）时，他重发第三条。（每次发送者发送一条 TCP 报文段后，都会再次启动一次时钟:RTT）。

这次第三条被成功接收，接收方可以直接确认第 5 条，因为 4，5 两条已收到。



3. 连接终止

连接终止使用了四路握手过程（four-way handshake），在这个过程中每个终端的连接都能独立地被终止。因此，一个典型的拆接过程需要每个终端都提供一对 FIN 和 ACK。



4. 拥塞控制

使用滑动窗实现 sliding window。

TCP 协议软件依靠滑动窗口机制解决传输效率和流量控制问题。它可以在收到确认信息之前发送多个数据分组。这种机制使得网络通信处于忙碌状态，提高了整个网络的吞吐率，它还解决了端到端的通信流量控制问题，允许接收端在拥有容纳足够数据的缓冲之前对传输进行限制。在实际运行中，TCP 滑动窗口的大小是可以随时调整的。收发端 TCP 协议软件在进行分组确认通信时，还交换滑动窗口控制信息，使得双方滑动窗口大小可以根据需要动态变化，达到在提高数据传输效率的同时，防止拥塞的发生。

5. SOKET 通讯与 TCP 原语

socket 工作过程如下:

服务器		客户端
Socket()		socket()
Bind()		↓
Listen()		↓
等待连接请求	←————请示连接 ————	connet()
accept()		↓
Recv()&Senc()	←————数据交互———→	Recv()&Send()
closesocket()		Closesocket()

服务器首先启动，通过调用 socket()建立一个套接字，然后调用 bind()将该套接字和本地网络地址联系在一起，再调用 listen()使套接字做好侦听的准备，并规定它的请求队列的长度,之后就调用 accept()来接收连接。客户在建立套接字后就可调用 connect()

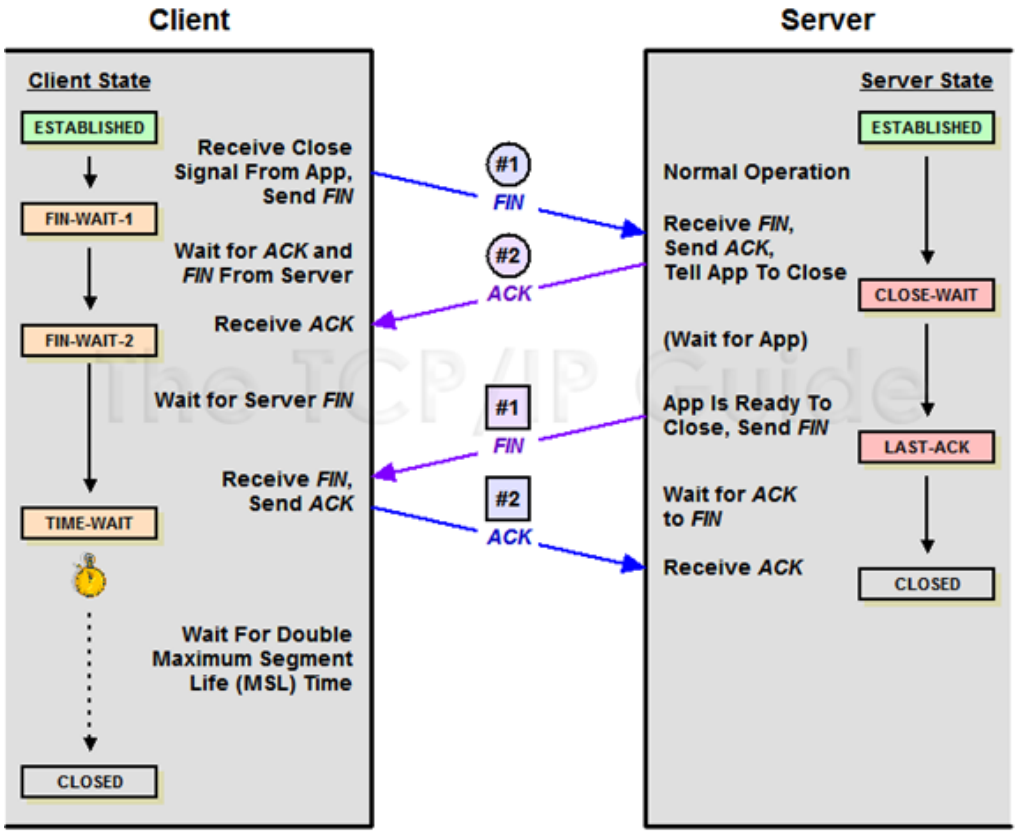
和服务器建立连接。连接一旦建立，客户机和服务器之间就可以通过调用 read()和 write()来发送和接收数据。最后，待数据传送结束后，双方调用 close()关闭套接字。

从而，conenct 是客户端使用的原语，而 listen accept bind 都是服务器端使用的原语。(创新工场 2014 校招)

在一个已经建立好的 TCP socket 连接中，发端调用 send()发五次，每次发 100 字节，问收端最少要 recv()收几次最多要收几次？(暴风 2014 校招)

综合数据传输部分，如果五次较短时间内同时到达，可以仅接收一次即可；TCP 是流式传输，如果过长，可以将信息拆分为若干段进行传输，接收时是从 buffer 中接收，视 buffer 的大小可以一次接受，buffer 最少 1 个字节，此时需要 500 次。

连接与断开状态示意图: http://robinjie.iteye.com/blog/289843



3. UDP 协议

用户数据报协议（User Datagram Protocol, UDP）是一个简单的面向数据报的传输层协议。

TCP/IP 模型中，UDP 为网络层以上和应用层以下提供了一个简单的接口。UDP 只提供数据的不可靠传递，它一但把应用程序发给网络层的数据发送出去，就不保留数据备份（所以 UDP 有时候也被认为是不可靠的数据报协议）。UDP 在 IP 数据报的头部仅仅加入了复用和数据校验（字段）checksum。

与 TCP 相比，UDP 是一种单向的数据传播，不需要应答，因而也不会有握手过程，不会建立特定的端到端的链接。UDP 不是一种可靠的数据传播，不确定发出的报文是否能够到达目的地；UDP 缺乏拥塞控制。

详见：https://zh.wikipedia.org/wiki/%E7%94%A8%E6%88%B7%E6%95%B0%E6%8D%AE%E6%8A%A5%E5%8D%8F%E8%AE%AE

4. 分组交换

分组交换（Packet switching）在计算机网络和通讯中是一种相对于电路交换的通信范例，分组（又称消息、或消息碎片）在节点间单独路由，不需要在传输前先建立通信路径。

分组交换是数据通信中一种新的且重要的概念，现在是世界上互联网通讯、数据和语音通信中最重要的基础。在此之前，数据通信是基于电路交换的想法，就像在传统的电话电路一样，在通话前先建立专有线路，通信双方要在电路的两端。

分组通过最佳路径(取决于 路由算法)路由到目标。但并不是所有在相同两个主机之间传送的分组（即使是来自同一消息的那些分组）一定要沿着相同的路径传送。一个数据连接通常传送数据的分组流，它们将不必全部以相同的方式路由过物理网络。目的计算机把收到的所有报文按照适当的顺序重新排列，就能合并恢复出原来的内容。

分组交换也可分为面向连接(Connection oriented)和无连接(Connectionless)传输。前者即**虚电路**方式(Virtual Path)，后者即**数据报**方式(Datagram)。

前者需要在发送端和接收端之间先建立一个逻辑连接，然后才开始传送分组，所有分组沿相同的路径进行交换转发，通信结束后再拆除该逻辑连接。网络保证所传送的分组按发送的顺序到达接收端。所以网络提供的服务是可靠的，也保证服务质量。每个分组头只需要标明虚电路标识符和序号，开销小，适合长报文传送。

后者每个分组按一定格式附加源与目的地址、分组编号、分组起始、结束标志、差错校验等信息，以分组形式在网络中传输。网络只是尽力地将分组交付给目的主机，但不保证所传送的分组不丢失，也不保证分组能够按发送的顺序到达接收端。所以网络提供的服务是不可靠的，也不保证服务质量。其优点是传输延时小，当某节点出现故障时不影响后续分组的传输。

5. HTTP 协议

1) HTTP 协议简介

From wiki: 超文本传输协定（HTTP）是网际网路上应用最为广泛的一种网路协议。设计 HTTP 最初的目的是为了提供一种发布和接收 HTML 页面的方法。通过 HTTP 或者 HTTPS 协议请求的资源由统一资源标识符（Uniform Resource Identifiers，URI）来标识。

HTTP 是一个客户端终端（用户）和服务器端（网站）请求和应答的标准。通过使用 Web 浏览器、网络爬虫或者其它的工具，客户端发起一个 HTTP 请求到服务器上指定端口（默认端口为 80）。我们称这个客户端为用户代理程式（user agent）。应答的服务器上存储着一些资源，比如 HTML 文件和图像。我们称这个应答服务器为源服务器（origin server）。在用户代理和源服务器中间可能存在多个中间层，比如代理，网关或者隧道。

通常，由 HTTP 客户端发起一个请求，建立一个到服务器指定端口（默认是 80 端口）的 TCP 连接。HTTP 服务器则在那个端口监听客户端的请求。一旦收到请求，服务器会向客户端返回一个状态，比如"HTTP/1.1 200 OK"，以及返回的内容，如请求的文件、错误消息、或者其它信息。

2) HTTP 协议方法

HTTP/1.1 协议中共定义了八种方法（也叫“动作”）来以不同方式操作指定的资源：

- OPTIONS:可使服务器传回该资源所支持的所有 HTTP 请求方法。用'*'来代替资源名称，向 Web 服务器发送 OPTIONS 请求，可以测试服务器功能是否正常运作。

- HEAD:与 GET 方法一样，都是向服务器发出指定资源的请求。只不过服务器将不传回资源的本文部份。它的好处在于，使用这个方法可以在不必传输全部内容的情况下，就可以获取其中“关于该资源的信息”(元信息或称元资料)。
- GET:向指定的资源发出“显示”请求。使用 GET 方法应该只用在读取资料，而不应当被用于产生“副作用”的操作中，例如在 Web Application 中。其中一个原因是 GET 可能会被网络蜘蛛等随意访问。
- POST:向指定资源提交数据，请求服务器进行处理（例如提交表单或者上传文件）。数据被包含在请求本文中。这个请求可能会建立新的资源或修改现有资源。
- PUT:向指定资源位置上传其最新内容。
- DELETE:请求服务器删除 Request-URI 所标识的资源。
- TRACE:回显服务器收到的请求，主要用于测试或诊断。
- CONNECT: HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。通常用于 SSL 加密伺服器的连结(经由非加密的 HTTP 代理伺服器)。

HTTP 服务器至少应该实现 GET 和 HEAD 方法，其他方法都是可选的。

3) HTTP 响应

所有 HTTP 响应的第一行都是状态行，依次是当前 HTTP 版本号，3 位数字组成的状态代码，以及描述状态的短语，彼此由空格分隔。

状态代码的第一个数字代表当前响应的类型：

1xx 消息——请求已被服务器接收，继续处理

2xx 成功——请求已成功被服务器接收、理解、并接受

3xx 重定向——需要后续操作才能完成这一请求

4xx 请求错误——请求含有词法错误或者无法被执行

5xx 服务器错误——服务器在处理某个正确请求时发生错误

例子:”404 not found”代码 404 的第一个“4”代表客户端的错误，如错误的网页位址；后两的数字码则代表着特定的错误讯息。HTTP 的三字元代码跟早期通讯协定 FTP 和 NNTP 的代码相当类似。

从 HTTP 的层面来看，404 讯息码之后通常会有一个可读的讯息“Not Found”，许多网路伺服器的预设页面也都有“404”代码跟“Not Found”的词汇。

404 错误讯息通常是在目标页面被更动或移除之后显现的页面。

4) 示例

下面是一个 HTTP 客户端与服务器之间会话的例子，运行于 www.google.com，端口 80

客户端请求：

```
GET / HTTP/1.1
Host:www.google.com
```

（末尾有一个空行。第一行指定方法、资源路径、协议版本；第二行是在 1.1 版里必带的一个 header 作用指定主机）

服务器应答：

```
HTTP/1.1 200 OK
Content-Length: 3059
Server: GWS/2.0
Date: Sat, 11 Jan 2003 02:44:04 GMT
Content-Type: text/html
Cache-control: private
Set-Cookie:
PREF=ID=73d4aef52e57bae9:TM=1042253044:LM=1042253044:S=SMCc_
HRPCQiqy
X9j; expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/;
domain=.google.com
Connection: keep-alive
```

（紧跟着一个空行，并且由 HTML 格式的文本组成了 Google 的主页）

6. IP 协议

IPv4 是一种无连接的协议，操作在使用分组交换的链路层上。此协议会尽最大努力交付分组，意即它不保证任何分组均能送达目的地，也不保证所有分组均按照正确的顺序无重复地到达。这些方面是由上层的传输协议（如传输控制协议）处理的。

1) IPV4

IPv4 地址可被写作任何表示一个 32 位整数值的形式，但为了方便，它通常被写作点分十进制的形式，即四个字节被分开用十进制写出，中间用点分隔。一共可以划分为 A~E 五类地址。D 类 E 类有特殊用途，其中前三类的地址分布如下：

类别	起始位	开始	结束	掩码
A	0	1.0.0.0	127.0.0.0	255.0.0.0
B	10	128.0.0.0	191.255.0.0	255.255.0.0
C	110	192.0.0.0	223.255.255.0	255.255.255.0

某些地址块不允许分配给主机，有专门的用途如 C 类地址中以.0 或.255 结尾的 IP,B 类地址中以.0.0 与.255.255 结尾的地址用于广播。以及 24 位块 10.0.0.0–10.255.255.255，20 位块 172.16.0.0–172.31.255.255，16 位块 192.168.0.0–192.168.255.255 均有特殊的用途。

2) 子网划分

具有相同的前半部分地址的一组 IP 地址，可以利用地址的前半部分划分组成子网。

子网掩码（subnet mask），它是一种用来指明一个 IP 地址的哪些位标识的是主机所在的子网以及哪些位标识的是主机的位掩码。如 192.0.2.96/28 表示的是一个前 28 位被用作网络号的 IP 地址（和子网掩码 255.255.255.240 的意思一样）

子网掩码的好处就是：不管网络有没有划分子网，只要把子网掩码和 IP 地址进行逐位的“与”运算（AND）。就立即得出网络地址来。这样在路由器处理到来的分组时就可以采用同样的方法。

IP 地址可以分为网络地址和主机地址两部分，而子网掩码即指令了两者之间的位数划分。子网的划分是一个将主机部分的若干位分配到网络部分的过程。子网划分是借助于取走主机位，把这个取走的部分作为子网位.因此这个意味划分越多的子网，每个子网容纳的主机将越少.

上图中有各类地址及其默认子网划分，可以得出网络地址与主机地址，当在该地址的基础上借用 1 位主机地址则可划分为 2 个子网，产生 2 个子网，每个子网有 126 个主机地址；相应的借用 2 位主机位，产生 4 个子网，每个子网有 62 个主机地址（除去.0 与.255）；例如：A 类地址 10.0.0.0，子网掩码：255.255.0.0 可以将其划分为 256 个子网（从 10.0.0.0 到 10.255.0.0）；

例如将对 B 类网络 135.41.0.0/16 需要划分为 20 个能容纳 200 台主机的网络（即：子网）。因为 16<20<32，即：2 的 4 次方<20<2 的 5 次方，所以，子网位只须占用 5 位主机位就可划分成 32 个子网，可以满足划分成 20 个子网的要求。B 类网络的默认子网掩码是 255.255.0.0。现在子网又占用了 5 位主机位，根据子网掩码的定义，划分子网后的子网掩码转换为十进制应该为 255.255.248.0。

现在我们再来看一看每个子网的主机数。子网中可用主机位还有 11 位，2 的 11 次方 =2048，去掉主机位全 0 和全 1 的情况，还有 2046 个主机 ID 可以分配，而子网能容纳 200 台主机就能满足需求，按照上述方式划分子网，每个子网能容纳的主机数目远大于需求的主机数目，造成了 IP 地址资源的浪费。

为了更有效地利用资源，我们也可以根据子网所需主机数来划分子网。还以上例来说，128<200<256，即 2^7<200<2^8，也就是说，在 B 类网络的 16 位主机位中，保留 8 位主机位，其它的 16－8=8 位当成子网位，可以将 B 类网络 138. 96.0.0 划分成 256（2^8）个能容纳 256－1－1=254 台（去掉全 0 全 1 情况）主机的子网。转换为十进制为 255.255.255.0。

7. ICMP

Internet Control Message Protocol 网络控制消息协议用于 TCP/IP 网络中发送控制消息，提供可能发生在通信环境中的各种问题反馈，通过这些信息，令管理者可以对所发生的问题作出诊断，然后采取适当的措施解决。

Ping: 用来测试数据包能否通过 IP 协议到达特定主机。ping 的运作原理是向目标主机传出一个 ICMPecho 要求封包，并等待接收 echo 回应封包。程式会按时间和成功响应的次数估算丢失封包率（丢包率）和封包往返时间。

Tracert: 现代 Linux 系统称为 tracepath，Windows 系统称为 tracert，是一种电脑网络工具。它可显示封包在 IP 网络经过的路由器的 IP 位址。

8. ARP 与 RARP

ARP(Address Resolution Protocol，地址解析协议)是获取物理地址的一个 TCP/IP 协议。某节点的 IP 地址的 ARP 请求被广播到网络上后，这个节点会收到确认 其物理地址的应答，这样的数据包才能被传送出去。RARP(逆向 ARP)经常在无盘工作站上使用，以获得它的逻辑 IP 地址。

五． 数据库

1. 主键/超键/候选键

主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。

超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

候选键：是最小超键，即没有冗余元素的超键。

2. ACID

数据库事务 transanction 正确执行的四个基本要素。ACID, Atomicity(原子性)、correspondence(一致性)、隔离性(isolation)、持久性(Durability)。

原子性:整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

一致性:在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。

隔离性:隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请求，使得在同一时间仅有一个请求用于同一数据。

持久性:在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

由于一项操作通常会包含许多子操作，而这些子操作可能会因为硬件的损坏或其他因素产生问题，要正确实现 ACID 并不容易。ACID 建议数据库将所有需要更新以及修改的资料一次操作完毕，但实际上并不可行。

目前主要有两种方式实现 ACID：第一种是 Write ahead logging，也就是日志式的方式。第二种是 Shadow paging。

3. 数据库范式

数据库规范化，又称数据库或资料库的正规化、标准化，是数据库设计中的一系列原理和技术，以减少数据库中数据冗余，增进数据的一致性。

现在数据库设计最多满足 3NF，普遍认为范式过高，虽然具有对数据关系更好的约束性，但也导致数据关系表增加而令数据库 IO 更易繁忙，原来交由数据库处理的关系约束现更多在数据库使用程序中完成。

1NF: 如果关系模式中所有属性的值域内每一个值都不可分，则称其符合第一范式；要求每条记录都只能存放单一值，而且每笔记录都用主键来加以识别。

2NF: 在满足第一范式的基础上，所有记录都要和主键有完全依赖关系；如果有记录只和主键的一部份有关，就得把它们独立出来变成另一个资料表。如果一个资料表的主键只有单一一个栏位的话，它就一定符合第二正规化。

3NF: 在满足第二范式的基础上，要求每个非主属性都不传递依赖 R 的候选键；即

要求所有的非键属性互相之间应该是无关的。

BCNF: 在第三范式的基础上要求，任何属性（包括非主键和主键）都不能被非主键所决定。

4. 数据库中的基本语句

LIMIT 子句可以被用于强制 SELECT 语句返回指定的记录数。LIMIT 接受一个或两个数字参数。参数必须是一个整数常量。如果给定两个参数，第一个参数指定第一个返回记录行的偏移量，第二个参数指定返回记录行的最大数目。初始记录行的偏移量是 0(而不是 1);

SELECT * FROM table LIMIT 5,10; // 检索记录行 6-15

5. 游标

可以把游标当做一个指针，用于定位结果集中的行，通过判断全局变量 @@FETCH_STATUS 可以判断其是否到了最后。

6. 索引

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。建立索引的目的是加快对表中记录的查找或排序。为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。

数据库索引可以分为，聚集索引与非聚集索引。索引键值的逻辑顺序与索引所服务的表中相应行的物理顺序相同的索引，被称为聚集索引，反之为非聚集索引，索引一般使用二叉树排序索引键值的，聚集索引的索引值是直接指向数据表对应元组的，而非聚集索引的索引值仍会指向下一个索引数据块，并不直接指向元组，因为还有一层索引进行重定向，所以非聚集索引可以拥有不同的键值排序而拥有多个不同的索引。而聚集索引因为与表的元组物理顺序一一对应，所以只有一种排序，即一个数据表只有一个聚集索引。

Hash 索引与 B-tree 索引，Hash 索引结构的特殊性，其检索效率非常高，索引的检索可以一次定位，不像 B-Tree 索引需要从根节点到枝节点，最后才能访问到页节点这样多次的 IO 访问，所以 Hash 索引的查询效率要远高于 B-Tree 索引。Hash 索引仅仅能满足"=","IN"和"<=>"查询，不能使用范围查询；由于 Hash 索引比较的是进行 Hash 运算之后的 Hash 值，所以它只能用于等值的过滤，不能用于基于范围的过滤，因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证和 Hash 运算前完全一样。Hash 索引无法被用来进行数据的排序操作；由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值，而且 Hash 值的大小关系并不一定和 Hash 运算前的键值完全一样，所以数据库无法利用索引的数据来进行任何排序运算。

B-Tree 索引是 MySQL 数据库中使用最为频繁的索引类型。一般来说， MySQL 中的 B-Tree 索引的物理文件大多都是以 Balance Tree 的结构来存储的，也就是所有实际需要的数据都存放于 Tree 的 Leaf Node ，而且到任何一个 Leaf Node 的最短路径的长度都是完全相同的，所以我们大家都称之为 B-Tree 索引。当然可能各种数据库（或 MySQL 的各种存储引擎）在存放自己的 B-Tree 索引的时候会对存储结构稍作改造。如 Innodb 存储引擎的 B-Tree 索引实际使用的存储结构实际上是 B+Tree ，也就是在 B-Tree 数据结构的基础上做

了很小的改造，在每一个 Leaf Node 上面出了存放索引键的相关信息之外，还存储了指向与该 Leaf Node 相邻的后一个 LeafNode 的指针信息，这主要是为了加快检索多个相邻 Leaf Node 的效率考虑。在 Innodb 存储引擎中，存在两种不同形式的索引，一种是 Cluster 形式的主键索引（ Primary Key ），另外一种则是和其他存储引擎（如 MyISAM 存储引擎）存放形式基本相同的普通 B-Tree 索引，这种索引在 Innodb 存储引擎中被称为 Secondary Index。

7. 语句

1) 更新数据库表的第 n 条记录该怎么写 sql？比如更新名为 table 的表中的第 100 条记录，将其 colunm1 域更新为 aaa,如何写这个 sql 呢？数据库是 oracle 的。

   update table set column1 ='aaa' where table.id = (select id from table limit 99,1)

2) Where 与 Having 的区别

   WHERE 在分组和聚集计算之前选取输入行（因此，它控制哪些行进入聚集计算），而 HAVING 在分组和聚集之后选取分组的行。因此，WHERE 子句不能包含聚集函数；因为试图用聚集函数判断那些行输入给聚集运算是没有意义的。相反，HAVING 子句总是包含聚集函数。

3) Delete 与 truncate 的区别

   delete 从已经建立的表中删除行数据；如果要删除整列需要使用 Update;而且删除表时，删除的是表的内容而不是表本身；如果要删除表，使用 Truncate Table 可以更快，实际上它会删除原来的表并重新创建一个表，而非逐行删除表中的数据。

8. 内连接与外连接

join_type 指出连接类型，可分为三种：内连接、外连接和交叉连接。内连接(INNER JOIN)使用比较运算符进行表间某(些)列数据的比较操作，并列出这些表中与连接条件相匹配的数据行。根据所使用的比较方式不同，内连接又分为等值连接、自然连接和不等连接三种。外连接分为左外连接(LEFT OUTER JOIN 或 LEFT JOIN)、右外连接(RIGHT OUTER JOIN 或 RIGHT JOIN)和全外连接(FULL OUTER JOIN 或 FULL JOIN)三种。与内连接不同的是，外连接不只列出与连接条件相匹配的行，而是列出左表(左外连接时)、右表(右外连接时)或两个表(全外连接时)中所有符合搜索条件的数据行。

9. 视图

视图是虚拟的表，与包含数据的表不一样，视图只包含使用时动态检索数据的查询；不包含任何列或数据。使用视图可以简化复杂的 sql 操作，隐藏具体的细节，保护数据；视图创建后，可以使用与表相同的方式利用它们。

视图不能被索引，也不能有关联的触发器或默认值，如果视图本身内有 order by 则对视图再次 order by 将被覆盖。

创建视图：create view XXX as XXXXXXXXXXXXX;

对于某些视图比如未使用联结子查询分组聚集函数 Distinct Union 等，是可以对其更新的，对视图的更新将对基表进行更新；但是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。

六． 算法及智力题目

1. 小白鼠试毒问题及扩展

(MS2013intern)1000 个瓶子水里有 1 瓶有毒，任何生物喝下任意剂量的毒后 7 天后死亡，现有若干只小白鼠，问至少需要多少只才能知道哪瓶水有毒？（MS2013）

2 的 10 次方=1024，现在先将老鼠排成一列，做一个数列，1 表示老鼠是活的，0 表示老鼠是死地，做完实验之后这十只老鼠的状态可以有 1024 种，现在从结果来推断实验方法。因为做实验，我们不算没有老鼠死亡的那一种情况。

把 1000 瓶水编号：0~999，再转化成二进制数：0000000000～1111101000，每个二进制数都是唯一的，且与每个瓶子一一对应。找到有毒的瓶子 == 找到有毒瓶子的二进制编号 == 确定有毒瓶子的二进制编号的每一位是 0 还是 1。在该十位数中，将第 i 位编号为 1 的药水喂给第 i 只小白鼠，如果该小白鼠死去，说明该位为 1，否则为 0。最后根据 10 个小白鼠的存活情况可以确定出药水的编号。

另： 如果你有两个星期的时间（换句话说你可以做两轮实验），为了从 1000 个瓶子中找出毒药，你最少需要几只老鼠？

7 只老鼠就足够了。事实上，7 只老鼠足以从 3^7 = 2187 个瓶子中找出毒药来。首先，把所有瓶子从 0 到 999 编号，然后全部转换为 7 位三进制数。现在，让第一只老鼠喝掉所有三进制数右起第一位是 2 的瓶子，让第二只老鼠喝掉所有三进制数右起第二位是 2 的瓶子，等等。一星期之后，如果第一只老鼠死了，就知道毒药瓶子的三进制编号中，右起第一位是 2；如果第二只老鼠没死，就知道毒药瓶子的三进制编号中，右起第二位不是 2，只可能是 0 或者 1……也就是说，每只死掉的老鼠都用自己的生命确定出了，三进制编号中自己负责的那一位是 2；但每只活着的老鼠都只能确定，它所负责的那一位不是 2。于是，问题就归约到了只剩一个星期时的情况。在第二轮实验里，让每只活着的老鼠继续自己未完成的任务，喝掉它负责的那一位是 1 的所有瓶子。再过一星期，毒药瓶子的三进制编号便能全部揭晓了。**//确定三进制编号**

另： n 只小白鼠和 t 周的时间可以从(t+1)^n 个瓶子中检验出毒药（一瓶）来。（是否是“至多”(t+1)^n 瓶希望你们能想个法子证明或证伪：P）(信息论的观点即可解答)

2. 天平寻找次品球问题及扩展

(MS2013)N 个球中有一个是次球， 已知该球比其他球都重或轻，现有一架天平，问称三次可以找出此球，则 N 可以为多少？（A12 B 16 C20 D24）

A 分为三份，每份为 4，如果 1 与 2 平衡，则次球在 3 中，否则在 1 或 2 中。而后将 4 个球分为 2 份，再次锁定范围。再次 2 个球称即可。

B 可分为 6/6/4 三份，如果在 6 中，再分为 3/3，再分为 1/1，如平衡则剩下的即是，否则即可看出。

C 可分为 9/9/2 三份，如在 9 中，则分为 3/3/3 而后再称一次 1/1 即可

D 可分为 9/9/6 三份。

推广：有 n 个球，其中 1 个球是次品，质量比其他球都要重。现在有一个没有砝码的天平，

要求称 t 次，将次品球挑出。请问能否做到？

将 n 个球分别编号为 1-n。 我们首先来分析结果的可能性数目（以下简称判断）。共有 n 种，即从 1 重、2 重一直到 n 重。而每一次称，都会出现三种结果：左重，左轻，左右等重。则 t 次之后，共至多会出现 3^t 种情形。我们所要做的，就是合理放置球，使得每一种判断都有一个情形对应，且一个情形至多对应一种判断。

因此当 3^t>=n 时，可以做到，否则不能。因此当 n<=3^t 时有解；该问题有解。如取三次，则至多会有 27 个球。

考虑到题设，每次比较在天平两端上球的数目必须相同。我们要做的就是找到一个划分，使得三次之内可以找到结果。

推广：有 12 个球，其中 1 个球是次品，质量与其它球不同（不知轻重）。现在有一个没有砝码的天平，要求称 3 次，将次品球挑出，请问应当如何操作。

将 12 个球分别编号为 1-12。可见共有 1 重、2 重、.....12 重、1 轻、2 轻、.....12 轻，共 24 种判断，而 3 次称量，共会产生 27 种情形，27>24，所以本题有解。接着考虑左右各放几个球，空闲几个球。设各方 a 个球，空闲 m 个球，由于当天平两端等重时，次品球必在，m 个球中，则有 2m 种判断。而剩下 2 次称量机会，即至多有 9 种情形，所以 2m<=9，m 至多为 4。当天平 2 边不等重时，由于对称等价，不妨设左边重，则有左边 a 球重和右边 a 球轻共 2a 种判断。2a<=9，所以 a 至多为 4。又 2a+m=12，所以 a=m=12，两边各放 4 球，空闲 4 球。

分别为 a b c d, e f g h, i j k l，取出 abcd, efgh

第一种情形：

如果重量相等，则说明所求在 ijkl 中，称量 i j ，

如果相等，比较 a k ，如果 a=k，则所求为 l ；如果 ak 不等，则所求为 k 。

如果不等，比较 a i ，如果 a=i，则所求为 j ；如果不等，则所求为 i 。

第二种：

如果 abcd 轻，在 efgh 中取出 fgh ，替掉 abcd 中 bcd，从 ijkl 中取出 ijk 个放入 e 中填补空位：如果 afgh 轻：则说明所求在 a 或 e，拿 e 和除 a 以外的任意一球比较，如果重量相等，则所求的球是 a ；如果不等，则所求的球是 e 。

如果 afgh 重：说明所求在 fgh 中，且所求较重；比较 f g ，等重则所求为 h ；不等则重的为所求。如果一样重：说明所求在 bcd 中，且所求较轻；以下同 afgh 重的情形。

第三种：

如果 abcd 重，在 efgh 中取出 fgh ，替掉 abcd 中 bcd，从 ijkl 中取出 ijk 个放入 e 中填补空位：如果 afgh 重：则说明所求在 a 或 e，拿 e 和除 a 以外的任意一球比较，如果重量相等，则所求的球是 a ；如果不等，则所求的球是 e 。

如果 afgh 轻：说明所求在 fgh 中，且所求较轻；比较 f g ，等重则所求为 h ；不等则重的为所求。如果一样重：说明所求在 bcd 中，且所求较重；以下同 afgh 轻的情形。

结论：K 次知道轻重可以从 3^K 个球中找出不同的球出来，如果不知道轻重就只能从（3^K-1）/2 个球中找出不同的球出来(仍然可从信息论角度直接求解)

3. 抽扑克牌问题

(Nokia2013)一副扑克牌有 52 张，最上面一张是红桃 A，如果每次把最上面的 10 张移到最下面而不改变它们的顺序及朝向，那么，至少经过()次移动，红桃 A 会出现在最上面。

把从第一次移动 10 张到若干次移动后，红桃 A 再次出现在最上面看做是一个周期；可以先求出这一个周期一共移动了多少张牌，也就是求出 52 和 10 的最小公倍数，由此即可解决问题．52 和 10 的最小公倍数是：2×13×5=260；260÷10=26（次）；

4. 三密码锁问题

一个三密码锁坏了，且也不知密码。但只要每次调对其中两个数字就可打开，问最多调多少次可以打开这把锁？(世通量化基金)

三个密码锁结果一共可能有 1000 种，而最终成功的可能有 298 种。如果只试两个格，则 100 次一定可以打开。然而密码锁的成功率约 3%，故一定可以小于 100 打开。此题可以转化为多少个三位数可以包含任意三位数中的至少两位？

视 1000 组密码为 1000 个堆积成正方体的小方块，其编号由顶点坐标编为(x,y,z)，其中正确密码(X,Y,Z)向三个方向投影，只要能确保选择的密码所对应的方块至少一次落在这个投影内就可以了。

```
4|4 0 1 2 3
3|3 4 0 1 2

x2|2 3 4 0 1
↑1|1 2 3 4 0
 0|0 1 2 3 4
 ------------------
  0 1 2 3 4
        →y
```

表内是 z 的值。

从这个表可以看出，将大正方体均分为 8 个小正方体，每一个小正方体含有 125 个小方块(x,y,z)，则以上表格所选出的小方块是包含在 125 个同一个小正方体内的，且 x、y、z 其中两个的取值固定时，另一个有唯一取值。这样就能保证这 25 组(x,y,z)所规定的小方块无论从正面、侧面、上面看，投影都恰好覆盖 25 格，也就是 1/4 个侧面。所以这样的三方向投影刚好能够覆盖 4 个小正方体，也就是大正方体的一半。另外半个大正方体就是由另一个矩阵的(x,y,z)解决。

图2

000	101	202	303	404					
011	112	213	314	410					
022	123	224	320	421					
033	134	230	331	432					
044	140	241	342	443					
					555	656	757	858	959
					566	667	768	869	965
					577	678	779	875	976
					588	689	785	886	987
					599	695	796	897	995

5. 猜数字问题

1-20 的两个数把和告诉 A,积告诉 B，A 说不知道是多少，B 也说不知道，这时 A 说我知道了，B 接着说我也知道了，那么这两个数是多少？(Tencent2013)

首先假设两个数可以相等，即没有限制。A 说不知道说明无法推知结果，故不可能为（2=1+1,3=1+2,39=10+19,40=20+20），两个数的和 S=[4,38];B 说不知道结果，说明乘积 M 不是质数或者分解唯一的数（1,2,3,5,7,11,13,17,19）。A 由 B 不知道而得出结果说明 S 只有一种非质数的分解方式。

如 4 = 2+2 = 1+3；但是如果是 1+3 B 可以直接猜到结果。故为 2 和 2。

如 5 = 1+4 = 2+3；1*4 =4 与 2*3 =6 均不能使 B 得到结果。

A 知道结果后 B 以 A 的观点来看也可得到结果。

如两个数不相等，则 S 的若干种分解中只有一种无法使 B 直接得到结果。S =[5,37];5=1+4=2+3;1*4 = 4 只有一种分解方式，故可以为 2 和 3。

6=1+5=2+4；同上，可以为 2 和 4。

6. 最大连续子序列问题

有一个环状数列，找出 其和为最大的连续子序列。用 C 或 C++实现算法并分析复杂度

此题是对线性数列最大连续子序列的扩展，对于最大连续子序列的求解可以有 O(N^2) O(NlogN)及 O(N)三种实现方式。

该问题还可以通过分治法来求解，最大连续子序列和要么出现在数组左半部分，要么出现在数组右半部分，要么横跨左右两半部分。因此求出这三种情况下的最大值就可以得到最大连续子序列和。此时时间复杂度为 O(NlogN)

```
1.  int maxsequence2(int a[], int l, int u)
2.  {
3.      if (l > u) return 0;
4.      if (l == u) return a[l];
5.      int m = (l + u) / 2;
6.
7.      /*求横跨左右的最大连续子序列左半部分*/
8.      int lmax=a[m], lsum=0;
9.      for (int i=m; i>=l; i--) {
10.         lsum += a[i];
11.         if (lsum > lmax)
12.             lmax = lsum;
13.     }
14.
15.     /*求横跨左右的最大连续子序列右半部分*/
16.     int rmax=a[m+1], rsum = 0;
17.     for (int i=m+1; i<=u; i++) {
18.         rsum += a[i];
19.         if (rsum > rmax)
20.             rmax = rsum;
21.     }
22.     return max3(lmax+rmax, maxsequence2(a, l, m), maxsequence2(a, m+1, u)); //返回三者最大值
23. }
24.
25. /*求三个数最大值*/
26. int max3(int i, int j, int k)
27. {
28.     if (i>=j && i>=k)
29.         return i;
30.     return max3(j, k, i);
31. }
```

还有一种更好的解法，只需要 O（N）的时间。因为最大连续子序列和只可能是以位置 0～n-1 中某个位置结尾。当遍历到第 i 个元素时，判断在它前面的连续子序列和是否大于 0，如果大于 0，则以位置 i 结尾的最大连续子序列和为元素 i 和前面的连续子序列和相加；否则，则以位置 i 结尾的最大连续子序列和为元素 i。

```
1.  int maxsequence3(int a[], int len)
2.  {
3.      int maxsum, maxhere;
4.      maxsum = maxhere = a[0];   //初始化最大和为 a【0】
5.      for (int i=1; i<len; i++) {
6.          if (maxhere <= 0)
7.              maxhere = a[i];  //如果前面位置最大连续子序列和小于等于 0，则以当前位置 i 结尾的最大连续子序列和为
    a[i]
8.          else
9.              maxhere += a[i]; //如果前面位置最大连续子序列和大于 0，则以当前位置 i 结尾的最大连续子序列和为它们两
    者之和
10.         if (maxhere > maxsum) {
11.             maxsum = maxhere;  //更新最大连续子序列和
12.         }
13.     }
14.     return maxsum;
15. }
```

在环形数组情况下，从环状数组中任一点 A，从 0 开始编号，假设环状数组中，和最大的连续子序列，是从下标 i 开始，到下标 j(不包括 j)结束。
若 i < j, 则可以从 A 点前面断开，问题转为“求普通数组的最大连续子序列和”。若 i > j，由于:　所有元素和 = i->j 的子序列和 + j->i 的子序列和，求“i->j 的子序列和最大” 等价于求 “j->i 的子序列和最小”。即“求普通数组的最小连续子序列和”。

```
1.  int max_ring_continuous_sum(const int arr[], size_t len)
2.
3.  {
4.    assert(arr && len > 0);
5.    int sum = arr[0];
6.    int cur_min_sum = sum, min_sum = sum;
7.    int cur_max_sum = sum, max_sum = sum;
8.    for (size_t i = 1; i < len; ++i) {
9.      const int value = arr[i];
10.     sum += value;
11.     cur_max_sum = max(cur_max_sum + value, value);
12.     max_sum     = max(cur_max_sum, max_sum);
13.     cur_min_sum = min(cur_min_sum + value, value);
14.     min_sum     = min(cur_min_sum, min_sum);
15.   }
16.   return max(max_sum, sum - min_sum);
17. }
```

7. 优惠券问题

一个饭店发行一套优惠券，一套里面总共有 n 张不同的优惠券，顾客每次吃一次，可以随机获得一张优惠券。如果收集齐一套，下次吃饭可以打折。请问：顾客要来多少次才能收集齐一套优惠券？

假设顾客第一次来，他一定会得到一张独一无二的优惠券，第二次来的时候，得到的优惠券和上次不重复的概率是 (n-1)/n， 第三次来的时候，得到与上两次不同的概率是(n - 2)/n，…. ，到第 n 次来的时候，与前 n - 1 次收集到的优惠券不同的概率是 1/n。

换句话说，拿到第一张不重复的优惠券需要的次数是 1， 拿到第二张与前一张不同的优惠券需要的次数是 n/(n-1)，拿到第三张与前两张不同的优惠券需要的次数是 n/(n - 2)，以此类推，拿到最后一张不重复的优惠券所需要的次数是 n/1.
所以，总的次数是 1 + n/(n-1) + n/(n-2) + n/(n -3) + ... + n ~= nlgn.

8. 间隔翻眼镜问题

一百个眼镜，摆成一个圈，全部正面向上，第一个人将每个翻动一次，一共翻了 100 次；第二个人从 no.2 开始隔一个翻一次，也翻 100 次；第 3 个人从 no.3 开始隔两个翻一次，翻 100 次，问 100 个人之后，多少眼镜正面向上?

第一次沿顺时针将所有眼镜翻转,而第 99 次从第 99 个眼镜开始每隔 98 个翻转一次,则按逆时针将所有眼镜翻转一次。类似的，第 K 次与第 100-K 次的效果可以抵消。
故最终的结果相当于只有第 100 个人每隔 99 个翻一次翻 100 次，相当于将 100 翻 100 次，故最终还是有 100 个眼镜朝上。

9. 扔鸡蛋确定楼层问题

300 层楼，3 个一样的小球，设计一个策略，得到小球摔碎的临界层数，并且要求最坏情况下所试次数最少。与程序员面试宝典中扔围棋问题类似。

经典的扔鸡蛋问题，只不过现在有三个。解题思路一样的，都是动态规划。

我们可以将这样的问题简记为 W(n,k)，其中 n 代表可用于测试的鸡蛋数，k 代表被测试的楼层数。对于问题 W(2,36)我们可以如此考虑，将第 1 颗鸡蛋，在第 i 层扔下（i 可以为 1～k 的任意值），如果碎了，则我们需要用第 2 颗鸡蛋，解决从第 1 层到第 i-1 层楼的子问题 W(1,i-1)，如果这颗鸡蛋没碎，则我们需要用这两颗鸡蛋，解决从 i+1 层到第 36 层的子问题 W(2,36-i)，解决这两个问题，可以分别得到一个尝试次数 p，q，我们取这两个次数中的较大者(假设是 p)，与第 1 次在 i 层执行测试的这 1 次相加，则 p+1 就是第一次将鸡蛋仍在 i 层来解决 W(2，36)所需的最少测试次数次数 ti。对于 36 层楼的问题，第一次，我们可以把鸡蛋仍在 36 层中的任何一层，所以可以得到 36 中解决方案的测试次数 T{t1,t2,t3,……,t36}，在这些结果中，我们选取最小的 ti，使得对于集合 T 中任意的值 tj(1<=j<=36,j!=i)，都有 ti<=tj，则 ti 就是这个问题的答案。

用公式来描述就是 W(n, k) = 1 + min{max(W(n -1, x -1), W(n, k - x))}, x in {2, 3, ……, k}，其中 x 是第一次的测试的楼层位置。

```
1.  unsigned int DroppingEggsPuzzle(unsigned int eggs, unsigned int floors)
2.  {
3.      unsigned int i, j, k, t, max;
4.
5.      unsigned int temp[eggs + 1][floors + 1];
6.
7.      for(i = 0; i < floors + 1; ++i)
8.      {
9.          temp[0][i] = 0;
10.         temp[1][i] = i;
11.     }
12.
13.     for(i = 2; i < eggs + 1; ++i)
14.     {
15.         temp[i][0] = 0;
16.         temp[i][1] = 1;
```

```
    }

    for(i = 2; i < eggs + 1; ++i)
    {
        for(j = 2; j < floors + 1; ++j)
        {
            for(k = 1, max = UINT_MAX; k < j; ++k)
            {
                t = temp[i][j - k] > temp[i - 1][k -1] ?  temp[i][j - k] : temp[i - 1][k -1];

                if(max > t)
                {
                    max = t;
                }
            }

            temp[i][j] = max + 1;
        }
    }

    return temp[eggs][floors];
}
```

该算法的空间复杂度是 O(nk)，时间复杂度是 O(nk^2)，对于规模较大的问题，无论是空间还是时间复杂度都很可观。

这里引出了一个问题：n 个鸡蛋，测试 m 次(简记为 D(n,m))，最大可以解决几层楼的问题，通过对递推结果表格的观察，我们可以得到如下结论

D(1,m) = m

D(n,n) = 2^n - 1

D(n,m){m <= n} = D(m,m)

D(n,m){n > m} = D(n -1,m-1) + 1 + D(n,m-1)

```
unsigned int DroppingMax(unsigned int eggs, unsigned times)
{
    if(eggs == 1)
    {
        return times;
    }

    if(eggs >= times)
    {
        return (unsigned int)pow(2, times) - 1;
    }

    return DroppingMax(eggs, times -1) + DroppingMax(eggs -1, times - 1) + 1;
}
unsigned int DroppingMax(unsigned int eggs, unsigned times)
{
    if(eggs == 1)
    {
        return times;
    }

    if(eggs >= times)
    {
        return (unsigned int)pow(2, times) - 1;
    }

    return DroppingMax(eggs, times -1) + DroppingMax(eggs -1, times - 1) + 1;
}
```

又：一百层高楼和两个棋子,求棋子撞碎的临界高度。

假设第一颗在第 c1 层测试，有两种可能：

1) 摔碎

2) 没摔碎

对于情况 1，说明 X 在 1 到 c1 之间，用递增法，c1 - 1 次可测得。加上第一次，共测试 c1 次。对于情况 2，说明 X > c1，于是进行第二轮。

【第二轮】

对于情况 2，用第一颗在第 c2 层测试（c2 > c1），有两种可能：

2.1) 摔碎

2.2) 没摔碎

对于情况 2.1，说明 X 在 c1 + 1 到 c2 之间，用递增法，c2 - c1 - 1 次可测得。加上前两次，共测试 c2 - c1 + 1 次。对于情况 2.2，说明 X > c2，进行第三轮。

【第三轮】

对于情况 2.2，用第一颗在第 c3 层测试（c3 > c2），仍有两种可能：

2.2.1) 摔碎

2.2.2) 没摔碎

对于情况 2.2.1，说明 X 在 c2 + 1 到 c3 之间，用递增法，c3 - c2 - 1 次可测得。加上前三次，共测试 c3 - c2 + 2 次。对于情况 2.2.2，说明 X > c3，进行第四轮。

……

以此类推下去，直到第 n 轮 c(n) >= 总楼层数（即 100），这时候没摔碎的情况已经不会出现。测试结束。

总测试次数为：

```
max {
        c1,
        c2 - c1 + 1,
        c3 - c2 + 2,
        c4 - c3 + 3,
        ...
        c(n) - c(n-1) + (n-1)
    }
```

题目中说，求最少多少次能够测出，在这里也就是说，如何让这个最大值最小化。观察这个数列（c1, c2 - c1 + 1, c3 - c2 + 2,….）你会发现前一项与后一项的关系，如果前一项特别大的话，就会是后一项特别小。那么为了使最大值比较小的话，就要让这些项都相等，于是

c2 = 2 * c1 - 1

c3 = c1 + c2 - 2 = 3 * c1 - 3

c4 = c1 + c3 - 3 = 4 * c1 - 6

…

c(n) = n * c1 - n(n-1)/2

以上，即保证在 n 轮 c1 次测试中，必能穷尽 c(n) 层楼。

现在，我们要求测试次数最少，也就是求 c1 的最小值（每一项都是 c1 的增函数，如果 c1 最小，函数也会最小）。

因为 c(n) >= 100

所以 n * c1 - n(n-1)/2 >= 100

即 c1 >= 100/n + (n-1)/2 = 100/n + n/2 - 1/2 >= 2 * sqrt(100/n * n/2) - 1/2 =

13.6 （以上用到均值不等式）
因此 c1 的最小值为 14，即最少需要测试 14 次。
分别在 14, 27, 39, 50, 60, 69, 77, 84, 90, 95, 99, 100 层进行 12 轮测试。
具体解释:

对任意自然数 n, 以及满足 c1 < c2 < ... < c(n-1) < c(n) = 100 的任意 n 个自然数 c1,c2,...c(n), 定义多元函数

f(n,c1,c2,...c(n)) =
max {
c1,
c2 - c1 + 1,
c3 - c2 + 2,
c4 - c3 + 3,
...
c(n) - c(n-1) + (n-1)
}求 f 的极小值

只是在后面求此极小值的过程让楼主有疑问.我在这里试着把求值过程更形式化一些,希望能解决楼主的疑问.

由 f 的定义知:

f >= c1
f >= c2 - c1 + 1
f >= c3 - c2 + 2,
f >= c4 - c3 + 3,
...
f >= c(n) - c(n-1) + (n-1)

把上面 n 个式子相加,得:

n*f >= c(n) + 1 + 2 + 3 + ... + (n-1)
= 100 + 1 + 2 + 3 + ... + (n-1)
= 100 + n(n-1)/2

因此, f >= 100/n + (n-1)/2 = 100/n + n/2 - 1/2 >= 2 * sqrt(100/n * n/2) - 1/2 = 13.6

另：这题可以转化为信息论来解，两颗碎棋子，表示 100 种不同的状态，其中存在碎 1 颗和碎 2 颗的情况，求最短的编码长度。

碎了为 1，不碎为 0，这样的话，100 层楼就可以转化为 100 种不同的 01 编码，编码中最多允许有 2 个 1，也可以只有一个 1，比如 14 层碎了，然后从 1 试到 13 都没有碎，编码就是 10000000000000。对应的是 14 楼。

即：C(n,1) + C(n,2) >= 100，其中 C(n,1)表示碎 1 颗的情况，C(n,2)表示碎 2 颗的情况。

n + n*(n-1)/2 >= 100
=>
n(n+1) >= 200
=>
n >= 14

另有人提出用这种信息编码的方式考虑这道题，可以把问题扩展到 m 层楼，n 个棋子，而不只局限于当前这道题。如果是 1000 层楼，3 个棋子的情况，用这种编码方式可以很轻松的求得解。

对于信息论，所知有限，不过网上的资料不少，可以自行搜索一下，

有不少问题都可以用信息论求得一个下限，比如经典的 12 彩球问题，另外附带一大堆用天平秤球的问题(本总结中第二题)都可以用信息论来解。

对于 1000 层，3 颗棋子，第一次实验的层数，只需要求得

c(n,3) + c(n,2) + c(n,1) >= 1000 其中的 n 就可以了

如果碎了，问题就转化为 2 颗棋子 n 层楼，如果没有碎，问题转化为 3 颗棋子 1000-n 层楼。

10.左上右下最大流问题

说有一个 m*n 的矩阵格子，里面每个格子有一个正数，一个人从左上要走到右下，每次只能往右或者下走，每经过一个格子他就拿走其分数，拿了格子里面的分数就没有了（为 0），求如何走让分数最大。 （动态规划问题）

```
/*****************************************************************
funciton:给定一个数字矩阵求从左上角到右下角距离最大。输出最大距离值。
condition:路径只能是往右与往下走。
Example
Input:
2 2
0  6
3  4
3 2
8 1 9
2 2 9
Output:
10
27
Method
给定一个距离矩阵用来存放从左上角到该位置的最大距离值。
每输入一个值就计算该点的最大距离值。可以知道，该点的最大
距离值就是他的上方和左方两者之间最大的距离值。
*****************************************************************/
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <malloc.h>
int main(void)
{
    int n;
    int m;
```

```
    int i;
    int j;
    int *a;// 数字矩阵
    int *d;// 距离矩阵
    srand((unsigned int)time(NULL));// 随机种子
    while(scanf("%d%d", &n, &m))
    {
        if (m<=n&&n)
        {
            a = (int*)malloc(sizeof(int)*n*m);
            d = (int*)malloc(sizeof(int)*n*m);

            for (i=0; i<m; i++)
            {
                for (j=0; j<n; j++)
                {
                    a[i*n+j] = rand()%10;
                    if (i==0 && j==0)
                        d[i+j] = a[i+j];// 距离矩阵左上角第一个元素
                    else if (i==0 &&j!=0)
                    {
                        d[i*n+j] = d[i*n+j-1]+a[i*n+j];// 第一行的数字
                    }
                    else if (i!=0&&j==0)
                    {
                        d[i*n+j] = d[(i-1)*n+j] + a[i*n+j];// 第一列的数字
                    }
                    else
                    {
             d[i*n+j]= d[(i-1)*n+j] > d[i*n+j-1] ? d[(i-1)*n+j]: d[i*n+j-1];
             d[i*n+j] += a[i*n+j];// 中间的数字都取上方和左方两者最大的
                    }
                    printf("%3d", a[i*n+j]);
                }// end for
                printf("/n");
            }//end for
        }// end if
        else
            return 0;
        printf("%d", d[n*m-1]);
        free(a);
        free(d);
    }// end while
    return 0;
```

```
72. }
```

另：给出一个 n*n 大小的矩阵,要求从左上角走到右下角,每次只能向下走或者向右走并取数,某位置取过数之后就只为数值 0,现在求解从左上角到右下角走 K 次的最大值.

将矩阵的每个元素 m[i][j]拆成两个点 u=(i-1)*n+j 和 v=n*n+(i-1)*n+j,从 u 到 v 连两条边:

1> 连边(u,v),容量为 1,费用值为 m[i][j],这样可以保证每一个位置的数只被取一次

2> 连边(u,v),容量为 INF,费用值为 0,这样可以保证某位置取数被置为 0 之后,随便怎么取对最后的费用值不会产生影响

由于每次只能向下走或者向右走,所以需要连两条边:

1> 向下走:即从(i,j)到(i+1,j),连边 v=n*n+(i-1)*n+j 到 i*n+j,容量为 INF,费用为 0

2> 向右走:即从(i,j)到(i,j+1),连边 v=n*n+(i-1)*n+j 到(i-1)*n+j+1,容量为 INF,费用为 0.

然后为了限制只走 K 次,需要添加源点 s=0 和汇点 t=2*n*n+1,连边:s……矩阵左上角顶点 1,容量为 k,费用为 0,连边矩阵右下角顶点被拆之后对应的顶点 2*n*n……t,容量为 K,费用为 0,然后对所建立的图求解最大费用流即可.

http://hi.baidu.com/zhuangxie1013/item/f4bb010867f1563d4bc4a31a

```
1.  #include<iostream>
2.  #include<cstring>
3.  #include<cstdlib>
4.  #include<cmath>
5.  #include<algorithm>
6.  using namespace std;
7.  const int MAXN=55;
8.  const int MAXM=200005;
9.  const int INF=0x3f3f3f3f;
10. struct Edge
11. {
12.         int u,v;  //起点和终点
13.         int cap;  //容量
14.         int cost;  //费用
15.         int next;
16. }e[MAXM];
17. int p[MAXM],q[MAXM],d[MAXM],inq[MAXM],head[MAXM],m[MAXN][MAXN];
18. int n,k,c,f,pos;
19. void addedge(int u,int v,int cap,int cost)
20. {
21.      e[pos].u=u; e[pos].v=v;  e[pos].cap=cap; e[pos].cost=cost;
22.      e[pos].next=head[u]; head[u]=pos++;
23.      e[pos].u=v; e[pos].v=u;  e[pos].cap=0;
24.      e[pos].cost=-cost; e[pos].next=head[v]; head[v]=pos++;
25. }
26. bool SPFA(int s,int t)  //查找从 s……t 是否存在增广路
```

```
{
    int u,v,now,front,rear;
    memset(inq,0,sizeof(inq));
    memset(p,-1,sizeof(p));
    for(u=0;u<=t;u++)
      d[u]=(u==s?0:-1);
    front=rear=0;
    q[rear++]=s;
    inq[s]=true;
    while(front<rear)
    {
          u=q[front++];
          inq[u]=false;
          for(now=head[u];now!=-1;now=e[now].next)
          {
               v=e[now].v;
      if(e[now].cap>0 && d[v]<d[u]+e[now].cost)  //注意这里是求解最大费用流
               {
                   d[v]=d[u]+e[now].cost;
                   p[v]=now;
                   if(!inq[v])
                   {
                        inq[v]=true;
                        q[rear++]=v;
                   }
               }
          }
    }
    if(p[t]!=-1 && d[t]!=-1) //存在增广路径
       return true;
    return false;
}
void Max_Cost(int s,int t) //求解最大费用流
{
    int i,u,a;
    c=f=0;
    while(SPFA(s,t))
    {
          a=INF;
          u=t;
          while(u!=s)
          {
              a=min(a,e[p[u]].cap);
              u=e[p[u]].u;
```

```
            }
            u=t;
            while(u!=s)
            {
                e[p[u]].cap-=a;
                e[p[u]^1].cap+=a;
                u=e[p[u]].u;
            }
            c+=d[t]*a;
            f+=a;
     }
}
int main()
{
    int i,j,u,v,s,t,tmp;
    while(scanf("%d%d",&n,&k)!=EOF)
    {
         s=pos=0;
         tmp=n*n;
         t=2*tmp+1;
         memset(head,-1,sizeof(head));
         for(i=1;i<=n;i++)
          for(j=1;j<=n;j++)
            scanf("%d",&m[i][j]);
         for(i=1;i<=n;i++)
         {
             for(j=1;j<=n;j++)
             {
                  u=(i-1)*n+j;  //将位置(i,j)处得顶点表示成顶点 u 和所对应的 v
                    v=u+tmp;
                    addedge(u,v,1,m[i][j]);
                    addedge(u,v,INF,0);
                    if(i<n)
                       addedge(v,u+n,INF,0);
                    if(j<n)
                       addedge(v,u+1,INF,0);
               }
         }
         addedge(s,1,k,0);
         addedge(2*tmp,t,k,0);
         Max_Cost(s,t);
         printf("%d\n",c);
     }
     system("pause");
```

```
115.      return 0;
116.  }
```

11.三角形内产生随机数

要求三角形内均匀产生。(完美世界 2013intern)

12.赛马问题

1. 25 匹马，5 个跑道，求跑的最快的 3 匹马需要比赛的场次。

   分 A.B.C.D.E 五组，各跑一场，选出 A1.B1.C1.D1.E1 再用他们比赛一场取出第一名，即是 25 匹马中的第一名；假定 A1>B1>C1>D1>E1 因仅取前三名，D 组与 E 组可以全部淘汰，因为比 D 组和 E 组快的至少有 A1B1C1 三个。

   故可能占据前三的为 A2 A3 B2 再加上 A1 B1 C1，决出第一名后剩余的五名再比赛一次即可获得第二名第三名。从而得出结果。共需要七场。

2. 25 匹马，5 个跑道，获得前 5 名。(存疑)

   (1)首先将 25 匹马分成 5 组，并分别进行 5 场比赛之后得到的名次排列如下：

   A 组： [A1 A2 A3 A4 A5]

   B 组： [B1 B2 B3 B4 B5]

   C 组： [C1 C2 C3 C4 C5]

   D 组： [D1 D2 D3 D4 D5]

   E 组： [E1 E2 E3 E4 E5]

   其中，每个小组最快的马为[A1、B1、C1、D1、E1]。

   (2)将[A1、B1、C1、D1、E1]进行第 6 场，选出第 1 名的马，不妨设 A1>B1>C1>D1>E1. 此时第 1 名的马为 A1。

   (3)将[A2、B1、C1、D1、E1]进行第 7 场，此时选择出来的必定是第 2 名的马，不妨假设为 B1。因为这 5 匹马是除去 A1 之外每个小组当前最快的马。

   (3)进行第 8 场，选择[A2、B2、C1、D1、E1]角逐出第 3 名的马。

   (4)依次类推，第 9，10 场可以分别决出第 4，5 名的吗。

   因此，依照这种竞标赛排序思想，需要 10 场比赛是一定可以取出前 5 名的。

   仔细想一下，如果需要减少比赛场次，就一定需要在某一次比赛中同时决出 2 个名次，而且每一场比赛之后，有一些不可能进入前 5 名的马可以提前出局。比如 E2E3E4E5D3D4D5C4C5B5 当然要做到这一点，就必须小心选择每一场比赛的马匹。我们在上面的方法基础上进一步思考这个问题，希望能够得到解决。

   (1) 首先利用 5 场比赛角逐出每个小组的排名次序是绝对必要的。

   (2) 第 6 场比赛选出第 1 名的马也是必不可少的。假如仍然是 A1 马(A1>B1>C1>D1>E1)。那么此时我们可以得到一个重要的结论：有一些马在前 6 场比赛之后就决定出局的命运了(下面绿色字体标志出局)。

   A 组： [A1 A2 A3 A4 A5]

   B 组： [B1 B2 B3 B4 B5]

   C 组： [C1 C2 C3 C4 C5]

D 组： [D1 D2 D3 D4 D5]
E 组： [E1 E2 E3 E4 E5]
(3) 第 7 场比赛是关键，能否同时决出第 2，3 名的马呢？我们首先做下分析：
在上面的方法中，第 7 场比赛[A2、B1、C1、D1、E1]是为了决定第 2 名的马。但是在第 6 场比赛中我们已经得到(B1>C1>D1>E1)，试问？有 B1 在的比赛，C1、D1、E1 还有可能争夺第 2 名吗？
当然不可能，也就是说第 2 名只能在 A2、B1 中出现。实际上只需要 2 条跑道就可以决出第 2 名，剩下 C1、D1、E1 的 3 条跑道都只能用来凑热闹的吗？
能够优化的关键出来了，我们是否能够通过剩下的 3 个跑道来决出第 3 名呢？当然可以，我们来进一步分析第 3 名的情况？
● 如果 A2>B1(即第 2 名为 A2)，那么根据第 6 场比赛中的(B1>C1>D1>E1)。 可以断定第 3 名只能在 A3 和 B1 中产生。
● 如果 B1>A2(即第 2 名为 B1)，那么可以断定的第 3 名只能在 A2,
B2,C1 中产生。
好了，结论也出来了，只要我们把[A2、B1、A3、B2、C1]作为第 7 场比赛的马，那么这场比赛的第 2，3 名一定是整个 25 匹马中的第 2，3 名。
我们在这里列举出第 7 场的 2，3 名次的所有可能情况：
① 第 2 名=A2，第 3 名=A3
② 第 2 名=A2，第 3 名=B1
③ 第 2 名=B1，第 3 名=A2
④ 第 2 名=B1，第 3 名=B2
⑤ 第 2 名=B1，第 3 名=C1
(4) 第 8 场比赛很复杂，我们要根据第 7 场的所有可能的比赛情况进行分析。
① 第 2 名=A2，第 3 名=A3。那么此种情况下第 4 名只能在 A4 和 B1 中产生。
● 如果第 4 名=A4，那么第 5 名只能在 A5、B1 中产生。
● 如果第 4 名=B1，那么第 5 名只能在 A4、B2、C1 中产生。
不管结果如何，此种情况下，第 4、5 名都可以在第 8 场比赛中决出。其中比赛马匹为[A4、A5、B1、B2、C1]

② 第 2 名=A2，第 3 名=B1。那么此种情况下第 4 名只能在 A3、B2、C1 中产生。而第 7 场中三者已经比赛过，故此时第 4 名已经得出。
● 如果第 4 名=A3，那么第 5 名只能在 A4、B2、C1 中产生，三者比赛即可。
● 如果第 4 名=B2，那么第 5 名只能在 A3、B3、C1 中产生，三者比赛即可。
● 如果第 4 名=C1，那么第 5 名只能在 A3、B2、C2、D1 中产生，四者比赛即可。
//那么，第 4、5 名需要在马匹[A3、B2、B3、C1、A4、C2、D1]七匹马中产生，则必须比
//赛两场才行，也就是到第 9 场角逐出全部的前 5 名。
所以需要八场即可。

③ 第 2 名=B1，第 3 名=A2。那么此种情况下第 4 名只能在 A3、B2、C1 中产生。
情况和②一样，必须角逐第 8 场

④ 第 2 名=B1，第 3 名=B2。 那么此种情况下第 4 名只能在 A2、B3、C1 中产生。因在第 7 场中 A2 C1 已经比赛过，已经知道其先后顺序，故第四名只在 B3 和两者之间的快者中产生。

- 如果第 4 名=A2，那么第 5 名只能在 A3、B3、C1 中产生。
- 如果第 4 名=B3，那么第 5 名只能在 A2、B4、C1 中产生。
- 如果第 4 名=C1，那么第 5 名只能在 A2、B3、C2、D1 中产生。

那么，第 4、5 名需要在马匹[A2、B3、B4、C1、A3、C2、D1]七匹马中产生，则必须比赛两场才行，也就是到第 9 场角逐出全部的前 5 名。

⑤ 第 2 名=B1，第 3 名=C1。那么此种情况下第 4 名只能在 A2、B2、C2、D1 中产生。

- 如果第 4 名=A2，那么第 5 名只能在 A3、B2、C2、D1 中产生。
- 如果第 4 名=B2，那么第 5 名只能在 A2、B3、C2、D1 中产生。
- 如果第 4 名=C2，那么第 5 名只能在 A2、B2、C3、D1 中产生。
- 如果第 4 名=D1，那么第 5 名只能在 A2、B2、C2、D2、E1 中产生。

那么，第 4、5 名需要在马匹[A2、B2、C2、D1、A3、B3、C3、D2、E1]九匹马中产生，因此也必须比赛两场，也就是到第 9 场决出胜负。

13.过河问题(INTEL)

有 N 个人想要用一条每次只能坐两人的船过河，因此，需要合理的安排来、以使所有的人都能顺利过河。每个人过河的速度不同，两个人速度取决于速度较慢的一个。你的任务就是想一个策略让所有的人在最短的时间内均过河。

如果 n=3, 过河时间为 A+B+C

如果 n<=2, 不费口舌了

如果 n>=4, 这个是重点:

每次优先考虑把最慢两人送过河，把 n 人中最快两人记为 A,B, 最慢两人记为 C,D(过河时间 A<B<C<D), n 人问题实质上转换为 4 人过河问题, 参考到 4 人过河时的优化,

记 AB 过河, A 回, CD 过河, B 回, 为方法 X, 实质是利用最快两人进行优化, 耗时 A+2B+D

记 AD 过河, A 回, AC 过河, A 回, 为方法 y, 实质是利用最快一人来过河, 耗时 2A+C+D

每次比较这两个方法, 如果 x 快, 使用 x 方法, 如果 y 快, 则用 y, 并且, 一旦某次使用 y 方法后, 以后都不用比较了, 全部使用 y 方法过河

例如：1 99 100 101

用 x 方法是 99+1+101+99= 300

y 方法是 101+1+100+1 = 203

y 比 x 快的原因是 2A+C+D < A+2B+D, 即 A+C<2B

容易想到, 从此以后 A+C 都会小于 2B 了(因为 C 越来越小)

```
#include<iostream>
#include<algorithm>
using namespace std;
int a[1001];
int main()
{
```

```
int t,i,n,fast1,fast2,slow1,slow2,slow3,sum,l,r;
cin>>t; //测试 t 轮
while(t--)
    {
        sum=0;
        cin>>n;
        for(i=1;i<=n;i++)
            cin>>a[i];
        sort(a+1,a+n+1); //对其进行排序
        fast1=a[1];fast2=a[2];slow1=a[n-1];slow2=a[n];slow3=0;
        l=1;r=n;
        while(n!=0)
        {
            if(n==1) {sum+=slow2;break;}
            if(n==2) {sum+=slow2;break;}
            if(n==3) {sum+=(slow2+slow1+fast1);break;}
            if(2*fast2>=fast1+slow1)
{sum+=(slow1+slow2+2*fast1);r-=2;slow2=a[r];slow1=a[r-1];n-=2;}
            else {sum+=2*fast2+fast1+slow2;r-=2;slow2=a[r];slow1=a[r-1];n-=2;}
        }
        cout<<sum<<endl;
        }
     return 0;
   }
```

14.数星星问题

阿里 2013 第一次实习笔试题目：有 N 颗星星，A 与 B 两人轮流数，每次只能数 K 个(30≥K≥20)，A 先数，谁数最后一组谁赢，问 N 符合什么条件时，A 必赢？

类似题目扩展可见：http://dwz.cn/azvhj

假设 N 可以表示为 N=50*C+D 的形式；

如果 D 在[20,30]，则 A 必赢；可以按照如下策略取；A 先数 D 个星星；随后 B 数 X 个星星，A 都数相应的 50-X 个星星，则如此下去，最后一轮剩余的个星星必然由 A 来数得。

当 D>30 时，B 必赢。因 A 第一次必须取 K 个，而后 B 每次都取 50-K 个，最终必然剩余 D 个，此时，无论 A 取多少个，剩余的最后一局必然由 B 取得。B 必赢。

如果 D<20 时，A 必赢。设 A 第一次取 20 个，尔后每次 B 数 K 个，A 都取 50-K 个，最终轮到 B 某次必然剩余 30+D 个，此时 A 取完，必赢。故当总数量可以表示为上式，并且 D<=30 时，A 必赢。

15.交流问题/GOSSIP PROBLEM

战场上不同的位置有 N 个战士（N>4），每个战士知道当前的一些战况，现在需要这 N 个战士通过通话交流，互相传达自己知道的战况信息，每次通话，可以让通话的双方知道对

方的所有情报，设计算法，使用最少的通话次数，是的战场上的 n 个士兵知道所有的战况信息，不需要写程序代码，得出最少的通话次数。（阿里 2013 实习/2014 校招第二次笔试）

此题的原型见：http://www.matrix67.com/blog/archives/1078

1) N-1 个人围成一个环，将知道的消息告诉都第 N 个人，需要 N-1 次，同时第 N-1 个人与第 N 个人交流时，两人互相交流消息，这样子第 N-1 与第 N 个人都知道了所有人的信息，接下来第 N-1 人沿着环将消息传递下去，需要 N-2 次。所以需要 N-1+(N-2)=2N-3. 但是实际上这个问题有着更优的解法。

2) 当 N=4 时，假设 ABCD，可以让 AB 互相通话，CD 互相通话，此时每个人都知道了（包括自己的）两条消息；然后 A 和 C 通话，B 和 D 通话，从而使得每个人都获知另外两条自己还不知道的消息。显然，对于 4 个人的情况，4 通电话已经是最少的了。

而对于四个人以上的情况可以以此类推。对于 n>4 的情况，有一种算法可以保证在 2n-4 通电话内解决问题。首先，选出 4 个人作为消息汇总人。其余每个人都选择一个汇总人并与之交流；然后 4 个汇总人再用 4 次交流互相更新一下消息（同上）；最后 4 个汇总人把电话再打回去，实现所有消息全部共享。此时攻需要 2(N-4)+4=2N-4.

关于其最小性的证明可以通过上述网址查询到结果。

16.交换问题

有一个淘宝商户，在某城市有 n 个仓库，每个仓库的储货量不同，现在要通过货物运输，将每次仓库的储货量变成一致的，n 个仓库之间的运输线路围城一个圈，即 1->2->3->4->...->n->1->...，货物只能通过连接的仓库运输，设计最小的运送成本（运货量*路程）达到淘宝商户的要求，并写出代码。（阿里 2013intern）

见：http://blog.csdn.net/linygood/article/details/8895902

类似 ACM 中的题目：http://www.lydsy.com/JudgeOnline/problem.php?id=1045

有 n 个小朋友坐成一圈，每人有 ai 个糖果。每人只能给左右两人传递糖果。每人每次传递一个糖果代价为 1，求使所有人获得均等糖果的最小代价。

数学表示：设第 i 个小朋友分给第 i-1 个小朋友的糖果数为 P[i](注意, P[i]可正可负, 负数理解成为第 i-1 个小朋友分给第 i 个小朋友-Pi 个糖果),P[1]为第一个小朋友分给第 n 个小朋友的, 显然最后的答案就是要求|P1|+|P2|+..+|Pn|的最小值.

而我们的最总目的是，将 p[i]全部用 p[1]表示。

设 ave 为最终每个人拿到的糖果数目，也就是平均数。

则 ave= -p[i]+p[i+1]+a[i],第 i+1 个人给 i 的加上原来的去掉第 i 个人给第 i-1 的即最终的结果。再设 w[i]=ave-a[i] 则变成 w[i]= -p[i]+p[i+1]

设 s[i]=w[1]+w[2]+⋯+w[i] 则 s[i]= -p[1]+p[2]-p[2]+p[3]-...-p[i]+p[i+1]=p[i+1]-p[1],有 p[i+1]=p1+s[i]，则有：

|P1|+|P2|+..+|Pn|=|p1|+|p1+s[1]|+..+|p1+s[n-1]|

设 Q=-p1

则 原式变为|0-Q|+|s[1]-Q|+..+|s[n-1]-Q|

这个式子可以看成 求 s[1],s[2]⋯s[n-1]到 Q 的 最短距离之和，

这个数列的中位数就是 Q 值，此即-p1，由此得出 1 交给 n 的最优值，而由此也可以得出上述式子的最小值。(类似的题目即是，在平面上若干点中，寻找到各点曼哈顿距离和最小的点，即各点坐标的中位数)

```cpp
#include <cstring>
#include <iostream>
#include <algorithm>
using namespace std;
const int X = 1000005;
typedeflong longll;
ll sum[X],a[X];
ll n;
ll Abs(ll x){
return max(x,-x);
}
int main(){
//freopen("sum.in","r",stdin);
while(cin>>n){
ll x;
ll tot = 0;
for(inti=1;i<=n;i++){
scanf("%lld",&a[i]);
tot += a[i];
}
ll ave = tot/n;
for(inti=1;i<n;i++)
sum[i] = a[i]+sum[i-1]-ave;
sort(sum+1,sum+n);
ll mid = sum[n/2];
ll ans = Abs(mid);
for(inti=1;i<n;i++)
ans += Abs(sum[i]-mid);
cout<<ans<<endl;
}
return0;
}
```

17.换数

在黑板上写下 50 个数字：1 至 50.在接下来的 49 轮操作中，每次做如下动作：选取两个黑板上的数字 a 和 b，擦去，在黑板上写|b - a|。请问最后一次动作之后剩下数字可能是什么？为什么？（阿里 2014 校招）

详见：http://www.360doc.com/content/11/0312/12/6307578_100433348.shtml 搜索黑板即可。

黑板上开始时所有数的和为

S=1+2+3+…+50=1275，是一个奇数，而每一次“操作”，将（a+b）变成了|a-b|，实际上减少了 2*(min(a,b))，即减少了一个偶数。因为从整体上看，总和减少了一个偶数，其奇偶性不变，所以最后黑板上剩下一个奇数。

18.消耗问题

1) 运水问题(往返)

人人 2014 校招北航。

一共 240kg 水，每次最多运 60kg 水，每运一公里就要喝掉一公斤水，起点时水费 0 元，10km 时每公斤 10 元，20km 时每公斤 20 元，要保证运水员能够回到原地的情况下，最多能获得多少钱？

每次运 60kg 水，假设走 X 的距离，则到达 X 距离后保证归来的情况下可以获得(60-2X)*X 元，此函数在 X=15 处取得最大值。即(60-30)*15=450 * 4=1800

2) 小毛驴运玉米(单程)

赶集网 2014 校招。

有一条 1000 公里的路，小毛驴在起点，它每走一公里要吃一斤玉米，小毛驴最多能背 1000 斤玉米，正常走完全程玉米会正好吃光。在起点有 3000 斤玉米，请问小毛驴如何安排，可以让它把最多的玉米**背到终点**？

第一段：驴先驮 1000 根胡萝卜走 200 公里，卸下 600 根，归途消耗 200 根；回到出发点再运 1000 根，到 200 公里处，卸下 600 根，归途消耗 200 根。最后运剩下 1000 根，到 200 公里处，不用返回。此时总共剩胡萝卜 2000 根，驴吃掉 1000 根。

第二阶段：由 200 公里处向前推进 333 公里，即走到 533 公里处，运 2 次，方法同上。此时驴又吃掉 999 根胡萝卜，还剩 1001 根胡萝卜。

第三阶段：将 1 根胡萝卜喂驴，剩下 1000 根由驴驮着前进，走 467 公里至终点，驴又吃掉 466 根胡萝卜（先前已喂 1 根），最终仅剩 534 根胡萝卜。

见：http://blog.sina.com.cn/s/blog_54379bfa0100dhiv.html

19.四则算式

1) 输入表达式输出结果。例如输入 1+2*3-6/(3-2)

其中仅包含数字及+-*/四个符号且+-不用做正负号。并陈述测试用例.

(小米 2014 测试笔试)(另外两道题目为奇偶排序和树转双向链表)

此题与 2)类似。相对 2)来说，需要自己考虑计算顺序，首先解括号，将括号内的部分单独算出，重新压栈，并且需要注意多重括号及括号不匹配的情况；将括号解除后，算式将变为简单的加减乘除四则算式，此时利用双栈进行处理即可，但是要计算两次首先计算乘除，而后计算加减，最后返回结果。因为双栈之间反复压栈，所以要注意操作符的运算顺序。

测试用例：

a.空串

b.一般输入， 1+2*(3-4)

c.多层括号嵌套， 1+2*((3-4)+8)

d.非法字符输入，9+A+8

e.非法计算符号，*-20+5

f.括号不匹配，1+2*(3-4)(

g.括号内为空，1+2*()*9

h.除零情况，1+2*6/(3-3)

```
struct Nnode                                    {
```

```
    float num;
    char c;
};
float computestr(char *str)
{
    int left=0,right=0;
    int len=strlen(str);
    int bracketNUM(0);
    stack<Nnode> s1;
    stack<Nnode> s2;
    int number=0;
    Nnode temp;
    for (int i=0;i<len;)
    {
        int original=i;
while(i<len&&str[i]>='0'&&str[i]<='
9'){
    number=number*10+str[i]-'0';
    i++;
    }
    if (i!=original){
            temp.num=number;
            temp.c=NULL;
            s1.push(temp);
            number=0;
        }
if (i<len&&(str[i]=='+' || str[i]=='-'||
str[i]=='*'|| str[i]=='/'))
        {
            temp.c=str[i];
            temp.num=NULL;
            s1.push(temp);
            i++;
        }
if (i<len&&str[i]=='(')
//if includes brackets
{
bracketNUM=1;
//multi brackets
left=i;
int j=i+1;
for (;j<=len;j++)
{
    if (bracketNUM==0)
    //bracket matches
    {
right=j-1;
char *bracket=(char
*)malloc((right-left)*sizeof(char));
int bracketLen=0;
for (int k=left+1;k<right;k++)
{
    bracket[bracketLen++]=str[k];
}

    bracket[bracketLen]='\0';
float ret = computestr(bracket);
//recursively the ones in brackets
temp.num = ret;
temp.c = NULL;
s1.push(temp);
free(bracket);
break;
    }
if (str[j]==')')
{bracketNUM--;
}
if (j==len&&bracketNUM!=0)
{
    cout<<"bracket not matches";
    return -1;
    break;
    }
if (str[j]=='(')//multi brackets
{
bracketNUM++;
}
}
  i=j;
  }
}
while(!s1.empty())
//compute with the multi & divide
{
```

```
Nnode ch=s1.top();
s1.pop();
if (ch.c!='*'&&ch.c!='/')
{
    s2.push(ch);
}
else if(ch.c!='*'||ch.c!='/')
{
float v1=s1.top().num;
float v2=s2.top().num;
s1.pop();
s2.pop();
if (ch.c=='*')
{temp.num=v1*v2;
temp.c=NULL;
s2.push(temp);
}
else if (ch.c=='/')
{
    if (v2==0)
    {
    cout<<"除数为 0"<<endl;
    return -1;
    break;
    }
    else
    {
    temp.num=v1/v2;
    temp.c=NULL;
    s2.push(temp);
    }
}

}
}

    while (!s2.empty())
    {
        Nnode ch=s2.top();
        s2.pop();
        if (ch.c!='+'&&ch.c!='-')
        {
            s1.push(ch);
        }
        else if(ch.c=='+'||ch.c=='-')
        {
        float v1=s2.top().num;
        float v2=s1.top().num;
        s1.pop();
        s2.pop();
        if (ch.c=='+')
        {
        temp.num=v1+v2;
        temp.c=NULL;
                s2.push(temp);
            }
            else if (ch.c=='-')
            {
            //-has changed twice
            temp.num=v2-v1;
            temp.c=NULL;
                s2.push(temp);
            }
        }
    }
    return s1.top().num;
}
```

2) 输入表达式为中缀表达式，输出结果

2+(3+4)*5

按照中缀式加上括号(2+((3+4)*5))　然后把运算符写在括号前面是 +(2*(3 4) 5)

把括号去掉即是中缀式：+ 2 * + 3 4 5

据此求中缀表达式的值。(赶集网 2014 年笔试)

利用双栈进行数据的存储，首先将数字及符号分别存储并压入栈 1，而后逐个弹出栈 1 中的元素，如果栈 1 中的元素为数字，则哨兵+1，并将当前数值压入堆栈 2；如

果哨兵超过 2 并且当前弹出值是操作符，则从堆栈 2 中弹出 2 个数字与该操作符操作后重新压入栈 2，同时哨兵-1。

```
struct Nnode
{
    float num;
    char c;
};
float computestrmid(char *str)
{
    int len=strlen(str);
    stack<Nnode> s1;
    stack<Nnode> s2;
    int number=0;
for (int i=0;i<len;)
{
    if (str[i]==' ')
    //ignore the space
    {
        i++;
        continue;
    }
    int original=i;
while(i<len&&str[i]>='0'&&str[
i]<='9'){

 number=number*10+str[i]-'0';
        i++;
    }
if (i!=original)
//push the number maybe 0
    {
        Nnode numb;
        numb.num=number;
        numb.c=NULL;
        s1.push(numb);
        number=0;
    }
if (i<len&&(str[i]=='+' || str[i]=='-'||
str[i]=='*'|| str[i]=='/'))
//push the symbol
    {
        Nnode ch;
        ch.c=str[i];
        ch.num=0;
        s1.push(ch);
        i++;
    }
}
int flagnum(0);
//two stack works below
while(!s1.empty())
{
    Nnode elem=s1.top();
    s1.pop();
    if (elem.c==NULL)
//it's a number
    {
        s2.push(elem);
        flagnum++;
    }
    else//it's a symbol
    {
        if (flagnum>=2)
//two operator has appeared
    {
    float num1=s2.top().num;
    s2.pop();
    float num2=s2.top().num;
    s2.pop();
    Nnode temp;
    if (elem.c=='+')
    {
    float result=num1+num2;
    temp.num=result;
    }
    else if (elem.c=='-')
    {
    float result=num1-num2;
    temp.num=result;
    }
    else if (elem.c=='*')
    {
```

```
float result=num1*num2;
temp.num=result;
        }
else if (elem.c=='/')
        {
float result=num1/num2;
        if (num2==0)
        {
cout<<"除数为 0"<<endl;
        return -1;
        break;
        }
temp.num=result;
                }
        temp.c=NULL;
        s2.push(temp);
        flagnum--;
        }
        else
        {
        s2.push(elem);
        }
        }
}
  return s2.top().num;
}
```

20.国王与魔鬼下棋问题

(世通量化基金)国王和魔鬼在一个无限大的围棋棋盘上下棋：国王有一个棋子，最初处在坐标为（0, 0）的格点上，每一回合棋子可以移动到与其邻近的 8 个格点上；国王每走一次，魔鬼可以破坏掉除棋子所在位置的任意一个格点，破坏后的格点棋子不能再走上去。

a) 如果魔鬼要将国王限制在 Y=A 以下，A 满足什么条件时魔鬼必赢？

b) 如果魔鬼要阻止国王进入 X>B 且 Y>B 的区域，B 满足什么条件则魔鬼必赢？

c) 如果魔鬼要将国王限制在-C<X<C，-C<Y<C 的区域内，C 满足什么条件则魔鬼必赢？

七． 数学与逻辑

1. 停时定理

一个随机变量序列 X1,X2,X3,…如果满足下列两个条件，就可称为离散时间鞅:

1. E(|Xn|)<∞;
2. E(Xn+1｜X1,…,Xn)=Xn.

“停时”的直观含义是赌徒决定退出的时间: 由于无法预测接下来的赌局的胜负，他只能根据现在和以前的胜负情况决定是否退出. 例如预先决定只玩 t 局，玩到了 t 局后无论胜负均退出游戏，这时 τ=t 是一个“停时”. 又例如采取“加倍投注”策略，最初投注 1 元，赢了就结束，输了就投注 2 元，… 直到赢一盘(总获利为 1 元)或者输光为止，这一策略下赌徒的退出时间 τ 也是一个“停时”. 也就是说，鞅 X1,X2,X3,…的“停时” τ 是一个取值为非负整数的随机变量，{τ=t}仅取决于 X1,X2,X3,…,Xt 的取值.

满足一定条件的鞅位于停时的期望等于其初值的期望，这就是“停时定理”. 用数学符号描述如下. 对于鞅 X1,X2,X3,…及其停时 τ，如果

1. E(τ)<∞，
2. 存在常数 c 使得 E(|Xi+1 − Xi|)≤c 对于所有 i 成立,

那么 E(Xτ)=E(X1)=E(Xi),i=1,2,3,….鞅 X1,X2,X3,…具有一个基本性质: E(Xi)=E(X1),i=1,2,3,…，这一点可以根据鞅的定义和期望的迭代性质 E(E(X｜Y))=E(X) 进行证明. 停时定理相当于将此性质由固定下标 i 推广到随机下标 τ，当然必须满足上述两个条件才能成立.

停时定理告诉我们，永远赚钱的赌博策略是不存在的. 因为一个赌徒寿命有限(满足停时定理的条件一)，而且每一次的赌注也有限(满足停时定理的条件二)，那么根据停时定理，他退出赌局时赌资总额的期望与刚开始赌博时相等. 以“加倍投注”策略为例，假设一开始有赌资 2n − 1 元，最初投注 1 元，赢了就结束，输了就投注 2 元，… 直到赢一盘(总获利为 1 元)或者输光为止，这样至多可以玩 n 局，停时的赌资总额或者是 2n 元或者是 0. 停时赌资总额的期望等于

2^n×(1/2+1/4+…+1/2^n)+0×1/2^n=2^n−1,

与初始赌资相等. 这个结果与停时定理的结论一致.

停时定理还告诉我们，散户永远赌不过庄家. 如果你有 a 元，我有 b 元，我俩进行公平的抛硬币赌博，直到一方输光为止. 那么停时你的赌资总额或者是 a+b 元或者是 0，根据停时定理,其期望等于 a 元. 设你笑到最后的概率为 p，则有 a = p(a+b), p = a/(a+b). 赌博最终就是看谁的本钱多，而不是谁的运气更好.

例如：一个国家人们只想要男孩，每个家庭都会一直要孩子，只到他们得到一个男孩。如果生的是女孩，他们就会再生一个。如果生了男孩，就不再生了。那么，这个国家里男女比例如何？

最严格的方法是直接计算男孩和女孩数量的期望，都是 1，所以男女比例平衡。另一种说法是每个孩子的概率都是 1：1，所以总的来看，最终的比例也将是 1:1。这种说

法可以用概率里面的"停时"的概念严格化，其实也不简单。比如，如果每个家庭一直要孩子，直到男孩比女孩数多一个，这时候如何分析？
"停时"是一个很深刻的概念，也非常有用。比如如果题目改成“如果每个家庭一直都要孩子，直到男孩比女孩数多一个，或者生到了 100 个小孩(是不是有点太多了)为止”，这时候如果直接算期望可不是一件容易的事情，但用停时定理一步就能得出结果。

2. 基本公式

- 如果从 n 个元素中取出 k 个元素，这个元素的**排列**是多少呢？公式如下：

$$P_k^n = \frac{n!}{(n-k)!}$$

- 和排列不同的是，在组合中取出元素的顺序则不在考虑之中。从 n 个元素中取出 k 个元素，这个元素可能出现的组合数为：

$$C_n^k = \binom{n}{k} = \frac{P_n^k}{k!} = \frac{n!}{k!(n-k)!}$$

- 多次组合情况下，如果每次抽出的结果放回再重复，这时的组合的可能性则是：

$$H_n^k = C_{n+k-1}^k = \binom{n+k-1}{k}$$

3. 实现’A+B’

将 a 和 b 以计算机的观点看均为二进制数，当计算机在做加法时，如果两位都是 1 则归零进位，如果仅有 1 位是 1 则加上当前的进位为结果，故加法的目的就是确定进位。位运算即可提供此种观点的解法。如下：

```
int addT(int a ,int b)
{
    if (a==0)
        return b;
    if (b==0)
        return a;
    int p1=a&b;//两位都是 1 的位数
    int p2=a^b;//两位只有 1 位是 1 的情况；
    p1=p1<<1;//进位
    return addT(p2,p1);
}
```

4. 估算 N! 的位数

如输入 9！有 362,880，有 6 位。当我们在表示很大的位数的时候，如果我们只是为了表示其的位数，可以使用科学计数法，比如光速约等于 3×10^8，该数共有九位；当转变为科学技术法后，即可一目了然；

我们照此想下去，如果 N!≈a×10^b;我们对该数取对数即可。而对数可以将阶乘转为加法，由此可得结果

5. N 的开方

编程实现正整数 N 的开方，精确到 0.0001。（创新工场）

二分法：

```
double sqrt_test(int N)
{
    double low,high;
    if (N<0)
    {
        return -1;
    }
    double eps=0.0001;
    if (N>=1)
    {
        low=1;
        high=N;
    }
    double mid =(low+high)/2;
    double sq=mid*mid;
    while(abs(sq-N)>eps)
    {
        if (sq<N)
        {
            low=mid;
        }
        else
        {
            high=mid;
        }
        mid =(low+high)/2;
        sq=mid*mid;
    }
    return mid;
}
```

牛顿迭代法：

例如求 5 的开方，先求 5/2=2.5 -> (2.5+2)/2=2.25-> 5/2.25=X-> mid=(X+2.25)/2

相对于二分法，牛顿法能够更快的下降。

```
double newtonsqrt(int N)
{
    double mid=N/2;
    double temp=2;
    double eps=0.0001;
    while (abs(mid*mid-N)>eps)
    {
        temp=N/mid;
        mid = (mid+temp)/2;
    }
    return mid;
}
```

卡马克开平方法：

相对于牛顿下降法来说，卡马克牛就牛在选择了 0x5f3759df 这个开始值，使得迭代的时候收敛速度暴涨，对于 Quake III 所要求的精度 10 的负三次方，只需要一次迭代就能够得到结果。

```
float kamake_sqr(float number) {

    long i;
    float x, y;
    const float f = 1.5F;
    x = number * 0.5F;
    y = number;
    i = *(long *) &y;
    i = 0x5f3759df - (i >> 1);
    y = *(float *) &i;
    y = y * (f - (x * y * y));
    y = y * (f - (x * y * y));
    return number * y;
    }
```

牛顿法附录：牛顿法一般用于求函数的零点，首先，选择一个接近函数$f(x)$零点的x_0，计算相应的$f(x_0)$和切线斜率$f'(x_0)$。然后我们计算穿过点$(x_0, f(x_0))$并且斜率为$f'(x_0)$的直线和x轴的交点的x坐标，也就是求如下方程的解：

$$f(x_0) = (x_0 - x) \cdot f'(x_0)$$

我们将新求得的点的x坐标命名为x_1，通常x_1会比x_0更接近方程$f(x) = 0$的解。因此我们现在可以利用x_1开始下一轮迭代。迭代公式可化简为如下所示：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

已经证明，如果f'是连续的，并且待求的零点x是孤立的，那么在零点x周围存在一个区域，只要初始值x_0位于这个邻近区域内，那么牛顿法必定收敛。并且，如果$f'(x)$

不为 0，那么牛顿法将具有平方收敛的性能. 粗略的说，这意味着每迭代一次，牛顿法结果的有效数字将增加一倍。

6. 三个数组求最大距离

已知三个升序整数数组 a[l], b[m]和 c[n]。请在三个数组中各找一个元素，是的组成的三元组距离最小。三元组的距离定义是：假设 a[i]、b[j]和 c[k]是一个三元组，那么距离为:**Distance = max(|a[I] – b[j]|, |a[I] – c[k]|, |b[j] – c[k]|)**

请设计一个求最小三元组距离的最优算法，并分析时间复杂度。（阿里 2014 校招）

分析：Distance 是以 a,b,c 任意两点间距离的最大值来度量的。将 a,b,c 看做一条线上的三个点，求 Distance 的最小值，即求三个最邻近的点。设三点 abc，从左到右依次为 a,b,c，则如果 c 增加结果必然增加，b 增加或者结果不变或者结果增加，只有 a 增加结果才可能减小。可以想象同一直线上三个点的移动。故每次均移动最小的数的索引，求得最小数。最终复杂度为 O(l+m+k)。

另外原式 Distance 可以化简。根据 max(a,b)=(|a+b|+|a-b|)/2,有

max{|x1-x2|,|y1-y2|} =(|x1+y1-x2-y2|+|x1-y1-(x2-y2)|)/2

Distance = max(|x1 – x2|, |x1 – x3|, |x2 – x3|)

= max(max(|x1 – x2|, |x1 – x3|) , |x2 – x3|)

=max(|x1 – x2|, |x1 – x3|) = 1/2 (|2x1 – x2 – x3| + |x2 – x3|)

= max(1/2 (|2x1 – x2– x3| + |x2 – x3|) , |x2 – x3|)

=1/2 * max(|2x1 – x2 – x3| , |x2 – x3|) + 1/2*|x2 – x3|

=1/2 * max(|2x1 – (x2 + x3)| , |x2 – x3|) + 1/2*|x2 – x3|

=1/2 * 1/2 *((|2x1 – 2x2| + |2x1 – 2x3|) + 1/2*|x2 – x3|

=1/2 *|x1 – x2| + 1/2 * |x1 – x3| + 1/2*|x2 – x3|

=1/2 *(|x1 – x2| + |x1 – x3| + |x2 – x3|)

如此编程实现即可。

7. 6,9,140 可以组合成大于 N 的所有数请问 N 最小为？

美团 2014 中科院笔试。

美团有三种糖果，分别装有 6、9、140 颗糖果，当给一般用户装糖时必须要分拆，比如要装 8 颗糖果，但是当需要的糖果大于 N 时，将不必分拆，请问 N 为多少？

解：因 6 和 9 均可被 3 整除，从而>6 能被 3 整除的数均可由 6x+9y 表示；而 140%3=2；被 3 整除余 1 的数则必须要 2 个 140 的袋子，比如 280=93*3+1=140*2；之后的数如果%3=2，则减去 140，剩下的数可以被 3 整除；如果%3=1，则减去 140*2，剩余的数用 6 和 9 表示。从而 2*140+6=286；

而 285%3=0；284%3=2=140+48*3；而 283%3=1=140*2+3 无法单独表示 3。

故而 N=283.

8. 判断一个点位于一个多边形的内部？

美团 2014 北邮笔试。

过该点以平行于 X 轴划一条直线，如果两端与多边形的交点数均为奇数，则在多边形内部，否则为外部。

这个问题可以这样考虑，当我们进入一个封闭空间（多边形）时，如果在外部，每穿过一条线进入，必然要穿过一条线穿出，所以最终的结果必然是偶数。如果在内部，则必然是奇数。

9. 求连续数组的最大乘积

美团 2014 北邮笔试。

10.台阶接水问题

用一组数组表示一组台阶的高度，两个高的台阶中间的低洼处可以盛水，求一个台阶序列最多可以盛多少水？(美团 2014 中科院笔试)

11.最小交集

在 100 个人中，75%的人有个人电脑，68%的人有洗衣机，85%的人有电冰箱，80%的有录像机。请问，至少有多少人同时拥有以上四件物品？

100=15+20+25+32+8

没有 A 的人组成的集合 A、没有 B 的集合 B、没有 C 的集合 C、没有 D 的集合 D

所有人全集 W

有四种东西的人 x=W-(AUBUCUD)

为了使 x 最少，应该使 A B C D 的交集尽量的小。具体到这个问题 可以取 他们的交集为零，从而并集的元素数是 15+20+25+32.

12.概率问题

1) 生日悖论之二

某大公司有这么一个规定：只要有一个员工过生日，当天所有员工全部放假一天。但在其余时候，所有员工都没有假期，必须正常上班。这个公司需要雇用多少员工，才能让公司一年内所有员工的总工作时间期望值最大？

假设一年有 365 天，每个员工的生日都概率均等地分布在这 365 天里。

你的第一感觉或许是，公司应该雇用 100 多人，或者 200 多人吧。答案或许会让你大吃一惊：公司应该雇用 365 个人。注意，雇用 365 个人并不意味着全体员工全年的总工作时间为 0 ，因为 365 个人的生日都是随机的，恰好每天都有一个人过生日的概率极小极小。下面我们就来证明，这个问题的最优解就是 365 人。

由于期望值满足线性关系（即对于随机变量 X 和 Y 有 E(X) + E(Y) = E(X+Y) ），因此我们只需要让每一天员工总工作时间的期望值最大就可以了。假设公司里有 n 个人，那么在特定的一天里，没有人过生日的概率是 (364/365)^n 。因此，这一天的期望总工作时间就是 n*(364/365)^n 个工作日。为了考察函数 n*(364/365)^n 的增减性，我们来看一下 ((n+1)*(364/365)^n+1) / (n*(364/365)^n) 的值，它等于 (364*(n+1)) / (365^n) 。如果分子比分母小，解得 n > 364 。可见，要到 n = 365 以后，函数才是递减的。

2) 升级概率问题

英雄升级，从 0 级升到 1 级，概率 100%。

从 1 级升到 2 级，有 1/3 的可能成功；1/3 的可能停留原级；1/3 的可能下降到 0 级；

从 2 级升到 3 级，有 1/9 的可能成功；4/9 的可能停留原级；4/9 的可能下降到 1 级。

每次升级要花费一个宝石，不管成功还是停留还是降级。

求英雄从 0 级升到 3 级平均花费的宝石数目。（网易游戏笔试）

详细分析:http://www.cppblog.com/sosi/archive/2013/08/28/193719.html

从 0 到 1 所需要的期望步数很简单就是 1.

从 1 到 2 的期望步数假设是 X ，则在走了一步之后，我们分情况讨论所处位置。可以列出 X = 1/3*1+ 1/3*(X+1) + 1/3*(2+X) (即有 1/3 的可能升级成功，1/3 的可能维持原状此时浪费一个宝石，还有 1/3 的可能降级，此时浪费两个宝石因为还要回到 2) 所以 X=4 ,从 1 都 2 的期望步数是 4.

同理，从 2-3 时：X = 1/9+ 4/9*(X+1) + 4/9*(5+X) X =25. 从 2 到 3 的期望步数是 25. 所以从 0 到 3 的期望步数是 30.

3) 碰撞概率

美团的销售专家需要拜访商家，假设本市总共有 M 家商家，美团派出 N 位专家来拜访本市的商家，假设专家之间互不通信，每位专家随机选择 K 个商家，如果有多位专家拜访了一个商家，我们称之为“拜访碰撞”。问发生拜访碰撞的概率是多少？

如果 N*K>M,则一定会发生碰撞，P=1;

如果 N*K=M,发生碰撞的概率为=1-未碰撞的概率

=1-C(M,K)*C(M-K,K)*...*C(K,K)/(C(M,K)^N)

如果 N*K<M,发生碰撞的概率为=1-未碰撞的概率

=1-C(M,K)*C(M-K,K)*...*C(M-(N-1)*K,K)/(C(M,K)^N)

4) 布丰投针问题

维基：https://zh.wikipedia.org/wiki/%E5%B8%83%E8%B1%90%E6%8A%95%E9%87%9D%E5%95%8F%E9%A1%8C

BD2014.平面上有相距为 t 的若干条平行线，一根长为 l 的针(l<t),问向该平面投针，与平行线相交的概率是多少？并用程序模拟求的圆周率 pi。百度 2014 笔试。(另外题目为 SIFT/OSI 七层协议/进程共享内存/输出组合数及二叉搜索/人脸检测 CNN 等)

据维基可得：设 x 为针的中心和最近的平行线的距离，θ为针和线之间的锐角。则 x 在 [0,t/2]间均匀分布，且概率密度函数为 2/t dx 同理θ的概率密度函数为 2/pi dθ，两个随机变数互相独立，因此两者结合的机率密度函数只是两者的积：

4/(pi*t)dxdθ。

当 x<L*sin(θ)/2 时，针和线相交。从而求上式的积分，得针与线相交的机率：

$$P=\int_0^{\frac{\pi}{2}}\int_0^{\frac{l\sin(\theta)}{2}}\frac{4}{\pi t}dx d\theta=\frac{2l}{\pi t}$$

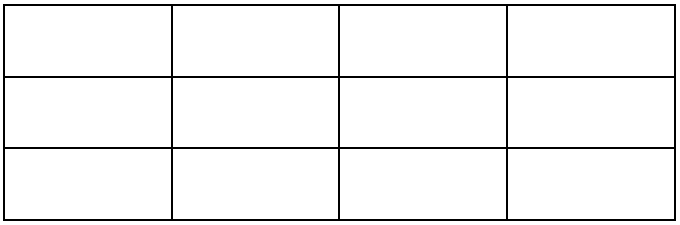
由此可求得π为 2*l/(p*t);

用程序模拟：主要是生成随机数用以随机 X 与θ两个变量。

```
double simulationPI(double t, double l)
//t:distance of the parallel lines    l:length of the pin
{
 assert(l<t);
 int testRounds=900000000;
 double pi=0.0;
 int intersection(0);
 for (int i=0;i<testRounds;i++)
 {
        double sita= 90*(double)rand()/((double)(RAND_MAX)) ;//0~90°
        sita=sita*PI/180;
        double x=rand()/((double)(RAND_MAX)/(t/2));
        if (x<l*sin(sita)/2){intersection++;}
 }
 double p=(double)intersection/testRounds;
 return 2*l/(p*t);
}
```

5) 概率组合示例

(1) **3×4grid** 有多少个 rectangular?



1(1×1)个小矩形：12

2(1×2 / 2×1)个组合：9+8 =17

3(1×3 / 3×1)个组合：4+6 =10

4(1×4 / 2×2)个组合：3+6=9

6(3×2 / 2×3)个组合：3+4 =7

8(2×4)个组合：2

9(3×3)个组合：2

12(3×4)个组合：1

共有:60 个组合

(2) 同一平面内 N 条非平衡不多点相交直线可以将平面分成多少部分?

（n^2 + n +2）/2

(3) 设在 n 进制下，下面的等式成立，567*456=150216，n 的值是（）.

依题意，在 n 进制下， 567=5*(n^2)+6*n+7；

456=4*(n^2)+5*n+6；

150216=n^5+5*(n^4)+2*(n^2)+n+6

(5*(n^2)+6*n+7)*(4*(n^2)+5*n+6)=n^5+5*(n^4)+2*(n^2)+n+6，n=18 时等式成立。

(4) AB 两人参加一个在 6 点到 8 点开的会，但是因为时间紧迫，AB 都只选自己感兴趣的部分听，其中 A 会参加一个小时的会，B 会参加半个小时的会。那么 AB 遇到的概率有多大?

A 要参加一个小时的会，则其到达时间需要满足：0<X<1 (1)

同理，B 要参加半个小时的会，其到达时间： 0<Y<1.5 (2)

如果 AB 相遇，则 -1.5<x-y<0.5 (3)

(3)式占(1)(2)式所构成的方形的面积即是相遇的概率，5/6

13.排列组合问题

1) 组合

求解组合数。 BD2014

```
int combine(int a[], int n, int m)
{
 int* order = new int[m+1];
 for(int i=0; i<=m; i++)
       order[i] = i-1;
 int count = 0,k = m;
 bool flag = true;
 while(order[0] == -1){
       if(flag) {
       for(int i=1; i<=m; i++)
             cout << a[order[i]] << " ";
cout << endl;
count++;
flag = false;}

order[k]++;
if(order[k] == n){
       order[k--] = 0;
       continue;}
if(k < m){
```

```
            order[++k] = order[k-1];                }
            continue;}                              delete[] order;
    if(k == m)                                      return count;
            flag = true;                            }
```

2) 全排列

用 123 来示例下。123 的全排列有 123、132、213、231、312、321 这六种。首先考虑 213 和 321 这二个数是如何得出的。显然这二个都是 123 中的 1 与后面两数交换得到的。然后可以将 123 的第二个数和每三个数交换得到 132。同理可以根据 213 和 321 来得 231 和 312。因此可以知道——全排列就是从第一个数字起每个数分别与它后面的数字交换。

详见：http://www.cnblogs.com/bakari/archive/2012/08/02/2620826.html

http://blog.csdn.net/e3399/article/details/7543861

全排列的递归实现

```
#include <stdio.h>
#include <string.h>
void Swap(char *a, char *b)
{
    char t = *a;
    *a = *b;
    *b = t;
}
//k 表示当前选取到第几个数,m 表示共有多少数.
void AllRange(char *pszStr, int k, int m)
{
    if (k == m)
    {
        static int s_i = 1;
        printf("  第%3d 个排列\t%s\n", s_i++, pszStr);
    }
    else
    {
  for (int i = k; i <= m; i++)
//第 i 个数分别与它后面的数字交换就能得到新的排列
   {
            Swap(pszStr + k, pszStr + i);
            AllRange(pszStr, k + 1, m);
            Swap(pszStr + k, pszStr + i);
    }
  }
```

```
}
void Foo(char *pszStr)
{
    AllRange(pszStr, 0, strlen(pszStr) - 1);
}
```

对于 n 个不同的数，上述方法可以输出 n!种结果，但是如果输入有重复，上述程序则会出现重复，此时需要**去重全排列**。

```
bool IsSwap(char *pszStr, int nBegin, int nEnd)
{
    for (int i = nBegin; i < nEnd; i++)
        if (pszStr[i] == pszStr[nEnd])
            return false;
    return true;
}
void Perm(char *pszStr, int k, int m)
{
    if (k == m)
    {
        static int s_i = 1;
        cout<<"第"<<s_i ++<<"个排列"<<pszStr<<endl;
    }
    else
    {
        for (int i = k; i <= m; i++)
        //第 i 个数分别与它后面的数字交换就能得到新的排列
        {
            if (IsSwap(pszStr, k, i))    //添加的判断语句，判断是否相等
            {
                swap(pszStr[k], pszStr[i]);
                Perm(pszStr, k + 1, m);
                swap(pszStr[k], pszStr[i]);
            }
        }
    }
}
```

非递归的全排列：

算法如下：

与上述 3)类似，已知序列 P =A1A2A3.....An 对 P 按字典**排序**，得到 P 的**最小排列** Pmin = A1A2A3....An 满足 Ai >= A(i-1) (1 < i <= n)从 **Pmin 开始,找到刚好比 Pmin 大的一个排列 P(min+1)**，再找到刚好比 P(min+1)大的一个排列，如此重复。

(1) 从后向前（即从 An->A1）,找到第一对为升序的相邻元素，即 Ai < A(i+1)。若找不到这样的 Ai，说明已经找到最后一个全排列，可以返回了。

(2) 从后向前,找到第一个比 Ai 大的数 Aj，交换 Ai 和 Aj。

(3) 将排列中 A(i+1)A(i+2)....An 这个序列的数逆序倒置，即 An.....A(i+2)A(i+1)。因为由前面第 1、2 可以得知，A(i+1)>=A(i+2)>=.....>=An,这为一个升序序列，应将该序列逆序倒置，所得到的新排列才刚刚好比上个排列大。

(4) 重复步骤 1-3,直到返回。

```
1.  //交换数组 a 中下标为 i 和 j 的两个元素的值
2.  void swap(int *a,int i,int j)
3.  {
4.      a[i]^=a[j];  a[j]^=a[i]; a[i]^=a[j];
5.  }
6.  //将数组 a 中的下标 i 到下标 j 之间的所有元素逆序倒置
7.  void reverse(int a[],int i,int j)
8.  {
9.      for(; i<j; ++i,--j)
10.         swap(a,i,j);
11. }
12. void print(int a[],int length)
13. {
14.     for(int i=0; i<length; ++i)
15.         cout<<a[i]<<" ";
16.     cout<<endl;
17. }
18. //求取全排列,打印结果
19. void combination(int a[],int length)
20. {
21.     if(length<2) return;
22.     bool end=false;
23.     while(true) {
24.         print(a,length);
25.         int i,j;
26.         //找到不符合趋势的元素的下标 i
27.         for(i=length-2; i>=0; --i) {
28.             if(a[i]<a[i+1]) break;
29.             else if(i==0) return;
30.         }
31.         for(j=length-1; j>i; --j) {
```

```
            if(a[j]>a[i]) break;
        }
        swap(a,i,j);
        reverse(a,i+1,length-1);
    }
}
```

递归情况下各种数字组合排列问题

详见：http://blog.sina.com.cn/s/blog_63e908970100vddo.html

3) 错排问题

错排问题是组合数学中的问题之一。考虑一个有 n 个元素的排列，若一个排列中所有的元素都不在自己原来的位置上，那么这样的排列就称为原排列的一个错排。n 个元素的错排数记为 Dn。 研究一个排列错排个数的问题，叫做错排问题。

对于排列数较多的情况，难以采用枚举法。这时可以用递归思想推导错排数的递推公式。显然 D1=0，D2=1。当 n≥3 时，不妨设 n 排在了第 k 位，其中 k≠n，也就是 1≤k≤n-1。那么我们现在考虑第 n 位的情况。

- 当 k 排在第 n 位时，除了 n 和 k 以外还有 n-2 个数，其错排数为 Dn-2。
- 当 k 不排在第 n 位时，那么将第 n 位重新考虑成一个新的“第 k 位”，这时的包括 k 在内的剩下 n-1 个数的每一种错排，都等价于只有 n-1 个数时的错排（只是其中的第 k 位会换成第 n 位）。其错排数为 Dn-1。

所以当 n 排在第 k 位时共有 Dn-2+Dn-1 种错排方法，又 k 有从 1 到 n-1 共 n-1 种取法，我们可以得到：Dn=(n-1)(Dn-1+Dn-2) 由此递归式即可求得错排概率为 Dn/n!.

From: http://zh.wikipedia.org/wiki/%E9%94%99%E6%8E%92

约等于: $$D_n = \left[\frac{n!}{e} + 0.5\right].$$

4) 输入 N，输出对应的所有长度为 N 的二进制串

暴风 2012 笔试。输入 2，输出 00 01 10 11

```
void print(unsigned int n)
{
    int max = pow((double)2,(int)n);
    unsigned int  MASK = 0x00000001;
    stack<int> s;
    for(unsigned int i=0; i<max; i++)
    {
        for(int j=0; j<n; j++)
```

```
        {
            s.push(((i & MASK)==MASK) ? 1:0);
            MASK=MASK<<1;
        }
        while(s.size())
        {
            cout << s.top();
            s.pop();
        }
        MASK=MASK>>n;
        cout << endl;
    }
}
```

```
void binary_output(int n)
{
    char a[100];
    int pa=0;
    for (int i=0;i<n;i++)
        a[i]=47;//表示'0'的前一个 ASCII 码
    while (pa>=0)
    {
        if(pa>=n)
        {
            a[pa]=0;
            cout<<a<<endl;
            pa--;
            continue;
        }
        if(a[pa]>='1')
        {
            a[pa--]=47;
            continue;
        }
        a[pa]++;
        pa++;
    }
}
```

5) 输入 56，输出 11-16 21-26…51-56

```
int combineNUM()
{
	const int N=100;
	char a[N]={0},b[N]={0};int i=0;
	cout<<"input num:";
	while((a[i]=getchar())!='\n')
	{
		b[i]='1';
		i++;
	}
	int num=i;
	bool isstop=0,isBigger=0;
	while(!isstop)
	{
		for (int j=num-1;j>=0;j--)
		{
			if (b[j]>a[j])
			{
				isBigger=1;
				if (j==0)
				{
					isstop=1;
					break;
				}
				b[j-1]++;
				b[j]='1';
			}
		}
		if (isBigger==1)
		{
			isBigger=0;
			continue;
		}
		for (int jj=0;jj<num;jj++)
		{
			cout<<b[jj];
		}
		cout<<'\n';
		b[num-1]++;
	}
```

```
 return 0;
}
```

6) 已知字符串里的字符是互不相同的，现在任意组合，比如 AB，则输出 AA，AB，BA，BB，编程按照字典序输出所有的组合

将 2) 改写即可得到。

```
char foundnext(char *a, char b)
{       int i=0;
 for (i=0;a[i];i++)
 {
        if (a[i]==b)
                break;
 }
 return a[i+1];}
int test_combineNUM()
{
 char a[100]= {0};
 char b[100]={0};
 int len=0;
 while((a[len]=getchar())!='\n')
        len++;
 b[len]='\0';
 a[len]='\0';
 qsort(a,len,sizeof(char),comp);
 bool isstop=false, isBigger=false;
 for (int ai=0;ai<len;ai++)
        b[ai]=a[0];
  int aindex=0;
 while(!isstop)
 {
        for (int j=len-1;j>=0;j--)
        {
                if (b[j]=='\0')
                {
                        isBigger=true;
                        b[j]=a[0];
                        aindex=0;
                        if (j==0)
                        {
                                isstop=true;
                                break;
                        }
                b[j-1] = foundnext(a,b[j-1]);
                }
        }
//isbigger is for the last time
        if (isBigger)
        {
                isBigger=false;
                continue;
        }
        for (int k=0;k<len;k++)
        {
                cout<<b[k];
        }
        cout<<endl;
        b[len-1]=a[++aindex];
 }
 return 0;
}
```

八． 手写代码

1. STRCPY 函数

检查是否非空，并注意最后应添加字符串结束符号 **'\0'**

2. ATOI

注意处理开头的空格及**正负号**问题

```
int strtoint(const char *src, int num)
{
    if (*src==NULL)
    {
        printf("source string Null");
        return -1;
    }
    int flag(0);
    while(*src)
    {
        if (*src==' ')
        {
            ;
        }
        else if (*src=='+')
        {
            flag =0;
        }
        else if (*src == '-')
        {
            flag = 1;
        }
        else if (*src>='0'&& *src<='9')
        {
            num=num*10 + *src-'0';
        }
        else
        {
            printf("异常字符");
            num=0;
            return num;
        }

        src++;
    }

    if (flag== 1)
    {
        num = ~num + 1;
    }
    return num;
}
```

3. ITOA(INTEL)

```
void myitoa(int num, string &s)
{
 int flag;
 if (!num)//对零的处理
 {
        s.push_back('0');
        return;
 }
 if (num>0)//对负数的处理方式
 {
        flag=0;
 }
 else
 {
        flag=1;
        num = ~num+1;
 }
 while (num)
 {
        int a = num%10;
        const char ch=a+'0';//将数字转为字符
        s.push_back(ch);
        num /=10;
 }
 if (flag)
 {
        s.push_back('-');
 }
 reverse(s.rbegin(),s.rend());//反向表示
}
```

4. 约瑟夫环 (INTEL)

约瑟夫斯(Josephus problem)问题，是一个出现在计算机科学和数学中的问题。在计算机编程的算法中，类似问题又称为约瑟夫环。http://en.wikipedia.org/wiki/Josephus_problem

有 n 个囚犯站成一个圆圈，准备处决。首先从一个人开始，越过 k-2 个人（因为第一个人已经被越过），并杀掉第 k 个人。接着，再越过 k-1 个人，并杀掉第 k 个人。这个过程沿着圆圈一直进行，直到最终只剩下一个人留下，这个人就可以继续活着。

问题是，给定了 n 和 k，一开始要站在什么地方才能避免被处决？

最简单的方法可以用动态规划的观点来解决。n(0,1…n-1)个人时取到的人必然来自于 n-1 取到的人身后第 k 个人。复杂度为 O(n)(当然也可采用环形链表进行朴素操作)也即：

$$g(n,k) = (g(n-1,k)+k) \bmod n, \text{ with } g(1,k) = 0$$

利用递推公式，如从 1 开始数则为：

$$f(n,k) = ((f(n-1,k)+k-1) \bmod n)+1, \text{ with } f(1,k) = 1,$$

```
if n ==1:
    return 0
else:
    return
((josephus(n-1,k)+k) % n)
```

```
def josephus(n, k):
  r = 0
  i = 2
while i <= n:
        r = (r + k) % i;
        i = i+1;
return r+1
```

对于 k=2 时，

return 2*(n – 2^(int(log(n,2))))+1

```
#define    TOTAL 15
int Joseph_intel(int N)
{
int a[TOTAL]={1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
int nextstart =0;
for (int i=1;i<TOTAL;i++)//循环 total-1 次  每一次排除一个结果
{int counter =0;
//nextstart = (nextstart+N)%(i+1);
while (counter<N)
{
        if (++counter == N )
        {
        a[nextstart]=0;
        }
    do
    {
    //略过为 0 的向量
nextstart = (nextstart+1)%TOTAL;
    } while (!a[nextstart]);
}
}
    return nextstart+1;
}
```

5. 二分查找函数

```
int binarysearch(int *srclist,int num, int begin, int end)
{
    int mid = (begin + end)/2;

    if (begin > end)
    {
        return -1;
    }

    if (num == *(srclist+mid))
    {
        return mid;
    }
    else if (num > *(srclist +mid))
    {
        binarysearch(srclist,num,mid+1,end);
    }
```

```
        else
        {
            binarysearch(srclist,num,0,mid-1);
        }
}
```

6. 实现栈或者树的建立查找删除销毁操作

7. 斐波那契数列

```
long fib[41] = {0,1};
int i;
for(i=2;i<41;i++)fib[i] = fib[i-1]+fib[i-2];
 for(i=1;i<41;i++)printf("F%d==%d\n",i,fib[i]);

 long fib(int n)
 {
 if (n==0) return 0;
 if (n==1) return 1;
 if (n>1) return fib(n-1)+fib(n-2);
 }
```

斐波那契数列递归方式的时间复杂度为 O(1.618^n);而递推方式的时间复杂度为 O(n);如果使用矩阵向量的方式来解答则可以达到时间复杂度为 O(logn).

链接：http://www.gocalf.com/blog/calc-fibonacci.html

8. 求两个数组中的相同元素

9. 查找一个中间大的数

（即除了该数外，一半比它大，另一半比它小）k=n/2 个数。

可以使用分治快速排序的方式找，首先寻找一个数，小的放在左边有 m 个，大的放在右边有 n-m-1 个。

如该数位于中间 m=k-1 则解决；

如 m<k-1 则从（m，n）中寻找第 k-1 个数；

如 m>k-1,则从（0，m）中寻找第 k-1 个数；

10.编写类 STRING 的构造析构赋值函数

已知类 String 的原型为：

```
class String
{
public:
String(const char *str = NULL);//普通构造函数
String(const String &other);//拷贝构造函数
~ String(void);// 析构函数
String & operate =(const String &other);// 赋值函数
private:
char* m_data;// 用于保存字符串
};
```

请编写 String 的上述 4 个函数。

标准答案：

```
// String 的析构函数
String::~String(void) // 3 分
{
delete [] m_data;
// 由于 m_data 是内部数据类型，也可以写成 delete m_data;
}

// String 的普通构造函数
String:: String(const char *str) // 6 分
{
if(str==NULL)
{
m_data = new char[1]; // 若能加 NULL 判断则更好
*m_data = '/0';
}
else
{
int length = strlen(str);
m_data = new char[length+1]; // 若能加 NULL 判断则更好
strcpy(m_data, str);
}
}

// 拷贝构造函数
String::String(const String &other) // 3 分
{
```

```
int length = strlen(other.m_data);
m_data = new char[length+1]; //  若能加 NULL  判断则更好
strcpy(m_data, other.m_data);
}

//  赋值函数
String & String::operate =(const String &other) // 13 分
{
// (1)  检查自赋值  // 4 分
if(this == &other)
return *this;
// (2)  释放原有的内存资源  // 3 分
delete [] m_data;
//  （）分配新的内存资源，并复制内容// 3 分
int length = strlen(other.m_data);
m_data = new char[length+1]; //  若能加 NULL  判断则更好
strcpy(m_data, other.m_data);
//  （）返回本对象的引用  // 3 分
return *this;
}
```

11.输入两个字符串，输出第二个字符串在第一个字符串中的位序

如输入 abdbcc   abc 输出 125 146 145 146

```
void PrintArray(char *pstr,char *sstr,int* printarr,int plen,int slen,int printarrnum,int
pstart,int sstart)
{
	int pstartNum = pstart;
	int sstartNum = sstart;
	int printNum= printarrnum;

	if (printNum==slen)
	{
		for (int i=0;i<slen;i++)
		{
			cout<<*(printarr+i);
			cout<<" ";
		}
		cout<<endl;
		return;
```

```
        }

        for (int i=pstartNum;i<plen;i++)
        {
            for (int j=sstartNum;j<slen;j++)
            {
                if (*(pstr+i)==*(sstr+j))
                {
                    printarr[printNum]=i+1;
                    pstartNum=i;
                    sstartNum=j;
        PrintArray(pstr,sstr,printarr,plen,slen,printNum+1,pstartNum+1,sstartNum+1);
                }
            }
        }
    }

    void connectSequence(char *pstr,char* sstr)//pstr 父串  sstr 子串
    {
        int plen=strlen(pstr);
        int slen=strlen(sstr);
        int *print_arr = new int[slen];
        unsigned int print_arr_num=0;
        if (pstr==NULL || sstr==NULL)
        {
            cout<<"string null"<<endl;
            return;
        }
        PrintArray(pstr,sstr,print_arr,plen,slen,0,0,0);
    }
```

12.方块寻径

编程计算一个 n×n 的方格中，由左上角到右下角一共有多少条不同的路径。例如上图 2×2 图中共有 6 条不同路径。暴风 2014 校招最后一题。

可以将此问题想象成一个由(0,0)到(n,n)的问题，每次增长的时候都有两种选择，左边增加或者右边增加，每次选择都以为着增加一种可能。

```
static int num=0;
```

```
void pathnum(int n,int a1=0,int a2=0)
{
 if (a1==n&&a2==n)
 {
        num++;
        return;
 }
 if (a1<=n&&a2<=n)
 {
        int a3=a1;
        int a4=a2;
        if (a1<n)
        {
              a3++;
              pathnum(n,a3,a2);
        }
        if (a2<n)
        {
              a4++;
              pathnum(n,a1,a4);
        }
 }
}
```

BF2014/RR2014 笔试。

13.实现积分图

九. 图像处理

1. RANSAC

RANSAC(RANdom SAmple Consensus), 随机抽样一致。它是一种迭代的方法，用来在一组包含离群(outlier)的被观测数据中估算出数学模型的参数。 RANSAC 是一个非确定性算法，在某种意义上说，它会产生一个在一定概率下合理的结果，其允许使用更多次的迭代来使其概率增加。

RANSAC 的基本假设是：

1. “内群”数据可以通过几组模型参数来叙述其数据分布，而“离群”数据则是不适合模型化的数据。

2. 数据会受杂讯影响,杂讯指的是离群，例如从极端的杂讯或错误解释有关数据的测量或不正确的假设。

3. RANSAC 假定，给定一组（通常很小）的内群，存在一个程序，这个程序可以估算最佳解释或最适用于这一数据模型的参数。

RANSAC 概述：

1. 在数据中随机选择几个点设定为内群
2. 计算适合内群的模型
3. 把其它刚才没选到的点带入刚才建立的模型中，计算是否为内群
4. 记下内群数量
5. 重复以上步骤多做几次
6. 比较哪次计算中内群数量最多,内群最多的那次所建的模型就是我们所要求的解

这里有几个需要考虑的问题，一开始的时候我们要随机选择多少点以及要重复做多少次

2. MEAN SHIFT

Mean Shift 偏移均值向量，一般是指一个迭代的步骤,即先算出当前点的偏移均值,移动该点到其偏移均值,然后以此为新的起始点,继续移动,直到满足一定的条件结束.可以用于图像平滑、分割及跟踪等。

一般的说来,离 x 越近的采样点对估计 x 周围的统计特性越有效,因此引进核函数的概念,在计算时可以考虑距离的影响;同时我们也可以认为在这所有的样本点中,重要性并不一样,因此我们对每个样本都引入一个权重系数.

$$M(x) \equiv \frac{\sum_{i=1}^{n} G_H(x_i - x) w(x_i)(x_i - x)}{\sum_{i=1}^{n} G_H(x_i - x) w(x_i)}$$

首先，我们对某一点求其平均偏移向量并将其作为其结果，以该结果为起点继续做如上变换知道结果收敛为止。此时即是最终的结果。

3. EM 算法

d

d

4. 分类

1) SVM

[理解 SVM 的三重境界]