目录

1. 软件工程和计算机有什么区别？

1) 基础课程重复度较高。
2) 计算机偏学术研究，软件工程偏工程实践。一般来说计算机的学习偏重学习计算机的原理。学习偏理论，学习内容涉及软件也涉及硬件。软件工程，简称(SE)。SE 的学习主要是围绕着软件的应用、设计、开发、维护架构这几个模块等，偏应用、工程、实践，学习内容涉及一些基本的硬件，但更多是工程的理论和大量的软件实践知识。
3) 软件工程培养计划里面一般有项目管理，架构，测试等科目。

2. 算法的基本特征和复杂度

（1）基本特征

输入、输出、有穷性、确定性、可行性

（2）算法的复杂度

详见笔记 03.

3. 用循环比递归效率高吗？

递归和循环两者完全可以互换。不能完全决定性地说循环地效率比递归的效率高。

（1）递归算法

- 优点：代码简洁、清晰，并且容易验证正确性。
- 缺点：它的运行需要较多次数的函数调用，如果调用层数比较深，需要增加额外的堆栈处理（还有可能出现**堆栈溢出**的情况），比如参数传递需要压栈等操作，会对执行效率有一定影响。但是，对于某些问题，如果不使用递归，那将是极端难看的代码。在编译器优化后，对于多次调用的函数处理会有非常好的效率优化，效率未必低于循环。

（2）循环算法

- 优点：速度快，结构简单。
- 缺点：并不能解决所有的问题。有的问题适合使用递归而不是循环。如果使用循环并不困难的话，最好使用循环。

4. P 问题、NP 问题、NP 完全问题的概念

1) P (polynominal)问题-多项式问题

- 存在多项式时间算法的问题。

2) NP (Nondeterministic Polynominal)问题-非确定多项式问题

- **能在多项式时间内验证得出一个正确解的问题。**
- 关于 P 是否等于 NP 是一个存在了很久的问题，这里不做讨论。
- 通俗的理解这两个问题的话：在借助计算机的前提下，P 问题很容易求解；NP 问题不容易求解，但对于某一答案我们可以很快验证这个答案是否正确。

举例：

[1] 最简单，最基本的：枚举集合 S 的所有子集的问题

[2] 旅行推销员问题

[3] N 皇后问题

[4] 背包问题（是一种组合优化的 NP 完全问题。问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。问题的名称来源于如何选择最合适的物品放置于给定背包中。）

3) NPH (Nondeterminism Polynomial Hard)问题-NP 难问题

- 它不一定是一个 NP 问题
- 其他属于 NP 的问题都可在多项式时间内归约成它。通俗理解，NP 难问题是比所有 NP 问题都难的问题。

4) NPC (Nondeterminism Polynomial Complete)问题-NP 完全问题

- 它是一个 NP 问题
- 其他属于 NP 的问题都可在多项式时间内归约成它。
- 通俗理解，NP 完全问题是介于 NP 问题和 NP 难问题之间的一类问题。

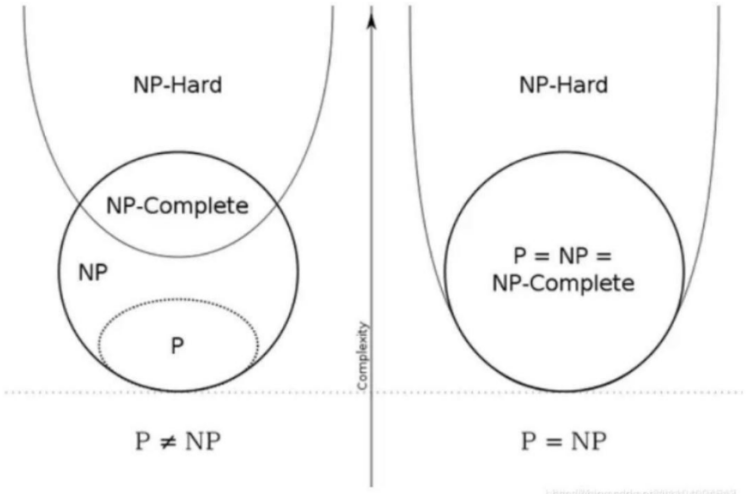


图 1　各问题的关系

5. 几种树

（1）二叉搜索树 (binary search tree)

详见笔记 74.

二叉搜索树满足的条件：

1. 非空左子树的所有键值小于其根节点的键值
2. 非空右子树的所有键值大于其根节点的键值
3. 左右子树都是二叉搜索树

（2）AVL 树 (Self-balancing binary search tree)

AVL 树又称为高度平衡的二叉搜索树。一棵 AVL 树或者是空树，或者是具有下列性质的二叉搜索树：它的左子树和右子树都是 AVL 树，且左子树和右子树的高度之差的绝对值不超过 1。

（3）红黑树(red-black tree)

红黑树是这样的一棵二叉搜索树：数中的每一个结点的颜色不是黑色就是红色。可以把一棵红黑树视为一棵扩充二叉树，用外部结点表示空指针。其特性描述如下：

- 特性 1：根结点和所有外部结点的颜色是黑色。
- 特性 2：从根结点到外部结点的途中没有连续两个结点的颜色是红色。
- 特性 3：所有从根到外部结点的路径上都有相同数的黑色结点。

从红黑树中任一结点 x 出发（不包括结点 x），到达一个外部结点的任一路径上的黑结点个数叫做结点 x 的黑高度，亦称为结点的阶（rank），记作 bh(x)。红黑树的高度定义为其根结点的黑高度。

对普通二叉搜索树进行搜索的时间复杂度为 O(h)，对于红黑树则为 O(log2n)。

（4）B 树（特例：2-3 树、2-3-4 树）

一棵 m 阶 B 树(balanced tree of order m)是一棵平衡的 m 路搜索树，它或者是空树，或者是满足下列性质的树：

①根结点至少有两个子女。

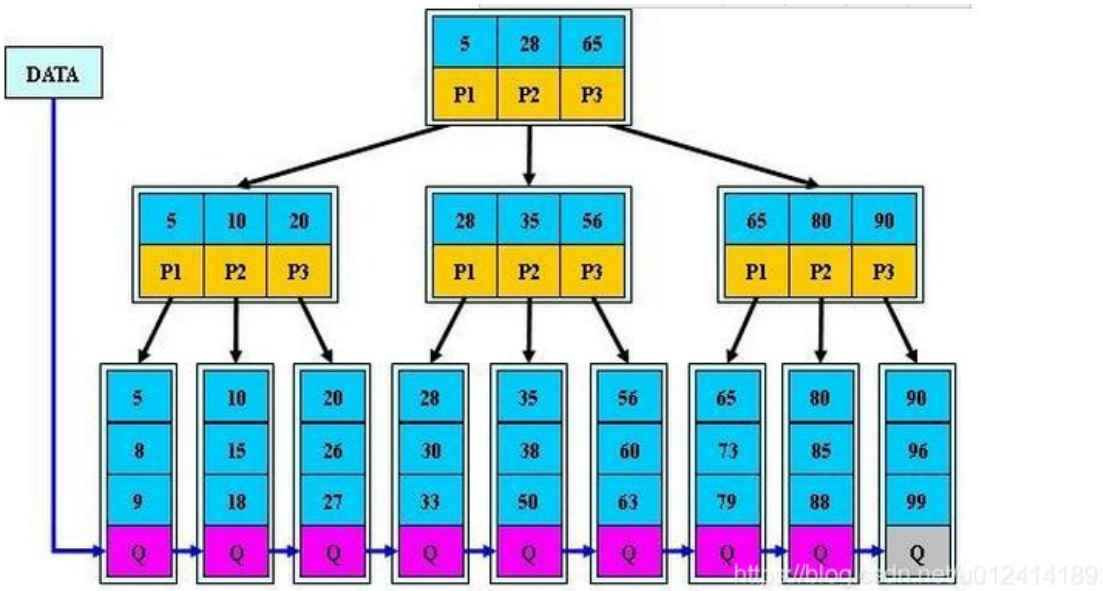
②除根结点以外的所有结点（不包括失败节点）至少有$\lceil m/2 \rceil$个子女。

③所有的失败结点都位于同一层。

（5）B+树

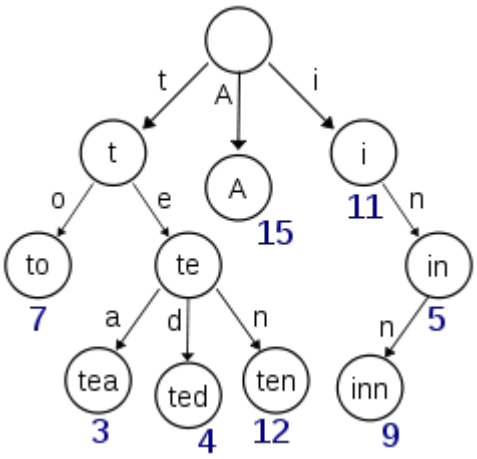
B+树是 B 树的一个升级版，B+树是 B 树的变种树，有 n 棵子树的节点中含有 n 个关键字，**每个关键字不保存数据，只用来索引，数据都保存在叶子节点**。是为文件系统而生的。

相对于 B 树来说 B+树更充分的利用了节点的空间，让查询速度更加稳定，其速度完全接近于二分法查找。



- 特点：在 B 树的基础上每个节点存储的关键字数更多，树的层级更少所以查询数据更快，所有指关键字指针都存在叶子节点，所以每次查找的次数都相同所以查询速度更稳定。
- 应用场景： 用在磁盘文件组织 数据索引和数据库索引。

（6）Trie 树（字典树）



Trie，又称前缀树，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。

6. 邻接矩阵与邻接表

- 邻接矩阵表示法：在一个一维数组中存储所有的点，在一个二维数组中存储顶点之间的边的权值
- 邻接表表示法：图中顶点用一个一维数组存储，图中每个顶点 vi 的所有邻接点构成单链表

对比

1） 在邻接矩阵表示中，无向图的邻接矩阵是对称的。矩阵中第 i 行或 第 i 列有效元素个数之和就是顶点的度。在有向图中第 i 行有效元素个数之和是顶点的出度，第 i 列有效元素个数之和是顶点的入度。

2） 在邻接表的表示中，无向图的同一条边在邻接表中存储的两次。如果想要知道顶点的度，只需要求出所对应链表的结点个数即可。有向图中每条边在邻接表中只出现一次，求顶点的出度只需要遍历所对应链表即可。求入度则需要遍历其他顶点的链表。

3） 邻接矩阵与邻接表优缺点

- 邻接矩阵的优点是可以快速判断两个顶点之间是否存在边，可以快速添加边或者删除边。而其缺点是如果顶点之间的边比较少，会比较浪费空间。因为是一个 n*n 的矩阵。
- 邻接表的优点是节省空间，只存储实际存在的边。其缺点是关注顶点的度时，就可能需要遍历一个链表。

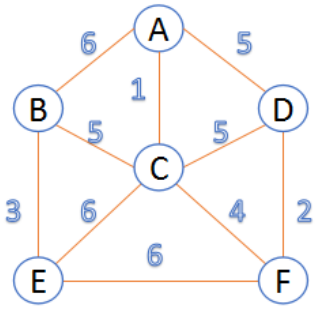
7. 最小生成树(Minimum Spanning Tree, MST)

一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图连通的最小权值的边。最小生成树可以用 kruskal（克鲁斯卡尔）算法或 prim（普里姆）算法求出。

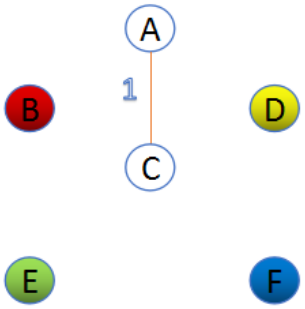
（1）Kruskal（克鲁斯卡尔）算法

此算法可以称为“加边法”，初始最小生成树边数为 0，每迭代一次就选择一条满足条件的最小代价边，加入到最小生成树的边集合里。

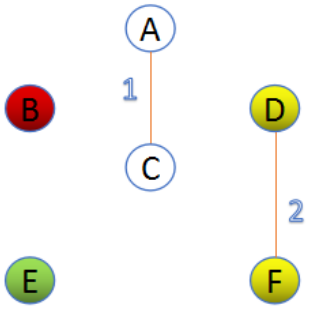
针对边展开，稀疏图有优势。



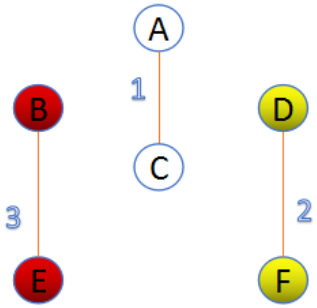
连通网G



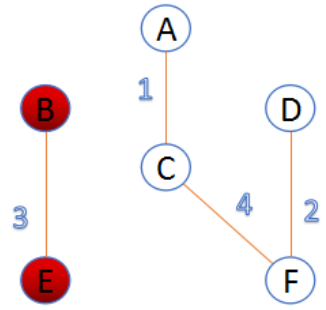
1.选择代价最小的边(A,C);并保证A,C不在同一颗树上,然后合并A,C



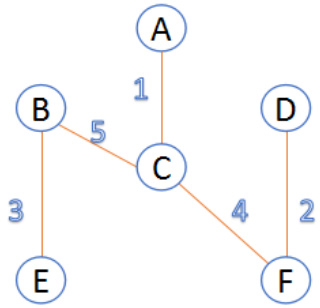
2.选择代价最小的边(D,F);并保证D,F不在同一颗树上,然后合并D,F



3.选择代价最小的边(B,E);并保证B,E不在同一颗树上,然后合并B,E



4.选择代价最小的边(C,F);并保证C,F不在同一颗树上,然后合并C,F所在的树

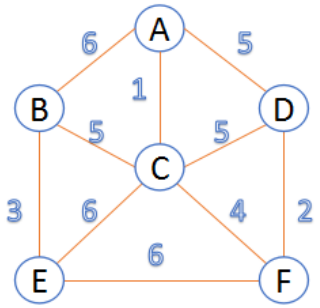


5.选择代价最小的边(A,D),顶点A,D在同一颗树上,丢弃;
选择最小的边(C,D),顶点C,D在同一颗树上,丢弃;
选择最小的边(B,C),顶点B,C不在同一颗树上,加入此边,然后合并B,C所在的树,此时所有顶点在同一颗树上，返回;

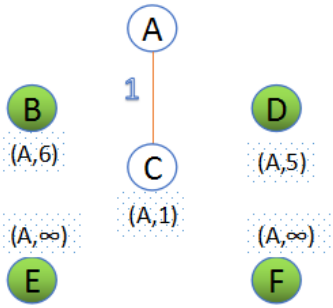
（2）Prim（普里姆）算法

此算法可以称为“加点法”，每次迭代选择代价最小的边对应的点，加入到最小生成树中。算法从某一个顶点 s 开始，逐渐扩大到覆盖整个连通网的所有顶点。

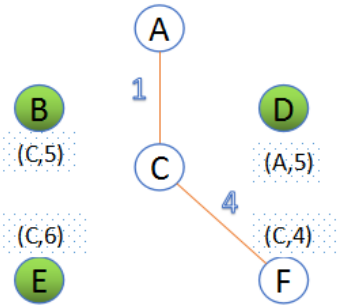
针对顶点展开，稠密图有优势。



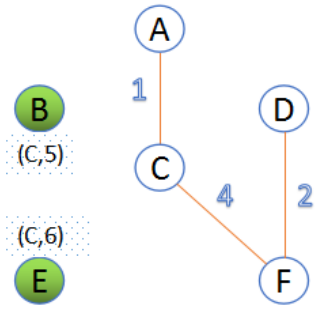
连通网G



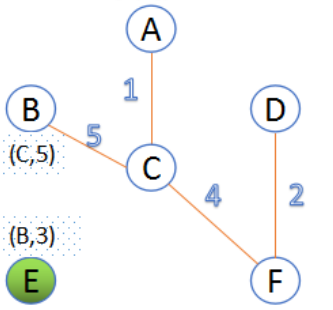
1.初始u={A},v={B,C,D,E,F};顶点B下方(A,6),表示与集合u中A的代价为6作为最小代价边。选择最小的代价边(A,C),把C并入到集合u中。



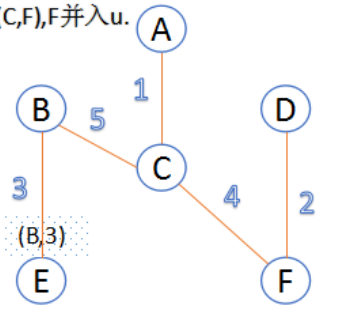
2.u={A,C},v={B,D,E,F};更新v中顶点与集合u的最小的代价边;例如:顶点E之前为(A,∞),更新为(C,6)；选择最小代价边(C,F),F并入u.



3.u={A,C,F},v={B,D,E};更新v中顶点与集合u的最小的代价边;选择最小代价边(F,D),D并入u.



4.u={A,C,F,D},v={B,E};更新v中顶点与集合u的最小的代价边;选择最小代价边(C,B),B并入u.



5.u={A,C,F,D,B},v={E};更新v中顶点与集合u的最小的代价边;选择最小代价边(B,E),E并入u.

8. 最短路径算法(Shortest Path Algorithm)

（1）Dijkstra 算法——从某一源点到其余各顶点的最短路径，O(n^2)

算法特点：

- 每次找出前一次迭代后具有最低费用的节点，添加到集合中。
- 第 k 次迭代后，可以知道源点到 k 个目的节点的最低费用路径。

（2）Floyd 算法——每一对顶点之间的最短路径，O(n^3)

基本思想：最开始只允许经过 1 号顶点进行中转，接下来只允许经过 1 号和 2 号顶点进行中转.....允许经过 1~n 号所有顶点进行中转，来不断动态更新任意两点之间的最短路程。即求从 i 号顶点到 j 号顶点只经过前 k 号点的最短路程。

（3）深度或广度优先搜索算法（解决单源最短路径）

9. 深度优先搜索（Depth-First-Search, DFS）和广度优先搜索（Breadth-First-Search, BFS）

（1）深度优先搜索（DFS）

深度优先搜索在搜索过程中访问某个顶点后，需要**递归地访问此顶点的所有未访问过的相邻顶点**。

初始条件下所有节点为白色，选择一个作为起始顶点，按照如下步骤遍历：

a. 选择起始顶点涂成灰色，表示还未访问。

b. 从该顶点的邻接顶点中选择一个，继续这个过程（即再寻找邻接结点的邻接结点），一直深入下去，直到一个顶点没有邻接结点了，涂黑它，表示访问过了。

c. 回溯到这个涂黑顶点的上一层顶点，再找这个上一层顶点的其余邻接结点，继续如上操作，如果所有邻接结点往下都访问过了，就把自己涂黑，再回溯到更上一层。

d. 上一层继续做如上操作，直到所有顶点都访问过。

（2）广度优先搜索（BFS）

广度优先搜索在进一步遍历图中顶点之前，**先访问当前顶点的所有邻接结点**。

a. 首先选择一个顶点作为起始结点，并将其染成灰色，其余结点为白色。

b. 将起始结点放入队列中。

c. 从队列首部选出一个顶点，并找出所有与之邻接的结点，将找到的邻接结点放入队列尾部，将已访问过结点涂成黑色，没访问过的结点是白色。如果顶点的颜色是灰色，表示已经发现并且放入了队列，如果顶点的颜色是白色，表示还没有发现。

d. 按照同样的方法处理队列中的下一个结点。

基本就是出队的顶点变成黑色，在队列里的是灰色，还没入队的是白色。

10. 并查集(union-find set)

并查集顾名思义就是有"合并集合"和"查找集合中的元素"两种操作的关于数据结构的一种算法。

并查集，在一些有 N 个元素的集合应用问题中，我们通常是在开始时让每个元素构成一个单元素的集合，然后按一定顺序将属于同一组的元素所在的集合合并，其间要反复查找一个元素在哪个集合中。

并查集是一种**树形的数据结构**，用于**处理一些不相交集合（Disjoint Sets）的合并及查询问题**。

并查集也是使用树形结构实现。不过，不是二叉树。每个元素对应一个节点，每个组对应一棵树。在并查集中，哪个节点是哪个节点的父亲以及树的形状等信息无需多加关注，整体组成一个树形结构才是重要的。类似森林。

11. 什么是稳定性排序？为什么排序要稳定？

如果两个具有相同键的对象以相同的顺序出现在排序输出中，则排序算法是稳定的。（假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i]=r[j]，且 r[i]在 r[j]之前，而在排序后的序列中，r[i]仍在 r[j]之前，则称这种排序算法是稳定的；否则称为不稳定的。）

稳定性的好处：**排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。**基数排序就是这样，先按低位排序，逐次按高位排序，低位相同的元素其顺序再高位也相同时是不会改变的。

12. 冒泡排序

冒泡排序顾名思义就是整个过程像气泡一样往上升，单向冒泡排序的基本思想是（假设由小到大排序）：对于给定 n 个记录，从第一个记录开始依次对相邻的两个记录进行比较，当前面的记录大于后面的记录时，交换位置，进行一轮比较和换位后，n 个记录的最大记录将位于第 n 位，然后对前（n-1）个记录进行第二轮比较；重复该过程，直到记录剩下一个为止。

13. 快速排序算法

（1）请你介绍一下快排算法

根据哨兵元素，用两个指针指向待排序数组的首尾，首指针从前往后移动找到比哨兵元素大的，尾指针从后往前移动找到比哨兵元素小的，交换两个元素，直到两个指针相遇，这是一趟排序，经常这趟排序后，比哨兵元素大的在右边，小的在左边。经过多趟排序后，整个数组有序。

稳定性：不稳定

平均时间复杂度：O(nlogn)

（2）简述快速排序过程

[1] 选择一个基准元素，通常选择第一个元素或者最后一个元素。

[2] 通过一趟排序将待排序的记录分割成独立的两部分，其中一部分记录的元素值均比基准元素值小，另一部分记录的元素值均比基准元素值大。

[3] 此时基准元素在其排好序后的正确位置。

[4] 然后分别对这两部分记录用同样的方法继续进行排序，直到整个序列有序。

（3）快排是稳定的吗？

快排算法是不稳定的排序算法。例如：

待排序数组：int a[] ={1, 2, 2, 3, 4, 5, 6}；

若选择 a[2]（即数组中的第二个 2）为枢轴，而把大于等于比较子的数均放置在大数数组中，则 a[1]（即数组中的第一个 2）会到 pivot 的右边，那么数组中的两个 2 非原序。

若选择 a[1]为比较子，而把小于等于比较子的数均放置在小数数组中，则数组中的两个 2 顺序也非原序。

（4）快排算法最差情况推导公式

在快速排序的早期版本中呢，最左面或者是最右面的那个元素被选为枢轴，那最坏的情况就会在下面的情况下发生啦：

1）数组已经是正序排过序的。 （每次最右边的那个元素被选为枢轴）

2）数组已经是倒序排过序的。 （每次最左边的那个元素被选为枢轴）

3）所有的元素都相同（1、2 的特殊情况）

因为这些案例在用例中十分常见，所以这个问题可以通过要么选择一个随机的枢轴，或者选择一个分区中间的下标作为枢轴，或者（特别是对于相比更长的分区）选择分区的第一个、中间、最后一个元素的中值作为枢轴。有了这些修改，那快排的最差的情况就不那么容易出现了，但是如果输入的数组最大（或者最小元素）被选为枢轴，那最坏的情况就又来了。

快速排序，在最坏情况退化为冒泡排序，需要比较 O(n2)次（n(n - 1)/2 次）。

（5）其他关于快速排序的知识

以从小到大为例：

快速排序的基本思想是任取待排序序列的一个元素作为中心元素(可以用第一个，最后一个，也可以是中间任何一个)，习惯将其称为 pivot，枢轴元素；将所有比枢轴元素小的放在其左边，将所有比它大的放在其右边，形成左右两个子表；然后对左右两个子表再按照前面的算法进行排序，直到每个子表的元素只剩下一个。

可见快速排序用到了分而治之的思想。

将一个数组分成两个数组的方法为：先从数组右边找到一个比枢轴元素小的元素，将数组的第一个位置赋值为该元素；再从数组的左边找到一个比枢轴元素大的元素，将从上面取元素的位置赋值为该值；依次进行，直到左右相遇，把枢轴元素赋值到相遇位置。

快速排序之所比较快，因为相比冒泡排序，每次交换是跳跃式的。每次排序的时候设置一个基准点，将小于等于基准点的数全部放到基准点的左边，将大于等于基准点的数全部放到基准点的右边。这样在每次交换的时候就不会像冒泡排序一样每次只能在相邻的数之间进行交换，交换的距离就大的多了。因此总的比较和交换次数就少了，速度自然就提高了。当然在最坏的情况下，仍可能是相邻的两个数进行了交换。因此快速排序的最差时间复杂度和冒泡排序是一样的都是 O(N2)，它的平均时间复杂度为 O(NlogN)。

（6）解释下快排为什么快？不要说快排的什么复杂度或者算法过程，回答为什么快。（这问题我蒙蔽的一匹，最后听老师意思是说从存储中内存和硬盘读取数据频率那里谈，莫非是数据量太大的时候其他排序涉及到外排序、快排二分几次后就避免了外排序的硬盘交互问题？？）

（7）归并排序的最坏时间复杂度优于快排，为什么我们还是选择快排？

[1] 快排查找的常量要比归并小。

[2] 绝大多数情况下，快排遇到的都是平均情况，也就是最佳情况，只有极个别的时候会是最坏情况，因此往往不考虑这种糟糕的情况。

或：

[1] C++模板有很强的 inline 优化机制，比较操作相对于赋值（移动）操作要快的多（尤其是元素较大时）

[2] 另一方面，一般情况下，归并排序的比较次数小于快速排序的比较次数，而移动次数一般多于快速排序的移动次数，二者大约都是 2~3 倍的差距。

因为这样，在 C++中快排要比归并排序更快，但其实在 Java 中恰恰相反，移动（赋值）一般比比较快。

14. 快速排序和归并排序的优缺点

1) 快速排序的优缺点

- 优点：快，平均性能好，时间复杂度为 O(nlogn)。
- 缺点：不稳定，初始序列有序或基本有序时，时间复杂度降为 O(n^2)。

2) **归并排序的优缺点**

- 优点：快，时间复杂度为 O(nlogn)；是一种稳定的算法；是最常用的外部排序算法（当待排序的记录放在外存上，内存装不下全部数据时，归并排序仍然适用，当然归并排序同样适用于内部排序）。
- 缺点：空间复杂度高（需要 O(n)的辅助空间）。

15. 插入排序

（1）基本思想

每一步将一个待排序的数据插入到前面已经排好序的有序序列中，直到插完所有元素为止。

插入排序的工作方式像许多人排序一手扑克牌。开始时，我们的左手为空并且桌子上的牌面向下。然后，我们每次从桌子上拿走一张牌并将它插入左手中正确的位置。为了找到一张牌的正确位置，我们从右到左将它与已在手中的每张牌进行比较。拿在左手上的牌总是排序好的，原来这些牌是桌子上牌堆中顶部的牌[1] 。

插入排序是指在待排序的元素中，假设前面 n-1(其中 n>=2)个数已经是排好顺序的，现将第 n 个数插到前面已经排好的序列中，然后找到合适自己的位置，使得插入第 n 个数的这个序列也是排好顺序的。按照此法对所有元素进行插入，直到整个序列排为有序的过程，称为插入排序[3] **。**

（2）复杂度

插入排序的平均时间复杂度也是`O(n^2)`，空间复杂度为常数阶`O(1)`，具体时间复杂度和数组的有序性也是有关联的。

16. 希尔排序(Shell sort)

希尔排序(Shell's Sort)是插入排序的一种又称“缩小增量排序”（Diminishing Increment Sort），是直接插入排序算法的一种更高效的改进版本。希尔排序是**非稳定排序**算法。

它通过比较相距一定间隔的元素来进行，各趟比较所用的距离随着算法的进行而减小，直到只比较相邻元素的最后一趟排序为止。

基本思想：希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至 1 时，整个文件恰被分成一组，算法便终止。

17. 选择排序

（1）基本思想

找到当前数字序列中最大（最小）的数，记录其所在位置，将其和最前面（最后面）的数进行交换，使最大（最小）的元素上浮（下沉）到本次排序的最前面（最后面），从而完成一趟(pass)排序。下一趟排序时，已经有序的元素不再参与。这样的话，n 个元素需要进行 n-1 趟排序！！！

（2）举个例子

4 个数字 4,6,7,5 进行从大到小的排序。

[1] 第一趟：参与数字序列：4,6,7,5

- 找到该数字序列中最大的数 7，记录其所在位置，将其和第一个位置的数 4 进行比较，7 大于 4，所以 7 和 4 交换，得到新的序列(7),6,4,5
- 经过第一趟的排序，使数字序列中最大的数 7 上浮到最前面，此时 7 属于有序的元素，不需要参与到下一趟的排序。

[2] 第二趟排序：参与数字序列：6,4,5

- 找到该数字序列中最大的数 6，记录其所在位置，将其和第一个位置的数 6 进行比较，6 等于 6，不需要交换，得到新的序列(7,6),4,5
- 经过第二趟的排序，使数字序列中第二大的数 6 上浮到第二个前面，此时 7,6 属于有序的元素，不需要参与到下一趟的排序。

[3] 第三趟排序：参与数字序列 4,5

- 找到该数字序列中最大的数 5，记录其所在位置，将其和第一个位置的数 4 进行比较，5 大于 4，所以 5 和 4 交换，得到新的序列(7,6,5),4
- 经过第三趟的排序，使数字序列中第三大的数 5 上浮到第三个前面，此时 7,6,5 属于有序的元素，不需要参与到下一趟的排序。

[4] 最后就剩下一个数 4，就不需要进行排序，排序最终得到的数字序列为：7,6,5,4。

（3）选择排序的关键点

- 采用双层循环：时间复杂度是 **O(n^2)**

- 外层循环表示的是排序的趟数，n 个数字需要 n-1 趟，因此，外层循环的次数是 n-1 次；同时也代表数的位置。
- 内层循环表示的是每一趟排序的数字。根据选择排序的思想，第 i 趟排序时，最前面的位置就是 i，用一个循环来不断更新。

- 找到最值数的位置，将该位置的数和当前序列最前面（最后面）位置的数进行交换。（**稳定排序**）

18. 堆排序

（1）堆排序的基本思路

a. 将无序序列构建成一个堆，根据升序降序需求选择大顶堆或小顶堆（一般升序采用大顶堆，降序采用小顶堆）。

b. 将堆顶元素与末尾元素交换，将最大元素"沉"到数组末端。

c. 重新调整结构，使其满足堆定义，然后继续交换堆顶元素与当前末尾元素，反复执行调整+交换步骤，直到整个序列有序。

（2）堆的操作

在堆的数据结构中，堆中的最大值总是位于根节点（在优先队列中使用堆的话堆中的最小值位于根节点）。堆中定义以下几种操作：

- 最大堆调整（Max Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点
- 创建最大堆（Build Max Heap）：将堆中的所有数据重新排序
- 堆排序（HeapSort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算

（3）堆排序的优缺点

- 比快速排序的优点：在最坏情况下它的性能很优越。
- 比归并排序的优点：使用的辅助存储少。
- 缺点：不适合太小的待排序列（因为需要建堆）；不稳定，不适合对象的排序。

19. 计数排序(Counting Sort)

计数排序，不是基于元素比较，而是**利用数组下标确定元素的正确位置**。

计数排序是一个非基于比较的排序算法。它的优势在于在对一定范围内的整数排序时，它的复杂度为O(n+k)（其中 k 是整数的范围），快于任何比较排序算法。当然这是一种牺牲空间换取时间的做法，而且当O(k)>O(n*log(n))的时候其效率反而不如基于比较的排序（基于比较的排序的时间复杂度在理论上的下限是O(n*log(n)), 如归并排序，堆排序）。计数排序是一个稳定的排序算法。

20. 介绍下桶散列

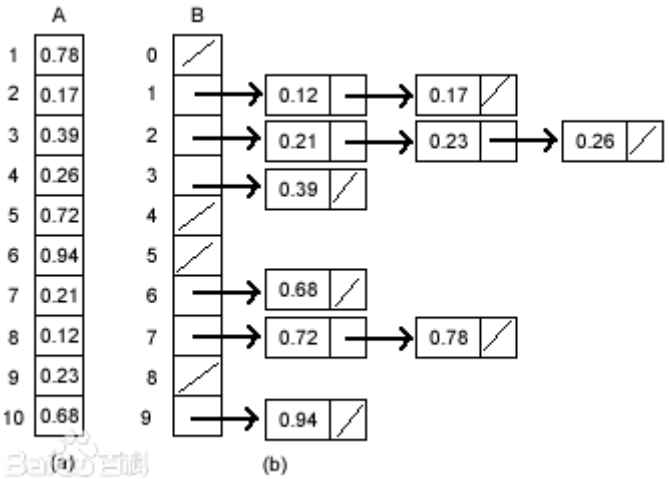
21. 桶排序(Bucket sort)

（1）基本思想：划分多个范围相同的区间，每个子区间自排序，最后合并

桶排序是将待排序集合中处于同一个值域的元素存入同一个桶中，也就是根据元素值特性将集合拆分为多个区域，则拆分后形成的多个桶，从值域上看是处于有序状态的。对每个桶中元素进行排序，则所有桶中元素构成的集合是已排序的。

（2）算法过程

[1] 根据待排序集合中最大元素和最小元素的差值范围和映射规则，确定申请的桶个数；
[2] 遍历待排序集合，将每一个元素移动到对应的桶中；
[3] 对每一个桶中元素进行排序，并移动到已排序集合中。

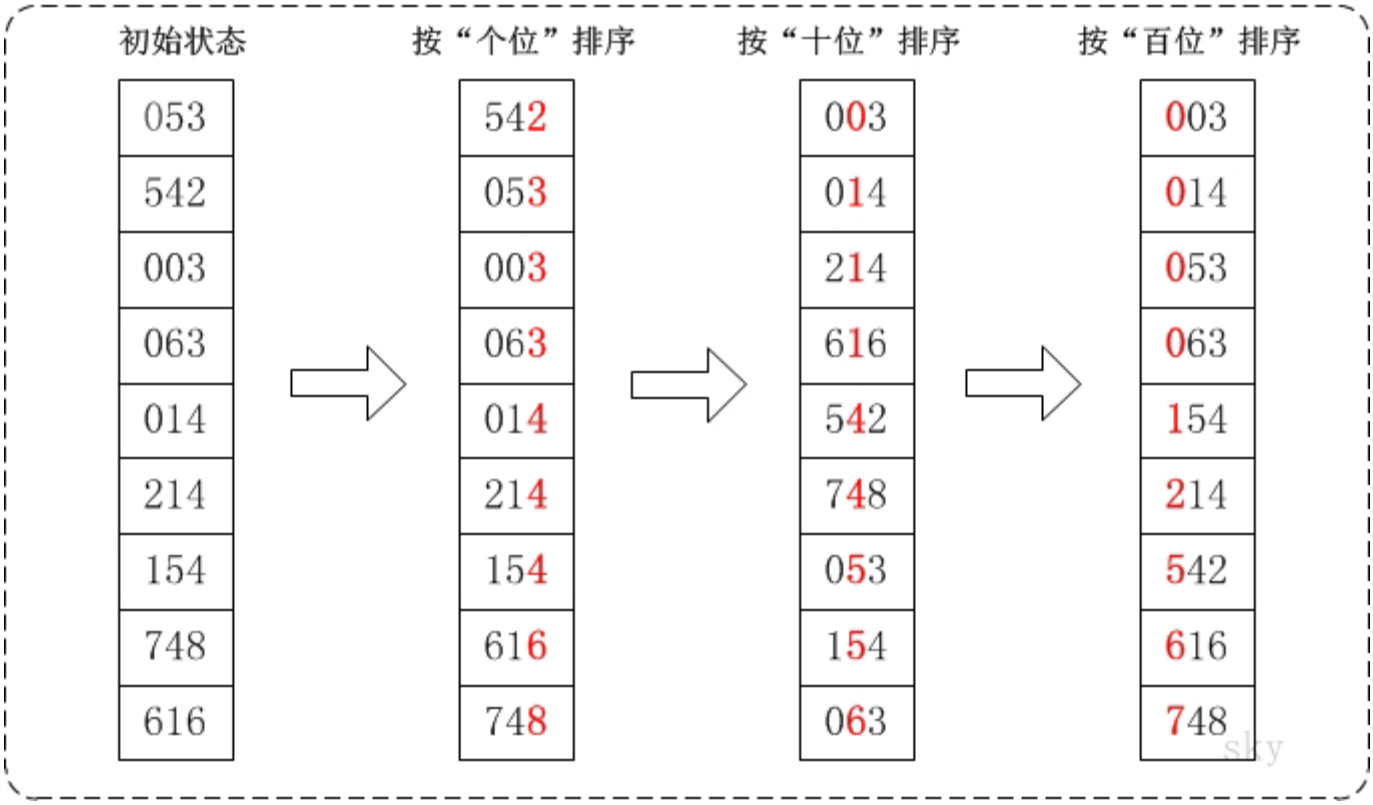


22. 基数排序(Radix Sort)

基数排序是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。

基数排序（radix sort）属于“分配式排序”（distribution sort），又称“桶子法”（bucket sort）或 bin sort，顾名思义，它是透过键值的部份资讯，将要排序的元素分配至某些“桶”中，藉以达到排序的作用，基数排序法是属于**稳定性**的排序，其时间复杂度为 O (nlog(r)m)，其中 r 为所采取的基数，而 m 为堆数，在某些时候，基数排序法的效率高于其它的稳定性排序法。

基数排序的方式可以采用 LSD（Least significant digital）或 MSD（Most significant digital），LSD 的排序方式由键值的最右边开始，而 MSD 则相反，由键值的最左边开始。



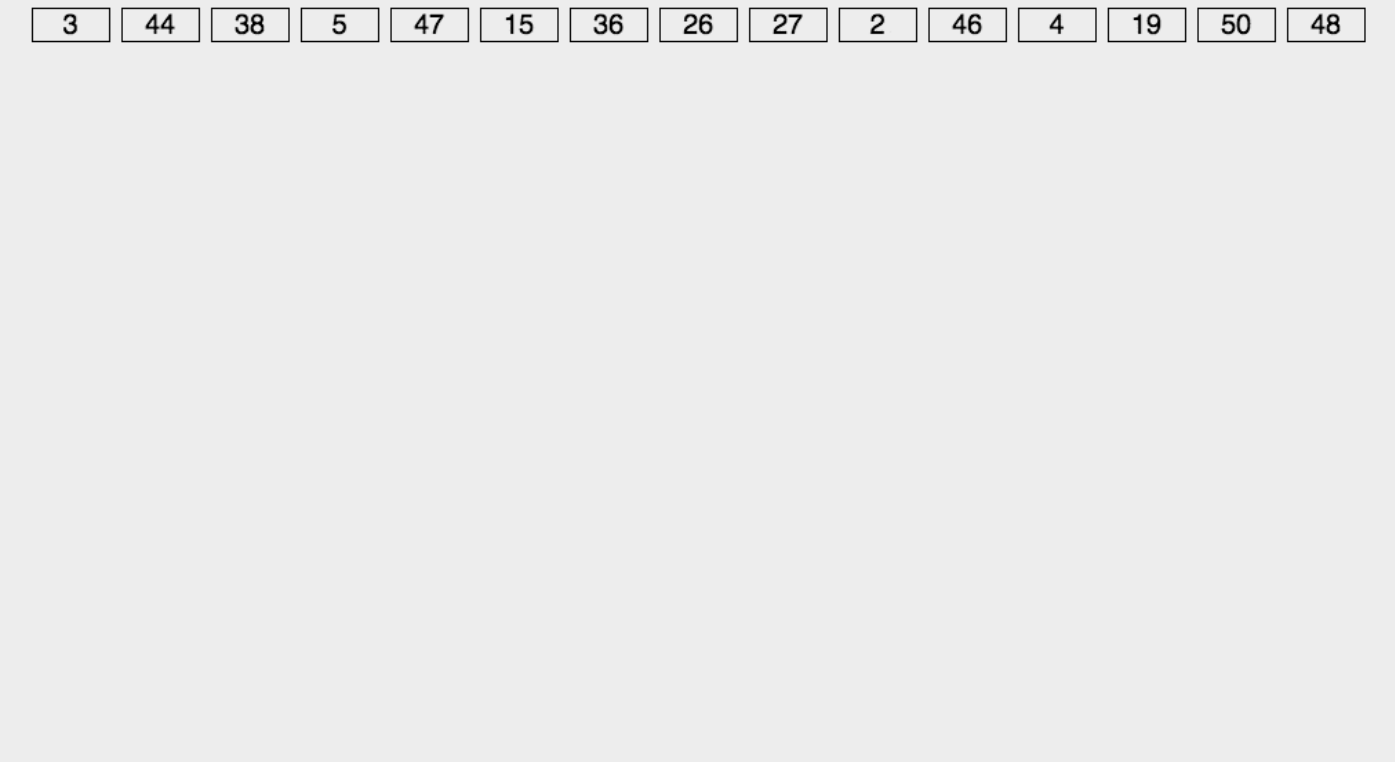


图 2 LSD 基数排序动图演示

基数排序是一种稳定的排序算法，但有一定的局限性：

[1] 关键字可分解

[2] 记录的关键字位数较少，如果密集更好

[3] 如果是数字时，最好是无符号的

23. 基数排序 vs 计数排序 vs 桶排序

这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：

- 基数排序：根据键值的每位数字来分配桶。
- 计数排序：每个桶只存储单一键值。
- 桶排序：每个桶存储一定范围的数值。

24. 各类排序算法对比

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序		O(d(r+n))	O(d(n+rd))	O(d(r+n))	O(rd+n)	稳定

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数。

（1）时间复杂度

- 平方阶($O(n^2)$)排序——各类简单排序：直接插入、直接选择和冒泡排序
- 线性对数阶(O(nlog2n))排序——快速排序、堆排序和归并排序
- $O(n^{1+\S})$)排序,§是介于 0 和 1 之间的常数——希尔排序（O(n^1.3)）
- 线性阶(O(n))排序——基数排序，此外还有桶、箱排序

说明：

- 当原表有序或基本有序时，直接插入排序和冒泡排序将大大减少比较次数和移动记录的次数，时间复杂度可降至 O(n)。
- 而快速排序则相反，当原表基本有序时，将蜕化为冒泡排序，时间复杂度提高为 $O(n^2)$。
- 原表是否有序，对简单选择排序、堆排序、归并排序和基数排序的时间复杂度影响不大。

（2）稳定性

排序算法的稳定性：若待排序的序列中，存在多个具有相同关键字的记录，经过排序，这些记录的相对次序保持不变，则称该算法是稳定的；若经排序后，记录的相对次序发生了改变，则称该算法是不稳定的。

- 稳定的排序算法：**冒泡排序、插入排序、归并排序和基数排序**
- 不是稳定的排序算法：**选择排序、快速排序、希尔排序、堆排序**

（3）选择排序算法准则

一般而言，需要考虑的因素有以下四点：

设待排序元素的个数为 n.

[1] 当 n 较大，则应采用时间复杂度为 O(nlog2n)的排序方法：快速排序、堆排序或归并排序。

[2] 当 n 较大，内存空间允许，且要求稳定性：归并排序。

[3] 当 n 较小，可采用直接插入或直接选择排序。

a) 直接插入排序：当元素分布有序，直接插入排序将大大减少比较次数和移动记录的次数。

b) 直接选择排序：元素分布有序，如果不要求稳定性，选择直接选择排序。

[4] 一般不使用或不直接使用传统的冒泡排序。

25. 冒泡排序算法的改进

1） 设置一标志性变量 pos，用于**记录每趟排序中最后一次进行交换的位置**。由于 pos 位置之后的记录均已交换到位，故在进行下一趟排序时只要扫描到 pos 位置即可。

2） 传统冒泡排序中每一趟排序操作只能找到一个最大值或最小值，我们考虑利用**在每趟排序中进行正向和反向两遍冒泡**的方法**一次可以得到两个最终值(最大值和最小值)**，从而使排序趟数几乎减少了一半。

26. 快速排序的改进

（1）只对长度大于 k 的子序列递归调用快速排序，让原序列基本有序，然后再对整个基本有序序列用插入排序算法排序

实践证明，改进后的算法时间复杂度有所降低，且当 k 取值为 8 左右时，改进算法的性能最佳。

（2）选择基准元的方式

对于分治算法，当每次划分时，算法若都能分成两个等长的子序列时，那么分治算法效率会达到最大。也就是说，基准的选择是很重要的。选择基准的方式决定了两个分割后两个子序列的长度，进而对整个算法的效率产生决定性影响。最理想的方法是，选择的基准恰好能把待排序序列分成两个等长的子序列。

[1] 方法 1 固定基准元

如果输入序列是随机的，处理时间是可以接受的。如果数组已经有序时，此时的分割就是一个非常不好的分割。

[2] 随机基准元

这是一种相对安全的策略。由于基准元的位置是随机的，那么产生的分割也不会总是会出现劣质的分割。在整个数组数字全相等时，仍然是最坏情况，时间复杂度是 O(n^2）。实际上，随机化快速排序得到理论最坏情况的可能性仅为 1/(2^n）。所以随机化快速排序可以对于绝大多数输入数据达到 O(nlogn）的期望时间复杂度。

[3] 三数取中

- 引入的原因：虽然随机选取基准时，减少出现不好分割的几率，但是还是最坏情况下还是 O(n^2)，要缓解这种情况，就引入了三数取中选取基准。
- 分析：最佳的划分是将待排序的序列分成等长的子序列，最佳的状态我们可以使用序列的中间的值，也就是第 N/2 个数。可是，这很难算出来，并且会明显减慢快速排序的速度。这样的中值的估计可以通过随机选取三个元素并用它们的中值作为基准元而得到。事实上，随机性并没有多大的帮助，因此一般的做法是**使用左端、右端和中心位置上的三个元素的中值作为基准元**。

27. 散列表（哈希表）查找

（1）详见 84-86.

哈希表（Hash table，也叫散列表），是根据关键码值(Key value)而直接进行访问的数据结构。

（2）解决哈希冲突的方法

[1] 线性探测法
[2] 平方探测法
[3] 伪随机序列法
[4] 拉链法

28. 哈希表除留取余法的桶个数为什么是质数？

除留取余，就是使用哈希函数将关键字被某个不大于哈希表长 m 的数 p 除后所得的余数作为哈希地址。如果散列值的因数越多，可能导致的散列分布越不均匀，所以在 p 的选择上需要选择约数少的数值，所以往往将桶个数设置为质数或不包含小于 20 的质因数的合数。

29. 字符串的，模式匹配算法

（1）BF 算法（属于朴素的模式匹配算法）

（2）KMP 算法

- 在一个字符串中查找是否包含目标的匹配字符串。其主要思想是每趟比较过程让子串向后滑动一个合适的位置。当发生不匹配的情况时，不是右移一位，而是移动（当前匹配的长度 - 当前匹配子串的部分匹配值）位。
- 核心：避免不必要的回溯。

30. 贪心算法 vs 动态规划 vs 分治法

分治法，动态规划法，贪心算法这三者之间有类似之处，比如都需要将问题划分为一个个子问题，然后通过解决这些子问题来解决最终问题。

（1）分治法

分治法（divide-and-conquer）：将原问题划分成 n 个规模较小而结构与原问题相似的子问题；递归地解决这些子问题，然后再合并其结果，就得到原问题的解。

分治模式在每一层递归上都有三个步骤：

[1] 分解（Divide）：将原问题分解成一系列子问题；

[2] 解决（Conquer）：递归地解各个子问题。若子问题足够小，则直接求解；

[3] 合并（Combine）：将子问题的结果合并成原问题的解。

合并排序（merge sort）是一个典型分治法的例子。其对应的直观的操作如下：

a) 分解：将 n 个元素分成各含 n/2 个元素的子序列；

b) 解决：用合并排序法对两个子序列递归地排序；

c) 合并：合并两个已排序的子序列以得到排序结果。

（2）动态规划

动态规划算法的设计可以分为如下 4 个步骤：

[1] 描述最优解的结构

[2] 递归定义最优解的值

[3] 按自底向上的方式计算最优解的值

[4] 由计算出的结果构造一个最优解

分治法是指将问题划分成一些独立的子问题，递归地求解各子问题，然后合并子问题的解而得到原问题的解。与此不同，动态规划适用于子问题独立且重叠的情况，也就是各子问题包含公共的子问题。在这种情况下，若用分治法则会做许多不必要的工作，即重复地求解公共的子问题。动态规划算法对每个子子问题只求解一次，将其结果保存在一张表中，从而避免每次遇到各个子问题时重新计算答案。

适合采用动态规划方法的最优化问题中的两个要素：最优子结构和重叠子问题。

- 最优子结构：如果问题的一个最优解中包含了子问题的最优解，则该问题具有最优子结构。

- 重叠子问题：适用于动态规划求解的最优化问题必须具有的第二个要素是子问题的空间要很小，也就是用来求解原问题的递归算法课反复地解同样的子问题，而不是总在产生新的子问题。对两个子问题来说，如果它们确实是相同的子问题，只是作为不同问题的子问题出现的话，则它们是重叠的。

分治法：各子问题独立　　**动态规划：各子问题重叠**

（3）贪心算法

贪心算法是使所做的选择看起来都是当前最佳的，期望通过所做的**局部最优选择**来产生出一个全局最优解。贪心算法对大多数优化问题来说能产生最优解，但也不一定总是这样的。

贪心算法与动态规划与很多相似之处。特别地，贪心算法适用的问题也是最优子结构。贪心算法与动态规划有一个显著的区别，就是贪心算法中，是以自顶向下的方式使用最优子结构的。贪心算法会先做选择，在当时看起来是最优的选择，然后再求解一个结果子问题，而不是先寻找子问题的最优解，然后再做选择。

贪心算法是通过做一系列的选择来给出某一问题的最优解。对算法中的每一个决策点，做一个当时看起来是最佳的选择。这一点是贪心算法不同于动态规划之处。在动态规划中，每一步都要做出选择，但是这些选择依赖于子问题的解。因此，解动态规划问题一般是自底向上，从小子问题处理至大子问题。贪心算法所做的当前选择可能要依赖于已经做出的所有选择，但不依赖于有待于做出的选择或子问题的解。**因此，贪心算法通常是自顶向下地做出贪心选择，不断地将给定的问题实例归约为更小的问题。贪心算法划分子问题的结果，通常是仅存在一个非空的子问题。**

31. 求拓扑排序的几种方法

32. 如何判断一个图是否有环？

33. 给你 20G 的数据，在 3G 内存里进行排序，怎么操作？

34. 堆和栈

1) **程序内存的区域**

a) 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，内存的分配是连续的，类似于平时我们所说的栈，如果还不清楚，那么就把它想成数组，它的内存分配是连续分配的，即，所分配的内存是在一块连续的内存区域内．当我们声明变量时，那么编译器会自动接着当前栈区的结尾来分配内存。

b) 堆区（heap）

一般由程序员分配释放， 若程序员不释放，程序结束时可能由操作系统回收。类似于链表，在内存中的分布不是连续的，它们是不同区域的内存块通过指针链接起来的。一旦某一节点从链中断开，我们要人为的把所断开的节点从内存中释放。

c) 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，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。 程序结束后由系统释放。

d) 文字常量区

常量字符串就是放在这里的。 程序结束后由系统释放。

e) 程序代码区

存放函数体的二进制代码。

2) 堆和栈的区别

1、概念
- 栈 stack：存放函数的参数值、局部变量，由编译器自动分配释放。
- 堆 heap：是由 new 分配的内存块，由应用程序控制，需要程序员手动利用 delete 释放，如果没有，程序结束后，操作系统自动回收。

2、因为堆的分配需要使用频繁的 new/delete，造成内存空间的不连续，会有大量的碎片。

3、堆的生长空间向上，地址越大，栈的生长空间向下，地址越小。

a) 申请方式不同

- 栈：由系统自动分配。例如：在函数中定义一个局部变量 int a = 0; 系统会在栈上自动开辟相应大小。

  注意：系统首先会去查看栈上是否有足够的区域去开辟该空间，如果有就直接开辟，如果没有则栈溢出。
- 堆：由程序员自己去申请开辟，并且指明大小。(利用 new/malloc)

b) 申请大小的限制

- 栈：在 Windows 下，栈是**向低地址扩展**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**。

c) 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/malloc 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。（new/malloc 后一定要显示的调用 free/delete 去释放内存）

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。

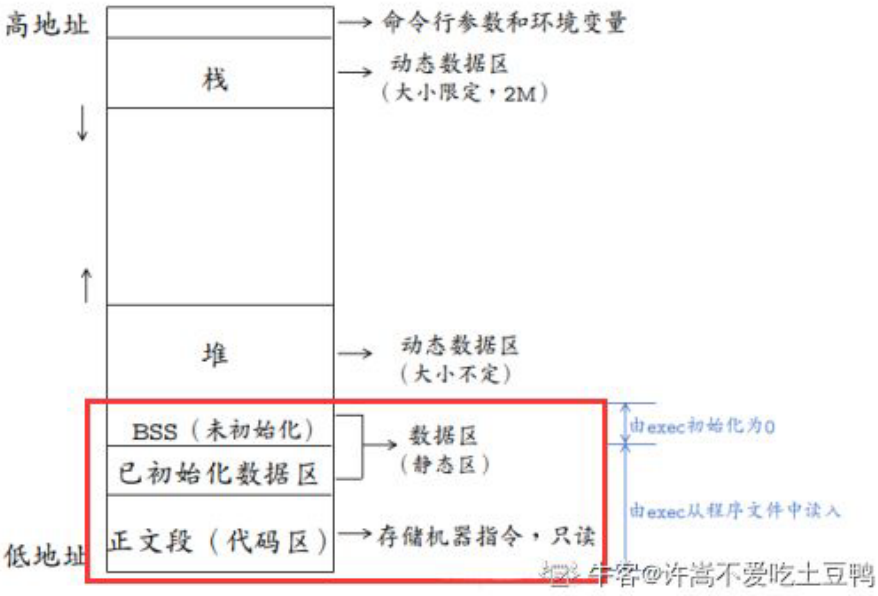
d) 堆和栈中的存储内容

- 栈：在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的 C 编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意**静态变量是不入栈的**。

  当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。
- 堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

e) 底层不同

- 栈：是**连续的空间。**
- 堆：**不是连续的空间。**



请注意：在栈上所申请的内存空间，当我们出了变量所在的作用域后，系统会自动我们回收这些空间，而在堆上申请的空间，当出了相应的作用域以后，我们需要显式 的调用 delete 来释放所申请的内存空间，如果我们不及时得对这些空间进行释放，那么内存中的内存碎片就越来越多，从而我们的实际内存空间也就会变的越来越少，即，孤立的内存块越来越多。在这里，我们知道，堆中的内存区域不是连续的，还是将有效的内存区域经过链表指针连接起来的，如果我们申请到了某一块内存，那么这一块内存区将会从连续的（通过链表连接起来的）内存块上断开，如果我们在使用完后，不及时的对它进行释放，那么它就会被孤立开来，由于没有任何指针指向它，所以这个区域将成为内存碎片，所以在使用完动态分配的内存（通过 new 申请）后，一定要显式的对它进行 delete 删除．对于这一点，一定要切记...

35. 如何用栈实现汉诺塔问题？

汉诺塔实现的基本思路是：不断将 n 个盘的汉诺塔问题转换为 2 个(n-1)个盘的汉诺塔问题，用递归实现比较好理解。设 n 盘问题为(n, a, b, c)，其中参数如下结构体所定义，第一个参数表示需要移动的盘子的数量，第二个参数表示 n 个盘子起始所在柱子 a, 第三个参数表示会被借用的柱子 b, 第四个参数表示这 n 个盘子所在的目标柱子 c。

1) 递归思路

假设(n, a, b, c)表示把 a 柱子上的 n 个盘借助 b 柱子移动到 c 柱子上，这个问题的递归求解方式是：

[1] 先把 a 柱子的(n-1)盘子借助 c 柱子移动到 b 柱子上(n-1, a, c, b),

[2] 然后把 a 柱子剩下的一个盘子移动到 c 柱子上(1, a, b, c)，

[3] 最后把 b 柱子上的(n-1)个盘子移动到 c 柱子上(n-1, b, a, c).

则问题求解可转换为对(n - 1, a, c, b)、(1, a, b, c)、(n - 1, b, a, c）这三个问题的求解，其中(1, a, b, c）不需要递归，可直接实现，将 n 个盘的汉诺塔问题转换为 2 个(n-1)个盘的汉诺塔问题，然后使用递归将(n-1)盘问题转换成(n-2)盘问题，直到盘数为 1。

2) 非递归的方式

使用 3 个栈模拟 3 个塔，每一步的移动，都按照真实情况进行。

递归方式本质上使用栈来实现的，所以如果采用非递归的方式也是使用栈来辅助实现。

但是若是用堆栈来实现的话，当将分解出的上述三个问题压入栈时，应该按照“需要先求解的问题后压入”的顺序，也就是压入顺序为：(n - 1, b, a, c), (1, a, b, c), (n - 1, a, c, b).

```
typedef struct {      //汉诺塔问题的结构类型
    int N;
    char A;           //起始柱
    char B;           //借助柱
    char C;           //目标柱
}ElementType;     //汉诺塔问题的结构类型
```

```
//借助栈的非递归实现
void Hanoi(int n)
{
    ElementType P, toPush;
    Stack S;

    P.N = n; P.A = 'a'; P.B = 'b'; P.C = 'c';
    S.top = -1;

    Push(&S, P);
    while (S.top != -1)           //当堆栈不为空时
    {
        P = Pop(&S);
        if (P.N == 1)
            printf("%c    -> %c\n", P.A, P.C);
        else
        {
            toPush.N = P.N - 1;
            toPush.A = P.B; toPush.B = P.A; toPush.C = P.C;
            Push(&S, toPush);          //将第二个待解子问题(n - 1, b, a, c)入栈
            toPush.N = 1;
            toPush.A = P.A; toPush.B = P.B; toPush.C = P.C;
            Push(&S, toPush);          //将可直接求解的子问题(1, a, b, c)入栈
            toPush.N = P.N - 1;
            toPush.A = P.A; toPush.B = P.C; toPush.C = P.B;
            Push(&S, toPush);          //将第一个待解子问题(n - 1, a, c, b)入栈
        }
    }
}
```

```
//借助栈的非递归实现
void Hanoi(int n)
{
    ElementType P, toPush;
    Stack S;

    P.N = n; P.A = 'a'; P.B = 'b'; P.C = 'c';
    S.top = -1;

    Push(&S, P);
    while (S.top != -1)           //当堆栈不为空时
```

```
    {
        P = Pop(&S);
        if (P.N == 1)
            printf("%c    -> %c\n", P.A, P.C);
        else
        {
            toPush.N = P.N - 1;
            toPush.A = P.B; toPush.B = P.A; toPush.C = P.C;
            Push(&S, toPush);            //将第二个待解子问题(n - 1, b, a, c)入栈
            toPush.N = 1;
            toPush.A = P.A; toPush.B = P.B; toPush.C = P.C;
            Push(&S, toPush);            //将可直接求解的子问题(1, a, b, c)入栈
            toPush.N = P.N - 1;
            toPush.A = P.A; toPush.B = P.C; toPush.C = P.B;
            Push(&S, toPush);            //将第一个待解子问题(n - 1, a, c, b)入栈
        }
    }
}
```

36. 编译与解释

翻译高级语言编写的程序的方式有**编译**和**解释**两种。

编译(compile)是用**编译器(complier)**程序把高级语言所编写的源程序(source code)翻译成用机器指令表达的目标代码，使目标代码和源程序在功能上完全等价，通过**连接器(linker)**程序将目标程序与相关连接库连接成一个完整的可执行程序。其优点是执行速度快，产生的可执行程序可以脱离编译器和源程序存在，反复执行。

解释(interpret)是用**解释器(interpreter)**程序将高级语言编写的源程序逐句进行分析翻译，解释一句，执行一句。当源程序解释完成时目标程序也执行结束，下次运行程序时还需要重新解释执行。其优点是移植到不同平台时不用修改程序代码，只要有合适的解释器即可。

37. new、delete、malloc、free 的关系

- 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。
- delete 会调用对象的析构函数，和 new 对应，new 调用构造函数，free 只会释放内存。

- 对于非内部数据类型的对象而言，光用 maloc/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，以及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。

1) C++中 new 和 malloc 的区别

[1] 属性

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

[2] 参数

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

[3] 返回类型

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

[4] 自定义类型

new 会先调用 operator new 函数，申请足够的内存（通常底层使用 malloc 实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete 先调用析构函数，然后调用 operator delete 函数释放内存（通常底层使用 free 实现）。

malloc/free 是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

[5] **“重载”**

C++允许自定义 operator new 和 operator delete 函数控制动态内存的分配。

内存区域

new 做两件事：分配内存和调用类的构造函数，delete 是：调用类的析构函数和释放内存。而 malloc 和 free 只是分配和释放内存。

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

[6] **分配失败**

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

[7] **内存泄漏**

内存泄漏对于 new 和 malloc 都能检测出来，而 new 可以指明是哪个文件的哪一行，malloc 确不可以。

38. C++有哪些性质（面向对象特点）

封装，继承和多态。

39. 多态性 (polymer phism)

- 多态是指相同的操作或函数、过程可作用于多种类型的对象上并获得不同的结果。不同的对象，收到同一消息可以产生不同的结果，这种现象称为多态。
- **多态是指同样的消息被不同类型的对象接收时导致不同的行为。**所谓消息是指对类成员函数的调用，不同的行为是指不同的实现，也就是调用了不同的函数。（C++课本）
- 多态性是指允许同一个函数（或操作符）有不同的版本，对于不同的对象执行不同的版本。C++支持以下两种多态性：（数据结构课本）

  [1] 编译时的多态性，表现为函数名（或操作符）的重载；

  [2] 运行时的多态性，通过派生类和虚函数来实现。

40. 虚函数(virtual function)与纯虚函数(pure virtual function)

（1）虚函数

一个虚函数是一个在基类中被声明为"virtual"，并在一个或多个派生类中被重定义的函数。

虚函数只能是类中的一个成员函数，且不能是静态的。

利用虚函数，可在基类和派生类中使用相同的函数名定义函数的不同实现，从而实现“一个接口、多种方式”。当用基类指针或引用对虚函数进行访问时，系统将根据运行时指针或引用的实际对象来自动确定调用对象所在类的虚函数版本。

使用虚函数，系统要增加一定的空间开销用来存储虚函数表，但系统在进行动态联编时的时间开销是很少的，因此，多态性是高效的。

（2）纯虚函数

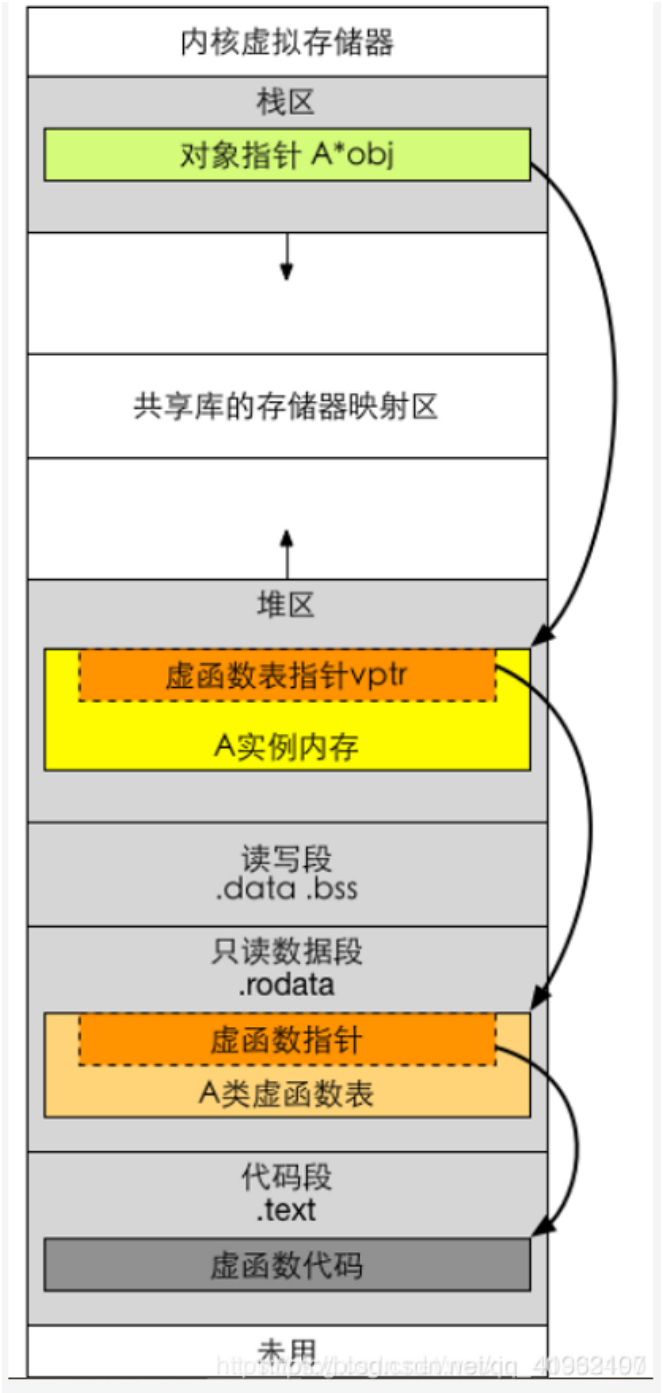
在许多情况下，不能在基类中为虚函数给出一个有意义的定义，这时可以将它说明为纯虚函数，将具体定义留给派生类去做。纯虚函数的定义形式为：**virtual 返回类型 函数名(形式参数列表)＝0;** 即在虚函数的声明原型后加上“=0”，表示纯虚函数根本就没有函数体。

纯虚函数的作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行定义。如果在一个类中声明了纯虚函数，而在其派生类中没有对该函数定义，则该虚函数在派生类中仍然为纯虚函数。

（3）虚函数表

虚函数（*Virtual Function*）是通过一张虚函数表（*Virtual Table*）来实现的，简称为 *V-Table*。在这个表中，主要是一个类的虚函数的地址表，这张表解决了继承、覆盖的问题，保证其真实反应实际的函数。这样，在有虚函数的类的实例中这个表被分配在了这个实例的内存中，所以，当我们用父类的指针来操作一个子类的时候，这张虚函数表就显得由为重要了，它就像一个地图一样，指明了实际所应该调用的函数。

*C++*的编译器应该是保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证取到虚函数表有最高的性能——如果有多层继承或是多重继承的情况下）。 这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。



（4）在内存中的存储

C++中，

- 虚函数表位于只读数据段（.rodata），即：C++内存模型中的常量区。
- 虚函数代码则位于代码段（.text），也就是 C++内存模型中的代码区。

（5）哪些函数不能声明成虚函数

在 C++，有五种函数不能被声明成虚函数，分别是：**非成员函数、构造函数、静态成员函数、内联成员函数、友元函数**这五种，下面分别解释为什么这五种函数不能被声明成虚函数。

- **非成员函数：**非成员函数只能被重载(overload)，不能被继承(override)，而虚函数主要的作用是在继承中实现动态多态，非成员函数早在编译期间就已经绑定函数了，无法实现动态多态，那声明成虚函数还有什么意义呢？
- **构造函数：**要想调用虚函数必须要通过“虚函数表”来进行的，但虚函数表是要在对象实例化之后才能够进行调用。而在构造函数运行期间，还没有为虚函数表分配空间，自然就没法调用虚函数了。
- **静态成员函数：**静态成员函数对于每个类来说只有一份，所有的对象都共享这一份代码，它是属于类的而不是属于对象。虚函数必须根据对象类型才能知道调用哪一个虚函数，故虚函数是一定要在对象的基础上才可以的，两者一个是与实例相关，一个是与类相关。
- **内联成员函数：**内联函数是为了在代码中直接展开，减少函数调用花费的代价，虚函数是为了在继承后对象能够准确的执行自己的动作，并且 inline 函数在编译时被展开，虚函数在运行时才能动态地绑定函数。
- **友元函数：**因为 C++不支持友元函数的继承，对于没有继承特性的函数没有虚函数的说法。友元函数不属于类的成员函数，不能被继承。

41. 关于链表的一些问题（涉及到了一点顺序表）

42. 数组和链表的区别

（1）从逻辑结构上来看，数组必须实现定于固定的长度，不能适应数据动态增减的情况，即数组的大小一旦定义就不能改变。当数据增加时，可能超过原先定义的元素的个数；当数据减少时，造成内存浪费；链表动态进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。

（2）从内存存储的角度看，数组从栈中分配空间（用 new 则在堆上创建），对程序员方便快速，但是自由度小；链表从堆中分配空间，自由度大但是申请管理比较麻烦。

（3）从访问方式类看，数组在内存中是连续的存储，因此可以利用下标索引进行访问；链表是链式存储结构，在访问元素时候只能够通过线性方式由前到后顺序地访问，所以访问效率比数组要低。

43. 如何判断一个链表是否有环，如何找到这个环的起点？

给定一个单链表，只给出头指针 h：

1、如何判断是否存在环？

2、如何知道环的长度？

3、如何找出环的连接点在哪里？

4、带环链表的长度是多少？

解法：

1、对于问题 1，使用追赶的方法，设定两个指针 slow、fast，从头指针开始，每次分别前进 1 步、2 步。如存在环，则两者相遇；如不存在环，fast 遇到 NULL 退出。

2、对于问题 2，记录下问题 1 的碰撞点 p，slow、fast 从该点开始，再次碰撞所走过的操作数就是环的长度 s。

3、问题 3：有定理：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。

4、问题 3 中已经求出连接点距离头指针的长度，加上问题 2 中求出的环的长度，二者之和就是带环单链表的长度。

44. 既然已经有数组了，为什么还要链表？

链表是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存储到下一个节点的指针(Pointer)。

从本质上来讲，链表与数组的确有相似之处，他们的相同点是都是线性数据结构，这与树和图不同，而它们的不同之处在于数组是一块连续的内存，而链表可以不是连续内存，链表的节点与节点之间通过指针来联系。

45. 单向链表和双向链表的优缺点及使用场景

（1）单向链表：只有一个指向下一个节点的指针。

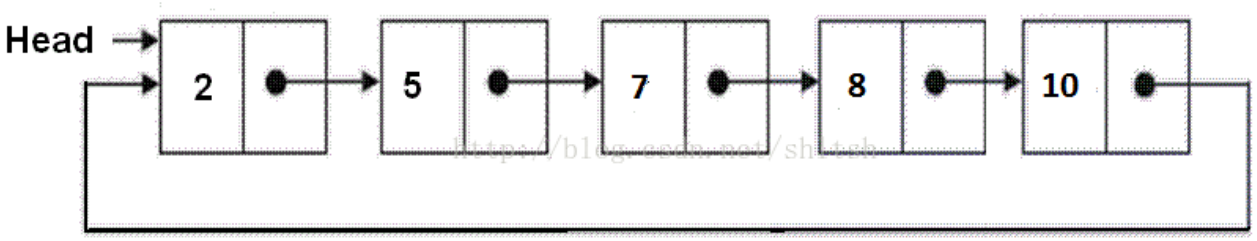
- 优点：单向链表增加删除节点简单。遍历时候不会死循环；
- 缺点：只能从头到尾遍历。只能找到后继，无法找到前驱，也就是只能前进。
- 适用于节点的增加删除。

（2）双向链表：有两个指针，一个指向前一个节点，一个后一个节点。

- 优点：可以找到前驱和后继，可进可退；
- 缺点：增加删除节点复杂，需要多分配一个指针存储空间。
- 适用于需要双向查找节点值的情况。

46. 循环链表

循环链表是一个所有节点相互连接，形成一个环的数据结构。链表尾部没有 null 节点。循环链表可以是一个单向链表，也可以是双向链表。



循环链表的好处

1) 任何节点都可以做为头节点。可以从任何节点开始进行链表的遍历。只要当第一个节点被重复访问时，则意味着遍历结束。
2) 用于实现队列数据结构是很有帮组的。如果使用循环链表，则不需要为了队列而维护两个指针(front 以及 rear)。只需要维护尾节点一个指针即可，因为尾节点的后向节点就是 front 了。
3) 循环链表常用于各应用程序中。例如，当运行多个应用程序时，操作系统通常会把这些程序存入至一个链表，并进行循环遍历，给每个应用程序分配一定的时间来执行。此时循环链表对于 OS 是很有帮助的，当达到链表尾部时，可以方便的从头部重新开始遍历。
4) 循环双向链表可以用于实现高级数据结构，例如斐波那契堆(Fibonacci Heap)。

47. 指针和引用

（1）指针

a) 概念

一个对象的地址称为该对象的指针。

通过对象地址访问对象的方式称为指针间接访问。

用来存放对象地址（即指针）的变量称为指针变量。

b) 指针的有效性

程序中的一个指针必然是以下 3 种状态之一：①指向一个已知对象；②0 值；③未初始化的、或未赋值的、或指向未知对象。

- 如果指针的值为 0 ，称为 0 值指针，又称空指针(null pointer)，空指针是无效的。
- 如果指针未经初始化，或者没有赋值，或者指针运算后指向未知对象，那么该指针是无效的。
  - 一个指针还没有初始化，称为“野指针”(wild pointer)。严格地说，每个指针在没有初始化之前都是“野指针”，大多数的编译器都对此产生警告。
  - 一个指针曾经指向一个已知对象，在对象的内存空间释放后，虽然该指针仍是原来的内存地址，但指针所指已是未知对象，称为“迷途指针”(dangling pointer)。

在实际编程中，程序员要始终确保引用的指针是有效的，对尚未初始化或为赋值的指针一般先将其初始化为 0 值，引用指针之前检测它是否为 0 值。

（2）如何理解引用(reference)?

简单地说，引用就是一个对象的别名(alias name)，其声明形式为：**引用类型 &引用名称 = 对象名称, …**

引用的本质是位于某个内存地址上的一个指定类型的对象。

在 C++中，引用全部是 const 类型，声明之后不可更改。引用一经定义，就不能指向别的地址，也不能指向别的类型，编译器不会专门开辟内存单元存储引用，而是将有引用的地方替换为对象的地址，接受引用的地方替换为指针。

（3）引用与指针的区别

（1）引用必须被初始化，指针不必。

（2）引用初始化以后不能被改变，指针可以改变所指的对象。

（3）不存在指向空值的引用，但是存在指向空值的指针。

48. 数组名，数组首地址，数组指针的区别

例如：int array[5] = {0};

众所周知，其中的&array 是整个数组 array 的首地址，array 是数组首元素的首地址（和&array[0]一样），其值相同，但是“意义不同”。

静态数组中，数组名在进行地址操作时，&arr 和 arr 值虽相同，但意义不同：&arr 移动的单位是整个数组，而 arr 移动的单位是数组元素!!

数组名字、数组名字取地址、数组首元素取地址、指向首元素的指针这四个变量的数值大小是相等的，但是在后面的地址加 1 的操作中，数组名字取地址所得地址在+1 之后所得的结果与其他变量不同。

1) 数组指针

数组指针，指的是数组名的指针，即数组首元素地址的指针。即是指向数组的指针。例：int (*p)[10]；p 即为指向数组的指针，又称数组指针。

2) 指针与数组名的区别

- 指针：也是一个变量，存储的数据是地址。
- 数组名：代表的是该数组最开始的一个元素的地址。

```
int a[10];
int *p;
p = &a[0] // 可以写成 p = a;
```

- 对数组元素 a[i]的引用也可以写成*(a+i)这种形式。
- 赋值语句 p=&a[0] 也可以写成下列形式: p=a。
- p 是个指针，p[i]与*(p+i)是等价的。

区别：指针是一个变量，可以进行数值运算。数组名不是变量，不可以进行数值运算。

3) 二维数组中的指针

[1] a+n 表示第 n 行的首地址，在一维数组中，a+n 表示的是数组的第 n+1 个元素的地址；

[2] &a[0][0]既可以看作数组 0 行 0 列的首地址，同样还可以看作二维数组的首地址。&a[m][n]就是第 m 行第 n 列的元素的地址；

[3] &a[0]是第 0 行的首地址，当然 &a[n]就是第 n 行的首地址；

[4] a[0]+n 表示第 0 行第 n 个元素的地址；

[5] ((a+n)+m) 表示第 n 行第 m 列元素；

[6] *(a[n]+m) 表示第 n 行第 m 列元素；

4) C++中的 int*、int**、int&、int*&、int *a[]、int(*a)[]：

```
int a;      //a 是一个 int 型【变量】
int *a;     //a 是一个指向 int 型变量的【指针】
int **a;    //a 是一个指向 int 型变量指针的指针，也就是【二级指针】
```

```
int &a;      //a 是一个【普通变量型引用】，若 int &a = i;则 a 是变量 i 的一个别名，&a=&i，即 a 和 i 的地址一样
int *&a;     //a 是一个【指针变量型引用】，若 int *&a = i;则 a 是指针 i 的一个引用
int a[2];    //a 是一个含有两个 int 型变量的【数组】
int *a[2];   //a 是一个【指针数组】，数组 a 里存放的是两个 int 型指针
int (*a)[2];//a 是一个【数组指针】，a 指向一个含有两个 int 型变量的数组
```

49. 函数名，函数指针，函数的入口地址的区别

- **指针函数**是指带指针的函数，即**本质是一个函数**，函数返回类型是某一类型的**指针。**
- **函数指针**是指向函数的指针变量，即**本质是一个指针变量。**

主要的区别是一个是指针变量，一个是函数。

函数指针：1. 指针变量 2. 指针变量指向函数

这正如用指针变量可指向整型变量、字符型、数组一样。

在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。

可利用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样，在这些概念上一致的。事实上，每一个函数，即使它不带有返回某种类型的指针，它本身都有一个入口地址，该地址相当于函数名。尽管函数不是变量，但它在内存中仍有其物理地址，该地址能够赋给指针变量。获取函数方法是：**用不带有括号和参数的函数名得到。**

函数名相当于一个指向其函数入口指针常量。

函数名后面加圆括号，表示函数调用。

若要得到函数的地址，直接用函数名就可以了

##

指针/函数和函数/指针的区别：

1. 指针函数：指带指针的函数，即本质是一个函数。
2. 指针函数返回类型是某一类型的指针。

##

函数指针有两个用途：调用函数和做函数的参数。函数指针的说明方法为：

数据类型标志符 （指针变量名）（形参列表）；

注 1：“函数类型”说明函数的返回类型，由于“()”的优先级高于“*”,所以指针变量名外的括号必不可少，后面的“形参列表”表示指针变量指向的函数所带的参数列表。例：

```
int func(int x); /* 声明一个函数 */
int (*f) (int x); /* 声明一个函数指针 */
f=func; /* 将 func 函数的首地址赋给指针 f */
```

赋值时函数 func 不带括号，也不带参数，func 代表函数的首地址。

注 2：函数括号中的形参可有可无，视情况而定。

下面的程序说明了函数指针调用函数的方法：

```
#include
int max(int x,int y){ return(x>y?x:y); }
void main()
{
    int (*ptr)(int, int);
    int a,b,c;
    ptr=max;
    scanf("%d,%d",&a,&b);
    c=(*ptr)(a,b);
    printf("a=%d,b=%d,max=%d",a,b,c);
}
```

实际上 ptr 和 max 都指向同一个入口地址，不同就是 ptr 是一个指针变量，不像函数名称那样是死的，它可以指向任何函数。

注意，指向函数的指针变量没有++和--运算。

50. const 与 #define 的比较，const 有什么优点?

（1） const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误（边际效应） 。

（2） 有些集成化的调试工具可以对 const 常量进行调试，但是不能对宏常量进行调试。

51. 内存的分配方式有几种?

（1）从**静态存储区域**分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如**全局变量**。

（2）在**栈上创建**。在执行函数时，**函数内局部变量**的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3）从**堆上分配**，亦称**动态内存分配**。程序在运行的时候用 **malloc 或 new** 申请任意多少的内存，程序员自己负责在何时用 **free 或 delete 释放内存**。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

52. 全局变量和局部变量有什么区别？是怎么实现的？操作系统和编译器是怎么知道的？

- 生命周期不同：全局变量随主程序创建和创建，随主程序销毁而销毁；局部变量在局部函数内部，甚至局部循环体等内部存在，退出就不存在。
- 使用方式不同：通过声明后全局变量程序的各个部分都可以用到；局部变量只能在局部使用；分配在栈区。

操作系统和编译器通过**内存分配的位置**来知道的，全局变量分配在**全局数据段**并且在程序开始运行的时候被加载。局部变量则分配在**堆栈**里面 。

53. 进程 process 与线程 thread

进程是操作系统分配资源的单位，线程是 CPU 调度的基本单位，线程之间共享进程资源。

进程与线程的一个简单解释(很形象)

深入理解进程和线程

进程与线程概念

（1）进程

计算机的核心是 CPU，它承担了所有的计算任务，而操作系统是计算机的管理者，它负责任务的调度，资源的分配和管理，统领整个计算机硬件；应用程序是具有某种功能的程序，程序是运行于操作系统之上的。

进程是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立单位，是应用程序运行的载体。进程是一种抽象的概念，从来没有统一的标准定义。**进程一般由程序，数据集合和进程控制块三部分组成**。程序用于描述进程要完成的功能，是控制进程执行的指令集；数据集合是程序在执行时所需要的数据和工作区；程序控制块包含进程的描述信息和控制信息是进程存在的唯一标志。

进程具有的特征：

- 动态性：进程是程序的一次执行过程，是临时的，有生命期的，是动态产生，动态消亡的。
- 并发性：任何进程都可以同其他进行一起并发执行。

- 独立性：进程是系统进行资源分配和调度的一个独立单位。
- 结构性：进程由程序，数据和进程控制块三部分组成。

（2）线程

在早期的操作系统中并没有线程的概念，进程是拥有资源和独立运行的最小单位，也是程序执行的最小单位。任务调度采用的是时间片轮转的抢占式调度方式，而进程是任务调度的最小单位，每个进程有各自独立的一块内存，使得各个进程之间内存地址相互隔离。

后来，随着计算机的发展，对 CPU 的要求越来越高，进程之间的切换开销较大，已经无法满足越来越复杂的程序的要求了。于是就发明了线程，**线程是程序执行中一个单一的顺序控制流程，是程序执行流的最小单元，是处理器调度和分派的基本单位。一个进程可以有一个或多个线程，各个线程之间共享程序的内存空间(也就是所在进程的内存空间)。**一个标准的线程由线程 ID，当前指令指针 PC，寄存器和堆栈组成。而进程由内存空间(代码，数据，进程空间，打开的文件)和一个或多个线程组成。

（3）进程与线程的区别

a) 线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位.

b) 一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线。

c) 进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段，数据集，堆等)及一些进程级的资源(如打开文件和信号等)，某进程内的线程在其他进程不可见。

d) 调度和切换：线程上下文切换比进程上下文切换要快得多。

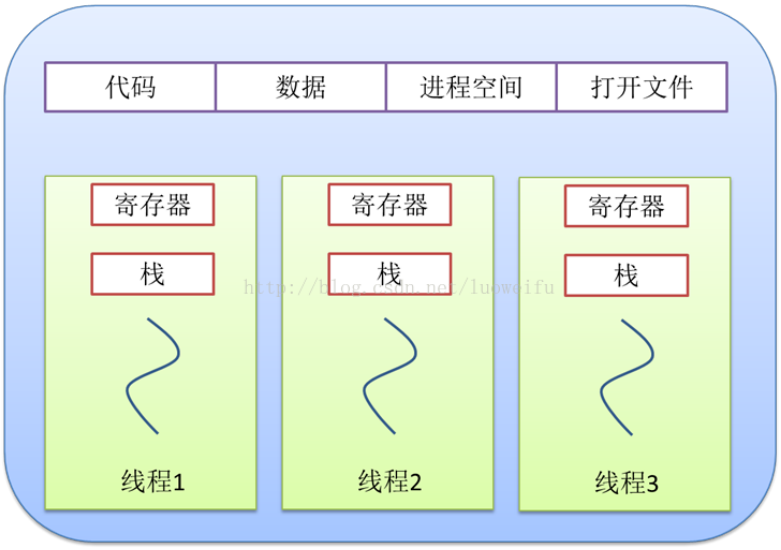
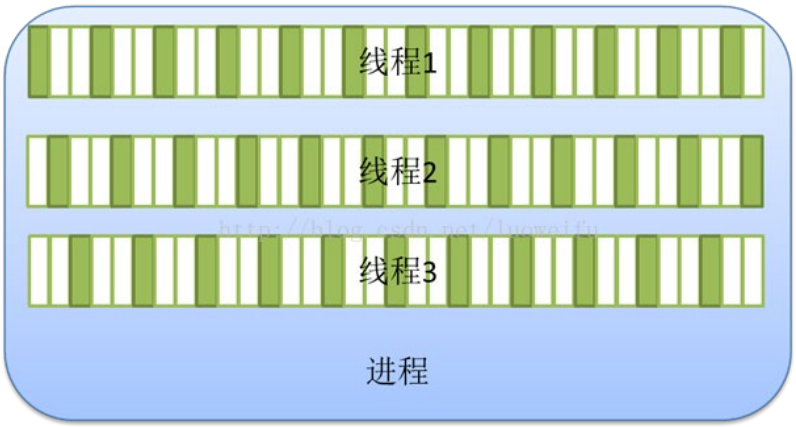


图 3 线程和进程关系示意图

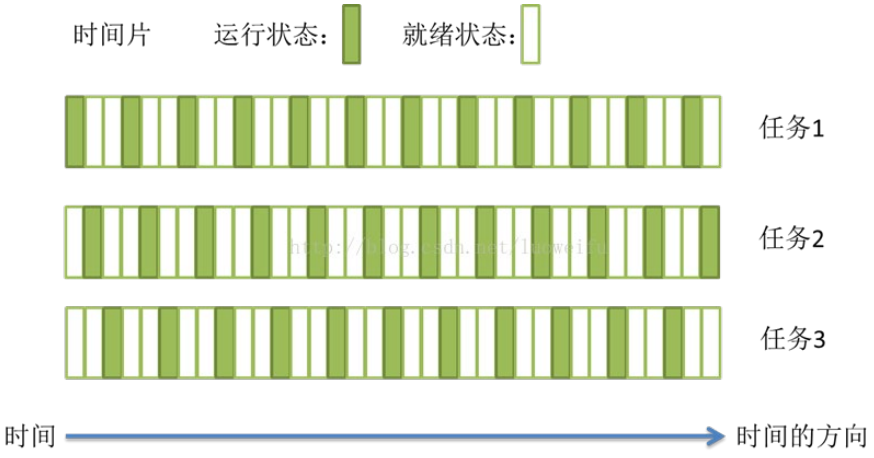
总之，线程和进程都是一种抽象的概念，线程是一种比进程还小的抽象，线程和进程都可用于实现并发。在早期的操作系统中并没有线程的概念，进程是能拥有资源和独立运行的最小单位，也是程序执行的最小单位，它相当于一个进程里只有一个线程，进程本身就是线程。所以线程有时被称为轻量级进程。

后来，随着计算机的发展，对多个任务之间上下文切换的效率要求越来越高，就抽象出一个更小的概念-线程，一般一个进程会有多个(也可以是一个)线程。



（4）任务调度

大部分操作系统的任务调度是采用时间片轮转的抢占式调度方式，也就是说一个任务执行一小段时间后强制暂停去执行下一个任务，每个任务轮流执行。任务执行的一小段时间叫做时间片，任务正在执行时的状态叫运行状态，任务执行一段时间后强制暂停去执行下一个任务，被暂停的任务就处于就绪状态，等待下一个属于它的时间片的到来。这样每个任务都能得到执行，由于 CPU 的执行效率非常高，时间片非常短，在各个任务之间快速地切换，给人的感觉就是多个任务在“同时进行”，这也就是我们所说的并发。



（5）为何不使用多进程而是使用多线程？

线程廉价，线程启动比较快，退出比较快，对系统资源的冲击也比较小。而且线程彼此分享了大部分核心对象(File Handle)的拥有权。

如果使用多重进程，但是不可预期，且测试困难。

（6）一个简单的比喻

做个简单的比喻：进程=火车，线程=车厢

[1] 线程在进程下行进（单纯的车厢无法运行）
[2] 一个进程可以包含多个线程（一辆火车可以有多个车厢）
[3] 不同进程间数据很难共享（一辆火车上的乘客很难换到另外一辆火车，比如站点换乘）
[4] 同一进程下不同线程间数据很易共享（A 车厢换到 B 车厢很容易）
[5] 进程要比线程消耗更多的计算机资源（采用多列火车相比多个车厢更耗资源）
[6] 进程间不会相互影响，一个线程挂掉将导致整个进程挂掉（一列火车不会影响到另外一列火车，但是如果一列火车上中间的一节车厢着火了，将影响到所有车厢）
[7] 进程可以拓展到多机，进程最多适合多核（不同火车可以开在多个轨道上，同一火车的车厢不能在行进的不同的轨道上）
[8] 进程使用的内存地址可以上锁，即一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。（比如火车上的洗手间）－"互斥锁"
[9] 进程使用的内存地址可以限定使用量（比如火车上的餐厅，最多只允许多少人进入，如果满了需要在门口等，等有人出来了才能进去）－“信号量”

54. 介绍下几种常见的进程调度算法及其流程（FCFS，SJF，剩余短作业优先，优先级调度，轮转法，多级反馈队列等等）

先来先服务　　最高响应比优先
短作业优先　　时间片轮转法
优先级调度　　多级反馈队列调度

55. 虚拟内存

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

56. 内存管理的方式及优点？

Windows 内存管理方式主要分为：**页式管理、段式管理、段页式管理**。

（1）页式管理

- 基本原理：将各进程的虚拟空间划分为若干个长度相等的页。把内存空间按页的大小划分为片或者页面，然后把页式虚拟地址与内存地址建立一一对应的页表，并用相应的硬件地址转换机构来解决离散地址变换问题。页式管理采用请求调页和预调页技术来实现内外存存储器的统一管理。
- 优点：没有外碎片，每个内碎片不超过页的大小。
- 缺点：程序全部装入内存，要求有相应的硬件支持，如地址变换机构缺页中断的产生和选择淘汰页面等都要求有相应的硬件支持。增加了机器成本和系统开销。

（2）段式管理

- 基本思想：把程序按内容或过程函数关系分成段，每段有自己的名字。一个用户作业或者进程所包含的段对应一个二维线性虚拟空间，也就是一个二维虚拟存储器。段式管理程序以段为单位分配内存，然后通过地址映射机构把段式虚拟地址转换为实际内存物理地址。
- 优点：可以分别编写和编译，可以针对不同类型的段采取不同的保护，可以按段为单位来进行共享，包括通过动态链接进行代码共享。
- 缺点：会产生碎片。

（3）段页式管理

- 基本思想：系统必须为每个作业或者进程建立一张段表以管理内存分配与释放、缺段处理等。另外，由于一个段又被划分为若干个页，每个段必须建立一张页表以把段中的虚页变换为内存中的实际页面。显然与页式管理时相同，页表也要有相应的实现缺页中断处理和页面保护等功能的表项。
- 优点：段页式管理是段式管理和页式管理相结合而成，具有两者的优点。
- 缺点：由于管理软件的增加，复杂性和开销也增加。另外需要的硬件以及占用的内存也有所增加，使得执行速度下降。

57. 页表、反向页表

58. 计算机是如何启动的？

1) BIOS 上电自检 **2) 查找启动设备**

3) MBR→引导程序　　5) OS 取得系统控制权

4) 加载 OS 内核

按下电源开关后，主板各个器件会供电，CPU 供电完成后会进行一段复位的操作。复位完成后会从主板上的存储芯片（BIOS 芯片）里读取一段启动代码进行上电自检。如果硬件设备都检测正常的话，它就会进行下一步：检测启动设备。

假设它查找的启动设备是一块硬盘。接下来，它会到硬盘的第一个扇区去读取一段程序（主引导记录 Master Boot Record，512 字节——引导程序+硬盘分区表、分区表数据+结束标志 AA55）。

当 BIOS 程序找到 MBR 这段数据的时候，它会把数据加载到内存中，然后把系统的控制权交给 MBR 中的那段引导程序。

MBR 的引导程序取得系统的控制权之后，它会把操作系统的内核加载到内存，然后把系统的控制权交给操作系统的内核。这个时候，操作系统就取得了系统的控制权。接下来的过程就是操作系统正常启动的过程。

59. 数据库系统中事务的概念，事务的特性

（1）事务的概念

- 事务是完成用户某一特定任务的与数据库的一次或多次交互的逻辑单位。
- 对数据库来讲，事务是和数据库交互的一个程序片段。

  对于一般意义上的用户来讲，事务是完成一个功能的程序片段或计算机的指令集合。
- 事务（Transaction） 是对数据库进行访问或修改的一个或多个操作，这组操作组成一个单位，共同完成一个任务。（参考资料）

（2）事务的 ACID 特性分别是什么？

- 原子性(Atomicity)：执行事务中的操作要么都做，要么都不做。
- 一致性(Consistency)：一致性要求事务维护数据库的完整性约束。
- 隔离性(Isolation)：并发执行的事务之间不能相互影响。
- 持续性(Durability)：事务一旦提交，它一定是永久生效的。

（3）事务的 ACID 特性怎么保证？（REDO/UNDO 机制）

60. 事务隔离等级

- READ UNCOMMITTED

  未提交读取（脏读）——有可能读到别的事务尚未提交的数据

- READ COMMIT

  提交读取——只能读到别的事务已经提交的数据

- REPEATABLE READ

  可重复读——在同一个事务中，对同一个数据的多次读，值是一样的（取决于第一次读取到的数据），即使在读的过程中别的事务是数据变化了，对你没有影响。

- SERIALIZABLE

  串行化——对同一个数据是串行化执行的

61. 黑盒测试和白盒测试

软件测试要经过的步骤：单元测试→集成测试→系统测试→验收测试

（1）黑盒测试

黑盒测试指测试人员通过各种输入和观察软件的各种输出结果来发现软件的缺陷，而不关心程序具体如何实现的一种测试方法。

- 黑盒测试用例设计方法：**等价类划分　　边界值划分　　错误推测法　　因果图法　　正交表试验法　场景图　　功能图**

（2）白盒测试

白盒测试又叫做结构测试，把程序看成装在一个透明的白盒子里，按照程序内部的逻辑测试程序，检测程序中的主要执行通路是否都能按预定要求正确工作。

- 白盒测试常用测试用例设计方法：**逻辑覆盖法（逻辑驱动测试）　　基本路径测试方法**

白盒测试法的覆盖标准有逻辑覆盖、循环覆盖和基本路径测试。其中逻辑覆盖包括语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖和路径覆盖。六种覆盖标准发现错误的能力呈由弱到强的变化。

1.语句覆盖每条语句至少执行一次。

2.判定覆盖每个判定的每个分支至少执行一次。

3.条件覆盖每个判定的每个条件应取到各个可能的值。

4.判定/条件覆盖的同时满足判定覆盖条件覆盖

5.条件组合覆盖每个判定中各条件的每一种组合至少出现一次。

6.路径覆盖使程序中每一条可能的路径至少执行一次。

62. 关于 5G

第五代移动通信技术（英语：5th Generation Mobile Communication Technology 简称 5G）是具有高速率、低时延和大连接特点的新一代宽带移动通信技术，是实现人机物互联的网络基础设施。

5G 作为一种新型移动通信网络，不仅要解决人与人通信，为用户提供增强现实、虚拟现实、超高清(3D)视频等更加身临其境的极致业务体验，更要解决人与物、物与物通信问题，满足移动医疗、车联网、智能家居、工业控制、环境监测等物联网应用需求。最终，5G 将渗透到经济社会的各行业各领域，成为支撑经济社会数字化、网络化、智能化转型的关键新型基础设施。

63. 云计算、云存储、物联网的概念

（1）云计算

云计算（cloud computing）是分布式计算的一种，指的是通过网络“云”将巨大的数据计算处理程序分解成无数个小程序，然后，通过多部服务器组成的系统进行处理和分析这些小程序得到结果并返回给用户。云计算早期，简单地说，就是简单的分布式计算，解决任务分发，并进行计算结果的合并。因而，云计算又称为网格计算。通过这项技术，可以在很短的时间内（几秒钟）完成对数以万计的数据的处理，从而达到强大的网络服务。[1]

现阶段所说的云服务已经不单单是一种分布式计算，而是分布式计算、效用计算、负载均衡、并行计算、网络存储、热备份冗杂和虚拟化等计算机技术混合演进并跃升的结果。[1]

（2）云存储

云存储是一种网上在线存储（英语：Cloud storage）的模式，即把数据存放在通常由第三方托管的多台虚拟服务器，而非专属的服务器上。托管（hosting）公司运营大型的数据中心，需要数据存储托管的人，则透过向其购买或租赁存储空间的方式，来满足数据存储的需求。数据中心营运商根据客户的需求，在后端准备存储

虚拟化的资源，并将其以存储资源池（storage pool）的方式提供，客户便可自行使用此存储资源池来存放文件或对象。实际上，这些资源可能被分布在众多的服务器主机上。

（3）物联网

物联网即物物相连的互联网。物联网是由 Kevin Ashton 教授首次提出，它的基础与核心依旧是互联网，是在互联网的基础上延伸及拓展的网络。

物联网的用户端延伸和拓展到任何的物品与物品之间，进行通信以及交换信息，即物物相息。物联网广泛应用在网络融合中，它是通过识别技术、智能感知以及普适计算等通信感知的技术来应用的。物联网是继计算机、互联网之后世界信息产业发展的第三次浪潮。

64. 输入网址点击转到之后发生的事

（1）用户在浏览器（客户端）里输入或者点击一个连接；

（2）浏览器向服务器发送 HTTP 请求；

（3）服务器处理请求，如果查询字符串或者请求体里含有参数，服务器也会把这些参数信息考虑进去；

（4）服务器更新、获取或者转换数据库里的数据；

（5）浏览器接受 HTTP 响应；

（6）浏览器以 HTML 或者其他格式（比如 JPEG、XML 或者 JSON）把 HTTP 响应呈现给用户。

或：

- **第一步：对网址进行 DNS 解析**

DNS 解析的过程就是寻找在哪台主机上有你需要的资源的过程，我们通常使用机器的域名来访问这台机器，而不是直接使用其 IP 地址，而将机器的域名转换为 IP 地址就需要域名查询服务，这个过程称为 DNS 解析，它主要充当一个翻译的角色，实现网址到 IP 地址的转换。

- **第二步：进行 TCP 连接**

TCP 连接也就是我们常说的三次握手，首先客户端向服务器端发送是否可以连接的请求，服务器端接受到请求后确认客户的 SYN，并向客户端发送自己的 SYN 包，客户端接收到服务器发来的包之后向服务器发送确认包从而完成三次握手。

- **第三步：发送 HTTP 请求**

在完成 TCP 连接后，接下来做的事情就是客户端向服务器端发送 http 请求。

- **第四步：服务器处理请求并返回 HTTP 报文**

服务器端接到 http 请求后在会作出响应。

- **第五步：浏览器解析渲染页面**

浏览器在收到 HTML,CSS,JS 文件后，它将这些信息渲染到客户端页面上。

浏览器是一个边解析边渲染的过程。首先浏览器解析 HTML 文件构建 DOM 树，然后解析 CSS 文件构建渲染树，等到渲染树构建完成后，浏览器开始布局渲染树并将其绘制到屏幕上。这个过程比较复杂，涉及到两个概念: reflow(回流)和 repain(重绘)。DOM 节点中的各个元素都是以盒模型的形式存在，这些都需要浏览器去计算其位置和大小等，这个过程称为 relow;当盒模型的位置,大小以及其他属性，如颜色,字体,等确定下来之后，浏览器便开始绘制内容，这个过程称为 repain。页面在首次加载时必然会经历 reflow 和 repain。reflow 和 repain 过程是非常消耗性能的，尤其是在移动设备上，它会破坏用户体验，有时会造成页面卡顿。所以我们应该尽可能少的减少 reflow 和 repain。

- **第六步：连接结束，关闭连接请求**

65. TCP/IP，TCP 和 UDP 的区别

- TCP/IP 协议(Transmission Control Protocol/Internet Protocol)叫做传输控制/网际协议，又叫网络通讯协议，这个协议是 Internet 国际互联网络的基础。
- TCP/IP 是网络中使用的基本的通信协议。虽然从名字上看 TCP/IP 包括两个协议，传输控制协议(TCP)和网际协议(IP)，但 TCP/IP 实际上是一组协议，它包括上百个各种功能的协议，如：远程登录、文件传输和电子邮件等，而 TCP 协议和 IP 协议是保证数据完整传输的两个基本的重要协议。通常说 TCP/IP 是 Internet 协议族，而不单单是 TCP 和 IP。
- TCP/IP 是用于计算机通信的一组协议，我们通常称它为 TCP/IP 协议族。它是 70 年代中期美国国防部为其 ARPANET 广域网开发的网络体系结构和协议标准，以它为基础组建的 INTERNET 是目前国际上规模最大的计算机网络，正因为 INTERNET 的广泛使用，使得 TCP/IP 成了事实上的标准。
- 之所以说 TCP/IP 是一个协议族，是因为 TCP/IP 协议包括 TCP、IP、UDP、ICMP、RIP、TELNET、FTP、SMTP、ARP、TFTP 等许多协议，这些协议一起称为 TCP/IP 协议。以下我们对协议族中一些常用协议英文名：
- TCP/IP 是供已连接因特网的计算机进行通信的通信协议。
- TCP/IP 指传输控制协议/网际协议 (Transmission Control Protocol / Internet Protocol)。
- TCP/IP 定义了电子设备（比如计算机）如何连入因特网，以及数据如何在它们之间传输的标准。

66. aloha，CSMA/CA，以太网和 Internet 区别，有了以太网为什么还有 Internet？

以太网只是组成互联网的一个子集，以太网是现在主流的局域网标准，而互联网是指将大量的局域网连接起来，进行资源的分享。

或：

以太网（英语：Ethernet）是为了实现局域网通信而设计的一种技术，它规定了包括物理层的连线、电子信号和介质访问层协议的内容。以太网是目前应用最普遍的局域网技术，取代了其他局域网标准如令牌环、FDDI 和 ARCNET。

互联网（英语：Internet）是一个网络的网络，它是由从地方到全球范围内几百万个私人的，政府的，学术界的，企业的和政府的网络所构成，通过电子，无线和光纤网络技术等等一系列广泛的技术联系在一起。简单地说，以太网是一直为了实现局域网通信而设计的一系列方法，包括物理层传输媒介和 CSMA/CD 协议等内容，而互联网是计算机网络。

67. 流量控制在哪一层起作用？

数据链路层、传输层

68. 解释下什么是 DMA，介绍下 DMA 流程

通过在 I/O 设备和内存之间开启一个可以直接传输数据的通路，采用 DMA 控制器来控制一个数据块的传输，CPU 只需在一个数据块传输开始阶段设置好传输所需的控制信息，并在传输结束阶段做进一步处理。

69. 程序的编译执行过程

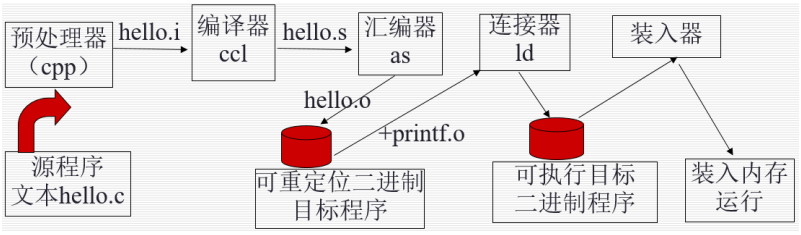
构建 C 程序需要 4 个步骤，分别使用 4 个工具完成： preprocessor, compiler, assembler, and linker.四步完成后生成一个可执行文件。

第一步，预处理. 这一步处理 头文件、条件编译指令和宏定义。

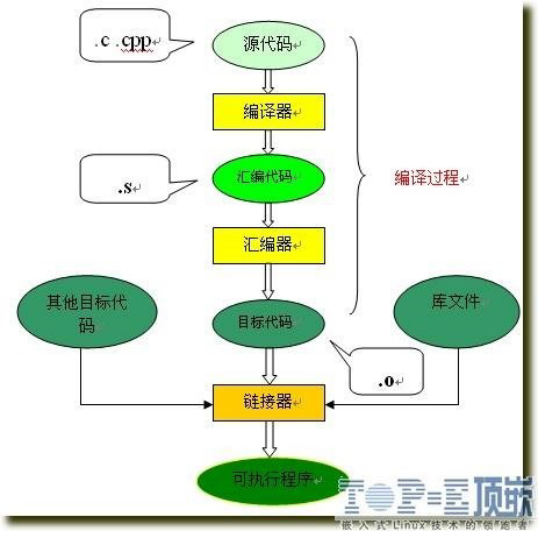
第二步，编译. 将第一步产生的文件连同其他源文件一起编译成汇编代码。

第三步，汇编。将第二步产生的汇编源码转换为 object file.

第四步，链接. 将第三步产生的一些 object file 链接成一个可执行的文件。



或：



C 语言的编译链接过程要把我们编写的一个 C 程序（源代码）转换成可以在硬件上运行的程序（可执行代码），需要进行编译和链接。编译就是把文本形式源代码翻译为机器语言形式的目标文件的过程。链接是把目标文件、操作系统的启动代码和用到的库文件进行组织，形成最终生成可执行代码的过程。

从图上可以看到，整个代码的编译过程分为编译和链接两个过程，编译对应图中的大括号括起的部分，其余则为链接过程。

70. 说说你对编译原理的理解

“编译原理课程”讲述高级程序设计语言源程序转换成汇编语言或机器语言的程序时使用的技术、数据结构和算法(即编译程序原理)。

编译器就是将“一种语言(通常为高级语言)”翻译为“另一种语言(通常为低级语言)”的程序。一个现代编译器的主要工作流程:源代码 (source code) → 预处理器 (preprocessor) → 编译器 (compiler) → 目标代码 (object code) → 链接器(Linker) → 可执行程序 (executables)。

71. Top k 问题

1) 排序（选取前 k 个数）

2) 快排

快排的 partition 划分思想可以用于计算某个位置的数值等问题，例如用来计算中位数；显然，也适用于计算 TopK 问题

每次经过划分，如果中间值等于 K ，那么其左边的数就是 Top K 的数据；当然，如果不等于，只要递归处理左边或者右边的数即可。

该方法的时间复杂度是 O(n) ，简单分析就是第一次划分时遍历数组需要花费 n，而往后每一次都折半（当然不是准确地折半），粗略地计算就是 n + n/2 + n/4 +... < 2n，因此显然时间复杂度是 O(n)。

缺点：在海量数据的情况下，我们很有可能没办法一次性将数据全部加载入内存，这个时候这个方法就无法完成使命了

3) 利用分布式思想处理海量数据

面对海量数据，我们就可以往分布式的方向去思考了。

我们可以将数据分散在多台机器中，然后每台机器并行计算各自的 TopK 数据，最后汇总，再计算得到最终的 TopK 数据。

4) 利用最经典的方法（堆），一台机器也能处理海量数据

其实提到 Top K 问题，最经典的解法还是利用堆。

维护一个大小为 K 的小顶堆，依次将数据放入堆中，当堆的大小满了的时候，只需要将堆顶元素与下一个数比较：如果大于堆顶元素，则将当前的堆顶元素抛弃，并将该元素插入堆中。遍历完全部数据，Top K 的元素也自然都在堆里面了。

当然，如果是求前 K 个最小的数，只需要改为大顶堆即可

对于海量数据，我们不需要一次性将全部数据取出来，可以一次只取一部分，因为我们只需要将数据一个个拿来与堆顶比较。

另外还有一个优势就是对于动态数组，我们可以一直都维护一个 K 大小的小顶堆，当有数据被添加到集合中时，我们就直接拿它与堆顶的元素对比。这样，无论任何时候需要查询当前的前 K 大数据，我们都可以里立刻返回给他。

整个操作中，遍历数组需要 O(n) 的时间复杂度，一次堆化操作需要 O(logK)，加起来就是 O(nlogK) 的复杂度，换个角度来看，如果 K 远小于 n 的话， O(nlogK) 其实就接近于 O(n) 了，甚至会更快，因此也是十分高效的。

最后，对于 Java，我们可以直接使用优先队列 PriorityQueue 来实现一个小顶堆。

72. 如何写代码计算根号 n

1) 袖珍计算器算法

「袖珍计算器算法」是一种用指数函数 exp 和对数函数 ln 代替平方根函数的方法。我们通过有限的可以使用的数学函数，得到我们想要计算的结果。

我们将 $\sqrt{x}$ 写成幂的形式 $x^{1/2}$，再使用自然对数 e 进行换底，即可得到

$$\sqrt{x} = x^{1/2} = (e^{\ln x})^{1/2} = e^{\frac{1}{2}\ln x}$$

这样我们就可以得到 $\sqrt{x}$ 的值了。

2) 二分法

由于 x 平方根的整数部分 ans 是**满足** $k^2 \le x$ **的最大** k **值**，因此我们可以对 k 进行二分查找，从而得到答案。

二分查找的下界为 0，上界可以粗略地设定为 x。在二分查找的每一步中，我们只需要比较中间元素 mid 的平方与 x 的大小关系，并通过比较的结果调整上下界的范围。由于我们所有的运算都是整数运算，不会存在误差，因此在得到最终的答案 ans 后，也就不需要再去尝试 $ans+1$ 了。

3) 牛顿迭代法

- 已知 $f(x) = x^2 - t$，$f'(x) = 2x$，$f(x_0) = 0$
- 求 x_0
- $\because x_{n+1} = x_n - f(x_n)/f'(x_n)$
- $\therefore x_{n+1} = x_n - (x_n^2 - t)/(2x_n)$

73. 如何一次写出正确的程序？\ 如何知道你写的程序是对的？

74. 微信红包随机金额怎么实现？

1) 关于分配算法，红包里的金额怎么算？为什么出现各个红包金额相差很大？

答：随机，额度在 0.01 和剩余平均值 2 之间。

例如：发 100 块钱，总共 10 个红包，那么平均值是 10 块钱一个，那么发出来的红包的额度在 0.01 元～20 元之间波动。当前面 3 个红包总共被领了 40 块钱时，剩下 60 块钱，总共 7 个红包，那么这 7 个红包的额度在：0.01～（60/7 * 2）=17.14 之间。

注意：这里的算法是每被抢一个后，剩下的会再次执行上面的这样的算法（Tim 老师也觉得上述算法太复杂，不知基于什么样的考虑）。这样算下去，会超过最开始的全部金额，因此到了最后面如果不够这么算，那么会采取如下算法：保证剩余用户能拿到最低 1 分钱即可。如果前面的人手气不好，那么后面的余额越多，红包额度也就越多，因此实际概率一样的。

2) **微信的金额什么时候算？**

答：微信金额是拆的时候实时算出来，不是预先分配的，采用的是纯内存计算，不需要预算空间存储。

为什么采取实时计算金额？原因是：实时效率更高，预算才效率低下。预算还要占额外存储。因为红包只占一条记录而且有效期就几天，所以不需要多大空间。就算压力大时，水平扩展机器是。

75. 概率计算题：一副扑克牌平均分成三堆，大小王同时在一堆的概率

假设有 1 2 3 三组，我们先求大王小王在同在 1 组的概率：

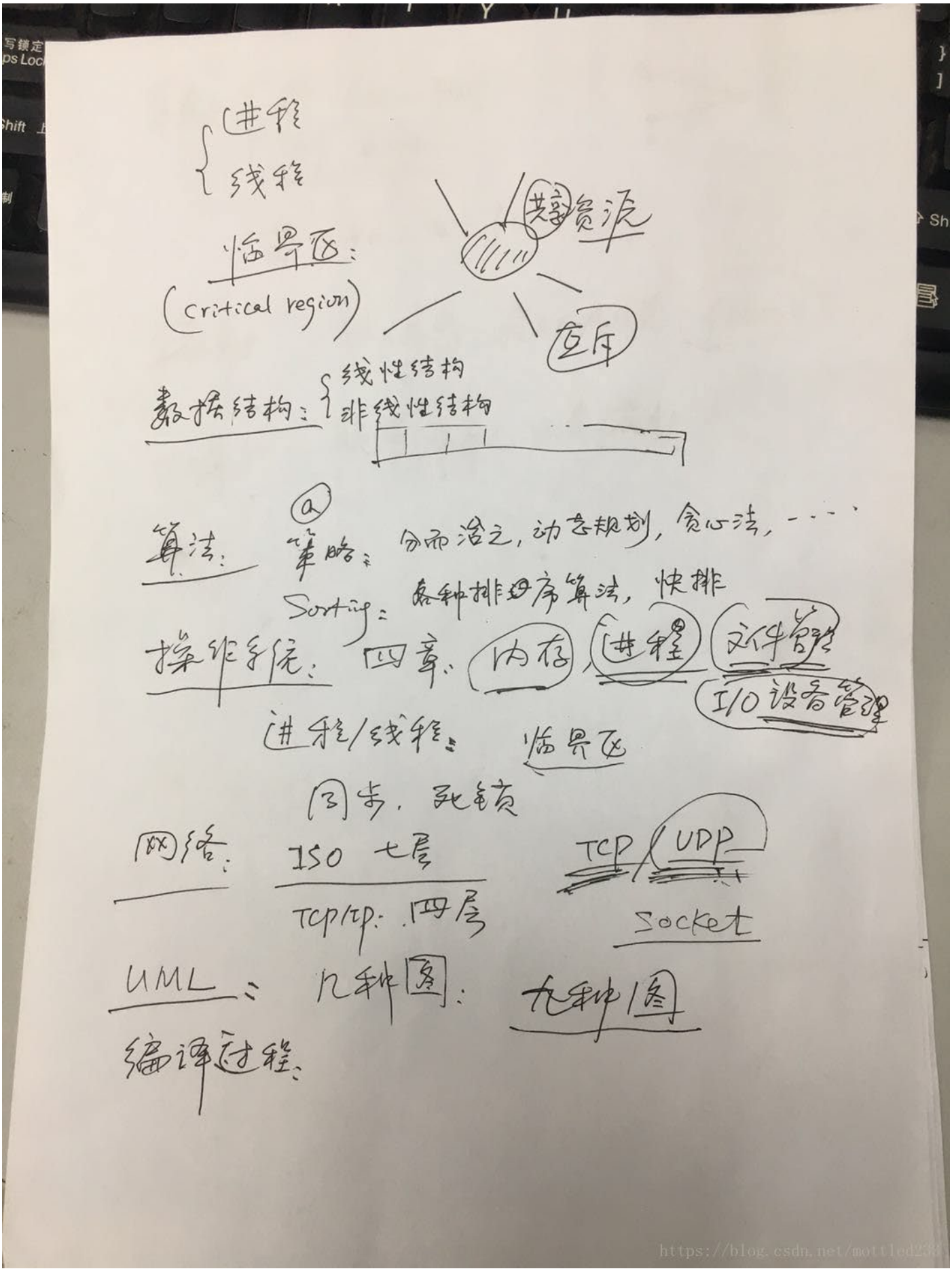
- 大王在 1 组 P(B)：18/54
- 大王已经在 1 组的条件下小王在 1 组 P(A|B)：17/53（因为把大王放在 1 组用掉了一个空位）
- 那么大王小王同在 1 组的概率 P：P(A,B) = P(B)*P(A|B) = 18/54 * 17/53

1，2，3 组都有可能选择，3 * 18/54 * 17/53 = 17/53

76. 命题逻辑的联结词有哪些？

非、析取、合取、蕴涵、等价

北航计算机学院往年夏令营+考研面试题目汇总（1）（2）



1. 哪些图算法中用到了动态规划的思想。
2. 编译原理中的 0、1、2、3 型文法及其关系。
3. 中间代码优化的目的。
4. 进程之间的通信。
5. TCP 协议和 UDP 协议不同的应用场景。
6. 原码、反码、补码。
7. 超标量处理机。
8. 五级流水线。
9. dfs、bfs 描述。√
10. 如何编程判断一棵二叉树是完全二叉树。
11. 如何高效地判断链表是否有环。（leetcode 快慢指针）
12. 最小生成树：prim 和 kruscal。
13. 最短路 dijkstra。
14. 拓扑排序。
15. 欧拉回路。
16. KMP 算法。
17. 排序：快排、归并。
18. 工厂方法模式的思想。
19. 数据库的各种范式。

https://blog.csdn.net/zqm_0015/article/details/109236372?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-11.base&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-11.base

2020 我的计算机保研历程

https://blog.csdn.net/HNUCSEE_LJK/article/details/109025471?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-14.control&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-14.control

计组：

解释下什么是 DMA

说下五级流水 CPU 的各阶段

执行单条指令时单周期 CPU 和五级流水 CPU 谁更快？为什么？

操作系统：

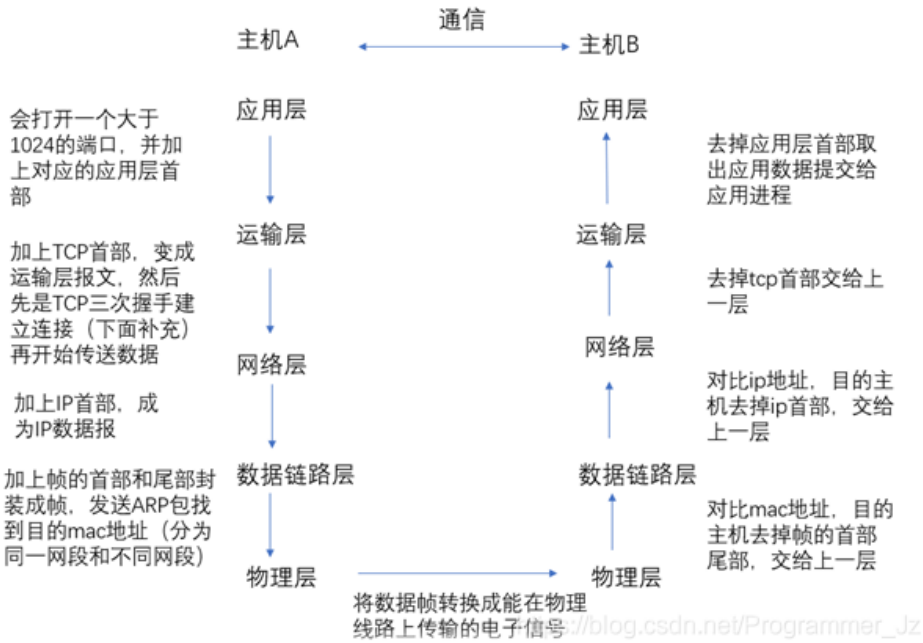
说下进程间通信的几种方式

简要介绍一下分页分段

计网：

说下 TCP 和 UDP 区别

说下网络中的主机通信流程



一个主机将两个端口接到网络上是否会提升吞吐量？为什么？

数据结构：

说下快排过程

如何判断一个单链表是否是循环链表（要给出一个比遍历更快的方法，貌似是两个指针一个每次前进 1 步、一个每次前进 2 步，相遇则循环）

介绍下平衡二叉树

线代：

介绍下什么是矩阵的秩

介绍下特征值与特征向量的意义

介绍下线性相关和线性无关

离散：解释下等价关系和等价类

概率论：

解释下大数定律

说一下全概率公式和贝叶斯公式

介绍下正态分布

机器学习：

机器学习和深度学习的差别联系

梯度下降法和牛顿迭代法的算法过程

编译原理：因为我没学过所以没问

计算机研究生复试面试题目

Q：堆排序与快排的区别

A： 相同点：平均时间均为 o(nlogn)。

不同点：堆排最坏情况为 o(nlong),快排最坏为 o(n^2)。空间堆排 o(1),快排 o(logn)。

Q：c 语言栈溢出的一个例子/c 语言没有可靠性检查（栈溢出）

A：在一个过程 A 中分配了一个数组，数组的数据存储在栈上，将这个数组名传入到另一个过程 B 中，这时数组退化为指针，并在过程 B 中写该指针，若超出这个指针范围，那么就会使得返回地址被覆盖。插入金丝雀值检测栈是否被修改。

Q：c 语言程序优化

A：利用栈变量加快执行速度。循环展开利用流水线能力，多个累积变量，

Q：存储器山（访问步长，数据集大小）

A：

Q：const 和 volatile

A：const 全局无法修改，被 const 修饰的局部变量仍然是变量，存储于栈上，并不是真正的常量，并没有存储在只读区。const 只是告诉编译器这个常量后面不能跟一个=号。真正存在常量区的是 char*s= “ asdasd”,所以 const 的值是可以被改变的，只是需要通过其他方式。

Q:一个参数既可以是 const 还可以是 volatile 吗？解释为什么

A：是的。一个例子是只读的状态寄存器。它是 volatile 因为它可能被意想不到地改变。它是 const 因为程序不应该试图去修改它。

Q：解释 volatile

A：被 volatile 修饰的变量，要求每次在读这个变量，需要小心地从内存中读取，而不是从寄存器去取。这使得在多线程环境下，任意一个线程修改了这个值，别的线程就能立即发现，并使用这个新值。

Q：alloc 函数与 malloc 函数，

A：alloc 在栈上分配，malloc 在堆上分配。

Q：malloc(0)，malloc(-1)

A:仍然都能能够返回一个可访问的指针，具体原因未知。

Q：malloc 函数会内存泄露吗？

A：不会的。

Q：数组名与指针的区别

A：数组名不能够自增自减（a+1 是正确的，数组名可理解为常量），指针能够自动加减并且按照其数据类型进行缩放。sizeof(数组名)返回数组空间按字节计算，对指针使用则是返回固定指针字节值。当数组名作为参数传递给函数时，退化为指针。数组只能被分配在静态区或栈上，指针可以指向任何一个地方。

Q：float 类型如何与 0 比较是否相等

A：定义任意一个精度范围，用 if(a<= && a>=)。推广对任意浮点数进行比较时应该使用 abs(a-b) <= eps;这样的方法来判断。

Q：ifndfe 的作用

A：避免头文件重复展开。

Q：“file.h”与<file.h>的区别

A：“file.h”编译器从用户目录寻找文件，<file.h>则是从标准库中寻找文件。

Q：#define 与 type def 的区别

A：如#define DPS struct S*，DPS s1,s2;其中 s2 是结构，s1 则是一个结构体指针。

Q：c 语言结构体如何比较

A：memcmp 进行内存上的比较

Q:可以用 free 去释放掉用 new 申请的内存吗？

A：这么做并不会出错，用 free 依然能够释放掉内存，但因为 free 是一个函数不是运算符，因此无法调用对象的析构函数。

Q：c 语言可变参数列表

A：在函数形参表中用三个…表示可变参数。使用 va_list 的一组宏取出参数。

Q：#define 与 const 的区别

A：对于 define 编译器只是简单的替换，而 const 具有数据类型，编译器可以据此对 const 变量进行安全检查。

Q：typecast 一个整数使其成为一个地址

A：int p; p = (int)0x1008;

Q:static 局部变量，全局变量，函数

A：static 局部变量只被初始化一次，在随后使用该函数的过程中，总是保持上次的值，作用域被限制在这个函数体内部。static 全局变量仅仅只在本文件内部有效，对其他文件不可见。全局变量与 static 全局变量的区别在于全局变量对所有其他文件是可见的。statci 函数同理。

Q：strcp 为什么要返回一个 char*

A：为了方便链式计算，例如 strlen（strcp）。strcp 是以’\0’计算结尾。

Q：函数体返回一个用 malloc 生成的指针的做法正确吗？

A：这样做法是极不正确的，因为这会导致内存泄漏，因为被调用者无法保证调用者能够及时释放掉内存。

A：野指针无法通过 NULL 来判断，因为野指针会指向一个随机值。

A：静态区会初始化值为 0，而堆栈区则不会对值进行初始化。

A：短小且经常调用的函数应该被写成宏函数，c++中则应该是 inline

A：运算符优先级（）>[]>*，因此 *(a ++)[5] 应该被翻译为 a[6][0];

A:二维数组传递参数的正确方式，f(int a[][5]),f(int (*a)[5]);

A:为什么不能把 p 直接指向二维数组，这是因为二维数组在内存中的存储实际上仍然是连续，不是以前自己动态分配那样，一个 p 指向两个*p，*p 再指向 int。

A:asser 断言宏函数两种形式

assert Expression1，assert Expression1:Expression2

Q：在什么情况下快排为 o(n^2)?

A：当代排关键字有序或基本有序时，快排退化为冒泡排序，为 o(n^2)。假设一个有序的长度为 n 的关键字序列，那么每次 partition 的子序列，将总是划分成一个长度为 1 的子序列，和另一个长度为 n-1 的子序列。

Q:无限制的使用 malloc 会导致程序崩溃么

A：it depends， 如果这个程序是运行在一个拥有虚拟内存的操作系统上，内存不会消耗完，因为它会使用硬盘顶替堆。

Q：B+与 B-树的区别

A：B+树所有数据都存在叶子结点上且叶子结点按顺序存储，而 B-树不是这样，因此在查找时，B+树上的每一次查找都是走了一条从根到叶子结点的路径。同时也因为这样的特性，使得 B+树有两种查找，一种是从根节点开始的随机查找，一种是从最小关键字开始的顺序查找。B+树比 B-树更适合文件系统和数据库索引。因为 B+树内部结点可以不包含其他信息，仅仅存储关键字值，存储空间比 B-树消耗的少。同时在范围查询时，B+树明显优于 B 树，B-树需要中序遍历，而 B+树则不用这样，因为它的叶节点是有序的。

B+树 vsB-树，聚集索引 vs 非聚集索引

结论聚集索引就是 B+树

Q：平衡二叉树(o(logn))与 B+树。

A：操作系统中磁盘管理系统的基本单位是盘块，而文件系统的基本单位是簇(Cluster)。即使我们只需要这个簇上的一个字节的内容，也要把一整个簇上的内容读完。而 B+树的每一个结点之所以会被设计为包含多个关键字，一方面是为了减少树的层数，另一方面也是为了尽可能的让一个结点充满簇，避免浪费。反观平衡二叉树，一个父结点仅连接两个左右孩子，树的深度远远大于 B+树。并且在物理存储上是极不规整的。

Q:50W 数据如何排序

A：外部排序需要多次 I/O 读写操作，I/O 操作随着归并的趟数增多而增多。归并的趟数 s 则由 m 和 k 决定。s = logkm。单纯的增加 k 会增加内部归并的时间，因为在 k 个关键字中求最小关键字需要进行 k-1 次比较，为得到 u 个有序记录需要进行(u-1)(k-1)次比较。随着 k 的增大，内部归并时间也会增大，将会抵消掉由于增大 k 而减少外存信息读写时间所得效益。因此需要制定一个数据结构败者树减少内部排序时间，败者树中选择出一个最小关键字记录所需 log2k 次比较。置换选择排序优化 m

Q：什么是稳定排序和不稳定排序？

A：设一个含 n 个记录的序列为{r1,r2,r3,…rn}，其对应的关键字序列{k1,k2,k3….kn}。假设存在 ki=kj，且在排序前 ri 领先于 rj。若在排序之后，ri 仍然领先于 rj，则称排序方法是稳定的，否则则是不稳定的。

Q：什么是二叉查找树？

A：二叉查找树是一个递归定义的树形数据结构。1.若左子树不空，则所有左子树节点的值均小于根节点。2.右子树同理。3.左右子树均为二叉排序树。在这样的定义之上，能够制定出一个高效快速的查找关键字的算法。略

Q：什么是哈夫曼树？

A：设有 n 个权值{w1,w2,w3,w4…,wn}，用这样的权值做节点值去构造一颗二叉树。在所有这样的树中，树的带权路径长度最小的二叉树，就是哈夫曼树。

Q：什么是带权路径长度？

A：从树中一个节点到另一个节点之间的分支构成这两个节点之间的路径，路径上的分支数目称为路径长度。树的路径长度是从树根到每一个节点的路径长度之和。

Q：图的深度优先遍历

A：若用邻接矩阵存储图，深度遍历复杂度为 o(n^2)。若用邻接表存储图，深度遍历负载度为 o(n+e)。

Q：解释生成树

A：联通图的生成树是一个极小联通子图，它含有图中所有顶点，但只有足以构成一棵树的 n-1 条边。若在一颗生成树上添加一条边，必定会构成一个环。但是有 n-1 条边的图不一定就是生成树。现为图中的边赋予权值，

设 n 个权值{w1,w2,w3,w4……wn}。使用这些含权值的边构造生成树，在所有生成树中，权值之和最小的生成树，就是最小生成树。

Q：解释虚拟内存

A：虚拟内存打破需要程序一次性全部装入才能运行的限制，之所以可以这么做是基于两个原理，时间局部性原理和空间局部性原理。为了实现虚拟内存，需要硬件上的支持和软件上的支持。硬件上需要缺页中断机构，地址翻译机构。软件上需要制定页面置换算法。因为虚拟内存为用户虚拟出一个用户地址空间，这简化了编译器和连接器的设计。

Q:解释分段与分页的区别

A：分页要对物理内存进行分块，每块物理内存的大小容量是固定，然后将进程的逻辑地址空间分成若干页，然后再将逻辑页映射到物理块上，正是因为这样的方法使得分页系统容易产生内部碎片。而分段系统不对物理内存进行分块，而是将进程的逻辑地址空间，按照用户需求，进行逻辑分段。然后将这些逻辑分段映射到物理内存上，这样使得分段系统容易产生外部碎片。较之于连续内存分配方案，分段与分页都采用离散分配方式，且都需要地址变换机构。其中分页以页为最小单位，页内总是连续的。分段以段为最小单位，段内总是连续的地址。分页对用户是可不见的，而分段对用户却不是透明的。分段系统较之于分页系统一个最突出的优点就是易于实现数据的共享，且对段的保护也十分简单，分段系统的诞生是为了适应软件工程的需求。

Q：连续内存分配方案与算法

A：单一连续，固定分区，动态分区（首次适应，最佳适应，最坏适应，邻近适应）

Q：简述系统调用过程

A：系统调用本质上就是一种过程调用，但它是一种特殊的过程调用。除了像普通过程调用一样，保护 cpu 现场，传递参数之外，系统调用还需要使用 trap 指令以软中断的形式进入到内核态，然后才能转入到相应的服务程序中执行。因为只有在内核态下才能够使用 cpu 指令集中的特权指令。最后在服务程序执行完毕返回调用处，并将状态模式设置为用户态，随后将控制返回给用户程序。

Q：中断服务程序的过程

A：当一个用户程序正在执行，被一个外部中断所打断时，cpu 在正式进入去执行中断服务程序的时候，硬件要执行一些操作。这包括，关闭中断响应，保存当前用户程序的程序计数器，引出中断程序。这些操作被称为中断隐指令。随后在中断服务程序中将保存用户程序的 cpu 现场。

Q：为什么需要系统调用

A：在 cpu 指令集中有一部分指令为特权指令，这些指令只能让操作系统使用，如启动 I/O 设备指令，存取一些特殊寄存器等。限制用户程序无法直接使用特权指令，而是通过让用户程序使用系统调用间接使用特权指令，就能够保证用户程序不会随意破坏某些特殊寄存器的值，从而避免了系统崩溃。

Q：交换与覆盖的区别

A：覆盖是人为的将内存划分为固定区和覆盖区。覆盖区中的数据可以被替换，固定区中的则不行。程序运行时可以将必要的代码和数据存入固定区，而将那些不怎么活跃的代码段存入覆盖区。这样就使得程序不必一次性全部装入。但同时这样也增加了编写程序的难度。究竟哪一段程序是可以被覆盖的，哪一段是固定的，这都需要编程人员指出。而交换是将处于等待状态的进程换出内存，将就绪好的进程换入内存执行。目的是为了增加系统吞吐量。

目的不同：覆盖是为了解决程序必须一次型全部装入才能运行的限制。而交换是为了提高系统的吞吐量。

Q：TLB 与 cache 与内存的区别

A：TLB 与 cache 都是按内容访问的存储器，因为采用 SRAM 构成，这使得它们的访存速度大大快于使用 DRAM 组成的内存。但正因为采用 SRAM 使得它们的功耗高于内存，集成度低于内存。因此在计算机系统中，它们通常都是位于 cpu 内部，作为 cpu 与内存之间的高速缓存，用于解决高速 cpu 与慢速内存访问速度之间的矛盾。

不同点：按内容访问存储器（SRAM 构成），按地址访问存储器（DRAM 构成）。

Q：解释哈希表是什么

A：在记录的存储位置和它的关键字之间建立一个确定的对应关系 f，这个 f 能使每一个关键字和结构中一个唯一的存储位置相对应。这样一来在查找时，只需根据这个对应关系 f 就能够快速找到关键字的存储位置，不需要进行比较便可直接取得所查记录。因此，称这个对应关系为哈希函数。按这个思想建立的表，称为哈希表。

哈希函数：直接定址法，数字分析法，平方取中法，除留余数法。

Q：解释哈希冲突

A：哈希冲突就是指多个不同的关键字记录被哈希函数映射到同一个存储位置上，这就是哈希冲突。解决哈希冲突的方法通常有 1.开放定址法 2.链地址法。

Q：图的相关概念

A： 顶点：图中的数据元素，记 V 是顶点集合。

弧：记 VR 是两个顶点之间的关系集合。若有<v,w>属于 VR，则表示从顶点 v 到顶点 w 的一条弧。若存在<v,w>属于 VR，必有<w,v>属于 VR，则称(v，w)为顶点 v 和 w 的一条边。

图：图就是顶点集合 V 和关系集合 VR 所构成的一个有序二元组(V,VR)。

顶点数目与边数目：

1.在无向图中，若给定顶点数目 n，那么边的取值范围便在[0,1/2n(n-1)]这个区间上。边数目为 1/2n(n-1)的图，就叫做完全图

2.在有向图中，若给定顶点数目 n，那么边的取值范围便在[0,n(n-1)]这个区间上。边数目为 n(n-1)的图，就叫做有向完全图。

路径：是一个顶点序列所构成的有序序列。若在序列中，第一个顶点和最后一个顶点相同，称这个路径为环。若序列中不出现重复的路径称为简单路径。

联通图：若从顶点 v1 到 v2 存在路径，那么就说 v1 和 v2 之间是联通的。若在一个图中的任意两个顶点之间都是联通的，那么这个图就是联通图。

Q:解释硬中断与软中断区别

A：中断源是引起中断的源头，它来自 cpu 内部或 cpu 外部。通常来讲，来自 cpu 内部的中断称作软中断，是程序在执行过程中所发生的中断。常见的软中断有缺页异常，除 0 异常，内存访问越界等。而来自 cpu 外部的中断称作硬中断，是外部机器硬件发出信号请求 cpu 服务的中断。常见的硬中断有设备 I/O 完成，定时器时间到等。在一般情况下，软件中断是不可屏蔽的，硬件中断却被分为可屏蔽中断源和不可屏蔽中断源。在 cpu 内部有一个中断控制逻辑对外延伸出两个针脚，一个叫 NMI，另一个叫 INTR。连接到 NMI 的中断请求是不可屏蔽的，必须立即响应。连接到 INTR 的中断请求是可屏蔽的，可以由此来制定中断请求的优先级。从 cpu 响应中断的时间不同来讲，软件中断一般发生在一条指令执行过程中发生的，在指令执行途中便响应中断。而响应硬件中断则是在一条指令执行结束后才响应，cpu 内部也有一个周期称作中断周期。

Q：说明一下操作系统中的数据结构。

A：操作系统中为了描述程序的运行情况，每一个程序都需要一个 PCB 来记录它的运行情况。将这些 PCB 组织起来就形成 PCB 表。PCB 表的实现方式有多种，线性方式，连接方式，索引方式。在文件系统中，为了能对一个文件进行正确的按名存取，操作系统需要一个 FCB 来描述文件名的控制文件和文件的硬盘地址。将这些 FCB 组织起来，就形成了目录文件。常见的目录实现方式有单级文件目录，两级文件目录，树形目录。在管理组织磁盘时，通常有链接组织方式，连续组织方式和索引组织方式。在其中链接组织方式中，最著名的数

据结构就是 FAT 表。追溯 FAT 表的思想起源，可以发现是来自数据结构中的静态链表。静态链表的思想在树的双亲法表示中也有体现。在处理机调度时，常用到的数据结构是队列，常见的是多级反馈队列。

Q：解释文件控制块(FCB)

A：为了能对一个文件进行正确的按名存取，操作系统需要用文件控制块（FCB）来记载文件名，文件物理位置，文件物理结构等信息。将许多文件控制块组织起来，就形成了目录，称为目录文件。在早期使用文件名检索目录时，因单个文件控制块存储数据过大，使得不能一次性读入整个目录文件，这使得检索文件时需要多次读硬盘，这样降低了检索速度。为解决这个问题，引入索引节点。将除文件名之外的所有信息，从文件控制块中剥离，并将这些单独存储在硬盘。随后使用一个盘块编号指向这个盘块。这样大大地减少了目录文件存储内容，使得内存能够一次性读入目录文件，加快了按名存取文件的速度。

Q：解释数据库的二级映像功能

A：当模式发生改变时，可以通过修改外模式/模式的映像从而保证外模式不变。又因为应用程序是依据外模式编写的，外模式不变就能够保证程序代码不用变。保证了数据与程序的逻辑独立性，称数据的逻辑独立性。当内模式发生改变时，可以通过对模式/内模式映像作出改变来保证模式不发生变化。模式不变，这就保证了应用程序不用改变，简称数据的物理独立性。

Q：解释数据库第一范式，第二范式，第三范式

A： 第一范式要求每一个属性只有一个值，不允许一个属性有多个值

若关系属于第一范式，在这之上若关系中每一个非主关键字段都完全依赖于主关键字段，没有部分依赖于主关键字段，则称其满足第二范式。举例：（学号，课号）

若关系属于第一范式，在这之上若关系中每一个非主关键字段都只依赖于主关键字字段，没有传递依赖，则称其符合第三范式。举例：（学号）年龄->身份证号->学号，年龄->学号。

Q：不满足第二范式和第三范式所带来的问题

A：若一个关系的主码有两个属性组成(x,y),且不满足第二范式，那么一旦依赖于 y 的某个属性的域发生了变化，那么就需要同时修改多个元组。同时缺少属性 y 的元祖无法插入到数据库中。依赖于 y 的某个属性的域被多次重复存储。造成数据冗余。

若一个关系中存在传递依赖，画图示意。插入异常，数据冗余。

Q:数据库故障及其恢复策略

A：事务故障，介质故障（磁盘丢失），系统故障（因 os，dbms 故障或断电故障）。恢复策略－ 定期转储整个数据库

－ 建立事务日志 ，要求事务的每一个修改操作都要写入日志文件，在修改数据库时，要求先将记录写入日志磁盘文件中之后，才能正式开始。undo 撤销，redo 重新做。undo 比 redo 需要更多 io 操作。redo 延迟更新。

－ 通过备份和日志进行恢复

Q:数据库的完整性约束条件

A：primary 要求被修饰属性不能取空值，unique 要求被修饰属性不能出现相同值，fpreign 要求被修饰属性值必须来源于被参照关系的取值，chech 要求被修饰属性值必须满足语义要求，default 修饰属性取默认值

Q：解释 GROUP BY，HAVING，JOIN ，ON 关键字

A：GROUP BY 是将查询结果集按一列或多列取值相等的原则进行分组。GROUP BY 子句的列名必须要是 FORM 子句中列表的列名。在使用 GROPU BY 时要求 FROM 子句选出来的列名，要么在 GROUP BY 子句中，要么就在一个统计函数中，否则将是错误的。在分组之后，使用 HAVING 可以筛选组。where 作用于表中元组，而 HAVING 作用于组。JOIN 用于链接两张表，而 ON 则用于给出这两张表的链接条件。

Q:解释聚集索引和非聚集索引的区别

A：聚集索引要改变数据的物理存储位置，会按照指定的索引关键字对数据文件进行排序。非聚集索引则不会对数据文件进行排序，仅仅只是为数据文件建立索引，不改变数据文件的物理存储位置。一张表只能有一个聚集索引，在建立非聚集索引之前，应当先建立聚集索引。否则建立聚集索引将会改变数据文件物理位置，这将会使非聚集索引被重新构造一次。聚集索引和非聚集索引都是用 B 树来实现的，不同点在于聚集索引的叶节点是数据页，而非聚集索引的叶节点是指向数据的指针。

Q：数据库的并发访问会带来哪些数据不一致的问题？

A：事务是并发控制的基本单位，多个事务同时对数据操作会导致事务的原子特性遭到破坏。1：当两个事务同时访问一个共享数据时，并修改这个数据时，后面的那个事务将会覆盖前面事务所进行的操作，这就是丢失修改。2：当两个事务一个事务读数据，另一个事务写数据时，就会造成不可重复读现象。3：当一个事务正在对数据进行修改，而这种修改还没有提交到数据库中，这时，另外一个事务开始访问这个数据，并使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是脏数据。

Q：并发控制的技术有哪些？

A：主要依靠共享锁和排他锁来完成。一个要读取修改数据的事务，首先要获得该数据的共享锁。共享锁限制事务只能读数据，允许多个事务同时获得共享锁。一旦获得了共享锁的事务就可以读数据，当事务要写数据时需要获取数据的排他锁。排他锁限制其他一切事务对数据的读操作，不允许其他任何事务获取共享锁，仅允许拥有排他锁的事务对数据进行读写。获取了共享锁的事务要将锁转变为排他锁，就必须等到所有事务释放共享锁之后，才能转变为排他锁。这就会引发出另一个问题，若有两个事务均获有共享锁，并都要转换为排他锁，这就会发生事务相互等待对方释放共享锁。因此将发生死锁。解决死锁的方法是使用更新锁，需要修改数据的事务在开始时便申请更新锁，因为一次只有一个事务能拥有更新锁，这就解决了死锁问题。

Q:简述三级封锁协议

A： 一级封锁协议：写数据时必须要加排他锁，直到事务结束时再释放，但是读数据却不要求加锁

二级封锁协议：在一级封锁协议之上，要求读数据要加共享锁，读完就可以释放。

三级封锁协议：在一级封锁协议之上，要求读数据要加共享锁，直到事务结束才能释放。

Q：解释触发器

A：触发器一般有两种，一种是 AFTER 触发器，一种是 INSTEAD OF 触发器。after 触发器是在某一类操作结束之后，触发器才会触发，而 instead of 触发器将替换某一类操作。触发器能够实现比 check 约束更复杂的约束条件，从而有力的保障了数据库的一致性和完整性。在数据库中存有 inserted 和 deleted 两张表，这两张表暂存了 sql 操作之后的数据内容。比如被 delete 操作所删除掉的数据都会存在 deleted 表中，在触发器代码中可以通过访问 deleted 表，查询出上一次 delete 操作所删除的数据，从而根据数据执行响应操作。

sql 语句为 create trigger 触发器名 on 表名 for 操作名

Q：解释存储过程和 T-sql 的区别

A：存储过程比 T-sql 代码执行速度快，因为存储过程是预编译，在第一次运行存储过程时就会被编译，优化；在随后调用存储过程便不再进行编译优化，而是直接执行。但 T-sql 代码每次运行，都需要编译优化过后再执行。这样就使得存储过程的执行效率高于 T-sql。当客户计算机操作访问数据库时，存储过程能够有效的减少网络流量。客户计算机只需发送调用存储过程的命令就可以了，但反观 T-sql，客户计算机需要传送整段 T-sql 代码。同时从另一方面讲，存储过程是很好的封装，能够避免用户对数据库进行非法的访问，提高了数据安全。

Q：简述关系模型

A：给定一组域 D1，D2，D3……Dn。D1,D2,…Dn 的笛卡尔积称为元组。D1,D2,…Dn 的笛卡尔的子集，称作在域 D1，D2，D3……Dn 上的关系。

Q:什么是软件生命周期?

A:一个软件从定义，开发，使用，维护，直至被弃用，要经历一个漫长的周期。通常就把这段时间称为软件生命周期。概括的说一般将生命周期分为三个阶段，软件定义，软件开发，软件运行维护。将软件生命周期中的各项开发活动的流程用一个合理的框架——开发模型来规范描述，就称是软件过程模型。它规定了各项任务完成的步骤。

软件定义：问题定义，需求分析，可行性研究

软件开发时期：概要设计，详细设计，编码和单元测试，综合测试

Q：常见过程模型及其主要特点。

A：瀑布模型：后一阶段的工作必须要等前一阶段的工作完成之后才能进行。这使得瀑布模型极其依赖前一段工作，若某一段的工作没有输出正确结果，那么后续所有工作输出的结果都将是不正确的。这样的依赖性的特征，也使得瀑布模型并不适合现代化软件开发，因为用户的需求总是在不停变化的。同时每个阶段之间都会产生大量的文档，工作量很大。

增量模型：首先实现一个核心的产品，这通常是第一个增量，这个增量能够满足用户的基本需求，但还有很多特征没有加上。然后立即将这个核心产品交与用户检验，用户对其使用评估，用户的评估结果都将指导下一个增量的开发。增量模型要求每一个增量都是可运行的产品。这样极大的增加了用户与开发人员的交流，能够很好的适应用户的需求变化。这样的特征要求软件满足开闭原则。在具体编程中可以使用设计模式中的依赖倒转原则和里氏替换原则来指导实现。

里氏替换原则:程序中能使用基类对象完成的任务,将其替换为子类对象也能够完成,且程序不会出错和异常。面向对象语言达成里氏原则的具体实现方法就是继承机制，最好的办法就是将父类申明为一个接口或抽象类。

依赖倒转原则：客户端代码要使用抽象类编程，而不是使用具体类编程。在依赖倒转原则中，以抽象方式耦合两个类之间的关系是实现依赖倒转原则的关键。依赖注入是传递对象之间依赖关系的指导思想。依赖注入将一个类的对象传入另一个类时，应当传入其父类对象，在程序运行中再通过子类对象覆盖父类对象。

螺旋模型：在快速原型模型和瀑布模型结合起来，在其中增加了风险分析。

A：什么是软件?什么是软件工程?软件工程常见的名词并解释?

Q: 软件：程序是按照一系列特定顺序所组织的计算机数据和指令的集合。与程序相关的文档是软件的一部分。软件就是程序+文档。

耦合：模块之间的联系。模块之间联系的越紧密，耦合性就越强，模块独立性就也越差。

内聚：模块内各元素之间的联系。元素之间的联系越紧密，内聚性就越高。

Q：什么是黑盒测试和白盒测试

A： 黑盒测试：只通过研究程序组件的输入和输出来确定程序是否有问题，不用考虑程序内部的逻辑结构和内部特性。又称为功能测试和数据测试。在设计具体的测试用例上，有等价类划分，边界值分析，因果图等方法。因果图用于分析应该为哪些情况设计测试用例。

白盒测试：需要对程序具体代码实现进行分析，了解代码的逻辑，设计测试用例，检测每条分支和路径。常用到的一些标准是

语句覆盖：设计足够多的测试用例，使得程序中每个语句至少能执行一次。

判定覆盖：设计足够多的测试用例，使得程序中每个判定至少都获得一次真值和假值。每个分支至少通过一次。

条件覆盖：设计足够多的测试用例，使每个判断中每个条件的可能取值至少满足一次，但未必能覆盖全部分支。

单单仅看即使每个 if 语句框至少取一次，不考虑组合。

判定/条件覆盖：设计足够多的测试用例，同时满足条件覆盖和判定覆盖。

条件组合覆盖：每个判定框之间相互组合。每个 if 语句里之间条件逻辑之间相互组合。

数据结构代码手写：

插入排序(done)，折半插入排序(done)，快速排序(done)，归并排序(done)，堆排序(done)，KMP(done)。

离散数学：

求解线性递推关系的解。

Q：解释 ICMP 及其原理

A：ICMP 是网际控制报文协议，用于在主机，路由器之间传递控制消息。ICMP 是使用 IP 分组，是一种无链接的协议。常使用的命令是 ping，用于测试网络是否通畅。

Q：解释 DNS 域名系统（分布式数据库，联系到存储过程和 T-sql 代码，联系到 b+树和字典序 域名 -> IP）

A：DNS 是一个数据库系统，这个系统能够把互联网上的主机名转换为 IP 地址。这样用户就可以不用去记住主机的 IP 地址，而是通过主机名去链接它。DNS 系统实质上是一个数据库，这个数据库上记载了主机名和 ip 地址的之间的对应关系。它被设计成分布式数据库系统，并采用客户服务器方式。之所以一个主机名会有多个域，是因为 DNS 是分布式数据库，为了能够快速检索到准确的对应关系，需要为这个数据库建立索引，而将一个主机名划分成多个域，就是达成这一目的的手段。

DNS 查询通常有两种方式，一种是迭代查询，一种是递归查询。DNS 的查询请求通常被封装在 UDP 数据报中。

Q：解释 ARP 协议(IP -> MAC)

A：ARP 协议是用来寻找 IP 地址所对应的 MAC 地址，因为最终在实际网络链路上进行传送数据帧时，必须要使用硬件地址。因此仅仅知道 IP 地址还不能传送数据，需要找到 IP 地址所对应的 MAC 地址。当一个主机需要查询一个 IP 地址所对应的 MAC 地址时，会在全网进行广播，一旦当目标主机收到该数据帧之后，就会以单播的形式会送自己的 MAC 地址。

Q：解释 DHCP 协议

A：DHCP 主要是用来给一台新加入计算机网络的主机自动配置 IP 的地址的协议。这样一台主机就不需要人工配置 ip 地址，使得主机可以即插即用。当一台新主机联入到网络后，就会立即使用 UDP 发送广播报文，寻找网络中的 DHCP 服务器。当 DHCP 服务器接收到这个信息之后，就会对此进行应答为其分配 ip 地址。

Q：解释 CSMA/CD 协议**（联系发散）主要是用于广播信道，仅最大努力交付不可靠**

A：CSMA/CD 协议主要用在共享式网络当中，随着交换机的广泛使用，共享式网络转变至交换式网络，这使得 CSMA/CD 协议逐渐消失。CSMA/CD 协议是用来让多台计算机共用一个线路进行通信的协议。CSMA/CD 协议要求网络上的计算机在发送数据之前，需要对通信线路进行监听，只有当信道空闲时，主机才能发送数据。因为数据在信道上传播有延迟，所以即便是主机在发送数据时也仍需要不断监听信道。一旦发现别的主机也在向信道上送往数据时，就必须停止发送数据，等待一段时间之后再发送。(截断二进制指数退避算法)

最短帧 64B，发送时延 = 传播时延 x 2。争用期 51.2 us

Q：停止等待协议（数据链路层通信协议）

A：停止等待协议就是每次发送完数据报之后，必须要等到确认消息之后才能继续发送下一个。在停止等待协议中有一个超时计时器，如果在计时器到期之前都没有收到对方的确认，就会重新发送数据报。

Q：ARQ（用于数据链路层通信）与滑动窗口协议

A：连续 ARQ 协议是数据链路层的，ARQ 的窗口是固定的。

当发送窗口和接收窗口的大小都等于 1 时，就是停止等待协议。

当发送窗口大于 1，接收窗口等于 1 时，就是回退 N 步协议。

当发送窗口和接收窗口的大小均大于 1 时，就是选择重发协议。

滑动窗口协议中的窗口是可变的，它受限于流量控制与拥塞控制，是接收窗口与拥塞窗口的最小值。

Q:解释为什么 TCP 需要三次握手

A:三次握手是为了避免出现如下所述情况。当客户端发送一个链接请求至服务器端，这个链接请求因某些原因在网络中滞留了很长时间但并未丢失。这对客户端来说，将迟迟收不到服务器端的链接确认消息。若这个等待时间过长，客户端将会自动释放掉这个链接，认为链接无法建立。但是链接请求却能正确达到服务器端，只是时间用了很久。这时候，如果不采用三次握手，即服务器端只要回送确认消息，这个链接就建立了。但是客户端却认为这个链接失效了。这样一来，服务器端将会一直等待永远客户端发送数据，服务器资源就被浪费掉了。

Q：IPv4 地址缺乏的解决办法以及 IPv4 的替代方案以及 IPv4 和 IPv6 如何相互通信？

A：IPv4 地址缺乏的解决方法是使用 CIDR 编址，在 CIDR 中取消了传统 IP 地址中分类的概。将 ip 地址划分为前后两个部分，前面的部分就叫做网络前缀，用来指明网络，后面的部分就用来指明主机。通过这样灵活的 ip 地址划分方式就解决了 ipv4 地址缺乏的问题。

解决 IPv4 和 IPv6 相互通信的方式有两种。一种从硬件上，为每一个路由器都配备 ipv4 和 ipv6 协议。这样路由器就能根据 ip 地址的不同选择不同的协议。另一种是从软件上，只要把 ipv6 数据报所有部分全部封装进入 ipv4 的数据报部分。这样 ipv6 的数据报就能在 ipv4 的网络上进行传输。

Q:解释虚拟网是什么

A：虚拟网 vpn 就是利用公用互联网作为 n 个专用网之间通信载体，虚拟的意思就是表面上机构内的所有计算机看起来好像是一个专用网里交流，但是实际上还是通过了互联网进行交换信息，只是在效果上很像专用网。

Q：解释拥塞控制，和它的具体做法。

A：当一个网络对资源的需求超过了资源所能提供的时候，就会出现网络吞吐量大幅下降的现象。拥塞控制就是为了防止这一现象的产生。拥塞控制可以防止过多的数据注入到网络中，这样就可以使得网络中的路由器和链路不致过载。

拥塞控制：设置一个拥塞窗口，再设置一个拥塞阈值。在 tcp 报文段中还有一个叫做窗口的字段，这是用来进行流量控制的。在慢开始阶段，每一次确认都会使拥塞窗口增大。当拥塞窗口增大至拥塞阈值时，这时将采用拥塞避免算法。拥塞窗口每一个传输轮次只会增大一。

出现超时：重新设置拥塞阈值为拥塞窗口的一半，并将拥塞窗口置为 1.

连续收到 3 个 ack：重新发送丢失报文，并重新设置拥塞阈值为拥塞窗口的一半，拥塞窗口=拥塞阈值。

Q：电路交换，报文交换，分组交换**（联系到单周期数据通路和流水线）**

A：电路交换要求建立一条专用线路。报文交换与分组交换不需要建立专用线路，采用存储转发的方式转发。区别在于数据报的粒度大小。

计算机网络 OSI 体系结构

物理层：频分复用，时分复用，码分复用。

数据链路层：ARQ 协议，CSMA/CD 协议

网络层：IP，CIDR，RIP，OSPF，BGP，ICMP，VPN，NAT

运输层：TCP，UDP，拥塞控制，三次握手

应用层：DNS，DHCP，

物理层作用：因为物理的传输媒体有很多，像是双绞线，同轴电缆，光纤等。为了能使这些不同的传输媒体之间能够相互传输信号，需要对他们的一些物理特征进行规定。像是规定电气特性，功能特性，机械特性等。

数据链路层作用：在物理层服务之上，在数据链路层实体之间建立链路链接，传输以帧为单位的数据包。使用流量控制和差错控制，使得有差错的物理链路，变成无差错的数据链路。

网络层：通过 IP 地址，将离散的 MAC 地址抽象形成拓扑图结构。多个 MAC 地址会对应着同一个 IP 网络号，一个网络号对应一个图顶点。在这之上就能图结构之上，实现路由选择，分组转发和拥塞控制，流量控制等。

Q：协议与服务

A: 网络层向上为传输层提供无链接，尽最大努力交付的数据服务

传输层向上为应用层提供端到端的可靠传输服务，流量控制，差错控制等。

移动通信略

802.11 mac 三地址数据帧

802.3

Q：堆排 vs 快排，平衡二叉树 vs b+树

A：就堆排序来讲，在调整堆的时候，总是需要频繁地访问父结点 i，和它的孩子结点 2i 和 2i+1。当 i 增大的时候，i 与 2i 之间距离也会增大。一旦 i 和 2i 的距离大过了 cache 容量，那么就会 cache 不命中，那么这时候就需要去访问内存。并且在一个有虚拟内存的操作系统上，当 i 和 2i 之间的距离大过了一页的容量，且 2i 所在页面又正好不在物理内存中，这时候需要从磁盘上读取数据。但反观快速排序，总是连续的访问数据，空间局部性十分良好。

我们知道操作系统磁盘管理的基本单位是盘块，而文件系统的基本单位是簇。即便是只需要一个簇上的一个字节也需要读入整个簇。B+树中之所以一个结点会有多个关键字，一方面是为了能让一个结点充满一个簇，不出现浪费。另一方面是为了减少树的高度。反观平衡二叉树，虽然在算法层面查找一个关键字所需时间是 long2n，但是他实际物理存储是极其不规范的。逻辑上相邻的位置，在物理存储上却很远。虽然比较次数少了，但是 i/o 读写却增多了。这使得平衡二叉树的实际效率不如 b+树。同时数据库经常需要进行范围查找，而在平衡二叉树和 b 树上进行范围查找时，是极其不方便的，因为他们只能随机查找，只有 b+树支顺序查找。

Q：IP 地址与面向对象分析

A：首先我认为他们都提供了一种分析和处理复杂系统的思路，较之于早期直接使用 mac 地址进行通信，ip 将多个 mac 地址映射到同一个网络号。将一个网络也就是 mac 地址的集合，作为系统分析的基本要素。同理较之于早期的结构化分析，面向对象分析将多个过程与数据结构集合形成对象，将对象作为系统分析的基本要素。不再关注数据的流向，而是关注对象与对象之间的通信。

Q：什么是第一范式

A：第一范式要求元祖的每个属性只含有一个值，若不满足第一范式，若在以后新增属性时，将出现更新异常。例子，（学号，课程），课程含有多个值。

Q：什么是 BCNF

A：非平凡函数依赖：若 x->y,Y 不为 x 的子集，那么就称 x->y 是非平凡函数依赖。

平凡函数依赖：若 x->y,但 y 是 x 的子集，那么就称 x->y 是平凡函数依赖。

部分函数依赖：若 x->y,但存在 x 的真子集 x1，使得 x1->y,那么就称这个是部分函数依赖。

传递函数依赖：若 x->y,y->z，则有 x->z,那么就称这个为传递函数依赖。

在第一范式的基础之上，若关系中的所有非平凡函数依赖的左边，都是候选码，那么这个关系就是 BCNF

Q：如何测试出 90%的 bug

A：我会使用白盒测试法中的条件组合覆盖设计尽可能多的测试用例，来测试程序。

Q：什么是概要设计

A：按照业务功能，将功能分解为不必再分解的的模块，使得模块达到高内聚，低耦合，每个模块完成一定的功能，为一个或多个父模块服务，也接受一个或多个子模块的服务。然后将这些模块的调用关系形成软件结构图。

Q：用例图作用

A：用例图，是用来描述人们如何使用一个系统，保证在不揭示系统内部构造的前提下，描述系统的业务功能。用例图被用来获取客户的需求。但用例图是静态的，因此需要活动图来动态的描述业务流程。

Q：活动图的作用

A：活动图用来描述业务用例实现的工作流程，在活动图使用垂直的线，可以划分出泳道。一条泳道就表明某个部分必须要完成的职责。

Q：什么是类图？

A：从泳道图识别出具体的实现类以及该类所需要完成的操作，这个类就需要完成泳道所要求的职责。但类图是静态的模型，无法表示对象之间的交互。因此就需要顺序图，来描述对象与对象之间的交互动态模型。

Q:什么是顺序图

A：顺序图描述了对象的生命周期，对象与对象之间的交互关系。

Q：为了能够从一个更高的角度去描述一个系统，就需要将多个类图组织起来，形成包图，然后确立包图与包图之间的关系，从而形成整个系统的体系结构。

Q：什么是数据字典

A：数据字典是给数据流图上的每个成分加以定义和说明，主要目的是为了程序员与用户沟通交流。

A：结构化分析，通过顶层数据流图与用户交流，DFD 图仅描述功能，通过此与用户沟通交流。然后对顶层数据流图逐步求精。但是数据流图是静态的。因此需要状态转换图，描述系统工作

Q：什么是面向对象分析

A：需求->业务用例图->活动图(可以变形为泳道图，泳道就是一个矩形，要求矩形内对象必须要完成的操作)->从图中识别出类，形成类模型->确定类模型中对象与对象之间的交互模型(顺序图)->类模型与类模型形成包图->包图与包图之间形成系统体系结构。

Q：什么是结构化分析

A：数据流模型（描述系统中数据处理的过程,关系数据的流动和数据的转换）->从数据流图中识别出模块->形成模块与模块之间的关系(软件结构图)->为每个模块设计算法与数据结构（流程图，盒图，PAD 图，HIPO 图，判定表与判定树，PDL）

Q:RAID0 与交叉存储器

A:我认为他们都体现了流水线化的思想。为了加快数据传输率，RAID0 与交叉存储器都使用冗余的存储体，将原本连续的数据进行切割，然后分别存储在不同存储体中。在 RAID0 中是通过条带化的方法，而在交叉存储器中通过将物理地址中的体号字段置于低位，从而将连续的物理地址映射到不同存储体中。然后这样一来，就能以流水线的方式去读取存储内容。

Q：解释 RAM，ROM，掩摸 ROM，PROM，EPROM，EEROM，FLASH

A：RAM 随机读写存储器，又可以分为 SRAM 和 DRAM。DRAM 会发生漏电现象，这使得 DRAM 需要不停的补充电荷，称补充电荷为刷新。而 SRAM 不需要刷新，但它的集成度低。主存常用 DRAM 构成，高速缓存 cache 通常用 SRAM 构成。

ROM 是只读存储器，其中的信息不会因为去掉电源而丢失，再次加电时存储信息依然存在。

PROM，是可以编程的只读存储器但仅能编程一次。

EPROM，是可以擦除重写的只读存储器，用紫外线擦存储内容，用加电的方法写入内容。

EEPROM，是电可擦除重写的只读存储器，EEPROM 可以使用电擦除存储内容，不需要放置于特殊的擦除器中，在电脑上就能擦除。常用于银行卡

FLASH，也是 EEPROM，只不过它的擦除和写入速度很快，为强调其速度快所以才称它为 FLASH，常见的有 U 盘，MP4 等

Q：cpu 寻址方式

A：隐含寻址，立即数寻址，直接寻址，简介寻址，寄存器寻址，寄存器间址，相对寻址，基址寻址，变址寻址。

为什么要对寄存器编号，是为了进行寄存器寻址。

8086 十六位 cpu，可寻址 2^ 20 的地址空间。使用段+偏移的方式寻址 2^20 地址空间

Q：I/O 端口与 I/O 接口

A：在一些寄存器如数据寄存器，状态寄存器，控制寄存器，命令寄存器之上制定控制 I/O 设备 I/O 控制逻辑就形成了 I/O 接口。I/O 接口是对 I/O 设备的抽象，它屏蔽了 I/O 设备的机械特性，数据格式等细节。I/O 接口中的数据，状态，控制寄存器为 I/O 设备与总线之间提供了 3 中 I/O 信息的传输通道，将这些传输通道称为 I/O 端口。

I/O 端口编址，独立编址和统一编址。

Q：解释中断向量

A：中断向量就是中断服务程序的入口地址。在 8086 中，有一部分内存区域被固定为中断向量存放的位置，称为中断向量表。中断向量占 4B，被分为两个部分，一个部分是段基值，另一个部分是段偏移。CS（段寄存器）和 IP（段偏移）。形成物理实际地址的方式是 CS 寄存器左移 4 位，然后加上段偏移，就形成了物理地址。

Q：什么是 8086 的实模式和保护模式(联系发散到 x86 指令集，intel 向下兼容金手铐)

A：给定一个基地址，进程能够不加以限制的访问 64kb 连续的空间。通过改变段寄存器的内容，一个进程可以随心所欲地访问内存中的任何一个单元，而丝毫不受限制。不能对一个进程的内存访问加以限制，也就谈不上对其他进程以及系统本身的保护。

Q：什么是芯片组

A：芯片组主要是指主板上的南桥芯片，北桥芯片和 bios 芯片等。北桥芯片是 cpu 用来与主存，显卡，南桥芯片进行通信的通道。南桥芯片内部则集成了许多 I/O 设备控制器，像硬盘，键盘等，所有外设的数据都会在南桥芯片中集合，然后送外北桥芯片，再到达 cpu。bios 芯片是一个只读存储器，用于开机过程中各种硬件设备的初始化和检测。在计算机刚启动时，无法从硬盘中获取数据，因为内存没有任何能够启动并处理硬盘的服务程序代码。bios 芯片中的代码就会经过南桥，北桥，送至 cpu 中，并开始处理。

随着技术的发展，现在 cpu 已经不需要通过北桥去访问主存，北桥中的主存控制器被移到 cpu 的内部结构中，这样大大提高了数据传输率。后来显卡的 pcie 控制器也被移动至 cpu 内部，这样北桥大部分主要功能都被移动至 cpu 内部，北桥也没有存在的意义，将北桥中剩下的功能合并至南桥芯片中。

Q：简单工厂模式与工厂模式的区别

A：简单工厂模式集中了所有产品的创建逻辑，当系统需要扩展时，将不得不修改工厂的代码，这是因为在简单工厂模式中负责创建对象的是一个具体工厂类。若对工厂进行抽象，将工厂做为接口，将对象的创建交付给实现接口的具体工厂类，这就是工厂模式。这使得系统需要扩展时不需要修改客户端原有代码，只需增添新的工厂即可。

Q：解释单例模式

A：一个使用单例模式设计的类，在整个程序生命周期里，它只能创建一个对象。具体做法就是将该类的构造函数申明为私有，并声明一个静态私有类对象，对外只提供一个静态的工厂方法，用于构造该类对象。因为没有抽象层，所以单例模式很难进行扩展。

Q：适配器模式

A：适配器模式有类适配器和对象适配器。若客户端代码针对一个 Target 接口或类进行编码，当系统发生变化，要求 target 提供不一样的操作时，这时候就可以使用适配器模式。设置一个适配器类和一个适配者类，在适配者类中实现特殊操作，然后通过适配器类将转换 Target 的操作替换为适配者类中实现的操作。这样就可以不用修改客户端代码，达成新需求，完全符合开闭原则。使用适配器模式要求目标类中的方法应为虚函数，因为适配器要通过继承去覆盖原方法。

Q：什么是桥接模式

A：在一个抽象类中，声明一个接口成员，通过设值注入的方式，为这个接口成员赋予具体实现类，这就是桥接模式。

Q：什么是组合模式

A：设置一个抽象元素类，再设置一个容器类，这个容器类继承于抽象元素类，同时容器类申明一个父类集合对象，这个集合对象能够收集所有派生于父类的对象。若在抽象元素类中声明了对元素的增删改查操作，那么这个组合模式就是透明组合模式。若在抽象类中没有声明对元素的增删改查操作，而是在容器类中增添了增删改查操作，这样的组合模式就是安全的组合模式。

Q：解释继承和装饰者模式的区别**（装饰者模式与组合模式的区别）**

A：设有一个接口类 A，再设一个实现接口类 A 的具体类 B，在类 B 中声明了有一个接口类 A 的对象，并通过设值注入为这个对象赋予值，称这个类为抽象装饰者类。可以通过继承抽象装饰者类派生新的方法。这样一来，凡是所有实现接口类 A 的对象，都可以将自身传递到具体装饰者类中，然后就可以调用具体装饰者类中

新增的方法。继承是静态的，客户端代码不能控制改变方法和时机。而装饰者模式是动态的，可以在程序运行过程中按需添加功能。

Q：外观模式

A：略

Q：什么是代理模式

A：给某一个对象提供一个代理，并由代理对象控制对原对象的引用，这就是代理模式。抽象主题角色申明了代理对象与被代理对象的共同接口，在客户端代码中要求使用抽象主题角色来编码。代理对象内部包含了对被代理对象的引用，被代理对象中实现了真实的业务代码，而代理对象仅仅只是引用。

Q：什么是命令模式

A：命令模式的核心思想就是将一个命令封装为一个对象，调用者与被调用者之间不存在直接引用，而是调用者通过命令类间接的引用被调用者的方法。在调用者类中声明抽象命令类，并通过设值注入具体命令类。通过这样的方式，就能使调用者与被调用者之间完全解耦，因此调用者可以对应不同的被调用者。

Q：什么是组合命令模式

A：组合命令模式是组合模式和命令模式的结合。为抽象命令类新增一个容器类，这样就可以达到命令的批处理。

Q：什么是迭代器模式

A：将聚合对象中的数据操作从其中分离出来，形成一个单独数据操作类，这样一来就可以不用暴露这个聚合对象的内部表示，这样就被称做迭代器模式。聚合类就仅仅只存储数据，不对外提供访问这些数据的具体方法，迭代器类则负责实现访问数据的具体方法。

Q：什么是观察者模式**（桥接模式与观察者模式区别）**

A：系统中有观察者与被观察目标，观察者监听观察目标，一旦当观察目标发生改变，那么观察者将收到消息通知，根据目标变化自动作出相应动作，这就是观察者模式。设置一个抽象目标类，和一个观察者接口，在这个抽象目标类里声明一个观察者集合。抽象目标类应具有 attach 与 detach 操作，attach 用于注册观察者，而 detach 用于撤销观察者。在目标类的编码中要求通知每一个观察者。

Q：各类排序算法最坏最好比较次数

A：直接插入排序（稳定）：

比较次数最小 n-1，0 次移动。

比较次数最大(n+2)(n-1)/2,(n+4)(n-1)/2 次移动

直接插入排序（稳定）：

最好 0 次移动，最坏(n+4)(n-1)/2 次移动

移动次数不变，仅比较次数减少

冒泡排序（稳定）：

比较次数最小 n-1 次，最多 n(n-1)/2 次。

简单选择排序（稳定）：

比较次数总需要 n(n-1)/2 次

最小需要 0 次移动，最多需要 3(n-1)次移动

快速排序(不稳定)

最好 o(nlog2n),最坏 o(n^2)

堆排序（不稳定）

最坏最好都是 o(nlog2^2)

归并排序(稳定)

最坏最好都是 o(nlog2^2)

按稳定划分：仅快排与堆排是不稳定的，除此之外都是稳定的

按情况划分：堆排，归并，最好最坏都是 o(nlog2^2)，简单选择最好最坏比较次数都是 n(n-1)/2 次

特殊记忆：(n+2)(n-1)/2，(n+4)(n-1)/2

Q：结构化分析与面向对象分析的区别

A：在分析设计同一个系统上，结构化设计与面向对象设计的视角尺度不一样。结构化分析视角尺度大，大到认真仔细观察系统中的每一个过程。仔细观察数据在过程与过程之间的流向。面向对象分析视角尺度小，概略观察系统中的对象。概率观察对象与对象之间消息的传递。在一个大型系统会存在十分多的过程，如果仍不能以对象这个更高的视角去审视系统，那么搭建维护一个大型系统是十分困难的。

Q：单片机与 CPU 的区别

A：单片机不仅包含了 cpu，还有内存，各类 I/O 接口等，看起来就像是一个小型的计算机系统。

Q：交叉存储器（物理）与 RAID

A：RAID0 与交叉存储器都需要将一个连续的数据切分，然后将它分布的存储在不同的存储体中。在 RAID0 中这样的操作被称为条带化，在交叉存储器中则是通过将物理地址中的“体号”字段放置于低位，从而将连续的物理地址映射到不同存储体中。一旦一个连续的数据被离散的存储在不同的存储体中时，这时候读取数据就能够以流水线的形式连续的读取，多个存储体同时并行工作极大的增加了数据传输率。但是这样做风险也是极高的，在 RAID0 系统中，虽然读取速度很快，但是任何一个磁盘损坏，会使整个系统失效。因此 RAID0 以后的系统，为了增加数据的安全性，通常系统中会有一份完整的数据库备份。这样 RAID 系统中任意一个磁盘损坏只需要更换损坏的磁盘，RAID 系统就能够自动的重建数据。

Q：cache 映射方式

A：直接映射与组相联映射地址结构相同，区别在于 cache 结构，直接映射的 cache 每一个块号只能装入一个内存块 ，组相联映射的 cache 按组划分，组内是全相连映射。全相联映射。

cache 替换算法仅在组相联映与全相联映射中使用。

Q：流水线冒险

A：结构冒险（指令访问内存冲突取指令，寄存器写冲突寄存器读），数据冒险（数据更新不一致，旁路），控制冒险

Q：微程序控制器与硬布线

A：硬布线，组合电路逻辑直接产生电路信号控制 cpu。微程序控制器将一条指令所需要产生的控制信号，存于控制存储器中。一条指令对应着一个微程序，微程序由多个微指令组成，微指令产生控制信号。微程序控制器就像是 cpu 内部中的 cpu，它有普通 cpu 一样的结构和工作原理，取指，译指，运行等等。

微指令产生控制信号，微指令的编码方式有很多。一种最简单的方式就是一个位对应一个控制信号，这样直接编码就无须译码，信号产生速度极快，但是指令会很长。因此也有译码编码方式。

Q：CF 与 OF

A：CF 借/进位标志，CF = Cin 异或 Cout，cin 是减法进一。加法时 CF=1 表示有进位，减法时 CF=0 表示有借位。OF＝Cn 异或 Cn-1，Cn 是最高位进位，Cn-1 是 n-1 的进位。有进位的时候不一定有溢出，有溢出的时候不一定有进位。4 位无符号数 3+5。4 位符号数-2+（-4）。

Q：什么是半加器和全加器

A：半加器 = 一个异或门+一个与门。半加器只有两个输入，无法计算进位。全加器 = 两个半加器。全加器有三个输入，比半加器多了一个进位输入。

Q：行波进位加法器与超前进位加法器

A：行波进位加法器是将多个全加器串联而成，高位运算总需要等待低位云散全加器的进位输出，因此速度很慢。超前进位加法器，通过计算可以求得高位的进位输出。例如一个 4 位超前进位加法器，一旦知道了最低位的三个输入，就能并行计算出每四位的进位输入。因此设置一个超前进位逻辑，这个逻辑为每一个全加器提供进位输入。因为超前进位逻辑的计算是并行的，能够同时提供进位输入，因此高位运算便不再需要等待低位运算。

Q：程序装入方式

A：绝对装入方式，逻辑地址与物理地址相同。

可重定位装入方式，逻辑地址与物理地址不同，在程序被装入后再修改其指令地址，以及指令中的数据地址。装入后便不能移动位置。

动态运行时的装入方式。

Q：内存保护方案

A：cpu 中设置上下限寄存器，采用重定位寄存器和界地址寄存器。

Q：程序链接方式

A：静态连接，编译后的各个小模块直接拼成一个大模块，将这个大模块称为“可执行文件”，以后都不再拆开作为一个整体调度加载运行。

装入时动态连接，在装入一个目标模块时，若发生了一个外部模块调用，将引起装入程序去找出相应的某个目标模块，并将它装入内存，然后修改模块中的地址。

运行时动态连接，在执行过程中，当发现一个被调用模块未装入内存时，OS 立刻去寻找该模块，然后装入内存并连接到调用者模块上。

Q：作业与进程的区别

A：作业是用户需要计算机完成的某项任务，是要求计算机所做工作的集合。一个作业至少有一个进程组成。进程是执行作业中某一任务的实体，完成一个作业通常需要多个任务，系统资源以进程为单位分配。

Q：什么是微内核

A：微内核是操作系统的子集，它包含了操作系统最基本核心的功能。而将一些诸如负责虚拟存储器管理，I/O 设备管理这些功能模块从操作系统本身中剥离，将他们作为操作体系中的服务进程，这些进程为操作系统提供服务。微内核是基于客户端/服务器模式的操作系统。

Q：模拟信号，数字信号，基带信号，带通信号

A： 模拟信号是连续的类似于正弦波一样，数字信号是离散的脉冲信号。

基带信号，数字信号经过调制后再传输（曼切斯特编码，差分曼切斯特）。

带通信号，数字信号经过调制后再传输（调幅，调频，调相）。

Q：扩展以太网

A：在物理层上扩展以太网，使用转发器扩展以太网地理范围。在数据链路层上扩展以太网，使用网桥和交换机

Q：ppp 协议

A：与某个 ISP 进行通信时使用的协议

语法分析与上下文无关文法（CFG）

在上下文无关文法中一个句子可以对应着多个派生。去掉派生顺序，仅仅只保留“替换”，那么就能形成派生树（CFG-only）。

二义性例子：

Gexp1：E->E+T

T->TF

Gexp2：E->E+E|E/E|EE

Gexp1 和 Gexp2。文法 Gexp2 有二义性，但文法 Gexp1 却没有。我认为出现这样的原因是因为在 Gexp2 中将所有运算符的优先级视为一样的。而在 Gxp1 中能够表达运算符的优先级的原因是在于对不同优先级运算操作的分批定义。例如在 Gexp1 中将加法和减法这两个运算优先级相同的运算定义为由同一个语法变量产生，而乘法和除法这两个运算优先级相同的运算为由另一个语法变量产生，并且这两个语法变量有先后次序关系，要

想产生乘法操作必须先产生加或减操作。但是在 Gexp2 中所有的运算操作都被定义由同一个语法变量 E 产生。所有运算操作都是相等级别的，在第一步便可以选取任意的产生式来产生同一个句子，这样便造成了二义性。同时这部分内容提到关于如何判定一个文法是否为二义性是一个不可解的问题。

词法分析

相关理论问题：

NFA 与 DFA 等价，都能识别相同的语言。

构造与 NFA 等价的 DFA。

Q:DFA(determinisitic finite automaton)与 NFA(non-determinisitic)的区别

A:区别在于状态转移函数，DFA 的状态转移函数都只有一个状态与之对应,f(q,a) =p。而 NFA 的状态转移函数可能会对应多个状态 f(q,a)={p1,p2,p3,p4….pn}。

词法分析就是采用正则文法。

FA 是正则语言的识别器，FA 识别右线形文法。FA 与 3 型文法等价。

构造识别右线形文法的自动机 FA。每一个语法变量就是一个状态，每次吃掉一个就移动到下一个状态。（顺着吃）

构造识别左线形文法的自动机 FA。（反着吃，形如 A->w 这种需要构造新状态）

句法分析

相关理论问题：

CFG 转 GNF，PDA 的状态转移函数 f (q,a,Z) = {(p1,r1),(p2,r2)}，左递归消除。

直接左递归消除

A->Aa1 A -> b1

A ->b1 = A -> b1B

B -> a1

B -> a1B

间接左递归消除，化简为直接左递归。

1) LL(1)

first 集，follow 集，select 集。这些集合在计算时，都按语法变量计算。难点在于空串 e，按语法变量依次计算。

step1. 先计算 first 集，使用 first 集求解 follow 集。

step2. 对生成式 A->a（其中 a 为语法变量），select(A->a)，依赖于 firsrt(a)，如果 firsrt(a)中不含有 e，那么 select(A->a) = firsr(a)。否则，select(A->a) = {firsr(a) - e} + follow(a)。特殊的，对于生成式 A->e。select(A->e) = follw(A)

step3. 求出 select 集之后，在 select 集之上有递归分析法和非递归分析法。

step4. 出错处理将每个语法变量的 follow 集填入到预测表中，并标记为 synch，表项中为 synch 的单元将出栈同时报错。

option1. 递归分析法：为每一个语法变量书写一段识别程序，按照生成式书写递归程序。当一个语法变量有多个 select 集合时候，例如 select(S->AaS) = {a}，select(S->BbS) = {c,b}，select(S->d) = {d}。这些都是语法变量 S 的情况，应该将将这些集合在一个函数体 ParseS(){}里面。若当前字符是 a，那么就应该按照 S->AaS 编写程序。

option2. 非递归分析法：将 select 集合形成预测表，将所有终结符形成列，语法变量作为行。设置栈，每次将生成式的右部压入栈。

2) LR(0)

ACTION-GOTO 表。ACTION-GOTO 表有两个表项一个 ACTION，一个 GOTO 表。GOTO 表的单元项是状态。ACTION 的单元项是动作，其中动作有 Sn 与 Rn 动作，Sn 动作即使移入并转移到 n 号状态。Rn 动作即是使用 n 号生成式规约，每一次规约过后，需要查看 GOTO 表进行状态的跳转。

step1:将文法改造成为增广文法

step2:寻找等价项目。

LR(0)中存在归约/移进冲突。E->T. 和 E->T.*。究竟是移进，还是规约？

Q：为什么需要中间代码

A：汇编器使用中间代码将其最终翻译成为机器语言，之所以需要中间代码的是因为这样做就能够屏蔽掉具体机器的指令集，同一种中间代码能够被多种不同机器的汇编器翻译成目标代码。能够在中间代码的基础之上，进行与机器无关的代码优化。

printf(“%020d”,i)，用于设置宽度为 20，不够补 0

for(j = 1; j ++ < 4;j ++) ,虽然比较结果为 false，但是 j 仍然需要自增,比较结果并不影响 j 的自增。

a[10-10]，先运算括号内，然后再取值

scanf(“i=%d”,&i),在输入的时候，要求按照格式依次输入。

scanf()在使用指针时，不需要加上&

scnaf 在对指针所指向的区域赋值的时候，总会清空掉上一次赋值

%只能作用于整数。

将数组名指向一个字符串是错误的，只能用指针指向

strcmp(str1,str2)

若 str1=str2，则返回零；

若 str1<str2，则返回负数；

若 str1>str2，则返回正数

main(int abc,char **arg)

++和- -的优先级大于* （* ++ptr）.age,先++再*

宏可以定义在函数任何一个位置

a+ 若文件存在，打开文件，并指向它的末尾。不存在，则创建文件。

r 表可读，r+表可读写文件不存在，不会建立文件

w 打开只写文件，若文件存在，则文件长度清零，即文件内容会消失，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，则文件长度清零，即文件内容会消失，若文件不存在则建立该文件

记住规则，a 是附加，w 是写，r 是读。+号表示可读可写，若文件不存在时，只有 r 不会建立文件。w 和 a 都要建立文件，w 直接无脑暴力覆盖文件。

如果 EOF 返回值为 1，说明达到了文件的末尾。

char * fgets(char * s, int n,FILE *stream);

从 stream 中读取 n-1 个字符。

int putchar(int ch);

int puts(const char *string);

一个输出字符，一个输出字符串。

按’\0’确定结尾，如果没有’\0’，将会出错

strlen，puts，

char *strcat(char *dest, const char *src);

功能：将 src 字符串连接到 dest 的尾部，‘\0’也会追加过去

fpus(char *s,FILE *fp)

成功则返回非负数，否则返回 EOF

getch 与 getchar。getchar：等待用户回车之后，读取缓冲区内容。getch，按一个字符立即响应。

fopen 打开错误，则返回 null