2022

408 计算机学科专业基础

细节和必要的知识点汇总

橙子 MUSIC

2022 年 12 月 18 日

目录

摘要: 本文自 2022 年 10 月 5 日开始整理，将 408 大题的考察内容根据知识模型进行汇总分析，同时补充相应的必备知识。

一、数据结构:

历年考试大题考察内容及特点		
年份	41 题	42 题
2009	图单源最短路径	链表双指针查询（线性表算法）
2010	散列表、平均查找长度	数组原地逆置（线性表算法）
2011	有向带权图、邻接矩阵、关键路径	两个数组的中位数（线性表算法）
2012	哈夫曼树合并有序表	公共链表节点（线性表算法）
2013	数组中的主元素（线性表算法）	查找长度
2014	二叉树带权路径长度（后序遍历）	带权无向图，Dijkstra 算法
2015	链表删除重复节点（线性表算法）	邻接矩阵
2016	正则二叉树	数组快速划分（线性表算法）
2017	二叉树中缀表达式（中序遍历）	Prim 算法
2018	未出现的最小正数（线性表算法）	最小生成树
2019	链表逆置插入（线性表算法）	设计循环队列
2020	三元组最小距离（线性表算法）	前缀编码和哈夫曼树
2021	科尼斯堡七桥问题（图算法）	计数排序
2022	数组形式的二叉排序树验证（中序遍历）	TopK
2023 押题	找出数组前 k 个数据中的最小值（线性表算法，使用双栈）	有向图的存储和应用

第一章：概述

----##算法大题可以使用一般的解法就不要花太多时间，可以使用 set、map、priority_queue、vector 等数据结构，而考察具体的实现时，要写出具体操作（红黑树不可能，但是堆的实现（adjust）有可能）

----时间复杂度通常指最坏情况下的时间复杂度，比如未优化的并查集高度为 n
----##数组的逻辑不要想得太复杂，“三元组“题的方法是最普遍而简单的从 0 到 n-1，要会设置一些结果变量如 Dmin

第二章：线性结构

----将长度分别为 m、n 的两个有序表归并成一个有序表最少比较次数为 min(m,n)，最多比较次数为 m+n-1.
----在双端队列的序列模拟中要严格模拟，不要记忆结论，栈也是;
----在数组中找到所有两两之和等于 x 的数对方法：快速排序、从两端开始找（若 a[i]+a[j]<x,则 i++，若 a[i]+a[j]>x，则 j--，否则输出 a[i] a[j] i++;j--;直到 i>j）
----栈和队列拥有相同的逻辑结构、不同的是运算
----判断链表是否中心对称可以使用栈
----链式存储的队列如果不带头结点，则 front 和 rear 分别指向队首和队尾节点，如果带头结点，则 front 指向头节点，rear 指向队尾节点；在删除时队头和队尾可能都需要修改（最后一个的时候要把 rear=front）
----队列的应用：页面替换算法、进程调度、缓冲区
栈的应用：括号匹配、迷宫求解、递归、进制转换

----双栈：可以在 O(1)时间内实现 push、pop、min 操作，主栈是普通栈，用来实现 push 和 pop，辅助栈和主栈是同步的，每次 push 都将 stack_min[top++]=(x<min)?x:min; 每次 pop 时辅助栈也会 pop，这样辅助栈栈顶永远是当前的最小值。
----KMP 算法中要求的最关键的 init 即从-1、0 开始的 next 数组，根据它产生相应的 next（都+1）和 nextval（从后往前，在 init 上将 init[i]与自身相等的再往前迭代一次）数组。
----循环链表的逆置也使用头插法
----同一组不重复输入序列执行不同的入栈出栈操作，得到的结果也可能相同，因为初态终态并没有限制
----##循环队列“在队列末尾使用牺牲一个单元的方法判断队空队满“，队空条件为 rear==front，队满为（rear+1）%n==front，rear 指向的是队尾元素的下一个存储单位

----##如果线性表相关算法题的具体操作代码太长，则可以写一部分伪代码，节省答题空间
----矩阵每行有序、每列有序的查找算法:

```
int i = 0, j = n-1;
while (1) {
    if (a[i][j] == x)break;
    if (a[i][j] > x)j--;
    else i++;
}
```

----链表逆序：//高频考点

```
Void reverese(listNode* L){//l 指向头节点
    listNode* p=l->next;
    l->next=NULL;
    while(p){
        listNode* temp=p;
        p=p->next;
```

```
            temp->next=l->next;
            l->next=temp;
        }
    }
```

----有序链表求交集：

```
    listNode* ins(listNode* A,listNode* B, listNode* C){
        listNode* a=A->next,*b=B->next,*c=C;
        while(a&&b){
            if(a->data==b->data){
                listNode*p=new listNode(a->data);
                c->next=p;
                c=c->next;
            }
            Else if(a->data>b->data)b=b->next;
            Else a=a->next;
        }
    }
```

----链表中也可以使用 for：for (listNode*p=L;p!=NULL;p=p->next)//要使用这样比较规范的代码

```
    Dnode* locate(Dnode* l, int x) {
        //用于实现链表根据访问频率使元素向前移动
        for (Dnode* p = l->next;p != NULL;p = p->next) {
            if (p->data == x) {
                p->fre++;//频度加 1
                Dnode* xpre = p->pre, * xnext = p->next;//局部变量，函数结束即失效，不会占用内存
                xpre->next = xnext;
                xnext->pre = xpre;//完成摘下

                Dnode* pos = l;
                for (;pos->fre > p->fre;pos = pos->next);
                    //找到第一个 fre 小于等于 p 的节点，准备将插到前面

                Dnode* ppre = pos->pre;
                p->next = pos;
                pos->pre = p;
                ppre->next = p;
                p->pre = ppre;//双向链表的头插法
                return p;
            }
        }
        return NULL;
    }
```

----判断链表是否有环：

```
listNode* loop(listNode* l){
    listNode*p=l;
    unordered_set<listNode*>se;
    while(1){
        if(se.count(p))//如果 se 里已经有 p
            return p;
        else{
            se.push(p);
            p=p->next;
        }
    }
}
```

----TopK: （这里的比较符号应该是<）

```
vector<int> getLeastNumbers(vector<int>& arr, int k) {
    vector<int>ans(k);
    if (k == 0)return ans;
    int n = arr.size();
    //priority_queue<int>qu;//自动将所有元素从大到小排序,qu.top()是最大的
    //priority_queue<int, vector<int>, less<int> >qu;//大顶堆
    priority_queue<int, vector<int>, greater<int> >qu;//小顶堆
    for (int i = 0;i < k;i++) qu.push(arr[i]);
    /* ... */
    for (int i = k;i < n;i++) {
        if (arr[i] > qu.top()) {
            qu.pop();
            qu.push(arr[i]);
        }
    }
    for (int i = 0;i < k;i++) {
        ans[i]=qu.top();
        qu.pop();
    }
    return ans;
}
```

----单调栈在 nums2 中寻找 nums1 中每个元素的下一个最大数（应该不太可能考）

```
vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
    int n1=nums1.size(),n2=nums2.size();
    int i=n2-1;
    stack<int>st;
    unordered_map<int,int>hashmap;
    for (;i>=0;i--){
        while(!st.empty()&&nums2[i]>st.top()) st.pop();
        hashmap[nums2[i]]=(st.empty())?(-1):(st.top());
        st.push(nums2[i]);//找到nums2中每个数的下一个最大数
    }
    i++;
    vector<int>ans;
    for (;i<n1;i++){
        ans.push_back(hashmap[nums1[i]]);//建立第一个数组与第二个数组的映射
    }
    return ans;
}
```

第三章：树形结构

----把二叉树所有节点（包括 NULL 和空位）都保存（规范顺序存储，正逆向都要注意空位要补全）
----按层次从上到下、同一层从左到右的顺序存储在数组中，不保存 NULL 节点（不规范的顺序存储）
----顺序存储的第 i 个节点的祖先是 ceiling(m/2)-1（编号从 0 开始，如果从 1 开始就另外推算推算）,依此可以寻找其各代祖先
----二叉树是逻辑结构，但线索二叉树是加上线索之后的二叉链表结构，属于物理结构
----**非空二叉树**（空二叉树不成立！）的叶子节点数目等于度为二的节点个数加一（n0=n2+1）----卡特兰数：C(n,2n)/(n+1);

----##路径长度计算**需要计算所有的叶子节点**
树的路径长度是根到**每个节点**的路径长度之和
哈夫曼树带权路径长度是所有**叶节点**的带权路径之和
树的度是所有节点度的最大值
----给定权值总数为 n 的哈夫曼树节点总数为 2n-1
----哈夫曼树中如果有负权值则叶子节点到根节点的路径序列不一定有序
----哈夫曼编码要小心 0 和 1 的走向、只有叶节点可以编码

----判断树是否镜像：
Ismirror（a,b） 判断两棵树是否镜像（递归比较 ismirror(a->left,b->right)&& ismirror(a->right,b->left)），再用 mirror（root）
----树的算法题**一定要会非递归的后序遍历**，找叶子节点和计算深度之用
----一般情况下，先思考使用递归的先序、后序遍历算法；
----在孩子兄弟表示法存储的树中，主要的“左右孩子”是“孩子节点”和“兄弟节点”，故在递归的时候可以把这两个当作二叉树的左右孩子来写代码
----将二叉链表中的二叉树放进数组里：

```
Void trans(TreeNode* root, int b[], int k){
    If(root){
        b[k]=root->data;
        trans(root->left,b,2*k+1);
        trans(root->right,b,2*k+2);
    }
}
```

----使用 buildTreeNode_from_to(int pre[], int l_pre, int r_pre, int in[], int l_in, int r_in)函数对“由中序、先序遍历序列构造二叉树”等问题求解
----判断一个数是否是完全二叉树：将所有节点入队（包括空节点），遍历时如果碰到空节点，那么这时队列需要为空，否则不是完全二叉树
----后序遍历的递归栈存了正在访问的节点的所有祖先，如果要逐个访问其祖先，则可以使用数组栈，并用下标访问；
----后序线索树的后序遍历仍然需要栈的支持

----树的算法题的时间复杂度一般都是 O(n),因为即使是递归遍历也是每个节点遍历一次

----二叉树的遍历和应用

A）先序遍历

递归版代码	非递归版代码

Void preOrder(TreeNode* root){
If (!root){
Visit (root);
preOrder(root->left);
preOrder(root->right);
}
}
	版本一：（与中序遍历保持一致的版本）
	Void preOrder(TreeNode* root){
	If (!root) return;
	TreeNode* p=root;
	Stack<TreeNode*> st;
	While(!st.empty() || p){
	If(p){
	Visit(p);
	St.push(p);
	P=p->left;
	}
	Else{
	P=st.top();
	St.pop();
	P=p->right;
	}//else
	}//while
	}
	版本二：（先序独有的版本）
	Void preOrder(TreeNode* root){
	If(!root) return;
	TreeNode * p=root;
	Stack<TreeNode*> st;
	St.push(p);
	While(!st.empty()){
	P=st.top();
	St.pop();
	Visit(p);
	If(p->right) st.push(p->right);
	If(p->left) st.push(p->left);
	}
	}

应用：生成先序序列	

B）中序遍历

递归版代码	非递归版代码
Void inOrder(TreeNode* root){ If (!root){	Void inOrder(TreeNode* root){ If (!root) return;

inOrder(root->left);
Visit (root);
inOrder(root->right);
}
}
	TreeNode* p=root;
	Stack<TreeNode*> st;
	While(!st.empty() || p){
	If(p){
	St.push(p);
	P=p->left;
	}
	Else{
	P=st.top();
	St.pop();
	Visit(p);
	P=p->right;
	}//else
	}//while
	}

应用:	

C）后序遍历

递归版代码	非递归版代码

Void postOrder(TreeNode* root){
If (!root){
postOrder(root->left);
postOrder(root->right);
Visit (root);
}
}
	Void inOrder(TreeNode* root){
	If (!root) return;
	TreeNode* p=root , * r=NULL;
	Stack<TreeNode*> st;
	While(!st.empty() || p){
	If(p){
	St.push(p);
	P=p->left;
	}
	Else{
	P=st.top();
	If (p->right && p->right!=r){
	P=p->right;
	}//如果右边有 且没有被访问过 则转向右边
	Else {
	St.pop();
	Visit(p);
	R=p;
	P=NULL;
	}//否则访问 p，四步缺一不可
	}//else
	}//while
	}

应用：求二叉树深度：遍历过程中最大的栈深度即为二叉树深度 求两个节点层数最高的共同祖先，使用双栈记录二者各自的祖先。

D） 层序遍历

简单版代码	带层数版代码

void levelOrder(TreeNode* root) {
if (!root) return;
queue<TreeNode*> qu;
TreeNode* p = root;
qu.push(p);
while (!qu.empty()) {
p=qu.front();
if (p->left)qu.push(p->left);
if (p->right)qu.push(p->right);
visit(p);
qu.pop();
}
}
	void levelOrderLev(TreeNode* root) {
	if (!root) return;
	queue<pair<TreeNode*,int>> qu;
	TreeNode* p = root;
	qu.push(make_pair(p,1));
	while (!qu.empty()) {
	p = qu.front().first;
	int lev = qu.front().second;
	qu.pop();
	if (lev != qu.front().second || qu.empty()) {
	//表明已经弹出当前层的最后一个节点
	visit1(p, lev);
	}
	Else visit2(p,lev);
	if (p->left)qu.push(make_pair(p->left,lev+1));
	if (p->right)qu.push(make_pair(p->right,lev+1));
	}
	}

应用：	

第四章：图

----##默认认为“有向图”是联通的，有向图邻接矩阵中，如果主对角线一下元素全为 0，则其拓扑序列一定存在但不唯一（要会举例），如果主对角线以上元素全为 1，主对角线以下元素全为 0，则拓扑序列唯一;如果一个有向图存在有序拓扑排序序列，则其邻接矩阵一定是三角矩阵

----有向图这种数据结构默认没有插入和删除操作

----有向图边<v,w>中，v 表示弧尾，w 表示弧头，箭头向右打

----##“一个数字的含义是什么？”可以为“路径的条数”而不是“存在路径”

----##图的应用题要把题目问的元素（比如每次选择的边）写清楚

----使用邻接表存储无向完全图需要 n(n-1)个边界点，而使用邻接多重表需要 n(n-1)/2 个;

----强连通是针对有向图说的，连通是针对无向图说的，强连通分量是有向图的极大连通子图

----AOV 网只讨论拓扑排序，AOE 网只讨论关键路径
----计算关键路径时需要仔细十以内加减法的正确性
----并不是任意顶点和任意边的组合都是子图，因为这样的组合可能不是图
----连通分量就是极大连通子图，强连通分量就是极大强连通（任意两个顶点之间都有互相到达的路径）子图
----极小连通子图是连通无向图的生成树。极小和极大是在满足连通的情况下，针对边的数目而言的。
----有向图顶点的度是入度和出度的和
----回路对应于路径，简单回路对应于简单路径

----十字链表（有向图，弧用 arc）：
顶点节点：data、firstin、firstout
弧节点：head_link, head, info, tail, tail_link（与下面的其实是类似的）
----邻接多重表（无向图，边用 edge）：
顶点节点：data、firstedge
边节点：v1_link, v1,info, v2, v2_link
----判断一个无向图是不是一棵树：一次深度遍历，如果访问的顶点数是 n 且边数为 2(n-1)，则这是一颗树：

```
void DFS(graph g, int v, int& Vnum, int& Enum, vector<bool>visited) {
    Vnum++;
    visited[v] = 1;
    int w = firstNeighbor(g, v);
    while (w != -1) {
        Enum++;//找到一个边，那么Enum++
        if(!visited[w])DFS(g, w, Vnum, Enum, visited);//递归遍历
        w = nextneighbor(g, v);//递归遍历中for循环和while循环的写法
    }
}
bool istree(graph g) {
    int n = g.vecnum;
    vector<bool>visited(n, 0);
    int Vnum = 0, Enum = 0;
    DFS(g, 1, Vnum, Enum,visited);//Vnum和Enum接收遍历的顶点和边的个数
    //用函数参数的方式返回，相当于全局变量
    if (Vnum = n && Enum == 2 * (n - 1)) return 1;
    else return 0;
}
```

----在 DFS 中使用参数返回如“can_reach”、“length”等
----使用 DFS：判断一个有向图是否存在这样的顶点，该顶点到其他所有顶点都有路径；
的方法：从每个顶点出发都使用一次 DFS，若能遍历所有顶点，则该点符合条件，时间复杂度 O(n^2)
关键代码：

```
For(int i=0;i<G.n;i++){
    Int N=0;
    DFS(G,I,visited,N);//注意参数设置
    If(N==G.n)return true;
```

}

----DFS 压栈的顺序是拓扑序列，那么退栈的顺序就是逆拓扑序列，在 DFS 的末尾（如果放在开头，则结论相反，这是由递归的特性决定的）加上“time++;finishtime[v]=time”则可以计算所有顶点递归遍历时结束的时间，结束时间最晚的就是拓扑排序的开始节点，而最早结束的是拓扑序列中的最后节点

----Prim 算法的时间复杂度：使用邻接矩阵 O(V^2)，使用邻接表为 O(E*logV);

Kruskal 算法的时间复杂度是 O(ElogV),一般不会使用邻接矩阵计算 Kruskal，因为寻找边集不方便

----邻接表表示的拓扑排序的时间复杂度是 O(V+E)（要想象消边的过程），邻接矩阵表示的是 O(V^2)

----在邻接矩阵的复杂度判断中不考虑 E

----Prim 算法每次选择离 MST 中**任意节点**最近的下一条边，而 Dijkstra 算法每次选择**离源点**最近的边；进而 Dijkstra 算法可以产生生成树，但不一定是最小生成树，至少起点所连的边（如果值相同）必须全部加入

----Floyd 算法求每对不同顶点之间最短路径时，允许弧上的权值为负，但不能有包含负权值边的回路

----Kruskal 算法要用到并查集（在选边之前先判断两个顶点是否属于同一个集合，如果是则不加入，如果不是，则加入并且把集合合并），无向图的连通性判断要用到并查集（遍历图的每个边，每次把边两端的顶点所在的集合合并，直到遍历完所有的边，如果只剩下一个集合则说明无向图是连通的，否则不连通）

----图的遍历算法

A）广度优先遍历

```
void BFS(Gragh G, int v) {
        queue<int> qu;
        qu.push(v);
        while (!qu.empty()) {
            int p = qu.front();
            qu.pop();
            visit(p) ; visited[p] = 1;
            for (int x = firstNeighber(G, p) ; x < n ; x = nextNeighber(G, p, x)) {
                if (!visited[x]) qu.push(x);
            }//把所有的邻居都加进来
        }
}
void bTraverse(Gragh G) {
        int n = G.vexnum();
        static vector<int> visited;
        for (int i = 0;i < n;i++) visited.push(0);
        for (int v = 0;v < n;v++)
            //对于每个连通分量
            if (!visited[v])BFS(G, v);
}
```

应用：广度遍历求无权值单源最短路径

```
void BFS(Gragh G, int v) {
```

```
        int n=G.vexnum();
        vector<int>d;
        for (int i=0;i<n;i++) d.push(INT_MAX);
        d[v]=0;
            queue<int> qu;
            qu.push(v);
            while (!qu.empty()) {
                int p = qu.front();
                qu.pop();
                for (int x = firstNeighber(G, p) ; x < n ; x = nextNeighber(G, p, x)) {
                    if (!visited[x]){
                        qu.push(x);visited[x]=1;
                        d[x]=d[p]+1;//到 x 的距离是当前点距离加 1，这就是最短的路
径，因为单权值单源最短路径问题中，广度遍历时同一层的结果相同，都是最短的
                    }
                }//把所有的邻居都加进来
            }
}
```

B）深度优先遍历

```
void DFS(Gragh G, int v) {//递归版
    visit(v);
    visited[v]=1;
    for (int x = firstNeighber(G, p) ; x >=0 ; x = nextNeighber(G, p, x)) {
            if( !visited[x] ) DFS(G,x);
        }//对所有的邻居都深度遍历一次
}
Void DFS(Gragh G,int v){//非递归版
     //思路是使用栈记录接下来可能要访问的顶点
     //同时用数组标记那些已经入栈过，实际上入栈也就代表访问过
    Stack<int> st;
    St.push(v);
    Visited[v]=1;
    While( !st.empty() ){
        Int p=st.top(); st.pop();
        Visit(p);
        for (int x = firstNeighber(G, p) ; x >=0 ; x = nextNeighber(G, p, x)) {
            if( !visited[x] ){
                st.push(x); visited[x]=1;
            }
        }
    }
}
void bTraverse(Gragh G) {
```

```
        int n = G.vexnum();
        static vector<int> visited;
        for (int i = 0;i < n;i++) visited.push(0);
        for (int v = 0;v < n;v++)
            //对于每个连通分量
            if (!visited[v])DFS(G, v);
}
```

第五章：集合

----没有太多考察内容，要理解并查集的应用场景：Kruskal、无向图连通性的判断（逐个加边法）

第六章：查找

----##查找失败的长度需要计算最后寻找空位的那一次；
----若 n 表示顺序表长度，则顺序查找成功的 ASL=a1=1/2(n+1)；查找失败的 ASL=a2=n+1；如果两者概率相等，则总 ASL=(a1+a2)/2，注意区分概率和长度，举例时注意严格根据例子来算与 n 的关系
----在二叉排序树中计算查找长度的关键是画出二叉排序树
----二叉排序树是增加了排序性质的二叉树，因此算法题仍然是基于二叉树的递归特点，只是需要多考虑排序性质而已。
----注意二叉排序树的概念：根节点要小于右子树中所有节点，而不仅是要小于右子树的根（容易混淆）

----nh 表示高度为 h 的平衡二叉树含有的最少节点数，则 n1=1；n2=2；nh=(nh-1)+(nh-2)+1
----##在平衡树中删除一个顶点再加入同一个顶点，得到的树可能相同，参考“1、2、3“，在开始重要善于直接写出实例
----注意平衡子树指的是平衡因子（h 左-h 右）小于等于 1 的子树（只差 1 仍是平衡的）

----红黑树的性质：
    每个节点或红或黑
    根节点是黑色的
    叶节点（虚构的外部节点、NULL 节点）都是黑色的，这说明终端节点（最底层的非叶节点）可红可黑
    不存在相邻的红节点（任意红节点的父节点和孩子节点都是黑色的）
    对于每个节点，从该节点到任意叶节点的简单路径上所含的黑结点数目相同
----红黑树的推论：
    从根节点到叶节点最长路径不大于最短路径的 2 倍
    红黑树将 AVL 树的“高度平衡”，降低到“任意节点左右子树的高度相差不超过两倍”
----红黑树和 AVL 数的插入、查找、删除的平均复杂度都是 O(log n)，但红黑树的统计

性能更好

----B 树的插入总体是在最下层堆积，堆不下了才往上溢；而删除（只会删除终端节点，要删除非终端节点时先用相邻的终端节点替代他然后在删除该终端节点）是在最下层沉淀，只要能沉下来就会把上面的往下沉，熟练掌握。
----B 树的所有终端节点的高度是一致的。
----具有 n 个关键字的 B 树应该有 n+1 个叶节点（即查找失败的外部节点）

----##B+树比 B 树更加适用于实际应用中的文件索引和数据库索引
----B+树每个索引项只包含对应子树的**最大关键字**和指向该子树的指针
----##二叉查找树的画法要注意使用统一的方式：一般是向下取整
----注意提到二分查找树的时候，一般认为关键字是有序的，而是否随机分布决定了平均查找长度能否达到 logn

----散列表楚除留余数法选用的 p 一般是小于等于 m 的最大质数（留下一部分用来处理冲突）
----注意双散列法的第二个散列函数计算的是地址增量，Hi=(H(key)+i*Hash2(key))%m, i 表示冲突次数，
----散列表查找成功的平均查找长度与装填因子直接相关，与表长无关
----Hash 聚集：非同义词争夺一个地址的现象；一般是线性探测产生的
----Hash 冲突：同义词争夺一个地址的现象；都会产生
----二分查找

```
int bsearch(vector<int>a, int low, int high, int x){
    Int mid=low;
    While(low<=high){
        Mid=low+(high-low)>>1;
        If(a[mid]==x)break;
        Else if(a[mid]>x) High=mid-1;
        Else Low=mid+1;
    }
    Retrun low;//返回的是插入位置
}
```

第七章：排序

----##快速排序每次划分后**先处理短分区可以减小递归深度但不能减少递归次数**；**如果分区均衡，可以减少递归次数和运行时间**。
----Partirion 函数如果没有递归 z 就没有递归深度，可以使用栈或者队列来实现快速排序的非递归算法，只要用栈或队列记录每次划分的两个端点，在下一次处理时取出即可，他们产生的顺序会不同
----要牢记的是 quicksort 和 heapsort!!

----荷兰国旗问题：即三数排序

```cpp
//荷兰国旗问题（按照123的顺序）
void HelanSort(vector<int>& a) {
    int n = a.size();
    int left = 0, right = a.size() - 1;
    int i = 0;
    while (i<=right) {
        if (a[i] == 1) {
            swap(a[left], a[i]);
            left++;i++;
        }//1与left交换
        else if (a[i] == 2) {
            i++;
        }//2不变
        else {
            swap(a[right], a[i]);
            right--;
        }//3与right交换
        //没有i++是为了防止交换以后a[i]还是3
        //这个细节需要手动模拟验证，实际上是因为i从左边开始
    }
}
```

----构建 n 个记录的初始堆，其时间复杂度为 O(n)（比较不直观，需要死记）

----堆排序属于选择排序，其思想为“每次从堆中选择最大的”，而堆的作用就是 logn 时间内找到最大的元素

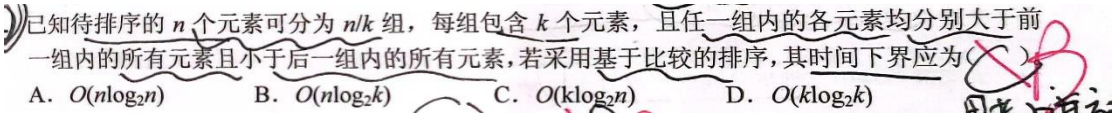
----直接选择排序是不稳定的，基数排序（因为分配和回收处理不会影响同义词）和归并排序是稳定的

----归并排序需要会把两个数组并排放，依次比较，计算比较次数

```cpp
//合并有序数组
vector<int> mergeSorted(vector<int> a, vector<int> b) {
    vector<int> c;
    int n1 = a.size(), n2 = b.size();
    int i = 0, j = 0, k = 0;
    while (i < n1 || j < n2) {
        int s1 = (i==n1) ? (INT_MAX) : (a[i]);
        int s2 = (j==n2) ? (INT_MAX) : (b[j]);
        if (s1 < s2) {
            c.push_back(s1);i++;
        }
        else {
            c.push_back(s2);j++;
        }
    }
    return c;
}
```

----m 路归并需要 m 个输入缓冲区和一个输出缓冲区

----关于分块排序：要弄清楚块与块之间的关系是怎样的，“仅块内有序”可以使用归并方式思考，二题中给出的是“块内无序，块间有序”，只需要块内排序即可。



----要掌握的排序算法

A）简单插入排序

```
void insertSort(vector<int>& a) {
        int n = a.size();
        for (int i = 1;i < n;i++) {
            if (a[i] < a[i - 1]) {
                int temp = a[i];
                int j = i - 1;
                for (;a[j] > temp;j--) {
                    a[j + 1] = a[j];
                }
                a[j + 1] = temp;
            }
        }//for
}
```

是否稳定	稳定
最好时间复杂度	O(n)
最坏时间复杂度	O(n2)
辅助空间	无
能否每一躺确定一个位置	不能
应用	在元素基本有序或者数组不长的时候很有效

B）希尔排序

```
void shellSort(vector<int>& a) {
    int n = a.size();
    for (int dis = n / 2;dis >= 1;dis /= 2) {
        for (int i = dis;i < n;i++) {
            if (a[i] < a[i - dis]) {
                int temp = a[i];
                int j = i - dis;
                for (;a[j] > temp && j > 0;j -
= dis) {
                    a[j + dis] = a[j];
                }
                a[j + dis] = temp;
            }
        }//for
    }
}
```

是否稳定	不稳定
时间复杂度	O(n1.2-n1.8)
辅助空间	无
能否每一躺确定一个位置	不能
应用	在元素基本有序或者数组不长的时候很有效

C）折半插入排序

```
void halfInsertSort(vector<int>& a) {
    int n = a.size();
    for (int i = 1;i < n;i++) {
        if (a[i] < a[i - 1]) {
            int temp = a[i];
            int low = 0, high = i;
            int mid = 0;
            while (low <= high) {
```

是否稳定	稳定
最好时间复杂度	O(n2)
最坏时间复杂度	O(n2)
辅助空间	无
能否每一躺确定一个位置	不能

mid = (low + high) >> 1;
if (a[mid] > temp)
high = mid - 1;
else low = mid + 1;
}
int j = i - 1;
for (;j > high;j--)a[j + 1] = a[j];
a[j + 1] = temp;
}
}
}
	应用	直接插入排序的变体，只减少了比较次数，没有减少移动次数，折半查找必须记住

D） 简单选择排序

void selectSort(vector<int>& a) {
int n = a.size();
for (int i = 0;i < n;i++) {
int index = i;
for (int j = i;j < n;j++) {
if (a[j] < a[index])
index = j;
}
swap(a[i], a[index]);
//在代码中能使用 swap 就不要使用 temp；
}
}
	是否稳定	不稳定：在选完交换的时候可能调换同义词的位置

	最好时间复杂度	O(n2)
	最坏时间复杂度	O(n2)
	辅助空间	无
	能否每一躺确定一个位置	能，确定最小的那个排到最前面
	应用	代码非常简单，在数据不多时仍然应用广泛，始终需要 O(n2)的复杂度

E） 冒泡排序

void bubbleSort(vector<int>& a) {
int n = a.size();
for (int i = 0;i < n - 1;i++)
for (int j = i + 1;j < n;j++)
if (a[i] > a[j])
swap(a[i], a[j]);
}
	是否稳定	稳定

	最好时间复杂度	O(n2)
	最坏时间复杂度	O(n2)
	辅助空间	无
	能否每一躺确定一个位置	不能
	应用	代码非常简单，在数据不多时仍然应用广泛，始终需要 O(n2)的复杂度

F） 快速排序

```
int partition(vector<int>&a, int low, int high) {
    int index = rand() % (high - low + 1) + low;
    swap(a[low], a[index]);
    int pivot = a[low];
    while (low < high) {
//选择 low 作 pivot 则从 high 开始变
        while (low < high && a[high] >= pivot)
            high--;
        a[low] = a[high];
        while (low < high && a[low] <= pivot)
            low++;
        a[high] = a[low];
    }
    a[low] = pivot;
    return low;
}
void quickSort(vector<int>&a, int begin, int end)
{
    if (begin < end) {
        int pos = partition(a, begin, end);
        quickSort(a, begin, pos - 1);
        quickSort(a, pos + 1, end);
    }
}
```

是否稳定	不稳定
最好时间复杂度	O(nlogn)
最坏时间复杂度	O(nlogn) 假如没有随机选 pivot 就是 O(n2)
辅助空间	无
能否每一躺确定一个位置	能，每次都确定了 povit 的最终位置
应用	性能最佳的排序算法，其分治思想在真题中考查过

G） 归并排序

```
vector<int>b(1010);
void merge(vector<int>& a, int low, int mid, int high) {
    //将 a 中 low..mid 和 mid..high 有序的两段合并,并放回 a
    for (int i = low;i <= high;i++)b[i] = a[i];
    int k = 0, i = 0, j = 0;
    for (i = low, j = mid + 1, k = i;
        i <= mid && j <= high;k++) {
        a[k] = (b[i]<=b[j]) ? (b[i++]) : (b[j++]);
    }//这里必须是<=才能保证稳定
    while (i <= mid)a[k++] = b[i++];
    while (j <= high)a[k++] = b[j++];
}
void mergeSort(vector<int>&a, int low, int high) {
    if (low < high) {
```

是否稳定	稳定
最好时间复杂度	O(nlogn)
最坏时间复杂度	O(nlogn)
辅助空间	无
能否每一躺确定一个位置	不能
应用	归并的思想很常用

```
        int mid = (low + high) >> 1;
        mergeSort(a, low, mid);
        mergeSort(a, mid + 1, high);
        merge(a, low, mid, high);
    }
}
```

H） 堆排序//这是具体实现，而要用堆解决问题的时候要使用 priority_queue

```
void heapAdjust(vector<int>& a, int k, int n) {
    int temp = a[k];
    for (int i = 2 * k + 1;i < n;i = i * 2 + 1) {
        //注意从哪里开始编号
        if (i + 1 < n && a[i] < a[i + 1]) i++;
        if (temp >= a[i]) break;
        a[k] = a[i];
        k = i;//更新最终位置
    }
    a[k] = temp;//放到最终位置上
}
void heapSort(vector<int>& a) {
    int n = a.size();
    for (int i = (n - 1) / 2;i >= 0;i--)
        heapAdjust(a, i, n);//建立初始大根堆,处理结果是升序。
    for (int i = n - 1;i > 0;i--) {
        swap(a[i], a[0]);//把最大值 a[0]交换到后面去
        heapAdjust(a, 0, i);
    }
}
```

是否稳定	不稳定
最好时间复杂度	O(nlogn)
最坏时间复杂度	O(nlogn)
辅助空间	无
能否每一躺确定一个位置	能，每次都选出最大的放在最后面
应用	堆这种数据结构善于从很多数中选出最小（大）的哪些

I） 基数排序

不会靠代码，但是要知道排序原理，注意**先以低位**为关键字。

J） 置换选择排序

不会考代码，但是需要知道这是外部排序在生成初始归并段时使用的算法，要回手动计算

二、计组:

历年真题考察内容及特点		
年份	43 题	44 题
2009	CPU 性能指标、DMA 方式	硬件控制功能和控制信息
2010	指令字段格式、寻址方式	Cache 直接映射、数组的局部性原理
2011	寄存器、数据的标识和运算	虚拟地址、物理地址、页表和 Cache
2012	CPU、DMA、传输速率	指令系统、指令流水线、数据运算
2013	CPU、Cache、传输速率	指令格式，标志位、硬件名称
2014	指令系统、汇编指令	寄存器、Cache 和主存
2015	ALU 硬件	指令和控制信号
2016	CPU 和 IO 方式	页式虚拟存储器
2017	浮点数、整数运算	指令系统、机器代码
2018	IO 方式	页式虚拟存储管理
2019	机器代码、指令寻址	分页存储、Cache 映射
2020	乘法指令、硬件、标志位	Cache 映射、缺失

2021	指令系统、寄存器和数据运算	组相联 TLB
2022	CPU 硬件控制	磁盘结构和 DMA
2023 押题	机器指令综合	Cache 与主存

第一章：概述

----冯诺依曼机：采用存储程序原理，基本工作方式是控制流驱动方式
----硬件和软件在逻辑上是等效的，硬件有更高的执行速度，软件实现有更好的灵活性
----寄存器的存在一般都是为了解决速度差异
----数据库系统（比较大的概念）是指在计算机系统中引入数据库之后的系统，包括数据库、数据库管理系统、应用系统、数据库管理员组成，其中数据库管理系统是系统程序
----如今的计算机 CPU 主频、内存容量等都增幅有限，超级计算机主要采用多处理器提高并行处理能力来增强计算机性能
----给出了 CPU 频率和 MIPS 两个值就可以直接计算 CPI，其他的信息要辨别是否多余
----##“颜色深度为 24 位”意思是每个颜色需要 24 个 bit，依此，可类比其他关于“需要多少位”的说法
----注意说法：局部性原理意味着对主存的访问是不均匀的（集中于某一部分）

----##“CPU 速度提升 50%”并不是 CPU 运算时间减少一半
----##带宽单位为 bps 或 Bps（可以把两个单位的数值都写出来作为答案）
----计算机系统层次（自下向上）：微程序机器层、传统机器层（实际的机器层）、操作系统层（软硬件的交界面）、汇编语言层、高级语言层
----汇编语言是计算机硬件语言，在应用中很少见
----计算机体系结构：机器语言或汇编语言程序员看到的东西，包括指令集、数据类型、存储器寻址技术等抽象属性
　　计算机组成：硬件工程师看到的东西，包括对程序员来说透明的硬件细节，例如，一台计算机是否具有乘法指令是结构问题，而实现乘法指令采用什么方式则是一个组成问题
----一定要注意什么是“透明”：透明=看不到、管不了、不属于自己的范围
　　不透明=看得到、能使用、属于自己的范围

第二章：数据表示与运算

----##任何时候都要注意题中的数字有没有 H，没有的话就是十进制!!
----补码 1000 表示-8
----从 x 的补码求-x 的补码的方法：全部取反末位加一（等价于从后往前第一个“1”的左边全部取反，其余不变）
----##unsigned 类型的减法操作和 int 减法操作是一样的，都是“减数全部取反、末尾加一”再“补码相加”；
----##八位补码表示的范围是-128~127

----##深刻理解 ALU 和 MUX 的配合工作
----加法器中运算延迟主要来自于进位传递延迟
----进位生成：g=x*y（只有 x 和 y 都是 1，它们相加才会进位）
----加法器可以提高运算速度的是并行进位加法器，但这会带来硬件的成本
----##在 ALU 加减法中，无符号数和有符号数的四标志生成方式是一样的（溢出到底看哪个要根据指令内容和微操作来判断）：

OF（溢出标志位）=最高位进位 ^ 次高位进位（即符号位进位和数值最高位进位不同时为 1）

CF（进位/借位标志位）=**最高位进位 ^ sub**（因为 CF 关系到加法和减法的不同，所以要考虑 sub 位）

SF（符号标志位）=结果的最高位 ##!!

ZF（零标志位）：结果为零则 1，非零则 0

----双符号位溢出判断：

00 结果为正，无溢出；11 结果为负，无溢出

01 正溢出；10 负溢出

----##乘法电路中（其阵列乘法器可以实现在一个周期内完成乘法指令）：

无符号数：用 2n 位保存结果是，不会发生溢出；截取最后 n 位作为最终结果时，如果结果超过 n 位可以表示的最大结果，则溢出；机器判断方法是看最高 n 位是否全为 0 或全为 1；

有符号数：用 2n 位保存中间结果，仅当前 n+1 位全为 1 或全为 0 时不溢出（参考符号扩展）

----数值位 n 位时，补码一位乘法中最多需要 n 次移位、n+1 次加法（产生 n+1 位结果）；原码乘法中移位和加法都最多 n 次

----原码一位乘法中，符号位不参与运算，单独处理方式：异或；两个 n 位数进行原码一位乘，则部分积至少需要 n+1 位

----补码不恢复余数除法：异号相除时，够减商 0，不够减商 1

----##要分清题中所给的一条指令的 CPI 和一个处理需要多少时钟周期，不能把数字弄混

----##double 占 8B

----double + float 可能会导致 float 对阶时尾数溢出导致丢失精度

----float 左规时尾数增大，阶码减小，可能导致阶码下溢；而尾数舍入的情况只会发生在右规格化或者对阶的时候

----注意浮点数舍入的几种方式：截断法是最简单的

----浮点数是为了表示小数，所以出现“尾数用补码表示”时，指的是小数的补码，而无论多少位小数补码最小值永远是-1

----浮点数乘法是考纲之外的东西，但是要注意原码介于 1.0~2 之间，结果介于 1.0~4 之间，故乘法可能需要右规而不需要左规。

----补码表示的尾数最高位与尾数的符号位相异时表示该数是规格化的（注意理解补码的表示方法和尾数规格化的含义（小于 1 的小数，原码第一位是 1））（**IEEE754 的规格化包括 1.0**）

----小数补码的运算方法坚持使用 1.001 这种格式（第一位表示符号，该数的绝对值小于等于 1）

----IEEE754 阶码范围：1~254 即-126~127，尾数使用原码，有一个个位的 1 是省略的
----##注意 IEEE754 的阶码大小，例如真值是 0.0111 则阶码为-2，保持对数字的敏感!!
----float：1+8+23；double：1+11+52；long double: 1+15+64
----浮点数格式相同时，基数越大则范围越大、精度越低

----举例：用小数原码二进制表示的真值，基数改为 4 后符合规格化的条件是：小数点后两位不全为 0。因为这样保证该小数大于 0.25，而以 4 为基数的规格化小数最小是 0.25
----##1 表述负，0 表示正，各种十六进制码要准确，**不能疏忽**
----进制转换要严格遵循基数次幂*位值的原则
----在进行十六进制加法的时候要关注符号拓展，尤其在变址寻址的加法中
----算术右移：进位标志位移至符号位，顺序右移一位，最低位移至进位标志位
----补码除法中，余数的符号与除数的符号相同时，上商 1，反之商 0

第三章：存储系统

----链接方式：静态链接、装入时动态链接、运行时动态链接，连接阶段形成逻辑地址
装入方式：绝对装入（编译时装入）、静态重定位（定位=装入）、动态重定位（在执行阶段而非装入阶段进行（或者说没有明确的装入阶段））（重定位即把逻辑地址变为内存中的物理地址）
----带有“分区”两个字的是分配策略，和上述装入方式完全不同，包括单一连续分配、固定分区分配（每个分区只装一道作业）、动态分区分配（包含首次适应、邻近适应、最佳适应、最坏适应等分配策略）

----固定分区式存储管理会产生内部碎片，段式管理换产生外部碎片
----段页式存储管理的交换单位是页（先分段、再分页）
----分页管理是在硬件和此操作系统层面完成的，对上层（编译器、链接程序、用户）是不可见的
----##访问缺页时，进行缺页处理（访问磁盘把对应的页写入内存）之后把页表项加入 TLB，再访问一次 TLB，根据地址再一次访存（用来取指）；即在 TLB 之后也需要再次访存（每次都至少要访存一次）；引起缺页中断的指令需要执行两次：一次触发调页，一次访问存储盘（真正执行）
----TLB 和 Cache 的缺失都不会导致程序出错，缺页会（缺页中断）。
----页地址计算判别时可能发生：
缺页中断：得出的页号合法（在页长之内），却不在页表内（有效位为 0）；
越界中断：地址计算非法（超过页长）
访问权限错误：执行的操作于页表中的保护位（如读写位、用户/系统属性位等）不一致，比如想要写不允许写的页面。
----##页表项的物理地址=页表始址地址+页号*页表项字节数（按字节编址!!）
----##多级页表找到了中间页表的物理地址之后，要**根据地址组成中其后的字段进行补充**，得到具体内容的地址，一般都是四的倍数。
----外层页表不能表示页面物理位置，只是给出了页表的物理地址（用于放进页表基址

寄存器里）

----区分普通的分页和请求分页

----纯粹的分页存储系统无法解决内存共享和保护的问题，这需要借助于编译器实现

----页类型为零页时表示该页在分配物理块时需要清零块空间，为写回 Swap 文件页时表示在写回时必须分配 swap 空间，并写回到 swap 空间中，未设置页类型时则按照正常方式处理；因此程序处理的数据所在的页类型决定了数据的读取方式，为零页时不读取，直接清零现分配的快空间即可，后两者则需要从文件系统读取。

----调页缺页的处理顺序：缺页中断（最先产生中断）->决定淘汰页->页面调出->页面调入

----如果某进程包含 n 个不同的页号，则至少会产生 n 次缺页。

----适合使用虚拟页式存储的情况：所用虚拟地址符合局部性原理。如堆栈，只有在栈顶跨页时才需要缺页处理；“矢量操作”类似于线性代数中的向量，是存储在数字中的一组数据，符合局部性原理；“间接寻址”存放间接地址的页面、存放直接地址的页面以及存放内容的页面三者之间没有规律，可能会导致三次缺页，故不符合局部性原理

----考察页表查询时，一般会考察关于访问越界的问题，要对“是否从 0 开始”、“是否有效”等出题点足够警惕

----内存越界：某进程访问了不属于自己的空间（如 int* p=new int[50]; free(p); free(p); 中，第二个 free 访问了已经不属于自己的空间）

内存泄漏：已经申请的空间地址丢失（如 int * p =new int[50]; p=new int[20]; 中，第一次分配的 50 个 int 发生了内存泄漏）

----##PDBR 是页表基址寄存器，进程切换时，PDBR 的内容会改变，而同一进程的线程切换时不改变，因为进程与 PDBR 是一一对应的，线程共享进程的地址空间。

----产生内存抖动的原因是页面置换算法不合理，毕竟页面置换算法的目的就是防止抖动

----交换区是在磁盘中开辟的一片区域，与内存交换数据，实现虚拟内存。

----高速缓冲存储器（Cache）是 CPU 内复制了经常访问的内存数据，高速磁盘缓存是内存中复制了经常访问的磁盘数据

----ROM 写入较慢，读取很快，不适合做高速缓存

----ROM 的升级版：EPROM、EEPROM、Flash

----CD-ROM 是顺序存储，属于 ODM（光盘存储），不属于 ROM，ROM（非易失）和 RAM（易失）一样可以随机存储，硬盘使用直接存取方式（先找到一小块，再找到地址）介于顺序存取、随机存取之间。

----四体低位交叉存储器的带宽是单存储器的四倍，存储周期是其四分之一，例如每个存储器的存储周期是 40ns 则四体交叉存储器存储周期为 10ns。

----高位交叉存储器的字是连续存放的，不满足局部性原理，但仍有可能一次连续读出相差一个存储器距离的四个字，只是概率很小

----##256MB 的存储器由若干 4M*8 位的 DRAM 组成，则地址线仅根据 4M*8 位计算，为 22/2=11 根（DRAM 默认使用地址服复用，要除以 2，而未提及 DRAM 时不除以 2），数据线为 8 根.（这是因为片选功能是由译码器实现的，片选线不算做地址线）

----在地址线和数据线之外，还需要 1 根片选线+1 或 2 根读写控制线+供电引线+接地引线

----SRAM：非破坏性读出，不需要刷新。断电即丢失，易失性。存储速度快，但集成度低，功耗较大，常用于 Cache

DRAM：破坏性读出，需要刷新。断电即丢失，易失性。存储速度慢，但集成度高，功耗较低，价格低，常用于主存

----DRAM 刷新：硬件自动刷新，只要有电就行，不需要 CPU，一次刷新占用一次存储周期（集中刷新存在死区，分散刷新无死区但加长了存取周期，异步刷新存在死区但整体比较 ok）

----##Cache 包括：标记位、脏位（一致性维护位）、替换信息位、有效位、内存数据副本（也称数据区，题中可能会要求"数据区"大小）、Cache 行号（隐含的）；计算总容量时都需要考虑!

----注意 Cache 数据区，如果一行 256b=32B，即主存地址的低 5 位（注意理解依字节编址）是块内偏移，而不是第 8 位

----如果在 Cache 总容量不变的情况下，把 Cache 块的大小增加一倍，则子块内地址位增加 1b，Cache 行数减少一倍，导致 tag 位增加一位（存疑，块内地址位增加 1，行数位减少 1，则 tag 位不变）

----Cache 内的数据全都是内存的副本，不会增加内存的容量

----计算 Cache 或 TLB 命中率时要分清命令执行的周期一共有多少次访问 Cache，比如 a[i]=a[i]+1;语句就需要两次 Cache 访问，访问指令 Cache 的命中率计算时分母是总指令数（=循环次数*每次循环的指令数）

----##Cache 组相联映射**"路数"是指"每个组内有多少行"**

----##Cache 组相联映射方式有两种方法，一种是高位编组，一种是低位编组，一般使用低位编组

----相联存储器即可按地址寻址，又可按内容寻址

----##指令 Cache 的使用方法和数据 Cache 一样，是把指令当作数据存取，他们分开可以减少指令流水线的冲突

----关于 Cache 的 CPU 执行时间=Cache 命中的指令执行时间+Cache 缺失时带来的额外开销

----##直写策略需要同时写快设备和慢设备，磁盘比主存慢得多，故 Cache-主存可使用直写，而主存-磁盘层次总使用写回策略;

----Cache 与主存之间的地址映射是有硬件自动完成的；主存与外存之间的地址映射是由硬件和操作系统共同完成的（或者说前者是由硬件完成的，后者是由操作系统完成的）

----题目默认 Cache 的访问与 TLB 访问有两种方式：同时访问与未命中再访问低速设备

----题中给出一些目标序号的时候，确认两个重要信息：**序号单位和序号起始**

----##尤其要关注从零开始的第一次算不算。

第四章：指令系统

----##RISC（如 ARM、MIPS）：硬布线控制，CISC：微程序控制，其微程序控制存储器（CM，与 CU 区分!）可能占据 CPU 芯片的 50%以上;

----##寻址时，分清问题问的是地址还是操作数

----转移指令实现程序控制，但仍然是从 PC 中读取

----##为了保持数组的随机访问，数组的虚拟地址是连续的，这和变址寻址有关
----一地址指令格式中，可能有多个操作数，比如另一个操作数由 ACC 给出，或使用堆栈指针
----**各种寻址**（一定要分清）：
隐地址可以简化地址结构
堆栈寻址：指令中没有地址字段，操作数地址隐含在堆栈指针中（注意寻址时要画好图：低、高地址、数据方向等）
相对寻址：便于在程序内部浮动，与当前 PC 值有关
基址寻址：便于编制浮动程序（整段浮动），可以扩大寻址范围
变址寻址：便于编制循环程序（访问数组等），面向用户，用户直接操控的是变址寄存器的内容，将它与形式地址相加得到变址结果
寄存器寻址：寄存器里直接存放数据，不用访存，可以有效地减少地址字段位数
----##跳转指令（jxxx、bne）的 PC 计算公式需要结合编址方式、OFFSET 含义以及是否先将 PC+“1”有关；跳转指令会引起控制相关阻塞，解决办法一般是在 IF 段后面增加三个时钟周期
----题中出现“PC 总是默认+1”的说法是错误的，应该说 PC 加上指令长度

第五章：CPU 系统

----##硬件工作流程“取数据”要加括号
----##要分清指令在存放数据之前还是之后改变地址值
----##汇编程序员可见的寄存器有 基址寄存器、标志寄存器、PC、通用寄存器组，别的一般不可见（如 MAR、MDR、IR），而硬件设计者一般都可见以上的东西。
----##常见控制部件：三态门（三角形，控制开关）、MUX（多路选择器，梯形，用来控制哪一条路接通）、译码器（矩形，用来把一个信号翻译为多个 bit）、加法器（平角内裤，实现加法）、ALU（平角内裤，实现多种基于加法的运算）
----##ALU 宽度即 ALU 运算对象的宽度，一般和字长（等于机器字长）相等，通用寄存器位数也和机器字长相同
----##MAR 和地址线数量相同（为了满足存储扩展的需要，即 MAR 的位数是存储空间的上限，而不一定等于存储空间），MDR 和数据线数量相同，所谓“n 位 CPU”指的是数据总线位数，而“数据字长”指的是例如 int 占 4B，double 占 8B，这与数据线数无关
----##MUX 和 ALU 可以实现 PC+“1”的操作，需要进行多路选择。

----运算器包括：算术逻辑单元（ALU）、暂存寄存器、累加器（ACC）、通用寄存器组（REG）、程序状态字寄存器（PSW）、移位器等
控制器包括：指令部件、时序部件、微操作信号发生器（CU）、中断控制逻辑等
其中指令部件包括：程序计数器（PC）、指令寄存器（IR）和指令译码器（ID）
----##数据通路指数据在功能部件之间传送的路径，包括数据通路上流经的部件，如 PC、ALU、通用寄存器、PSWR、异常和中断处理逻辑、Cache、MMU（内存管理单元，包括分页管理机制的硬件）等，不包括控制部件。

----一个指令周期可能包含不同的机器周期数（每个机器周期都是一个存取周期，每个

存取周期（机器周期）可能都会被 DMA 窃取（DMA 的数据太重要了，使得它的中断响应比别的中断更及时），在一个指令周期中，每个机器周期都有可能被 DMA 中断打断，而不需要整个指令周期结束），如果在末尾检测到其他中断请求，其后就会出现“中断相应周期”

----硬布线控制单元中，CU 的输入信号主要来源是译码器产生的 指令信息 和时序系统产生的 机器周期信号和节拍信号

----##“XXXop”是读写控制信号，可以理解为水龙头的开关，由 CU 发出，其作用是：决定总线上数据流方向、控制存储器的读写类型和方向

----##描述指令执行所需的周期：如“sub R1，R3，（R2）”为“ 1 取出 R1 的数据放进暂存寄存器，2 取出 R2 的数据与 R1 相减写进寄存器，3 读取 R2 的地址数据， 4 根据该地址数据把运算结果写回主存“，共四个时钟周期，这种题需要把文字描述写清楚

----##一定要注意考察的时钟周期从哪到哪儿，依照时间点来斟酌!!

----##指令跳转距离最大不会超过寄存器位数所能表示的数字范围;

----在微程序中“操作控制信号”就是微命令，微操作是微命令的执行过程，他们是一一对应的，而微指令是若干微命令的集合（一条微指令会发出多个微命令），微程序是微指令的有序集合。形成微程序的入口地址的是机器指令的操作码字段，毕竟微程序是机器指令的划分，其开头由机器指令给出

----微指令字段编码方式中，互斥性微命令可以放在同一个字段中，但是该字段还需要留出一个全零以表示不进行任何操作，因此 3 位的字段最多可以表示 7 个互斥微命令

----指令有若干条微指令组成，如 32 条指令，每条指令平均由 4 条微指令组成，再加上两条公共微指令，则总共 130 条微指令，断定法（下地址字段法）微指令寻址至少需要 8 位（与之类似的，下地址字段如果占 8 位，则对多可以表示 256 个微指令）

----计算微指令发出微操作的过程中，紧扣“操作是否相容”，相容则可以出在不同字段同时发出，不相容则需要处于同一个字段（互斥发出）

----微指令寻址方式有两种，一是微指令计数器（CMAR，相当于 PC）给出，二是微指令的下地址字段直接给出

----若指令系统中有 n 种机器指令（微程序），则控制存储器中有 n+1（必须的取址微程序）+1（可能的中断微程序）

----微程序控制器由控制存储器和微指令寄存器直接控制实现，硬布线控制器由逻辑门电路组合实现。前者较慢的原因是需要从控制存储器中读出微指令，但只需要根据时序依次进行即可，时序系统比较简单

----组合逻辑电路不包含记忆存储组件，具有一组输入和一组输出，其输出仅与输入有关，与电路作用前的状态、时刻都无关

时序逻辑电路中的输出不仅与输入的信号有关，还和电路原来的状态有关，时序电路必然包含存储记忆单元

----机器指令的地址字段的作用是存取数据，微指令的地址字段的作用是确定执行顺序

----注意“时钟的前半周期写寄存器，后半周期读寄存器”的方式下，同一个时钟周期内的写、读不冲突;

----指令流水线的五段式：IF（取指，控制器自动进行，根据指令长度不同，操作可能不同）、ID（译码和取数，这虽然在同一段，但确实两个阶段）、EX（执行）、M（访问存储器）、WB（写回寄存器）

----解决数据相关的办法：硬件阻塞（暂停相关指令的执行）、软件插入（nop）、采用旁路电路技术（专门的数据通路，直接把结果送进 ALU 做下一次运算）

解决控制相关的办法：分支预测，尽早生成转移目标地址（算上一卦、准确率挺高）、加快和提前形成条件码

解决结构冒险（资源冲突）的方法：把数据和指令 Cache 分开、访存指令之后的指令暂停一个周期

----采用转发技术时，需要一个周期进行转发，即将数据冲突的中间三个 nop 改为一个 nop

----超标量流水线：空分复用，使用多个功能部件，在同一个时钟周期内发射多条独立命令

超长指令字技术：将多条可以并行的指令组合成一个具有多个操作码的超长指令字，需要多个处理部件

超流水线技术：时分复用，把时钟周期再分段

----若每个指令由 n 个阶段，每个阶段时间为 t，则在理想状态下执行 m 条指令最短的时间为**（m+n-1）t**.（该结论适用于很多场合）

第六章：总线和 I/O 系统（磁盘）

----字符设备：信息交换以字符为单位，不可寻址、效率低，如打印机、鼠标

块设备：以数据块为单位，可寻址，如磁盘

虚拟设备：把一个物理设备映射为多个逻辑设备

----设备分配时可以不考虑及时性

----缓冲池是系统共用资源，多个进程共享，为了管理方便，可以将相同类型的缓冲区连成一个队列

----缓冲主要为了解决 CPU 比 IO 快很多而导致数据积压的问题，所以 IO 速度很快时，缓冲池没有必要设置

----##IO 传输速率要关注单位时间，抓住“毫秒“或”秒“的周期。

----IO 逻辑即设备控制器（中最主要的部分），用于实现对设备的控制

----设备的高效利用最离不开多道程序设计技术

----CRT：一种老式显示器；总线标准有：ISA EISA VESA PCI PCI-EXPRESS（高性能串行总线） AGP USB RS-232C

----##系统处理 IO 请求的顺序：用户程序、系统调用处理程序、设备驱动程序、中断处理程序

----##设备驱动程序可以处理关于设备寄存器的相关读写和寻址，包括计算数据所在的柱面号（高位最耗时，移动磁头）、磁头号（中位，选择磁头）、扇区号（低位，依靠旋转最先动）（磁盘地址的构成），向设备寄存器写入控制命令等，每个类型的设备需要一个设备驱动程序

----与设备相关的中断处理过程是由设备驱动程序完成的，或者说“设备驱动程序可以处理相应控制器或通道发出的中断请求”

----设备驱动程序不能直接控制 IO 设备工作（它只是个中间人啊）

----突发传输的单位可以默认为“磁头不需要移动所能读到的最多的数据”即“同一柱面的所有数据”

----注意 DRAM 一次访问一个字，即使是突发传输，“每次访问需要 16 个时钟”时，总共也需要 64 个时钟。
----提到突发传输时一定要注意考点是什么：一般都是“传输过程包括传输一个地址和一连串的数据“，要区分地址和数据，这是最常考的点。
----##使用地址、数据线复用技术可以节省地址线，并不能提高传输率

----IO 方式：
程序查询：CPU 与设备串行，传送与主程序串行
IO 中断方式：CPU 与设备并行，传送与主程序串行
DMA 方式：CPU 与设备并行，传送与主程序并行（理解），纯硬件电路方式，别的都需要程序干预
通道控制方式（已经从大纲删除）：专门用于解决 IO 的硬件机制（也需要程序控制，即存储在内存里的“通道程序”），缓冲池、覆盖技术、SPOOLing 技术都是软件技术；
----通道与设备控制器、设备的关系：
通道 控制 设备控制器，设备控制器 控制 设备
----##IO 地址线和控制线都是单向的从 CPU 到 IO 接口的
----##在同一个周期内，如果 CPU 处理时间>设备处理时间，则可能导致 IO 数据丢失
----##DMA 方式中如果 DMA 请求得不到相应可能会导致 IO 数据丢失，一般 DMA 可以和 CPU 并行执行（一般的 IO 方式都不会采用串行，除了程序直接控制方式（程序查询方式））
----IO 中断处理方式申请 CPU 处理时间，发生在指令执行结束之后，是在软件的控制下完成的，不会涉及访问内存，故不会传递主存地址数据
----##DMA 方式申请总线时间，直接控制总线传送整个数据块，是在硬件控制器的控制下完成的。
----##用户程序不能直接与 DMA 交流，DMA 获得具体存储位置的信息来源是设备驱动程序
----##DMA 方式与 CPU 使用主存的方式有：
停止 CPU 访存：**直到 DMA 传送完成一块数据才释放总线**
周期挪用：挪用 CPU 一个或几个访存周期，并且优先级高于 CPU，**完成一个字后立即释放总线**
交替访存：把 CPU 周期直接分一部分给 DMA 访存
----##磁盘块的修改（例如更改下地址指针）需要读一次、写一次，共两次访问磁盘。
----##地址位数往往决定了存储单元的个数!!
----##单总线结构的数据通路需要多个时钟周期来完成一条指令，故不适用于单周期处理器。

----总线仲裁方式：
独立请求方式：有 n 根总线请求线和 n 根总线同意线，响应快但很废线
计数器定时方式：n 个设备需要 logn（向上取整）根设备地址线
链式查询方式：优先级固定
分散仲裁：大家举牌，看谁优先级高
----系统使用总线：有利于增减外设、减少信息传输线的条数
----总线宽度等于数据线根数（因为总线是用来传数据的），与地址线、控制线数无关
----总线需要负责裁决，而各种“总线忙”、“总线请求”等信号由申请总线的设备发出

----IO 总线中

控制线：各种时序、控制、相应信号

数据线：数据（寄存器、命令状态等内容）

地址线：与 CPU 交换数据的地址码，用来指明 CPU 要访问的主存或 IO 接口的地址

----高速设备使用局部总线链接，有利于节省系统总线带宽

----##总线事务与定时：同步方式采用统一的时钟信号，异步方式则没有时钟控制信号，依靠双方的"握手"信号，即按需分配时间，一次通信往往会交换多位而非一位数据

----异步传输方式适合于速度差较大的两个设备之间：

一般认为速度由慢到快是：打印机<<<<磁盘<<IO 接口<主存<<Cache<CPU=总线(小于号越多表示速度差异越大，越应该使用异步传输)

----设备控制器 = IO 接口 = IO 控制器

----打印机分为：点阵式、活字式（按能否打印出汉字来）；击打式、非击打式（按打字原理分）

----IO 接口统一编址（要在相对固定的部分编址）时，使用指令系统中的访存指令来完成输入输出操作，独立编址时则需要专门的 IO 指令

----磁盘是一种典型的共享设备，一段时间（并非同一时间）内允许多个用户访问，同一时刻只能有一个进程使用，共享设备是不会引起死锁的

----##磁盘驱动器由磁头、磁盘、读写电路等组成，也就是磁盘本身

----##磁头调度算法：SCAN 是双向的、CSCAN 是单向的，除了 FCFS 之外，都会产生磁壁黏着

----##低级格式化（也称物理格式化）：为磁盘划**分扇区（物理层面的，划出辐射）**、每个扇区采用特别的数据结构，包括校验码;

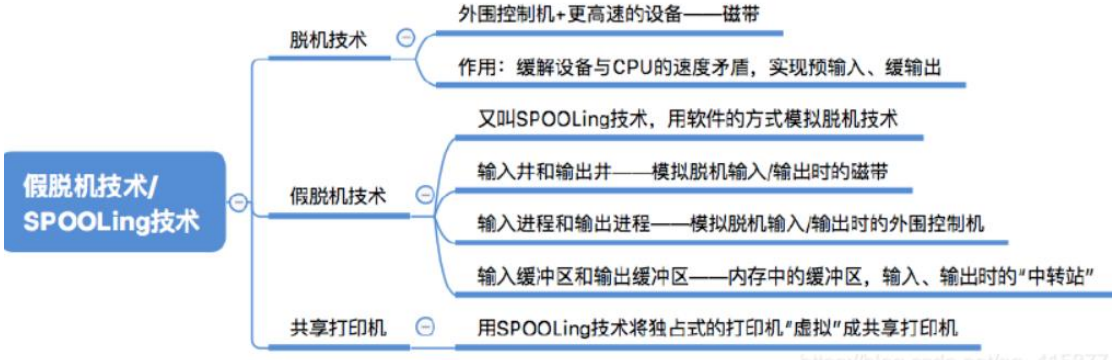
磁盘分区（不属于物理格式化）：将磁盘**按柱面分区（中间层面的，让设备区分 C/，D/）**，每个分区可以作为一个单独的磁盘

逻辑格式化：**创建文件系统（逻辑层面的）**，将初始的文件系统数据结构（包括空闲和已分配的空间以及一个初始为空的目录）存储到磁盘上;

----独立冗余磁盘阵列：RAID0 是无冗余、无校验的磁盘阵列，RAID1~5 均是有冗余（镜像）或校验的阵列，RAID 可以类似流水线那样读取不同磁盘来提高读取速度

----条带化技术：将连续的数据分成很多小部分，分别存储到不同磁盘上去，使多个进程能同时访问数据的多个不同部分而不冲突，而且在需要对这种数据进行顺序访问的时候可以获得最大程度上的 I/O 并行能力。（与 RAID 无关）

----SPOOLING 技术由系统控制 设备与 磁盘中的输入输出井 之间的数据传送



----闪存中静态磨损均衡算法比动态算法更先进，SSD 会在数据写入前，自动分配较新的数据块。

三、操作系统:

历年真题考察内容及特点		
年份	45 题	46 题
2009	生产者消费者问题（PV）	快表、页地址结构
2010	磁盘空闲管理、巡道算法（CSCAN、FCFS）	页地址结构、页面置换算法（FIFO、CLOCK）
2011	银行叫号问题（PV）	文件系统、文件目录结构
2012	空闲页框链表分页系统	索引结构文件系统

2013	生产者消费者问题（PV）	分页管理系统
2014	文件删除和插入	生产者消费者问题（PV）
2015	生产者消费者问题（PV）	二级页表
2016	优先级调度算法	文件目录、FAT
2017	分页虚拟存储管理	线程互斥（PV）
2018	分页虚拟存储	文件系统：索引节点
2019	哲学家进餐问题（PV）	磁盘寻址
2020	前驱关系（PV）	二级页表、局部性原理
2021	信号量、中断方法	操作系统引导
2022	文件系统、读写磁盘	前驱关系（PV）
2023 押题	读者写者问题（PV）	请求分页机制

第一章：概述

----要注意操作系统的几大功能：进程管理、存储管理、文件管理、设备管理，这四个词对应四章，不同章节的内容不能混淆，比如“按名存取“就是专属于文件管理的内容，别的部分不包含这个功能。
----##注意题目的问题所属的类别，当问到“是否属于操作系统的操作”时，要思考选项是否属于硬件操作。
----广义指令就是系统调用命令，而命令解释器属于命令接口，shell 是命令解析器，也属于命令接口，系统中的缓存全部由操作系统管理，对用户是透明的
----系统调用包括很多内容（进程控制、文件系统控制、系统控制、内存管理、网络管理、用户管理、进程间通信等），几乎各个功能都需要用到系统调用。
----多道程序的引入相比单道程序来说，失去了封闭性和顺序性，由于资源竞争产生了制约性
----实时操作系统必须在控制对象规定时间内完成对外部事件的处理，采用抢占式优先级调度算法
----操作系统完成进程调度可以不需要硬件支持

----##内部异常：指来自 CPU 内部的异常，如电源掉电、地址非法、校验错、页面失效（缺页属于异常!!）、非法指令、地址越界、除数为零、陷入指令、算术溢出等，都在指令执行过程中产生，由于内部异常不能被屏蔽，所以产生立马处理，所以也是在执行过程中处理，有的异常必须终止进程，则不能回到原断点执行
----中断：指来自 CPU 执行指令以外事件的发生，如 IO 中断、时钟中断，他们与 CPU 当前运行的程序无关
----可以引发中断的情况：外部事件（如按 ESC），内部异常（如缺页、浮点数上溢（浮点数下溢则直接当作 0 处理，不会中断））、Cache 完全是硬件机构，不会中断
----应用程序在用户态下想执行高危指令（例如需要输入输出）需要发出访管（即 trap

指令、陷入指令）中断，由硬件完成用户态到核心态的转变

----##中断服务:

中断**隐指令**（硬件直接控制执行，不属于指令系统，因此不是指令）：关中断（将中断触发器置 0）、保存断点（保存 PC 的内容）、识别中断源、形成中断服务程序入口地址并送入 PC（用来开启中断服务程序!）

中断**服务程序**（操作系统）：保存现场（保护通用寄存器的内容）、中断事件处理、恢复现场、开中断、中断返回（中断返回指令不等于无条件返回指令，它不会保存 TLB 和 Cache，需要恢复所有寄存器的状态）

----##中断屏蔽字需要写清矩阵，中断优先级比当前执行程序优先级低的中断信号 CPU 是检测不到的，被屏蔽的中断源发出的信号 CPU 也检测不到，屏蔽字中的“1”越多，表示自身的优先级越高。

----中断向量本身是中断入口程序的地址，则中断向量地址就是该中断入口程序地址的地址

----中断 PC 保护由硬件完成，是为了保证安全可靠

----中断优先级：硬件故障>访管指令>外部中断>程序性中断>重新启动

----中断判优可以用软件或硬件实现，一般是硬件排队器

----中断优先级分为响应优先级和处理优先级，中断屏蔽字可以改变后者

----##系统调用、外部中断、缺页：在用户态发生，要执行的程序在核心态下执行，进程切换属于系统调用执行的程序，只会发生在核心态（要区分“调用”和“执行”）

----##开/关中断操作属于特权指令，只能在内核态下运行

----需要切换到内核态执行的指令（特权指令）有：有关对 IO 设备操作的指令、有关访问程序状态的指令、存取特殊寄存器指令、修改页表

不需要切换到内核态执行的有：**通用寄存器清零**

----##子程序调用：保存程序断点（该指令的下一条指令的地址），不用保存 PSWR;

----分层式操作系统结构清晰，便于调试，各层之间只能单向依赖或单向调用，调整时只要不改变相应层间接口，就不会影响其他层，易于扩充维护，但依赖关系固定，不够灵活，由于指令往往需要穿越多层，导致效率不高。

----宏内核与微内核是对立的，优缺点也是对立的，微内核的内核态、用户态切换频繁，导致操作系统开销较大（降低了效率），但可靠性好、易拓展、易移植，如今的多数支持多 CPU 的 OS 都是微内核的，包括 windows。

----##应用软件给用户提供解决具体问题的界面，系统软件（如操作系统）为用户提供基础操作界面

----Linux 系统支持多用户、多任务：多用户指的是一台主机允许多个用户（用户=账号）同时登陆；多任务指的是一台主机允许多个进程客户端通过同一个账户登录

----##操作系统引导:

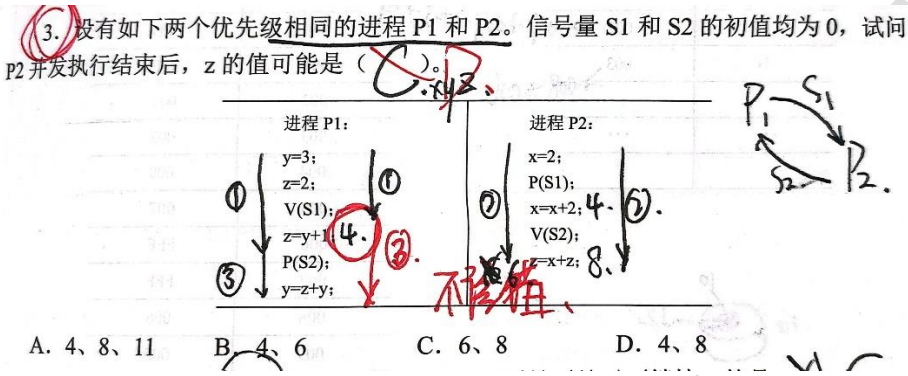
激活 CPU、启动 BIOS 程序（位于 ROM 的自举程序、用于启动具体的设备，**所以在硬件自检之前**）、硬件自检、加载带有操作系统的硬盘、加载主引导记录（MBR）、扫描硬盘分区表、加载分区引导记录（PBR）、加载启动管理器（即操作系统的引导程序）、加载操作系统内核

----##操作系统的初始化过程不包括文件系统建立过程、更不包括硬盘分区，但包括中断向量表的创建，这些的包含关系应当在理性思考之后得出结论

----##操作系统引导：操作系统的程序最终会被送到主存（是 RAM）的系统区中
----32 位操作系统指的是操作系统最多能同时访问 2^32 个地址，而 32 位处理器指的是 CPU 的硬件特性，低位数的 CPU 不能运行高位的操作系统，而高位的 CPU 可以运行低位的操作系统

第二章：进程与同步

----要根据语句的态度优选最可能的答案
----进程异步的伪代码题目中，使用向下的箭头+标号的方法进行模拟计算如图，但应该注意要把每个进程都运行完整：



----##PCB 中的内容包括：进程描述信息、进程控制和管理信息、资源分配清单、CPU 相关信息，（还有进程通信信息，包括消息队列指针、消息资源信号量、消息互斥访问信号量）。PCB 中不包含进程地址空间大小（我的理解是每个进程的地址空间不需要在 PCB 中显示，只需要访问时由页地址机制管理即可）

----进程调度属于进程管理，和进程通信无关
----##管道（进程通信）可以实现半双工通信，但不能实现“双向数据传输”
----管道系统中允许多个进程向管道写入数据，允许多个进程从管道中读数据，但不是同时。
----进程通信中有共享缓冲区，它和进程自己的内存空间是两块地方

----C 语言编写内存分布：正文段（代码和全局变量、常量）、数据堆段（动态分配的存储区）、数据栈段（临时使用的数据变量）
----##睡眠=阻塞，唤醒=就绪（可以重新竞争 CPU），程序退出临界区（释放资源）会唤醒处于等待队列中的就绪态进程（记住!）等待进入临界区的进程不会主动放弃 CPU
----##某进程请求的 IO 完成发出的中断操作会将该进程的状态从等待变为就绪

----多线程系统适用于有多个输入的环境（例如矩阵乘法运算、HTTP 请求等）
----进程中的线程共享进程内的全部资源，但某线程的栈指针对其他线程是透明的
----如果一个进程有多个内核级线程，则只要有一个线程处于就绪状态则该进程处于就绪状态，如果有一个线程正在执行，则该进程正在执行，进程阻塞指的是所有线程都阻塞

----##关于进程的状态变化需要把所有可能发生的变化都写清楚
----调度分为作业调度（高级，每个作业只执行一次，表示作业开始）、内存调度（中级，把作业放到辅存挂起，或从辅存调入就绪）、进程调度（低级，让进程获得 CPU）
----进程处于临界区时，只要不破坏临界区的使用规则，就可以进行处理机调度，比如打印机资源，必须允许调度否则性能很差
----带权周转时间=周转时间/实际运行时间，这个值肯定是大于 1 的

----FCFS 不能抢占，时间片轮转不能不抢占
----FCFS 算法不利于 IO 繁忙型作业，有利于 CPU 繁忙型作业，因为 CPU 繁忙型作业可以一直计算，从容完成，而 IO 繁忙型作业是指在 CPU 处理时要经常进行 IO，导致完成后还需要排队等待调度，因此等待时间长。SJF 算法与之相反。
----用户态和内核态的切换叫做模式切换，不是上下文切换，因为没有改变当前进程，上下文切换只能发生在核心态；调度是决策行为，切换是执行行为
----##不同进程创建的变量不互斥！不同线程对属于同一进程的变量要互斥访问；
----进程互斥原则：空闲让进、忙则等待、有限等待（进程不能等太久，会饥饿）、让权等待（不能进入时，立即放弃处理机）

----可重入代码（即纯代码）允许多个进程访问，不允许任何进程修改。如果要实现修改，进程把执行中部分拷贝到其数据区，只修改该数据区中的内容，不改变共享的代码（适用于共享程序段），其提高系统性能的原因是将多个程序段映射到同一段内存中去，减少了程序段的换入换出，即减少了兑换数量
----相关临界区指的是访问临界资源的代码段
----##皮特森算法（双让步）不会导致饥饿，但不满足“让权等待”，会主动放弃 CPU 的只有信号量机制，即实现让权等待
----硬件指令方法包括 TestAndSet、Swap，都是原子操作，调度时在等待进程中随机选择，优点是简单，缺点是会忙等、会饥饿

----饥饿的判断：如果有源源不断的同类进程进入，别类进程有没有获得处理机的机会
----semaphore：同步信号量，是用来实现同步的（而不是互斥，虽然一般也用来实现互斥），循环过程需要加 while（）（先在草稿纸上写好）
----互斥信号量的初值一般设为 1，同步信号量的初值要根据存量资源确定
----进程在进入管程之前使用 x.wait()把自己放进 x 的阻塞队列里，出管程的时候使用 signal()释放管程
----mutex 要尽量靠近
----在第一段写出问题属于什么互斥同步问题（前驱关系、生产者消费者、哲学家进餐、吸烟者、读者写者）
----哲学家进餐问题模板：

```
定义大锁：semaphore lock=1;
定义资源数：int a=9,b=8,c=6;
Process(){
    While(1){//循环检查资源是否足够
        P(lock);//lock 的作用是对资源变量的互斥访问
        If(所有的资源都够){
            所有资源 int 减小;
```

```
                取 xxx 资源;
                V(lock);
                Break;
            }
            V(lock);
        }//一口气拿完所有的资源
        做进程该做的事;
        P(lock);
        归还所有资源;
        V(lock);
    }
```

----##读者写者问题：关键点在于进入临界区的进程判断自己是不是第一个到的，离开临界区的时候判断自己是不是第一个走的

----吸烟者问题：关键在于（互斥关系：组合资源信号量）和 （同步关系：一手交钱一手交货需要两对互斥信号量，每个人都需要 P 一下、V 一下（普通的同步问题只需要一对就可以了））

----如果刚开始没有该资源（比如服务员在睡觉）则互斥量的初值为 0;

----如果某资源可以同时给三个进程使用，则可以把它的信号量设为 3n

----##进程调度时要分清优先数的大小!!

----读者之间不需要互斥，但读者与写者、写着与写者需要互斥

----##死锁预防：破坏死锁产生的四个条件：

破坏互斥条件：是系统资源能够共享使用

破坏不剥夺条件：一个以保持某些资源的进程如果不能申请到足够的资源，那么必须释放已持有资源，可能导致前一阶段工作失效，效率较低

破坏请求和保持条件：进程运行前必须一次性申请所有资源（即静态资源分配策略）;

破坏循环等待条件：顺序资源分配法，给资源编号，进程按照顺序申请

死锁避免：银行家算法（区分皮特森算法（双让步）），需要知道进程所需的资源数，不会给可能造成死锁的进程分配资源，避免进入不安全状态（不一定就是死锁状态，但会导致死锁）

死锁检测：分配资源时不采取任何措施，但提供死锁的检测和解除手段（死锁定理）

----系统中有 n 个进程时，阻塞队列最多有 n 个进程（死锁了）

第三章：存储管理（见计组）

第四章：文件管理

----系统运行以进程为基本单位，而用户进行输入、输出时以文件为基本单位（逻辑文件是文件存取的基本单位）

----UNIX 系统中，输入输出设备被视为特殊文件，其文件的索引结构放在索引节点中，超级快用来描述文件系统

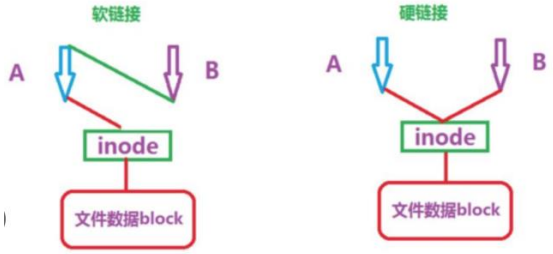
----##**文件目录项=FCB**（组成文件目录的目录项，也不要太死板，需要看题目要求），

包含文件基本信息、存取控制信息、使用信息
----系统级安全管理包括注册、登录
----对文件的访问限制通常由用户访问权限和文件属性决定
----对文件的访问需要路径名（会包括文件名）即可
----为防止文件受损，通常采用备份的方法，而“存取控制矩阵“适用于多用户之间的存取权限保护
----“平均查找的记录数“相当于成功查找长度，例如 100 个顺序文件的”平均查找记录数“为 50

----逻辑文件分为无结构文件（即流式文件）和有结构文件（顺序文件、索引文件、索引顺序文件、直接文件或散列文件）；
物理文件分为连续分配、连接分配（默认隐式分配：目录项含有文件第一块和最后一块的指针，每个块都含有指向下一个盘块的指针；显式分配链接：使用 FAT）、索引分配：inode（重点是混合索引分配）
----磁盘顺序存储并不是一无是处，其存储速度很快，但一般认为是低效的
----##FCB 一般集中存放，这样查找文件的速度更快

----##一个文件的 inode 只有一个，无论被多少个进程打开
----inode 将文件的描述信息从文件目录项中分离出来，减少了查找文件时的 IO 信息量
----##索引节点总数即文件总数，与单个文件的大小无关
----##inode 访问磁盘块时，如果经过一级索引节点，则需要把该节点写入内存，算作一次访问磁盘。读文件至少要经过一次索引节点，再读磁盘具体内容，所以至少会读两次磁盘
----时刻注意文件的读入和写回都会启动磁盘一次
----注意 1596（从 1 开始）/512=3..60，则在第 3 个记录（从 0 开始）中查找第 60 字节（从 0 开始），**对于题目中的每个数字都要注意是从 0 开始还是从 1 开始!!**
----树形目录一般都会使用相对目录，即从相对当前文件位置开始往下检索（如果某个分量名未找到，则停止寻找），以减少从根目录查找带来的时间开销
----文件检索完成后得到的是文件的逻辑文件
----查询方式分为顺序检索和 Hash 法，通常使用顺序检索

----文件共享包括硬链接和符号链接（软连接），如下图示，箭头表示用户文件目录项，左图 inode 引用数为 1，右图引用数为 2



----##空闲盘块管理一般使用空闲空间链表和位示图
----##位图法表示空闲盘块的时候每一个比特可以表示一个盘块

----##简答题需要答出“关键点”比如 FCFS 算法在 Flash 存储器里比 CSCAN 算法在机械磁盘里的优势在于“不需要寻道和旋转延时”
----##open()指的是把指定文件的目录复制到内存，需要路径名、文件名，而 read()需要文件描述符（对于每个进程来说是独立的）、缓冲区首地址、传送的字节数
----Linux 中的虚拟文件系统（VFS）包括四个对象类型：超级块对象、索引节点对象、目录项对象、文件对象

第五章：设备管理（见计组）

四、计网：

历年真题考察内容及特点	
年份	47 题
2009	网络层：子网划分、路由表
2010	传输层：CSMA/CD

2011	网络层、链路层：以太网帧和 IP 分组
2012	网络层、传输层：IP 分组、TCP 链接、路由
2013	网络层：网络拓扑
2014	网络层：路由表
2015	网络层：网络拓扑、IP 和 MAC
2016	链路层：滑动窗口机制，TCP 链接
2017	链路层：后退 N 帧协议、 数据确认
2018	网络层：子网划分
2019	网络层：网络拓扑、IP 分配
2020	网络层：NAT
2021	链路层~应用层：ARP、DNS、广播地址
2022	链路层、网络层：DHCP、CSMA/CD、802.11
2023 押题	网络层：IP、子网、CIDR、路由器

第一章：概述

----计算机网络的资源是指：计算机硬件（路由器、主机）、软件（操作系统、数据库）和数据（web 等）
----分组交换网络的缺点：附加信息开销大
----协议中的要素：语法（数据与控制信息之间的结构或格式，或传输数据的格式），语义（需要发出何种控制信息、完成何种动作以及做出何种响应，或要完成的功能）、同步（规定各种操作的条件、时序关系）
----接口是同一节点内相邻两层之间交换信息的连接点，协议是控制对等实体间进行通信的规则集合
----物理媒体是物理层下面的层，OSI 模型中的会话层可以在文件中插入同步点，避免网络故障重传

	网络层	传输层
OSI 体系	无连接/面向连接	面向连接
TCP/IP 体系	无连接	无连接/面向连接

----网络体系结构是抽象的，不包括各层协议及功能的具体细节，它的分层保证了网络的灵活性和封装性；分层的原则时把网络的功能划分，而不是把相关的功能组合

HTTP报文	HTTP报文分为请求报文&响应报文 请求报文: 1.请求行：请求方法（常用get/post）、请求URL、HTTP协议版本 2.首部行 3.请求体/实体主体 响应报文: 1.状态行 2.响应头部 3.响应体
UDP数据报	1.首部8B，由4个字段组成（都是2B） 2.长度字段包括首部+数据部分 3.检验和检验首部+数据部分（可选）
TCP报文段	1.首部固定部分为20B，最大值为60B（和IP分组一样） 2.源端口和目的端口各占2B 3.序号（本报文段第一个字节的序号）和确认号（期望收到下一个的序号）各占4B 4.数据偏移=首部长度（4B整数倍） 5.确认位ACK、同步位SYN、终止位FIN什么时候为0/1 6.窗口字段表示允许对方发送的数据量（流量控制用）
IP分组	1.首部固定部分为20B，首部最大值为60B 2.总长度（1）+片偏移的单位（8）+首部长度（4）（“一种八片首饰”） 3.标志位MF和DF在分片时的取值 4.生存时间TTL，经过一个路由器减去1，直到为0 5.首部校验和字段只校验首部 6.源地址和目的地址字段长度都为4B
MAC帧	1.前同步码8B 2.MAC地址长度6B 3.数据长度为46-1500B，首部和尾部是18B，因此最短帧长64B。

第二章：物理层

----奈氏准则给出了码元传输速率的限制，但没有对信息传输速率给出限制，有噪音的条件则奈氏准则和香农定理都要计算并取最小值（奈氏准则没有噪音条件，有系数 2）
----频率范围的跨度就是带宽，采样频率是带宽的两倍
----曼彻斯特编码的时钟同步是位于信号中间的边界
----信号有基带信号（将数字信号 1、0 直接用两种不同的电压表示，然后放到数字信道上传输）和宽带信号（将基带信号进行调制后形成频分复用模拟信号，然后放到模拟信道上传输）之分，其区别在于是否对数据进行调制（波形表示）、
----以太网使用两个比特表示一个码元
----报文交换不能用于实时通信环境，因为数据单位是报文，存储转发的时间比较大、不固定
----虚电路有临时性和非临时性两种
----本题需要注意的是：题目考察的内容是电路交换，那么已经给出了“分组长度”等干扰信息，但是不会给太多干扰信息，即使不确定“链路上的延迟”是什么，也应当把它当作非干扰信息，使这个题目的答案看上去合理。

. 设待传送数据总长度为 L 位，分组长度为 P 位，其中头部开销长度为 H 位，源结点到目的结点之间的链路数为 h，每个链路上的延迟时间为 D 秒，数据传输率为 B bps，电路交换建立连接的时间为 S 秒，则电路交换方式传送完所有数据需要的时间是（　　）秒。

A．$hD+L/B$　　B．$S+hD+L/B$

C．$S+hD+PL/((P-H)B)$　　D．$S+L/B$

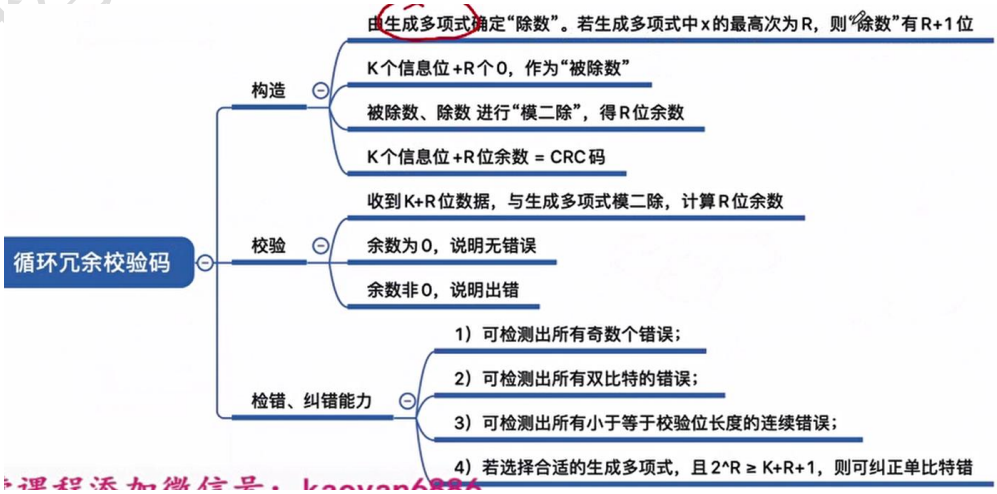
----543 规则：中继器或集线器互联的网络中，任意发送方和接收方最多只能经过 4 个设备、5 个网段，3 个主机段
----放大器用于远距离模拟信号传输，会同时放大信号和噪声，引起失真

中继器将信号再生（不是简单的放大）。
----两个网段在物理层互联时要求数据传输速率相同，但链路层协议可以不同（者可以推广到别的层）
----集线器从一个端口收到数据后，从**其他**所有端口转发出去，而以太网一个主机发送数据后，该以太网内的所有主机（**包括自己**）都会受到该数据

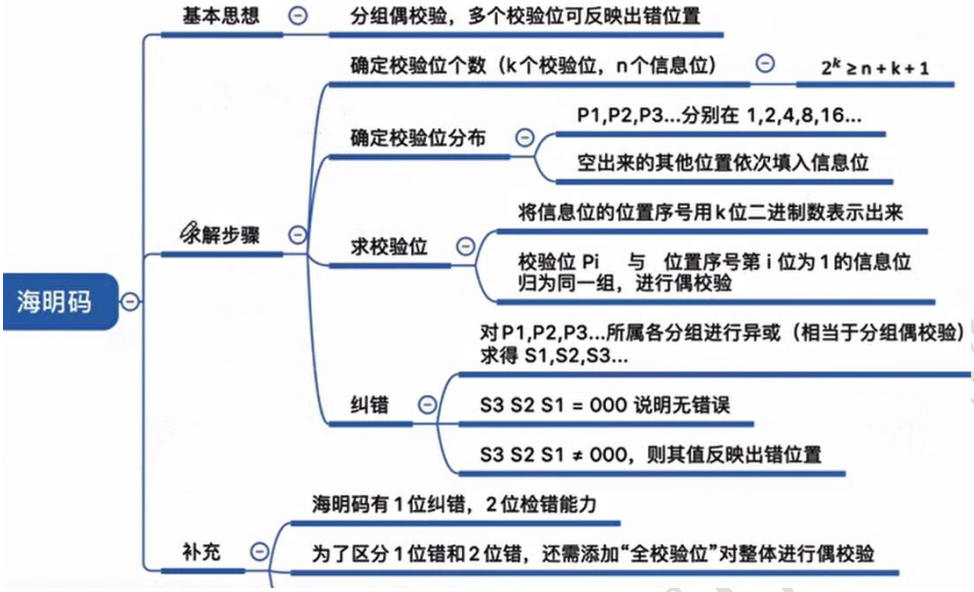
第三章：链路层

----链路层不需要考虑物理层的实现细节
----CSMA 协议中：
1 坚持：可能发生冲突
非坚持：会导致网络空闲
P 坚持：比较综合
----##链路层协议一般都认为是 CSMA/CD
----链路层流量控制有：停止-等待协议（最基础的一个）和滑动窗口协议（**强调滑动窗口，包括 GBN 和 SR**），**这三个都是连续 ARQ（自动重传请求，强调自动重传）协议**
----滑动窗口协议中：窗口大小=发送窗口+接收窗口；分为两种：
GBN（后退 N 帧）：n = n-1 + 1;
SR（选择重传）： n = n/2 + n/2
----注意 GBN 中的用词：“接收窗口内的序号为 4“表示接收方此时期待的是 4 号帧，别的帧一律不收。GBN 中也不存在缓存，故关于”将 5 号帧缓存下来“的说法都是错的
----TDM（时分复用）可以用于数字传输，而 FDM（频分复用）不行
----##链路层在不可靠的物理介质上提供可靠传输，作用有：物理寻址、将数据封装成（定义了数据结构）、流量控制、差错校验，没有拥塞控制功能

----两个码字间的距离：合法码字逐位对比，不同的位的个数
码距：一种编码方案中所有码字之间的最小距离
奇偶校验：加入一位校验位之后，码字中所有 1 的个数为奇数（或偶数）
循环冗余校验：位数不等式和海明码相同，计算方法要会：其校验码为模二除出的余数（除数时 R+1 位，在信息位后增加 R 个 0，这是为了得到 R 位余数后加上去位数刚好）



##**海明码**校验位数 k 需要满足 **2^k>=n+k+1**，其含义是：k 个校验位能表示的错误位数 大于等于 总位数加一个正确情况。



循环冗余校验和海明码校验的信息位效率是差不多的，但是 CRC 没有位置信息所以不能寻位纠错，而计网因为使用自动重传请求，所以不太需要纠错。

----##广播 mac 地址：ff-ff-ff-ff-ff-ff
----同一局域网中有两个设备有相同的 MAC 地址时，它们都不能正常通信
----##以太网 mac 地址最高位为 1 时表示组播地址，这需要分组管理的支持
----**从组播 IP 地址到组播 mac 地址的方法**：
把 IP 地址从后往前数的第 24 位改为 0，然后把低 24 位直接映射为 MAC 地址，前面再加上 MAC 组播的地址（已知）得到结果
----802.11 是无线局域网标准，802.3 是以太网标准
----802.3 中无中继的同轴电缆：粗缆 500m，细缆 185m，双绞线 100m，光纤对 2000m
----##**802.11** 中的地址分为三个，地址 1 是直接接收地址、地址 2 是直接发送地址、地址 3 是额外地址
来自 AP：1 目的地址、2 AP 地址、3 源地址
去往 AP：1 AP 地址、2 源地址、3 目的地址
----##“接收方总是以与数据帧等长的帧进行确认”，则一个周期最短的时间=最短的数据上传时间 +往返传输时延+最短的数据上传时间（需要画图理解）
----“使用稍带确认”指的是确认帧和数据帧一样长，而这个返回来的确认帧不当作有效数据传输
----##以太网数据帧有效载荷的最小长度为 46B，低于这个长度的帧需要填充，设置最短真的目的是为了区分噪声和因发生冲突而异常终止的短帧，设置最长帧的目的是保证所有的站点都能公平竞争接入以太网
----##以太网交换机的交换方式
直通交换方式：只检查帧的目的地址，不进行差错检查，速度很快
存储转发方式：能够丢弃错误分组，隔离碎片和超长帧，速度慢
碎片隔离方式：能隔离碎片，但不隔离超长帧，速度中
----以太网的逻辑结构是总线型的，物理结构是星形的

----互联网可以连接不同类型（广域网、局域网）的网络，在互联网看来，广域网和局域网是平等的

----##交换机的任务：缓存收到的分组，检查其首部，查找转发表，根据首部中的目的地址，将分组转发到交换机的合适端口，然后从缓冲区提取下一个分组。
----交换机中每个端口所占用的带宽不会因为端口节点数目增加而减少，每个都占有全部带宽
----集线器无法隔离广播域，网桥和交换机都能隔离冲突域，但不能隔离广播域，路由器和 VLAN 可以隔离广播域
----##100Base-T 的数据传输速率为 100Mbps
----100Base-T、吉比特以太网都既支持全双工方式、又支持半双工方式，10 吉比特以太网之工作在全双工方式，在全双工模式下都不需要使用 CSMA/CD 协议

----**2*（最远传输距离/传播速度+额外耗时）=最短帧长/传输速率!!!**
----##GBN 协议是哪个层次的？答：GBN 和 SR 都是 TCP 的应用；接收方只会顺序接收帧，失序帧都会被丢弃，所以发送方的重传需要重传未确认的所有帧
----##CDMA 方式中，将站点的码片序列和收到的码片序列进行规格化内积，得到结果为 1 表示发送了 1，**得到-1 表示发送了 0**，得到 0 表示没有发送数据
----CSMA/CA 协议中所有站点发送完数据后，都要等待一段时间才能发送下一帧，成为帧间间隔，帧间间隔短的站点优先级高；
----VLAN 使用交换技术，其划分与计算机的实际物理位置不同，使用软件方式实现逻辑分组管理，其中的计算机可以处于不同的局域网中；它可以有效共享网络资源、简化网络管理、提高网络安全性；可以隔离冲突域和广播域

----##PPP 协议（为两个点建立连接传送数据）的特点：有差错控制，没有流量控制，不可靠；仅支持点对点，不支持多点；只支持全双工链路；面向字节；两端可以运行不同的网络层协议；
----##二进制退避算法：
基本退避时间为 c（往返传播时延，端到端时延的两倍）；从 0 1 . . 2^k-1（k=min(重传次数，10)）中随机选择一个数 r，则 rc 即为重传需要的退避时间；当重传 16 次仍不成功时，则丢弃该帧并报告上层

第四章：网络层

----路由器互联多个局域网，要求每个局域网物理层、链路层、网络层的协议可以不同，但网络层以上的高层协议必须相同
----##DHCP 请求报文段源 IP 为 0.0.0.0，目的 IP 为 255.255.255.255
----IP 分组中的检查字段检查范围是分组首部
----IP 类别：
A：1~126；环回自检地址：127；B：128~191；C：192~223；D：224~239（注意大于等于 224 的是组播地址!!）；
私有地址：
A：10.0.0.0~10.255.255.255（10.开头）

B：**172.16.0.0~172.31**.255.255（**要记住**）

C：192.168.0.0~192.168.255.255（192.168.开头）

----##IP 地址需要用点分十进制回答，但也可以用十六进制进行计算

----##IP 数据报经过路由器转发时可能改变的值：如果主机使用的是私有地址（记 NAT 内部地址），那么源 IP 地址字段需要更换为路由器的全球 IP 地址；生存时间减 1；校验和字段重新计算。如果 IP 分组长度大于 MTU，则需要重新分片（总长度、标志字段、片偏移字段也要发生变化）

----IP 源地址和目的地址只要不经过 NAT 就不会改变，而路由器在收到分组后，剥去链路层协议头，在加上新的链路层协议头转发出去，这个过程中把 mac 地址都改变了，毕竟 mac 地址只有本地意义

----把 IP 网络划分子网的好处是减小广播域

----##“目的 IP 地址“一定要指向最终的地址，而”下一跳地址“则用与路由选择

----##计算“剩余 IP“地址的时候要找准图中已经使用过哪些地址，包括路由器端口用掉的

----##辨别 IP 设置时使用的是 NAT 内部的地址还是全球地址

----NAT 的表项都由管理员添加，不会自学习，这样才能控制内网到外网的链接，其内容包括内网地址和端口、外网地址和端口（可能会出现多个表项是同一个地址，他们的端口不通，即一个主机多个进程的 NAT 表项）

----子网掩码中的 0 和 1 不一定要连续但建议连续

----##**IP 首部**

总长度：单位是 1B，表示本数据包的总长度（**即分片之后的**），而不是整个 IP 数据的长度

标识：辨别不同任务发出的 IP 数据报总体，分开的多个 IP 数据报保持同样的标识

标志：指的是 MF（more fragments）、DF(don’t fragment)

片偏移：单位是 8B，表示分片后，某片在原分组中的位置

生存时间：路由器再转发之前，先把 TTL 减 1，如果 TTL 减到 0，则该丢弃该分组（如果这时的目标是目的主机，则可以发送）。

协议字段：表示上层协议，6 表示 TCP，17 表示 UDP

版本字段：表示本层协议，即 IP 版本，4 表示 Ipv4，6 表示 Ipv6

----##IP 数据报分片的工作是在网络层进行的，注意“一总，八片，首四”，首部的总长度字段是 16 位，则 IP 数据报最大长度为 65535B（这需要分片）

----**CIDR** 中的地址由网络前缀（代替原来的网络号和子网号）和主机号组成，作用是划分子网和聚合超网

----如果没有使用 CIDR 的标志，则按照 IP 类型和来判断子网划分（虽然是已经过时的方式，但仍然可能考察）

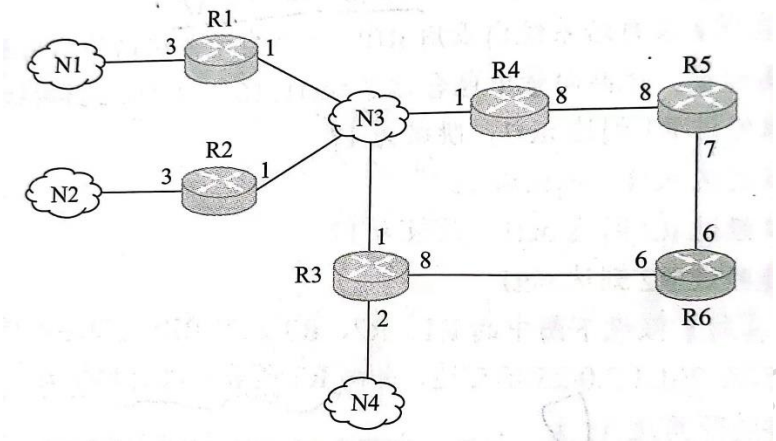
----##PPP 是链路层协议，为网络层提供服务，ICMP 使用 IP 的服务

----对于 ICMP 报文不再发送 ICMP 报文、对第一个分片之后的所有分片都不再产生 ICMP、对组播数据报不产生 ICMP、对 127、0 开头的地址的数据报不产生 ICMP

----PING（应用层、跳过 TCP 直接使用 IP 协议）和 traceroute（网络层协议）可以测试两台主机之间的连通性，PING 使用了 ICMP 询问报文、traceroute 使用了 ICMP 差错报告报文

----PING 的过程：先尝试 127.0.0.0；如果 ping 通则说明本地 TCP/IP 正常工作，接下来 ping 本地 IP 地址，若通，则说明网络适配器（网卡）正常，再 ping 同网段某其他主机 IP 地址，若通，则说明网络线路（网络接口）正常

----##网络拓扑图画法（接口处写 IP 地址）：



----##两个路由器端口之间的网段拥有同样的 IP 网络号，依此可以推断一些端口的完整 IP 地址
----##最小子网指的是可以分配的主机数最少的子网
----子网划分要注意不等长划分方法，类似于不等长的哈夫曼树编码，采用不能让前缀重叠的递推法

----Ipv6 数据包如果太大不能放到链路上，则路由器会将之抛弃，因为 ipv6 不能分片
----路由表是根据路由选择算法得出的，而转发表是路由表构造出来的，包含到输出端口或 mac 地址的映射；路由表由软件实现，而转发表可以用软件实现，也可以用特殊的硬件实现；路由表查找主要靠高速缓存、路由表压缩技术和快速查找算法

----##RIP 协议中矢量值为 16 表示不可达（退避二进制也有这个数字，要敏感）
----RIP（路由选择协议）：距离-向量（路由算法），##基于 UDP，只交换自己的路由表，由于慢收敛导致路由器接受了无效的路由信息，可能导致路由回路（即不收敛，即网络设备的路由表与整个网络拓扑结构不一致，要一致才是好路由表）问题

OSPF（开放最短路径优先协议）：链路状态（路由算法）（Dijkstra），基于 IP，交换与本路由器邻近的所有路由器的链路状态，可以用于很大的自治系统中，分为五种分组，其中的 hello 分组用于保持与邻居的链接

——以上是 IGP（内部网关协议）

BGP（边界网关协议）：路径-向量（路由算法），基于 TCP，——EGP（外部网关协议），首次是交换所有路由表，然后是交换有变化的部分（交换到达某个网络所经过的路径）

----路由选择有直接交付和间接交付，前者发送方和接收方在同一网段内，不需要经过路由器；后者不在同一网段内，最后一步是直接交付
----移动节点到达新的网络后，通过注册把自己的可达信息通知本地代理，本地代理之后接收到发送给他的分组时，将转交其外部代理
----主机离开了它属于的子网后，将不能直接发送和接受分组，但可以通过转交地址来

间接接收和发送

第五章：传输层

----可靠传输协议中的可靠指的是使用确认机制来确保传输的数据不丢失，并不需要很高质量的传输链路。在应用层也可以通过检错、纠错、应答机制保证数据最后准确的传输到目的地。
----TCP 的流量控制使用大小可变的滑动窗口协议，它是 GBN 和 SR 的结合
----##发送端的发送窗口值表明在收到下一个确认之前可以再发送的最大字节数，是 2000 就可以再发送 2000B，具体问题要具体分析，多看选项，斟酌题意，区分“发送窗口”和“接收窗口”，TCP 是甲发出的，那么它只能表示甲所拥有的窗口，即接收窗口，而别人的发送窗口要根据它已经发送的数据报来决定
----##SYN 报文段不能携带数据，但要消耗一个序号，序号是按照字节编号的
----##“慢开始算法“中“快恢复”部分的初始窗口值是“收到重复确认，进行快速重传”时的拥塞窗口的一半，而不是上一次最大拥塞窗口的一半。
----##严格计算经过多少个 RTT 后的窗口大小，把 RTT 和窗口大小的关系找清楚，窗口大小的值是一个时刻点，而 RTT 是时间段
----##考虑确认机制时重点抓住下次传送开始的时间，因为那之前是算作一个周期

----##四次挥手：C-S、S-C、S-C、C-S，其中 C 表示客户端，S 表示服务器，C 在最后一次挥手之后会停留一个 RTT，而 S 可以在第三次挥手后结束链接。
----## TCP 连接建立阶段
客户端：SYN-SENT \ ESTABLISHED
服务器：LISTEN \ SYN-RCVD \ ESTABLISHED
TCP 连接释放阶段
客户端：ESTABLISHED \ FIN-WAIT-1 \ FIN-WAIT-2 \ TIME-WAIT \ CLOSED
（注意其中的常考点：“TIME-WAIT“的作用是确保对方所有的帧都接收完毕，避免旧的 TCP 报文段对后续连接产生错误干扰，其长度为 2*WSL，即两倍的最长报文段寿命）
服务器：ESTABLISHED \ CLOSE-WAIT \ LAST-ACK \ CLOSED

----##熟知端口：FTP：服务器主动方式数据端口 20，控制端口 21；服务器被动方式由客户端决定；HTTP：80（服务器端，客户端的 HTTP 端口号由客户端操作系统分配）；SMTP：25；TELNET：23；
----TCP 协议字段
序号：（TCP 是面向字节流的）表示本报文段的数据的第一个字节的序号
数据偏移（即首部长度）：表示数据起始处距离报文段起始处的距离
----##TCP 首部最短 20B，UDP 首部最短 8B，IP 首部最短 20B，MAC 帧首部加尾部长 18B，MAC 帧数据部分长度 46B~1500B（即 MAC 帧长为 64B~1518B）
----TCP 报文源目端口长度为 2B，IP 分组源目地址为 4B，MAC 数据帧源目地址长度为 6B

第六章：应用层

----客户/服务器模型中，浏览器显示的内容来服务器
----同一域名在不同时间可能解析出不同的 IP 地址，多个域名可以指向同一台主机 IP，Internet 上提供访问服务的主机一定要求 IP 地址，但不一定有域名
----##FTP 的"命令"（包括用户名和密码）是通过控制连接传送的，控制连接全程保持打开，数据连接在每次数据传输完成后就关闭
----FTP 可以从远程计算机获取文件、也能将文件从本地机器传送到远程计算机
----FTP 协议中每个匿名用户都可以使用 anonymous 的 ID，该用户的密码可以是任意的

----##BGP（边界网关协议）适用于不同的自治系统之间的路由信息交换，属于应用层协议，依靠 TCP 的服务
----##单纯的访问 Web 不会用到 SMTP，但会用到 PPP（接入网络）、ARP（寻 MAC 地址）、UDP（DNS 域名解析是基于 UDP 的，是属于 C/S 模型的分布式系统）
----DNS 递归查询中，由最先链接的服务器器给客户端返回地址，两个服务器之间一般都有两次 UDP 报文传输，主机与本地域名服务器之间也有两次
----##当主机不知道对方地址的时候要 ARP 广播查询，如不知道本地域名服务器 mac、或不知道路由器 mac，都需要 ARP 广播
----每个主机都要在授权域名服务器（一般是本地域名服务器）处登记，它将管辖的主机名转换为主机的 IP 地址

----主机访问 web 需要经过：检查主机内有没有服务器 IP，如果没有则与本地域名服务器请求，本地域名服务器经过递归迭代（查一次网络服务器需要一次 RTT，最后查到 abc.com 即可，不需要到 www.abc.com）查询各级服务器给主机返回目的服务器的 IP 地址，主机再与服务器建立 TCP 链接（一个 RTT），然后发送请求收到结果（一个 RTT），计算时间的时候要注意从哪里开始到哪里结束
----##POP3 依靠 TCP 的服务，使用明文来传输密码，不对密码加密
----SMTP 协议规定了两个互相通信的 SMTP 进程之间如何交换信息，采用"推"的通信方式，而 POP3 使用使用"拉"的通信方式
----HTTP 的请求报文中：GET 请求读取由 URL 标识的信息，HEAD 仅请求读取前者的首部，POST 给服务器添加信息（如注释），CONNECT 用于代理服务器
----TELNET 将主机变成远程服务器的一个虚拟终端