Code

- 数组
  - 合并排序的数组
  - 约瑟夫环问题——高效解法
- 栈
  - 栈实现队列
  - 最小栈
  - 逆波兰表达式求值
- 队列
  - 设计循环队列
- 链表
  - 删除链表节点
  - 删除链表中间节点
  - 删除链表的倒数第n个节点
  - 删除链表中的重复元素
  - 相交链表
  - 链表中环的入口点
  - 反转链表
  - 旋转链表
  - 合并两个链表
  - 重排链表
  - 链表排序——插入
  - 链表排序——归并
- 二叉树
  - 中序遍历
  - 前序遍历
  - 后序遍历
  - 二叉树的层序遍历
  - 前序 + 中序 构建二叉树
  - 有序数组转为二叉搜索树
  - 将二叉搜索树变平衡
  - 二叉树的最近公共祖先
  - 层数最深的叶子节点的和
  - 对称二叉树
  - 二叉树的右视图
- 排序
  - 插入 冒泡 选择
  - 希尔 快排 堆排 归并
  - 基数 计数
  - 双轴快排
  - 按奇偶排序数组——快排
  - 荷兰国旗问题——颜色分类——快排
  - 数组中的多数元素——快排
  - 逆序对问题——归并
  - Top K 问题——堆排
  - 最小的 K 个数——堆排
  - 最大间距——基排
  - 排序杂题
    - 相对名次
- 双指针
  - 无重复字符的最长子串
- 二分
  - 旋转数组的最小数字
  - 0~n-1 中缺失的数字
- 深度优先搜索 dfs
  - 括号生成
  - 路径总和
- 贪心
  - 三元组最小距离
- 动态规划
  - 最大子数组和
  - 最大乘积子数组
  - 最长公共子序列
  - 最长公共子数组（最长重复子数组）
  - 最长递增子序列
  - 01背包 *
  - 完全背包 *
  - 零钱兑换——完全背包变体 *
  - 零钱兑换II ——完全背包变体 *
- 杂题

回文数

字符串压缩——模拟

> 背包问题已经标 *，有一说一，几率应该不大，所以不是重点，可以理解则理解，否则 pass。重要的还是排序、树的遍历，递归，非递归这些。图论算法也的看看，尤其是图的遍历，其实和树的一样，就是需要标记一下访问过的节点，避免死循环。
>
> 相信自己，写的代码很多了，一定可以的，即使一眼看着不太会，冷静思考分析一下，或许就会有想法哦，加油！

Code

数组

合并排序的数组

> 给定两个排序后的数组 A 和 B，其中 A 的末端有足够的缓冲空间容纳 B。 编写一个方法，将 B 合并入 A 并排序。
>
> 初始化 A 和 B 的元素数量分别为 m 和 n。

```
void merge(int* A, int ASize, int m, int* B, int BSize, int n){
    if(ASize == 0) return;
    int * c = (int *)malloc(sizeof(int )*ASize);
    int num = 0,j,k;
    for(int i = 0;i < ASize;i++)
        c[i] = 0;
    for(j = 0,k = 0;j < m && k < n;num++)
    {
        if(A[j] <= B[k])
            c[num] = A[j++];
        else
            c[num] = B[k++];
    }
    while(j < m) c[num++] = A[j++];
    while(k < n) c[num++] = B[k++];
    for(int i = 0;i < ASize;i++)
        A[i] = c[i];
    return;
}
```

约瑟夫环问题——高效解法

> 0,1,···,n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字（删除后从下一个数字开始计数）。求出这个圆圈里剩下的最后一个数字。
>
> 例如，0、1、2、3、4这5个数字组成一个圆圈，从数字0开始每次删除第3个数字，则删除的前4个数字依次是2、0、4、1，因此最后剩下的数字是3。

```

int f(int n, int m) {
    if (n == 1) {
        return 0;
    }
    int x = f(n - 1, m);
    return (m + x) % n;
}
int lastRemaining(int n, int m) {
    return f(n, m);
}
```

k 神题解

栈

栈实现队列

> 请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（push、pop、peek、empty）：
>
> 实现 MyQueue 类:
>
> - void push(int x) 将元素 x 推到队列的末尾
> - int pop() 从队列的开头移除并返回元素
> - int peek() 返回队列开头的元素
> - boolean empty() 如果队列为空，返回 true ； 否则，返回 false

> **注意：** 出栈的元素为空时 需要将入栈的元素全部挪到出栈当中

```c
typedef struct {
    int *in_stack;
    int *out_stack;
    int top_in;
    int top_out;
    int size;
} MyQueue;

MyQueue* myQueueCreate() {
    MyQueue* m = (MyQueue *)malloc(sizeof(MyQueue)*1);
    m->in_stack = (int*)malloc(sizeof(int)*110);
    m->out_stack = (int*)malloc(sizeof(int)*110);
    m->top_in = -1;
    m->top_out = -1;
    m->size = 0;
    return m;
}

void myQueuePush(MyQueue* obj, int x) {
    obj->in_stack[++(obj->top_in)] = x;
    obj->size++;
    return ;
}

int myQueuePop(MyQueue* obj) {
    if(obj->top_out == -1)
    {
        while(obj->top_in >= 0)
            obj->out_stack[++(obj->top_out)] = obj->in_stack[(obj->top_in)--];
    }
    (obj->size)--;
    return obj->out_stack[(obj->top_out)--];
}

int myQueuePeek(MyQueue* obj) {
    if(obj->top_out == -1)
    {
        while(obj->top_in >= 0)
            obj->out_stack[++(obj->top_out)] = obj->in_stack[(obj->top_in)--];
    }
    return obj->out_stack[obj->top_out];
}

bool myQueueEmpty(MyQueue* obj) {
    if(obj->size == 0)
        return true;
    else
        return false;
}

void myQueueFree(MyQueue* obj) {
    free(obj->in_stack);
    free(obj->out_stack);
    obj->top_out = -1;
    obj->top_in = -1;
    obj->size = 0;
    return ;
}
/**
 * Your MyQueue struct will be instantiated and called as such:
 * MyQueue* obj = myQueueCreate();
 * myQueuePush(obj, x);
 * int param_2 = myQueuePop(obj);
 * int param_3 = myQueuePeek(obj);
 * bool param_4 = myQueueEmpty(obj);
 * myQueueFree(obj);
*/
```

最小栈

> 设计一个支持 push ， pop ， top 操作，并能在常数时间内检索到最小元素的栈。
>
> - push(x) —— 将元素 x 推入栈中。
> - pop() —— 删除栈顶的元素。
> - top() —— 获取栈顶元素。
> - getMin() —— 检索栈中的最小元素。

```c
typedef struct {
    int *arr;
    int top;
    int size;//xieshangzaishuo
    int min_num;
    int *brr;
} MinStack;


MinStack* minStackCreate() {
    MinStack* m = (MinStack *)malloc(sizeof(MinStack)*1);
    m->arr = (int *)malloc(sizeof(int)*101000);
    m->brr = (int *)malloc(sizeof(int)*101000);
    m->top = -1;
    m->size = 0;
    m->min_num = 2147483647;
    return m;
}

void minStackPush(MinStack* obj, int val) {
    obj->arr[++(obj->top)] = val;
    (obj->size)++;
    if(val < (obj->min_num))
    {
        obj->min_num = val;
        // obj->brr[(obj->top)] = obj->min_num;
    }
    obj->brr[(obj->top)] = obj->min_num;
    // else
        // obj->brr[(obj->top)] =
}

int minStackTop(MinStack* obj) {
    return obj->arr[(obj->top)];
}

void minStackPop(MinStack* obj) {
    (obj->size)-= 1;
    (obj->top)--;
    // 最小的弹出去之后 记得更新 min_num
    if(obj->size == 0)
        obj->min_num = 2147483647;
    else
        obj->min_num = obj->brr[(obj->top)];
}

int minStackGetMin(MinStack* obj) {
    return obj->brr[obj->top];
}

void minStackFree(MinStack* obj) {
    free(obj->arr);
    free(obj->brr);
    obj->size = 0;
    obj->top = -1;
    obj->min_num = 0;
}
```

逆波兰表达式求值

> 根据 逆波兰表示法，求表达式的值。
>
> 有效的算符包括 +、-、*、/ 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。
>
> 说明：

- 整数除法只保留整数部分。
- 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

```c
/*
遇到数字就入栈，遇到符号就将栈顶的两个元素取出进行计算，将计算的结果入栈
*/
int evalRPN(char ** tokens, int tokensSize){
    int *stack = (int *)calloc(tokensSize,sizeof(int));
    int top = 0;
    int a,b;
    for(int i = 0;i < tokensSize;i++)
    {
        char *c = tokens[i];
        if(strlen(c) == 1 && c[0] >= 42 && c[0] <= 47 )
        {
            b = stack[top-1];
            a = stack[top-2];
            top = top- 2;
            switch(c[0])
            {
                case '+':
                    stack[top++] = a+b;
                    break;
                case '-':
                    stack[top++] = a-b;
                    break;
                case '*':
                    stack[top++] = a*b;
                    break;
                case '/':
                    stack[top++] = a/b;
                    break;
            }
        }
        else
            stack[top++] = atoi(c);
    }
    return stack[--top];
}
```

队列

设计循环队列

> 设计你的循环队列实现。 循环队列是一种线性数据结构，其操作表现基于 FIFO（先进先出）原则并且队尾被连接在队首之后以形成一个循环。它也被称为“环形缓冲器”。
>
> 循环队列的一个好处是我们可以利用这个队列之前用过的空间。在一个普通队列里，一旦一个队列满了，我们就不能插入下一个元素，即使在队列前面仍有空间。但是使用循环队列，我们能使用这些空间去存储新的值。
>
> 你的实现应该支持如下操作：
>
> - MyCircularQueue(k): 构造器，设置队列长度为 k 。
> - Front: 从队首获取元素。如果队列为空，返回 -1 。
> - Rear: 获取队尾元素。如果队列为空，返回 -1 。
> - enQueue(value): 向循环队列插入一个元素。如果成功插入则返回真。
> - deQueue(): 从循环队列中删除一个元素。如果成功删除则返回真。
> - isEmpty(): 检查循环队列是否为空。
> - isFull(): 检查循环队列是否已满。

```c
typedef struct {
    int k;
    int *queue;
    int length;
    int front,rear;
} MyCircularQueue;

MyCircularQueue* myCircularQueueCreate(int k) {
    MyCircularQueue *q = (MyCircularQueue *)calloc(1,sizeof(MyCircularQueue));
    q->queue = (int *)calloc(k,sizeof(int));
    q->k = k;
    q->front = -1;
```

```
    q->rear = -1;
    q->length = 0;
    return q;
}

bool myCircularQueueEnQueue(MyCircularQueue* obj, int value) {
    if(obj->length == obj->k)
        return false;

    if(obj->front == -1 && obj->rear == -1)
    {
        obj->front = 0;
        obj->rear = 0;
    }
    obj->queue[obj->rear] = value;
    obj->rear = (obj->rear+1)%obj->k;
    obj->length++;
    return true;
}

bool myCircularQueueDeQueue(MyCircularQueue* obj) {
    if(obj->length == 0)
        return false;

    obj->front = (obj->front+1)%obj->k;
    obj->length--;
    return true;
}

int myCircularQueueFront(MyCircularQueue* obj) {
    if(obj->length == 0)
        return -1;

    return obj->queue[obj->front];
}

int myCircularQueueRear(MyCircularQueue* obj) {
    if(obj->length == 0)
        return -1;

    if(obj->rear - 1 == -1)
        return obj->queue[obj->rear-1+obj->k];

    return obj->queue[obj->rear-1];
}

bool myCircularQueueIsEmpty(MyCircularQueue* obj) {
    if(obj->length != 0)
        return false;

    return true;
}

bool myCircularQueueIsFull(MyCircularQueue* obj) {
    if(obj->length != obj->k)
        return false;

    return true;
}

void myCircularQueueFree(MyCircularQueue* obj) {
    obj->front = -1;
    obj->rear = -1;
    obj->length = 0;
    if(obj->queue != NULL)
    {
        free(obj->queue);
        obj->queue = NULL;
    }
    if (obj != NULL) {
        free(obj);
```

```
        obj = NULL;
    }
}

/**
 * Your MyCircularQueue struct will be instantiated and called as such:
 * MyCircularQueue* obj = myCircularQueueCreate(k);
 * bool param_1 = myCircularQueueEnQueue(obj, value);
 * bool param_2 = myCircularQueueDeQueue(obj);
 * int param_3 = myCircularQueueFront(obj);
 * int param_4 = myCircularQueueRear(obj);
 * bool param_5 = myCircularQueueIsEmpty(obj);
 * bool param_6 = myCircularQueueIsFull(obj);
 * myCircularQueueFree(obj);
 */
```

链表

删除链表节点

> 给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。
>
> 返回删除后的链表的头节点。

```c
struct ListNode* deleteNode(struct ListNode* head, int val){
    int num = val;
    struct ListNode* s = (struct ListNode*)malloc(sizeof(struct ListNode)*1);
    s->next = head;
    struct ListNode*p,*q;
    p = s;
    while(p->next!=NULL)
    {
        if(p->next->val == num)
        {
            q = p->next;
            p->next = q->next;
            break;
        }
        p = p->next;
    }
    return s->next;
}
```

删除链表中间节点

> 给你一个链表的头节点 head 。删除 链表的 中间节点 ， 并返回修改后的链表的头节点 head 。
> 长度为 n 链表的中间节点是从头数起第 [n / 2] 个节点（下标从 0 开始），其中 [x] 表示小于或等于 x 的最大整数。
>
> - 对于 n = 1、2、3、4 和 5 的情况，中间节点的下标分别是 0、1、1、2 和 2 。

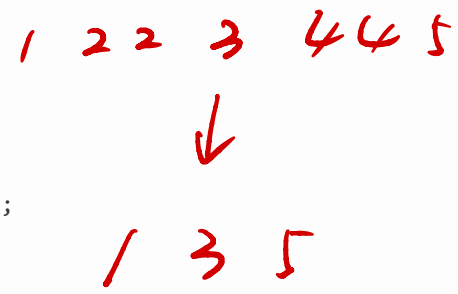
```c
// 一波优雅的快慢指针
struct ListNode* deleteMiddle(struct ListNode* head){
    if(head->next == NULL)
        return NULL;
    struct ListNode* l = (struct ListNode*)calloc(1,sizeof(struct ListNode));
    l->next = head;
    struct ListNode*p,*q;
    q = head;
    p = l;
    while(q != NULL && q->next != NULL)
    {
        q = q->next->next;
        p = p->next;
    }
    // q = p->next;
    p->next = p->next->next;
    return l->next;
}
```

删除链表的倒数第n个节点

```c
struct ListNode* removeNthFromEnd(struct ListNode* head, int n){
    struct ListNode *p,*q;
    struct ListNode *l = (struct ListNode*)malloc(sizeof(struct ListNode)*1);
    l->next = NULL;
    l->next = head;
    p = q = l; //p->next指向第n-1结点 q->next指向后边的结点
    while(n--)
    {
        p = p->next;
    }
    while(p->next != NULL)
    {
        p = p->next;
        q = q->next;
    }
    p = q->next;
    q->next = q->next->next;
    free(p);
    return l->next;
}
```

删除链表中的重复元素

> 存在一个按升序排列的链表，给你这个链表的头节点 head ，请你删除链表中所有存在数字重复情况的节点，只保留原始链表中 没有重复出现 的数字。
>
> 返回同样按升序排列的结果链表。



```c
struct ListNode* deleteDuplicates(struct ListNode* head){
    if(head == NULL)
        return NULL;
    if(head->next == NULL)
        return head;
    struct ListNode* s,* p,* pre,* q;
    struct ListNode* l = (struct ListNode *)malloc(sizeof(struct ListNode)*1);
    s = head->next;
    l->next = head;
    pre = l;
    // int x = pre->next->val;
    p = head;
    while(p != NULL && p->next != NULL)
    {
        if(p->val == s->val)
        {
            while(p->val == s->val && s->next != NULL)
            {
                q = s;
                s = s->next;
                p->next = s;
                free(q);
            }
            if(s->val == p->val && s->next == NULL)
            {
                p = NULL;
                pre->next = p;
                break;
            }
            q = p;
            pre->next = q->next;
            free(q);
            // pre = pre->next;
            p = pre->next;
            s = p->next;
        }
        else
        {
            pre = p;
            p = s;
            s = s->next;
        }
```

```
    }
    return l->next;
}
```

相交链表

> 给你两个单链表的头节点 `headA` 和 `headB` ，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 `null` 。

```
// 觉得优雅的不太优雅代码
struct ListNode *getIntersectionNode(struct ListNode *headA, struct ListNode *headB) {
    struct ListNode *s = (struct ListNode *)malloc(sizeof(struct ListNode)*1);
    struct ListNode * a, * b,* a_x,* b_x;
    a = headA;
    b = headB;
    int len = 0;
    int flag = 1;
    while(a != NULL && b != NULL)
    {
        a = a->next;
        b = b->next;
    }
    while(a!= NULL)
    {
        len++;
        flag = 1;
        a = a->next;
    }
    while(b!= NULL)
    {
        len++;
        flag = 0;
        b = b->next;
    }
    a_x = headA;
    b_x = headB;
    while(len > 0)
    {
        if(flag == 0)
            b_x  = b_x->next;
        else
            a_x = a_x->next;
        len--;
    }
    while(a_x != NULL)
    {
        if(a_x == b_x)
            return a_x;
        else
        {
            a_x = a_x -> next;
            b_x = b_x -> next;
        }
    }
    return NULL;
}

// 觉得不优雅的优雅代码
struct ListNode *getIntersectionNode(struct ListNode *headA, struct ListNode *headB) {
    if (headA == NULL || headB == NULL) {
        return NULL;
    }
    struct ListNode *pA = headA, *pB = headB;
    while (pA != pB) {
        pA = pA == NULL ? headB : pA->next;
        pB = pB == NULL ? headA : pB->next;
    }
    return pA;
}
```

链表中环的入口点

首先快慢指针一起走，直到相遇。然后将快指针置为头节点，快慢指针同步向前走，相遇就是入口节点。

```c
struct ListNode *detectCycle(struct ListNode *head) {
    struct ListNode *slow = head;
    struct ListNode *first = head;
    while(1)
    {
        if(first == NULL || first -> next == NULL)
            return NULL;

        slow = slow -> next;
        first = first -> next -> next;
        if(first == slow)
            break;
    }

    first = head;
    int k = 0;
    while(first != slow)
    {
        first = first -> next;
        slow = slow -> next;
        k++;
    }
    return slow;
}
```

反转链表

```c
struct ListNode* reverseList(struct ListNode* head){
    struct ListNode* pre;
    struct ListNode* a;
    struct ListNode* end;
    pre = NULL;

    a = head;
    while(a != NULL)
    {
        end = a->next;
        a->next = pre;
        pre = a;
        a = end;
    }
    return pre;
}
```

旋转链表

> 给你一个链表的头节点 `head` ，旋转链表，将链表每个节点向右移动 `k` 个位置。

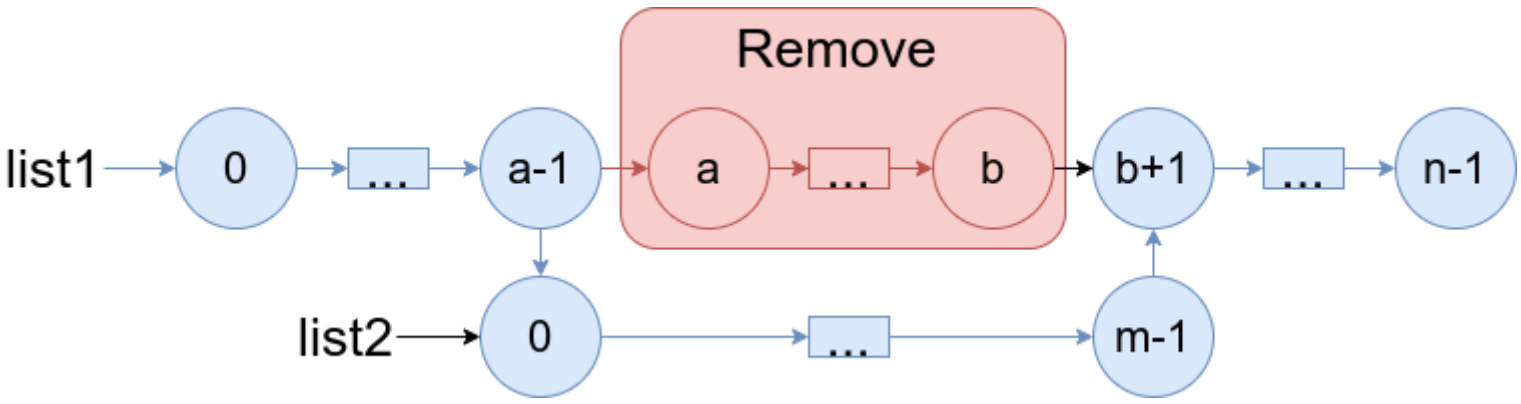
```c
struct ListNode* rotateRight(struct ListNode* head, int k){
    if(head == NULL)
        return NULL;
    if(head->next == NULL)
        return head;

    struct ListNode* slow,* first;
    struct ListNode* s = head;
    int len = 0;
    while(s!= NULL)
    {
        s = s->next;
        len++;
    }
    k = k%len;
    // printf("%d",k);
    if(k == 0)
        return head;
    struct ListNode* lnode = (struct ListNode* )malloc(sizeof(struct ListNode)*1);
    lnode->next = head;
    slow = first = lnode;
```

```
    while(k--)
    {
        first = first->next;
    }

    while(first->next != NULL)
    {
        slow = slow->next;
        first = first->next;
    }
    struct ListNode* l;
    l = slow->next;
    slow->next = NULL;
    slow = l;
    // if(slow == head)
        // return head;
    while(l->next != NULL && l != NULL)
    {
        l = l->next;
    }
    l->next = head;
    return slow;
}
```

合并两个链表

> 给你两个链表 list1 和 list2 ， 它们包含的元素分别为 n 个和 m 个。
>
> 请你将 list1 中下标从 a 到 b 的全部节点都删除，并将list2 接在被删除节点的位置。
>
> 下图中蓝色边和节点展示了操作后的结果：
>
> 
>
>
> 请你返回结果链表的头指针。

```
struct ListNode* mergeInBetween(struct ListNode* list1, int a, int b, struct ListNode* list2){
    struct ListNode* s;
    struct ListNode* pre;
    struct ListNode* f;
    s = list1;
    int cha = b - a;
    while(--a)
        s = s->next;

    pre = s;
    s = s->next;
    while(cha--)
    {
        f = s;
        s = s->next;
        free(f);
    }
    pre -> next = list2;

    while(pre -> next != NULL)
        pre = pre->next;

    pre->next = s->next;
    return list1;
}
```

重排链表

> 给定一个单链表 L 的头节点 head ， 单链表 L 表示为：
>
> L0 → L1 → … → Ln - 1 → Ln 请将其重新排列后变为：
>
> L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → … 不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

```c
struct ListNode* reverseList(struct ListNode* head) {
    struct ListNode thead;
    thead.next = NULL;
    struct ListNode* pre = &thead, *cur = head, *nxt = NULL;
    while (cur) {
        nxt = cur->next;
        cur->next = pre->next;
        pre->next = cur;
        cur = nxt;
    }
    return thead.next;
}

void reorderList(struct ListNode* head){
    if (head == NULL) return head;
    struct ListNode *fp = head, *sp = head;
    while (fp->next && fp->next->next) {
        fp = fp->next->next;
        sp = sp->next;
    }
    struct ListNode *sh = reverseList(sp->next);
    sp->next = NULL;
    struct ListNode *th = head, *tp = NULL;
    while (th && sh) {
        tp = sh->next;
        sh->next = th->next;
        th->next = sh;
        th = sh->next;
        sh = tp;
    }
    return head;
}
```

链表排序——插入

```c
struct ListNode* insertionSortList(struct ListNode* head){
    if(head == NULL || head->next == NULL)
        return head;

    struct ListNode *l = (struct ListNode *)malloc(sizeof(struct ListNode));
    l->val = 0;
    l->next = head;
    struct ListNode *pre = head;
    struct ListNode *cur = head->next;
    //cur外循环遍历
    while (cur != NULL)
    {
        if(pre->val <= cur->val)
            pre = pre->next;
        else
        {
            struct ListNode *move = l;
            while(move->next->val <= cur->val)
                move = move->next;

            pre->next = cur->next;
            cur->next = move->next;
            move->next = cur;
        }
        cur = pre->next;
    }
    return l->next;
}
```

链表排序——归并

```c
struct ListNode * merge(struct ListNode *m,struct ListNode *n)
{
    struct ListNode *d = (struct ListNode *)malloc(sizeof(struct ListNode)*1);
    struct ListNode *head = d;
    while(m!=NULL && n!=NULL)
    {
        if(m->val <= n->val)
        {
            head->next = m;
            head = head->next;
            m = m->next;
        }
        else
        {
            head->next = n;
            head = head->next;
            n = n->next;
        }
    }
    if(m!=NULL)
        head->next = m;
    if(n!=NULL)
        head->next = n;
    return d->next;
}
struct ListNode* mergesort(struct ListNode *low,struct ListNode *mid)
{
    if(low == NULL)
        return NULL;
    if(low -> next == mid)
    {
        low->next = NULL;
        return low;
    }
    //归并
    struct ListNode *s = low;
    struct ListNode *f = low;
    while(f != mid && f->next != mid)
    {
        s = s->next;
        f = f->next->next;
    }
    return merge(mergesort(low,s),mergesort(s,mid));
}
struct ListNode* sortList(struct ListNode* head){
    if(head == NULL) return NULL;
    return mergesort(head,NULL);
}
```

二叉树

中序遍历

```c
// 递归
void inorder(struct TreeNode* root,int returns[],int *returnSize)
{
    if(root == NULL)
        return;
    inorder(root->left,returns,returnSize);
    returns[(*returnSize)++] = root->val;
    inorder(root->right,returns,returnSize);
}

// 非递归
int* inorderTraversal(struct TreeNode* root, int* returnSize){
    *returnSize = 0;
    int* returns = (int *)malloc(sizeof(int )*110);
    struct TreeNode *s[110];
    int top = 0;
    struct TreeNode *m = root;
```

```
    while(m!=NULL || top != 0)
    {
        if(m)
        {
            s[top++] = m;
            m = m->left;
        }
        else
        {
            m = s[--top];
            returns[(*returnSize)++] = m->val;
            m = m -> right;
        }
    }
    return returns;
}
```

前序遍历

```
// 递归
void preorder(struct TreeNode* root,int returns[],int *returnSize)
{
    if(root == NULL) return;

    returns[*returnSize] = root->val;
    (*returnSize)++;
    preorder(root->left,returns,returnSize);
    preorder(root->right,returns,returnSize);
}

// 非递归
typedef struct stack{
    struct TreeNode* data[110];
    int top;
}stack;
void push(stack *s,struct TreeNode *x)
{
    s->data[(s->top)] = x;
    (s->top)++;
}
struct TreeNode *pop(stack *s,struct TreeNode *x)
{
    x = s->data[--(s->top)];
    return x;
}
void init(stack *s)
{
    (s->top) = 0;
}
int empty(stack s)
{
    if(s.top == 0)
        return 0;
    return 1;
}
void visit(struct TreeNode* x,int returns[],int *returnSize)
{
    returns[(*returnSize)++] = x->val;
}
int* preorderTraversal(struct TreeNode* root, int* returnSize){
    *returnSize = 0;
    int* returns = (int *)malloc(sizeof(int)* 110);
    struct TreeNode *m;
    m = root;
    stack s;
    init(&s);
    while(m != NULL || empty(s)!= 0)
    {
        if(m)
        {
            visit(m,returns,returnSize);
```

```
            push(&s,m);
            m = m->left;
        }
        else
        {
            m = pop(&s,m);
            m = m->right;
        }
    }
    return returns;
}
```

后序遍历

```
// 递归
void postorder(struct TreeNode* root,int returns[],int *returnSize)
{
    if(root == NULL) return;

    postorder(root->left,returns,returnSize);
    postorder(root->right,returns,returnSize);
    returns[(*returnSize)++] = root->val;
}

// 非递归
int* postorderTraversal(struct TreeNode* root, int* returnSize){
    *returnSize = 0;
    int *returns = (int *)malloc(sizeof(int)* 110);
    struct TreeNode* data[110];
    int top = 0;
    struct TreeNode* m = root;
    struct TreeNode* pre = NULL;
    while(m || top != 0)
    {
        while(m)
        {
            pre = m;
            data[top++] = m;
            m = m->left;
        }
        m = data[--top];
        if(m->right != NULL && m->right != pre)
        {
            data[top++] = m;
            m = m->right;
        }
        else
        {
            pre = m;
            returns[(*returnSize)++] = pre->val;
            m = NULL;
        }
    }
    return returns;
}
```

二叉树的层序遍历

> 给你一个二叉树，请你返回其按 **层序遍历** 得到的节点值。 （即逐层地，从左到右访问所有节点）。

```
// em 优雅!
// bfs 模板题 一层一层访问
void bfs(struct TreeNode* root,int **returns,int *returnSize,int **returnColumnSizes)
{
    int front = 0;
    int rear = 0;
    struct TreeNode* queue[1010];
    queue[rear++] = root;

    while(front != rear)
```

```
    {
        int cha = rear - front;
        returns[(*returnSize)] = (int *)malloc(sizeof(int)*(cha));
        for(int i = 0;i < cha;i++)
        {
            returns[(*returnSize)][i] = queue[front]->val;
            if(queue[front]->left != NULL)
                queue[rear++] = queue[front]->left;

            if(queue[front]->right != NULL)
                queue[rear++] = queue[front]->right;

            front++;
        }
        (*returnColumnSizes)[*returnSize] = cha; //这句zhejvlueluelue
        (*returnSize)++;
    }
}
int** levelOrder(struct TreeNode* root, int* returnSize, int** returnColumnSizes){
    *returnSize = 0;
    if(root == NULL)
        return NULL;

    int **returns = (int **)malloc(sizeof(int*)*1010);

    *returnColumnSizes = (int *)malloc(sizeof(int)*1010);
    bfs(root,returns,returnSize,returnColumnSizes);
    return returns;
}
```

前序 + 中序 构建二叉树

```
/*
给定二叉树的前序和中序遍历序列，还原二叉树
先序遍历访问顺序是：根 左 右
中序遍历访问顺序是：左 根 右
则通过先序遍历的根，可以将中序遍历分成左右两个部分，然后将左右两个部分分别生成左右子树
*/
struct TreeNode* dfs(int preorder[],int p_start,int p_end,int inorder[],int i_start,int i_end)
{
    if(p_start == p_end)
        return NULL;
    int root = preorder[p_start];
    int i_index;
    for(int i = i_start;i < i_end;i++)
    {
        if(inorder[i] == root)
        {
            i_index = i;
            break;
        }
    }
    int p_num = i_index-i_start;
    struct TreeNode* t = (struct TreeNode*)malloc(sizeof(struct TreeNode)*1);
    t->val = root;
    t->left = dfs(preorder,p_start+1,p_start+p_num+1,inorder,i_start,i_index);
    t->right = dfs(preorder,p_start+p_num+1,p_end,inorder,i_index+1,i_end);
    return t;
}
struct TreeNode* buildTree(int* preorder, int preorderSize, int* inorder, int inorderSize){
    return dfs(preorder,0,preorderSize,inorder,0,inorderSize);
}
```

有序数组转为二叉搜索树

> 给你一个整数数组 nums ，其中元素已经按 升序 排列，请你将其转换为一棵 高度平衡 二叉搜索树。
>
> 高度平衡 二叉树是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1 」的二叉树。

```
struct TreeNode* dfs(int returns[],int low,int high)
{
```

```
    if(low > high)
        return NULL;

    int mid = (low + high)/2;
    struct TreeNode *t = (struct TreeNode* )malloc(sizeof(struct TreeNode)*1);
    t->val = returns[mid];
    t->left = dfs(returns,low,mid-1);
    t->right = dfs(returns,mid+1,high);
    return t;
}

struct TreeNode* sortedArrayToBST(int* nums, int numsSize){
    return dfs(nums,0,numsSize-1);
}
```

将二叉搜索树变平衡

> 给你一棵二叉搜索树，请你返回一棵 平衡后 的二叉搜索树，新生成的树应该与原来的树有着相同的节点值。
>
> 如果一棵二叉搜索树中，每个节点的两棵子树高度差不超过 1 ，我们就称这棵二叉搜索树是 平衡的 。
>
> 如果有多种构造方法，请你返回任意一种。

```
struct TreeNode* dfs(int returns[],int low,int high)
{
    if(low > high)
        return NULL;

    int mid = (low + high)/2;
    struct TreeNode *t = (struct TreeNode* )malloc(sizeof(struct TreeNode)*1);
    t->val = returns[mid];
    t->left = dfs(returns,low,mid-1);
    t->right = dfs(returns,mid+1,high);
    return t;
}

void visit(struct TreeNode* root,int returns[],int *returnSize)
{
    if(root == NULL)
        return;
    visit(root->left,returns,returnSize);
    returns[(*returnSize)++] = root->val;
    visit(root->right,returns,returnSize);
}

struct TreeNode* balanceBST(struct TreeNode* root){
    //先遍历 存储到一个数组中，将整棵树 以一个 升序序列存放
    int *returns = (int *)malloc(sizeof(int)* 10010);
    int returnSize = 0;
    visit(root,returns,&returnSize);
    //再以中间建立 二叉搜索平衡树
    return dfs(returns,0,returnSize-1);
}
```

二叉树的最近公共祖先

> 最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
struct TreeNode* dfs(struct TreeNode* root,struct TreeNode* p,struct TreeNode* q)
{
    struct TreeNode* l,* r;
    if(root == NULL)
        return NULL;
```

```
    if(root == p || root == q)
        return root;
    l = dfs(root->left,p,q);
    r = dfs(root->right,p,q);
    if(l != NULL && r != NULL)
        return root;
    else if(l == NULL && r!= NULL)
        return r;
    else
        return l;
}

struct TreeNode* lowestCommonAncestor(struct TreeNode* root, struct TreeNode* p, struct TreeNode* q) {
    //已知层序遍历的数组
    return dfs(root,p,q);
}
```

层数最深的叶子节点的和

> 给你一棵二叉树的根节点 `root` ，请你返回 **层数最深的叶子节点的和** 。

```
/*
通过三个参数分别记录递归过程中 达到的最大深度（max_deepth），当前的深度（deepth），最大深度的和(sum)

如果达到新的最大深度，更新最大深度，最大深度和归零
如果当前节点是叶子节点，判断当前深度是否是最大深度，如果是，将当前节点的值累加到最大深度和
*/
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
struct TreeNode* dfs(struct TreeNode* root,int deepth,int * sum,int *max_deepth)
{
    if(root == NULL)
        return NULL;
    if(deepth > (*max_deepth))
    {
        (*max_deepth) = deepth;
        (*sum) = 0; //guiling heihei
    }
    struct TreeNode *left,*right;
    left = dfs(root->left,deepth+1,sum,max_deepth);
    right = dfs(root->right,deepth+1,sum,max_deepth);
    if(left == NULL && right == NULL && deepth == (*max_deepth))
        (*sum) += root->val;
    return root;
}

int deepestLeavesSum(struct TreeNode* root){
    int sum = 0;
    if(root == NULL)
        return 0;
    int max_deepth = 0;
    dfs(root,0,&sum,&max_deepth);
    return sum;
}
```

对称二叉树

> 给定一个二叉树，检查它是否是镜像对称的。

```
bool dfs(struct TreeNode* left_,struct TreeNode* right_)
{
    if(left_ == NULL &&right_ == NULL)
        return true;
    if(left_ == NULL || right_ == NULL || right_->val != left_->val)
        return false;
```

```
    // 左的左子树和右的右子树   左的右子树和右的左子树 均互相对称
    return dfs(left_->left,right_->right) && dfs(left_->right,right_->left);
}

bool isSymmetric(struct TreeNode* root){
    if(root == NULL)
        return true;
    return dfs(root->left, root->right);
}
```

二叉树的右视图

> 给定一个二叉树的 **根节点** `root`，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

```
/*
bfs 记录每一层的最后一个节点
*/
int* rightSideView(struct TreeNode* root, int* returnSize){
    *returnSize = 0;
    if(root == NULL)
        return NULL;
    int *returns = (int *)malloc(sizeof(int)*110);
    struct TreeNode* queue[120];
    int front = 0;
    int rear = 0;
    memset(returns,0,sizeof(returns));
    queue[rear++] = root;
    while(front != rear)
    {
        int cha = rear - front;
        while(cha--)
        {
            if(queue[front]->left != NULL)
                queue[rear++] = queue[front]->left;
            if(queue[front]->right != NULL)
                queue[rear++] = queue[front]->right;
            front++;
        }
        returns[(*returnSize)++] = queue[front-1]->val;
    }
    return returns;
}
```

排序

> 本来想粘过来的 想想还是算了 直接上大佬的博客

插入 冒泡 选择

> em 首先有请 n^2 的登场
>
> biu：(25条消息) 排序算法总结—时间复杂度O(n^2)—冒泡/插入/选择小记_一君子兮-CSDN博客

希尔 快排 堆排 归并

> 然后是 $n * log_n$ di
>
> biu：(25条消息) leetcode排序算法总结—时间复杂度o(nlogn)-希尔/堆排/快排/归并小记_一君子兮-CSDN博客

基数 计数

> 最后是 n 啦
>
> biu：(25条消息) 排序算法总结—时间复杂度O(n)—基数排序/计数排序小记_一君子兮-CSDN博客

双轴快排

```
void swap(int *a,int *b)
{
    int t = *a;
    *a = *b;
```

```
    *b = t;
}
void sort(int *nums,int start,int end)
{
    if(start > end) return;
    int left = start;
    int right = end;

    if(nums[start] == nums[end])
    {
        for(int i = start;i < end;i++)
        {
            if(nums[i] != nums[end])
            {
                swap(&nums[i],&nums[start]);
                break;
            }
        }
    }
    if(nums[start] > nums[end])
        swap(&nums[start],&nums[end]);

    int privot1 = nums[start];
    int privot2 = nums[end];
    while(left+1 <= end && nums[left+1] < privot1)
        left++;
    while(right-1 >= start && nums[right-1] > privot2)
        right--;
    int k = left+1;

    while(k < right)
    {
        if(nums[k] < privot1)
        {
            left++;
            swap(&nums[left],&nums[k]);
            k++;
        }
        else if(nums[k] <= privot2)
            k++;
        else
        {
            right--;
            swap(&nums[right],&nums[k]);
        }
    }
    swap(&nums[left],&nums[start]);
    swap(&nums[right],&nums[end]);
    sort(nums,start,left-1);
    sort(nums,left+1,right-1);
    sort(nums,right+1,end);
}
int* sortArray(int* nums, int numsSize, int* returnSize){
    int start,end,k,left,right;
    *returnSize = numsSize;
    start = 0;
    end = numsSize - 1;
    sort(nums,start,end);
    return nums;
}
```

按奇偶排序数组——快排

> 给定一个非负整数数组 `A`，返回一个数组，在该数组中， `A` 的所有偶数元素之后跟着所有奇数元素。
>
> 你可以返回满足此条件的任何数组作为答案。

```
void swap(int *a,int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
```

```
}
int* sortArrayByParity(int* nums, int numsSize, int* returnSize){
    //kuaipaima
    *returnSize = numsSize;
    int i,j;
    for(i = 0,j = numsSize-1;i < j;)
    {
        if(nums[i]%2 == 0)
            i++;
        if(nums[j]%2 == 1)
            j--;
        if(i > j)
            break;
        swap(&nums[i],&nums[j]);
    }
    return nums;
}
```

荷兰国旗问题——颜色分类——快排

> 给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。
>
> 此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

```
// 你的 c 是 n 方的复杂度，建议看一下边 n 的复杂度
void sortColors(int* nums, int numsSize){
    for(int i = 1;i < numsSize;i++)
    {
        int t = nums[i];
        int j = i;
        for(;j > 0;j--)
        {
            if(t < nums[j-1])
                nums[j] = nums[j-1];
            else
                break;
        }
        nums[j] = t;
    }
}
//-------------------------------------------------------------------------------------------------------//
void swap(int *a, int *b) {
    int tp = *a;
    *a = *b;
    *b = tp;
}
void sortColors(int* nums, int numsSize){
    // 初始化左指针为第一个元素的前一个位置   右指针为最后一个元素的后一个位置
    // 这样相当于左指针所指向的及其左边的元素均为0
    //          右指针所指向的及其右边的元素均为2
    int l = -1, r = numsSize;
    // 从左到右遍历每一个元素
    for (int i = 0; i < r; ++i) {
        // 如果是 0，和左指针的下一个元素交换
        if (nums[i] == 0) {
            swap(&nums[++l], &nums[i]);
        } else if (nums[i] == 2) {
            // 如果是2，和右指针的前一个元素交换，但是交换之后需重新判断当前位置元素的情况，所以对 i--
            swap(&nums[--r], &nums[i--]);
        }
        // 如果是 1 不做处理
    }
}
```

数组中的多数元素——快排

> 给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数 大于 ⌊ n/2 ⌋ 的元素。
>
> 你可以假设数组是非空的，并且给定的数组总是存在多数元素。

```
void swap(int a[],int x,int y)
{
```

```
    int t = a[x];
    a[x] = a[y];
    a[y] = t;
}
int partition(int a[],int start,int end) //分区 找枢纽 然后left right交换数据
{
    int temp = a[start];
    int left = start + 1;
    int right = end;
    while(left < right)
    {
        while(left < right && a[left] <= temp) left++;
        while(left < right && a[right] >= temp) right--;
        if(left < right)
        {
            swap(a,left,right);
            left++;
            right--;
        }
    }
    if(right == left && a[right] > temp) right--;
    swap(a,right,start);
    return right;
}
void quicksort(int a[],int start,int end) //快排 左右排序 递归
{
    if(start >= end) return;
    int mid = partition(a,start,end);
    quicksort(a,start,mid-1);
    quicksort(a,mid+1,end);
}
int majorityElement(int* nums, int numsSize){
    quicksort(nums,0,numsSize-1);
    return nums[numsSize/2];
}
```

逆序对问题——归并

```
int merge(int a[],int com[],int low,int mid,int high,int count)
{
    for(int i = low; i <= high;i++)
        com[i] = a[i];

    int i,j,k;
    for(i = low,j = mid+1,k = low;i <= mid && j <= high;k++)
    {
        if(com[i] <= com[j])
        {
            a[k] = com[i];
            i++;
        }
        else
        {
            a[k] = com[j];
            j++;
            count += mid-i+1; //每一次后边的小放到前面就形成了前一个数组所剩元素数量的逆序对
        }
    }
    while(i <= mid) a[k++] = com[i++];
    while(j <= high) a[k++] = com[j++];
    return count;
}
int mergesort(int a[],int com[],int low,int high,int count)
{
    if(low < high)
    {
        int mid = (low+high)/2;
        count = mergesort(a,com,low,mid,count);
        count = mergesort(a,com,mid+1,high,count);
        count = merge(a,com,low,mid,high,count);
    }
```

```
    return count;
}

int reversePairs(int* nums, int numsSize){
    int count = 0;
    int *com = (int *)malloc(sizeof(int)*numsSize);
    count = mergesort(nums,com,0,numsSize-1,count);
    return count;
}
```

Top K 问题——堆排

> 找出第 k 大的数

```
void buildmaxheap(int * a,int len)
{
    for(int i = len/2;i > 0;i--)
        headadjust(a,i,len);
}

void headadjust(int * a,int k,int len){
    a[0] = a[k];
    for(int i = k*2;i <= len;i*=2)
    {
        if(i < len &&a[i] < a[i+1])
            i++;
        if(a[0] >= a[i])
            break;
        else {
            a[k] = a[i];
            k = i;
        }
    }
    a[k] = a[0];
    return;
}
int findKthLargest(int* nums, int numsSize, int k){
    int * a = (int *)malloc(sizeof(int )*(numsSize+1));
    int m = 1;
    for(int j = 0;j < numsSize;j++)
        a[m++] = nums[j];

    int len = numsSize; //len是第n个 可以取到值，所以应该等于nums 这样就alen有意义
    buildmaxheap(a,len);

    for(int i = len,m = 0;m < k-1 ;i--,m++)
    {
        a[1] = a[i];
        headadjust(a,1,i-1);
    }
    return a[1];
}
// ------------------------------------------------------------------------//
void swap(int arr[],int i,int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
int findKthLargest(int* nums, int numsSize, int k){
    int maxindex,minindex;
    int i,j;
    for(i = 0;i < numsSize/2;i++)
    {
        maxindex = i;
        minindex = i;
        for(j = i+1;j < numsSize-i;j++)
        {
            if(nums[maxindex] < nums[j])
                maxindex = j;
            if(nums[minindex] > nums[j])
```

```
                minindex = j;
        }
        if(maxindex == minindex)
            break;
        swap(nums,maxindex,i);
        if(minindex == i)
            minindex = maxindex;
        swap(nums,minindex,numsSize-1-i);
    }

    return nums[k-1];
}
```

最小的 K 个数——堆排

> 输入整数数组 `arr` ，找出其中最小的 `k` 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

```
//堆排序
void heapAdjust(int a[],int k,int len)
{
    int temp = a[k];
    for(int j = k*2 ;j <= len;j *= 2)//第2个结点-1 = 下标
    {
        if(j < len && a[j] > a[j+1])
            j++;
        if(a[j] >= temp)
            break;
        else
        {
            a[k] = a[j];
            k = j;
        }
    }
    a[k] = temp;
}
void buildheap(int a[],int len)
{
    for(int j = len/2;j > 0;j--)
        heapAdjust(a,j,len);
}
void swap(int *a,int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
int* getLeastNumbers(int* arr, int arrSize, int k, int* returnSize){
    int a[arrSize+1];
    int m = 1;
    for(int j = 0;j < arrSize;j++)
        a[m++] = arr[j];

    *returnSize = k;
    int *returns = (int *)malloc(sizeof(int) * k);
    buildheap(a,arrSize);
    for(int i = 0,x = arrSize;i < k;i++,x--)
    {
        returns[i] = a[1];
        swap(&a[1],&a[x]);
        heapAdjust(a,1,x-1);
    }
    return returns;
}
```

最大间距——基排

> 给定一个无序的数组，找出数组在排序之后，相邻元素之间最大的差值。
>
> 如果数组元素个数小于 2，则返回 0。

```
void copy(int arr[],int nums[],int len)
```

```
{
    for(int i = 0;i < len;i++)
        nums[i] = arr[i];
}
void set_0(int arr[],int len)
{
    for(int i = 0;i < len;i++)
        arr[i] = 0;
}
int maximumGap(int* nums, int numsSize){
    if(numsSize < 2) return 0;
    int cha = 0;
    int max = 0,max_length = 0;
    int count[10];
    int arr[numsSize];
    int ten = 1;
    set_0(arr,numsSize);
    for(int i = 0;i < numsSize;i++)
    {
        if(max < nums[i])
            max = nums[i];
    }
    while(max > 0)
    {
        max = max / 10;
        max_length++;
    }
    set_0(count,10);
    while(max_length--)
    {
        for(int j = 0;j < numsSize;j++)
        {
            int r = nums[j] / ten % 10;
            count[r]++;
        }
        for(int k = 1;k < 10;k++)
            count[k] += count[k-1];
        for(int m = numsSize-1;m >= 0;m--)
        {
            int r = nums[m] / ten % 10;
            arr[count[r]-1] = nums[m];
            count[r]--;
        }
        copy(arr,nums,numsSize);
        ten *= 10;
        set_0(count,10);
    }
    for(int k = 0; k < numsSize-1;k++)
    {
        if(cha < nums[k+1]-nums[k])
            cha = nums[k+1] - nums[k];
    }
    return cha;
}
```

排序杂题

相对名次

> 给你一个长度为 n 的整数数组 score ， 其中 score[i] 是第 i 位运动员在比赛中的得分。所有得分都 互不相同 。
>
> 运动员将根据得分 决定名次 ， 其中名次第 1 的运动员得分最高，名次第 2 的运动员得分第 2 高，依此类推。运动员的名次决定了他们的获奖情况：
>
> 名次第 1 的运动员获金牌 "Gold Medal" 。 名次第 2 的运动员获银牌 "Silver Medal" 。 名次第 3 的运动员获铜牌 "Bronze Medal" 。 从名次第 4 到第 n 的运动员，只能获得他们的名次编号（即，名次第 x 的运动员获得编号 "x"）。 使用长度为 n 的数组 answer 返回获奖，其中 answer[i] 是第 i 位运动员的获奖情况。

```
char ** findRelativeRanks(int* score, int scoreSize, int* returnSize){
    *returnSize = scoreSize;
    int a[scoreSize];
    for(int m = 0;m < scoreSize;m++)
        a[m] = score[m];

```

```
    char **returns = (char **)malloc(sizeof(char*)*scoreSize);
    int temp,index;
    for(int dk = scoreSize/2;dk > 0;dk /= 2)
    {
        for(int i = dk;i < scoreSize;i++)
        {
            if(score[i-dk] < score[i])
            {
                temp = score[i];
                index = i-dk;
                while(index >= 0 && temp > score[index])
                {
                    score[index+dk] = score[index];
                    index -= dk;
                }
                score[index+dk] = temp;
            }
        }
    }
    for(int k = 0;k < scoreSize;k++)
    {
        for(int m = 0;m < scoreSize;m++)
        {
            if(a[k] == score[m])
            {
                if(m == 0)
                    returns[k] = "Gold Medal";
                else if(m == 1)
                    returns[k] = "Silver Medal";
                else if(m == 2)
                    returns[k] = "Bronze Medal";
                else
                {
                    returns[k] = (char*)malloc(sizeof(char)*10);
                    sprintf(returns[k],"%d",m+1) ;
                }
                break;
            }
        }
    }
    return returns;
}
```

双指针

无重复字符的最长子串

> 给定一个字符串 `s` ，请你找出其中不含有重复字符的 **最长子串** 的长度。

```
/*
就设置两个指针：一个起始边界指针，一个遍历指针
判断当前指针指向的元素是否在 [i, j) 区间内出现过，如果出现过，向右更新 i 指针，比较当前区间长度是否可以更新答案
需要注意的是：走到最后需要判断一下 n-i 的长度，因为最后遍历指针到头，没有比较这个距离
*/
int max(int a,int b)
{
    return a > b ? a : b;
}
int lengthOfLongestSubstring(char* s) {
    int n = strlen(s);
    int i = 0, j = 0;
    bool exist[256] = {false};
    int max_len = 0;
    while(j < n){
        if(exist[s[j]]){
            max_len = max(max_len, j - i);
            while(s[i] != s[j]){
                exist[s[i]] = false;
                i++;
            }
```

```
            i++;
            j++;
        }else{
            exist[s[j]] = true;
            j++;
        }
    }
    max_len = max(max_len, n - i);
    return max_len;
}
```

二分

旋转数组的最小数字

> 把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。
>
> 给你一个可能存在 重复 元素值的数组 numbers ，它原来是一个升序排列的数组，并按上述情形进行了一次旋转。请返回旋转数组的最小元素。例如，数组 [3,4,5,1,2] 为 [1,2,3,4,5] 的一次旋转，该数组的最小值为1。

```
int minArray(int* numbers, int numbersSize){
    //生活太累了，我可以邀请你去云朵☁上打呼噜吗
    // 好呀好呀 那真是太开心啦!
    // int min_num = numbers[0];
    int low = 0;
    int high = numbersSize-1;
    int mid = (low + high)/2;
    while(low < high)
    {
        if(numbers[mid] > numbers[high])
            low = mid+1;
        else if(numbers[mid] < numbers[high])
            high = mid;
        else
            high--;
        mid = (low+high)/2;
    }
    return numbers[low];
}
```

0~n-1 中缺失的数字

> 一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围0～n-1之内。在范围0～n-1内的n个数字中有且只有一个数字不在该数组中，请找出这个数字。

```
/*
二分判断 mid 指向的元素是否和下标相等，如果相等，则缺失的元素在 [mid+1, end)，否则在[0, mid]

需要注意的是缺失的可能是第一个元素和最后一个元素
*/
int missingNumber(int* nums, int numsSize){
    int mid;
    int low = 0;
    int high = numsSize-1;
    if(nums[low] == 1)
        return low;
    if(nums[high] == high)
        return high+1;
    while(low < high)
    {
        mid = (low+high)/2;
        if(nums[mid] != mid)
            high = mid;
        else
            low = mid+1;
    }
    return low;
}
```

深度优先搜索 dfs

括号生成

> 数字 `n` 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 **有效的** 括号组合。

```c
void bracket(int left,int right,int n,char arr[],int idx,char c,char **returns,int *returnSize)
{
    if(left < right || left > n || right > n)
        return;
    if(left == n && right == n)
    {
        returns[(*returnSize)] = (char *)calloc((2*n+1),sizeof(char));
        strcpy(returns[(*returnSize)],arr);
        (*returnSize)++;
        return;
    }
    arr[idx] = '(';
    bracket(left+1,right,n,arr,idx+1,'(',returns,returnSize);
    arr[idx] = ')';
    bracket(left,right+1,n,arr,idx+1,')',returns,returnSize);
    return;
}

char ** generateParenthesis(int n, int* returnSize){
    *returnSize = 0;
    char ** returns = (char **)malloc(sizeof(char *)*1500);
    char * arr = (char *)calloc((2*n+1),sizeof(char));
    bracket(0,0,n,arr,0,'(',returns,returnSize);
    return returns;
}
```

路径总和

> 给你二叉树的根节点 root 和一个整数目标和 targetSum ， 找出所有 从根节点到叶子节点 路径总和等于给定目标和的路径。
>
> 叶子节点 是指没有子节点的节点。

```c
void digui(struct TreeNode* root,int targetSum,int arr[],int **returns,int* returnSize,int index,int **returnColumnSizes)
{
    if(root == NULL) return;
    arr[index] = root->val;
    targetSum -= root->val;

    if(targetSum == 0 && root->left == NULL && root->right == NULL)
    {
        returns[(*returnSize)] = (int *)calloc((index+1),sizeof(int ));
        for(int i = 0;i <= index;i++)
            returns[(*returnSize)][i] = arr[i];

        (*returnColumnSizes)[*returnSize] = index+1;
        (*returnSize)++;
        return;
    }
    digui(root->left,targetSum,arr,returns,returnSize,index+1,returnColumnSizes);
    digui(root->right,targetSum,arr,returns,returnSize,index+1,returnColumnSizes);
    return ;
}

int** pathSum(struct TreeNode* root, int targetSum, int* returnSize, int** returnColumnSizes){
    *returnSize = 0;
    *returnColumnSizes = (int *)malloc(sizeof(int)*5050);
    int **returns = (int **)malloc(sizeof(int *)*5050);
    int *arr = (int *)malloc(sizeof(int)*5050);
    int index = 0;
    digui(root,targetSum,arr,returns,returnSize,index,returnColumnSizes);
    return returns;
}
```

贪心

三元组最小距离

> 定义三元组 (a,b,c)（a,b,c 均为整数）的距离 D=|a−b|+|b−c|+|c−a|
>
> 给定 3 个非空整数集合 S1,S2,S3，按**升序**分别存储在 3 个数组中。
>
> 请设计一个尽可能高效的算法，计算并输出所有可能的三元组 (a,b,c)（a∈S1,b∈S2,c∈S3）中的最小距离。
>
> 例如 S1={−1,0,9},S2={−25,−10,10,11},S3={2,9,17,30,41} 则最小距离为 2，相应的三元组为 (9,10,9)。

```cpp
/*
假设构成当前三元组的三个数大小关系是：s1[i] <= s2[j] <= s3[k]；由于三个数组元素都是升序排列的。则：
    1. 如果往后移动 k，则三元组的距离会增大；
    2. 如果移动 j：
        2.1 如果 s2[j + 1] > s3[k]，则三元组距离增大
        2.2 如果 s2[j + 1] <=  s3[k]，则三元组距离不变
    3. 如果移动 i：
        3.1 如果 s1[i + 1] <= s3[k]，则距离减小
        3.2 如果 s1[i + 1] > s3[k]，则距离可能增大或者减小
因此，只有移动最小的元素，三元组的距离才可能减小。
*/
#include<bits/stdc++.h>
using namespace std;

const int N = 1e5 + 5;

int main() {
    int l, m, n;
    scanf("%d %d %d", &l, &m, &n);

    long long s1[N], s2[N], s3[N];
    for (int i = 0; i < l; ++i) scanf("%lld", &s1[i]);
    for (int i = 0; i < m; ++i) scanf("%lld", &s2[i]);
    for (int i = 0; i < n; ++i) scanf("%lld", &s3[i]);

    long long ans = LONG_LONG_MAX;
    for (int i = 0, j = 0, k = 0; i < l && j < m && k < n; ) {
        long long dis = abs(s1[i] - s2[j]) + abs(s2[j] - s3[k]) + abs(s1[i] - s3[k]);
        ans = min(ans, dis);

        if (s1[i] <= s2[j] && s1[i] <= s3[k]) i++;
        else if (s2[j] <= s1[i] && s2[j] <= s3[k]) j++;
        else if (s3[k] <= s2[j] && s3[k] <= s1[i]) k++;
    }
    cout << ans << endl;
    return 0;
}
```

动态规划

最大子数组和

> 子数组连续

```cpp
// nums[i] 表示以 i 结尾的最大子数组和
int maxSubArray(int* nums, int numsSize){
    int sum = nums[0];
    for(int i = 1;i < numsSize;i++)
    {
        if(nums[i-1] > 0)
            nums[i] += nums[i-1];
        if(nums[i] > sum)
            sum = nums[i];
    }
    return sum;
}
```

最大乘积子数组

> 给你一个整数数组 `nums` ，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

```cpp
// 小苏的代码 em 周久良和孟鹤堂 都搬上来了
int max_num(int a,int b,int c)
{
```

```
    if(a >= b && a >= c)
        return a;
    else if(b >= a && b >= c)
        return b;
    else if(c >= a && c >= b)
        return c;
    else
        return 0;
}
int min_num(int a,int b,int c)
{
    if(a <= b && a <= c)
        return a;
    else if(b <= a && b <= c)
        return b;
    else if(c <= a && c <= b)
        return c;
    else
        return 0;
}
int maxProduct(int* nums, int numsSize){
    if(numsSize == 0)
        return 0;
    // 以第 i 个元素结尾的子数组乘积最大值
    int zjl_max = nums[0];
    // 以第 i 个元素结尾的子数组乘积最小值
    int zjl_min = nums[0];
    int mht_max = nums[0];
    for(int i = 1;i < numsSize;i++)
    {
        int temp = zjl_max;
        zjl_max = max_num(zjl_max*nums[i],zjl_min*nums[i],nums[i]);
        zjl_min = min_num(temp*nums[i],zjl_min*nums[i],nums[i]);
        if(mht_max < zjl_max)
            mht_max = zjl_max;
    }
    return mht_max;
}

// dage 的优雅
int min(int a, int b) {return a > b ? b : a;}
int max(int a, int b) {return a > b ? a : b;}
int maxProduct(int* nums, int numsSize){
    int ans = nums[0];
    int dp_mi[numsSize + 5];
    int dp_ma[numsSize + 5];
    dp_ma[0] = dp_mi[0] = nums[0];
    for (int i = 1; i < numsSize; ++i) {
        dp_mi[i] = min(nums[i], min(nums[i] * dp_mi[i - 1], nums[i] * dp_ma[i - 1]));
        dp_ma[i] = max(nums[i], max(nums[i] * dp_mi[i - 1], nums[i] * dp_ma[i - 1]));
        ans = max(ans, dp_ma[i]);
    }
    return ans;
}
```

最长公共子序列

> 给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。
>
> 一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。
>
> 例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。 两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。

```
/*
dp[i][j] 表示第一个串的前 i 个元素和第二个串的前 j 个元素的最长公共子序列

状态转移一：当第一个串的第 i 个元素等于第二个串的第 j 个元素的时候，那么第一个串的前 i 个元素和第二个串的前 j 个元素的最长公共子序列的长度为第一个串的前 i-1 个元素和第二个串的前 j-1 个元素的最长公共子序列长度 +1，即 dp[i][j] = dp[i-1][j-1] + 1

状态转移二：当第一个串的第 i 个元素不等于第二个串的第 j 个元素的时候，那么第一个串的前 i 个元素和第二个串的前 j 个元素的最长公共子序列的长度为 第一个串的前 i-1 个元素和第二个串的前 j 个元素的最长公共子序列长度（dp[i-1][j]）和
   第一个串的前 i 个元素和第二个串的前 j-1 个元素的最长公共子序列长度（dp[i][j-1]）的最大值
```

```c
*/
int max(int a,int b)
{
    return a > b ? a : b;
}
int longestCommonSubsequence(char * text1, char * text2){
    int len1 = strlen(text1);
    int len2 = strlen(text2);
    int dp[1010][1010];
    int i,j;
    dp[0][0] = 0;
    for(i = 1;i <= strlen(text1);i++)
    {
        for(j = 1;j <= strlen(text2);j++)
        {
            if(text1[i-1] == text2[j-1])
                dp[i][j] = dp[i-1][j-1]+1;
            else
                dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
        }
    }
    return dp[len1][len2];
}
```

最长公共子数组（最长重复子数组）

> 给两个整数数组 `A` 和 `B` ，返回两个数组中公共的、长度最长的子数组的长度。

```c
/*
dp[i][j] 表示第一个串以第 i 个元素结尾和第二个串的以第 j 个元素结尾的最长公共子数组（当然这里你是倒序的啦，意思是一样的）

那那那 当 第 i 个元素和第 j 个元素不一样的时候，当然 dp[i][j] = 0 唠

*/
int max(int a,int b)
{
    return a > b ? a : b;
}
int findLength(int* nums1, int nums1Size, int* nums2, int nums2Size){
    int dp[nums1Size+1][nums2Size+1];
    memset(dp, 0, sizeof(dp));
    dp[nums1Size][nums2Size] = 0;
    int ans = 0;
    for(int i = nums1Size-1;i >= 0;i--)
    {
        for(int j = nums2Size-1;j >= 0;j--)
        {
            if(nums1[i] == nums2[j])
                dp[i][j] = dp[i+1][j+1]+1;
            else
                dp[i][j] = 0;
            ans = max(ans,dp[i][j]);
        }
    }
    return ans;
}
```

最长递增子序列

> 给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。
>
> 子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。
>
> **哦对，思考一个问题?**
>
> - 如果求的是最长递增子数组呢

```c
/*
dp[i] 表示以 i 结尾的递增子序列的最大长度
*/
int max(int a,int b)
{
```

```
    return a > b ? a : b;
}
int lengthOfLIS(int* nums, int numsSize){
    int max_num = 1;
    int dp[numsSize];
    // memset(dp,1,sizeof(dp));
    dp[0] = 1;
    for(int i = 1;i < numsSize;i++)
    {
        dp[i] = 1;
        for(int j = 0;j < i;j++)
        {
            // 相当于是将 第 i 个数字 加在 第 j 个数字之后，所以是 dp[j]+1，这里求的是 dp[i] 的最大值，因此是 max(dp[i], dp[j]+1)
            if(nums[j] < nums[i])
                dp[i] = max(dp[i],dp[j]+1);
        }
        // 更新结果
        max_num = max(max_num,dp[i]);
    }
    return max_num;
}
```

em 如果求的是最长递增子数组的话那么当然是 双指针啦！

01背包 *

> 有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。
>
> 第 i 件物品的体积是 vi，价值是 wi。
>
> 求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。 输出最大价值。

```
//板子板子
#include <bits/stdc++.h>
using namespace std;

const int N = 1010;

int n,m;
int f[N][N];
int v[N],w[N];

int main()
{
    cin >> n >> m;
    //小白想检验一下生成的f数组是不是全0数组
    // for(int j = 0;j < n;j++) cout<<f[j][4];
    for(int i = 1;i <= n;i++)
    {
        cin >> v[i] >> w[i];
    }
    //下面这个循环是 依次考虑 当第i个物品时，体积j依次增大，直到大于v[i]时，即当前可以装下第i个物品，即拥有了第i个物品的价值。下面进行降维优化时，j从最大体积m依次减小考虑情况。
    for(int i = 1;i <= n;i++)
    {
        for(int j = 1;j <= m;j++)
        {
            f[i][j] = f[i - 1][j];
            if(j >= v[i])
            {
                f[i][j] = max(f[i][j],f[i - 1][j - v[i]] + w[i]);
                //小白手动跑了一遍检验发现 f[n][m]已经是最大的价值 因为赋值时进行了选择
            }
        }
    }

    // int res = 0;
    // for(int i = 0;i <= m;i++)
    // {
    //     res = max(res,f[n][i]);
    // }
    cout<<f[n][m]<<endl;
```

```
    return 0;
}

// 优化
#include <bits/stdc++.h>
using namespace std;
const int N = 1010;

int n,m;
int f[N]; //这里进行降维优化
int v[N],w[N];

int main()
{
    cin >> n >> m;
    for(int i = 1;i <= n;i++)
    {
        cin >> v[i] >> w[i];
    }
    for(int i = 1;i <= n;i++)
    {
        for(int j = m;j >= v[i];j--)
        {
            //由于此时变成了一位数组 由于比较的是和它前一位f[i-1][j-v[i]]为了更好地表示发生变化，我们应选择体积从最大的体积m依次递减
            f[j] = max(f[j],f[j - v[i]] + w[i]);
        }
    }
    cout<<f[m]<<endl;
    return 0;
}

// dage 代码
#include<bits/stdc++.h>
using namespace std;

const int N = 1005;
const int V = 1005;
int n, v;
int dp[N];
// int size[N], weight[N];

/*
dp[i][j] 表示前 i 件物品背包容量为 j 的情况下可以取到的最大价值

因为最初的 01 背包的转移方程是 dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - v] + w), (j >= v) 即:
    1. dp[i - 1][j] 表示不取第 i 件物品得到的价值，即前 i-1 件物品，背包容量为 j 是可以得到的最大价值
    2. dp[i - 1][j - v] + w 表示取第 i 件物品得到的价值。即【前 i-1 件物品】在【背包容量为 j-v 】的时候【可以得到最大价值】加上【第 i 件物品的价值w】
--------------优化--------------
由于我们每次计算只需要前 i-1 件物品的情况，而不需要之前的，所以可以使用一维数组进行优化。
即: dp[i] = max(dp[i], dp[i - volume] + weight)
    dp[i] 表示【当前物品及之前物品】，在 【体积为 i】的情况下可以取得的最大值

假设当前考虑的是【第 n 件】物品，则未更新 dp 数组时【dp数组的状态】 表示的是 【前 n-1 件】物品在 【体积为 i】 时可以取到的最大价值

由上述二维数组的转移方程我们发现，我们需要用到 【前 n-1 件物品在 j-v 体积】情况下的最大价值，dp数组未更新时正好保存的就是前 n-1 件物品的情况，如果正向更新 dp 数组，dp[j - v] 将会在 dp[j] 之前被更新，无法计算 dp[j]，因此需要倒序更新 dp 数组
*/
// 封装的 01 背包，v 是背包的总体积，volume 是当前物品的体积，weight 是当前物体的价值
// 01 背包倒叙遍历
void zero_one_package(int volume, int weight) {
    for (int i = v; i >= volume; --i) {
        dp[i] = max(dp[i], dp[i - volume] + weight);
    }
}

int main() {
    cin >> n >> v;
    memset(dp, 0, sizeof dp);
    int volume, weight;
    // 获取 n 个物品的体积和价值
    for (int i = 0; i < n; ++i) {
```

```
        cin >> volume >> weight;
        zero_one_package(volume, weight);
    }
    cout << dp[v] << endl;
    return 0;
}
```

参考链接

完全背包 *

> 有 N 件物品和一个容量是 V 的背包。每件物品可以使用无限次。
>
> 第 i 件物品的体积是 vi，价值是 wi。
>
> 求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。 输出最大价值。

```
#include<bits/stdc++.h>
using namespace std;

const int N = 1005;
const int V = 1005;
int n, v;
int dp[N];

// 封装的 完全 背包，v 是背包的总体积，volume 是当前物品的体积，weight 是当前物体的价值
// 完全 背包正序遍历
/*
转移方程：dp[i] = max(dp[i], dp[i - volume] + weight)
假设当前考虑的是【第 n 件物品】，完全背包【不仅】考虑的是【前 n-1 件物品】的情况，他还需要考虑【假设已经选择了 k 件第 n 件物品】的情况。（因为完全背包物品的选择是没有数量限制的）。而如果 dp 数组正向更新，则 dp[i] 之前正好是 【考虑了前 n-1 件物品和 k 件第 n 件物品情况的叠加】
*/
void complete_package(int volume, int weight) {
    for (int i = volume; i <= v; ++i) {
        dp[i] = max(dp[i], dp[i - volume] + weight);
    }
}

int main() {
    cin >> n >> v;
    int volume, weight;
    memset(dp, 0, sizeof dp);
    for (int i = 0; i < n; ++i) {
        cin >> volume >> weight;
        complete_package(volume, weight);
    }
    cout << dp[v] << endl;
    return 0;
}
```

零钱兑换——完全背包变体 *

> 给你一个整数数组 coins ， 表示不同面额的硬币；以及一个整数 amount ， 表示总金额。
>
> 计算并返回可以凑成总金额所需的 最少的硬币个数 。如果没有任何一种硬币组合能组成总金额，返回 -1 。
>
> 你可以认为每种硬币的数量是无限的。

```
// 一看无限次就是完全背包【指遍历顺序是从前往后】
/*
但是这里要的是凑成 总金额 的最小硬币个数。即相当于必须把背包装满。所以最终的状态都需要从 0 转移过来。
什么意思呢?
假如上边的裸背包有如下情形:
    背包大小是 20，只有一件物品的体积是 3，价值为 2，则最大的价值为 18*2 = 36。
    有  dp[20] = dp[17] + 2;
        dp[17] = dp[14] + 2;
        ...
        dp[5] = dp[2] + 2; 这样的状态转移过程
但假如 amount = 20，只有一种钱币 3，则无法凑成 20，
    因为要凑成 20 需要凑成 17，要凑成 17 需要 14，最终需要凑成 2，但是只有 3 的面额，所有无法凑成 2

这说明了一个什么问题呢?
```

```
    1. 即对于不要求装满的那种情况下，我最终的情况可以由 dp[0]~dp[20] 任意一种情况转移过来。比如只有体积为 19 的物品，则 dp[20] = dp[1] + w，即 dp[20] 由初始的 dp[1] 状态转移过来
    2. 而对于凑钱的问题（或者说要求正好装满的情况下）则不行，你必须最终由 dp[0] 转移过来。也就是说，dp[1]~dp[20] 不能作为最初始的合法状态。
*/
int min(int a, int b) { return a > b ? b : a; }

int coinChange(int* coins, int coinsSize, int amount){
    int dp[amount + 5];
    // 所有的状态都需要从 0 转移而来，所有其它需要初始化为最大（因为要最小钱币数）
    memset(dp, 0x3f, sizeof dp);
    dp[0] = 0;
    for (int i = 0; i < coinsSize; ++i) {
        for (int j = coins[i]; j <= amount; ++j) {
            dp[j] = min(dp[j], dp[j - coins[i]] + 1);
        }
    }
    return dp[amount] > amount ? -1 : dp[amount];
}
```

零钱兑换II ——完全背包变体 *

> 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
>
> 请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0 。
>
> 假设每一种面额的硬币有无限个。
>
> 题目数据保证结果符合 32 位带符号整数。

```
/*
这个是求方案数，最终状态同样只能由 0 状态转移过来，dp[0] 等于 1，表示组合成 0 的方案有 1 种
dp[j] 表示组合成 j 面额的方案数
假设现在是第 n 种面额的硬币，计算 dp[j] 的时候，dp[j] 保存的是前 n-1 种金额的硬币组合成 j 的方案数。dp[j - coins[i]] 表示的是使用 k（某个具体的值） 个第 n 种硬币和前 n-1 种面额的硬币组合成 j-coins[n] 的方案数，两者相加即为 dp[j] = dp[j] + dp[j - coins[i]];
*/
int change(int amount, int* coins, int coinsSize){
    int dp[amount + 5];
    // 所有的状态都需要从 0 转移而来，最初组合成 0 的方案为 1，其它的方案为 0
    memset(dp, 0, sizeof dp);
    dp[0] = 1;
    for (int i = 0; i < coinsSize; ++i) {
        for (int j = coins[i]; j <= amount; ++j) {
            dp[j] += dp[j - coins[i]];
        }
    }
    return dp[amount];
}
```

杂题

回文数

> 这解法牛！！！

```
bool isPalindrome(int x){
    // 特殊情况：
    // 如上所述，当 x < 0 时，x 不是回文数。
    // 同样地，如果数字的最后一位是 0，为了使该数字为回文，
    // 则其第一位数字也应该是 0
    // 只有 0 满足这一属性
    if (x < 0 || (x % 10 == 0 && x != 0))
        return false;

    int revertedNumber = 0;
    while (x > revertedNumber) {
        revertedNumber = revertedNumber * 10 + x % 10;
        x /= 10;
    }

    // 当数字长度为奇数时，我们可以通过 revertedNumber/10 去除处于中位的数字。
    // 例如，当输入为 12321 时，在 while 循环的末尾我们可以得到 x = 12，revertedNumber = 123，
    // 由于处于中位的数字不影响回文（它总是与自己相等），所以我们可以简单地将其去除。
```

```
    return x == revertedNumber || x == revertedNumber / 10;
}
```

字符串压缩——模拟

> 给你一个字符数组 chars ， 请使用下述算法压缩:
>
> 从一个空字符串 s 开始。对于 chars 中的每组 连续重复字符 ：
>
> 如果这一组长度为 1 ， 则将字符追加到 s 中。 否则，需要向 s 追加字符，后跟这一组的长度。 压缩后得到的字符串 s 不应该直接返回 ， 需要转储到字符数组 chars 中。需要注意的是，如果组长度为 10 或 10 以上，则在 chars 数组中会被拆分为多个字符。
>
> 请在 修改完输入数组后 ， 返回该数组的新长度。

```
int compress(char* chars, int charsSize){
    int index = 0;
    int i,j;
    for(i = 0;i < charsSize;)
    {
        j = i+1;
        // 要注意先判断是否越界呀小苏 小脑袋瓜 记住！记住！记住！！！
        while(j < charsSize && chars[j] == chars[i])
            j++;

        if(j-i > 1)
        {
            chars[index++] = chars[i];
            int d = j-i;
            int d_index = 0;
            int arr[5];
            while(d > 0)
            {
                arr[d_index] = d%10;
                d = d/10;
                d_index++;
            }
            while(d_index--)
            {
                chars[index++] = arr[d_index]+'0';
            }
        }
        else if(j-i == 1)
            chars[index++] = chars[i];

        if(j == charsSize)
            break;
        i = j;
    }
    charsSize = index;
    return charsSize;
}
```