编程40题答案.md

1. 略

2. 略

3.

```
#include <stdio.h>
const int MOD = 100007;
int main() {
        int N;
        int fib1 = 1, fib2 = 1, fib = 1;
        int i;
        scanf("%d", &N);
        for (i = 3; i <= N; ++i) {
                fib = (fib1 + fib2) % MOD;
                fib1 = fib2;
                fib2 = fib;
        }
        printf("%d", fib);
        return 0;
}
```

设走上第N级台阶的走法数为f(N)。我们知道，走上第N级台阶只有两种可能，要么是从第N−1级台阶跨1级，要么是从第N−2级台阶跨2级。所以得出：`f(N)=f(N−1)+f(N−2),N≥3`

可以看出，这一公式正是斐波那契数列的形式，可以轻易写出。

但需要注意一点，1≤N≤1000，当N稍大时，f(N)会超出int所能表示的范围，而这也正是题目中要求输出走法数目对100007取余后的结果的原因。所以我们不能直接求出f(N)后，再对100007取余；而应当在迭代的过程中就做取余的运算。而取余具有如下的性质：`(a+b) % c =((a % c)+(b % c)) % c`

4.

```
#include <stdio.h>
const int MOD = 100007;
int main() {
        int N, K;
        int i, j;
        scanf("%d%d", &N, &K);
        N--;
        int fib[1001] = {1, 1};
        for (i = 1; i <= K; ++i)
                fib[i] = 1;
        for (i = 2; i <= N; ++i) {
                for (j = i - 1; j >= i - K && j > 0; --j)
                        fib[i] = (fib[i] + fib[j]) % MOD;
        }
        printf("%d", fib[N]);
        return 0;
}
```

此题类似与第3题，但又是对第3题的扩展。同样，设走上第N级台阶的走法数为f(N)。我们知道，走上第N级台阶只有K种可能：从第N−1级台阶跨1级，从第N−2级台阶跨2级，…，从第N−K级台阶跨K级。那么很容易得出： $f(N)=\Sigma f(N-i),1<=i<=K$

5.

```
#include <stdio.h>
int main() {
        int n;
        int i;
        int res = 2;
        scanf("%d", &n);
        for (i = 2; i <= n; ++i)
                res = 2 * res + 2;
        printf("%d", res);
        return 0;
}
```

对于这题，如果理解了汉诺塔的递归思想，应当很容易解出。

设f(n)为2n个原盘从A柱移动到C柱所需的最小移动次数。这个过程可分为三步：首先，将A柱上面的2(n−1)个原盘从A柱移动到B柱，这一步所需的最小移动次数为f(n−1)；然后，将A柱下面剩余的2个原盘依次移动到C柱，这一步所需要的最小移动次数为2；最后，将B柱的2(n−1)个原盘从B柱移动到C柱，这一步所需的最小移动次数为f(n−1)。故，可推出： $f(n)=2f(n-1)+2$

6.

```
#include <stdio.h>
int moveN_HanoiFromSrcToDest(const char* src, const char* mid,
                             const char* dest, int n) {
        int num = 0;
        if (n == 0)
                return 0;
        num += moveN_HanoiFromSrcToDest(src, mid, dest, n - 1);
        printf("Move %d from %s to %s\n", n, src, mid);
        ++num;
        num += moveN_HanoiFromSrcToDest(dest, mid, src, n - 1);
        printf("Move %d from %s to %s\n", n, mid, dest);
        ++num;
        num += moveN_HanoiFromSrcToDest(src, mid, dest, n - 1);
        return num;
}
int main() {
        int n;
        scanf("%d", &n);
        printf("%d", moveN_HanoiFromSrcToDest("A", "B", "C", n));
        return 0;
}
```

对于这题，如果理解了汉诺塔的递归思想，应当很容易解出。

设f(n)为n个原盘从A柱移动到C柱所需的步骤数。想将n个原盘从A柱移动到C柱，这个过程可分为5步：首先，将A柱上面的n−1个原盘从A柱移动到C柱，这一步所需的移动步数为f(n−1)；然后，将A柱最下面的原盘从A柱移动到B柱，这一步所需的移动步数为1；然后，将C柱的n−1个原盘从C柱移动到A柱，这一步所需的移动步数为f(n−1)；然后，将B柱的1个原盘从B柱移动到C柱，这一步所需的移动步数为1；最后，将A柱的n−1个原盘从A柱移动到C柱，这一步所需的移动步数为f(n−1)。

函数moveN_HanoiFromSrcToDest接受四个参数，分别是src，mid，dest和n，表示将n个原盘从名为src的柱子移动到名为dest的柱子，而名为mid的柱子，则是移动过程中可供临时暂存原盘的柱子。该函数的功能是，输出n个原盘从名为src的柱子移动到名为dest的柱子的步骤，并返回移动的步骤数。

我在移动的过程中顺便求了所需的步骤数。也可以根据递推公式，直接算出步骤数： `f(n)=3f(n-1)+2`

7.

```
int BinomialCoefficient(int n, int k) {
        if (k > n / 2)
                k = n - k;
        if (k == 0 || n == 1)
                return 1;
        if (k == n)
                return 1;
        return BinomialCoefficient(n - 1, k - 1) +
               BinomialCoefficient(n - 1, k);
}
```

在宏观上把握递归的思想，会更容易理解递归。

8.

```
int is_sorted(int arr[], int left, int right) {
        for ( ; left < right; ++left)
                if (arr[left] > arr[left + 1])
                        return -1; // not sorted
        return 0; // sorted
}
```

注意是非降序

9.

```
void swap(int arr[], int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
}//交换数组中两个元素的值
void BubbleSort(int arr[], int left, int right) {
        int i, j;
        for (i = 1; i <= right - left; ++i) {
                for (j = left; j <= right - i; ++j) {
                        if (arr[j] > arr[j + 1])
                                swap(arr, j, j + 1);
                }
        }
}
```

10.

递归版本

```
int binary_search_recursive(int arr[], int left, int right, int query) {
        int mid;
        if (left > right)
```

```
                return -1;
        mid = left + (right - left) / 2;
        if (arr[mid] == query)
                return mid;
        if (arr[mid] > query)
                return binary_search_recursive(arr, left, mid - 1, query);
        return binary_search_recursive(arr, mid + 1, right, query);
}
```

迭代版本

```
int binary_search_iterative(int arr[], int left, int right, int query) {
        int mid;
        while (left <= right) {
                mid = left + (right - left) / 2;
                if (arr[mid] == query)
                        return mid;
                if (arr[mid] > query)
                        right = mid - 1;
                else
                        left = mid + 1;
        }
        return -1;
}
```

需注意递归版本的递归终止条件，循环版本的循环判断条件。另外，在对mid的计算时，我使用的是：`mid = left + (right - left) / 2;` 而非：`mid = (left + right) / 2;` 这样做是为了避免left + right的整型溢出风险。另外，须注意，这种二分查找的实现仅适用于当数组arr是严格升序时的情形。

11.

```
void InsertionSort(int arr[], int left, int right) {
        int i, j, temp;
        for (i = left + 1; i <= right; ++i) {
                temp = arr[i];
                for (j = i - 1; j >= left && temp < arr[j]; --j)
                        arr[j + 1] = arr[j];
                arr[j + 1] = temp;
        }
}
```

注意，这种插入排序的实现，在寻找插入位置时是线性查找。所以，我们也可以使用二分查找的方法来寻找插入位置，这是对插入排序的一个优化方式。

12.

```
void SelectionSort(int arr[], int left, int right) {
        int i, j, minIdx;
        for (i = left; i < right; ++i) {
                minIdx = i;
                for (j = i + 1; j <= right; ++j)
                        if (arr[j] < arr[minIdx])
                                minIdx = j;
                swap(arr, i, minIdx);
        }
}
```

swap同第9题。

13.

```
void k_reverse(char* str, int k) {
        int len = strlen(str);
        int i, left, right;
        char temp;
        for (i = 0; i + k <= len; i += k) {
                for (left = i, right = left + k - 1;
                        left < right; ++left, --right) {
                        temp = str[left];
                        str[left] = str[right];
                        str[right] = temp;
                }
        }
}
```

最后不足 k 个的字符不用反转

14.

```
#include <stdio.h>
#include <string.h>
#define N 100000
void add(int num1[], int len1, int num2[], int len2, int sum[], int* len) {
        int i = 0, mod = 0, temp;
        while (i < len1 && i < len2) {
                temp = num1[i] + num2[i] + mod;
                sum[i++] = temp % 10;
                mod = temp / 10;
        }
        while (i < len1) {
                temp = num1[i] + mod;
                sum[i++] = temp % 10;
                mod = temp / 10;
        }
        while (i < len2) {
                temp = num2[i] + mod;
                sum[i++] = temp % 10;
                mod = temp / 10;
        }
        if (mod > 0)
                sum[i++] = mod;
        *len = i;
}
void to_arr(const char* str, int arr[], int len) {
        int i;
        for (i = 0, --len; len >= 0; ++i, --len)
                arr[i] = str[len] - '0';
}
int main() {
        char str1[N], str2[N];
        int num1[N], num2[N], sum[N];
        int len1, len2, len;
        scanf("%s%s", str1, str2);
        len1 = strlen(str1);
        len2 = strlen(str2);
```

```
        to_arr(str1, num1, len1);
        to_arr(str2, num2, len2);
        add(num1, len1, num2, len2, sum, &len);
        return 0;
}
```

15.

```
#include <stdio.h>
#define N 25
int main() {
        int spiral_matrix[N][N];
        int n, i;
        int row = 1, col = 1, val = 1;
        scanf("%d", &n);
        for (i = 1; i <= (n + 1) / 2; ++i) {
                while (col <= n - i + 1)
                        spiral_matrix[row][col++] = val++;
                --col;
                while (row <= n - i)
                        spiral_matrix[++row][col] = val++;
                while (col > i)
                        spiral_matrix[row][--col] = val++;
                while (row > i + 1)
                        spiral_matrix[--row][col] = val++;
                ++col;
        }
        return 0;
}
```

for循环的每一次迭代填充矩阵的一整圈,其中,i用来控制圈数。

16. 略

17.

```
int my_strlen(const char* str) {
        if (str == NULL)
                return 0;
        int len;
        for (len = 0; str[len]; ++len);
        return len;
}
```

18.

```
void my_strcpy(char* dest, const char* src) {
        assert(dest != NULL);
        assert(src != NULL);
        while (*src)
                *dest++ = *src++;
        *dest = 0;
}
```

19.

```
int my_strcmp(const char* str1, const char* str2) {
        assert(str1 != NULL);
        assert(str2 != NULL);
        while (*str1 && *str2) {
                if (*str1 == *str2) {
                        ++str1;
                        ++str2;
                } else
                        return *str1 > *str2 ? 1 : -1;
        }
        return *str1 ? 1: (*str2 ? -1 : 0);
}
```

20.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct MS {
        char name[11];
        char num[11];
        int grade;
};
int main() {

        int n=0;
        int i=0;
        int min=0,max=0;
        int mindex=0;
        int maxdex=0;
        struct MS *stu;
        scanf("%d",&n);
        stu=(struct MS *)malloc(n*sizeof(struct MS));
        for(i=0; i<n; i++) {
                scanf("%s %s %d",stu[i].name,stu[i].num,&stu[i].grade);
        }
        min=stu[0].grade;
        max=stu[0].grade;
        for(i=0; i<n; i++) {
                if(stu[i].grade<min) {
                        min=stu[i].grade;
                        mindex=i;
                }
                if(stu[i].grade>max) {
                        max=stu[i].grade;
                        maxdex=i;
                }
        }
        printf("%s %s\n",stu[maxdex].name,stu[maxdex].num);
        printf("%s %s\n",stu[mindex].name,stu[mindex].num);
        return 0;
}
```

21.

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
        FILE* fp;
        int ch;
        if (argc != 2) {
                printf("Error in format. Usage: show file\n");
                return 1;
        }
        if ((fp = fopen(argv[1], "r")) == NULL) {
                printf("The file <%s> can not be opened.\n", argv[1]);
                return 2;
        }
        ch = fgetc(fp);
        while (ch != EOF) {
                putchar(ch);
                ch = fgetc(fp);
        }
        fclose(fp);
        return 0;
}
```

22.

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
        int ch;
        FILE* srcfp, *destfp;
        if (argc != 3) {
                printf("Error in format. Usage: copy source destination\n");
                return 1;
        }
        if ((srcfp = fopen(argv[1], "r")) == NULL) {
                printf("The file <%s> can not be opened.\n", argv[1]);
                return 2;
        }
        if ((destfp = fopen(argv[2], "w")) == NULL) {
                printf("The file <%s> can not be opened.\n", argv[2]);
                return 2;
        }
        ch = fgetc(srcfp);
        while (ch != EOF) {
                fputc(ch, destfp);
                ch = fgetc(srcfp);
        }
        fclose(srcfp);
        fclose(destfp);
        return 0;
}
```

23.

```c
#include <stdio.h>
long long sum(long long i) {
        long long cnt;
        if (i & 1)
                cnt = (i + 1) >> 1;
        else {
```

```
                cnt = i >> 1;
                --i;
        }
        return (1 + i) * cnt / 2;
}
int main() {
        int n;
        scanf("%d", &n);
        long long res = 0;
        while (n) {
                res += sum(n);
                n >>= 1;
        }
        printf("%lld", res);
        return 0;
}
```

f(x)为x最大的奇数约数，则显然，x为奇数时，f(x)=x；x为偶数时，x最大的两个约数分别为x与x2，故此时，f(x)=x2。那么，

$$f(1)+f(2)+\cdots+f(N)=1+f(1)+3+f(2)+5+f(3)+\cdots$$

可以把上式分为两部分，一部分是奇数值构成的等差数列，这部分可以直接利用求和公式计算出来；另一部分是偶数折半后的最大奇约数，重复同样的过程即可。

代码中的函数sum是完成一次上述的过程。

24.

```
int bin_insert(int n, int m, int j, int i) {
        m <<= j;
        n |= m;
        return n;
}
```

25.

```
int sum(int n) {
        n && (n += sum(n - 1));
        return n;
}
```

这里用到了短路求值

26.

```
int add(int n1, int n2) {
        if (n2 > 0) {
                for (int i = 1; i <= n2; ++i)
                        ++n1;
        } else if (n2 < 0) {
                for (int i = -1; i >= n2; --i)
                        --n1;
        }
```

```
        return n1;
}
```

27.

```
int find(const char* str, const char* substr) {
        int i, j, k;
        for (i = 0; str[i]; ++i) {
                for (k = i, j = 0; str[k] && substr[j]
                        && str[k] == substr[j]; ++k, ++j);
                if (!substr[j])
                        return i;
        }
        return -1;
}
```

原题中函数名为substr，与该函数的第二个形参命名相同，虽未发生命名冲突，但这其实不是一个很好的习惯。我在出题时疏忽了，特此将该函数名更正find。即在主串str中查找字串substr，并返回其第一次出现的位置；若为查找到，则返回-1。

这是一道字符串匹配题。我在此处用的是普通的匹配方式，即试探所有的情况。此题另有更好的做法，叫做KMP算法。

28.

```
int count_substr(const char* str, const char* substr) {
        int i, j, k;
        int cnt = 0;
        for (i = 0; str[i]; ) {
                for (k = i, j = 0; str[k] && substr[j]
                        && str[k] == substr[j]; ++k, ++j);
                if (!substr[j]) {
                        ++cnt;
                        i = k;
                } else
                        ++i;
        }
        return cnt;
}
```

可以使用更好的KMP算法

29.

```
void Merge(int arr[], int left, int mid, int right) {
        int size = right - left + 1;
        int* temparr = (int*)malloc(size * sizeof(int));
        int i = left, j = mid + 1, k = 0;
        while (i <= mid && j <= right) {
                if (arr[i] <= arr[j])
                        temparr[k++] = arr[i++];
                else
                        temparr[k++] = arr[j++];
        }
        while (i <= mid)
                temparr[k++] = arr[i++];
```

```
        while (j <= right)
                temparr[k++] = arr[j++];
        memcpy(arr + left, temparr, size * sizeof(int));
        free(temparr);
}
void MergeSort(int arr[], int left, int right) {
        int mid = left + (right - left) / 2;
        if (left >= right)
                return;
        MergeSort(arr, left, mid);
        MergeSort(arr, mid + 1, right);
        Merge(arr, left, mid, right);
}
```

函数MergeSort如题所述；函数Merge实现将两个有序（非降序）序列归并为一个有序（非降序）序列，其中arr[left .. mid]与arr[mid + 1 .. right]均已有序（非降序）。

我用到了malloc函数进行动态内存分配。而有些同学在此处，可能会写为：

```
int temparr[right – left + 1];
```

虽然这个写法在使用gcc编译器进行编译时不会出现编译错误，甚至在运行时也几乎不会出现运行时错误，但我建议还是慎用这种写法。出于以下两点理由：

其一，这个特性叫变长数组。我们知道，C语言在定义数组时，数组的长度应当是一个常量，或者说，编译期常量，即，在编译期就已经确定了其值，也就是说，在编译完成后，编译器须要知道某一数组的具体大小，这样才能在程序运行时为函数栈分配确定的空间。不过，变长数组是C99的一个新特性，也就是说，C语言标准支持这种写法。然而，未必所有的C语言编译器都实现了这个特性，我们目前所使用的gcc的确是支持变长数组，但换一个编译器，就可能出现编译错误。

其中，C99标准是ISO/IEC 9899:1999 - Programming languages -- C的简称，是C语言的官方标准第二版，与1999年发布。你可以了解有关C99的内容。

其二，这种写法存在一个问题，就是，当该变长数组的长度过大时，在这个例子中，即，当right – left + 1过大时，可能会超出一个栈的最大可分配空间，这样就造成了运行时错误。

所以，慎用变长数组这一特性。使用动态内存分配是一个更安全更妥善的方法。当然，在使用动态分配的内存完毕后，应当使用free来释放内存，否则就造成了内存泄漏。你可以了解有关内存泄露 的内容。

另外，我用到了memcpy函数，实现将数组temparr中的内容拷贝到数组arr中。它在语义逻辑上等价于：

```
for (i = left, k = 0; i <= right; ++i)
        arr[i++] = temparr[k++];
```

memcpy的使用方法可以参考百度百科。

30. 略

31. 略

32.

```
struct Node* construct(int arr[], int size) {
        int i;
```

```
        struct Node head;
        struct Node *prev, *p;
        head.next = NULL;
        prev = &head;
        for (i = 0; i < size; ++i) {
                p = (struct Node* )malloc(sizeof(struct Node));
                p->val = arr[i];
                p->next = NULL;
                prev->next = p;
                prev = p;
        }
        return head.next;
}
```

33.

```
struct Node* insert_to_head(struct Node* head, int val) {
        struct Node* newhead = (struct Node* )malloc(sizeof(struct Node));
        newhead->val = val;
        newhead->next = head;
        return newhead;
}
```

34.

```
struct Node* insert_to_tail(struct Node* head, int val) {
        struct Node* tail = (struct Node* )malloc(sizeof(struct Node));
        struct Node* p;
        tail->val = val;
        tail->next = NULL;
        if (head == NULL)
                return tail;
        for (p = head; p->next; p = p->next);
        p->next = tail;
        return head;
}
```

35.

```
struct Node* insert(struct Node* head, int val) {
        struct Node* node = (struct Node* )malloc(sizeof(struct Node));
        struct Node* prev = NULL;
        struct Node* p = head;
        node->val = val;
        node->next = NULL;
        while (p && p->val <= val) {
                prev = p;
                p = p->next;
        }
        if (prev == NULL) {
                node->next = head;
                return node;
        }
        prev->next = node;
        node->next = p;
```

```
        return head;
}
```

36.

```
struct Node* delete_node(struct Node* head, struct Node* target) {
        struct Node* prev = NULL;
        struct Node* p = head;
        struct Node* post;
        while (p) {
                post = p->next;
                if (p == target) {
                        if (prev == NULL) {
                                free(p);
                                return post;
                        }
                        prev->next = post;
                        free(p);
                        return head;
                }
                prev = p;
                p = post;
        }
        return head;
}
```

37.

```
struct Node* delete_val(struct Node* head, int val) {
        struct Node* prev = NULL;
        struct Node* p = head;
        struct Node* post;
        while (p) {
                post = p->next;
                if (p->val == val) {
                        if (prev == NULL) {
                                head = post;
                                free(p);
                        } else {
                                prev->next = post;
                                free(p);
                        }
                        p = post;
                } else {
                        prev = p;
                        p = post;
                }
        }
        return head;
}
```

38.

```
struct Node* reverse(struct Node* head) {
        struct Node* prev = NULL;
```

```
        struct Node* p = head;
        struct Node* post;
        while (p) {
                post = p->next;
                p->next = prev;
                prev = p;
                p = post;
        }
        return prev;
}
```

39.

```
struct Node* merge(struct Node* head1, struct Node* head2) {
        struct Node *p1 = head1, *p2 = head2;
        struct Node head;
        struct Node *prev, *p;
        head.next = NULL;
        prev = &head;
        while (p1 && p2) {
                p = (struct Node* )malloc(sizeof(struct Node));
                p->next = NULL;
                if (p1->val <= p2->val) {
                        p->val = p1->val;
                        p1 = p1->next;
                } else {
                        p->val = p2->val;
                        p2 = p2->next;
                }
                prev->next = p;
                prev = p;
        }
        while (p1) {
                p = (struct Node* )malloc(sizeof(struct Node));
                p->next = NULL;
                p->val = p1->val;
                p1 = p1->next;
                prev->next = p;
                prev = p;
        }
        while (p2) {
                p = (struct Node* )malloc(sizeof(struct Node));
                p->next = NULL;
                p->val = p2->val;
                p2 = p2->next;
                prev->next = p;
                prev = p;
        }
        return head.next;
}
```

40.

```
void clear(struct Node* head) {
        struct Node* p;
        while (head) {
                p = head;
                head = head->next;
```

```
                free(p);
            }
        }
```