

1.

解答: 算法完成了求解 $\sum_{i=1}^n i^2$ 的功能, 算法的基本语句是 $S=S+i*i$; 基本语句执行了 n 次, 时间复杂度为 $O(n)$ 。

2.

解答: 通过分析双重循环的循环次数, 可以得到基本语句执行次数为 $n(n+1)$, 算式如下:

$$\sum_{i=1}^n 2i = n(n+1)$$

3. 求解下面的递归式子

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/3) + n, & n > 1 \end{cases}$$

解答: 该递归式满足 Master Theorem 的第三种情况。

对递归式 $2T(n/3) + n$, 有 $a = 2, b = 3, f(n) = n, n^{\log_b a} = n^{\log_3 2} = O(n^{\log 0.631})$ 。因为 $f(n) = \Omega(n^{\log_3 2 + \varepsilon})$, 其中 $\varepsilon \approx 0.37$ 。验证递归式是否满足第三种情况: 对足够大的 n , 有 $af\left(\frac{n}{b}\right) = 2\left(\frac{n}{3}\right) \leq cf(n)$, 其中 $c = \frac{2}{3}$, 因此递归式的解为 $T(n) = \Theta(f(n)) = \Theta(n)$ 。

4. 有一猴子第 1 天摘下若干桃子, 当即吃了一半, 又多吃了 m 个。第 2 天又将剩下的桃子吃掉一半, 并多吃了 m 个, 以后每天都吃了前一天剩下的一半后又多吃 m 个, 到第 n 天再想吃时, 见只剩下 d 个桃子, 问第一天共摘了多少个桃子?

解答: 通过递归求解。设 $Q(m, n, d)$ 代表第 n 天要吃的时候剩余 d 个桃子, 则 $Q(m, n-1, 2*(m+d))$ 成立, 意思是第 $n-1$ 天要吃的时候剩余 $2(m+d)$ 个桃子, 由此建立递归, 求出 $n=1$ 时剩余的桃子, 即第一天摘得桃子个数。由于递归了 n 次, 时间复杂度为 $O(n)$ 。

代码:

```
#include<iostream>
using namespace std;

int solve(int m,int n,int d){
    if(n==1) return d;
    else return solve(m,n-1,2*(d+m));
}

int main(){
    int m,n,d;
    cin>>m>>n>>d;
    cout<<solve(m,n,d)<<endl;
    return 0;
}
```

运行结果:

```
1 2 4
10
```

分析:

设猴子每天吃一半，又多了 1 个，第二天要吃的时候剩余 4 个，则猴子第一天一共摘了 10 个桃子。

5. 递归求解双递归摆动数列。已知递归数列:

$a[1]=1$ $a[2i]=a[i]+1$ $a[2i+1]=a[i]+a[i+1]$

i 为正整数，试建立递归，求该数列的第 n 项与前 n 项的和。

解答: 如题目建立递归即可求出任意 $a[i]$ ，同时用 $\text{sum}(i) = a(i) + \text{sum}(i-1)$ 建立递归即可求出前 n 项的和。

代码:

```
// 求解双递归摆动数列
#include <iostream>
using namespace std;

int a(int i) {
    if (i == 1) return 1;
    if (i % 2)
        return a(i / 2) + a(i / 2 + 1); // 奇数情况
    else
        return a(i / 2) + 1; // 偶数情况
}

int sum(int i){
    if(i==1) return a(i);
    return a(i)+sum(i-1);
}

int main() {
    int i;
    cin >> i;
    cout << a(i) << " " << sum(i) << endl;
    return 0;
}
```

运行结果 1:

```
3
3 6
```

6. 用单向链表表示十进制数，求两个整数的和。

解答：

用链表将输入的数存储下来，之后将链表倒序，再逐位求和，最后将结果链表倒序，就得到答案。

代码：

// 用链表实现整数求和

// 输入->倒序->求和->倒序->输出

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <string>
using namespace std;

// Data Structure of Node
struct Node {
    int num;
    Node *next;
    Node() {
        num = 0;
        next = NULL;
    }
    Node(int _num) {
        num = _num;
        next = NULL;
    }
    bool setNext(Node *next_node) {
        if (next_node != NULL) {
            next = next_node;
            return true;
        }
        else
            return false;
    }
};

// Data Structure of Linked List
// #-----头结点不实际存储数据-----#
struct LinkedList {
    Node *head;
    int length = 0;
    LinkedList() { head = new Node(); }
    ~LinkedList() {
```

```

    Node *p, *temp;
    p = head;
    while (p != NULL) {
        temp = p;
        p = p->next;
        delete (temp);
    }
}
// 找到最后一个元素
Node *findLastNode() {
    Node *p = head;
    while (p->next != NULL) {
        p = p->next;
    }
    return p;
}
// 添加一个数
void addNum(int num) {
    Node *node = new Node(num);
    Node *last = findLastNode();
    last->next = node;
    length++;
}
// 逆序链表
void reverse() {
    if (head->next != NULL) {
        head->next = sub_reverse(head->next);
    }
}
// 辅助函数，将以 head 为头结点的链表倒转
// 返回值为逆转后的头结点
Node *sub_reverse(Node *head) {
    if (head->next == NULL) return head;

    Node *new_head = sub_reverse(head->next);
    head->next->setNext(head);
    head->next = NULL;
    return new_head;
}
// 打印链表
void printList() {
    Node *p;
    p = head->next;
    while (p != NULL) {

```

```

        cout << p->num;
        p = p->next;
    }
}
// 深度复制链表
LinkedList *makeCopy() {
    LinkedList *a = new LinkedList();
    Node *p1, *p2;
    p1 = head->next;
    p2 = a->head;
    while (p1 != NULL) {
        Node *node = new Node(p1->num);
        p2->next = node;
        p2 = node;
        p1 = p1->next;
    }
    return a;
}
};

// 链表大数求和，返回结果链表的指针
LinkedList *linkedSum(LinkedList *a, LinkedList *b) {
    if (a->length < b->length) return linkedSum(b, a); //保证较长的在前面
    LinkedList *c = a->makeCopy();
    Node *p1, *p2; // p1 遍历 c, p2 遍历 b
    a->reverse();
    b->reverse();
    c->reverse();
    p1 = c->head->next;
    p2 = b->head->next;
    while (p2 != NULL) {

        int promote = (p1->num + p2->num) / 10;
        if (p1->next == NULL) {
            if (promote > 0) {
                c->addNum(0);
                p1->next->num = p1->next->num + promote;
            }
        }
        else {
            p1->next->num = p1->next->num + promote;
        }
        p1->num = (p1->num + p2->num) % 10;
        p1 = p1->next;
    }
}

```

```

        p2 = p2->next;
    }
    a->reverse();
    b->reverse();
    c->reverse();

    return c;
}

string num1, num2;

int main() {
    cin >> num1 >> num2;
    LinkedList *a = new LinkedList();
    LinkedList *b = new LinkedList();
    LinkedList *c = new LinkedList();

    // 将字符串转化为数字
    for (int i = 0; i < num1.size(); i++) {
        a->addNum(num1[i] - '0');
    }
    for (int i = 0; i < num2.size(); i++) {
        b->addNum(num2[i] - '0');
    }

    c = linkedSum(a, b);
    c->printList();
    cout << endl;
    return 0;
}

```

运行结果 1:

```

123456 654321
777777

```

运行结果 2:

```

325 888
1213

```

7. 删除数组 $r[n]$ 中的重复元素，要求移动元素的次数尽可能少，且相对次序保持不变

解答: 利用双重循环查找并标记重复元素（这里也可以借助 `map` 降低时间复杂度，但是会增加空间开销）。标记删除元素之后，使用两个指针 `i` 和 `j`，令 `j` 指向从前向后数第一个被删除的元素位置，`i` 指向 `j` 之后第一个未被删除的元素，之后进行元素交换与状态切换，重复至数组中所有元素处理完毕。

代码:

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <utility>
#define N 1000
using namespace std;

int r[N] = { 1, 1, 2, 3, 1, 4, 1, 5, 2, 6, 7, 8, 9 };
int length = 13;
int new_length;          // 删除重复元素之后的长度
bool isDel[N] = { 0 }; // 标记元素是否被删除, 0 代表未被删除
void deleteRepetition() {
    memset(isDel, 0, sizeof(isDel));
    // 发现重复元素, 并标记删除
    new_length = length;
    for (int i = 0; i < length; i++) {
        if (!isDel[i]) {
            for (int j = i+1; j < length; j++) {
                if (r[i] == r[j]) {
                    isDel[j] = 1;
                    new_length--;
                }
            }
        }
    }
    // 进行元素移动, 并记录元素移动次数
    int count = 0;
    int i = 0, j = 0;
    for (i = 0; i < length; i++) {
        // 让 j 停留在第一个被删除位置
        if (isDel[j] == 0) {
            j++;
        }
        // 让 i 停留在 j 之后的第一个未被删除位置
        else if (isDel[i] == 0) {
            // 切换元素和删除状态
            r[j] = r[i];
            isDel[i] = 1;
            isDel[j] = 0;
            j++;
            count++;
        }
    }
}
```

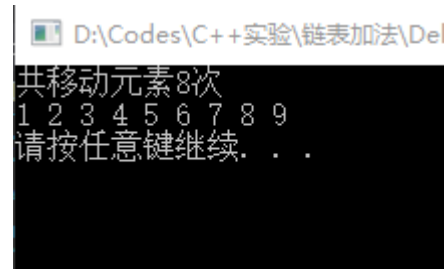
```

        cout << "共移动元素" << count << "次" << endl;
    }

    int main() {
        deleteRepetition();
        for (int i = 0; i < new_length; i++) {
            cout << r[i] << " ";
        }
        cout << endl;
        system("pause");
        return 0;
    }

```

运行结果:



```

D:\Codes\C++实验\链表加法\Del
共移动元素8次
1 2 3 4 5 6 7 8 9
请按任意键继续. . .

```

8. 拆分表 A 而不利用额外空间

解答: 利用两个指针 i 和 j , i 从前向后遍历寻找 <0 的元素, j 从后向前遍历寻找 ≥ 0 的元素, i 和 j 都找到目标之后进行元素交换, 最终将表 A 划分成表 B 和表 C。索引 0 至 $i-1$ 为表 B, 索引 i 至 N 为表 C。

代码:

```

#include <iostream>
#include <utility>
using namespace std;

#define N 10
int A[10] = {2, -1, 3, 4, -5, -6, 5, 7, 9, 0};

int main() {
    // 打印 A 表
    cout << "List A: ";
    for (int k = 0; k < N; k++) {
        cout << A[k] << " ";
    }
    cout << endl;
    // 拆分 A 表
    int i = 0, j = N - 1;

```



```

while (i < j) {
    if (A[i] >= 0) {
        i++;
    } else if (A[j] < 0) {
        j--;
    } else {
        swap(A[i], A[j]);
    }
}
cout << "List B: ";
for (int k = 0; k < i; k++) {
    cout << A[k] << " ";
}
cout << "\nList C: ";
for (int k = i; k < N; k++) {
    cout << A[k] << " ";
}
}

```

运行结果:

```

List A: 2 -1 3 4 -5 -6 5 7 9 0
List B: 2 0 3 4 9 7 5
List C: -6 -5 -1

```

9. 判断两个单词是否是同位词:

解答: 将单词排序后逐个字符对比即可。

代码:

```

// 同位词判别
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;

char a[1000], b[1000];
void solve() {
    cin >> a >> b;
    if (strlen(a) != strlen(b)) {
        cout << "No" << endl;
        return;
    }
    sort(a, a + strlen(a));
    sort(b, b + strlen(b));
    for (int i = 0; i < strlen(a); i++) {
        if (a[i] != b[i]) {

```

```
        cout << "No" << endl;  
        return;  
    }  
}  
cout << "Yes" << endl;  
}
```

```
int main() {  
    solve();  
    return 0;  
}
```

运行结果 1:

```
eat tea  
Yes
```

运行结果 2:

```
offset static  
No
```