数据与算法知识与方法

T$^{\mathrm{T}}$T

2024 年 10 月 16 日

目录

0 绪论：数据、数学模型、算法

0.1 数据及其结构

定义 0.1.1. 数据的基本单元称为**数据元素**。

数据元素并不是孤立存在的，而是存在密切的联系。

数据的逻辑结构：

- **集合结构**：数据元素之间没有任何关系
- **线性结构**：数据元素之间存在一对一的关系，如线性表、栈、队列、串
- **树形结构**：数据元素之间存在一对多的关系，如树
- **图形结构**：数据元素之间存在多对多的关系，如图

数据的存储结构：

- **顺序存储结构**：将数据元素存放在地址连续的存储单元里
- **链式存储结构**：将数据元素存放在任意的存储单元里，通过指针相连

0.2 数学模型

定义 0.2.1. **数学模型**是对于现实世界的某一特定对象，根据其内在规律，为特定目的而得到的一个抽象的、简化的数学结构。

常用的数学模型的类型：

- **线性方程组**：超定线性方程组的线性拟合、欠定线性方程组的线性规划
- **非线性方程组**：非线性方程组的求解
- **微分方程**：常微分方程的数值解法、偏微分方程的数值解法
- **概率模型**：概率预测、概率统计
- **统计模型**：统计预测、统计分析
- **离散模型**：线性结构、树结构、图结构
- **优化模型**：线性规划、整数规划、非线性规划

0.3 算法

定义 0.3.1. **算法**是用以解决某一问题的有限长度的指令序列。

算法的基本特点：

- **有穷性**：算法必须在执行有穷步之后能够结束，且每一步都可在有穷时间内完成
- **确定性**：算法的每一步必须有确切的含义，算法的实际执行结果是确定的、且精确地符合要求或期望
- **可行性**：算法中描述的操作都可以通过已经实现的基本操作运算的有限次执行来实现
- **输入**：算法必须有零个或多个输入
- **输出**：算法必须有一个或多个输出

好的算法应该具有以下特点：

- **正确性**：算法应该能够解决问题
  - 不含语法错误

– 对一般的输入数据能够产生正确的输出结果
– 对精心选择的苛刻数据也能产生正确的输出结果
– 对于所有的输入数据都能产生正确的输出结果

- **可读性**：算法应该容易理解
- **健壮性**：算法应该能够处理各种异常情况
- **高效性**：算法应该能够在合理的时间和空间开销内解决问题

一个特定算法的运行工作量的大小，与问题规模的大小有关，这种关系称为算法的复杂度。

定义 0.3.2. 渐进时间复杂度

算法的**（渐进）时间复杂度**是指算法的运行时间与问题规模之间的关系。

例 0.3.1. 现有程序如下：

```
for(int i = 0; i < n; i++)
    for(int j = i; j < n; j++)
        if(a[i] > a[j])
            swap(a[i], a[j]);
```

其中，基本操作 swap 函数的时间复杂度为 $O(1)$，其执行次数为 $\sum\limits_{i=0}^{n-1}\sum\limits_{j=i}^{n-1}1=\sum\limits_{i=0}^{n-1}(n-i)=\sum\limits_{i=0}^{n-1}i=\dfrac{n(n-1)}{2}$，则该程序的时间复杂度为 $O\left(\dfrac{n(n-1)}{2}\right)=O(n^2)$。

例 0.3.2. 现有程序如下：

```
for(int i = 1; i < n; i++)
    for(int j = 0; j < i; j *= 2)
        a[i] += a[j];
```

其内层循环的时间复杂度为 $O(\log_2 i)$，则该程序的时间复杂度为

$$\begin{aligned}\sum_{i=2}^{n-1}O(\log_2 i)=O\left(\sum_{i=2}^{n-1}\log_2 i\right)=O\left(\log_2\left(\prod_{i=2}^{n-1}i\right)\right)\\=O\left(\log_2(n-1)!\right)=O\left(\log_2 n!\right)=O(n\log_2 n)\end{aligned}$$

定义 0.3.3. 渐进空间复杂度

算法的**（渐进）空间复杂度**是指算法的空间开销与问题规模之间的关系。

1 数据结构

1.1 抽象数据类型

定义 1.1.1. 数据类型

数据类型是元素的集合 D、集合中元素的关系 R 和定义在此集合上的对这些元素的操作的集合 C 的总称。

数据类型是一种封装好的数据结构，把用户无须了解的实现细节封装起来，只提供刻画外在特性的接口。对于高级语言，除开语言本身定义好的**原子类型**外，用户还可以自定义**结构类型**，即**抽象数据类型（ADT）**。

例 1.1.1. 复数的抽象数据类型定义如下：

```
ADT Complex {
    数据对象: D = {e1, e2 | e1, e2 ∈ ℝ}
    数据关系: R = {⟨e1, e2⟩ | e1 = 𝔢(D), e2 = 𝔪(D)}
    基本操作:
        InitComplex(&Z, v1, v2)
            操作结果: 构造复数 Z，其实部和虚部分别被赋以参数 v1 和 v2 的值。
        DestroyComplex(&Z)
            初始条件: 复数 Z 存在;
            操作结果: 复数 Z 被销毁。
        Add(z1, z2, &sum)
            初始条件: z1, z2 是复数。
            操作结果: 用 sum 返回两个复数 z1, z2 的和值。
        Sub(z1, z2, &sub)
            初始条件: z1, z2 是复数。
            操作结果: 用 sub 返回两个复数 z1, z2 的差值。
} ADT Complex
```

例 1.1.2. 圆柱体的抽象数据类型定义如下：

```
ADT CYLinder {
    数据对象: D = {r, h | r, h ∈ ℝ}
    数据关系: R = {⟨r, h⟩ | r 为圆柱底面半径，h 为圆柱高}
    基本操作:
        InitCyld(r, h)
            操作结果: 构造圆柱体，底面半径 r，圆柱高 h。
        BaseArea(r, &bArea)
            初始条件: 圆柱体存在。
            操作结果: 计算圆柱体底面积，用 bArea 返回。
        SideArea(r, h, &sArea)
            初始条件: 圆柱体存在。
            操作结果: 计算圆柱体侧面积，用 sArea 返回。
        Volume(r, h, &vol)
            初始条件: 圆柱体存在。
            操作结果: 计算圆柱体体积，用 vol 返回。
} ADT CYLinder
```

1.2 线性结构

1.2.1 线性表

定义 1.2.1. 线性表

线性表是一种「有序」结构，即在数据元素的非空有限集合中，

- 存在唯一的一个被称为「**第一个**」的数据元素，无前驱；
- 存在唯一的一个被称为「**最后一个**」的数据元素，无后继；
- 除第一个之外，每个数据元素均只有一个直接**前驱**；
- 除最后一个之外，每个数据元素均只有一个直接**后继**。

对呈现这样结构的数据，可以记为 $L=(a_0,\ a_1,\ a_2,\ \cdots,\ a_{i-1},\ a_i,\ a_{i+1},\ \cdots,\ a_n)$，并称

- a_i 为**第 i 个**数据元素，i 为数据元素 a_i 的**位序**；
- a_{i-1} 为 a_i 的**直接前驱**；
- a_{i+1} 为 a_i 的**直接后继**；
- $n+1$，即线性表中数据元素的个数，为表的**长度**。

线性表的抽象数据类型可以定义为：

```
ADT List {
    数据对象: D = {a_0, a_1, a_2, ···, a_n | a_i ∈ ElemType, 1 ≤ i ≤ n}
    数据关系: R = {⟨a_i, a_{i+1}⟩ | a_i为第 i 个数据元素, a_{i+1}为第 i+1 个数据元素}
    基本操作:
        InitList(&L)
            操作结果: 构造一个空的线性表 L。
        DestroyList(&L)
            初始条件: 线性表 L 存在。
            操作结果: 销毁线性表 L。
        IsEmpty(L)
            初始条件: 线性表 L 存在。
            操作结果: 若 L 为空表，则返回 true，否则返回 false。
        ListLength(L)
            初始条件: 线性表 L 存在。
            操作结果: 返回 L 中数据元素的个数。
        GetElem(L, i, &e)
            初始条件: 线性表 L 存在, 0 <= i < ListLength(L)。
            操作结果: 用 e 返回 L 中第 i 个数据元素的值。
        LocateElem(L, e, compare())
            初始条件: 线性表 L 存在, compare() 是数据元素判定函数。
            操作结果: 返回 L 中第一个与 e 满足 compare() 的数据元素的位序; 若不存在, 则返
                回 -1。
        PriorElem(L, cur_e, &pre_e)
            初始条件: 线性表 L 存在。
            操作结果: 若 cur_e 是 L 的数据元素, 且不是第一个, 则用 pre_e 返回它的直接前
                驱; 否则操作失败。
        NextElem(L, cur_e, &next_e)
            初始条件: 线性表 L 存在。
            操作结果: 若 cur_e 是 L 的数据元素, 且不是最后一个, 则用 next_e 返回它的直接
                后继; 否则操作失败。
        ClearList(&L)
            初始条件: 线性表 L 存在。
```

```
            操作结果：将 L 重置为空表。
        ListInsert(&L, i, e)
            初始条件：线性表 L 存在，0 <= i <= ListLength(L)。
            操作结果：在 L 的第 i 个位置（第 i 个元素之前）插入新的数据元素 e，L 的长度加
                1。
        ListDelete(&L, i, &e)
            初始条件：线性表 L 存在，0 <= i < ListLength(L)。
            操作结果：删除 L 的第 i 个数据元素，并用 e 返回其值，L 的长度减 1。
        ListTraverse(L, visit())
            初始条件：线性表 L 存在，visit() 是对数据元素操作的函数。
            操作结果：依次对 L 的每个数据元素调用 visit()，一旦 visit() 失败，则操作失
                败。
} ADT List
```

A) 线性表的实现 按照存储结构的不同，线性表可以分为顺序表和链表。

- **顺序表**：线性表的存储结构为顺序存储结构，即用一组地址连续的存储单元依次存放线性表中的元素。

可以用 C++ 语言实现一个顺序表：

```cpp
class List {
private:
    ElemType *elem;
    int length, maxSize;
public:
    List(int size) : maxSize(size)
        { elem = new ElemType[maxSize]; length = 0; }
    ~List()                          { delete[] elem; }
    bool isEmpty() const             { return length == 0; }
    int listLength() const           { return length; }
    ElemType getElem(int i) const    { return elem[i]; }
    int locateElem(ElemType e) const {
        for (int i = 0; i < length; i++)
            if (elem[i] == e) return i;
        return -1;
    }
    ElemType priorElem(ElemType e) const {
        int i = locateElem(e);
        if (i > 0) return elem[i - 1];
        return -1;
    }
    ElemType nextElem(ElemType e) const {
        int i = locateElem(e);
        if (i >= 0 && i < length - 1) return elem[i + 1];
        return -1;
    }
    void clearList()                 { length = 0; }
    bool listInsert(int i, ElemType e) {
        if (i < 0 || i > length || length == maxSize) return false;
        for (int j = length; j > i; j--)
            elem[j] = elem[j - 1];
        elem[i] = e;
        length++;
        return true;
```

```
    }
    ElemType listDelete(int i) {
        if (i < 0 || i >= length) return -1;
        ElemType e = elem[i];
        for (int j = i; j < length - 1; j++)
            elem[j] = elem[j + 1];
        length--; return e;
    }
    void listTraverse(void (*visit)(ElemType)) const {
        for (int i = 0; i < length; i++)
            visit(elem[i]);
    }
};
```

1.2.2 栈

定义 1.2.2. 栈

栈是一种特殊的线性表，其插入和删除操作只能在表的同一端进行，这一端被称为**栈顶**，另一端被称为**栈底**。

栈的抽象数据类型可以定义为：

```
ADT Stack {
    数据对象: D = {a_0, a_1, a_2, ···, a_n | a_i ∈ ElemType, 1 ≤ i ≤ n}
    数据关系: R = {⟨a_i, a_{i+1}⟩ | a_i为第 i 个数据元素, a_{i+1}为第 i+1 个数据元素}
    基本操作:
        InitStack(&S)
            操作结果: 构造一个空栈 S。
        DestroyStack(&S)
            初始条件: 栈 S 存在。
            操作结果: 销毁栈 S。
        IsEmpty(S)
            初始条件: 栈 S 存在。
            操作结果: 若 S 为空栈, 则返回 true, 否则返回 false。
        GetTop(S, &e)
            初始条件: 栈 S 存在, S 非空。
            操作结果: 用 e 返回 S 的栈顶元素。
        Push(&S, e)
            初始条件: 栈 S 存在。
            操作结果: 将元素 e 压入栈顶。
        Pop(&S, &e)
            初始条件: 栈 S 存在, S 非空。
            操作结果: 弹出栈顶元, 素并用 e 返回其值。
        ClearStack(&S)
            初始条件: 栈 S 存在。
            操作结果: 将 S 重置为空栈。
        StackLength(S)
            初始条件: 栈 S 存在。
            操作结果: 返回 S 的元素个数。
} ADT Stack
```

A) 栈的实现 按照存储结构的不同，栈可以分为顺序栈和链式栈。

- **顺序栈**：栈的存储结构为顺序存储结构，即用一组地址连续的存储单元依次存放栈中的元素。

  可以用 C++ 语言实现一个顺序栈：

```cpp
class Stack {
private:
    ElemType *bottom;
    int stackDepth, maxSize;
public:
    Stack(int size) : maxSize(size)
        { bottom = new ElemType[maxSize]; stackDepth = 0; }
    ~Stack()                        { delete[] bottom; }
    bool isEmpty() const            { return stackDepth == 0; }
    ElemType getTop() const         { return bottom[stackDepth - 1]; }
    int stackLength() const         { return stackDepth; }
    void push(ElemType e)           { bottom[stackDepth++] = e; }
    ElemType pop()                  { return bottom[--stackDepth]; }
    void clearStack()               { stackDepth = 0; }
};
```

  在初始化顺序栈时，需要指定栈的最大容量 maxSize，如果栈中元素已经达到最大容量，继续压栈会导致**栈溢出**。在具体实现中，根据需要可以具体设计如何处理栈溢出的情况。

- **链式栈**：栈的存储结构为链式存储结构，即用链表存放栈中的元素。

  可以用 C++ 语言实现一个链式栈：

```cpp
struct Node {
    ElemType data;
    Node *next;
};
typedef struct Node* Link;
class Stack {
private:
    Link head;
public:
    Stack()                 { head = nullptr; }
    ~Stack()                { clearStack(); }
    bool isEmpty() const    { return head == nullptr; }
    ElemType getTop() const { return head->data; }
    int stackLength() const
        { int len = 0; for (Link p = head; p; p = p->next) len++; return len; }
    void push(ElemType e)
        { Link p = new Node; p->data = e; p->next = head; head = p; }
    ElemType pop()
        { Link p = head; head = head->next; ElemType e = p->data; delete p; return
            e; }
    void clearStack()       { while (!isEmpty()) pop(); }
};
```

  链式栈的实现中，栈顶元素位于链表的头部，每次压栈时，新元素插入到链表的头部；每次弹栈时，栈顶元素从链表的头部删除。

例 1.2.1. 后缀表达式是一种不含括号的表达式，其操作符位于操作数后面。例如，后缀表达式 $34+5*$ 对应的

中缀表达式为 $(3+4)*5$。在计算机处理中，后缀表达式的计算更加方便。用栈实现四则运算后缀表达式的计算如下：

```
double calcPostfix(const char *postfix) {
    strstream ss(postfix);                  // 用字符串流读取后缀表达式
    Stack values;
    int curValue = 0; char curChar = '\0';
    while (ss >> curChar) {                 // 逐个读取字符
        if (isdigit(curChar)) {             // 对数字字，符转换为数字并入记录数值
            curValue = curValue * 10 + curChar - '0';
        } else {                            // 对操作，符弹栈计算
            int preValue = values.pop();
            switch (curChar) {
            case '+': values.push(preValue + curValue); break;
            case '-': values.push(preValue - curValue); break;
            case '*': values.push(preValue * curValue); break;
            case '/': values.push(preValue / curValue); break;
            }
            curValue = 0;                   // 重置记录数值
        }
    }
    if (values.stackLength() != 1) throw ValueException("Invalid postfix expression!");
    return values.pop();
}
```

B) 栈与递归 递归是一种常见的算法设计方法，递归函数的调用过程可以看作是一个栈的过程。递归函数的调用过程中，每次调用函数时，都会将当前函数的局部变量、返回地址等信息压入栈中，当递归函数返回时，会将这些信息弹出。

例 1.2.2. 计算斐波那契数列的第 n 项的递归函数如下：

```
int fibonacci(int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

在计算斐波那契数列的第 n 项时，递归函数 `fibonacci` 会不断调用自身，直到递归到 `n = 0` 或 `n = 1` 时返回，这个过程可以看作是一个栈的过程。

递归算法可能导致较高的时间复杂度和空间复杂度，因此有时需要考虑使用非递归的方法来实现，即**递归的消除**。以下两类递归函数可以比较容易地消除：

- **尾递归**：如果一个递归函数中只有唯一一个递归调用语句，并且递归调用语句在递归算法的最后，称为尾递归。尾递归可以通过循环来实现，从而避免栈的过多压入和弹出。
- **单向递归**：如果一个递归函数中虽然有多处递归调用语句，但各递归调用语句的参数之间无关，并且递归调用语句都在递归算法的最后。单向递归可以通过设置变量保存中间结果，从而用循环结构代替递归。

例 1.2.3. 例 1.2.2 中的斐波那契数列的递归函数是一个单向递归函数，消除递归的算法设计为：

```java
int fibonacci(int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;
    int pre = 0, cur = 1, next;
    for (int i = 2; i <= n; i++) {
        next = pre + cur;
        pre = cur; cur = next;
    }
    return next;
}
```

1.2.3 队列

定义 1.2.3. 队列

队列是一种特殊的线性表，其插入操作只能在表的一端进行，这一端被称为**队头**；删除操作只能在表的另一端进行，称为**队尾**。

队列的抽象数据类型可以定义为：

```
ADT Queue {
    数据对象: D = {a_0, a_1, a_2, ···, a_n | a_i ∈ ElemType, 1 ≤ i ≤ n}
    数据关系: R = {⟨a_i, a_{i+1}⟩ | a_i为第 i 个数据元素, a_{i+1}为第 i+1 个数据元素}
              约定: 队头为 , a_0队尾为 a_n。
    基本操作:
        InitQueue(&Q)
            操作结果: 构造一个空队列 Q。
        DestroyQueue(&Q)
            初始条件: 队列 Q 存在。
            操作结果: 销毁队列 Q。
        IsEmpty(Q)
            初始条件: 队列 Q 存在。
            操作结果: 若 Q 为空队列, 则返回 true, 否则返回 false。
        QueueLength(Q)
            初始条件: 队列 Q 存在。
            操作结果: 返回 Q 的元素个数。
        GetHead(Q, &e)
            初始条件: 队列 Q 存在, Q 非空。
            操作结果: 用 e 返回 Q 的队头元素。
        EnQueue(&Q, e)
            初始条件: 队列 Q 存在。
            操作结果: 将元素 e 入队列 Q 的队尾。
        DeQueue(&Q, &e)
            初始条件: 队列 Q 存在, Q 非空。
            操作结果: 将 Q 的队头元素出队, 列并用 e 返回其值。
        ClearQueue(&Q)
            初始条件: 队列 Q 存在。
            操作结果: 将 Q 重置为空队列。
        QueueTraverse(Q, visit())
            初始条件: 队列 Q 存在, visit() 是对数据元素操作的函数。
```

```
            操作结果：依次对 Q 的每个数据元素调用 visit()，一旦 visit() 失败，则操作失
                败。
} ADT Queue
```

A) **队列的实现** 按照存储结构的不同，队列可以分为顺序队列和链式队列。

- **顺序队列**：队列的存储结构为顺序存储结构，即用一组地址连续的存储单元依次存放队列中的元素。为了避免队列整体移动，通常采用**循环队列**的方式实现顺序队列。

  可以用 C++ 语言实现一个顺序队列：

```cpp
class Queue {
private:
    ElemType *base;
    int front, rear, maxSize;
public:
    Queue(int size) : maxSize(size)
        { base = new ElemType[maxSize]; front = rear = 0; }
    ~Queue()                        { delete[] base; }
    bool isEmpty() const            { return front == rear; }
    ElemType getHead() const        { return base[front]; }
    int queueLength() const         { return (rear - front + maxSize) % maxSize; }
    void enQueue(ElemType e)
        { base[rear] = e; rear = (rear + 1) % maxSize; }
    ElemType deQueue()
        { ElemType e = base[front]; front = (front + 1) % maxSize; return e; }
    void clearQueue()               { front = rear = 0; }
};
```

- **链式队列**：队列的存储结构为链式存储结构，即用链表存放队列中的元素。链式队列是只在头尾两端进行插入和删除操作的更简单的链表。

  可以用 C++ 语言实现一个链式队列：

```cpp
struct Node {
    ElemType data;
    Node *next;
};
typedef struct Node* Link;
class Queue {
private:
    Link front, rear;
public:
    Queue()                         { front = rear = new Node; front->next =
        nullptr; }
    ~Queue()                        { clearQueue(); delete front; }
    bool isEmpty() const            { return front == rear; }
    ElemType getHead() const        { return front->next->data; }
    int queueLength() const
        { int len = 0; for (Link p = front->next; p; p = p->next) len++; return len
            ; }
    void enQueue(ElemType e)
        { Link p = new Node; p->data = e; p->next = nullptr; rear->next = p; rear =
            p; }
```

```
    ElemType deQueue()
        { Link p = front->next; front->next = p->next; ElemType e = p->data; delete
            p; return e; }
    void clearQueue()               { while (!isEmpty()) deQueue(); }
};
```

B) 优先级队列 给队列中的每一个元素都赋予一个优先级，优先级高的元素先出队列，这种队列称为**优先级队列**。

优先级队列的抽象数据类型可以定义为：

```
ADT PQueue {
    数据对象: D = {a_0, a_1, a_2, ···, a_n | a_i ∈ map<int, ElemType>, 1 ≤ i ≤ n}
    数据关系: R = {⟨a_i, a_{i+1}⟩ | a_i为第 i 个数据元素, a_{i+1}为第 i+1 个数据元素}
              约定: 队头为 a_0, 队尾为 a_n。
    基本操作:
        InitPQueue(&Q)
            操作结果: 构造一个空优先级队列 Q。
        DestroyPQueue(&Q)
            初始条件: 优先级队列 Q 存在。
            操作结果: 销毁优先级队列 Q。
        IsEmpty(Q)
            初始条件: 优先级队列 Q 存在。
            操作结果: 若 Q 为空队列, 则返回 true, 否则返回 false。
        PQueueLength(Q)
            初始条件: 优先级队列 Q 存在。
            操作结果: 返回 Q 的元素个数。
        GetHead(Q, &e)
            初始条件: 优先级队列 Q 存在, Q 非空。
            操作结果: 用 e 返回 Q 中优先级最高的元素。
        EnPQueue(&Q, e)
            初始条件: 优先级队列 Q 存在。
            操作结果: 将元素 e 插入优先级队列 Q。
        ChangePQueue(&Q, ei, ef)
            初始条件: 优先级队列 Q 存在, Q 非空, ei 是 Q 中的元素。
            操作结果: 将 Q 中元素 ei 的优先级改为 ef。
        DePQueue(&Q, &e)
            初始条件: 优先级队列 Q 存在, Q 非空。
            操作结果: 将 Q 中优先级最高的元素出优先级队, 列并用 e 返回其值。
        ClearPQueue(&Q)
            初始条件: 优先级队列 Q 存在。
            操作结果: 将 Q 重置为空优先级队列。
        PQueueTraverse(Q, visit())
            初始条件: 优先级队列 Q 存在, visit() 是对数据元素操作的函数。
            操作结果: 依次对 Q 的每个数据元素调用 visit(), 一旦 visit() 失败, 则操作失
                败。
} ADT PQueue
```

优先级队列可以基于有序或无序的顺序表或链表实现。有序的优先级队列可以在插入时按照优先级顺序插入，插入操作的时间复杂度为 $O(n)$；而无序的优先级队列则需要在出队列时遍历所有元素找到优先级最高的元素，出队列操作的时间复杂度为 $O(n)$。各种实现方式中，入队、修改、出队操作的时间复杂度如表 1 所示。

表 1：优先级队列的时间复杂度

	操作	有序顺序表	无序顺序表	有序链表	无序链表
入队	EnPQueue(&Q, e)	$O(n)$	$O(1)$	$O(n)$	$O(1)$
修改	ChangePQueue(&Q, ei, ef)	$O(n)$	$O(1)$	$O(n)$	$O(1)$
出队	DePQueue(&Q, &e)	$O(n)$	$O(n)$	$O(1)$	$O(n)$

若优先级队列需要频繁进行入队操作，可以考虑使用无序顺序表或无序链表实现；若优先级队列需要频繁进行出队操作，可以考虑使用有序链表实现。

注 1.1. 优先级队列与栈、队列的联系

优先级队列是一种特殊的队列，其出队操作是根据元素的优先级进行的。同时，优先级队列可以看作是栈和队列的推广，若设定元素在队中的停留时间为其优先级，那么栈就是时间越短优先级越高的优先级队列，队列就是时间越长优先级越高的优先级队列。

1.2.4 串

定义 1.2.4. 串

串是由零个或多个字符组成的有限序列，又称**字符串**。串中的字符数目称为串的**长度**；长度为零的串称为**空串**，记作 $\emptyset$。两个串**相等**当且仅当它们的长度相等，且对应位置的字符都相同。

串中任意多个连续字符组成的子序列称为该串的**子串**，子串**首字符**在原串中的位置称为子串的**位置**。

串的抽象数据类型可以定义为：

```
ADT String {
    数据对象：D = {a_0, a_1, a_2, ···, a_n | a_i ∈ char, 1 ≤ i ≤ n}
    数据关系：R = {⟨a_i, a_{i+1}⟩ | a_i为第 i 个字符，a_{i+1}为第 i+1 个字符}
    基本操作：
        StrAssign(&S, chars)
            初始条件：chars 是字符串常量。
            操作结果：生成一个其值等于 chars 的串 S。
        StrCopy(&S, T)
            初始条件：串 T 存在。
            操作结果：由串 T 复制得到串 S。
        StrEmpty(S)
            初始条件：串 S 存在。
            操作结果：若 S 为空串，则返回 true，否则返回 false。
        StrLength(S)
            初始条件：串 S 存在。
            操作结果：返回 S 的长度。
        StrCompare(S, T)
            初始条件：串 S 和 T 存在。
            操作结果：若 S > T，则返回正数；若 S = T，则返回 0；若 S < T，则返回负数。
        Concat(&S, T1, T2)
            初始条件：串 T1 和 T2 存在。
            操作结果：用 S 返回由 T1 和 T2 连接而成的新串。
        SubStr(&S, T, pos, len)
```

```
            初始条件：串 T 存在，1 <= pos <= StrLength(T)，0 <= len <= StrLength(T) - pos
                + 1。
            操作结果：用 S 返回串 T 的第 pos 个字符起长度为 len 的子串。
        Index(S, T, pos)
            初始条件：串 S 和 T 存在，1 <= pos <= StrLength(S)。
            操作结果：若 S 中存在和 T 值相同的子串，则返回 S 中从第 pos 个字符起第一个这
                样的子串的位置，否则返回 0。
        Replace(&S, T, V)
            初始条件：串 S、T 和 V 存在，T 非空。
            操作结果：用 V 替换 S 中出现的所有与 T 相等的不重叠的子串。
        StrTraverse(S, visit())
            初始条件：串 S 存在，visit() 是对数据元素操作的函数。
            操作结果：依次对 S 的每个数据元素调用 visit()，一旦 visit() 失败，则操作失
                败。
} ADT String
```

对串操作的最小子集包括串赋值 StrAssign()、串复制 StrCopy()、串求长 StrLength()、串比较 StrCompare()、串连接 Concat()、求子串 SubStr() 操作。

A) 串的实现 串的存储和一般的线性表类似，可以用顺序存储结构或链式存储结构实现。

- **串的顺序存储**：串的顺序存储结构是用一组地址连续的存储单元依次存放串中的字符。根据其长度规定，又可以分为：
  - **定长顺序存储**，即串的长度是固定的，不足部分用特殊字符填充；
  - **变长顺序存储**，即串的长度是动态分配的，可以根据需要动态调整，串的结束标志是特殊字符。
- **串的链式存储**：串的链式存储结构是用链表存放串中的字符，每个结点存放一定数量的字符，一个结点存储的字符个数称为**结点大小**。

B) 串的模式匹配 给定一个文本串 T 和一个模式串 P，模式匹配是指在文本串 T 中查找模式串 P 的过程。若 T 中存在一个子串与 P 完全相同，则称模式串 P 在文本串 T 中匹配成功。

蛮力穷举是一种简单的模式匹配算法，其基本思想是：

- 从文本串 T 的第一个字符开始，依次与模式串 P 的第一个字符开始比较；
- 若不匹配，则文本串向后移动一位，模式串从头开始比较；
- 若匹配，则继续比较下一个字符，直到模式串匹配成功或文本串遍历完毕。

蛮力穷举算法的时间复杂度为 $O(m \cdot n)$，其中 m 为模式串 P 的长度，n 为文本串 T 的长度。C++ 中 std::string 类的 find 函数即是蛮力穷举算法的实现。

算法 1.1. KMP 算法

思想 在模式串 P 与文本串 T 的匹配过程中，当遇到不匹配的字符时，根据已经匹配的信息，尽量减少不必要的比较次数。

步骤 (1) **构造模式串的部分匹配表**：对于模式串 P，求取一个部分匹配表 next，其中 next[i] 表示模式串 $P[0:i]$ 的最长相同前后缀的长度。

(2) **匹配文本串**：在文本串 T 中匹配模式串 P，若遇到不匹配的字符，则根据部分匹配表 next 调整模式串的位置。

算法的关键是构造模式串部分匹配表 next 的函数 Next()，其 C++ 实现如下：

```
void Next(char *P, int next[]) {
    int m = strlen(P), i = 1, j = 0;
    next[0] = 0;
    while (i < m) {
        if (P[i] == P[j]) next[i++] = ++j;      // 相同字符，next[i] = j + 1
        else if (j > 0) j = next[j - 1];        // 不同字符，回溯到前一个字符
        else next[i++] = 0;                     // 不同字符且无法回溯，next[i] = 0
    }
}
```

这样，将文本串 T 与模式串 P 进行匹配的函数 KMP() 的 C++ 实现如下：

```
int KMP(char *T, char *P) {
    int n = strlen(T), m = strlen(P), i = 0, j = 0;
    int *next = new int[m];
    Next(P, next);
    while (i < n) {
        if (T[i] == P[j]) {
            if (j == m - 1) return i - j;       // 匹配成功
            else { i++; j++; }
        } else if (j > 0) j = next[j - 1];
        else i++;
    }
    delete[] next;
    return -1;                                  // 匹配失败
}
```

KMP 算法的时间复杂度为 $O(m+n)$，其中 m 为模式串 P 的长度，n 为文本串 T 的长度。

1.3 树

1.3.1 树的基本概念

树是一种由**一对多**分支关系定义的层级结构。

定义 1.3.1. 树

树是由 $n(n \geq 0)$ 个结点组成的有限集合，满足：

- 若 $n = 0$，则称为**空树**；
- 对 $n > 0$ 的非空树：
  - 有且仅有一个特定的称为**根**（root）的结点，只有直接后继，没有直接前驱；
  - 除根以外的其余结点划分为 $m(m > 0)$ 个互不相交的有限集合 $T_1, T_2, \cdots, T_m$，每个集合本身又是一棵树，称为根的**子树**（subtree）。子树的根节点有且仅有一个直接前驱，可以有 0 个或多个直接后继。没有直接后继的结点称为**叶子**（leaf）。

定义 1.3.2. 一个结点处的子树的个数称为该结点的**度**，树中结点的最大度称为树的**度**。如果树的度为 k，则称该树为 **k 叉树**。

定义 1.3.3. 对树中的结点 v，其直接后继（或者说其子树的根）称为 v 的**孩子**（*child*）结点，v 称为孩子结点的**双亲**（*parent*）结点；结点 v 的孩子结点之间互为**兄弟**（*sibling*）结点，v 的不同孩子结点的孩子结点之间互为**堂兄弟**（*cousin*）结点；从根结点到结点 v 的路径上的结点都是 v 的**祖先**（*ancestor*）结点，v 的所有子树中的结点都是 v 的**后裔**（*descendant*）结点。

定义 1.3.4. 树中结点的**层次**从根开始定义，根结点的层次为 *1*，其余结点的层次等于其双亲结点的层次加 *1*；树中（叶子）结点的最大层次称为树的**深度**（*depth*）。

从离散数学的角度，树结构有如下性质：

定理 1.3.1. 树中结点的数目等于所有结点的度数和加 1。

定理 1.3.2. 度为 k 的树中第 i 层上最多有 k^{i-1} 个结点。

定理 1.3.3. 深度为 h 的 k 叉树最多有 $\dfrac{k^h-1}{k-1}$ 个结点；有 n 个结点的 k 叉树的最小深度为 $\lceil \log_k(n\cdot(k-1)+1) \rceil$。

1.3.2 二叉树

二叉树是一种特殊的树，其每个结点最多有两个子树，且子树有左右之分，次序不能颠倒。

定义 1.3.5. 二叉树

二叉树是一个包含 n 个结点的有限集合，满足：

- 若 $n=0$，则称为**空二叉树**；
- 若 $n>0$，则满足：
  - 有且仅有一个特定的称为**根**（root）的结点，只有直接后继，没有直接前驱；
  - 除根以外的其余结点划分为两个互不相交的有限集合 L 和 R，每个集合本身又是一棵二叉树，分别称为根的**左子树**（left subtree）和**右子树**（right subtree）；L 和 R 的根节点分别称为根的**左孩子**（left child）和**右孩子**（right child）。

定义 1.3.6. 完全二叉树、满二叉树

若一棵深度为 h 的二叉树，除第 h 层外，其它各层的结点数都达到最大值，且第 h 层的结点都连续集中在最左边，则称其为**完全二叉树**；若一棵二叉树每一层的结点数都达到最大值，则称其为**满二叉树**。

定理 1.3.4. 具有 n 个结点的完全二叉树的深度为 $\lceil \log_2(n+1) \rceil$。

定理 1.3.5. 完全二叉树的结点编号 对 n 个结点的完全二叉树自顶向下、自左向右进行 *0* 至 $n-1$ 的编号，则对结点 i 有如下性质：

- 若 $i=0$，则为根结点，无双亲；否则，其双亲结点编号为 $\left\lfloor \dfrac{i-1}{2} \right\rfloor$；
- 若 $2i+1>n$，则该结点无左孩子；否则，其左孩子编号为 $2i+1$；
- 若 $2i+2>n$，则该结点无右孩子；否则，其右孩子编号为 $2i+2$；
- 若 i 为非零偶数，则其左兄弟结点编号为 $i-1$；
- 若 i 为不大于 $n-2$ 的奇数，则其右兄弟结点编号为 $i+1$，否则该结点无右兄弟；
- 结点 i 的层次为 $\lceil \log_2(i+2) \rceil$。

1.3.3 二叉树的数据结构

二叉树的抽象数据类型可以定义为：

```
ADT BiTree {
    数据对象：D = {a_0, a_1, a_2, ···, a_n | a_i ∈ ElemType, 1 ≤ i ≤ n}
    数据关系：
        若 D = ∅，则 R = ∅；
        若 D ≠ ∅，则 R = H，其中 H 是一个二元关系，满足：
        + ∃root ∈ D，∀v ∈ D，⟨v, root⟩ ∉ H；
        + 若 D − {root} ≠ ∅，则 ∃D_l, D_r ⊆ D − {root} 为 D 的划分，；
        + 若 D_l ≠ ∅，则 ∃!x_l ∈ D_l，使得 ⟨root, x_l⟩ ∈ H，且 D_l 上的关系 H_l ⊂ H；
          若 D_r ≠ ∅，则 ∃!x_r ∈ D_r，使得 ⟨root, x_r⟩ ∈ H，且 D_r 上的关系 H_r ⊂ H；
          并且，上述的 {⟨root, x_l⟩, ⟨root, x_r⟩} 以及 H_l 和 H_r 是 H 的三划分。
        + D_l 连带 H_l、D_r 连带 H_r 是符合本定义的二叉树。
    基本操作：
        InitBiTree(&T)
            操作结果：构造一个空二叉树 T。
        DestroyBiTree(&T)
            初始条件：二叉树 T 存在。
            操作结果：销毁二叉树 T。
        CreateBiTree(&T, definition)
            初始条件：二叉树 T 不存在。
            操作结果：按 definition 中给出的定义建立二叉树 T。
        ClearBiTree(&T)
            初始条件：二叉树 T 存在。
            操作结果：将二叉树 T 清空。
        isEmpty(T)
            初始条件：二叉树 T 存在。
            操作结果：若 T 为空二叉树，则返回 true，否则返回 false。
        GetDepth(T)
            初始条件：二叉树 T 存在。
            操作结果：返回 T 的深度。
        GetRoot(T)
            初始条件：二叉树 T 存在。
            操作结果：返回 T 的根。
        GetValue(T, cur_e)
            初始条件：二叉树 T 存在，cur_e 是 T 中的某个结点。
            操作结果：返回 cur_e 的值。
        Assign(&T, cur_e, value)
            初始条件：二叉树 T 存在，cur_e 是 T 中的某个结点。
            操作结果：将 cur_e 结点的值赋为 value。
        GetParent(T, cur_e)
            初始条件：二叉树 T 存在，cur_e 是 T 中的某个结点。
            操作结果：返回 cur_e 的双亲；若 cur_e 是根，则返回空。
        LeftChild(T, cur_e)
            初始条件：二叉树 T 存在，cur_e 是 T 中的某个结点。
            操作结果：返回 cur_e 的左孩子；若 cur_e 无左孩子，则返回空。
        RightChild(T, cur_e)
            初始条件：二叉树 T 存在，cur_e 是 T 中的某个结点。
            操作结果：返回 cur_e 的右孩子；若 cur_e 无右孩子，则返回空。
        LeftSibling(T, cur_e)
            初始条件：二叉树 T 存在，cur_e 是 T 中的某个结点。
            操作结果：返回 cur_e 的左兄弟；若 cur_e 无左兄弟，则返回空。
        RightSibling(T, cur_e)
```

```
            初始条件：二叉树 T 存在，cur_e 是 T 中的某个结点。
            操作结果：返回 cur_e 的右兄弟；若 cur_e 无右兄弟，则返回空。
        InsertChild(&T, p, LR, c)
            初始条件：二叉树 T 存在，p 是 T 中的某个结点，LR = 0 时插入左子树，LR = 1 时
                插入右子树；c 为待插入的非空二叉树，与 T 不相交且右子树为空。
            操作结果：将 p 的左子树或右子树插入 c。
        DeleteChild(&T, p, LR)
            初始条件：二叉树 T 存在，p 是 T 中的某个结点，LR = 0 时删除左子树，LR = 1 时
                删除右子树。
            操作结果：删除 p 的左子树或右子树。
        PreOrderTraverse(&T, visit())
            初始条件：二叉树 T 存在，visit() 是对结点操作的函数。
            操作结果：前序遍历 T，对每个结点调用 visit()。
        InOrderTraverse(&T, visit())
            初始条件：二叉树 T 存在，visit() 是对结点操作的函数。
            操作结果：中序遍历 T，对每个结点调用 visit()。
        PostOrderTraverse(&T, visit())
            初始条件：二叉树 T 存在，visit() 是对结点操作的函数。
            操作结果：后序遍历 T，对每个结点调用 visit()。
        LevelOrderTraverse(&T, visit())
            初始条件：二叉树 T 存在，visit() 是对结点操作的函数。
            操作结果：层序遍历 T，对每个结点调用 visit()。
} ADT BiTree
```

A) 二叉树的存储 二叉树的存储结构有顺序存储结构和链式存储结构两种。

- **二叉树的顺序存储**：二叉树的顺序存储结构是用一维数组存储二叉树中的结点，按照完全二叉树的结点编号规则存储。对于结点 i，其左孩子结点编号为 $2i+1$，右孩子结点编号为 $2i+2$，双亲结点编号为 $\left\lfloor \frac{i-1}{2} \right\rfloor$。对于非完全二叉树，顺序存储浪费的空间极大，如对于单分支的二叉树，顺序存储所真正使用的空间仅为所占用空间的 $\frac{n}{2^{n-1}-1}$。
- **二叉树的链式存储**：二叉树的链式存储结构是用链表存储二叉树中的结点，每个结点包含数据域和两个指针域，分别指向左孩子和右孩子。考虑到寻求双亲结点的需求，也可在结点中增加一个指向双亲结点的指针域。

  链式存储结构的二叉树的结点定义如下：

```
typedef struct nodeForBiTree {
    ElemType data;
    struct TNode *lchild, *rchild, *parent;
} TNode;
```

  这样，可以用 C++ 实现二叉树的数据结构如下：

```
class BiTree {
private:
    TNode *root;
public:
    BiTree()                                    { root = nullptr; }
    ~BiTree()                                   { DestroyBiTree(); }
    void InitBiTree()                           { root = nullptr; }
```

```
    void DestroyBiTree()
        { void deleteTNode(TNode *t) { delete t; }
          PostOrderTraverse(deleteTNode); root = nullptr; }
    void CreateBiTree( /* 规则 */ )             { /* 递归创建二叉树 */ }
    void ClearBiTree()                          { DestroyBiTree(); }
    bool isEmpty()                              { return root == nullptr; }
    int getDepth()                              { /* 递归计算二叉树深度 */; }
    TNode* getRoot()                            { return root; }
    ElemType getValue(TNode *cur_e)             { return cur_e->data; }
    void Assign(TNode *cur_e, ElemType value)   { cur_e->data = value; }
    TNode* Parent(TNode *cur_e)                 { return cur_e->parent; }
    TNode* LeftChild(TNode *cur_e)              { return cur_e->lchild; }
    TNode* RightChild(TNode *cur_e)             { return cur_e->rchild; }
    TNode* LeftSibling(TNode *cur_e)            { return cur_e->parent->lchild; }
    TNode* RightSibling(TNode *cur_e)           { return cur_e->parent->rchild; }
    void InsertChild(TNode *p, int LR, BiTree c)        { /* 插入子树 */ }
    void DeleteChild(TNode *p, int LR)                  { /* 删除子树 */ }
    void PreOrderTraverse(void (*visit)(TNode*))        { /* 前序遍历 */ }
    void InOrderTraverse(void (*visit)(TNode*))         { /* 中序遍历 */ }
    void PostOrderTraverse(void (*visit)(TNode*))       { /* 后序遍历 */ }
    void LevelOrderTraverse(void (*visit)(TNode*))      { /* 层序遍历 */ }
};
```

B) 二叉树的遍历 二叉树是一种递归结构，可以用递归思路来研究二叉树遍历方法。根据访问根节点时机，二叉树的遍历可分为**前序遍历**、**中序遍历**、**后序遍历**；还可以对二叉树自上而下逐层遍历，即**层序遍历**。

算法 1.2. 二叉树的递归遍历

思想 在遍历左、右子树时，递归调用遍历函数；在访问根节点时，执行边界操作。

步骤 容易实现。

- **前序遍历**：先访问根节点，再前序遍历左子树，最后前序遍历右子树，即

```
void PreOrderTraverse(Tnode* t, void (*visit)(Tnode*)) {
    if (t) {                                    // 若二叉树非空
        visit(t->data);                         // 访问根节点
        PreOrderTraverse(t->lchild, visit);     // 前序遍历左子树
        PreOrderTraverse(t->rchild, visit);     // 前序遍历右子树
    }
}
```

- **中序遍历**：先中序遍历左子树，再访问根节点，最后中序遍历右子树，即

```
void InOrderTraverse(Tnode* t, void (*visit)(Tnode*)) {
    if (t) {                                    // 若二叉树非空
        InOrderTraverse(t->lchild, visit);      // 中序遍历左子树
        visit(t->data);                         // 访问根节点
        InOrderTraverse(t->rchild, visit);      // 中序遍历右子树
    }
}
```

- **后序遍历**：先后序遍历左子树，再后序遍历右子树，最后访问根节点，即

```
void PostOrderTraverse(Tnode* t, void (*visit)(Tnode*)) {
    if (t) {                                    // 若二叉树非空
        PostOrderTraverse(t->lchild, visit);    // 后序遍历左子树
        PostOrderTraverse(t->rchild, visit);    // 后序遍历右子树
        visit(t->data);                         // 访问根节点
    }
}
```

递归算法简介、清晰，但是递归深度过大时会导致栈溢出，因此可以考虑用非递归算法实现二叉树的遍历，即**消除递归**。对这类较复杂的递归情况，消除递归就是用一个显式的栈来模拟递归调用的隐式的栈。

算法 1.3. 二叉树的非递归遍历

思想 用一个栈来模拟递归调用的隐式栈，实现二叉树的非递归遍历。

步骤 （1）创建一个栈，用于保存结点；（2）按照需要的遍历顺序开始遍历，遇到要经过但没有访问的结点或没有访问某一侧孩子的结点时，将其入栈；（3）访问到叶子结点，出栈一个结点继续遍历。

- **前序遍历**：用栈保存没有访问右孩子的结点，即

```
void PreOrderTraverse(Tnode* t, void (*visit)(Tnode*)) {
    stack<Tnode*> s;                // 创建栈保存结点
    Tnode* p = t;                   // 从根结点出发遍历
    while (p || !s.empty()) {
        if (p) {                    // 若结点非空
            visit(p);               // 访问结点
            s.push(p);              // 结点入栈
            p = p->lchild;          // 移至左孩子
        } else {                    // 若结点为空
            p = s.top(); s.pop();   // 出栈一个结点
            p = p->rchild;          // 移至右孩子
        }
    }
}
```

- **中序遍历**：用栈保存移入了左孩子但没有访问的结点，即

```
void InOrderTraverse(Tnode* t, void (*visit)(Tnode*)) {
    stack<Tnode*> s;                // 创建栈保存结点
    Tnode* p = t;                   // 从根结点出发遍历
    while (p || !s.empty()) {
        if (p) {                    // 若结点非空
            s.push(p);              // 结点入栈
            p = p->lchild;          // 移至左孩子
        } else {                    // 若结点为空
            p = s.top(); s.pop();   // 出栈一个结点
            visit(p);               // 访问结点
```

```
            p = p->rchild;          // 移至右孩子
        }
    }
}
```

- **后序遍历**：用栈保存访问过右孩子但没有访问的结点，即

```
void PostOrderTraverse(Tnode* t, void (*visit)(Tnode*)) {
    stack<Tnode*> s;                // 创建栈保存结点
    Tnode* p = t, *r = nullptr;     // 从根结点出发遍历
    while (p || !s.empty()) {
        if (p) {                    // 若结点非空
            s.push(p);              // 结点入栈
            p = p->lchild;          // 移至左孩子
        } else {                    // 若结点为空
            p = s.top();            // 取栈顶结点
            if (p->rchild && p->rchild != r) {  // 若右孩子存在且未访问
                p = p->rchild;      // 移至右孩子
            } else {                // 若右孩子不存在或已访问
                visit(p);           // 访问结点
                r = p;              // 记录访问过的结点
                s.pop();            // 出栈一个结点
                p = nullptr;        // 结点置空
            }
        }
    }
}
```

算法 1.4. 二叉树的层序遍历

思想 用队列保存每一层的结点，实现二叉树的层序遍历。

步骤 （1）创建一个队列，用于保存每一层的结点；（2）从根结点开始，将根结点入队；（3）反复执行：出队一个结点，访问该结点，将其左右孩子入队；（4）直至队列为空。

```
void LevelOrderTraverse(Tnode* t, void (*visit)(Tnode*)) {
    queue<Tnode*> q;                    // 创建队列保存结点
    Tnode* p = t;                       // 从根结点出发遍历
    q.push(p);                          // 根结点入队
    while (!q.empty()) {
        p = q.front(); q.pop();         // 出队一个结点
        visit(p);                       // 访问结点
        if (p->lchild) q.push(p->lchild);   // 左孩子入队
        if (p->rchild) q.push(p->rchild);   // 右孩子入队
    }
}
```

C) **二叉树的建立** 二叉树的遍历是把树状的二叉树结构转换为线性的遍历序列，这个过程显然是不可逆的，即**根据一个遍历序列无法唯一建立一棵二叉树**。要根据遍历序列建立二叉树，需要更多的信息：

- 同时给定**前序序列和中序序列**可唯一地建立一棵二叉树。其基本思想是，前序序列的第一个结点是根结点，根据中序序列将结点分为左右子树，于是可以回到前序序列中找到左右子树的根结点，递归建立左右子树。
- 同时给定**后序序列和中序序列**可唯一地建立一棵二叉树，其基本思想也类似。
- 同时给定**前序序列和后序序列**无法唯一地建立一棵二叉树，因为没有提供更多关于左右子树划分的信息。

A LambdaOJ 平台介绍

A.1 判题结果

判题有如下结果：

- `ACCEPTED`：结果完全正确。
- `WRONG_ANSWER`：程序正常执行，也没有超时和超内存，但是答案不对。
- `TIME_LIMIT_EXCEEDED`：程序超时，可能是死循环，也可能是算法不够优，或者实现不够优；还没到核对答案那一步，内存情况也不明。
- `MEMORY_LIMIT_EXCEEDED`：程序超过内存限制，可能是算法不够优，或者实现不够优；也没有到核对答案那一步。
- `OUTPUT_LIMIT_EXCEEDED`：输出数据过多，要么程序有 bug，要么是恶意代码。
- `BAD_SYSCALL`：使用了非法的系统调用，要么是恶意代码，大部分情况是 C++ 在 `new` 一块内存的时候，超过了限制失败了，C++ 运行时环境需要一个系统调用关闭信号，然后杀死这个进程。
- `RUN_TIME_ERROR`：运行时错误，可能是由于除零、引用空指针或者数组越界等等。

A.2 注意事项

尽量使用 `stdio.h` 而不使用 `iostream`，前者的输入输出效率更高，时间和空间开销更小。