

数据与算法知识与方法

T^TT

2024 年 10 月 8 日

目录

0 绪论：数据、数学模型、算法	2	1.2 线性结构	5
0.1 数据及其结构	2	1.2.1 线性表	5
0.2 数学模型	2	1.2.2 栈	7
0.3 算法	2	1.2.3 队列	10
1 数据结构	4	1.2.4 串	13
1.1 抽象数据类型	4	A LambdaOJ 平台介绍	16
		A.1 判题结果	16
		A.2 注意事项	16

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：

```
1  for(int i = 0; i < n; i++)
2      for(int j = i; j < n; j++)
3          if(a[i] > a[j])
4              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 $\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1 = \sum_{i=0}^{n-1} (n-i) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$ ，则该程序的时间复杂度为 $O\left(\frac{n(n-1)}{2}\right) = O(n^2)$ 。

例 0.3.2. 现有程序如下：

```
1  for(int i = 1; i < n; i++)
2      for(int j = 0; j < i; j *= 2)
3          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

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

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：

```

1  ADT Complex {
2      数据对象:  $D = \{e_1, e_2 \mid e_1, e_2 \in \mathbb{R}\}$ 
3      数据关系:  $R = \{\langle e_1, e_2 \rangle \mid e_1 = \Re(D), e_2 = \Im(D)\}$ 
4      基本操作:
5          InitComplex(&Z, v1, v2)
6              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。
7          DestroyComplex(&Z)
8              初始条件: 复数 Z 存在;
9              操作结果: 复数 Z 被销毁。
10         Add(z1, z2, &sum)
11             初始条件: z1, z2 是复数。
12             操作结果: 用 sum 返回两个复数 z1, z2 的和值。
13         Sub(z1, z2, &sub)
14             初始条件: z1, z2 是复数。
15             操作结果: 用 sub 返回两个复数 z1, z2 的差值。
16     } ADT Complex

```

例 1.1.2. 圆柱体的抽象数据类型定义如下：

```

1  ADT CYLinder {
2      数据对象:  $D = \{r, h \mid r, h \in \mathbb{R}\}$ 
3      数据关系:  $R = \{\langle r, h \rangle \mid r \text{ 为圆柱底面半径, } h \text{ 为圆柱高}\}$ 
4      基本操作:
5          InitCyld(r, h)
6              操作结果: 构造圆柱体, 底面半径 r, 圆柱高 h。
7          BaseArea(r, &bArea)
8              初始条件: 圆柱体存在。
9              操作结果: 计算圆柱体底面积, 用 bArea 返回。
10         SideArea(r, h, &sArea)
11             初始条件: 圆柱体存在。
12             操作结果: 计算圆柱体侧面积, 用 sArea 返回。
13         Volume(r, h, &vol)
14             初始条件: 圆柱体存在。
15             操作结果: 计算圆柱体体积, 用 vol 返回。
16     } 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 $L = (a_0, a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ ，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：

```

1  ADT List {
2     数据对象:  $D = \{a_0, a_1, a_2, \dots, a_n \mid a_i \in \text{ElemType}, 1 \leq i \leq n\}$ 
3     数据关系:  $R = \{\langle a_i, a_{i+1} \rangle \mid a_i \text{ 为第 } i \text{ 个数据元素, } a_{i+1} \text{ 为第 } i+1 \text{ 个数据元素}\}$ 
4     基本操作:
5         InitList(&L)
6             操作结果: 构造一个空的线性表 L。
7         DestroyList(&L)
8             初始条件: 线性表 L 存在。
9             操作结果: 销毁线性表 L。
10        IsEmpty(L)
11            初始条件: 线性表 L 存在。
12            操作结果: 若 L 为空表, 则返回 true, 否则返回 false。
13        ListLength(L)
14            初始条件: 线性表 L 存在。
15            操作结果: 返回 L 中数据元素的个数。
16        GetElem(L, i, &e)
17            初始条件: 线性表 L 存在,  $0 \leq i < \text{ListLength}(L)$ 。
18            操作结果: 用 e 返回 L 中第 i 个数据元素的值。
19        LocateElem(L, e, compare())
20            初始条件: 线性表 L 存在, compare() 是数据元素判定函数。
21            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。
22        PriorElem(L, cur_e, &pre_e)
23            初始条件: 线性表 L 存在。
24            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。
25        NextElem(L, cur_e, &next_e)
26            初始条件: 线性表 L 存在。
27            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。
28        ClearList(&L)
29            初始条件: 线性表 L 存在。

```

```

30      操作结果：将 L 重置为空表。
31      ListInsert(&L, i, e)
32      初始条件：线性表 L 存在， $0 \leq i \leq \text{ListLength}(L)$ 。
33      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。
34      ListDelete(&L, i, &e)
35      初始条件：线性表 L 存在， $0 \leq i < \text{ListLength}(L)$ 。
36      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。
37      ListTraverse(L, visit())
38      初始条件：线性表 L 存在，visit() 是对数据元素操作的函数。
39      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。
40  } 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：

```

1  class List {
2  private:
3      ElemType *elem;
4      int length, maxSize;
5  public:
6      List(int size) : maxSize(size)
7      { elem = new ElemType[maxSize]; length = 0; }
8      ~List() { delete[] elem; }
9      bool isEmpty() const { return length == 0; }
10     int listLength() const { return length; }
11     ElemType getElem(int i) const { return elem[i]; }
12     int locateElem(ElemType e) const {
13         for (int i = 0; i < length; i++)
14             if (elem[i] == e) return i;
15         return -1;
16     }
17     ElemType priorElem(ElemType e) const {
18         int i = locateElem(e);
19         if (i > 0) return elem[i - 1];
20         return -1;
21     }
22     ElemType nextElem(ElemType e) const {
23         int i = locateElem(e);
24         if (i >= 0 && i < length - 1) return elem[i + 1];
25         return -1;
26     }
27     void clearList() { length = 0; }
28     bool listInsert(int i, ElemType e) {
29         if (i < 0 || i > length || length == maxSize) return false;
30         for (int j = length; j > i; j--)
31             elem[j] = elem[j - 1];
32         elem[i] = e;
33         length++;
34         return true;

```

```

35     }
36     ElemType listDelete(int i) {
37         if (i < 0 || i >= length) return -1;
38         ElemType e = elem[i];
39         for (int j = i; j < length - 1; j++)
40             elem[j] = elem[j + 1];
41         length--; return e;
42     }
43     void listTraverse(void (*visit)(ElemType)) const {
44         for (int i = 0; i < length; i++)
45             visit(elem[i]);
46     }
47 };

```

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：

```

1  ADT Stack {
2      数据对象：  $D = \{a_0, a_1, a_2, \dots, a_n \mid a_i \in \text{ElemType}, 1 \leq i \leq n\}$ 
3      数据关系：  $R = \{\langle a_i, a_{i+1} \rangle \mid a_i \text{ 为第 } i \text{ 个数据元素, } a_{i+1} \text{ 为第 } i+1 \text{ 个数据元素}\}$ 
4      基本操作：
5          InitStack(&S)
6              操作结果：构造一个空栈 S。
7          DestroyStack(&S)
8              初始条件：栈 S 存在。
9              操作结果：销毁栈 S。
10         IsEmpty(S)
11             初始条件：栈 S 存在。
12             操作结果：若 S 为空栈，则返回 true，否则返回 false。
13         GetTop(S, &e)
14             初始条件：栈 S 存在，S 非空。
15             操作结果：用 e 返回 S 的栈顶元素。
16         Push(&S, e)
17             初始条件：栈 S 存在。
18             操作结果：将元素 e 压入栈顶。
19         Pop(&S, &e)
20             初始条件：栈 S 存在，S 非空。
21             操作结果：弹出栈顶元素并用 e 返回其值。
22         ClearStack(&S)
23             初始条件：栈 S 存在。
24             操作结果：将 S 重置为空栈。
25         StackLength(S)
26             初始条件：栈 S 存在。
27             操作结果：返回 S 的元素个数。
28     } 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:

```

1  class Stack {
2  private:
3      ElemType *bottom;
4      int stackDepth, maxSize;
5  public:
6      Stack(int size) : maxSize(size)
7          { bottom = new ElemType[maxSize]; stackDepth = 0; }
8      ~Stack() { delete[] bottom; }
9      bool isEmpty() const { return stackDepth == 0; }
10     ElemType getTop() const { return bottom[stackDepth - 1]; }
11     int stackLength() const { return stackDepth; }
12     void push(ElemType e) { bottom[stackDepth++] = e; }
13     ElemType pop() { return bottom[--stackDepth]; }
14     void clearStack() { stackDepth = 0; }
15 };

```

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:

```

1  struct Node {
2      ElemType data;
3      Node *next;
4  };
5  typedef struct Node* Link;
6  class Stack {
7  private:
8      Link head;
9  public:
10     Stack() { head = nullptr; }
11     ~Stack() { clearStack(); }
12     bool isEmpty() const { return head == nullptr; }
13     ElemType getTop() const { return head->data; }
14     int stackLength() const
15         { int len = 0; for (Link p = head; p; p = p->next) len++; return len; }
16     void push(ElemType e)
17         { Link p = new Node; p->data = e; p->next = head; head = p; }
18     ElemType pop()
19         { Link p = head; head = head->next; ElemType e = p->data; delete p; return e; }
20     void clearStack() { while (!isEmpty()) pop(); }
21 };

```

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：

```

1  double calcPostfix(const char *postfix) {
2      stringstream ss(postfix);           // 用字符串流读取后缀表达式
3      Stack values;
4      int curValue = 0; char curChar = '\0';
5      while (ss >> curChar) {             // 逐个读取字符
6          if (isdigit(curChar)) {          // 对数字字，符转换为数字并入记录数值
7              curValue = curValue * 10 + curChar - '0';
8          } else {                          // 对操作，符弹栈计算
9              int preValue = values.pop();
10             switch (curChar) {
11                 case '+': values.push(preValue + curValue); break;
12                 case '-': values.push(preValue - curValue); break;
13                 case '*': values.push(preValue * curValue); break;
14                 case '/': values.push(preValue / curValue); break;
15             }
16             curValue = 0;                  // 重置记录数值
17         }
18     }
19     if (values.stackLength() != 1) throw ValueException("Invalid postfix expression!");
20     return values.pop();
21 }

```

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：

```

1  int fibonacci(int n) {
2      if (n == 0) return 1;
3      if (n == 1) return 1;
4      return fibonacci(n - 1) + fibonacci(n - 2);
5  }

```

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：

```

1  int fibonacci(int n) {
2      if (n == 0) return 1;
3      if (n == 1) return 1;
4      int pre = 0, cur = 1, next;
5      for (int i = 2; i <= n; i++) {
6          next = pre + cur;
7          pre = cur; cur = next;
8      }
9      return next;
10 }

```

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：

```

1  ADT Queue {
2      数据对象:  $D = \{a_0, a_1, a_2, \dots, a_n \mid a_i \in \text{ElemType}, 1 \leq i \leq n\}$ 
3      数据关系:  $R = \{\langle a_i, a_{i+1} \rangle \mid a_i \text{ 为第 } i \text{ 个数据元素, } a_{i+1} \text{ 为第 } i+1 \text{ 个数据元素}\}$ 
4              约定: 队头为  $a_0$  队尾为  $a_n$ 。
5      基本操作:
6          InitQueue(&Q)
7              操作结果: 构造一个空队列 Q。
8          DestroyQueue(&Q)
9              初始条件: 队列 Q 存在。
10             操作结果: 销毁队列 Q。
11          IsEmpty(Q)
12              初始条件: 队列 Q 存在。
13              操作结果: 若 Q 为空队列, 则返回 true, 否则返回 false。
14          QueueLength(Q)
15              初始条件: 队列 Q 存在。
16              操作结果: 返回 Q 的元素个数。
17          GetHead(Q, &e)
18              初始条件: 队列 Q 存在, Q 非空。
19              操作结果: 用 e 返回 Q 的队头元素。
20          EnQueue(&Q, e)
21              初始条件: 队列 Q 存在。
22              操作结果: 将元素 e 入队列 Q 的队尾。
23          DeQueue(&Q, &e)
24              初始条件: 队列 Q 存在, Q 非空。
25              操作结果: 将 Q 的队头元素出队, 并用 e 返回其值。
26          ClearQueue(&Q)
27              初始条件: 队列 Q 存在。
28              操作结果: 将 Q 重置为空队列。
29          QueueTraverse(Q, visit())
30              初始条件: 队列 Q 存在, visit() 是对数据元素操作的函数。

```

```

31         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
32         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：

```

1  class Queue {
2  private:
3      ElemType *base;
4      int front, rear, maxSize;
5  public:
6      Queue(int size) : maxSize(size)
7          { base = new ElemType[maxSize]; front = rear = 0; }
8      ~Queue() { delete[] base; }
9      bool isEmpty() const { return front == rear; }
10     ElemType getHead() const { return base[front]; }
11     int queueLength() const { return (rear - front + maxSize) % maxSize; }
12     void enqueue(ElemType e)
13         { base[rear] = e; rear = (rear + 1) % maxSize; }
14     ElemType dequeue()
15         { ElemType e = base[front]; front = (front + 1) % maxSize; return e; }
16     void clearQueue() { front = rear = 0; }
17 };

```

- 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：

```

1  struct Node {
2      ElemType data;
3      Node *next;
4  };
5  typedef struct Node* Link;
6  class Queue {
7  private:
8      Link front, rear;
9  public:
10     Queue() { front = rear = new Node; front->next =
11         nullptr; }
12     ~Queue() { clearQueue(); delete front; }
13     bool isEmpty() const { return front == rear; }
14     ElemType getHead() const { return front->next->data; }
15     int queueLength() const
16         { int len = 0; for (Link p = front->next; p; p = p->next) len++; return len; }
17     void enqueue(ElemType e)
18         { Link p = new Node; p->data = e; p->next = nullptr; rear->next = p; rear = p; }

```

```

18     ElemType dequeue()
19     { Link p = front->next; front->next = p->next; ElemType e = p->data; delete
      p; return e; }
20     void clearQueue()          { while (!isEmpty()) dequeue(); }
21 };

```

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:

```

1  ADT PQueue {
2      数据对象:  $D = \{a_0, a_1, a_2, \dots, a_n \mid a_i \in \text{map}\langle \text{int}, \text{ElemType} \rangle, 1 \leq i \leq n\}$ 
3      数据关系:  $R = \{\langle a_i, a_{i+1} \rangle \mid a_i \text{ 为第 } i \text{ 个数据元素, } a_{i+1} \text{ 为第 } i+1 \text{ 个数据元素}\}$ 
4              约定: 队头为  $a_0$ , 队尾为  $a_n$ 。
5      基本操作:
6          InitPQueue(&Q)
7              操作结果: 构造一个空优先级队列 Q。
8          DestroyPQueue(&Q)
9              初始条件: 优先级队列 Q 存在。
10             操作结果: 销毁优先级队列 Q。
11          IsEmpty(Q)
12              初始条件: 优先级队列 Q 存在。
13              操作结果: 若 Q 为空队列, 则返回 true, 否则返回 false。
14          PQueueLength(Q)
15              初始条件: 优先级队列 Q 存在。
16              操作结果: 返回 Q 的元素个数。
17          GetHead(Q, &e)
18              初始条件: 优先级队列 Q 存在, Q 非空。
19              操作结果: 用 e 返回 Q 中优先级最高的元素。
20          EnPQueue(&Q, e)
21              初始条件: 优先级队列 Q 存在。
22              操作结果: 将元素 e 插入优先级队列 Q。
23          ChangePQueue(&Q, ei, ef)
24              初始条件: 优先级队列 Q 存在, Q 非空, ei 是 Q 中的元素。
25              操作结果: 将 Q 中元素 ei 的优先级改为 ef。
26          DePQueue(&Q, &e)
27              初始条件: 优先级队列 Q 存在, Q 非空。
28              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。
29          ClearPQueue(&Q)
30              初始条件: 优先级队列 Q 存在。
31              操作结果: 将 Q 重置为空优先级队列。
32          PQueueTraverse(Q, visit())
33              初始条件: 优先级队列 Q 存在, visit() 是对数据元素操作的函数。
34              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。
35  } 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：

```
1 ADT String {
2   数据对象:  $D = \{a_0, a_1, a_2, \dots, a_n \mid a_i \in \text{char}, 1 \leq i \leq n\}$ 
3   数据关系:  $R = \{\langle a_i, a_{i+1} \rangle \mid a_i \text{ 为第 } i \text{ 个字符, } a_{i+1} \text{ 为第 } i+1 \text{ 个字符}\}$ 
4   基本操作:
5       StrAssign(&S, chars)
6           初始条件: chars 是字符串常量。
7           操作结果: 生成一个其值等于 chars 的串 S。
8       StrCopy(&S, T)
9           初始条件: 串 T 存在。
10          操作结果: 由串 T 复制得到串 S。
11       StrEmpty(S)
12          初始条件: 串 S 存在。
13          操作结果: 若 S 为空串，则返回 true，否则返回 false。
14       StrLength(S)
15          初始条件: 串 S 存在。
16          操作结果: 返回 S 的长度。
17       StrCompare(S, T)
18          初始条件: 串 S 和 T 存在。
19          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。
20       Concat(&S, T1, T2)
21          初始条件: 串 T1 和 T2 存在。
22          操作结果: 用 S 返回由 T1 和 T2 连接而成的新串。
23       SubStr(&S, T, pos, len)
```

```

24      初始条件: 串  $T$  存在,  $1 \leq \text{pos} \leq \text{StrLength}(T)$ ,  $0 \leq \text{len} \leq \text{StrLength}(T) - \text{pos} + 1$ 。
25      操作结果: 用  $S$  返回串  $T$  的第  $\text{pos}$  个字符起长度为  $\text{len}$  的子串。
26      Index( $S, T, \text{pos}$ )
27      初始条件: 串  $S$  和  $T$  存在,  $1 \leq \text{pos} \leq \text{StrLength}(S)$ 。
28      操作结果: 若  $S$  中存在和  $T$  值相同的子串, 则返回  $S$  中从第  $\text{pos}$  个字符起第一个这样的子串的位置, 否则返回 0。
29      Replace(& $S, T, V$ )
30      初始条件: 串  $S$ 、 $T$  和  $V$  存在,  $T$  非空。
31      操作结果: 用  $V$  替换  $S$  中出现的所有与  $T$  相等的非重叠的子串。
32      StrTraverse( $S, \text{visit}()$ )
33      初始条件: 串  $S$  存在,  $\text{visit}()$  是对数据元素操作的函数。
34      操作结果: 依次对  $S$  的每个数据元素调用  $\text{visit}()$ , 一旦  $\text{visit}()$  失败, 则操作失败。
35  } 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：

```
1 void Next(char *P, int next[]) {  
2     int m = strlen(P), i = 1, j = 0;  
3     next[0] = 0;  
4     while (i < m) {  
5         if (P[i] == P[j]) next[i++] = ++j;           // 相同字符, next[i] = j + 1  
6         else if (j > 0) j = next[j - 1];             // 不同字符, 回溯到前一个字符  
7         else next[i++] = 0;                           // 不同字符且无法回溯, next[i] = 0  
8     }  
9 }
```

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：

```
1 int KMP(char *T, char *P) {  
2     int n = strlen(T), m = strlen(P), i = 0, j = 0;  
3     int *next = new int[m];  
4     Next(P, next);  
5     while (i < n) {  
6         if (T[i] == P[j]) {  
7             if (j == m - 1) return i - j;           // 匹配成功  
8             else { i++; j++; }  
9         } else if (j > 0) j = next[j - 1];  
10        else i++;  
11    }  
12    delete[] next;  
13    return -1;                                       // 匹配失败  
14 }
```

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。

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。