数据结构与算法期末复习
Homework

尹超

中国科学院大学，北京 100049

Carter Yin

University of Chinese Academy of Sciences, Beijing 100049, China

2024.8 – 2025.1

序言

本文为笔者数据结构与算法的期末复习笔记。

望老师批评指正。

目录

题目 第四章

63

> 栈 S 初始为空，进行以下操作后从栈顶到栈底的元素依次为：
> S.push(5);
> S.push(4);
> S.pop();
> S.push(2);
> S.pop();
> S.pop();
> S.push(1)
> A. 5, 4, 2, 1
> B. 1, 2, 4, 5
> C. 1
> D. 5, 4

Solution 1. 正确答案是 *C*。

详细解答：

我们一步步追踪栈 *S* 的状态。栈的特点是后进先出 *(LIFO)*。我们将栈顶表示在列表的右侧。

(1) 初始状态： *S = [] (空栈)*

***(2) S.push(5)*：** 元素 *5* 入栈。*S = [5] (栈底 -> 栈顶: 5)*

***(3) S.push(4)*：** 元素 *4* 入栈。*S = [5, 4] (栈底 -> 栈顶: 5, 4)*

***(4) S.pop()*：** 栈顶元素 *4* 出栈。*S = [5] (栈底 -> 栈顶: 5)*

***(5) S.push(2)*：** 元素 *2* 入栈。*S = [5, 2] (栈底 -> 栈顶: 5, 2)*

***(6) S.pop()*：** 栈顶元素 *2* 出栈。*S = [5] (栈底 -> 栈顶: 5)*

***(7) S.pop()*：** 栈顶元素 *5* 出栈。*S = [] (空栈)*

***(8) S.push(1)*：** 元素 *1* 入栈。*S = [1] (栈底 -> 栈顶: 1)*

操作完成后，栈 *S* 中只有一个元素 *1*。从栈顶到栈底的元素依次为：*1*。

因此，选项 *C* 是正确的。

64

> 当扫描到一个左括号时：
> A. 出栈
> B. 进栈
> C. 跳过该字符
> D. 算法结束

Solution 2. 正确答案是 *B*。

详细解答：

这个问题通常出现在处理算术表达式、括号匹配或类似的算法场景中，这些算法通常会使用栈来辅助操作。

当扫描到一个左括号 *'('* 时，一般的处理规则是将其压入栈中。

- **表达式求值/转换（如中缀转后缀）**：左括号通常被压入栈，以标记一个子表达式的开始。它会停留在栈中，直到遇到对应的右括号。
- **括号匹配**：左括号被压入栈。当遇到右括号时，会从栈顶弹出一个元素进行匹配。如果栈顶不是对应的左括号，则表示括号不匹配。

具体分析选项：

- ***A.* 出栈**：出栈操作通常与遇到右括号或操作符优先级处理相关，而不是左括号。
- ***B.* 进栈**：这是处理左括号的标准操作。
- ***C.* 跳过该字符**：跳过左括号会导致无法正确处理表达式的结构或括号的匹配关系。
- ***D.* 算法结束**：扫描到左括号通常是算法处理过程的一部分，而不是结束的标志。

因此，当扫描到一个左括号时，正确的操作是将其进栈。

65

> 3,1,2,4 是否是 1,2,3,4 的栈混洗?
>
> A. 是
>
> B. 不是

Solution 3. 正确答案是 *B (不是)*。

详细分析：

我们要判断序列 $P = \{3, 1, 2, 4\}$ 是否可以作为输入序列 $I = \{1, 2, 3, 4\}$ 的一个栈混洗（也称为栈置换或出栈序列）。这意味着我们尝试使用一个栈，按照以下规则操作，从输入序列 I 生成输出序列 P：

(1) 按顺序从输入序列 I 中取出元素。

(2) 取出的元素可以被压入栈中。

(3) 栈顶的元素可以被弹出，并作为输出序列 P 的下一个元素。

我们来模拟这个过程，尝试生成序列 $P = \{3, 1, 2, 4\}$：

设栈为 S。输入序列 $I = (1, 2, 3, 4)$。目标输出序列 $P = (3, 1, 2, 4)$。

(1) **期望输出** $P_1 = 3$：

- 从 I 中取出 *1*，压入 S。$S = [1]$ *(栈底在左，栈顶在右)*
- 从 I 中取出 *2*，压入 S。$S = [1, 2]$
- 从 I 中取出 *3*，压入 S。$S = [1, 2, 3]$
- 栈顶元素是 *3*，与 P_1 匹配。从 S 弹出 *3*。 输出的第一个元素是 *3*。 $S = [1, 2]$

(2) **期望输出** $P_2 = 1$：

- 当前栈 $S = [1, 2]$。栈顶元素是 *2*。
- 我们期望输出的下一个元素是 *1*。
- 由于栈顶是 *2 (不等于 1)*，我们不能直接弹出 *1*。
- 输入序列 I 中还剩下元素 *4*。如果我们将 *4* 压入栈，S 变为 $[1, 2, 4]$，栈顶是 *4*，仍然不是 *1*。
- 元素 *1* 确实在栈中，但它在元素 *2* 的下方。要使元素 *1* 出栈，必须先将元素 *2* 从栈中弹出。
- 如果我们此时弹出元素 *2*，那么输出序列将变为 $(3, 2, \ldots)$。
- 这与目标输出序列 $(3, 1, \ldots)$ 的第二个元素不符。

由于在生成第一个元素 *3* 之后，无法在不违反栈操作规则（即必须先弹出栈顶元素）的情况下使得下一个输出元素为 *1*，因此序列 $\{3,1,2,4\}$ 不可能是序列 $\{1,2,3,4\}$ 的栈混洗。

另一种判断方法是检查是否存在一个"禁止模式"。对于输入序列 $1,2,\ldots,n$，如果输出序列中存在三个元素 x,y,z（它们在输入序列中的原始顺序是 $x<y<z$），并且它们在输出序列中出现的顺序是 $z\ldots x\ldots y$（即 z 先出现，然后是 x，然后是 y），那么这个输出序列不可能是合法的栈混洗。在我们的例子中：输入序列是 $(1,2,3,4)$。输出序列是 $(3,1,2,4)$。考虑元素 $x=1,y=2,z=3$。它们在输入序列中的顺序是 $1\to2\to3$。它们在输出序列 $(3,1,2,4)$ 中的顺序是 3 *(在位置 1)*，然后是 1 *(在位置 2)*，然后是 2 *(在位置 3)*。这形成了 $z\ldots x\ldots y$ 的模式。因此，这也不是一个合法的栈混洗。

66

> 长度为 4 的序列共有多少个不同的栈混洗?

Solution 4. 答案：*14*

详细解答：

对于一个长度为 n 的输入序列（例如 $\{1,2,\ldots,n\}$），其所有可能的不同栈混洗（出栈序列）的数量由第 n 个卡特兰数 *(Catalan number)* 给出，记为 C_n。

卡特兰数的计算公式为：

$$C_n=\frac{1}{n+1}\binom{2n}{n}=\frac{(2n)!}{(n+1)!n!}$$

在本题中，序列的长度为 $n=4$。我们需要计算 C_4：

$$C_4=\frac{1}{4+1}\binom{2\times4}{4}=\frac{1}{5}\binom{8}{4}$$

首先，计算组合数 $\binom{8}{4}$：

$$\binom{8}{4}=\frac{8!}{4!(8-4)!}=\frac{8!}{4!4!}$$

$$\binom{8}{4}=\frac{8\times7\times6\times5\times4\times3\times2\times1}{(4\times3\times2\times1)(4\times3\times2\times1)}$$

$$\binom{8}{4}=\frac{8\times7\times6\times5}{4\times3\times2\times1}=\frac{1680}{24}=70$$

然后，将 $\binom{8}{4}=70$ 代入 C_4 的公式：

$$C_4=\frac{1}{5}\times70=14$$

因此，长度为 *4* 的序列共有 *14* 个不同的栈混洗。

卡特兰数的前几项为：$C_0=1$ $C_1=1$ $C_2=2$ *(对于序列 1,2，栈混洗有 1,2 和 2,1)* $C_3=5$ *(对于序列 1,2,3，栈混洗有 1,2,3, 1,3,2, 2,1,3, 2,3,1, 3,2,1)* $C_4=14$

67

> 利用栈结构进行中缀表达式求值，什么时候进行实际的运算?
> A. 每遇到一个新的操作数
> B. 每遇到一个新的操作符
> C. 当前的操作符比栈顶的操作符优先级高
> D. 当前的操作符比栈顶的操作符优先级低

Solution 5. 正确答案是 *D*。

详细解答:

在利用栈结构进行中缀表达式求值的典型算法中（通常使用两个栈：一个操作数栈，一个操作符栈），实际的运算（如加减乘除）通常在以下情况下发生:

(1) **当遇到的当前操作符的优先级低于或等于栈顶操作符的优先级时（对于左结合操作符）:** 在这种情况下，栈顶的操作符具有更高或相等的优先级，意味着它应该先被计算。因此，从操作符栈中弹出栈顶操作符，从操作数栈中弹出相应的操作数，执行运算，并将结果压回操作数栈。这个过程会持续进行，直到栈顶操作符的优先级低于当前操作符，或者栈为空，或者栈顶是左括号。

(2) **当遇到右括号 ')' 时:** 右括号表示一个子表达式的结束。此时，应不断从操作符栈中弹出操作符并执行运算，直到遇到匹配的左括号 '(' 为止。左括号随后也从栈中弹出。

(3) **当整个表达式扫描完毕时:** 如果操作符栈中仍然有操作符，应依次弹出并执行运算，直到操作符栈为空。

现在我们分析给出的选项:

- ***A.* 每遇到一个新的操作数:** 遇到操作数时，通常是将其压入操作数栈，不直接进行运算。
- ***B.* 每遇到一个新的操作符:** 遇到新的操作符时，需要根据其与栈顶操作符的优先级关系来决定是将其压栈还是先执行栈顶的运算。并非每次都立即运算。
- ***C.* 当前的操作符比栈顶的操作符优先级高:** 如果当前操作符的优先级高于栈顶操作符，那么当前操作符应该被压入操作符栈，等待其操作数。此时不会执行栈顶的运算。例如，在 '$2 + 3 * 4$' 中，遇到 '*' 时，其优先级高于栈顶的 '+'，所以 '*' 被压栈。
- ***D.* 当前的操作符比栈顶的操作符优先级低:** 如果当前操作符的优先级低于栈顶操作符，这意味着栈顶的操作符（及其操作数）应该先进行计算。例如，在 '$2 * 3 + 4$' 中，遇到 '+' 时，其优先级低于栈顶的 '*'，所以先计算 '$2 * 3$'。这是进行实际运算的一个关键时机。

虽然更完整的条件是“当前操作符优先级低于或等于栈顶操作符优先级（对左结合操作符）”，但在给出的选项中，选项 *D* 描述了一个明确会触发运算的场景。当栈顶操作符的优先级确实高于当前扫描到的操作符时，栈顶的运算必须先执行。

因此，选项 *D* 是最合适的答案。

68

> 利用栈结构进行逆波兰表达式的求值算法中，什么时候进行一次实际的运算?
> A. 每遇到一个新的操作数
> B. 每遇到一个新的操作符
> C. 当前操作符优先级高于栈顶
> D. 当前操作符优先级低于栈顶

Solution 6. 正确答案是 *B*。

详细解答:

逆波兰表达式（*Reverse Polish Notation, RPN*），也称为后缀表达式，其求值算法通常使用一个栈。算法步骤如下:

(1) 从左到右扫描逆波兰表达式。

(2) 如果扫描到的是一个操作数，则将其压入栈中。

(3) 如果扫描到的是一个操作符，则从栈中弹出所需数量的操作数（对于二元操作符，通常是两个），执行该操作符所代表的运算，然后将运算结果压回栈中。

当整个表达式扫描完毕后，栈中唯一剩下的元素就是表达式的最终结果。

根据这个算法:

- ***A.* 每遇到一个新的操作数:** 当遇到操作数时，它被压入栈，不进行运算。
- ***B.* 每遇到一个新的操作符:** 当遇到操作符时，会从栈中取出操作数，并执行该操作符指定的运算。这是进行实际运算的时机。
- ***C.* 当前操作符优先级高于栈顶:** 逆波兰表达式的求值不涉及操作符优先级的比较。操作符按其出现的顺序直接应用于栈顶的操作数。
- ***D.* 当前操作符优先级低于栈顶:** 同上，操作符优先级在逆波兰表达式求值过程中不起作用。

因此，在利用栈结构进行逆波兰表达式求值的算法中，每当遇到一个新的操作符时，就会进行一次实际的运算。

69

> 思考一下如何将逆波兰表达式还原为中缀表达式呢？试将下列逆波兰表达式还原为中缀表达式:
> $12 + 34\hat{}*$
> A. (1-2)*3^4
> B. (1+2)*3^4
> C. (1+2)^3*4
> D. (1+3)*2^4

Solution 7. 正确答案是 *B*。

思考与转换方法:

将逆波兰表达式（后缀表达式）转换为中缀表达式，通常也使用栈结构。算法步骤如下:

(1) 初始化一个空栈。

(2) 从左到右扫描逆波兰表达式的每一个元素（操作数或操作符）。

(3) **如果当前元素是操作数:** 将其压入栈中。

(4) **如果当前元素是操作符:**

　(1) 从栈中弹出栈顶元素，作为右操作数（*operand2*）。

　(2) 再次从栈中弹出栈顶元素，作为左操作数（*operand1*）。

　(3) 构建一个新的字符串，形式为 *"(operand1 操作符 operand2)"*。注意，为了保证运算顺序的正确性，通常需要在子表达式两侧加上括号。

　(4) 将这个新构建的子表达式字符串压回栈中。

(5) 当整个逆波兰表达式扫描完毕后，栈中应该只剩下一个元素，这个元素就是转换后的中缀表达式。

对给定表达式进行转换: 逆波兰表达式为: *1 2 + 3 4 ^ **

栈的状态变化如下（栈顶在右侧）:

(1) *扫描到* ***1*** *(操作数): 栈: '["1"]'*

(2) *扫描到* ***2*** *(操作数): 栈: '["1", "2"]'*

(3) *扫描到* ***+*** *(操作符):*

- *弹出"2" (作为 operand2)*
- *弹出"1" (作为 operand1)*
- *构建子表达式: '(1 + 2)'*
- *压栈: '["(1+2)"]'*

栈: '["(1+2)"]'

(4) *扫描到* ***3*** *(操作数): 栈: '["(1+2)", "3"]'*

(5) *扫描到* ***4*** *(操作数): 栈: '["(1+2)", "3", "4"]'*

(6) *扫描到* ***^*** *(操作符, 代表乘方):*

- *弹出"4" (作为 operand2)*
- *弹出"3" (作为 operand1)*
- *构建子表达式: '(3 ^4)'*
- *压栈: '["(1+2)", "(3^4)"]'*

栈: '["(1+2)", "(3^4)"]'

(7) *扫描到* ***** *(操作符):*

- *弹出"(3^4)" (作为 operand2)*
- *弹出"(1+2)" (作为 operand1)*
- *构建子表达式: '((1+2) * (3^4))'*
- *压栈: '["((1+2)*(3^4))"]'*

栈: '["((1+2)(3^4))"]'*

表达式扫描完毕，栈中剩下的元素是 *'((1+2)*(3^4))'*。这个表达式可以简化书写为 *'(1+2)*3^4'*，因为乘方 *'^'* 的优先级通常高于乘法 *'*'*，所以 *'(3^4)'* 外的括号可以省略，而 *'(1+2)'* 的括号是必需的，以确保加法先于乘法执行。

对照选项：*A. (1-2)*3^4 (错误的操作符'-') B. (1+2)*3^4 (匹配) C. (1+2)^3*4 (运算顺序和操作数错误) D. (1+3)*2^4 (操作数错误)*

因此，还原后的中缀表达式是 *(1+2)*3^4*。

70

> 栈初始为空，依次经过以下操作：
> push(5);
> push(8);
> pop();
> push(5);
> top();
> push(1);
> push(3);
> pop();
> pop();
> push(2);

> 此时从栈顶到栈底依次为：
> A. 2, 5, 5
> B. 2, 3, 1
> C. 5, 5, 2
> D. 1, 3, 2

Solution 8. 正确答案是 *A*。

详细解答：

我们一步步追踪栈 *S* 的状态。栈的特点是后进先出 *(LIFO)*。我们将栈顶表示在列表的右侧（或者说，新元素加在右边，从右边弹出）。

(1) 初始状态： *S = [] (空栈)*
***(2) push(5)*：** *元素 5 入栈。S = [5] (栈底 → 栈顶: 5)*
***(3) push(8)*：** *元素 8 入栈。S = [5, 8] (栈底 → 栈顶: 5, 8)*
***(4) pop()*：** *栈顶元素 8 出栈。S = [5] (栈底 → 栈顶: 5)*
***(5) push(5)*：** *元素 5 入栈。S = [5, 5] (栈底 → 栈顶: 5, 5)*
***(6) top()*：** *查看栈顶元素。栈顶元素是 5。栈状态不变。S = [5, 5] (栈底 → 栈顶: 5, 5)*
***(7) push(1)*：** *元素 1 入栈。S = [5, 5, 1] (栈底 → 栈顶: 5, 5, 1)*
***(8) push(3)*：** *元素 3 入栈。S = [5, 5, 1, 3] (栈底 → 栈顶: 5, 5, 1, 3)*
***(9) pop()*：** *栈顶元素 3 出栈。S = [5, 5, 1] (栈底 → 栈顶: 5, 5, 1)*
***(10) pop()*：** *栈顶元素 1 出栈。S = [5, 5] (栈底 → 栈顶: 5, 5)*
***(11) push(2)*：** *元素 2 入栈。S = [5, 5, 2] (栈底 → 栈顶: 5, 5, 2)*

操作完成后，栈 *S* 中的元素，如果按照从栈底到栈顶的顺序是 *[5, 5, 2]*。题目要求"从栈顶到栈底依次为"，所以顺序应该是：*2, 5, 5*。

因此，选项 *A* 是正确的。

71

> 阅读下面函数 (其中 1≤x, y≤16)，指出其功能：
>
> ```
> char digits[]={'0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'};
> void convert(int y,int x){
>     if(x!=0){
>         convert(y,x/y);
>         printf("%c",digits[x%y]);
>     }
> }
> ```
>
> A. 打印十进制整数 x 的 y 进制表示
> B. 打印 x 进制整数 100 的 y 进制表示
> C. 打印十进制整数 y 的 x 进制表示
> D. 打印 y 进制整数 100 的 x 进制表示

Solution 9. 正确答案是 *A*。

详细分析：

该函数 *convert(int y, int x)* 是一个递归函数，用于将一个整数转换为特定进制的表示形式。

- *digits[]* 数组存储了 *0-15* 对应的字符表示，用于输出 *0-9* 和 *a-f*（对应十六进制）。
- 函数参数 y 代表目标进制的基数。
- 函数参数 x 代表要转换的十进制整数。
- 递归的终止条件是 *x == 0*。
- 在递归步骤中：
  (1) *convert(y, x/y)*: 递归调用自身，处理整数 x 除以基数 y 的商。这相当于处理更高位的数字。
  (2) *printf("%c", digits[x%y])*: 在递归调用返回后，打印整数 x 对基数 y 取余的结果所对应的字符。这个余数是当前最低位的数字。

由于 *printf* 语句在递归调用之后执行，这意味着数字是按照从高位到低位的顺序生成的（因为最深的递归对应最高位，它最先完成其 *printf* 之前的递归调用，然后当递归逐层返回时，较低位的数字被打印出来）。

举例说明：假设调用 *convert(2, 10)*，即把十进制数 *10* 转换为 *2* 进制。

(1) *convert(2, 10)*: $10 \neq 0$
- 调用 *convert(2, 10/2)* 即 *convert(2, 5)*
- *convert(2, 5)*: $5 \neq 0$
  - 调用 *convert(2, 5/2)* 即 *convert(2, 2)*
  - *convert(2, 2)*: $2 \neq 0$
    - 调用 *convert(2, 2/2)* 即 *convert(2, 1)*
    - *convert(2, 1)*: $1 \neq 0$
      - 调用 *convert(2, 1/2)* 即 *convert(2, 0)*
      - *convert(2, 0)*: $0 == 0$，函数返回。
      - *(从 convert(2,1) 返回后)* 打印 *digits[1%2]* 即 *digits[1] ('1')*。
    - *(从 convert(2,2) 返回后)* 打印 *digits[2%2]* 即 *digits[0] ('0')*。
  - *(从 convert(2,5) 返回后)* 打印 *digits[5%2]* 即 *digits[1] ('1')*。
- *(从 convert(2,10) 返回后)* 打印 *digits[10%2]* 即 *digits[0] ('0')*。

输出顺序将是*'1', '0', '1', '0'*，即*"1010"*，这是十进制数 *10* 的二进制表示。

因此，函数的功能是打印十进制整数 x 的 y 进制表示。这与选项 *A* " 打印十进制整数 x 的 y 进制表示" 相符。

72

> 对序列 2, 3, 5, 7, 11 进行栈混洗得到 3, 5, 2, 11, 7 的过程中用于中转的栈 S 进行的操作是:
> A. push, pop, pop, push, push, pop, push, push, pop, pop
> B. push, push, pop, pop, pop, pop, push, push, push, pop
> C. push, push, pop, push, pop, pop, push, pop, pop, pop
> D. push, push, pop, push, pop, pop, push, push pop, pop

Solution 10. 正确答案是 *D*。*(选项 D 中"push, push pop, pop" 应理解为"push, push, pop, pop")*

详细分析:

设输入序列为 $I = \{2, 3, 5, 7, 11\}$，目标输出序列为 $O = \{3, 5, 2, 11, 7\}$。我们使用一个栈 S 来模拟这个过程。栈顶在右侧。

***(1)* 目标输出** $O_1 = 3$*:*
- 从 I 取出 *2*，执行 *push(2)*。$S = [2]$。操作序列*: push*
- 从 I 取出 *3*，执行 *push(3)*。$S = [2, 3]$。操作序列*: push，push*
- 栈顶元素 *3* 与 O_1 匹配。执行 *pop()*。输出 *3*。$S = [2]$。操作序列*: push，push，pop*

***(2)* 目标输出** $O_2 = 5$*:*
- 栈顶元素 *2* 不等于 $O_2 = 5$。
- 从 I 取出 *5*，执行 *push(5)*。$S = [2, 5]$。操作序列*: push，push，pop，push*
- 栈顶元素 *5* 与 O_2 匹配。执行 *pop()*。输出 *5*。$S = [2]$。操作序列*: push，push，pop，push，pop*

***(3)* 目标输出** $O_3 = 2$*:*
- 栈顶元素 *2* 与 O_3 匹配。执行 *pop()*。输出 *2*。$S = []$ *(*空栈*)*。操作序列*: push，push，pop，push，pop，pop*

***(4)* 目标输出** $O_4 = 11$*:*
- 栈为空。
- 从 I 取出 *7*, 执行 *push(7)*。$S = [7]$。操作序列*: push，push，pop，push，pop，pop，push*
- 栈顶元素 *7* 不等于 $O_4 = 11$。
- 从 I 取出 *11*, 执行 *push(11)*。$S = [7, 11]$。操作序列*: push，push，pop，push，pop，pop，push，push*
- 栈顶元素 *11* 与 O_4 匹配。执行 *pop()*。输出 *11*。$S = [7]$。操作序列*: push，push，pop，push，pop，pop，push，push，pop*

***(5)* 目标输出** $O_5 = 7$*:*
- 栈顶元素 *7* 与 O_5 匹配。执行 *pop()*。输出 *7*。$S = []$ *(*空栈*)*。操作序列*: push，push，pop，push，pop，pop，push，push，pop，pop*

最终的操作序列为*: push，push，pop，push，pop，pop，push，push，pop，pop*。这与选项 *D (*在修正了可能的打印错误后*)* 相符。

73

> 1,2,3... i...j...k...n: 下列哪一个序列一定不是 1,2,3⋯i⋯j⋯k⋯n 的栈混洗:
> A. ⋯i⋯j⋯k⋯
> B. ⋯k⋯j⋯i⋯
> C. ⋯k⋯i⋯j⋯
> D. ⋯j⋯k⋯i⋯

Solution 11. 正确答案是 C。

详细分析:

一个序列 P 不是输入序列 I 的栈混洗，有一个著名的判断条件（禁忌模式）: 如果输入序列 I 中有三个元素 x, y, z 依次出现（即 x 在 y 之前，y 在 z 之前），那么在任何合法的栈混洗输出序列 P 中，不可能出现 z 先输出，然后 x 输出，最后 y 输出的相对顺序。也就是说，子序列 $z \dots x \dots y$ 是被禁止的。

在题目中，输入序列是 $\{1, 2, 3, \dots, i, \dots, j, \dots, k, \dots, n\}$。这意味着元素 i, j, k 在输入流中出现的顺序是 $i \to j \to k$。

根据上述禁忌模式，令 $x = i, y = j, z = k$。那么，任何包含 $k \dots i \dots j$ 作为子序列（保持此相对顺序）的输出序列，一定不是一个合法的栈混洗。

我们来分析各个选项中 i, j, k 的相对顺序：

- ***A. ···i···j···k···*** 相对顺序是 $i \to j \to k$。这是可能的。例如：依次将 i, j, k 入栈并立即出栈。*(push i, pop i; push j, pop j; push k, pop k)*
- ***B. ···k···j···i···*** 相对顺序是 $k \to j \to i$。这是可能的。例如：依次将 i, j, k 入栈，然后依次出栈。*(push i, push j, push k; pop k, pop j, pop i)*
- ***C. ···k···i···j···*** 相对顺序是 $k \to i \to j$。这符合我们上面讨论的禁忌模式 $z \dots x \dots y$（其中 $x = i, y = j, z = k$）。为了更具体地理解为什么这是不可能的：
  - ***(1)*** 要想首先输出 k，元素 i, j, k 必须都已经被压入栈中（或者 i, j 在栈中，k 刚被压入并立即弹出）。假设 i, j, k 都已按顺序入栈，栈的状态（从底到顶）为 $[\dots, i, j, k]$。
  - ***(2)*** 此时 k 出栈。输出序列得到 k。栈变为 $[\dots, i, j]$。
  - ***(3)*** 接下来，我们希望输出 i。但是，元素 j 在栈中位于 i 的上方。根据栈的 *LIFO* 原则，必须先将 j 弹出，然后才能弹出 i。
  - ***(4)*** 如果 j 先弹出，那么输出序列就会变成 $k \dots j \dots$，而不是期望的 $k \dots i \dots$。

  因此，序列 $\dots k \dots i \dots j \dots$ 一定不是合法的栈混洗。
- ***D. ···j···k···i···*** 相对顺序是 $j \to k \to i$。这是可能的。例如：*(push i; push j; pop j; push k; pop k; pop i)* 这样得到的输出子序列是 j, k, i。

综上所述，序列$\cdots k \cdots i \cdots j \cdots$一定不是给定输入序列的栈混洗。

74

> 以下几个量中相等的是:
> 1 不同的 n 位二进制数个数
> 2 对小括号所能构成的合法括号匹配个数
> 3 1, 2 .. n 的不同栈混洗个数
> 4 含 n 个运算符的中缀表达式求值过程中运算符栈 push 操作的次数
> A. 12
> B. 23
> C. 34
> D. 24

Solution 12. 正确答案是 B。

详细分析： 我们逐个分析这四个量，假设 n 是一个正整数。

1 **不同的 *n* 位二进制数个数：** 一个 n 位的二进制数，每一位都有 *2* 种选择（*0* 或 *1*）。因此，总共有 $2 \times 2 \times \dots \times 2$ *(n 次)* $= 2^n$ 个不同的 *n* 位二进制数。

2 **对小括号所能构成的合法括号匹配个数：** 这指的是用 *n* 对小括号（即 *n* 个左括号和 *n* 个右括号）可以形成的合法匹配序列的数量。这个数量由第 *n* 个卡特兰数 C_n 给出。$C_n = \frac{1}{n+1}\binom{2n}{n}$。

3 ***1, 2 .. n*** **的不同栈混洗个数：** 一个包含 *n* 个不同元素的序列（如 *1, 2, ..., n*）通过一个栈进行操作，所有可能的出栈序列（栈混洗）的数量也是由第 *n* 个卡特兰数 C_n 给出。

4 **含 *n* 个运算符的中缀表达式求值过程中运算符栈 *push* 操作的次数：** 在标准的中缀表达式求值算法中（例如使用迪克斯特拉的双栈算法或改进版），表达式中的每一个运算符最终都会被压入操作符

栈一次。如果一个中缀表达式含有 n 个运算符，那么这 n 个运算符中的每一个都会被执行一次 *push* 操作到操作符栈。因此，这个次数是 n。(注意：左括号也会被压入操作符栈，但题目关注的是由 n 个运算符引起的 *push* 次数，或者说，如果只考虑运算符本身的 *push*，则是 n 次)。

总结各个量：

- 量 *1* = 2^n
- 量 *2* = C_n
- 量 *3* = C_n
- 量 *4* = n

比较这些量，我们可以看到量 *2* 和量 *3* 都等于第 n 个卡特兰数 C_n，因此它们是相等的。

- 2^n 通常不等于 C_n *(*例如, $n = 3$, $2^3 = 8$, $C_3 = 5$*)*。
- C_n 通常不等于 n *(*例如, $n = 3$, $C_3 = 5$, $n = 3$*;* 它们仅在 $n = 1$ 和 $n = 2$ 时碰巧相等*)*。

因此，只有量 *2* 和量 *3* 是普遍相等的。所以选项 *B (23)* 是正确的。

75

> 在中缀表达式求值中，某时刻运算数栈从栈顶到栈底依次为: 6，2，1 运算符栈从栈顶到栈底依次为: x, +, (剩下的待处理表达式为:)/(4 × 5 −7) 在接下来的过程中运算符的入栈顺序以及最终的计算结果分别为：
>
> A. /(x-最终结果为 8
>
> B. /(x-) 最终结果为 8
>
> C. /(x-最终结果为 1
>
> D. /(x-) 最终结果为 1

Solution 13. 正确答案是 *C*。

详细分析:

假设题目中的 '*x*' 代表乘法运算符 '*'。初始状态：

- 运算数栈 *(S_num)*，从栈底到栈顶*: '[1, 2, 6]'* (栈顶是 *6*)
- 运算符栈 *(S_op)*，从栈底到栈顶*: '[(, +, *]'* (栈顶是 '*')
- 待处理表达式 *(Rem): ')/(4 * 5 −7)'*

追踪步骤：

(1) 读取 *Rem* 的第一个字符*: ')'.*

  - 遇到右括号，处理 *S_op* 直到 '('.
  - *Pop '*' from S_op. Pop '6', '2' from S_num.* $2 * 6 = 12$*. Push '12' to S_num. S_num: '[1, 12]'. S_op: '[(, +]'.*
  - *Pop '+' from S_op. Pop '12', '1' from S_num.* $1 + 12 = 13$*. Push '13' to S_num. S_num: '[13]'. S_op: '[(]'.*
  - *Pop '(' from S_op. S_op: '[]'.*

  此时，"接下来的过程" 中被推入运算符栈的运算符序列 *(PushedOps)* 为空。*Rem* 变为 *'/(4 * 5 −7)'.*

(2) 读取 *Rem* 的 *'/'. Push '/' to S_op. S_op: '[/]'. PushedOps: '[/]'.*

(3) 读取 *Rem* 的 *'('. Push '(' to S_op. S_op: '[/, (]'. PushedOps: '[/, (]'.*

(4) 读取 *Rem* 的 *'4'. Push '4' to S_num. S_num: '[13, 4]'.*

(5) 读取 *Rem* 的 *'*' (*即 *'x')*. *S_op* 项为 *'('. '*'* 优先级高. *Push '*' to S_op. S_op: '[/, (, *]'. PushedOps: '[/, (, *]'.*

(6) 读取 *Rem* 的 *'5'. Push '5' to S_num. S_num: '[13, 4, 5]'.*

(7) 读取 *Rem* 的 *'−'. S_op* 项为 *'*'. '−'* 优先级低.

- *Pop '*'. Pop '5', '4'.* $4 * 5 = 20$. *Push '20' to S_num. S_num: '[13, 20]'. S_op: '[/, (]'.*

S_op 项为 *'('. '−'* 优先级高. *Push '−' to S_op. S_op: '[/, (, −]'. PushedOps: '[/, (, *, −]'.*

(8) 读取 *Rem* 的 *'7'. Push '7' to S_num. S_num: '[13, 20, 7]'.*

(9) 读取 *Rem* 的 *')'.*

- *Pop '−'. Pop '7', '20'.* 20–7 = 13. *Push '13' to S_num. S_num: '[13, 13]'. S_op: '[/, (]'.*
- *Pop '('. S_op: '[/]'.*

(10) *Rem* 为空. 处理 *S_op.*

- *Pop '/'. Pop '13', '13'.* $13/13 = 1$. *Push '1' to S_num. S_num: '[1]'. S_op: '[]'.*

最终结果为 *'1'. "* 接下来的过程中运算符的入栈顺序 *"* 为 *'/, (, *, −' (*即选项中的 *'/(x-'*，其中 *'x'* 为 *'*')*。这与选项 *C "/(x-*最终结果为 *1 "* 相符。

76

> (1+2×3!)/(4×5-7) 的逆波兰表达式为 (表达式中的整数都是一位数)

Solution 14. ***逆波兰表达式 (后缀表达式) 为:*** *1 2 3 ! × + 4 5 × 7 − /*

详细转换步骤:

我们将使用标准的运算符优先级（从高到低）:

(1) 阶乘 *'¡*

(2) 乘法 *'×'*，除法 *'/' (*同级，从左到右*)*

(3) 加法 *'+'*，减法 *'-' (*同级，从左到右*)*

(4) 括号 *'()'* 具有最高优先级，改变运算顺序。

原中缀表达式为*: (1+2×3!)/(4×5−7)*

我们可以将其看作是 *Numerator / Denominator* 的形式。*Numerator (*分子*):*（*1+2×3!*）*Denominator (*分母*):*（*4×5−7*）

1. **转换分子**（*1+2×3!*）

- 首先处理内部最高优先级的运算 *'3¡*。中缀*: '3¡* 后缀*: '3 ¡*
- 接下来是 *'2×3¡*，可以看作 *'2 × (3!)'*。中缀*: '2 × 3¡* 后缀 *(*将 *'3¡* 的后缀代入*): '2 (3 !) ×' → 2 3 ! ×*
- 然后是 *'1 + (2×3!)'*。中缀*: '1 + 2×3¡* 后缀 *(*将 *'2×3¡* 的后缀代入*): '1 (2 3 ! ×) +' → 1 2 3 ! × +*

所以，分子的后缀表达式是*: 1 2 3 ! × +*

2. **转换分母**（*4×5−7*）

- 首先处理 *'4×5'*。中缀*: '4×5'* 后缀*: '4 5 ×'*
- 然后是 *'(4×5) - 7'*。中缀*: '4×5 - 7'* 后缀 *(*将 *'4×5'* 的后缀代入*): '(4 5 ×) 7 -' → 4 5 × 7 −*

所以，分母的后缀表达式是*: 4 5 × 7 −*

3. **组合分子和分母** 原表达式是 *Numerator / Denominator*。后缀形式为*: (*后缀 *Numerator) (*后缀 *Denominator) /* 代入已转换的后缀表达式:（*1 2 3 ! × +*）（*4 5 × 7 −*）*/ → 1 2 3 ! × + 4 5 × 7 − /*

使用 *Shunting-yard* 算法（调度场算法）验证: 输入*:*（ *1 + 2 × 3 !* ）*/*（ *4 × 5 − 7* ）输出队列 *(Q):* 运算符栈 *(S):*

Token	*Action*	*Q*	*S*
(	Push (to S		(
1	Add 1 to Q	1	(
+	Push + to S (S top is (, or lower prec)	1	(+
2	Add 2 to Q	1 2	(+
×	Push × to S (prec(×) > prec(+))	1 2	(+ ×
3	Add 3 to Q	1 2 3	(+ ×
!	Push ! to S (prec(!) > prec(×))	1 2 3	(+ × !
)	Pop S to Q until (found		
	Pop !	1 2 3 !	(+ ×
	Pop ×	1 2 3 ! ×	(+
	Pop +	1 2 3 ! × +	(
	Pop (	1 2 3 ! × +	
/	Push / to S	1 2 3 ! × +	/
(	Push (to S	1 2 3 ! × +	/ (
4	Add 4 to Q	1 2 3 ! × + 4	/ (
×	Push × to S (S top is (, or lower prec)	1 2 3 ! × + 4	/ (×
5	Add 5 to Q	1 2 3 ! × + 4 5	/ (×
-	Pop S to Q (prec(-) $\leq$ prec(×))		
	Pop ×	1 2 3 ! × + 4 5 ×	/ (
	Push - to S (S top is (, or lower prec)	1 2 3 ! × + 4 5 ×	/ (-
7	Add 7 to Q	1 2 3 ! × + 4 5 × 7	/ (-
)	Pop S to Q until (found		
	Pop -	1 2 3 ! × + 4 5 × 7 -	/ (
	Pop (	1 2 3 ! × + 4 5 × 7 -	/
End	Pop all from S to Q		
	Pop /	1 2 3 ! × + 4 5 × 7 - /	

最终的逆波兰表达式为: 1 2 3 ! × + 4 5 × 7 - /