【模拟】-TLV编码

题目描述与示例

题目

TLV 编码是按 TagLengthValue 格式进行编码的。

一段码流中的信元用 `tag` 标识， `tag` 在码流中唯一不重复， `length` 表示信元 `value` 的长度， `value` 表示信元的值，码流以某信元的 `tag` 开头， `tag` 固定占一个字节， `length` 固定占两个字节，字节序为**小端序**。

现给定 TLV 格式编码的码流以及需要解码的信元 `tag` ，请输出该信元的 `value` 。

输入码流的 16 进制字符中，不包括小写字母；且要求输出的 16 进制字符串中也不要包含小写字母；码流字符串的最大长度不超过 `50000` 个字节。

输入描述

第一行为第一个字符串 ，表示待解码信元的 `tag` ；输入第二行为一个字符串， 表示待解码的 16 进制码流；字节之间用 `空格` 分割。

输出描述

输出一个字符串，表示待解码信元以 16 进制表示的 `value` 。

示例

输入

```
31
32 01 00 AE 90 02 00 01 02 30 03 00 AB 32 31 31 02 00 32 33 33 01 00 CC
```

输出

```
32 33
```

说明

需要解析的信源的 tag 是 `31`；从码流的起始处开始匹配：

第一个信元的tag 为 `32` ，其长度为 `1` (`01 00` ，小端序表示为十六进制的 `0001` ，十进制为 `1`)，匹配接下来的 `1` 个字节 `AE` ；

第二个信元的 tag 为 `90` 其长度为 `2` (`02 00` ，小端序表示为十六进制的 `0002` ，十进制为 `2` ，匹配接下来的 `2` 个字节 `01 02` ；

第三个信元的 tag 为 `30` 其长度为 `3` (`03 00` ，小端序表示为十六进制的 `0003` ，十进制为 `3` ，匹配接下来的 `3` 个字节 `AB 32 31` ；

第四个信元的 tag 为 `31` 其长度为 `2` (`02 00` ，小端序表示为十六进制的 `0002` ，十进制为 `2` ，匹配接下来的 `2` 个字节 `32 33` ；

所以返回长度后面的两个字节即可 为 `32 33` 。

解题思路

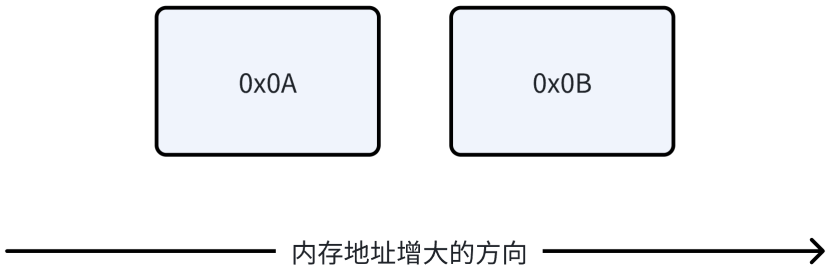
大端序和小端序

这道题涉及一个前置知识，大端序和小端序。

这两种排序都是**字节序**，表示的是数据字节在内存中存储方式。

大端序（Big-Endian）指的是数据的低位字节存放在内存的高位地址，高位字节存放在内存的低位地址。这种排列方式与数据用字节表示时的书写顺序一致，**符合人类的阅读习惯**。

小端序（Little-Endian）指的数据的低位字节存放在内存的低位地址，高位字节存放在高位地址。**小端序与人类的阅读习惯相反，但更符合计算机读取内存的方式**，因为CPU读取内存中的数据时，是从低地址向高地址方向进行读取的。



大端序阅读这两个字节，从左到右，得到16进制的0A0B，表示10进制的2571

小端序阅读这两个字节，从右到左，得到16进制的0B0A，表示10进制的2826

看起来有点拗口，但只需要记住**大端序从左到右排列符合人类阅读习惯，小端序从右到左排列不符合人类阅读习惯即可。**

TLV码流的解码过程

下图显示了示例的码流的解码过程

- 蓝绿红构成的一整段内容表示一个信元，其中
  - 蓝色表示信元的唯一标识 `tag`
  - 绿色表示信元的 `value` 的长度
  - 红色表示信元的 `value` 的的内容

```
32 01 00 AE 90 02 00 01 02 30 03 00 AB 32 31 31 02 00 32 33 33 01 00 CC
```

我们要做的事情，就是找到 `tag` 为所第一行输入的 `target_tag` 的那个信元，所对应的 `value` 的内容。

模拟过程

本题剩余内容，只需要按照题目要求进行模拟即可。

代码

Python

```python
# 题目：2024E-TLV编码
# 分值：100
# 作者：许老师-闭着眼睛学数理化
# 算法：字符串模拟
# 代码看不懂的地方，请直接在群上提问


# 构建一个用于解码的辅助函数
def help(stream, idx):
    cur_tag = stream[idx]
    # 信元长度length为小端序排布，得到十六进制的字符串结果
    cur_length_hex = stream[idx+2] + stream[idx+1]
    # 将十六进制转换为十进制
    cur_length = int(cur_length_hex, base = 16)
    return cur_tag, cur_length


# 待解码信元的tag
target_tag = input()
# 待解码的编码流
stream = input().split()

# idx是stream流中的索引
idx = 0

# 进行循环，循环条件为idx对应的字符串不是目标信元target_tag
while stream[idx] != target_tag:
    # 根据当前的idx，获得当前信元的tag和长度
    cur_tag, cur_length = help(stream, idx)
    # 信元和长度一共占3位，信元中的信息占cur_length位
    # 下一个信元tag的位置位于 idx + 3 + cur_length 处
    # 更新idx
    idx += 3 + cur_length

# 退出循环后再进行一次信元解码
# 此时得到的cur_tag一定是target_tag
cur_tag, cur_length = help(stream, idx)

# 此时stream中，从 idx+3 到 idx+3+cur_length 位置的切片
# 即为信元target_tag解码得到的字节
print(" ".join(stream[idx+3:idx+3+cur_length]))
```

Java

```java
import java.util.*;

public class Main {

    // 构建一个用于解码的辅助函数
    public static String[] decodeHelper(String[] stream, int idx) {
        String curTag = stream[idx];
        // 信元长度length为小端序排布，得到十六进制的字符串结果
        String curLengthHex = stream[idx + 2] + stream[idx + 1];
        // 将十六进制转换为十进制
        int curLength = Integer.parseInt(curLengthHex, 16);
        return new String[] { curTag, String.valueOf(curLength) };
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 待解码信元的tag
        String targetTag = scanner.nextLine();
        // 待解码的编码流
        String[] stream = scanner.nextLine().split(" ");

        // idx是stream流中的索引
        int idx = 0;

        // 进行循环，循环条件为idx对应的字符串不是目标信元targetTag
        while (!stream[idx].equals(targetTag)) {
            // 根据当前的idx，获得当前信元的tag和长度
            String[] result = decodeHelper(stream, idx);
            String curTag = result[0];
            int curLength = Integer.parseInt(result[1]);

            // 信元和长度一共占3位，信元中的信息占curLength位
            // 下一个信元tag的位置位于 idx + 3 + curLength 处
            idx += 3 + curLength;
        }

        // 退出循环后再进行一次信元解码
        // 此时得到的curTag一定是targetTag
        String[] result = decodeHelper(stream, idx);
        String curTag = result[0];
        int curLength = Integer.parseInt(result[1]);
```

```
43
44          // 输出信元的字节
45          System.out.println(String.join(" ", Arrays.copyOfRange(stream, idx +
    3, idx + 3 + curLength)));
46
47          scanner.close();
48      }
49  }
50
```

C++

```
1  #include <iostream>
2  #include <vector>
3  #include <sstream>
4
5  using namespace std;
6
7  // 构建一个用于解码的辅助函数
8  pair<string, int> decodeHelper(const vector<string>& stream, int idx) {
9      string cur_tag = stream[idx];
10     // 信元长度length为小端序排布，得到十六进制的字符串结果
11     string cur_length_hex = stream[idx + 2] + stream[idx + 1];
12     // 将十六进制转换为十进制
13     int cur_length = stoi(cur_length_hex, nullptr, 16);
14     return {cur_tag, cur_length};
15 }
16
17 int main() {
18     string target_tag;
19     getline(cin, target_tag);
20
21     string line;
22     getline(cin, line);
23
24     // 将输入的编码流分割并存入vector中
25     vector<string> stream;
26     stringstream ss(line);
27     string token;
28     while (ss >> token) {
29         stream.push_back(token);
30     }
31
32     // idx是stream流中的索引
33     int idx = 0;
```

```cpp

    // 进行循环，循环条件为idx对应的字符串不是目标信元target_tag
    while (stream[idx] != target_tag) {
        // 根据当前的idx，获得当前信元的tag和长度
        auto [cur_tag, cur_length] = decodeHelper(stream, idx);

        // 信元和长度一共占3位，信元中的信息占cur_length位
        // 下一个信元tag的位置位于 idx + 3 + cur_length 处
        idx += 3 + cur_length;
    }

    // 退出循环后再进行一次信元解码
    // 此时得到的cur_tag一定是target_tag
    auto [cur_tag, cur_length] = decodeHelper(stream, idx);

    // 输出信元的字节
    for (int i = idx + 3; i < idx + 3 + cur_length; ++i) {
        cout << stream[i] << " ";
    }
    cout << endl;

    return 0;
}

```

时空复杂度

时间复杂度：`O(N)`。需要一次遍历码流数组`stream`。

空间复杂度：`O(1)`。进入若干长度变量