对于 KMP 算法中间的部分记忆数组做以下约定：

- 模版串 s 长度为 n，模式串 t 长度为 m，下标均从 1 开始
- `next[]` 数组简记为 `ne[]`
- `ne[j]` 存储的信息为模式串中「以 `t[j]` 结尾的非平凡前缀和非平凡后缀的最大匹配长度」即 `t[1 : ne[j]] = t[j - ne[j] + 1 : j]`

作业四

T1

> 计算下列串的next数组：
>
> (1) "ABCDEFG"
>
> (2) "AAAAAAAA"
>
> (3) "BABBABAB"
>
> (4) "AAAAAAB"
>
> (5) "ABCABDAAABC"
>
> (6) "ABCABDABEABCABDABF"
>
> (7) "ABBACXY"

我们直接编程实现。

```cpp
vector<int> getNext(const string& child) {
    int m = child.size();
    vector<int> ne(m + 1);
    string t = " " + child;

    for (int i = 2, j = 0; i <= m; i++) {
        while (j && t[i] != t[j + 1]) {
            j = ne[j];
        }
        if (t[i] == t[j + 1]) {
            j++;
        }
        ne[i] = j;
    }
```

```
15
16      return ne;
17  }
```

最终计算可得：

(1) "ABCDEFG" = `0 0 0 0 0 0 0`

(2) "AAAAAAAA" = `0 1 2 3 4 5 6 7`

(3) "BABBABAB" = `0 0 1 1 2 3 2 3`

(4) "AAAAAAB" = `0 1 2 3 4 5 0`

(5) "ABCABDAAABC" = `0 0 0 1 2 0 1 1 1 2 3`

(6) "ABCABDABEABCABDABF" = `0 0 0 1 2 0 1 2 0 1 2 3 4 5 6 7 8 0`

(7) "ABBACXY" = `0 0 0 1 0 0 0`

T2

> 要求输入两个字符串s和t，统计s包含串t的个数。

可以直接 $O(mn)$ 暴力匹配统计，也可以用 KMP 优化到 $O(n+m)$。

```
1  int countKMP(const string& s = "abccabaccaba", const string& t = "ab") {
2      int cnt = 0;
3      int n = s.size(), m = t.size();
4      string news = " " + s;
5      string newt = " " + t;
6      vector<int> ne = getNext(t);
7
8      for (int i = 1, j = 0; i <= n; i++) {
9          while (j && news[i] != newt[j + 1]) {
10             j = ne[j];
11         }
12         if (news[i] == newt[j + 1]) {
13             j++;
14         }
15         if (j == m) {
16             cnt++;
17             j = ne[j];
18         }
```

```
    }

    return cnt;
}
```

T3

> 编写从串s中删除所有与串t相同的子串的算法

匹配的逻辑相同，可以暴力也可以 KMP。至于删除，没有必要在模板串 s 上进行，可以重新构造一个答案串，构造方法比较简单，利用一个变量 l 存储「不需要删除的子串的起始下标」，后续匹配成功时将 $s[l:j-m+1]$ 拼接到答案串并更新 $l=i+1$ 即可。

```
string deleteKMP(const string& s = "abccabaccaba", const string& t = "ab") {
    string res;
    int n = s.size(), m = t.size();
    string news = " " + s;
    string newt = " " + t;
    vector<int> ne = getNext(t);

    int l = 1;
    for (int i = 1, j = 0; i <= n; i++) {
        while (j && news[i] != newt[j + 1]) {
            j = ne[j];
        }
        if (news[i] == newt[j + 1]) {
            j++;
        }
        if (j == m) {
            res += news.substr(l, i - m - l + 1);
            l = i + 1;
            j = ne[j];
        }
    }

    // get possible end
    res += news.substr(l);

    return res.size() ? res : s;
}
```

T4

> 试给出求串s和串t的最大公共子串的算法

注：对于求解 LongestCommomSubstring 一类的问题。本章涉及到的 KMP 算法仅适用于 10^3 级别的数据量，更大的数据量需要使用别的算法，例如适用于 10^4 级别的动态规划算法和 10^5 级别的后缀数组 Suffix Array 算法。此处仅讨论前两个算法。

思路一：**枚举子串+KMP匹配**。不难想到我们枚举 t 串的左右端点来 $O(m^2)$ 的枚举出其所有子串，接着对每一个枚举出来的子串和 s 串进行 KMP 匹配统计。时间复杂度为 $O(m^2(n+m))$。

```cpp
vector<string> getLongestCommomSubstring_bf(const string& s = "abaadqbacaba", const string& t = "abac") {
    vector<string> res;

    int m = t.size();
    for (int len = m; len >= 1; len--) {
        bool ok = false;
        for (int i = 0; i < m - len + 1; i++) {
            string tt = t.substr(i, len);
            if (countKMP(s, tt)) {
                ok = true;
                res.push_back(tt);
            }
        }
        if (ok) {
            return res;
        }
    }

    return res;
}
```

思路二：**动态规划**。时间复杂度为 $O(nm)$，空间复杂度为 $O(nm)$

我们定义 `dp[i][j]` 表示 s 串以 `s[i]` 结尾的子串与 t 串以 `t[j]` 结尾的子串中最长公共子串的长度。不难发现状态转移方程就是：

- `dp[i][j] = dp[i - 1][j - 1] + 1`, if and only if $\mathrm{s[i] == t[j]}$
- `dp[i][j] = 0`, if and only if $\mathrm{s[i] != t[j]}$

评测 OJ：https://www.nowcoder.com/practice/f33f5adc55f444baa0e0ca87ad8a6aac

```cpp
vector<string> getLongestCommomSubstring_dp(const string& s = "abaadqbacaba", const string& t = "abac") {
    string news = " " + s;
    string newt = " " + t;
    int n = s.size(), m = t.size();

    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
    int ma = 0;

    // update dp table
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (news[i] == newt[j]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                ma = max(ma, dp[i][j]);
            }
        }
    }

    // find all longest common substrings
    unordered_map<string, bool> dict;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (dp[i][j] == ma) {
                dict[news.substr(i - ma + 1, ma)] = true;
            }
        }
    }

    // get result
    vector<string> res;
    for (auto& [lcs, _]: dict) {
        res.push_back(lcs);
    }

    return res;
}
```

T5

> 编写一个函数来颠倒单词在字符串里的出现顺序。例如，把字符串"Do or do not, there is no try. "转换为"try．no is there not do, or Do"。假设所有单词都以空格为分隔符，标点符号也当做字母来对待。请对你的设计思路做出解释，并对你的解决方案的执行效率进行评估。

思路一：原地解决。先将原字符串左右翻转，然后对其中每一个单词进行左右翻转即可。时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

```
string reverseOrigin(string s = "Do or do not, there is no try.") {
    int n = s.size();

    reverse(s.begin(), s.end());

    int i = 0;
    while (i < n) {
        int j = i;
        while (j < n && s[j] != ' ') {
            j++;
        }
        reverse(s.begin() + i, s.begin() + j);
        i = j + 1;
    }

    return s;
}
```

思路二：利用栈结构做一个中转。我们扫描一遍原字符串，将扫描到的单词按顺序入栈，最后从栈顶开始拼接答案字符串即可。时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

```
string reverseWithStack(const string& s = "Do or do not, there is no try.") {
    int n = s.size();

    // get words
    stack<string> stk;
    int i = 0;
    while (i < n) {
        int j = i;
        while (s[j] != ' ') {
            j++;
        }
        stk.push(s.substr(i, j - i));
```

```
        i = j + 1;
    }

    // splice result
    string res;
    while (stk.size()) {
        res += stk.top() + " ";
        stk.pop();
    }

    return res;
}
```

实验四

T1

实验题 4.1 实现串的模式匹配等算法。

（1）采用顺序存储方式存储串，建立两个字符串 s、t，利用 BF 算法求串 t 在串 s 中首次出现的位置。

（2）采用顺序存储方式存储串，建立两个字符串 s、t，利用 KMP 算法求串 t 在串 s 中首次出现的位置。

（3）采用顺序存储方式存储串，建立两个字符串 s、t，利用改进的 KMP 算法求串 t 在串 s 中首次出现的位置。

上述作业中已经使用了改进的 KMP 算法进行匹配了，下面给出三种匹配代码。其中，未改进的 KMP 算法就是**在一次失配后不再继续匹配长度更短的前缀**，而是直接重新开始。

BF 算法：

```
int bruteForce(const string& s = "abccabaccaba", const string& t = "aba") {
    string news = " " + s;
    string newt = " " + t;
    int n = s.size(), m = t.size();

    for (int i = 1; i <= n - m + 1; i++) {
        int j = 1;
        for (int k = i; j <= m; j++, k++) {
            if (news[k] != newt[j]) {
                break;
            }
        }
        if (j == m + 1) {
```

```
            return i - 1;
        }
    }

    return -1;
}
```

未改进的 KMP 算法：

```
int originalKMP(const string& s = "abccabaccaba", const string& t = "aba") {
    string news = " " + s;
    string newt = " " + t;
    int n = s.size(), m = t.size();

    Homework_4 obj;
    vector<int> ne = obj.getNext(newt);

    for (int i = 1, j = 0; i <= n; i++) {
        if (j && news[i] != newt[j + 1]) {
            j = ne[j];
        }
        if (news[i] == newt[j + 1]) {
            j++;
        }
        if (j == m) {
            return i - m;
        }
    }

    return -1;
}
```

改进的 KMP 算法：

```
int optimizedKMP(const string& s = "abccabaccaba", const string& t = "aba") {
    string news = " " + s;
    string newt = " " + t;
    int n = s.size(), m = t.size();

    Homework_4 obj;
    vector<int> ne = obj.getNext(newt);

```

```
    for (int i = 1, j = 0; i <= n; i++) {
        while (j && news[i] != newt[j + 1]) {
            j = ne[j];
        }
        if (news[i] == newt[j + 1]) {
            j++;
        }
        if (j == m) {
            return i - m;
        }
    }

    return -1;
}
```

T2

实验题 4.2 利用恺撒密码对文件进行加解密。

恺撒密码是一种置换密码，它的加密原理是将字母替换为它后面的另一个字母，从而起到加密作用。假如有这样一段明文"security"，用偏移量为 3 的恺撒密码加密后，密文为"vhfxulwb"。

这种加密方法可以依据移位的不同产生新的变化。将明文记为 ch，密文记为 c，位移量（密钥）记作 key，更具一般性的恺撒密码加密过程可记为如下的变换：

$$c \equiv (ch + key) \bmod n \quad （其中 key 为位移量，n 为基本字符个数）$$

同样，解密过程可表示为

$$ch \equiv (c - key + n) \bmod n \quad （其中 key 为偏移量，n 为基本字符个数）$$

基本要求：

（1）输入一段英文（字符串），采用恺撒密码加密成密文，偏移量（密钥）由用户输入。

（2）读入密文字符串，解密成明文，偏移量（密钥）由用户输入。

（3）用文件实现输入和输出。

核心就是一个字符的循环移位操作，解密可以复用加密的算法，例如加密对应偏移 dx 位，则解密对应偏移 $26 - dx$ 位。

加密算法：

```
void caesarCipher(string in = "Exp4_T4_in.txt", string out = "Exp4_T4_cipher.txt", int dx = 3) {
    string cwd = std::filesystem::current_path().string() + "\\Code\\chapter4_String\\";

    ifstream fin(cwd + in);
    ofstream fout(cwd + out);

```

```
    string s;
    while (getline(fin, s)) {
        string trans;
        for (char ch: s) {
            if (islower(ch)) {
                trans += (ch - 'a' + dx) % 26 + 'a';
            } else if (isupper(ch)) {
                trans += (ch - 'A' + dx) % 26 + 'A';
            } else {
                cerr << "invalid character\n";
                exit(1);
            }
        }
        fout << trans << "\n";
    }

    fin.close();
    fout.close();
}
```

解密算法：

```
void caesarDecipher(string cipher = "Exp4_T4_cipher.txt", string decipher = "Exp4_T4_decipher.txt", int dx = 3) {
    caesarCipher(cipher, decipher, 26 - dx);
}
```