Lesson1（数组模拟数据结构）

> **链表与邻接表（数组模拟）**
>
> 单链表&邻接表（n个链表）：邻接表主要用于存储图与树
>
> - e[N]:值　ne[N]:next
> - e与ne数组通过下标关联，空结点下标用-1表示，链表最后一个有效结点ne为-1

```cpp
#include <iostream>

using namespace std;

const int N = 100010;

// head表示头结点的下标
// e[i]表示结点i的值
// ne[i]表示结点i的后一个结点下标
// idx表示当前已经使用到了哪个点
int head, e[N], ne[N], idx;

//初始化
void init() {
    head = -1, idx = 0;
}

//将x插到头结点
void add_to_head(int x) {
    e[idx] = x, ne[idx] = head, head = idx++;
}

//将x插到第k个结点之后
void add(int k, int x) {
    e[idx] = x, ne[idx] = ne[k], ne[k] = idx++;
}

//将下标是k的点的后面的点删掉
void remove(int k) {
    ne[k] = ne[ne[k]];
}

int main(void) {
    int m;
    scanf("%d", &m);

    init();

    while (m--) {
        int k, x;
        char op;
        // scanf("%c", &op);
        cin >> op;

        if (op == 'H') {
            scanf("%d", &x);
            add_to_head(x);
        }
        else if (op == 'D') {
            scanf("%d", &k);
            if (!k) head = ne[head];
            else remove(k-1);
        }
        else {
            scanf("%d%d", &k, &x);
            add(k-1, x);
        }
    }

    for (int i=head; i!=-1; i=ne[i]) printf("%d ", e[i]);

    return 0;
}
```

双链表：优化某些特定问题

- 在单链表基础上增加指向前一个结点的指针

    - l[N], r[N]
- 不定义头结点与尾结点，0：head　1：tail

```
#include <iostream>

using namespace std;

const int N = 100010;

int l[N], r[N], e[N], idx;

//初始化双链表
void init() {
    //0表示最左端结点，1表示最右边结点，起标志作用，不能被删除或修改
    //idx从2开始
    r[0] = 1, l[1] = 0;
    idx = 2;
}

//在第k个插入的结点右侧插入一个结点
void add(int k, int x) {
    e[idx] = x;
    l[idx] = k, r[idx] = r[k];
    l[r[k]] = idx, r[k] = idx++;
}

//删除第k个插入的结点
void remove(int k) {
    r[l[k]] = r[k];
    l[r[k]] = l[k];
}

int main(void) {
    int m;
    scanf("%d", &m);

    init();

    while (m--) {
        int k, x;
        string op;
        cin >> op;

        if (op == "L") {
            scanf("%d", &x);
            add(0, x);
        }
        else if (op == "R") {
            scanf("%d", &x);
            add(l[1], x);
        }
        else if (op == "D") {
            scanf("%d", &k);
            remove(k+1);    //注意从第k个插入的数下标为k+1
        }
        else if (op == "IL") {
            scanf("%d%d", &k, &x);
            add(l[k+1], x);
        }
        else {
            scanf("%d%d", &k, &x);
            add(k+1, x);
        }
    }

    for (int i=r[0]; i!=1; i=r[i]) printf("%d ", e[i]);

    return 0;
}
```

邻接表（多个单链表）

- head[1]->...
- head[2]->...
- ...
- head[3]->...

栈（先进后出）

```
//stk表示栈，tt表示栈顶
int stk[N], tt;

//插入x
stk[++tt] = x;

//弹出栈顶
tt--;

//判断是否为空
if (tt > 0) not empty
else empty

//栈顶
stk[tt];
```

- e1:

```
#include <iostream>

using namespace std;

const int N = 100010;

int stk[N], tt;

int main(void) {
    int m;
    scanf("%d", &m);

    while (m--) {
        int x;
        string op;
        cin >> op;

        if (op == "push") {
            scanf("%d", &x);
            stk[++tt] = x;
        }
        else if (op == "pop") tt--;
        else if (op == "empty") {
            if (tt > 0) puts("NO");
            else puts("YES");
        }
        else printf("%d\n", stk[tt]);
    }

    return 0;
}
```

- e2: 表达式求值

```
#include <iostream>
#include <cstring>
#include <unordered_map>

using namespace std;

const int N = 100010;

char str[N];
int num[N];     //数字栈
char op[N];     //表达式栈
int tt1, tt2;

void eval() {
    int b = num[tt1--];
    int a = num[tt1--];
    char c = op[tt2--];

    int x = 0;
    if (c == '+') x = a+b;
    else if (c == '-') x = a-b;
    else if (c == '*') x = a*b;
    else if (c == '/') x= a/b;

    num[++tt1] = x;
```

```
}

int main(void) {
    scanf("%s", str);

    unordered_map<char, int> pr{{'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}};

    for (int i=0; str[i]; i++) {
        auto c = str[i];

        if (isdigit(c)) {
            int x = 0, j = i;
            while (str[j] && isdigit(str[j])) {
                x = x*10+str[j++]-'0';
            }

            num[++tt1] = x;
            i = j-1;
        }
        else if (c == '(') op[++tt2] = c;
        else if (c == ')') {
            while (op[tt2] != '(') eval();
            tt2--;
        }
        else {
            while (tt2 && op[tt2]!='(' && pr[op[tt2]]>=pr[c]) eval();
            op[++tt2] = c;
        }
    }

    while (tt2) eval();
    printf("%d", num[tt1]);

    return 0;
}
```

队列（先进先出）

```
//在队尾插入元素，在队头弹出元素
int q[N], hh, tt = -1;

//插入
q[++tt] = x;

//弹出
hh++;

//判断是否为空
if (hh <= tt) not empty
else emtpy

//取队头
q[hh]
```

- e1

```
#include <iostream>

using namespace std;

const int N = 100010;

int q[N], hh, tt = -1;

int main(void) {
    int m;
    cin >> m;

    while (m--) {
        int x;
        string op;
        cin >> op;

        if (op == "push") {
            scanf("%d", &x);
            q[++tt] = x;
```

```
        }
        else if (op == "pop") hh++;
        else if (op == "empty") {
            if (hh <= tt) puts("NO");
            else puts("YES");
        }
        else cout << q[hh] << endl;
    }

    return 0;
}
```

单调栈

常见模型：在一个序列中，找出每一个数左边离它最近的且满足某种性质（如 最大最小）的数的位置

先想暴力算法，再利用单调性进行优化

```
#include <iostream>

using namespace std;

const int N = 100010;

int stk[N], tt;

int main(void) {
 int m;
 scanf("%d", &m);

 while (m--) {
     int x;
     scanf("%d", &x);

     while (tt && stk[tt] >= x) tt--;
     if (tt) printf("%d ", stk[tt]);
     else printf("-1 ");

     stk[++tt] = x;
 }

 return 0;
}
```

单调队列

经典应用：求滑动窗口中最大值or最小值

```
#include <iostream>

using namespace std;

const int N = 1000010;

int n, k;
int a[N];
//q中存储的是下标
int q[N], hh, tt = -1;

int main(void) {
 scanf("%d%d", &n, &k);

 for (int i=0; i<n; i++) scanf("%d", &a[i]);

 //打印所有最小值
 for (int i=0; i<n; i++) {
     //窗口容量已满，出队
     if (hh<=tt && i-k+1>q[hh]) hh++;
     //单调性优化
     while (hh<=tt && a[q[tt]]>=a[i]) tt--;
     //入队
     q[++tt] = i;

     if (i-k+1 >= 0) printf("%d ", a[q[hh]]);
 }

 puts("");
```

```
30
31  //最大值
32  hh = 0, tt = -1;
33  for (int i=0; i<n; i++) {
34      if (hh<=tt && i-k+1>q[hh]) hh++;
35      while (hh<=tt && a[q[tt]]<=a[i]) tt--;
36      q[++tt] = i;
37
38      if (i-k+1 >= 0) printf("%d ", a[q[hh]]);
39  }
40
41  return 0;
42 }
```

KMP

习惯下标从1开始

1. 暴力算法如何做

```
1  S[N], p[M];
2
3  //朴素做法
4  for (int i=1; i<=n; i++) {
5      bool flag = true;
6      for (int j=1; j<=m; j++) {
7          if (s[i+j-1] != p[j]) {
8              flag = false;
9              break;
10         }
11     }
12
13     if (flag) 匹配成功
14 }
```

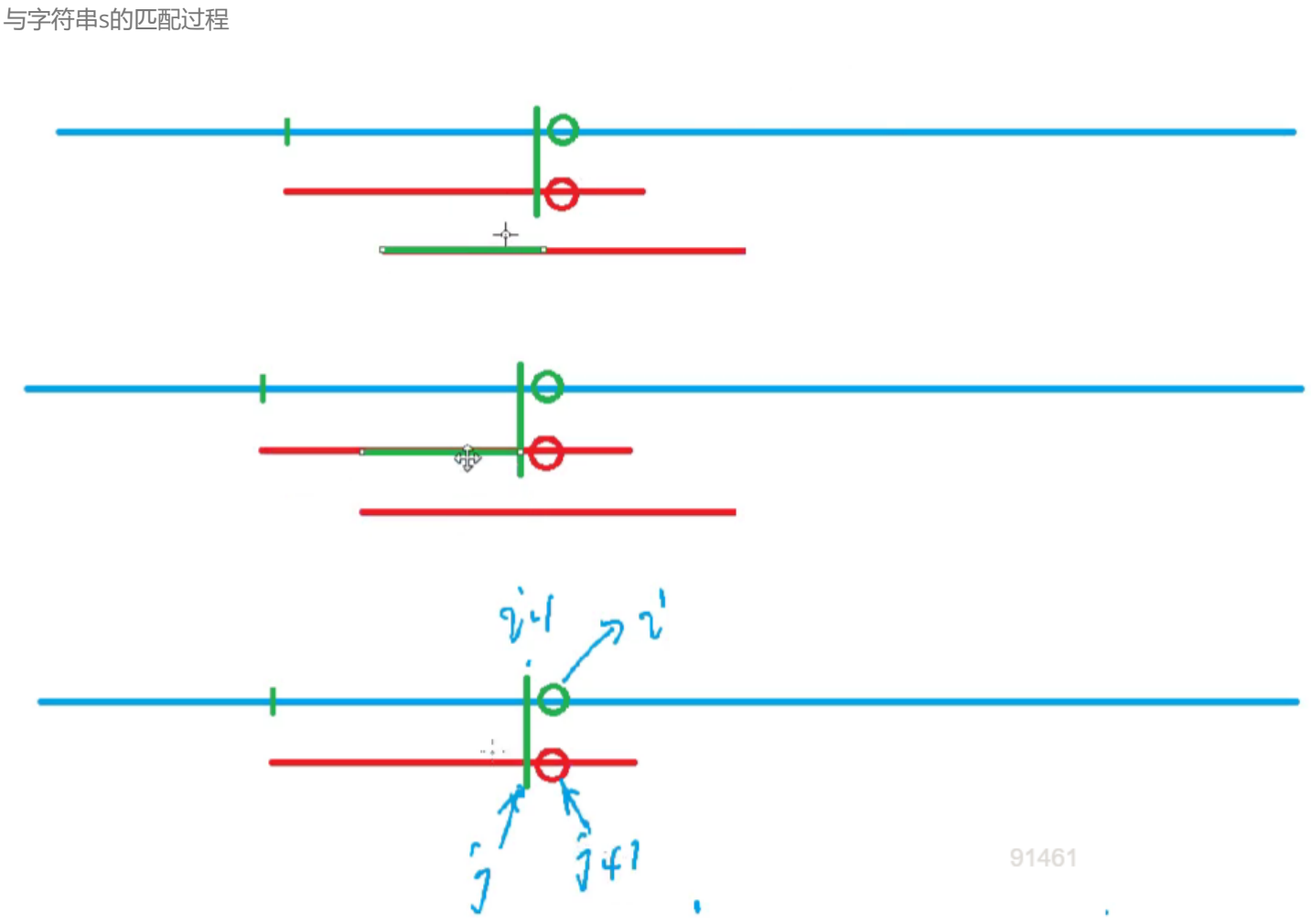
2. 如何去优化

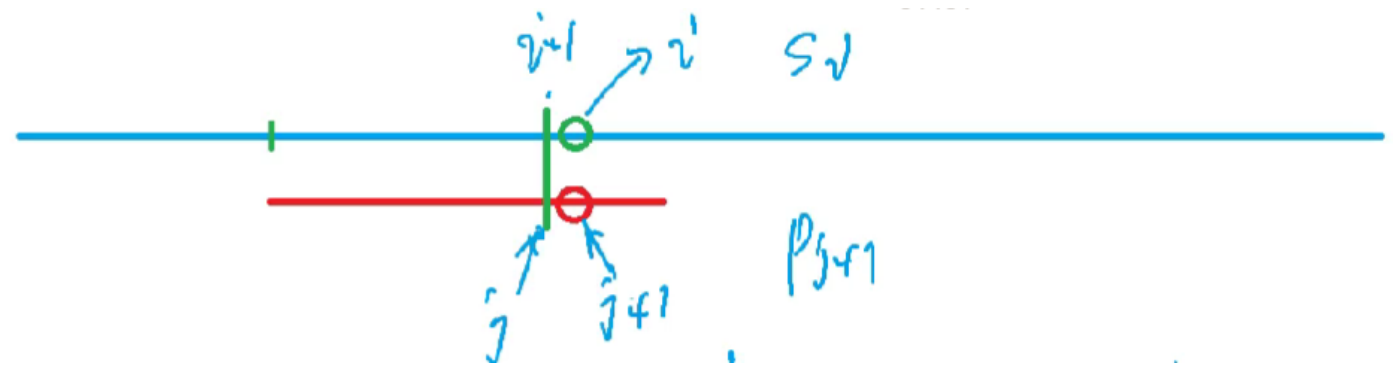
- **next[i] 表示以i为终点，且后缀和前缀相等的子串（长度严格小于原串）的最大长度。**

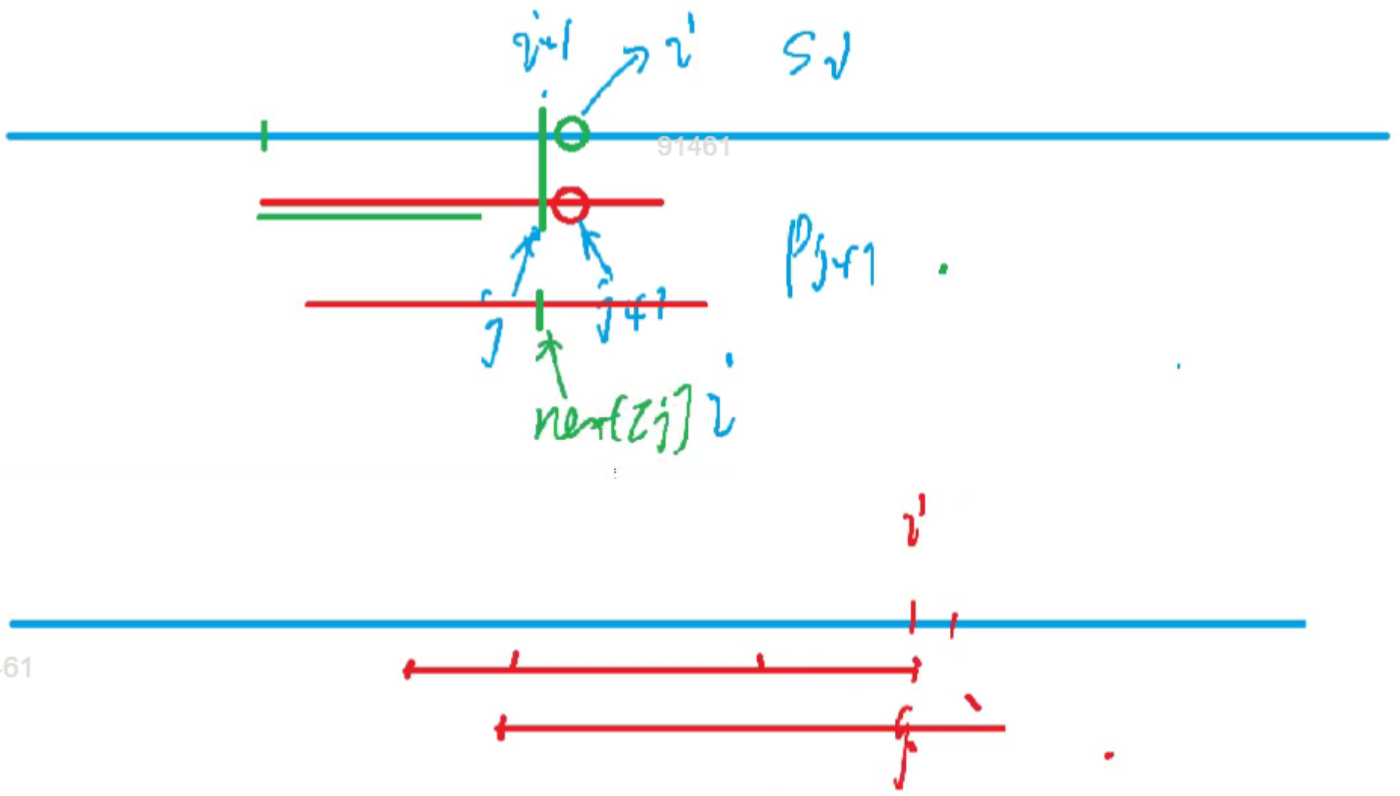
  下标从1开始时，定义next[1] = 0，即如果模板串第二(j+1)个字符没有匹配，则只能从头开始匹配，**即定义长度为1的串没有前后缀子串。**

  next[i] = j　　p[1, j] = p[i-j+1, i]

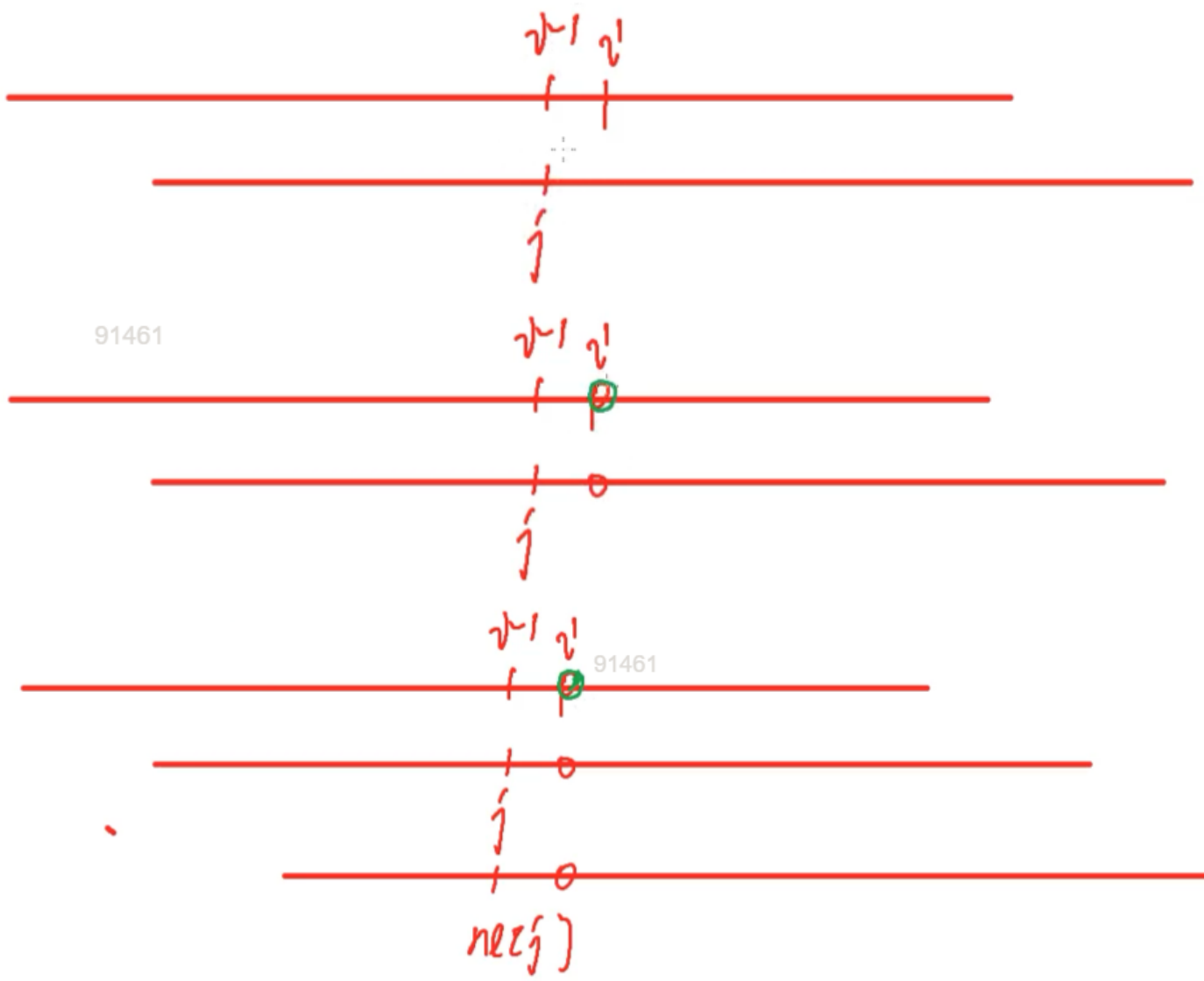
- 与字符串s的匹配过程

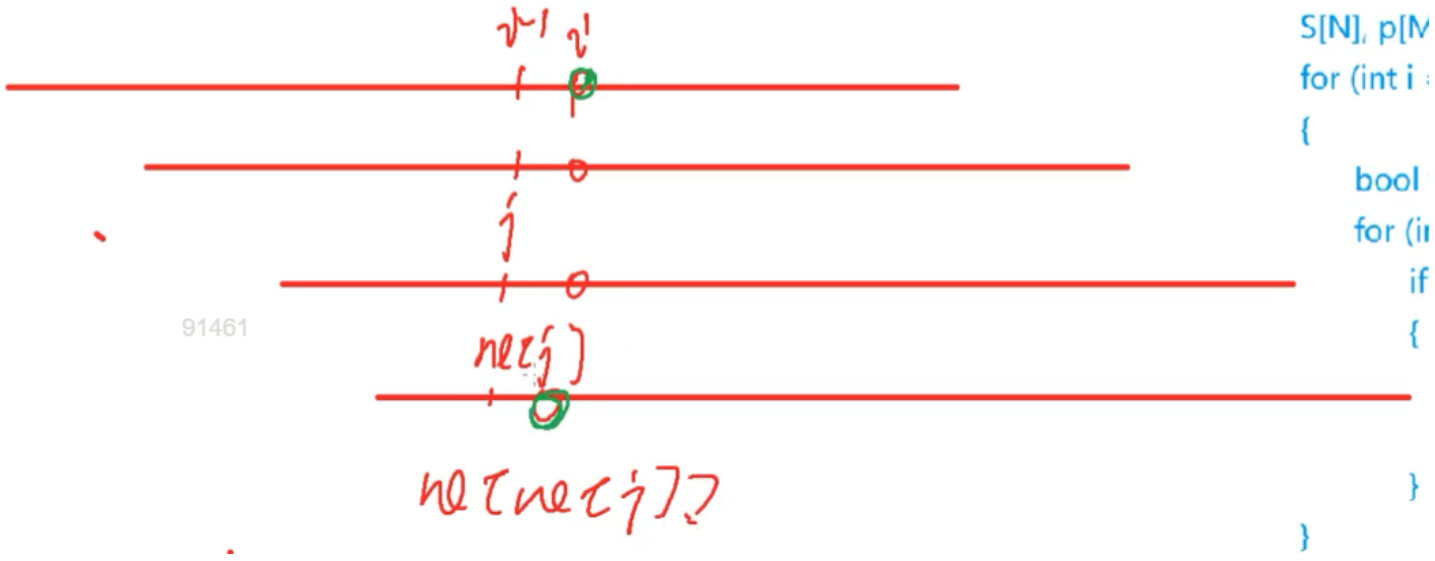






    - 匹配成功之后，由于j==n，不能再增加，需要更新j=ne[j]
- 求next数组过程





- 代码实现

```
#include <iostream>

using namespace std;

const int N = 100010, M = 1000010;

int n, m;
char p[N], s[M];
int ne[N];

int main(void) {
    scanf("%d%s%d%s", &n, p+1, &m, s+1);


    //求出模板串p的next数组，定义ne[1]=0，即如果第一个字符没有匹配
    //,则只能从头开始匹配。所以i从2开始，j=0表示从头匹配
    for (int i=2, j=0; i<=n; i++) {
        //结束条件：j=0 或者 已经匹配
        while (j && p[i]!=p[j+1]) j = ne[j];
        if (p[i] == p[j+1]) j++;    //i与j+1匹配
        ne[i] = j;  //更新ne数组
    }


    //求与s串的匹配位置
    //假设待匹配串s的[1, i-1]与模板串p[1, j]已经匹配，
    //当前比较的位置是s[i]与p[j+1]
    for (int i=1, j=0; i<=m; i++) {
        //j没有退回起点 并且 当前位置不匹配
        while (j && s[i]!=p[j+1]) j = ne[j];
        if (s[i] == p[j+1]) j++;
        if (j == n) {   //匹配成功
            printf("%d ", i-n);
            j = ne[j];  //匹配成功之后，后退一步，
        }
    }

    return 0;
}

```

- 时间复杂度O(n+m)

  对于第二大个循环，j最多加m次

  且ne[j]严格小于j，所以内部while最多减m次，即内部while最多执行m次，即最坏情况下是O(2m)

作用

- 可用于求循环节（字符串哈希无法使用）