# 2024年秋季 数据结构课程笔记

## Lecture 4 栈与队列

### 栈扩容

> 采取容量加倍策略 n次push元素时间复杂度$O(n)$， 每次均摊为$O(1)$ 而采取容量递增策略(栈的空间每次加d) n次push元素时间复杂度为$O(n^2)$，每次均摊为$O(n)$

实际上这边在比较算法的改进

### 栈混洗 （出栈次序问题）

#### 栈混洗出栈序列合法性判别

P85，86例题均选择D

##### 判别算法

> （总遵循原则：在栈里面底下的不可能比上面的先出去）对于出栈序列 只能是入栈序列当前值后者以后进入栈的值，不可能是在当前栈首元素再下面的值 时间复杂度：$O(n)$

P91 选C

##### 312模式判定

用于判定非法出栈序列

#### 栈混洗总数

P87页 输入序列 a1, a2, ....., an，关于出栈序列总数计算

等价于n对括号的合法序列数

> $Catalan(n) = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n}$

P100 选择B （栈后进先出 栈内已有a, b, c） P101 选择C （这种题分类讨论列举情况即可）

### 递归与回溯

栈帧：一次函数调用所需要的信息（存放于栈区） 递归算法关系到**空间复杂度**

回溯算法代码的困扰：条件转化为数据的表示形式 待调试 ~~LG P1605 (奇怪的Bug)~~ ~~LG B3625 (莫名TLE) 仍然存在问题(TLE)~~ 对于LG B3625的反思

```
void walk(int x, int y){
    if(x == n && y == m){
        cout << "Yes";
        exit(0);
        return ;
    }
    for(int i = 0;i < 4;i ++){
        if(Feasible(x + Dx[i], y + Dy[i])){
            Vis[x][y] = 1;
            walk(x + Dx[i], y + Dy[i]);
            //Vis[x][y] = 0; // 问题就出在这行代码上！！！
            // 只要不是求方案数的，就不用回溯
        }
    }
}
```

> 对于不能回溯的解释：此处是判断路径存在与否 返回上一级递归说明此路不通 故无需回溯；而统计路径数量的题，可能这条路是通的，因此需要回溯

~~待解决 HDU 2553 八皇后问题 (Acwing 843 同类题)~~

```
bool IsFeasible(int x, int y){
    for(int i = 1;i <= x - 1;i ++){
        if(Pos[i][y]) return false;
        if(i + y - x > 0 && i + y - x <= n && Pos[i][i + y - x]) return false;
        if(x + y - i >= 1 && x + y - i <= n && Pos[i][x + y - i]) return false;
    }
    return true;
}
```

> 上面这边注意判定条件的数组边界值简化判断

~~POJ 1028(栈的基础操作...)~~ 这题用到了双栈，不过更值得注意的是以下这段代码

```
void VisOp(){
    while (!Forward.empty())
    {
        Forward.pop();
    }
    char* Place = new char[100];
    // 特别注意这段代码 new出来的空间在堆区，而直接 char place[100];的空间在栈区，函数执行完就没了，会导致后面字符串操作错
误
    scanf("%s", Place);
    printf("%s\n", Place);
    Back.push(Place);
}
```

**快速幂**

**算法1：递归求解**

> $a^n = \begin{cases} a^{\frac{n}{2}} \times a^{\frac{n}{2}} & n为偶数 \\ a^{\lfloor \frac{n}{2} \rfloor} \times a^{\lfloor \frac{n}{2} \rfloor} \times a & n为奇数 \end{cases}$

对应代码如下

```
double myPow(double x, int n){
        if(n == 0 || x == 1) return 1;
        if(n == 1) return x;
        if(n == -1) return 1.0 / x;
        double result;
        double Pre = myPow(x, n / 2);
        result = Pre * Pre;
        if(n % 2) result *= x;
        return result;
    }
```

**算法2：快速幂 + 迭代**

> 对于$x^n$，因为$n = 2^{i_0} + 2^{i_1} + ..... + 2^{i_n}$ ，所以$a^n = a^{2^{i_0}} \times a^{2^{i_1}} \times ..... \times a^{2^{i_n}}$ 也就是说，可以将n转换为二进制数，决定$a^1, a^2, a^3 ..... a^n$ 是否被乘入结果

对应代码如下

```
double myPow(double x, int n){
        if(n < 0) return 1.0 / myPow(x, -(n + 1)) / x;
        double res = 1.0;
        double x_use = x;
        while (n > 0)
        {
            if(n % 2) res *= x_use;
            n /= 2;
            x_use = x_use * x_use;
        }
        return res;
    }
```

队列

**循环队列 && 广度优先搜索**

P153 选A ~~相关待解决题目 Openjudge3752(似乎是规定方向)~~ 水题... 解题思路：广度优先搜索 代码如下

```
int BFS(){
    Vis[1][1] = 1; Distance[1][1] = 1; f.push_back(pir(1, 1));
    while (!f.empty())
    {
        int x = f.front().first;
        int y = f.front().second;
        f.pop_front();
        for(int i = 0;i < 4;i ++){
            if(IsFeasible(x + Dx[i], y + Dy[i])){
                f.push_back(pir(x + Dx[i], y + Dy[i]));
                Vis[x + Dx[i]][y + Dy[i]] = 1;
                Distance[x + Dx[i]][y + Dy[i]] = Distance[x][y] + 1;
                if(x + Dx[i] == n && y + Dy[i] == m) return Distance[x + Dx[i]][y + Dy[i]];
            }
        }
    }
    return -1;
}
bool IsFeasible(int x, int y){
    if(x >= 1 && x <= n && y >=1 && y <= m && !Vis[x][y] && !Pos[x][y]) return true;
    return false;
}
```

> 当然也可以像PPT里面不用STL里面的deque，而是自己写一个, 丐版deque代码如下

```
class Deque{
public:
    void Push(pir a){
        Store[Rear].first = a.first; Store[Rear].second = a.second;
        Rear ++;
    };
    pir Pop(){
        return Store[Front ++];
    }
    bool Empty(){
        return Rear == Front;
    }
private:
    int Front = 0, Rear = 0;
    pir Store[N * N];
};
int BFS(){
    Deque fk;
    Vis[1][1] = 1; Distance[1][1] = 1; fk.Push(pir(1, 1));
    while (!fk.Empty())
    {
        pir firPoint = fk.Pop();
        int x = firPoint.first; int y = firPoint.second;
        for(int i = 0;i < 4;i ++){
            if(IsFeasible(x + Dx[i], y + Dy[i])){
                fk.Push(pir(x + Dx[i], y + Dy[i]));
                Vis[x + Dx[i]][y + Dy[i]] = 1;
                Distance[x + Dx[i]][y + Dy[i]] = Distance[x][y] + 1;
                if(x + Dx[i] == n && y + Dy[i] == m) return Distance[x + Dx[i]][y + Dy[i]];
            }
        }
    }
    return -1;
}
```

# Lecture 5 数组与矩阵

## 存储与寻址

注意存储单元容量和个数 实际上是字节的计算的问题

## 特殊矩阵的压缩存储

**对角矩阵的压缩存储**

这个把对角线上的变成横的就行了

**下三角矩阵压缩**

实际上是统计之前出现的元素个数求和，并将其作为下标 对于存储数组d,$M(i,j)$**前面**有个k个元素，则$M(i,j)$存储在$d[k]$ 公式如下
$M(i,j)=\{d[i\times(i-1)/2+j-1]\quad i\geq j\ 0\quad i<j$

**对称矩阵的压缩**

$\forall\quad 1\leq i,j\leq n,M(i,j)=M(j,i)$ 由此，存储位置计算为 $M(i,j)=\{d[i\times(i-1)/2+(j-1)]\quad i\geq j;\ d[j\times(j-1)/2+(i-1)]\quad i<j;$
P20题计算： $M(7,2)=M(2,7)=\frac{(1+6)\times 6}{2}+1=22$

**三对角矩阵M的压缩存储**

定义:$\forall\quad 1\leq i,j\leq n,$当$|i-j|>1,M(i,j)=0$恒成立 存储位置M(i, j) 前有k个元素 计算为 $k=(i-2)\times 3+2+(j-i)+1=2i+j-3$ 故
$M(i,j)=\{d[i+2j-3]\quad |i-j|\leq 1;\ 0\quad |i-j|>1$

## 三元组表

**稀疏矩阵的压缩存储**

定义：存在大量零元素的矩阵 压缩存储：仅存储非零元素 P29选A 三元组储存的形式

```
struct Triple{
    int row;
    int col;
    int val;
};
```

**稀疏矩阵的转置存储**

关于**快速转置算法**： 还有一件重要的事情！！！：**三元组表中的行列下标是从1开始的！！！** ~~待解决：P36页算法~~ 这个算法的疑难点在于这个$start[]$数组到底是用来干嘛的 解释：$start[]$数组用于存储转置后第k行前面有多少个元素，$start[k]$即代表着当前存储的第k行目的元素之前有多少个元素，然后将统计值作为新的矩阵中的下标

## 十字链表

定义：

```
struct ListNode{
    int row, col;
    int data;
    ListNode* up;
    ListNode* left;
};
```

注意在十字链表行与列中哨兵节点的设置

## 动态规划初步

使用动态规划解决的问题范围：**最优子结构**， **无后效性**， **重叠子问题**

**典例：斐波那契数列的计算**

最佳优化：时间复杂度$O(n)$,空间复杂度$O(1)$

### 二维情况

对于公式$F(i,j)=\{1 \quad i=1或j=1\ F(i-1,j)+F(i,j-1)$ 的优化

#### 滚动数组优化

使用空间优化，由二维压缩为一维 示例代码如下

```
int uniquePaths(int m, int n) {
        int Ans[200] = {0};
        for(int i = 1;i <= n;i ++)
            Ans[i] = 1;
        for(int i = 2;i <= m;i ++){
            for(int j = 1;j <= n;j ++){
                Ans[j] = Ans[j] + Ans[j - 1];
            }
        }
        return Ans[n];
    }
```

#### 直接使用组合数计算

路径的总数 = 从$n+m-2$步中选择$m-1$条向下的步数 = $\binom{n+m-2}{m-1}$ 问题便来到了：**如何高效地计算**$\binom{n}{m}$**?** 推导过程

> $$\binom{n}{m}=\frac{n!}{(n-m)!\times m!}=\frac{n}{m}\times\frac{n-1}{m-1}\times\frac{n-2}{m-2}\times.....\times\frac{n-m+2}{2}\times\frac{n-m+1}{1}$$

注意到上述公式从左向右计算中间结果可能会是小数 但是从右向左看，可以发现公式为$\binom{n-m+k}{k}$ 算法代码对应如下

```
ll CalculateC(int n, int k){
    if(k > n / 2) k = n - k; // 特别注意这行代码！！！会大量减少计算量，防止超时
    ll res = 1;
    for(int i = 1;i <= k;i ++)
        res = res * (n - k + i) / i;
    return res;
}
```

### 最大子数组和

#### C++中用于表示正负无穷大的方法

> 正无穷大: 0x3f3f3f3f 负无穷大: 0xc0c0c0c0

对应的，将数组a中每个元素初始化为正负无穷大的方法

```
memset(a, 0x3f, sizeof(a));
memset(a, 0xc0, sizeof(a));
```

对于最大子数组和的线性时间算法

```
int maxSubArray(vector<int>& nums) {
        int MaxSum = 0, RecordInterval = 0;
        MaxSum = RecordInterval = nums[0];
        for (int i = 1; i < nums.size(); i++) {
            if (RecordInterval > 0)
                RecordInterval += nums[i];
            else
                RecordInterval = nums[i];
            MaxSum = max(RecordInterval, MaxSum);
        }
```

```
        return MaxSum;
    }
```

对于该线性算法的阐释：

1. 首先，在for循环语句中加入if-else条件判断的速度比直接调用max函数快。这就非常的奇怪。
2. 记$f(i-1)$为位置$i-1$结尾的最大子数组，那么即有$f(i)=max(f(i)+nums[i],nums[i])$，因为无论如何，$nums[i]$总要计入下一个子区间，当前的问题即在于之前的子区间是否需要计入。

#### 对于最大子数组和对应数组的求取

细节非常多，相应算法代码如下

```
int MaxSum, RecordInterval, MaxLeft, MaxRight, Lf, Rg;
        MaxLeft = MaxRight = Lf = Rg = 0;
        MaxSum = RecordInterval = array[0]; // 首先这边得把第一个值赋给MaxSum，因为这个数组可能所有元素都是负的，初值赋0有漏洞
        for (int i = 1; i < array.size(); i ++) {
            if (RecordInterval >= 0) RecordInterval = RecordInterval + array[i];
            else {
                RecordInterval = array[i];
                Lf = i;
            }
            Rg = i;
            if (RecordInterval >= MaxSum) {
                if(RecordInterval > MaxSum){ // 这边这个分支条件判断又是一个坑点，当区间和更大时，需要及时更新。
                    MaxSum = RecordInterval;
                    MaxLeft = Lf;
                    MaxRight = Rg;
                } else {
                    MaxSum = RecordInterval;
                    if (Rg - Lf + 1 >= MaxRight - MaxLeft + 1) {
                        MaxLeft = Lf;
                        MaxRight = Rg;
                    }
                }
            }
        }
        vector<int> ans;
        for (int i = MaxLeft; i <= MaxRight; i ++) ans.emplace_back(array[i]);
        return ans;
```

#### 最大子数组积

关于最大子数组积和最大子数组和的区别，在于$f(i-1)\times nums[i]$中的$nums[i]$可能是负的，因此，需要另外维护一个当前下标对应的最小子数组积$g(i)$，由此，满足
$\{f(i)=max(f(i-1)\times nums[i],nums[i],g(i-1)\times nums[i])\backslash g(i)=min(f(i-1)\times nums[i],nums[i],g(i-1)\times nums[i])$

### 区间处理技巧

#### 前缀和

对于数组$a[i](0<i\leq n)$，构建数组$sum[i](0<i\leq n)$，使得$sum[i]=\sum_{j=1}^{i}a[j]$，由此，便有$\sum_{k=i}^{j}a[k]=sum[j]-sum[i-1]$（实际上从这里也可以看出为什么下标从1开始）。

#### 差分数组

适用场合：频繁地对数组某个区间的元素进行增减 先使用公式$diff[i]=a[i]-a[i-1]$构建数组$diff[n]$，对区间$l\rightarrow r$操作时，对$a[l]$执行正向操作，对$a[r+1]$执行反向操作(实际上是为了抵消效果)。 算法代码如下

```
    for(int i = 1;i <= n;i ++)
        scanf("%d", &nums[i]), diff[i] = nums[i] - nums[i - 1];
    while (p --)
    {
```

```
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        diff[x] += z;
        if(y + 1 <= n) diff[y + 1] -= z;
    }
    for(int i = 1;i <= n;i ++){
        nums[i] = nums[i - 1] + diff[i];
    }
```

**ST表（稀疏表）**

用于求取区间最值问题

定义$F(i,j)$表示数组$array$中从下标为$i$开始的$2^i$个数的最大值，即区间$[i,i+2^j-1]$的最大值 有如下递推公式
$F(i,j)=\{max(F(i,j-1),F(i+2^{j-1},j-1)),\quad 0<j\leq \log_2 n;\ array[i],\quad j=0$ 算法思路分析:实际上构建的$F(i,j)$方程，维护了对$array[]$数组$(1\leq i\leq n)$中任意元素$i$，可以在$O(1)$的时间内查询区间$[i,j](n\geq j\geq i)$内的最大值。而ST表（即$F(i,j)$）的构建时间复杂度为$O(nlogn)$ 算法对应代码如下

```
int n, m;
    scanf("%d%d", &n, &m);
    for(int i = 1;i <= n;i ++){
        scanf("%d", &nums[i]);
        ST[i][0] = nums[i];
    }
    for(int j = 1;j <= log2(n);j ++){
        for(int i = 1;i <= n + 1 - (1 << j);i ++){
            ST[i][j] = max(ST[i][j - 1], ST[i + (1 << (j - 1))][j - 1]);
        }
    }
    for(int i = 1;i <= m;i ++){
        int l, r;
        scanf("%d%d", &l, &r);
        int logRecord = log2(r - l + 1);
        printf("%d\n", max(ST[l][logRecord], ST[r - (1 << logRecord) + 1][logRecord]));
    }
```

**关于$2^i$的高效算法**

使用位运算

```
2 ^ i = (1 << i);
```

**尺取法**

即**双指针**算法，维护两个快慢指针即可。 算法对应代码如下

```
int FindMinSequence(){
    memset(a, 0, sizeof(a));
    int n, s;
    scanf("%d%d", &n, &s);
    for(int i = 0;i < n;i ++) scanf("%d", &a[i]);
    int lf = 0, rf = 0, MinLength = n + 1, Sum = 0;
    for(;;){
        while (rf < n && Sum < s)
        {
            Sum += a[rf ++];
        }
        if(Sum < s) break;
        MinLength = min(MinLength, rf - lf);
        Sum -= a[lf ++];
    }
    if(MinLength == n + 1) MinLength = 0;
    return MinLength;
}
```

### 子集生成

对于子集的生成有两种做法

**递归求取**

实际上就是深度优先搜索 算法对应代码如下

```
vector<vector<int>> subsets(vector<int>& nums){
        DFS(nums, 0);
        return ans;
    }
    void DFS(vector<int>& nums, int k){
        if(k == nums.size()){
            ans.push_back(arr);
            return ;
        }
        DFS(nums, k + 1);
        arr.push_back(nums[k]);
        DFS(nums, k + 1);
        arr.pop_back();
    }
```

时间复杂度为$O(n \times 2^n)$

**迭代求取**

算法解释：集合的第$i$个元素是否构成子集中的元素有$0,1$两种状态，由此，$n$个元素便有$2^n$个状态。对于状态$(a_n a_{n-1} \dots a_1 a_0)$，与集合储存元素数组$nums[]$的第$i$个元素对应的位数（即$2^i$）相与，由此决定该元素是否放入子集中。 算法对应代码如下：

```
vector<vector<int>> subsets(vector<int>& nums) {
        int Len = nums.size();
        for (int i = 0; i < (1 << Len); i++) {
            arr.clear();
            for (int j = 0; j < Len; j++) {
                if (i & (1 << j))
                    arr.push_back(nums[j]);
            }
            ans.push_back(arr);
        }
        return ans;
    }
```

## Lecture 6 字符串模式匹配

### 模式串匹配

即在目标串（文本）中查找模式串（关键词）

### 朴素模式匹配算法

**最坏**时间复杂度为$O(n \times m)$

命题：假定目标串$S$长度为$n$，模式串$P$长度为$m$，则朴素模式匹配算法在平均时间情况下的字符比较次数不超过$2n$ 注意到平均时间复杂度与字符集数量，模式串每位比较概率相关。

> 对于$T_i$（文本串中每位开始的平均比较次数）最终计算得公式$T_i = \sum_{i=0}^{m-1} \frac{1}{k^i} \leq 2$，由此$T \leq n \times T_i \leq 2 \times n$，注意到$k =$ 字符集大小

由此，我们得出结论，**字符集包含的字符种类越多，平均时间复杂度越低**

对于朴素模式匹配算法，字符集越大，时间越趋于线性。字符集越小，越接近最坏情况，算法性能越差。

### KMP算法

寻找**最长相等**的前后缀

> 算法解释：对于目标串$S = s_0 s_1 \dots s_{n-1}$，模式串$P = p_0 p_1 \dots p_{m-1}$，匹配至$s_{i-j} \dots s_{i-1}$与$p_0 p_1 \dots p_{j-1}$相匹配，但是$s_i != p_j$，此时，在失配位置对应子串$p_0 p_1 \dots p_{j-1}$中，寻找长度最大的相等前后缀，使得$s_{i-k} \dots s_{i-1} = p_{j-k} \dots p_{j-1} = p_0 \dots p_{k-1}$ 此时右移模式串，使得$s_{i-k} \dots s_{i-1}$与$p_0 \dots p_{k-1}$相对齐，**也就是说，让$s_i$和$p_k$继续比对。**

注意到：**k是模式串$p_0 p_1 \dots p_{j-1}$中的最长相等前后缀长度，k标识了模式串P在位置j匹配失败后，下一次匹配的位置**

由此，我们引出$Next()$函数（又称为失败函数、前缀函数） $Next(j) = \{ -1 \quad j = 0; \ max{p_0 p_1 \dots p_{k-1} = p_{j-k} \dots p_{j-1}} \quad 0 \le k \le j; \ 0 \quad$ 不存在$k$
注意到$j = -1$时，实际上是为了使得在文本串中下一位进行匹配

P52选择$D$， 注意到在位置$j = 7$处失配，实际上是在前7位中寻找最长相等前后缀（因为下标从0开始）

**关于$Next()$函数的构建**

对于$Next()$函数构建的朴素算法时间复杂度为$O(m^2)$，显然不可取 采用递推思想构建，已知$Next(j) = k$，求取$Next(j+1)$ 算法代码对应如下

```
int strStr(string haystack, string needle) {
        const int N = 1e4 + 10;
        int Next[N] = {0}, m = needle.length(), n = haystack.length();
        Next[0] = -1;
        for(int i = 0;i < m - 1;i ++){
            int k = Next[i];
            while (k >= 0 && needle[i] != needle[k]) // 特别注意此处，不是needle[i + 1]而是needle[i]！！！
            {
                k = Next[k];
            }
            Next[i + 1] = ++ k;
        }
        for(int i = 0, j = 0;i < n && j < m;){
            if(j < 0 || haystack[i] == needle[j]){
                i ++;
                j ++;
                if(j == m) return i - m;
            } else{
                j = Next[j];
            }
        }
        return -1;
    }
```

> 实际上上面这段代码中有一个很重要的总结：对于$Next()$函数，$Next(i)$指的是从0到$i-1$区间范围内的最长相等前后缀，而**str[i]处是匹配失败的位置!!!**

**关于$KMP$算法的时间复杂度分析**

$Next()$函数的构建部分，注意到$k$最多加$m$次，而$k$每次减至少减1，$k$总计减的步数必定小于等于$m$次，故总执行步数$n$满足$n \le 2m$，由此时间复杂度为$O(m)$。同理，下面的字符串匹配过程的时间复杂度为$O(n)$ 综上，时间复杂度为$O(n+m)$

**P76那道题的含义**

对于寻找一个字符串的最长回文前缀，由于$p_0 p_1 \dots p_m = p_m \dots p_0$，由此将原字符串翻转后，拼接到初始字符串上，形成$p_0 p_1 \dots p_n p_n \dots p_1 p_0$，从而寻找新字符串的最长相等前后缀，也就是构建$Next()$数组

**P77 添加字符生成最短回文串思路**

原字符串s，在s前面添加一段字符p，使得$p + s$构成回文串。**注意到$s$的前$len(s) - len(p)$个字符同样构成回文串。**利用回文串的性质，将其翻转后，**等价于前缀和后缀相同**，由此联系到$KMP$算法中的前后缀字符串匹配问题。 相应算法对应代码如下

```
string shortestPalindrome(string s) {
        const int Len = s.length();
        string model = s;
        for(int i = 0, j = Len - 1;i <= j;i ++,j --){
            swap(s[i], s[j]);
        }
        const int N = 5e4 + 10;
        int next[N];
```

```
        next[0] = -1;
        for(int i = 0;i < Len - 1;i ++){
            int k = next[i];
            while (k >= 0 && model[k] != model[i])
            {
                k = next[k];
            }
            next[i + 1] = ++k;
        }
        int position;
        int i = 0, j = 0;
        while (i < Len && j < Len)
        {
            if(j < 0 || model[j] == s[i]){
                i ++; j ++;
            } else {
                j = next[j];
            }
        }
        string ans = model.substr(j, Len - j);
        reverse(ans.begin(), ans.end());
        ans += model;
        return ans;
    }
```

注意$substr$和$reverse$两个库函数的使用方式

**$KMP$算法的应用：循环节**

细节还是在于对$Next()$函数意义的理解，$Next(i)$**指的是在**$i$**位置失配，值的含义是求取得的**$[p_0p_1\dots p_{i-1}]$**中的最大前后缀长度。**因此此处计算循环节，需要计算整个字符串的最大前后缀长度，需要多算一位。 算法对应代码如下

```
bool repeatedSubstringPattern(string s) {
        const int N = 1e4 + 10;
        int next[N], len = s.length();
        next[0] = -1;
        for (int i = 0; i < len; i++) { // 特别注意此处的"i < len", 而不是"i < len - 1"!!!
            int k = next[i];
            while (k >= 0 && s[i] != s[k]) {
                k = next[k];
            }
            next[i + 1] = ++k;
        }
        if (len % (len - next[len]) == 0 && next[len])
            return true;
        else
            return false;
    }
```

**纯粹应试（两个定义的差别）**

关于教材中的失败函数$f(i)$和本笔记中的$Next(i)$定义的区别，详见ppt第86页。

## 对于$KMP$算法的改进

对于原来的$Next(i)$，当寻找到的前缀$p_0p_1\dots p_{k-1}$，但是，仍然有$p_k == p_i$，也就是说，移动模式串后下一次匹配还是失配。实际上，可以在构建$Next()$时进行判别改进。 满足
$Next2(j) = \{-1 \quad j = 0;\ max p_0\dots p_{k-1} = p_{j-k}\dots p_{j-1}$并且$p_k! = p_j$ 存在$k$ 0 不存在前后缀，但是$p_0! = p_j$ $-1$ 不存在前后缀题在于代码 $p_j$
好像根本跑不通...