Chapter 1

STL

1.1 bitset

作用：判断每位状态 0 或者 1
定义：bitset<MAX_N> b; //MAX_N 是长度下标从 0 开始，默认状态值都为 0

1.1.1 用 unsigned long 值初始化 bitset 对象

将 unsigned long 值转化为二进制的为模式，而 bitset 对象中的位集作为这种模式的副本。如果 bitset 类型大于 unsigned long 的二进制位数，则其余高阶位位置为 0，如果 bitset 类型长度小于 unsigned long 值的二进制位数，则只使用 unsigned long 值中的低阶位，超过 bitset 类型长度的高阶位将会被舍弃。例如：
bitset<32> bs(0xffff); // bits 0 ... 15 are set to 1; 16 ... 31 are 0

1.1.2 用 string 对象初始化 bitset 对象

string 对象只能为 01 串。如果 string 对象中的字符个数小于 bitset 类型的长度，则高阶位为 0. 从 string 对象读入位集的顺序是从右向左（反向转化）:string 对象最右边的字符（即下标最大的那个字符）用来初始化低阶位（即下标为 0 的为）。可以指定某个子串作为初始值。例如：

```
string str("11111110000000011001101");
bitset<32> bs1(str); //str is fully used to initialize bs1
bitset<32> bs2(str, 5, 4); // 4 bits starting at str[5], 1100
bitset<32> bs3(str, str.size() - 4);      // use last 4 characters
cout << "bs1 = " << bs1 << endl;
cout << "bs2 = " << bs2 << endl;
cout << "bs3 = " << bs3 << endl;
/*
Result:
bs1 = 00000000001111111000000011001101
bs2 = 00000000000000000000000000001100
bs3 = 00000000000000000000000000001101
*/
```

1.1.3 操作

b.any()	b 中是否存在置为 1 的二进制位
b.none()	b 中不存在置为 1 的二进制位吗？
b.count()	b 中置为 1 的二进制位的个数
b.size()	b 中二进制位的个数
b[pos]	访问 b 中在 pos 处的二进制位
b.test(pos)	b 中在 pos 处的二进制位是否为 1 ？
b.set()	把 b 中所有二进制位都置为 1
b.set(pos)	把 b 中在 pos 处的二进制位置为 1
b.reset()	把 b 中所有二进制位都置为 0

b.reset(pos) 把 b 中在 pos 处的二进制位置为 0
b.flip() 把 b 中所有二进制位逐位取反
b.flip(pos) 把 b 中在 pos 处的二进制位取反
b.to_ulong() 用 b 中同样的二进制位返回一个 unsigned long 值

1.1.4 测试

注意：使用 string 初始化时从右向左处理，如下初始化的各个位的值将是 110，而非 011 string strVal("011");

```
bitset<3> bs1(strVal);
cout << "bs1[0] is " << bs1[0] << endl;
cout << "bs1[1] is " << bs1[1] << endl;
cout << "bs1[2] is " << bs1[2] << endl;
cout << bs1 << endl;
//any() 方法如果有一位为 1 ，则返回1
cout << "bs1.any() = " << bs1.any() << endl;
//none() 方法，如果有一个为 1 none 则返回 0 ，如果全为 0 则返回 1
bitset<3> bsNone;
cout << "bsNone.none() = " << bsNone.none() << endl;
//count() 返回几个位为1
cout << "bs1.count() = " << bs1.count() << endl;
//size() 返回位数
cout << "bs1.size() = " << bs1.size() << endl;
//flip() 诸位取反
bitset<3> bsFlip = bs1.flip();
cout << "bsFlip = " << bsFlip << endl;
//to_ulong: 用 bs1 中同样的二进制位返回一个 unsigned long 值
unsigned long val = bs1.to_ulong();
cout<< val << endl;
/*
Result:
bs1[0] is 1
bs1[1] is 1
bs1[2] is 0
011
bs1.any() = 1
bsNone.none() = 1
bs1.count() = 2
bs1.size() = 3
bsFlip = 100
4
*/
```

1.2 单调栈、单调队列

1.2.1 单调栈

单调队列和单调栈的时间复杂度都是:$O(n)$。
单调栈主要用于解决某个元素它向左向右为最大值或最小值的最大范围是什么。如果是最大值，那就要维护单调非递增栈（唯一最大就是单调递减栈），如果是最小值就要维护单调非递减栈（唯一最小就是单调递增栈）。

```
//求数组每个数以其为区间唯一最小值的最大区间左右端点
int top = 0, cur;
for (int i = 1; i <= n; ++i) {
    while (top) {
        cur = sta[top];
        if (data[cur] <= data[i]) break;//非唯一时去掉等号
        --top;
    }
    if (top == 0) left[i] = 1;  //data[i] 的区间左端点
    else left[i] = sta[top] + 1;
    sta[++top] = i;
}
top = 0;
for (int i = n; i >= 1; --i) {
    while (top) {
        cur = sta[top];
        if (data[cur] <= data[i]) break;
        --top;
    }
    if (top == 0) right[i] = n;
    else right[i] = sta[top] - 1;
    sta[++top] = i;
}
//如果是求以其为最大值，只需要把 <= 换为 >=
```

[POJ 3494]：给出一个 $n*m$ 的 01 矩阵，求出最大全 1 子矩阵面积。数据范围：$n, m \leq 2000$

我们把每一行单独处理，把从这行向上连续延伸全为 1 的最大长度看成是矩形的高，那么每行其实就是求个最大矩形面积。

$height[i][j]$: 第 i 行第 j 列元素往上最长的连续 1 长度

需要用 $O(n^2)$ 的复杂度预处理出 $height[]$，然后需要枚举每行，每行利用单调栈可以在 $O(n)$ 复杂度得到最大矩形面积. 总的时间复杂度是:$O(n^2)$。

```
const int MAX_N = 2010;

int n, m, ans;
int mat[MAX_N][MAX_N];
int height[MAX_N][MAX_N], sta[MAX_N], L[MAX_N], R[MAX_N];

//height[i][j]: 第 i 行第 j 列元素往上最长的连续 1 长度
//维护单调非递减栈
void solve(int row)
{
    int top = 0, cur;
    height[row][m + 1] = 0;
    for (int j = 1; j <= m + 1; ++j) {
        while (1) {
            cur = sta[top];
            if (height[row][cur] <= height[row][j]) break;
            R[cur] = j;
            --top;
        }
        L[j] = cur;
```

```
        sta[++top] = j;
    }
    for (int j = 1; j <= m; ++j) {
        if(mat[row][j] == 0) continue;
        int len = R[j] - L[j] - 1;
        ans = max(ans, height[row][j] * len);
    //  printf("height[%d][%d] = %d len = %d\n", row, j, height[row][j], len);
    }
}

int main()
{
    while (~scanf("%d%d", &n, &m)) {
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= m; ++j) {
                scanf("%d", &mat[i][j]);
            }
        }
        memset(height, 0, sizeof(height));
        for (int j = 1; j <= m; ++j) {
            for (int i = 1; i <= n; ++i) {
                if (mat[i][j] == 1) {
                    height[i][j] = 1;
                    while (mat[++i][j] == 1) {
                        height[i][j] = height[i - 1][j] + 1;
                    }
                    --i;
                }
            }
        }
        ans = 0;
        for (int i = 1; i <= n; ++i) { solve(i); }
        printf("%d\n", ans);
    }
    return 0;
}
```

1.2.2 单调队列

单调队列主要用于解决满足特定条件的区间问题（如：区间最大值不超过 k 的最大区间长度, 所有区间长度为 k 的最大元素值，长度不超过 k 的最大连续子序列和，区间最值差 $\in [m,k]$ 的最大区间长度）。往往和前缀和结合在一起。需要判别队列应是何种单调性。

【*HDU* 3530 *Subsequences* :】给 $n(n \leq 10^5)$ 个数，求区间最值差 $\in [m,k]$ 的最大区间长度。

维护一个单调非递增队列和一个单调非递减队列，通过当前位置元素维护其单调性，然后调整队列首元素之差 $diff$，使 $diff \leq k$，判断调整后的 $diff$ 是否满足 $diff \geq m$, 如果满足更新 ans。

```
memset(dec, 0, sizeof(dec));
memset(inc, 0, sizeof(inc));
int ans = 0, diff, pre = 0; //额外注意 pre 赋初值为0
for (int i = 0; i < n; ++i) {
    while (head_inc != tail_inc && data[i] < data[inc[tail_inc - 1]]) --tail_inc;
    inc[tail_inc++] = i;

    while (head_dec != tail_dec && data[i] > data[dec[tail_dec - 1]]) --tail_dec;
    dec[tail_dec++] = i;

    while(1) {
        diff = data[dec[head_dec]] - data[inc[head_inc]];
        if (diff > k) {
            if (dec[head_dec] < inc[head_inc]) {
                pre = dec[head_dec] + 1;  //注意区间首的位置
                head_dec++;
```

```
            } else {
                pre = inc[head_inc] + 1;
                head_inc++;
            }
        } else break;
    }
    diff = data[dec[head_dec]] - data[inc[head_inc]];
    if(diff >= m) ans = max(ans, i - pre + 1);
}
```

1.3 list

1.3.1 CF 350 E

给出一个长度为偶数的只含'(' 和')' 并且两者个数相等的字符串, 初始指针位置是 p, 下标从 1 开始. 有三种操作:

- R 指针位置右移, 即 $p++$
- L 指针位置左移, 即 $p--$
- D 删除 p 位置和相对应括号这个区间的所有括号

输出若干次操作后的字符串.

需要预处理出每个括号和相对应的括号的下标.

```
const int MAX_N = 500010;

int match[MAX_N];

int main()
{
    int n, m, p;
    while(cin >> n >> m >> p) {
        string s1, s2;
        cin >> s1;
        cin >> s2;
        stack <int> s;
        for(int i = 0; i < n; i++){
            if (s1[i] == '(') {
                s.push(i);
            } else {
                int t = s.top();
                s.pop();
                match[t] = i;
                match[i] = t;
            }
        }
        list <int> lis;
        for (int i = 0; i < n; i++) { lis.push_back(i); }
        list<int> ::iterator pos = lis.begin();
        for (int i = 1; i < p; i++) pos++;
        for (int i = 0; i < m; i++) {
            if (s2[i] == 'L') pos--;
            else if (s2[i] == 'R') pos++;
            else {
                list<int> ::iterator tmp = pos, it;
                if (s1[*pos] == '('){
                    while ((*pos) != match[*tmp]) {
                        pos++;
                    }
                } else {
                    while ((*tmp) != match[*pos]) {
```

```
38                          tmp--;
39                      }
40                  }
41                  for (it = tmp; it != pos; ){
42                      lis.erase(it++);
43                  }
44                  lis.erase(pos++);
45                  if (pos == lis.end()) pos--;
46              }
47          }
48
49          for (it = lis.begin(); it != lis.end(); it++){
50              cout << s1[*it];
51          }
52          cout << endl;
53      }
54      return 0;
55  }
```