基础算法

快速排序:

```
void quick_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j); quick_sort(q, j + 1, r);
}
```

归并排序:

```
void merge_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int mid = l + r >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else {
            tmp[k ++ ] = q[j ++ ];
            // ans += mid - i + 1 //逆序对数量
        }

    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r)  tmp[k ++ ] = q[j ++ ];

    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

插入排序:

```
void insertion_sort(int q[], int len) {
  for (int i = 1; i < len; i++) {
    int key = q[i];
    int j = i - 1;
    // 寻找 key 的位置
    while (j >= 0 && q[j] > key) {
      q[j + 1] = q[j]; j--;
    }
    arr[j + 1] = key; // 插入 key
  }
}
```

二分:

```
bool check(int x) {/* ... */} // 检查x是否满足某种性质

// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用：
while (l < r)
{
    int mid = l + r >> 1;
    if (check(mid)) r = mid;    // check()判断mid是否满足性质
    else l = mid + 1;
}

// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用：
while (l < r)
{
    int mid = l + r + 1 >> 1;
    if (check(mid)) l = mid;
    else r = mid - 1;
}

// 浮点数二分
while (r - l > 1e-6) { // 1e-6 表示精度，取决于题目对精度的要求
    double mid = (l + r) / 2;
    if (check(mid)) r = mid;
    else l = mid;
}
```

位运算:

```
n的第k位设置为0: n & ~(1 << k)
n的第k位设置为1: n | (1 << k)
n的第k位取反: n ^ (1 << k)
求n的第k位数字: n >> k & 1
返回n的最后一位1：lowbit(n) = n & -n
```

递推求组合数:

```
// c[a][b] 表示从a个苹果中选b个的方案数
for (int i = 0; i < N; i ++ )
    for (int j = 0; j <= i; j ++ )
        if (!j) c[i][j] = 1;
        else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
```

双指针:

```
//     (1) 对于一个序列，用两个指针维护一段区间
//     (2) 对于两个序列，维护某种次序，比如归并排序中合并两个有序序列的操作
for (int r = 0, l = 0; r < n; r ++ )
{
    while (l < r && check(l, r)) l ++ ;

    // 具体问题的逻辑
}
```

区间合并(将所有存在交集的区间合并):

```
void merge(vector<PII> &segs)
{
    vector<PII> res;

    sort(segs.begin(), segs.end());

    int st = -2e9, ed = -2e9;
    for (auto seg : segs)
        if (ed < seg.first)
        {
            if (st != -2e9) res.push_back({st, ed});
            st = seg.first, ed = seg.second;
        }
        else ed = max(ed, seg.second);

    if (st != -2e9) res.push_back({st, ed});

    segs = res;
}
```

单调栈:

```
stack<int> stk;
for (int i = 1; i <= n; i ++ )
{
    while (stk.size() && check(stk.top(), i)) stk.pop() ;
    stk.push(i);
}
```

单调队列(常见模型：找出滑动窗口中的最大值/最小值):

```
int hh = 0, tt = -1; // hh 表示队头，tt表示队尾
for (int i = 0; i < n; i ++ )
{
    while (hh <= tt && check_out(q[hh])) hh ++ ;  // 判断队头是否滑出窗口
    while (hh <= tt && check(q[tt], i)) tt -- ;
    q[ ++ tt] = i;
}
```

最大连续子段和:

```
f = [i for i in a]
for i in range(1, n):
    f[i] = max(f[i - 1] + a[i], a[i])
mx = max(f)
```

求所有组合序列:

```
vector<vector<int>> ans;
vector<int> temp; // 记录路径

void dfs(int start, int n, int k) {
    if (temp.size() == k) {
        ans.push_back(temp);
        return;
    }

    for (int i = start; i <= n; i++) {
        temp.push_back(i);
        dfs(i + 1, n, k);
        temp.pop_back(); // 回溯
    }
}

vector<vector<int>> combine(int n, int k) {
    dfs(1, n, k);
    return ans;
}
```

单个状态对二维数组进行dfs:

```
# p*q的网格
def dfs(pos, cnt):
    global p,q,k,res,vis
    if cnt == k: # 终止条件
        res += 1
        res %= mod
        return
    for i in range(pos, p*q):
        x = i // q # 获取行
        y = i % q # 获取列
        if (not vis[x][y]) and check(x, y):
            vis[x][y] = True
            dfs(i+1, cnt+1) # 开始下一个位置
            vis[x][y] = False # 回溯
```