```
#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<queue>
#include<stack>
#include<algorithm>
#include<string>
#include<map>
#include<set>
#include<vector>
using namespace std;

void merge_sort(int *arr, int l, int r) {
    if(l >= r) return ;
    int mid = (l + r) / 2;

    cout << endl;
    cout << "sort : " << l << " <--->" << r << " : " << endl;
    for(int i = l; i <= r; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    merge_sort(arr, l, mid);
    merge_sort(arr, mid + 1, r);
    vector<int> temp(r - l + 1);
    int k = 0, p1 = l, p2 = mid + 1;
    // 当左右两个区间还有元素的时候
    while(p1 <= mid || p2 <= r) {
        // 1. 右区间为空
        // 2. 左区间没空，并且，左区间的元素比较小
        if((p2 > r) || (p1 <= mid && arr[p1] <= arr[p2])) {
            temp[k] = arr[p1];
            k++, p1++;
        } else {
            temp[k] = arr[p2];
            k++, p2++;
        }
    }
    for(int i = l; i <= r; i++) {
        arr[i] = temp[i - l];
    }
    for(int i = l; i <= r; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return ;
}

int main() {
    int a[10] = {1, 9, 0, 2, 5, 6, 2, 7, 1, 9};
    merge_sort(a, 0, 9);
    for(int i = 0; i < 10; i++) {
        cout << a[i] << " ";
    }
    return 0;
}
```

1. 数组中的逆序对

```
class Solution {
public:
    vector<int> temp;
    int merge_sort(vector<int>& nums, int l, int r) {
        if (l >= r) return 0;
        int mid = (l + r) / 2, ans = 0;
        ans += merge_sort(nums, l, mid);
        ans += merge_sort(nums, mid + 1, r);
        int k = l, p1 = l, p2 = mid + 1;
        while ((p1 <= mid) || (p2 <= r)) {
            if ((p2 > r) || (p1 <= mid && nums[p1] <= nums[p2])) {
                temp[k++] = nums[p1++];
            } else {
                temp[k++] = nums[p2++];
                ans += (mid - p1 + 1);
            }
        }
        for (int i = l; i <= r; i++) nums[i] = temp[i];
        return ans;
    }
    int reversePairs(vector<int>& nums) {
        while (temp.size() < nums.size()) temp.push_back(0);
        return merge_sort(nums, 0, nums.size() - 1);
    }
};
```

2. 合并K个升序链表 (分治合并链表)

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */

struct cmp {
    bool operator() (ListNode* a, ListNode* b) {
        return a->val > b->val;
    }
};

class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if(lists.size() == 0) {
            return nullptr;
        }
        // 小顶堆
        priority_queue<ListNode*, vector<ListNode*>, cmp> que;
        for(auto list : lists) {
            if(list == nullptr) continue;
            que.push(list);
        }

        ListNode dummy(-1);
        ListNode *ans = &dummy;
        while(!que.empty()) {
            ListNode* p = que.top();
            que.pop();
            ans->next = p;
            ans = p;
            p = p->next;
            if(p != nullptr) {
                que.push(p);
            }
        }
        return dummy.next;
    }
};
```

3. 排序链表

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode *mergeSort(ListNode *head, int n) {
        if (n <= 1) return head;
        int l_cnt = (n / 1), r_cnt = n - l_cnt;
        ListNode ret, *l = head, *r = l, *p = l;
        for (int i = 1; i < l_cnt; i++) p = p->next;
        r = p->next; p->next = nullptr;
        l = mergeSort(l, l_cnt);
        r = mergeSort(r, r_cnt);
        p = &ret;
        while (l || r) {
            if (r == nullptr || (l && l->val <= r->val)) {
                p->next = l; p = l; l = l->next;
            } else {
                p->next = r; p = r; r = r->next;
            }
        }
        return ret.next;
    }
    ListNode* sortList(ListNode* head) {
        int n = 0;
        ListNode *p = head;
        while (p) n += 1, p = p->next;
        return mergeSort(head, n);
    }
};
```

4. 两棵二叉搜索树中的所有元素 (树)

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    void getNums(TreeNode *root, vector<int> &nums) {
        if (root == nullptr) return ;
        getNums(root->left, nums);
        nums.push_back(root->val);
        getNums(root->right, nums);
        return ;
    }
    vector<int> getAllElements(TreeNode* root1, TreeNode* root2) {
        vector<int> l, r;
        getNums(root1, l);
        getNums(root2, r);
        vector<int> temp;
        int p1 = 0, p2 = 0;
        while (p1 < l.size() || p2 < r.size()) {
            if ((p2 >= r.size()) || (p1 < l.size() && l[p1] <= r[p2])) {
                temp.push_back(l[p1++]);
            } else {
                temp.push_back(r[p2++]);
            }
        }
        return temp;
    }
};
```

5. 子数组和排序后的区间和

```
class Solution {
public:
    struct Date {
        Date(int i, int j, int sum) : i(i), j(j), sum(sum) {}
        int i, j, sum;
    };
    struct cmp {
        bool operator()(const Date &a, const Date &b) {
            return a.sum > b.sum;
        }
    };
    int rangeSum(vector<int>& nums, int n, int left, int right) {
        priority_queue<Date, vector<Date>, cmp> q;
        for(int i = 0; i < n; i++) {
            q.push(Date{i, i, nums[i]});
        }
        int ans = 0, mod = 1e9 + 7;
        for(int i = 1; i <= right; i++) {
            Date d = q.top();
            q.pop();
            if(i >= left) {
                ans = (ans + d.sum) % mod;
            }
            if(d.j + 1 < n) {
                q.push(Date{d.i, d.j + 1, (d.sum + nums[d.j + 1]) % mod});
            }
        }
        return ans;
    }
};
```

6. 区间和的个数 (难)

```
class Solution {
public:

    int merge_sort(vector<long>& arr, int l, int r, int lower, int upper) {
        if (r - l <= 1) return 0;

        int mid = (l + r) / 2, ans = 0;
        ans += merge_sort(arr, l, mid, lower, upper);
        ans += merge_sort(arr, mid, r, lower, upper);

        // 横跨左右两部分区间的答案信息
        int k1 = mid, k2 = mid;
        for(int i = l; i < mid; i++) {
            // 我们要的是 >= lower的k1, 所以只要小于lower, 就不要, k1之前的都不要
            while(k1 != r && arr[k1] - arr[i] < lower) k1++;
            // 我们要的是 <= upper的, 只要<=upper, 我们就要, k2之前的我们要
            while(k2 != r && arr[k2] - arr[i] <= upper) k2++;
            ans += (k2 - k1);
        }

        // 合并有序数组
        inplace_merge(arr.begin() + l, arr.begin() + mid, arr.begin() + r);
        return ans;
    }

    int countRangeSum(vector<int>& nums, int lower, int upper) {
        long s = 0;
        vector<long> arr{0};
        for(auto v: nums) {
            s += v;
            arr.push_back(s);
        }

        int ans = merge_sort(arr, 0, arr.size(), lower, upper);
        return ans;
    }
};
```

7. 计算右侧小于当前元素的个数 (难)

```
class Solution {
public:
    // 从大到小排序
    struct Data {
        Data(int val, int ind, int cnt) : val(val), ind(ind), cnt(cnt) {}
        int val, ind, cnt;
    };

    vector<Data> temp;

    void mergeSort(vector<Data> &arr, int l, int r) {
        if (l >= r) return ;
        int mid = (l + r) / 2;
        mergeSort(arr, l, mid);
        mergeSort(arr, mid + 1, r);
        int k = l, p1 = l, p2 = mid + 1;
        // 第一个区间不为空，或者第二个区间不为空
        while (p1 <= mid || p2 <= r) {
            // 为什么是大于？因为完全大于的时候才统计元素，等于的时候不统计答案
            if ((p2 > r) || (p1 <= mid && arr[p1].val > arr[p2].val)) {
                // 右侧区间还剩下多少个元素？ r - p2 + 1
                arr[p1].cnt += (r - p2 + 1);
                temp[k++] = arr[p1++];
            } else {
                temp[k++] = arr[p2++];
            }
        }
        for (int i = l; i <= r; i++) arr[i] = temp[i];
        return ;
    }

    vector<int> countSmaller(vector<int>& nums) {
        while (temp.size() < nums.size()) temp.push_back(Data{0, 0, 0});
        vector<Data> arr;
        for (int i = 0; i < nums.size(); i++) {
            arr.push_back(Data{nums[i], i, 0});
        }
        mergeSort(arr, 0, arr.size() - 1);

        vector<int> ret(nums.size());
        for (int i = 0; i < arr.size(); i++) {
            ret[arr[i].ind] = arr[i].cnt;
        }
        return ret;
    }
};
```

8. 最大子序和：

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        vector<long long> sum;
        sum.push_back(0);
        for (auto x : nums) sum.push_back(sum[sum.size() - 1] + x);

        long long ans = sum[1];
        for (long long pre = 0, i = 1; i < sum.size(); i++) {
            ans = max(sum[i] - pre, ans);
            pre = min(sum[i], pre);
        }
        return ans;
    }
};
```

9. 首个共同祖先

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    // p和q分别存在当前节点的左边和右边，则当前节点就是最近公共祖先
    // p和q都在左边的子树里面，则最近公共祖先在左边
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode*
q) {
        if (root == nullptr) return nullptr;
        if (root == p || root == q) return root;
        TreeNode *l = lowestCommonAncestor(root->left, p, q);
        TreeNode *r = lowestCommonAncestor(root->right, p, q);
        if (l != nullptr && r != nullptr) return root;
        if (l != nullptr && r == nullptr) return l;
        return r;
    }
};
```

10. 层数最深叶子节点的和

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
 * };
 */
class Solution {
public:
    void getAns(TreeNode *root, int k, int& max_k, int& ans) {
        if (root == NULL) return ;
        if (k == max_k) ans += root->val;
        else if (k > max_k) {
            max_k = k, ans = root->val;
        }
        getAns(root->left,  k + 1, max_k, ans);
        getAns(root->right, k + 1, max_k, ans);
        return ;
    }
    int deepestLeavesSum(TreeNode* root) {
        int ans = 0, max_k = 0;
        getAns(root, 0, max_k, ans);
        return ans;
    }
};
```