unordered_map

遍历

1.pair

```
for(const auto& p:m){   //遍历pair<TKey,TValue>
    cout<<p.first<<" "<<<p.second<<endl;
}
```

2.C++17绑定

```
    for (const auto& [k, v] : m)cout << k << " " << v << endl;
    for (const auto& [_, v] : m)cout << v << endl;  //只需要遍历Value
```

去重

集合法

```
vector<int> a{1,2,2,1,1,3,3};
unordered_set<int> s(a.begin(),a.end());
a.assign(s.begin(),s.end());
```

排序法

```
sort(a.begin(),a.end());
a.erase(unique(a.begin(),a.end()),a.end());
```

离散化

```
#include<algorithm>

vector<int> tmp(nums);
sort(tmp.begin(), tmp.end());
for(int& num: nums)
    num = lower_bound(tmp.begin(), tmp.end(), num) - tmp.begin() + 1;
```

字符串

KMP

求next数组

```
vector<int> get_next(string& p){
    int n=p.size();
    vector<int> next(n,0);next[0]=-1;
    int j=-1,i=0;
    while(i<n-1){
        if(j==-1 || p[i]==p[j]){
```

```
            i++;j++;
            next[i]=(p[i]==p[j])?next[j]:j;
        }else{
            j=next[j];
        }
    }
    return next;
}
```

split

```
//不考虑前导和后导
inline vector<string> split(string s,string delim=" "){
    vector<string> res;
    int l=0,r=0,n=s.size();
    while((r=s.find(delim,l))!=string::npos){
        string wrd=s.substr(l,r-l);
        res.push_back(wrd);
        l=r+1;
    }
    res.push_back(s.substr(l));
    return res;
}
```

字符串哈希

```
//下标越大，权重越大
using ull = unsigned long long;
ull hash_string(string s, int base = 131, ull moder = 1e18 + 7) {
    ull hash_value = 0;
    ull base_power = 1;
    for (const auto& ch : s) {
        hash_value = (hash_value + base_power * ch) % moder;
        base_power = (base_power * base) % moder;
    }
    return hash_value;
}
```

#

区间合并

```
vector<vector<int>> merge(vector<vector<int>>& intervals) {
    sort(intervals.begin(),intervals.end());
    vector<vector<int>> res;
    int l=intervals[0][0],r=intervals[0][1];

    for(const auto& interval:intervals){
        if(interval[0]>r){
            res.push_back({l,r});
            l=interval[0];
        }
        r=max(r,interval[1]);
    }
```

```
    res.push_back({l,r});
    return res;
}
```

数据结构

树状数组

```
class FenwickTree {
private:
    int length;
    vector<int> tree;
public:
    FenwickTree(int length) {
        this -> length = length;
        tree = vector<int>(length + 1, 0);
    }
    int lowbit(int x) {
        return x & (-x);
    }
    void update(int idx, int val) {
        while (idx <= length) {
            tree[idx] += val;
            idx += lowbit(idx);
        }
    }
    int query(int idx) {
        int res = 0;
        while (idx > 0) {
            res += tree[idx];
            idx -= lowbit(idx);
        }
        return res;
    }
};


vector<int> tmp(h);
// 离散化
sort(tmp.begin(), tmp.end());
for (int& num : h) {
    num = lower_bound(tmp.begin(), tmp.end(), num) - tmp.begin() + 1;
}
auto tree_right = FenwickTree(n);
```

字典树

```
class Trie {
private:
    bool is_end;
    Trie* next[26];
public:
    Trie() {
```

```
        is_end = false;
        memset(next, 0, sizeof(next));
    }
    void insert(string word) {
        Trie* node = this;
        for (const auto& ch : word) {
            if (node -> next[ch - 'a'] == nullptr) {
                node -> next[ch - 'a'] = new Trie();
            }
            node = node -> next[ch - 'a'];
        }
        node -> is_end = true;
    }
    bool search(string word) {
        Trie* node = this;
        for (const auto& ch : word) {
            if (node -> next[ch - 'a'] == nullptr) {
                return false;
            }
            node = node -> next[ch - 'a'];
        }
        return node -> is_end;
    }
    bool startsWith(string prefix) {
        Trie* node = this;
            for (const auto& ch : prefix) {
                if (node -> next[ch - 'a'] == nullptr) {
                    return false;
                }
                node = node -> next[ch - 'a'];
            }
        return true;
    }
};
```

哈希

自定义优先队列

```
struct node {
    int need;
    int get;
    int idx;

};

struct cmp {
    bool operator()(node a, node b) {
        return a.need > b.need;
    }
};
priority_queue<node, vector<node>, cmp> can_do;
```

自定义哈希规则

```
struct TreeNodeHash {
    size_t operator()(const pair<TreeNode*, bool>& key) const {
        auto h1 = hash<TreeNode*>{}(key.first);
        auto h2 = hash<bool>{}(key.second);
        return h1 ^ h2;
    }
};

unordered_map<pair<TreeNode*, bool>, int, TreeNodeHash> memo;
```

ST表 / 可重复贡献问题

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>

using namespace std;

int opt(int a, int b) {
    return max(a, b);
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<int> a(n);
    for (int i = 0; i < n; ++i) {
        cin >> a[i];
    }
    int lenj = ceil(log2(n)) + 1;
    vector<vector<int>> f(n, vector<int>(lenj));
    vector<int> log(n + 1);
    for (int i = 2; i <= n; ++i) {
        log[i] = log[i >> 1] + 1;
    }
    for (int i = 0; i < n; ++i) {
        f[i][0] = a[i];
    }
    for (int j = 1; j < lenj; ++j) {
        for (int i = 0; i < n + 1 - (1 << j); ++i) {
            f[i][j] = opt(f[i][j - 1], f[i + (1 << (j - 1))][j - 1]);
        }
    }
    auto qry = [&](int l, int r) {
        int k = log[r - l + 1];
        return opt(f[l][k], f[r - (1 << k) + 1][k]);
    };
    for (int i = 0; i < m; ++i) {
        int l, r;
        cin >> l >> r;
        cout << qry(l - 1, r - 1) << '\n';
    }
    return 0;
```

```
}
```

图论 / 树

换根DP

```
class Solution {
using ull = unsigned long long;
public:
    void STAStation(int n, vector<vector<int>>& edges) {
        vector<int> cnt(n + 1, 1);
        vector<int> depth(n + 1);
        vector<ull> res(n + 1);
        function<void(int, int)> dfs1 = [&](int u, int fa) -> void {
            depth[u] = depth[fa] + 1;
            for (const auto& v : edges[u]) {
                if (v == fa) continue;
                dfs1(v, u);
                cnt[u] += cnt[v];
            }
        };
        function<void(int, int)> dfs2 = [&](int u, int fa) -> void {
            for (const auto& v : edges[u]) {
                if (v == fa) continue;
                res[v] = res[u] + n - 2 * cnt[v];
                dfs2(v, u);
            }
        };
        dfs1(1, 0);
        for (int i = 1; i <= n; i++) res[1] += depth[i];
        dfs2(1, 0);
        cout << *max_element(res.begin() + 1, res.end());
    }
};
```

倍增LCA

```
class Solution {
public:
    // s 是 树根,f[u][i] 表示u 节点向上跳2^i 的节点，dep[u] 表示深度
    void solve(int n, int m, int s, vector<vector<int>>& edges, vector<pair<int,
int>>& query) {
        const int N = n + 10, MX = int(log2(n));
        vector<int> dep(N); vector<vector<int>> f(N, vector<int>(MX + 1));

        function<void(int, int)> dfs = [&](int u, int fa) -> void {
            f[u][0] = fa;
            dep[u] = dep[fa] + 1;
            for (int i = 1; i <= MX; i++) f[u][i] = f[f[u][i - 1]][i - 1];
            for (int v : edges[u])
                if (v != fa) dfs(v, u);
            };
        function<void(int, int)> lca = [&](int u, int v) -> void {
            if (dep[u] < dep[v]) swap(u, v);
            for (int i = MX; i >= 0; i--) {
```

```
                if (dep[f[u][i]] >= dep[v])
                    u = f[u][i];
            }
            if (u == v) {
                printf("%d\n", u);
                return;
            }
            for (int i = MX; i >= 0; i--) {
                if (f[u][i] != f[v][i]) u = f[u][i], v = f[v][i];
            }
            printf("%d\n", f[u][0]);
            };
        dep[s] = 1;
        for (int v : edges[s]) {
            dfs(v, s);
        }
        for (const auto& p : query)
            lca(p.first, p.second);
    }
};
```

print 可变长参数

```
template<typename T, typename ...Args>
inline void print(T first, Args...rest) {
    ostream& stream = cout;
    stream << first;
    if constexpr (sizeof...(rest) > 0) {
        stream << ' ';
        print(rest...); // 递归调用打印剩余参数
    } else {
        stream << '\n'; // 所有参数打印完后添加换行符
    }
}
template<typename T>
inline void print(vector<T>& vec) {
    for (const auto& elem : vec) {
        cout << elem << " ";
    }printf("\n");
}

template<typename T>
inline void print(queue<T> q) {
    while(!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }
    printf("\n");
}
template<typename T>
inline void print(stack<T> stk) {
    while(!stk.empty()) {
        cout << stk.top() << " ";
        stk.pop();
    } printf("\n");
}
```

字符串

替换字符串内容（原地算法）

```
void replace(string& base, string src, string dst)
{
    int pos = 0, srclen = src.size(), dstlen = dst.size();
    while ((pos = base.find(src, pos)) != string::npos)
    {
        base.replace(pos, srclen, dst);
        pos += dstlen;
    }
}
```

单调结构

单调队列

```
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    vector<int> res; int n = nums.size();
    deque<int> q;
    for (int i = 0; i < n; i++) {
        int x = nums[i];
        // 1.入队
        while (q.size() > 0 && x >= nums[q.back()]) {
            q.pop_back();
        }
        q.push_back(i);
        // 2.出队
        if (i - q.front() + 1 > k) q.pop_front();

        // 3.记录结果
        if (i >= k - 1) res.push_back(nums[q.front()]);
    }
    return res;
}
```

单调栈

```
nt trap(vector<int>& height) {
        stack<int> stk; int res = 0, n = height.size();
        for (int i = 0; i < n; i++ ) {
            //1. 不为空且违反单调性
            while (!stk.empty() && height[i] > height[stk.top()]) {
                //2. 出栈
                int top = stk.top(); stk.pop();
                //3. 特判
                if (stk.empty()) break;
                //4. 获得左边界、宽度
                int left = stk.top();
                int width = i - left - 1;
```

```
                //5. 计算
                res += width * (min(height[i], height[stk.top()]) -
height[top]);
            }
            //6. 入栈
            stk.push(i);
        }
        return res;
    }
```

数学

异或性质

1.交换律：$A \oplus B = B \oplus A$

2.结合律：$(A \oplus B) \oplus C = A \oplus (B \oplus C)$

3.自反性：$A \oplus B \oplus B = A$

4.推论：如果$A \oplus B = X$, 则有$X \oplus A = B, X \oplus B = A$

1.交换变量

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

2.排除偶次重复

在整数数组中，仅存在一个不重复的数字，其他数字均出现两次（或者偶数次），找出不重复数字。

利用任何一个数和某一个数异或两次都会消去的定律。

```
int res = 0;
for (int& num : nums) res ^= num;
```

快速幂

递归

```
using ull = unsigned long long; using ll = long long;
double fast_pow_ull(double a, ull n) {
    if (!n)return 1;
    double half = fast_pow_ull(a, n >> 1);
    return (n & 1) ? half * half * a : half * half;
}
double fast_pow(double a, ll n) {       //支持负数幂
    return n >= 0 ? fast_pow_ull(a, n) : fast_pow_ull(1 / a, -n);
}
```

迭代+二进制

```
double fast_pow(double a, ll n) {
    if (n == 0)return 1;
    double res = 1;
    if (n < 0) { n = -n; a = 1 / a; }
    while (n) {
        if (n & 1)res *= a;
        a *= a;
        n >>= 1;
    }
    return res;
}
```

矩阵快速幂（2x2）

```
vector<vector<ull>> pow(vector<vector<ull>>& a,ull n){
    vector<vector<ull>> res{    //其他形状的改成nxn的E矩阵
        {1,0},
        {0,1}
    };
    while(n){
        if(n&1)res=mul(res,a);
        a=mul(a,a);
        n>>=1;
    }
    return res;
}
```

矩阵乘法

```
const ull moder=1e9+7;
vector<vector<ull>> mul(vector<vector<ull>>& a,vector<vector<ull>>& b){
        int m_a=a.size(),n_a=a[0].size();
        int m_b=b.size(),n_b=b[0].size();
        int c=n_a;      //可以加一个n_a和m_b的判等
        vector<vector<ull>> res(m_a,vector<ull>(n_b,0));
        for(int i=0;i<m_a;i++){
            for(int j=0;j<n_b;j++){
                ull tmp=0;
                for(int k=0;k<c;k++){
                    //tmp=(tmp+(a[i][k]*b[k][j])%moder)%moder;  //如果需要取模
                    tmp+=a[i][k]*b[k][j];
                }
                res[i][j]=tmp;
            }
        }
        return res;
    }
```

语法 / 其他板子

卡常

```
std::cin.tie(nullptr);
std::ios::sync_with_stdio(false);
```

Lambda表达式

```
auto func1=[&](vector<int> seg)->int{return seg[0]};

// function
function<vector<int>(int)> dfs=[&](int i)->vector<int>{
    return xxxx;
};

sort(intervals.begin(),intervals.end(),
     [](vector<int> a,vector<int> b)->bool{
         return a[0]<b[0];
     });
```

Count

```
#include <algorithm>
char target='1';
int c=count(str.begin(),str.end(), target);
```

queue

```
q.push(elem);
q.pop();
q.front();
q.back();
q.empty();
```

deque

```
deq.push_back();
deq.pop_back();
deq.push_front();
deq.pop_front();
deq.front();
deq.back();
```

全排列

```cpp
#include<algorithm>
vector<int> a{1,2,3,4};
do{
    //每次拿到a的一个全排列，覆盖到a中
    for(const auto& elem:a)cout<<e<<" ";
    cout<<endl;
}while(next_permutation(a.begin(),a.end()));
```