基础算法

```
/**
 * 不定向输入        UndirectedInput
 * 二分            binary
 * 离散化          discretization
 * 进制转换        conversion
 * 高精度          high_precision
 * 搜索            Bfs_Dfs
 * STL
*/
#include <iostream>
#include <algorithm>
#include <string>
#include <vector>
#include <sstream> // 需要包含这个头文件
using namespace std;


namespace golitter {
namespace UndirectedInput {
#include <sstream> // 需要包含这个头文件
void solve() {
    stringstream put_str;
    string str;
    getline(cin, str); // 获取一行字符串
    int n(0), p;
    put_str<<str; // 将str重定向输入到put_str
    while(put_str>>p) n++; // 从put_str重定向读入数据
    cout<<n;
}

}}

namespace golitter {
namespace binary {

bool check(int mid) {
    ;
}
void solve() {
    int l,r;
    int ans;
    while(l <= r) {
        int mid = l + r >> 1;
        if(check(mid)) {
            // ans = mid; // 最小值最大
            l = mid + 1;
        } else {
            // ans = mid; // 最大值最小
            r = mid - 1;
```

```
        }
    }
    {
        // 最大值最小
        while(l < r) {
            int mid = (l + r) >> 1;
            if(check(mid)) r = mid;
            else l = mid + 1;
        } // output: r
    }
    {
        // 最小值最大
        while(l < r ) {
            int mid = (l + r + 1) >> 1; // [2, 2]; -->
            if(check(mid)) l = mid;
            else r = mid - 1;
        } // output: l
    }
}

}}

namespace golitter {
namespace discretization {

const int N = 333;
int a[N],last[N],id[N];
void test1() { // 重复数字一样 1 222 222     -----> 1 2 2 url:
https://www.luogu.com.cn/record/115366968
    int n; cin>>n; for(int i = 1; i <= n; ++i) cin>>a[i],id[i] = a[i];
    sort(id+1, id+1+n);
    int cnt = unique(id+1, id+n+1) - id - 1;
    for(int i = 1; i <= n; ++i) {
        last[i] = lower_bound(id+1, id+cnt+1, a[i]) - id;
    }
    {        // STL处理
            // n
        vector<int> a,id,last; id.assign(a.begin(), a.end());
        sort(id.begin(), id.end());
        id.erase(unique(id.begin(), id.end()), id.end());
        for(int i = 0; i < n; ++i) {
            last[i] = lower_bound(id.begin(), id.end(),a[i]) - id.begin();
        }
    }
}

void solve() {
        ;
    }

}}

namespace golitter {
namespace conversion {
/**
 * @brief 将数制为base的value转为十进制
 * @return 转换为十进制的数
*/
```

```
int conversion_from_other_2_base10(int base, int value) {
    string str = to_string(value);
    int res = 0;
    int p = 1;
    int len = str.size();
    for(int i = len - 1; i >= 0; --i) {
        res += p * (str[i] - '0');
        p *= base;
    }
    return res;
}
/**
 * @brief 将value转换为base进制的数
 * @return 转换为base进制的数
*/
int conversion_from_base10_2_other(int value, int base) {
    string str = "";
    while(value) {
        str += value % base + '0';
        value /= base;
    }
    int res = 0;
    int len = str.size();
    for(int i = len - 1; i >= 0; --i) {
        res = res * 10 + str[i] - '0';
    }
    return res;
}
/**
 * @brief 将数制为A_base的数A_value转为数制为B_base的数B_value
 * @param A_base 将要转换的值的数制类型
 * @param A_value 将要转换的值
 * @param B_base 转换后的数制类型
 * @param B_value 转换后的数值（引用类型）
 * @return void
*/
void conversion_from_baseA_2_baseB(int A_base, int A_value, int B_base, int&
B_value) {
    if(A_base != 10) {
        A_value = conversion_from_other_2_base10(A_base, A_value);
    }
    if(B_base != 10) {
        B_value = conversion_from_base10_2_other(A_value, B_base);
    } else B_value = A_value;
    // cout<<"进制: "<<A_base<<" 的数 ( "<<A_value<<" ) 转为 ==> 进制: "<<B_base<<"
的数 ( "<<B_value<<" )"<<endl;
}

// 由m进制转换成n进制
string conversion(string num, int m, int n){
    int l = num.size(), k = 0;
    string ans = "";
    for(int i = 0; i < l; ){
        k = 0;
        // k是 a/b 的余数，因为在 a/b 的过程中我们要不断更新商的值，所以要不断更新 num[j]
        // 单纯求余数的话我们 k * m + num[j] 计算若干次就够了
        for(int j = i; j < l; j ++){
            int t = (k * m + num[j] - '0') % n;
```

```
            num[j] = (k * m + num[j] - '0') / n + '0';
            k = t;
        }
        ans += (k + '0');
        // 如果 num[i] == 0 说明商在该位上没有值，比如 0001，那值就是 1，跳过去就好了
        while(num[i] == '0') i ++;
    }
    return ans; // 反转即可
}


}}

namespace golitter {
namespace high_precision {
/**
 * 使用python char a = 'a'; ord(a) == 97 将字符转为对应的ASCII码
 *                          chr(97) == 'a' 将ascii码转为对应的字符
*/


}}



#include <vector>
#include <stack>
#include <queue>
#include <set>
#include <map>
#include <unordered_map>
#include <unordered_set>
#include <string>
namespace golitter {
namespace STL {
void Vector() {
    /**
     * vector<int> vi || vi(n)
     * size()      返回元素个数
     * clear()     清空
     * front() back()    第一个，最后一个元素
     * []
    */
}
void String() {
    /**
     * string str;
     * substr(pos, len);
     * size()
     * reverse(bg, ed);
    */
}
void Queue() {
    /**
     * queue<int> q;
     * size()
     * clear()
     * push()
```

```
     * front()
     * pop()
    */

    /**
     * deque<int> dq; //
https://blog.csdn.net/mataojie/article/details/122310752?
ops_request_misc=%257B%2522request%255Fid%2522%253A%252216899453581680019222744 6
%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=16899
4535816800192227446&biz_id=0&utm_medium=distribute.pc_search_result.none-task-
blog-2~all~top_positive~default-1-122310752-null-
null.142^v90^insert_down1,239^v3^control&utm_term=deque&spm=1018.2226.3001.4187
     * size()
     * empty()
     * front() back()
     * push_front() push_back()
     * pop_front() pop_back()
     * 可以数组下标访问
     * 可以排序
    */

   // 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;
// 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;
// }
}
void Priority_queue() {
    /**
     * priority_queue<int> heap 大顶堆 [大的在上面] 默认大顶堆
     *      等价于 priority_queue<int,vector<int>,less<int>> heap 大顶堆
     * priority_queue<int, vector<int>, greater<int>> q; 小顶堆 [小的在上面]
     * size()    push()     pop()     top()
    */
}
// 用priority_queue 自定义堆 http://www.cbww.cn/news/37826.shtml
//      要重载 < 操作符 ，注意两个const才可以通过编译
// 方法一 重载运算符<
struct adt { // 小顶堆
    int a;
    bool operator<(const adt& rhs) const { // 优先队列的><与sort的><相反. ** 没有
const会报错
        return a > rhs.a; // 这里 从大到小进行排序，队列从最右边开始，所以是小顶堆
    }
};
// 方法二 使用lambda表达式
void test_priority_queue() {
    auto cmp = [](int pre, int suf) { return pre > suf; }; // 小顶堆
    priority_queue<int,vector<int>, decltype(cmp)> pq(cmp); // decltype 类型说明符

    // 实现自定义PII堆结构
    auto pii_cmp = [](PII pre, PII suf) {return pre.vf < suf.vf; };
    priority_queue<PII, vector<PII>, decltype(pii_cmp)> heap(pii_cmp);
```

```
}
void Stack() {
    /**
     * stack<int> s;
     * size()
     * clear()
     * push()
     * pop()
     * top()
    */
   // 单调栈

// 常见模型：找出每个数左边离它最近的比它大/小的数
// auto linear_stack = [&]() {
//     int tt = 0;
//     for (int i = 1; i <= n; i ++ )
//     {
//         while (tt && check(stk[tt], i)) tt -- ;
//         stk[ ++ tt] = i;
//     }
// }
}
void Map() {
    /**
     *
     *  map 自带大常数，但是卡不掉map，
     * stl 里 套 stl会很慢 ***
     * size()   clear()
     *
     * map:
     * 可以元组映射
     * map<PII,int> mpi; mpi[{1, 2}] = 3;
     *
     * unordered_map 不可以元组映射
     * multimap:
     *  multimap<PII,PII> mmpp;
     * mmpp.insert(pair<PII,PII>({x1,y1}, {x2,y2}));
     * count()    find()
     *
     * ** multimap不支持 [] 操作。** *
     *
     * map 和 unordered_map 比较：
     *              unordered_map最坏O(n)，会被卡
     *              # cf 有专门卡umap的
    */
   struct custom_hash { // 防止卡umap
    static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
chrono::steady_clock::now().time_since_epoch().count(); // <chrono>
        return splitmix64(x + FIXED_RANDOM);
    }
```

```
};
    unordered_map<int,int,custom_hash> umii;

}
void Set() {
    /**
     * set<int> s;
     * insert()   erase()
     * count()
    */
}
void Unordered_All() {
    /**
     *
    */
}
}}

namespace golitter {
namespace Bfs_Dfs {
// void dfs(int k)
//   {
//     if  (到目的地) 输出解;
//      else
//        for (i=1;i<=算符种数;i++)
//          if  (满足条件)
//              {
//                保存结果;
//                          Search(k+1)
//                                恢复：保存结果之前的状态{回溯一步}
//              }
//   }

// void bfs() {
//     // queue
// }
}}
```