Main

```
#include <bits/stdc++.h>
using namespace std;

using i64 = long long;

void solve() {
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int _ = 1;
    cin >> _;
    while (_ --)
        solve();
}
```

进制转换

```
string toBase(i64 num, int base){
    if(num == 0)
        return "0";
    string idx = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    string res;
    while(num){
        res += idx[num % base];
        num /= base;
    }
    reverse(res.begin(), res.end());
    return res;
}
```

并查集

```
struct DSU {
    vector<int> f, siz;

    DSU() {}
    DSU(int n) {
        init(n);
    }

    void init(int n) {
        f.resize(n);
        iota(f.begin(), f.end(), 0);
        siz.assign(n, 1);
    }

    int find(int x) {
        while (x != f[x]) {
            x = f[x] = f[f[x]];
        }
        return x;
    }

    bool cmp(int x, int y) {
        return find(x) == find(y);
    }

    bool merge(int x, int y) {//将y合并至x中
        x = find(x);
        y = find(y);
        if (x == y) {
            return false;
        }
        siz[x] += siz[y];
        f[y] = x;
        return true;
    }

    int size(int x) {
        return siz[find(x)];
    }
};
```

快速幂

```
i64 qmi(i64 a, i64 b, int p){
    i64 res = 1;
    for( ;b ; b >>= 1, a = a * a % p){
        if(b & 1)
            res = a * res % p;
    }
    return res;
}
```

线性筛

```
vector<int> p;

void build(int MAX){
    vector<bool> vis(MAX + 1);
    for(int i = 2; i <= MAX; i ++){
        if(!vis[i])
            p.push_back(i);
        for(int j = 0; j < p.size() && p[j] <= MAX / i; j ++){
            vis[p[j] * i] = true;
            if(i % p[j] == 0)
                break;
        }
    }
}
```

SparseTable

```cpp
#include <bits/stdc++.h>
using namespace std;

using i64 = long long;

void solve() {
    int n;
    cin >> n;
    vector<int> a(n);
    for(int i = 0; i < n; i ++)
        cin >> a[i];

    vector<vector<int>> st(n, vector<int>(20));
    for(int i = 0; i < n; i ++)//p = 0时
        st[i][0] = a[i];

    for(int p = 1; n >> p > 0; p ++)//枚举所有p
        for(int i = 0; i + (1 << p) <= n; i ++)//从下标 i 开始的长度为 2^p 的区间 不能超过n
            st[i][p] = max(st[i][p - 1], st[i + (1 << (p - 1))][p - 1]);

    auto query = [&](int l, int r)->int{
        int p = log2(r - l + 1);
        return max(st[l][p], st[r - (1 << p) + 1][p]);
    };

    for(int i = 0; i < n; i ++){
        for(int j = i; j < n; j ++)
            cout << "i: " << i + 1 << ' ' << "j: " << j + 1 << ' ' << "MAX: " << query(i, j) << '\n';
    }
}

signed main() {
//    freopen("../data/data.in", "r", stdin), freopen("../data/data.out", "w", stdout);
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int _ = 1;
    // cin >> _;
    while (_ --)
        solve();
}
```

Fenwick

```cpp
template <typename T>
struct Fenwick {
    int n;
    std::vector<T> a;

    Fenwick(int n_ = 0) {
        init(n_);
    }

    void init(int n_) {//可用于清空树状数组，并重新定义大小
        n = n_;
        a.assign(n + 1, T{});//下标为从1开始，需要初始化大小为n + 1
    }

    void add(int x, const T &v) {//单点修改(下标为从1开始)
        while(x <= n)
            a[x] += v, x += x & -x;
    }

    T sum(int x) {//查询1~x的和(下标为从1开始)
        T ans{};
        while(x > 0)
            ans += a[x], x -= x & -x;
        return ans;
    }
};
```

多项式乘法(NTT)

```
const int P = 998244353;
vector<int> rev, roots{0, 1};

int qmi(int a, int b) {
    int res = 1;
    for (; b; b >>= 1, a = 1ll * a * a % P){
        if (b & 1)
            res = 1ll * res * a % P;
    }
    return res;
}

void dft(vector<int> &a) {//正变换，迭代版NTT，将每个ai转换为对应的单位根
    int n = a.size();
    if (int(rev.size()) != n) {
        int k = __builtin_ctz(n) - 1;
        rev.resize(n);
        for (int i = 0; i < n; i++) {
            rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
        }
    }
    for (int i = 0; i < n; i++) {
        if (rev[i] < i) {
            swap(a[i], a[rev[i]]);
        }
    }
    if (roots.size() < n) {
        int k = __builtin_ctz(roots.size());
        roots.resize(n);
        while ((1 << k) < n) {
            int e = qmi(31, 1 << (__builtin_ctz(P - 1) - k - 1));
            for (int i = 1 << (k - 1); i < (1 << k); i++) {
                roots[2 * i] = roots[i];
                roots[2 * i + 1] = 1LL * roots[i] * e % P;
            }
            k++;
        }
    }
    for (int k = 1; k < n; k *= 2) {
        for (int i = 0; i < n; i += 2 * k) {
            for (int j = 0; j < k; j++) {
                int u = a[i + j];
                int v = 1LL * a[i + j + k] * roots[k + j] % P;
                a[i + j] = (u + v) % P;
```

```
                a[i + j + k] = (u - v) % P;
            }
        }
    }
}

void idft(vector<int> &a) {//逆变换
    int n = a.size();
    reverse(a.begin() + 1, a.end());
    dft(a);
    int inv = (1 - P) / n;
    for (int i = 0; i < n; i++) {
        a[i] = 1ll * a[i] * inv % P;
    }
}

vector<int> mul(vector<int> a, vector<int> b) {//卷积操作，操作的多项式为系数表达式
    int n = 1, tot = a.size() + b.size() - 1;
    while (n < tot) {
        n *= 2;
    }
    if (tot < 128) {
        vector<int> c(a.size() + b.size() - 1);
        for (int i = 0; i < (int)a.size(); i++) {
            for (int j = 0; j < (int)b.size(); j++) {
                c[i + j] = (c[i + j] + 1ll * a[i] * b[j]) % P;
            }
        }
        return c;
    }
    a.resize(n);
    b.resize(n);
    dft(a);
    dft(b);
    for (int i = 0; i < n; i++) {
        a[i] = 1ll * a[i] * b[i] % P;
    }
    idft(a);
    a.resize(tot);
    return a;
}
```

组合计数

```
const int mod = 998244353;
vector<i64> fac, inv;

int qmi(i64 a, i64 b, int p){
    i64 res = 1;
    for( ;b ; b >>= 1, a = a * a % p){
        if(b & 1)
            res = a * res % p;
    }
    return res;
}

void init(int n) {
    //求阶乘
    fac.assign(n + 1, 1);
    for (int i = 1; i <= n; i++)
        fac[i] = fac[i - 1] * i % mod;

    //求某个阶乘的逆元
    inv.assign(n + 1, 1);
    inv[n] = qmi(fac[n], mod - 2, mod);
    for (int i = n - 1; i >= 0; i--)
        inv[i] = inv[i + 1] * (i + 1) % mod;
}

int C(int N, int M) {//求n取m
    if (M > N || M < 0)
        return 0;
    return fac[N] * inv[M] % mod * inv[N - M] % mod;
}
```

懒标记线段树(基础区间和)

```
using i64 = long long;

struct Info{//主要信息
    i64 sum;
    Info(i64 s = 0) : sum(s) {}
};

struct Lazy{//懒信息
    i64 add;
    Lazy(i64 x1 = 0) : add(x1) {}
};

Info operator+(const Info &a, const Info &b){//合并两个子区间的规则
   return Info(a.sum + b.sum);
}

struct SegmentTree{
    int n;
    vector<Info> info;
    vector<Lazy> lazy;//懒信息

    SegmentTree() {}
    SegmentTree(int _n) {
        init(_n);
    }

    void init(int _n){
        n = _n;
        info.assign(n << 2, Info());
        lazy.assign(n << 2, Lazy());
        build(1, 1, n);
    }
    //回溯，更新i的值
    void up(int p){
        info[p] = info[2 * p] + info[2 * p + 1];
    }
    //对i节点进行+v的懒操作
    void apply(int p, i64 val, int siz){
        info[p].sum += val * siz;
        lazy[p].add += val;
    }
    //懒信息下发，对左右进行懒操作，然后清空本身的懒标记
    void down(int p, int sizL, int sizR){
        if (lazy[p].add != 0){
```

```
            apply(2 * p, lazy[p].add, sizL);
            apply(2 * p + 1, lazy[p].add, sizR);
            lazy[p].add = 0;//清空
        }
    }
    //构建线段树 O(N)
    void build(int p, int l, int r){
        if (l == r){
            info[p] = Info(0);
        }else{
            int mid = (l + r) / 2;
            build(2 * p, l, mid);
            build(2 * p + 1, mid + 1, r);
            up(p);//回溯更新info[i]
        }
    }
    //区间修改 最坏2*O(logN)
    void modify(int p, int l, int r, int L, int R, i64 val){
        if (L > r || R < l)
            return;

        if (L <= l && r <= R){//l~r被ql~qr完全包含时，进行懒操作，不用往下递归了
            apply(p, val, r - l + 1);
        }else{
            int mid = (l + r) / 2;
            down(p, mid - l + 1, r - mid);//l~r不完全被ql~qr包含，懒信息下发,
注意右边应该为r - (mid + 1) + 1
            modify(2 * p, l, mid, L, R, val);
            modify(2 * p + 1, mid + 1, r, L, R, val);
            up(p);//回溯更新info[i]
        }
    }
    //区间查询 最坏2*O(logN) 注意: 不需要回溯
    i64 query(int p, int l, int r, int L, int R){
        if(L > r || R < l)
            return 0;

        i64 res = 0;
        if(L <= l && r <= R){//l~r被ql~qr完全包含时，直接返回整段值，不用往下递归
了
            res = info[p].sum;
        }else{
            int mid = (l + r) / 2;
            down(p, mid - l + 1, r - mid);//l~r不完全被ql~qr包含，懒信息下发
            res += query(2 * p, l, mid, L, R);
            res += query(2 * p + 1, mid + 1, r, L, R);
        }
        return res;
    }
```

```
    //0-based
    void modify(int L, int R, i64 val){
        if (L ⩽ R)
            modify(1, 1, n, L + 1, R + 1, val);
    }
    i64 query(int L, int R){
        if(L ⩽ R)
            return query(1, 1, n, L + 1, R + 1);
        else
            return -1ll;
    }
};
```

懒标记线段树(区间重置+RMQ)

```
using i64 = long long;

struct Info{//主要信息
    i64 mx;
    Info(int x = 0) : mx(x) {}
};

struct Lazy{//懒信息
    i64 add, setL;
    bool need;
    Lazy(i64 x1 = 0, i64 x2 = 0, bool x3 = false) : add(x1), setL(x2),
need(x3) {}
};

Info operator+(const Info &a, const Info &b){//合并两个子区间的规则
    if(a.mx ⩾ b.mx)
        return a;
    if(b.mx > a.mx)
        return b;
}

struct SegmentTree{
    int n;
    vector<Info> info;
    vector<Lazy> lazy;

    SegmentTree() {}
    SegmentTree(int _n) {
        init(_n);
    }

    void init(int _n){
        n = _n;
        info.assign(n << 2, Info());
        lazy.assign(n << 2, Lazy());
        build(1, 1, n);
    }
    //对i节点进行+v的懒操作
    void applyAdd(int p, i64 val){
        info[p].mx += val;
        lazy[p].add += val;
    }
    //对i节点进行=v的懒操作
    void applySet(int p, i64 val){
```

```
        info[p].mx = val;
        lazy[p] = Lazy(0, val, true);
    }
    //懒信息下发，对左右进行懒操作，然后清空本身的懒标记，
    //如果同时有重置信息和添加信息，那么一定是先重置，不然add将无效
    void down(int p){
        if(lazy[p].need){
            applySet(2 * p, lazy[p].setL);
            applySet(2 * p + 1, lazy[p].setL);
            lazy[p].need = false;
        }
        if(lazy[p].add != 0){
            applyAdd(2 * p, lazy[p].add);
            applyAdd(2 * p + 1, lazy[p].add);
            lazy[p].add = 0;
        }
    }
    //回溯，更新i的值
    void up(int p){
        info[p] = info[2 * p] + info[2 * p + 1];
    }
    //构建线段树 O(N)
    void build(int p, int l, int r){
        if (l == r){
            info[p] = Info(0);
        }else{
            int mid = (l + r) / 2;
            build(2 * p, l, mid);
            build(2 * p + 1, mid + 1, r);
            up(p);//回溯更新info[i]
        }
    }
    //区间修改与重置(下标为0) 最坏2*O(logN)
    void modify(int p, int l, int r, int L, int R, i64 val, int sign){
        if (L > r || R < l)
            return;

        if (L <= l && r <= R){//l~r被ql~qr完全包含时，进行懒操作，不用往下递归了
            if(sign == 1)//增加
                applyAdd(p, val);
            if(sign == 2)//重置
                applySet(p, val);
        }else{
            int mid = (l + r) / 2;
            down(p);//l~r不完全被ql~qr包含，懒信息下发
            modify(2 * p, l, mid, L, R, val, sign);
            modify(2 * p + 1, mid + 1, r, L, R, val, sign);
            up(p);//回溯
        }
```

```
    }
    //区间查询(下标为0) 最坏2*O(logN) 注意: 不需要回溯
    i64 query(int p, int l, int r, int L, int R){
        i64 res = -1e17;
        if(L > r || R < l)
            return res;
        if(L <= l && r <= R){//l~r被ql~qr完全包含时，直接返回整段值，不用往下递归了
            res = info[p].mx;
        }else{
            int mid = (l + r) / 2;
            down(p);//l~r不完全被ql~qr包含，懒信息下发
            res = max(res, query(2 * p, l, mid, L, R));
            res = max(res, query(2 * p + 1, mid + 1, r, L, R));
        }
        return res;
    }

    //0-based
    void modify(int L, int R, i64 val, int sign){
        if(L <= R)
            modify(1, 1, n, L + 1, R + 1, val, sign);
    }
    i64 query(int L, int R){
        if(L <= R)
            return query(1, 1, n, L + 1, R + 1);
    }
};
```

一些小trick

O(n)：$n \leq 5 \times 10^8$ O($nlogn$)：$n \leq 2 \times 10^7$ O(n^2)：$n \leq 2 \times 10^4$ O(n^3)：$n \leq 500$

一维数组转二维数组下标：y = idx % n, x = idx / m 二维转一维：idx = n * y + x

C++ 向上取整: a / b = (a + b - 1) / b

builtin函数：https://www.luogu.com/article/0gvfhhdd

__builtin_popcount(x):二进制x中1的个数

__builtin_ffs(x):低位开始第一个 1 的位置 例x = 00000011000时，返回4

__builtin_ctz(x):后导零的个数

__builtin_clz(x):前导零的个数

线性求1~n逆元： inv[1] = 1; for (int i = 2; i <= n; i ++) inv[i] = (int)(p - (p / i) * (i64)inv[p % i] % p);//O(N) (p为mod)

set/map: auto it = s.lower_bound(x) : 以x为下界的区间内的最小元素，即第一个 >= x 的迭代器(s.end()时为不存在)。 --it: 最大的 < x 的迭代器(s.begin()时为不存在)

auto it = s.upper_bound(x) : 以x为上界的区间外的最小元素，第一个 > x 的迭代器(s.end()时为不存在)。 --it: 最大的 <= x 的迭代器(s.begin()时为不存在)