's Algorithm Template

> 该模板库中所有用到的数组，除了特殊说明，一般下标都是**从1开始**

数论

素数

判断素数

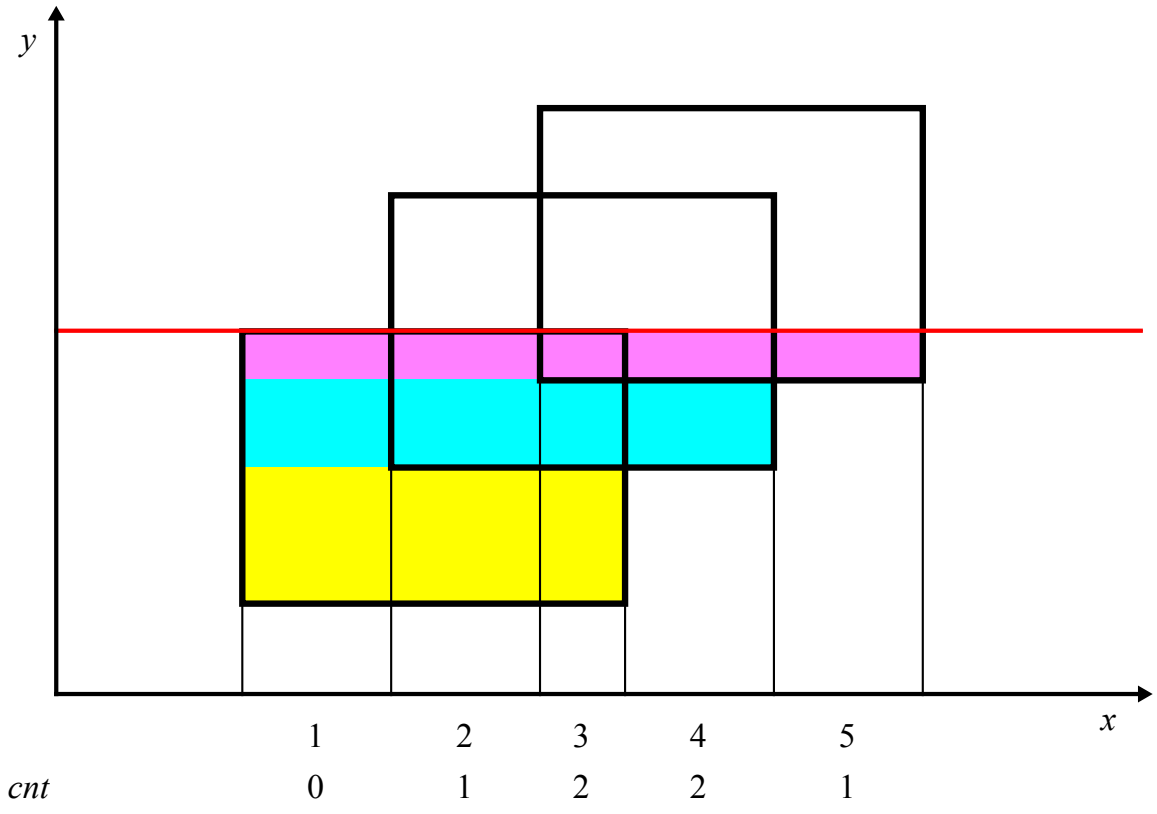
```cpp
auto is_prime = [&](int x) -> bool{
  if(x < 2)return false;
  for(int i = 2;i * i <= x;i++)
    if(x % i == 0)return false;
  return true;
};
```

欧拉筛

```cpp
std::vector<int> vis(n + 5), prime;
auto euler = [&](int n) -> void{
  for(int i = 2;i <= n;i++){
    if(!vis[i])prime.push_back(i);
    for(auto j : prime){
      if(j * i > n)break;
      vis[j * i] = true;
      if(i % j == 0)break;
    }
  }
};
```

计算几何

扫描线



> 扫描线的思路为，将需要操作的矩阵以y轴升序排序，然后用线段树统计区间

每一个操作可以被抽象成一个std::array<int, 4>, 代表y, x_begin, x_end, type

若type为1，表示这是起始线段，type为-1表示为末尾线段

那么对于每个矩阵，只需要放入两个操作

1. (y_begin, x_begin, x_end, 1)
2. (y_end, x_begin, x_end, -1)

例题：P1884 [USACO12FEB] Overplanting S

```
std::vector<std::array<int, 4>> a;
std::vector<int> x;
for(int i = 1;i <= n;i++){
  int x_begin, x_end, y_begin, y_end;
  std::cin >> x_begin >> x_end >> y_begin >> y_end;
  a.push_back({y_begin, x_begin, x_end, 1});
  a.push_back({y_end, x_begin, x_end, -1});
  x.push_back(x_begin);
  x.push_back(x_end);
}
std::sort(a.begin(), a.end(), [&](const std::array<int, 4> &xx, const std::array<int, 4> &yy) -> bool{
  if(xx[0] != yy[0])return xx[0] < yy[0];//将y轴升序排序
  return xx[3] < yy[3];//如果y轴相同，先将-1放在前面
});
SegmentTree st(x);
int ans = 0, last = a[0][0];
for(int i = 0;i < a.size();i++){
  auto [y, x1, x2, t] = a[i];
  if(i)ans += (y - last) * st.getlen();
  st.apply(x1, x2, t);
  last = y
}
std::cout << ans << endl;
```

上述代码中的线段树自带离散，代码如下：

```
class SegmentTree{
private:
  std::vector<int> xs;
  std::vector<int> cover;
  std::vector<int> len;
  int sz;
  void build(int k, int l, int r){
    if(l == r){
      len[k] = 0;
      return ;
    }
    int mid = (l + r) >> 1;
    build(k << 1, l, mid);
```

```
    build(k << 1 | 1, mid + 1, r);
    pushup(k, l, r);
  }

  void pushup(int k, int l, int r){
    if(cover[k])len[k] = xs[r + 1] - xs[l];
    else{
      if(l == r)len[k] = 0;
      else len[k] = len[k << 1] + len[k << 1 | 1];
    }
  }

  void update(int k, int l, int r, int x, int y, int val){
    if(x > r || y < l) return ;
    if(x <= l && r <= y){
      cover[k] += val;
      pushup(k, l, r);
      return ;
    }
    int mid = (l + r) >> 1;
    update(k << 1, l, mid, x, y, val);
    update(k << 1 | 1, mid + 1, r, x, y, val);
    pushup(k, l, r);
  }
public:
  SegmentTree(std::vector<int> &x){
    std::sort(x.begin(), x.end());
    x.erase(std::unique(x.begin(), x.end()), x.end());
    sz = x.size();
    xs.resize(sz + 1);
    for(int i = 0;i < sz;i++)
      xs[i + 1] = x[i];
    cover.resize((sz + 1) << 2, 0);
    len.resize((sz + 1) << 2, 0);
    build(1, 1, sz);
  }

  void apply(int x1, int x2, int val){
    int l = std::lower_bound(xs.begin() + 1, xs.end(), x1) - xs.begin();
    int r = std::lower_bound(xs.begin() + 1, xs.end(), x2) - xs.begin();
    if(l >= r) return ;
    update(1, 1, sz, l, r - 1, val);
  }

  int getlen(){
    return len[1];
  }
};
```

图论

建边

邻接表

```cpp
std::vector<int> e[n + 5];
//若x y表示x指向y的单向边
for(int i = 1;i <= m;i++){
  int x, y;
  std::cin >> x >> y;
  e[x].push_back(y);
}
//若x y表示x与y的双向边
for(int i = 1;i <= m;i++){
  int x, y;
  std::cin >> x >> y;
  e[x].push_back(y);
  e[y].push_back(x);
}
//若u表示当前节点 v表示要访问的节点 则邻接表的访问方式为
//for each写法
for(auto v : e[u]){
  //在此对v进行操作
}
//普通for写法
for(int i = 0;i < e[u].size();i++){
  v = e[u][i];
  //在此对v进行操作
}
```

链式前向星

```cpp
struct edge{
  int next, to;
};
std::vector<edge> e(m * 2 + 5);//双倍边
std::vector<int> head(n + 5, -1);
int cnt = 0;
auto add = [&](int x, int y) -> void{
  //此为x->y
  e[cnt].next = head[x];
  e[cnt].to = y;
  head[x] = cnt++;
  //此为y->x
  e[cnt].next = head[y];
  e[cnt].to = x;
  head[y] = cnt++;
};
//此处为遍历方式
for(int i = head[u];i != -1;i = e[i].next){
  int v = e[i].to;
  //此处对v进行操作
```

```
}
```

拓扑排序

```
std::vector<int> ind(n + 5), e[n + 5];
for(int i = 1;i <= n;i++){
  int x, y;//这里的x y表示有一条x指向y的单向边
  std::cin >> x >> y;
  e[x].push_back(y);
  ind[y]++;
}
std::queue<int> q;
for(int i = 1;i <= n;i++)
  if(!ind[i])
    q.push(i);
while(!q.empty()){
  int u = q.front();
  q.pop();
  for(auto v : e[u]){
    ind[v]--;
    //这里进行操作
    if(!ind[v])q.push(v);
  }
}
```

> 例题：P4017 最大食物链计数

tarjan实现缩点

```
//此处使用邻接表存储图
std::vector<int> belong(n + 5), e[n + 5], dfn(n + 5), vis(n + 5), low(n + 5), s(n + 5);
int tot = 0, index = 0, t = 0;
std::function<void(int)>tarjan = [&](int x) -> void{
  dfn[x] = low[x] = ++t;
  s[++index] = x;
  vis[x] = 1;
  for(auto v : e[x]){
    if(!dfn[v]){
      tarjan(v);
      low[x] = std::min(low[x], low[v]);
    }else if(vis[v])
      low[x] = std::min(low[x], dfn[v]);
  }
  if(low[x] == dfn[x]){
    tot++;
    while(1){
      belong[s[index]] = tot;
      vis[s[index]] = 0;
      index--;
      if(x == s[index + 1])break;
```

```
      //此处进行合并操作
    }
  }
};
for(int i = 1;i <= n;i++)
  if(!dfn[i])tarjan(i);
```

> 缩点，即将一个环进行操作，并将一整个环抽象成一个点
>
> 例题：P3387 【模板】缩点

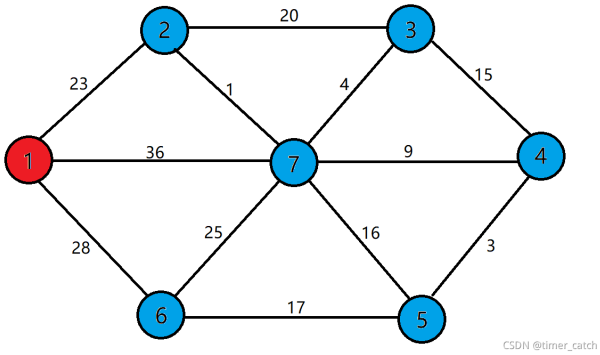
最小生成树

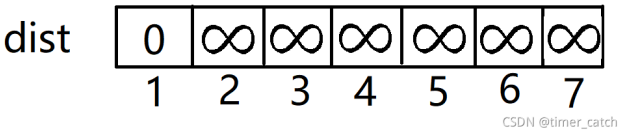
Prim

```
//此处使用链式前向星建图
int cnt = 0, cur = 1, tot = 0, ans = 0;
struct edge{
  int next, to, val;
};
std::vector<int> dis(n + 5, 1e9), head(n + 5, -1), vis(n + 5);
std::vector<edge> a(m + 5);
auto add = [&](int u, int v, int val) -> void{
  a[cnt].next = head[u];
  a[cnt].to = v;
  a[cnt].val = val;
  head[u] = cnt++;
};
for(int i = 1;i <= m;i++){
  int u, v, val;
  //此处以双向边为例子
  add(u, v, val);
  add(v, u, val);
}
//此处为prim算法
for(int i = head[1];i != -1;i = e[i].next)
  dis[e[i].to] = std::min(dis[e[i].to], e[i].val);
while(++tot < n){
  int mn = 1e9;
  vis[cur] = 1;
  for(int i = 1;i <= n;i++)
    if(!vis[i] && minn > dis[i]){
      minn = dis[i];
      cur = i;
    }
  ans += minn;
  for(int i = head[cur];i != -1;i = e[i].next){
    int v = e[i].to, val = e[i].val;
    if(!vis[v] && dis[v] > val)
      dis[v] = val;
  }
}
```

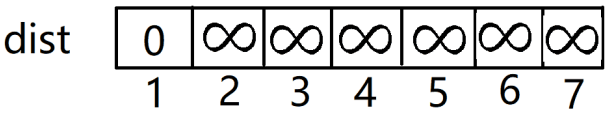
```
38  std::cout << ans << endl;
```

算法实现原理：

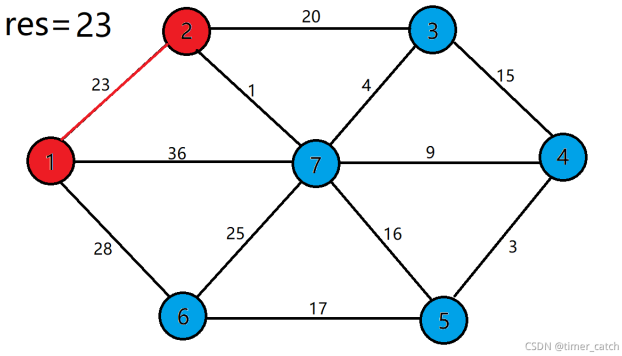




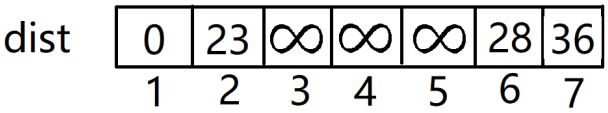
通过点1，对相邻点的dist进行更新，结果如下：



将与1最近的点2加入生成树中

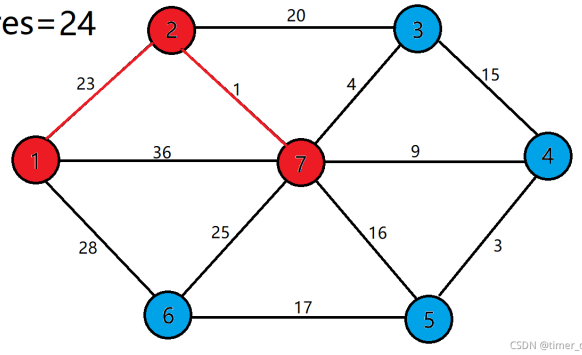


此时用2来更新dist数组



重复上述步骤，直到所有的点都加入到最小生成树中

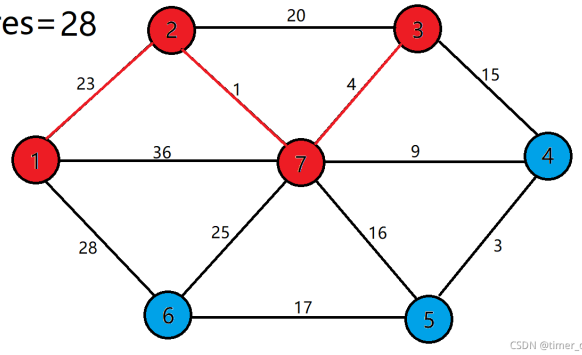
res=24



dist

0	23	4	9	16	25	1
1	2	3	4	5	6	7

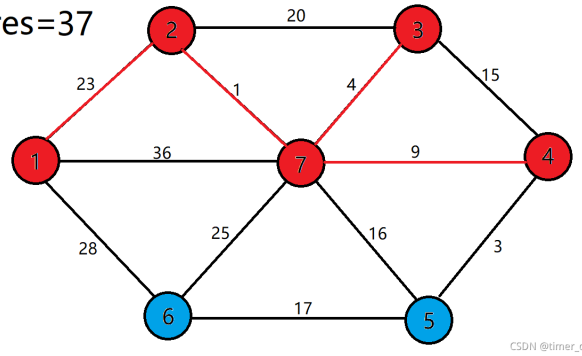
res=28



dist

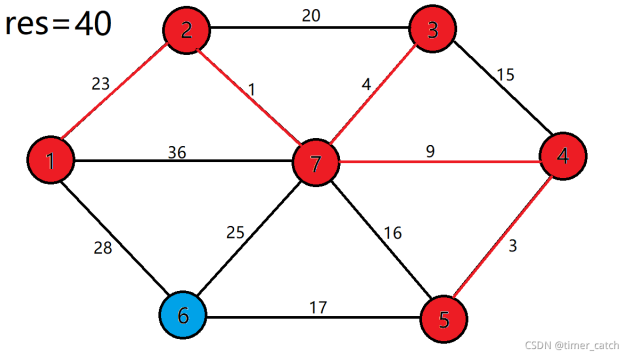
0	23	4	9	16	25	1
1	2	3	4	5	6	7

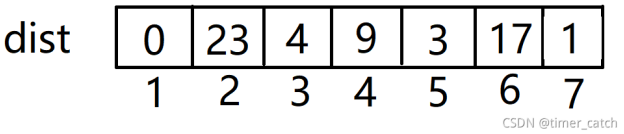
res=37

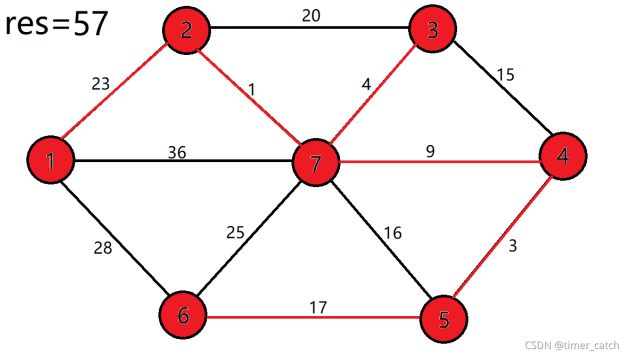


dist

0	23	4	9	3	25	1
1	2	3	4	5	6	7







Kruskal

```
int cnt = 0, ans = 0, cc = 0;
std::vector<int> f(n + 5);
std::vector<std::array<int, 3>> e;
for(int i = 1;i <= n;i++)f[i] = i;
for(int i = 1;i <= m;i++){
  int x, y, val;
  std::cin >> x >> y >> val;
  //若存在x->y的单向边
  e.push_back({x, y, val});
  //若存在y->x的单向边
  e.push_back({y, x, val});
}
std::sort(e.begin(), e.end(), [&](const std::array<int, 3> &x, const std::array<int, 3> &y) -> bool{
  return x[2] < y[2];
});
for(auto [u, v, val] : e){
  int cx = find(u), cy = find(v);//此处的find使用的是dsu的find，详见下面
  if(cx != cy){
    f[cy] = cx;
    ans += val;
    cc++;
    if(cc == n - 1)break;
  }
```

```
}
std::cout << ans << endl;
```

> dsu详解点此处，下面仅展示上述find代码的实现
>
> ```
> std::function<int(int)>find = [&](int x) -> int{
> if(x != f[x])f[x] = find(f[x]);
> return f[x];
> };
> ```

单源最短路

Dijkstra

> 只适用于不含**负权边**的图

SPFA

> 只适用于不含**正权边**的图

数据结构

并查集

> 例题：P1536 村村通

```
class DSU{
private:
  int n;
  std::vector<int> f, sz;
public:
  DSU(int x){
    n = x;
    f.resize(n + 5);
    sz.resize(n + 5, 1);
    for(int i = 1;i <= n;i++)f[i] = i;
  }

  int find(int x){
    if(f[x] != x)f[x] = find(f[x]);
    return f[x];
  }
  //合并x y
  void merge(int x, int y){
    int cx = find(x), cy = find(y);
    f[cy] = cx;
    sz[cy] += sz[cx];
  }
```

```
  //判断x y是否属于一个联通块
  bool same(int x, int y){
    return find(x) == find(y);
  }
  //判断某个联通块有几个节点
  int get_size(int x){
    return sz[x];
  }
};
```

线段树

SegmentTree（不带LazyTag）

Ice's线段树模板使用注意事项

注意此线段树下标**从1开始(1-based)**，并且**操作区间为左闭右闭区间**！！！

有两种构造方式，方式一为直接指定大小

```
SegmentTree<Info> sgt(n);
```

调用的构造函数原型为

```
SegmentTree(int _n, Info _v = Info()){
  init(_n, _v);
}
```

方式二为传入初始化数组以及大小（初始化数组长度任意，但是**一定要保证数据存在1-n**！！

```
std::vector<Info> a(n + 5);
for(int i = 1;i <= n;i++)
  //此处对a进行输入
SegmentTree<Info> sgt(n, a);
```

调用的构造函数原型为

```
template<class T>
SegmentTree(int _n, std::vector<T> _init){
  init(_n, _init);
}
```

init函数为

```
template<class T>
void init(int _n, std::vector<T> _init){
  n = _n;
  info.resize(4 * n + 5, Info());
```

```
5
6     std::function<void(int, int, int)>build = [&](int k, int l, int r) -> void{
7       if(l == r){
8         info[k] = _init[l];
9         return ;
10      }
11      int mid = (l + r) >> 1;
12      build(lc(k), l, mid);
13      build(rc(k), mid + 1, r);
14      pushup(k);
15    };
16
17    build(1, 1, n);
18  }
```

上述两种方法传入的第一个参数都为n，指的是线段树处理的区间是**1～n**

线段树模板

```
1   template<class Info>
2   class SegmentTree{
3     #define lc(x) (x << 1)
4     #define rc(x) (x << 1 | 1)
5   private:
6     int n;
7     std::vector<Info> info;
8   public:
9     SegmentTree(int _n, Info _v = Info()){
10      init(_n, _v);
11    }
12
13    template<class T>
14    SegmentTree(int _n, std::vector<T> _init){
15      init(_n, _init);
16    }
17
18    //若_init大小为n+5，则需要传入题目长度n，以及_init
19    template<class T>
20    void init(int _n, std::vector<T> _init){
21      n = _n;
22      info.resize(4 * n + 5, Info());
23
24      std::function<void(int, int, int)>build = [&](int k, int l, int r) -> void{
25        if(l == r){
26          info[k] = _init[l];
27          return ;
28        }
29        int mid = (l + r) >> 1;
30        build(lc(k), l, mid);
31        build(rc(k), mid + 1, r);
```

```
      pushup(k);
    };

    build(1, 1, n);
  }

  //可以直接传入n的大小
  void init(int _n, Info _v = Info()){
    init(_n, std::vector<Info>(_n + 5, _v));
  }

  void pushup(int k){
    info[k] = info[lc(k)] + info[rc(k)];
  }

  void update(int k, int l, int r, int x, const Info &v){
    if(l == r){
      info[k] = v;
      return ;
    }
    int mid = (l + r) >> 1;
    if(x <= mid)update(lc(k), l, mid, x, v);
    else update(rc(k), mid + 1, r, x, v);
    pushup(k);
  }

  void update(int k, const Info &v){
    update(1, 1, n, k, v);
  }

  Info query(int k, int l, int r, int x, int y){
    if(l > y || r < x)return Info();
    if(x <= l && r <= y)return info[k];
    int mid = (l + r) >> 1;
    return query(lc(k), l, mid, x, y) + query(rc(k), mid + 1, r, x, y);
  }

  Info query(int l, int r){
    return query(1, 1, n, l, r);
  }

  #undef lc(k)
  #undef rc(k)
};

struct Info {
  //在此处存放变量
};

Info operator+(const Info &a, const Info &b){
  Info c;
  //在此处重载规则
```

```
  return c;
}
```

> 在使用此线段树前，请确保你已经看过了Ice's线段树模板使用注意事项
>
> > 即此Tag的SegmentTree下面的灰色文字部分，这部分讲了此线段树初始化的方式以及传入的参数，并且说明了此线段树为**1-based**

Info类型变量的书写规则以及Info重载运算符的方法

Info结构体内定义的为你想要线段树能操作的变量，例如区间元素和sum，元素区间的最大值mx，区间最小值mn等

Info重载的运算符即你希望**pushup**的规则

例如常规线段树当中的

```
struct Node{
  int sum, mx, mn;
}t[maxn * 4];
//....
void pushup(int k){
  t[k].sum = t[k << 1].sum + t[k << 1 | 1].sum;
  t[k].mx = std::max(t[k << 1].mx, t[k << 1 | 1].mx);
  t[k].mn = std::min(t[k << 1].mn, t[k << 1 | 1].mn);
}
```

在此板子中需要这样写：

```
struct Info{
  int sum, mx, mn;
  Info(): sum(0), mx(0), mn(0) {}
  Info(int x): sum(x), mx(x), mn(x) {}
};

Info operator+(const Info &a, const Info &b){
  Info c;
  c.sum = a.sum + b.sum;
  c.mx = std::max(a.mx, b.mx);
  c.mn = std::min(a.mn, b.mn);
  return c;
}
```

update函数（单点修改）

其中，**update函数**为**单点**修改，有两种使用方式

第一种，直接指定需要操作的**下标x(1-based)**和需要**修改为的Info_val（不是相加，而是直接修改成）**

```
SegmentTree<Info> sgt(n);
sgt.update(index, Info_val);
```

如果想要相加，例如想要将index的值加上y，则需要如此操作：

```
struct Info{
  //....
  Info(int x = 0): x(x) {}
}
update(index, Info(a[index].val += val));
```

第二种，按照常规线段树的update，传入根，线段树左右区间，需要修改的下标，需要**修改为的Info_val**

```
SegmentTree<Info> sgt(n);
sgt.update(1, 1, n, index, Info_val);
```

若想想加，则按照上面的方法进行操作

query函数（区间查询）

对于**query**函数，可以进行区间查询，有两种使用方式

第一种，直接指定需要查询的左右区间l，r，返回**Info类型变量**

```
SegmentTree<Info> sgt(n);
Info ans = sgt.query(l, r);
```

第二种，按照常规线段树的query，传入根，线段树左右区间，需要查询的左右区间l，r，返回**Info类型变量**

```
SegmentTree<Info> sgt(n);
Info ans = sgt.query(1, 1, n, l, r);
```

使用示例

例如我需要修改单点的值，查询区间gcd以及区间和，示例为：

```
struct Info {
  int x, d;
  Info(int x = 0) : x(x), d(x) {}
};

Info operator+(const Info &a, const Info &b){
  Info c;
  c.x = a.x + b.x;
  c.d = gcd(a.d, b.d);
  return c;
```

```
}

std::vector<Info> a(n + 5);
for(int i = 1;i <= n;i++){
  int x;
  std::cin >> x;
  a[i] = Info(x);
}
SegmentTree<Info> sgt(n, a);
while(m--){
//此处当opt为1时，向第x位的数字+y
//当opt为2时，查询[x, y]的gcd和元素和
  int opt, x, y;
  std::cin >> opt >> x >> y;
  if(opt == 1){
    sgt.update(x, Info(a[x].x += y));
  }else std::cout << sgt.query(x, y).x << " " << sgt.query(x, y).d << endl;
}
```

LazySegmentTree（带LazyTag）

Ice's懒标记线段树模板使用注意事项

注意此线段树下标**从1开始(1-based)**，并且**操作区间为左闭右闭区间**！！！

有两种构造方式，方式一为直接指定大小

```
LazySegmentTree<Info, Tag> lsgt(n);
```

调用的构造函数原型为

```
LazySegmentTree(int _n, Info _v = Info()){
  init(_n, _v);
}
```

方式二为传入初始化数组以及大小（初始化数组长度任意，但是**一定要保证数据存在1-n**！！

```
std::vector<Info> a(n + 5);
for(int i = 1;i <= n;i++)
//此处对a进行输入
LazySegmentTree<Info, Tag> lsgt(n, a);
```

调用的构造函数原型为

```
template<class T>
LazySegmentTree(int _n, std::vector<T> _init){
  init(_n, _init);
}
```

init函数为

```
template<class T>
void init(int _n, std::vector<T> _init){
  n = _n;
  info.resize(4 * n + 5, Info());
  tag.resize(4 * n + 5, Tag());
  std::function<void(int, int, int)>build = [&](int k, int l, int r) -> void{
    if(l == r){
      info[k] = Info(_init[l], l, l);
      return ;
    }
    int mid = (l + r) >> 1;
    build(lc(k), l, mid);
    build(rc(k), mid + 1, r);
    pushup(k);
  };

  build(1, 1, n);
}
```

上述两种方法传入的第一个参数都为n，指的是线段树处理的区间是**1～n**

懒线段树板子

```
template<class Info, class Tag>
class LazySegmentTree{
  #define lc(x) (x << 1)
  #define rc(x) (x << 1 | 1)
private:
  int n;
  std::vector<Info> info;
  std::vector<Tag> tag;
public:
  LazySegmentTree(int _n, Info _v = Info()){
    init(_n, _v);
  }

  template<class T>
  LazySegmentTree(int _n, std::vector<T> _init){
    init(_n, _init);
  }

  //若_init大小为n+5，则需要传入题目长度n，以及_init
  template<class T>
  void init(int _n, std::vector<T> _init){
    n = _n;
    info.resize(4 * n + 5, Info());
    tag.resize(4 * n + 5, Tag());
```

```
    std::function<void(int, int, int)>build = [&](int k, int l, int r) -> void{
      if(l == r){
        info[k] = _init[l];
        return ;
      }
      int mid = (l + r) >> 1;
      build(lc(k), l, mid);
      build(rc(k), mid + 1, r);
      pushup(k);
    };

    build(1, 1, n);
  }

  //可以直接传入n的大小
  void init(int _n, Info _v = Info()){
    init(_n, std::vector<Info>(_n + 5, _v));
  }

  void pushup(int k){
    info[k] = info[lc(k)] + info[rc(k)];
  }

  void apply(int k, const Tag &v){
    info[k].apply(v);
    tag[k].apply(v);
  }

  void pushdown(int k){
    apply(lc(k), tag[k]);
    apply(rc(k), tag[k]);
    tag[k] = Tag();
  }

  //单点修改
  void update(int k, int l, int r, int x, const Info &v){
    if(l == r){
      info[k] = v;
      return ;
    }
    int mid = (l + r) >> 1;
    pushdown(k);
    if(x <= mid)update(lc(k), l, mid, x, v);
    else update(rc(k), mid + 1, r, x, v);
    pushup(k);
  }

  void update(int k, const Info &v){
    update(1, 1, n, k, v);
  }

  Info query(int k, int l, int r, int x, int y){
```

```
    if(l > y || r < x)return Info();
    if(x <= l && r <= y)return info[k];
    int mid = (l + r) >> 1;
    pushdown(k);
    return query(lc(k), l, mid, x, y) + query(rc(k), mid + 1, r, x, y);
  }

  Info query(int l, int r){
    return query(1, 1, n, l, r);
  }

  void Apply(int k, int l, int r, int x, int y, const Tag &v){
    if(l > y || r < x)return ;
    if(x <= l && r <= y){
      apply(k, v);
      return ;
    }
    int mid = (l + r) >> 1;
    pushdown(k);
    Apply(lc(k), l, mid, x, y, v);
    Apply(rc(k), mid + 1, r, x, y, v);
    pushup(k);
  }

  void Apply(int l, int r, const Tag &v){
    return Apply(1, 1, n, l, r, v);
  }

  #undef lc(k)
  #undef rc(k)
};

struct Tag{
  //定下要放什么标记
  void apply(Tag t){
    //怎么用父节点的标记更新儿子的标记
  }
};

struct Info {
  //在此处存放变量
  void apply(Tag t){
    //怎么用父节点的标记更新儿子存储的信息
  }
};

Info operator+(const Info &a, const Info &b){
  Info c;
  //在此处重载规则
  return c;
}
```

> 在使用此线段树前，请确保你已经看过了Ice's懒标记线段树模板使用注意事项
>
> > 即此Tag的LazySegmentTree下面的灰色文字部分，这部分讲了此线段树初始化的方式以及传入的参数，并且说明了此线段树为**1-based**
>
> 此懒线段树仍然保留了单点修改，其中**update函数**为**单点**修改，使用方式与上面的线段树使用方式一样

Info变量以及Tag变量的书写规则，以及Info运算符重载的书写规则

Info重载的运算符即你希望**pushup**的规则

Tag结构体中，重载的apply函数为你希望**pushdown**的规则

Info结构体中，重载的apply函数为你希望**pushdown**的规则

并且Tag和Info结构题中重载的apply函数，是以**子结点**为当前变量(this)，**父结点**为传入的Tag t

例如对于常规线段树，sum为区间和，add为加的**tag**

```
struct Node{
  int l, r, add, sum;
}t[maxn * 4];
void pushup(int k){
  t[k].sum = t[k << 1].sum + t[k << 1 | 1].sum;
}
void pushdown(int k){
  t[k << 1].sum += t[k << 1].add * (t[k << 1].r - t[k << 1].l + 1);
  t[k << 1].add += t[k].add;
  t[k << 1 | 1].sum += t[k << 1 | 1].add * (t[k << 1 | 1].r - t[k << 1 | 1].l + 1);
  t[k << 1 | 1].add += t[k].add;
  t[k].tag = 0;
}
```

在此板子中，则需要重载成这样（上面的sum变成此处的x）：

```
struct Tag{
  int add;
  Tag(): add(0) {}
  Tag(int a) : add(a) {}
  void apply(Tag t){
    add += t.add;
  }
};

struct Info {
  int x, l, r;
  Info(): x(0), l(0), r(0) {}
  Info(int val, int a, int b) : x(val), l(a), r(b) {}
  void apply(Tag t){
    x += (r - l + 1) * t.add;
  }
```

```
};

Info operator+(const Info &a, const Info &b){
    Info c;
    c.x = a.x + b.x;
    c.l = a.l;
    c.r = b.r;
    return c;
}
```

query函数（区间查询）

对于**query**函数，可以进行区间查询，有两种使用方式

第一种，直接指定需要查询的左右区间l，r，返回**Info类型变量**

```
LazySegmentTree<Info, Tag> lsgt(n);
Info ans = lsgt.query(l, r);
```

第二种，按照常规线段树的query，传入根，线段树左右区间，需要查询的左右区间l，r，返回**Info类型变量**

```
LazySegmentTree<Info, Tag> lsgt(n);
Info ans = lsgt.query(1, 1, n, l, r);
```

Apply函数（区间修改）

对于**Apply**函数，可以进行区间修改，有两种使用方式

第一种，直接指定需要修改的左右区间l，r，以及**需要更改为的Tag类型变量**

```
LazySegmentTree<Info, Tag> lsgt(n);
lsgt.Apply(l, r, Tag_val);
```

第二种，按照常规线段树方法，传入根，线段树左右区间，需要查询的左右区间l，r，以及**需要更改为的Tag类型变量**

```
LazySegmentTree<Info, Tag> lsgt(n);
lsgt.Apply(1, 1, n, l, r, Tag_val);
```

使用示例

例如，我需要区间加以及区间求和，例题为[P3372 【模板】线段树 1]

```
struct Tag{
```

```
  int add;
  Tag(): add(0) {}
  Tag(int a) : add(a) {}
  void apply(Tag t){
    add += t.add;
  }
};

struct Info {
  int x, l, r;
  Info(): x(0), l(0), r(0) {}
  Info(int val, int a, int b) : x(val), l(a), r(b) {}
  void apply(Tag t){
    x += (r - l + 1) * t.add;
  }
};

Info operator+(const Info &a, const Info &b){
  Info c;
    c.x = a.x + b.x;
    c.l = a.l;
    c.r = b.r;
    return c;
}

signed ICE(){
  int n, m;
  std::cin >> n >> m;
  std::vector<Info> a(n + 5);
  for(int i = 1;i <= n;i++){
    std::cin >> a[i].x;
    a[i].l = a[i].r = 1;
  }
  LazySegmentTree<Info, Tag> LSGT(n, a);
  while(m--){
    int opt, x, y, k;
    std::cin >> opt >> x >> y;
    //当opt为1时，对区间[x, y]增加k
    if(opt == 1){
      std::cin >> k;
      LSGT.Apply(x, y, Tag(k));
    }else{
      //当opt为2，求区间[x, y]的和
      std::cout << LSGT.query(x, y).x << endl;
    }
  }
  return awa;
}
```

平衡树

Splay

例题：P3369 【模板】普通平衡树

```cpp
class Splay{
private:
  int sz = 0, root = 0;
  std::vector<int> key, cnt, sizeT, f;
  std::vector<std::array<int, 2>> tree;

  void clear(int x){
    tree[x][0] = tree[x][1] = f[x] = cnt[x] = key[x] = sizeT[x] = 0;
  }
public:
  Splay(int n){
    key.resize(n + 5, 0);
    cnt.resize(n + 5, 0);
    sizeT.resize(n + 5, 0);
    f.resize(n + 5, 0);
    tree.resize(n + 5);
  }

  int get(int x){
    return tree[f[x]][1] == x ? 1 : 0;
  }

  void update(int x){
    if(x){
      sizeT[x] = cnt[x];
      if(tree[x][0]) sizeT[x] += sizeT[tree[x][0]];
      if(tree[x][1]) sizeT[x] += sizeT[tree[x][1]];
    }
  }

  void rotate(int x){
    int old = f[x], oldf = f[old], which = get(x);
    tree[old][which] = tree[x][which ^ 1];
    f[tree[old][which]] = old;
    f[old] = x;
    tree[x][which ^ 1] = old;
    f[x] = oldf;
    if(oldf)
      tree[oldf][tree[oldf][1] == old] = x;
    update(old);
    update(x);
  }

  void splay(int x, int goal){
    for(int fa; (fa = f[x]) != goal; rotate(x))
      if(f[fa] != goal)
        rotate(get(x) == get(fa) ? fa : x);
    if(!goal)root = x;
  }
```

```
50
51  void insert(int x){
52    if(!root){
53      sz++;
54      tree[sz][0] = tree[sz][1] = f[sz] = 0;
55      key[sz] = x;
56      cnt[sz] = 1;
57      sizeT[sz] = 1;
58      root = sz;
59      return ;
60    }
61    int now = root, fa = 0;
62    while(1){
63      if(key[now] == x){
64        cnt[now]++;
65        update(now);
66        update(fa);
67        splay(now, 0);
68        break;
69      }
70      fa = now;
71      now = tree[now][key[now] < x];
72      if(!now){
73        sz++;
74        tree[sz][0] = tree[sz][1] = 0;
75        key[sz] = x;
76        sizeT[sz] = 1;
77        cnt[sz] = 1;
78        f[sz] = fa;
79        tree[fa][key[fa] < x] = sz;
80        update(fa);
81        splay(sz, 0);
82        break;
83      }
84    }
85  }
86
87  int find(int x){
88    int ans = 0, now = root;
89    while(1){
90      if(x < key[now])
91        now = tree[now][0];
92      else{
93        ans += (tree[now][0] ? sizeT[tree[now][0]] : 0);
94        if(x == key[now]){
95          splay(now, 0);
96          return ans + 1;
97        }
98        ans += cnt[now];
99        now = tree[now][1];
100     }
101   }
```

```
}

int findx(int x){
  int now = root;
  while(true){
    if(tree[now][0] && x <= sizeT[tree[now][0]])
      now = tree[now][0];
    else{
      int tmp = (tree[now][0] ? sizeT[tree[now][0]] : 0) + cnt[now];
      if(x <= tmp)return key[now];
      x -= tmp;
      now = tree[now][1];
    }
  }
}

int pre(){
  int now = tree[root][0];
  while(tree[now][1])now = tree[now][1];
  return now;
}

int next(){
  int now = tree[root][1];
  while(tree[now][0])now = tree[now][0];
  return now;
}

void del(int x){
  find(x);
  if(cnt[root] > 1){
    cnt[root]--;
    update(root);
    return ;
  }
  if(!tree[root][0] && !tree[root][1]){
    clear(root);
    root = 0;
    return ;
  }
  if(!tree[root][0]){
    int oldroot = root;
    root = tree[root][1];
    f[root] = 0;
    clear(oldroot);
    return ;
  }else if(!tree[root][1]){
    int oldroot = root;
    root = tree[root][0];
    f[root] = 0;
    clear(oldroot);
    return ;
```

```
    }
    int leftbig = pre(), oldroot = root;
    splay(leftbig, 0);
    f[tree[oldroot][1]] = root;
    tree[root][1] = tree[oldroot][1];
    clear(oldroot);
    update(root);
    return ;
  }

  int id(int x){
    int now = root;
    while(1){
      if(x == key[now])return now;
      else{
        if(x < key[now])now = tree[now][0];
        else now = tree[now][1];
      }
    }
  }

  int get_key(int x){
    return key[x];
  }
};
```

需要使用，则

```
Splay splay(n);//此处的n为最大可能的操作次数
```

若要向M中插入一个数x

```
splay.insert(x);
```

若要删除M中一个数字（若多个相同，则只删除一个）

```
splay.del(x);
```

若要查询M中有多少个数比x小

```
splay.insert(x);
int ans = splay.find(x);
splay.del(x);
```

若要查询M从小到大排序后，排名第x位的数

```
splay.findx(x);
```

若要查询M的前驱（最大的小于x的数)

```
splay.insert(x);
int pre = splay.pre();
int ans = splay.get_key(pre);
splay.del(x);
```

若要查询M的后继（最小的大于x的数）

```
splay.insert(x);
int next = splay.next();
int ans = splay.get_key(next);
splay.del(x);
```

树状数组

模板

```
#define lowbit(x) (x & (-x))
class FenwickTree{
private:
  std::vector<int> t;
  int n;
public:
  void add(int i, int val){
    while(i <= n){
      t[i] += val;
      i += lowbit(i);
    }
  }

  int sum(int i){
    int res = 0;
    while(i > 0){
      res += t[i];
      i -= lowbit(i);
    }
    return res;
  }

  FenwickTree(int x){
    n = x;
    t.resize(n + 5);
  }
};
```

单点修改与区间求和

```
FenwickTree t(n);
//输入处理
for(int i = 1;i <= n;i++){
  int x;
  std::cin >> x;
  t.add(i, x);
}
//对a这个点加上val
t.add(a, val);
//要求[a, b]的区间和
int res = t.sum(b) - t.sum(a - 1);
```

区间修改和单点求和

```
FenwickTree t(n);
//输入处理
int last = 0;
for(int i = 1;i <= n;i++){
  int x;
  std::cin >> x;
  t.add(i, x - last);
  last = x;
}
//对[a, b]区间都加上val
t.add(a, val);
t.add(b + 1, -val);
//求x位置的数字是多少
int res = t.sum(x);
```

杂项

随机数以及对拍

> 头文件可以使用
>
> ```
> #include <bits/stdc++.h>
> ```
>
> 但当万能头文件不能使用时，需要使用下述同文件：
>
> ```
> #include <iostream>
> #include <chrono>
> #include <thread>
> #include <functional>
> #include <random>
> ```

随机数生成

单调时间戳生成种子

```
auto seed = std::chrono::steady_clock::now().time_since_epoch().count();
```

使用PID生成种子

```
auto thread_id = std::hash<std::thread::id>{}(std::this_thread::get_id());
```

使用高精度时钟时间戳

```
auto time_seed = std::chrono::high_resolution_clock::now().time_since_epoch().count();
```

随机数生成代码

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define awa 0
typedef long long ll;

signed ICE(){
  static std::mt19937 gen([]{
        auto time_seed = std::chrono::steady_clock::now().time_since_epoch().count();
        auto thread_id = std::hash<std::thread::id>{}(std::this_thread::get_id());
        auto seed =
std::chrono::high_resolution_clock::now().time_since_epoch().count();
        return seed + thread_id;
    }());
    std::uniform_int_distribution<int> dis(1, 200000);
    //在此处添加输出模块
  return awa;
}

signed main(){
  std::ios::sync_with_stdio(false),std::cin.tie(nullptr),std::cout.tie(nullptr);
  int T = 1;
  //std::cin >> T;
  while(T--)ICE();
  return 0;
}
```

> 其中std::mt19937中的return可以是三个种子自由组合
>
> uniform_int_distribution会产生这个区间内的随机数
>
> 用法:
>
> ```
> std::cout << dis(gen()) << endl;
> ```

> 且上述代码在windows, macOS, linux都可以使用

对拍脚本

> 对于下述脚本，**xxx__Generator.cpp是生成数据的，xxx__Good.cpp是暴力的正确代码，xxx.cpp是需要对拍的代码**

Linux/MacOS（check.sh）

> 使用时，记得更改下面的文件名，此脚本用main.cpp作为样例
>
> 最后的结果**会输出到终端以及统计目录的result.txt**

check.sh

```
#!/bin/bash

# 记得更改下面文件名
g++ -std=c++14 main__Generator.cpp -o generator
g++ -std=c++14 main__Good.cpp -o good
g++ -std=c++14 main.cpp -o test

> result.txt
epoch=1

while true; do
    echo "Testing epoch: $epoch"
    ./generator > input.txt
    ./good < input.txt > good.out
    ./test < input.txt > test.out

    if ! diff good.out test.out > /dev/null; then
        echo "WA found at epoch $epoch!" | tee -a result.txt
        {
            echo "INPUT:"
            cat input.txt
            echo "GOOD:"
            cat good.out
            echo "BAD:"
            cat test.out
        } >> result.txt
        cat result.txt
        break
    fi

    echo "AC"
    epoch=$((epoch+1))
done
```

> 若提示

```
permission denied: ./check.sh
```

则在终端中运行

```
chmod +x check.sh
```

Windows（check.bat）

> 使用时，记得更改下面的文件名，此脚本用main.cpp作为样例
>
> 最后的结果**会输出到终端以及统计目录的result.txt**

check.bat

```
@echo off
setlocal enabledelayedexpansion

:: 记得更改下面文件名
g++ -std=c++14 main__Generator.cpp -o generator.exe
g++ -std=c++14 main__Good.cpp -o good.exe
g++ -std=c++14 main.cpp -o test.exe

type nul > result.txt
set epoch=1

:loop
echo Testing epoch: %epoch%
generator.exe > input.txt
good.exe < input.txt > good.out
test.exe < input.txt > test.out

fc /b good.out test.out >nul
if errorlevel 1 (
    echo WA found at epoch %epoch%! >> result.txt
    echo WA found at epoch %epoch%!
    echo INPUT: >> result.txt
    type input.txt >> result.txt
    echo GOOD: >> result.txt
    type good.out >> result.txt
    echo BAD: >> result.txt
    type test.out >> result.txt
    type result.txt
    exit /b
)

echo AC
set /a epoch+=1
goto loop
```

前缀和

一维求和前缀和

```
std::vector<int> f(n + 5), a(n + 5);
for(int i = 1;i <= n;i++)
  std::cin >> a[i];
for(int i = 1;i <= n;i++)
  f[i] = f[i - 1] + a[i];
int l, r;
std::cin >> l >> r;
std::cout << f[r] - f[l - 1] << std::endl;
```

> 例题: P8218 【深进1.例1】求区间和

一维异或前缀和

```
std::vector<int> f(n + 5), a(n + 5);
for(int i = 1;i <= n;i++)
  std::cin >> a[i];
for(int i = 1;i <= n;i++)
  f[i] ^= f[i - 1] ^ a[i];
int l, r;
std::cin >> l >> r;
std::cout << f[r] ^ f[l - 1] << std::endl;
```

二维求和前缀和

```
std::vector<std::vector<int>> f(n + 5, std::vector<int>(m + 5)), a(n + 5,
std::vector<int>(m + 5));
for(int i = 1;i <= n;i++)
  for(int j = 1;j <= m;j++)
    std::cin >> a[i][j];
for(int i = 1;i <= n;i++){
  int sum = 0;
  for(int j = 1;j <= m;j++){
    sum += a[i][j];
    f[i][j] = f[i - 1][j] + sum;
  }
}
int x1, y1, x2, y2;
std::cin >> x1 >> y1 >> x2 >> y2;
std::cout << f[x2][y2] - f[x2][y1 - 1] - f[x1 - 1][y2] + f[x1 - 1][y1 - 1] <<
std::endl;
```

> 例题：P1719 最大加权矩形

差分

一维差分

```
std::vector<int> d(n + 5), a(n + 5);
for(int i = 1;i <= q;i++){
  int l, r;
  std::cin >> l >> r;
  d[l]++;
  d[r + 1]--;
}
for(int i = 1;i <= n;i++)
  a[i] = a[i - 1] + d[i];
for(int i = 1;i <= n;i++)
  std::cout << a[i] << " ";
std::cout << std::endl;
```

> 例题：P2367 语文成绩

二维差分

```
std::vector<std::vector<int>> d(n + 5, std::vector<int>(n + 5)), a(n + 5,
std::vector<int>(n + 5));
  for(int i = 1;i <= m;i++){
    int x1, x2, y1, y2;
    std::cin >> x1 >> y1 >> x2 >> y2;
    d[x1][y1]++;
    d[x2 + 1][y1]--;
    d[x1][y2 + 1]--;
    d[x2 + 1][y2 + 1]++;
  }
  for(int i = 1;i <= n;i++)
    for(int j = 1;j <= n;j++)
      a[i][j] = a[i - 1][j] + a[i][j - 1] - a[i - 1][j - 1] + d[i][j];
  for(int i = 1;i <= n;i++){
    for(int j = 1;j <= n;j++)
      std::cout << a[i][j] << " ";
    std::cout << std::endl;
  }
```

> 例题：P3397 地毯

滑动窗口

> 例题：P1638 逛画展
>
> 滑动窗口是一种贪心思想 通过动态调整双指针来处理问题 若长度为n 则其时间复杂度为O(n)
>
> 首先将右指针一直像右推，直到满足条件
>
> 然后左指针往右推，直到条件不满足
>
> 重复上述步骤，即可求得答案

```cpp
std::vector<int> a(n + 5);
for(int i = 1;i <= n;i++)
  std::cin >> a[i];
int l = 1, r = 1;
while(r <= n){
  //在这里对右指针指向的元素进行处理
  if(/*满足条件*/){
    while(l <= n && /*满足条件*/){
      //删去左指针指向的元素
      l++;
    }
    l--;//这里l--的原因是 上面的while会使得其**恰好**不满足条件 此时我退回一步操作 此时的区间**恰好**满足条件
    //更新答案
    l++;//这里将上面的操作回溯
  }
  r++;
}
```

二分

手写二分

```cpp
int l = 1, r = n, mid, ans = 0;
while(l <= r){
  mid = (l + r) >> 1;
  if(check(mid)){
    ans = mid;
    l = mid + 1;
  }else r = mid - 1;
}
```

STL二分写法

- lower_bound()

```cpp
int x = val;//val是你需要找的值
std::vector<int> a(n + 5);
for(int i = 1;i <= n;i++)
  std::cin >> a[i];
std::sort(a.begin() + 1, a.begin() + 1 + n);
int p = std::lower_bound(a.begin() + 1, a.begin() + 1 + n, x) - a.begin();
```

> lower_bound默认是对**非降序列**使用，返回的是第一个**大于等于**x的值对应的**迭代器**

- upper_bound()

```cpp
int x = val;//val是你需要找的值
std::vector<int> a(n + 5);
for(int i = 1;i <= n;i++)
  std::cin >> a[i];
std::sort(a.begin() + 1, a.begin() + 1 + n);
int p = std::upper_bound(a.begin() + 1, a.begin() + 1 + n, x) - a.begin();
```

> upper_bound默认是对**非降序列**使用，返回的是第一个**大于**x的值对应的**迭代器**

高精度

高精度加法

高精度减法

高精度乘法

高精度除法

STL函数

max_element

```cpp
std::vector<int> a(n + 5);
for(int i = 1;i <= n;i++)
  std::cin >> a[i];
int mx = *max_element(a.begin() + 1, a.begin() + 1 + n);
```

> max_element是返回[begin, end]中最大元素对应的**迭代器**

min_element

```cpp
std::vector<int> a(n + 5);
for(int i = 1;i <= n;i++)
  std::cin >> a[i];
int mn = *min_element(a.begin() + 1, a.begin() + 1 + n);
```

> min_element事返回[begin, end]中最小元素对应的**迭代器**

next_permutation

```cpp
std::vector<int> a(4);
a = {0, 1, 2, 3};//模板数组下标从1开始，即“有效部分”为{1,2,3}
do{
  for(int i = 1;i <= 3;i++)
    std::cout << a[i] << " ";
  std::cout << std::endl;
}while(next_permutation(a.begin() + 1, a.begin() + 1 + 3));
```

> next_permutation求的是[begin, end]的当前排列的**下一个排列**，若当前排列**不存在**下一个排列，则返回**false**，否则返回**true**

prev_permutation

```
std::vector<int> a(4);
a = {0, 3, 2, 1};//模板数组下标从1开始，即“有效部分”为{3,2,1}
do{
  for(int i = 1;i <= 3;i++)
    std::cout << a[i] << " ";
  std::cout << std::endl;
}while(prev_permutation(a.begin() + 1, a.begin() + 1 + 3));
```

> prev_permutation求的是[begin, end]的当前排列的**上一个排列**，若当前排列**不存在**上一个排列，则返回**false**，否则返回**true**

greater

> 对于**数组** 若**从左到右遍历下表时** 变成**降序** 即**从大到小**
>
> 对于**建堆时** 变成**大根堆** 即**从下层到上层** 堆元素**从大到小**

less

> 对于**数组** 若**从左到右遍历下表时** 变成**升序** 即**从小到大**
>
> 对于**建堆时** 变成**小根堆** 即**从下层到上层** 堆元素**从小到大**

unique

```
std::vector<int> a{0, 1, 1, 2, 2, 3, 3, 4};
std::sort(a.begin() + 1, a.begin() + 1 + 7);
a.erase(std::unique(a.begin() + 1, a.begin() + 1 + 7), a.end());
```

> 若**原数组无序，一定要先排序**
>
> unique函数**并不是移除**重复元素，而是将重复元素**置于数组末尾**，并且返回**去重后的末尾元素指针**

reverse

```
std::vector<int> a(n + 5);
for(int i = 1;i <= n;i++)
  std::cin >> a[i];
std::reverse(a.begin() + 1, a.begin() + 1 + n);
```

> reverse是将[begin, end]的元素倒过来

STL

vector

vector的初始化

代码	意义
vector<T> v1	v1是一个元素类型为T的空vector
vector<T> v2(v1)	使用v1中所有元素初始化v2
vector<T> v2=v1	同上
vector<T> v3(n, val)	v3中包含了n个值为val的元素
vector<T> v4(n)	v4大小为n，所有元素默认初始化为0
vector<T> v5{a, b, c}	使用a,b,c初始化v5
vector<vector<T>> v6(n, vector<T>(m, val))	初始化一个n*m大小，值为val的二维矩阵v6

vector常用基础操作

代码	意义
v.empty()	如果v为空则返回**true**,否则返回**false**
v.size()	返回v中元素的个数
v1 == v2	**当且仅当**拥有**相同数量**且**相同位置上值相同**的元素时返回true
v1 != v2	
<, <=, >, >=	以**字典序**进行比较
v.push_back()	将某个元素添加到v后面，并且将其大小+1
v.resize(val)	将v的大小resize成val的大小
v.begin()	返回指向容器**第一个元素**的**迭代器**
v.end()	返回指向容器**尾端（非最后一个元素）**的**迭代器**
v.rbegin()	返回指向容器**最后一个元素**的**逆向迭代器**
v.rend()	返回指向容器**前端（非第一个元素）**的**逆向迭代器**

stack

> 栈满足**先进后出（FILO）**原则

代码	意义
stack<T> s	创建一个类型为T的栈
s.push(val)	将val压入栈顶
s.top()	返回栈顶元素
s.pop()	弹出栈顶元素
s.size()	返回栈的大小
s.empty()	若栈空，则返回true，否则返回false

array

代码	意义
array<T, val> a0	初始化一个大小为val，类型为T的数组a0
array<T, 3> a1={1,2,3}	用{1,2,3}初始化a1，类型为T
array<T, val> a2 = a0	用a0初始化a2
a.begin()	返回指向容器**第一个元素**的**迭代器**
a.end()	返回指向容器**尾端（非最后一个元素）**的**迭代器**
a.rbegin()	返回指向容器**最后一个元素**的**逆向迭代器**
a.rend()	返回指向容器**前端（非第一个元素）**的**逆向迭代器**

set

> set内部封装了红黑树 默认是**有排序且从小到大排序**的 且set中**元素值不重复**

代码	意义
set<T> s	初始化一个类型T的set
s.clear()	删除s中的所有元素
s.empty()	若set为空，则返回true，否则返回false
s.insert(val)	将val插入set
s.erase(it)	将**迭代器it**指向的元素删掉
s.erase(key)	将**值为key**的元素删掉
s.find(val)	查找**值为val**的元素，并返回指向该元素的**迭代器**，若**没找到则返回end()**
s.lower_bound(val)	返回第一个**大于等于val**的元素对应的**迭代器**
s.upper_bound(val)	返回第一个**大于val**的元素对应的**迭代器**
s.begin()	返回指向容器**第一个元素**的**迭代器**
s.end()	返回指向容器**尾端（非最后一个元素）**的**迭代器**
s.rbegin()	返回指向容器**最后一个元素**的**逆向迭代器**
s.rend()	返回指向容器**前端（非第一个元素）**的**逆向迭代器**

set重写排序规则

> 想要实现自定义类型的元素排序规则重写，例如pair或者vector，只需要将代码里的int改为对应类型即可

第一种方法（普通函数指针）

```
bool cmp(const int &x, const int &y){
  return x > y;
}
std::set<int, bool(*)(const int &x, const int &y)> a(cmp);
```

第二种方法（仿函数）

```
class cmp{
public:
  bool operator()(int x, int y) const {
    return x > y;
  }
};
std::set<int, cmp> a;
```

第三种方法（库函数）

```
std::set<int, std::greater<int>> a;//greater是从大到小排序
```

multiset

> multiset内部同样封装了红黑树 默认是**有排序且从小到大排序**的 但multiset**允许元素值重复**
>
> 若想通过**key值**删除multiset的元素，则需要使用**s.erase(s.find(val))**

代码	意义
multiset<T> s	初始化一个类型为T的multiset
s.clear()	删除s中的所有元素
s.empty()	若multiset为空，则返回true，否则返回false
s.insert(val)	将val插入multiset
s.erase(it)	将**迭代器it**指向的元素删掉
s.find(val)	查找**值为val**的元素，并返回指向该元素的**迭代器**，若**没找到则返回end()**
s.lower_bound(val)	返回第一个**大于等于val**的元素对应的**迭代器**
s.upper_bound(val)	返回第一个**大于val**的元素对应的**迭代器**
s.begin()	返回指向容器**第一个元素**的**迭代器**
s.end()	返回指向容器**尾端（非最后一个元素）**的**迭代器**
s.rbegin()	返回指向容器**最后一个元素**的**逆向迭代器**
s.rend()	返回指向容器**前端（非第一个元素）**的**逆向迭代器**

multiset重写排序规则

见[set重写排序规则]

map

> map容器的每一个元素都是一个**pair**类型的数据

代码	意义
map<T1, T2> a	初始化一个类型T1映射到T2的map a
a.clear()	删除所有元素
a.erase(val)	删除**key为val**的元素
a.erase(it)	删除**迭代器it**对应的元素
a.find(val)	查找**值为val**的元素，并返回指向该元素的**迭代器**，若**没找到则返回end()**
a.empty()	若map为空，则返回true，否则返回false
a.count(val)	返回**key为val**是否存在于map，若存在则为1，否则为0
a.lower_bound(val)	返回第一个**大于等于**key的键值对对应的**迭代器**
a.upper_bound(val)	返回第一个**大于**key的键值对对应的**迭代器**

> 对于map的lower_bound的用法例子
>
> ```
> std::map<int, int> a;
> //此处对a进行处理
> auto it = a.lower_bound(3);
> if(it != a.end()){
> auto [key, value] = *it;
> std::cout << key << " " << value << endl;
> }
> ```
>
> 一般判断map中某个元素是否存在，**不用if(a[val])**，而是用**if(!a.count())**
>
> 因为前者会创建一个val的映射，后者并不会
>
> 例如我bfs的时候，需要判断val是否被走过，一般不用
>
> ```
> std::map<int, int> vis;
> if(!vis[val])
> q.push(val);
> ```
>
> 而是使用
>
> ```
> if(vis.count(val) && !vis[val])
> q.push(val)
> ```
>
> 这样，当我下面代码需要判断val是否存在时，就不会出错（因为如果我用了前者，很可能会创建一个(val, 0)的映射，影响下面的判断）

map重写排序规则

第一种方法（库函数）

```cpp
std::map<int, int, std::greater<int>> a;//这样能让map以key为关键词从大到小排序
```

第二种方法（仿函数）

```cpp
class cmp{
public:
  bool operator()(int x, int y) const {
    return x > y;
  }
};
std::map<int, int, cmp> a;
```

若要以**value**作为关键词排序

> **不能**使用stl的sort函数，因为sort函数只能对**线性**容器进行排序，而map是**集合容器**，存储的是pair且非线形存储，则只能将其放到vector里后排序

```cpp
std::map<int, int> vis;
//此处对map进行了操作
std::vector<PII> a(vis.begin(), vis.end());
std::sort(a.begin(), a.end(). [&](const PII &x, const PII &y) -> bool{
  return x.second < y.second;
});//此处是从小到大排序
for(auto [key, value] : a)
  std::cout << key << " " << value << endl;
```

queue

> 队列满足**先进先出（FIFO）**原则

代码	意义
queue<T> q	创建一个类型为T的队列q
q.push(val)	在队尾插入一个元素val
q.pop()	删除队列第一个元素
q.size()	返回队列中元素个数
q.empty()	若队列为空则返回true，否则返回false

priority_queue

代码	意义
priority_queue<T, std::vector<T>, std::greater<T>> q	创建一个类型为T，**从小到大**排序的优先队列q
priority_queue<T, std::vector<T>, std::less<T>> q	创建一个类型为T，**从大到小**排序的优先队列q
q.push(val)	将**值为val**的元素插入优先队列中
q.top()	返回优先队列中的最高优先级元素
q.pop()	删除优先队列中的最高优先级元素
q.empty()	若优先队列为空则返回true，否则返回false
q.size()	返回优先队列中的元素个数

deque

代码	意义
deque<T> q	创建一个双向队列q
q.emplace_back(val)/q.push_back(val)	在队列尾部插入值为val的元素
q.emplace_front(val)/q.push_front(val)	在队列头部插入值为val的元素
q.pop_back()	删除队列尾部元素
q.pop_front()	删除队列头部元素
q.begin()	返回指向容器**第一个元素**的**迭代器**
q.end()	返回指向容器**尾端（非最后一个元素）**的**迭代器**
q.rbegin()	返回指向容器**最后一个元素**的**逆向迭代器**
q.rend()	返回指向容器**前端（非第一个元素）**的**逆向迭代器**
q.size()	返回双端队列的元素个数
q.empty()	若双端队列为空，则返回true，否则返回false
q.clear()	清空队列

list

代码	意义
list<T> a	创建一个类型为T的列表a
a.push_front(val)	向a的头部添加值为val的元素
a.push_back(val)	向a的尾部添加值为val的元素
a.pop_front()	将a头部的元素删去
a.pop_back()	将a尾部的元素删去
a.size()	返回列表元素的个数
a.begin()	返回指向第一个元素的**迭代器**
a.end()	返回指向最后一个元素**下一个位置**的**迭代器**
a.rbegin()	返回指向最后一个元素的**迭代器**
a.rend()	返回指向第一个元素**前一个位置**的**迭代器**
a.sort()	将所有元素从小到大排序，可以填入std::greater<T>来从大到小排序
a.remove(val)	删除值为val的元素
a.remove_if(func)	若元素满足func，则删除
a.reverse()	将元素按原来相反的顺序排序

> 注意，list**没有提供[]**

TODO

Dijkstra

Floyd

Bellman-Ford

SPFA

二分图

LCA

网络流

背包

KMP

快速幂

排列组合

Int128的使用

费马小定理

欧拉函数

快读 快写

二叉树前中后序遍历

RMQ

splay的原理，P3391 【模板】文艺平衡树

min25筛

增加高精度加减乘除模板

所有模板的教程