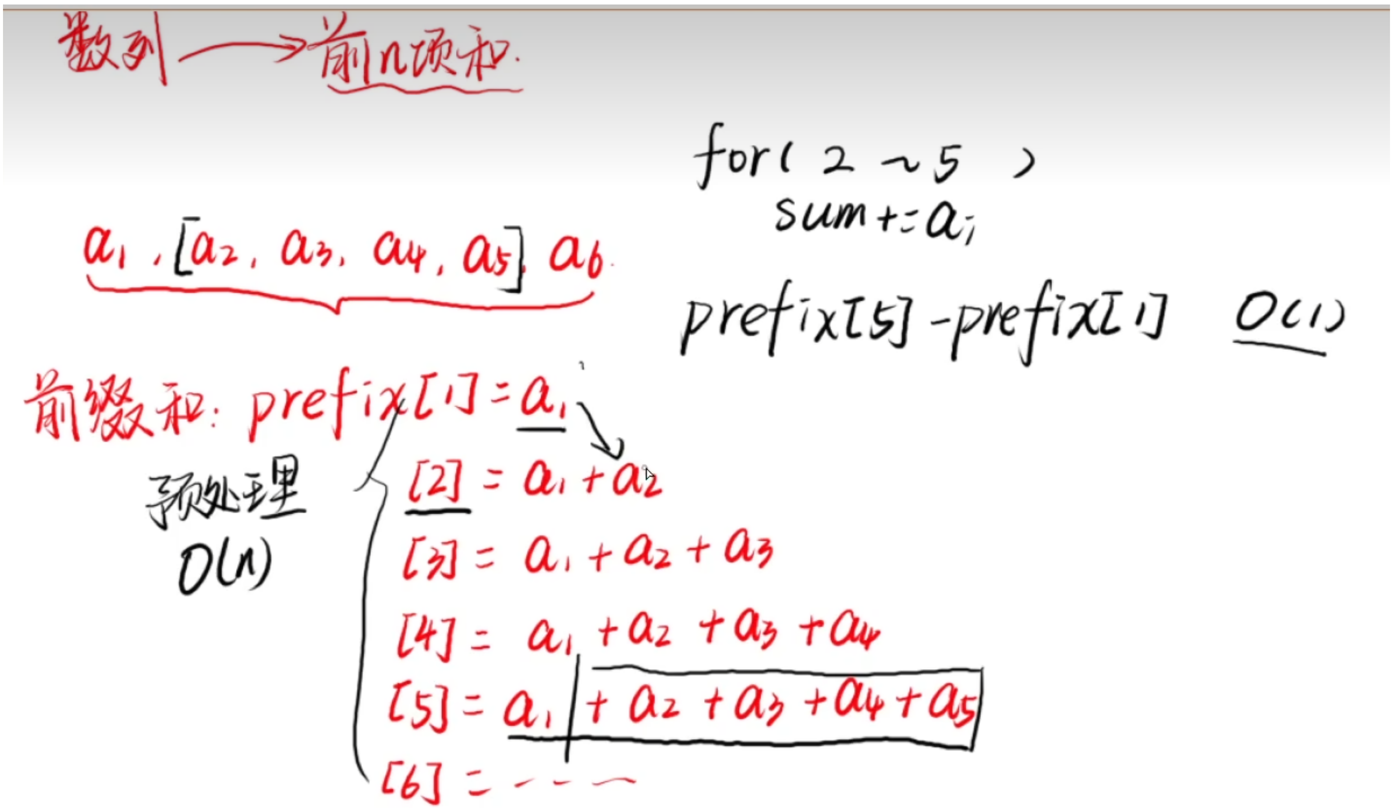
part1

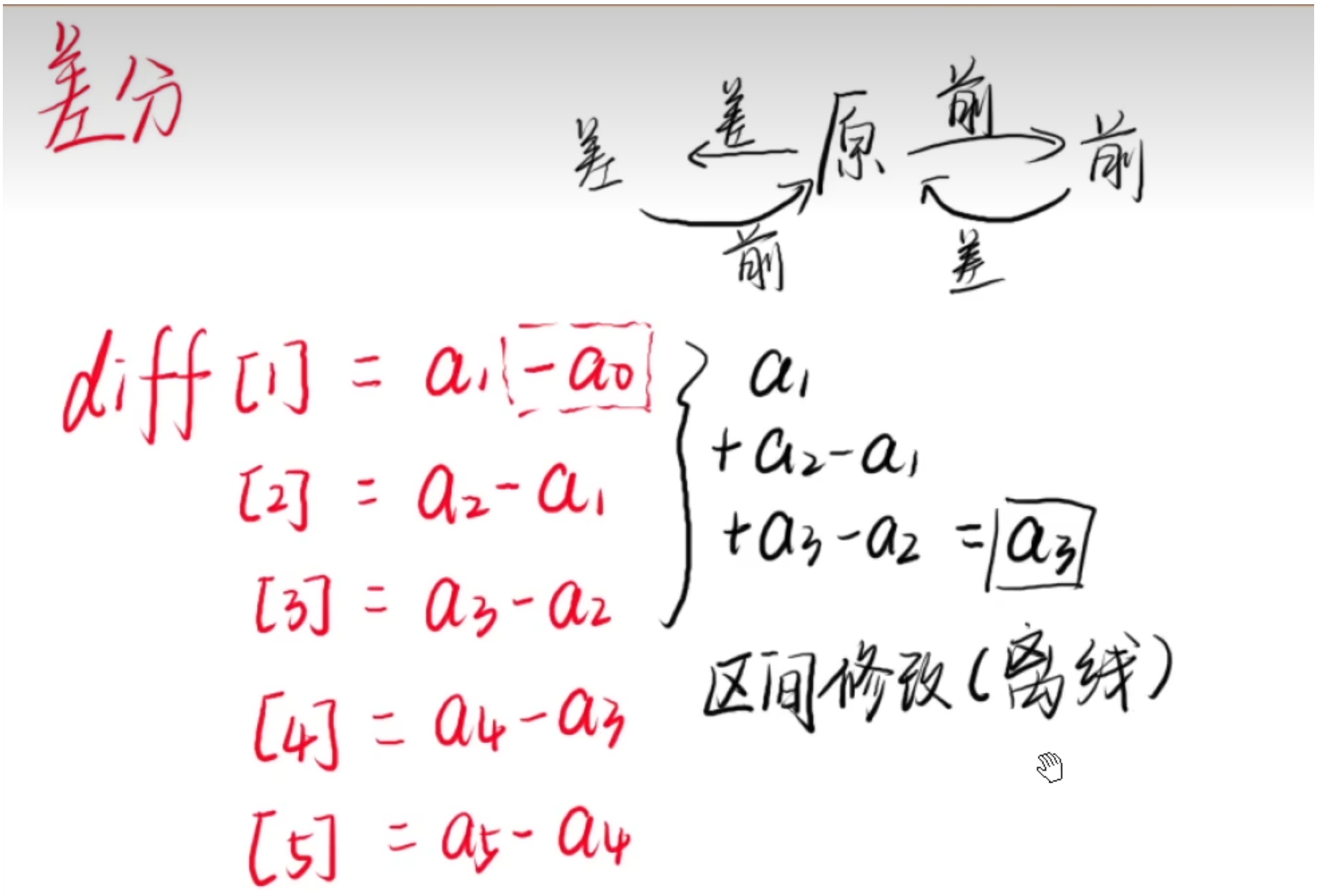
1.差分（Erik_Tse_p9）

前缀和的应用：求区间长度

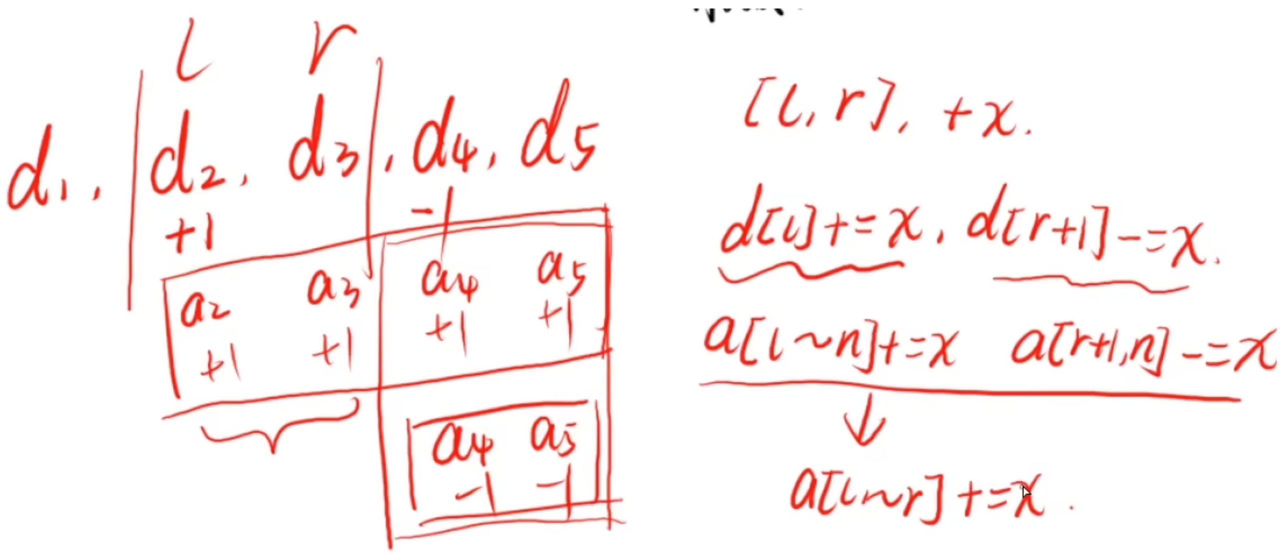
对于需要多次求区间长度来说：先求前缀和预处理一下



差分及其三者关系



差分操作过程：



如图所示：

```
d2+1会导致求差分数组的前缀和（a数组）的时候 a2到a5都+1 （相当于状态继承一样）(原理：详见差分数组转原数组)

如何只想让d2和d3两个+1 就需要把后面状态转移给抵消掉所以d3后面的d4就-1将后面的抵消掉
```

示例代码：

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

const int MAXN = 1e5 + 5;
int a[MAXN] = {0}, b[MAXN] = {0};

int main() {
  int n, m;
  cin >> n >> m;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
    b[i] = a[i] - a[i - 1];  // 构建差分数组
  }
  while (m--) {
    int l, r, d;
    cin >> l >> r >> d;
    b[l] += d;       // 区间左端点加d
    b[r + 1] -= d;   // 区间右端点的下一个位置减d
  }

  for (int i = 1; i <= n; i++) {
    b[i] += b[i - 1];   // 求前缀和得到原数组
    cout << b[i] << " ";
  }
  // 最后得到操作之后的原数组
  return 0;
}
```

2.贪心（test1_2)

题意

n 个砝码，只能放天平的一边，问最小的无法称重的重量。

思路

如果你现在能称 [l,r] ，新加入的砝码重量为 a ,显然，必须使用这颗砝码能称的重量区间为 [l+a,r+a] 。

这时，如果 r≥l+a−1 ,那么称重连续区间就可以扩展成 [l,r+a] 。显然，初始区间为 [0,0] ；然后按砝码升序（题目已经给了，不需要排序）计算即可。 时间复杂度为 O(n) 。

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

int main() {
    int n;
    cin >> n;
    vector<int> weights(n);
    for (int i = 0; i < n; i++) {
        cin >> weights[i];
    }
    sort(weights.begin(), weights.end());
```

```
    int sum = 0;
    for (int i = 0; i < n; i++) {
        if (weights[i] > sum + 1) {
            break;
        }
        sum += weights[i];
    }
    cout << sum + 1 << endl;
    return 0;
}
```

解释如下：

```
这里的r、l和a分别代表当前可以称出的重量的连续区间的右端点、左端点和新加入的砝码的重量。当r >= l + a - 1
时，表示新加入的砝码的重量a可以通过已有的砝码组合称出，因此可以将连续区间扩展到r + a。

为什么是r >= l + a - 1呢？这是因为在连续区间[l, r]中，我们可以称出的重量包括l, l+1, ..., r。如果新加
入的砝码的重量a满足a <= r - l + 1，那么我们就可以通过已有的砝码组合称出重量a，因此连续区间可以扩展到r +
a。

例如，假设当前可以称出的重量的连续区间是[1, 5]，也就是我们可以称出重量1, 2, 3, 4, 5，那么如果新加入的砝
码的重量a是3，因为5 >= 1 + 3 - 1，我们可以通过已有的砝码组合称出重量3，因此连续区间可以扩展到5 + 3 =
8，也就是[1, 8]。

例如，[1,7] a=7,前面的1到7看作是7个独立的小块，后加入的7作为一个单独的大块，本身就可以实现1到7，再以大块
7为跳板后面接上1到7 结果就是 [1,14]

反例：[1,7] a=100,不太记得题目了，以100为跳板就 [101,107]
规则简单来说就是不必须用到之前的区间
```

为什么没有写出来?

```
想到了维护一个区间，但是没有仔细想，觉得自己会超时
```

3.二进制枚举（test1_4）

> 对于正反两种情况用0和1表示恰到好处

题意

T个样例，输入一个数列（输入长度为n），问是否能添加一个等号和若干加减号，让其成为一个等式。

题解

1.等号相当于把数列分成两部分，前半部和后半部

2.如果等号是固定的，那么算前半部和后半部所有的取值，然后看这两个集合是否存在交集。 所以，枚举等号的位置，对于左右半部，使用二进制枚举加减号的所有组合，判断其是否存在交集。这时的时间复杂度为 $O(n\cdot 2^n)$ 。

有四个数字1，2，3，4 保持1，2，3，4的顺序不变，每两个数字之间可以插入+号或者-号，如何快速求得所有的结果

```
000` -> `1 + 2 + 3 + 4
001` -> `1 + 2 + 3 - 4
010` -> `1 + 2 - 3 + 4
011` -> `1 + 2 - 3 - 4
100` -> `1 - 2 + 3 + 4
101` -> `1 - 2 + 3 - 4
110` -> `1 - 2 - 3 + 4
111` -> `1 - 2 - 3 - 4
```

用二进制表示+ - 这样对于符号的位置的判断就比较清晰，就可以更方便的枚举出来

总共 2^n总情况数

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

// Function to evaluate an expression given the numbers and a binary mask
int evaluateExpression(const vector<int>& numbers, int mask) { // 这的mask就是上次使用的mask
    int result = numbers[0];
    for (int i = 0; i < 3; ++i) { // We have 3 positions for operators between 4 numbers
        if (mask & (1 << i)) {  // 位运算：使得mask的值从000到111
                               // 每次都要与运算 三次：：最终结果就是000到111
            result -= numbers[i + 1];
        } else {
            result += numbers[i + 1];
        }
    }
    return result;
}

// Function to generate the expression string based on the binary mask
string generateExpressionString(const vector<int>& numbers, int mask) {
    string expression = to_string(numbers[0]);
    for (int i = 0; i < 3; ++i) {
        if (mask & (1 << i)) {  // 精华 1 << i 表示1移动i位
            expression += " - ";
        } else {
            expression += " + ";
        }
        expression += to_string(numbers[i + 1]);
    }
    return expression;
}

int main() {
    vector<int> numbers = {1, 2, 3, 4};
```

```
    // There are 2^3 = 8 possible combinations of + and -
    for (int mask = 0; mask < 8; ++mask) {
        string expression = generateExpressionString(numbers, mask);
        int result = evaluateExpression(numbers, mask);
        cout << expression << " = " << result << endl;
    }
    // 注意这里是循环 0到7  感觉效果还是bitset<3>(i)更好点

    return 0;
}
```

> 下面的这种方式更容易理解些

关于如何快速生产需要的二进制数位：比如已知3位 -> 对应2的3次方即0到7

```
#include <iostream>
#include <bitset>
using namespace std;

int main() {
    for(int i = 0; i < 8; i++) {
        // 使用bitset来快速生成二进制表示
        cout << bitset<3>(i) << endl;
    }
    return 0;
}
```

输出结果是：

```
000
001
010
011
100
101
110
111
```

如果需要将结果表示为string：

```
#include <iostream>
#include <bitset>
using namespace std;

int main() {
    for(int i = 0; i < 8; i++) {
        // 使用bitset来快速生成二进制表示
        string binary = bitset<3>(i).to_string();
        cout << binary << endl;
    }
    return 0;
}
// 也就是使用to_string即可：：简简单单
```

> 显然使用 bitset<3>(i) 稍微没那么容易忘记，使用性稍微高点

4.基于比较的排序和桶排序

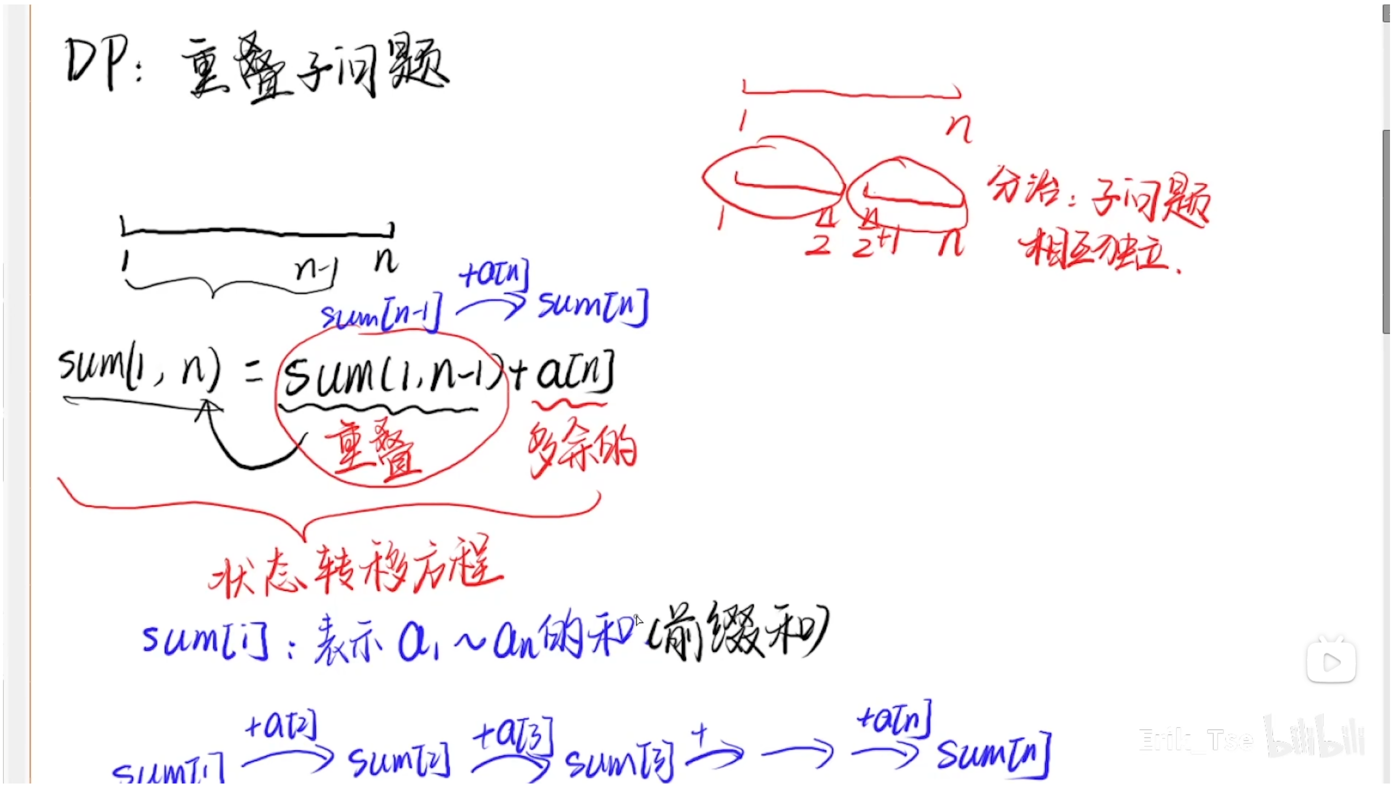
基于比较

```
sort(arr.begin(),arr.end());

//sort(arr,arr+n);
```

基于桶

```
#include <bits/stdc++.h>
using namespace std;
const int N = 2e6 + 9;
int a[N];//a[i]表示数字i出现的次数

signed main()
{
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int n, m;cin >> n >> m;
    for(int i = 1;i <= m; ++ i)
    {
        int x;cin >> x;
        a[x] ++;
    }
    for(int i = 0;i <= n; ++ i)
    {
        for(int j = 1;j <= a[i]; ++ j)
        {
            cout << i << ' ';
        }
    }
    return 0;
}
```

5.动态规划（重叠子问题）（Erik_Tse_p10)



6.map

```
insert(key,value);
// insert({1,"hello"})
// insert(make_pair(1,"world"))
erase(key);
find(key);
count(key);
// 然后都是键的操作
size();
clear();
empty();
begin();
end();
// 然后是整体性操作
```

常用举例：

```
// 输出
if(mp.find(key)!=mp.end()) cout<<mp[key]<<endl;

if(mp.count(key))cout<<mp[key]<<endl;


// 遍历
for(auto &it:map)cout<<it.firse<<" "<<it.seconde<<endl;
```

```
for(map<int,int>::iterator it=mp.begin(),it!=it.end();it++)cout<<it->firse<<" "<<it-
>seconde<<endl; // 类似于解引用


// 重载运算符使用map
struct Node{
    int x,int y;
    bool operator < (const  Node& u)const{
        return x==u.x?y<u.y:x<u.x;
    }
    // 第一依据和第二依据：依旧是<符号
}
```

7.set

基本操作

```
// 重载操作运算符
struct Node{
    int x,y;
    bool operator < (const  Node& u)const{
        return x==u.x?y<u.y:x<u.x;
    }
} // 重载 < 运算符

struct cmp{
    bool operator () (const int &u,const int &v)const{
        return u>v;
    }
} // 重载 () 运算符


insert(x);
erase(x);
find(x);
count(x);
// 单个数据的操作
clear();
size();
empty();
begin();
end();
// 然后后面都是整体操作
```

常见用法与map基本一致：

```
for(auto &it :set)cout<<it<<endl;


for(set<int>::iterator it=ms.begin();it!=ms.end();it++){
    cout<<*it<<endl;
}  // 类似于解引用
```

迭代器方式也是和map基本一致

8.大小堆/优先队列

默认是大根堆

```
priority_queue<int>maxMp;
priority_queue<int,vector<int>,greater<int>>minMp; // 最小堆：正确
```

常用方法

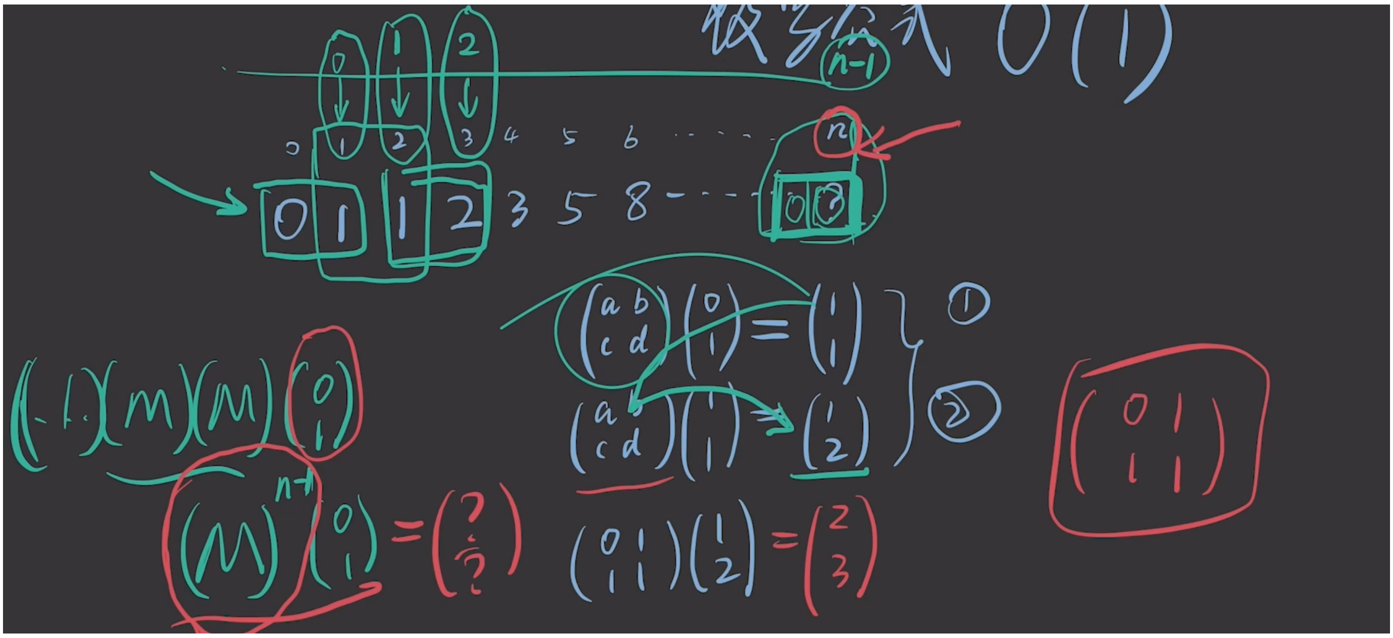
```
push();
pop();
top();
size();
empty();
```

9.矩阵快速幂（long long）

> 参考别人的答案那里也是写的一坨shit

何时使用mod？ 在任何运算会大于mod这个值的时候就使用一次mod就行

原理：（核心：幂的拆解：：数学推导） 仔细看这个图就知道最后是返回 0 1 还是 1 0：：显然是1 0



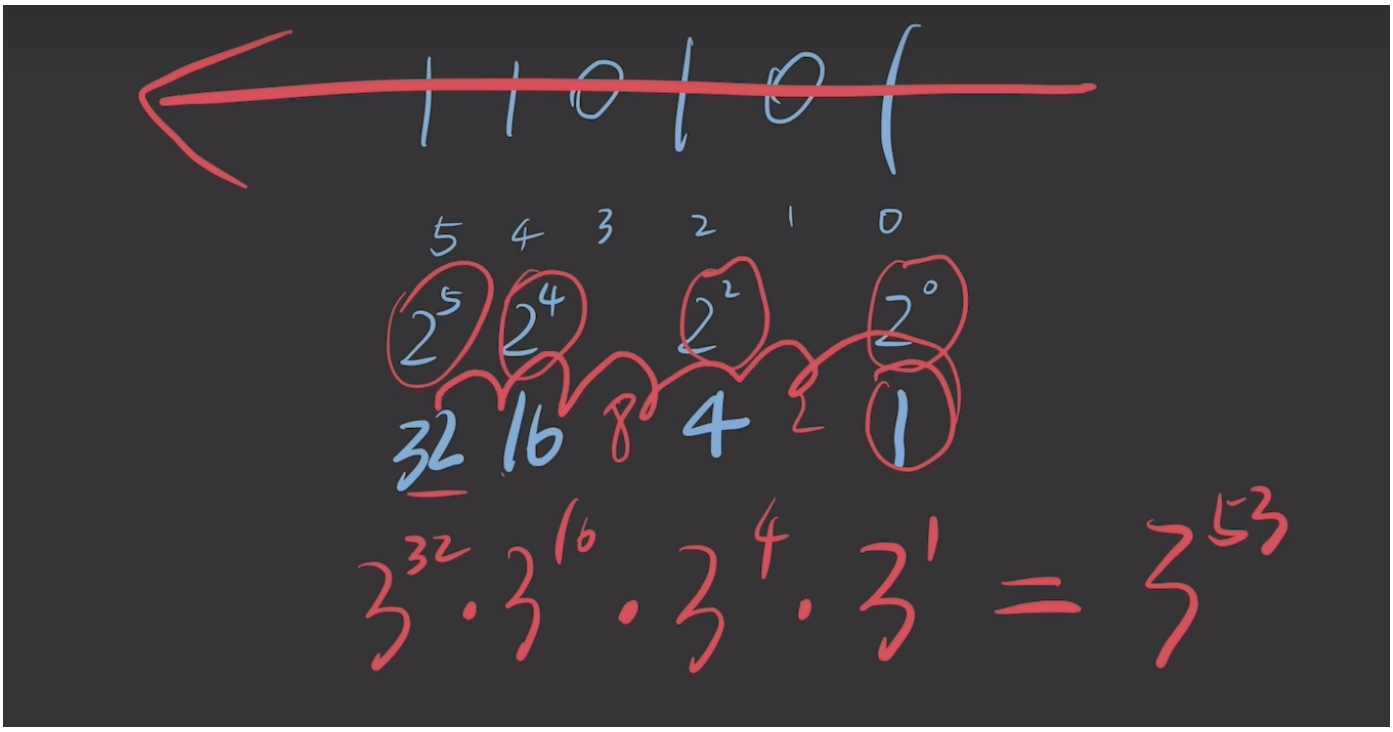
一个神秘的 2X2 的矩阵 与两两作为一项的矩阵

所以结果例如斐波那契第n行

[M]^n-1 次方 乘以 [0 1] 那个矩阵

当然是先求得 [M]^n-1 次方

下面是传统快速幂的计算：



以上面3^53次方为例：不难看出来快速幂的运算技巧

将53拆解为2进制数，可见每个大项都可以由前一项推导而出（有一种保留结果的感觉）

[M]^n-1 的拆解同理

示例代码:(python)

```
def matmul(m1, m2):
    res = [[0]*len(m2[0]) for _ in range(len(m1))]
    for y in range(len(m1)):
        for x in range(len(m2[0])):
            n = 0
            for a, b in zip(m1[y], (line[x] for line in m2)):
                n += a * b
            res[y][x] = n
    return res

def fab(n):
    if n < 0:
        return 0
    m = [
        [0, 1],
        [1, 1],
    ]
```

```python
    resM = [[1, 0], [0, 1]]
    while n != 0:
        if n & 1 == 1:
            resM = matmul(m, resM)
        m = matmul(m, m)
        n >>= 1
    c, d = resM[0]
    return c

def main():
    for i in range(1000):
        print(fab(i))
```

(c++) 版:

```cpp
#include<bits/stdc++.h>

using namespace std;

vector<vector<int>> matmul(vector<vector<int>> m1, vector<vector<int>> m2) {
    vector<vector<int>> res(m1.size(), vector<int>(m2[0].size(), 0));
    for (int y = 0; y < m1.size(); ++y) {
        for (int x = 0; x < m2[0].size(); ++x) {
            int n = 0;
            for (int i = 0; i < m1[y].size(); ++i) {
                n += m1[y][i] * m2[i][x];
            }
            res[y][x] = n;
        }
    }
    return res;  // 依旧是一个2x2的矩阵
}
// 模拟矩阵乘法罢了: 那么关于如何运算矩阵乘法? 这点需要充分明确 (这里倒是还好只好2x2的矩阵)

int fab(int n) {
    if (n < 0) {
        return 0;
    }
    vector<vector<int>> m = {{0, 1}, {1, 1}};  // 魔法矩阵
    vector<vector<int>> resM = {{1, 0}, {0, 1}};  // 单位矩阵 (用于储存中间变量) 到最后就是结果
    while (n != 0) {
        if (n & 1 == 1) {
            resM = matmul(resM, m);
        }
        m = matmul(m, m);
        n >>= 1;
    }
    return resM[1][0];  // 这里我更认可为 1 0 : 可以自己画图参考
}
```

```
int main() {
    for (int i = 0; i < 1000; ++i) {
        cout << fab(i) << endl;
    }
    return 0;
}
```

已知是2x2的矩阵相乘（不要理解错误为2x2的矩阵乘 1x2的矩阵了，那么矩阵乘法那里可以优化一下：

```
#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> matmul(const vector<vector<int>>& m1, const vector<vector<int>>& m2) {
    vector<vector<int>> res(2, vector<int>(2, 0));
    res[0][0] = m1[0][0] * m2[0][0] + m1[0][1] * m2[1][0];
    res[0][1] = m1[0][0] * m2[0][1] + m1[0][1] * m2[1][1];
    res[1][0] = m1[1][0] * m2[0][0] + m1[1][1] * m2[1][0];
    res[1][1] = m1[1][0] * m2[0][1] + m1[1][1] * m2[1][1];
    return res;
}

int fab(int n) {
    if (n < 0) {
        return 0;
    }
    vector<vector<int>> m = {{0, 1}, {1, 1}};  // 魔法矩阵
    vector<vector<int>> resM = {{1, 0}, {0, 1}};  // 单位矩阵（用于储存中间变量）到最后就是结果
    while (n != 0) {
        if ((n & 1) == 1) {  // 这里注意优先级
                resM = matmul(resM, m);  // 注意乘法顺序
        }
        m = matmul(m, m);
        n >>= 1;
    }
    return resM[1][0];  // 返回斐波那契数列的结果
}

int main() {
    for (int i = 0; i < 1000; ++i) {
        cout << fab(i) << endl;
    }
    return 0;
}
```

因为矩阵乘法不满足交换律：所以 resM = matmul(resM, m);这里的顺序需要额外注意一下：以及最后的返回值的情况

通过实测：是

return resM 0 1

或者 resM 1 0 都是可以的，但是0 0 是错误的

10.二分查找

> 适用于 "在从小到大的排序中寻值：核心提示就是顺序"

find()函数（针对于迭代器）

vector

基于暴力查找 O(n)

map&set

其两者底层都是红黑树：也就是其两者的find函数基于红黑树：时间复杂的是O(logn)：但是代码实现并不是二分查找

标准

```
binary(int key,vector<int> mv){
    int l=0,r=mv.size()-1;
    while(l<=r){
        int mid=(l+r)/2;
        if(mv[mid]==key)  return mid;
        else if(mv[mid]<key)  l=mid+1;
        else r=mid-1;
    }
    return 0;
}
// 使用纯实心的端点值
```

11.DFS

```
vector<int> a; // 记录每次排列
vector<int> book; //标记是否被访问

void dfs(int cur, int k, vector<int>& nums){
    if(cur == k){ //k个数已经选完，可以进行输出等相关操作： : 终点判断
        for(int i = 0; i < cur; i++){
            printf("%d ", a[i]);
        }
        return;
    }

    for(int i = 0; i < k; i++){ //遍历 n个数，并从中选择k个数
        if(book[nums[i]] == 0){ //若没有被访问

            a.push_back(nums[i]); //选定本输，并加入数组            : : 前->加入
            book[nums[i]] = 1; //标记已被访问
```

```
            dfs(cur + 1, k, nums); //递归，cur+1                    : : 中->递归

            book[nums[i]] = 0; //释放，标记为没被访问，方便下次引用      : : 后->归还
            a.pop_back(); //弹出刚刚标记为未访问的数
        }
    }
}
```

总结上来说就是我总结的三个小点：（牢记这三个步骤就行）

前 -> 中 -> 后 <-> （加入 -> 递归 -> 归还）

12.BFS（数字bfs）

> 补充一点：
>
> 对于用途为bool数组 最方便最省空间的方式是定义bitset数组：0则为false 1则为true

在范围内直接bfs即可：

```
#include <bits/stdc++.h>

using namespace std;

const int maxn = 1e5 + 5;
int vis[maxn << 1];

class Solution {
 public:
  void solve() {
    int o;
    cin >> o;
    while (o--) {
      int x, y;
      cin >> x >> y;
      cout << bfs(x, y) << endl;
    }
  }
  int bfs(int s, int t) {
    memset(vis, -1, sizeof(vis));  // 定义和初始化判断数组
    queue<int> q;
    q.push(s);  // 定义和初始化判断队列
    vis[s] = 0;   // 初始化
    while (!q.empty()) {
      int u = q.front();
      q.pop();  // 队列基操
      if (u == t) return vis[u];
      if (u - 1 >= 0 && vis[u - 1] == -1) {  // -1大于0且非检索
        vis[u - 1] = vis[u] + 1;
        q.push(u - 1);
      }
      if (u + 1 <= 2 * t && vis[u + 1] == -1) {  // +1小于t且非检索
```

```
        vis[u + 1] = vis[u] + 1;
        q.push(u + 1);
      }
      if (u * 2 <= 2 * t && vis[u * 2] == -1) {  // *2小于2t且非检索
        vis[u * 2] = vis[u] + 1;
        q.push(u * 2);
      }
      if (u / 2 >= 0 && vis[u / 2] == -1) {  // /2大于0且非检索
        vis[u / 2] = vis[u] + 1;
        q.push(u / 2);
      }
    }
    return -1;
  }
};

int main() {
  Solution s;
  s.solve();
  return 0;
}
```

注意那几种基操即可：难点是如何判断这里使用 BFS

> 范围 -> 几种一致操作 -> 能否查找判断

13.前后指针（左右指针）

马里奥跳砖块，马里奥最多跳的距离是 d ,要坐标 1 的砖块跳到坐标 w 的砖块上，中间有 n 个砖块，问最少去掉几块中间的砖块，使得马里奥无法完成任务。

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

using namespace std;

int main() {
    int n, w, d;
    cin >> n >> w >> d;

    vector<int> a(n + 2);
    a[0] = 1;
    a[n + 1] = w;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    // 排序以确保砖块位置按顺序排列
    sort(a.begin(), a.end());
```

```
    if (a[n + 1] - a[0] <= d) {
        // 如果一开始马里奥就可以直接跳过去
        cout << "-1" << endl;
        return 0;
    }

    int left = 0, right = 0, minRemove = n;
    while (right <= n + 1) {
        while (right <= n + 1 && a[right] - a[left] <= d) {
            right++;
        }
        // 更新最少移除的砖块数量
        minRemove = min(minRemove, n - (right - left - 1));
        // 移动左指针
        left++;
        if (minRemove == 0) {
            // 如果已经没有砖块需要移除，直接结束
            break;
        }
    }

    cout << minRemove << endl;
    return 0;
}
```

> 虽然我觉得找到 所有砖块中 已经存在的最长距离 然后用这个距离和d去比较就行，如果这个距离大大于d就说明无法跳过，否则针对最长的距离进行处理就行：：假设模拟
>
> 这个双指针反而我有点不懂
>
> 典型题目：：三数之和

14.奇怪的进制

斐波那契进制

核心：逆向思维（看了不知道抄的谁的代码：7-80行只能说找对方向和重要：不然复杂度翻倍）

以考试1为例：第一题找对发现非常简单：代码也非常短

第二题：一样：：就一个更新策略

第三题：模拟：虽然我写的不好太冗余了

第四题：二进制加减模拟

回到正题：该题解答：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
int t;
```

```
ll f[50], s[50];

int main() {
  f[0] = 0, f[1] = 1, f[2] = 1, s[0] = 0, s[1] = 1, s[2] = 2;
  for (int i = 3; i <= 42; i++) {
    f[i] = f[i - 1] + f[i - 2];
    s[i] = s[i - 1] + f[i];  // 前缀和
  }
  cin >> t;
  while (t--) {
    ll ans = 0, x;
    cin >> x;
    for (int i = 42; i >= 0; i--) {  // 核心代码：对于前一项的加入：以及后面二进制的处理（使用）
      if (s[i - 1] < x) {  // 当前 i-1 个 Fibonacci 数之和小于 x 时，则第 i 个数必须取
        x -= f[i];
        ans += (ll)1 << (i - 1);   //使用 二进制的座椅运算就行处理 ：：  最后就不哟昂将二进制数单独转
为十进制数
      }
    }
    cout << ans << endl;
  }
  return 0;
}
// 关于为何座移的是 i-1 而不是 i个  ：：  其实这里就是配合上面的 i-1  因为确定的是 i-1 而不是 i：
// 这里是易错点：别忘记了
```

核心：见注释部分

15.素数筛

> 如果想使用 bitset 需要长度为定值：：如果没有开启O2优化的话不要使用

埃式筛法*（推荐使用）

> 感觉很好理解：就是把素数向后的一轮筛掉就行，被筛掉的数就跳过，否则针对该数进行筛选循环
>
> 理解算法实现就不难写出该段代码了

板子：

```
#include<bits/stdc++.h>

using namespace std;

void qPrime(int n) {
  vector<bool> prime(n + 1, true);  // vector的初始化：初始全为true
  prime[0] = prime[1] = false;  // 初始化0和1
  for (int p = 2; p * p <= n; p++) {  // 从2开始
    if (prime[p] == true) {  // 如果没有被筛：如果已经被筛了那么p的倍数依旧被筛了就不要考虑了
      for (int i = p*2; i <= n; i += p) prime[i] = false;  // 从2j开始，筛为非素数
    }
  }
```

```
  // 信息打印:打印全部的素数
  for (int p = 2; p <= n; p++) 范围见下面的范围值
    if (prime[p]) cout << p << endl;
  }

int main() {
  ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);
  int n = 100000;
  cout << "以下是小于等于 " << n << " 的素数: ";
  qPrime(n);
  return 0;
}
// 代码核心见注释部分: for循环的位置
```

将埃式筛 截取出来如下:

```
void qPrime(int n) {
  vector<bool> prime(n + 1, true);  // vector的初始化: 初始全为true
  prime[0] = prime[1] = false;  // 初始化0和1
  for (int p = 2; p * p <= n; p++) {  // 从2开始
    if (prime[p] == true) {  // 如果没有被筛: 如果已经被筛了那么p的倍数依旧被筛了就不要考虑了
      for (int i = p*2; i <= n; i += p) prime[i] = false;  // 从2j开始, 筛为非素数
    }
  }
}
```

> 小结：注意初始化（true） -> 注意范围值 -> 注意合时开始筛（用true去筛） -> 注意筛的范围（从2p开始，然后每次递增一个p）

> 总结一下思路清晰多了

使用bitset的板子 --> 特别强调：使用在o2优化的oj中：一般别随便使用

```
bitset<N> vis;

void Prime(int n) {
  vis.set();  // 调用标准函数将其全部置为1
  vis[0] = vis[1] = 0;
  for (int i = 2; i * i <= n; i++) {
    if (vis[i]) {
      for (int j = i * 2; j <= n; j += i) vis[j] = 0;
    }
  }
}
// 实现素数打表
// 注意两个for玄幻的起点和红点
```

欧拉筛法（线性筛）

> 每个被筛去的数都是被其最小因数给筛去的
>
> 合数=最小质因数*其他数 -> 核心
>
> 考试的时候想起哪个用哪个：实际上很大区别没有的

```
vector<boot> isp(N,true);
vector<int> prime;
for (int i = 2;i < N;i++) {  // 注意是从2开始
    if (isp[i]) {
        prime.push_back(i);
    }
    for (int j = 0;j < prime.size() && prime[j] * i < N;j++) {  // 循环每个素数
        isp[prime[j] * i] = false;  // 这里筛
        if (i % prime[j] == 0) break;  // 算法精华
    }
}
```

16.双指针

> 左右指针属于 双指针 这点无疑

逆向双指针

例题：

两数求和（充分利用数组已经排好序的特性：两数之和的大小就是确定的）

同向双指针（滑动窗口）（快慢指针）

盛最多水的容器：：创建一个变量保存最大的盛水量：：然后左右指针高度比较：谁短就移动谁

> 注意：不多说了我觉得github desktop还是不如Gitkraken：：可以直观看见分支的状况

17.树的几种基本遍历方式

> 不多说：核心就是输出的顺序的位置罢了

pre

```
void pre(Node* node) {
    if (node == nullptr) return;
    cout << node->value << ' ';
    pre(node->left);
    pre(node->right);
}
```

order

```
void inorder(Node* node) {
    if (node == nullptr) return;
    inorder(node->left);
    cout << node->value << ' ';
    inorder(node->right);
}
```

post

```
void post(Node* node) {
    if (node == nullptr) return;
    post(node->left);
    post(node->right);
    cout << node->value << ' ';
}
```

> 只是稍微复习下不赘述咯

18.辗转相除法 (gcd)

> 用于快速求得 两数的最小的公因数

```
int gcd(int a,int b)
{
  int t;
    while(b!=0)
    {
        t=a%b;
        a=b;
        b=t;
    }
    return a;
}
```

口诀：：传入大小，循环小不等于0，大等于小，小等于余

19.四大常见贪心算法

> 以前还用的txt和源码做笔记：：非常不直观的方法：：笔记效率太低:复习不方便也基本上没怎么看过
>
> 我现在对markdown给予我最大的诚意说nb，极为方便

最短路径

Dijkstra（迪杰斯特拉算法）

算法实现：

```
#include<stdio.h>
```

```
#include<string.h>
#include<stdbool.h>
#define max 0x3f3f3f3f

int shuzu[50][50];  //邻接矩阵
int d[50];  //起点到各个点的最短距离::最开始全部初始化为无穷大
bool judge[50];  //判断每个点是否已经在最短路径集合中

void initial(int y){
    memset(shuzu, max, sizeof(shuzu));
    memset(d, max, sizeof(d));
    memset(judge, 0, sizeof(judge));
    for(int i = 0; i < y; i++) {
        shuzu[i][i] = 0;
    }
}
// 数据初始化

void getshuzu(int q){
    while(q--){
        int a, b, len;
        scanf("%d %d %d", &a, &b, &len);
        shuzu[a][b] = len;
        shuzu[b][a] = len;
    }
}
// 获得初始数组

void dijkstra(int start, int y){
    d[start] = 0;
    for(int i = 1; i <= y; i++){
        int min = max, u = -1;
        for(int j = 1; j <= y; j++){
            if(!judge[j] && d[j] < min){
                u = j;
                min = d[j];
            }
        }
        // 每次先找到一个最近的点：这几种算法这里的操作都是类似的
        if(u == -1) break;
        judge[u] = true;

        for(int v = 1; v <= y; v++){
            if(!judge[v] && shuzu[u][v] < max && d[u] + shuzu[u][v] < d[v]){
                d[v] = d[u] + shuzu[u][v];
            }
        }
    }
}
// dijkstra算法 主要看这个函数：其他初始化和调试的地方 稍微看看就好

int main(){
```

```
    int start, end, q, y;  // 起点 终点 多少段数据 点个数
    scanf("%d %d %d %d", &start, &end, &q, &y);
    initial(y);
    getshuzu(q);
    dijkstra(start, y);
    printf("The shortest distance from %d to %d is %d\n", start, end, d[end]);
    return 0;
}
```

> 这个算法本身不是很难：需要自己掌握这种思维：而不是去死记硬背代码（算法）：算法细节多加注意
>
> 死记硬背是记不住代码的（理所当然）

Floyd(弗洛伊德算法)

算法实现：

```
#include<stdio.h>
#include<string.h>
#define N 0x3f3f3f
int A[50][50];
int v,e;
void init(){
  for(int i = 0 ;i < 50;i++)
    for(int j = 0;j < 50;j++){
    if(i == j)    A[i][j] = 0;
    else        A[i][j] = N;
  }
  printf("顶点个数:\n");
  scanf("%d",&v);
  printf("边的条数:\n");
  scanf("%d",&e);
}

void input(int n){
  printf("输入所有的邻接的两个点和他们的边:\n");
  while(n--){
    int a,b,l;
    scanf("%d %d %d",&a,&b,&l);
    A[a][b] = l;      //len-边的长度
    A[b][a] = l;
  }
}

void cal(int k){
  if(k>v) return;
  for(int i = 1;i <= v;i++){
    for(int j = 1; j <= v; j++){
      if(A[i][j] >  A[i][k] + A[k][j]){
        A[i][j] = A[i][k] + A[k][j];
      }
    }
```

```
  }
  cal(++k);


int main(){
  init();
  input(e);
  cal(1);
  for(int i=1;i<=6;i++){
    for(int j=1;j<=6;j++){
      printf("%5d\t",A[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

> 删去冗余的调试信息：剩下的就是比较简洁的算法了
>
> 虽然我绝地将

最小生成树

Prim（普利姆算法）

算法实现：

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

using namespace std;

const int MAXN = 1000, INF = 0x3f3f3f3f;
vector<vector<int>> g(MAXN, vector<int>(MAXN, INF));  // 比较复杂的初始化方式
vector<int> dist(MAXN);
vector<bool> book(MAXN);
int n, m, res;

void prim() {
    fill(dist.begin(), dist.end(), INF);
    fill(book.begin(), book.end(), false);
    dist[1] = 0;
    book[1] = true;
    for (int i = 2; i <= n; i++) dist[i] = g[1][i];

    for (int i = 2; i <= n; i++) {
        int temp = INF, t = -1;
        for (int j = 2; j <= n; j++) {
            if (!book[j] && dist[j] < temp) {
                temp = dist[j];
                t = j;
```

```
            }
        }
        if (t == -1) {
            res = INF;
            return;
        }  // 和上面算法几乎一样的思想
        book[t] = true;
        res += dist[t];
        for (int j = 2; j <= n; j++) dist[j] = min(dist[j], g[t][j]);
    }
}

int main() {
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int a, b, w;
        cin >> a >> b >> w;
        g[a][b] = g[b][a] = w;
    }
    // 初始化
    res = 0;
    prim();
    if (res == INF) cout << "orz";
    else cout << res;
    return 0;
}
```

> 还是注重强调算法思想

Kruskal（克鲁斯卡尔算法）

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

const int MAXV = 110;  // 最多多少个点
const int MAXE = 10010;  // 最多多少个边

struct Edge {
    int u, v, cost;
};

vector<Edge> edges;
int father[MAXV];

bool cmp(const Edge& a, const Edge& b) {
    return a.cost < b.cost;
}
// 注意cmp的自定义都是< 从小到大排)

int findFather(int x) {
```

```
    return father[x] == x ? x : father[x] = findFather(father[x]); // 这里注意路径压缩：虽然都是对的
}

int kruskal(int n, int m) {
    int ans = 0, numEdge = 0;
    for(int i = 0; i < n; ++i) father[i] = i;
    sort(edges.begin(), edges.end(), cmp);
    for(int i = 0; i < m; ++i) {
        int faU = findFather(edges[i].u);
        int faV = findFather(edges[i].v);
        if(faU != faV) {
            father[faU] = faV;
            ans += edges[i].cost;
            if(++numEdge == n - 1) break;
        }
    }
    return numEdge == n - 1 ? ans : -1;
}

int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    edges.resize(m);
    for(int i = 0; i < m; ++i) {
        scanf("%d%d%d", &edges[i].u, &edges[i].v, &edges[i].cost);
    }
    printf("%d\n", kruskal(n, m));
    return 0;
}
```

part2

> 主要参考来源：
>
> https://www.desgard.com/algo/

好话：

数据结构到底有多优雅？看一看**并查集**和**树状数组**你就知道什么才是真正的优雅。第一次看到 `lowbit` 居然能用一句简单的位运算来描述。每接触一个高效的数据结构，都会让我去挖掘其实现的方式，最后的赞叹永远是优美、神奇、精悍！

1.时间复杂度估算土法

1. 土法一：执行一行约是一次运算
2. 土法二：以经验计算时间

一般的计算机，在处理 10^7计算的时候需要消耗一秒的时间。

3. 土法三：取极限估算复杂度

2.快速幂

> 算法中数学思维

快速幂运算

例如计算x^n，当n为10的时候将n二进制拆分 1010

代码实现

```
#include <iostream>
using namespace std;

int main() {
    int n = 10; // 幂指数，下面通过二进制拆分成 1010
    int x = 2; // 底数
    int res = 1; // 累乘的答案
    while (n) {
        // 去除二进制的最低位，也就是上面推导中的右式，如果 n & 1 == 1，说明是 *1
        if (n & 1) {
            // 如果是 *1，则根据我们观察出来的规律，对维护的结果做累乘
            res *= x;
        }
        // 转换到下一位
        x *= x;
        // 二进制右移一位，目的是取到下一个低位二进制
        n >>= 1;
    }
    cout << res << endl; // 1024
    return 0;
}
```

快速幂口诀：

> 二进制，一个存底，一个存积
>
> 稍微多看看源码就能理解我这个意思了

代码简化与应用示例：

```
#include <iostream>
using namespace std;

int qpow(int x, int n) {
    int res = 1;
    while (n) {
        if (n & 1) res *= x;
        x *= x;
        n >>= 1;
```

```
    }
    return res;
}

int main() {
    cout << qpow(2, 10) << endl; // 1024
    cout << qpow(4, 2) << endl;  // 16
    cout << qpow(5, 3) << endl;  // 125
    cout << qpow(10, 6) << endl; // 1000000
    return 0;
}
```

如果需要取模运算：

```
int qpow(int x, int n, int m) {
    int res = 1;
    while (n) {
        if (n & 1) res = res * x % m;
        x = x * x % m;
        n >>= 1;
    }
    return res;
}

int main() {
    cout << qpow(10, 3, 997) << endl; // 3
    cout << qpow(10, 2, 997) << endl; // 100
    return 0;
}
```

矩阵快速幂

先看一段结构体的定义：也就是相当于一个类的处理方式

```
#define N 2

struct matrix {
    int m[N][N];
    matrix() {
        memset(m, 0, sizeof(m));
    }
    void prt();  // 定义一个函数：外部实现
};

// 重载该结构体的操作运算符
matrix operator * (const matrix a, const matrix b) {
    matrix ans;
    for (int i = 0; i < N; ++ i) {
        for (int j = 0; j < N; ++ j) {
            for(int k = 0; k < N; ++ k) {
                ans.m[i][j] += a.m[i][k] * b.m[k][j];
```

```
            }
        }
    }
    return ans;
}

// 打印测试代码 ::是作用域解析符
void matrix::prt() {
    for (int i = 0; i < N; ++ i) {
        for (int j = 0; j < N; ++ j) {
            cout << this -> m[i][j] << " ";
        }
        cout << endl;
    }
}
```

然后就是一样的了：

```
// 这是一个函数返回值为matrix
matrix qpow(matrix x, int n) {
    matrix res;
    for (int i = 0; i < N; ++ i) {
        res.m[i][i] = 1;
    }
    // 强调以及注意这里是单位矩阵的初始化
    while (n) {
        if (n & 1) res = res * x;
        x = x * x;
        n >>= 1;
    }
    return res;
}
```

fib封装

```
int fib(int n) {
    matrix a;
    a.m[0][0] = a.m[1][0] = a.m[0][1] = 1;

    matrix base;
    base.m[0][0] = 1;

    matrix ans = qpow(a, n - 1);
    ans = ans * base;

    return ans.m[0][0];
}
```

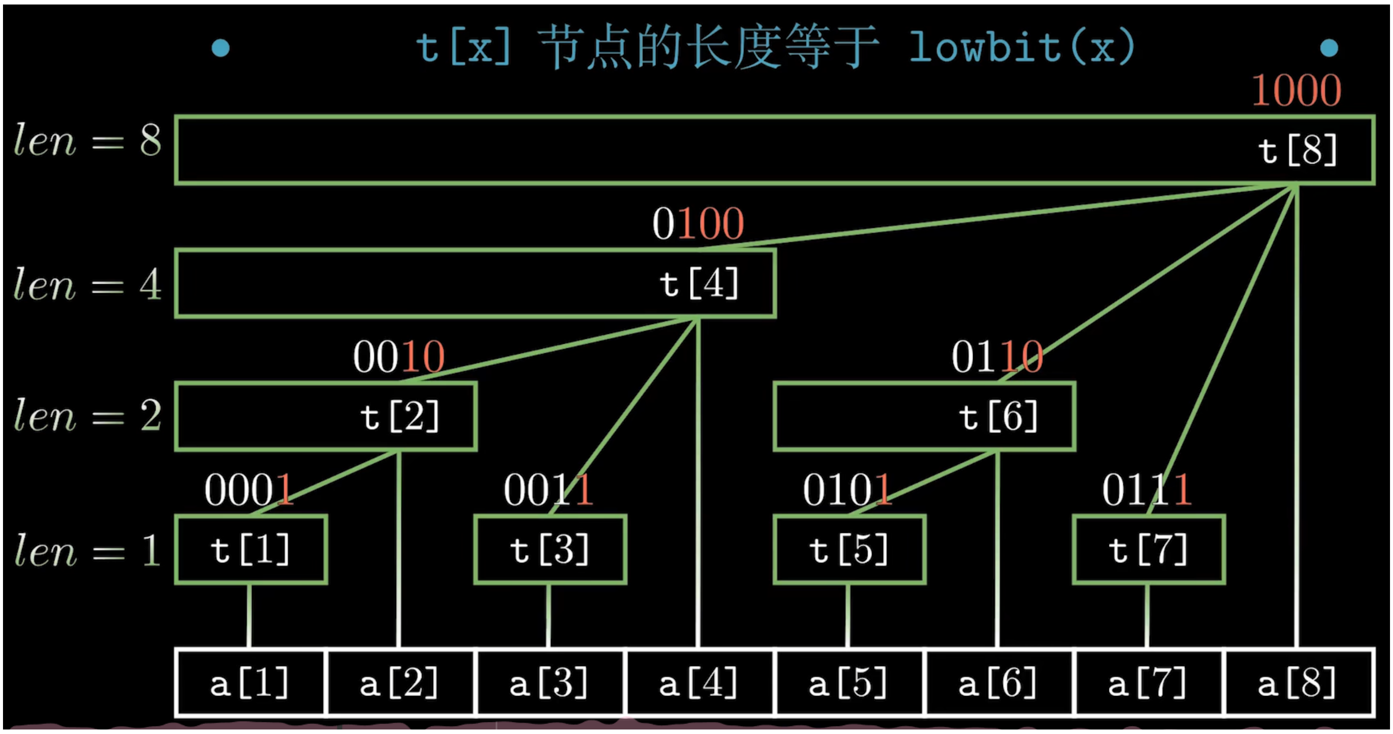
main封装

```
int main() {
    cout << fib(1) << endl; // 1
    cout << fib(2) << endl; // 1
    cout << fib(3) << endl; // 2
    cout << fib(4) << endl; // 3
    cout << fib(5) << endl; // 5
    cout << fib(6) << endl; // 8
    cout << fib(7) << endl; // 13
}
```

结合就算了：快速幂前面是学过的：但是这边总结的更精简：而且又有点忘记了

3.树状树（->RMQ问题Range Minimum/Maximum Query）

核心图片：



核心代码部分：(单点)

单点修改 查询前缀和

```
void add(int x,int k){
    for(;x <= n;x += x & -x,) t[x]+=k;
}
int ask(int x){
    int ans=0;
    for(;x;x -= x & -x) ans+=t[x];
  return ans;
}
```

> 鉴于创建树状数组的难度过高：现在先不讨论该问题：：记录树状数组的常规操作即可

单点修改 单点查询

```
add(x,k);
ask(x)-ask(x-1);
```

区间修改 单点查询

```
add(l,d); add(r+l,-d);
a[x]+ask(x);
```

区间修改 区间查询：暂时先略过

4.线段树

> 线段树和树状数组之间的区别和联系：

5.算法思想

这里就简单将算法思想分类为下面两种

- 简单模拟
- 算法模拟

> 也就是都仅仅是按照某种结果或者要求模拟某个过程

关于算法 学习：提出一宗总结的思想：用 总结的 简记口诀 对算法的执行过程会非常清晰